Feature Extraction for Policy Search

Feature Extraction für Policy Search
Bachelor-Thesis von Sandra Christina Amend aus Hanau
Mai 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Feature Extraction for Policy Search
Feature Extraction für Policy Search

Vorgelegte Bachelor-Thesis von Sandra Christina Amend aus Hanau

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Dr. Gerhard Neumann

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 28. Mai 2014

(Sandra Christina Amend)

Thesis Statement

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, May 28, 2014

(Sandra Christina Amend)

Abstract

Features for reinforcement learning are usually handcrafted for specific tasks. The reliance on handcrafted features, however, limits the autonomy of robots and other learning agents. We therefore investigate learning features for policy search methods. We focus on the relative entropy policy search algorithm (REPS). REPS requires features for representing the value function, as well as the policy. The features are learned using extremely randomized trees (extra trees). The trees define a partitioning of the state space, which is transformed into a suitable feature representation. The proposed method was evaluated using a simulated pool task. The tree based approach was able to learn policies from scratch for a limited region of the state space. Due to the tree structure, the learned policies were capable of targeting multiple pockets.

Contents

List of Figures

1 Introduction

In the future, robots and other autonomous agents should be able to perform complex tasks in real world environments. Real world environments are, however, high dimensional and can change over time. Therefore, the robot should autonomously adapt to changes in the context of the task. An agent can learn to perform a task through trial and error using reinforcement learning.

In a reinforcement learning setting, the agent is provided with a reward signal that indicates the goal of the task. Based on this reward signal, the agent must learn to choose suitable actions depending on the current state. For continuous real world scenarios, the state is usually abstracted using a set of features. These features indicate similarities between states and, thus, allow the agent to generalize its knowledge to new situations. In most cases, the features are designed for a specific task by a human. This reliance on pre-constructed features limits the autonomy of the agent.

In this thesis, we address this issue and explore the topic of learning features for reinforcement learning. In particular we focus on learning features for policy search methods, which are known to be suitable for learning in high dimensional state spaces. The policy search method we use in this thesis is relative entropy policy search (REPS), which is a state-of-the-art method and has been used for learning motor skills for robots [1, 2]. Policy search methods use features of the state for two key components. The features are used to model the value function, which indicates how good states are. The features are also used to define the agent's policy, which selects actions based on the current state.

We propose using extremely randomized trees (extra trees) in order to learn the features. Extra trees are often used for regression problems, where they learn a mapping from the input space to a one dimensional output. The trees partition the state space such that the variance of the output is small in each partition. The extra tree algorithm builds several trees using a randomized building process. The trees are combined into a forest which merges their individual predictions.

Extra trees have been used to learn the Q-function for performing Q-Iteration. We use the trees to learn features for the value function and policy of REPS. The trees are trained by learning mappings from states to rewards or actions. The resulting partitions of the state space are then used as features. We evaluate our proposed method on a simulated pool task. Pool is a difficult task to learn, as the state space is continuous and the reward function is discontinuous.

In Chapter 2 we give an overview of reinforcement learning and policy search. The REPS algorithm is described in Chapter 3. The standard extra trees are introduced in Chapter 4. The extensions for representing the value function and the policy using extra trees are given in Chapters 5 and 6 respectively. The proposed method was evaluated on a pool task. The results of the experiments are given in Chapter 7.

2 Background

2.1 Learning Scenario

Our learning scenario is based on a method from the reinforcement learning sub-field policy search. Reinforcement learning is based on an agent that learns to achieve a goal in an environment through trial and error. The goal is defined by a reward function that provides feedback to the agent. The agent derives an optimal policy based on the rewards it gains. Policy search methods use a parametric representation of the policy. The policy's parameters are optimized, to obtain an optimal policy, based on the rewards.

2.1.1 Reinforcement Learning

Reinforcement learning [3] is based on the insight that the agent can learn from trial and error. In a standard reinforcement learning scenario, an agent interacts with its environment to receive rewards. Through these interactions, and the provided feedback, it must learn which of the actions lead to high rewards. To be able to find an optimal action, the agent must explore new actions but also exploit the already gained knowledge. Once it has a good estimate over the quality of the actions, it should favor the optimal actions and reduce exploration.

The reinforcement learning scenario can be formalized as a Markov decision process (MDP). In a MDP, an agent is in a state $\boldsymbol{s} \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states. When the agent performs an action $\boldsymbol{a} \in \mathcal{A}$, it transitions to the next state $\boldsymbol{s}'$ and receives a reward $r(\boldsymbol{s},\boldsymbol{a}) \in \mathbb{R}$. The set $\mathcal{A}$ contains all possible actions. The agent chooses the action based on a stochastic policy $\boldsymbol{a} \sim \pi(\boldsymbol{a}|\boldsymbol{s})$, which depends on the state. By performing an action $\boldsymbol{a}$ the agent transitions to the next state $\boldsymbol{s}'$. The distribution over next states is given by the transition probabilities $p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$. The reward function $r(\boldsymbol{s},\boldsymbol{a})$ denotes the immediate reward that is received by the agent for performing action $\boldsymbol{a}$ in state $\boldsymbol{s}$.

Rather than considering an infinite sequence of actions, we are interested in the episodic case. The scenario starts in an initial state $\hat{\boldsymbol{s}}$, which can either be a fixed state or sampled from an initial state distribution. The episode terminates after some predefined condition is fulfilled. This condition can, for example, be time dependent or depend upon reaching a goal state. Given an episode with T time steps, the accumulated reward

$$\mathcal{R}_{\boldsymbol{s}\boldsymbol{a}} = \sum_{i=1}^{T} r(\boldsymbol{s}_i, \boldsymbol{a}_i) \tag{2.1}$$

is given as the sum over all of the rewards the agent obtained during the episode.

Instead of maximizing the immediate reward, the agent should maximize the accumulated reward, which is often called the return. The accumulated reward is closely related to the value function, which is given by

$$V_t^{\pi}(\boldsymbol{s}) = E_{\pi}\left\{ \sum_{i=t}^{T} r(\boldsymbol{s}_i, \boldsymbol{a}_i) \,\middle|\, \boldsymbol{s}_t = \boldsymbol{s} \right\}. \tag{2.2}$$

The recursive formulation of the value function can be written as

$$V_t^{\pi}(\boldsymbol{s}) = E_{\pi}\left\{ r(\boldsymbol{s}_t, \boldsymbol{a}_t) + V_{t+1}^{\pi}(\boldsymbol{s}_{t+1}) \,\middle|\, \boldsymbol{s}_t = \boldsymbol{s} \right\}. \tag{2.3}$$

The value function denotes the expected accumulated reward that can be obtained when the agent starts in the state $\boldsymbol{s}$. Hence, the value function can be seen as a quality measure for the state $\boldsymbol{s}$. If a state has a low immediate reward and a high value, it is still desirable for the agent to transition to the state in order to achieve high rewards in the future. An optimal policy π^* selects at every time step an action that maximizes the expected accumulated reward, which is given by the immediate reward plus the expected value of the next state $V^{\pi^*}(\boldsymbol{s}_{t+1})$. Value function methods, such as Q-learning [4], approximate the value function in order to compute a policy that selects near optimal actions. Approximating the value function is, however, not a trivial task for continuous and high dimensional state spaces [5, 3].

2.1.2 Policy Search

Policy Search is a sub-field of reinforcement learning [6]. In contrast to value function based methods, policy search aims to find an optimal policy directly in the policy space. For continuous state spaces a policy cannot be defined as a look-up table for each individual state, as is often done in finite state spaces. Instead the agent must use a policy that is defined by a finite set of parameters. Hence, a change of the parameters affects the behavior of the agent, and thus also changes the rewards. The goal is to find the parameters of the policy that optimize the reward. The agent may need to learn to perform the same task in different contexts. In these cases, the agent can learn a higher level policy to select suitable parameters depending on the current context. The low level policy would perform the task as before using these parameters. This framework is known as contextual policy search.

Policy search methods are usually local and therefore might not find the global optimum. Gradient based methods, e.g., the natural actor critic algorithm [7], are a common approach to policy search. These algorithms compute the gradient of the return with respect to the parameters based on local samples in the proximity of the current parameter set. Subsequently, they update the parameters by taking a small step in the direction of the gradient. The main problem with gradient based methods is that they are sensitive to the step size. A too big step size might cause a jump over the maximum and it might not converge. If the step size is too small they converge only very slowly. Expectation-maximization methods, e.g., PoWER [8], do not employ a step size. Instead they compute a lower bound on the return and then shift to the maximum of the lower bound. Information-theoretic approaches aim to limit the loss of information when updating the policy. This approach prevents the algorithm from stepping too far away from the previous policy. REPS, as discussed in Chapter 3, is an information-theoretic approach.

2.2 Related Work

Reinforcement learning in large state spaces can quickly become intractable if each state is considered independently, without generalization. State aggregation methods abstract the state space by combining sets of states into meta-states [9, 10]. The states assigned to a meta-state should have similar values to minimize the approximation error. However, the values for the individual states are typically not known, nor can they be computed efficiently. Baras and Borkar propose aggregating states in parallel to learn the value function, by clustering states based on approximated values [11]. Rather than using the value directly, Ren and Krogh combine states with similar immediate rewards and transitions [12]. The parti-game algorithm, is a variable resolution reinforcement learning method. It combines the partitioning of the high dimensional state space with planning and learning [13]. The partitioning of the state space is achieved with a single tree which is refined during learning only in critical areas where the previous partitioning was insufficient. In our method, each tree creates a state aggregation. All the states in one leaf are assigned to one meta-state, as we will explain in Section 4.2.

When using state aggregation, we would still need to learn a parameter for each meta state. A more compact representation of the value function can be learned using linear features [9]. Usually, these features are handcrafted using prior knowledge of the task. There are, however, methods to learn the features from data. A state-of-the-art method is deep belief networks [14]. Deep belief networks often consist of stacks of restricted Boltzmann machines or auto-encoders. First each layer is trained separately to encode its input in an unsupervised manner. The first layer is trained on the real input data. The second layer is then trained on the output of the hidden neurons of the first layer. Each following layer is trained on the hidden part of the previous layer. All layers are then combined to form one network. Finally, the weights in the network are fine-tuned. Lange and Riedmiller use auto-encoder deep belief networks for encoding observations to the feature space to then learn the Q-function [15]. Faulkner and Precup learn a forward model with deep belief networks in order to simulate additional data for model-based reinforcement learning [16]. Mnih et al. combine a convolutional neural network with a Q-learning algorithm to learn control policies for Atari games based on visual data [17].

Ernst et al. compare different tree-based methods for extracting relevant information for the control policy [18]. They also evaluate extra trees in this context. They use fitted Q-Iteration to obtain an optimal policy, in contrast to our approach, where we use a policy search method.

In Chapter 7, we evaluate our method by learning to play pool in a simulator. There have been many approaches to pool playing robots and programs. Pastor et al. use the policy search method PI^2 to learn a pool stroke with a real robot [19]. The goal is to shoot the cue ball in a straight line as fast as possible. Deep Green, developed by Greenspan et al., is aimed to play competitively against human opponents [20]. It plans its shots through a minimax based game tree obtained with the help of a simulator. PickPocket, developed by Smith, is a program that uses an expectimax based game tree to play simulated pool games [21]. To achieve this goal, a set of possible shots is created by using prior knowledge of pool and the current geometric layout of the balls. Its main focus is on playing strategically against an opponent. In our evaluations we try to learn a policy for single shots. The learning algorithm starts with no prior knowledge and has to figure out how to pocket a ball on its own. The initial state is sampled from a state distribution and, as a result, the state can change for each episode.

3 Learning Skills with REPS

This chapter discusses Relative Entropy Policy Search (REPS) [22]. REPS is a Policy Search method that aims to learn an improved policy based on observed data. The main constraint that REPS introduces is the limitation of the loss of information during policy updates. We will use REPS for the episodic case. In the episodic case, the agent is given the initial state at the beginning of an episode. This initial state defines the context of the task. The agent must then select a set of parameters for the policy based on the context before the episode begins. The episode is then executed according to these parameters, and the agent receives the accumulated reward for the entire episode. This accumulated reward is also known as the return, and can be used to learn better parameters for the future.

