Bandit Algorithms and Tree Search (Bats) toolbox

Louis Dorard **louis@dorard.me**
University College London
Department of Computer Science
London WC1E 6BT, U.K.

Editor:

Abstract

The Bats toolbox for Matlab provides implementations of popular bandit algorithms (UCB, LinRel, GPB) and bandit-based tree search algorithms (UCT, BAST, HOO, GPTS), under common interfaces. These algorithms are presented in the first section of this technical report, while the second section explains the functioning of the toolbox.

Keywords: Bandits, Tree Search, Upper Confidence Bounds, Gaussian Processes, Matlab, Object Oriented Programming

1. Background

We briefly review the algorithms implemented in the Bats toolbox. We will refer to the Equations and notations introduced here, in Section 2. The reader is invited to consult the suggested references for more information on the algorithms.

1.1 Bandit algorithms

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 samples are iid (identically and independently distributed). 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 $\mathbf{f} = (f(1),...f(N))$ 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, and to exploit arms with high averages.

$$\mu_t(i) = \hat{\mathbb{E}}P_i = \frac{1}{\nu(i,t)} \sum_{\tau=1}^{t} \delta_{i,i_\tau} y_\tau$$

where $\nu(i,t)$ is the number of plays of arm i up to time t. We write $\boldsymbol{\mu}_t = (\mu_t(1),...\mu_t(N))$. 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 $\mathbf{f}$.

Once a policy has been chosen, we denote by $(I_t)_t$ the stochastic process that corresponds to the sequence of chosen arms at all time steps (these are random variables). We denote by $(i_t)_t$ a realisation of $(I_t)_t$, and by y_t the sequence of observed rewards. We also write $\mathbf{y}_t$ the vector of concatenated reward observations up to time t. By definition of f and the

fact that rewards are iid, we can write $y_t = f(i_t) + \epsilon_t$ where ϵ_t is a martingale difference sequence that we call the noise sequence.

1.1.1 UCB1

Reward estimates $\mu_t(i)$ and 'uncertainty measures' $\sigma_t(i)$ are maintained for each arm. We write $\boldsymbol{\mu}_t = (\mu_t(1), ...\mu_t(N))$ and $\boldsymbol{\sigma}_t = (\sigma_t(1), ...\sigma_t(N))$. $\mu_t(i)$ is, as previously, the empirical average of the rewards observed for i. The expression for $\sigma_t(i)$ is chosen 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. Under the reasonable assumption that the support of the P_i reward distributions is in $[-1, 1]$, the Chernoff-Hoeffding inequality bounds the probability that $f(i)$ is further away from its empirical average than a given distance:

$$P(|f(i) - \mu_t(i)| \geq a) \leq 2\exp(-\frac{2a^2}{n}) \tag{1}$$

The arm to be played at each time step is the one for which the upper bound of the confidence interval is the highest. This is a popular strategy which implements the principle of "Optimism in the Face of Uncertainty". The UCB1 algorithm 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 (which is equivalent to say that their support is in $[-1, 1]$). The proposed algorithm is frequentist. 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. This is done by being 'optimistic' and making the algorithm play at time t the arm which maximises the value of the upper confidence function

$$f_t(i) = \mu_t(i) + \sqrt{\beta}_t\sigma_t(i)$$

The UCB1 algorithm is as follows:

- Initialisation:
  - Play each arm once
  - Define $\boldsymbol{\mu}_N$ and $\boldsymbol{\sigma}_N$ from observed data $(i_1, y_1)...(i_N, y_N)$
  - $t = N$
- 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 $\boldsymbol{\sigma}_{t+1}$
  - $t = t + 1$

UCB1 achieves optimal regret growth-rate by choosing:

$$\beta_t = 2\log(t) \tag{2}$$

$$\sigma_t^2(i) = \frac{1}{\nu(i,t)} \tag{3}$$

1.1.2 LinRel

We write $\mathbf{a}_i$ the feature vector of arm i, and we model the mean reward function f as a linear function in the space of arms $\mathcal{X} = \{\mathbf{a}_1, ...\mathbf{a}_N\}$. Again, the reward support is assumed to be bounded in $[-1, 1]$. We write $\mathbf{x}_t = \mathbf{a}_{i_t}$, we denote by training data at time t the set $\{(\mathbf{x}_1, y_1), \dots (\mathbf{x}_t, y_t)\}$, and we write $\mathbf{y}_t$ the vector of training outputs. LinRel (Auer, 2003) adopts the same policy as UCB but defines its confidence intervals differently. First, it defines $\mu_t(\mathbf{x}) = \mathbf{w}_t^\mathsf{T}\mathbf{x}$ for all $\mathbf{x} \in \mathcal{X}$, where $\mathbf{w}_t$ is the Least Squares weight vector estimated from the training data at time t. Thus, the reward estimate can be written as a weighted sum of previous rewards: $\mu_t(\mathbf{x}) = \boldsymbol{\alpha}_t(\mathbf{x})^\mathsf{T}\mathbf{y}_t$. If the y_τ's were independent variables, the variance of this estimate would be bounded by $||\boldsymbol{\alpha}_t(\mathbf{x})||^2 /4$, and we could also apply the Azuma-Hoeffding inequality to determine an upper confidence bound for $\mathbf{w}_t^\mathsf{T}\mathbf{x}(t)$ of the form $\boldsymbol{\alpha}_t(\mathbf{x})^\mathsf{T}\mathbf{y}_t + \sqrt{\beta_t}\,||\boldsymbol{\alpha}_t(\mathbf{x})||^2$. 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.

The kernelised and regularised version of LinRel is given by:

$$\begin{aligned}
\boldsymbol{\mu}_t(\mathbf{x}) &= \mathbf{k}_t^\mathsf{T}(\mathbf{x})\mathbf{C}_t^{-1}\mathbf{y}_t \\
\boldsymbol{\sigma}_t(\mathbf{x}) &= \left|\left|\mathbf{k}_t^\mathsf{T}(\mathbf{x})\mathbf{C}_t^{-1}\right|\right| \\
\beta_t &= \log(2Nt/\delta) \text{ or constant}
\end{aligned}$$

where $\mathbf{k}_t(\mathbf{x})$ is the vector of kernel products between $\mathbf{x}$ and the training inputs, and $\mathbf{C}_t$ is the covariance matrix between the training inputs.

From one play, we learn about all arms. The reward estimates and uncertainty measures need to be updated for all arms. While UCB 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 $\boldsymbol{\sigma}_{t+1}$
  - $t = t + 1$

1.1.3 GPB

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 rewards

when always playing the same arm, as Gaussian noise. If arms have feature vectors in $\mathbb{R}^d$, for example, the covariance function can be chosen to be a squared exponential whose smoothness is adjusted to fit the characteristic length scale which is assumed for f – but more generally, we do not need explicit feature representations of arms. Note that in the case of a finite number of arms, the GP prior on f is equivalent to an N-variate Gaussian prior on $\mathbf{f}$:

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

The GP posterior at time t has same mean $\mu_t(\mathbf{x})$ as LinRel (GP regression and least squares regression are equivalent) and variance $\sigma_t^2(\mathbf{x}) = \kappa(\mathbf{x}, \mathbf{x}) - \mathbf{k}_t(\mathbf{x})^\mathsf{T} \mathbf{C}_t^{-1} \mathbf{k}_t(\mathbf{x})$. The GPB algorithm, as introduced by Dorard et al. (2009), works in the same way as LinRel – but with different confidence interval widths:

- 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 $\boldsymbol{\mu}_{t+1}$ and $\boldsymbol{\sigma}_{t+1}$
  - $t = t + 1$

Srinivas et al. (2010) give an upper regret bound for GPB with high probability, that relies on the fact that the f values lie between their lower and upper confidence bounds. If $\mathcal{X}$ is finite, this happens with probability $1 - \delta$ if:

$$\beta_t = 2 \log(\frac{|\mathcal{X}| \, t^2 \pi^2}{6\delta})$$

1.2 Bandit-based 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 $D \geq 1$ and branching factor $B \geq 2$, and the observation of which can be noisy. We look for the node 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. 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 (in game tree search for instance), or when a maximum number of iterations has been specified (this is referred to as "fixed horizon" exploration, by opposition to "anytime"). 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.

1.3 Many-bandits algorithms: Upper Confidence Trees

Path selection as a sequence of bandit problems Many-bandits tree search algorithms use bandit problems at each non-leaf 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, 2007 and Bubeck et al., 2010, but B is already reserved here for the branching factor). 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. The first many-bandits tree search algorithm was proposed by Kocsis and Szepesvári (2006) and called Upper Confidence Trees, as it used UCB instances. Gelly and Wang (2006) reported that UCT performed significantly better than other approaches for searching Go game trees.

