Policy Extraction via Online Q-Value Distillation

by

Aman Jhunjhunwala

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Recently, deep neural networks have been capable of solving complex control tasks in certain challenging environments. However, these deep learning policies continue to be hard to interpret, explain and verify which limits their practical applicability. Decision Trees lend themselves well to explanation and verification tools but are not easy to train especially in an online fashion. The aim of this thesis is to explore online tree construction algorithms and demonstrate the technique and effectiveness of distilling reinforcement learning policies into a Bayesian tree structure.

We introduce Q-BSP Trees and an Ordered Sequential Monte Carlo training algorithm that helps condense the Q-function from fully trained Deep Q-Networks into the tree structure. QBSP Forests generate partitioning rules that transparently reconstruct the Value function for all possible states. It convincingly beats performance benchmarks provided by earlier policy distillation methods resulting in performance closest to the original Deep Learning policy.

Acknowledgements

I would like to thank all the people who made this thesis possible. I would begin by thanking my supervisor Dr. Krzysztof Czarnecki for his extraordinary support and guidance throughout my time here. Professor Czarnecki gave me the freedom and support to work on problems I was interested in, for which I am very grateful.

I thank Professor Mark Crowley and Professor Peter van Beek for agreeing to be the readers of this thesis.

I would also like to thank Sean Sedwards, Jaeyoung Lee, Vahdat Abdelzad, Ashish Gaurav, Aravind Balakrishnan and all my colleagues at Waterloo Intelligent System Engineering Lab (WISELab) for all the discussion and feedback that enriched the project.

I am grateful to Xuhui Fan for his BSP Forest open-source libraries.

Finally, I would like to thank my family for their constant support and encouragement.

Dedication

This thesis is dedicated to everyone who directly or indirectly, made it possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Deep Reinforcement Learning has excelled at automatically learning human-level [27] policies for a certain number of control and gaming tasks. Recently these policies exceeded human capabilities in complex video games like StarCraft or Dota [30]. Inspite of these impressive gains in learning ability, the policy decisions remain impenetrable and this can lead to undesirable and even dangerous behaviour [2]. The pace of safety research has not kept up with the advances in policy design and it keeps getting harder to apply any oversight to these policies.

For applicability in the real world, we need to be able to satisfy safety characteristics of these policies which includes providing stability guarantees for the controllers [7], verifying their correctness [24] and measuring their robustness [5]. Confirming these characteristics directly for deep reinforcement learning policies is very inefficient and often not possible at all [17]. This may lead to distrust between the regulators, end users and the policy designers.

Distillation [22] has been an effective method to train relatively more compressed [12], interpretable[13], structured [40, 8, 15] or shallower [3] models. Recently, researchers have applied forms of imitation learning[1] to distill reinforcement learning policies from trained deep neural networks to tree structure representations in an active-play setting [6, 26]. In an active play setup, a fully-trained deep reinforcement learning algorithm (eg. DQN [27]) is observed as it performs actions in an environment (eg. Cartpole). All the transitions executed by the policy are passed through the online distillation algorithm that aims to

condense information into our desirable tree structure. Figure 3.1 depicts a summary of the process.

We aim to improve and extend prior research by proposing a novel tree distillation method that is self-consistent ,i.e., it can be trained in an online fashion efficiently and achieves better performance than other online and offline learning methods. This allows us to distill more complex state spaces in Reinforcement Learning environments than were previously possible due to memory constraints while also exceeding the regression performance and gameplay performance of previous online approaches.

Regressing over the Q-values of the network instead of classifying over actions allows us to structure the factors that led to a particular action being taken as well as the influences that impaired other unsuccessful actions.

1.2 Overview

Decision Trees and Random Forests have been introduced decades ago and remain one of the most popular machine learning algorithms even today. Over the years they have proved to be scalable, robust and accurate for real-world classification and regression tasks. However, most of the decision tree and random forest algorithms used today work in an offline manner. They need the entire view of the data before they build the tree. Attempts have been made to design decision trees for online data processing but very few have been as convincing as the Mondrian Tree Process [34] and the BSP Tree Process [18]. Hence we use the BSP-Tree Process as the base for creating our QBSP Forest algorithm. The visual differences between a standard decision tree policy and our QBSP Tree policy is shown in Figure 1.1

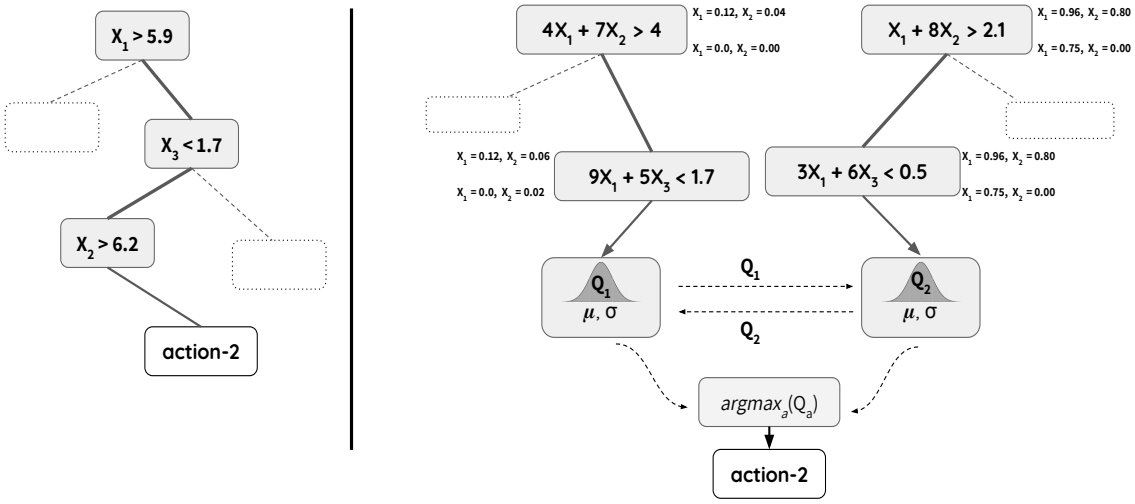

Figure 1.1: Representative Structural Comparison between a standard decision tree used by Viper[6] and our Q-BSP Tree for a Cartpole environment with 4 state dimensions ($X_1 ... X_4$) and 2 actions. Note that in our tree structure, there are 2 trees, each representing the Q-value for one action. Each node encodes more mathematically complex boundary expressions and contains the boundaries of operation. The output of one *action tree* is fed into the other. Just like in a DQN, the argmax over the Q values chooses the action.

In this thesis we present :

- A novel data-structure which we call the Q-BSP Tree to learn distilled reinforcement learning policies. Q-BSP Tree Nodes are strictly more expressive than standard decision tree nodes and are effective at capturing pair-wise dependencies among input features.
- A new combined regression and ranking algorithm based on the Particle Gibbs sampler that enables the online distillation of deep reinforcement learning policies into Q-BSP Trees. This method performs better and is orders of magnitude more scalable than any previous distillation approach applied to reinforcement learning policies.
- We present and compare our regression and gameplay performance with the current state of the art methods and show that the policies distilled by the trees closely resembles the neural network in terms of feature importance. We introduce this as a new effectiveness metric for distillation. In line with the work done earlier [6], we easily verify the correctness of our distilled policy trees for the Cartpole environment.

1.3 Thesis Organization

The thesis is organized into five distinct chapters.

- **Chapter 1** provides the motivation and high-level overview of this thesis.
- **Chapter 2** delves into important background concepts that led to the development of QBSP Distillation Method. We go over the basic concepts associated with Reinforcement Learning, Q-Learning and Deep Q-Networks. The Bayesian Additive and Regressive Trees model (BART) provides fundamental building blocks for understanding most Bayesian Tree construction algorithms and we formally describe their form and structure here. Finally we provide a condensed recap of the Particle Filtering Algorithm which is crucial for the training of QBSP Forests.
- **Chapter 3** describes into the architecture of the distillation process and presents the training process for the QBSP Forests.
- **Chapter 4** provides a description of the environments in which the Distillation process was tested and presents the results for our QBSP Distillation technique. The regression accuracy and gameplay performance provides empirical performance statistics. Examples of policies learnt by the trees are presented for each environment along with the influence transfer metrics. Finally the distilled cartpole policy is evaluated for correctness using Z3 SMT Solver.
- **Chapter 5** concludes the thesis and sums up the results. It also presents possible future directions to extend this work in the future.

Chapter 2

Background

2.1 Reinforcement Learning

2.1.1 Definitions

Reinforcement learning is the problem of getting an agent to act in an environment so as to maximize its cumulative rewards [39]. An agent performs actions; for example, a car navigating a street, Super Mario overcoming obstacles in a video game or a robot trying to stand. An agent chooses an action among a list of possible actions based on a strategy. The strategy is implemented through a policy π, which is a mapping from state to action : $\pi(s) = a$. At any particular state, Super Mario can choose to move left, right, jump high, crouch or stand still. The *environment* in which the agent acts takes the agent's current state and action as input and returns the agent's reward and next state.

The agent and the environment interact through a series of actions and obtained rewards over timesteps $t = 1, 2, \cdots, T$ as shown in Figure 2.1. During the process, the agent gains some experience acting in the environment. It tries to learn the optimal policy by choosing the next action in a manner that helps it learn the best policy. At time t, labelling the current state s_t, action taken a_t, and reward r_t, the *episode* or *trial* is fully described by the following sequence terminating in s_t: $(s_1, a_1, r_1, s_2, a_2, \cdots, s_t)$. A single record of this trail, from state s_t to s_{t+1} is often referred to as a *transition* : (s_t, a_t, s_{t+1}, r_t).

Markov Decision Processes (MDP) formally describe an environment for reinforcement learning. A Markov decision process is a tuple $(S, \mathcal{A}, P, \mathcal{R})$ where S is a finite set of states, $\mathcal{A}$ is a finite set of actions, $P(s, s')$ is the probability that action a taken in state s at time

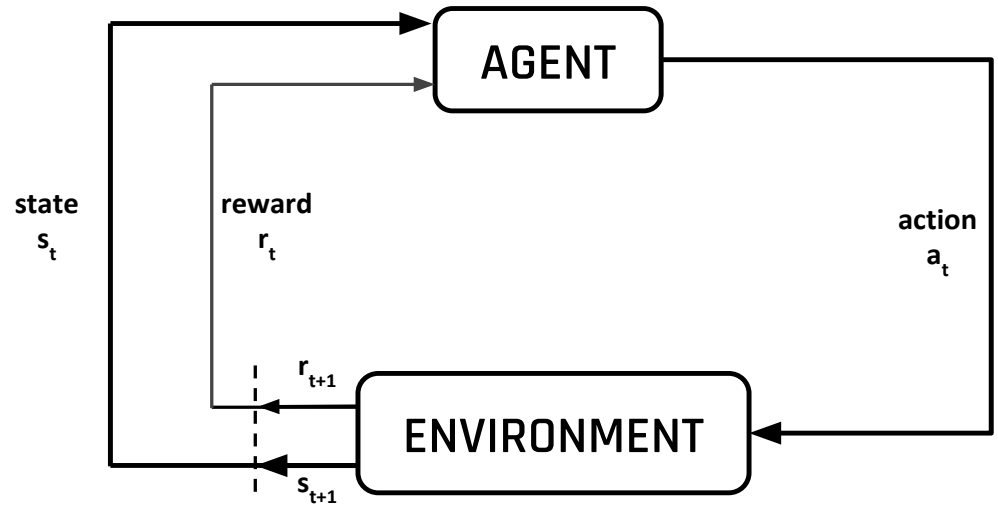

Figure 2.1: Overview of the Reinforcement Learning process

t will lead to state s' at time $t+1$ and $\mathcal{R}(s, a, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s' due to action a.

