Improved Policy Extraction via Online Q-Value Distillation

Aman Jhunjhunwala
Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
a3jhunjh@uwaterloo.ca

Jaeyoung Lee
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
jaeyoung.lee@uwaterloo.ca

Sean Sedwards
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
sean.sedwards@uwaterloo.ca

Vahdat Abdelzad
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
vahdat.abdelzad@uwaterloo.ca

Krzysztof Czarnecki
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
k2czarne@uwaterloo.ca

***Abstract*—Deep neural networks are capable of solving complex control tasks in challenging environments, but their learned policies are hard to interpret. Not being able to explain or verify them limits their practical applicability. By contrast, decision trees lend themselves well to explanation and verification, but are not easy to train, especially in an online fashion. In this work we introduce Q-BSP trees and propose an Ordered Sequential Monte Carlo training algorithm that efficiently distills the Q-function from fully trained deep Q-networks into a tree structure. Q-BSP forests are used to generate the partitioning rules that transparently reconstruct an accurate value function. We explain our approach and provide results that convincingly beat earlier online policy distillation methods with respect to their own performance benchmarks.**

I. Introduction

Deep reinforcement learning (RL) has excelled at automatically learning human-level performance in a number of control and gaming tasks [1]. Indeed, deep learned policies now exceed human capabilities in complex video games like StarCraft and Dota [2]. Despite impressive performance, the reasoning behind such policies remains impenetrable, which can lead to undesirable and even dangerous behaviour [3]. For applicability in the real world, it is necessary for the policies to satisfy certain guarantees, such as stability [4], correctness [5] and robustness [6]. Confirming these characteristics directly for deep learned policies is typically intractable [7].

Distillation [8] is an effective method to train models that are often more compressed [9], more interpretable [10], more structured [11]–[13], or shallower [14]. Recently, researchers studying the safety of reinforcement learning have applied forms of imitation learning [15] to distill policies from trained deep neural networks to tree structure representations in an active-play setting [16], [17]. In this context, a fully-trained deep reinforcement learning algorithm [1] is observed as it performs actions in an environment (e.g., Cart-Pole [18]). All the transitions executed by the policy are passed through an online distillation algorithm that aims to condense the information into a desirable tree structure. Figure 1 presents a summary of the process.

We improve and extend the research of [16], [17] by proposing a novel, efficient, self-consistent tree distillation method that may be performed online and that achieves better accuracy than other online and offline methods.

In summary, our contributions are as follows:

- We present a novel data structure 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.
- We present a new combined regression and ranking algorithm based on the particle Gibbs sampler, which enables the online distillation of deep reinforcement learning policies into Q-BSP trees. We find that this method performs better and is relatively more scalable than previous distillation approaches applied to reinforcement learning policies.
- In addition to the best regression and gameplay performance, the policies distilled by the trees closely resemble the neural network in terms of feature importance. We introduce this as a new effectiveness metric for distillation. In common with the work in [16], we formally verify the correctness of our distilled policy trees for the Cart-Pole environment.

This work is supported by the Japanese Science and Technology agency (JST) ERATO project JPMJER1603: HASUO Metamathematics for Systems Design, by the Natural Sciences and Engineering Research Council of Canada (NSERC) CREATE in Product-Line Engineering for Cyber-physical Systems (PLoCS) grant and by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant: Model-Based Synthesis and Safety Assurance of Intelligent Controllers for Autonomous Vehicles.

