Bandit algorithms for searching large spaces

Louis Richard Michel Dorard

A dissertation submitted in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

of

UCL.

Department of Computer Science

University College London

2012

Statement of originality

I, Louis Dorard, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated.

Abstract

Bandit games consist of single-state environments in which an agent must sequentially choose actions to take, for which rewards are given. The objective being to maximise the cumulated reward, the agent naturally seeks to build a model of the relationship between actions and rewards. The agent must both choose uncertain actions in order to improve its model (*exploration*), and actions that are believed to yield high rewards according to the model (*exploitation*). The choice of an action to take is called a *play* of an *arm* of the bandit, and the total number of plays may or may not be known in advance.

Algorithms designed to handle the exploration-exploitation dilemma were initially motivated by problems with rather small numbers of actions. But the ideas they were based on have been extended to cases where the number of actions to choose from is much larger than the maximum possible number of plays. Several problems fall into this setting, such as information retrieval with relevance feedback, where the system must learn what a user is looking for while serving relevant documents often enough, but also global optimisation, where the search for an optimum is done by selecting where to acquire potentially expensive samples of a target function. All have in common the search of large spaces.

In this thesis, we focus on an algorithm based on the Gaussian Processes probabilistic model, often used in Bayesian optimisation, and the Upper Confidence Bound action-selection heuristic that is popular in bandit algorithms. In addition to demonstrating the advantages of the GP-UCB algorithm on an image retrieval problem, we show how it can be adapted in order to search tree-structured spaces. We provide an efficient implementation, theoretical guarantees on the algorithm's performance, and empirical evidence that it handles large branching factors better than previous bandit-based algorithms, on synthetic trees.

Acknowledgments

This work was undertaken as part of the LeStruM project (http://www.lestrum.org/), which provided motivation for bandit-based tree search methods, and the PinView project (http://www.pinview.eu/), which provided an application for many-armed bandit algorithms. I am grateful for the financial support I received from these projects.

Personally, I would like to thank:

- John Shawe-Taylor, my supervisor, who I feel privileged to have had the opportunity to work with for the last few years. Thank you for taking me as a PhD student, for giving me complete academic freedom while exposing me to great ideas, and for providing crucial feedback on my work;

- My two examiners, Rémi Munos and David Silver, who kindly accepted to read this thesis and took time to give very detailed feedback;

- My colleagues at the CSML, at the UCL CS department, and all the Machine Learners I have had interesting and useful discussions with at conferences and workshops. I would especially like to thank: Niranjan Srinivas, Csaba Szepesvari, Thore Graepel and Philippe Rolet, for answering questions and providing details on their respective work;

- Orlando Döhring, Richard H. Dorard and Matthew Higgs, for proof-reading parts of this thesis; Afra Mashhadi, for her help answering my various questions regarding the formatting and submission of the thesis;

- Zakria Hussain and Alex Leung, who briefed me on the PinView project and answered my many questions;

- David Barber and Zhaoping Li, who provided critical feedback as examiners at my 1st and 2nd year vivas, and helped me better understand how to do research;

- My friends who supported me during this long process, and those who provided free accommodation when my funding ran out (Nicolas, Richard, Bastien, Loïc)!

My last thanks go to my parents for their continuous and unwavering support, and to which this thesis is dedicated. *Grazie !*

Contents

List of Figures

List of Tables

List of Symbols

β_t	confidence term at time t, page 26
δ	confidence threshold, page 62
Δ_i	sub-optimality of arm i, measured as $f^* - f(i)$, page 23
$(\epsilon_\tau)_\tau$	noise sequence, page 22
$\hat{\lambda}_i$	i^{th} largest eigenvalue of $\mathbf{K}$, page 73
κ	kernel/covariance function, page 31
$\mu_t(i)$	reward estimate at time t for arm i, page 22
$\boldsymbol{\mu}_t$	vector of $\mu_t(i)$ values for all i, page 22
$\nu(i,t)$	number of plays of arm i up to time t, page 22
ρ_d	parameter of the smoothness assumption of BAST, page 45
$\sigma_t(i)$	uncertainty measure at time t for arm i, page 24
$\boldsymbol{\sigma}$	vector of $\sigma_t(i)$ values for all i, page 25
χ_d	value of the kernel product between paths of a tree that have d nodes not in common, page 84
$\mathbf{a}_i$	feature vector of arm i, page 32
B	tree branching factor, page 43
C_T	computational cost at time T, page 65
$\mathcal{D}_t$	training set after t observations, page 22

D	maximum tree depth, page 44
$\mathbf{f}$	vector of $f(i)$ values for all i, page 21
f	(unknown) function that takes an arm index or feature representation in input and returns its mean-reward value, page 21
f^*	optimal f value, page 22
f_t	upper confidence function at time t, page 25
G_T^*	maximum possible information gain after T observations of f, page 73
G_T^g	information gain after T observations chosen sequentially by the information-greedy algorithm, page 73
$\mathcal{I}$	set of arm indices $\{1, \ldots, N\}$, page 22
$\mathcal{I}_t$	set of training data input indices after t observations, page 22
i^*	optimal arm, i.e. arm whose f value is the highest, page 22
$\mathbf{K}$	matrix of kernel products between the elements of $\mathcal{X}$ (when the latter is finite), page 61
$\mathbf{K}_t$	matrix of kernel products between the inputs of $\mathcal{D}_t$, page 31
$\bar{N}$	number of nodes in the tree, page 47
N	number of arms in a bandit problem, page 21
$\tilde{O}$	$u_n = \tilde{O}(v_n) \Leftrightarrow \exists \alpha, \beta > 0, u_n \leq \alpha \log(v_n)^\beta v_n$, page 27
R_T	regret at time step T, page 22
r_t	immediate regret at time step t, page 22
S_t	upper bound on the size of the (reduced) training set after t bandit plays, page 69
U	upper bound on the best reward that can be obtained from a given node, page 43
$\mathcal{X}$	space of arms/inputs, page 32
$\mathbf{X}_t$	matrix of t concatenated training input vectors, page 30
$\mathbf{x}_t$	feature representation of the arm played at time t, page 32

$\mathbf{y}_t$	vector of t concatenated reward observations, page 22
y_t	reward at time t, page 22

1 Introduction

1.1 Motivation

The present work finds its motivation in problems where one must navigate a large search space in order to find an element of that space that is of interest. An everyday example is that of recommender systems on the internet: they must search a library of items in order to find those that will be of interest to a given user. For instance, Amazon.com recommends items to buy to each of its users individually, based on their purchase history; Last.fm creates a personalised radio station based on a user's listening habits. In both cases, the interest in an item is measured by the reaction of the user (a purchase or a click on a "Like" button). We can formalise the recommendation problem by assuming the existence of a function f that maps a representation of a user and of an item to a number that quantifies the interest of the user for the item. The recommender system makes noisy observations of f at each interaction with a user, and uses these observations in order to figure out the areas of the search space where f has high values.

In optimisation tasks, one also needs to find regions of the search space where an unknown function f is high, as the objective is to find the maximum of f. This task falls into the class of problems we consider: the "interest" for an element $\mathbf{x}$ of the search space is measured by $f(\mathbf{x})$. Finding the maximum of a function requires building a model of it through repeated observations. We may choose to observe f values in areas of high uncertainty of the model, in order to improve it (exploration), or in areas where the model predicts high values (exploitation). We say that a space is large when its cardinality is large compared to the maximum possible number of elements for which we can make observations. Other examples from the same class of problems include: Operations Research problems (such as scheduling), in which the search space is a set of solutions to a combinatorial problem, and the interest for a solution is measured by its cost; Decision Processes, in which one makes a sequence of decisions and receives a sum of reward values for each of them; Games, as an application where decisions are moves in a game.

Multi-armed bandit problems are a simple model of the exploration and exploitation dilemma. They take their name from an analogy with slot machines in casinos: one must allocate coins, one at a time, to slot machines, in order to maximise the cumulative payoff in the end of the process (also called the *reward*). It is assumed that, throughout the process, the mean payoffs of the different slot machines are constant. The problem becomes to find the machine with highest mean payoff, so that we lock on to that machine and get the maximum expected payoff from then on. When allocating a coin to a slot machine, we say that we *pull* or that we *play* an *arm* of the bandit. The difficulty in this type of problem does not come from the number of elements in the search space (here we can make observations for each arm at least once) but from the high variability of the payoff: we need many plays of each machine to estimate its mean payoff accurately.

If this sounds as a rather unrealistic problem to tackle in practise, consider the following problem (described on the Wikipedia page for the Gittins index[1]):

We can take two examples from a developing sector, such as from electricity generating technologies: wind power and wave power. If we are presented with the two technologies when they are both proposed as ideas we cannot

[1]http://en.wikipedia.org/wiki/Gittins_index

> *say which will be better in the long run as we have no data, as yet, to base our judgments on. It would be easy to say that wave power would be too problematic to develop as it seems easier to put up lots of wind turbines than to make the long floating generators, tow them out to sea and lay the cables necessary.*
> *If we were to make a judgment call at that early time in development we could be condemning one technology to being put on the shelf and the other would be developed and put into operation. If we develop both technologies we would be able to make a judgment call on each by comparing the progress of each technology at a set time interval such as every three months. The decisions we make about investment in the next stage would be based on those results.*

In the example above, allocating capital to one of the technologies can be seen as a play of a bandit problem. Allocating capital to get more data is exploration, whereas allocating capital to further develop a technology that is thought to be better is exploitation. Exploration aims at reducing the model's uncertainty, exploitation consists in making decisions according to the model. The study of bandit problems did not actually originate in capital or resource allocation problems, but in medical trials. N drugs are proposed to cure a disease and they are supposed to each have the same effect on all patients. The success of a drug can be seen as a Bernoulli random variable characterised by a fixed, unknown mean value. The objective is to heal as many patients as possible, i.e. the cumulative reward is the number of healed patients.

Although we are not interested in such problems, we will see that the techniques used in multi-armed bandit problems in order to balance exploration and exploitation have been used successfully in problems where the number of arms is much larger than the maximum possible number of plays, such as image retrieval in large databases. Such problems are referred to as *many-armed* bandit problems. In order to deal with a large number of arms, we need arms to be related to one another, so that an observation made about one arm can also provide information regarding other arms. Despite differences in terminology, strong parallels have been made with global optimisation techniques. The recent use of Gaussian Processes as a regression tool and a probabilistic model of the smoothness of the function that maps arm representations to mean-reward values is where the bandits and Bayesian optimisation communities have met.

In many applications, the spaces to be searched are particularly large but have a tree structure that make them relatively easy to navigate. Each element is a leaf or, equivalently, a path down the tree. We can summarise the learnings gained from observations about paths that share nodes in common, by storing statistics at the last node they have in common. When searching the space, we start from the root and make a rather small number of decisions (equal to the depth of the tree) on the next child node to move to, based on the statistics of all the child nodes. We can consider each of these decisions as a multi-armed bandit problem. There are as many bandit problems as there are interior nodes, and the number of arms is given by the branching factor.

This strategy has been used in Artificial Intelligence for Games, where we look for sequences of moves from the current state of the game to a final state in which the game is won. For computational reasons, we do not usually get to explore all leaves of the game tree. Instead, we generate a stochastic reward by rolling out the game randomly and receiving +1 when winning, 0 otherwise. Bandit-based approaches to tree search are responsible for significant progress in AI for the game of Go (Gelly and Wang, 2006), in which the number of available moves at each turn is about 5 times greater than in Chess, on average.

Searching trees with large branching factors is computationally challenging. It requires efficiently selecting branches to explore based on their estimated potential (i.e. how good the rewards can be at the leaves) and the uncertainty in the estimates. High depths can be unattainable due to lack of computation time. A tree search algorithm should not waste time exploring sub-optimal branches too frequently, while still exploring them enough in order not to miss the optimal branch because of high noise in the observations (as in game tree search). For this, multi-armed bandit algorithms can be used at each node of the tree in order to guide the selection of branches to explore.

In this work, however, we are interested in applying many-armed bandit algorithms to the search of trees that have large branching factors. Instead of viewing tree search as a sequence of multi-armed bandit problems, we view it as a single many-armed bandit problem in which information can be shared between paths that have nodes in common.

1.2 Outline and contributions

The contributions of this work concern many-armed bandit algorithms that make use of Gaussian Processes to model dependencies between arms, and are designed to focus the exploration of large spaces. In order to present these contributions, this thesis is organised as follows:

- Chapter 2 provides background information on bandits, tree search, and Gaussian Processes. It also contains a review of some related work on techniques that deal with dependencies between arms, on Bayesian optimisation, and on the application of tree search algorithms to the problem of planning in Markov Decision Processes.

- Chapter 3 is divided in two. Section 3.1 presents original work, comprising a formulation of the Gaussian Process Upper Confidence Bound algorithm (GP-UCB) – also referred to as the Gaussian Process Bandits algorithm (GPB) – and an analysis of its computational complexity. The algorithm was first introduced by Dorard, Glowacka, and Shawe-Taylor (2009) and empirical results were obtained by Glowacka et al. (2009). To the best of our knowledge, we were the first to apply Gaussian Processes to bandit problems. Srinivas et al. (2010) subsequently provided a regret analysis of GPB, and their contribution is reviewed in Section 3.2.

  More specifically, the original work presented in this chapter consists of:

  – The formulation of the GPB algorithm, in Section 3.1.1. We also make connections with the well-known UCB1 and LinRel algorithms.

  – A computational analysis of GPB and LinRel, in Section 3.1.2, along with optimisations based on original online computation “tricks” that reduce the algorithms’ theoretical complexity.

  – Additional details to the work of Srinivas et al., in Section 3.2.

- **Chapter 4 (new work)** discusses the application of GPB to tree search. The resulting algorithm, GPTS, considers tree paths as arms of a bandit problem. We show how the algorithm can be efficiently implemented, even when the tree is very large. We provide a computational analysis of GPTS, theoretical guarantees on its performance, and we compare it to related algorithms (UCT, BAST and OLOP).

- **Chapter 5 (new work)** describes a new open source toolbox, "Bats", for programming with bandits and tree search algorithms in Matlab. We designed and used the toolbox to provide empirical observations of the performance of GPB on an Information Retrieval task, and to study how the performance of GPTS was affected by the parameters of synthetic trees it was applied to (branching factor, depth, smoothness of the target).
- Chapter 6 concludes this work and offers ideas for future work.
- Appendix A provides mathematical results taken from the literature and used in this work, as well as proofs of lemmas given in the background review of Chapter 2.

Summaries are given at the beginning of each chapter, thus providing a summary for the whole thesis.

2 Background and literature review

The background review presented in this chapter introduces three main concepts: the stochastic bandit problem, the Gaussian Processes (GP) framework, and bandit-based tree search techniques.

Firstly, we formalise the assumptions that characterise the stochastic bandit problem and we define different measures of performance of a bandit algorithm. We give relationships between these measures and the regret, defined as the expected difference between the best possible cumulative reward and the cumulative reward of the algorithm at time T. After a brief look at the first approaches to bandit games, we focus on a particular class of frequentist algorithms that achieve asymptotically optimal regret growth rate (in the square root of T) and are computationally tractable for finitely many armed bandits – namely the Upper Confidence Bound algorithms. These algorithms establish confidence intervals at each time step for the mean-reward values, and always choose to play the arm with highest upper confidence bound. We give an overview of the regret analysis of one such algorithm, UCB1. We then review LinRel, a UCB-type algorithm that makes use of arm feature descriptions, looks for linear relationships between these and the observed rewards, and establishes pseudo-confidence intervals for the reward values. We also review the extension of this algorithm to non-linear arm-reward relationships through the use of kernel Ridge Regression (kRR).

Secondly, we present the Gaussian Processes Bayesian regression model that will be at the core of GPB, the UCB-type bandit algorithm that we propose in Chapter 3 as an alternative to LinRel. Roughly, this model expresses a prior belief that similar inputs are likely to yield similar outputs. Similarity is modelled by a kernel/covariance function on the inputs. GPs, as a probabilistic model, allow us to model uncertainty, and to update our belief of what the target f may be after we have observed input-output pairs – it has to come relatively close to the sample values (we are only off because of the noise in the observations) but at the same time it has to agree with the level of smoothness dictated by the prior covariance function. In addition to creating a "statistical picture" of f, encoded in the posterior mean, the GP framework gives us error bars for this picture, through the posterior variance. We show that the GP posterior can be expressed in closed form, and also that the Bayesian framework enables us to perform Maximum Likelihood model selection in order to estimate the potential parameters of the covariance function.

Thirdly, we motivate the use of bandit algorithms for problems where the target function f is defined on tree leaves, i.e. for tree search problems – in particular, those with large branching factors that make it crucial to explore branches efficiently. We present the "many-bandits" approach, based on the use of bandit instances at each node that decide which branches to explore. The Bandit Algorithm for Smooth Trees (BAST) builds confidence intervals at all nodes for the maximum reward values that can be attained from these nodes, and adapts to different levels of smoothness of the target function. This approach has sparked significant interest since a special case of BAST, Upper Confidence Trees (UCT), was successfully applied to the Go game tree search and outperformed classical Branch-and-bound approaches used in Chess tree search (such as alpha-beta search, see Gelly, 2007). We mention how BAST has been extended to the problem of planning in Markov Decision Processes, in which a decision maker must take actions sequentially and may have a large number of actions to choose from. Later, in Chapter 4, we will see how GPB can be applied to tree search and how it can provide a useful alternative to BAST.

The background review is followed by a literature review of some related work on Bayesian optimisation (based on GPs), on continuum-armed bandit algorithms and on their application to global optimisation.

2.1 The stochastic bandit problem

2.1.1 Problem statement

The *multi-armed bandit problem* is a simple model of the tradeoff between exploration and exploitation. In an analogy with a slot machine, known as a one-armed bandit, but with multiple arms, a player receives a reward when pulling (or 'playing') an arm selected from a finite set of arms. In the stochastic bandit scenario, the reward is modelled as a sample from a fixed probability distribution associated to the chosen arm. This is in contrast with the 'restless' bandit problem in which the distributions are allowed to change through time, and with the adversarial setting in which the sequence of reward values for each arm are chosen by an adversary (see Bubeck, 2010, and the references therein for an overview of the different types of bandit problems).

The objective of the player is to maximise the collected reward sum (or 'cumulative reward') through iterative plays of the bandit. A good policy to choose arms to play requires optimally balancing the learning of the probability distributions and the exploitation of arms which have been learnt as having high expected rewards. Real-world applications are varied and include for instance advertisement on the web, where pulling an arm corresponds to placing an ad on a given webpage, and rewards are visitor clicks.

Formulation It is assumed that there is a fixed number of arms N, that the reward obtained when playing arm i is a sample from a distribution P_i, unknown to the player, and that successive plays of arm i yield identically and independently distributed (iid) rewards. A stochastic bandit problem is thus characterised by a set of probability distributions $(P_i)_{1\leq i\leq N}$. The vector of means of these distributions is notated $\mathbf{f} = (f(1), \dots, f(N))$ where $f(i) = \mathbb{E}P_i$. As the number of arms is finite (and usually smaller than the number of experiments allowed), it is possible to explore all the possible options (arms) a certain number of times, thus building empirical averages $\mu_t(i)$ estimating $f(i)$ for all i at time step t, i.e. after t plays of the bandit, and to exploit arms with high averages.

Notations Once a policy has been chosen, we denote by $(I_\tau)_\tau$ the stochastic process that corresponds to the sequence of chosen arms at all time steps (these are random variables because the policy is based on the observed rewards, which are stochastic, and the policy can be stochastic itself). We denote by $(i_\tau)_\tau$ a realisation of $(I_\tau)_\tau$, and by $(y_\tau)_\tau$ the sequence of observed rewards. By definition of f and the fact that rewards are iid, we can write $y_\tau = f(i_\tau) + \epsilon_\tau$ where $(\epsilon_\tau)_\tau$ is a martingale difference sequence that we call the *noise sequence*. The set of observations $\mathcal{D}_t = \{(i_1, y_1), \ldots, (i_t, y_t)\}$ up to time t is called the *training data set*. We write $\mathcal{I} = \{1, \ldots, N\}$, $\mathcal{I}_t = \{i_1, \ldots, i_t\}$ for the set of training data input indices, and $\mathbf{y}_t$ for the vector of concatenated reward observations. We thus have, by definition:

$$\mu_t(i) = \hat{\mathbb{E}} P_i = \frac{1}{\nu(i,t)} \sum_{\tau=1\ldots t \text{ s.t. } i_\tau = i} y_\tau \tag{2.1}$$

where $\nu(i,t)$ is the number of plays of arm i up to time t. We write $\boldsymbol{\mu}_t = (\mu_t(1), \ldots, \mu_t(N))$. As the number of times we play the same arm i grows, we expect our *reward estimates* $\boldsymbol{\mu}_t$ to improve and to get closer to $\mathbf{f}$. The *policy* is the set of rules that determine which arm is played at each time step.

2.1.1.1 Performance measures for bandit algorithms

The optimal arm selection policy/strategy $\mathcal{S}^*$, i.e. the policy that yields maximum expected cumulative reward, would consist in selecting arm $i^* = \text{argmax}_i\{f_i\}$ to play at each iteration. We write $f^* = f_{i^*}$. The expected cumulative reward of $\mathcal{S}^*$ at time T (after T iterations) is Tf^*. The performance of a policy $\mathcal{S}$ is assessed by the analysis of its expected cumulative regret at time T, defined as the difference between the expected cumulative reward of $\mathcal{S}^*$ and $\mathcal{S}$ at time T. Note that $\mathcal{S}^*$ is unknown to the agent, and therefore the agent can only attempt to bound the regret based on assumptions on f. The regret can only be computed if we have access to the f^* value for the function f that is used by the environment.

We define the *immediate regret* at time step t as:

$$r_t = f^* - \mathbb{E}\, f_{I_t}$$

where the expectation is taken with respect to the random drawing of the arms (f_{I_t} is a random variable because of the index I_t). The expected cumulative regret R_T, also called *regret*, is defined as:

$$R_T = \sum_{t=1}^{T} r_t$$

We can also consider a performance measure based on the observed reward values, which will depend on the realisation of $(I_t)_t$, and which we call the *empirical regret*:

$$\begin{aligned} r'_t &= f^* - y_t \\ R'_T &= \sum_{t=1}^{T} r'_t \end{aligned}$$

It can be shown that $|R_T - R'_T|$ scales in $O(\sqrt{T})$ with high probability. This is thanks to Azuma's inequality (A.7) applied to the martingale difference sequence $Y_\tau = R_\tau - R'_\tau$ for τ from 1 to T (see Coquelin and Munos, 2007b, for details).

As we will see in the following, bandit algorithms can be applied as a way to focus exploration in optimisation problems. In that case, it is not the cumulative regret we are ultimately interested in, but the *simple regret*, defined as the expected difference between f^* and f_{J_T} where J_T is the arm recommended by the algorithm after T exploration rounds. One way to recommend an arm is to take $J_T = i$ with probability $\nu(i, T)/T$, in which case the simple regret is equal to $\frac{1}{T} R_T$. An algorithm is said to be *no-regret* when this quantity tends to 0 when T tends to infinity.

In the rest of this work, we will focus on bounds on the expected regret, which can then be used to derive other regret bounds (with high probability). A regret bound is said to be *problem-specific* when it involves constants that are specific to the current bandit problem, such as the *sub-optimality* values of arms:

$$\Delta_i = f^* - f(i)$$

Problem-independent bounds, however, do not involve such quantities and hold uniformly over the space of bandit problems with N arms (regardless of the arms' mean-reward values). The bounds also hold uniformly over time.

We are especially interested in the growth rate of these bounds in time, and in the dependency of constants on the size N of the bandit problem.

2.1.1.2 First approaches

The mean-reward estimates defined in Equation (2.1) improve when the number of training data points t increases. A good policy should balance the improvement of these estimates with the exploitation of arms with high empirical means , which are considered likely to give good rewards. A first approach to reach a good balance is the *epsilon-greedy policy* which consists in fixing ϵ and either choosing actions randomly with probability ϵ, or choosing actions greedily otherwise. As the number of iterations increases, our empirical estimates improve and it becomes more important to exploit than to explore. In order to reflect this, we can choose ϵ to be a decreasing function of the number of iterations. However, a disadvantage of this strategy is that it does not adapt its exploration to the problem at hand and to the relative values of the arms' average rewards. For certain arms, it does not seem necessary to refine reward estimates much, if, based on previous observations, we can be fairly confident that these arms are not the best. Auer et al. (2002) report that epsilon-greedy can be made competitive with other algorithms in practise, when tuning ϵ to the problem at hand, but there is no known automated way to obtain good results without prior knowledge of the problem.

Gittins and Jones (1979) proposed another approach, based on Bayesian theory and some further assumptions on the reward distributions. It consists in computing indices, for all arms and at each time step, and always playing the arm with highest index. The Gittins index for arm i at time t is defined as the maximum expected sum of rewards, from time t until the end of the bandit game, that can be attained when starting from arm $I_{t+1} = i$. Gittins and Jones proved this policy was optimal, and Tsitsiklis (2002) later gave a simpler proof of this result. We thus see that in the Bayesian probabilistic approach, playing optimally is a computational problem.[1] The algorithms that we study in this work consider instead the statistical problem of achieving low regret, as a relaxation of the problem of finding the optimal playing strategy, and focus on finding computationally efficient solutions which do not lose asymptotically in comparison with the optimal strategy.

2.1.2 Upper Confidence Bound algorithms

2.1.2.1 Arm selection

A popular policy for balancing exploration and exploitation in bandit problems consists in applying the so-called *Optimism in the Face of Uncertainty (OFU) principle*. First of all, reward estimates $\mu_t(i)$ and *uncertainty measures* $\sigma_t(i)$ are maintained for each arm.

[1]This can be very long or even intractable, but in the case of Bernoulli reward distributions the computations can be performed exactly and efficiently for modest values of T.

We write $\boldsymbol{\sigma}_t = (\sigma_t(1), \dots, \sigma_t(N))$. $\mu_t(i)$ is usually the empirical average of the rewards observed for i. The expression for $\sigma_t(i)$ is chosen along with a positive and monotonic factor β_t, so that the probability that $f(i)$ is outside of its *confidence interval* of the form $[\mu_t(i) - \sqrt{\beta_t}\sigma_t(i); \mu_t(i) + \sqrt{\beta_t}\sigma_t(i)]$ drops quickly in time. The random quantities considered here are $\mu_t(i)$ and $\sigma_t(i)$, whereas f is fixed.

The OFU principle states that the arm to be played at each time step is the one for which the upper bound of the confidence interval is the highest. As with the epsilon-greedy algorithm, we want to decrease the amount of exploration in time, but when we do decide to explore, we should rather explore promising arms rather than any arm. We define the *upper confidence function at time t* as $f_t = \mu_t + \sqrt{\beta_t}\sigma_t$. Each implementation of the OFU principle will specify its own expression for μ_t, β_t, σ_t, but will always pick at any time step the arm that maximises the upper confidence function.

UCB1 The UCB1 algorithm which implements the OFU principle has been shown by Auer et al. (2002) to achieve optimal regret growth-rate for problems with independent arms. The setting here is agnostic as no assumption is made on the nature of the reward distributions, other than the fact that they have bounded support (and so can be transformed into a problem with support in $[0, 1]$). UCB1 takes μ_t to be the empirical average, and:

$$\beta_t = 2\log(t) \qquad (2.2)$$

$$\sigma_t^2(i) = \frac{1}{\nu(i,t)} \qquad (2.3)$$

It proceeds as follows:

- Initialisation:
  - Play each arm once
  - Define $\boldsymbol{\mu}_N$ and $\boldsymbol{\sigma}_N$ from observed data $(i_1, y_1), \dots, (i_N, y_N)$
  - $t = N$
- Loop:
  - Play $i_{t+1} = \text{argmax}_{1 \le i \le N} f_t(i)$ and break ties arbitrarily
  - Get reward y_{t+1}, which defines $\boldsymbol{\mu}_{t+1}$ and $\boldsymbol{\sigma}_{t+1}$
  - $t = t + 1$

Confidence intervals Under the assumption that the support of the P_i reward distributions is in $[0,1]$, Hoeffding's inequality (A.8) bounds the probability that, given the number of times n that arm i has been played, the empirical average $\mu_t(i)$ goes further away from the true expectation $f(i)$ than a given distance:

$$\mathbb{P}(f(i) \geq \mu_t(i) + \epsilon) = \mathbb{P}(f(i) \leq \mu_t(i) - \epsilon) \leq \exp(-2\epsilon^2 n) \tag{2.4}$$

Applying this with $\epsilon = \sqrt{\beta}_t \sigma_t(i)$, the probability of the upper confidence bound being below $f(i)$ is:

$$\forall i, \mathbb{P}(f(i) \geq \mu_t(i) + \sqrt{\beta}_t \sigma_t(i)) \leq t^{-4} \tag{2.5}$$

By symmetry, the same holds for the probability of being below the lower confidence bound.

Interpretation The arm selection problem can be seen as an active learning problem, as we are able to decide which data we want to observe. By selecting arms with highest upper confidence bound, we want to learn f values accurately when we estimate that they can be potentially high, and we do not care much if our estimates are inaccurate as long as we are confident that the true function values are low. β_t balances exploration and exploitation: the bigger it is, the more it favours points with high $\sigma_t(i)$ (exploration), while if $\beta_t = 0$, the algorithm is greedy and is thus only doing exploitation. For this reason, we say that f_t is the sum of an exploitation term (the reward estimate $\mu_t(i)$), and an exploration term (the uncertainty measure $\sigma_t(i)$) times a confidence term β_t . By having β_t grow with time, we ensure that asymptotically we will continue to play even the worst performing arms. Also note that the bigger β_t, the wider the confidence intervals and the more likely rewards are within their confidence intervals.

Note that we can also relate the immediate regret to the confidence width. If all f values are within their confidence intervals:

$$\begin{aligned} f(i^*) &\leq f_t(i^*) \\ &\leq f_t(i_{t+1}) \text{ by definition of } i_{t+1} \\ f(i^*) - f(i_{t+1}) &\leq \sqrt{\beta}_t \sigma_t(i_{t+1}) + \mu_t(i_{t+1}) - f(i_{t+1}) \end{aligned}$$

and thus:

$$r_{t+1} \leq 2\sqrt{\beta}_t \sigma_t(i_{t+1}) \tag{2.6}$$

We thus see that reducing the uncertainty of our predictions has an effect on the regret.

Translating and scaling rewards Rewards are usually taken in $[0, 1]$ in the bandit literature, but it may be more convenient to have output spaces centred around 0 (when dealing with probabilistic models, for instance), and the rewards may take values in a wider range (as in Section 2.4.2 for instance). We assume that the expression for μ_t is linear in $\mathbf{y}_t$ and that the expression for σ_t does not involve the observed y values, which will be the case for all algorithms considered in this work. We thus see that translating rewards does not affect the order of the f_t values, hence it does not affect the algorithm, but if we scale rewards by a factor a, we should scale β_t by a^2.

Extensions We can improve the way that we build the confidence intervals when we know the type of the P_i distributions. For instance, the UCB1-normal algorithm also proposed by Auer et al. (2002) was devised for the case of Gaussian reward distributions, and it takes advantage of the extra knowledge on the P_i's in order to derive tighter confidence intervals. It is also possible to obtain better intervals when we do not know the type of these distributions, by taking into account the empirical variance of the distributions, which is what the UCB-V algorithm does (see Audibert et al., 2009).

2.1.2.2 Overview of the regret analysis

UCB1 achieves a regret with problem-specific upper bound in $O(\log(T))$, and problem-independent upper bound in $\tilde{O}(\sqrt{T})$, where we write $u_n = \tilde{O}(v_n)$ if there exist $\alpha, \beta > 0$ such that $u_n \leq \alpha \log(v_n)^\beta v_n)$. This matches the lower bound rate of Lai and Robbins (1985) where they focused on Bernoulli reward distributions.The strategy used to derive the first bound is all the more interesting as it inspired the regret analyses of other algorithms mentioned in Sections 2.3.2 and 2.4.1.2. We first rewrite the regret as follows:

$$R_T = \sum_{i=1}^{N} \Delta_i \mathbb{E}\nu(i, T) \tag{2.7}$$

Hence, we can bound the regret by bounding the expected number of times each arm has been selected after T rounds. $\mathbb{E}\nu(i, T)$ is bounded by $l_i(T)$ (arbitrary) plus the sum over t of probabilities of the event $e_1(i, t)$ defined as playing arm i at round t after we have played it already more than $l_i(T)$ times in the past (Lemma 1).

$$\mathbb{E}\nu(i, T) \leq l_i(T) + \sum_{t=2}^{+\infty} \mathbb{P}(e_1(i, t))$$

We want $l_i(T)$ to grow with T, and the rest to be bounded by a constant. We expect that $\mathbb{P}(e_1(i, t))$ decreases when $l_i(T)$ increases, as the accuracy of the reward estimate for i will increase hence it will be easier to see it is a suboptimal arm.

We write $Y_{i,1}, Y_{i,2}, \ldots$ the rewards obtained for successive plays of arm i, and $\bar{Y}_{i,n}$ the average of the n first rewards. In order to determine the rate at which $\mathbb{P}(e_1(i,t))$ decreases, we first state Lemma 2 (see proof in Appendix A.2): $e_1(i,t)$ implies that there exist ν_i and ν^* such that:

$$\bar{Y}_{i,\nu_i} + \sqrt{\frac{\beta_t}{\nu_i}} \quad \geq \quad \bar{Y}_{i^*,\nu^*} + \sqrt{\frac{\beta_t}{\nu^*}}$$

In that case, Lemma 3 states that if $f^* \geq f(i) + 2\sqrt{\frac{\beta_t}{\nu_i}}$, then either f^* is above its upper confidence bound or $f(i)$ is below its lower confidence bound – which are events we can bound the probability of when summing over all possible values that ν_i and ν^* can take (see Inequality 2.5). Therefore, it is all down to having $\nu_i \geq l_i(T)$ high enough so that this condition is always met. Choosing $l_i(T) = \left\lceil \frac{8\log(T)}{\Delta_i^2} \right\rceil$, we thus have:

$$\begin{aligned}
\sum_{t=2}^{+\infty} \mathbb{P}(e_1(i,t)) \quad &\leq \quad \sum_{t=2}^{+\infty} \sum_{\nu^*=0}^{t-1} \sum_{\nu_i=l_i(T)}^{t-1} 2t^{-4} \\
&\leq \quad 2\sum_{t=2}^{+\infty} t^2 t^{-4} \\
&\leq \quad \frac{\pi^2}{3}
\end{aligned}$$

Note that the application of Hoeffding's inequality requires the number of points in the empirical average to be fixed. Therefore, we had to consider all the values that ν_i and ν^* could take, so that they became indices instead of random variates.

From this result follows that:

$$\mathbb{E}R_T \leq 8\sum_{i \neq i^*} \frac{1}{\Delta_i} \log(T) + (1 + \frac{\pi^2}{3}) \sum_{i=1}^{N} \Delta_i \tag{2.8}$$

In order to get a problem-independent bound, we follow the trick used in the fourth step of the proof of Theorem 2.2 of Bubeck (2010). We first write $\mathbb{E}R_T = \sum_i \Delta_i \sqrt{\mathbb{E}\nu(i,T)}\sqrt{\mathbb{E}\nu(i,T)}$.

- $\Delta_i \sqrt{\mathbb{E}\nu(i,T)}$ is bounded by $\sqrt{8\log(T) + 1 + \frac{\pi^2}{3}}$ (using the fact that $\Delta_i \leq 1$ because rewards are bounded in $[0,1]$) which is an expression that is independent of i;

- $\sum_i \sqrt{\mathbb{E}\nu(i,T)}$ is bounded by $N\sqrt{\frac{1}{N}\sum_i \mathbb{E}\nu(i,T)}$ owing to the concavity of the square root

Using the fact that $\sum_i \nu(i,T) = T$, we thus have:

$$\mathbb{E}R_T \leq \sqrt{N\ T\ (8\log(T) + 1 + \frac{\pi^2}{3})} \tag{2.9}$$

2.1.3 Arm-reward regression

In practise, the arms' mean reward values are often related to one another and correlations are observed. Arm feature representations can be given, and the mean reward function f on all arms can be modelled as a function in the feature space. We thus get, from one play, information about all the other arms, which allows us to deal with the many-armed bandit problem that we mentioned in Chapter 1. Here, we review the LinRel approach (Auer, 2003) which represents arms by feature vectors $(\mathbf{a}_i)_{1\leq i\leq N}$ and looks for a linear mapping from a vector-space $\mathcal{X}$ to the mean-reward values. We give an overview of some other approaches in Section 2.4.1.

2.1.3.1 Linear Regression

We start by providing some background on linear regression, which will be used by LinRel to learn mean rewards as a function of arms' feature representations.

In our learning setting, we observe $(\mathbf{x}, y)$ pairs that we model as samples from a fixed (but unknown) probability distribution $p(\mathbf{x}, y)$. The aim of regression is to find a functional relationship between the inputs and outputs that are given for us to observe. For simplicity, we assume the existence of a linear relationship: there exists $\mathbf{w}$ such that for any given $\mathbf{x}$, the output is drawn from a unimodal and symmetric distribution with mean $\mathbf{w}^\mathsf{T}\mathbf{x}$. The latter distribution accounts for the noise that may come from the observations or from the fact that the true relationship between $\mathbf{x}$ and y is not linear. Our aim is to estimate $\mathbf{w}$ from training data $\mathcal{D}_t = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_t, y_t)\}$, and to make predictions by computing $\mathbf{w}^\mathsf{T}\mathbf{x}$. We call the regression model *parametric* because it involves a parameter vector $\mathbf{w}$.

Least Squares In practise, we can expect that there will not exist $\mathbf{w}$ such that for all t we will have $\forall 1 \leq \tau \leq t, y_\tau = \mathbf{w}^\mathsf{T}\mathbf{x}_\tau$ (unless the functional is indeed linear and is observed without noise). Least Squares consists in looking for $\mathbf{w}$ that explains the past observations well in the sense that it minimises a certain measure of the errors of the predictions that would be given by $\mathbf{w}$. One way to define this error on $\mathcal{D}_t$ is by taking the sum of squared differences : $\sum_{\tau=1}^{t}(y_\tau - \mathbf{w}^\mathsf{T}\mathbf{x}_\tau)^2$. We denote by $\mathbf{X}_t$ the concatenation of all vectors of inputs in training data. Finding $\mathbf{w}_t$ that minimises the sum of squared differences is a simple optimisation problem which can be solved by setting the differential of the objective to $\mathbf{0}$:

$$\begin{aligned}\frac{d}{d\mathbf{w}}((\mathbf{y}_t - \mathbf{X}_t^\mathsf{T}\mathbf{w})^\mathsf{T}(\mathbf{y}_t - \mathbf{X}_t^\mathsf{T}\mathbf{w})) &= -2(\mathbf{y}_t - \mathbf{X}_t^\mathsf{T}\mathbf{w})^\mathsf{T}\mathbf{X}_t^\mathsf{T} \\ &= \mathbf{0} \\ \mathbf{w}_t &= (\mathbf{X}_t\mathbf{X}_t^\mathsf{T})^{-1}\mathbf{X}_t\mathbf{y}_t\end{aligned}$$

if $\mathbf{X}_t\mathbf{X}_t^\mathsf{T}$ is invertible.

We use the $\mu_t(\mathbf{x})$ notation for the regressor $\mathbf{w}^\mathsf{T}\mathbf{x}$. Indeed, the previous definition of μ_t given in Equation (2.1) is a special case corresponding to a space $\mathcal{X}$ that is the set of indicator vectors with N components (one component has value equal to one, the others are all zero): $\mathbf{w}^\mathsf{T}\mathbf{x}$ is thus equal to the empirical average of the outputs we observed when having $\mathbf{x}$ in input.

2.1.3.2 Extension to non-linearly transformed feature spaces

If we do not believe f to be a linear function, we can choose to perform linear regression in a transformed feature space $\phi(\mathcal{X})$ defined by a mapping ϕ to an n-dimensional space, such that we believe f to be linear in that space.[2] Performing linear regression in the transformed space is a way to perform non-linear regression in the original space. Thus, our predictions are now $\mathbf{w}^\mathsf{T}\phi(\mathbf{x})$ where $\mathbf{w}_t = (\phi(\mathbf{X}_t)\phi(\mathbf{X}_t)^\mathsf{T})^{-1}\phi(\mathbf{X}_t)\mathbf{y}_t$.

We show that the predictor can be expressed only in terms of inner products involving $\mathbf{x}$ and the inputs in training data. For this, we use the fact that $\mathbf{A}(\mathbf{A}^\mathsf{T}\mathbf{A})^{-1} = (\mathbf{A}\mathbf{A}^\mathsf{T})^{-1}\mathbf{A}$:

$$\begin{aligned}\phi(\mathbf{x})^\mathsf{T}\mathbf{w} &= \phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{X}_t)(\phi(\mathbf{X}_t)^\mathsf{T}\phi(\mathbf{X}_t))^{-1}\mathbf{y}_t \\ &= (\phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{x}_1)\dots\phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{x}_t))\begin{pmatrix}\phi(\mathbf{x}_1)^\mathsf{T}\phi(\mathbf{x}_1) & \dots & \phi(\mathbf{x}_1)^\mathsf{T}\phi(\mathbf{x}_t) \\ \vdots & \ddots & \vdots \\ \phi(\mathbf{x}_t)^\mathsf{T}\phi(\mathbf{x}_1) & \dots & \phi(\mathbf{x}_t)^\mathsf{T}\phi(\mathbf{x}_t)\end{pmatrix}^{-1}\mathbf{y}_t\end{aligned}$$

[2]This is only interesting for non-linear mappings.