2.1.2 Value Function

Each state is associated with a value function $V(s)$ predicting the expected amount of future rewards the agent will be able to receive in this state by acting according to the the same policy. In other words, the value function quantifies how good a state is. The future *return* G_t is defined as the total sum of discounted rewards going forward. We can calculate the return G_t starting from time t:

$$G_t = r_t + \gamma r_{t+1} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.1}$$

where $\gamma \in [0, 1]$ discount rewards into the future.

The Value function $V^{\pi}(s)$ approximates the expected return if the agent is in state s at time t and is following policy π into the future :

$$V^{\pi}(s_t) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k}] \tag{2.2}$$

For deterministic environments, among all the possible value functions, there exists an optimal value function that has higher value than other functions for **all** states

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in S \tag{2.3}$$

The optimal policy π^* is the policy corresponding to the optimal value function

$$\pi^* = \operatorname*{argmax}_{\pi} V^{\pi}(s) \quad \forall s \in S \tag{2.4}$$

2.1.3 Q-Function

Q function is a function of a state-action pair and returns a real value $Q : S \times \mathcal{A} \rightarrow \mathbb{R}$. $Q_{\pi}(s, a)$ is a measure of the overall expected reward assuming the agent is in state s and performs action a, and then continues playing until the end of the episode following some policy π. Since $V^*(s)$ is the maximum expected total reward when starting from state s, it will be the maximum of $Q^*(s, a)$ over all possible actions. Thus,

$$V^*(s) = \max_{a} Q^*(s, a) \quad \forall s \in S \tag{2.5}$$

The optimal policy can be formed by choosing the action a that gives the maximum $Q^*(s, a)$ at every state s:

$$\pi^*(s) = \operatorname*{argmax}_{a} Q^*(s, a) \quad \forall s \in S \tag{2.6}$$

2.1.4 Bellman Equations

Bellman Equations use dynamic programming to provide a recursive definition for the optimal Q function. Function $Q^*(s, a)$ is the summation of immediate reward after performing action a in state s and the discounted expected future reward after transition to the next state s'

$$Q^*(s, a) = \mathcal{R}(s, a, s') + \gamma \mathbb{E}_{s'}[V^*(s')] \tag{2.7}$$

$$Q^*(s,a) = \mathcal{R}(s,a,s') + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s') \tag{2.8}$$

Using Equation 2.5,

$$V^*(s) = \max_a \left[\mathcal{R}(s,a,s') + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s') \right] \tag{2.9}$$

2.1.5 Deep Q Learning

A DQN [27] learns a deep neural network as a Q-function approximator. Q-Learning is based on the sample principle as Equation 2.9.

Algorithm 1: Basic Q-Learning Algorithm

```
Start with an initial guess Q_0(s,a) ∀s ∈ S, ∀a ∈ A
The agent begins at an initial state s
for k = 1, 2, ··· till convergence do
    Sample action a
    Perform a to get to next state s'
    if s' is terminal then
        target = R(s,a,s')
        Sample new initial state s
    else
        target = R(s,a,s') + γ max_{a'} Q_k(s',a')
    Q_{k+1}(s,a) = (1 − α)Q_k(s,a) + α[target]
    s = s'
```

Algorithm 1 however is unable to deal with a large number of states and actions. It is also extremely memory and computation intensive and hence a deep neural network is often used to approximate $Q(s,a)$. To train these neural networks and to counter a potentially non-stationary Q-function target two solutions are often used:

- Experience Replay : A large number of transitions are placed in a buffer $\mathcal{B}$ and minibatches are sampled from this buffer to train the neural network. This makes

the input dataset distribution relatively stable for training. Random sampling allows datapoints closer to i.i.d..

- Target Network : Instead of training one neural network, two neural networks are used with parameters θ and $\tilde{\theta}$. The first network is mostly fixed and used to estimate Q values while the second is used to perform updates at every training iteration. After C iterations , θ is copied into $\tilde{\theta}$. This prevents the moving target estimation problem.

The complete DQN algorithm is presented next.

Algorithm 2: Deep Q Learning Algorithm

Initialize replay memory $\mathcal{B}$ to capacity N
Initialize the main Q-function neural network Q with random weights θ
Initialize the target Q function neural network Q' with weight $\tilde{\theta} = \theta$
for *episode* $= 1, 2, \cdots$, M **do**
  for $t = 1, 2, \cdots$, T **do**
    With probability ϵ select a random action a_t
    Else select $a_t = \underset{a}{\text{argmax}}\, Q(s_t, a; \theta)$
    Execute action a_t in the environment and observed reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$
    Sample random minibatch of transitions from $\mathcal{B}$ as (s_j, a_j, r_j, s_{j+1})
    if *episode terminates at step* $j + 1$ **then**
      set $y_j = r_j$
    else
      $y_j = r_j + \gamma \max_{a'} Q'(s_{j+1}, a'; \widetilde{\theta})$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to network parameters θ
    Every C steps reset $Q' = Q$

2.2 Bayesian Additive and Regressive Tree Model

Primarily the BART model [14] consists of two parts: a sum-of-trees model and a regularization prior on the parameters of that model.

2.2.1 Sum of Trees model

BART Trees consist of interior nodes with decision rules and terminal nodes (leaves) with a number / parameter value. $\boldsymbol{\mu} = \{\mu_1, \mu_2, \ldots, \mu_b\}$ represents a set of leaf values associated with each of the b terminal nodes of a tree T. The decision rules in the nodes are typically based on the single components of $x = (x_1, \ldots, x_p)$ and are of the form $\{x_i \leq c\}$ vs $\{x_i > c\}$ for continuous x_i. Every input x value is associated with a single leaf of T by traversing through decision rules from top to bottom. The μ_i value associated with the selected leaf is the predicted output value. For a given T and $\boldsymbol{\mu}$. In the following sections, we use $g(x; T, \boldsymbol{\mu})$ to denote the function which assigns a $\mu_i \in \boldsymbol{\mu}$ to x. Hence, assuming homoschedastic data,

$$Y = g(x; T, \boldsymbol{\mu}) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{2.10}$$

The conditional mean of Y given x, $E(Y|x)$ equals the terminal node parameter μ_i assigned by $g(x; T, \boldsymbol{\mu})$. Now the sum-of-trees model can be written as

$$Y = \sum_{j=1}^{m} g(x; T_j, \boldsymbol{\mu}_j) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{2.11}$$

where for each binary regression tree T_j and its associated leaf parameters $\boldsymbol{\mu}_j$, $g(x; T_j, \boldsymbol{\mu}_j)$ is the function assigning $\mu_{ij} \in \boldsymbol{\mu}_j$ to x.

2.2.2 Regularization Priors

The BART model imposes a prior over all the parameters of the sum-of-trees model, namely $(T_1, \boldsymbol{\mu}_1), \cdots, (T_m, \boldsymbol{\mu}_m)$ and σ. The specifications of prior effectively regularize the fit by subduing the individual tree effects. Without the regularization, some larger trees could eclipse the output prediction hindering the advantages of the additive formulation.

Prior independence and symmetry

The specification of the regularization prior for BART is immensely unraveled by restricting attention to priors for which

$$\begin{aligned} p((T_1, \boldsymbol{\mu}_1), \ldots, (T_m, \boldsymbol{\mu}_m), \sigma) &= \left[\prod_j p(T_j, \boldsymbol{\mu}_j)\right] p(\sigma) \\ &= \left[\prod_j p(\boldsymbol{\mu}_j | T_j) p(T_j)\right] p(\sigma) \end{aligned} \tag{2.12}$$

and

$$p(\boldsymbol{\mu}_j | T_j) = \prod_i p(\mu_{ij} | T_j), \tag{2.13}$$

where $\mu_{ij} \in \boldsymbol{\mu}_j$. Under such priors, the tree components $(T_j, \boldsymbol{\mu}_j)$ are independent of each other and of σ, and the leaf parameters of every tree are independent. Equation 2.12 simplifies the prior specification problem to $p(T_j), p(\mu_{ij}|T_j)$ and $p(\sigma)$.

The T_j prior

The prior $p(T_j)$ is specified by three aspects:

(i) the probability that a node at depth d $(= 0, 1, 2, \ldots)$ is non-terminal is given by

$$\alpha(1+d)^{-\beta}, \qquad \alpha \in (0,1), \beta \in [0, \infty), \tag{2.14}$$

(ii) the distribution on the splitting variable assignments at each interior node, and

(iii) the distribution on the splitting rule assignment in each interior node, conditional on the splitting variable.

The default of uniform prior on available variables for (ii) and uniform prior on the discrete set of available splitting values for (iii) are chosen.

For the a sum-of-trees model, especially with m large, the regularization prior keeps the individual tree components small, which is desirable. Generally $\alpha = 0.95$ and $\beta = 2$ is used.

The $\mu_{ij}|T_j$ prior

The prior for μ_{ij} is set to zero $\mu_\mu = 0$ and σ_μ is chosen such that $k\sqrt{m}\sigma_\mu = 0.5$ for a suitable value of k, leading to

$$\mu_{ij} \sim N(0, \sigma_\mu^2) \qquad \text{where } \sigma_\mu = 0.5/k\sqrt{m}. \tag{2.15}$$

This prior shrinks the tree parameters μ_{ij} toward zero, limiting the effect of the individual trees by keeping them small. As k and/or the number of trees m is increased, this prior will become tighter and apply larger shrinkage to the μ_{ij}'s. BART researchers found k between 1 and 3 yielded good results with 2 being the default choice.

The σ prior

A conjugate prior is used for $p(\sigma)$: the inverse chi-square distribution $\sigma^2 \sim \nu\lambda/\chi^2_\nu$. The hperparameters ν and λ need to assign substantial probability to all the plausible values of σ while avoiding overconcentration and overdispersion. The prior degree of freedom ν and scale λ can be approximated using an estimate $\hat{\sigma}$ of σ. Two choices for $\hat{\sigma}$ are (1) $\hat{\sigma}$ can be computed as the sample standard deviation of Y (2) the "linear model" specification, in which $\hat{\sigma}$ is the residual standard deviation from a least squares linear regression of Y on the original X's. Then ν can be sampled between 3 and 10 to get an appropriate shape, and a value of λ can be sampled so that the qth quantile of the prior on σ is located at $\hat{\sigma}$, that is, $P(\sigma < \hat{\sigma}) = q$. Values of q such as 0.75, 0.90 or 0.99 can be used to center the distribution below $\hat{\sigma}$.

2.2.3 Training BART Models

Given the observed data y, our Bayesian setup induces a posterior distribution

$$p((T_1, \boldsymbol{\mu}_1), \ldots, (T_m, \boldsymbol{\mu}_m), \sigma|y) \tag{2.16}$$

on all the unknowns that determine a sum-of-trees model. To sample from this posterior, BART used something similar to a Gibbs sampler. It took $T_{(j)}$ be the set of all trees in the sum *except* T_j. $\boldsymbol{\mu}_{(j)}$ is defined similarly. Observe that $T_{(j)}$ is now a set of $m-1$ trees. The Gibbs sampler here entails m successive draws of $(T_j, \boldsymbol{\mu}_j)$ conditional on $(T_{(j)}, \boldsymbol{\mu}_{(j)}, \sigma)$:

$$(T_j, \boldsymbol{\mu}_j)|T_{(j)}, \boldsymbol{\mu}_{(j)}, \sigma, y, \tag{2.17}$$

$j = 1, \ldots, m$, followed by a draw of σ from the full conditional:

$$\sigma | T_1, \ldots, T_m, \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_m, y. \tag{2.18}$$

This algorithm is referred to as Backfitting Makov Chain Monte Carlo and σ in (2.18) can be drawn from an inverse gamma distribution.