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

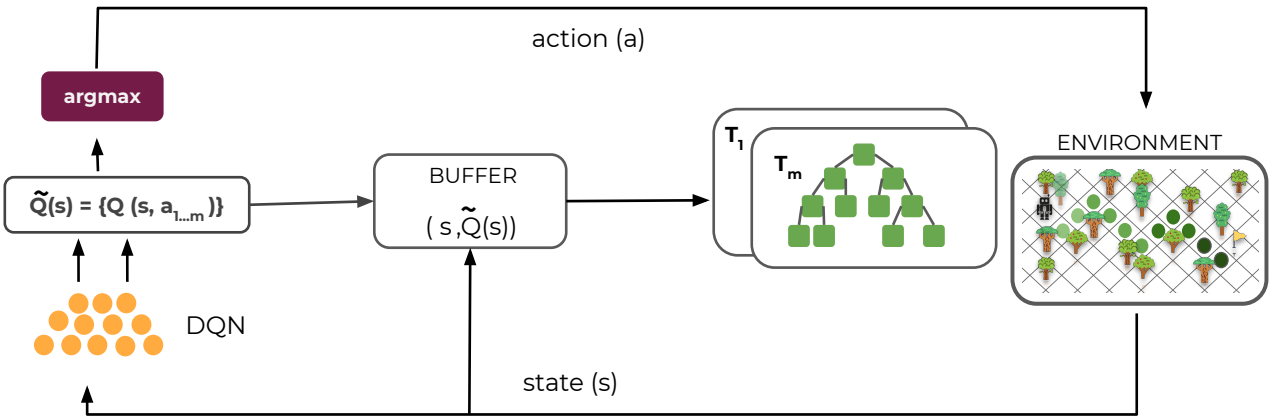

Fig. 1. Architecture of our Distillation Process. The Deep Q-Network outputs a Q-value for every action in the action space. The argmax of those Q-values (shown in red) is used to pick the best action to perform in the environment (denoted by a in figure). The action is performed in the simulation environment and the new states are collected in a buffer along with the Q-values that will result from feeding those states into the DQN. The data collected in the buffer is finally used to modify distilled decision trees, after which the buffer will be cleared.

II. Preliminaries

A. Reinforcement Learning and Q-function

The objective of reinforcement learning is to find a policy that guides the agent's actions in an environment so as to maximize the cumulative rewards the agent ultimately receives [19]. The environment here is formally described by a Markov Decision Process. A Markov Decision Process is a tuple $(S, \mathcal{A}, P, \mathcal{R})$ where S defines the state space, $\mathcal{A}$ is a finite set of actions, $P(s, a, s')$ is the probability that action a taken in state s will lead to state s' and $\mathcal{R}(s, a, s')$ is the immediate reward received after transitioning from state s to state s' due to action a.

Q-function is a function of a state-action pair and returns a real value $Q : S \times A \rightarrow \mathbb{R}$. $Q(s, a)$ represents the expected cumulative future reward assuming the agent in state s and performs action a and then continues playing until the end of the episode by following the optimal policy. While training, in any state s, either a random action (for exploration) or an action a with the highest Q-value is performed, i.e. $a = \text{argmax}_{a_i \in \mathcal{A}} Q(s, a_i)$ (for exploitation). The Q-function for continuous or large state spaces is intractable to tabulate explicitly, and hence is often represented by a neural network. This network is referred to as a Deep Q-Network (DQN) [1] and is the focus of distillation in this paper.

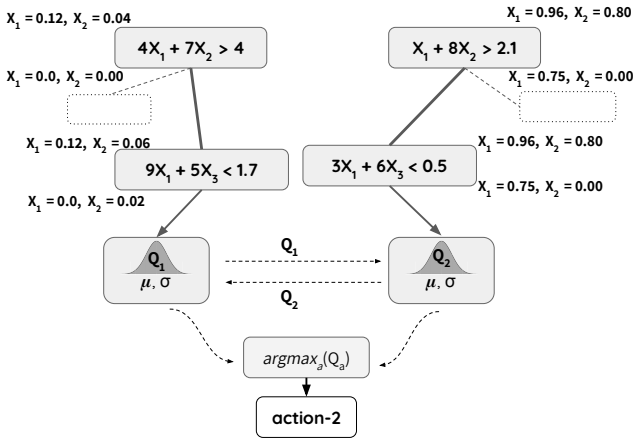

Fig. 2. Structural Representation of the Q-BSP Tree for a Cart-Pole environment with 4 state dimensions $(X_1, \cdots, X_4)$ and 2 actions. Note that in our tree structure, there are 2 trees, each representing the Q-value for a single action. Each node encodes mathematically richer boundary expressions and 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

B. Decision Trees : Evolution from CART to BSP Trees

Decision trees and random forests have a rich literature [20]–[22]. In CART [20], 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 related to information gain. These decision trees are used by [16] and require the entire dataset to be loaded in memory to compute this gain metric for every split. This prevents distillation in an online fashion.

Self-consistency, as described in [23], 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 [23], [24] and their recent successor Binary Space Partitioning (BSP) forests [25], [26] exploit this property to create the generative process for their respective trees. They are self-consistent online forest methods that have consistently matched the accuracy of the state-of-the-art offline regression methods trained on the same dataset. The BSP process [25] 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. This is why we use BSP processes as the basis for our Q-BSP forest structure.