3.1 Optimization Problem

The goal of REPS is to maximize the expected reward, i.e.

$$\max_{\mu,\pi} \sum_{\boldsymbol{s},\boldsymbol{a}} \mu(\boldsymbol{s})\pi(\boldsymbol{a}|\boldsymbol{s})\mathcal{R}_{\boldsymbol{s}\boldsymbol{a}}, \tag{3.1}$$

where $\mathcal{R}_{\boldsymbol{s}\boldsymbol{a}}$ is the accumulated reward, for performing an action $\boldsymbol{a}$ in a given state $\boldsymbol{s}$, as defined in Section 2.1.1, $\mu(\boldsymbol{s})$ is the state distribution and $\pi(\boldsymbol{a}|\boldsymbol{s})$ is the policy. The main constraint is the limitation of the information loss during the policy updates to avoid deterioration of the policy. Without such a constraint, the policy search algorithm is more prone to getting stuck in a poor local optimum and erratic jumps in the policy space may occur. The old state action distribution $q(\boldsymbol{s},\boldsymbol{a})$ is obtained from the observed samples of the old policy. The new state action distribution $p(\boldsymbol{s},\boldsymbol{a}) = \mu(\boldsymbol{s})\pi(\boldsymbol{a}|\boldsymbol{s})$ is generated by the new policy π that we want to optimize. The resulting constraint that bounds the Kullback-Leibler divergence between $q(\boldsymbol{s},\boldsymbol{a})$ and $p(\boldsymbol{s},\boldsymbol{a})$ is

$$\varepsilon \geq \sum_{\boldsymbol{s},\boldsymbol{a}} p(\boldsymbol{s},\boldsymbol{a}) \log\left(\frac{p(\boldsymbol{s},\boldsymbol{a})}{q(\boldsymbol{s},\boldsymbol{a})}\right). \tag{3.2}$$

REPS optimizes over the joint state action distribution $p(\boldsymbol{s},\boldsymbol{a})$. However, the initial state is determined by the environment and, therefore, the state distribution cannot be modified by the agent. The state distribution should therefore not be changed during optimization. Hence, we need to introduce a constraint for the state distribution $\mu(\boldsymbol{s})$ which ensures that it matches the observed initial state distribution $\mu_0(\boldsymbol{s})$

$$\forall \boldsymbol{s} \ : \ \mu(\boldsymbol{s}) = \mu_0(\boldsymbol{s}). \tag{3.3}$$

This constraint has to be adapted if the state space is continuous. In this case, state features $\boldsymbol{\phi}(\boldsymbol{s})$ are introduced as an abstract description of the initial state. The modified constraint

$$\sum_{\boldsymbol{s}} \mu(\boldsymbol{s})\boldsymbol{\phi}(\boldsymbol{s}) = \hat{\boldsymbol{\phi}}_0 \tag{3.4}$$

matches the feature expectations to the observed feature averages $\hat{\boldsymbol{\phi}}_0$ from the given state distribution. The last constraint

$$\sum_{\boldsymbol{s},\boldsymbol{a}} \mu(\boldsymbol{s})\pi(\boldsymbol{a}|\boldsymbol{s}) = 1 \tag{3.5}$$

ensures that the resulting state action distribution is a valid probability distribution.

This constrained optimization problem can be rewritten using Lagrange multipliers

$$L = \sum_{\boldsymbol{s},\boldsymbol{a}} \mu(\boldsymbol{s})\pi(\boldsymbol{a}|\boldsymbol{s})\mathcal{R}_{\boldsymbol{s}\boldsymbol{a}} + \eta\left(\varepsilon - \sum_{\boldsymbol{s},\boldsymbol{a}} p(\boldsymbol{s},\boldsymbol{a})\log\left(\frac{p(\boldsymbol{s},\boldsymbol{a})}{q(\boldsymbol{s},\boldsymbol{a})}\right)\right) + \boldsymbol{\theta}^{\mathrm{T}}\left(\hat{\boldsymbol{\phi}}_0 - \sum_{\boldsymbol{s}} \mu(\boldsymbol{s})\boldsymbol{\phi}(\boldsymbol{s})\right) + \lambda\left(1 - \sum_{\boldsymbol{s},\boldsymbol{a}} \mu(\boldsymbol{s})\pi(\boldsymbol{a}|\boldsymbol{s})\right). \tag{3.6}$$

The closed form solution of the optimization is

$$p(\boldsymbol{s},\boldsymbol{a}) \propto q(\boldsymbol{s},\boldsymbol{a})\exp\left(\frac{\mathcal{R}_{\boldsymbol{s}\boldsymbol{a}} - \boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{s})}{\eta}\right). \tag{3.7}$$

The new state action distribution $p(\boldsymbol{s},\boldsymbol{a})$ is therefore given by weighted samples of the old state action distribution $q(\boldsymbol{s},\boldsymbol{a})$. The Lagrange multipliers η and $\boldsymbol{\theta}$ are obtained by optimizing the dual function. The parameter η has a similar effect as the temperature parameter in a softmax policy. A large value for η will result in a more uniform weighting of the samples, and the resulting distribution $p(\boldsymbol{s},\boldsymbol{a})$ will be similar to $q(\boldsymbol{s},\boldsymbol{a})$. For a small value of η, only a few samples will have a high weighting. The $\boldsymbol{\theta}$ parameters are a result of the constraint in Equation (3.4). Together with the state features $\boldsymbol{\phi}(\boldsymbol{s})$, they form the baseline value function. The baseline is subtracted from the accumulated reward $\mathcal{R}_{\boldsymbol{sa}}$ to be able to judge the actions without the influence of the initial state.

The Lagrange multipliers η and $\boldsymbol{\theta}$ can be obtained by optimizing the dual function

$$g(\eta,\boldsymbol{\theta}) = \eta\varepsilon + \boldsymbol{\theta}^{\mathrm{T}}\hat{\boldsymbol{\phi}}_0 + \eta \log \sum_{\boldsymbol{s},\boldsymbol{a}} q(s,a) \exp\left(\frac{\mathcal{R}_{\boldsymbol{sa}} - \boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{s})}{\eta}\right). \tag{3.8}$$

The optimal parameter values can be efficiently computed using a gradient based approach [6].

3.2 Policy Update

Having performed the optimization, we now generate a new policy based on the old one. However, REPS only defines a relationship between the old policy and the new policy for the sampled points. Furthermore, the relationship is defined for the joint distributions rather than the policies directly. It therefore does not define a new policy for the entire state space. Instead the parameters of the new policy have to be estimated from the weighted samples given by REPS. The optimization defines a weight for every sample, given by

$$w_i = \exp\left(\frac{\mathcal{R}_{\boldsymbol{sa}} - \boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{s})}{\eta}\right). \tag{3.9}$$

The new policy is obtained by computing a weighted maximum likelihood estimate

$$\max_{\beta} \sum_i w_i \log(\pi(\boldsymbol{a}_i|\boldsymbol{s}_i;\boldsymbol{\beta})), \tag{3.10}$$

where $\boldsymbol{\beta}$ are the parameters of the policy. An example for a weighted maximum likelihood policy update can be found in Section 6.3. Due to the weighting, samples with large accumulated rewards relative to the baseline will have a larger influence on the new policy. Similarly, actions that perform poorly will be less likely to occur in the future.

3.3 Role of Features in REPS

Features are used in two components of episodic REPS. The first component is the policy. Features can be used to create a policy that adapts to different initial states. By using suitable features, the policy can generalize better to new states. Our feature based policy is explained in Chapter 6.

Features are also used to define the value function. The value function defines a baseline for comparing actions. The value parameter θ_i depends on all the samples associated with its feature. Actions with a better performance than the baseline will receive a larger weighting. Our feature generating process, using extra trees, is described in Chapter 5.

4 Trees

Tree based approaches have been used for several years for classification and regression problems [23, 24]. Trees are hierarchical structures consisting of nodes and links between the nodes, called edges. Each node can have several children or none. If a node has no children it is called a leaf node. Each non-leaf node contains a test that determines the next node in the path down the tree when traversing it. The leaf node predicts the final output of the tree.

4.1 Regression Trees and Forests

Regression trees are used to learn a mapping from a multidimensional input $\boldsymbol{x} \in \mathbb{R}^m$ to an output $y \in \mathbb{R}$. A regression tree consists of nodes which are connected in a hierarchical manner. Each node in the tree has a test $[x_d < t]$ and two child nodes ξ_l and ξ_r, which makes it a binary tree. The test consists of the dimension d of the input to check and the threshold t for splitting the data. Each leaf ζ in a tree T corresponds to a region in the input space. Therefore, a tree splits the input space into smaller regions with axes parallel boundaries. All areas defined by the leaves of one tree are disjoint. A visualization of such a partitioning of a two dimensional input space can be found in Figure 4.1 and the corresponding tree in Figure 4.2.

Building a tree based on a given training set can be achieved by different methods. One method is the extremely randomized trees approach, also called extra trees, which is described in Section 4.2.

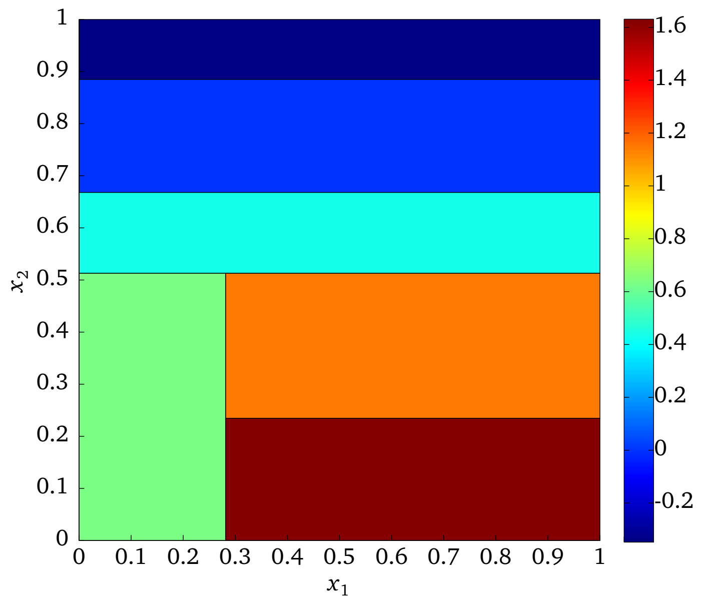


Figure 4.1: Partitioning of the input space based on the regression tree in Figure 4.2.

A forest of trees is an ensemble method. Ensemble methods create several simple models and combine them to obtain a more accurate prediction. In a forest F, each of the trees T corresponds to one model. The forest combines the predictions of all the trees, for example, by averaging over them. The number of trees in the forest is denoted by M. A forest is trained based on a training set $\mathcal{TS} = \{(\boldsymbol{x}_i, y_i) \,|\, i = 1, 2, \ldots, n\}$, containing n inputs $\boldsymbol{x}$ and their respective outputs y, by training each of the trees in the forest individually. The assumption for an ensemble approach is that each tree provides an independent estimate of the output value. Therefore, the trees should not be too similar. To achieve this effect, different approaches can be used, that typically randomize a part of the learning algorithm. For example, bagging creates bootstrap training sets by sampling uniformly with replacement from the original training set $\mathcal{TS}$. Alternatively, the tree building process itself can be randomized [18]. The extra trees method, that we use is, based on the latter approach, as explained in Section 4.2. By averaging over all of the trees, the forest's prediction is robust to the local inaccuracies of individual trees.