Kernels A kernel κ is a function of two variables such that there exists ϕ such that $\forall(\mathbf{x}, \mathbf{x}'), \kappa(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^{\mathsf{T}}\phi(\mathbf{x}')$. The value of κ at $(\mathbf{x}, \mathbf{x}')$ is called the *kernel product* of $\mathbf{x}$ and $\mathbf{x}'$. We write $\mathbf{K}_t$ for the matrix of kernel products between the inputs of $\mathcal{D}_t$, and $\mathbf{k}_t$ the operator that associates to $\mathbf{x}$ the vector of kernel products between $\mathbf{x}$ and the inputs of $\mathcal{D}_t$, so that the regressor associated to our kernel can be written:

$$\mu_t(\mathbf{x}) = \mathbf{k}_t(\mathbf{x})^{\mathsf{T}}\mathbf{K}_t^{-1}\mathbf{y}_t$$

Remarks

- When κ is a kernel, $\mathbf{K}_t$ is a positive semi-definite matrix, and conversely, if $\mathbf{K}_t$ is a positive semi-definite matrix for any t and any set of inputs $(\mathbf{x}_\tau)_{1\leq\tau t}$, then κ is a kernel.
- The kernel corresponding to $\forall\mathbf{x}, \phi(\mathbf{x}) = \mathbf{x}$ is called the linear kernel ($\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathsf{T}}\mathbf{x}'$) as it leads to linear regression. If we take features that are powers of the components of $\mathbf{x}$, we can perform polynomial regression. We will say more on the choice of κ in the next section, where we review the Gaussian Process model for which the regressor has the same expression. We will give examples of common kernels for which the associated feature spaces can be infinite dimensional, and we will see that we do not need to work with an explicit representation of $\phi(\mathbf{x})$.
- Predictions can be computed either in the *weight-space view*, also called the *primal*, as the inner product between two vectors of n components ($\mathbf{w}_t$ and $\phi(\mathbf{x})$), or they can be computed in the *dual* as the inner product between two vectors of t components ($\mathbf{k}_t(\mathbf{x})$ and $\mathbf{K}_t^{-1}\mathbf{y}_t$).

2.1.3.3 Regularisation

In kernel regression, we do not control the number of dimensions of the underlying feature space, and it can actually be much larger than the number of observed data points. As a consequence, there is a risk of over-fitting the data when simply looking for a $\mathbf{w}_t$ vector that brings the least squares error on the training set to 0. This does not mean that we are learning f accurately. Perhaps the simplest illustration of this is the existence, for any training set $(x_\tau, y_\tau)_{1\leq\tau\leq t}$, of a polynomial p of degree t such that $\forall\tau, p(x_\tau) = y_\tau$. If the data was generated from a function f to which noise was added, doing this will amount to learning noise, instead of learning a pattern. The least squares error of a regressor that would know the true f would be strictly positive (due to the noise), whereas the error of p would be 0.

One solution to this issue consists in penalising $\mathbf{w}$ vectors that are too complex, i.e. that have a high norm. We thus consider a new objective for $\mathbf{w}_t$:

$$\mathbf{w}_t = \text{argmin}_{\mathbf{w}} \left|\left|\mathbf{y}_t - \mathbf{X}_t^\mathsf{T}\mathbf{w}\right|\right|^2 + \rho \left|\left|\mathbf{w}\right|\right|^2$$

This is called *kernel Ridge Regression (kRR)*. The value for ρ that will provide the best weight vectors depends on the particular learning problem that is being considered. It can chosen by cross-validation: we try different splittings of the training data in two and, for each, we learn from the first subset of the data with different values of ρ and assign them a score based on the error we measure on the second subset of the data; the ρ value with lowest overall score is chosen. Values of ρ that are too small or too big will have high scores.

The solution to this new optimisation problem is:

$$\mathbf{w}_t = (\mathbf{X}_t\mathbf{X}_t^\mathsf{T} + \rho\mathbf{I}_n)^{-1}\mathbf{X}_t\mathbf{y}_t$$

which implies:

$$\mu_t(\mathbf{x}) = \mathbf{k}_t(\mathbf{x})^\mathsf{T}(\mathbf{K}_t + \rho\mathbf{I}_t)^{-1}\mathbf{y}_t$$

We invite the reader to refer to Shawe-Taylor and Cristianini (2004) for a complete introduction to kernel methods, for a review of their applications to regression and other Machine Learning problems, and for examples of common kernels.

2.1.3.4 The LinRel algorithm

We write $\mathbf{a}_i$ for the feature vector of arm i, and we model the mean reward function f as a linear function in the space of arms $\mathcal{X} = \{\mathbf{a}_1, \ldots, \mathbf{a}_N\}$. We write f as a function of an arm's feature representation, or equivalently as a function of an arm's index: $f(i) = f(\mathbf{a}_i)$. Again, the reward support is assumed to be bounded in $[-1, 1]$. We write $\mathbf{x}_t = \mathbf{a}_{i_t}$ so that the training data set is $\mathcal{D}_t = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_t, y_t)\}$.

LinRel (Auer, 2003) adopts the same policy as UCB1 but defines its confidence intervals differently. μ_t is chosen to be the kRR regressor determined from

$\mathcal{D}_t$. Thus, the reward estimate can be written as a weighted sum of previous rewards: $\mu_t(\mathbf{x}) = \boldsymbol{\alpha}_t(\mathbf{x})^\mathsf{T}\mathbf{y}_t$ where $\boldsymbol{\alpha}_t(\mathbf{x})^\mathsf{T} = \mathbf{k}_t^\mathsf{T}(\mathbf{x})(\mathbf{K}_t+\rho\mathbf{I}_t)^{-1}$. $\sigma_t(\mathbf{x})$ is taken to be $||\boldsymbol{\alpha}_t(\mathbf{x})|| /\sqrt{2}$ When the horizon T is fixed, β_t can be replaced by β_T and, because $t \leq T$ and β is strictly increasing, the f values will still be within their confidence intervals with high probability. As a consequence, the confidence term can become a constant that can be tuned to the problem at hand. Otherwise, Auer proposes to take $\beta_t = 2\log(2Nt/\delta)$ when rewards are bounded in $[0, 1]$.

From one play, we learn about all arms. The reward estimates and uncertainty measures need to be updated for all arms. While UCB1 needed to play each arm once in the initialisation phase, in order to define all the $\sigma_t(i)$ values, LinRel only needs to have played one (randomly chosen) arm:

- Initialisation:
  - Play i_1 chosen randomly
  - Get reward y_1, which defines $\boldsymbol{\mu}_1$ and $\boldsymbol{\sigma}_1$
  - $t = 1$
- Loop:
  - Play $i_{t+1} = \text{argmax}_{1\leq i\leq N} f_t(i)$ and break ties arbitrarily
  - Get reward y_{t+1}, which defines $\boldsymbol{\mu}_{t+1}$ and $\boldsymbol{\sigma}_{t+1}$
  - $t = t + 1$

Remarks on theory If the y_τ's were independent variables and if $\boldsymbol{\alpha}_t(\mathbf{x})$ did not depend on them (through the fact that the observed outputs influence the choice of inputs), the variance of this estimate would be bounded by $\sigma_t^2(\mathbf{x}) = ||\boldsymbol{\alpha}_t(\mathbf{x})||^2 /2$, using the fact that the variance of any random variable bounded in $[-1, 1]$ is at most $1/2$. However, the $(y_\tau)_{1\leq\tau\leq t}$ are actually not independent since past rewards influence future choices. This is why Auer devised the SupLinRel algorithm, which calls LinRel as a subroutine with training sets $\{(\mathbf{x}_\tau, y_\tau)_{\tau\in\Psi(t)}\}$ where the $\Psi(t)$'s are designed so that the $(y_\tau)_{\tau\in\Psi(t)}$ are independent variables. In practise, LinRel performs at least as well as SupLinRel. Dani et al. (2008) later proved a regret upper-bound in $\tilde{O}(\sqrt{T})$ for a similar algorithm called ConfidenceBall that does not require $\mathcal{X}$ to be finite. In the following we focus on LinRel.

2.2 Bayesian regression models

We now introduce a regression method in which our belief on f is formalised with a probabilistic model, namely the Gaussian Process model. GPs can be seen as probability distributions over functions that say, roughly, that similar inputs are likely to yield similar outputs. The similarity between inputs is defined by a kernel/covariance function (the two terms are equivalent).

We will then be able to introduce, in the next chapter, a new bandit algorithm based on GP regression, as an alternative to LinRel which is based on kRR. Actually, we will see that the expression for the GP regressor is the same as for LinRel. The advantage of probabilistic models is that they model uncertainty in a principled way. This is useful in order to trade exploration and exploitation based on the uncertainty of our predictions, but also for other things such as choosing parameters of the model by maximising the likelihood of the observations.

We refer the reader to Bishop (2006) for a complete introduction to probabilistic models in Machine Learning, and to Rasmussen and Williams (2006) for more information on the Gaussian Processes model.

2.2.1 Parametric models

2.2.1.1 Bayesian learning theory

As we saw previously, we see input-output observations as realisations of random variables. Furthermore, we make assumptions on the nature of the probability distribution of the output conditioned on the input, with density notated $p(y|\mathbf{x})$ for a realisation $(\mathbf{x}, y)$ of the input-output pair of random variables (we do not need to make assumptions on the marginal distribution of inputs, $p(\mathbf{x})$, in order to make predictions). These models are said to be *generative* as they provide an explanation on how the data was generated, in the form of a probability distribution we could draw data from. In the case of parametric models, the two random variables are assumed to depend on an additional random variable, so that we can write:

$$p(y|\mathbf{x}) = \int_{\mathbf{w}} p(y|\mathbf{x}, \mathbf{w}) p(\mathbf{w}) d\mathbf{w}$$

where $p(y|\mathbf{x}, \mathbf{w})$ is fixed by the model and $\mathbf{w}$ is called the parameter of the model.

Ultimately, we are interested in making predictions on unseen data, i.e. in mapping a new input $\mathbf{x}$ given by the real world to a predicted output y_{pred}. This can be done by taking the most likely value of y:

$$y_{pred} = \text{argmax}_y \, p(y|\mathbf{x})$$

The latter distribution is called the *marginal likelihood*. When it is unimodal and not skewed, the most likely output coincides with the mean of the marginal likelihood. The standard deviation, as a dispersion indicator, gives a measure of the "uncertainty" of our prediction. The objective of learning is to reduce this uncertainty by the observation of data, i.e. samples from $p(\mathbf{x}, y)$ given by the real world.

In Bayesian learning theory, the probabilistic formalism is used to encode our expectations on what $\mathbf{w}$ may be, also called our 'belief'. Our *prior* belief $p(\mathbf{w})$ is readjusted after observing data $\mathcal{D}$, according to Bayes rule:

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})}$$

The new distribution $p(\mathbf{w}|\mathcal{D})$ is called the *posterior*, as it characterises our belief on $\mathbf{w}$ a posteriori (after observing data). We say that the posterior is equal to the prior multiplied by the *likelihood* of the data (also called the *evidence*) and divided by a normalising constant, also called the *marginal probability* of the data. Note that the posterior may be of a different family to the prior distribution, because the latter is multiplied by the likelihood. The prior and posterior are said to be *conjugate distributions* and the family of the prior distribution is the *conjugate prior* of the family of the likelihood. For instance, the Beta distribution is the conjugate prior of the Bernoulli distribution, and the Gaussian distribution is conjugate to itself.

The assumptions we made in our model, in the form of distributions $p(y|\mathbf{x}, \mathbf{w})$ and $p(\mathbf{w})$, imply that:

$$p(y|\mathbf{x}, \mathcal{D}) = \int_{\mathbf{w}} p(y|\mathbf{x}, \mathbf{w})p(\mathbf{w}|\mathcal{D})d\mathbf{w}$$

Making predictions using the predictive distribution $p(y|\mathbf{x}, \mathcal{D})$ is referred to as *probabilistic inference*. We consider as regressor $\mu(\mathbf{x})$ the most likely output according to this distribution. Again, when it is unimodal and not skewed, it coincides with its mean.

2.2.1.2 Bayesian linear regression

We assume a linear relationship between input and output random variables, observed with noise:

$$y = \mathbf{w}^\mathsf{T}\mathbf{x} + \epsilon \tag{2.10}$$

The noise is assumed to be Gaussian: $\epsilon \sim \mathcal{N}(0, s^2_{\text{noise}})$, which gives:

$$p(y|\mathbf{x},\mathbf{w}) \propto \exp\left(-\frac{\left|y - \mathbf{w}^\mathsf{T}\mathbf{x}\right|^2}{2s^2_{\text{noise}}}\right)$$

Because of the problem of over-fitting caused by the noise in our observations, we model our prior belief on $\mathbf{w}$ with a zero-mean multivariate Gaussian distribution: we believe that a small norm is more probable, and the probability drops quickly as the norm gets bigger. We write $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$. We assume that dimensions are independent of each other, and without further knowledge on the relative importance of each dimension, $\mathbf{\Sigma}$ can be taken to be $\lambda \mathbf{I}_t$. Thus, the probability density for $\mathbf{w}$ can be written

$$p(\mathbf{w}) \propto \exp\left(-\frac{1}{2\lambda}\left|\left|\mathbf{w}\right|\right|^2\right)$$

We now determine the posterior, which we know will be Gaussian. The likelihood is a product of one-dimensional independent Gaussians:

$$\begin{aligned} p(\mathcal{D}_t|\mathbf{w}) &= \prod_{i=1}^{t} p(y_i|\mathbf{x}_i,\mathbf{w})p(\mathbf{x}_i) \\ &\propto \exp\left(-\sum_i \frac{\left|y_i - \mathbf{w}^\mathsf{T}\mathbf{x}_i\right|^2}{2s^2_{\text{noise}}}\right) \\ &\propto \exp\left(-\frac{\left|\left|\mathbf{y}_t - \mathbf{X}_t^\mathsf{T}\mathbf{w}\right|\right|^2}{2s^2_{\text{noise}}}\right) \end{aligned}$$

The posterior is obtained by multiplying this by the prior and normalising. With a bit of algebra we get:

$$p(\mathbf{w}|\mathcal{D}_t) \propto \exp\left(-\frac{\left|\left|\mathbf{y}_t - \mathbf{X}_t^\mathsf{T}\mathbf{w}\right|\right|^2}{2s^2_{\text{noise}}} - \frac{1}{2\lambda}\left|\left|\mathbf{w}\right|\right|^2\right)$$

The mean of this Gaussian is the value $\mathbf{w}_t$ which maximises this, also called the *Maximum A Posteriori* (MAP) estimate of $\mathbf{w}$. The objective is similar to the one for regularised Least Squares regression in the primal, and by setting the derivative of its log to zero we get:

$$\mathbf{w}_t = (\mathbf{X}_t\mathbf{X}_t^\mathsf{T} + \frac{s^2_{\text{noise}}}{\lambda}\mathbf{I}_n)^{-1}\mathbf{X}_t\mathbf{y}_t$$

We use this expression in order to rewrite the previous expression of the posterior as follows:

$$\begin{aligned} p(\mathbf{w}|\mathcal{D}_t) &\propto \exp(-\frac{1}{2s^2_{\text{noise}}}(\mathbf{w}-\mathbf{w}_t)^\intercal(\mathbf{X}_t\mathbf{X}_t^\intercal + \frac{s^2_{\text{noise}}}{\lambda}\mathbf{I}_n)(\mathbf{w}-\mathbf{w}_t)) \\ \mathbf{w}|\mathcal{D}_t &\sim \mathcal{N}(\mathbf{w}_t, s^2_{\text{noise}}(\mathbf{X}_t\mathbf{X}_t^\intercal + \frac{s^2_{\text{noise}}}{\lambda}\mathbf{I}_n)^{-1}) \end{aligned}$$

The predictive distribution $p(y|\mathbf{x},\mathcal{D}_t)$ at $\mathbf{x}$ is obtained by multiplying the $p(\mathbf{w}|\mathcal{D}_t)$ posterior by $p(y|\mathbf{x},\mathbf{w})$ and integrating the result over all possible $\mathbf{w}$ vectors. From Equation (2.10) and by symmetry, we see that this is a Gaussian with mean equal to $\mu_t(\mathbf{x}) = \mathbf{w}_t^\intercal\mathbf{x}$: Bayesian linear regression is equivalent to performing linear regression with the MAP estimator of $\mathbf{w}$. When the noise variance goes to 0 or when λ goes to infinity, this is equivalent to Least Squares regression.

2.2.2 Non-parametric models: Gaussian Processes

2.2.2.1 Definition of a Gaussian Process

A Gaussian Process is a collection of random variables, any finite number of which have a joint Gaussian distribution. This condition that the random variables must satisfy is also called the *consistency property*. Let us write F for one such collection. In particular, we are interested in collections indexed by $\mathcal{X}$, and denote by $F_\mathbf{x}$ the random variable associated to $\mathbf{x} \in \mathcal{X}$. A realisation f of F is a collection of real values indexed by the input space. This is analogous to a function from $\mathcal{X}$ to $\mathbb{R}$. As a consequence, GPs are a way to represent our belief of what an unknown function may be.

The Gaussian distributions on any finite number of random variables still need to be specified. For this, we characterise a GP by a mean function of one variable $m(\mathbf{x})$ and a covariance function of two variables $\kappa(\mathbf{x},\mathbf{x}')$ in $\mathcal{X}$ such that, for any finite number of elements $\{\mathbf{x}_1,\ldots,\mathbf{x}_t\}$ of $\mathcal{X}$, we write $\mathbf{m}_t$ for the vector of mean values, $\mathbf{f}_t$ a vector of realisations of $(F_{\mathbf{x}_1},\ldots,F_{\mathbf{x}_t})$, and we have:

$$\mathbf{f}_t \sim \mathcal{N}(\mathbf{m}_t, \mathbf{K}_t) \tag{2.11}$$

We summarise this by simply writing $f \sim \mathcal{GP}(m,\kappa)$. Covariance functions have the same properties as kernel functions, and in the rest of this work we will use both terms equivalently.

Interpretation GPs can be seen as extensions of multi-variate Gaussians to an infinite number of variables (an N-variate Gaussian is actually a distribution over functions defined on spaces of exactly N elements), where the mean has an infinite number of components, and the covariance has an infinite number of rows and columns. Also note that, in informal terms, the closer two given inputs, the more likely their function values will also be close. The covariance function takes the role of modelling the smoothness of a GP function. This comes from the fact that, given two inputs $\mathbf{x}_a$ and $\mathbf{x}_b$, the probability density for their respective function values is a 2-variate Gaussian with covariance matrix

$$\begin{pmatrix} \kappa(\mathbf{x}_a, \mathbf{x}_a) & \kappa(\mathbf{x}_a, \mathbf{x}_b) \\ \kappa(\mathbf{x}_b, \mathbf{x}_a) & \kappa(\mathbf{x}_b, \mathbf{x}_b) \end{pmatrix}$$

By specifying how much function values co-vary, we express a belief on the smoothness of a function.

2.2.2.2 Inference

We assume that y is a noisy observation of a functional of $\mathbf{x}$: $y = f(\mathbf{x}) + \epsilon$. Furthermore, we assume white Gaussian noise, as we did before. Instead of assuming that f is linear and putting a prior distribution on the weight vectors, we characterise our prior belief of what the function values may be with a Gaussian Process with zero mean and covariance function κ. In this model, the form of the functional is not specified by a parameter – as it was in the linear model with the weight vector – and the model is therefore said to be *non-parametric*. In other terms, if one considers the collection of random variables that consists of the possible values of $f(\mathbf{x})$ for all possible input $\mathbf{x}$, in the linear model these variables are linked to each other through the parameter $\mathbf{w}$, whereas in the GP model they are linked to each other through the consistency property.

We write $(\mathbf{y}_t\ z)^\mathsf{T}$ for the vector of output variates associated to the training inputs and the function variate associated to a new input $\mathbf{x}$:

$$\begin{pmatrix} \mathbf{y}_t \\ z \end{pmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K}_t + s_{\text{noise}}^2 \mathbf{I}_t & \mathbf{k}_t(\mathbf{x}_{t+1}) \\ \mathbf{k}_t(\mathbf{x}_{t+1})^\mathsf{T} & \kappa(\mathbf{x}_{t+1}, \mathbf{x}_{t+1}) \end{pmatrix}\right)$$

The function of $\mathbf{x}$ that gives the mean of the predictive distribution $p(z|\mathbf{x}, \mathcal{D})$ is the *GP posterior mean* $\mu_t(\mathbf{x})$, the function that gives the variance of this probability is the *GP* *posterior variance* $\sigma_t(\mathbf{x})$, and by Equation (A.5) we have:

$$\mu_t(\mathbf{x}) \quad = \quad \mathbf{k}_t(\mathbf{x})^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{y}_t \tag{2.12}$$

$$\sigma_t^2(\mathbf{x}) \quad = \quad \kappa(\mathbf{x},\mathbf{x}) - \mathbf{k}_t(\mathbf{x})^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}) \tag{2.13}$$

where matrix $\mathbf{C}_t$ and vector $\mathbf{k}_t(i)$ are defined as follows:

$$\begin{aligned}(\mathbf{C}_t)_{p,q} &= \kappa(i_p, i_q) \text{ if } p \neq q \\ &\quad\ \kappa(i_p, i_q) + s_{\text{noise}}^2 \text{ otherwise} \\ (\mathbf{k}_t(i))_p &= \kappa(i, i_p)\end{aligned}$$

Equation (A.5) can also be used to characterise $p(z, z'|\mathbf{x}, \mathcal{D})$ and to derive the following expression for the posterior covariance between $\mathbf{x}$ and $\mathbf{x}'$:

$$\text{cov}_t(\mathbf{x},\mathbf{x}') = \kappa(\mathbf{x},\mathbf{x}') - \mathbf{k}_t(\mathbf{x})^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}') \tag{2.14}$$

Inference is simple and efficient in the Gaussian model, as we get expressions in closed form – this is not always the case with probabilistic models. The regressor μ_t is the same as in kRR. The kernel and covariance function play indeed the same role, and the two terms can be used equivalently. Predictions in the noise-free case are obtained by letting s_{noise} tend to 0. With the Bayesian approach we have made additional, probabilistic assumptions, in order to model uncertainty. The fact that $p(y|\mathbf{x}, \mathcal{D}_t)$ is a one-dimensional Gaussian with mean equal to $\mu_t(\mathbf{x})$ and variance equal to $\sigma_t^2(\mathbf{x})$ implies:

$$\mathbb{P}(|f(\mathbf{x}) - \mu_t(\mathbf{x})| \geq \sqrt{\beta_t}\sigma_t(\mathbf{x})) \quad = \quad \text{erfc}\left(\sqrt{\frac{\beta_t}{2}}\right) \tag{2.15}$$

by definition of the complementary error function erfc (Equation A.3).

2.2.2.3 Reproducing Kernel Hilbert Spaces

μ_t is a linear combination of functions from $\mathcal{X}$ to $\mathbb{R}$ of the form $k_\mathbf{x} = \kappa(\mathbf{x}, .)$ where $\mathbf{x}$ is a fixed element in $\mathcal{X}$. More precisely, it is a linear combination of the $k_{\mathbf{x}_\tau}$ functions for τ from 1 to t. Consider the vector space of these $k_\mathbf{x}$ functions, for all $\mathbf{x} \in \mathcal{X}$, with inner product between two functions $k_\mathbf{x}$ and $k_{\mathbf{x}'}$ defined as $\kappa(\mathbf{x},\mathbf{x}')$. We denote by $\mathcal{H}_\kappa$ the Hilbert space obtained when completing this vector space of functions with the limits of Cauchy sequences with respect to the norm defined by the previous inner product. An interesting property of this space is that, for any $f \in \mathcal{H}_\kappa$ and any $\mathbf{x} \in \mathcal{X}$, the inner product between f and $k_{\mathbf{x}}$ is equal to $f(\mathbf{x})$. We say that $\mathcal{H}_\kappa$ is the *Reproducing Kernel Hilbert Space* induced by the kernel κ.

A few remarks:

- $\mathcal{H}_\kappa$ can be made an almost arbitrarily rich space of functions, depending on the choice of κ. Kondor (2003) notes that for the Gaussian kernel, $\mathcal{H}$ can be shown to be a dense subset of $L_2(\mathcal{X})$

- Defining a probability distribution $p(f) \propto \exp(-\frac{||f||_\kappa^2}{2})$ in the RKHS and using it as a prior on f implies the GP consistency property. This is easier to see when $\mathcal{X}$ is finite and $\mathbf{K}$ is invertible, as we can write $\forall \mathbf{x}, f(\mathbf{x}) = \sum_i \alpha_i \kappa(\mathbf{a}_i, \mathbf{x})$ where $\boldsymbol{\alpha} = \mathbf{K}^{-1}\mathbf{f}$, and thus:

$$\begin{aligned}
||f||_\kappa^2 &= \langle f, f \rangle \\
&= \sum_i \sum_j \alpha_i \alpha_j \langle k_{\mathbf{a}_i}, k_{\mathbf{a}_j} \rangle \\
&= \sum_i \sum_j \alpha_i \alpha_j K_{i,j} \\
&= \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} \\
&= \mathbf{f}^T \mathbf{K}^{-1} \mathbf{f}
\end{aligned}$$

- In the next chapter we will give a regret bound for a GP-based bandit algorithm in terms of the norm of f in the RKHS induced by the chosen covariance function.

2.2.3 Covariance functions

In many problems we can assume that the covariance should be *stationary*, meaning that $\kappa(\mathbf{x}, \mathbf{x}')$ should be a function of $\mathbf{x} - \mathbf{x}'$ and thus invariant to translation, or even that it is *isotropic*, meaning that it should be a function of the distance between $\mathbf{x}$ and $\mathbf{x}'$ and thus invariant to all rigid motions. One such covariance function that is widely used in practise in $\mathbb{R}^n$ is the ISO-SE which is a Squared Exponential (SE) on the Euclidian distance between two vectors, with a width adjusted to fit the *characteristic length-scale* s which is assumed for f along each of its dimensions,[3] multiplied by a

[3]The name ISO-SE comes from the fact that the length scale is the same along each dimension.

signal variance term s_f:

$$\kappa(\mathbf{x},\mathbf{x}') = s_f^2 \exp\left(-\frac{||\mathbf{x}-\mathbf{x}'||^2}{2s^2}\right) \tag{2.16}$$

We refer the reader to Shawe-Taylor and Cristianini (2004) and Rasmussen and Williams (2006) for proofs that this is indeed a valid kernel/covariance function. Sample functions for the ISO-SE covariance function are particularly smooth as they are differentiable to any order almost surely.

s and s_f are called *hyper-parameters*: they are parameters of the covariance function, hence of the model, but not of the regression method (which is non parametric). The signal variance s_f^2 characterises our prior belief on the value z of f at any point $\mathbf{x}$:

$$\begin{aligned} p(z|\mathbf{x}) &= \mathcal{N}(m(\mathbf{x}),\kappa(\mathbf{x},\mathbf{x})) \\ &= \mathcal{N}(m(\mathbf{x}),s_f^2) \end{aligned}$$

An advantage of the ISO-SE covariance function is that we can encode, through s, a belief on the length-scale of f which, loosely speaking, characterises how much f is expected to change on a given scale, or the distance between two inputs from which they will become practically uncorrelated. It can be shown that the mean number of level-zero crossings on a unit interval for a one-dimensional SE process with zero mean is $(2\pi s)^{-1}$. Note that if s is overestimated, the GP regressor will be generalising too much, and the $\mu_t(\mathbf{x})$ estimates will be too close to each other. If s is underestimated, we will not be sharing enough information from one input to the other and we will over-fit the data.

We now consider a similar covariance function, but with different length-scales $(s_i)_{1\leq i\leq n}$ along the different dimensions of the input space.

$$\kappa(\mathbf{x},\mathbf{x}') = s_f^2 \exp(-\frac{(x_1-x_1')^2}{2s_1^2})\dots\exp(-\frac{(x_n-x_n')^2}{2s_n^2}) \tag{2.17}$$

The $1/s_i$ terms are analogous to (independent) weights given to the different dimensions of the input space when combining the differences between the components of two vectors in order to determine their Euclidian distance. A lower length-scale on the i^{th} dimension will give it more importance. The covariance above is called the ARD-SE, for Automatic Relevance Determination.

Although we may be able to determine from the context whether the SE covariance functions would be suitable for a particular application, in practise we only have a vague idea of what the length-scales should be. We can set their values by maximising the likelihood of the observations (which is expressed as a function of the hyper-parameters).[4] In the ARD-SE case, we can thus learn the relative importance of each dimension on the f values. Maximising the likelihood is equivalent to minimising minus its log, for which the derivatives with respect to each s_i can be given in terms of $\frac{\partial \mathbf{C}_t}{\partial s_i}$ (see Rasmussen and Williams, 2006, for a complete proof):

$$\begin{aligned}
p(\mathbf{y}_t|\mathbf{X}_t, s_1, \ldots, s_n, s_f, s_{\text{noise}}) &= \mathcal{N}(0, \mathbf{K}_t + s_{\text{noise}}^2 \mathbf{I}_t) \\
-\log(p(\mathbf{y}_t|\mathbf{X}_t, s_1, \ldots, s_n, s_f, s_{\text{noise}})) &= \frac{1}{2}\log(|\mathbf{C}_t|) + \frac{n}{2}\log(2\pi) + \frac{1}{2}\mathbf{y}_t^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{y} \\
\frac{\partial}{\partial s_i}(-\log(p(\ldots))) &= \frac{1}{2}\,\mathrm{tr}\left((\mathbf{C}_t^{-1} - \mathbf{C}_t^{-1}\mathbf{y}_t\mathbf{y}_t^\mathsf{T}\mathbf{C}_t^{-1})\frac{\partial \mathbf{C}_t}{\partial s_i}\right)
\end{aligned}$$

The partial derivative of $\mathbf{C}_t$ with respect to s_i is a matrix whose components are:

$$\frac{(x_{\tau,i} - x_{\tau',i})^2}{s_i^3}\kappa(\mathbf{x}_\tau, \mathbf{x}_{\tau'})$$

As a consequence, gradient search methods can be employed, but they may have difficulties in finding an optimal setting of the hyper-parameters as the likelihood function may suffer from multiple local optima.

2.3 Tree search

The exploration/exploitation balance achieved by bandit algorithms can be applied to the search of very large spaces organised in tree structures. We consider functions defined on leaves of a tree with finite depth and branching factor, and noisy observations. We look for the leaf with highest function value, f^*. We consider cases where an exhaustive search of the tree is prohibitive due to its size.

Typically, algorithms proceed in iterations of tree traversals starting from the root. After the t^{th} iteration, a leaf node n_t is selected and a reward y_t is received. It is usually assumed that there exists a mean-reward function f such that y_t is a noisy observation of $f(n_t)$. Other common assumptions are that f^*, the highest value of f, is known (or an upper bound on f^* is known) and is always bigger than y_t. The algorithm stops when a convergence criterion is met, when a computational/time budget is exhausted

[4]If we have prior knowledge regarding the hyper-parameters, we can seek a MAP estimator instead. Sparsity-inducing hyper-priors can be useful to remove variables that are irrelevant to the learning problem.

(in game tree search for instance), when a maximum number of iterations has been specified (this is referred to as *fixed horizon* exploration, as opposed to *anytime*), or when uncertainty measures drop below a certain threshold. In the end, a path through the tree is given. This can simply be the path that leads to the leaf node that received the highest reward, or that has the highest estimated reward, or that has the highest lower confidence bound at a given confidence threshold. These last two might be more robust to the variability of the rewards (we could be misled by an unlikely high reward value for a mediocre path).

2.3.1 Many-bandits algorithms: Upper Confidence Trees

2.3.1.1 Path selection as a sequence of bandit problems

Many-bandits tree search algorithms use bandit problems at each interior node of the tree in order to assign high-probability upper bounds on the best reward values that can be obtained by continuing the exploration from each of this node's children. We call these upper bounds *U-values* (they are called B-values in the notations of Coquelin and Munos (2007b) and Bubeck et al. (2010), but B is already reserved here for the branching factor of the tree). The children of a given node are the arms of its associated bandit problem, and the U-values of the children are defined as the f_t values that are assigned to them by their parent's bandit algorithm. At each iteration of the tree search algorithm, we start from the root and repeatedly select the child node with highest U-value, until a leaf j is reached and a reward y is received. Then, for each ancestor i of j we add the observation (i, y) to the training set of the bandit algorithm of the parent of i, and thus we update the U-values of all ancestors of j.

2.3.1.2 UCT

Kocsis and Szepesvári (2006) proposed a many-bandits tree search algorithm based on UCB1, which they called Upper Confidence Trees and which is described below. Gelly and Wang (2006) reported that UCT performed significantly better than previous approaches to Go game trees, when generating stochastic rewards at leaves by rolling out the game randomly and receiving $+1$ when winning, 0 otherwise.[5]

[5]Note that, for two-player game trees, when UCT has to choose a move for the opponent, it uses $1 - \mu_t(i)$ instead of $\mu_t(i)$ in the UCB formula (the exploration term stays the same).

- Repeat while stopping/convergence criterion has not been met:
    - Set node x to the root
    - Repeat while x is not a leaf:
        - If x has never been seen before, associate a new UCB1 algorithm to x, notated $A(x)$, with arms corresponding to the children of x. The f_t values given by $A(x)$ to its arms are initialised to infinity and used to assign U values to the children of x.
        - Set x to the node with highest U value (break ties arbitrarily).
    - Get reward value y for the leaf x that has been reached
    - "Back-propagate" the reward, i.e. repeat while x has a parent:
        - Add (x, y) to the training data of $A(\text{parent}(x))$. This updates the f_t values given by this algorithm, which are, by definition, the U values for all siblings of x.
        - Set x to parent(x)

2.3.1.3 Measure of performance

A Tree Search algorithm's performance can be measured, as for a bandit algorithm, by its cumulative regret $R_T = Tf^* - \sum_{t=1}^{T} f(n_t)$. However, although this is a good objective to achieve a good exploration/exploitation balance, we might be ultimately interested in a bound on how far the reward value for the best node we would see after T iterations is from the optimal f^*. Or it might be more useful to bound the regret after a given execution time (instead of a number of iterations) in order to compare algorithms that have different computational complexity.

2.3.1.4 Tree growing methods

The trees we set to search are usually too big to be represented in memory, which is why we "grow" them iteratively by only adding the nodes that are needed for the implementation of our algorithm. The *fixed-depth* tree growing method consists, at each iteration, in selecting child nodes sequentially until reaching a maximum depth D. Another method of growing the tree is *iterative-deepening*, which consists in stopping the traversal of the tree only after having created a new node. Thus, the maximum depth can potentially be increased at each iteration. In Go tree search (Coulom, 2006), the method used to evaluate leaf nodes does not depend on their (variable) depth as it simply consists in random rollouts. The tree may grow asymmetrically as it contains paths that have different numbers of nodes. Hopefully iterative-deepening helps to go deeper in the tree in regions where f has high values, and keeps the paths short in the rest of the tree. This saves time and memory by stopping the exploration early and not creating nodes that would belong to sub-optimal paths.

Note that iterative-deepening is not suitable for cases where reward values depend on the number of nodes in a path (as for sums of intermediate reward values, for instance), because this would favour the exploration of average nodes that are deep in the tree, rather than the exploration of promising nodes that are not that deep.

2.3.2 Revised upper confidence bounds: the Bandit Algorithm for Smooth Trees

2.3.2.1 Description

Despite the good performances of UCT on Go, Coquelin and Munos (2007b) showed that it can behave poorly in certain situations because of "overly optimistic assumptions in the design of its upper confidence bounds" (Bubeck and Munos, 2010), leading to a high lower bound on its cumulative regret. Consequently, they proposed a revised definition of U to overcome this problem, based on a regularity assumption on f: there exist decreasing values $\rho_{0\leq d\leq D}$ such that, for all i at depth d and for all descendants j of i, $f(i) - f(j) \leq \rho_d$. This implies that the more ancestors in common between two leaves, the closer their f values will be. This assumption was subsequently relaxed to η-suboptimal nodes only, for a fixed η.

The Bandit Algorithm for Smooth Trees of Coquelin and Munos follows the algorithmic description of UCT given in Section 2.3.1.2 and only differs in the way that the U-values are updated at interior nodes when rewards are back-propagated:

$$U(i) = \min\{f_t(i) + \rho_d, \max_{j \text{ child of } i}\{U(j)\}\} \tag{2.18}$$

where $f_t(i)$ is defined as $\mu_t(i) + \sqrt{\frac{\log(\bar{N}\nu(i,t)(\nu(i,t)+1)\delta^{-1})}{2\nu(i,t)}}$ and $\mu_t(i)$ is the reward estimate for node i given by the UCB1 algorithm associated to the parent node of i – in other terms it is the average of the rewards obtained after selecting i.

2.3.2.2 Theoretical insights

We first explain how the regularity assumption on f is used to build true upper confidence bounds at interior nodes and to finally arrive at the above formulation. For this, we extend the definition of f to all nodes: we set f on any interior node to be the maximum value of f on tree paths that go through this node. The f_t values of UCT do not represent true upper confidence bounds on the f values (except for leaf nodes), because the rewards are not iid: the leaf nodes for which the rewards are obtained depend on a node selection process which is not stationary. Therefore, Hoeffding's inequality does not apply. However, owing to the regularity assumption on f we can relate its value at a given node i to its value at leaf nodes $j \in \mathcal{L}(i)$ that are descendants of i, and thus derive true confidence bounds. We assume that the number of times n_j that node $j \in \mathcal{L}(i)$ has been selected by the tree search algorithm is given. We write $y_{j,\tau}$ for the τ-th reward received at leaf j, so that:

$$\begin{aligned} n_i &= \sum_{j \in \mathcal{L}(i)} n_j \\ \mu_t(i) &= \frac{1}{n_i} \sum_{j \in \mathcal{L}(i)} \sum_{\tau=1}^{n_j} y_{j,\tau} \end{aligned}$$

We have:

$$\begin{aligned} f(i) &= \frac{1}{n_i} \sum_{j \in \mathcal{L}(i)} n_j f(i) \\ &\leq \frac{1}{n_i} \sum_{j \in \mathcal{L}(i)} n_j (f(j) + \rho_d) \\ &\leq \mu_t(i) + \rho_d + \frac{1}{n_i} \sum_{j \in \mathcal{L}(i)} \sum_{\tau=1}^{n_j} f(j) - y_{j,\tau} \end{aligned}$$

Azuma's inequality (A.7) can be applied to the $(f(j) - y_{j,\tau})_{j,\tau}$ difference sequence. There are n_i elements in this sequence and they have range in $[-1, 1]$ since reward values are in $[0, 1]$. By the fact that rewards obtained when selecting the same leaf are iid and by definition of f on leaves, we have: $\forall j \in \mathcal{L}(i), \forall \tau \in [1, n_j], \mathbb{E}(f(j) - y_{j,\tau}) = 0$. This proves that, whatever the order of its elements, this sequence is a martingale. As a result, the probability of the sum of elements of the sequence being bigger than $\epsilon = \sqrt{\frac{\log(\delta^{-1}) n_i}{2}}$ is bounded above by δ. This gives, with probability $1-\delta$:

$$\begin{aligned} f(i) &\leq \mu_t(i) + \rho_d + \frac{1}{n_i}\epsilon \\ &\leq \mu_t(i) + \rho_d + \sqrt{\frac{\log(\delta^{-1})}{2n_i}} \end{aligned}$$

This true upper bound is bigger than the UCT pseudo upper bound by a ρ_d term, which shows that UCT is indeed overly optimistic given our assumptions of smoothness.

Recall that the role of the U-values is to put a tight, optimistic, high-probability upper bound on the best mean-reward value that can be achieved from a given node. We have seen a way to derive true upper confidence bounds at interior nodes. We could also get true bounds by considering at depth $D-1$ the max of the true upper confidence bounds of the children (i.e. their f_t values since they are leaves), and so on for depths $D-2$ to 1. As a consequence, the U-values in BAST were defined so as to benefit from these two ways of constructing upper confidence bounds. The choice of the expression for f_t was motivated by the regret analysis which can be carried out when the f values of all the $\bar{N} = \frac{B^{D+1}-1}{B-1} - 1$ nodes of the tree are within their confidence intervals, with high probability. Note that the exploration term expression contains no term in t only, so we should write:

$$\begin{aligned} \beta_t &= 1 \\ \sigma_t^2(i) &= \frac{\log(\bar{N}\nu(i,t)(\nu(i,t)+1)\delta^{-1})}{2\nu(i,t)} \end{aligned}$$

BAST is parameterised by ρ_d and can therefore adapt to different levels of smoothness of the reward function. A time-independent regret upper bound was derived, expressed in terms of the sub-optimality values Δ_i of nodes (dependent on the reward f on nodes, hence unknown to the algorithm) and was thus problem specific. Also, quite paradoxically, the bound could become very high for smooth functions (because of $1/\Delta_i$ terms).