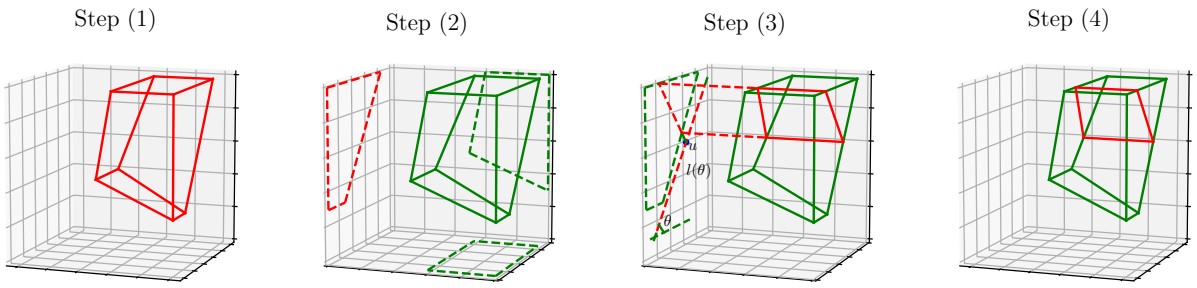

Fig. 3. Diagrammatic representation of the cutting process of a BSP tree leaf [25], [26]. Each node is represented by a polytope and the leaf partition is shown in red in Step (1). Step (2) shows all the two dimensional projections, red signifying the projection chosen for split in Step (3) .The generation of θ and u is shown as Step (3). Step (4) shows the parent node divided into two children nodes.

Algorithm 1: Sampling a leaf node to cut k^* and generating a cutting hyperplane $H\left(k^*, (d_1^*, d_2^*), \theta, u\right)$ to divide the selected leaf from [26]

Input: dataset $(x, y)^{1:N}$; leaf index $k \in \{1, \cdots, b\}$

Output: cutting hyperplane $H(k^*, (d_1^*, d_2^*), \theta, u)$; cost c

for $(d_1, d_2) \in \{(1,2), \cdots, (1,d), \cdots, (d-1,d)\}$ **do**
  for $k \in \{1, \cdots, b\}$ **do**
    Project data points $\{x\}^k \subset \{x\}^{1:N}$ reaching the leaf k onto the (d_1, d_2) dimensions to get a projection: $\Pi^k_{(d_1,d_2)} := \{(x_{d_1}, x_{d_2}) \in \mathbb{R}^2\}^k$
    Compute the convex hull $\mathbb{C}^k_{(d_1,d_2)}$ on $\Pi^k_{(d_1,d_2)}$
    Calculate the perimeter $\mathbb{P}^k_{(d_1,d_2)}$ of $\mathbb{C}^k_{(d_1,d_2)}$
Sample
  a leaf node k^* to cut, proportional to the sum of projection perimeters $\left\{\sum_{d_1,d_2} \mathbb{P}^k_{(d_1,d_2)}\right\}_{k=1}^{b}$
  a dimension pair (d_1^*, d_2^*) for the leaf node k^*, proportional to the perimeters $\left\{\mathbb{P}^{k^*}_{(d_1,d_2)}\right\}_{\forall (d_1,d_2)}$
  a direction $\theta \in (0, \pi]$, proportional to $\boldsymbol{l}(\theta)$, onto which $\mathbb{C}^{k^*}_{(d_1^*,d_2^*)}$ is projected (see Fig. 3)
  u uniformly on the projection of $\mathbb{C}^{k^*}_{(d_1^*,d_2^*)}$ (Fig. 3)
Calculate $H\left(k^*, (d_1^*, d_2^*), \theta, u\right)$ 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.

C. Binary Space Partitioning Tree

We summarize the tree generation process for BSP trees in Algorithm 1 [25], [26]. The levels of the tree are recursively generated through a series of cutting hyperplanes bounded by the node polytope. A BSP tree T is implemented through a set of partitions in the data space. Each partition is typically represented by a convex polytope $\square \subset \mathbb{R}^d$ and each cutting hyperplane is parallel to the $d-2$ dimensions it does not cut through. For any arbitrary pair of dimensions (d_1, d_2) in the d-dimensional input $x = (x_1, ..., x_d) \in \mathbb{R}^d$, the selection and splitting process for a leaf node k^* in T, given a dataset $(x, y)^{1:N}$ of size N, is summarized as follows.

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 (d_1, d_2) represents pair of dimensions 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 decided. An angle is chosen at random from $(0, \pi]$, with a probability density 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 that line segment $\boldsymbol{l}(\theta)$. The cutting hyperplane for the node k along dimensions (d_1, d_2), denoted $H\left(k, (d_1, d_2), \theta, u\right)$, 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 an exponential distribution with the sum of the projection perimeters for all leaf nodes as its rate. If the cost exceeds a predefined budget (discussed in Sections III-A and IV), the newly formed hyperplane is discarded.