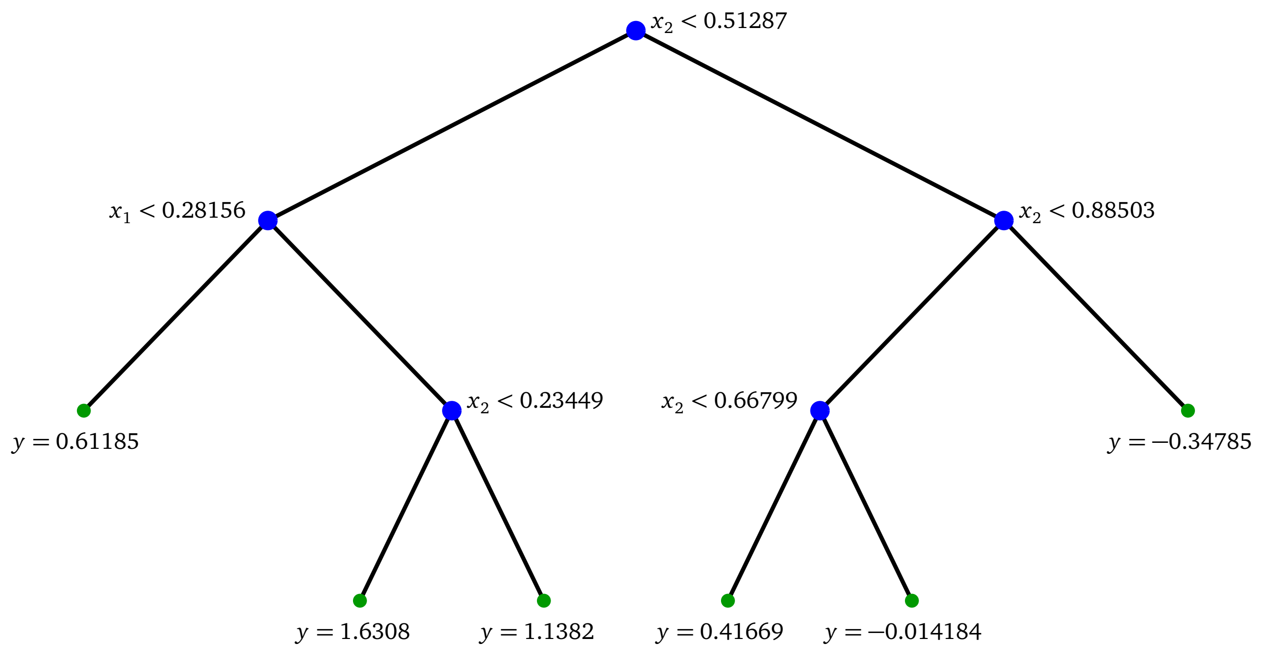


Figure 4.2: Regression tree structure. The non-leaf nodes (blue) are associated with a test, the leaf nodes (green) contain an output.

4.2 Extra Trees

Extremely **Ra**ndomized Trees, as introduced by Geurts et al. [24, 18], are based on randomizing the building process of the individual trees in the ensemble. This section describes how extra trees are commonly used for regression. In Chapters 5 and 6, we will explain how extra trees can be used to learn features for REPS.

4.2.1 Building an Extra Tree Forest

To build an extra tree forest, we need a training set $\mathcal{TS}$ containing inputs $\boldsymbol{x}$ and their corresponding outputs y. Each tree in the forest starts with all of the training data in a single node, known as the root node. In order to split a node into two branches, a set of K tests is randomly generated. Each test has the form $[x_d < t]$ where d denotes the dimension of the cut and t denotes the threshold for splitting the data. The dimension d is chosen randomly for each test, and the threshold t is sampled uniformly in the range of the input values in dimension d. The K tests are then evaluated with the help of a scoring function and the training data $\mathcal{D} = \left\{(\boldsymbol{x}_i, y_i) \,\middle|\, i = 1, 2, \ldots, n_\xi\right\}$ in the node ξ. The test $[x_d < t]$ splits the data set $\mathcal{D}$ into a left part $\mathcal{D}_l = \{(\boldsymbol{x}, y) \,|\, x_d < t \wedge (\boldsymbol{x}, y) \in \mathcal{D}\}$ and a right part $\mathcal{D}_r = \{(\boldsymbol{x}, y) \,|\, x_d \geq t \wedge (\boldsymbol{x}, y) \in \mathcal{D}\}$. As scoring function, the relative variance reduction

$$\operatorname{score}(\mathcal{D}, [x_d < t]) = -\frac{|\mathcal{D}_l| \operatorname{var}(y|\mathcal{D}_l) + |\mathcal{D}_r| \operatorname{var}(y|\mathcal{D}_r)|}{|\mathcal{D}|}, \tag{4.1}$$

where $|\cdot|$ denotes the cardinality of a set, is used. The test with the highest score is assigned to the node and two child nodes are created containing the data $\mathcal{D}_l$ and $\mathcal{D}_r$ respectively. The leaves in the tree are split further as long as they contain more than n_{min} data entries.

4.2.2 Obtaining a Prediction

To obtain an output $\hat{y}$ for an input $\hat{\boldsymbol{x}}$, the tree is traversed node by node. Starting at the root node, $\hat{\boldsymbol{x}}$ is evaluated by the test of the root node. The result of the test determines the path that is taken in the tree. If the test is true, the next node is the left child ξ_l. Otherwise the right child ξ_r is the next node. This process is repeated until a leaf ζ is reached. The leaf ζ is then considered active for the input $\hat{\boldsymbol{x}}$. We also say that ζ contains $\hat{\boldsymbol{x}}$ and write $\hat{\boldsymbol{x}} \in \zeta$.

For each leaf an output value $\tilde{y}$ is stored, which is returned by the tree. The tree output is computed by taking the mean over all data sample outputs of the leaf ζ

$$\tilde{y} = \frac{1}{|\mathcal{D}_\zeta|} \sum_{s \in \mathcal{D}_\zeta} y_s, \tag{4.2}$$

where $\mathcal{D}_\zeta \subseteq \mathcal{TS}$ is the dataset for which ζ is active.

The forest computes its prediction based on the outputs of all of the trees T_i. The prediction is given by the mean over all outputs $\tilde{y}_i$

$$\hat{y} = \frac{1}{M}\sum_{i=1}^{M}\tilde{y}_i. \tag{4.3}$$

This prediction can also be seen as the weighted average over all the training data outputs. A training sample receives a larger weight, if it appears in more active leaves in the forest. The weight of a sample will also increase more if it is in a leaf with fewer data samples. A sample which does not appear in an active leaf receives a weight of zero. An example of how one tree is traversed to obtain a prediction can be found in Figure 4.3.

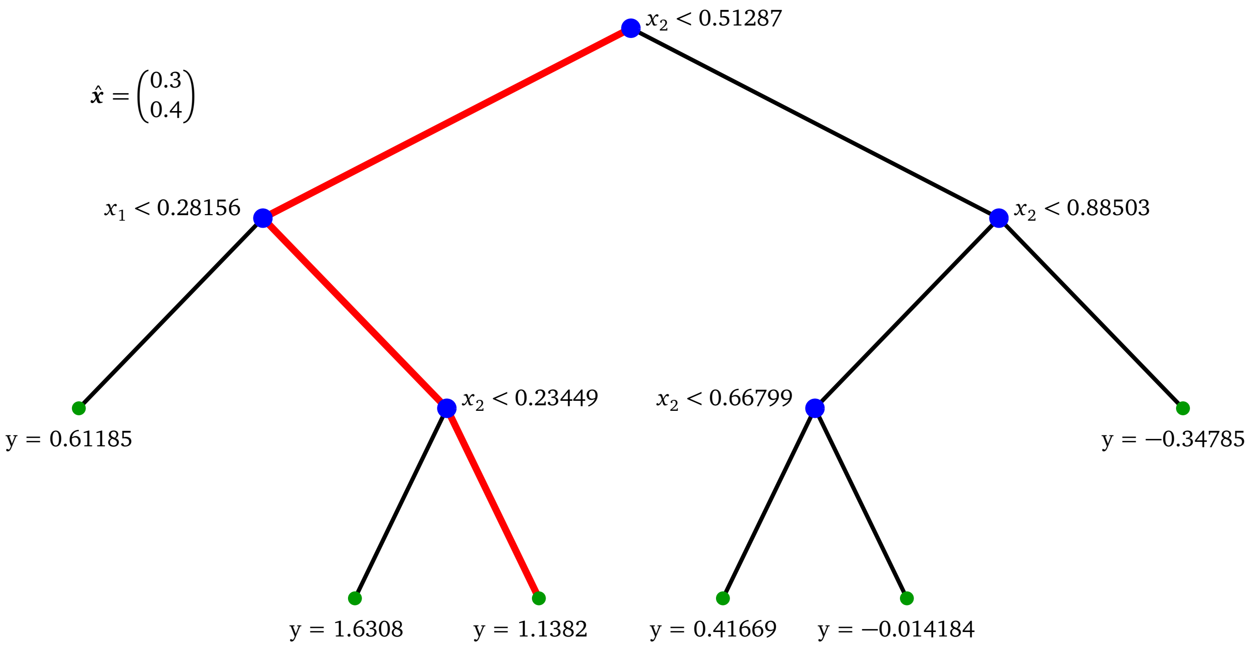


Figure 4.3: The path that is traversed for the input $\hat{x} = (0.3, 0.4)^{\mathrm{T}}$ is marked in red. The predicted output $\hat{y}$ is 1.1382.

4.3 Parameters of the Extra Tree Approach

In this section, we take a closer look at the individual parameters of the extra trees algorithm and explain their effects on the tree building process and the prediction accuracy.

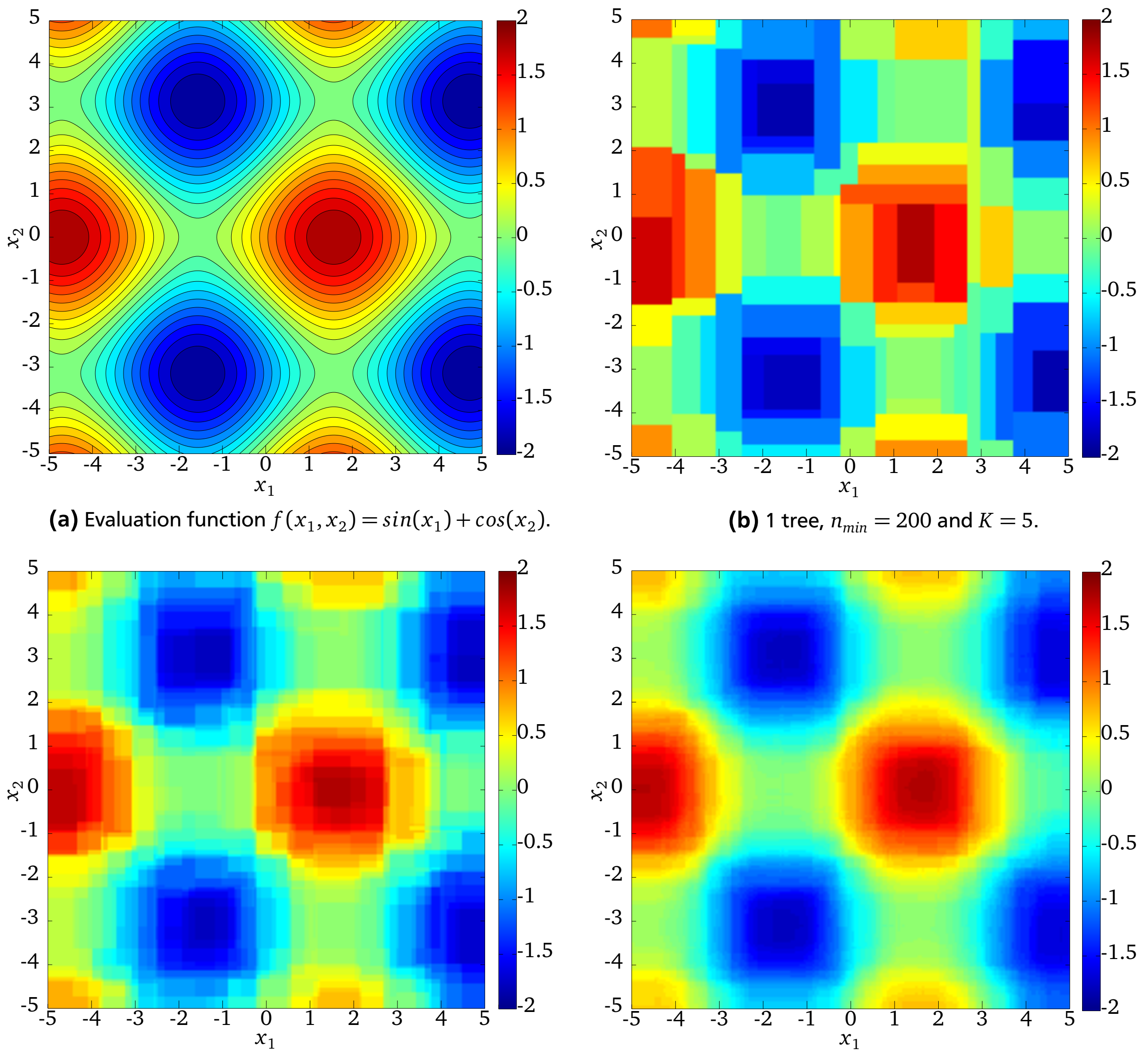
4.3.1 Number of Random Cuts

The number of cuts K denotes the number of cuts to evaluate for each split during the training of the extra trees. Each cut corresponds to a test of the form $[x_d < t]$. For small values of K, the trees in the forest become more random. A very high value for K, leads to very similar trees and in the case of $K \to \infty$, the trees would be identical. The best choice for K depends on the specific task and should be determined through evaluations. Ernst et al. propose the number of input dimensions as a default value for K for regression tasks [18].