Drawing $(T_j, \boldsymbol{\mu}_j)$ m times in (2.17) can be performed through the following steps. First, observe that the conditional distribution $p(T_j, \boldsymbol{\mu}_j | T_{(j)}, \boldsymbol{\mu}_{(j)}, \sigma, y)$ depends on $(T_{(j)}, \boldsymbol{\mu}_{(j)}, y)$ only through

$$R_j \equiv y - \sum_{k \neq j} g(x; T_k, \boldsymbol{\mu}_k), \tag{2.19}$$

the n-vector of partial residuals based on a fit that excludes the jth tree. Thus, the m draws of $(T_j, \boldsymbol{\mu}_j)$ given $(T_{(j)}, \boldsymbol{\mu}_{(j)}, \sigma, y)$ in (2.17) are equivalent to m draws from

$$(T_j, \boldsymbol{\mu}_j) | R_j, \sigma, \qquad \text{for } j = 1, \ldots, m \tag{2.20}$$

Equation (2.20) is formally equivalent to the posterior of the single tree model $R_j = g(x; T_j, \boldsymbol{\mu}_j) + \epsilon$ where R_j plays the role of the target data y. Because a conjugate prior was used for $\boldsymbol{\mu}_j$,

$$p(T_j | R_j, \sigma) \propto p(T_j) \int p(R_j | \boldsymbol{\mu}_j, T_j, \sigma) p(\boldsymbol{\mu}_j | T_j, \sigma) \, d\boldsymbol{\mu}_j \tag{2.21}$$

can be obtained in closed form up to a normalizing constant. Each draw from (2.20) can be carried in two consecutive steps as

$$T_j | R_j, \sigma, \tag{2.22}$$
$$\boldsymbol{\mu}_j | T_j, R_j, \sigma. \tag{2.23}$$

The draw of $\boldsymbol{\mu}_j$ in (2.23) is a set of independent draws of the leaf μ_{ij}'s from a normal distribution. The draw of $\boldsymbol{\mu}_j$ enables the calculation of the next residual R_{j+1} which is then used for the next draw of T_j. The chain is initialized with m single node trees, and then iterations are repeated until it convergences within satisfication.

2.3 Conditional Sequential Monte Carlo Filters

Particle Filters or Sequential Monte Carlo Filters (SMC) provides sampling based approximations of a sequence of posterior distributions over parameter vectors which increase in dimension, and allow fast efficient inference in complex dynamic statistical models. Particle Filters sample a random set of values (called *particles*) from a parameter which are propogated over time to track the posterior distribution of the parameter at each point in time. Every particle is scored commensurate with its posterior probability and the weights are used to then resample particles and allow for the final posterior approximation. We now gradually build up the final algorithm for Particle Filtering from the basics of Importance Sampling.

2.3.1 Importance Sampling

Importance Sampling is a Monte Carlo intergration technique often used to approximate expected values of random variables. The expected value of a random variable is generally computed by averaging over samples drawn from the random variable. Importance Sampling is based on the same idea of but samples are drawn from a distribution different from the true distribution which is called the *importance distribution*. If the true distribution is difficult to sample from, the importance distribution comes in very handy. To correct for the fact that the samples were drawn from the importance distribution and not the true distribution, scores or weights are assigned to the sampled values which reflect the difference between the importance and target distribution. The final estimate is then a weighted average of those randomly sampled values.

Given a function f of a random variable Y which is distributed according to a probability distribution p, the expected value of f is defined as:

$$\mathbb{E}_p[f(Y)] = \int f(y)p(y)dy \tag{2.24}$$

If $f(y) = y$, equation 2.24 results in computing the mean of Y. If the analytical solution of integral is not tractable, we turn to the Monte Carlo approximation

In Algorithm 3, the samples $y^{(i)}$ are termed as *particles*. As the number of particles N approaches infinity, according to the law of large numbers, the Monte Carlo approximation will converge to the true expected value [32]. If the distribution p (from which the particles were sampled) in Algorithm 3 is intractable, importance sampling is used to sample from

Algorithm 3: Standard Monte Carlo integration for $\mathbb{E}_p[f(Y)]$

for $i = 1, \cdots, N$ **do**
  sample $y^{(i)} \sim p(y)$; // Sample
Compute the sample average to obtain Monte Carlo approximation of the expected value $E^{MC} = \frac{1}{N}\sum_{i=1}^{N} f(y^{(i)})$; // Estimate

an arbitrary distribution q. Importance sampling is based on the basic algebric identity $a = \frac{a}{b} \times b$. The fundamental importance sampling identity is derived as

$$\mathbb{E}_p[f(Y)] = \int \frac{p(y)}{q(y)} q(y) f(y) dy = \mathbb{E}_q[w(Y)f(Y)] \tag{2.25}$$

where the importance weight is defined as $w(y) = \frac{p(y)}{q(y)}$.

The expected value of $f(Y)$ under the target distribution p is same as the expected value of $w(Y)f(Y)$ under the importance distribution q. The importance distribution allows for convenient sampling and more efficient estimation. The importance distribution however must fulfil the condition that if $f(y)p(y) \neq 0$ then $q(y) > 0$ i.e. when p assigns non-zero probability to a value y, q should also do so; $q(y) > 0$ when $p(y) > 0$.

The following algorithm demonstrates the importance sampling method

Algorithm 4: Importance Sampling (IS) for an expected value $\mathbb{E}_p[f(Y)]$

for $i = 1, \cdots, N$ **do**
  sample $y^{(i)} \sim q(y)$; // Sample
for $i = 1, \cdots, N$ **do**
  compute the importance weight $w^{(i)} = \frac{p(y^{(i)})}{q(y^{(i)})}$; // Weight
Compute a weighted average to obtain the IS estimate $E^{IS} = \frac{1}{N}\sum_{i=1}^{N} w^{(i)} f(y^{(i)})$

2.3.2 Sequential Importance Sampling

For our online tree model, we need to infer unknown parameters of our statistical model after each new observation is seen. Given a sequence of observations $y_{1:t} = (y_1, y_2, \cdots, y_t)$, we want to compute a sequence of posterior distributions $p(\theta|y_1), p(\theta|y_{1:2}), \cdots, p(\theta|y_{1:t})$. θ

denotes a vector of parameters. Approximating the posterior distribution $p(\theta|y_{1:t})$ with importance sampling, requires an importance distribution $q_t(\theta)$ for sampling with importance weights $w_t^{(i)} = \frac{p(\theta^{(i)}|y_{1:t})}{q_t(\theta^{(i)})}$.

Following the algorithm above, at every timestep, we need to generate a new important sample and browse the entire history of observations to calculate the importance weights. As more timesteps pass, we would need to sample over the complete trajectory of the state from $t = 1$ and the size of the importance samples keeps getting larger.

Sequential importance sampling (SIS) utilizes information from previous observations and samples to help in incremental importance sampling; requiring only a fixed computational cost for every new observation(timestep) .

SIS allows importance weights to be calculated incrementally, by multiplying the importance weight at the previous timestep $t-1$ with an incremental weight update $a_t^{(i)}$. We can rewrite the importance weights as

$$w_t^{(i)} = \frac{p(\theta^{(i)}|y_{1:t})q_{t-1}(\theta^{(i)})}{p(\theta^{(i)}|y_{1:t-1})q_t(\theta^{(i)})} \frac{p(\theta^{(i)}|y_{1:t-1})}{q_{t-1}(\theta^{(i)})} = a_t^{(i)} w_{t-1}^{(i)} \tag{2.26}$$

where the incremental weight update is defined as

$$a_t^{(i)} = \frac{p(\theta^{(i)}|y_{1:t})}{p(\theta^{(i)}|y_{1:t-1})} \frac{q_{t-1}(\theta^{(i)})}{q_t(\theta^{(i)})} \tag{2.27}$$

Now to obtain the final SIS algorithm, our observations are conditionally independent given the parameters and each of the posteriors can be expressed through Bayes Theorem as

$$p(\theta|y_{1:t}) = \frac{p(\theta)\prod_{i=1}^{t} p(y_i|\theta)}{p(y_{1:t})} \tag{2.28}$$

Using this, the LHS in equation 2.27 is simplified as

$$\frac{p(\theta^{(i)}|y_{1:t})}{p(\theta^{(i)}|y_{1:t-1})} = p(y_t|\theta^{(i)})p(y_t|y_{1:t-1}) \tag{2.29}$$

Using the same importance distribution over all timesteps $q_t(\theta) = q_{t-1}(\theta) = q(\theta)$, the RHS ratio in 2.27 evaluates to $\frac{q_{t-1}(\theta^{(i)})}{q_t(\theta^{(i)})} = 1$. The incremental weight update reduces to $a_t^{(i)} = p(y_t|\theta^{(i)})p(y_t|y_{1:t-1})$.

Algorithm 5: Sequential Importance Sampling (SIS) for parameters θ

for $i = 1, \cdots, N$ **do**
  sample $\theta^{(i)} \sim q(\theta)$; // Initialize
  compute normalized weights $W_0^{(i)} \propto \frac{p(\theta)}{q(\theta)}$ with $\sum_{i=1}^{N} W_0^{(i)} = 1$
for $t = 1, \cdots, T$ **do**
  for $i = 1, \cdots, N$ **do**
    compute $W_t^{(i)} \propto p(y_t|\theta^{(i)}) W_{t-1}^{(i)}$ with $\sum_{i=1}^{N} W_t^{(i)} = 1$; // Reweight
  compute the self normalized SIS estimate $E_t^{SIS} = \sum_{i=1}^{N} W_t^{(i)} f(\theta^{(i)})$; // Estimate

If importance weights are self normalized, the term $p(y_t|y_{1:t-1})$ can be ignored resulting in the final SIS scheme shared below

Sequential importance sampling is computationally efficient since we can approximate the relevant distributions sequentially without revisiting all previous observations or redrawing the entire importance sample. However, the performance of SIS detoriates over time and after some iterations, only one particle has non-negligible weight.

To counter the "weight degeneracy" problem, particle filtering algorithms have a resampling step, where particles are sampled with replacement from the set of all particles with the probability of selection being proportional to the importance weights. After resampling, the weights are reset or equalized.