III. The Q-BSP Forest

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 in the action space $\mathcal{A} = \{a_1, a_2, \cdots, a_m\}$. As indicated in Figure 1, a Q-value is generated for every possible action $a_i \in \mathcal{A}$ in a given state s. This set of Q-values is denoted $\widetilde{Q}(s) = \{Q(s, a_i) \mid a_i \in \mathcal{A}\}$. The action to be performed is selected as

$$a = \underset{a_i \in \mathcal{A}}{\text{argmax}}\, Q(s, a_i). \tag{1}$$

With a slight abuse of notations, we also refer to $\widetilde{Q}(s)$ as a vector in $\mathbb{R}^m$ whose ith element is $Q(s, a_i)$. 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 learning task is then defined as a regression problem to fit our tree-based model onto the input-output map $s \mapsto \widetilde{Q}(s)$.

An independent collection of trees $\{T_1, .., T_m\}$ is created to form a forest. Each tree T_i is responsible for regressing the Q-value $Q(\cdot, a_i)$ for 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 2. Using notation from [22], 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 the leaf node z. The function $g(\cdot)$ encapsulates this tree traversal and expresses the Q-value for the ith action as

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

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, assuming homoschedastic data. Once we predict Q-values for all the m actions, we use Equation (1) to predict the next action. Finally, we can trivially extend the forest such that each $Q(s, a_i)$ value can be predicted by more than one tree.

A. Online Expansion of Node Partitions

In Section II-C it is shown that every node in the BSP tree (including the *root*) is bounded by the convex hull of the points it has just observed. The original BSP trees were not designed for online operation and hence did not include any way of expanding the partition boundaries in case the incoming data lay outside root node boundaries. Reinforcement learning environments often have intractable state space, resulting from a large number of state dimensions or dimensions with large ranges. 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. We note that if a new datapoint is observed, only one of two things can occur for any given tree: (i) an existing split boundaries are expanded to include the new datapoint, or (ii) a new split is generated above the generated split (the split is reset). These cases are illustrated in Figure 4. Algorithm 2 is summarized below.

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 afresh, following the BSP tree leaf generation algorithm described in Section II-C.

Algorithm 2: 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 $(d_1, d_2) \in \{(1,2), \cdots, (1,d), \cdots, (d-1,d)\}$ **do**

  Compute

    the projections $(x_{d_1}, x_{d_2})^k \subset \{s'\}^k \in \mathbb{R}^2$ onto dimensions (d_1, d_2);

    the new convex hull $\mathbb{C}^*_{(d_1,d_2)}$ on old boundary points and new projection points

$$\left\{\mathbb{C}_{(d_1,d_2)} \bigcup (x_{d_1}, x_{d_2})^k\right\}$$

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

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

Expand the cutting hyperplane $H\,(k, (d_1^*, d_2^*), \theta, u)$ to mirror the line $\boldsymbol{l}(\theta)$

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 : (θ, u) on the newly computed convex hulls $\mathbb{C}^*$

IV. The Distillation Training Process

A fixed interval based approach $\{(\mathbb{B}_i, \mathbb{B}_{i+1})\}_{i=0}^{S}$ is implemented for Q-BSP training. At the ith stage ($i = 0, \cdots, S$), all the trees $T_1, \ldots, T_m$ are under the same budget constraints $(0, \mathbb{B}_i]$ and there might be zero, one or more cuts at every stage, with the sum of the costs within the pre-allocated budget $\mathbb{B}_i$ (refer Section II-C). When a tree T is initialized, a portion of the tree is generated in every stage. The tree generated up to stage i, denoted by T^i, is assigned a numeric value $\omega_i \in \omega_{1:S}$ that describes its *goodness of fit* (regression fit) to the incoming data so far.

We give a high-level summary of our training method presented in Algorithm 3. Given the original tree, we proceed in a sequential top-down manner. At every given tree level, R mutations of the tree are created. The winner of a particular level is used to create the R clones at the next level (line 4). At every level, mutations can include splitting a leaf into two nodes (line 7), destroying a node along with its children at that level and recomputing the boundary line, or no change at all (line 9). All the mutations and the original tree compete to get the highest score on the training dataset (line 16). These