Totally randomized trees are the special case of extra trees where $K = 1$. At each split, only one cut is chosen, and, hence, it automatically has the maximum score. As a result, the trees in the forest have very different structures and, do not depend on the outputs in the training set. The structure of the learned trees is independent of the task and is only influenced by the distribution of the input data.

4.3.2 Minimum Size of Leaves

The minimum size of the leaves n_{min} refers to the number of data points, a node needs to contain to be still considered for splitting. This value determines the size of the trees, where a very small value leads to almost fully grown trees. A



(a) Evaluation function $f(x_1, x_2) = sin(x_1) + cos(x_2)$.

(b) 1 tree, $n_{min} = 200$ and $K = 5$.

(c) 5 trees, $n_{min} = 200$ and $K = 5$

(d) 30 trees, $n_{min} = 200$ and $K = 5$

Figure 4.4: Predictions of extra tree forests trained with 10000 random samples from the function given in Subfigure 4.4a. The parameters $n_{min} = 200$ and $K = 5$ were fixed for all of the forests. The prediction is less coarse if more trees are added to the forest.

high number leads to very shallow trees. The extreme case, where n_{min} is equal to the number of training samples $|\mathcal{TS}|$, results in a tree with only one node. Setting n_{min} greater than one is a pre-pruning method. It stops further splitting without building a fully grown tree and therefore reduces the possibility of overfitting. In conclusion, n_{min} influences the coarseness of the partitions that the trees can create for the input space.

4.3.3 Number of Trees

The number of trees M determines how many trees are created for the whole forest during training. To obtain an output prediction from a tree, the outputs of all trees in the forest are averaged. This strategy has a smoothing effect on the output and even if there are some trees that predict outliers they do not have a significant influence on the overall prediction for a reasonable value of M. It is generally better to build more trees, apart from the higher computational cost resulting from many trees. More trees in the ensemble do not lead to overfitting as the results of the trees are averaged. Figure 4.4 provides example predictions of a function obtained from forests with different numbers of trees.

5 Trees as Features

Instead of using extra trees to predict the output for a given input, we will use them to obtain features for REPS. In this use case, the input for the trees are states or state-action pairs and the output are the corresponding obtained rewards. This means that the trees learn an approximation of the value function $V(\boldsymbol{s})$ or of the reward function $r(\boldsymbol{s}, \boldsymbol{a})$.

5.1 Building the Trees

Building the extra trees is the same as how it is described in Section 4.2. The training set $\mathcal{TS}$ contains the inputs and outputs used for training the tree. They are obtained by the sampling performed in REPS. The inputs can either be the states $\boldsymbol{s}$ or the state-action pairs $\langle \boldsymbol{s}, \boldsymbol{a} \rangle$. The output is the corresponding accumulated reward $\mathcal{R}_{\boldsymbol{sa}}$. Hence, we are approximating either the value function $V(\boldsymbol{s})$ in the first case, or the reward function $r(\boldsymbol{s}, \boldsymbol{a})$ in the second. The value function describes what value we expect from the state in general without taking the action that was chosen into account. The reward function on the other hand returns the expected reward for the state $\boldsymbol{s}$ when the action $\boldsymbol{a}$ is chosen. A more detailed description of the two functions can be found in Section 2.1.1.

If the tree is based on the state-action pairs, they are treated as one input. For example, if the state has four dimensions and the action has two dimensions, splitting in the fifth dimension actually means splitting the first dimension of the action.

We introduce an additional parameter n_c which constrains the set of splits that are considered during training. A potential split creating a node with less than n_c data entries is ignored. Hence the resulting tree will have at least n_c data entries per leaf and no more than n_{min}.

5.2 Obtaining the Feature Vector

There are two possible ways to obtain a feature vector for a new state $\boldsymbol{s}$ from a tree. This section describes both methods in detail. We will call the first method “active leaves”, as it creates for each leaf in the forest an entry in the feature vector. The second method is called “active samples”, because the feature vector contains an entry for every training sample.

5.2.1 Active Leaves

In this setting, every leaf in the forest corresponds to exactly one entry in the feature vector $\boldsymbol{\phi}(\boldsymbol{s})$. Each entry in this feature vector will be called a feature $\psi(\boldsymbol{s})$. The forest F consists of M trees T_i, where $i = 1, 2, \ldots, M$. Each of the trees has a number of leaves n_i. The number of leaves is likely to be different for each tree T_i, because of the randomized tree building process. If a leaf ζ is active for the given input $\boldsymbol{s}$ in a tree, the feature is active and set to one. In the other case, when ζ is not active for $\boldsymbol{s}$, the feature is set to zero. We can write this intuition more formally with the help of the indicator function

$$\mathbb{I}(\boldsymbol{s}, \zeta) = \begin{cases} 1 & \text{if } \boldsymbol{s} \in \zeta \\ 0 & \text{if } \boldsymbol{s} \notin \zeta \end{cases}. \tag{5.1}$$

The feature $\psi_j(\boldsymbol{s})$ is the j$^{\text{th}}$ element in the feature vector $\boldsymbol{\phi}_i(\boldsymbol{s})$ for $j = 1, 2, \ldots, n_i$. The feature vector $\boldsymbol{\phi}_i(\boldsymbol{s})$ of the tree T_i is now

$$\boldsymbol{\phi}_i(\boldsymbol{s}) = \left(\psi_1(\boldsymbol{s}), \psi_2(\boldsymbol{s}), \ldots, \psi_{n_i}(\boldsymbol{s})\right)^{\mathrm{T}}, \tag{5.2}$$

where each $\psi_j(\boldsymbol{s})$ is given by

$$\psi_j(\boldsymbol{s}) = \mathbb{I}(\boldsymbol{s}, \zeta_j). \tag{5.3}$$

Thus, the feature vector $\boldsymbol{\phi}(\boldsymbol{s})$ can be obtained from the forest F by concatenating the feature vectors of each tree T_i, i.e.

$$\boldsymbol{\phi}(\boldsymbol{s}) = \left(\boldsymbol{\phi}_1(\boldsymbol{s})^{\mathrm{T}}, \boldsymbol{\phi}_2(\boldsymbol{s})^{\mathrm{T}}, \ldots, \boldsymbol{\phi}_M(\boldsymbol{s})^{\mathrm{T}}\right)^{\mathrm{T}}. \tag{5.4}$$

Note that, if the forest is trained on state-action pairs, multiple leaves in a tree might be active. This results from the fact that the feature vector only depends on the state. If a test in a tree checks an action dimension $d > \dim(\boldsymbol{s})$, the test cannot be evaluated as the input $\boldsymbol{s}$ does not contain the dimension. Therefore, the path is split, for the left and right child, and both paths are further evaluated.

5.2.2 Active Samples

In this setting, each sample in the training set $\mathcal{TS}$ from the extra trees corresponds to exactly one entry in the feature vector. The resulting feature vector $\hat{\boldsymbol{\phi}}(\boldsymbol{s})$ will therefore contain $|\mathcal{TS}|$ entries. The forest F is queried for the active leaves for the input $\boldsymbol{s}$. This is equivalent to the feature vector $\boldsymbol{\phi}(\boldsymbol{s})$ from the active-leaves method (see Section 5.2.1) for the input $\boldsymbol{s}$. Each leaf ζ contains a subset of the training samples, therefore $\mathcal{D}_\zeta \subseteq \mathcal{TS}$. We consider the samples in $\mathcal{D}_\zeta$ as active for the given input $\boldsymbol{s}$, if ζ is active. The resulting feature vector $\hat{\boldsymbol{\phi}}(\boldsymbol{s})$ contains the number of times the forest considers each training sample as active, normalized by the number of trees M. This can be obtained by taking the scalar product between the feature vector $\boldsymbol{\phi}(\boldsymbol{s})$ and the matrix $\boldsymbol{\Phi}$. The feature vector $\boldsymbol{\phi}(\boldsymbol{s})$ is obtained by the active-leaves method for the input $\boldsymbol{s}$. The matrix $\boldsymbol{\Phi}$ contains the feature vectors $\boldsymbol{\phi}(\boldsymbol{d}_i)$ for each training sample $\boldsymbol{d}_i \in \mathcal{TS}$, also obtained with the active-leaves method. Thus, the matrix

$$\boldsymbol{\Phi} = \left[\boldsymbol{\phi}(\boldsymbol{d}_1), \boldsymbol{\phi}(\boldsymbol{d}_2), \dots, \boldsymbol{\phi}(\boldsymbol{d}_{|\mathcal{TS}|})\right] \tag{5.5}$$

has size $n_F \times |\mathcal{TS}|$, where $n_F = \sum_{i=1}^{M} n_i$ is the number of leaves in the forest. The feature vector $\hat{\boldsymbol{\phi}}(\boldsymbol{s})$ is therefore

$$\hat{\boldsymbol{\phi}}(\boldsymbol{s}) = \frac{1}{M}\left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{s})\right). \tag{5.6}$$

For large forests, this method creates feature vectors with a smaller dimension than the active-leaves method. It should therefore increase the computation speed during optimization in REPS.

6 Non-Linear Policy Based on Trees

The forest can also be used to define a policy. In this case, we will base the policy on locally weighted regression, where the weights are determined using a forest and the weighting from REPS. This learned policy is then used to sample an action for a given state.

6.1 Building the Forest

The trees in the forest F for the policy are built with the states $\boldsymbol{s}$ as input and the actions $\boldsymbol{a}$ as output. In this case, the score function has to be adapted to support multidimensional outputs as the actions can also have more than one dimension. The score function now sums over the relative variance score (see Equation (4.1)) of all m output dimensions

$$\mathrm{score}(\mathcal{D},[s_d < t]) = -\sum_{i=1}^{m} \frac{|\mathcal{D}_l|\,\mathrm{var}(a_i|\mathcal{D}_l) + |\mathcal{D}_r|\,\mathrm{var}(a_i|\mathcal{D}_r)S}{|\mathcal{D}|}, \tag{6.1}$$

where a_i is the i^th dimension of the action vector. When using the trees in conjunction with REPS, the reward weighting should also be taken into consideration when the score for a split is determined. The weight $\boldsymbol{w} \in \mathbb{R}^n$ is a vector containing the reward weighting for each sample. Actions that have a high reward weighting, should have more influence when building the trees. The score function therefore replaces the computation of the variance with the computation of the weighted variance

$$\mathrm{var}_w(y|\mathcal{D}) = \frac{\sum_{i=1}^{|\mathcal{D}|} w_i y_i^2}{\sum_{i=1}^{|\mathcal{D}|} w_i} - \left(\frac{\sum_{i=1}^{|\mathcal{D}|} w_i y_i}{\sum_{i=1}^{|\mathcal{D}|} w_i} \right)^2 . \tag{6.2}$$

The weighted score function is now given by

$$\mathrm{score}_w(\mathcal{D},[s_d < t]) = -\sum_{i=1}^{m} \frac{\left(\sum_{j=1}^{|\mathcal{D}_l|} w_{lj}\right)\mathrm{var}_w(a_i|\mathcal{D}_l) + \left(\sum_{j=1}^{|\mathcal{D}_r|} w_{rj}\right)\mathrm{var}_w(a_i|\mathcal{D}_r)}{\sum_{j=1}^{|\mathcal{D}|} w_j}. \tag{6.3}$$

The predicted output $\hat{\boldsymbol{a}}$ of the forest is obtained by using Equation (4.3).