2.3.3 Particle Filters / Sequential Monte Carlo Models

For our distillation problem, we are specifically interested in state space models. State space models are defined by an observable time series $y_{1:t}$ through a sequence of latent states $\phi_{0:t}$. The latent states in this thesis refer to our intermediate trees in the process to the final distilled tree. In state space models, we have primary assumptions. First, it is assumed that each observation y_t depends only on the current state ϕ_t and the observations are conditionally independent given the states ϕ_t:

$$p(y_{1:T}|\phi_{0:T}) = \prod_{t=1}^{T} p(y_t|\phi_t) \tag{2.30}$$

Secondly it is assumed that hidden states change over time according to the first-order

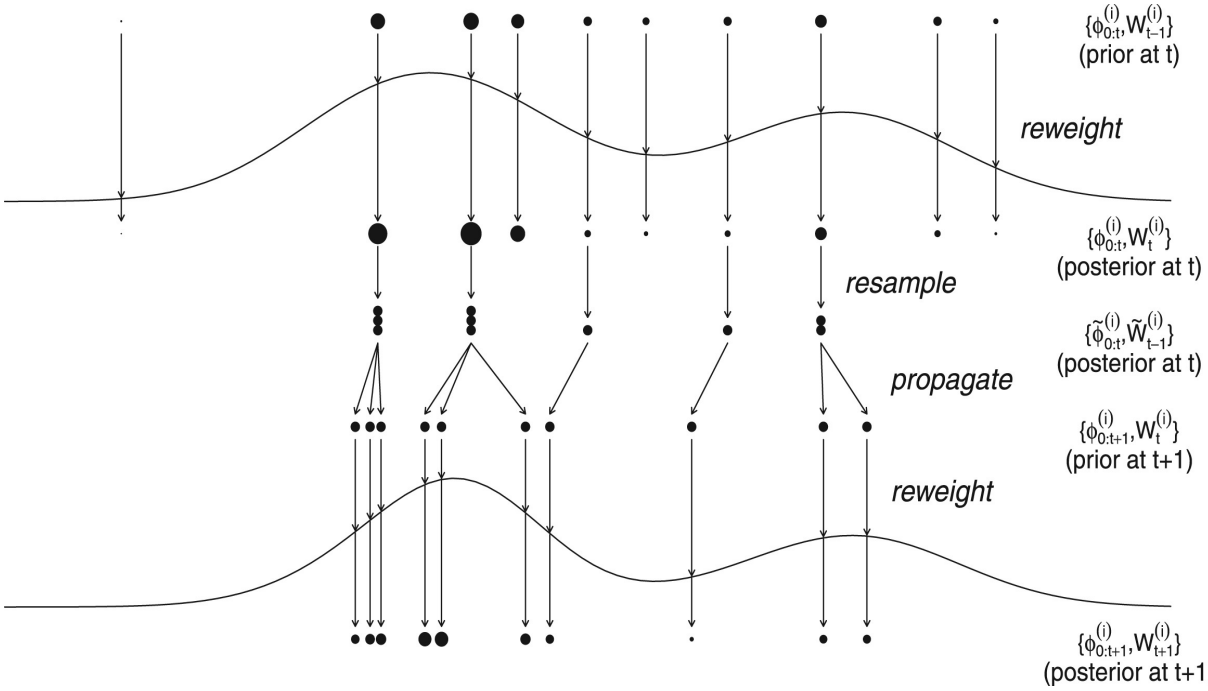

Figure 2.2: Schematic representation of a generic particle filter (from [37]). At time t, a set of weighted particles $\left\{\phi_{0:t}^{(i)}, W_{t-1}^{(i)}\right\}$ representing the prior distribution at t. In this thesis, each particle represents a tree and the weight describes the goodness of fit to an input dataset. Each particle $\phi_{0:t}^{(i)}$ is a multidimensional variable which represents the whole path of the particle state from time 0 up to the current time point t. The location of the dots in the graph represent $\phi_t^{(i)}$, the value of the state at the current time point. The size of each dot reflects the weight $W_{t-1}^{(i)}$ (prior at t). In the *reweight* step, the weights are updated to $W_t^{(i)}$ partly as a function of $p(y_t|\phi_t^{(i)})$, the likelihood of observation y_t according to each sampled state value $\phi_t^{(i)}$ (solid line). The resulting set of weighted particles $\left\{\phi_{0:t}^{(i)}, W_t^{(i)}\right\}$ approximates the posterior distribution (posterior at t) of the state paths. The *resampling step* duplicates values $\phi_{0:t}^{(i)}$ with higher weights $W_t^{(i)}$, and eliminates those with low weights, resulting in the set of uniformly weighted particles $\left\{\widetilde{\phi}_{0:t}^{(i)}, \widetilde{W}_t^{(i)} = 1/N\right\}$ which is approximately distributed according to the posterior (second posterior at t). In the *propagate step*, values of states $\phi_{t+1}^{(i)}$ at the next time point are sampled and added to each particle to account for state transitions, forming a prior distribution for time $t+1$ (prior at $t+1$). Thus, at each new time point, the particles grow in dimension because the whole path of the latent state now incorporates the new time point as well. The particles are then reweighted in response to the likelihood of the new observation y_{t+1} to approximate the posterior distribution at $t+1$ (posterior at $t+1$), etc.

Markov Process, such that the current state depends only on the state at the immediately preceding timestep :

$$p(\phi_{0:T}) = p(\phi_0) \prod_{t=1}^{T} p(\phi_t|\phi_{t-1}) \tag{2.31}$$

The posterior distribution over the hidden states can now be written with these assumptions as :

$$p(\phi_{0:T}|y_{1:T}) = \frac{p(\phi_0) \prod_{t=1}^{T} p(y_t|\phi_t)p(\phi_t|\phi_{t-1})}{p(y_{1:T})} \tag{2.32}$$

Now the posteriors can be recursively calculated as:

$$p(\phi_{0:t}|y_{1:t}) = \frac{p(y_t|\phi_t)p(\phi_t|\phi_{t-1})}{p(y_t|y_{1:t-1})} p(\phi_{0:t-1}|y_{1:t-1}) \tag{2.33}$$

Estimating the hidden states $\phi_{0:t}$ can be seen as estimating a vector of parameters θ whose dimensions increase at each timestep t , such that, at time t, we estimate $\theta = \phi_{0:t}$, and at time $t+1$ another dimension is added to the parameter vector and now we estimate $\theta = \phi_{0:t+1}$. We could draw a new importance sample $\{\phi_{0:t}^{(i)}\}$ at each time t using Importance Sampling. Using Sequential Importance Sampling, the importance sample can be built up incrementally, starting at time $t = 0$ with a sample $\{\phi_0^{(i)}\}$, then sampling values $\{\phi_1^{(i)}\}$ at time 1 conditional on the sample at time 0, then adding sampled values $\{\phi_2^{(i)}\}$ at time 2 conditional on the sample at time 1, etc. Formally, this means we define the importance distribution at time t as

$$q_t(\phi_{0:t}) = q_t(\phi_t|\phi_{0:t-1})q_{t-1}(\phi_{0:t-1}) \tag{2.34}$$

Using this conditional importance distribution, and noting that $q_{t-1}(\phi_{0:t}) = q_{t-1}(\phi_{0:t-1})$, the right-hand ratio in 2.27 simplifies to

$$\frac{q_{t-1}(\phi_{0:t})}{q_t(\phi_{0:t})} = \frac{1}{q_t(\phi_t|\phi_{0:t-1})}$$

Combining this with 2.33, we can write the incremental weight update as

$$a_t^{(i)} = \frac{p(y_t|\phi_t^{(i)})p(\phi_t^{(i)}|\phi_{t-1}^{(i)})}{p(y_t|y_{1:t-1})q_t(\phi_t^{(i)}|\phi_{0:t-1}^{(i)})} \tag{2.35}$$

The term $p(y_t|y_{1:t-1})$ can be ignored if normalized importance weights are used. Now we finally describe the (approximately) constant computational cost algorithm capable of updating the weights of the particles without revisiting the previous observations and hidden states

Algorithm 6: Generic Particle Filtering algorithm to approximate a sequence of posterior distributions $p(\phi_{0:1}|y_1), p(\phi_{0:2}|y_{1:2}), \cdots, p(\phi_{0:t}|y_{1:t})$

For $i = 1, \cdots, N$, sample $\widetilde{\phi}_0^{(i)} \sim q(\phi_0)$ and compute the normalized importance weights $\widetilde{W}_0^{(i)} \propto \frac{p(\phi_0^{(i)})}{q(\widetilde{\phi}_0^{(i)})}$ with $\sum_{i=1}^N \widetilde{W}_t^{(i)} = 1$.

for $t = 1, \cdots, T$ **do**

  for $i = 1, \cdots, N$ **do**

    sample $\phi_t^{(i)} \sim q_t(\phi_t|\widetilde{\phi}_{0:t-1}^{(i)})$; // Propagate

    add this new dimension to the particles, setting $\phi_{0:t}^{(i)} = (\widetilde{\phi}_{0:t-1}^{(i)}, \phi_t^{(i)})$

  for $i = 1, \cdots, N$ **do**

    compute normalized weights $W_t^{(i)} \propto \frac{p(y_t|\phi_t^{(i)})p(\phi_t^{(i)}|\phi_{t-1}^{(i)})}{q_t(\phi_t^{(i)}|\phi_{0:t-1}^{(i)})}\widetilde{W}_{t-1}^{(i)}$; // Reweight

    with $\sum_{i=1}^N W_t^{(i)} = 1$

  Compute the required estimate $E_t = \sum_{i=1}^N f(\phi_{0:t}^{(i)})W_t^{(i)}$; // Estimate

  If $N_e \leq cN$, resample $\{\widetilde{\phi}_{0:t}^{(i)}\}$ with replacement from $\{\phi_{0:t}^{(i)}\}$ using the normalized weights $W_t^{(i)}$ and set $\widetilde{W}_t^{(i)} = 1/N$ to obtain a set of equally weighted particles $\{\widetilde{\phi}_t^{(i)}, \widetilde{W}^{(i)} = 1/N\}$ else set $\{\widetilde{\phi}_{0:t}^{(i)}, \widetilde{W}_t^{(i)}\} = \{\phi_{0:t}^{(i)}, W_t^{(i)}\}$; // Resample

Conditional Sequential Monte Carlo Algorithm is almost exactly like the particle filtering algorithm; the only difference being that the parent tree or particle (from which all the particles are sampled) is kept alive throughout the algorithm.

Chapter 3

The Distillation Process

We formally describe the architecture of our distillation process in the next section. Here we give an informal overview of the architecture depicted in Figure 3.1. The Distillation process can be split into three phases. In the first phase, the deep neural network policy controls the agent policy and interacts with the environment over multiple rollouts. Once a fixed size data buffer is filled from the data gathered from those interactions, the data is sent to the tree process for constructing the tree to fit the given data. In the second phase, the parent tree and its altered clones try to fit the data to their best capabilities. Each tree fits the Q-value for a single predetermined action by taking the state of the agent as input and once the training algorithm alters the forest to fit the data, the buffer is emptied. In the third and final phase, the tree controls the agent in a number of games to assess the distilled policy knowledge learnt by the tree. This setup is commonly known as an *active play* setup.

3.1 QBSP Forest

To more formally describe our distillation process, consider an environment where we represent the agent state s at any step by a d-dimensional vector $s = (x_1, ..., x_d) \in \mathbb{R}^d$. The agent can perform one of m discrete actions $a_1, ..., a_m \in \mathcal{A}$. As indicated in Figure 3.1, a Q-value is generated for every action $a_i \in \mathcal{A}$ possible in a given state s denoted as $\widetilde{Q}(s) = \{\ Q(s, a_i) \mid \forall a_i \in \mathcal{A}\}$. The action to be performed is selected as

$$a = \underset{\forall a_i \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a_i) \tag{3.1}$$

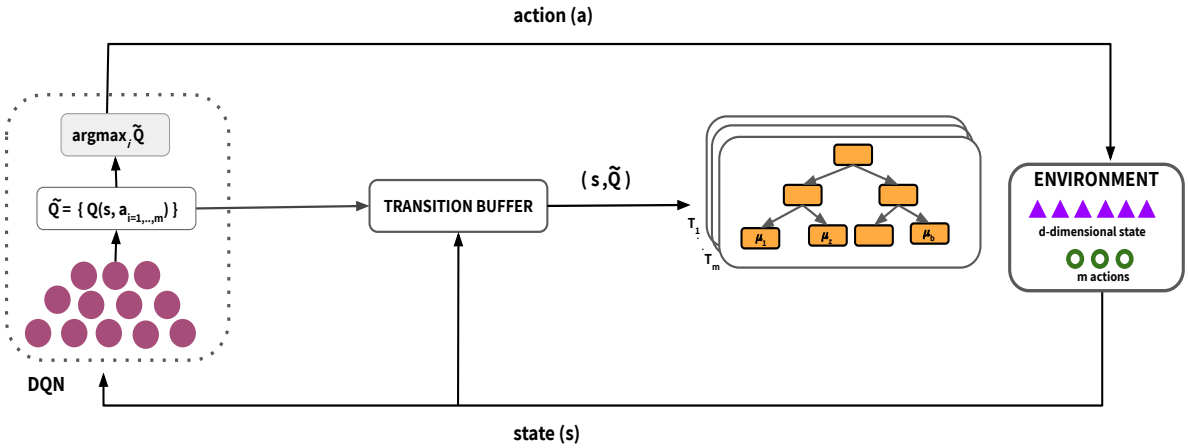

Figure 3.1: Architecture of our Distillation Process

The transition between the agent and the environment are recorded as tuples of the form of $(s, \widetilde{Q}(s)) \in \mathbb{R}^d \times \mathbb{R}^m$. The supervised regression task is then defined from $s \rightarrow \widetilde{Q}(s)$.

