Q-Learning with Online Trees

by

Joosung Min

B.Econ., Sungkyunkwan University, 2014

Project Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Statistics and Actuarial Science
Faculty of Science

© Joosung Min 2021
SIMON FRASER UNIVERSITY
Summer 2021

Copyright in this work rests with the author. Please ensure that any reproduction
or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name:	**Joosung Min**
Degree:	**Master of Science (Statistics)**
Title:	**Q-Learning with Online Trees**
Committee:	**Chair:** Himchan Jeong Assistant Professor, Statistics and Actuarial Science

Lloyd T. Elliott
Supervisor
Assistant Professor, Statistics and Actuarial Science

Jiguo Cao
Committee Member
Professor, Statistics and Actuarial Science

Thomas Loughin
Examiner
Professor, Statistics and Actuarial Science

Abstract

Reinforcement learning is one of the major areas of artificial intelligence that has been studied rigorously in recent years. Among numerous methodologies, Q-learning is one of the most fundamental model-free reinforcement learning algorithms, and it has inspired many researchers. Several studies have shown great results by approximating the action-value function, one of the essential elements in Q-learning, using non-linear supervised learning models such as deep neural networks. This combination has led to the surpassing human-level performances in complex problems such as the Atari games and Go, which have been difficult to solve with standard tabular Q-learning. However, both Q-learning and the deep neural network typically used as the function approximator require very large computational resources to train. We propose using the online random forest method as the function approximator for the action-value function to mitigate this. We grow one online random forest for each possible action in a Markov decision process (MDP) environment. Each forest approximates the corresponding action-value function for that action, and the agent chooses the action in the succeeding state according to the resulting approximated action-value functions. When the agent executes an action, an observation consisting of the state, action, reward, and the subsequent state is stored in an experience replay. Then, the observations are randomly sampled to participate in the growth of the online random forests. The terminal nodes of the trees in the random forests corresponding to each sample randomly generate tests for the decision tree splits. Among them, the test that gives the lowest residual sum of squares after splitting is selected. The trees of the online random forests grown in this way age each time they take in a sample observation. One of the trees that is older than a certain age is then selected at random and replaced by a new tree according to its out-of-bag error. In our study, forest size plays an important role. Our algorithm constitutes an adaptation of previously developed Online Random Forests to reinforcement learning. To reduce computational costs, we first grow a small-sized forest and then expand them after a certain period of episodes. We observed in our experiments that this forest size expansion showed better performances in later episodes. Furthermore, we found that our method outperformed some deep neural networks in simple MDP environments. We hope that this study will be a medium to promote research on the combination of reinforcement learning and tree-based methods.

Keywords: Reinforcement learning, Q-learning, online random forest;

Acknowledgements

First of all, I would like to express my deepest appreciation to my supervisor Dr. Lloyd Elliott for his endless support and commitment. His guidance and encouragement were truly irreplaceable in completing this project. Thank you so much for giving me the opportunity and confidence to challenge new ideas and accomplish my goals. I am looking forward to our future projects.

I wish to thank my fellow grad students Louis Arsenault-Mahjoubi, Zubia Mansoor, Peter Tea, Seyeon Kim, Claudine Tu, Renny Doig, Robyn Ritchie, and Ahmad Mokhtar who brightened up my grad student life at SFU with great friendship and joy. I also have to thank my friends back in South Korea, Gihyun Jeong and Youngsu Cheon, for their endless and invaluable friendship.

Finally, I would like to acknowledge the unparalleled love and support of my parents and brother. You are the pillars of my life, and I am forever indebted to you for always being there for me.

Table of Contents

Declaration of Committee ii

Abstract iii

Acknowledgements v

Table of Contents vi

List of Tables viii

List of Figures ix

1 Introduction 1

2 Literature Review 2

2.1 Reinforcement Learning: Overview 2

2.1.1 Temporal Difference Learning 4

2.1.2 SARSA: On-policy TD Control 5

2.1.3 Q-learning: Off-policy TD Control 6

2.2 Value-function approximation 7

2.3 Tree-based Batch Mode Reinforcement Learning 8

2.3.1 Fitted Q-Iteration 8

2.3.2 Adapting to different action spaces 10

2.4 Deep Q-learning 10

2.4.1 Q-network: Non-linear Q-function approximator 10

2.4.2 Experience replay 11

2.5 Online random forests 12

2.5.1 Extremely randomized trees 13

2.5.2 Online bagging 14

2.5.3 Online decision trees 15

2.5.4 Temporal knowledge weighting 16

2.5.5 Summary 17

3 Methods 18
3.1 Online random forest in regression setting . . . 18
3.2 Computing $\max_{a \in \mathcal{A}} \hat{Q}(S, a)$ when $|A| > 1$ . . . 21
3.3 Partial randomness in split point selection . . . 22
3.4 Expanding ensemble size . . . 22