6.2 Computing Sample Weighting from Forest

Given a new state $\hat{\boldsymbol{s}}$ the policy needs to sample an action $\hat{\boldsymbol{a}}$. This action should be based on past actions from similar states that achieved high rewards. We therefore weight the samples based on the forest and the weighting $\boldsymbol{w}$ from REPS.

Samples are considered to be more similar if they activate more of the same leaves in the forest F. The weight $\hat{w}_i$ for a training sample $(\boldsymbol{s}_i, \boldsymbol{a}_i)$ is given by

$$\hat{w}_i = \frac{w_i}{M} \boldsymbol{\phi}(\boldsymbol{s}_i)^{\mathrm{T}} \boldsymbol{\phi}(\hat{\boldsymbol{s}}), \tag{6.4}$$

where the two feature vectors $\boldsymbol{\phi}(\boldsymbol{s}_i)$ and $\boldsymbol{\phi}(\hat{\boldsymbol{s}})$ are computed based on the forest F as described in Section 5.2.1, and w_i is the sample weighting from REPS in Section 3.2. Hence, a sample only gets a high weight, if it is similar to the new state $\hat{\boldsymbol{s}}$ and received a large weight during the policy update.

6.3 Policy

The policy is modeled as a Gaussian distribution

$$\pi(\hat{\boldsymbol{a}}|\hat{\boldsymbol{s}}) = \mathcal{N}\left(\hat{\boldsymbol{a}}|\boldsymbol{\mu}(\hat{\boldsymbol{s}}), \boldsymbol{\Sigma}(\hat{\boldsymbol{s}})\right). \tag{6.5}$$

The parameters for the Gaussian distribution, the mean $\boldsymbol{\mu}(\hat{\boldsymbol{s}})$ and the variance $\boldsymbol{\Sigma}(\hat{\boldsymbol{s}})$, are computed based on the weights $\hat{w}_i$ which are obtained as described in the previous Section 6.2. The mean and variance depend on the state $\hat{\boldsymbol{s}}$ as the new action will be based on past samples with states similar to $\hat{\boldsymbol{s}}$. The computation of the parameters is done by locally weighted regression

$$\begin{bmatrix} \boldsymbol{\alpha}^{\mathrm{T}} \\ \boldsymbol{\beta}^{\mathrm{T}} \end{bmatrix} = \left(\boldsymbol{S}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{S} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{S}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{A}, \tag{6.6}$$

where the i^th row of the matrix $\boldsymbol{S}$ is given by $\left[1, \boldsymbol{s}_i^{\mathrm{T}}\right]$ and the i^th row of the matrix $\boldsymbol{A}$ is given by $\boldsymbol{a}_i^{\mathrm{T}}$. The i^th diagonal element of the matrix $\boldsymbol{W}$ is given by $\hat{w}_i$, such that $\boldsymbol{W} = \mathrm{diag}(\hat{\boldsymbol{w}})$. $\boldsymbol{I}$ is the identity matrix with ones on the diagonal and zeros everywhere else. The variable λ is a regularization term. Large values of λ penalize large $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ values more. In our evaluations we used a standard value of $\lambda = 10^{-10}$.

After estimating the $\boldsymbol{\alpha}$ vector and the $\boldsymbol{\beta}$ matrix, the mean and the variance are computed by

$$\boldsymbol{\mu}(\hat{\boldsymbol{s}}) = \boldsymbol{\beta}\hat{\boldsymbol{s}} + \boldsymbol{\alpha}, \tag{6.7}$$

$$\boldsymbol{\Sigma}(\boldsymbol{s}) = \frac{\sum_i \hat{w}_i \left(\boldsymbol{a}_i - \boldsymbol{\mu}(\hat{\boldsymbol{s}})\right)\left(\boldsymbol{a}_i - \boldsymbol{\mu}(\hat{\boldsymbol{s}})\right)^{\mathrm{T}}}{Z}, \tag{6.8}$$

$$Z = \frac{\left(\sum_i \hat{w}_i\right)^2 - \sum_i \hat{w}_i^2}{\sum_i \hat{w}_i}. \tag{6.9}$$

With the parameters determined above, the new action can then be obtained by sampling from the distribution $\pi(\hat{\boldsymbol{a}}|\hat{\boldsymbol{s}})$. Locally weighted regression is a form of lazy learning. Therefore the mean and variance must be recomputed for every new state. However, we can use locally weighted regression to learn non-linear policies instead of just linear policies that can be obtained by linear regression.

7 Evaluations

To evaluate our method, we decided to learn to pocket balls in a simulated pool game. In the game pool, the dimensionality can be easily scaled by adding or removing balls. Every additional ball adds two dimensions to the state, which contains the x and y positions of all of the balls present on the pool table. Additionally, not every dimension of the state is relevant in pool. If, for example, an additional ball does not block the pocket it can easily be ignored by the learning algorithm. As a first step to evaluate our method, the experiments in this thesis are done with two balls on the table.

7.1 Pool Simulator

To be able to execute pool games, we implemented a two dimensional simulator in Matlab and C (mex). It simulates the behavior of the balls given a valid starting state and an initial velocity of the cue ball.

The pool table has a length ratio of 2 : 1, with the longer side having a length of 1 meter. All of the n balls have a radius of 0.015 m and the pockets have a radius of 0.03 m. The number of balls is the same for all episodes. One of the balls is always the white cue ball which can be influenced by the player directly, by giving the initial velocity to the simulator. There also has to be at least one blue ball which is used to define the reward. Red balls serve as obstacles. There are six pockets, four placed in the corners and two placed centered on the longer side. A picture of the pool simulation is shown in Figure 7.1. An episode is considered to be finished when all balls have come to a rest.

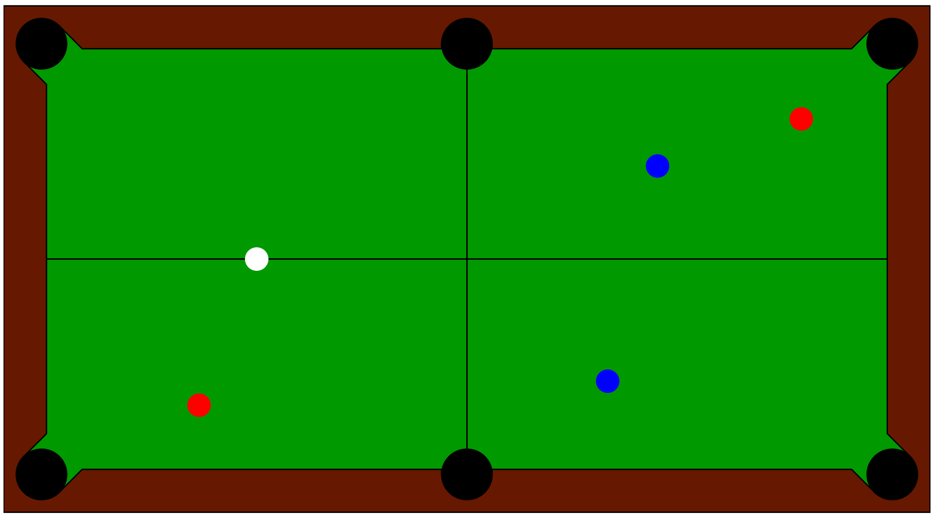


Figure 7.1: Pool table visualized by the simulator in Matlab.

7.1.1 Initialization

The simulator is initialized with a state $\boldsymbol{s}$ and an action $\boldsymbol{a}$. Given n balls the state vector $\boldsymbol{s} \in \mathbb{R}^{2n}$ contains all of the ball positions

$$\boldsymbol{s} = \left[\boldsymbol{c}^{\mathrm{T}}, \boldsymbol{r}_1^{\mathrm{T}}, \dots, \boldsymbol{r}_b^{\mathrm{T}}, \boldsymbol{o}_1^{\mathrm{T}}, \dots, \boldsymbol{o}_{n-(b+1)}^{\mathrm{T}}\right]^{\mathrm{T}} \tag{7.1}$$

where $\boldsymbol{c} = (c_x, c_y)^{\mathrm{T}}$ is the white cue ball, $\boldsymbol{r} = (r_x, r_y)^{\mathrm{T}}$ are the blue reward balls and $\boldsymbol{o} = (o_x, o_y)^{\mathrm{T}}$ are the red obstacle balls. The number of blue reward balls is denoted by b. The state vector is sampled uniformly

$$\boldsymbol{s} \sim \mathcal{U}(\gamma_l, \gamma_u), \tag{7.2}$$

where $\mathcal{U}$ is the uniform distribution and $[\gamma_l, \gamma_u]$ define its interval. The action vector $\boldsymbol{a} \in \mathbb{R}^2$ contains the initial velocities of the cue ball $\dot{\boldsymbol{c}} = (\dot{c}_x, \dot{c}_y)^{\mathrm{T}}$ and it is sampled from the policy π

$$\boldsymbol{a} \sim \pi(\boldsymbol{a}|\boldsymbol{s}). \tag{7.3}$$

The policy used in our experiments is described in Chapter 6.

7.1.2 Pool Physics Simulation

In each time step, collisions are checked between each pair of balls and each ball with the walls. A collision is detected when an overlap occurs and the relative velocities point towards each other. For a collision between a ball and a wall, we can decompose the velocity of the ball into a normal component v_n and a tangential component v_t. After collision, the resulting normal velocity v_n' is

$$v_n' = -\kappa v_n, \tag{7.4}$$

where κ is the coefficient of restitution. The tangential velocity v_t remains unchanged. For a collision between two balls, we can again decompose the velocities into normal components v_{1n} and v_{2n} and tangential components v_{1t} and v_{2t}. Given that the balls have the same masses the normal components of the balls are exchanged, i.e.

$$v_{1n}' = v_{2n}, \tag{7.5}$$

$$v_{2n}' = v_{1n}. \tag{7.6}$$

Again, the tangential velocities v_{1t} and v_{2t} remain unchanged. The spin of the balls is not simulated. Balls come to a rest by either friction or falling into a pocket. Friction is modeled by reducing the velocity by a fixed percentage at each time step. The velocity of a ball is set to zero if it falls below a small threshold. A ball falls into a pocket if the center of the ball is within the area of a pocket.

7.2 Reward Function

The quality of the reward function is crucial to the success of the system. The reward function should guide the agent towards performing the desired task. In pool the desired task is to shoot the reward balls into the pockets. In our evaluations, we are only focusing on two neighboring corner pockets. If the reward ball is pocketed, the agent receives the maximum reward of zero. The agent otherwise receives a negative reward for the final distance between the reward ball and a pocket. The return is

$$\mathcal{R}_{\boldsymbol{sa}} = -\min_{i,j} \|\boldsymbol{r}_i - \boldsymbol{p}_j\|, \tag{7.7}$$

where $\boldsymbol{r}_i$ is the ith reward ball and $\boldsymbol{p}_j$ is the jth pocket position. An additional penalty is incurred if the cue ball fails to hit the reward ball. This penalty is given by the minimum distance between the cue ball and the reward ball times ten. The reward function therefore first guides the agent to hitting the reward ball with the cue ball, and then towards pocketing the reward ball. Although this reward function may seem simple, the minimization over the trajectories leads to a complicated reward function with many local optima. An example of the reward function can be found in Figure 7.2. The trajectories generated by two actions from the previous plot can be found in Figure 7.3. One of the actions hits the upper pocket and the other the lower pocket. Averaging over these two solutions leads to poor performance. Therefore, there are multiple solutions for the same state which are similar but also distinct. Such solutions can be challenging for the learning process.