2.3.2.3 UCT as a special case of BAST

Let us show that BAST with $\rho_d = 0$ for all d is equivalent to UCT. For this, we show by induction that the f_t values at all nodes, as given by their parent's bandit algorithm, coincide with their U-values. This is true at depth D, by definition of BAST. Assuming that the result is true at depth d, we show by contradiction that, for all nodes i at depth $d-1$, $f_t(i)$ is smaller than the maximum of all the U-values of its children. If this was not the case, then for all children j of i, we would have $\mu_t(i) + \beta_t\sigma_t^2(i) > \mu_t(j) + \beta_t\sigma_t^2(j)$ since the children are at depth d and their U-values coincide with their f_t values; hence $\mu_t(i) > \mu_t(j)$ since i has been played more times that its child j and consequently $\sigma_t^2(j) - \sigma_t^2(i) > 0$; this last result on μ_t contradicts the fact that $\mu_t(i)$ is a weighted average of the $\mu_t(j)$ values. As a consequence, $U(i) = \min\{f_t(i) + \rho_{d-1}, \max_{j \text{ child of } i}\{U(j)\}\} = f_t(i)$.

2.3.3 Applications to planning in Markov Decision Processes

We present an application of bandit based tree search to the problem of planning in Markov Decision Processes. Beforehand, we introduce the notion of MDP as a formalism of decision making in environments in which the actions taken by an agent can change its state and give rewards. We present classical planning methods and their disadvantages in order to motivate the use of Open Loop planning methods. These are applicable when a generative model of the environment is available. We introduce the OLOP algorithm, inspired by previous tree search algorithms, where the tree represents the possible sequences of actions from the current state, and which exploits the smoothness induced by the particular form of the reward function as an exponentially discounted sum of bounded intermediate rewards.

2.3.3.1 Background

Markov Decision Processes are a formalism for sequential decision-making problems, which are numerous in operations research (inventory control, optimising transportations systems, schedules, production, etc.) and also occur in the control of chemical, electronic or mechanical systems (Szepesvári, 2010). The characteristics of the problems modelled by MDPs are that:

- a decision maker, called the "agent", acts within an environment in a sequential fashion;
- each action that the agent takes changes its state and the actions that will be available from then on;
- a reward is given by the environment for each action taken by the agent;

- the agent must decide which action to take at each time step, in order to maximise the rewards given by the environment.

Note that being greedy at each time step and selecting the action that maximises the reward may lead to a state from which only actions with poor rewards will be available.

We consider finite and deterministic MDPs in which the set of actions $\mathcal{A}$ is finite and the transition from one given state to the next one, after taking a given action, is deterministic. Such MDPs are characterised by $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathbb{P}, \gamma)$ tuplets where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{T}$ is a function that takes a state-action pair and returns a new state, $\mathbb{P}$ is a probability distribution over the possible reward values in $\mathbb{R}$, and γ is a *discount factor*. We introduce some terms and notations in order to define the role of γ. Let $S_t \in \mathcal{S}$ denote the state at time t, $A_t \in \mathcal{A}$ the action taken, and Y_t the reward obtained from the environment after taking this action: $Y_t \sim \mathbb{P}(.|X_t, A_t)$. The agent selects actions to take based on the observed history, according to certain "rules" that we call the *behaviour* of the agent. The *return* of a given behaviour is the discounted sum of all the rewards incurred: $\sum_{t=0}^{\infty} \gamma^t Y_{t+1}$. The goal of the agent is to adopt a behaviour that is as close as possible to the optimal one, i.e. the behaviour that maximises the expected return.

The optimal behaviour is the one that always takes the action a that maximises the *optimal action-value function* $Q^*(x, a)$, when in state x. This is defined as the maximum expected return under the constraints that the process starts at state x and the first action chosen is a. Therefore, it verifies the following *Bellman equation*:

$$Q^*(x, a) = r(x, a) + \gamma \max_{a' \in \mathcal{A}} Q^*(\mathcal{T}(x, a), a')$$

where $r(x, a)$ is defined as the mean of $\mathbb{P}(.|x, a)$. We can get a near-optimal behaviour by learning the optimal action-value function and taking actions greedily with respect to this value function. We refer the reader to Szepesvári (2010) for a review of the techniques that follow this idea (Value iteration, Dynamic Programming, Temporal-difference learning). However, this approach can become problematic for large state-action spaces, for which it will be difficult to store all values in memory: some sort of function approximation is required.

2.3.3.2 Open-Loop Optimistic Planning

Another approach to choose which actions to take is *open-loop planning*, which is used when a generative model of the MDP is available, i.e. we know $\mathcal{T}$ and have a model of $\mathbb{P}$. We perform a search of the tree representing the succession of actions available from the current state we are in (seen as the root of the tree). A sample-based look-ahead search starts exploring the tree from the root, stops when reaching a certain depth, observes rewards generated from the model for the actions taken, and starts again from the root. The exploration and exploitation of branches of the tree should be balanced. One approach proposed by Kocsis and Szepesvári consists in using UCT, which works in an any-time fashion and returns the root action from the current state of the MDP for which the average of the rewards generated during planning was the highest.

The Open Loop Optimistic Planning (OLOP) algorithm of Bubeck and Munos is based on BAST and builds true upper confidence bounds by exploiting the smoothness of f, induced by the discount factor: $\rho_d = \frac{\gamma^{d+1}}{1-\gamma}$. The other difference with BAST comes from the fact that, in the MDP setting, we observe the intermediate reward values that make up the reward function on paths. OLOP takes advantage of this extra information by updating each bandit instance along each node of a path that was played, using the corresponding intermediate reward value instead of back-propagating the reward value for the whole path.

The branching factor of the tree is the cardinality of $\mathcal{A}$. The depth of the tree is potentially infinite, but we need to stop each exploration of the tree (an iteration of the tree search algorithm) at a certain depth. The cumulative regret $\bar{R}_n$ considered by Bubeck and Munos is measured as a function of the number of calls n to the generative model, which is equal to $D\ T$ for us. It is the sum for t from 1 to T of the immediate regrets $\bar{r}_t$ defined as the difference between the infinite sum of discounted rewards for the sequence of nodes chosen by the optimal policy, and for the sequence of nodes given by following our policy for D actions and switching to the optimal policy from then on. Consider the t^{th} path exploration. Let us write $n_{D,t}$ for the node that we have after following our policy for D actions. It may be different from the node n^*_D that we would have had with the optimal policy. For this reason, n^*_{D+1} may not be available after $n_{D,t}$, which implies that the sequences of nodes that follow can be different, even though we are using the same, optimal policy. Consequently, $\bar{r}_t$ is equal to r_t, measured up to depth

D, plus $\gamma^D \sum_{i=1}^{+\infty} 2\gamma^{i-1}$, where the intermediate reward differences after D actions are all bounded by 2 (since rewards lie in $[-1,1]$). OLOP works with $D = \log_B(T)$ so that the cost of stopping the exploration at depth D is not linear but is in the order of $T\gamma^D = T^{1-a}$ where $a = \log_B(1/\gamma) > 0$. This gives $\bar{R}_n = R_T + \tilde{O}(T^{1-a})$.

The authors derive regret bounds expressed in terms of a measure $b \in [1, B]$ of the quantity of near-optimal branches in the tree:

$$\begin{aligned} \frac{1}{n}\bar{R}_n &= \tilde{O}(n^{-\log_b(1/\gamma)}) \text{ if } \gamma > \frac{1}{\sqrt{b}} \\ &= \tilde{O}(n^{-1/2}) \text{ otherwise} \end{aligned}$$

If $\gamma \leq \frac{1}{\sqrt{B}}$ then $\gamma \leq \frac{1}{\sqrt{b}}$ and the second bound applies. Otherwise, the first bound or the second bound applies. $\gamma > \frac{1}{\sqrt{B}}$ implies $a < 1/2$ and thus $n^{-1/2} < n^{-a}$; it also implies $n^{-\log_b(1/\gamma)} < n^{-a}$. As a consequence, we can give a result which does not involve problem-specific quantities:

$$\begin{aligned} \frac{1}{n}\bar{R}_n &= \tilde{O}(n^{-a}) \text{ if } \gamma > \frac{1}{\sqrt{B}} \\ &= \tilde{O}(n^{-1/2}) \text{ otherwise} \end{aligned}$$

We show that these bounds are equivalent to the following cumulative regret bounds on the tree search problem with maximum depth equal to $\log_B(T)$:

$$\begin{aligned} R_T &= \tilde{O}(T^{1-a}) \text{ if } \gamma > \frac{1}{\sqrt{B}} \\ &= \tilde{O}(\sqrt{T}) \text{ otherwise} \end{aligned}$$

We use the fact that $n = D\,T = T\log_B(T)$ by definition.

$$\begin{aligned} \frac{1}{n}\bar{R}_n &= \tilde{O}(n^{-a}) \\ \exists \alpha, \beta > 0, \bar{R}_n &\leq \alpha \log(n^{-a})^{\beta} n^{1-a} \text{ where } \beta \text{ can only be even} \\ &\leq \frac{(-a)^{\beta}}{(1-a)^{\beta}} \alpha \log(n^{1-a})^{\beta} n^{1-a} \\ \bar{R}_n &= \tilde{O}(n^{1-a}) \end{aligned}$$

$$\begin{aligned} \exists \alpha, \beta > 0, \bar{R}_n &\leq \alpha \log(n^{1-a})^{\beta} n^{1-a} \\ &\leq \alpha(1-a)\log(n)^{\beta} T^{1-a} \log_B(T)^{1-a} \\ &\leq \alpha' \log(T)^{\beta+1-a} T^{1-a} \\ &\leq \frac{\alpha'}{(1-a)^{\beta+1-a}} \log(T^{1-a})^{\beta+1-a} T^{1-a} \\ \bar{R}_n &= \tilde{O}(T^{1-a}) \end{aligned}$$

We thus have

$$R_T = \tilde{O}(T^{1-a}) - \tilde{O}(T^{1-a}) \text{ if } \gamma > \frac{1}{\sqrt{B}}$$
$$R_T = \tilde{O}(T^{1/2}) - \tilde{O}(T^{1-a}) \text{ otherwise}$$

which proves the result on R_T.

2.4 Related work

Firstly, we show how bandit-based tree search algorithms have been applied to design a global framework for online optimisation, namely the Hierarchical Optimistic Optimisation framework. For this framework to be applicable, we only need to be given a tree of coverings of the (potentially infinite) input space. We select a point (arm) to sample the objective function (reward) at by growing a tree structure representing the input space and sampling randomly within the subspace associated to a chosen leaf. We give an overview of the regret analysis of HOO. We then briefly review the UCB-AIR algorithm that also deals with infinitely many arms by making an assumption on the probability of selecting suboptimal arms, another algorithm that deals with many arms by clustering them, and one that works with taxonomies, i.e. hierarchies of clusters.

Secondly, as a prelude to the next chapter where we will be introducing the Gaussian Processes Bandit algorithm, we review some applications of the GP modelling tool to online function optimisation, in which points where to sample the function are chosen sequentially, based on the current function estimation and uncertainty measures. In particular, we show how the optimal allocation of function samples (in terms of the expected final loss) can be approximately determined, which is a technique very similar in spirit to the first Bayesian approaches to bandit problems.

2.4.1 Bandit problems with many and infinitely-many arms

2.4.1.1 Some algorithms

As pointed out in the introduction, some problems have a number of options that is much larger than the number of observations we can hope to make. In this case, if no assumption is made on the smoothness of f, the search might be arbitrarily hard. The key idea is, as we did with LinRel, to model dependencies between arms through smoothness assumptions on f, so that information can be gained about several arms (if not the whole set of arms) when playing only one arm. Pandey et al. (2007) have developed an algorithm which exploits cluster structures among arms in order to share knowledge between them, motivated by a content-matching problem (matching webpages to ads). Wang et al. (2008) make a probabilistic assumption on f: the probability that an arm chosen uniformly at random is ϵ-optimal scales in ϵ^{β}. Thus, when there are many near-optimal arms and when choosing a certain number of arms uniformly at random, there exists at least one which is very good with high probability. The regret bound of their UCB-AIR algorithm is in $\tilde{O}(\sqrt{T})$ when $\beta < 1$ and $f^* < 1$, and in $\tilde{O}(T^{\frac{\beta}{1+\beta}})$ otherwise.

Bandit problems in continuous arm spaces have been studied notably by Auer et al. (2007), Kleinberg et al. (2008), Wang et al. (2008) and Bubeck et al. (2009). Kleinberg et al. (2008) consider metric spaces, Lipschitz functions, and derive a regret growth-rate in $\tilde{O}(\exp(n)T^{\frac{n+1}{n+2}})$, which strongly depends on the dimension n of the input space. The algorithm of Bubeck et al. (2009), HOO, follows a similar idea to that of Kleinberg et al. which is to "zoom" the discretisation of $\mathcal{X}$ in regions of interest. They consider weak-Lipschitz functions in arbitrary topological spaces, and derive a bound with a similar growth-rate. However, when $\mathcal{X} = [0,1]^n$, when the number of maxima is finite and f is locally Hölder with a known exponent around any maxima, their bound becomes $\tilde{O}(\sqrt{\exp(O(n))T})$. This is strictly better than in the work of Kleinberg et al., is independent of the dimension of the input space and matches the bound derived by Auer et al. – but HOO deals with multiple dimensions, does not waste computational resources on discretising the input space finely where such a fine discretisation is not needed, and is therefore a viable algorithm in practise.

2.4.1.2 Hierarchical Optimistic Optimisation

The *Hierarchical Optimistic Optimisation (HOO)* algorithm uses BAST on a recursive splitting of the space where each node corresponds to a region, or *covering*, of the space, and all interior nodes have two children representing two-halves of the corresponding space. BAST is used to go down the tree of coverings of $\mathcal{X}$ in an iterative-deepening fashion, thus selecting smaller and smaller regions to randomly sample f in. A first version of this idea was given by Coquelin and Munos (2007b) with an application of BAST to the problem of optimising an unknown function in $[0,1]$. Here, the diameter of a covering at depth h is assumed to decrease exponentially. More precisely, it is assumed that there exists $\nu_1 > 0$ and $0 < \rho < 1$ such that for any h this diameter is bounded by $\nu_1 \rho^h$ (which we use as ρ_h value in BAST).

We give a brief overview of the strategy used to deal with the fact that HOO builds an infinitely deep tree in the derivation of a regret bound. As is usual with UCB-type algorithms, the fact that arms' mean-reward values lie within their confidence intervals with a high probability that rapidly decreases in time allows us to upper bound the number of times that arms are selected. The regret at time T is equal to the sum for each arm of its sub-optimality multiplied by the number of times it has been selected up to time T. HOO builds an infinite tree and constantly considers new arms which have sub-optimalities that become smaller and smaller as the algorithm narrows down the location of the optimum: Δ_{min} is not a constant anymore, as it decreases with time.

The regularity assumption on f (*weak-Lipschitzness* with respect to a *dissimilarity* l) and the assumption that the size of the coverings decreases exponentially with the depth in the tree are used in order to prove a key result for the following, owing to concentration of measure inequalities: if the sub-optimality of a node is bounded by $c\nu_1\rho^h$, then the sub-optimality of all descendants of this node will be bounded by $\max\{2c, c+1\}\nu_1\rho^h$. This is used so that we only need to analyse the regret contributions of nodes up to a depth H and bound the contributions of the descendants. We divide the tree into 'good' nodes and 'bad' nodes, i.e. nodes which sub-optimality values are smaller than $2\nu_1\rho^h$ when they are at depth h, and nodes which sub-optimality values are bigger. The value of H will have to be chosen so that it minimises the sum of the contributions of the good nodes and of the bad nodes on the regret. The tree is divided in 3:

- Good nodes at depth H and their descendants: we have selected T of these nodes at most, and their sub-optimalities are bounded by $4\nu_1\rho^H$.
- Good nodes at depth $h = 0 \dots H-1$. Each node is played once at most (because HOO uses iterative deepening to grow the tree) and the number of good nodes at depth h is bounded in terms of a quantity called the near-optimality dimension of f with respect to l.
- Bad nodes at depth $h = 0 \dots H-1$ that have a good parent (at depth $h-1$), and their descendants. The number of times that such nodes are played is bounded

by the sum over h of the number of bad nodes at depth h that have a good parent (this is smaller than twice the number of good nodes at depth $h-1$, by definition), multiplied by the number of times that a bad node at this depth is expected to have been chosen (meaning that the node was on a path chosen by the algorithm). Once this has been bounded, it can be multiplied by the sub-optimality bound derived from the fact that the parent is a good node.

2.4.2 Bayesian global optimisation

2.4.2.1 Motivation

Global optimisation is one area where the exploration/exploitation dilemma appears: we need to learn a function and to optimise it at the same time. Research in this field has produced methods such as Lipschitz optimisation, homotopy methods, simulated annealing, genetic algorithms and Bayesian response-surface methods (Lizotte, 2008). These methods all deal in their own way with the exploration and exploitation tradeoff, for instance through the acceptance probability function in simulated annealing, or through crossovers and mutations in genetic algorithms.

Bayesian approaches typically use Gaussian Processes to model a belief on f, as we have seen in Section 2.2. Samples of the target function are acquired iteratively and used to maintain a posterior belief on f, and thus to decide where we want to sample next. The combination of function estimates and uncertainty measures is particularly useful as, for the problem of global optimisation, we are not interested in learning f accurately where we are confident that its values are low. GP optimisation methods (also referred to as “Kriging” and “response-surface” optimisation) are very popular due to the flexibility and power of GPs (see Brochu et al., 2009, for a review of Bayesian optimisation using GPs) and their applicability in practise in engineering problems (see Grünewalder et al., 2010, and references therein). Their main limitation, however, seems to be a poor scalability with respect to the dimension of the search space (Rolet, 2011).

When function evaluations are expensive, it is important to choose samples carefully, as we do in bandit problems. We can thus see the problem of deciding which samples to acquire as a bandit problem: each point in the search space is an arm and, when it is played, a potentially noisy observation of f at that point is given as a reward. The UCB heuristic, which is popular in bandit algorithms, could be used to focus exploration in global optimisation. Even though the actual objective may not be to minimise the cumulative regret but to minimise the simple regret, we have seen in Section 2.1 how a bound on the former can give a bound on the latter. Minimising the cumulative regret forces algorithms not to waste samples, which can be costly to acquire in certain applications as they might involve a physical and expensive action for instance, such as deploying a sensor or taking a measurement at a particular location (see the experiments on sensor networks performed by Srinivas et al., 2010), or they can simply be computationally costly because of lengthy computer simulations for instance: the less samples, the quicker we can find a maximum. Such problems include robot gait design, online path planning, algorithm configuration, sensor placement and reinforcement learning (see references within Hoffman et al., 2011).

2.4.2.2 Sample acquisition criteria

GP optimisation algorithms differ in their sample acquisition function f_t. We assume that this function is given in closed form or is easy to evaluate, so that its maximisation can be carried out with standard numerical techniques, sequential quadratic programming or the DIRECT algorithm (Hoffman et al., 2011). Even when f_t is multimodal, Brochu et al. (2009) showed that global search heuristics are very effective.

The GPGO algorithm of Osborne et al. (2009) considers a fixed horizon T and computes an approximation of the Expected Improvement (EI) provided by the selection of $\mathbf{x}$ at time t, over all possible $T-t$ remaining allocations. The improvement is defined as the difference between the best observed function value at time T and at time t. For this, the probability of improvement is broken down into the probability of improvement given the arms at times t to T, times the probability of picking these arms, which can also be broken down recursively. This is similar in spirit to the work of Gittins and Jones. The computations have a very high computational cost (hopefully warranted by the cost of function samples) and, in the experiments of Osborne et al., the number of iterations was only twice the dimension of the problem. However, the algorithm was found to perform better than other optimisation methods on most benchmark problems. A lower bound on the simple regret of this algorithm was given by Grünewalder et al. (2010) in the case where observations are not noisy. Different variants of Expected Improvement exist, such as the myopic version which assumes that the next sample will be the last one – see Bull (2011) for more on Expected Improvement and for convergence rates.

In the UCB heuristic, f_t takes a form also encountered in sequential design (see the Sequential Design for Optimisation algorithm of Cox and John, 1997): $f_t = \mu_t + \sqrt{\beta_t}\sigma_t$. Note that if we wanted to maximise the information gained at each time step, we would take $f_t = \sigma_t$ – but some of this information is not useful as we do not need to learn f accurately in regions where its values are low. We present GP-UCB in more detail in the next Chapter, and we also present the theoretical regret bounds derived by Srinivas et al. (2010). They are based on the rate of decay of the eigenvalues of the kernel matrix on the whole set of arms, if finite, or of the kernel operator. The regret is $O(n\sqrt{T})$ for the linear kernel and $O(\sqrt{T \log(T)^{n+1}})$ for the ISO-SE kernel, where n is the dimension of the input space. Bounds are given when f follows a GP distribution, but also in the setting where f has finite norm in the RKHS induced by the covariance function. However, the convergence rates for optimisation in that setting are not optimal (Bull, 2011). While Srinivas et al. report that, on an application to sensor networks, they obtained their best results with the UCB heuristic, Hoffman et al. (2011) show that there is no sample acquisition technique that consistently performs better than others. The algorithm they propose, GP-Hedge, uses a portfolio of heuristics including UCB, Expected Improvement and Maximum Probability of Improvement (see Mockus, 1989; Lizotte et al., 2007, for practical applications) and adaptively learns which ones are better for the problem at hand. Another efficient way to trade exploration and exploitation in practice is Thompson sampling (Chapelle and Li, 2011), which is not based on the maximisation of a f_t function but on drawing $\mathbf{x}_{t+1}$ from the posterior distribution at time t.

Finally, note that, in practise, the covariance function is not entirely specified as its parameters are not held fixed but they are estimated from previous observations. The problem is that our GP optimisation algorithm chooses where it wishes to observe f, and thus there is bias in the areas where it observes the function. Bull (2011) shows that using the Maximum Likelihood (or MAP) estimates of the hyper-parameters may cause GP-EI to never converge. The author proposes alternative estimators for which the convergence rates with a fixed prior still apply, with constants that are minimised.

2.4.2.3 Comparison to HOO and other bandit algorithms

The GP-based optimisation methods do not need f to be bounded in a known interval (a common assumption in bandit algorithms), but instead they have a prior on the signal variance. Bull (2011) note that the HOO assumption that f has a finite number of global optima and is quadratic in a neighbourhood of each – under which the regret growth rate matches that of GP-UCB, in $\tilde{O}(\sqrt{T})$ – is quite restrictive. GPs specify global smoothness properties, unlike the local assumptions of the continuum-armed bandit algorithms, but Srinivas et al. say that these algorithms' regularity assumptions can be too coarse-grained, whereas the GP assumption is neither too weak nor too strong in practise. One added benefit of the Bayesian framework is the possibility of tuning the parameters of our smoothness assumption (encoded in the covariance function) by maximising the likelihood of the observed data, which can be written in closed-form for the commonly used ARD-SE.

Interestingly, Graepel et al. (2010) have used ideas that are similar to what we have reviewed here, for the problem of Click-Through Rate prediction in sponsored search advertising systems: for a given ad impression characterised by a feature vector $\mathbf{x}$, the probability of a click is modelled as a probit function on top of the inner product between a weight vector $\mathbf{w}$ and $\mathbf{x}$; a factorising Gaussian prior distribution over $\mathbf{w}$ is assumed; ad impressions are chosen by Thompson sampling.

3

Gaussian Process Bandits

We have seen in Chapter 2 that GPs perform the same regression as kRR, but, owing to additional probabilistic assumptions, they are also able to model uncertainty. By assuming a GP prior on the mean reward as a function of arm feature representations, we are able to derive true upper confidence bounds for any arm, formed by a multiple of the GP posterior variance added to the posterior mean. In the rest of this thesis, we focus on the GP-UCB bandit algorithm, also referred to as *Gaussian Process Bandits (GPB).*

We first show how UCB1 can be seen as a special case of GPB, when the kernel is the Kronecker function. This is followed by an analysis of the computational cost of GPB. We reduce its complexity by deriving linear update expressions which involve an expensive matrix-vector product whose value can be shared for the updates of the reward estimates and uncertainty measures of all arms. We write N for the number of arms and T for the number of iterations. We obtain two formulations with total costs in $O(N\ T^2)$ or in $O(N^2\ T)$, instead of $O(T^4 + N\ T^3)$ for a naive formulation of the algorithm. The same results also apply to LinRel. In addition to this, we propose an approximation called *GPB-red* that helps make the algorithm run faster by occasionally removing the oldest training data point, so that the size of the training set is bounded by an increasing function S. The cost thus becomes $O(T(S_T^2 + N\ S_T))$.

In the second section of this chapter, we present an information-theoretic analysis of the regret of GPB due to Srinivas et al., in which the regret is bounded with high probability (up to constant and logarithmic factors) by the square root of T multiplied by the maximum possible information gain after T iterations. We focus on bandit problems with finite number of arms N, and we will see that the information gain can be expressed in terms of the eigenvalues of the total kernel matrix on the input space. As a consequence, the information gain can be bounded by a constant that depends on N. We finish by mentioning how the analysis presented here is extended to infinitely large spaces of arms, and to a more agnostic setting where the target function f has finite norm in a given Reproducing Kernel Hilbert Space and the noise sequence is a martingale difference sequence.

3.1 Gaussian Processes for bandit problems

3.1.1 The GPB algorithm

We assume a GP prior with covariance function κ on the mean-reward function f. In the absence of any extra knowledge on the problem at hand, f is flat and centred in the output space, so our GP prior mean is the $\mathbf{0}$ function. We model the variability of the reward, when always playing the same arm, as Gaussian noise with variance s_{noise}^2 (where s_{noise} is a parameter of the model). In the case of a finite number of arms, the GP prior on f is equivalent to an N-variate Gaussian prior on $\mathbf{f}$:

$$f \sim \mathcal{GP}(\mathbf{0}, \kappa) \Leftrightarrow \mathbf{f} \sim \mathcal{N}(\mathbf{0}, \mathbf{K})$$

where we write $\mathbf{K}$ for the total kernel matrix on $\mathcal{X}$. The posterior at time t after seeing data $\mathcal{D}_t$ has mean $\mu_t(\mathbf{x})$ and variance $\sigma_t^2(\mathbf{x})$, as given in Equations (2.12) and (2.13).

3.1.1.1 Arm selection

For a given training set $\mathcal{D}_t$, GPB has the same confidence interval centres as LinRel, but different widths. As a UCB-type algorithm, it selects arms iteratively by maximising the upper confidence function:

- Initialisation:
  - For all $\mathbf{x} \in \mathcal{X}$, we set $\mu_0(\mathbf{x}) = 0$ and $\sigma_0^2(\mathbf{x}) = \kappa(\mathbf{x}, \mathbf{x})$
  - $t = 0$
- Loop:
  - Play $\mathbf{x}_{t+1} = \text{argmax}_{\mathbf{x} \in \mathcal{X}} f_t(\mathbf{x})$ and break ties arbitrarily
  - Get reward y_{t+1}, which defines $\mu_{t+1}(\mathbf{x})$ and $\sigma_{t+1}(\mathbf{x})$ (and thus $f_{t+1}(\mathbf{x})$) for all $\mathbf{x}$:

$$\begin{aligned} \mu_{t+1}(\mathbf{x}) &= \mathbf{k}_{t+1}(\mathbf{x})^\mathsf{T} \mathbf{C}_{t+1}^{-1} \mathbf{y}_{t+1} \\ \sigma_{t+1}^2(\mathbf{x}) &= \kappa(\mathbf{x}, \mathbf{x}) - \mathbf{k}_{t+1}(\mathbf{x})^\mathsf{T} \mathbf{C}_{t+1}^{-1} \mathbf{k}_{t+1}(\mathbf{x}) \end{aligned}$$

  - $t = t + 1$

3.1.1.2 Choice of β_t

A choice of β_t corresponds to a choice of confidence interval width. The GPB regret bound given by Srinivas et al. (2010), that we present in the second section of this chapter, relies on the fact that the f values lie between their lower and upper confidence bounds. If $\mathcal{X}$ is finite and of cardinality N, this happens with probability $1-\delta$ if:

$$\beta_t = 2\log\left(\frac{Nt^2\pi^2}{6\delta}\right)$$

Justification δ is referred to as the *confidence threshold* and lies between 0 and 1. We write p for the probability that at least one f value is not in its confidence interval at some point in time. Applying Inequality (2.15) for all t and for $\mathbf{x} = \mathbf{a}_i$ for all i gives:

$$\begin{aligned} p &= N\sum_{t=1}^{\infty}\operatorname{erfc}(\sqrt{\frac{\beta_t}{2}}) \\ &\leq N\sum_{t=1}^{\infty}\exp(-\frac{\beta_t}{2}) \\ &\leq N\frac{6\delta}{N\pi^2}\sum_{t=1}^{\infty}\frac{1}{t^2} \\ &\leq \delta \end{aligned}$$

where we have used the upper bound on erfc given in Inequality (A.4). As a consequence, all f values are within their confidence intervals at all times with probability $1-\delta$.

The above expression for β_t is due to Srinivas et al.. Note that they did not optimise the constants in this expression, and it is usually beneficial in practise to scale β_t by a constant specific to the problem at hand. In their sensor network application, the scaling parameter was tuned by cross validation.

3.1.1.3 Case where $\mathbf{K} = \mathbf{I}_N$ and similarities with UCB1

If $\mathbf{K} = \mathbf{I}_N$, the arm feature representations are orthogonal. Since the dimensions of $\mathbf{w}$ are independent, the $f(i)$ variates, hence the arms, are independent too.

Posterior mean and variance Let us re-order the arms in the training data so that the $\nu(i,t)$ first arms that were played are i (this does not affect the values of the posterior mean and variance). We write $\mathbf{A}$ the covariance matrix between the remaining training inputs. This gives:

$$\begin{aligned}
\mathbf{k}_t(i) &= (1 \dots 1\ 0 \dots 0)^\mathsf{T} \\
\mathbf{C}_t &= \begin{pmatrix} 1+s^2_{\text{noise}} & & & & \\ & \ddots & 1 & & 0 \\ & 1 & \ddots & & \\ & & & 1+s^2_{\text{noise}} & \\ & & & & \\ & 0 & & & \mathbf{A} \end{pmatrix} \\
\mathbf{C}_t\mathbf{k}_t(i) &= (\nu(i,t)+s^2_{\text{noise}}\ \dots\ \nu(i,t)+s^2_{\text{noise}}\ 0\ \dots\ 0)^\mathsf{T} \\
&= (\nu(i,t)+s^2_{\text{noise}})\mathbf{k}_t(i) \\
\mathbf{C}_t^{-1}\mathbf{k}_t(i) &= \frac{1}{\nu(i,t)+s^2_{\text{noise}}}\mathbf{k}_t(i)
\end{aligned}$$

From this and Equations (2.12) and (2.13) we derive the following expressions for the posterior mean and variance:

$$\begin{aligned}
\mu_t(i) &= \frac{1}{\nu(i,t)+s^2_{\text{noise}}}\sum_{j=1}^{\nu(i,t)} y_j \\
\sigma_t^2(i) &= 1 - \frac{1}{\nu(i,t)+s^2_{\text{noise}}}\mathbf{k}_t(i)^\mathsf{T}\mathbf{k}_t(i) \\
&= \sqrt{\frac{s^2_{\text{noise}}}{\nu(i,t)+s^2_{\text{noise}}}}
\end{aligned}$$

Regret bound Even though the regret has a different nature in the Bayesian setting (it is a random variable since f is not fixed anymore but drawn from a probability distribution), we could adapt the regret analysis of UCB1 and use the GP error bars in place of the Hoeffding inequalities in order to determine confidence intervals for $f(i)$. As seen in Section 2.1.2.2, we would need to have

$$\begin{aligned}
\sum_{t=1}^{\infty} t^2\mathbb{P}(f^* \geq \mu_t(i^*) + \sqrt{\beta}_t\sigma_t(i^*) \text{ or } f(i) \leq \mu_t(i) - \sqrt{\beta}_t\sigma_t(i)) &= \sum_{t=1}^{\infty} t^2 \text{erfc}\left(\sqrt{\frac{\beta_t}{2}}\right) \\
&< \infty
\end{aligned}$$

so that the ideas of the UCB1 regret proof can be applied. This is the case for $\beta_t > 6\log(t)$. One could thus give a problem-specific regret upper bound with high probability in $O(\log(T))$ for GPB. In the general case where $\mathbf{K}$ is not necessarily the identity matrix, we can expect the posterior variance to be even lower (sharing information between arms helps to reduce the variance). We will see in the second section of this chapter how the regret relates to spectral properties of this matrix, namely the rate of decay of the eigenvalues.

Relationship to UCB1 For $\beta_t = \frac{4 \log t}{s^2_{\text{noise}}}$ the upper confidence function for arm i becomes:

$$\frac{1}{\nu(i,t) + s^2_{\text{noise}}} \sum_{\tau=1 \dots t \text{ s.t. } i_\tau = i} y_\tau + \sqrt{\frac{2 \log t}{\nu(i,t) + s^2_{\text{noise}}}}$$

which tends to the value of the UCB1 upper confidence function when s_{noise} tends to 0.

3.1.1.4 Remarks on the UCB maximisation for infinitely many arms

GPB can also be used in infinite spaces of arms. If the horizon T is known in advance, β_t can be replaced by a constant; otherwise, we give an expression for β_t in Proposition 3 on page 82. While our model stays the same, infinite spaces introduce difficulties in finding the upper confidence maximiser. In our approach, the problem of finding the maximum of the function f is replaced by iterations of a simpler problem, which is to maximise the function f_t, given in closed form. In the case where the kernel is normalised ($\kappa(\mathbf{x}, \mathbf{x}) = 1$ for all $\mathbf{x}$, as with the Gaussian and the cosine kernels), $f_t(\mathbf{x})$ is a concave function of $\mathbf{k}_t(\mathbf{x})$ and we look for the maximum over $\mathbf{k}$ of the following expression:

$$\mathbf{k}^\mathsf{T} \mathbf{C}_t^{-1} \mathbf{y}_t + \sqrt{\beta_t (1 - \mathbf{k}^\mathsf{T} \mathbf{C}_t^{-1} \mathbf{k})}$$

However, in most cases, the constraints on $\mathbf{k}$ that result from the fact that $\exists \mathbf{x}, \mathbf{k} = \mathbf{k}_t(\mathbf{x})$ cannot all be written in the form required for convex optimisation, which would be:

$$\begin{aligned} g_i(\mathbf{k}) &\leq 0 \text{ where } g_i \text{ is convex} \\ h_i(\mathbf{k}) &= 0 \text{ where } h_i \text{ is affine} \end{aligned}$$

We could optimise without considering these constraints, but this could lead to situations where we obtain a maximiser $\mathbf{k}^*$ such that there are no solutions to $\mathbf{k}(\mathbf{x}) = \mathbf{k}^*$. We could look for $\mathbf{x}$ such that $\mathbf{k}(\mathbf{x})$ approximates $\mathbf{k}^*$ – for Gaussian kernels, with triangulation for instance. However, our attempts have been unsuccessful in practise, as the f_t value at the chosen location was found to be far from its maximum.

We have mentioned techniques to deal with the maximisation of f_t in Section 2.4.2.2. The most straightforward approach is probably to discretise the input space. The problem of a fixed-size discretisation is that the regret in T is linear. Nonetheless, if we know the total number of iterations in advance, we can make the discretisation depend on T and avoid this problem. HOO and the Zooming Algorithm do not need to fix T in advance: they constantly refine the discretisation in regions of the input space that they find "interesting". For instance, the strategy of the Zooming Algorithm is, when playing an arm $\mathbf{x}$ already played before, to add other arms close to $\mathbf{x}$ to the current set of arms being considered.

3.1.2 Computational analysis, optimisations and approximations

Assuming the set of arms is finite and fixed, we now study the cost of updating the upper confidence bounds of the GPB algorithm after observing $(\mathbf{x}_{t+1}, y_{t+1})$. We can expect it to be in $O((t+1)^\alpha) = O(t^\alpha)$ where α is an integer, hence the cost of one iteration to be in $O(t^\alpha + N)$ by adding the cost of finding the upper confidence maximiser among the N arms. The total cost of T iterations of the algorithm would thus be $C_T = O(T^{\alpha+1} + N\,T)$. We write C_T as a function of N too, so that the expression can be used for the case where N is a function of time. We assume that the algorithm is allowed to learn its hyper-parameters only up to a fixed time T_0, so that the kernel matrix is fixed thereafter, and the extra computational cost associated to this transitory phase can be considered a constant.

3.1.2.1 Default algorithm

Cost At each iteration $t+1$, once $\mathbf{x}_{t+1}$ has been chosen, we have to do the following in order to determine $\boldsymbol{\mu}_{t+1}$ and $\boldsymbol{\sigma}_{t+1}$:

- Compute the covariance matrix inverse $\mathbf{C}_{t+1}^{-1}$, used for the posterior mean and variance computation: $O((t+1)^3)$.
- Compute:

$$\boldsymbol{\mu}_{t+1} = \mathbf{K}(\mathcal{I}_{t+1}, \mathcal{I})^\mathsf{T} \mathbf{C}_{t+1}^{-1} \mathbf{y}_{t+1} \tag{3.1}$$

The cost of this step is $O((t+1)^2)$ for the right-most product and $O(N(t+1))$ for the remaining product, i.e. $O(N\ t + t^2)$.

- Compute $\boldsymbol{\sigma}_{t+1}^2$: this is more expensive as we have to compute a $\mathbf{k}_{t+1}(\mathbf{a}_i)^\top \mathbf{C}_{t+1}^{-1} \mathbf{k}_{t+1}(\mathbf{a}_i)$ term for each $\mathbf{a}_i$. This implies that we either perform a loop over the $\mathbf{a}_i$'s, or, in matrix form, we write:

$$\boldsymbol{\sigma}_{t+1}^2 = \mathrm{diag}(\mathbf{K} - \mathbf{K}(\mathcal{I}_{t+1}, \mathcal{I})^\top \mathbf{C}_{t+1}^{-1} \mathbf{K}(\mathcal{I}_{t+1}, \mathcal{I})) \tag{3.2}$$

  In both cases, the cost of this step is $O(N\ (t+1)^2)$.

As a consequence, $C_T = O(T^4 + N\ T^3)$.

Iterative matrix inversion It is possible to reuse the covariance matrix inverse at time t in order to compute the inverse at time $t+1$ more efficiently. For this, we write:

$$\begin{aligned}
\mathbf{C}_{t+1}^{-1} &= \begin{pmatrix} \mathbf{C}_t & \mathbf{k}_t(\mathbf{x}_{t+1}) \\ \mathbf{k}_t(\mathbf{x}_{t+1})^\top & \kappa(\mathbf{x}_{t+1}, \mathbf{x}_{t+1}) + s_{\text{noise}}^2 \end{pmatrix}^{-1} \\
&= \begin{pmatrix} \mathbf{A}_t & \mathbf{b}_t \\ \mathbf{b}_t^\top & d_t \end{pmatrix} \\
\text{where } \mathbf{A}_t &= \mathbf{C}_t^{-1} + \frac{\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1})(\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1}))^\top}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \\
\mathbf{b}_t &= -\frac{\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \\
d_t &= \frac{1}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2}
\end{aligned}$$

This technique is sometimes referred to as the Sherman-Woodbury-Morrison matrix inversion in the literature. We thus see that the covariance matrix inverse can be computed in $O(t^2)$, and therefore $C_T = O(N\ T^3)$.

Parallelisation We remark that the updates for all N arms could be done in parallel, so the factor in front of T^3 could be reduced, depending on the number of processors available.

3.1.2.2 Online updates

We can take advantage of the previous recursive formulation and of the fact that $\mathbf{k}_{t+1}(\mathbf{x})^\top = (\mathbf{k}_t(\mathbf{x})^\top\ \kappa(\mathbf{x}, \mathbf{x}_{t+1}))$ and $\mathbf{y}_{t+1}^\top = (\mathbf{y}_t^\top\ y_{t+1})$, in order to derive sequential update formulae for μ and σ^2:

$$\begin{aligned}
\mu_{t+1}(\mathbf{x}) &= \mu_t(\mathbf{x}) + (y_{t+1} - \mu_t(\mathbf{x}_{t+1}))\frac{\kappa(\mathbf{x}, \mathbf{x}_{t+1}) - \mathbf{k}_t(\mathbf{x})^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \\
\sigma_{t+1}^2(\mathbf{x}) &= \sigma_t^2(\mathbf{x}) - \frac{(\kappa(\mathbf{x}, \mathbf{x}_{t+1}) - \mathbf{k}_t(\mathbf{x})^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1}))^2}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2}
\end{aligned}$$

It is easy to see, in this formulation, that i) the amount of change of the estimated reward for $\mathbf{x}$ is small when the posterior covariance at time t between $\mathbf{x}$ and $\mathbf{x}_{t+1}$ is small, or when the new reward sample y_{t+1} agrees with the estimated reward for $\mathbf{x}$ at time t, and ii) the posterior variance can only decrease: the bigger the posterior covariance at time t between $\mathbf{x}$ and $\mathbf{x}_{t+1}$, the bigger the decrease.