An independent collection of trees $T_1, .., T_m$ is created to form the forest. Each tree T_i is responsible for regressing the Q-value $Q(\cdot, a_i)$ of a single fixed action $a_i \in \mathcal{A}$. A tree T consists of a set of internal nodes containing decision rules and a set of terminal nodes (leaves) containing parameter values as shown in Figure 1.1. Let $\boldsymbol{\mu} = \{\mu_1, \mu_2, ..., \mu_b\}$ denote the set of parameter values associated with each of the b leaf nodes. Every input is associated with a single leaf node $z \in T$ by a sequence of decision rules from top to bottom and is then assigned the value μ_z corresponding to this leaf node z. The function $g(\cdot)$ encapsulates this tree traversal.

$$Q(s, a_i) = g(s; T_i, \boldsymbol{\mu_i}) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{3.2}$$

where for each regression tree T_i and its associated leaf node parameters $\boldsymbol{\mu_i}$, $g(s; T_i, \boldsymbol{\mu_i})$ is the function that assigns a leaf node z and its parameter $\mu_z \in \boldsymbol{\mu_i}$ to the input s. ϵ denotes the observation variance. Once we predict Q-values for all the m actions, we use Equation 3.1 to predict the next action.

3.1.1 The BSP Tree

Decision Trees and Random Forests have a rich volume of past research. Generally a node in a decision tree is split in two steps. First, the feature to be split is decided and then

a location along the chosen feature is finalized for the splitting. The algorithm generally follows a greedy strategy to optimize for some metric (generally related to information gain). Binary Decision Trees such as those used by [6], require the entire dataset to be loaded in memory to compute this gain metric for each and every split. This prevents distillation in an online fashion.

We quickly recap the Binary Space Partitioning (BSP) Tree as described in [18, 19]. In the BSP Tree, the levels of the tree are recursively generated through a series of cutting hyperplanes and each cutting hyperplane is parallel to the $d-2$ dimensions it does not cut through. The cutting hyperplanes form a set of partitions in the data space. Each such partition is typically represented by a convex polytope $\square \subset \mathbb{R}^d$. For any arbitrary pair of dimensions (d_1, d_2) in the d-dimensional input state s, Algorithm 7 presents the selection and splitting process for a leaf node $k^* \in T_i$ given a dataset $(s, Q(s, a_i))^{1:N}$ of size N .

The algorithm can be summarized in the following steps. First to expand the tree and generate new leaves, an existing leaf node is sampled in proportion to $\left\{\sum_{d_1,d_2} \mathbb{P}^k_{(d_1,d_2)}\right\}_{k=1}^{b}$; where for every pair of dimension (d_1, d_2) from a total of d-dimensions. $\mathbb{P}^k_{(d_1,d_2)}$ denotes the perimeter of the projections of the input datapoints on dimensions (d_1, d_2) for leaf k. Once the node to split has been decided, a pair of dimensions are sampled (to create a cutting hyperplane) in proportion to the projection perimeters of the datapoints falling in that node.

Now that the node to cut and the dimensions of the cutting hyperplane have been finalized, the direction of the cutting hyperplane is to be decided. Figure 3.2 depicts the cutting process. An angle is chosen from $(0, \pi]$ with the probability density function in proportion to the length of the line segment $\boldsymbol{l}(\theta)$ onto which the hyperplane is projected. After choosing the angle, a random position u is chosen on the line segment. The cutting hyperplane is formed as a line passing through u and crossing the selected projection orthogonal to $\boldsymbol{l}(\theta)$ in the selected dimensions.

The cost to cut a node is sampled from the exponential distribution with the sum of the projection perimeters for all leaf nodes as its rate. If the cost exceeds a predefined budget (which we discuss later), the newly formed hyperplane is discarded.

Once the leafs are created with Algorithm 7, we can sample $(\mu_{leaf}, \sigma_{leaf})$ for each new leaf as $\sigma^2_{leaf} = \left(\frac{1}{\sigma^2_{parent}} + \frac{n}{\sigma^2}\right)^{-1}, \mu_{leaf} = \sigma^2_{leaf}\left(\frac{\mu_{parent}}{\sigma^2_{parent}} + \frac{\sum_i y_i}{\sigma^2}\right)$ where n denotes the number of points in the dataset falling in the newly created partition, σ^2 refers to the sample variance of these n datapoints and y_i denotes the regression output (Q values in our case) of the data reaching that leaf.

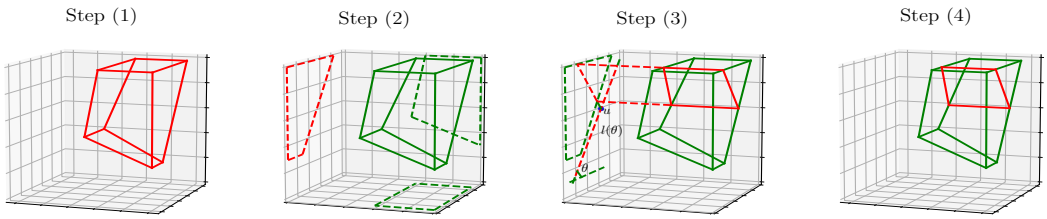

Figure 3.2: Diagrammatic representation of the cutting process of a BSP tree leaf from [18, 19]. The root node is represented by a three dimensional unit hypercube and the leaf partition is shown in red in Step (1). Step (2) shows all the two dimensional projections, red signifying the projection chosen for split in Step (3) .The generation of θ and u is shown as Step (3).

Algorithm 7: Sampling a leaf node to cut k^* and generating a cutting hyperplane to divide the selected leaf $H\left(k^*, (d_1^*, d_2^*), \theta, \boldsymbol{u}\right)$ as proposed in [19]

Input: dataset $(s, Q(s, a_i))^{1:N}$, leaf index $k = \{1, \cdots, b\}$

Output: cutting hyperplane $H\left(k^*, (d_1^*, d_2^*), \theta, \boldsymbol{u}\right)$, and cost c

for *every dim pair* $(d_1, d_2) \in \{(1,2), \cdots (d-1, d)\}, k = 1, \cdots, b$ **do**

  project datapoints $\{s\}^k \subset \{s\}^{1:N}$ that reach leaf k onto the (d_1, d_2) dimensions to get a projection $\{x_{d_1}, x_{d_2}\}^k \in \{s\}^k$;

  compute the convex hull on $\{x_{d_1}, x_{d_2}\}^k \in \{s\}^k$, denoted by $\mathbb{C}^k_{(d_1,d_2)}$;

  calculate the perimeter of $\mathbb{C}^k_{(d_1,d_2)}$, denoted by $\mathbb{P}^k_{(d_1,d_2)}$;

sample leaf k^* to cut proportional to the sum of projection perimeters $\left\{\sum_{d_1,d_2} \mathbb{P}^k_{(d_1,d_2)}\right\}_{k=1}^{b}$;

sample dimension pair (d_1^*, d_2^*) in proportion to the perimeters $\left\{\mathbb{P}^{k^*}_{(d_1,d_2)}\right\}_{\forall (d_1,d_2)}$

sample a direction $\theta \in (0, \pi]$, where the probability density function is proportional to the length of the line segment $\boldsymbol{l}(\theta)$, onto which $\mathbb{C}^{k^*}_{(d_1^*,d_2^*)}$ is projected (shown in Figure 3.2)

sample u uniformly on the projection of the convex hull $\mathbb{C}^{k^*}_{(d_1^*,d_2^*)}$ (shown in Figure 3.2);

create proposed cutting hyperplane $H\left(k^*, (d_1^*, d_2^*), \theta, u\right)$ as the straight line passing through u and crossing through the projection $\mathbb{C}^{k^*}_{(d_1^*,d_2^*)}$, orthogonal to $\boldsymbol{l}(\theta)$ creating two new leaves.

sample the cost $c \sim \text{Exp}\left(\sum_k \sum_{d_1,d_2} \mathbb{P}^k_{(d_1,d_2)}\right)$; If cost exceeds budget, reject the proposed cut.

3.1.2 Online Expansion of Node Partitions

In Algorithm 7, every node (including the *root*) is bounded by the convex hull of the points it has just observed (Figure 3.2).The original BSP Trees were not designed for online operation and hence did not include any way of expanding the partition boundaries in case incoming data lied outside root node boundaries. Reinforcement learning environments usually have a large number of state dimensions or state spaces with large dimension ranges which can be memory intensive. An effective way to overcome this constraint is to learn in an online setting where the training examples are presented as a stream of input data.

We add the flexibility to expand partition (*node*) boundaries to accommodate new datapoints (that lie outside the current node boundaries) while making sure that the partitions of their children are adapted consistently to the extended space. Algorithm 8 and Figure 3.3 explains this expansion process.