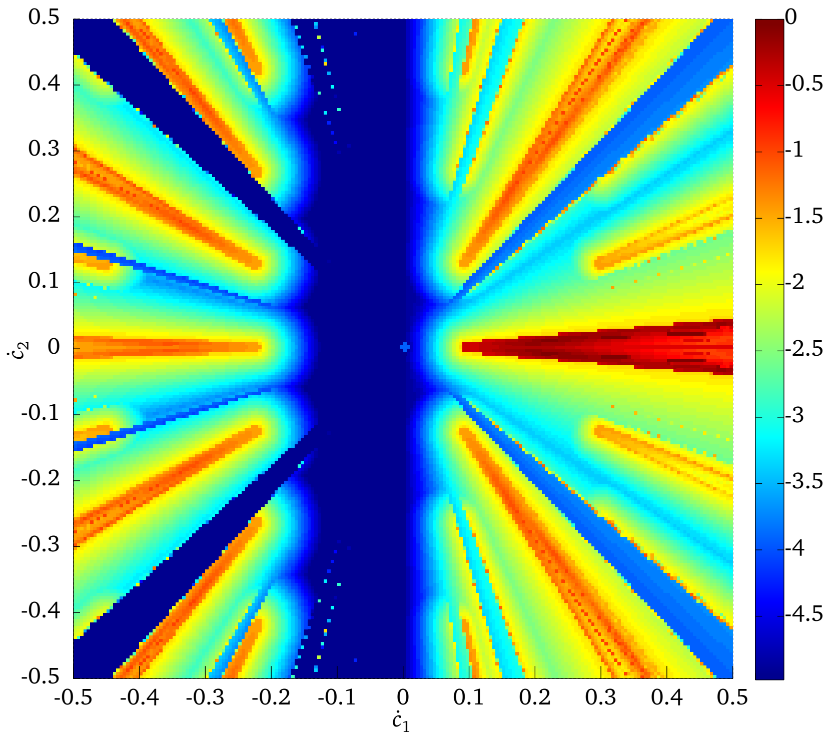


Figure 7.2: This plot of the reward function was generated by setting the cue ball and the reward ball to the fixed positions $(-0.25, 0)$ and $(0.1, 0)$ respectively. The action was varied and the resulting rewards are indicated by the colors at the corresponding position in the action space. Only the dark red spots are actions that received the highest possible reward of 0.

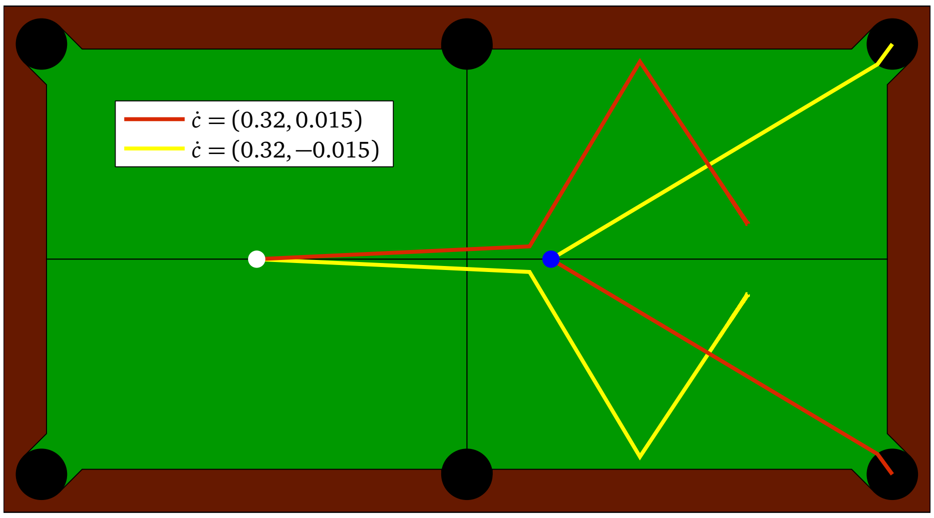


Figure 7.3: Two samples for actions that received 0 reward. One shot the reward ball into the lower pocket (red), and the other into the upper pocket (yellow). For the same position several optimal actions can be found.

7.3 Experiment 1: Representing the Reward Function

The goal of this experiment is to determine how well the trees can represent the reward function. This evaluation is important as the trees will be used as features for modeling the value function.

7.3.1 Setup

In this scenario there are only two balls on the table, the cue ball and one reward ball. The cue ball is always positioned at $(-0.25, 0)$ relative to the center of the table. The position of the reward ball is sampled uniformly from a square region with width 0.4 m and centered at $(0.25, 0)$. To create samples of the reward function the agent was programmed with a policy that directly aims the cue ball at the reward ball. A total of 1000 training samples and 1000 test samples were obtained using this approach. Each forest was always trained on a subset of only 200 samples at a time. The five trained forests were evaluated on 200 separate test samples. We evaluated different parameter values for n_{min}, K and M. As default values we used $n_{min} = 20$, $K = 10$ and $M = 30$. The results for changing n_{min} and M are shown in Figure 7.4a and Figure 7.4b respectively. In Figure 7.4c the effects of changing K can be seen.

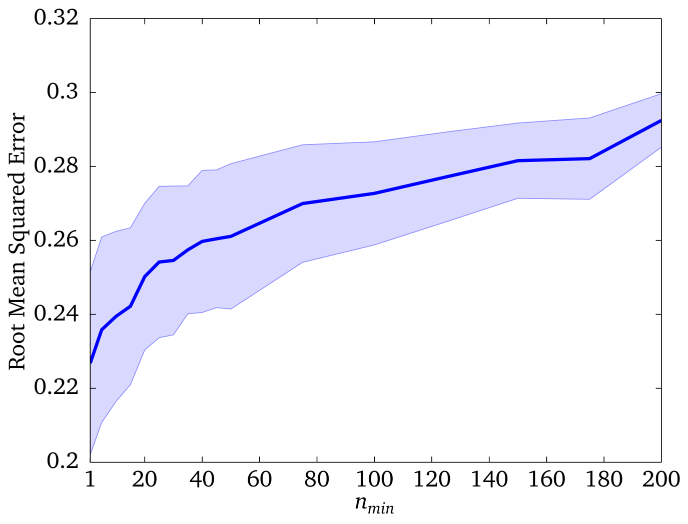
7.3.2 Discussion

The best performance was achieved using $K = 10$. The performance seems to slightly decrease when using more cuts, although it is not a significant result. A more noticeable decrease in performance occurs when the number of cuts tends to one. The totally randomized tree approach is therefore not suitable for this task.

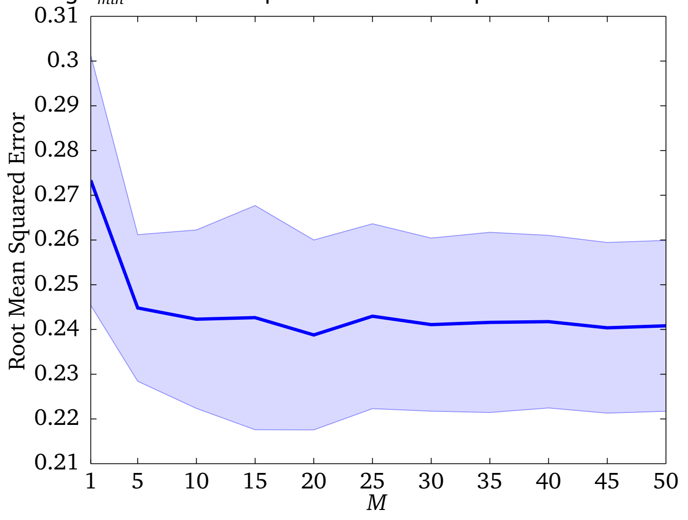
As one would expect using only a single tree led to poor performance. When using ten trees the performance is already almost as good as when using 50 trees. Adding more trees to the ensemble does not lead to overfitting. Therefore it is usually better to use more trees rather than too few.

A smaller value for n_{min} achieves the best performance. Even for the smallest value for n_{min} the forest does not overfit. This surprising effect may be the result of using a deterministic simulator and the uniform sampling. The performance decreases only slightly between 80 and 180. As expected the performance is the worst when $n_{min} = 200$, at which point there is no tree structure.

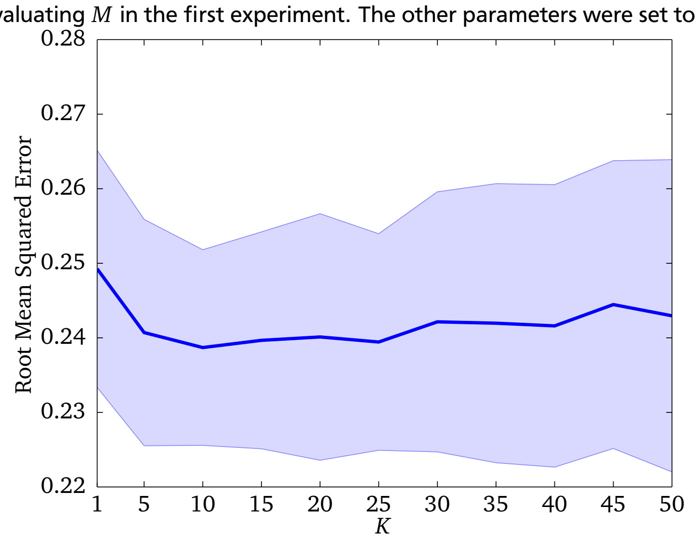
When looking at the predicted rewards, it can be seen that the forests are not well suited for representing large discontinuities in the reward function. The reward in regions where the reward ball is pocketed are generally underestimated. Although this underestimation may lead to low performance in regression tasks it may actually be beneficial for learning value functions. As the features are averaging over good and bad samples that are in the neighborhood, the good samples will have a higher advantage and therefore be assigned higher weights for the next policy update.



(a) The results from evaluating n_{min} in the first experiment. The other parameters were set to $K = 10$ and $M = 30$.



(b) The results from evaluating M in the first experiment. The other parameters were set to $K = 10$ and $n_{min} = 20$.



(c) The results from evaluating K in the first experiment. The other parameters were set to $M = 30$ and $n_{min} = 20$.

Figure 7.4: The results of using different parameters from the first experiment. The error bars indicate one standard deviation.

7.4 Experiment 2: Representation of an Optimal Policy

In the second experiment we investigate how well the tree based policy can represent optimal policies for pool.

7.4.1 Setup

In order to train the policy, samples were collected using the ghost ball strategy, as proposed by Smith [21]. The pocket closest to the reward ball is selected as the target. The direction from the reward ball to the pocket is computed and the ghost ball is placed next to the reward ball in the negative direction, as shown in Figure 7.5. The cue ball is then aimed at the ghost ball. When the cue ball and the ghost ball overlap, the reward ball is hit towards the target pocket.

Three different policies were evaluated in this experiment. In the first policy, the n_{min} for the trees is set to the number of samples. As a result, the tree only has the root node. The second policy learned trees based on the mapping from states to actions. The third policy was trained using the mapping from states to reward states, i.e., the final position of the reward ball. For the second and third policies the number of trees was set to 30 and the number of cuts to 10. The n_{min} parameter was adapted to the size of the training set, and was always set to a tenth of the number of training samples. The number of training samples ranged from 10 to 1000.

The cue ball was placed at $(-0.25, 0))$. The position of the reward ball was uniformly sampled from a square region with a width of 0.4 m centered on $(0.25, 0))$.

For each number of training samples, each policy was evaluated on 1000 test shots. The performances of the policies are shown in Figure 7.6. Similar to the training policy, the learned policies also aim for the two pockets.

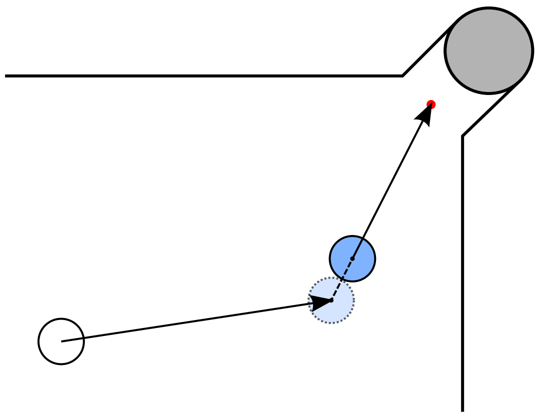


Figure 7.5: An optimal action hits the reward ball in the direction of the pocket. The center at the entry of the pocket corridor is used for aiming. The cue ball has to aim for the center of a "ghost ball", indicated by a dotted circle.

7.4.2 Discussion

In all three policies there is a significant drop in performance from the optimal training data. This drop in performance is expected given the precision that is needed for pool. A ball that is not pocketed is likely to bounce off of a nearby wall and may come to a rest quite far away from a pocket. The largest drop in performance can be seen for the policy with only a single node in each tree. This policy is not suitable for representing the multimodal policy. Instead, it averages over the solutions resulting in poor overall performance. The policy based on mappings from states to actions performed better, but not as well as the policy based on mappings from states to reward states. This result is due to the fact that the actions are not as discriminative as the reward states. In summary, the results of the experiment show that the trees allow the agent to learn multimodal policies, which can represent different solutions to the task. The mapping from states to reward states allows a better distinction between the two solutions, and therefore outperforms the other two policies.