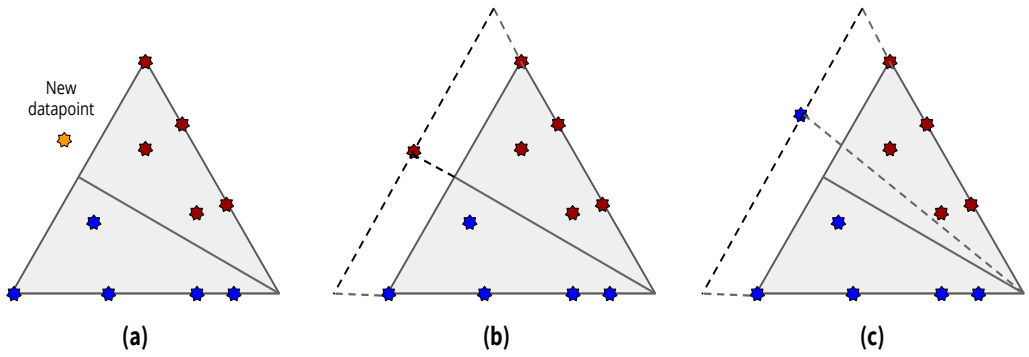

Fig. 4. Online expansion of a BSP block. Diagram(a) plots the initial boundary of a BSP block when a new out-of-boundary datapoint appears. Now there are only two ways in which expansion can work. In (b), we simply extrapolate the original boundary line to accommodate the new point. In (c), we try different new boundary partitions and choose the one which minimizes the given loss metric

Algorithm 3: The Ordered Sequential Monte Carlo training process for the j-th Q-BSP-tree T_j regressing over action a_j

Input: batch data $\mathcal{D} = (s, Q(s, a_j))^{1:N}$; the number of particles R; Q-values predicted by all other trees for given dataset $\mathcal{D}$: $\hat{Q}_{\mathcal{D}} = \left\{Q^*(s, a_z)\right\}^{1:N}_{z\in\{1,\cdots,j-1,j+1,\cdots,m\}}$

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

1 **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 $\boldsymbol{\mu}_i[1:R] = 0$.

2 Expand parent tree T_j using **Algorithm 2** for $(s, Q(s, a_j))^{1:N}$, if needed

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

4 $\quad V[1:R] = \mathbb{T}_{i-1}$; `// make R clones of last successful tree`

5 $\quad V[R+1] = T_j^i$; `// the state of original tree at stage i`

6 $\quad$ **for** $r = 1, \cdots, R$, **if** $\mathbb{B}_i < \sum_{i'=0}^{i-1} c_{i'}[r] < \mathbb{B}_{i+1}$, **do**

7 $\quad\quad$ Using Algorithm 1, recursively obtain

$$H_i[r] = \{\hat{H}_\lambda\}_{\lambda=1}^{\Lambda} \in V[r], \quad \boldsymbol{\mu}_i[r] \in V[r], \quad c_i[r] = \sum_{\lambda=1}^{\Lambda}\{c_\lambda\},$$

$\quad\quad$ where Λ is total number of cuts falling in $(\mathbb{B}_i, \mathbb{B}_{i+1}]$, **until** $\sum_{i'=0}^{i} c_{i'}[r] > \mathbb{B}_{i+1}$

8 $\quad\quad$ **if** $\Lambda = 0$ **then**

9 $\quad\quad\quad$ Set $H_i[r] = H_{i-1}[r]$, $\boldsymbol{\mu}_i[r] = \boldsymbol{\mu}_{i-1}[r]$, and $c_i[r] = 0$

10 $\quad$ **for** $r = 1, \cdots, R+1$ **do**

11 $\quad\quad$ Compute likelihood weight, $\omega[r] := \dfrac{prior(\boldsymbol{\mu}_i[r]) \cdot p(Q(s,a_j)|s, H_{1:i}[r], \boldsymbol{\mu}_{1:i}, \sigma^2)}{posterior(\boldsymbol{\mu}_i[r]) \cdot p(Q(s,a_j)|s, H_{1:i-1}[r], \boldsymbol{\mu}_{1:i-1}, \sigma^2)}$

12 $\quad\quad$ Compute ranking weight, $\varphi[r] = \boldsymbol{\rho}(V[r], \hat{Q}_{\mathcal{D}}, \mathcal{D})$, where $\boldsymbol{\rho}$ is the ranking loss function

13 $\quad$ **for** $r = 1, \cdots, R+1$ **do**

14 $\quad\quad$ **Normalize** weights $W[r] := \dfrac{(\omega[r] + \alpha.\varphi[r])}{\sum_{r'=1}^{R}(\omega[r'] + \alpha.\varphi[r'])}$, where α is a mixing parameter

15 $\quad$ **Sample** one particle $\mathbf{r} \propto W[r]$ as winner, $\mathbb{T}_i = V[\mathbf{r}]$

16 $T_j^* = \mathbb{T}_S$

scores are used to pick winners at every stage. The score is the average of two metrics: (i) improvement in regression accuracy on the training dataset over the base tree (line 11), and (ii) the relative sorted ordering of the predicted value over all other action-trees (or forests) (line 13).