Measure of performance A Tree Search algorithm's performance can be measured, as for a bandit algorithm, by its cumulative regret $R_T = Tf^* - \sum_{t=1}^{\mathsf{T}} f(n_t)$. 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^*.

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 *depth-first* 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*, used by Coulom (2006) for Go tree search: the current iteration is stopped after creating a new node (or reaching a maximum depth); a reward is obtained as a function of the visited path (not necessarily of depth D), or as a function of a randomly completed path of length D. The resulting tree is asymmetric and contains paths that have different numbers of nodes. Hopefully this 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 at a depth smaller than D and not creating nodes that would belong to sub-optimal paths, thus growing the tree asymmetrically.

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

Despite the good performances of UCT on Go, Coquelin and Munos (2007) 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, and proposed a revised definition of B to overcome this.

We extend the definition of f to all nodes: we set f on any non-leaf 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 doesn't apply. However, if we can relate the value of f at a given node i to its value at leaf nodes $j \in \mathcal{L}(i)$ in the branch i, through a smoothness assumption on f, we will be able to derive true confidence bounds. We assume that 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 means that, the more ancestors in common between two leaves, the closer their f values will be. We thus have:

$$\begin{aligned} f(i) &\leq \frac{1}{\nu(i,t)} \sum_{j \in \mathcal{L}(i)} \nu(j,t)(f(j) + \rho_d) \\ &\leq \mu_t(i) + f(i) - \mu_t(i) + \rho_d \\ &\leq \mu_t(i) + \sqrt{\frac{\beta_t}{\nu(i,t)}} + \rho_d \end{aligned}$$

with high probability, where $\beta_t = 2\log(\frac{2\bar{N}t(t+1)}{\delta})$ and $\bar{N} = \frac{B^{D+1}-1}{B-1}$ is the total number of nodes in the tree.

Indeed, for all i and for all t, Azuma's inequality applied to the martingale difference sequence sum

$$\frac{1}{\nu(i,t)} \sum_{\tau, \mathbf{x}_\tau \in \mathcal{L}(i)} f(\mathbf{x}_\tau) - y_\tau \leq f(i) - \mu_t(i)$$

states that the probability that it is bigger than $\sqrt{\beta_t}\nu(i,t)$ is smaller than δ. With $\beta_t = 2\log(\delta^{-1})$, this gives $f(i) \leq f_t(i) + \rho_d$ with probability $1 - \delta$. 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.

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 could also have benefited from true upper confidence bounds for non-leaf nodes 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. The Bandit Algorithm for Smooth Trees constructs its U-values in order to benefit from these two ways of constructing upper confidence bounds, and thus to get tighter bounds:

$$U(i) = \min\{f_t(i) + \rho_d, \max_{j \text{ child of } i} \{U(j)\}\} \tag{4}$$

BAST is parameterised by ρ_d and can therefore adapt to different levels of smoothness of the reward function. It can be shown that UCT is a particular case of BAST corresponding to $\rho_d = 0$.

1.5 Gaussian Processes for Tree Search

The GPTS algorithm (Dorard and Shawe-Taylor, 2010) considers tree paths as arms of a bandit problem. The number of arms is $N = B^D$ (number of leaves or number of tree paths). A path $\mathbf{x}$ is given by a sequence of nodes : $\mathbf{x} = x_1, \ldots, x_{D+1}$ where x_1 is always the

root node and has depth 0. We consider the feature space indexed by all the nodes x of the tree and defined by

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

The dimension of this space is equal to $\bar{N}$. The linear kernel in this space simply counts the number of nodes in common between two paths – and the more nodes in common, the closer the rewards of these nodes should be. We could model different levels of smoothness of f by considering a Gaussian kernel in this feature space and adapting the width parameter. The kernel κ could also be derived from the model assumptions. For instance, in a γ-discounted MDP problem where the actions at any given state are all independent and their intermediate rewards are assumed to be Gaussians:

$$\kappa(\mathbf{x}_1, \mathbf{x}_2) = \frac{1 - \gamma^{2\phi(\mathbf{x}_1)^\top \phi(\mathbf{x}_2)}}{1 - \gamma^2}$$

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 the large number of arms – as for most tree search applications. At time t we look for the path $\mathbf{x}$ that maximises $f_t(\mathbf{x})$. When $\kappa(\mathbf{x}, \mathbf{x})$ has the same value for all $\mathbf{x}$, 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 of kernel products with the arms in training, $\mathbf{k}$; all the paths that go through the same unexplored sub-tree share the same $\mathbf{k}$, and there are $O(t)$ maximum unexplored sub-trees.