Algorithm 8: Online Expansion of a BSP Tree T

Input: tree node k, datapoints reaching node k : $\{s\}^k$

find the datapoints $\{s'\}^k \subset \{s\}^k$ outside the partition boundaries $\mathbb{C}$ of node k

for *every dim pair* $(d_1, d_2) \in \{(1,2), \cdots, (d-1, d)\}$ **do**

  compute projections $\{x_{d_1}, x_{d_2}\}^k \in \{s'\}^k$ onto dimensions (d_1, d_2);

  compute the new convex hull $\mathbb{C}^*_{(d_1,d_2)}$ on old boundary points and new projection points $\left\{\mathbb{C}_{(d_1,d_2)} \bigcup \{x_{d_1}, x_{d_2}\}^k\right\}$;

  compute the perimeter of $\mathbb{C}^*_{(d_1,d_2)}$ as $\mathbb{P}^*_{(d_1,d_2)}$;

extrapolate the line segment $\boldsymbol{l}(\theta)$ to cut the new boundary $\mathbb{C}^*$

expand the cutting hyperplane $H\left(k, (d_1^*, d_2^*), \theta, u\right)$ to mirror the line

recompute the *cost* $c^* \sim \text{Exp}\left(\sum_k \sum_{d_1,d_2} \mathbb{P}^k_{(d_1,d_2)}\right)$;

if *cost* $c^*_{\square}$ *within* budget **then**

  recurse over children of k

else

  find new cuts : $(\theta, \boldsymbol{u})$ on the newly computed convex hulls $\mathbb{C}^*$

If a new datapoint is observed, for any given tree, only one of the following three things can occur: (i) an existing leaf node is split to generate two children (ii) existing split boundaries are expanded to include the new datapoint (iii) a new parent split is generated above the current split (the split is reset). Cases (ii) and (iii) are depicted pictorially in

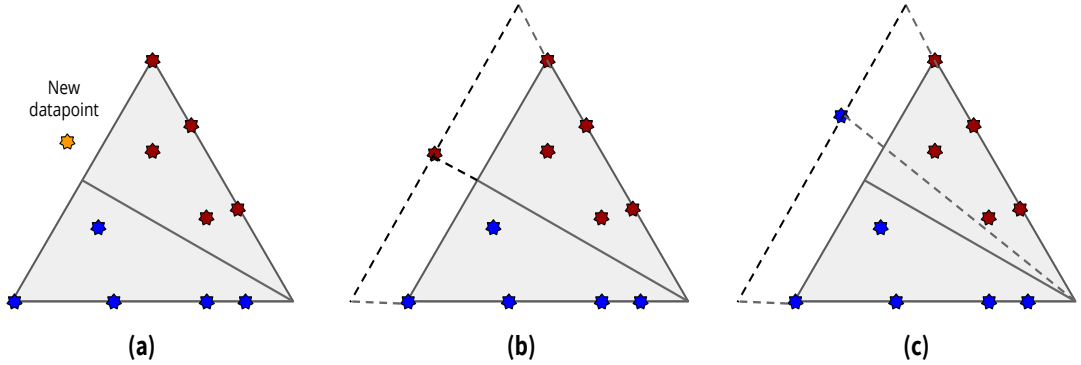

Figure 3.3: Online expansion of a BSP block. Diagram(a) plots the initial boundary of a BSP block when a new out-of-boundary datapoint appears. Now there are only two ways in which expansion can work. In (b), we simply extrapolate the original boundary line to accommodate the new point. In (c), we try different new boundary partitions and choose the one which minimizes the given loss metric

Figure 3.3. Case (i) was captured by Algorithm 7 earlier but Case (ii) and (iii) are handled by Algorithm 8.

If a new datapoint is observed that is outside the current boundary of the node, a new convex hull (boundary) is computed for every dimension-pair projection that includes the new datapoint. If the dimension-pair includes the cutting hyperplane, the dividing line is extrapolated with the sample angle θ for $l(\theta)$ and the corresponding hyperplane is expanded back in the original polytope to follow the line. The new cost with the new perimeters is sampled and if it exceeds the predefined budget, the cutting hyperplane and all the children of the node are discarded. A new dimension pair, cutting line and hyperplane are sampled from afresh following Algorithm 7.

3.2 Training Methodology

A fixed interval based approach $\{(\mathbb{B}_i, \mathbb{B}_{i+1})\}_{i=0}^{S}$ is implemented for QBSP training. At every stage $i = 0 \rightarrow S$, all the trees $T_1, ..., T_m$ are under the same budget constraints $(0, \mathbb{B}_i]$ and there might be zero, one or more cuts at every stage as long as the sum of the costs (refer Step 10 in Algorithm 7) are within the pre-allocated budget $\mathbb{B}_i$. The cuts generated at various stages are depicted geometrically in Figure 3.4 on the tree in Figure 3.5.

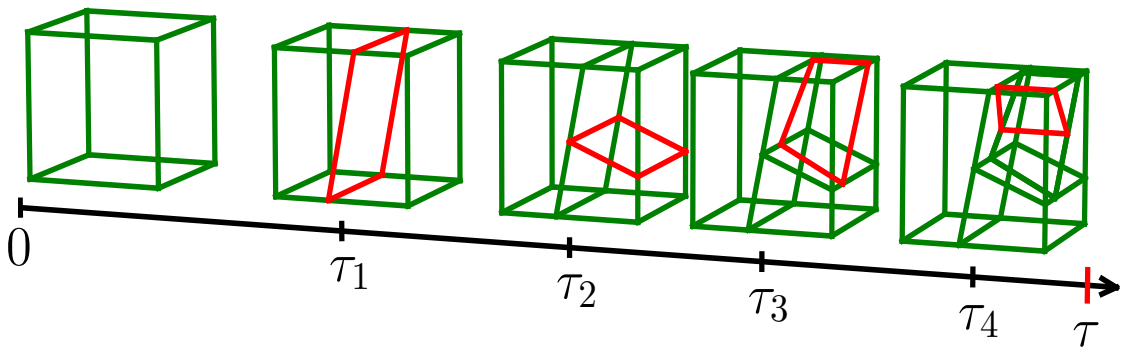

Figure 3.4: Depiction of one branch of a 3-dimensional tree process with total budget τ. Red denotes the new cutting hyperplane generated at that step. Source [18]

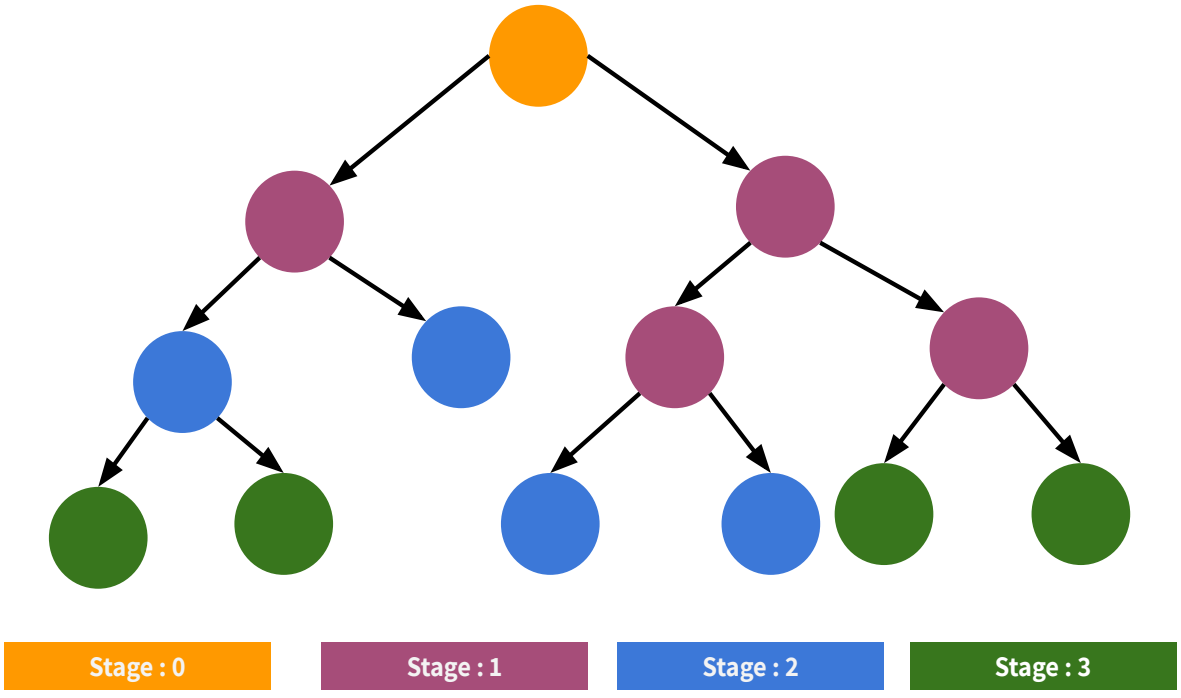

Figure 3.5: Depiction of the portion of trees generated over three stages in a single training iteration. Yellow represents the root node at stage 0 when we begin the training process. Violet represents the portion of the tree generated in stage 1, blue represents stage 2 and finally green represents the third and last stage

When a tree T is initialized, a portion of the tree is generated in every stage. The tree generated till stage i, denoted by T^i, is assigned a numeric value $\omega_i \in \omega_{1:S}$ that describes the *goodness of fit* of the tree till that stage to incoming data.

To get a complete understanding of the training algorithm, it is essential to study the Background section in detail. The Sequential Monte Carlo training process, as described earlier, is exactly like the particle filter method with the tree T being the high-dimensional variable whose posterior needs to be sampled. SMC differs from particle filter in that the original tree is kept alive (refer to line 5 of Algorithm 9) and compared to its clones until the very end, till it is replaced by the best performing mutation.

On an extremely high level, Algorithm 9 can be summarized as follows :

Given the original tree, proceed in a sequential top-down manner. At every given tree level, create R mutations of the tree. The winner of a particular level is used to create the R clones at the next level **(Step 4)**. Mutations at every level might include splitting a leaf into two nodes **(Step 7)**, destroying a node at that level (along with its children) and recomputing the boundary line or no change at all **(Step 9)**. All the mutations and the original tree compete to get the highest score on the training dataset. **(Step 16)** These scores are used to pick winners at every stage. The score is average of two metrics : (i) Improvement in Regression Accuracy on the training dataset over the base tree **(Step 11)** (ii) The relative sorted ordering of the predicted value over all other action-trees (or forests) **(Step 13)**.

The ordering metric has never been used for any reinforcement learning distillation process before and we summarize the motivation for it here. A score composed of regression accuracy alone leads to great regression performance and optimal MAE and RMSE errors (defined later). In our experiments, the distilled forest policy inspite of having impressive and minimal regression losses was unable to get high rewards in our test environments. This signified an important problem with Q-Value Distillation methods. For some crucial actions, the difference in Q-values for competing actions is often very minimal and regression error metrics proved insufficent to force the correct action from being chosen. Picking the wrong action can often lead to catastrophic failures in the game. Since we only pick the top action to perform irrespective of the difference in Q-values between the chosen action and the next best action, the ordering score builds this condition into the posterior

sampling problem.

Algorithm 9: The Ordered Sequential Monte Carlo training process for the j-th QBSP-tree T_j regressing over action a_j

Input: batch data $\mathcal{D} = (s, Q(s, a_j))^{1:N}$, no. of particles R, Q-values predicted by all other trees for dataset $\mathcal{D}$: $\hat{Q}_{\mathcal{D}} = \{Q^*(s, a_z)\}^{1:N}_{\forall z \in 1,..,j-1,j+1,..,m}$

Output: modified tree T_j^*, $\boldsymbol{\mu} \in T_j^*$

Initialization $i = 0$, $\mathbb{T}_0 =$ empty tree(root node), particle array $V[1 : R+1] = \mathbb{T}_0$, costs $c_i[1 : R+1] = 0$, leaf params $\boldsymbol{\mu}_i[1 : R] = 0$;

Expand parent tree T_j using **Algorithm 8** for $(s, Q(s, a_j))^{1:N}$, if needed

for $i = 1, \cdots, S$ **do**

  $V[1 : R] = \mathbb{T}_{i-1}$; `// make R clones of last successful tree`

  $V[R+1] = T_j^i$; `// the state of original tree at stage i`

  for $r = 1, \cdots, R$, ***if*** $\mathbb{B}_i < \sum_{i'=0}^{i-1} c_{i'}[r] < \mathbb{B}_{i+1}$ **do**

    using **Algorithm 7**, recursively obtain $H_i[r] = \{\hat{H}_\lambda\}_{\lambda=1}^{\Lambda} \in V[r]$, $\boldsymbol{\mu}_i[r] \in V[r]$, $c_i[r] = \sum_{\lambda=1}^{\Lambda}\{c_\lambda\}$ **until** $\sum_{i'=0}^{i} c_{i'}[r] > \mathbb{B}_{i+1}$; where $\Lambda =$ total number of cuts falling in $(\mathbb{B}_i, \mathbb{B}_{i+1}]$

    if $\Lambda = 0$ **then**

      set $H_i[r] = H_{i-1}[r]$, $\boldsymbol{\mu}_i[r] = \boldsymbol{\mu}_{i-1}[r]$, $c_i[r] = 0$;

  for $r = 1, \cdots, R+1$ **do**

    compute likelihood weight, $\omega[r] := \frac{prior(\boldsymbol{\mu}_i[r]) \cdot p(Q(s,a_j)|s, H_{1:i}[r], \boldsymbol{\mu}_{1:i}, \sigma^2)}{posterior(\boldsymbol{\mu}_i[r]) \cdot p(Q(s,a_j)|s, H_{1:i-1}[r], \boldsymbol{\mu}_{1:i-1}, \sigma^2)}$;

  for $r = 1, \cdots, R+1$ **do**

    compute ranking weight, $\varphi[r] = \boldsymbol{\rho}(V[r], \hat{Q}_{\mathcal{D}}, \mathcal{D})$, where $\boldsymbol{\rho}$ is the ranking loss function;

  for $r = 1, \cdots, R+1$ **do**

    normalize weights $W[r] := \frac{(\omega[r] + \alpha.\varphi[r])}{\sum_{r'=1}^{R}(\omega[r'] + \alpha.\varphi[r'])}$ where α is a mixing parameter;

  sample one particle $\mathbf{r} \propto W[r]$ as winner, $\mathbb{T}_i = V[\mathbf{r}]$

$T_j^* = \mathbb{T}_S$

Each tree T_j produces the Q-value $Q^*(s, a_j)$ for action a_j. Collectively all the trees reconstruct the original $\widetilde{Q}$ function and the prediction is denoted by $\widetilde{Q}^*$. The ranking function $\boldsymbol{\rho}$ used in Algorithm 9 computes the mean square error between the neural network ranking and the distillation tree ranking of the Q values in the dataset.