We can make a quick sanity check for $\mathbf{x} = \mathbf{x}_{t+1}$, independent arms, and $\mathbf{x}_{t+1}$ played once, so that $\sigma_t^2(\mathbf{x}_{t+1}) = s_{\text{noise}}^2$: in that case, $\mu_{t+1}(\mathbf{x}_{t+1})$ is the average between $\mu_t(\mathbf{x}_{t+1})$ and y_{t+1}, and $\sigma_{t+1}^2(\mathbf{x}_{t+1})$ is half of $\sigma_t^2(\mathbf{x}_{t+1})$.

We also see that all updates are expressed in terms of $\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1})$. We thus have to do the following in order to update $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$:

- Update the covariance matrix inverse: $O(t^2)$
- Compute $\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1})$: $O(t^2)$.
- For all $1 \leq i \leq N$:
  - Update μ and σ^2 for $\mathbf{a}_i$, which involves the computation of an inner product, and operations on scalars: $O(t)$.

The overall cost becomes: $C_T = O(T^3 + N\,T^2)$. We refer to this version of the algorithm as 'GPB-ONLINE1'.

Improved formulation The computational cost of an iteration can further be improved by computing and memorising the list of vectors $\mathbf{q}_t(\mathbf{x}) = \mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x})$ for $\mathbf{x} \in \{\mathbf{a}_1, \ldots, \mathbf{a}_N\}$. We write:

$$\alpha_{t+1}(\mathbf{x}) = \kappa(\mathbf{x}, \mathbf{x}_{t+1}) - \mathbf{q}_t(\mathbf{x}_{t+1})^\mathsf{T}\mathbf{k}_t(\mathbf{x}) \tag{3.3}$$

$$\mu_{t+1}(\mathbf{x}) = \mu_t(\mathbf{x}) + (y_{t+1} - \mu_t(\mathbf{x}_{t+1}))\frac{\alpha_{t+1}(\mathbf{x})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \tag{3.4}$$

$$\sigma_{t+1}^2(\mathbf{x}) = \sigma_t^2(\mathbf{x}) - \frac{\alpha_{t+1}(\mathbf{x})^2}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \tag{3.5}$$

These expressions do not involve the covariance matrix anymore, and $\mathbf{q}_{t+1}(\mathbf{x})$ is updated as follows:

$$\mathbf{q}_{t+1}(\mathbf{x}) = \begin{pmatrix} \mathbf{q}_t(\mathbf{x}) - \frac{\alpha_{t+1}(\mathbf{x})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \mathbf{q}_t(\mathbf{x}_{t+1}) \\ \frac{\alpha_{t+1}(\mathbf{x})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \end{pmatrix} \tag{3.6}$$

which has a cost in $O(t)$ for each $\mathbf{x}$.

The procedure to update $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ is now:

- For all $1 \leq i \leq N$:
  - Compute $\alpha_{t+1}(\mathbf{a}_i)$, based on $\mathbf{q}_t(\mathbf{x}_{t+1})$: $O(t)$;
  - Compute $\mu_{t+1}(\mathbf{a}_i)$ and $\sigma_{t+1}(\mathbf{a}_i)$, based on $\alpha_{t+1}(\mathbf{a}_i)$: $O(1)$;
  - Compute $\mathbf{q}_{t+1}(\mathbf{a}_i)$: $O(t)$.

This gives: $C_T = O(N\ T^2)$. Note that the memory requirements are different: there are $t \times N$ values to memorise for $\mathbf{q}$, and N for α. We refer to this version of the algorithm as 'GPB-online2'. Although the recursive formulation is similar in spirit to recursive least-squares estimation, we believe that the "trick" of memorising the $q_t(\mathbf{x})$ vectors is novel.

Numerical stability The previous formulae illustrate the numerical stability problems one encounters with GPs with very small, or zero noise. As we get training samples that are close to each other, the uncertainty about these samples gets very low and the $\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2$ denominator tends to zero. One way to deal with this is to not update μ_{t+1} values if $\sigma_t(\mathbf{x}_{t+1})$ is below a certain threshold. If all σ_t values are below that threshold, we switch to the greedy policy – there is nothing more to learn and we assume we have converged to the true f.

LinRel We can also use the "$\mathbf{q}_t$ trick" for LinRel where $\sigma_t^2(\mathbf{x}) = \mathbf{q}_t^\top(\mathbf{x})\mathbf{q}_t(\mathbf{x})$, hence its computational cost is the same as GPB's.

When N is relatively small ($\leq T$) Let us write $p_t(\mathbf{x}, \mathbf{x}') = \mathbf{k}_t(\mathbf{x})^\top \mathbf{C}_t^{-1} \mathbf{k}_t(\mathbf{x}')$ and $\mathbf{P}_t$ for the $N \times N$ matrix of such values for $\mathbf{x}, \mathbf{x}' \in \{\mathbf{a}_1, \ldots, \mathbf{a}_N\}$. We have:

$$\begin{aligned} \mu_{t+1}(\mathbf{x}) &= \mu_t(\mathbf{x}) + (y_{t+1} - \mu_t(\mathbf{x}_{t+1})) \frac{\kappa(\mathbf{x}, \mathbf{x}_{t+1}) - p_t(\mathbf{x}, \mathbf{x}_{t+1})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \\ \sigma_{t+1}^2(\mathbf{x}) &= \sigma_t^2(\mathbf{x}) - \frac{(\kappa(\mathbf{x}, \mathbf{x}_{t+1}) - p_t(\mathbf{x}, \mathbf{x}_{t+1}))^2}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \end{aligned}$$

P can be updated in $O(N^2)$ owing to the following recursive formulation:

$$\begin{aligned} p_{t+1}(\mathbf{x},\mathbf{x}') &= p_t(\mathbf{x},\mathbf{x}') + \frac{1}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2}(p_t(\mathbf{x},\mathbf{x}_{t+1})p_t(\mathbf{x}',\mathbf{x}_{t+1}) \\ &\quad -\kappa(\mathbf{x}',\mathbf{x}_{t+1})p_t(\mathbf{x},\mathbf{x}_{t+1}) - \kappa(\mathbf{x},\mathbf{x}_{t+1})p_t(\mathbf{x}',\mathbf{x}_{t+1}) \\ &\quad +\kappa(\mathbf{x},\mathbf{x}_{t+1})\kappa(\mathbf{x}',\mathbf{x}_{t+1})) \end{aligned}$$

Once **P** is known, the computation of $\mathbf{C}^{-1}$ is not required anymore, there are no more matrix-vector products to be performed, and the $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ updates can be done in $O(N)$, which means that the cost of iteration $t+1$ is $O(N^2)$ and $C_T = O(N^2\ T)$.

We refer to this version of the algorithm as 'GPB-ONLINE3'. It is particularly interesting for relatively small values of N, for instance when the number of iterations of the algorithm grows larger than N. Note that this cost is the same as for GP inference in the weight-space view. Also, for UCB1, $T \geq N$ and the cost of T iterations is in $O(N\ T)$.[1] We thus have a linear cost for both algorithms, with a constant in N for UCB1 and in N^2 for GPB-ONLINE3, and we have regret bounds in $\tilde{O}(\sqrt{T})$ in both cases – but supposedly with a better constant for GPB, as we learn more from each sample.

When N is large The exhaustive search for the upper confidence maximiser implies a cost in $O(N)$ at least, which can be very large. We will see in the next chapter how to exploit tree-like dependency structures among arms in order to only consider a subset of the whole set of arms that grows with t.

3.1.2.3 Reducing the size of the training set: GPB-RED

We can reduce the computational cost of GPB by reducing the size of the training set. For example, at each iteration we add an arm to the training set, but we can also decide to remove the oldest sample in the training set if the size of $\mathcal{D}_t$ is bigger than a certain function S_t. This new version of GPB will be referred to as GPB-RED. An interesting thing to notice with GPB-RED is that, when removing a data point from the training set, we increase the GP posterior variance at this point and points close to it. Therefore we give them more chances to be selected, as increasing the variance increases f_t, and thus GPB-RED explores more than GPB – empirical evidence of this was given by

[1]The cost of an update is constant, but the cost to find the arm with highest upper confidence bound scales linearly with N.

Dorard et al. (2009).

The cost of updating the posterior mean and variance at iteration t of GPB-RED is obtained by replacing t by a bound on the number of elements in training, S_t – which is smaller than t. We must also determine the cost of recomputing the posterior after the oldest data point has been removed (*downdate*).

Online downdates Let us write $\mathcal{I}_{t,l} = \{i_{t-l+1}, \dots, i_t\}$ and $\mathcal{D}_{t,l}$ for the corresponding training set, i.e. the training set at time t restricted to the l last elements that have been observed. We denote by a t, l subscript all quantities that are based on $\mathcal{I}_{t,l}$ instead of $\mathcal{I}_t$. We write:

$$\begin{aligned}
\mathbf{C}_{t,l+1}^{-1} &= \begin{pmatrix} d_l & \mathbf{b}_l^\top \\ \mathbf{b}_l & \mathbf{A}_l \end{pmatrix} \\
&= \begin{pmatrix} \kappa(\mathbf{x}_{t-l}, \mathbf{x}_{t-l}) + s_{\text{noise}}^2 & \mathbf{k}_{t,l}(\mathbf{x}_{t-l})^\top \\ \mathbf{k}_{t,l}(\mathbf{x}_{t-l}) & \mathbf{C}_{t,l} \end{pmatrix}^{-1}
\end{aligned}$$

$\mathbf{C}_{t,l}^{-1}$ can thus be determined from the block matrices that make up $\mathbf{C}_{t,l+1}^{-1}$:

$$\mathbf{C}_{t,l}^{-1} = \mathbf{A}_l - \frac{\mathbf{b}_l^\top \mathbf{b}_l}{d_l}$$

We thus see that the covariance matrix can be downdated in $O(l^2)$ when removing $\mathbf{x}_{t-l}$.

Similarly to the previous online updates formulae, we can show that

$$\mu_{t,l+1}(\mathbf{x}) = \mu_{t,l}(\mathbf{x}) + (y_{t-l} - \mu_{t,l}(\mathbf{x}_{t-l}))\frac{\kappa(\mathbf{x}, \mathbf{x}_{t-l}) - \mathbf{k}_{t,l}(\mathbf{x})^\top \mathbf{C}_{t,l}^{-1}\mathbf{k}_{t,l}(\mathbf{x}_{t-l})}{\sigma_{t,l}^2(\mathbf{x}_{t-l}) + s_{\text{noise}}^2} \tag{3.7}$$

$$\sigma_{t,l+1}^2(\mathbf{x}) = \sigma_{t,l}^2(\mathbf{x}) - \frac{(\kappa(\mathbf{x}, \mathbf{x}_{t-l}) - \mathbf{k}_{t,l}(\mathbf{x})^\top \mathbf{C}_{t,l}^{-1}\mathbf{k}_{t,l}(\mathbf{x}_{t-l}))^2}{\sigma_{t,l}^2(\mathbf{x}_{t-l}) + s_{\text{noise}}^2} \tag{3.8}$$

We have to do the following in order to downdate $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ when discarding the $(l+1)$th last observation:

- Downdate the covariance matrix inverse: $O(l^2)$
- Compute $\boldsymbol{\mu}_{t,l} = \mathbf{K}(\mathcal{I}_{t,l}, \mathcal{I})^\top \mathbf{C}_{t,l}^{-1}\mathbf{y}_{t,l}$: $O(l^2)$ for the right-most product, $O(N\ l)$ for the remaining product
- Compute $\sigma_{t,l}^2(\mathbf{x}_{t-l}) = \kappa(\mathbf{x}_{t-l}, \mathbf{x}_{t-l}) - \mathbf{k}_{t,l}(\mathbf{x}_{t-l})^\top \mathbf{C}_{t,l}^{-1}\mathbf{k}_{t,l}(\mathbf{x}_{t-l})$: $O(l^2)$

- Compute $\boldsymbol{\sigma}^2_{t,l}$:
  - Determine the list of $\kappa(\mathbf{x}, \mathbf{x}_{t-l}) - \mathbf{k}_{t,l}(\mathbf{x})^\mathsf{T}\mathbf{C}_{t,l}^{-1}\mathbf{k}_{t,l}(\mathbf{x}_{t-l})$ values for all arms, based on Equation (3.7) and the fact that $\boldsymbol{\mu}_{t,l}$, $\boldsymbol{\mu}_{t,l+1}$ and $\sigma^2_{t,l}(\mathbf{x}_{t-l})$ are already known: $O(N)$
  - Use these values to compute $\sigma^2_{t,l}(\mathbf{x})$ for all arms, using Equation (3.8) and the fact that $\boldsymbol{\sigma}^2_{t,l+1}$ is already known: $O(N)$

The total cost of this is $O(l^2 + N\ l)$.

Cost of GPB-red The cost of iteration $t+1$ is equal to the cost of an online update, plus potentially the cost of an online downdate. With the GPB-ONLINE1 algorithm, this is twice $O(S_t^2 + N\ S_t)$. The overall cost of T iterations C_T is thus bounded by $O(T(S_T^2 + N\ S_T) + N\ T) = O(T(S_T^2 + N\ S_T))$. If S is a constant function, the cost becomes linear, as it is with UCB1 for instance. However, if the value of S depends on N, the constant will be bigger than with UCB1. If S is a logarithmic function, then $C_T = \tilde{O}(T)$.

Remarks

- Although we do not give theoretical guarantees on the performance of GPB-RED, it seems preferable, if we want to keep the no-regret property, to impose at least to S to be a strictly increasing function that tends to infinity, so that there is no T_* such that no more data is added to the training set after T_*.

- An alternative way to decide when to remove samples would be by dropping the oldest sample of the training set as long as the resulting σ_t values do not exceed a certain threshold. We would need to determine a bound on the size of the training set at time T with this method if we want to determine its computational complexity.

- Choosing to remove the oldest sample is somewhat arbitrary: it may be better, instead, to remove samples based on the amount of information they provide.

3.2 Theoretical analysis

3.2.1 Overview

The GPB algorithm was analysed by Srinivas et al. in the cases of finite and infinite number of arms, under the assumption that f is drawn from a Gaussian Process with zero mean and given covariance function, and in a more agnostic setting where f has low complexity as measured under the RKHS norm induced by a given kernel. The rest of this chapter focuses on their work, reviewed in Propositions 1 to 4 and in the proofs to which we added details. Their results will be core to the regret bounds we give in Section 4.3.3. It is important to note that, whereas f was previously fixed, in the GP setting it is now a random variate. As a consequence, the cumulative regret defined in Section 2.1.1.1 is now a random quantity, and we aim to upper bound it with high probability with respect to the choice of f.

The regret analysis is based on a quantification of the reduction in uncertainty caused by the observation of data, through the *information gain*. If $\mathcal{A}$ is a subset of $\mathcal{X}$, getting a sample of outputs $\mathbf{y}_\mathcal{A}$ for the elements in this set reduces our uncertainty about the set of f values for all arms, $\mathbf{f}$. The information gain associated to $\mathcal{A}$ is defined as the mutual information between $\mathbf{y}_\mathcal{A}$ and $\mathbf{f}$, i.e. the entropy of $\mathbf{f}$ minus the entropy of $\mathbf{f}$ given $\mathbf{y}_\mathcal{A}$:

$$G(\mathcal{A}) = G(\mathbf{y}_\mathcal{A}; \mathbf{f}) = H(\mathbf{f}) - H(\mathbf{f}|\mathbf{y}_\mathcal{A}) \tag{3.9}$$

G is a monotonic function (Cover and Thomas, 1991): $\mathcal{A} \subset \mathcal{A}' \Rightarrow G(\mathcal{A}) \leq G(\mathcal{A}')$. It is also a submodular function (Krause and Guestrin, 2005): if $\mathcal{A} \subset \mathcal{A}'$, we gain less information when adding a new element to $\mathcal{A}'$ than when adding it to $\mathcal{A}$ (property of diminishing returns).

Proposition 1. *We restate Theorem 1 of Srinivas et al. (2010). We write G_T^u for the information gain after acquiring T samples iteratively by maximisation of the upper confidence function in a finite space $\mathcal{X}$. Assume that f is drawn from a Gaussian Process with zero mean and given covariance function κ, and that $\kappa(\mathbf{x}, \mathbf{x}) = \sigma_0^2$ for all $\mathbf{x}$. For any given δ between 0 and 1 and β_t defined accordingly (see 3.1.1.2 on page 65), we have:*

$$P\left(\forall T > 1, R_T \leq \sqrt{\frac{8\sigma_0^2}{\log(1 + s_{\text{noise}}^{-2}\sigma_0^2)}\beta_{T-1} T G_T^u}\right) \geq 1 - \delta$$

We can also write, "with high probability":

$$R_T = \tilde{O}(\sqrt{TG_T^u})$$

A proof is given in Section 3.2.2.

The distributions being Gaussian (Gaussian process and Gaussian noise), the information gain G_T^u is expressed in terms of the log determinant of $\mathbf{K}_T + s_{\text{noise}}^2 \mathbf{I}_T$. It can easily be written in terms of the eigenvalues of $\mathbf{K}_T$. The simplest case is for a linear kernel in d dimensions, which we illustrate in Section 3.2.4.1. However, in general there is no simple expression for these eigenvalues since we do not know which arms have been played.[2]

Instead, we aim to bound the *max infogain (maximum possible information gain)*, $\max_{\mathcal{A}\subset\mathcal{X},|\mathcal{A}|=T} G(\mathcal{A})$, which measures how quickly the function can be learnt in an information theoretic sense. We actually consider the max infogain G_T^* in the extended space $\mathcal{X}^e$ of linear combinations of the elements in $\mathcal{X}$ such that the vector of coefficients has norm 1, for a reason that will become clear later:

$$\mathcal{X}^e = \left\{ \sum_{i=1}^{N} v_i \mathbf{a}_i \text{ s.t. } ||\mathbf{v}|| = 1 \right\} \supset \mathcal{X}$$

Clearly, we have $G_T^* \geq G_T^u$. Intuitively, a small growth rate of the max infogain means that there is not much information left to be gained after some time, hence that we can learn quickly, which should result in small regrets. We can expect the max infogain to be a problem-dependent quantity, and that its growth is determined by properties of the kernel and of the input space. We write G_T^g for the *greedy infogain* after acquiring T samples in $\mathcal{X}^e$, i.e. the information gain of a "greedy" algorithm that selects at each iteration the arm that maximises its immediate information gain. Because the information gain is a monotonic and a sub-modular function, Nemhauser et al. (1978) state that:

$$G_T^* \leq \frac{1}{1-e^{-1}} G_T^g$$

where $e = \exp(1)$.

Proposition 2. *We write $\hat{\lambda}_1 \geq \ldots \geq \hat{\lambda}_N$ for the eigenvalues of $\mathbf{K}$. The greedy infogain can be bounded as follows:*

$$G_T^g = \frac{1}{2} \sum_{i=1}^{\min(T,N)} \log(1 + s_{\text{noise}}^{-2} m_i \hat{\lambda}_i) \tag{3.10}$$

[2]The process of selecting arms is non-deterministic because of the noise in the observations.

where $(m_i)_i$ is a sequence of positive or zero values which sum is equal to T and which maximises the expression above. A proof is given in Section 3.2.3.

Remarks

- $\min(T, N) \leq N$ hence $G_T^g = \tilde{O}(N)$ and thus:

$$R_T = \tilde{O}(\sqrt{N\, T}) \tag{3.11}$$

  However, in some applications, $N \gg T$, and it is more interesting to bound R_T in terms of T rather than N, when possible.

- We may be interested in cases where N depends on T, as in MDP planning (with deterministic transitions) where the tree of all possible sequences of actions has infinite depth and we typically choose D as a function of the horizon T (see Section 2.3.3).

3.2.2 Bounding the regret with the information gain

In this section we prove Proposition 1.

Information gain of a GP-based algorithm We start by writing:

$$G(\mathcal{D}_T) = H(\mathbf{f}) - H(\mathbf{f}|\mathbf{y}_T) = H(\mathbf{y}_T) - H(\mathbf{y}_T|\mathbf{f})$$

- The first term can be expressed recursively: $H(\mathbf{y}_T) = H(\mathbf{y}_{T-1}) + H(y_T|\mathbf{y}_{T-1})$. Conditioned on $\mathbf{y}_{T-1}$, the $(\mathbf{x}_t)_{1\leq t\leq T}$ are deterministic and thus $f(\mathbf{x}) \sim \mathcal{N}(0, \sigma_{T-1}^2(\mathbf{x}))$ for all $\mathbf{x}$. $y_T = f(\mathbf{x}_T) + \epsilon_T$ is a sum of two zero-mean Gaussians: one with variance equal to $\sigma_{T-1}^2(\mathbf{x}_T)$ and the other with variance equal to s_{noise}^2. By application of Equation (A.2) we have $H(y_T|\mathbf{y}_{T-1}) = \log(2\pi e(s_{\text{noise}}^2 + \sigma_{T-1}^2(\mathbf{x}_T)))$, from which we deduce:

$$H(\mathbf{y}_T) = \frac{1}{2}\sum_{t=1}^{T} \log(2\pi e(s_{\text{noise}}^2 + \sigma_{t-1}^2(\mathbf{x}_t)))$$

- The second term is easily determined, using the fact that $\mathbf{y}_T$ conditioned on $\mathbf{f}$ follows a zero-mean normal distribution with covariance matrix equal to $s_{\text{noise}}^2 \mathbf{I}_T$:

$$\begin{aligned} H(\mathbf{y}_T|\mathbf{f}) &= \frac{1}{2}\log(\left|2\pi e s_{\text{noise}}^2 \mathbf{I}_T\right|) \\ &= \frac{1}{2}\sum_{t=1}^{T}\log(2\pi e s_{\text{noise}}^2) \end{aligned}$$

Combining these two terms and bringing each term of the sum over t under one log, we get:

$$G(\mathcal{D}_T) = \frac{1}{2}\sum_{t=1}^{T} \log(1 + s_{\text{noise}}^{-2}\sigma_{t-1}^2(\mathbf{x}_t)) \tag{3.12}$$

Bound on the sum of squared immediate regrets Recall that, for a UCB-type algorithm, the immediate regret was bounded in terms of $\sigma_t(\mathbf{x}_{t+1})$ in Inequality (2.6). We now try to relate the immediate regret to an expression in $\log(1 + s_{\text{noise}}^{-2}\sigma_{t-1}^2(\mathbf{x}_t))$, so that later we can bound the regret in terms of the information gain. From Inequality (2.6), $r_t^2 \leq 4\beta_{T-1}s_{\text{noise}}^2 a$ where $a = s_{\text{noise}}^{-2}\sigma_{t-1}^2(\mathbf{x}_t) \leq s_{\text{noise}}^{-2}\sigma_0^2 = b$. We have $a \leq \frac{b}{\log(1+b)}\log(1+a)$ because $\frac{x}{\log(1+x)}$ is an increasing function, which gives:

$$r_t^2 \leq \frac{4\sigma_0^2}{\log(1 + s_{\text{noise}}^{-2}\sigma_0^2)}\beta_{T-1}\log(1 + s_{\text{noise}}^{-2}\sigma_{t-1}^2(\mathbf{x}_t))$$

Using Equation (3.12), we thus have:

$$\sum_{t=1}^{T} r_t^2 \leq \frac{8\sigma_0^2}{\log(1 + s_{\text{noise}}^{-2}\sigma_0^2)}\beta_{T-1}G_T^u \tag{3.13}$$

We can relate the cumulative regret R_T to $\sum r_t^2$ using the Cauchy-Schwarz inequality and the fact that R_T is the inner product between the vector of regrets r_t at each time $1 \leq t \leq T$ and the T-dimensional vector of ones.

$$R_T^2 \leq T\sum_{t=1}^{T} r_t^2$$

Combining this with Inequality (3.13) proves Proposition 1.

Remark on the noise-free case It is not straightforward to adapt the current analysis to the case where rewards are deterministic ($s_{\text{noise}} = 0$). Indeed, the information gain at time t in this case is equal to $H(\mathbf{y}_{t-1}) + H(y_t|\mathbf{y}_{t-1}) = H(\mathbf{y}_{t-1}) + 1/2\log(2\pi e\sigma_{t-1}^2(\mathbf{x}_t))$ and thus it is equal to $\sum_{t=1}^{T} 1/2\log(2\pi e\sigma_{t-1}^2(\mathbf{x}_t))$: the trick we used to relate the information gain to the sum of squared regrets cannot be applied here.

3.2.3 Bounding the information gain with the eigenvalues of the total kernel matrix

In this section we prove Proposition 2.

The greedy algorithm chooses at each time step the arm that maximises the immediate information gain:

$$\begin{aligned} G(\mathcal{D}_{t-1} \cup \{\mathbf{x}, y\}) - G(\mathcal{D}_{t-1}) &= H(\mathbf{y}_t) - H(\mathbf{y}_{t-1}) - (H(\mathbf{y}_t|\mathbf{f}) - H(\mathbf{y}_{t-1}|\mathbf{f})) \\ &= H(y_t|\mathbf{y}_{t-1}) - H(y_t|\mathbf{y}_{t-1}, \mathbf{f}) \\ &= 1/2 \log(2\pi e(s^2_{\text{noise}} + \sigma^2_{t-1}(\mathbf{x}))) - 1/2 \log(2\pi e s^2_{\text{noise}}) \end{aligned}$$

We thus see that this is equivalent to maximising the posterior variance.

We write $\mathbf{\Sigma}_t$ for the posterior covariance matrix on the whole set of arms at time t for the greedy algorithm. $\mathbf{x}$ can be written as $\sum_{i=1}^N v_i \mathbf{a}_i$ where $||\mathbf{v}|| = 1$.

$$\begin{aligned} \sigma^2_t(\mathbf{x}) &= \text{cov}_t(\mathbf{x}, \mathbf{x}) \\ &= \sum_{i_1} \sum_{i_2} v_{i_1} v_{i_2} \text{cov}_t(\mathbf{a}_{i_1}, \mathbf{a}_{i_2}) \\ &= \sum_{i_1} \sum_{i_2} v_{i_1} v_{i_2} (\mathbf{\Sigma}_t)_{i_1, i_2} \\ &= \mathbf{v}^\mathsf{T} \mathbf{\Sigma}_t \mathbf{v} \end{aligned}$$

Choosing an arm $\mathbf{x}$ is equivalent to choosing a vector $\mathbf{v}$ of norm 1. We denote by $\mathbf{v}_t$ the vector that corresponds to the arm chosen at time t by the greedy algorithm.

$$\mathbf{v}_{t+1} = \text{argmax}_{||\mathbf{v}||=1} \mathbf{v}^\mathsf{T} \mathbf{\Sigma}_t \mathbf{v} \tag{3.14}$$

3.2.3.1 The greedy posterior covariance matrix has same eigenbasis as $\mathbf{K}$

Relationship between the two matrices We have

$$\mathbf{\Sigma}_t^{-1} = \mathbf{K}^{-1} + s^{-2}_{\text{noise}} \mathbf{V}_t \mathbf{V}_t^\mathsf{T} \tag{3.15}$$

where $\mathbf{V}_t$ is the matrix of concatenated $\mathbf{v}_\tau$ vectors (for τ from 1 to t). Using (A.5) with the fact that $\mathbf{\Sigma}_t$ is the covariance matrix of $\mathbb{P}(\mathbf{f}|\mathbf{y}_t)$ and the vector $[\mathbf{f}\ \mathbf{y}_t]^\mathsf{T}$ is drawn from a Gaussian with zero mean and covariance matrix:

$$\begin{pmatrix} \mathbf{K} & \mathbf{K}\mathbf{V}_t \\ \mathbf{V}_t^\mathsf{T}\mathbf{K} & \mathbf{K}_t + s^2_{\text{noise}} \mathbf{I}_t \end{pmatrix}$$

Recursive proof they share the same eigenbasis Let us show by recursion that for all t, $\mathbf{\Sigma}_t$ and $\mathbf{K}$ have same eigenbasis. We write the eigen-decomposition $\mathbf{K} = \mathbf{U}\hat{\mathbf{\Lambda}}\mathbf{U}^\mathsf{T}$ where $\mathbf{U} = [\mathbf{u}_1 \ldots \mathbf{u}_N]$ is the matrix of normalised eigenvectors ordered by decreasing eigenvalues. We only need to show that for all i, $\mathbf{u}_i$ is an eigenvector of $\mathbf{\Sigma}_t$.

- The proposition is trivial for $t = 0$ since $\mathbf{\Sigma}_0 = \mathbf{K}$.

- Note that, if the proposition is true for a given $t-1$, $\mathbf{v}_t$ being the eigenvector of $\mathbf{\Sigma}_{t-1}$ with highest eigenvalue, it is also an eigenvector of $\mathbf{K}$.

- Let us assume that the proposition is true for 1 to $t-1$, and let us show that it is true for t.

  1. From the previous point, we know that $\mathbf{v}_1, \dots, \mathbf{v}_t$ are all eigenvectors of $\mathbf{K}$, thus $\mathbf{V}_t$ is composed of columns of $\mathbf{U}$, say columns $\hat{i_1} \dots \hat{i_t}$:
  2. $(\mathbf{V}_t\mathbf{V}_t^\mathsf{T})\mathbf{u}_i = \begin{pmatrix} \mathbf{u}_{\hat{i_1}} & \dots & \mathbf{u}_{\hat{i_t}} \end{pmatrix} \begin{pmatrix} \mathbf{u}_{\hat{i_1}}^\mathsf{T} \\ \dots \\ \mathbf{u}_{\hat{i_t}}^\mathsf{T} \end{pmatrix} \mathbf{u}_i = m_{i,t}\mathbf{I}_t\mathbf{u}_i$
  where $m_{i,t}$ is the number of occurrences of $\mathbf{u}_i$ in $\mathbf{V}_t$, i.e. the number of times the i^{th} arm has been selected up to time t
  3. Multiplying Equation (3.15) by $\mathbf{u}_i$ on the right-hand side, we find that each $\mathbf{u}_i$ is an eigenvector of $\mathbf{\Sigma}_t$

3.2.3.2 Decay of the eigenvalues of the covariance matrix

For all $\mathbf{u}_i$:

$$\mathbf{\Sigma}_t^{-1}\mathbf{u}_i = \mathbf{K}^{-1}\mathbf{u}_i + s_{\text{noise}}^{-2}\mathbf{V}_t\mathbf{V}_t^\mathsf{T}\mathbf{u}_i \tag{3.16}$$

$$= \hat{\lambda}_i^{-1}\mathbf{u}_i + s_{\text{noise}}^{-2} m_{i,t}\mathbf{u}_i \tag{3.17}$$

$$\mathbf{\Sigma}_t\mathbf{u}_i = \frac{\hat{\lambda}_i}{1 + s_{\text{noise}}^{-2} m_{i,t}\hat{\lambda}_i}\mathbf{u}_i \tag{3.18}$$

Interpretation At the first time step, $\mathbf{u}_1$ is selected. At the following time step, either the eigenvalue corresponding to $\mathbf{u}_1$ (which is $\frac{\hat{\lambda}_1}{1+s_{\text{noise}}^{-2}\hat{\lambda}_1}$) is still bigger than all the $\hat{\lambda}_{i\geq 2}$, and $\mathbf{u}_1$ is selected again, or it is not the case and the biggest eigenvalue is thus $\hat{\lambda}_2$ which makes us select $\mathbf{u}_2$. Then, either the eigenvalue corresponding to $\mathbf{u}_1$ shrinks again (if chosen previously) which might thus create an opportunity to select $\mathbf{u}_2$, or the eigenvalue corresponding to $\mathbf{u}_2$ shrinks (if chosen previously) which might create an opportunity to select $\mathbf{u}_3$ or $\mathbf{u}_1$ again; and so on. Repeating this argument shows that $\mathbf{v}_t = \mathbf{u}_{\iota(t)}$ with $\iota(t) \leq t$.

3.2.3.3 Impact on the information gain

Successive covariance matrices of $\mathbb{P}(\mathbf{f}|\mathbf{y}_t)$ share their eigenbasis with $\mathbf{K}$. Let us denote by $\hat{\hat{\lambda}}_{i,t}$ the eigenvalue of $\mathbf{\Sigma}_t$ corresponding to eigenvector $\mathbf{u}_i$.

$$\sigma_{t-1}^2(\mathbf{x}_t) = \max_{||\mathbf{v}||=1} \mathbf{v}^\mathsf{T}\mathbf{\Sigma}_{t-1}\mathbf{v} = \mathbf{u}_{i(t)}^\mathsf{T}\mathbf{\Sigma}_{t-1}\mathbf{u}_{i(t)} = \hat{\hat{\lambda}}_{i(t),t} = \frac{\hat{\lambda}_{i(t)}}{1 + s_{\text{noise}}^{-2} m_{i(t),t}\hat{\lambda}_{i(t)}}$$

We plug this in Equation (3.12), which gives:

$$G_T^g = \frac{1}{2}\sum_{t=1}^{T} \log(1 + s_{\text{noise}}^{-2}\hat{\hat{\lambda}}_{i(t),t})$$

Let us denote by $\mathcal{T}_i \subset [1, T]$ the set of t values such that $i(t) = i$, and by m_i the number of elements in $\mathcal{T}_i$ (i.e. the number of times i has been selected up to time T). Because we have done T iterations, the greedy algorithm will have picked vectors among the first T eigenvectors of $\mathbf{K}$, hence $i \leq T$, which, combined with $i \leq N$ (total number of eigenvectors), gives $i \leq \min(T, N)$. We denote by S_i the sum of $\log(1 + s_{\text{noise}}^{-2}\hat{\hat{\lambda}}_{i(t),t})$ for $t \in \mathcal{T}_i$.

$$\begin{aligned} S_i = \log(1 + s_{\text{noise}}^{-2}\hat{\lambda}_i) \quad &+ \quad \log\left(1 + \frac{s_{\text{noise}}^{-2}\hat{\lambda}_i}{1 + s_{\text{noise}}^{-2}\hat{\lambda}_i}\right) \\ &+ \quad \log\left(1 + \frac{s_{\text{noise}}^{-2}\hat{\lambda}_i}{1 + s_{\text{noise}}^{-2}2\hat{\lambda}_i}\right) \\ &+ \quad \ldots + \log\left(1 + \frac{s_{\text{noise}}^{-2}\hat{\lambda}_i}{1 + s_{\text{noise}}^{-2}m_{i,t}\hat{\lambda}_i}\right) \end{aligned}$$

We transform the sum of logs into a log of a product and expand the product, which gives $S_i = \log(1 + s_{\text{noise}}^{-2}m_i\hat{\lambda}_i)$. Summing the S_i gives:

$$G_T^g = \frac{1}{2}\sum_{i=1}^{\min(T,N)} \log(1 + s_{\text{noise}}^{-2}m_i\hat{\lambda}_i) \tag{3.19}$$

Since the m_i values are not known, we maximise over all positive m_i values whose sum equals T, which gives Proposition 2.

We see that comparing to the greedy algorithm is crucial to bound the information gain in terms of the eigenvalues of $\mathbf{K}$ – instead of the eigenvalues of $\mathbf{K}_t$.

3.2.4 Other notable results

3.2.4.1 Special case: the linear kernel

The linear kernel in $\mathbb{R}^n$ is defined by $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$, so that $\mathbf{K}_T = \mathbf{X}_T^\mathsf{T}\mathbf{X}_T$. Let us denote by $\mathbf{\Lambda}_T$ the diagonal matrix of eigenvalues $\lambda_{T,1} \geq \ldots \geq \lambda_{T,n}$ of $\mathbf{X}_T\mathbf{X}_T^\mathsf{T}$. The information gained from a training set $\mathcal{D}_T$ can be expressed in terms of $\mathbf{K}_T$:

$$\begin{aligned}
G(\mathbf{x}_1, \ldots, \mathbf{x}_T) &= H(\mathbf{f}) - H(\mathbf{f}|\mathbf{y}_T) \\
&= H(\mathbf{f}_T) - H(\mathbf{f}_T|\mathbf{y}_T) \\
&= H(\mathbf{y}_T) - H(\mathbf{y}_T|\mathbf{f}_T) \\
&= H(\mathcal{N}(\mathbf{0}, \mathbf{K}_T + s_{\text{noise}}^2\mathbf{I}_T)) - H(\mathcal{N}(\mathbf{f}_T, s_{\text{noise}}^2\mathbf{I}_T)) \\
&= 1/2\log(\left|\mathbf{I}_T + s_{\text{noise}}^{-2}\mathbf{X}_T^\mathsf{T}\mathbf{X}_T\right|) \\
&= 1/2\log(\left|\mathbf{I}_n + s_{\text{noise}}^{-2}\mathbf{X}_T\mathbf{X}_T^\mathsf{T}\right|) \text{ by Sylvester's determinant theorem} \\
&\leq 1/2\log(\left|\mathbf{I}_n + s_{\text{noise}}^{-2}\mathbf{\Lambda}_T\right|) \text{ by Hadamard's inequality} \\
&\leq \sum_{i=1}^{n} 1/2\log(1 + s_{\text{noise}}^{-2}\lambda_{T,i}) \\
&\leq (n/2)\log(1 + s_{\text{noise}}^{-2}\lambda_{T,1})
\end{aligned}$$

The largest eigenvalue of $\mathbf{X}_T\mathbf{X}_T^\mathsf{T}$ is $O(T)$, therefore the information gain scales in $O(n\log(T))$ and the regret in $\tilde{O}(\sqrt{n\log(N)T})$.

3.2.4.2 Infinite number of arms

Proposition 3. *We can bound with probability $1-\delta$ the regret of the GPB algorithm run on a compact and convex subset of $\mathbb{R}^n$ with a Squared Exponential kernel and $\beta_t = 2\log\left(\frac{t^{\frac{n}{2}+2}\pi^2}{3\delta}\right)$ as follows:*

$$R_T = \tilde{O}(\sqrt{nTG_T^u})$$

See Srinivas et al. (2010) for a proof and a characterisation of the kernels for which this result can be extended.

The analysis requires discretising the input space $\mathcal{X}$, and to use the regularity of the covariance function in order to have all f values within their confidence intervals with high probability. The discretisation $\mathcal{X}_T$ after T iterations is of cardinality $O(T)$, and the information gain is bounded by an expression of the eigenvalues of $\mathbf{K}_T$. The expected sum of these can be linked to the sum of eigenvalues of the kernel operator spectrum with respect to the uniform distribution over $\mathcal{X}$, for which an expression is known for common kernels such as the Gaussian and Matérn kernels.

3.2.4.3 Reward functions in a Reproducing Kernel Hilbert Space

As in the frequentist setting, we compare the performance of our algorithm on the specific problem at hand (f is fixed) to the performance of an optimal allocation that would know f in advance, without any assumption on the problem other than the reward distributions for any given arm being bounded. In the present setting, f is arbitrary (not drawn from any prior) and a regret bound is given in terms of its smoothness (measured by its RKHS norm). The GP prior is just an artefact of the algorithm that is only used to define an arm selection criterion. In the Proposition below, we give an upper bound on the algorithm's regret, which is now a fixed quantity (unlike in Proposition 1).

Proposition 4. *We restate Theorem 3 of Srinivas et al. (2010). If f is a fixed (but unknown) function that lies in the RKHS induced by κ on the (potentially infinite) input space $\mathcal{X}$, if it has finite norm $||f||_\kappa \leq c$ and if $(\epsilon_\tau)_\tau$ is a martingale difference sequence bounded by s_{noise} almost surely, then when running GPB with covariance function κ and with a certain expression for β_t based on c, we have:*

$$R_T \leq \tilde{O}(\sqrt{T}(||f||_\kappa \sqrt{G_T^*} + G_T^*)) \tag{3.20}$$

See Srinivas et al. (2010) for the exact expression for β_t and for a proof of this result. The authors stress that neither this Proposition nor Proposition 3 encompasses the other. The former holds uniformly over all functions of finite norm under the chosen RKHS, whereas the latter is a probabilistic statement. Moreover, in the GP setting, $||f||_\kappa$ is almost surely infinite.

If $\mathcal{X}$ is finite, for any set of function values $\mathbf{f} = (f(\mathbf{a}_1) \dots f(\mathbf{a}_N))^T$ we have $||f||_\kappa^2 = \mathbf{f}^T\mathbf{K}^{-1}\mathbf{f}$, so the norm of f in the RKHS is always finite (see Section 2.2.2.3). If we take a kernel for which the eigenvalues of $\mathbf{K}$ decay more rapidly , we can expect to get a smaller G_T^* bound but a larger $||f||_\kappa$ value. Note that this regret bound involves a G_T^* term – instead of $\sqrt{G_T^*}$ – and G_T^* may scale linearly with N. If N depends on T (for instance, $N = T$ in Section 4.4.5), the regret upper bound can be worse than linear.