2. The toolbox

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 next section of this chapter.

The use of the Object Oriented (OO) features of the Matlab language allows 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 to share code between algorithms that have things in common. For instance, in our framework, UCB-1 is an implementation of a UCB-type algorithm; LinRel and GPB are kernel extensions of such algorithms and only differ from each other by a few lines of code; UCT is based on instances of UCB-1; GPTS is based on an instance of GPB.

In OO 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 introduced in this thesis. 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. It can be downloaded at http://louis.dorard.me/bats, along with unit tests and the code for the applications considered in this work (CBIR and toy tree search). The Bats toolbox is released under a GPL license.

We denote properties, methods, functions and classes with `this typeface`; class names are capitalised.

2.1 Bandit problems

2.1.1 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 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.

2.1.2 Bandit algorithms

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

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 2.2.2 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`, which 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);
```

2.1.3 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 + beta(t) .* sqrt(V)` and deals nicely with cases where some `V` values are equal to infinity. 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 UCB-1 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 (1.1) and (3). `updateBeta` defines `beta` as specified in Equation (2). This expression doesn't involve any parameter, and as a consequence, `delta` is left unspecified in the constructor.

2.1.4 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 **??**, 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 they are the same for all kRR

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

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 (**??**)) 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 (**??**) and (**??**) for GPB). The `Q` property is updated according to Equation (**??**) which is implemented in the `updateQ` method. We don't support the removal of data from training.
- In the online1 mode, we don't 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 (**??**)) 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 isn't represented in the bandit algorithm's set of arms, based on its feature representation. This is the case, for instance, with the GP-UCT algorithm (see Section 2.2.3). 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 to provide 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 doesn't 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`.

2.1.5 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 $\boldsymbol{\alpha}_{t+1}$ the vector of $\alpha_{t+1}(\mathbf{x})$ values for all arms in $\mathcal{X}$. We also write $\mathbf{Q}_t$ the matrix of $\mathbf{q}_t(\mathbf{x})$ vectors for all arms. For GPB-online1:

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

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

For GPB-online2:

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

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 `U` values given by the different versions, 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.

2.2 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 'depth-first' 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.

2.2.1 Tree representation

The `Tree` class implements a tree structure which, essentially, is a set of nodes indexed in $[1, n]$ where n 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 sub-trees.

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 don't have feature representations, and their entries in `features` are columns of NaN values – 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.

```
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 non-leaf 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 doesn't affect the `U` value of the parent.

A new method called `trainBandit` allows 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 (4). 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.

2.2.2 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[3]. 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`.

2.2.3 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 doesn't exist yet), and this until reaching the maximum depth (in 'depth-first' 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` wasn't 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 the HOO algorithm (Bubeck et al., 2010) 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'.

GP_UCT The `GP_UCT` class wasn't included in the current version of the toolbox, but it would also derive from `ManyBanditsTS` and its `BTree` would contain GPB instances

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

(`tree.bType=='GPB'`). When calling `next` at a leaf node or at a node that has a dummy child, we would call `e.offspring` to list all possible children (even those not in the tree structure yet) and we would return the one with highest `U` value as given by the `predictU` method of the parent's GPB instance, after having created it if it wasn't in the tree structure yet.

2.3 The Experiments framework

The empirical evaluation of the performance of a bandit algorithm requires to perform 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 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 to select 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.

2.3.1 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 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.

2.3.2 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 is displayed:

```
>> 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)
```

References

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. 3

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. 2

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. 5

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. XArmed Bandits. *Projet*, 20:01, 2010. URL http://arxiv.org/pdf/1001.4475. 5, 16

Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *Uncertainty in Artificial Intelligence*, 2007. URL http://arxiv.org/abs/cs/0703062. 5

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. 5

Louis Dorard and John Shawe-Taylor. Gaussian Process Bandits for Tree Search. *Arxiv preprint arXiv:1009.0605*, pages 1–33, 2010. URL http://arxiv.org/pdf/1009.0605. 6

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. 4

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)*, 2006. 5

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, 2006. URL http://www.springerlink.com/index/d232253353517276.pdf. 5

Carl Edward Rasmussen. Gaussian Processes for Machine Learning (GPML) Toolbox. *Journal of Machine Learning Research*, 11:3011–3015, 2010. 12

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. 4