Prior Specification As discussed in the Background section, we need to set the prior distribution for σ^2 and $\{\boldsymbol{\mu}^i\}_{i=1}^m$ (leaf parameter array for all m trees). We follow the same prior and posterior specification as [19] i.e.

$$\sigma^2 \sim IGamma(1.5, \lambda)$$

, where given an estimate of the sample variance $\hat{\sigma}^2$, *lambda* is set to satisfy the equation $F(\hat{\sigma}^2; 1.5, \lambda) = 0.9$ where F represents the Inverse Gamma c.d.f. The leaf parameters μ are often stamdardized and hence the prior was chosen as

$$\mu \sim \mathcal{N}(0, \frac{1}{2\sqrt{m}})$$

Posterior Specification While the posterior of μ depends on the data (simple posterior mean calculation referred to after Algorithm 7) using conjugacy the full posterior conditional distribution of σ^2 is:

$$\sigma^2 \sim IGamma\left(\frac{3+N}{2}, \lambda + \frac{E}{2}\right)$$

Consistency BSP Trees are unique in the sense that they are self-consistent. Self-consistency is a property exhibited by a statistical process wherein restricting a process on a convex polygon $\square$ to any sub-region $\triangle \subseteq \square$, the resulting partitioning on the sub-region is distributed as if it is directly generated on $\triangle$. Mondrian Forests [34, 25] and their recent successor Binary Space Partitioning Forests [18, 19] exploit this property to create the generative process for their respective trees.They are self-consistent online forest methods that have constantly matched the accuracy of the state-of-the-art offline regression methods trained on the same dataset. The BSP Process [18] creates space partitions that are strictly more expressive than decision trees and as a result are generally much smaller than regular decision trees for the same accuracy. We use them as the base for our QBSP Forest structure.

3.3 Proof of Consistency

The proof for the consistency for the BSP Tree Process followed by the BSP Forests as presented in [18, 19].

Extension to QBSP Forests Applying Kolmogorov Extension Theorem, the extended BSP tree Process is self-consistent in d-dimensional ($d \geq 2$) space and maintains distributional invariance when restricting its domain from a convex polyhedron to a sub-domain.

Finally, the hyperplane restricted on the convex hull is distributed the same as if it was first partitioned on the $full$ polyhedron and then restricted attention to the convex hull since both the methods have identical equilibrium distribution in the MCMC algorithm.

Adding the ordering weight and online expansion does not change these fundamental characteristics.

Chapter 4

Experiments

4.1 Environment Description

4.1.1 OpenAI Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It contains a collection of test problems environments that can be to learn reinforcement learning strategies. These environments have a shared interface, allowing the training of generalizable algorithms. It makes no assumptions about the structure of the agent, and is compatible with any numerical computation libraries like Pytorch and Tensorflow.

The gym environment interfaces with the RL algorithm through the step function that implements the classic agent in the environment loop. At every timestep, the agent chooses an action from the policy, passes it to the *step* function and the environment returns the following information:

- observation (object): an environment-specific object representing the new state of the environment. For example, pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.
- reward (float): amount of reward achieved by the previous action.
- done (boolean): if its time to reset the environment again. Most tasks have fixed termination points, and done being True closes the episode.
- info (dict): diagnostic information useful for debugging

4.1.2 Cart Pole

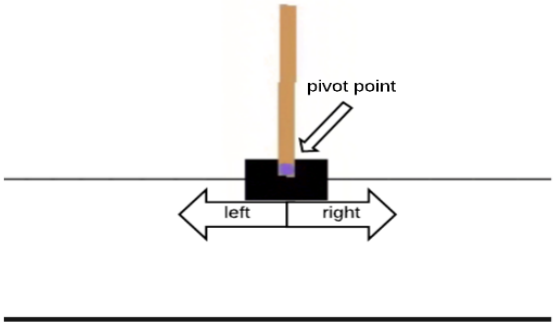

Figure 4.1: Screenshot of the Cartpole environment in OpenAI Gym

In the Cartpole experiment, a pole is attached by an unactuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The environment was first described in [39].

The state of the agent is defined through 4 continuous features and 2 discrete actions. Reward is +1 for every step survived and an episode terminates if (1) the pole angle is more than $\pm 12^\circ$ (2) episode length > 200 (3) Cart Position $> \pm 2.4$.

Table 4.1: Cartpole Observation Space

Feature	Min	Max
Cart Position	-2.4	2.4
Cart Velocity	-Inf	Inf
Pole Angle	-41.8°	41.8°
Pole Velocity(tip)	-Inf	Inf

Table 4.2: Cartpole Action Space

Code	Action
0	Push Cart to Left
1	Push Cart to Right

4.1.3 Mountain Car

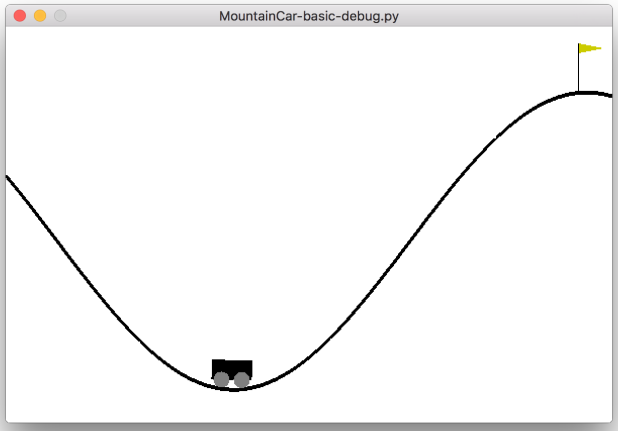

Figure 4.2: Screenshot of the Mountain Car environment in OpenAI Gym

A car is on a one-dimensional track, positioned between two 'mountains'. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. The state of the agent is defined through 2 continuous features and 3 discrete actions. Reward is -1 for every passing timestep until the goal is reached and the episode terminates on reaching the successful goal or if 200 iterations are reached. The starting state is a random position between -0.6 to -0.4 with 0 velocity.

This environment has a very tricky and complex solution policy. Training a DQN neural network as well as the distilled tree proves extremely challenging but doable.

Table 4.3: MC Observation Space

Feature	Min	Max
Position	-1.2	0.6
Velocity	-0.07	0.07

Table 4.4: MC Action Space

Code	Action
0	Accelerate to Left
1	No acceleration
2	Accelerate to Right

4.2 Training Setup

We provide an extensible platform for training neural network policies and then distilling the trained policies into tree structures followed by evaluation of the derived data structure. Mountain Car and Cartpole DQN were both trained using the same platform with generic Experience Replay and a Target Network networks. We use the same network configuration provided by [27].

For Cartpole, the network consisted of 3 fully connected hidden layers, each with 128 neurons, trained using the Adam optimizer with learning rate of 0.0001. The network was trained for 800 episodes with an Experience Buffer of size 10,000 and batch size of 20. Exploration decayed exponentially as epochs proceeded.

For Mountain Car, the network consisted of 2 fully connected hidden layers with sizes of 24 and 48 neurons, trained using the Adam optimizer with learning rate of 0.001. The network was trained for 400 episodes with an Experience Buffer of size 20,000 and batch size of 32. Exploration decayed exponentially as epochs proceeded.

For our distillation process, we provide the following hyperparameters:

- Iterations : Number of times the training loop needs to run. In each iteration, fresh data is obtained from observing the network perform in the environment.
- Transition Buffer Size (generally fixed at 5000)
- Trees per action : If the output of any action is to be regressed by more than 1 tree (generally set to 1 for QBSP Tree results below)
- Number of particles in the particle filter (set to 20)
- Number of stages for tree generation
- Games to evaluate performance on (set to 100 for all experiments)
- Total Budget (defined as $\mathbb{B}$ in the algorithm) : Decides the permitted cost to cut for a dividing plane over all stages

4.3 Results

The performance of QBSP-Tree and QBSP-Forests is compared to benchmark data from earlier papers [26] under the same evaluation environments implemented through the Gym [11] toolkit : Cartpole [4] and Mountain Car [29].

Like LMUT [26], we convincingly outperform all the offline methods including CART [10] and M5 model trees [31] for Q-value regression. The original BSP Forests [19] outperformed most online forests including Breimain-Random Forests [9], Bayesian Additive Regression Tree Forests [14], Extremely Randomized Forests [21] and batched versions of Mondrian Forests [25].

Fast Incremental Model Trees (FIMT) [23] is another online tree distillation method that builds a linear model that adapts to evolving data streams. LMUT and FIMT have been used for distillation of RL policies in the past. We formally present a quantitative comparison between our performance and theirs. However, note that since the authors of the works above did not provide any access to their code and implementation we are forced to use data directly from their papers. We expect to open-source our code on the publication of this thesis.

4.3.1 Regression Fidelity

Q-BSP Tree and Q-BSP Forests are evaluated based on how close its regression outputs are to the Q values from the Q-Networks. The standard regression metrics Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) are tabulated for each gym environment.

Mean Absolute Error can be computed as :

$$MAE = \frac{1}{a \times n} \sum_{\forall a \in \mathcal{A}} \sum_{j=1}^{n} |Q(s_j, a) - Q^*(s_j, a)| \tag{4.1}$$

Root Mean Square Absolute Error can be computed as :

$$MAE = \sqrt{\frac{1}{a \times n} \sum_{\forall a \in \mathcal{A}} \sum_{j=1}^{n} (Q(s_j, a) - Q^*(s_j, a))^2} \tag{4.2}$$

where Q^* represents the predicted Q-value and n represents the batch size.