4 Gaussian Process Tree Search

In our background review, we saw how using UCB1 bandit instances at each node of a tree could help tree search (the "many-bandits" approach). But with the more recent many-armed bandit algorithms that model dependencies between arms and that can be applied to optimisation problems, we can consider a single bandit instance applied to the search for an optimum in a tree-structured space. Each element of the search space on which the target f is defined is a tree leaf (or, equivalently, a tree path) and the cardinality N of the space is bounded by the maximum branching factor to the power of the maximum depth.

In this chapter, we introduce and study the *Gaussian Process Tree Search algorithm (GPTS)*, which consists of using GPB for searching tree-structured spaces. We start by explaining why GP models make sense for trees and we introduce some kernels of interest that are based on the number of nodes in common between two paths. The application of GPB to tree search is not as straightforward as one could imagine, because the algorithm requires searching for the element with the highest upper confidence bound in a large, discrete space, in which an exhaustive search will not be feasible. Also, the reward estimates and uncertainty measures of all of the N arms need to be updated after each new observation. These operations can be implemented efficiently by exploiting the tree structure and the fact that leaves in the same unexplored subtree share their upper confidence values. As a consequence, we only need to consider, at each time t, a set of arms which cardinality scales linearly with t. The computational complexity that follows from this scales in $D\ T^3$ (or $D\ T\ S_T^2 + D\ T^2\ S_T$ for GPTS-RED).

As we saw in the previous chapter, the information gain of the algorithm is bounded by a constant that depends on N, which can be extremely large here ($N \gg T$ for large branching factors). We improve this constant by providing a bound on the eigenvalues that exploits the properties of the kernel, and which is expressed in terms of the kernel parameters. We give a regret bound that scales in the square root of NT (up to logarithmic factors), and one that scales in $N^{\frac{1}{4}}T^{\frac{3}{4}}$. In both cases, and for a Gaussian tree paths kernel, the constant improves for higher values of the kernel width (resulting in smoother functions).

4.1 Introduction: GP models for Tree Search

Here, we consider the general problem of tree search, for which nodes may be unlabeled. Our intuition on the function f defined on tree leaves, or, equivalently, on tree paths, is that the more nodes in common between two paths, the closer their f values are likely to be.

We saw that BAST and UCT estimate the "quality" of a node with an average μ of the rewards obtained with all paths that were tried and that included this node. Thus, the algorithm expects the value of the reward that will be obtained in the end, if it chooses to go through this node, to be more or less close to μ: all paths that go through this node will all have rewards close to each other. For the MDP planning problem, Bubeck and Munos (2010) say that "the rewards obtained along any sequence provides information, not only about that specific sequence, but also about any other sequence sharing the same initial actions". This is a property that translates to other tree search problems and that many-bandits algorithms do not fully exploit: at each iteration, they only update the statistics at the nodes of the path that was just "played". It also relates to a correlation assumption that can be formalised with a GP prior, with a covariance function based on the number of nodes in common between two inputs. Owing to GPs, we can share information gained for playing a path with any other path that has nodes in common. The covariance function is between tree paths, i.e. sequences of nodes $\mathbf{x} = x_1, \ldots, x_D$ where x_1 is always a child of the root node and has depth 1.

Our Gaussian Processes Tree Search approach consists in applying GPB to the search for optimal values of f by considering tree paths as arms of a bandit problem. We refer to this as the *single-bandit* approach to tree search: unlike the many-bandits approach, we use only one instance of a bandit algorithm that is able to model dependencies between arms.[1] Perhaps the most natural features for tree paths consist of node indicators for all nodes of the tree. Another possible single-bandit algorithm would be based on LinRel instead of GPB. However, given the similarities between the two algorithms, we prefer to use GPB since the definition of its exploration term is more principled and performance guarantees have been given even when the Gaussian

[1] Bubeck and Munos (2010, sec. 4) also consider a similar approach when comparing UCB-AIR and HOO to OLOP.

assumptions do not hold.

In the following we consider trees with maximum branching factor $B \geq 2$ and depth $D \geq 1$. The number of arms is $N = B^D$.

4.1.1 Kernels based on the number of nodes in common

We consider a feature space indexed by all the nodes x of the tree, with the feature vector of any given path $\mathbf{x} = (x_1, ...x_D)$ defined by:

$$\phi_x(\mathbf{x}) = \begin{cases} 1; & \text{if } \exists 1 \leq i \leq D, x = x_i \\ 0; & \text{otherwise.} \end{cases}$$

The dimension of this space is equal to the number of nodes in the tree other than the root, $\bar{N}$. The linear kernel in this space simply counts the number of nodes in common between two paths. Note that paths that start from different children of the root node will have a linear kernel product of 0, meaning that their mean-reward values are considered to be independent. As a consequence, there will be many zero entries in $\mathbf{K}$ with the linear kernel.

Similarly to BAST, we wish to model different levels of smoothness of f. For this, we can extend the notion of characteristic length-scale to functions on tree paths by considering a Gaussian covariance function in their feature space. The squared Euclidian distance is twice the number of nodes d where they differ: path 1 contains nodes indexed by $i_1, \ldots, i_d$ that path 2 does not contain, and path 2 contains nodes indexed by $j_1, \ldots, j_d$ that path 1 does not contain, so the $i_1, \ldots, i_d$ and $j_1, \ldots, j_d$ components of the feature vectors differ. The components of the difference of the feature vectors will be 0 except at the d i-indices and at the d j-indices where they will be 1 or -1. Summing the squares gives $2d$. Consequently, the Gaussian kernel is an exponential on minus the number of nodes where two paths differ (from 0 to D): $\exp(-d/s^2)$ where s is the characteristic length-scale.

More generally, any kernel function that is based on the number of nodes in common between paths is characterised by a set of $\chi_0 > ... > \chi_{D-1}$ values in $[0, 1]$, where χ_d represents the value of the kernel product between two paths that have d nodes not in common, and $\chi_D = 0$. We can give an explicit feature mapping for such a kernel, in the same feature space as before:

$$\phi_x(\mathbf{x}) = \begin{cases} \sqrt{\chi_{D-i} - \chi_{D-i+1}}; & \text{if } \exists 1 \leq i \leq D, x = x_i \\ 0; & \text{otherwise.} \end{cases}$$

Indeed, consider two paths that differ on d nodes: the first $1 + D - d$ nodes only will be in common, hence the inner product of their feature vectors will be $\chi_D + \sum_{i=1}^{D-d}(\sqrt{\chi_{D-i} - \chi_{D-i+1}})^2 = \chi_d$, which is equal to the kernel product between the two paths, by definition of χ_d. Note that the kernel is normalised by imposing $\chi_0 = 1$, which will be required in Section 4.3.2.1.

4.1.2 Examples: discounted MDPs and game trees

4.1.2.1 Discounted MDPs with deterministic dynamics and normalised Gaussian rewards

We model our belief on what we expect the intermediate reward functions to be by considering, at each node x_τ in the sequence of actions being explored, a set of random variables $F_1^{(x_\tau)}, \ldots, F_B^{(x_\tau)}$ such that the intermediate reward function values for all possible actions from node x_τ is a realisation of this set of random variables. We assume that each of these random variables follows a normalised Gaussian distribution, and that they are all independent. The discounted sum of intermediate reward values is a sum of Gaussians, hence it is a Gaussian and the GP model makes sense. We now determine the tree paths covariance function that follows from our assumptions. A path is a list of nodes $x_0, x_1, \ldots, x_D$, where x_0 is the root, corresponding to a list of indices $i_1, \ldots, i_D$ of actions taken in the environment. Our belief on the function value for this path is represented by $\gamma^0 F_{i_1}^{(x_0)} + \ldots + \gamma^{D-1} F_{i_D}^{(x_{D-1})}$. If two paths $\mathbf{x}$ and $\mathbf{x}'$ have $h = \phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{x}')$ action indices in common, they can be represented by $i_1, \ldots, i_h, i_{h+1}, \ldots, i_D$ and $i_1, \ldots, i_h, i'_{h+1}, \ldots, i'_D$. The kernel product between these two paths is given by:

$$\begin{aligned} \kappa(\mathbf{x}, \mathbf{x}') &= \mathrm{cov}(\gamma^0 F_{i_1}^{(x_0)} + \ldots + \gamma^{h-1} F_{i_h}^{(x_{h-1})} + \gamma^h F_{i_{h+1}}^{(x_h)} + \ldots + \gamma^{D-1} F_{i_D}^{(x_{D-1})}, \\ &\quad \gamma^0 F_{i_1}^{(x_0)} + \ldots + \gamma^{h-1} F_{i_h}^{(x_{h-1})} + \gamma^h F_{i'_{h+1}}^{(n'_h)} + \ldots + \gamma^{D-1} F_{i'_D}^{(n'_{D-1})}) \\ &= \sum_{\tau=0}^{h-1} \gamma^\tau \gamma^\tau \, \mathrm{cov}(F_{i_{\tau+1}}^{(x_\tau)}, F_{i_{\tau+1}}^{(x_\tau)}) \\ &= \frac{1 - \gamma^{2h}}{1 - \gamma^2} \end{aligned}$$

where we used the bi-linearity of the covariance, the independence of the random variables, and the fact that their variances are always 1 since they follow normalised Gaussian distributions. This characterises our belief on the discounted sum of rewards f. Note that the kernel is not normalised: $\kappa(\mathbf{x}, \mathbf{x}) = \frac{1-\gamma^{2D}}{1-\gamma^2}$ which grows with D. This reflects the fact that the signal variance is higher for deeper trees.

We refer to this kernel as the *discounted kernel* ($\gamma < 1$). If $\gamma = 1$ (un-discounted MDP), we have the linear kernel. Note that f is made smoother by decreasing the value of the discount factor γ.

4.1.2.2 Game trees

Hennig et al. (2010) provide a probabilistic, generative model for the value of game tree nodes under the random rollout policy. We show that their model's assumptions imply a GP over the leaves and we give the expression of the covariance function.

The assumptions are that, for each node in the tree, there exists a latent variable called its *score* which represents the average of all possible outcomes ($+1$ for a win, -1 for a loss) from that node on. The prior for the score of the root node is a Gaussian with zero mean and standard deviation equal to 1 (this can actually be generalised to any values). The score of a node is generated from a Gaussian with mean equal to the parent's score, and with standard deviation equal to 1. As a consequence, the scores of sibling nodes are independent given their parent's score.

Ultimately, we are interested in learning the scores of leaves at depth D, and the assumptions on the scores at interior nodes are used to model the relationships between the leaves' scores. When arriving at a leaf, we get a reward by randomly finishing the game (as in classical Monte-Carlo game tree search, see Gelly and Wang, 2006). We can thus consider that this reward is a Bernoulli sample with mean equal to the score of the leaf node. Thus, we observe the true scores plus an arbitrary martingale difference sequence (the noise) with variance bounded by the maximum possible Bernoulli variance (given by $m(1-m)$ where m is the mean) which is $1/4$. When scaling rewards to -1 and 1 instead of 0 and 1, this variance is multiplied by 4. The variance being bounded, Inequality (3.20) could be applied to bound the regret of GPTS.

Let us now determine the covariance function between paths. We first show by induction that, for a node at depth d with score S, $\text{var}(S) = d$. For this, we write P for the score of the parent. We have that $\text{var}(P) = d - 1$. By the law of total covariance (Equation A.6), $\text{var}(S) = \mathbb{E}(\text{var}(S|P)) + \text{var}(\mathbb{E}(S|P)) = 1 + \text{var}(P) = 1 + (d - 1) = d$, which ends the proof. We can also apply the law of total covariance between two paths X and X' that have $D - d$ nodes in common: $\text{cov}(X, X') = \mathbb{E}(\text{cov}(X, X'|S)) + \text{var}(S) = 0 + \text{var}(S) = D - d$ where S is the score of the last node they have in common, which is at depth $D - d$. This is the linear covariance function.

Although their model's assumptions seem restrictive, Hennig et al. have shown with Q-Q plots that the game of Go is close to their model. They have also reported a "minor decay in performance" when true scores are drawn uniformly at random, rather than from Gaussians as the model assumes.

4.2 An efficient implementation of the GPTS algorithm

The difficulty in implementing the GPB algorithm is to find the maximum of the upper confidence function, when the computational cost of an exhaustive search is prohibitive due to a large number of arms – as for most tree search applications. At time t we look for the path $\mathbf{x}$ which maximises $f_t(\mathbf{x})$. Because $\kappa(\mathbf{x}, \mathbf{x})$ has the same value for all $\mathbf{x}$ (here, χ_0), we can benefit from the tree structure in order to perform this search in $O(t)$ only: f_t is a function of the vector $\mathbf{k}$ of kernel products with the arms in training, all the paths that go through the same unexplored subtree share the same $\mathbf{k}$, and there are $O(t)$ maximum unexplored subtrees. We first define some terminology and then prove this result.

Terminology A node x is said to be explored if there exists a path $\mathbf{x}_{i,i \leq t}$ in the training data such that $\mathbf{x}_i$ contains x, and it is said to be unexplored otherwise. A subtree is defined here to be a set of nodes that have a common ancestor called the *root* of the subtree, excluding this root node. A subtree is unexplored if no path in the training data goes through this subtree. A maximum unexplored subtree is a subtree such that its root belongs to an $\mathbf{x}_i$ in the training data.

Proof and procedure When $\kappa(\mathbf{x}, \mathbf{x})$ has same value for all $\mathbf{x}$, $f_t(\mathbf{x})$ can be expressed as a function of $\mathbf{k} = \mathbf{k}_t(\mathbf{x})$ (see Equations 2.12 and 2.13 on page 42) and we argue that all paths that go through a given unexplored subtree $\mathcal{S}$ will have same $\mathbf{k}$ value, hence same f_t value. Let $\mathbf{x} = (x_1, \ldots, x_l, \ldots, x_{D+1})$ be such a path, where $l \geq 1$ is defined such that node x_l has been explored but not x_j for $j > l$. All $\mathbf{x}$'s that go through $\mathcal{S}$ have the same first nodes $x_1, \ldots, x_l$, and the other nodes do not matter in kernel computations since they have not been visited.

Consequently we just need to evaluate $f_t(\mathbf{x})$ on one randomly chosen path that goes through the unexplored subtree $\mathcal{S}$, all other such paths having the same value for $f_t(\mathbf{x})$. We represent maximum unexplored subtrees by *dummy nodes* and, as we do for leaf nodes, we compute and store f_t values for dummy nodes. The number of dummy nodes in memory is 1 per visited node with unexplored siblings: it is the subtree containing the unexplored siblings and their descendants. There are at most $D + 1$ such nodes per path in the training data, and there are t paths in the training data, hence the number of dummy nodes is less than or equal to $(D + 1)t$.

This means that the number of nodes (leaf or dummy) to examine in order to find the maximiser of f_t is in $O(t)$. We denote this set of nodes $\mathcal{X}_t$. We do not need to represent all arms in memory, but only those in $\mathcal{X}_t$. After some time, all $\bar{N}$ nodes of the tree will have been explored and $\mathcal{X}_t$ will be equal to $\mathcal{X}$. Pseudo-code is given in Algorithm 1 on the next page. Note that with this algorithm, we might choose the same leaf node more than once unless $s_{\text{noise}} = 0$.

Computational analysis As we said previously, we only need to consider $O(D\ t)$ dummy nodes and t leaf nodes when maximising and updating f_t, instead of N leaf nodes. Replacing N by $D\ t$ in the GPB-online1 computational complexity formula, we get a cost in $O(S_t^2 + D\ t\ S_t)$ for the updates and potential downdates.

Here, GPB works with a finite but growing set of arms, so we need to consider the costs of adding an arm to $\mathcal{X}_t$. After choosing $\mathbf{x}_{t+1}$, we add up to D arms to $\mathcal{X}$, corresponding to dummy nodes. With GPB-online1 and for a new arm $\mathbf{x}$, we can compute $\mu_{t+1}(\mathbf{x})$ and $\sigma_{t+1}(\mathbf{x})$ directly, based on the covariance matrix inverse, which costs $O(S_t^2)$. As a consequence, adding dummy nodes at each iteration costs $O(D\ S_t^2)$.

This is to be added to the previous cost and to the argmax cost ($O(D\ t)$ instead of $O(N)$), which gives:

$$C_T = \sum_{t=1}^{T} O(D\ S_t^2 + D\ t\ S_t) \leq O(D\ T\ S_T^2 + D\ T^2\ S_T)$$

Algorithm 1 GPB *for Tree Search*

```
% Initialisation
$t = 0$ % number of iterations
create root and dummy child $d_0$
$\mathcal{X}_t = \{\mathbf{d}_0\}$ % set of arms that can be selected
% Iterations
repeat
  % Choose a path
  if $t == 0$ then
    $\mathbf{x} = \mathbf{d}_0$
  else
    choose $\mathbf{x}$ in $\mathcal{X}_t$ that has highest upper confidence value
  end if
  if $\mathbf{x}$ is a dummy node then
    % Random walk
    create sibling $\mathbf{x}'$ of $\mathbf{x}$
    if all siblings of $\mathbf{x}$ have been created then
      delete $\mathbf{x}$ from the tree and remove from $\mathcal{X}_t$
    end if
    $\mathbf{x} = \mathbf{x}'$
    while depth of $\mathbf{x}$ is strictly smaller than $D$ do
      create $\mathbf{x}'$ child of $\mathbf{x}$ and $\mathbf{d}$ dummy child of $x$
      add $\mathbf{d}$ to $\mathcal{X}_t$
      $\mathbf{x} = \mathbf{x}'$
    end while
    add $\mathbf{x}$ to $\mathcal{X}_t$ % chosen leaf
  end if
  % Get reward and add to training set
  compute the vector of kernel products $\mathbf{k}$ between $\mathbf{x}$ and the elements of $\mathbf{X}_t$
  append $\mathbf{x}$ to $\mathbf{X}_t$
  append reward($\mathbf{x}$) to $\mathbf{y}_t$
  $\mathbf{K}_t = \begin{pmatrix} \mathbf{K}_t & \mathbf{k} \\ \mathbf{k}^\mathsf{T} & \kappa(\mathbf{x}, \mathbf{x}) \end{pmatrix}$
  $\mathbf{C}_t^{-1} = (\mathbf{K}_t + s_{\text{noise}}^2 \mathbf{I}_{t+1})^{-1}$
  for all $\mathbf{x} \in \mathbf{X}_t$ do
    compute the vector of kernel products $\mathbf{k}$ between $\mathbf{x}$ and the elements of $\mathbf{X}_t$
    compute $f_t(\mathbf{x})$ (based on $\mathbf{k}, \mathbf{C}_t, \mathbf{y}_t$, see Equations (2.12) and (2.13))
  end for
  $t = t + 1$
until stopping criterion is met
% Define output
look for $\mathbf{x}$ in $\mathbf{X}_t$ that had highest reward value and output the corresponding path
```

Note that the cost of all online variants of GPB applied to tree search would be the same because the number of arms to be considered scales in $O(T)$.

On the choice of kernels The efficient implementation of GPTS through the use of dummy nodes was made possible by the fact that there is only a small set of possible kernel product values and we can easily identify paths that have same kernel products. In problems where we may have access to feature descriptions of nodes, as in Go where nodes are labelled by Go boards, we may be tempted to exploit the richer representations of tree paths in our kernel, thus modeling dependencies between paths more precisely. However, doing so would likely result in an intractable algorithm where we would have difficulties updating the upper confidence values for all arms – unless $N = B^D$ is small.

4.3 Theoretical analysis

4.3.1 Overview

We have seen in Section 4.1.1 that all kernels considered here are equivalent to a linear kernel in a certain feature space of dimension $\bar{N}$. Therefore, we can apply the linear kernel GPB regret bound given in Section 3.2.4.1. $n = \bar{N} = O(N)$ here, and thus we know that the regret scales in $\tilde{O}(\sqrt{NT})$ with high probability. However, we aim to provide regret bounds with better constants that are stated in terms of the kernel parameters – we expect smaller constants for smoother kernels. From Propositions 1 and 2, the regret is bounded as follows, with high probability:

$$R_T \leq \max_{(m_i)_i \geq 0 \text{ s.t. } \sum_i m_i = T} \sqrt{\frac{4\sigma_0^2 \beta_{T-1} T}{(1-e^{-1}) \log(1 + s_{\text{noise}}^{-2} \sigma_0^2)} \sum_{i=1}^{\min(T,N)} \log(1 + s_{\text{noise}}^{-2} m_i \hat{\lambda}_i)} \tag{4.1}$$

We therefore have to upper bound the $\hat{\lambda}_i$. For our analysis, we "expand" the tree by creating extra nodes so that all branches have the same branching factor B. This construction is purely theoretical as the algorithm does not need a representation of the whole tree, nor the expanded tree, in order to run. We first derive analytical expressions for the eigenvalues of $\mathbf{K}$ in terms of B, D, and the $\chi_{0 \leq d \leq D}$ values.

Proposition 5. *We write $\bar{\lambda}_1 < \ldots < \bar{\lambda}_{D+1}$ for the distinct eigenvalues of $\mathbf{K}$, and ν_i*

for their multiplicities.

$$\begin{aligned}\forall i \in [1, D], \bar{\lambda}_i^{(D)} &= \sum_{j=0}^{i-1} B^j(\chi_j - \chi_{j+1}) \text{ and } \nu_i^{(D)} = (B-1)B^{D-i} \\ \bar{\lambda}_{D+1}^{(D)} &= \sum_{j=0}^{D-1} B^j(\chi_j - \chi_{j+1}) + B^D \chi_D \text{ and } \nu_{D+1}^{(D)} = 1\end{aligned}$$

A proof is given in Section 4.3.2.2.

The $\hat{\lambda}_t$ values are the $\bar{\lambda}_i$ values repeated a number of times equal to their multiplicities, and in reverse order. We thus have $\hat{\lambda}_t = \bar{\lambda}_{D-i}$ with i such that $B^i < t \leq B^{i+1}$. For $1 < t \leq N$, $\log(t) = i\log(B) + r$ with $0 < r \leq \log(B)$ hence $B^i < t \leq B^{i+1}$. $i = \frac{\log(t)-r}{\log(B)}$ from which we have:

$$\forall t \in [1, N], \exists i \in [-1, D-1], \hat{\lambda}_t = \bar{\lambda}_{D-i} \text{ with } \log_B(t) - 1 \leq i < \log_B(t) \tag{4.2}$$

We can use this inequality in the previous proposition in order to study the decay rate of $\hat{\lambda}_t$ for chosen tree paths kernels. Note that the expression for $\bar{\lambda}_{D-i}$ always involves a B^D factor. We will therefore derive upper-bounds of the form $\hat{\lambda}_t \leq N\hat{l}(t)$.

Proposition 6. *We have, for all $t > 1$:*

$$\hat{\lambda}_t \leq N\hat{l}(t)$$

where

$$\begin{aligned}\hat{l}(t) &= \frac{B}{(B-1)Dt} \text{ for the linear kernel} \\ &= \frac{B}{\gamma(B-\gamma^2)t^{1+2\log_B(1/\gamma)}} \text{ for the discounted kernel, with } 0 < \gamma < 1 \\ &= O\left(\frac{1}{s^2 t}\right) \text{ for the Gaussian kernel}\end{aligned}$$

See proofs in Sections 4.3.2.3, 4.3.2.4 and 4.3.2.5.

Using this result in Equation 4.1, we can derive the following proposition.

Proposition 7. *Under the assumption that f is drawn from a GP prior, and by application of Proposition 1, the regret of GPTS can be upper-bounded "with high probability" by an expression that scales in $\tilde{O}(\sqrt{NT})$. In the cases of the linear and of the Gaussian kernels, it can also be bounded by an expression that scales in $\tilde{O}\left(N^{\frac{1}{4}}T^{\frac{3}{4}}\right)$. The constant in the bound improves for larger widths of the Gaussian kernel. Proofs are given in Sections 4.3.3.1 and 4.3.3.2.*

The ratio between these two rates is $\tilde{O}\left(\left(\frac{N}{T}\right)^{\frac{1}{4}}\right)$, hence the bound corresponding to the second rate is more interesting when $N \gg T$.

4.3.2 Eigenvalues of the kernel matrix on tree paths

In this section we prove Proposition 6, which is a direct application of Proposition 5 with the χ_d values corresponding to the kernels that are considered.

4.3.2.1 Recursive block representation of the kernel matrix

We write $\mathbf{K}_{B,D}$ for the kernel matrix on all paths through an expanded tree with branching factor B and depth D. These two integers completely characterise the tree. We start by giving an expression of $\mathbf{K}_{B,D}$ in terms of $\mathbf{K}_{B,D-1}$, which will be used in order to prove Proposition 5.

We write $\mathbf{J}_i$ for the matrix of ones of dimension $i \times i$. $\mathbf{K}_{B,D}$ can be expressed in block matrix form with $\mathbf{K}_{B,D-1}$ and $\mathbf{J}_{B^{D-1}}$ blocks:

$$\mathbf{K}_{B,1} = (\chi_0 - \chi_1)\mathbf{I}_B + \chi_1 \mathbf{J}_B \tag{4.3}$$

and

$$\mathbf{K}_{B,D} = \begin{pmatrix} \mathbf{K}_{B,D-1} & \chi_D \mathbf{J}_{B^{D-1}} & \cdots & \chi_D \mathbf{J}_{B^{D-1}} \\ \chi_D \mathbf{J}_{B^{D-1}} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \chi_D \mathbf{J}_{B^{D-1}} \\ \chi_D \mathbf{J}_{B^{D-1}} & \cdots & \chi_D \mathbf{J}_{B^{D-1}} & \mathbf{K}_{B,D-1} \end{pmatrix}$$

To see this, one must think of the (B,D)-tree as a root pointing to B $(B,D-1)$-trees. On the 1st diagonal block of $\mathbf{K}_{B,D}$ is the kernel matrix for the paths that go through the first $(B,D-1)$-tree. Because the kernel function is normalised, this stays the same when we prepend the same nodes (here the new root) to all paths, so it is $\mathbf{K}_{B,D-1}$. Similarly, on the other diagonal blocks we have $\mathbf{K}_{B,D-1}$. In order to complete the block matrix representation of $\mathbf{K}_{B,D}$ we just need to know that any two paths that go through different $(B,D-1)$-trees only have the root in common, and we use the definition of χ_D.

Let us denote by $\tilde{\mathbf{I}}^{(n)}(\mathbf{M})$ and $\tilde{\mathbf{J}}^{(n)}(\mathbf{M})$ the matrices of n blocks by n blocks:

$$\tilde{\mathbf{I}}^{(n)}(\mathbf{M}) = \begin{pmatrix} \mathbf{M} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \mathbf{M} \end{pmatrix}$$

$$\tilde{\mathbf{J}}^{(n)}(\mathbf{M}) = \begin{pmatrix} \mathbf{M} & \dots & \mathbf{M} \\ \vdots & \ddots & \vdots \\ \mathbf{M} & \dots & \mathbf{M} \end{pmatrix}$$

We can then write:

$$\mathbf{K}_{B,D} = \chi_D \tilde{\mathbf{J}}^{(B)}(\mathbf{J}_{B^{D-1}}) - \chi_D \tilde{\mathbf{I}}^{(B)}(\mathbf{J}_{B^{D-1}}) + \tilde{\mathbf{I}}^{(B)}(\mathbf{K}_{B,D-1}) \tag{4.4}$$

4.3.2.2 Eigenvalues

We prove Proposition 5 by recursion on D. We write $\bar{\lambda}_i^{(D)}$ for the $D+1$ distinct eigenvalues of $\mathbf{K}_{B,D}$ and $\nu_i^{(D)}$ their multiplicities. For this, we show that $\mathbf{J}_{B^D}$ and $\mathbf{K}_{B,D}$ share same eigenbasis, and the eigenvector $\mathbf{K}_{B,D}$ with highest eigenvalue is the vector of ones $\mathbf{1}_{B^D}$, which is also the eigenvector of $\mathbf{J}_{B^D}$ with highest eigenvalue.

Preliminary result: eigenanalysis of the block-matrix of ones $\mathbf{J}_B$ has two eigenvalues: 0 with multiplicity $B-1$ and B with multiplicity 1. We denote by $\mathbf{j}_1, \dots, \mathbf{j}_B$ the eigenvectors of $\mathbf{J}_B$, in decreasing order of corresponding eigenvalue. $\mathbf{j}_1$ is the vector of ones. The coordinates of $\mathbf{j}_i$ are notated $j_{i,1} \dots, j_{i,B}$. For all i from 1 to B we define $\tilde{\mathbf{U}}_i^{(B)}(.)$ as a concatenation of B vectors:

$$\tilde{\mathbf{U}}_i^{(B)}(\mathbf{v}) = \begin{pmatrix} j_{i,1}\mathbf{v} \\ \vdots \\ j_{i,B}\mathbf{v} \end{pmatrix}$$

For all $i \geq 2$, $\sum_l j_{i,l} = 0$ by definition of $\mathbf{j}_i$. For all n-dimensional vector $\mathbf{v}$ and $n \times n$ matrix M:

$$\begin{aligned}
\tilde{\mathbf{J}}^{(B)}(\mathbf{M})\tilde{\mathbf{U}}_i^{(B)}(\mathbf{v}) &= \begin{pmatrix} (\sum_k \mathbf{M}_{1,k} j_{i,1} v_k) + \ldots + (\sum_k \mathbf{M}_{1,k} j_{i,B} v_k) \\ \vdots \\ (\sum_k \mathbf{M}_{n,k} j_{i,1} v_k) + \ldots + (\sum_k \mathbf{M}_{n,k} j_{i,B} v_k) \end{pmatrix} \\
&= \begin{pmatrix} (\sum_k \mathbf{M}_{1,k} v_k)(\sum_l j_{i,l}) \\ \vdots \\ (\sum_k \mathbf{M}_{n,k} v_k)(\sum_l j_{i,l}) \end{pmatrix} \\
&= \mathbf{0}
\end{aligned}$$

Hence $\tilde{\mathbf{U}}_i^{(B)}(\mathbf{v})$ is an eigenvector of $\tilde{\mathbf{J}}^{(B)}(\mathbf{M})$ with eigenvalue equal to 0.

Recursion We propose eigenvectors of $\mathbf{K}_{B,D}$, use Equation (4.4) and determine the value of each term of the sum multiplied by the proposed eigenvectors, in order to get an expression for the eigenvalues.

- For $D = 1$. From Equation (4.3), $\mathbf{j}_1, \ldots, \mathbf{j}_{B-1}$ are also eigenvectors of $\mathbf{K}_{B,1}$ with eigenvalue $\bar{\lambda}_1^{(1)} = \chi_0 - \chi_1$, hence $\bar{\lambda}_1^{(1)}$ has multiplicity $\nu_1^{(1)} = B - 1$ as expected. $\mathbf{j}_B$ is also an eigenvector of $\mathbf{K}_{B,1}$ with eigenvalue $\bar{\lambda}_2^{(1)} = B\chi_1 + \chi_0 - \chi_1$, and $\nu_2^{(1)} = 1$.

- Let us assume the result is true for a given depth $D - 1$.

  - The largest eigenvalue of $\mathbf{K}_{B,D-1}$ is

    $$\bar{\lambda}_D^{(D-1)} = B^{D-1}\chi_{D-1} + \sum_{j=0}^{D-2} B^j(\chi_j - \chi_{j+1})$$

    with multiplicity 1. Let us apply $\tilde{\mathbf{U}}_B^{(B)}$ to the corresponding eigenvector $\mathbf{1}_{B^{D-1}}$, and multiply it by the expression of $\mathbf{K}_{B,D}$ given in Equation (4.4).

    * $\tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}}) = \mathbf{1}_{B^D}$ and $\tilde{\mathbf{J}}^{(B)}(\mathbf{J}_{B^{D-1}})$ is a matrix of ones in B^D dimensions, hence:

      $$\tilde{\mathbf{J}}^{(B)}(\mathbf{J}_{B^{D-1}})\tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}}) = B^D \tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}})$$

- $\mathbf{1}_{B^{D-1}}$ is also the highest eigenvector of $\mathbf{J}_{B^{D-1}}$, with eigenvalue B^{D-1}, hence:

$$\tilde{\mathbf{I}}^{(B)}(\mathbf{J}_{B^{D-1}})\tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}}) = B^{D-1}\tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}})$$

- By definition of $\mathbf{1}_{B^{D-1}}$ and $\bar{\lambda}_D^{(D-1)}$:

$$\tilde{\mathbf{I}}^{(B)}(\mathbf{K}_{B,D-1})\tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}}) = \bar{\lambda}_D^{(D-1)}\tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}})$$

As a consequence, $\tilde{\mathbf{U}}_B^{(B)}(\mathbf{1}_{B^{D-1}}) = \mathbf{1}_{B^D}$ is the eigenvector of $\mathbf{K}_{B,D}$ with highest eigenvalue (this will be confirmed later), equal to $\bar{\lambda}_{D+1}^{(D)} = B^D\chi_D + \sum_{j=0}^{D-1} B^j(\chi_j - \chi_{j+1})$.

– Let us apply $\tilde{\mathbf{U}}_k^{(B)}$ to $\mathbf{1}_{B^{D-1}}$ for all k from 1 to $B-1$.

- Owing to the preliminary result, we have:

$$\tilde{\mathbf{J}}^{(B)}(\mathbf{J}_{B^{D-1}})\tilde{\mathbf{U}}_k^{(B)}(\mathbf{1}_{B^{D-1}}) = \mathbf{0}$$

- Since $\mathbf{1}_{B^{D-1}}$ is the eigenvector of $\mathbf{J}_{B^{D-1}}$ with eigenvalue B^{D-1}:

$$\tilde{\mathbf{I}}^{(B)}(\mathbf{J}_{B^{D-1}})\tilde{\mathbf{U}}_k^{(B)}(\mathbf{1}_{B^{D-1}}) = B^{D-1}\tilde{\mathbf{U}}_k^{(B)}(\mathbf{1}_{B^{D-1}})$$

- Since $\mathbf{1}_{B^{D-1}}$ is the eigenvector of $\mathbf{K}_{B,D-1}$ with highest eigenvalue:

$$\tilde{\mathbf{I}}^{(B)}(\mathbf{K}_{B,D-1})\tilde{\mathbf{U}}_k^{(B)}(\mathbf{1}_{B^{D-1}}) = \bar{\lambda}_D^{(D-1)}\tilde{\mathbf{U}}_k^{(B)}(\mathbf{1}_{B^{D-1}})$$

for the same reasons as previously.

As a consequence, $\mathbf{K}_{B,D}\tilde{\mathbf{U}}_k^{(B)}(\mathbf{1}_{B^{D-1}}) = (-\chi_D B^{D-1} + \bar{\lambda}_{D+1}^{(D)})\tilde{\mathbf{U}}_k^{(B)}(\mathbf{1}_{B^{D-1}})$ and we have found $B-1$ eigenvectors of $\mathbf{K}_{B,D}$ with eigenvalue equal to $\bar{\lambda}_D^{(D)} = \sum_{j=0}^{D-1} B^j(\chi_j - \chi_{j+1})$. These vectors are also eigenvectors of $\mathbf{J}_{B^D}$ with eigenvalue 0, which comes from the preliminary result and the fact that $\mathbf{J}_{B^D} = \tilde{\mathbf{J}}^{(B)}(\mathbf{J}_{B^{D-1}})$.

– For i from 1 to $D-1$, let us apply $\tilde{\mathbf{U}}_k^{(B)}$, for all k from 1 to B, to all $(B-1)B^{D-1-i}$ eigenvectors $\mathbf{v}$ of $\mathbf{K}_{B,D-1}$ with eigenvalue equal to $\bar{\lambda}_i^{(D-1)}$. By definition of $\mathbf{v}$:

$$\tilde{\mathbf{I}}^{(B)}(\mathbf{K}_{B,D-1})\tilde{\mathbf{U}}_k^{(B)}(\mathbf{v}) = \bar{\lambda}_i^{(D-1)}\tilde{\mathbf{U}}_k^{(B)}(\mathbf{v})$$

$\mathbf{v}$ being also an eigenvector of $\mathbf{J}_{B^{D-1}}$ with eigenvalue 0:

$$\begin{aligned}\tilde{\mathbf{J}}^{(B)}(\mathbf{J}_{B^{D-1}})\tilde{\mathbf{U}}_k^{(B)}(\mathbf{v}) &= \mathbf{0}\\ \tilde{\mathbf{I}}^{(B)}(\mathbf{J}_{B^{D-1}})\tilde{\mathbf{U}}_k^{(B)}(\mathbf{v}) &= \mathbf{0}\end{aligned}$$

As a consequence, eigenvalues stay unchanged but their multiplicities are all multiplied by B (because k goes from 1 to B and we have identified B times as many eigenvectors) which gives $\nu_i^{(D)} = (B-1)B^{D-i}$. Again, the preliminary result allows us to show that the $\tilde{\mathbf{U}}_k^{(B)}(\mathbf{v})$ are also eigenvectors of $\mathbf{J}_{B^D}$ with eigenvalue 0.

– The total number of multiplicities for all found eigenvalues is equal to $(\sum_i^{D-1}\nu_i^{(D)}) + B - 1 + 1 = B^D$ so we have identified all the eigenvectors.

4.3.2.3 Linear kernel

The linear kernel is an inner product in the feature space, which amounts to counting how many nodes in common two paths have. It takes values from 0 to D. The normalised linear kernel divides these values by D. If two paths of depth D differ on d nodes, they have $D-d$ nodes in common:

$$\chi_d = \frac{D-d}{D}$$

For all j, $\chi_j - \chi_{j+1} = 1/D$, hence $\bar{\lambda}_i = \frac{1}{D}\sum_{j=0}^{i-1} B^j = \frac{B^i-1}{(B-1)D}$ for $i < D+1$.

- For $t > 1$: we use Inequality (4.2) to get a lower and an upper bound on $\hat{\lambda}_t$.

$$\begin{aligned} & \bar{\lambda}_{D-i} && = \frac{NB^{-i}-1}{(B-1)D}\\ \frac{NB^{-\log_B(t)}-1}{(B-1)D} \leq{} & \hat{\lambda}_t && \leq \frac{NB^{1-\log_B(t)}-1}{(B-1)D}\\ \frac{N-t}{(B-1)Dt} \leq{} & \hat{\lambda}_t && \leq \frac{NB-t}{(B-1)Dt}\end{aligned}$$

And thus we can take:

$$\hat{l}(t) = \frac{B}{(B-1)Dt}$$

- For $t = 1$:

$$\begin{aligned}\hat{\lambda}_1 &= \bar{\lambda}_{D+1}\\ &= \bar{\lambda}_D + B^D \chi_D\\ &= \hat{\lambda}_2\\ &\leq \frac{NB}{2(B-1)D}\end{aligned}$$

But more simply, we can also write $\hat{\lambda}_1 \leq N$.

4.3.2.4 Discounted kernel

If two paths differ on d nodes, they have $D - d$ nodes in common.

$$\begin{aligned}\chi_d &= \frac{1-\gamma^{2(D-d)}}{1-\gamma^2}\\ \chi_j - \chi_{j+1} &= \frac{(\gamma^2)^{D-(j+1)} - (\gamma^2)^{D-j}}{1-\gamma^2}\\ &= (\gamma^2)^{D-j-1}\\ \bar{\lambda}_i &= \sum_{j=0}^{i-1} B^j(\chi_j - \chi_{j+1})\\ &= \gamma^{2D-1}\sum_{j=0}^{i-1}\left(\frac{B}{\gamma^2}\right)^j\\ &= \frac{\gamma^{2D+1}}{B-\gamma^2}\left(\left(\frac{B}{\gamma^2}\right)^i - 1\right)\\ \bar{\lambda}_{D-i} &\leq \frac{N\gamma}{B-\gamma^2}\left(\frac{B}{\gamma^2}\right)^{-i}\\ \hat{\lambda}_t &\leq \frac{N\gamma}{B-\gamma^2}\left(\frac{B}{\gamma^2}\right)^{1-\log_B(t)}\\ \hat{\lambda}_t &\leq \frac{NB}{\gamma(B-\gamma^2)t^{1+2\log_B(1/\gamma)}}\end{aligned}$$

We note that, although the constant in the expression for $\hat{l}(t)$ grows for larger γ values, i.e. smoother functions, the decay rate in t gets faster.