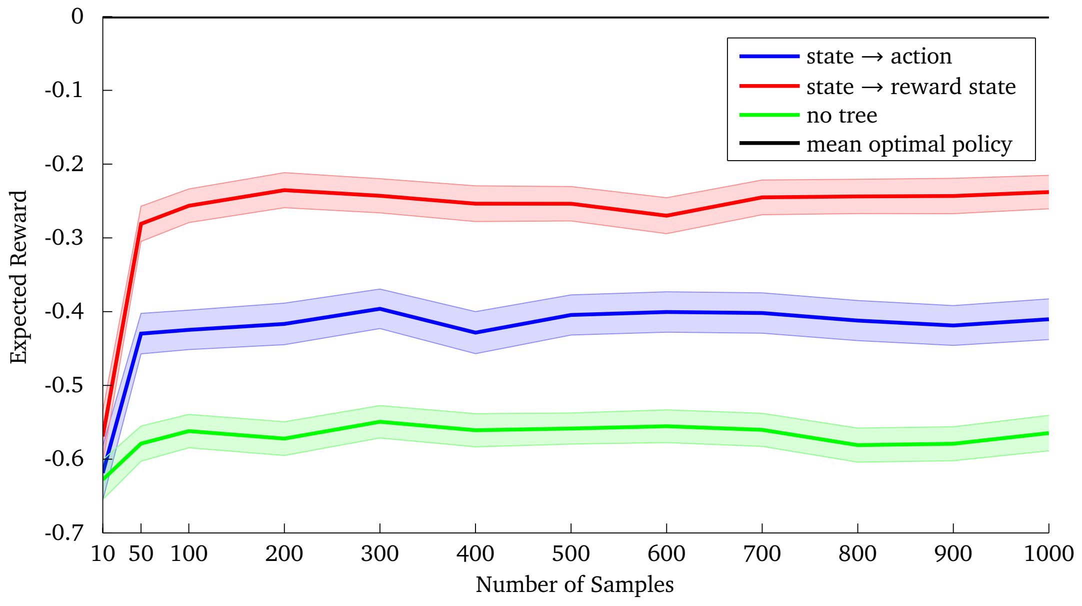


Figure 7.6: The results of modeling optimal samples using tree-based policies. The error bars indicate two standard error.

7.5 Experiment 3: Learning to Pocket a Ball with REPS

In the final experiment we use REPS to learn the policy for pocketing the reward ball.

7.5.1 Setup

For this evaluation, the cue ball was positioned at $(-0.25, 0)$. The initial state of the reward ball was sampled uniformly in a rectangular region with a width of 0.1 m and a height of 0.2 m centered at $(0.1, 0)$.

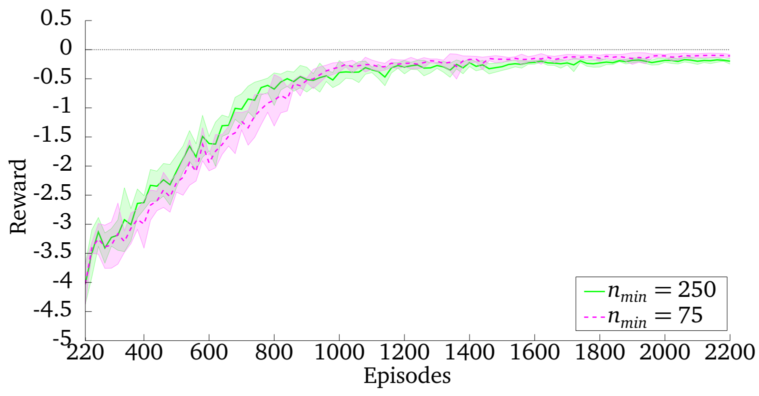
Three policies were evaluated in this scenario. The first policy used a n_{min} of 250. As a result, the policy represents a baseline where the tree only contains a single node. The second and third policies used a n_{min} of 75, the number of cuts K was set to 20 and the forest contained 10 trees. The second policy was trained using the mapping from states to reward states, while the third policy was based on the mapping from states to actions.

In addition to the different policies, we also tested two different score functions for learning the value function features. The first score function is the relative variance reduction, as explained in Chapter 4. The second score function is the negative relative variance reduction, i.e., the negative of the first score function. The latter score function attempts to maximize the variance within each partition, rather than reduce it. Hence, the advantage signal should be large, which may help the learning process.

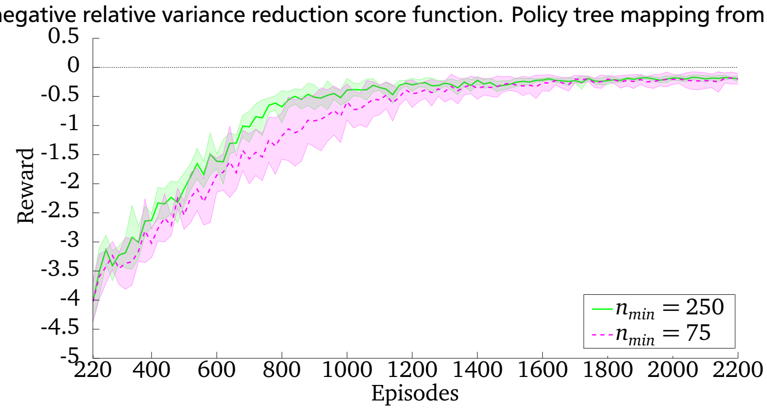
Each evaluation was initialized with 200 samples using a Gaussian policy with mean zero and isotropic variance of 0.01. The agent was then given 100 iterations to learn the task. In each iteration, 20 new samples were generated using the current policy. The policy was always based on the last 200 samples using importance weighting. The evaluation was repeated 5 times for each policy, in order to determine the repeatability of the performance. The average return for each iteration is shown in Figure 7.7. Figure 7.8 shows the final performance in more detail. For the final policies, the number of balls that were pocketed are shown in Figure 7.9. Additionally Figure 7.10 shows the distribution of the final distances between the reward ball and the nearest pocket.

7.5.2 Discussion

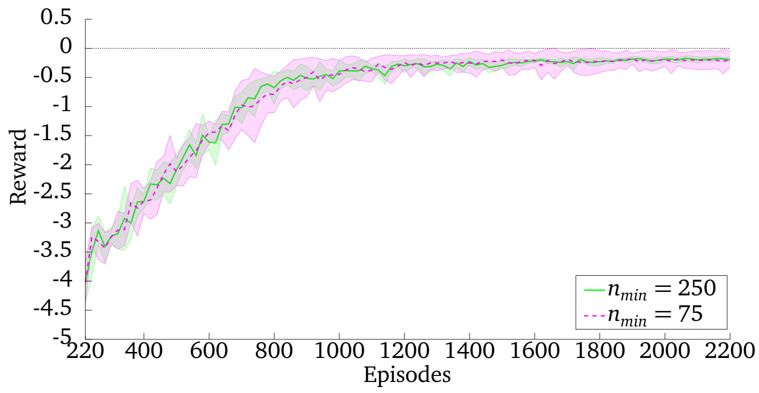
Overall none of the tree based policies performed considerably better than the baseline policy. The worst performance was achieved when using the relative variance reduction score, and a policy based on the mapping from states to actions. This policy has a large variance in the returns. It managed to pocket more balls than the baseline policy, but the balls that missed tended to stop further away from the pockets. This behavior is often caused by policies that use excessive velocities. The best performance was achieved when using the negative relative variance score, and a policy based on the mapping from states to reward states. The mean of the returns for this policy is approximately 0.1 greater than the baseline policy. This improvement corresponds to the ball being on average about 10 cm closer to the pocket, which is a significant improvement considering the precision required to play pool. The policy also distributed the pocketed balls almost evenly between the two pockets. The baseline policy pocketed the fewest balls. Using the negative relative variance score together with a policy based on the mapping from states to actions also led to several balls being pocketed. This policy, however, favored the lower pocket. Its overall performance is also decreased by the number of balls with a distance greater than 0.3. All of the policies converged after approximately 1400 episodes. Therefore, it seems like many of them converged to a poor local optima. This is not too surprising considering the many local optima in the reward function, and that the policy was learned from scratch. Better policies could potentially be learned by initializing the policy through imitation learning.



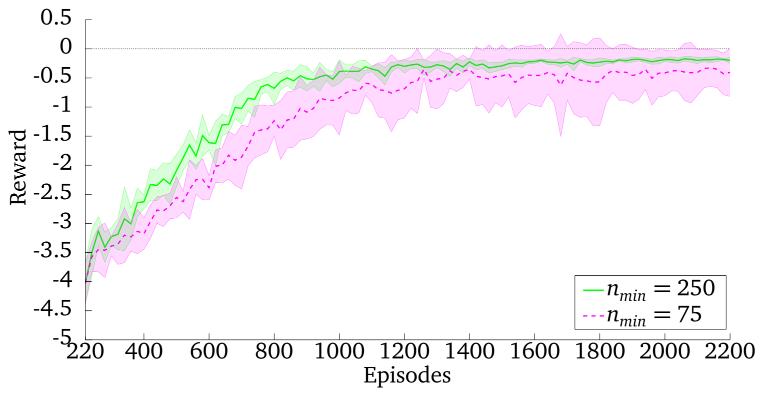
(a) Feature tree with negative relative variance reduction score function. Policy tree mapping from states to reward states.



(b) Feature tree with relative variance reduction score function. Policy tree mapping from states to reward states.

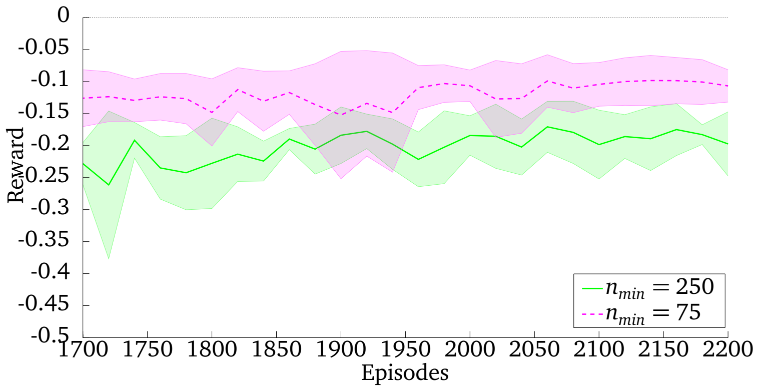


(c) Feature tree with negative relative variance reduction score function. Policy tree mapping from states to actions.

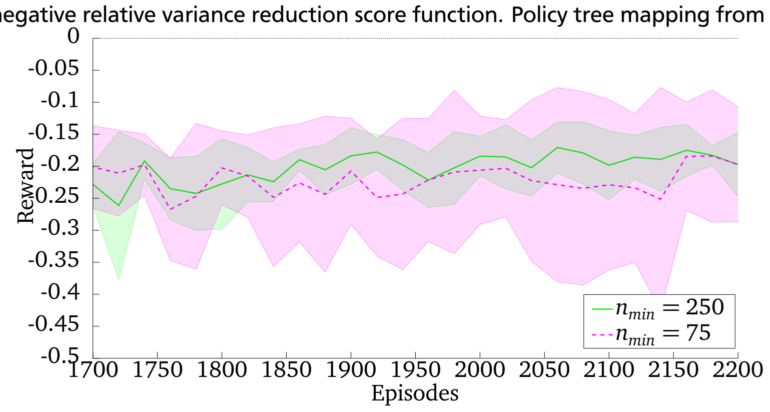


(d) Feature tree with relative variance reduction score function. Policy tree mapping from states to actions.

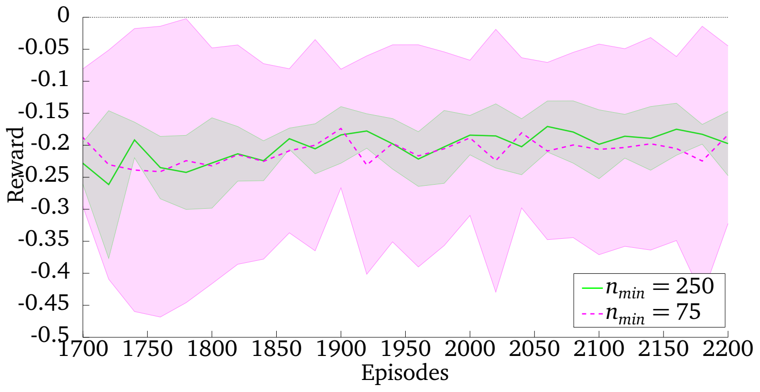
Figure 7.7: The performance of the policies evaluated in the third experiment.