4 Experiments and results 24
4.1 Blackjack . . . 25
4.1.1 Blackjack: Results . . . 25
4.1.2 Blackjack: Statistical tests . . . 28
4.2 Inverted pendulum . . . 30
4.2.1 Inverted pendulum: Results . . . 30
4.2.2 Inverted pendulum: Statistical tests . . . 32
4.3 Lunar lander . . . 35

5 Discussion 37

6 Conclusion 38

Bibliography 40

Appendix A List of Parameters 43

List of Tables

Table 4.1 Blackjack: Results of Shapiro-Wilk test for the normality of the average rewards data at episodes 300 and 1,000 obtained by DQN and RL-ORF. In all cases, we cannot reject the null hypothesis that the data is normally distributed. 29
Table 4.2 Blackjack: Results of one-sided T-test between the mean average cumulative rewards from DQN and RL-ORF. There is a statistical evidence that the mean from RL-ORF is greater than the mean from DQN at episode 1,000 with p-value of 2.272E-5. 29
Table 4.3 Inverted Pendulum: Results from Shapiro-Wilk tests for normality of the average rewards generated by RL-ORF with and without ensemble size expansion. With the p-values smaller than the significance level, we provide statistical evidence that the data is not normally distributed in all cases. 33
Table 4.4 Inverted Pendulum: Results on Mann-Whitney U test for comparison between the means of RL-ORF with ensemble size expansion and without expansion. We provide evidence that RL-ORF with ensemble size expansion outperforms the one without the expansion at episode 1,000. 33
Table 4.5 Inverted Pendulum: Results from the Shapiro-Wilk normality test on the average rewards per episode at episodes 300 and 1,000 generated by the best performing RL-ORF and DQN. The results reports that we have enough evidence to reject the null hypothesis that the data is normally distributed in all four cases. 35
Table 4.6 Inverted pendulum: Results from the Mann-Whitney U test comparing the means of DQN and RL-ORF at episodes 300 and 1,000. Small p-values indicate that RL-ORF outperforms DQN in both cases. 35

List of Figures

Figure 3.1 Schematics of Q-learning with online trees. Transitions from interactions with the environments are stored in replay memory. Random mini-batches of the transitions sequentially update the online trees, which approximate action-value functions for the succeeding state-action pairs. The behaviour policy chooses the action with the largest Q-function with the probability of $1 - \varepsilon$, or a random action with ε. See Algorithm 11 for detail. 19

Figure 4.1 Blackjack: Average rewards obtained using DQN as the Q-function approximator with different hidden node sizes and learning rates. (d) presents that DQN with hidden node size = 32x32 and $\alpha = 0.01$ performed the best at episode 1,000 among the combinations. . . . 26

Figure 4.2 Blackjack: Average rewards obtained from ORF with different η and whether ensemble size is expanded at episode 100. (d) shows that RL-ORF with η=32 and the ensemble expansion gives the best performance among the examined parameter sets. 27

Figure 4.3 Blackjack: Average cumulative reward per 100 episodes from the best ones of (a) DQN and (b) RL-ORF with the error regions. Neither of the two methods showed noticeably larger or smaller performance fluctuations compared to the other. 28

Figure 4.4 Blackjack: Comparison of the average cumulative rewards per 100 episodes between the best DQN, RL-ORF, and standard Q-learning. RL-ORF has a larger average cumulative reward than DQN at episode 1,000. Statistical evidence is provided in Table 4.2. 28

Figure 4.5 Inverted Pendulum: Average rewards per episode obtained by DQN with different hidden node sizes and learning rates. (d) shows that the best DQN has hidden node size 128x128 and $\alpha = 0.005$. 31

Figure 4.6 Inverted Pendulum: Average rewards per episode obtained by RL-ORF with varying sizes of η and whether ensemble size is expanded. In all (a), (b), and (c), expanding the ensemble size produces the best performance at episode 1,000. The best RL-ORF has $\eta = 256$ and the ensemble size expanded from 100 to 200 at episode 100. . . 32

Figure 4.7 Inverted Pendulum: Average rewards per episode from the best of (a) DQN and (b) RL-ORF with error regions. The figures show that neither of the two has recognizably smaller performance fluctuations compared to the other. 34
Figure 4.8 Inverted Pendulum: Comparison between the RL-ORF and DQN in average rewards per episode. The statistical test shown in Table 4.5 shows that our method outperforms the DQN. 34
Figure 4.9 Lunar lander: Average rewards per episode obtained by (a) DQN and (b) RL-ORF. Neither one of the function approximators performs well within 1,000 episodes. 36