4.3.2.5 Gaussian kernel

Value of χ_d and $\bar{\lambda}_i$

$$\chi_d = \exp(-\frac{d}{s^2})$$

For all j, $\chi_j - \chi_{j+1} = (1-\exp(-\frac{1}{s^2}))\exp(-\frac{j}{s^2})$, hence for all $i < D+1$,

$$\begin{aligned}\bar{\lambda}_i &= (1-\frac{q_s}{B})\sum_{j=0}^{i-1} q_s^j \\ &= c_s(q_s^i - 1)\end{aligned}$$

where

$$\begin{aligned}q_s &= B\exp(-\frac{1}{s^2})) \\ c_s &= \frac{1-\frac{q_s}{B}}{q_s - 1}\end{aligned}$$

By definition, $q_s < B$. Let us focus on the case where $1 < q_s$ so that c_s is always positive, which is equivalent to:

$$s > \frac{1}{\sqrt{\log(B)}}$$

Bounds on $\hat{\lambda}_t$ Once again, Inequality (4.2) gives us a lower and an upper bound on $\hat{\lambda}_t$:

$$\begin{aligned}c_s(q^D q^{-\log_B(t)} - 1) \leq \quad & \hat{\lambda}_t & \leq c_s(q^D q^{-\log_B(t)} q - 1) \\ & q^{-\log_B(t)} & = t^{-\log_B(q)} \\ & q^{-\log_B(t)} & = t^{-1+\frac{1}{s^2\log(B)}} \\ \frac{1}{t} \leq \quad & q^{-\log_B(t)} & \leq \frac{1}{t}\exp\left(\frac{D}{s^2}\right) \quad \text{since } t \leq \min(N,T) \leq B^D \\ \frac{1}{t} \leq \quad & q^{-\log_B(t)} & \leq \frac{N}{t}q^{-D} \\ \frac{c_s(N\exp(-\frac{D}{s^2}) - t)}{t} \leq \quad & \hat{\lambda}_t & \leq \frac{c_s(Nq_s - t)}{t}\end{aligned}$$

And thus we can take:

$$\hat{l}(t) = \frac{c_s q_s}{t}$$

Influence of the kernel width Note that

$$\begin{aligned}c_s q_s &= \frac{(B-q_s)q_s}{B(q_s-1)} \\ &= \left(1+\frac{1}{q_s-1}\right)\left(1-\frac{q_s}{B}\right)\end{aligned}$$

and q_s increases when s increases, hence $\frac{1}{q_s-1}$ decreases and $-q_s$ decreases. As a result, $c_s q_s$ decreases. Also, since q_s tends to B when s tends to infinity, the limit of $c_s q_s$ is 0 when s tends to infinity. The $\hat{\lambda}_t$ upper-bound improves over that of the linear kernel when s is big enough so that $c_s q_s \leq \frac{B}{(B-1)D}$.

Now, let us look at the rate at which $c_s q_s$ tends to zero: when s is bigger than $\frac{1}{\sqrt{\log\left(\frac{B}{2}\right)}}$, we have:

$$\begin{aligned} c_s q_s &\leq 2\left(1 - \exp\left(-\frac{1}{s^2}\right)\right) \\ &\leq 2\left(\frac{1}{\sigma^2} + o\left(\frac{1}{s^2}\right)\right) \end{aligned}$$

Hence:

$$c_s q_s = O\left(\frac{1}{s^2}\right) \tag{4.5}$$

4.3.3 Regret bounds

In this section we prove Proposition 7 by using Equation (4.1) as a starting point (see page 93). We lower-bound m_t by 0 and we upper-bound it by T. We assume that $\hat{\lambda}_t \leq N\alpha_1 t^{-\alpha_2}$, as it is the case for the linear, discounted and Gaussian kernels (see Proposition 6). One way to bound the sum of log terms is with a tail-sum of the terms inside the logarithm: $\log(1 + s_{\text{noise}}^{-2} m_t \hat{\lambda}_t) \leq s_{\text{noise}}^{-2} m_t \hat{\lambda}_t$. However, this introduces a T factor (upper bound on m_t) and a N factor from the upper bound on $\hat{\lambda}_t$, which results in a regret bound that is worse than linear and involves large constants. Another way is to bound this sum of log terms by a sum of log-eigenvalues, which we do in Section 4.3.3.1. We can actually stop this sum at $T' \leq \min(T, N)$ such that $\forall t > T', m_t = 0$, and we show in Section 4.3.3.2 that the smallest T' that verifies this scales in $O(\sqrt{NT})$ when the kernel is linear or Gaussian.

4.3.3.1 Bound based on the sum of log-eigenvalues

We bound the sum of $\log(1+s_{\text{noise}}^{-2} m_t \hat{\lambda}_t)$ by a sum of $\log(c'\hat{\lambda}_t) = \log(c'N\alpha_1)+\alpha_2 \log(1/t)$, which can in turn be bounded owing to a result on the sum of $\log(1/t)$:

$$\begin{aligned}
\sum_{t=2}^{T'} \log(1/t) &\leq \sum_{t=1}^{T'} \log(1/t) \\
&\leq \log\left(\frac{1}{T'!}\right) \\
&\leq -\log(\Gamma(T'+1)) \\
&\leq -T' \log\left(\frac{T'+1}{e}\right) \\
&\leq T' \log\left(\frac{e}{2}\right)
\end{aligned}$$

using the fact that $\Gamma(x) \geq (\frac{x}{e})^{x-1}$. Let us consider the Gaussian kernel.[2] Because $1 \leq \frac{\hat{\lambda}_t}{\hat{\lambda}_T}$ and $m_t \leq T$, we have:

$$\begin{aligned}
\log(1+s_{\text{noise}}^{-2} m_t \hat{\lambda}_t) &\leq \log\left(\left(\frac{1}{\hat{\lambda}_T} + s_{\text{noise}}^{-2} T\right)\hat{\lambda}_t\right) \\
\sum_{t=1}^{T'} \log(1+s_{\text{noise}}^{-2} m_t \hat{\lambda}_t) &\leq \log\left(\left(\frac{1}{\hat{\lambda}_T} + s_{\text{noise}}^{-2} T\right)\alpha_1 N\right) T' + \alpha_2 \sum_{t=2}^{T'} \log\left(\frac{1}{t}\right) + \log(\hat{\lambda}_1) \\
G_T^g &\leq \log\left(\left(\frac{1}{\hat{\lambda}_T} + s_{\text{noise}}^{-2} T\right)\alpha_1 \left(\frac{e}{2}\right)^{\alpha_2} N\right) T' + D\log(B)
\end{aligned}$$

By extracting $\log(\hat{\lambda}_t)$ terms from G_T^g, we take advantage of the log but we also introduce a $\frac{1}{\hat{\lambda}_T}$ term which will be larger for smoother kernels. Indeed, for the Gaussian kernel:

$$\begin{aligned}
\frac{1}{\hat{\lambda}_T} &\leq \frac{1}{\hat{\lambda}_N} = \frac{1}{\bar{\lambda}_1} \\
&\leq \frac{1}{1-e^{-1/s^2}} \\
&\leq \frac{1}{\frac{1}{s^2} + o\left(\frac{1}{s^2}\right)} \\
&= O(s^2)
\end{aligned}$$

Using this in the G_T^g bound with $\alpha_1 = c_s q_s = O\left(\frac{1}{s^2}\right)$ and $\alpha_2 = 1$ gives:

$$G_T^g \leq \log\left(\left(o(1) + s_{\text{noise}}^{-2} O\left(\frac{1}{s^2}\right)\right)\frac{e}{2} NT\right) T' + D\log(B) \tag{4.6}$$

We thus see that the infogain bound improves for smoother Gaussian kernels, i.e. bigger s. We either get a regret in $\tilde{O}(\sqrt{NT})$ or in $\tilde{O}(T)$ depending on whether we bound $T' \leq \min(T, N)$ by N or T: the former implies a smaller rate in time and a bigger constant, the latter implies a smaller constant but a rate too high to make the bound interesting.

[2] The derivations for the linear kernel are identical and just involve different constants.

4.3.3.2 Tighter bound on the number of different arms selected by the infogain greedy algorithm

We know that the infogain greedy procedure chooses eigenvectors of $\mathbf{K}$ among the T that have highest associated eigenvalues. However, we may have only picked T' different eigenvectors, because we picked several times the same ones (m_i gives the number of times we picked the i^{th} eigenvector of $\mathbf{K}$). We look for the smallest T' such that:

$$\forall t > T', m_t = 0 \tag{4.7}$$

The contrary to Equation (4.7) is equivalent to choosing $\mathbf{u}_{T'+1}$ at least once. This is equivalent to the fact that there exists t, first time we select $\mathbf{u}_{T'+1}$, such that all eigenvalues $\hat{\hat{\lambda}}_{i,t}$ of $\mathbf{\Sigma}_t$ are smaller than $\hat{\hat{\lambda}}_{T'+1,t} = \hat{\lambda}_{T'+1}$. This can be written:

$$\begin{aligned} \exists t \leq T, \forall i \leq T', \frac{\hat{\lambda}_i}{1 + s_{\text{noise}}^{-2} m_{i,t} \hat{\lambda}_i} &\leq \hat{\lambda}_{T'+1} && (4.8) \\ \frac{1}{\hat{\lambda}_{T'+1}} - \frac{1}{\hat{\lambda}_i} &\leq s_{\text{noise}}^{-2} m_{i,t} && (4.9) \end{aligned}$$

Therefore, Not Equation (4.7) is equivalent to Equation (4.9). Let us assume that the latter is true. We know that each $m_{i,t}$ is smaller than $m_{i,T}$ and that $\sum_{i=1}^{T'} m_{i,T} \leq T$, hence:

$$\sum_{i=1}^{T'} \left(\frac{1}{\hat{\lambda}_{T'+1}} - \frac{1}{\hat{\lambda}_i} \right) \leq s_{\text{noise}}^{-2} T \tag{4.10}$$

Thus, we can find T' such that Equation (4.7) is true by lower bounding $\sum_{i=1}^{T'} \left(\frac{1}{\hat{\lambda}_{T'+1}} - \frac{1}{\hat{\lambda}_i} \right)$ and looking for T' such that this lower bound is equal to $s_{\text{noise}}^{-2} T$. From the $\hat{\lambda}_t$ upper and lower bounds established in the previous section for the Gaussian kernel, we have:

$$\begin{aligned} \sum_{i=1}^{T'} \left(\frac{1}{\hat{\lambda}_{T'+1}} - \frac{1}{\hat{\lambda}_i} \right) &\geq \sum_{i=1}^{T'} \frac{T'+1}{N c_s q_s} - \frac{i}{c_s \left(N \exp\left(-\frac{D}{s^2}\right) - i \right)} \\ &\geq \frac{1}{a_s} T'(T'+1) \\ \text{where } a_s &= \frac{N c_s q_s \left(\exp\left(-\frac{D}{s^2}\right) - 1 \right)}{\exp\left(-\frac{D}{s^2}\right) - 1 - \frac{q_s}{2}} \end{aligned}$$

thus we look for $0 < T' < T$ such that $T'^2 + T' - s_{\text{noise}}^{-2} a_s T = 0$:

$$T' = \frac{\sqrt{1 + 4 s_{\text{noise}}^{-2} a_s T} - 1}{2} = O(\sqrt{NT})$$

Since a_s tends to 0 when s tends to infinity, the bigger s, the smaller the constant in the upper bound for T'.

The same idea can be applied to the case of the linear kernel by using the corresponding upper and lower bounds for $\hat{\lambda}_t$ – which only differ from those for the Gaussian kernel by constants.

4.4 Discussion

We discuss GPTS in comparison with other algorithms for tree search.

4.4.1 Model

Smoothness of f For any two nodes x and x' with same parent n at depth d, there exist two leaves $\mathbf{x}$ and $\mathbf{x}'$ (with ancestors x and x', respectively) such that $f(x) - f(x') = f(\mathbf{x}) - f(\mathbf{x}')$ (by definition of f on interior nodes, which is the maximum value of f on descendant leaves). With the GP assumption, $(f(\mathbf{x}), f(\mathbf{x}'))^\intercal$ lies with high probability within an ellipse determined by the kernel product between $\mathbf{x}$ and $\mathbf{x}'$ (equal to the depth of x and x', when considering the linear kernel). One can thus say how close the f values of two siblings may be, and thus bound $f(n) - f(\mathbf{x})$ in terms of d, with high probability, in order to give a rough comparison with the BAST smoothness assumption. Although this bound is only with high probability – while it would always hold with BAST – GPB makes an extra assumption on how the f values are distributed.

The GP smoothness assumption is global, whereas BAST only assumes smoothness for η-optimal nodes. We note that GPs can estimate the parameters of the covariance function by maximising the likelihood of the training data.

Reward variability BAST assumes that the reward at each leaf is always in $[0, 1]$ and is given by a probability distribution whose nature is unknown (it could be different for each node) and with mean equal to the f value at that leaf. GPB assumes that the reward distribution is Gaussian with standard deviation s_{noise}. However, Proposition 4 also provides theoretical guarantees when this is not the case and the noise sequence is an arbitrary martingale difference sequence uniformly bounded by s_{noise}.

4.4.2 Computational cost

With UCT and BAST, we only need to keep track of two quantities for each node, which define its U-value: the number of visits of this node and the average of the rewards obtained by paths that went through this node. At each iteration, the algorithm chooses a path to explore by starting from the root node and by repeatedly selecting a child node with highest U-value. The obtained reward is back-propagated from the leaf node of the path to the root node, in order to update the average reward values, to increment the number of visits, and thus to update the U-values of these nodes. As a consequence, the cost of an iteration is constant.

With GPTS, however, we need to update all dummy nodes at each iteration, and the number of dummy nodes grows in time. Each dummy node is the sibling of a node at a certain depth that belongs to a path in training, and as a consequence, all dummy nodes have different $\mathbf{k}_t$ vectors. As a consequence, even when benefiting from the computational improvements offered by the online GPB updates and when limiting the size of the training set to a logarithmic function, iterations of GPTS are expensive compared to BAST (they are at least quadratic in time). The dummy nodes that are created when exploring a new subtree at time $t+1$ have same $\mathbf{k}_t$ vectors, and they have $\mathbf{k}_{t+1}$ vectors that only differ at their last coordinate. It might be possible to further exploit the structure of the tree and the relationships between dummy nodes' $\mathbf{k}_t$ vectors in order to speed up the dummy nodes updates and therefore the GPTS algorithm. Besides, an interesting property of the updates is that all leaf nodes in a fully explored subtree share the same update after exploring a dummy node. Although using this property would not reduce the theoretical complexity of the algorithm, it could offer a minor improvement on the number of computations in certain cases, and it shows that the current implementation of GPTS does not fully exploit the tree structure.

4.4.3 Tree growing method

Because we consider tree paths as arms of a bandit problem in GPTS, we need all paths to have same length. BAST can either be run in iterative-deepening or fixed-depth mode. Supposedly, it is more efficient in its iterative-deepening version (but no regret bound was given by Coquelin and Munos for this version). Because GPTS creates dummy nodes at different depths and considers all of them when deciding where to explore, it also grows the tree asymmetrically by going deeper only in regions of interest – but it has a maximum depth.

It may be possible to run GPTS in iterative-deepening mode, by considering that there is no maximum depth and that all leaf nodes are dummies. A random walk would consist of creating only one child node. We would also need to work with a kernel which would only be based on the number of nodes that paths have in common (not on a maximum depth).

Finally, we remark that GPTS lends itself to *progressive widening* (Coulom, 2007) in a very simple way, owing to the use of dummy nodes. Progressive widening is a technique which is used for planning in MDPs with infinite action spaces, or more generally it is used in order to deal with cases where the branching factor is large with respect to the number of interactions with the environment. In the work of Rolet et al. (2009), for instance, a node's children correspond to a discretisation of a continuous space, and it is useful to refine the discretisation as the tree search algorithm keeps choosing this parent node. For this, when starting a random walk from a dummy node in GPTS, we can simply make sure that we never delete this dummy node, so that we always consider that more siblings exist. Each time a dummy node is chosen by the algorithm, the branching factor increases.

4.4.4 Regret bounds

Theorem 4 of Coquelin and Munos (2007b) gives a regret bound when ρ_d decreases exponentially: $\rho_d = \delta\gamma^d$. The bound is written in terms of the parameters of the smoothness assumption (namely η, δ, γ) and is independent of time. This bound is problem-specific as it involves the inverse of the Δ_{min} quantity, where $\Delta_{min} = \min_i\{\Delta_i = f^* - f(i)\}$. Note that when f has B^D possible inputs, $1/\Delta_{min}$ can easily be of the order of B^D. While the bound is interesting asymptotically, the number of iterations T of the tree search algorithm is unlikely to go past B^D for most interesting values of B and D. Another issue with the $1/\Delta_{min}$ term is that, the smoother f, the bigger $1/\Delta_{min}$ and the bigger the regret – whereas we would actually like to take advantage of the smoothness of f to improve the regret. The non-dependency w.r.t. Δ_{min} usually comes at the price of a stronger dependency on time T, as it is the case with UCB1 – $O(\log(T))$ vs. $O(\sqrt{T})$.

Our GPTS regret bound is problem-independent; it scales in the square root of T, up to logarithmic factors and constant factors that decrease for smoother functions. Since GPTS does more updates at each iteration than BAST (which only updates values at the nodes of the last path that was selected), it is learning more, and its regret should be smaller. One way to improve the current GPTS regret bounds would be to use the fact that we know the eigenvalues of $\mathbf{K}$ exactly in order to provide an expression (or an upper bound) in terms of t and T for the numbers of times that the greedy infogain algorithm selects each eigenvector of $\mathbf{K}$ – this was previously bounded by T.

4.4.5 MDP planning

Planning in MDPs with deterministic transitions is an example of a setting where D can be a function of the horizon T. As seen in Section 2.3.3, it is convenient to take $D = \log_B(T)$, which implies $N = T$. In this case, we don't have constants in the order of B^D anymore and we are interested in the growth rate in T of asymptotic regret bounds. We have:

$$\begin{aligned} G_T^g &\leq \max_{(m_i)_i} \frac{1}{2} \sum_{i=1}^{\min(T,N)} \log(1 + s_{\text{noise}}^{-2} m_i \hat{\lambda}_i) \\ &\leq \max_{(m_i)_i} \sum_{i=1}^{T} \log(1 + s_{\text{noise}}^{-2} m_i T \hat{l}(i)) \end{aligned}$$

When splitting the sum at $t = T_*$ and writing $r = \sum_{i=1}^{T_*} m_i$, we can bound m_i by r when $i \leq T_*$ and $T - r$ when $i > T_*$:

$$G_T^g \leq \max_{1 \leq r \leq T} T_* \log(1 + s_{\text{noise}}^{-2} r T \hat{l}(1)) + (T - r) T s_{\text{noise}}^{-2} \sum_{t=T_*+1}^{T} \hat{l}(i)$$

$\hat{l}$ is a decreasing function, so we can bound its tail-sum by N times a tail integral of $\hat{l}$. We write $a = \log_B(1/\gamma) > 0$.

$$\begin{aligned} \sum_{t=T_*+1}^{T} \hat{l}(i) &\leq \int_{t=T_*+1}^{T} \hat{l}(t) dt \\ &\leq \frac{B}{\gamma(B - \gamma^2) 2a T_*^{2a}} \end{aligned}$$

When $T_* = T^{1/a}$, the first term dominates in the upper bound of G_T^g, so that $r = T$ and we have:

$$R_T = \tilde{O}(T^{1/2 + 1/(2a)})$$

This is sub-linear for:

$$\begin{aligned} 1/(2a) &< 1/2 \\ \log_B(1/\gamma) &> 1 \\ \gamma &< 1/B \end{aligned}$$

Usually, the discount factor γ is close to 1 and the number of actions is at least 2, so the case where $\gamma < 1/B$ is not really interesting. Note that the bound we give suffers from the very loose bound on m_t ($\leq T$), and future work should address this.

OLOP uses the fact that the intermediate reward values are observed, whereas GPTS can only use information in the form of arm-reward pairs where arms are leaves. We would need interior nodes to be considered as arms of a bandit problem if we wanted to add intermediate reward values to the training data. The difference in performance between OLOP and GPTS can be thought of as the "price" of the information that GPTS is not able to take into account. The regret of GPTS is $T^{1/(2a)}$ times the regret of OLOP when $\gamma \leq \frac{1}{\sqrt{B}}$, and $T^{a+1/(2a)-1/2}$ times the regret of OLOP otherwise.

The relevance of GPTS for MDP planning is questionable, but the method we used for its theoretical analysis in this setting may be relevant for other problems where

- we only observe reward values on whole paths;
- we have to decide how deep to go down the tree based on the horizon;
- stopping at a fixed depth introduces a linear cost;
- the deeper we go down the tree, the higher the covariances between paths.

One such problem is hierarchical optimisation.

4.4.6 Application to optimisation

4.4.6.1 A model for GP hierarchical optimisation

Coquelin and Munos (2007b) present empirical results for BAST applied to a 1D optimisation problem, which has probably inspired the HOO algorithm. Similarly, we should be able to apply GPTS in a manner similar to HOO in order to find the maximum of a function in a space for which we are given a tree of coverings. As for discounted MDP planning, we can choose D as a function of T. Each leaf of the tree corresponds to a region of the search space (regions get smaller as D increases), and we aim at learning the average f values in these regions. When a leaf is reached, we receive a reward as the f value of a sample of the uniform distribution in the corresponding search region.

In GP optimisation, we assume a covariance function κ between inputs. Here, we consider the covariance between any two paths $\mathbf{p}_1$ and $\mathbf{p}_2$ of the tree of coverings, corresponding to regions $\mathcal{X}_1$ and $\mathcal{X}_2$ of the input space. This is the covariance of the function values for random variables X_1 and X_2 that follow uniform distributions in their respective regions. Recall that the (unknown) function value for an input x is represented by a random variable notated F_x.

$$\begin{aligned}
\operatorname{cov}(\mathbf{p}_1, \mathbf{p}_2) &= \operatorname{cov}(F_{X_1}, F_{X_2}) \\
&= \int f_1 f_2 p(f_1, f_2) df_1 df_2 \\
&= \int f_1 f_2 p(f_1, f_2 | X_1 = \mathbf{x}_1, X_2 = \mathbf{x}_2) p(\mathbf{x}_1, \mathbf{x}_2) df_1 df_2 d\mathbf{x}_1 d\mathbf{x}_2 \\
&= \int \kappa(\mathbf{x}_1, \mathbf{x}_2) p(\mathbf{x}_1, \mathbf{x}_2) d\mathbf{x}_1 d\mathbf{x}_2 \\
&= \frac{1}{|\mathcal{X}_1||\mathcal{X}_2|} \int \kappa(\mathbf{x}_1, \mathbf{x}_2) d\mathbf{x}_1 d\mathbf{x}_2
\end{aligned}$$

by definition of the covariance in $\mathcal{X}$, the fact that f has a zero-mean prior, and that X_1 and X_2 are independent and uniformly distributed in spaces of volumes $|\mathcal{X}_1|$ and $|\mathcal{X}_2|$.

We note, however, that there are as many possible kernel product values as there are leaves in the tree of coverings (i.e. 2^D), whereas our description and analysis of GPTS concerned cases where the covariance only depends on the number of nodes in common (there were D possible kernel product values). This was a precondition to our eigenvalue analysis, but also to the efficient implementation of GPB for tree search with the use of dummy nodes. We may depart from this model and consider a simpler one where, as with HOO, we do not account for the fact that leaves may correspond to regions of the search space that are very close, even when they have few ancestors in common. To illustrate, consider a recursive splitting in two of $[0, 1]$: there is a leaf that corresponds to the region immediately below 0.5, and another leaf that corresponds to the region immediately above. These two regions are next to each other, but the corresponding leaves only have the root node as a common ancestor.

We can choose to only model covariances between leaves based on the number of ancestors they share. Some covariances will be overestimated, and others will be underestimated. The model is characterised by a set of χ_d values, which are initially unknown but can be set to maximise the likelihood of the observed data. Before any data is observed, we can initialise these values to those of the linear kernel, scaled so that χ_0 corresponds to the signal variance (if it is known beforehand).

4.4.6.2 Remarks

Hierarchical optimisation has the following properties:

- it offers a unified framework for many optimisation problems (not just in $\mathbb{R}^n$): the search space can be the Cartesian product of an arbitrary (and possibly infinite) family of discrete and continuous sets;
- it simplifies the dependency structure, compared to "normal" GP models: there is only a finite number of possible kernel values;
- choosing a point where to sample the function is straightforward, as no search heuristics are required to approximate the sample acquisition criterion.

In their application, Coquelin and Munos tried several values for the smoothness parameter δ (ρ in our notations), but no method was given in order to tune its value for the problem at hand. By adapting the hierarchical optimistic strategy to the GP framework, we would benefit from the Bayesian way of updating the parameters of the model based on new observations. Besides, Bubeck et al. (2010) tell us that no performance guarantees are available for HOO with an overestimated smoothness, and that the algorithm may get stuck indefinitely in a local optimum of f. We expect the GP model to be more robust to misestimations of its parameters since regret bounds are also available for the more agnostic setting where f has finite norm in the RKHS induced by the covariance function.

It may be interesting to consider a tree of coverings with a branching factor that is greater than 2. For example, at each level of the tree we could split the search space in two according to each of its n dimensions: B would be equal to 2^n, and the size of the coverings would decrease more rapidly. By only splitting according to one dimension, HOO privileges the dimensions along which it splits first – which are chosen arbitrarily. Indeed, the more reliable reward estimates in tree search are at smaller depths.

5 Empirical Results

We introduce a new Matlab toolbox for bandits and tree search ("Bats") aimed to make the empirical evaluations of algorithms easy to perform by defining common interfaces, and to cut down development time and errors of implementation by sharing code among them. For instance, LinRel and GPB are kRR extensions of UCB1 that only differ from each other in the way that they compute their uncertainty estimates, BAST is based on instances of UCB1, and GPTS is based on an instance of GPB. We explain how the Object Oriented code of the framework was structured.

We have presented GPB as an alternative to LinRel, and, as such, we now test its performance on a Content-Based Image Retrieval (CBIR) task for which LinRel was found to be successful. The problem is to assist a user to find images that are relevant to his search, by using feature representations of the images and binary feedback in order to learn what the user is interested in, as he browses results pages. The bandit algorithm is used to deal with the exploration-exploitation tradeoff imposed by the necessity of reducing the uncertainty in the learnings and of presenting relevant images as early as possible. In our experiments, for each category of images taken from the VOC'2007 Challenge data (Everingham et al.), we consider the task of finding images that belong to this category; the algorithm chooses which images to present to the "user", and feedback is given for these images only, based on the fact that they belong or not to the category. Performance is measured by the average precision, which privileges algorithms that find relevant images early. Our results validate the pertinence of the GP model in practise. We were also able to improve the performance of GPB for a few categories, by periodically updating its hyper-parameters (the weights on the different image features) using the Maximum Likelihood principle. This means that, with the GP model, we may be able to learn which features are important for the current search session, based on the feedback given by the user.

We also study the performance of GPTS on different synthetic trees, with rewards that are built as discounted sums of unobserved, intermediate rewards at each node. We vary the parameters of the tree search problems (branching factor, depth, noise variance, discount factor) in order to study how they impact the regret of the algorithm. We compare to BAST and show that GPTS handles large branching factors ($B = 200$, as one could expect in Go game trees) particularly well. It is likely that this is due to the fact that, in BAST, the initialisation of the UCB1 instances at each node requires playing each child node at least once before being able to make an informed decision on the child node to select. GPTS, however, maintains confidence intervals for all unexplored subtrees (the number of which scales linearly in T) and updates each of them after each new observation. This is also what makes the GPTS iterations more costly than the BAST iterations.

All experiments were performed with the Bats toolbox for Matlab. The code for both the toolbox and the experiments was released under a GPL License and it can be downloaded at `http://louis.dorard.me/bats`, along with unit tests and the libraries required for its functioning.

5.1 The Bats toolbox for Matlab

The Bats toolbox for Matlab contains implementations of the Gaussian Process Bandit and Tree Search algorithms, as well as of other standard algorithms. In the following we give a high level description of the contents and functioning of the toolbox. We start by describing the Bandits framework, which will be used for the construction of the Tree Search framework. We also introduce the Experiments framework for running algorithms multiple times on several problem instances, which was used for the experiments presented in the rest of this chapter.

The use of the Object Oriented features of the Matlab language allows us to represent the interactions between agent and environment, and to define common interfaces that enable the testing of different algorithms on different problem instances with the same code – only the construction of the agent and environment changes. Inheritance allows us to share code between algorithms that have things in common. For instance, in this toolbox, UCB1 is an implementation of a UCB-type algorithm, and LinRel and GPB are implementations of a class of kRR-UCB-type algorithms, which derive from UCB-type algorithms.

In Object Oriented programming, *classes* are constituted of *properties*, and *methods* that perform operations on these properties. *Abstract* classes are used to define properties, method profiles and implementations that are common to a family of algorithms, but they are not instantiable themselves, as some methods are left unimplemented. We describe the relationships between the classes of this toolbox, their properties and methods, and how they implement the algorithms that we presented and that we introduced in this work. More information on the properties and methods can be found by browsing the Matlab HTML documentation (‘doc bats’). The source code also contains many comments that explain in detail how the implementations work. In the following, we denote properties, methods, functions and classes with `this typeface`; class names are capitalised.

5.1.1 Bandit problems

5.1.1.1 Environment

The environment is where the reward function is defined. The latter is kept as a private property of the `Environment` abstract class[1] so that it is not directly accessible by the outside. `reward` is expressed as a function but it can also represent a distribution through the use of random generators in Matlab. Reward samples can be obtained for a chosen input by calling the public method `play`, which calls `reward`, keeps track of the arms that have been played (`X`), of the rewards that have been obtained (`Y`), and of the number of plays (`t`).

The `EnvironmentBandit` class extends this base class for the multi-armed bandit problem. It keeps a list of mean-reward values `rewardList` (also a private property), used by the class to determine which arm is best and to define the regret `R` and the empirical regret `eR`. The constructor initialises the reward function based on a list of mean rewards given in input, and accepts 3 types of rewards:

- 'bernoulli', where the reward for arm i is 1 with probability `rewardList(i)` (must be in $[0, 1]$), and 0 otherwise;
- 'bernoulli2', same as the above but with reward -1 instead of 0;
- 'normal', where each arm i's reward distribution is a one-dimensional normal distribution with mean specified by `rewardList(i)` and variance given as an extra input;
- 'gp', which is similar except that the list of mean rewards is not given but is drawn from a multi-variate Gaussian with zero mean and covariance matrix given in input.

The environment also allows for feature representations of arms (given as optional inputs to the constructor) and can proceed to their normalisation, if specified. Arms can be added at any stage owing to the `addArm` method (which also returns the normalised feature representation of the arm). This can be particularly useful when we choose not

[1]Matlab is also a functional programming language and considers functions as an object type.

to represent all arms because of memory constraints, but to add them sequentially, as would be the case for GPTS and for item recommendation. Note that this poses a problem for determining the regret: the best arm can change as arms are added to the environment; however, we might know in advance what the best possible reward value `rbest` would be.

5.1.1.2 Bandit algorithms

Initialisation The `BAlg` abstract class defines basic methods that are shared by all bandit algorithms. The constructor is given the number of arms `N` specified by the environment, and optionally a list of feature representations of these arms. It is useful in certain cases to identify arms by labels (see 5.1.2.2 for instance), also given as optional inputs to the constructor. New arms can be added to the bandit algorithm after they have been added to the environment, and a normalised feature representation can be specified.

Training When reward samples are received from the environment, they are added to the bandit algorithm's training set `Tr` through the `train` method. Arms can be identified by their label, and when an arm's index has been found, it is fed to the `addTraining` method. If the size of the training set becomes larger than a given function `S` of the number of iterations, the oldest training point is removed. The `train` method relies on `removeOldestTraining` and `addTraining` to remove/add points from/to the training set. The former method is called first. Both methods are left for implementation and are expected to update the algorithm's knowledge. We increment the number of iterations `t` at each data point added to the training set. This should match the value of the `t` property of the environment, since training points are obtained by playing arms. However, we do not decrement `t` when removing points form the training set – the number of elements in the training set is given by `ntr`.

Choosing arms The most often-called method is `choose`, whose interface is defined in `BAlg` but left for implementation. By default, any arm can be chosen to be played, but this can be changed to restrict possible outputs to the list of arms indicated in the `playable` property. Also, the `chooseNew` property indicates whether we want to force the algorithm to always choose arms that have never been played before, or not. This is useful for applications to content-based document retrieval for instance, where we do not want to retrieve the same document twice. Besides, we may want to retrieve several documents at the same time. For this, the `chooseSimulated` method can return as many arms as specified, by duplicating the current instance and iteratively choosing and adding an arm to the training set, along with its estimated reward.

The `choose` method is always based on a list of estimations of all arms' mean-reward values. This list is defined in the `BAlg` class as the `M` property and is initialised to a list of 0 values. In the random implementation `RandBAlg`, the maximum size of the training set is set to 0 and `M` is never updated. Arms are chosen randomly among the list of playable arms. `M` corresponds to what we notated $\boldsymbol{\mu}_t$, and the output of `choose` corresponds to i_{t+1}.

Code snippet We give below a typical sequence of calls to the environment and to the bandit algorithm. Note that only their initialisation is application-specific.

```
rl = rand(1,N); % rewards list (N arms)
e = EnvironmentBandit('bernoulli', rl);

b = RandBAlg(N);
x = b.choose();
y = e.play(x);
b.train(x, y);

xs = b.chooseSimulated(3);
```

5.1.1.3 UCB algorithms

UCB algorithms represent their knowledge on the arms' reward distributions with a list of estimated means (`M`) and variances (`V`, initialised to infinity). The choice of an exploration/exploitation balance function `beta` defines upper confidence values `U` for all arms, through the `updateU` method which simply sets `U = M + sqrt(beta(t)) .* sqrt(V)` and deals nicely with cases where some `V` values are equal to infinity and `beta(t) = 0`. The form of `beta`, as a function of the number of iterations, is usually fixed for a given algorithm, and its expression can admit parameters (such as δ in GPB).

The `UcbAlg` abstract class defines a base constructor that takes the same arguments as the `BAlg` constructor, plus an optional `delta` argument. Note that `beta` often depends on the total number of arms, hence we specify the existence of an `updateBeta` abstract method which is used both to initialise `beta` and to update it when adding new arms. This method requires that we memorise (with a class property) the `delta` value initially given to the constructor. A setter on `beta` is defined so that a change of value of `beta` is always followed by a call to `updateU`. The setter can also be used if we want `beta` to take another form, but this should be reserved to tests.

The `choose` method can be implemented in `UcbAlg`: it simply picks an arm with highest `U` value. In order to learn from experience, we need to extend the `addTraining` method by calling an abstract method `updateMV`, and then `updateU`. In the UCB1 implementation of class `UCB`, we define the `nplayed` property as the list of the number of times that each arm has been played, which `updateMV` uses to perform the computations defined in Equation (2.1) and (2.3). `updateBeta` defines `beta` as specified in Equation (2.2). This expression does not involve any parameter, and as a consequence, `delta` is left unspecified in the constructor.

5.1.1.4 Kernel Ridge Regression UCB algorithms

Core properties Algorithms that derive from the `kRRUcbAlg` abstract class, such as KLinRel and GPB, use kernel Ridge Regression to learn non-linear relationships between arm feature vectors and mean-reward function values. The computation of `M` and `V` therefore requires that we either pass a kernel matrix `K` to the constructor, or a kernel/covariance function `covfunc` and arms' feature representations. In the second case where the kernel matrix is not explicitly given, it is computed owing to the `kernelProducts` method – which also serves in the rest of this class to compute kernel products for new arms or new hyper-parameters. The `logtheta` property is a list of the logarithms of the hyper-parameters of the covariance function. This does not include other hyper-parameters of the model such as the noise standard deviation `signoise`, which is kept as a separate property.

Computing `M` and `V` As we have seen for GPB in Section 3.1.2.2, the `M` and `V` properties can be updated online. The desired update mode ('default', 'online1' or 'online2') is specified to the `kRRUcbAlg` constructor.[2] The implementation of `updateMV` is fixed and relies on methods that compute `M` and `V` values, or incremental updates `dM` and `dV`. The methods that compute the `M` values are already implemented, because

[2]The 'online3' version was not implemented in this first version of the toolbox.

they are the same for all kRR algorithms. The methods that compute the `V` values are abstract and their implementation will vary from one algorithm to the other.

- In the default mode, `M` and `V` can be expressed as affine functions `k2M` and `k2V` of the matrix of kernel products between all arms and the arms in training, and of the covariance matrix inverse `Ci`. The latter is recomputed when adding/removing points to/from the training set through the `updateCi` and `downdateCi` methods which implement the online update and downdate formulae for $\mathbf{C}_t^{-1}$.
- In the online2 mode, the $\boldsymbol{\alpha}$ vector is computed by `k2al` (see Equation (3.3)) which makes use of the matrix `Q`. It is then fed to `al2dM` and `al2V` which compute the difference between the new and the previous `M` and `V` values (see Equations (3.4) and (3.5) for GPB). The `Q` property is updated according to Equation (3.6) which is implemented in the `updateQ` method. We do not support the removal of data from training.
- In the online1 mode, we do not use `Q` but `Ci` to compute $\boldsymbol{\alpha}$, that we feed to the same methods as above. `Ci` is updated after `M` and `V`, and not before as it is the case with the default algorithm. The `M` and `V` downdates are done in `removeOldestTraining`: `M` is computed from scratch (Equation (3.1)) and `V` is downdated through the `downdateV` abstract method.

Estimating U values In certain situations, we need to estimate the `U` value of an arm that is not represented in the bandit algorithm's set of arms, based on its feature representation. In the `estimateU` method, `kernelProducts` is applied to the feature representation given in input and to the arms in training, and the result is fed to `k2M` and `k2V` in order to determine the `U` value for this arm. The procedure is the same for the default and online1 update modes (but is not available in online2 mode).

Adding arms In the current implementation, adding new arms is restricted to the case where a covariance function has been defined. It requires providing a feature description of the new arm so that the kernel products with the previously defined arms can be computed and the total kernel matrix can be extended. These kernel products are also used to set the `M` and `V` values for the new arm. The procedure is the same for the default and online1 update modes (but is not available in online2 mode).

Setters We specify setters for `logtheta` and `K` as we may wish to change their values if we decide to learn the kernel/covariance function from observed data. When changing existing entries of `K`, `Ci` and `Q` must be recomputed, and, based on their new values, `M`, `V` and `U` must be recomputed too. However, adding arms to the bandit problem does not impact these properties because it augments but does not change existing values of `K`. When changing the value of `logtheta`, we must recompute the total kernel matrix and reset `K`.

5.1.1.5 GPB

Hyper-parameter learning In the Gaussian Processes framework, `logtheta` can be learnt by maximising the likelihood of the observed data. The likelihood is determined according to the model, and is therefore a function of its parameters. We rely on the GPML toolbox (Rasmussen, 2010), namely on two functions:

- `gpr`: in the way we use it, this function takes a covariance function and training data in input, and outputs minus the log likelihood of the data along with its partial derivatives with respect to the hyper-parameters;
- `minimize`: minimises a differentiable multivariate function using conjugate gradients, based on the partial derivatives of that function and an initial guess of where the minimum could be.

Here, we plug the output of `gpr` to the input of `minimize`. Note that for this to work with any covariance function, we must make sure that it has been implemented according to the specifications of the GPML toolbox.

Updates in matrix form In order to speed up the Matlab computations, we rewrite the update formulae in matrix form so that no loops are needed (loops are inefficient in Matlab). For this, we write $\boldsymbol{\alpha}_{t+1}$ for the vector of $\alpha_{t+1}(\mathbf{x})$ values for all arms in $\mathcal{X}$. We also write $\mathbf{Q}_t$ for the matrix of $\mathbf{q}_t(\mathbf{x})$ vectors for all arms. For GPB-ONLINE1:

$$
\begin{aligned}
\boldsymbol{\mu}_{t+1}(\mathbf{x}) &= \boldsymbol{\mu}_t(\mathbf{x}) + \frac{y_{t+1} - \mu_t(\mathbf{x}_{t+1})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2}(\mathbf{K}(:, i_{t+1}) - \mathbf{K}(:, \mathcal{I}_t)^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1})) \\
\boldsymbol{\sigma}_{t+1}^2(\mathbf{x}) &= \boldsymbol{\sigma}_t^2(\mathbf{x}) - \frac{1}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2}(\mathbf{K}(:, i_{t+1}) - \mathbf{K}(:, \mathcal{I}_t)^\mathsf{T}\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x}_{t+1}))^2
\end{aligned}
$$

where the squared operator for vectors corresponds to the component-wise exponentiation to the power of 2.

For GPB-ONLINE2:

$$\begin{aligned}
\boldsymbol{\alpha}_{t+1} &= \mathbf{K}(:, i_{t+1}) - \mathbf{K}(:, \mathcal{I}_t)\mathbf{Q}(:, i_{t+1}) \\
\boldsymbol{\mu}_{t+1} &= \boldsymbol{\mu}_t + \frac{y_{t+1} - \mu_t(\mathbf{x}_{t+1})}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2}\boldsymbol{\alpha}_{t+1} \\
\boldsymbol{\sigma}_{t+1}^2 &= \boldsymbol{\sigma}_t^2 - \frac{\boldsymbol{\alpha}_{t+1}^2}{\sigma_t^2(\mathbf{x}_{t+1}) + s_{\text{noise}}^2} \\
\mathbf{Q}_{t+1} &= \begin{pmatrix} \mathbf{Q}_t - \text{repmat}(\boldsymbol{\alpha}_{t+1}^{\mathsf{T}}, t, 1) \times \text{repmat}(\mathbf{Q}_t(:, i_{t+1}), 1, N) \\ \boldsymbol{\alpha}_{t+1} \end{pmatrix}
\end{aligned}$$

where $\times$ for vectors denotes the component-wise multiplication.