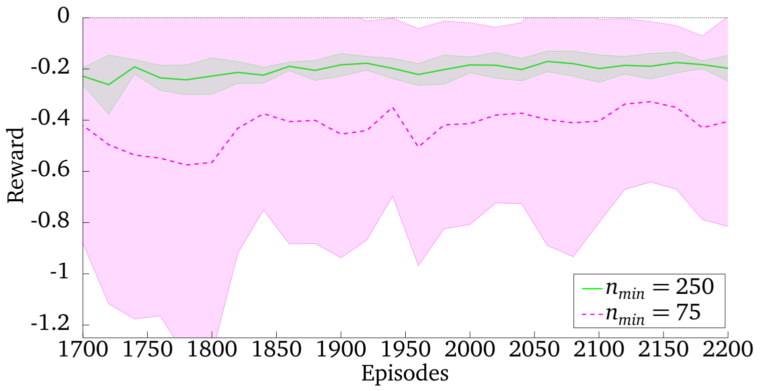
(a) Feature tree with negative relative variance reduction score function. Policy tree mapping from states to reward states.



(b) Feature tree with relative variance reduction score function. Policy tree mapping from states to reward states.



(c) Feature tree with negative relative variance reduction score function. Policy tree mapping from states to actions.



(d) Feature tree with relative variance reduction score function. Policy tree mapping from states to actions.

Figure 7.8: The performance of the policies in the last 500 episodes. The error bars indicate one standard deviation.

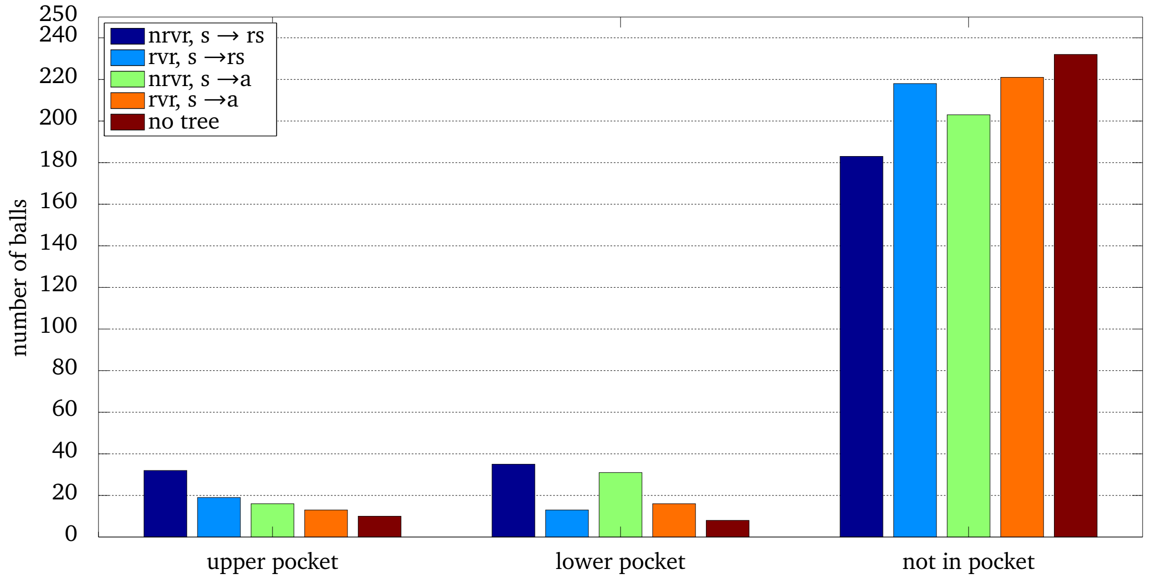


Figure 7.9: The histogram shows the pocketing statistics for the learned policies. Policies based on mappings from states to actions and from states to reward states are denoted by "s → a" and "s → rs" respectively. The value function was trained with the relative variance reduction score (rvr) or the negative rvr score (nrvr).

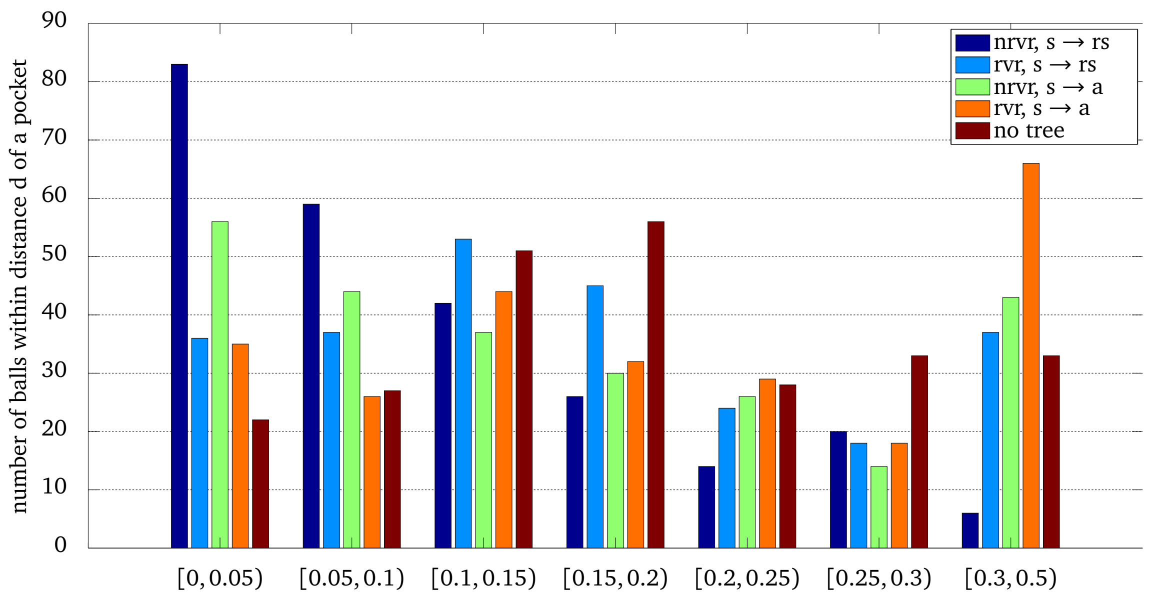


Figure 7.10: For each learned policy, the histogram indicates the final distance of the reward ball to the nearest pocket. Policies based on mappings from states to actions and from states to reward states are denoted by "s → a" and "s → rs" respectively. The value function was trained with the relative variance reduction score (rvr) or the negative rvr score (nrvr).

8 Conclusion and Future Work

In this thesis, we investigated using a tree based approach for learning features for policy search methods. Features are usually handcrafted specifically for a task. By learning suitable features the agent can become more autonomous. Our focus was on learning features for relative entropy policy search. In particular, we proposed using extra trees to generate the features for representing the value function and the policy. The value function features were learned by training forests to map from states to rewards. The partitioning of the state space, which is defined by the leaves of the forest, is transformed into a feature representation. The policy features were based on forests learning mappings from states to actions. In the experiments, we also investigated using mappings from states to reward states. The policies use the learned features as a basis for locally weighted regression.

The proposed method was evaluated on a simulated pool task. The experiments showed that our tree based approach is able to represent multimodal policies, which capture multiple solutions. The regression trees could also represent the general structure of the reward function, except for the large discontinuities. In some cases, our tree based REPS method was able to learn good policies from scratch. However, learning to pocket balls without prior information is a difficult task, and many learned policies converged to poor local maxima.

There are several possible ways to extend the proposed approach in the future. In the current framework, the tests for splitting a tree were based on the positions of individual balls. An alternative form of test could evaluate the number of balls in different regions of the table. This approach could be especially useful when dealing with many balls or even a varying number of balls.

The score function used to evaluate the tests in the trees for the value features should also be further investigated. Although we had expected that the relative variance reduction would lead to the best performance, using the negative of the score function resulted in a better performance. Determining which states should be combined in one partition remains an open problem.

Better policies could be learned by incrementally increasing the difficulty of the task. At the beginning the initial state could be fixed to allow the agent to hit and pocket the ball from this state. Over time the variance of the initial state could then be increased, resulting in more difficult tasks. This method might help to avoid poor local optima, but it would probably not learn multimodal solutions.

Another interesting extension to the pool task would be to look into sequential shots. The challenge in this setting would be to also position the balls for the next shot to achieve a high reward.

Finally, better policies can potentially be learned using hierarchical relative entropy policy search (HiREPS) [1]. The policy is currently learned for a limited region of the state space. Using HiREPS, lower level policies could be learned for different regions of the state space and then combined into one hierarchical policy.

Bibliography

[1] C. Daniel, G. Neumann, and J. Peters, “Learning concurrent motor skills in versatile solution spaces,” in *Proceedings of the International Conference on Robot Systems (IROS)*, 2012.

[2] C. Daniel, G. Neumann, O. Kroemer, and J. Peters, “Learning sequential motor tasks,” in *Proceedings of 2013 IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[3] R. S. Sutton and A. G. Barto, “Reinforcement learning: An introduction,” 1998.

[4] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[5] R. Bellman, *Dynamic Programming*. Dover Publications, 1957.

[6] M. P. Deisenroth, G. Neumann, and J. Peters, “A survey on policy search for robotics,” *Foundations and Trends in Robotics*, pp. 388–403, 2013.

[7] J. Peters and S. Schaal, “Natural actor critic,” *Neurocomputing*, no. 7-9, pp. 1180–1190, 2008.

[8] J. Kober and J. Peters, “Policy search for motor primitives in robotics,” *Machine Learning*, no. 1-2, pp. 171–203, 2011.

[9] J. N. Tsitsiklis and B. V. Roy, “Feature-based methods for large scale dynamic programming.” *Machine Learning*, vol. 22, no. 1-3, pp. 59–94, 1996.

[10] L. Li, T. J. Walsh, and M. L. Littman, “Towards a unified theory of state abstraction for mdps.” in *ISAIM*, 2006.

[11] J. S. Baras and V. S. Borkar, “A learning algorithm for markov decision processes with adaptive state aggregation,” in *Proceedings of the 39th IEEE Conference on Decision and Control*, 2000, pp. pp.3351–3356.

[12] Z. Ren and B. Krogh, “State aggregation in markov decision processes,” in *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, vol. 4, 2002, pp. 3819–3824 vol.4.

[13] A. W. Moore, “The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces.” in *NIPS*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds. Morgan Kaufmann, 1993, pp. 711–718.

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[15] S. Lange and M. Riedmiller, “Deep auto-encoder neural networks in reinforcement learning,” in *International Joint Conference on Neural Networks (IJCNN 2010)*, Barcelona, Spain, 2010.

[16] R. Faulkner and D. Precup, “Dyna planning using a feature based generative model,” in *Proceedings of NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, Whistler, Canada, 2010.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning.” *CoRR*, 2013.

[18] D. Ernst, P. Geurts, and L. Wehenkel, “Tree-based batch mode reinforcement learning,” in *Journal of Machine Learning Research*, 2005, pp. 503–556.

[19] P. Pastor, M. Kalakrishnan, S. Chitta, E. Theodorou, and S. Schaal, “Skill learning and task outcome prediction for manipulation.” in *ICRA*. IEEE, 2011, pp. 3828–3834.

[20] M. A. Greenspan, J. Lam, W. Leckie, M. Godard, I. Zaidi, K. Anderson, D. C. Dupuis, and S. Jordan, “Toward a competitive pool playing robot: Is computational intelligence needed to play robotic pool?” in *CIG*. IEEE, 2007, pp. 380–388.

[21] M. Smith, “Pickpocket: A computer billiards shark,” *Artificial Intelligence*, vol. 171, no. 16, pp. 1069–1091, 2007.

[22] J. Peters, K. Muelling, and Y. Altun, “Relative entropy policy search,” in *Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence (AAAI), Physically Grounded AI Track*, 2010.

[23] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[24] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.