Chapter 1

Introduction

In reinforcement learning (RL), agents learn to make good decisions through interaction with their environment. Such methods are used in object tracking, games, and recommendation systems and often involve online learning with big data in which observations arrive with volume and variety. Online random forests provide lightweight implementations suitable for such data [26]. In Q-learning for RL [29], the action-value function may be approximated by an arbitrary function. Variational methods [14], linear function approximation methods including polynomial [2], Fourier basis [17], radial basis function approximation [24], and non-linear methods such as tree-based methods [9] and neural networks [22] have long been a standard for functional approximation in Q-learning [10]. In this work, we explore online random forests (ORFs; [26]) for approximation of the Q-function. In order to operationalize ORFs for approximation of Q-functions, we solve two theoretical issues: 1) We bring methods from multiple output random forests [9] to ORFs, and 2) Previous work in ORFs is limited to categorical output, we extend this to regression trees so that the continuous Q-function can be approximated. We also introduce an *expanding trees* method to the ORF cannon wherein the number of trees used in random forest regression begins small when the first data points come in, and is increased as more data comes in (the new trees are centred at previously learned trees).

We apply our methods to several OpenAI gym environments [7]: blackjack, inverted pendulum, and lunar lander compared to state-of-the-art Deep Q-Networks (DQNs) and traditional discrete temporal difference (TD) learning. We show that our version of online random forests (which we refer to as RL-ORF for *reinforcement learning with online random forests*) can successfully approximate action-value functions for Q-learning in some gyms, with performance exceeding DQNs in the blackjack gym. In Section 2, we describe related work (including online random forests, and offline methods for tree based inference). In Section 3, we describe our methods. In Section 4, we describe our experiments on OpenAI RL gyms. In Section 5 and 6, we conclude and provide directions of future work.

Chapter 2

Literature Review

We provide an overview of reinforcement learning to introduce the key elements and terms of our work. Then, we discuss some recent work in function approximation methods for Q-learning, such as approximation through offline tree-based methods and deep neural networks, including experience replay. Lastly, we review online random forest methods and general advances in RL that we adapt for our model, including online bagging and temporal knowledge weighting.

2.1 Reinforcement Learning: Overview

Reinforcement learning (RL) is one of the three main machine learning paradigms [29]. RL differs from the other two main paradigms (supervised learning, and unsupervised learning) in that RL methods do not require pre-determined datasets from users. The RL agent learns optimal action policy, which maximizes expected total rewards, by updating the policy using data obtained from simultaneously interacting with the environment in a trial-and-error style. RL problems are usually formalized by Markov decision processes (MDP), defined by a tuple of 5 elements $M_t = (S, A, P, R, \gamma)$ [29]. Here, S is the state that represents the status of the environment at time t. A is the action that defines the possible actions available for the agent to choose from. P is the transition distribution $P(s_t, a_t, s_{t+1})$ where $s \in \mathcal{S}, a \in \mathcal{A}$ that maps the action executed by the agent at state s at time t with the succeeding state. R is the reward function that the agent receives as the immediate feedback for executing action a at a given state s_t. If the agent progresses along a sequence of interactions with the environment after time step t, the agent would receive corresponding rewards $R_{t+1}, R_{t+2}, \ldots$. Then, the total return the agent can expect at time step t can be computed as follows:

$$G_t = R_t + R_{t+1} + R_{t+2} + \ldots + R_T. \tag{2.1}$$

Here T is the final time step at which an episode ends. T can be either a fixed integer or ∞ depending on the environment. In this thesis, we only consider finite horizon cases. In RL, we make assumption that future rewards are discounted by a factor of γ at each time step

[22]. The goal of which the agent tries to maximize is defined as *return*, which is denoted:

$$G_t = \gamma^0 R_t + \gamma^1 R_{t+1} + \gamma^2 R_{t+2} + \ldots = \sum_{i=0}^{T} \gamma^i R_{t+i}. \tag{2.2}$$

$\gamma \in [0, 1]$ is a discount factor determining how much impact a future reward has in the present. For example, if $\gamma = 0$, the agent only learns to maximize the immediate reward. However, if $\gamma = 1$, the agent becomes more far-sighted and takes the future rewards into account more strongly when evaluating its actions [29]. For the agent to assess its performance during learning, it needs to be able to evaluate the return for being at a given state or executing an action at the state in terms of the expected reward. To do this, we use what is called *value functions* [29]. There are two major kinds of value functions. Firstly, the *state-value function* is defined as the expected return for the agent being at state s at time step t:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]. \tag{2.3}$$