The `kRRUcbAlg` class implements the above `M` $(= \boldsymbol{\mu})$ updates, while the `GPB` class implements the `V` $(= \boldsymbol{\sigma}^2)$ updates.

Code snippet We give below a typical sequence of calls to an environment and a bandit algorithm. Note that only their initialisation is application-specific.

```
% Initialise environment
% create N random vectors of dim dimensions, normally distributed
dim = 5;
features = randn(dim, N);
sigma = 1;
ker{1} = 'covSEiso';
% log of SE width and log of signal variance
ker{2} = [log(sigma); log(sqrt(1))];
normalise = true;
e = EnvironmentBandit('gp', {ker, signoise}, features, normalise);

% Initialise bandit
delta = 0.05;
labels = [];
b = GPB(ker, signoise, labels, e.features, delta, 'online1');
% note that the features are given by the environment
b.S = @(t) N./2;

e.iterations(b, N);
b.learnHyper();
xf = randn(dim, 1);
xfn = e.addArm('new', xf);
b.estimateU(xfn);
b.addArm('new', xfn);
```

Remark on the precision of Matlab's computations The Bats toolbox contains a bunch of tests to make sure that the implementations of the different versions of GPB are correct. In particular, we compare the `M` given by UCB1 to the `M` given by GPB with a kernel matrix equal to the identity matrix; we also compare the `U` values given by the different versions (default, online1, online2), with the same data in training. Although they should be the same, this is not the case in practise. Indeed, Matlab is imprecise when working with large vectors. For instance, if `x`, `y` and `z` are vectors, `[x';y']*z` is not exactly equal to `[x'*z;y'*z]`. These imprecisions are amplified in the online updates and after a large number of iterations, as previous computations are being reused and the imprecisions add up.

5.1.2 Tree Search

`TreeSearchInterface` specifies the profile of the `choose` method that should return a near-optimal path of given length `np` after a number `nit` of iterations. `BanditTS` provides a super class for single-bandit as well as many-bandits tree search algorithms: they all consist of an environment `e` and a tree structure `tree` where the explored nodes are stored. The `growMethod` property specifies how the tree should be grown – in a 'fixed-depth' or an 'iterative-deepening' way. In tree search environments, inputs to the reward function are sequences of node feature representations. `EnvironmentTS` extends `Environment` by providing one important additional property: the `offspring` function which lists the children (and their feature representations) that can be produced from a given node.

`BanditTS` implements the `choose` method by running the specified number of iterations, where each consists of searching the tree for a path to be played by the environment. The observed reward at time `t` is used to train the bandit(s) through the `train` method. At the end of these iterations, the `best` method selects the best path down the tree, based on the current learnings. Both methods are left for implementation. Besides, a `newChild` method is implemented and uses `tree` and `e` in order to explore a given node and create a new child to it: the method calls `offspring` in the environment in order to determine the features of possible child nodes at this place in the tree, selects one at random among those that are not yet in memory, and then stores it in the tree structure.

5.1.2.1 Tree representation

The `Tree` class implements a tree structure which, essentially, is a set of nodes indexed in $[1, nn]$ where nn is the number of nodes currently stored in the tree structure. We only store in the tree structure the nodes that have already been explored: the structure is made of regular nodes, and of dummy nodes that represent unexplored subtrees.

A number of properties (arrays) are used to describe the relationships between nodes: `parent`, `firstChild`, `lastChild`, `nextSibling`, `previousSibling`. They are not all necessary to characterise the tree, but keeping them in memory can facilitate certain operations and the navigation in the tree. The `getChildren` method, which lists the indices of children of a given node, and the `getPathTo` method, which lists the ancestors of a given node by increasing depth, are based on them.

The `features` property is used to store nodes' feature representations, which will be passed to the environment. Note that the tree search algorithms we consider here do not consider the node feature representations. Dummy nodes do not have feature representations, and their entries in `features` are columns of `NaN` values[3] – it is this property of dummy nodes which is used by the `isdummy` and `hasDummyChild` methods.

The tree structure can be made to have a maximum depth `maxDepth`, or, if this property is set to 0, it can be grown indefinitely, as for iterative-deepening search methods. At each time a new node is explored and added to the tree structure through the `createNode` method, the properties of the `Tree` class must be resized, which is costly. We avoid this by adopting the "doubling trick": we keep track of the number of nodes with the `nn` property and, when the length of `parent` becomes equal to `nn`, we double the size of all arrays.

We show below how a `Tree` object is displayed. The tree given in example is asymmetric (leaves are not always at the same depth). Indents are used to represent the tree structure in the "tree-like representation". Each line represents a node and gives the feature representation of the node, followed by its index in brackets. The node feature representation consists here of the depth of the node, an index among all nodes of same depth, and a binary intermediate reward value.

[3]`NaN` means "Not a Number" in Matlab.

```
obj =

  Tree handle

  Properties:
             parent: [0 1 1 1 4 3 3]
         firstChild: [2 0 6 5 0 0 0]
          lastChild: [3 0 7 5 0 0 0]
        nextSibling: [0 4 0 3 0 7 0]
    previousSibling: [0 0 4 2 0 0 6]
           features: [3x7 double]
                 nn: 7
           maxDepth: 0

  Methods, Events, Superclasses


Tree-like representation
------------------------

0  1  0 (1)
    1  1  1 (2)
    1  2  0 (4)
        2  1  1 (5)
    1  3  1 (3)
        2  2  0 (6)
        d (7)
```

Structure for many-bandits algorithms The `BTree` class extends `Tree` by storing instances of bandit algorithms at interior nodes in the `bandit` cell-array property, which will be useful for implementing many-bandits tree search algorithm. All bandits are of same type (for instance: Random, UCB, GPB) specified by `bType`, and with optional parameters `delta` and `paramA` passed to the bandit constructor when a new instance is created. The bandit instances have their corresponding node's children as arms (identified by their indices in the tree). `createNode` is extended so that it either creates a bandit at the parent node when creating a first child, or adds an arm to the parent's existing bandit. The `U` value of the new node is initialised to Inf, but this does not affect the `U` value of the parent.

A new method called `trainBandit` allows us to train the bandit instances at nodes that were in a path that was just played by the environment. The `b.U` values given to nodes by their parent's bandit `b` are used to define the `U` values for these nodes, which are updated as specified by Equation (2.18). ρ_d is specified by the `rho` property which is a function with inputs d and the maximum depth of the tree. In order to implement this update, `trainBandit` should be called from the bottom of a recently played path to the top: we need the children `U` values to be up-to-date, and changing the `U` value of the current node implies that the parent's `U` value will have to be updated.

5.1.2.2 A single-bandit algorithm: GPTS

Single-bandit algorithms implement superclass methods with a regular tree and only one bandit algorithm. In `GPTS`, the `BanditTS` class is extended with an instance of GPB, `b`. Because GPB can learn its hyper-parameters, we also add a `learnHyper` property that specifies how often we would like the algorithm to learn and update its hyper-parameters (0 for never).

In the GPB instance, arms correspond to dummy or leaf nodes. Their labels are the nodes' indices, which are used for identification. Their feature vectors are vectors of node indices, up to the depth of the dummy/leaf node, and `NaN` entries up to the maximum depth. We have implemented the discounted and the linear covariance functions (`covPathsDISC` and `covPathsLIN`), that work on these feature representations. They are based on the number of nodes in common between two paths.[4] The tree and the bandit are initialised in the constructor by doing a random walk down the tree, from the root node: the leaf node and the dummy nodes (`maxDepth` of them in this case) that are created during the `randomWalk` procedure constitute the initial arms of `b`.

The `train` method is simply implemented by calling `b.train` and, if specified by `learnHyper`, `b.learnHyper` is also called. The `search` method consists of calling a new method, `explorePathFromBandit`, with the result of `b.choose` passed in parameter. This method does the following: it takes an arm index and gets the feature representation of that arm from `b`; this is a path to a dummy or a leaf node, and in the former case, it performs a random walk until reaching a leaf node; the method then adds the leaf node and all the dummy nodes created by `randomWalk` to `b`, and returns the path to that leaf. The `best` method outputs the path with highest `b.M` value, and in case this is a path to a dummy node, it completes this path by calling `explorePathFromBandit`.

[4]We should consider that the number of nodes in common between a path to a dummy and itself is D.

5.1.2.3 Many-bandits algorithms

The `ManyBanditsTS` abstract class also derives from `BanditTS`. Its `tree` property is now a `BTree`, but the type of bandit algorithms to be used (UCB, GPB, etc.) is not specified here. `ManyBanditsTS` implements the superclass methods by calling the bandit algorithms that are stored at each node of the tree. For instance, `train` simply calls `tree.trainBandit` from the bottom to the top of the input path. The `search` method goes down the tree by starting from the root and sequentially calling the `next` abstract method to determine which child to go to next (possibly one that does not exist yet), and this until reaching the maximum depth (in 'fixed-depth' mode) or appending a child to a previously leaf node (in 'iterative-deepening' mode). `best` does something similar, except that it stops at the desired depth and always chooses children that already exist in the tree, based on the `M` values given to them by the bandit algorithm at the current node.

UCT, BAST and HOO The `UCT` class implements the `next` method of `ManyBanditsTS` by adding and returning a new child (with `BanditTS.newChild`) to the node given in input if the latter is a leaf node or has a dummy among its children, or by returning the child with highest U-value otherwise. The `UCT` constructor initialises `tree` to a `BTree` object with given `bType` which can be either 'random', 'ucb' or 'bast' – the latter means that the BAST exploration term should be used rather than UCB's. If `rho` was not specified to the `UCT` constructor, it is set to 0 and the algorithm behaves as UCT. Otherwise, `rho` is passed on to `BTree` and the algorithm behaves as BAST. Note that we can run HOO if `offspring` is based on a binary tree of coverings of the search space, `reward` is made to give the target function's value at a point sampled uniformly at random in the region of the space corresponding to the last node given in input to it, and `growMethod` is set to 'iterative-deepening'.

5.1.3 The Experiments framework

The empirical evaluation of the performance of a bandit algorithm requires performing several runs of this algorithm on the same problem: bandit algorithms often make randomised choices, and the rewards they get are usually stochastic. For the Pinview experiments, for instance, algorithms pick the 15 first images at random, and we can imagine that selecting a relevant image by chance can influence their subsequent performance. We may also want to evaluate algorithms on randomly selected instances of a same class of problems.

We introduce a simple framework that allows us to easily perform several runs of bandit algorithms and to summarise the results. Besides, we make sure that our implementations are randomised as they should by overloading the Matlab built-in `max` function: the UCB heuristic requires selecting arms with highest upper confidence bounds, and to break ties arbitrarily; for this, our version of `max` is such that if several elements of the input list have the same maximal value, one of them will be picked randomly and its index will be returned – whereas Matlab's built-in function would always return the index of the first element with maximal value.

5.1.3.1 Structure of the experiments

`ExperimentsAbstract` provides a super class for defining and running experiments. The `addAgent` and `addEnv` methods allow to define agents (i.e. bandit algorithms) and environments in which they will evolve, both characterised by a type (e.g. the name of a bandit algorithm) and parameters, and stored as cells of the `agent` and `env` properties. The `addExpe` method allows us to define experiments (cells of the `expe` property) that specify parameters which will be used by the `runOne` abstract method to run a given agent in a given environment.

Once experiments have been defined, they can be run a given number of times owing to `runExpe` or `runAll`. The entries of the `expe` property are also used to store the results of these runs. In order to deal with the stochasticity of bandit algorithms, we report their average performance over several runs (see next paragraph for an example of how this is displayed by the overridden `display` method). These results can be saved to a file owing to the `save` method.

5.1.3.2 An implementation for tree search problems

The `Experiments` class of the `ToyTS` package derives from `ExperimentsAbstract` and implements the `runOne` method as follows:

- it looks up the `expe{i}` entry, where `i` is the index of the experiment to run:

this is a struct that contains the index `ida` of an agent and the index `ide` of an environment

- an `EnvironmentTS` is created based on the parameters given by `env{ide}`
- a tree search algorithm is created based on this environment, and on the type ('Random', 'UCT', 'BAST' or 'GPTS') and parameters given by `agent{ida}`
- the tree search algorithm is run for the number of iterations specified by the parameter of `expe{i}`
- the cumulative reward obtained by the tree search algorithm is saved to `expe{i}.runs{nruns+1}.perf` where `nruns` is the previous total number of runs of this experiment

The class constructor defines some agents, environments, and experiments based on these. It then launches several runs of all experiments (through `runAll`, which calls `runOne`). We show below an example of how a `ToyTS.Experiments` object (`Ex`) is displayed after the experiments have been run:

```
>> Ex

Agents:
------------

1: Random
2: UCT
3: BAST with gamma=0.5
4: GPTS with gamma=0.5 and s_n=2
5: GPTS-red @(t)log(t) with gamma=0.5 and s_n=2


Environments:
------------

1: TS, B=5, D=10, s_n=2, gamma=0.5, offspringSum and rewardSum
 * Expe 4: 100 iterations;  Agent 4  -> mean perf: 111.4734 (100 runs)
 * Expe 2: 100 iterations;  Agent 2  -> mean perf: 78.932 (100 runs)
 * Expe 3: 100 iterations;  Agent 3  -> mean perf: 76.6744 (100 runs)
 * Expe 5: 100 iterations;  Agent 5  -> mean perf: 70.2699 (100 runs)
 * Expe 1: 100 iterations;  Agent 1  -> mean perf: 57.7434 (100 runs)
```

On this example, we see that 5 different algorithms (Agents 1 to 5) were tried on one tree search problem (Environment 1), which resulted in 5 different experiments that are listed below the information on Environment 1. Each of these experiments was run 100 times and consisted of 100 iterations of the agent. The experiments are sorted by decreasing mean performance of the associated agent interacting with Environment 1.

5.2 Content-based image retrieval

5.2.1 Introduction: the Pinview system

The goal of the PinView European research project is to develop a "proactive Personal Information Navigator that allows retrieval of multimedia - such as still images, text and video - from unannotated databases" (pinview.eu). Such an information retrieval system is therefore content-based, and needs to exploit relevance feedback given by its user during a search session in order to infer what he is looking for. The objective is to understand the intent of the user as quickly as possible, in order to present relevant documents as early as possible in the search. Here, we will consider the case of image retrieval, and of feedback given in the form of pointer clicks (the user clicks on images relevant to his search). Other types of feedback can be considered, such as implicit eye movement feedback for instance. A more in-depth review of the Pinview system can be found in Auer et al. (2010a).

We consider a filtering task, where a user wants to find a set of images relevant to his query. The feedback is received as clicks, which give rewards of 1, and -1 for no click. Images can be presented to the user one by one, or, more realistically, collages of images can be presented on a web page. In the first case, the feedback is said to be *immediate*: we receive feedback from image t before we choose image $t+1$ to present. In the second case, the feedback is said to be *delayed* : if we present collages of N_c images, we must choose image $t+2$ without receiving the feedback for the image picked at $t+1$, and so on, we must choose image $t+N_c$ without receiving the feedback at $t+1, t+2, \ldots, t+N_c-1$. The performance of an algorithm that selects which images to present to the user, will be measured by its *average precision*, a standard information retrieval metric which encodes our preference for algorithms that present relevant documents early in the results. We first define the precision of an algorithm at time t as the total number of relevant documents found up to time t, divided by t. The average precision is the sum of the precisions at time t going from 1 to the total number of documents shown T, divided by T. We assume that the algorithms to be compared always show the same number of images, for all queries (which is probably not the case in practise).

Presenting relevant images early to the user requires learning his intent and utilising this learning to optimise the selection of images. There is a tradeoff between exploration and exploitation that motivates the use of bandit algorithms. Indeed, Auer et al. (2010b) use the LinRel algorithm to select images to present to the user, and force it to select a new image at each time step.[5] LinRel also showed the best performance in the experiments carried out by Auer et al. (2011), when compared to other algorithms. The algorithm models the true relevance of an image as a function f of the feature representation of this image. When feedback is received from the user, (noisy) observations of f are made. When the feedback is immediate, it selects the next image to be presented as the one with highest upper confidence value among all the images that have not been selected yet. When the feedback is delayed and we must choose N_c images to present, selecting the N_c images with highest f_t values may lead to collages that consist of similar images. In order to remedy this and show a variety of images that make us learn more from the user, one strategy is to duplicate the current instance of the bandit algorithm, then to pick the image with highest upper confidence value, simulate feedback by incorporating this new image with its predicted relevance (as given by the model's reward estimate) to the training set, and so on N_c times.[6]

5.2.2 Experimental setup

We stick to the setup of Auer et al., unless otherwise indicated, and the immediate feedback mode.

Dataset We use a subset of the data of the VOC'2007 challenge, consisting of 2501 labelled images, belonging to 20 categories. We define search queries based on each category: the relevance of an image for the 'aeroplane' task is 1 if this image belongs to that category, -1 otherwise. We use the two-dimensional Self Organising Maps

[5]This forced exploration is necessary, otherwise, the best possible cumulated reward could be attained by simply always playing the first relevant image that we find.

[6]Note that the procedure is particularly slow for picking the first N_c images to present: they will involve the learning from "fake" observations (which has a non-negligible computational cost), but in the end they will be picked at random.

representations of 11 feature extraction methods used in the PicSOM system originally introduced by Laaksonen et al. (2001), which makes up $n = 22$ features in total, and we normalise the feature vectors individually.

Measure of performance We fix the maximum number of images to be seen to $T = 150$. Thus, the computations won't take too long, but it is also reasonable in practise as users of an image search tool are expected to see at most a number of images of this order. It is important to trade exploration and exploitation efficiently because we will only be able to show the user 6% of the whole dataset. Also, even if two algorithms find the same number of relevant images, a better tradeoff will show interesting images earlier in the search. As we said previously, performance is measured by average precision. However, the performance of an algorithm may be affected by the first, randomly selected image: we are more lucky if we select a relevant image first. For this reason, we run each algorithm 100 times and report the average results.

5.2.3 Methods and results

We aim to reproduce the results of Auer et al. by running LinRel on the image retrieval problem for a fixed number of iterations T, and to compare its performance to that of GPB. Note that the choice of a regression model, rather than a classification model, is motivated by the fact that other types of feedback could be continuous (eye movement feedback, for instance). The regression model of GPB is the same as LinRel's. The GP model, however, makes extra assumptions on the nature of the distribution of the rewards. As we have seen in Section 2.1.3.4, β_t can be replaced by a constant c when T is fixed in advance. Let us rewrite the arm selection formulae in GPB and LinRel in order to highlight the similarities between the two algorithms:

$$\begin{aligned}
\mathbf{x}_{t+1} &= \operatorname{argmax}_{\mathbf{x}} \mathbf{k}_t^{\mathsf{T}}(\mathbf{x})\mathbf{C}_t^{-1}\mathbf{y}_t + c\sqrt{\kappa(\mathbf{x},\mathbf{x}) - \mathbf{k}_t^{\mathsf{T}}(\mathbf{x})\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x})} \text{ for GPB} \\
\mathbf{x}_{t+1} &= \operatorname{argmax}_{\mathbf{x}} \mathbf{k}_t^{\mathsf{T}}(\mathbf{x})\mathbf{C}_t^{-1}\mathbf{y}_t + c\sqrt{\mathbf{k}_t^{\mathsf{T}}(\mathbf{x})\mathbf{C}_t^{-1}\mathbf{C}_t^{-1}\mathbf{k}_t(\mathbf{x})} \text{ for LinRel}
\end{aligned}$$

5.2.3.1 Model

We adapt the model and hyper-parameter settings of Auer et al.. The same values should be used by LinRel and GPB, because they perform the same regression, therefore their models of f should coincide. The Gaussian kernel with width $s = 1$ was found to be preferable. For both algorithms, we use an ISO-SE kernel, which is equivalent to a Gaussian kernel but is more suited to the GP model as it allows us to model signal variance. For GPB we choose a signal variance of 0.25, which corresponds to $s_f = 0.5$ and a 95% confidence that the f values are in $[-1, 1]$. LinRel, however, knows that they are in $[-1, 1]$ but has no knowledge on how they are distributed within this interval. The values of the exploration terms are all $\beta_0 s_f$ initially with GPB, and they decrease where we reduce uncertainty.

Noise/regularisation term Scaling the kernel products by s_f^2 has no impact on LinRel when also adapting the noise level: let us write $\kappa' = s_f^2 \kappa$, so that $\mathbf{k}' = s_f^2 \mathbf{k}$ and $\mathbf{C}'_t = \mathbf{K}'_t + s'^2_{\text{noise}} \mathbf{I}_t$ where $\mathbf{K}' = s_f^2 \mathbf{K}$ and $s'_{\text{noise}} = s_f s_{\text{noise}}$. We have $\mathbf{C}'_t = s_f^2 \mathbf{C}_t$ and thus $\mathbf{k}'^{\mathsf{T}}_t(\mathbf{x}) \mathbf{C}'^{-1}_t \mathbf{y}_t + c \left|\left|\mathbf{k}'^{\mathsf{T}}_t(\mathbf{x}) \mathbf{C}'^{-1}_t\right|\right| = \mathbf{k}^{\mathsf{T}}_t(\mathbf{x}) \mathbf{C}^{-1}_t \mathbf{y}_t + c \left|\left|\mathbf{k}^{\mathsf{T}}_t(\mathbf{x}) \mathbf{C}^{-1}_t\right|\right|$. Auer et al. take $s_f = 1$ and $s_{\text{noise}} = 1$ which means that if $s_f = 0.5$ here, we should take $s_{\text{noise}} = 0.5$.

5.2.3.2 Confidence term selection

The confidence term used by Auer et al. was 0.05, hence we should double this value for LinRel, since we scaled rewards from $[0, 1]$ to $[-1, 1]$ here. The exploration terms are constructed differently with GPB, so we need to tune the GPB confidence parameter independently. The noise level, the default SE width and the confidence term are common to all queries, so they can be tuned by running the algorithm with different parameter values on a given task for which we know in advance the labels of the images. We take the 'aeroplanes' task in order to choose GPB's confidence term: we know which are the relevant images for this specific task, and thus we can assess the performance of GPB run with different values of c. We perform 100 runs with immediate feedback.

Table 5.1 shows that the UCB arm selection criterion doesn't significantly improve over greedy arm selection (corresponding to $c = 0$). Indeed, there are little differences between the results observed for c from 0 to 0.5. It may be safer, however, to have an exploration term that is not 0, and for this reason we choose $c = 0.1$ for both LinRel and GPB.

	$c = 0$	0.05	0.1	0.2	$c = 0.5$	1
LinRel	$54.17 \pm 0.50\%$	$\mathbf{55.04} \pm 0.49\%$	$54.51 \pm 0.50\%$	$54.76 \pm 0.41\%$	$54.48 \pm 0.40\%$	$50.14 \pm 0.77\%$
GPB	$54.52 \pm 0.45\%$	$54.52 \pm 0.42\%$	$\mathbf{54.71} \pm 0.39\%$	$54.58 \pm 0.48\%$	$53.94 \pm 0.52\%$	$52.65 \pm 0.51\%$

Table 5.1: *Average precision (in %) of* GPB *and* LinRel *with immediate feedback for the 'aeroplanes' task, with several values of the confidence term c. 150 images shown, out of a total of 2501. Squared Exponential kernel with width* $s = 1$ *and scaling factor* $s_f^2 = 0.25$*; noise variance* $s_{\text{noise}} = 0.5$*. Results averaged over 100 runs. The numbers after the* $\pm$ *correspond to the relative standard error.*

5.2.3.3 Learning query-dependent kernels: Multiple Kernel Learning vs Maximum Likelihood

From one query to the other, we can expect some of the image features to be more relevant than others. In other terms, the similarities between images depend on what the user is looking for. This is always different and we cannot know in advance what the ideal kernel is for a new query. We can model the relative importance of each feature by considering an ARD-SE kernel. With no additional prior knowledge, the widths along each of the n dimensions of the feature space should all be the same and equal to s as chosen previously – which gives us the previous ISO-SE kernel. When receiving feedback, we progressively learn what the user is looking for and, in the GP framework, we can adjust the parameters of the kernel by maximising the likelihood of the observed data. In our experiments, our implementation of GPB updates the hyper-parameters s_i and s_f of its kernel every 15 iterations.[7] This is based on the GPML toolbox (Rasmussen, 2010) which uses gradient descent-based techniques in order to minimise minus the log likelihood of the data, starting the search from weights initialised to 1 and signal variance initialised to 0.25.

[7] s_{noise} stays fixed, however.

5.2.3.4 Results

In Table 5.2 we report the performance of LinRel, GPB and GPB with hyper-parameter learning every 15 iterations (GPB-h), as sample-means of their average precision computed over 100 runs. For each sample-mean value, we also report the relative standard error, i.e. the standard deviation of that mean divided by the latter and expressed as a percentage. We show in **bold** which of the average precision of the 3 algorithms was best. When all algorithms were worse than random, we show the random value in bold too.

All algorithms were run in the 'online2' mode. The experiments were run in parallel on four Intel Xeon cores at 3GHz and took two days. The 'online2' mode has the smallest cost for this application since N is larger than T. Our baseline for comparison is the average precision of a random algorithm, which we can show is equal to the proportion of relevant images in the dataset. As another baseline, Auer et al. have compared the use of the Gaussian kernel in LinRel to that of the polynomial and the linear kernels, and they have shown an improvement of around 0.5 and 1 point respectively.

5.2.3.5 Discussion

As we said before, the regression technique is identical for both algorithms, but the exploration is different because different confidence intervals are used. With GPB, we take advantage of extra assumptions on f and we can expect better confidence intervals if these assumptions are reasonable. But in this application, the two algorithms are practically the same, since the exploration term is low. Indeed, there are no significant differences between the overall average precisions reported for LinRel and GPB. GPB-h performed significantly better on some of the difficult tasks: bus, motorbike and pottedplant. However, it was far from being systematically better than the other two, as we could have hoped. This shows that hyper-parameter learning is challenging in the bandit setting and illustrates the problem raised by Bull (2011) of the algorithm's convergence when using the Maximum Likelihood estimators in that setting. Moreover, hyper-parameter learning is expensive as it takes a significant amount of compute time to recompute $\mathbf{K}$, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ every time new values are learnt.

Regarding the probabilistic model we used, we did not encode the fact that f takes

	Random	LinRel	GPB	GPB-h
bicycle	4.88	**11.43** $\pm$ 5.00%	9.86 $\pm$ 5.61%	9.04 $\pm$ 4.20%
bird	7.28	14.07 $\pm$ 2.18%	**14.14** $\pm$ 2.78%	13.91 $\pm$ 3.61%
boat	3.48	**24.54** $\pm$ 2.26%	23.99 $\pm$ 2.12%	13.64 $\pm$ 5.37%
bottle	**6.12**	3.91 $\pm$ 5.02%	**4.60** $\pm$ 5.02%	4.11 $\pm$ 5.40%
bus	**4.00**	0.00	0.15 $\pm$ 99.75%	**2.68** $\pm$ 7.77%
car	16.07	**33.86** $\pm$ 2.03%	33.21 $\pm$ 1.61%	26.18 $\pm$ 3.21%
cat	6.64	11.25 $\pm$ 3.30%	**11.45** $\pm$ 2.04%	9.33 $\pm$ 5.03%
chair	11.28	13.71 $\pm$ 5.29%	**14.25** $\pm$ 5.97%	10.68 $\pm$ 6.09%
cow	2.84	6.19 $\pm$ 1.84%	**6.59** $\pm$ 1.76%	5.89 $\pm$ 4.29%
diningtable	5.20	**6.51** $\pm$ 5.07%	5.95 $\pm$ 2.68%	6.25 $\pm$ 3.82%
dog	8.40	12.79 $\pm$ 2.24%	**13.58** $\pm$ 1.97%	11.41 $\pm$ 2.93%
horse	5.76	13.22 $\pm$ 3.75%	**14.17** $\pm$ 4.15%	12.77 $\pm$ 6.23%
motorbike	4.92	6.36 $\pm$ 5.24%	5.97 $\pm$ 5.12%	**9.53** $\pm$ 4.25%
person	36.15	46.83 $\pm$ 1.57%	**47.37** $\pm$ 1.48%	42.42 $\pm$ 1.93%
pottedplant	**6.12**	1.24 $\pm$ 16.58%	1.07 $\pm$ 17.54%	**4.29** $\pm$ 6.11%
sheep	1.96	**6.78** $\pm$ 4.42%	6.33 $\pm$ 3.31%	2.63 $\pm$ 9.52%
sofa	7.52	**7.79** $\pm$ 3.98%	7.17 $\pm$ 2.28%	6.67 $\pm$ 4.62%
train	5.12	11.35 $\pm$ 1.99%	**11.70** $\pm$ 2.23%	8.53 $\pm$ 6.38%
tvmonitor	5.76	9.51 $\pm$ 3.65%	**9.92** $\pm$ 3.28%	6.69 $\pm$ 6.47%
overall	7.87	12.70 $\pm$ 2.14%	**12.71** $\pm$ 2.13%	10.88 $\pm$ 2.19%

Table 5.2: *Average precision (in %) of* LinRel, GPB, *and* GPB *with hyper-parameter learning, for all tasks except 'aeroplanes'. 150 images shown, out of a total of 2501, and feedback given immediately.* $s = 1$, $s_{\text{noise}} = 0.5$, $s_f = 0.5$, $c = 0.1$ *for both* GPB *and* LinRel. *Results averaged over 100 runs. The numbers after the* $\pm$ *correspond to the relative standard error.*

values between -1 and 1. For this, it could be useful to observe and learn the logit[8] of $(f+1)/2$, which has values in all of $\mathbb{R}$ when f has values in $[-1, 1]$. Besides, the GP prior with zero mean does not encode the fact that we actually expect most images to be irrelevant (f value close to -1), and that the closer to 1, the less likely reward values would be.

We did not study the impact that delayed feedback had on the performance of the bandit algorithms, as this was already investigated by Auer et al. (2010b). The solution they propose to deal with delayed feedback requires copying the current bandit instance in order to train a new one with simulated feedback (see the `BAlg.chooseSimulated` method of the Bats toolbox). The mechanism used to copy objects in Matlab is particularly slow, which made it difficult to perform experiments. Finally, note that there are some significant differences with the average precisions reported by Auer et al., in particular in the fact that LinRel is always better than random in their results (which is not the case here).

5.3 Search of synthetic trees

5.3.1 Experimental setup

The tree search environment with which the bandits interact specifies what the children of any given node are – thus defining the tree structure – and how the rewards at leaf nodes are given. As in the theoretical analysis of GPTS, we consider symmetric trees with branching factor B and maximum depth D. To each node x is associated an intermediate reward value, a depth, and an index defined as $(id-1)B+i$ where id is the index of p the parent of x and i is the index of x among the B children of p. The reward value at any given node is chosen once and for all for a given tree search problem, by drawing a sample from $\mathcal{N}(0,1)$.

For a given path, we draw rewards from a Gaussian with mean equal to the (discounted) sum of its nodes' intermediate rewards, and standard deviation equal to s_{noise}. Thus, this setup resembles an MDP planning problem with deterministic transitions, fixed depth, and the mean-reward function verifies the GP assumption with the dis-

[8]The logit function is defined as $logit(x) = \log(x) - \log(1-x)$

counted kernel, or the linear kernel if $\gamma = 1$. Figures 5.1, 5.2 and 5.3 show what such functions look like for different values of the discount factor γ. We do not know in advance what the best mean-reward value f^* is – for this we would need to do an exhaustive search of the whole tree –, hence we cannot measure the regret of an algorithm as we have defined it so far. Instead, we look for the maximum reward y_{max} for all algorithms run on a given problem, and use it to approximate immediate regrets by $y_{max} - y_i$. We also normalise regrets by dividing them by f^*, once again approximated by y_{max}. As a consequence, it is the $\sum_i \frac{y_{max}-y_i}{y_{max}}$ value that we report and refer to as "regret".

In our experiments, for each parameter value for $B, D, \gamma, s_{\text{noise}}$, we fix one instance of the corresponding class of tree search problems, and we evaluate several algorithms on this problem. We average performance measures over several runs (100) because of the stochasticity of each experiment. We could also have decided to run these algorithms on several instances of the same class of tree search problems (characterised by a set of parameter values, and that are discounted sums of independent Gaussians with standard deviation equal to 1), but this would have introduced additional variability, and more runs would have been necessary to reduce this variability.

As default parameter values, we take $\gamma = 1$, so that the intermediate rewards at all levels of the tree have the same importance, and our experiments are not subject to the fact that the first levels could be accidentally "easier" than the other levels. We choose $B = 40$ which is rather large, in order to minimise the chance of getting easy levels where most values are positive (imagine $B = 5$). We choose a small value of s_{noise} $(= 0.1)$ to reduce the variability of our results.

5.3.2 Methods and results

GPTS was run with hyper-parameters set to the true γ and s_{noise} values, and with $\delta = 5\%$. The ρ_d value used in BAST corresponds to the maximum difference between the mean rewards of two paths that have d nodes in common. In our setup, when given d first nodes, the rewards for paths that share these nodes are drawn from a Gaussian with mean equal to the discounted sum of intermediate rewards for these nodes, and standard deviation equal to $\sum_{i=d+1}^{D} \gamma^i = \frac{\gamma^d - \gamma^D}{1-\gamma}$. BAST was run with ρ_d equal to the width of a 95% confidence interval in which the two paths' rewards

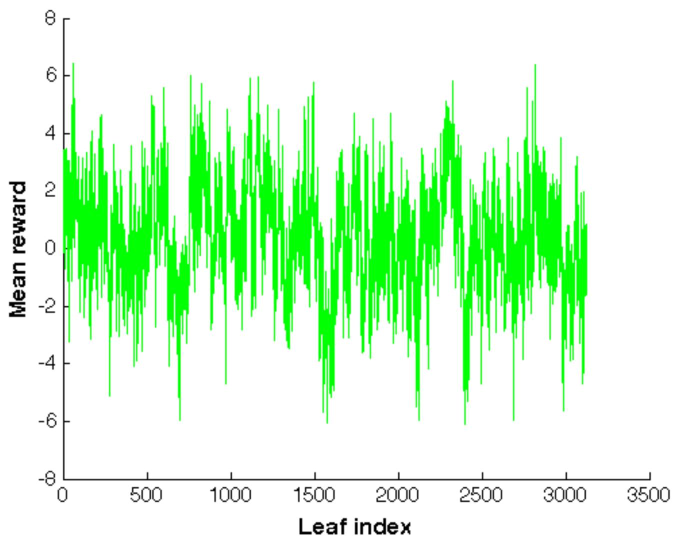

Figure 5.1: *Example of a function on tree paths that is a sum of intermediate values at each node, drawn from independent Gaussians with standard deviation equal to* 1*, for a tree with branching factor* $B = 5$ *and depth* $D = 5$*.*

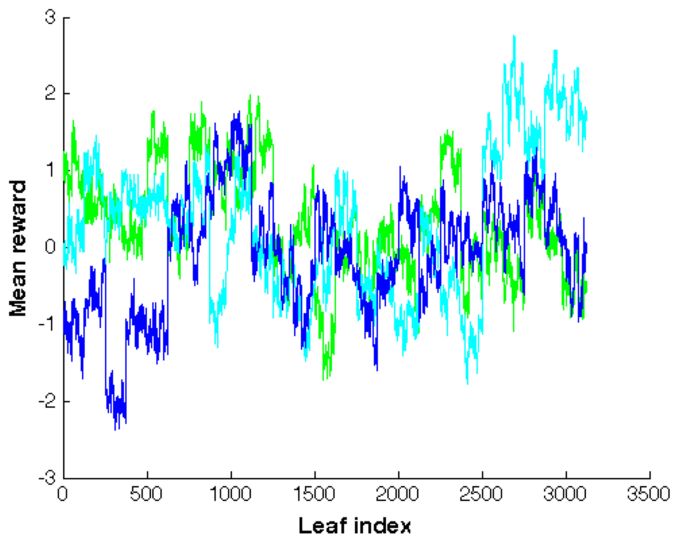

Figure 5.2: *Examples of three functions on tree paths, each of which is a* 0.5*-discounted sum of intermediate values at each node, drawn from independent Gaussians with standard deviation equal to* 1*, for a tree with* $B = 5$ *and* $D = 5$*.*

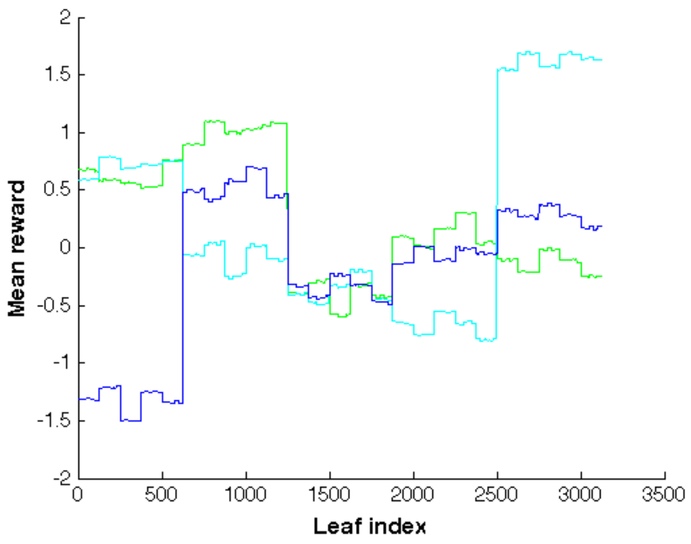

Figure 5.3: *Examples of three functions on tree paths, each of which is a* 0.1*-discounted sum of intermediate values at each node, drawn from independent Gaussians with standard deviation equal to* 1*, for a tree with* $B = 5$ *and* $D = 5$*.*

should be, corresponding to twice the standard deviation of this Gaussian, i.e. 4 times the previous value. Besides, each bandit instance in BAST expects rewards in $[0, 1]$, whereas rewards actually are in $\left[-\frac{2(1-\gamma^D)}{1-\gamma}, \frac{2(1-\gamma^D)}{1-\gamma}\right]$ with a probability slightly greater than 95%. For this reason, we scaled β_t by $\left(\frac{4(1-\gamma^D)}{1-\gamma}\right)^2$ (see Section 2.1.2.1). However, these theoretically good settings for BAST led to mediocre performance in practise, as the algorithm only slightly improved over random. We found that much better empirical results could be obtained when using UCB1 instances at each node of the tree. We call the resulting algorithm BAST1. We also found that UCT performed similarly to BAST1. In summary:

- When $\gamma < 1$:
  - We use GPTS with the γ-discounted kernel.
  - We use BAST1 with $\rho_d = 4\frac{\gamma^d - \gamma^D}{1-\gamma}$, with the β_t expression defined in UCB1 and scaled by $\left(\frac{4(1-\gamma^D)}{1-\gamma}\right)^2$.
- When $\gamma = 1$:
  - We use GPTS with the linear kernel.
  - We use BAST1 with $\rho_d = 4(D - d)$ and β_t scaled by $4D$.

First, we report the performance of GPTS, BAST1, and the random algorithm, after 1000 iterations on a problem with branching factor $B = 200$ and depth $D = 10$ (Figure 5.4). The resulting tree is similar in shape to a typical Go tree search problem (Gelly, 2007).

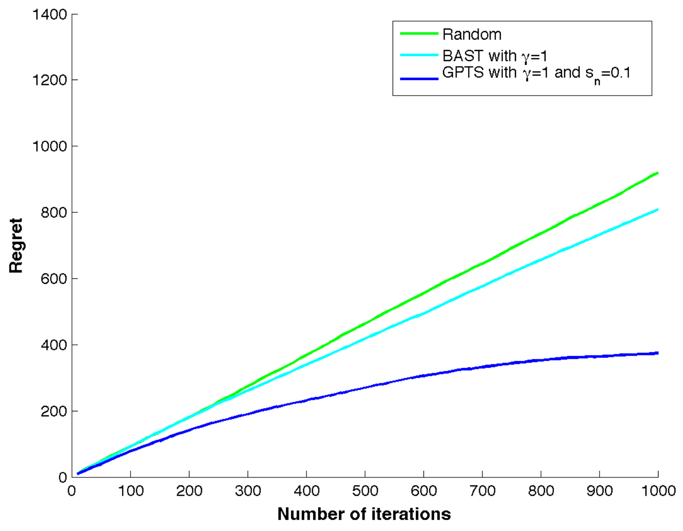

Figure 5.4: *Regret curves of GPTS, BAST1 and the random algorithm, over 1000 iterations. Tree search problem with discount factor* $\gamma = 1$*, branching factor* $B = 200$*, depth* $D = 10$ *and noise variance* $s_{\text{noise}} = 0.1$*. Results averaged over 100 runs.*

Next, we report the performance of GPTS and BAST1 after 100 iterations, for several values of γ. We normalise the reward values so that we can make comparisons between the performances obtained with different γ values. For instance, we would like that our performance measure for the random algorithm stays the same when varying γ. For this, we divide the rewards by D when $\gamma = 1$, or by $\frac{1-\gamma^D}{1-\gamma}$ when $\gamma < 1$. For the random algorithm, we can expect a (normalised) regret equal to the number of iterations, since intermediate rewards will be distributed around 0. Although the sum of rewards of the random algorithm averaged over 100 runs should be close to 0, we have found that it can be in the order of plus or minus 35 after 100 iterations (with B around 75 and D around 15). If it is positive, it means that the tree search problem is rather easy (hence smaller regrets), whereas if it is negative, the problem is rather difficult (hence larger regrets). For easier problems, we can expect y_{max} to be larger, and thus the regret of the random algorithm should be closer to T. Because of these considerations, we choose to report from now on the ratio between the regret and the regret of the random algorithm for each tree search problem. This should remove the effect of easier and harder problems on our performance measures.