To the best of our knowledge, the ordering metric has never been used for RL distillation before, so we summarize its motivation here. In our test environments, a score based on regression accuracy alone produced minimal regressive losses, but these failed to translate into high rewards. For some crucial

TABLE I
CART-POLE REGRESSION FIDELITY

Algorithm	Regression Loss MAE	RMSE	Leaves
FIMT[27]	32.744	62.862	2195
LMUT[17]	14.230	43.847	416
Q-BSP-Tree	**2.222**	**3.700**	45

TABLE II
MOUNTAIN-CAR REGRESSION FIDELITY

Algorithm	Regression Loss MAE	RMSE	Leaves
FIMT[27]	3.735	5.002	1021
LMUT[17]	0.475	1.015	453
Q-BSP-Tree	0.461	0.984	68

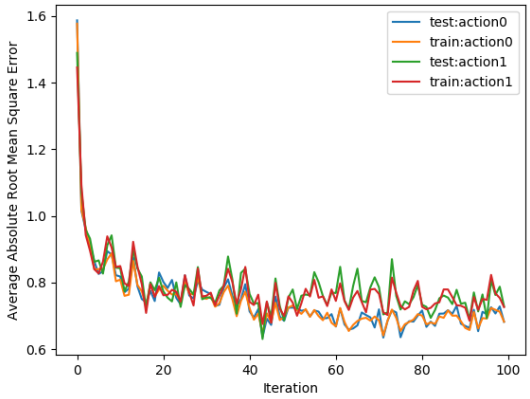

Fig. 5. RMSE loss curve averaged over 5 runs of the Cart-Pole task, each lasting 100 iterations. A 80:20 training and testing ratio was chosen for every set of collected trajectories.

states, the difference in Q-values for competing actions is often very small, such that regression error metrics prove insufficient to force the correct action to be picked. Picking the wrong action can often lead to catastrophic failures in the game. The ordering score builds the relative ordering of Q-values as a condition into the posterior sampling problem.

Each tree T_j produces the Q-value $Q^*(s, a_j)$ for action a_j. Collectively, all the trees reconstruct the original $\widetilde{Q}(s)$ function, denoted $\widetilde{Q}^*(s)$. The ranking function $\boldsymbol{\rho}$ used in Algorithm 3 computes the mean square error between the neural network ranking and the distillation tree ranking of the Q values in the dataset.

V. RESULTS

The performance of our algorithms Q-BSP-Tree and Q-BSP-Forests is compared to benchmark data from earlier works [17], [27], under the same evaluation environments, implemented through the Gym toolkit [28]. To make this comparison, we consider environments Cart-Pole [18] and Mountain-Car [29], but also include some results for the Lunar Lander environment.

Like LMUT [17], we convincingly outperform all the offline methods that perform Q-value regression, including CART [20] and M5 model trees [30]. The original BSP forests [26] outperformed most online forests, including Breimain-Random forests [21], Bayesian Additive Regression Tree forests [22], Extremely Randomized forests [31] and batched versions of Mondrian forests [24].

LMUT[17] and FIMT[27] have been used for distillation of RL policies in the past. We present a quantitative comparison between our performance and theirs. We refrain from providing detailed performance comparisons with Viper [16], as it lacks any regression component.

Note that since the authors of the above-mentioned algorithms do not provide any access to their implementations, we are forced to use data directly from their papers.

A. Regression Fidelity

Q-BSP-Tree and Q-BSP-Forests are evaluated on how close their 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. In the LMUT tests, the trees are trained over $30,000$ consecutive transitions and evaluated over another $10,000$ transitions.

Mean Absolute Error is computed as

$$\text{MAE} = \frac{1}{m \times n} \sum_{\forall a \in \mathcal{A}} \sum_{j=1}^{n} |Q(s_j, a) - Q^*(s_j, a)|. \quad (3)$$

Root Mean Square Error is computed as

$$\text{RMSE} = \sqrt{\frac{1}{m \times n} \sum_{\forall a \in \mathcal{A}} \sum_{j=1}^{n} (Q(s_j, a) - Q^*(s_j, a))^2}, \quad (4)$$

where Q^* represents the predicted Q-value, m is the size of set $\mathcal{A}$ and n represents the batch size.

Table I presents the distillation performance results for the Cart-Pole environment. Table II gives the comparative regression accuracy for the Mountain-Car environment. Q-BSP-Trees consistently outperforms older methods on all the environments. In Cart-Pole, the difference between our approach and the previous state-of-the-art is particularly convincing. Figure 5 presents the regression curve for the 2-action Cart-Pole environment.