Table 4.5: Cartpole Regression Fidelity

	Regression Loss		
Algorithm	MAE	RMSE	Leaves
FIMT	32.744	62.862	2195
LMUT	14.230	43.847	416
QBSP Tree	**2.222**	**3.700**	45

Table 4.6: Mountain Car Regression Fidelity

	Regression Loss		
Algorithm	MAE	RMSE	Leaves
FIMT	3.735	5.002	1021
LMUT	0.475	1.015	453
QBSP Tree	0.461	0.984	68

In the LMUT tests, the trees are trained over 30,000 consecutive transitions and evaluated over another 10,000 transitions. We follow the same baseline dataset. Table 4.5 presents the distillation performance results for the Cartpole environment. Table 4.6 depicts the comparative regression accuracy for the Mountain Car environment. QBSP Trees outperform older methods on all the environments consistently. In Cartpole, the difference between our approach and the previous state-of-the-art is particularly convincing.

4.3.2 Gameplay Performance

While regression might seem adequate to measure distillation performance, Q-values for different actions can often be close to each other and excellent regression performance might not translate to large rewards. Choosing sub optimal actions at certain crucial states can yield catastrophic results. Gameplay performance offers a more precise look into the robustness of distilled tree policies. For evaluation, the distilled tree policy controls the agent through 100 games or episodes. We use the Average Per Episode Reward (APER) metric to compare performance across various algorithms. Table 4.7 presents the comparative gameplay results over different distillation methods and it is clearly evident that QBSP Trees performs the best in both environments. However it beats the Oracle for Cartpole achieving perfect rewards for over 1000 continuous runs.

Table 4.7: Comparative Game Playing Performance

	Environment	
Model	Cart Pole	Mountain Car
DQN	193.88	-126.43
FIMT	40.54	-189.29
LMUT	147.91	-149.91
QBSP-Tree	**200.00**	**-141.65**

4.3.3 Feature Influence

Decision Trees are amenable to visual inspection and are hence often considered *interpretable* We p[33, 26]. To further develop our understanding of the distillation efficiency , we introduce a new metric and compare feature importance of the state inputs derived from our trained deep learning model and our distillation tree. Recently, Integrated Gradients (IG) [38] have been widely adopted for computing feature importance for neural networks. In brief, it computes the integral of the gradients obtained from a set of scaled input features and then takes the element-wise product of those features with the original input.

For decision trees, there have been two widely accepted feature importance metrics. Gini Importance (GI) [10] computes the importance of each feature as the weighted sum over the number of nodes (across all trees) that includes the feature multiplied by the number of samples that node splits. Mean Decrease in Accuracy(MDA) or Permutation Importance [20] is a more recent measure that estimates feature importance of an input dimension by randomly permuting that dimension for the input data and observing the decrease in accuracy.

We compute the relative importance of each feature by normalizing the feature importance values to sum up to one. The feature importance comparison between the deep learning policy and distilled tree policy for the Cartpole environment is represented in Table 4.8 and for the Mountain Car environment in Table 4.9. A rank of 1 indicates the most important feature and the rank 4 denotes the least important feature.

As is evident from the results, our distillation method transfers the feature importance from the deep learning policy to the tree to great extent. Most of the features share the same importance ranking across all the methods.

Table 4.8: Feature Influence for Cartpole

	Ranking(Relative Importance)		
Feature	GI	MDA	IG
Pole Angle	4 (0.106)	4 (0.123)	4 (0.06)
Cart Velocity	3 (0.209)	3 (0.244)	3 (0.17)
Cart Position	2 (0.231)	1 (0.339)	1 (0.49)
Pole Velocity	1 (0.455)	2 (0.295)	2 (0.28)

Table 4.9: Feature Influence for Mountain Car

	Ranking(Relative Importance)		
Feature	GI	MDA	IG
Velocity	2 (0.216)	2 (0.479)	2 (0.47)
Position	1 (0.784)	1 (0.521)	1 (0.53)

4.3.4 Verification

Bastani et.al [6] recently showed that tree policies can be used for verifying the correctness, stability and robustness of linear controllers. We repeat the same correctness experiments for our distilled Cartpole controller with the Z3 SMT Solver [16] under an identical linear dynamics approximations.

For proving correctness, a four dimensional state is defined (x, v, θ, ω) is defined. x denotes the cartpole position, v denotes the cart velocity, θ denotes the pole angle and finally ω signifies the angular velocity of the pole. The policy has to move the cart to the finish line while keeping the pole in a constant position. The action a provides linear force in each direction to the cart. The goal is to prove that the pole never falls below a certain height which is encoded as the formula (taken from [6]):

$$\psi \equiv s_0 \in S_0 \ \wedge \bigwedge_{t=0}^{\infty} |\phi(s_t)| \leq y_0 \tag{4.3}$$

Here , S_0 describes the set of initial states. $s_t = f(s_{t-1}, a_{t-1})$ is the state on step t, $\phi(s)$ is the deviation of the pole angle from upright in state s and y_0 is the maximum desirable deviation from the upright position. f is the transition function. We use a linear approximation for simplifying the dynamics :

$$f(s, a) \approx As + Ba \tag{4.4}$$

which is valid for small pole angles. We limit the time horizon to 8 steps in the rollout due to limited computational resources. If the controller is correct, $-\psi$ is unsatisfiable with the rules derived from our tree. Z3 was able to verify correctness in approximately 492 seconds.

Chapter 5

Conclusions

Through this thesis, we proposed a new approach to learning online decision tree policies. We explored a few different strategies for building online forests. This thesis was primarily focused on providing better distillation performance by using more powerful, expressive and condensed data structures. QBSP Trees outperformed other distillation methods and achieved performance closest to the Deep Q Network policy.

We also wanted to bring forward and highlight the self-consistency property of these Bayesian data structures. Self-consistent data structures are tailor-made for distillation methods and ours achieved impressive regressive and gameplay performance gains in Cart-pole while consuming only a third of runtime memory needed by VIPER and unlike VIPER, performed completely online.

QBSP Forests are able to scale better and handle more complex state spaces. In contrast to LMUT where trees can only expand linearly (only in the downward direction) in response to new data, every level of the QSP tree is evaluated for an incoming batch of data. As a result, the trees are of much smaller size and can easily be extended in a forest configuration. Linear tree structures are also more prone to forgetting earlier states in large state spaces; a problem mitigated our Ordered Sequential Monte Carlo training method.

Our proposed distillation method is not limited to DQN Policies and can be easily adapted to controllers trained using policy gradients (where the Q function is unavailable) by extracting the Q-values through the maximum entropy formulation of reinforcement learning [41].

We are the first research publication to pose Integrated Gradients as an "Influence Transfer Metric" for Distillation. In our tests QBSP Forests and the Deep Neural Network

ranked the features in the same order of importance signalling that we might have been able to capture intrinsic DNN policy knowledge into our distilled tree structure.

5.1 Future Work

Learning interpretable and explainable policies directly from the environment can help bring verifiable, safe and intelligent controllers into the real world. However, a lot of work remains to be done in the area. The primary challenge faced by researchers is the lack of defined gradients within tree structures. Tree Policies also possess a lower learning ceiling than neural networks due to their piece-wise regressive nature. Larger trees and forests are often uninterpretable and impair visual inspection of learnt policies. Generating shallow and expressive data structures remains an active area of research.

Distillation models can adopt techniques wherein specific areas in the state space are explored based on interchangable policy control between the oracle and the distilled policy. We term this as "Active Distillation" and hope that this method can drastically increase the efficiency of distillation for all data structures. The process saves times by not visiting swaths of feature space where the distilled structure is fairly confident of its abilities.

The Infinite Dirichlet Distribution provides an effective method to scale our QBSP approach to higher dimensions and state spaces. It allows for infinite clustering of datapoints around common centers; the processing of each of which is highly parallelizable. Instead of datapoints, the dimension-pairs can also be set as centers and mixing strategies can be used for forest growth.

Finally, advancing verification tools to accommodate complex environment and controller constraints can ease the need for simplifying approximations for verification and provide theoretical guarantees needed for real world applications.

References

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 1–, New York, NY, USA, 2004. ACM.

[2] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

[3] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2654–2662. Curran Associates, Inc., 2014.

[4] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[5] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 2621–2629, USA, 2016. Curran Associates Inc.

[6] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2494–2504. Curran Associates, Inc., 2018.

[7] Felix Berkenkamp, Matteo Turchetta, Angela P. Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 908–919, USA, 2017. Curran Associates Inc.

[8] Olcay Boz. Extracting decision trees from trained neural networks. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 456–461, New York, NY, USA, 2002. ACM.

[9] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[10] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[12] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, New York, NY, USA, 2006. ACM.

[13] Zhengping Che, Sanjay Purushotham, Robinder Khemani, and Yan Liu. Interpretable deep models for icu outcome prediction. In *AMIA Annual Symposium Proceedings*, volume 2016, page 371. American Medical Informatics Association, 2016.

[14] Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. Bart: Bayesian additive regression trees. *Ann. Appl. Stat.*, 4(1):266–298, 03 2010.

[15] Darren Dancey, Zuhair A Bandar, and David McLean. Logistic model tree extraction from artificial neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(4):794–802, 2007.

[16] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[17] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *J. Mach. Learn. Res.*, 6:503–556, December 2005.

[18] Xuhui Fan, Bin Li, and Scott Sisson. The binary space partitioning-tree process. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1859–1867, Playa Blanca, Lanzarote, Canary Islands, 09–11 Apr 2018. PMLR.

[19] Xuhui Fan, Bin Li, and Scott Anthony Sisson. Binary space partitioning forests. *arXiv preprint arXiv:1903.09348*, 2019.

[20] Aaron Fisher, Cynthia Rudin, and Francesca Dominici. All models are wrong but many are useful: Variable importance for black-box, proprietary, or misspecified prediction models, using model class reliance. *arXiv preprint arXiv:1801.01489*, 2018.

[21] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[22] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv e-prints*, page arXiv:1503.02531, Mar 2015.

[23] Elena Ikonomovska, João Gama, and Sašo Džeroski. Learning model trees from evolving data streams. *Data mining and knowledge discovery*, 23(1):128–168, 2011.

[24] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017.

[25] Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. Mondrian forests: Efficient online random forests. In *Advances in neural information processing systems*, pages 3140–3148, 2014.

[26] Guiliang Liu, Oliver Schulte, Wang Zhu, and Qingcan Li. Toward interpretable deep reinforcement learning with linear model u-trees. *CoRR*, abs/1807.05887, 2018.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[28] Christoph Molnar. Interpretable machine learning: A guide for making black box models explainable. https://christophm.github.io/interpretable-ml-book/feature-importance.html, 2016.

[29] Andrew William Moore. Efficient memory-based learning for robot control. 1990.

[30] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[31] J. R. Quinlan. Learning with continuous classes. pages 343–348. World Scientific, 1992.

[32] Christian Robert and George Casella. *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[33] Ivan Dario Jimenez Rodriguez, Taylor W. Killian, Sung-Hyun Son, and Matthew C. Gombolay. Interpretable reinforcement learning via differentiable decision trees. *CoRR*, abs/1903.09338, 2019.

[34] Daniel M Roy and Yee W. Teh. The mondrian process. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1377–1384. Curran Associates, Inc., 2009.

[35] Daniel Murphy Roy. *Computability, inference and modeling in probabilistic programming*. PhD thesis, Massachusetts Institute of Technology, 2011.

[36] D. Sculley. Combined regression and ranking. In *KDD*, 2010.

[37] Maarten Speekenbrink. A tutorial on particle filters. *Journal of Mathematical Psychology*, 73:140 – 152, 2016.

[38] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *CoRR*, abs/1703.01365, 2017.

[39] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[40] Gilles Vandewiele, Olivier Janssens, Femke Ongenae, Filip De Turck, and Sofie Van Hoecke. GENESIM: genetic extraction of a single, interpretable model. *arXiv e-prints*, page arXiv:1611.05722, Nov 2016.

[41] Brian D Ziebart, Andrew Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. 2008.