Figure 5.5 shows our results when varying the γ parameter. We see that BAST1 has difficulties for $\gamma = 1$ (and for $\gamma = 0.75$ to a lesser extent). In the rest of our experiments, we take $\gamma = 0.5$. We report the performance of GPTS and BAST1 when varying the value of B (Figure 5.6), D (Figure 5.7) and s_{noise} (Figure 5.8), with a default value of $B = 50$.

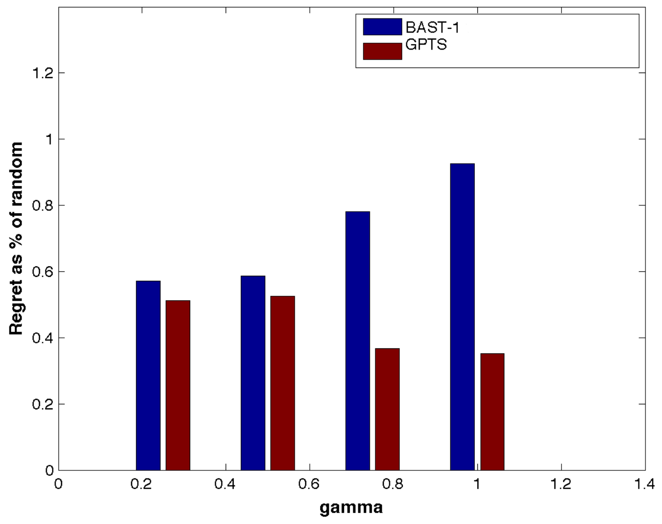

Figure 5.5: *Regrets after 100 iterations of GPTS and BAST1, divided by the regret of the random algorithm, for different values of the discount factor:* $\gamma = 0.25, 0.5, 0.75, 1$. *Tree search problem with* $B = 50, D = 10, s_{\text{noise}} = 0.1$. *Results averaged over 100 runs.*

5.3.3 Discussion

5.3.3.1 Large branching factors

The results presented in Figure 5.4 indicate that GPTS better handles large branching factors. Indeed, UCT and BAST tend to pure exploration for large branching factors, since the initialisation phase requires playing each child of a given node once, before applying the upper-confidence formula. BAST1 suffers from a linear regret, similar to that of the random algorithm, up to time $T = 200$ (equal to the branching factor). This period corresponds to the initialisation phase of the root UCB1 instance: BAST1 has to play all 200 children of the root before being able to make an informed decision on

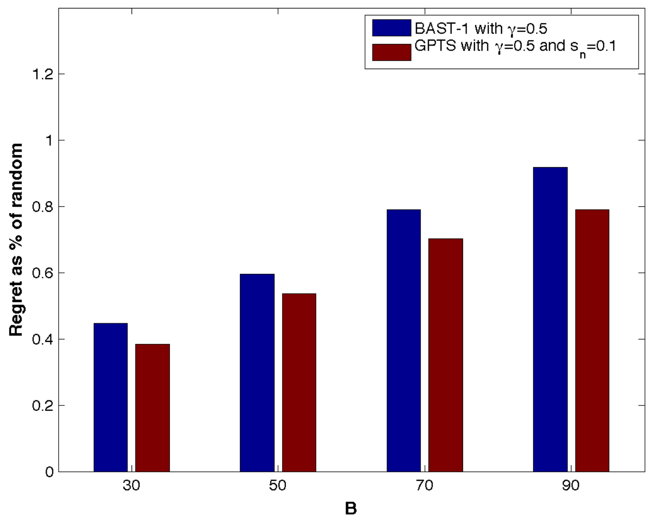

Figure 5.6: *Regrets after 100 iterations of GPTS and BAST1, divided by the regret of the random algorithm, for different values of the branching factor:* $B = 30, 50, 70, 90$*. Tree search problem with* $D = 10, \gamma = 0.5, s_{\text{noise}} = 0.1$*. Results averaged over 100 runs.*

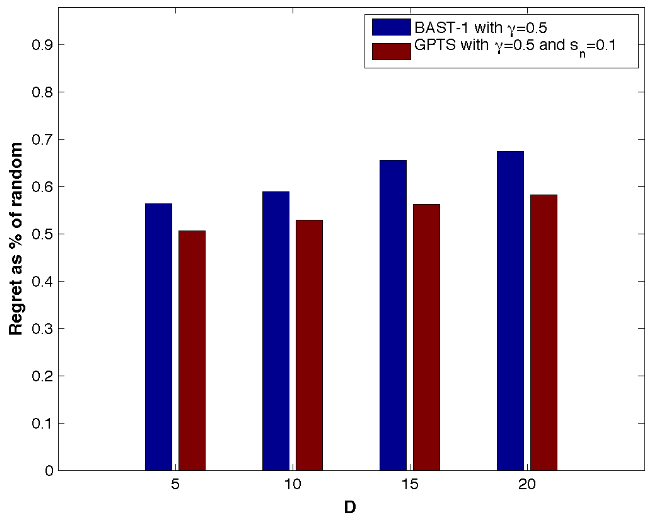

Figure 5.7: *Regrets after 100 iterations of GPTS and BAST1, divided by the regret of the random algorithm, for different values of the maximum depth:* $D = 5, 10, 15, 20$*. Tree search problem with* $B = 50, \gamma = 0.5, s_{\text{noise}} = 0.1$*. Results averaged over 100 runs.*

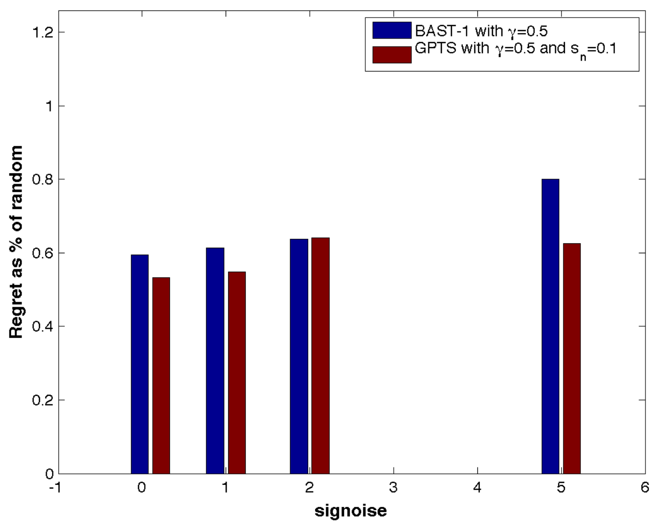

Figure 5.8: *Regrets after 100 iterations of GPTS and BAST1, divided by the regret of the random algorithm, for different values of the noise variance:* $s_{\text{noise}} = 0.1, 1, 2, 5$. *Tree search problem with* $B = 50, D = 10, \gamma = 0.5$. *Results averaged over 100 runs.*

where to explore the tree. However, we may have already found interesting branches of the tree after just a few iterations, and with GPTS we may decide to explore a dummy node which is at a depth greater than 1.

If GPTS finds a good path, it may decide to explore a subtree that branches off that path, rather than to continue exploring all of the root node's children (as BAST does). We can expect that the dummy node that represents this subtree will be somewhere in the middle of the tree: it should be close to the leaf of the previous path for which a good reward was obtained, so that its μ value is high; but it should also be close to the root so that its σ value is high. If the algorithm gets a good reward after selecting a path created from this dummy node, then it can be confident that the early nodes of that path are good (they are in common between the last two selected paths). If it gets a poor reward, then it is likely that the previous path was good because of the last nodes only (and not the ones in common with the newer path). Thus, GPTS can rapidly refine the confidence intervals of its dummy nodes.

UCB1 does not take into account the fact that the rewards are normally distributed (as sums of normally distributed intermediate rewards), but even if it did, the resulting BAST would still run into the same problem of spending a lot of time in the initialisation phases of the bandit algorithms, so we would not expect a significant improvement. We obtained regret curves similar in shape to those of Figure 5.4 when comparing Random, BAST1 and GPTS up to time $T = 100$ on a problem with $B = 20$ and with intermediate rewards drawn uniformly at random, instead of normally, which suggests that the advantage of GPTS persists even when the experimental setting does not exactly match the model's assumptions.[9] Indeed, GPB benefits from theoretical guarantees even when the GP assumption is not verified (see Proposition 4); but, as can be expected, these guarantees get worse the more irregular f is. The advantage of GPTS lies in the fact that, as a single-bandit algorithm, it builds upper confidence bounds for all unexplored subtrees – not only for nodes that have already been explored, as many-bandits algorithms do. Updating all these upper confidence bounds at every time step is costly but they provide more information to better trade exploration and exploitation. Because we learn more about f from each observation, we need less exploration.

5.3.3.2 Impact of the parameters of the tree search problem

As one can expect, the regrets of both algorithms increase with B, D and s_{noise}. GPTS seems to be slightly more robust to noise. The only counter-intuitive result is in Figure 5.5: where BAST1 performs better (relative to random) when the tree search problems are easier, i.e. when γ decreases and f is smoother, GPTS performs worse. We do indeed expect BAST1 to perform better when γ is smaller: the algorithm is likely to make poor choices of nodes to select as it gets closer to the bottom of the tree, because the deeper a node is, the less it has been explored and the less accurate the reward estimates are; but these poor choices only have a small impact on the total rewards, because the intermediate rewards obtained down the tree are further down-weighted when γ decreases.

We saw in Section 4.3.2.4 that the rate of decay of the eigenvalues of $\mathbf{K}$ is higher for smaller γ values. The sum of eigenvalues, however, does not change (it is equal to N), and therefore the largest eigenvalues get even larger when γ decreases. This explains why the GPTS regret upper bound could have a larger value when T is small – although for T big enough the bound would have to improve when the discount factor

[9] It is also interesting to note that Hennig et al. (2010) only reported a "minor decay in performance" of their Bayesian searcher when experimenting with synthetic game trees with intermediate scores generated uniformly at random, instead of normally as in their model.

decreases. Our observations suggest that the regret might also be lower bounded by a sum of expressions involving the eigenvalues of $\mathbf{K}$.

5.3.3.3 Computational aspects

Matlab being an interpreted language, it runs Object Oriented code very slowly (compared to lower-level implementations). This was particularly obvious with BAST1 which should have been much quicker than GPTS – but it was not, probably because its implementation involved as many bandit algorithm objects as there were nodes in the tree.

We had to run 51 experiments 100 times each, hence we could not run the algorithms for an extended number of iterations (100 only for the bar charts, 1000 for the regret curves). The computations took approximately 15 days on four Intel Xeon cores at 3GHz. In practise, for a fixed problem on which the algorithm is run only once, we should be able to perform many more iterations. Besides, based on usual benchmarks, we can expect that an optimised version in C++ or Java would be 20 times faster than its Matlab equivalent. We can also speed computations up with distributed computing.

Note that, owing to the use of dummy nodes in all algorithms (even in BAST), the running time is not affected by B.

6 Conclusions

6.1 Conclusions

6.1.1 Gaussian Processes for bandit problems

This work started with an application of Gaussian Processes to model and handle uncertainty in bandit problems. We showed connections between the resulting algorithm, GPB, and popular bandit algorithms from the literature, namely UCB1 and LinRel. We studied and improved the computational complexity of GPB owing to an online computation "trick" – also applicable to LinRel. We reviewed the work of Srinivas et al. who gave theoretical guarantees on the performance of GPB when the Gaussian assumptions of the model hold, but also when they do not. In the latter case, they showed a regret bound with a constant expressed in terms of the RKHS norm of f for the chosen covariance function, which can be seen as a quantification of the mismatch between the model and the reality.

As the PinView application to image retrieval showed, it can be useful to see certain problems involving the search of large spaces as many-armed bandit problems. An advantage of approaching many-armed bandits with linear regression or Gaussian Processes is that we are able to represent correlations between arms in a precise way, hence, if time allows, we can be smart about the way we explore the search space. We can deal with input spaces that are not continuous subsets of $\mathbb{R}^n$, such as text, as long as we can compute kernel products between inputs. Thus, we could use GPB to recommend text documents (such as news articles) to a user, or to maximise the relevance of text advertisements on the web for each individual user.

With Gaussian Processes, we also benefit from the range of tools available in this framework, for instance to perform classification (for problems with binary rewards), to select model parameters, or to summarise the training set with a smaller set (in order to speed up the inference). If we extend GPB with principled methods that make use of the GP assumption, we should be able to also extend our theoretical analyses in a similar way to the analysis of the version of the GP-EI algorithm that used estimators of the hyper-parameters from observations (Bull, 2011). One promising direction to improve the performance of GPB in the PinView application is, for that matter, to perform Bayesian hyper-parameter selection as proposed by Bull, which would amount to learning an image similarity metric specific to the user's query.

Processing all the arms correlation information has a large computational cost, even after using the specificities of the bandit setting to speed up probabilistic inference. Ranganathan and Yang (2008) suggest that further speed ups may be obtained when using certain kernel functions. In certain problems, the cost of a sample or of an error can be high enough to warrant the computational cost of GP-based algorithms. Large scale recommender systems, however, usually do not consider the content of items as this makes for inefficient algorithms. A partial solution may be to simplify the dependency structure between arms with a tree, which is what Pandey et al. (2007) did with hierarchical clusters of advertisements; we can assign arms to nodes of a tree of coverings of the search space, based on their feature representations.

6.1.2 Many-bandits vs single-bandit approaches to tree search

We developed the idea of using our new bandit algorithm for modelling and handling uncertainty in tree search by simply considering a covariance function between tree paths (seen as arms of a bandit problem), and we established parallels between the resulting model and a generative game-tree model found in the literature (Hennig et al., 2010). We also showed how we could use the GPTS algorithm in the (somewhat restrictive) problem of planning in MDPs with deterministic dynamics and normalised Gaussian rewards. However, due to computational considerations, we were not able to use feature descriptions of states/actions in our covariance function between tree paths: we need the number of possible kernel product values to be small in order to make GPB tractable when the number of arms (tree paths) is large, so that many different paths share the same kernel products with the training data and thus the same upper confidence bounds. We were able to improve the GPB regret bound under the GP assumption by providing a bound on the eigenvalues of the total kernel matrix that exploits the properties of the kernel, and thus problem-independent regret bounds with constants expressed in terms of the kernel parameters and such that smaller upper bounds could be obtained for smoother functions.

The idea of using GPB for tree search can be generalised to using a single many-armed bandit. Our experiments comparing GPTS (single-bandit) to BAST (many-bandits) suggest that, after the same number of iterations on some tree search problems with large branching factors, the cumulative regret for a single-bandit algorithm can be expected to be significantly smaller than for a many-bandits algorithm. This implies that single-bandits gain more from each sample, or in other terms that they have higher sample efficiency. However, many-bandits are more computationally efficient: in the same amount of time, a many-bandits algorithm may build a much larger tree if f is cheap to evaluate, and in the end it may gain more from this large tree – even though it gains less from each sample. In Go for instance, current approaches search thousands of times more nodes than in our experiments. In any case, we can argue that there is a crossover point where the single-bandit approach improves over the many-bandits approach. This is when there is an overly large number of options to choose from: the problem can be made arbitrarily difficult for a many-bandits algorithm by increasing B (which makes it tend to pure exploration, due to the initialisation phase of the bandits), but the options available at any given node may not be that different from each other.

Another situation in which a single-bandit approach is preferable is that of expensive target functions, in which case there is no hope of building large trees. For example, the evaluation of f could be based on a physical experiment, or on a lengthy computer simulation, or on a human evaluation. In recommender systems where items are organised in a taxonomy, the observations of f are based on human inputs, and inefficient exploration-exploitation tradeoffs may make us lose the user. The fact that we only have very few information regarding the user in the beginning (the "cold-start problem") motivates the use of an algorithm with high sample efficiency, i.e. a single-bandit algorithm.

For planning in MDPs, many-bandits algorithms may be preferable when we observe intermediate rewards for each action taken, because single-bandit algorithms do not offer a way to take these intermediate rewards into account. Many-bandits can also handle MDPs with stochastic transitions, whereas it is not straightforward how to extend the single-bandit idea to such MDPs, as we pick sequences of actions and therefore we need to be certain about which sequences are possible.

6.2 Ideas for future work

It is worth trying to improve the computational complexity of GPTS by exploiting the tree structure further, as well as the fact that the covariance matrix to be inverted has large blocks of zero values (when $\chi_D = 0$). Also, we may want to investigate which S function in GPTS-RED would give the best tradeoff between performance and computational cost.

We hope that the results we presented on the use of GP models for bandit problems and tree search will motivate these further studies. Hopefully they will provide groundwork to continue the analysis of the GPB and GPTS algorithms, to investigate other related algorithms, and to evaluate their practical performance on different problems involving the search of large spaces, as suggested in the next section.

We organise ideas for future work in three groups: theory, new algorithms, and example applications.

Theory The theoretical analysis of GPB could be complemented with:

- Regret lower bounds for both the GP and the RKHS settings, in terms of the eigenvalues of the total kernel matrix.
- Regret bounds in the noise-free case.
- Problem-specific regret bounds, that could be obtained by analysing the number of times we play sub-optimal arms.
- Theoretical guarantees on the performance of GPB-red, in terms of the bound S_T on the size of the training set. We hope that by having a continuously growing S_T, the infogain will continue to be bounded by a sub-linear expression of T (otherwise the regret would become linear and we would not be able to prove the no-regret property).

In addition, for GPTS with a given kernel, it may be possible to use the fact that we know the eigenvalues of the total kernel matrix $\mathbf{K}$ exactly, in order to derive a closed-form expression for the m_i values defined in Section 3.2.3.3, i.e. the number of times that the greedy infogain algorithm selects each eigenvector of $\mathbf{K}$ (previously bounded by T). This would impact the greedy infogain bound, and could thus improve the regret bound.

We mentioned in Section 4.4.5 that one way to bound G_T^g is by using the $\log(1+x) \leq x$ inequality. This bound is tighter for smaller $x = s_{\text{noise}}^{-2} m_t \hat{\lambda}_t$ values and using it results in a sum of eigenvalues, which can be bounded by an integral on the strictly decreasing $\hat{l}$ function. The quicker $\hat{l}$ decreases, the smaller the integral, hence the lower the information gain bound. The problem we saw with this technique was that bounding $\log(1 + s_{\text{noise}}^{-2} m_t \hat{\lambda}_t)$ by $s_{\text{noise}}^{-2} m_t \hat{\lambda}_t$ introduces a T factor (the upper bound on m_t), but we hope that this could be replaced with a smaller expression in T.

Then, an even better bound on G_T^g may be obtained by using the sum of log-eigenvalues bound (seen in Section 4.3.3.1) up to a certain time T_*, and the sum of eigenvalues bound for t from $T_* + 1$ to T.

New algorithms Planning in MDPs is closely related to Tree Search. However, an important property of MDPs is that the total reward for a given sequence of actions (a tree path) is a discounted sum of intermediate rewards that can be observed separately, whereas in tree search we only observe rewards for whole paths. In single-bandit algorithms, one way to use the extra information provided by the intermediate rewards that make up the total reward would be to consider all interior nodes as arms of the bandit problem, to add each node on a selected path to the training data, along with the discounted sum of rewards obtained up to this node, and to use a covariance function that would be based on the number of ancestors in common between any two given nodes.

In certain MDP problems, we may assume dependencies between the actions available at any given state. In game playing for instance, similar moves are likely to take us to similar states, from which the chances of winning will be close. According to Gelly and Silver (2011), one of the most interesting lines of research for Upper Confidence-type Tree Search algorithms is to generalise between nodes of the tree. For this, we could imagine a GP-UCT algorithm (and similarly, GP-BAST and GP-OLOP) that would replace UCB1 instances at all interior nodes by GPB instances to which we would give feature representations of actions. We would aim to learn immediate reward functions at each node, as functions of the children's feature representations.

Example applications

- Because we claim that GPTS handles large branching factors very well, the first application that comes to mind is Go AI – indeed, Gelly and Wang (2006) motivated the use of UCT for Go specifically because of the large branching factors of Go game trees. It would be interesting to also see how encoding prior domain knowledge and heuristics in the prior mean (see Stern, 2008) could improve the performance of the algorithm. Besides, if we could model similarities between Go moves/boards through a kernel function, GP-UCT would certainly be an interesting algorithm to try.

- As seen in Section 4.4.6, GPTS can be used for Hierarchical Optimisation and it can thus provide an interesting alternative to HOO if we work with trees of coverings that have large branching factors. It would be interesting to measure the performance of hierarchical methods in comparison with the Bayesian methods that we mentioned in Section 2.4.2 (GPGO and GP-Hedge), for the optimisation of standard test functions such as those considered by Hoffman et al. (2011) and Osborne et al. (2009).

- After having applied GPB in the Pinview setting to binary feedback data, we should apply it to eye-movement feedback measured in $[0, 1]$. Our results showed the potential of hyper-parameter learning but more investigation is required to make it perform consistently better than vanilla GPB – in particular, Bull (2011) showed that using maximum likelihood estimators, as we did, can be problematic in the bandit setting. Then, the Bayesian approach to learn a query-dependent similarity measure between images could be compared to the Multiple Kernel Learning approach.

- Our work on bandit-based tree search was initially motivated by the problem of generating expressive performances of scores of piano music. This can be formalised as a sequence labelling problem where we are given a sequence $\mathbf{m} = (m_1, \dots, m_D)$ of notes and harmonies, and we must come up with a sequence $\mathbf{x} = (x_1, \dots, x_D)$ of performance parameters for each of these notes. One approach to this problem consists in defining a compatibility measure $\Psi(\mathbf{m}, \mathbf{x})$ and, given $\mathbf{m}$, to search $\mathcal{X}$ for the maximum of $\Psi(\mathbf{m}, .)$. In Dorard et al. (2007), the score and its performance are seen as two views of the same object ("the music"), their representations are projected into a common semantic space owing to Kernel Canonical Correlation Analysis, and the compatibility between any given $(\mathbf{m}, \mathbf{x})$ pair is measured by their inner product in that space. The kernels to be used for the two views should account for both local and global similarities between sequences. The maximisation of Ψ can be carried out by choosing the x_i's one at a time in a sequence. Initially, this was done greedily, i.e. x_i was chosen so that it maximises $\Psi\left((m_1, \dots, m_i), (x_1, \dots, x_i)\right)$, but we would expect better results by taking the long term into account – as a game tree search algorithm does when choosing a move to play. Note that each x_i is a set of performance parameters for m_i that can take continuous values. This can be dealt with owing to a discretisation of the space of parameter values and progressive widening in the tree search.

A

Additional material

A.1 Mathematical and probabilistic identities

A.1.1 The Gaussian distribution

Density The probability density function of a random variate $\mathbf{x} \in \mathbb{R}^n$ that follows a Gaussian distribution with mean vector $\mathbf{m}$ and covariance matrix Σ is given by:

$$p(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n \det \mathbf{\Sigma}}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{m})^\mathsf{T}\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{m})\right) \tag{A.1}$$

Entropy From the above we derive that the entropy of $\mathbf{x} \sim \mathcal{N}(\mathbf{m}, \mathbf{\Sigma})$ is given by:

$$H(\mathbf{x}) = -\mathbb{E}(\log p(\mathbf{x})) = \frac{1}{2}\log((2\pi e)^n \det \mathbf{\Sigma}) \tag{A.2}$$

Complementary error function

$$\operatorname{erfc}(z) = \frac{2}{\sqrt{\pi}}\int_z^\infty \exp(-u^2)du \tag{A.3}$$

For $z = \frac{a}{\sqrt{2}}$ this is the probability that $x \sim \mathcal{N}(m, \sigma)$ is outside of $[m - a\sigma, m + a\sigma]$. The complementary error function can be upper-bounded as follows:

$$\operatorname{erfc}(z) \leq \exp(-z^2) \tag{A.4}$$

Conditional Gaussian Assuming the following joint probability distribution:

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mathbf{m_x} \\ \mathbf{m_y} \end{pmatrix}, \begin{pmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^\mathsf{T} & \mathbf{B} \end{pmatrix} \right)$$

we have:

$$\mathbf{x}|\mathbf{y} \sim \mathcal{N}(\mathbf{m_x} + \mathbf{C}\mathbf{B}^{-1}(\mathbf{y} - \mathbf{m_y}), \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\mathsf{T}) \tag{A.5}$$

A.1.2 Probabilities

Law of total covariance If X, Y, Z are random variables on the same probability space, and the covariance of X and Y is finite, then:

$$\operatorname{cov}(X, Y) = \mathbb{E}(\operatorname{cov}(X, Y|Z)) + \operatorname{cov}(\mathbb{E}(X|Z), \mathbb{E}(Y|Z)) \tag{A.6}$$

Azuma's inequality Let $(Y_\tau)_\tau$ be a martingale difference sequence, i.e. for all τ, $\mathbb{E}(Y_\tau|Y_1, \ldots, Y_{\tau-1}) = 0$. Assume that the range of these random variables is bounded in $[-1, 1]$, then for all $\epsilon > 0$:

$$\mathbb{P}\left(\sum_{\tau=1}^{n} Y_\tau \geq \epsilon \right) \leq \exp\left(-\frac{2\epsilon^2}{n} \right) \tag{A.7}$$

Hoeffding's inequality Let S_n be the sum of n independent random variables $(X_\tau)_{1 \leq \tau \leq n}$ with the same distribution P with mean m and range bounded in $[0, 1]$. Hoeffding's inequality can be seen as a special case of Azuma's inequality with $Y_\tau = X_\tau - m$ and $Y_\tau = m - X_\tau$:

$$\mathbb{P}(nm \geq S_n + \epsilon) = \mathbb{P}(nm \leq S_n - \epsilon) \leq \exp\left(-\frac{2\epsilon^2}{n} \right)$$

from which we get:

$$\mathbb{P}(m \geq \hat{m}_n + \epsilon) = \mathbb{P}(m \leq \hat{m}_n - \epsilon) \leq \exp(-2\epsilon^2 n) \tag{A.8}$$

where $\hat{m}_n$ is a random variable that represents an empirical average of n samples from P.

A.2 UCB1 regret bound

In this section, we give proofs of the lemmas invoked in Section 2.1.2.2.

Lemma 1 We start by introducing the following notation: if an event e is true, the expression $\{e\}$ takes the value 1; otherwise, it takes the value 0.

$$\begin{aligned}
\nu(i,T) &= \sum_{t=2}^{T}\{i_t = i\} \\
&= \sum_{t=2}^{T}\{i_t = i \text{ and } \nu(i,t-1) < l_i(T)\} + \{i_t = i \text{ and } \nu(i,t-1) \geq l_i(T)\} \\
&\leq l_i(T) + \sum_{t=2}^{T}\{i_t = i \text{ and } \nu(i,t-1) \geq l_i(T)\}
\end{aligned}$$

Lemma 2 $i_t = i$ implies, by definition of i_t and of the upper confidence function:

$$\bar{Y}_{i^*,\nu(i^*,t-1)} + \sqrt{\frac{\beta_t}{\nu(i^*,t-1)}} \leq \bar{Y}_{i,\nu(i,t-1)} + \sqrt{\frac{\beta_t}{\nu(i,t-1)}}$$

Note that the $\nu(i,t)$ values are interdependent random variates. We aim at working with independent random variates, and for this we write:

$$\min_{0<\nu^*<t} \bar{Y}_{i^*,\nu^*} + \sqrt{\frac{\beta_t}{\nu^*}} \leq \max_{l_i(T)\leq\nu_i\leq t} \bar{Y}_{i,\nu_i} + \sqrt{\frac{\beta_t}{\nu_i}}$$

This implies that $\bar{Y}_{i^*,\nu^*} + \sqrt{\frac{\beta_t}{\nu^*}} \leq \bar{Y}_{i,\nu_i} + \sqrt{\frac{\beta_t}{\nu_i}}$ for a certain value of ν^* and ν_i.

Lemma 3 Let us assume that f^* is strictly below its upper confidence bound and $f(i)$ is strictly above its lower confidence bound. Using the result from the previous lemma, there exist ν^* and ν_i such that:

$$f^* < \bar{Y}_{i^*,\nu^*} + \sqrt{\frac{\beta_t}{\nu^*}}$$

$$\bar{Y}_{i,\nu_i} < f(i) + \sqrt{\frac{\beta_t}{\nu_i}}$$

$$\bar{Y}_{i^*,\nu^*} + \sqrt{\frac{\beta_t}{\nu^*}} \leq \bar{Y}_{i,\nu_i} + \sqrt{\frac{\beta_t}{\nu_i}}$$

Our aim is to show that $f^* \geq f(i) + 2\sqrt{\frac{\beta_t}{\nu_i}}$ is not true:

$$\begin{aligned}
f^* &< \bar{Y}_{i^*,\nu^*} + \sqrt{\frac{\beta_t}{\nu^*}} \\
&< \bar{Y}_{i,\nu_i} + \sqrt{\frac{\beta_t}{\nu_i}} \\
&< f(i) + \sqrt{\frac{\beta_t}{\nu_i}} + \sqrt{\frac{\beta_t}{\nu_i}}
\end{aligned}$$

References

Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876–1902, 2009. ISSN 03043975. doi: 10.1016/j.tcs.2009.01.016. URL http://linkinghub.elsevier.com/retrieve/pii/S030439750900067X. 30

Peter Auer. Using Confidence Bounds for Exploitation-Exploration Trade-offs. *Journal of Machine Learning Research*, 3(3):397–422, March 2003. ISSN 1532-4435. doi: 10.1162/153244303321897663. URL http://jmlr.csail.mit.edu/papers/volume3/auer02a/auer02a.pdf. 32, 35, 36

Peter Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002. URL http://www.springerlink.com/index/L7V1647363415H1T.pdf. 27, 28, 30

Peter Auer, Ronald Ortner, and Csaba Szepesvári. Improved rates for the stochastic continuum-armed bandit problem. *Learning Theory*, pages 454–468, 2007. URL http://www.springerlink.com/index/44011766316x5x77.pdf. 56

Peter Auer, Zakria Hussain, Samuel Kaski, Arto Klami, Jussi Kujala, Jorma Laaksonen, Alex Leung, Kitsuchart Pasupa, and John Shawe-Taylor. Pinview: Implicit Feedback in Content-Based Image Retrieval. *Proceedings of the Workshop on Applications of Pattern Analysis*, 2010a. 130

Peter Auer, Alex Leung, Zakria Hussain, and John Shawe-Taylor. Report on using side information for exploration-exploitation trade-offs. *PinView Deliverable D4.2.1*, pages 1–14, 2010b. URL http://www.pinview.eu/files/pinview-d4-2-1-final.pdf. 131, 132, 133, 135, 137

Peter Auer, Dorota Glowacka, Mehmet Gönen, Zakria Hussain, Samuel Kaski, Jorma Laaksonen, Alex Leung, Shiau Hong Lim, Craig Saunders, and John Shawe-Taylor. Scaling the PinView prototype to millions of images. *Framework*, pages 1–24, 2011. 131

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer New York:, 2006. URL `http://www.library.wisc.edu/selectedtocs/bg0137.pdf`. 37

Eric Brochu, Mike Cora, and Nando de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. Technical report, TR-2009-23, UBC, 2009, 2009. 58, 59

Sébastien Bubeck. *Bandits Games and Clustering Foundations*. PhD thesis, 2010. 24, 31

Sébastien Bubeck and Rémi Munos. Open loop optimistic planning. In *23rd annual conference on learning theory (COLT)*, pages 1–18, 2010. URL `http://sequel.futurs.inria.fr/munos/papers/BM10.pdf`. 48, 53, 86

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. Online optimization in X-armed Bandits. *Advances in Neural Information Processing Systems*, 2009. 56

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. X–Armed Bandits. *Projet*, 20:01, 2010. URL `http://arxiv.org/pdf/1001.4475`. 46, 111

Adam D. Bull. Convergence rates of efficient global optimization algorithms. January 2011. URL `http://arxiv.org/abs/1101.3501`. 60, 61, 135, 148, 153

Olivier Chapelle and Lihong Li. An Empirical Evaluation of Thompson Sampling. In *Proceedings of the 2nd Workshop on the Online Trading of Exploration and Exploitation*, 2011. 60

Pierre-Arnaud Coquelin and Rémi Munos. Bandit Algorithms for Tree Search. (March), 2007a. URL `http://arxiv.org/abs/cs/0703062v1`. 106

Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *Uncertainty in Artificial Intelligence*, 2007b. URL `http://arxiv.org/abs/cs/0703062`. 26, 46, 48, 56, 107, 109, 111

Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. *Proceedings of the 5th international conference on Computers and games*, 2006. URL `http://portal.acm.org/citation.cfm?id=1777826.1777833`. 48

Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. In *Computer Games Workshop*, number June, 2007. URL `http://hal.inria.fr/inria-00149859/`. 107

T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley Interscience, 1991. 75

D. D. Cox and S. John. SDO: A statistical method for global optimization. *SIAM*, 1997. 60

Varsha Dani, T.P. Hayes, and Sham M Kakade. Stochastic linear optimization under bandit feedback. In *Proceedings of the 21st Annual Conference on Learning Theory (COLT 2008)*, pages 355–366, 2008. 36

Louis Dorard, David R Hardoon, and John Shawe-Taylor. Can style be learned? A machine learning approach towards 'performing' as famous pianists. 2007. URL `http://eprints.pascal-network.org/archive/00003541/`. 153

Louis Dorard, Dorota Glowacka, and John Shawe-Taylor. Gaussian Process Modelling of Dependencies in Multi-Armed Bandit Problems. In *Proceedings of the 10th International Symposium on Operational Research - SOR'09*, number x, Nova Gorica, Slovenia, 2009. ISBN 9781595937933. 19, 73

M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. Technical report. 114

Sylvain Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go.* PhD thesis, 2007. URL `http://www.citeulike.org/group/5884/article/2990577`. 22, 141

Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, pages 1–33, 2011. URL `http://www.sciencedirect.com/science/article/pii/S000437021100052X`. 152

Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for Monte-Carlo go. In *Twentieth Annual Conference on Neural Information Processing Systems (NIPS)*, 2006. 18, 46, 89, 152

J.C. Gittins and D.M. Jones. A dynamic allocation index for the discounted multiarmed bandit problem. *Biometrika*, 66(3):561–565, 1979. ISSN 0006-3444. doi: 10.1093/biomet/66.3.561. URL `http://www.jstor.org/stable/2335176`. 27, 59

Dorota Glowacka, Louis Dorard, Alan Medlar, and John Shawe-Taylor. Prior Knowledge in Learning Finite Parameter Spaces. *Proceedings of the 14th conference on Formal Grammar*, 2009. 19

Thore Graepel, Joaquin Quiñonero Candela, Thomas Borchert, and Ralf Herbrich. Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, number April 2009, 2010. URL `http://research.microsoft.com/pubs/122779/AdPredictorICML2010.pdf`. 61

Steffen Grünewalder, Jean-Yves Audibert, Manfred Opper, and John Shawe-Taylor. Regret Bounds for Gaussian Process Bandit Problems. In *Proceedings of the 13th International Conference on Articial Intelligence and Statistics (AISTATS) 2010, Chia Laguna Resort, Sardinia, Italy. Volume 6 of JMLR: W&CP 6.*, volume 6, 2010. URL `http://eprints.pascal-network.org/archive/00005869/`. 58, 59

Philipp Hennig, David Stern, and Thore Graepel. Coherent Inference on Optimal Play in Game Trees. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9, pages 326–333, Chia Laguna Resort, Sardinia, Italy, 2010. URL `http://eprints.pascal-network.org/archive/00007205/`. 89, 90, 145, 149

Matthew Hoffman, Eric Brochu, and Nando de Freitas. Portfolio Allocation for Bayesian Optimization. In *27th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011. 59, 60, 153

Robert Kleinberg, Aleksandrs Slivkins, and Eli Upfal. Multi-armed bandits in metric spaces. *Proceedings of the 40th annual ACM Symposium on Theory of Computing*, 2008. URL `http://portal.acm.org/citation.cfm?id=1374475`. 56

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, 2006. URL http://www.springerlink.com/index/d232253353517276.pdf. 46, 53

Risi Kondor. Regularization in Gaussian Processes. 2003. URL http://www.its.caltech.edu/~risi/notes/gp2.pdf. 43

A. Krause and C. Guestrin. Near-optimal nonmyopic value of information in graphical models. In *UAI*, 2005. 75

Jorma Laaksonen, Markus Koskela, Sami Laakso, and Erkki Oja. Self-organising maps as a relevance feedback technique in content-based image retrieval. *Pattern Analysis & Applications*, 4(2):140–152, June 2001. ISSN 1433-7541. doi: 10.1007/PL00014575. URL http://www.springerlink.com/index/10.1007/PL00014575http://www.springerlink.com/index/LQNBNAGNDN4DPUNN.pdf. 132

T. L. Lai and Herbert Robbins. Asymptotically Efficient Adaptive Allocation Rules. *Advances in Applied Mathematics*, 6(1):4—-22, 1985. 30

Daniel Lizotte. *Practical bayesian optimization.* PhD thesis, 2008. URL http://gradworks.umi.com/NR/46/NR46365.html. 58

Daniel Lizotte, Tao Wang, Michael Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression. *Proc. of IJCAI*, pages 944–949, 2007. 60

Jonas Mockus. *The Bayesian approach to global optimization*, volume 5 of *Mathematics and its Applications (Soviet Series)*. Indian Statist. Inst., 1989. ISBN 0792301153. 60

G.L. Nemhauser, L.A. Wolsey, and M.L. Fisher. An analysis of the approximations for maximizing submodular set functions II. *Mathematical Programming*, 14(1):265–294, 1978. URL http://www.springerlink.com/index/H474774824K48231.pdf. 76

Michael Osborne, Roman Garnett, and S.J. Roberts. Gaussian processes for global optimization. In *3rd International Conference on Learning and Intelligent Optimization (LION3)*, number x, pages 1–15, 2009. URL http://www.robots.ox.ac.uk/~mosb/OsborneGarnettRobertsGPGO.pdf. 59, 153

Sandeep Pandey, Deepayan Chakrabarti, and Deepak Agarwal. Multi-armed bandit problems with dependent arms. *Proceedings of the 24th international conference on Machine learning - ICML '07*, pages 721–728, 2007. doi: 10.1145/1273496.1273587. URL `http://portal.acm.org/citation.cfm?doid=1273496.1273587`. 56, 148

Ananth Ranganathan and MH Yang. Online sparse matrix Gaussian process regression and vision applications. *Computer Vision–ECCV 2008*, 2008. URL `http://www.springerlink.com/index/n74u0u060t8v7843.pdf`. 148

Carl Edward Rasmussen. Gaussian Processes for Machine Learning (GPML) Toolbox. *Journal of Machine Learning Research*, 11:3011–3015, 2010. 121, 134

Carl Edward Rasmussen and C K I Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006. URL `http://www.springerlink.com/index/3xevuky2p4272n75.pdf`. 37, 44, 45

Philippe Rolet. *Elements for Learning and Optimizing Expensive Functions*. PhD thesis, 2011. 58

Philippe Rolet, M. Sebag, and Olivier Teytaud. Boosting Active Learning to optimality: a tractable Monte-Carlo, Billiard-based algorithm. *Machine Learning and Knowledge Discovery in Databases*, pages 302–317, 2009. URL `http://www.springerlink.com/index/r7008820k488x214.pdf`. 107

John Shawe-Taylor and Nello Cristianini. *Kernel methods for pattern analysis*. Cambridge University Press, 2004. 35, 44

Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. *ICML*, December 2010. URL `http://arxiv.org/abs/0912.3995`. 19, 59, 60, 61, 63, 65, 75, 82, 83, 147

David Stern. *Modelling Uncertainty in the Game of Go*. PhD thesis, 2008. 152

Csaba Szepesvári. *Algorithms for Reinforcement Learning*, volume 4. January 2010. doi: 10.2200/S00268ED1V01Y201005AIM009. URL `http://www.morganclaypool.com/doi/abs/10.2200/S00268ED1V01Y201005AIM009`. 51, 52

John N. Tsitsiklis. A short proof of the Gittins index theorem. In *Decision and Control, 1993., Proceedings of the 32nd IEEE Conference on*, pages 389–390. IEEE,

2002. ISBN 0780312988. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=325122. 27

Yizao Wang, Jean-Yves Audibert, and R Munos. Algorithms for infinitely many-armed bandits. *Advances in Neural Information Processing Systems*, pages 1–9, 2008. 56