B. 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 informative view of 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 III presents the

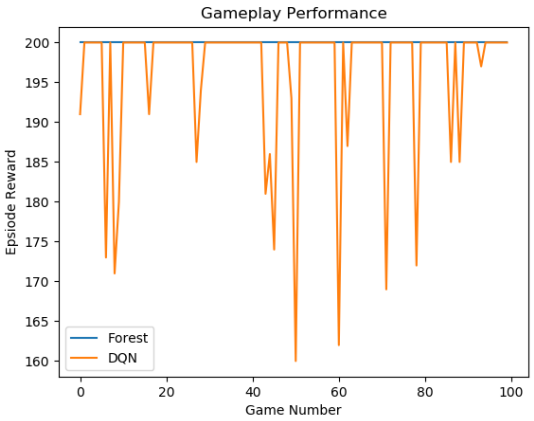

Fig. 6. Reward obtained over 100 games of Cart-Pole by our oracle (DQN) and distilled BSP Forest (one tree per action)

TABLE III
COMPARATIVE GAME PLAYING PERFORMANCE

	Environment		
Model	Cart Pole	Mountain-Car	Lunar Lander
DQN	194.85	-130.32	200.0
FIMT[27]	40.54	-189.29	-
LMUT[17]	147.91	-149.91	-
Q-BSP-Tree	**200.00**	**-141.65**	**200.00**

comparative gameplay results over different distillation methods and it is clear that Q-BSP-Tree performs the best; it matches the oracle in Lunar Lander and beats the oracle while achieving a perfect score in Cart-Pole. The gameplay performance for Cart-Pole using the oracle and our distilled tree is plotted in Figure 6.

C. Feature Influence

Decision Trees are amenable to visual inspection and are hence often considered *interpretable* [17], [32]. 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) [33] 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) [20] 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 [34] 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.

TABLE IV
FEATURE INFLUENCE FOR CART-POLE

	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 V
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)

We compute the relative importance of each feature by normalizing the feature importance values to sum to 1.0. The feature importance comparison between the deep learning policy and distilled tree policy for the Cart-Pole environment is presented in Table IV, and for the Mountain-Car environment in Table V. A rank of 1 indicates the most important feature, while a rank of 4 denotes the least important feature.

The results demonstrate that our distillation method transfers almost the entire feature importance from the deep learned policy to the tree.

D. Verification

Bastani et al. [16] recently showed that tree policies can be used for verifying the correctness, stability and robustness of linear controllers. In Cart-Pole, the policy tries to move the cart to the finish line while keeping the pole in a vertical position. The action a provides linear force in each direction to the cart. We repeat the same correctness experiments for our distilled Cart-Pole controller, using the Z3 SMT Solver [35] under the linear dynamics approximation $f(s,a) = As + Ba$, the small angle assumption, and a finite time horizon of $T = 8$ steps. Our controller was deemed correct, since the pole angle stayed between -10 and $+10$ degrees during the entire rollout.

VI. CONCLUSIONS

In this work we proposed a new approach to learning online decision tree policies. Q-BSP Trees outperformed other distillation methods and achieved performance closest to the oracle DQN policy. We ensure meaningful distillation by measuring feature influence transfer, while our generated trees are amenable to verification tools and visual inspection. Q-BSP distillation consumes only a third of the runtime memory needed by Viper [16] and, unlike that approach, performs distillation completely online. Relative to previous algorithms, our algorithm scales better and handles a larger range of state spaces. In contrast to greedy approaches like LMUT, where trees can only expand linearly (in the downward direction) in response to new data, every level of the Q-BSP tree is

evaluated with respect to an incoming batch of data. As a result, the trees are much smaller and are easily extensible in a forest configuration, with more than one tree regressing over a particular action.

As future work, for image based reinforcement learning tasks, feature vectors from pretrained feature extraction networks can serve as an effective state space for distillation. With enough resources, it would be interesting to scale our concept to these use-cases along with relatively more complex to deal with state spaces through Infinite Dirichlet Clustering on multiple compute nodes split by data points, state dimensions or trees (or all three).

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

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

[3] D. Amodei, C. Olah, J. Steinhardt, P. F. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," *CoRR*, vol. abs/1606.06565, 2016. [Online]. Available: http://arxiv.org/abs/1606.06565

[4] F. Berkenkamp, M. Turchetta, A. P. Schoellig, and A. Krause, "Safe model-based reinforcement learning with stability guarantees," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. USA: Curran Associates Inc., 2017, pp. 908–919. [Online]. Available: http://dl.acm.org/citation.cfm?id=3294771.3294858

[5] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," *CoRR*, vol. abs/1702.01135, 2017. [Online]. Available: http://arxiv.org/abs/1702.01135

[6] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. USA: Curran Associates Inc., 2016, pp. 2621–2629. [Online]. Available: http://dl.acm.org/citation.cfm?id=3157382.3157391

[7] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *J. Mach. Learn. Res.*, vol. 6, pp. 503–556, Dec. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1046920.1088690

[8] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv e-prints*, p. arXiv:1503.02531, Mar 2015.

[9] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 535–541. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150464

[10] Z. Che, S. Purushotham, R. Khemani, and Y. Liu, "Interpretable deep models for icu outcome prediction," in *AMIA Annual Symposium Proceedings*, vol. 2016. American Medical Informatics Association, 2016, p. 371.

[11] G. Vandewiele, O. Janssens, F. Ongenae, F. De Turck, and S. Van Hoecke, "GENESIM: genetic extraction of a single, interpretable model," *arXiv e-prints*, p. arXiv:1611.05722, Nov 2016.

[12] O. Boz, "Extracting decision trees from trained neural networks," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 456–461. [Online]. Available: http://doi.acm.org/10.1145/775047.775113

[13] D. Dancey, Z. A. Bandar, and D. McLean, "Logistic model tree extraction from artificial neural networks," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 37, no. 4, pp. 794–802, 2007.

[14] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2654–2662. [Online]. Available: http://papers.nips.cc/paper/5484-do-deep-nets-really-need-to-be-deep.pdf

[15] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the Twenty-first International Conference on Machine Learning*, ser. ICML '04. New York, NY, USA: ACM, 2004, pp. 1–. [Online]. Available: http://doi.acm.org/10.1145/1015330.1015430

[16] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 2494–2504. [Online]. Available: http://papers.nips.cc/paper/7516-verifiable-reinforcement-learning-via-policy-extraction.pdf

[17] G. Liu, O. Schulte, W. Zhu, and Q. Li, "Toward interpretable deep reinforcement learning with linear model u-trees," *CoRR*, vol. abs/1807.05887, 2018. [Online]. Available: http://arxiv.org/abs/1807.05887

[18] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834–846, 1983.

[19] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

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

[21] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[22] H. A. Chipman, E. I. George, and R. E. McCulloch, "Bart: Bayesian additive regression trees," *Ann. Appl. Stat.*, vol. 4, no. 1, pp. 266–298, 03 2010. [Online]. Available: https://doi.org/10.1214/09-AOAS285

[23] D. M. Roy and Y. W. Teh, "The mondrian process," in *Advances in Neural Information Processing Systems 21*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds. Curran Associates, Inc., 2009, pp. 1377–1384. [Online]. Available: http://papers.nips.cc/paper/3622-the-mondrian-process.pdf

[24] B. Lakshminarayanan, D. M. Roy, and Y. W. Teh, "Mondrian forests: Efficient online random forests," in *Advances in neural information processing systems*, 2014, pp. 3140–3148.

[25] X. Fan, B. Li, and S. Sisson, "The binary space partitioning-tree process," in *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, A. Storkey and F. Perez-Cruz, Eds., vol. 84. Playa Blanca, Lanzarote, Canary Islands: PMLR, 09–11 Apr 2018, pp. 1859–1867. [Online]. Available: http://proceedings.mlr.press/v84/fan18b.html

[26] X. Fan, B. Li, and S. A. Sisson, "Binary space partitioning forests," *arXiv preprint arXiv:1903.09348*, 2019.

[27] E. Ikonomovska, J. Gama, and S. Džeroski, "Learning model trees from evolving data streams," *Data mining and knowledge discovery*, vol. 23, no. 1, pp. 128–168, 2011.

[28] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[29] A. W. Moore, "Efficient memory-based learning for robot control," 1990.

[30] J. R. Quinlan, "Learning with continuous classes." World Scientific, 1992, pp. 343–348.

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

[32] I. D. J. Rodriguez, T. W. Killian, S. Son, and M. C. Gombolay, "Interpretable reinforcement learning via differentiable decision trees," *CoRR*, vol. abs/1903.09338, 2019. [Online]. Available: http://arxiv.org/abs/1903.09338

[33] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," *CoRR*, vol. abs/1703.01365, 2017. [Online]. Available: http://arxiv.org/abs/1703.01365

[34] A. Fisher, C. Rudin, and F. 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.

[35] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.