Here v(·) is the state-value function and π is the *policy* which refers to the probability distribution over all possible actions at given states that the agent follows when selecting an action [29]. Therefore, $v_\pi(s)$ can be interpreted as the expected return when starting in state s and the agent's action selections are based on π. One of the most commonly used policies in RL is *ε-greedy* policy. With this policy, the agent chooses a random action from the action space with a probability of ε, ε being a small real number between 0 and 1. The agent would select an action from the learned value-functions with a probability of 1-ε. We use ε-greedy policy as π throughout this paper. Similar to the state-value function, we can also determine the value of an action a taken at state s, which is referred to as the *action-value function*:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]. \tag{2.4}$$

These value functions are the building blocks of reinforcement learning algorithms. In the following subsections, we briefly go over some fundamental methods that utilize value functions for maximizing expected future returns. In subsection 2.1.1, we introduce a value iteration method called *temporal difference learning* including SARSA (State–action–reward–state–action) and Q-learning. Then, we move on to more complex state-of-the-art *value function approximation* methods such as linear function approximators, tree-based methods, and Q-networks in Section 2.2 to Section 2.5.

2.1.1 Temporal Difference Learning

By finding the optimal *state-value function* or *action-value function*, the agent can take an optimal sequence of actions that yields maximum *expected return*. However, since we do not know the true value functions, the agent must learn the value functions from experience. Hence, most of the RL algorithms' objective is to estimate the optimal value functions. The problem of estimating the value function for a given policy π is called the *prediction* problem. There are two most fundamental prediction methods in RL: Monte Carlo (MC) methods and temporal difference learning [28]. In a simple MC prediction, value function updates occur in the following manner:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]. \tag{2.5}$$

Here, $\alpha \in [0, 1]$ is the learning rate. We use an estimate V for v_π since we do not know the true v_π. The term in the brackets can be interpreted as the error term that the agent aims to minimize. From Equation 2.8, it is noticeable that the MC method must wait until the end of the episode to be able to compute G_t for determining the error term. This limitation can be troublesome when the episode is long or when $T = \infty$. Unlike MC methods, temporal difference learning does not need to wait for episodes to end before updating value function estimates. *Temporal difference learning* (TD learning) was first introduced in 1988 by Sutton et al. [28]. The algorithm uses a value function estimate from one step ahead to update the current value function. TD algorithms get around the limitation of MC methods by utilizing the following recurrence relation:

$$\begin{aligned} G_t &= R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots && (2.6)\\ &= R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \ldots) && (2.7)\\ &= R_t + \gamma G_{t+1}. && (2.8) \end{aligned}$$

When the agent takes an action at a given state, the state-value estimate can be updated:

$$V(S_t) \leftarrow V(S_t) + \alpha[r_t + \gamma G_{t+1} - V(S_t)]. \tag{2.9}$$

The fact that $V(S_{t+1})$ is an estimate for G_{t+1}, we get the following *TD-error* at time t defined as:

$$\delta_t = R_t + \gamma V(S_{t+1}) - V(S_t). \tag{2.10}$$

This gives rise to the simplest TD method that, which is specified by the following update:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)]. \tag{2.11}$$

At each time step, the agent chooses an action and observes the successor state and corresponding reward, which is then immediately used to update the estimate of the current state-value function. Because the value function estimate is updated at every step, this method is often referred to as TD(0), or the *one-step TD* method [29]. Both MC and TD methods are called *bootstrapping* methods since an estimate $V(S_{t+1})$ is used for updating another estimate $V(S_t)$. The full algorithm of the TD(0) prediction method is shown in Algorithm 1.

Algorithm 1 One-step TD method for estimation of v_π

Require: Learning rate $\alpha \in (0, 1]$
Require: Discount rate $\gamma \in [0, 1]$
Require: Initialize V(s) arbitrarily for all $s \in \mathcal{S}$, except $V(terminal) = 0$
1: **for** episode 1:E **do**
2: Initialize S
3: **for** time step t in $1 : T$ **do**
4: Select action $a \in \mathcal{A}$ by some policy π for S
5: Execute action a, observe R, S'
6: $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
7: $S \leftarrow S'$
8: **end for**
9: **end for**

As the updating progresses, δ converges to 0. The fact that TD methods do not require the agent to reach the end of the episode to update the estimate allows the algorithm to be implemented in an online, fully incremental fashion [28]. This characteristic is advantageous in environments where the episode's length is very long or in infinite horizon tasks with no terminal state. Also, it has been proven that in discrete settings for any fixed policy π, TD(0) converges to v_π if α is sufficiently small [28].

2.1.2 SARSA: On-policy TD Control

The goal of finding the optimal policy can be achieved by utilizing the predicted value function. The process of approximating optimal policies is referred to as the *control* problem. There are two major types of TD-control methods: SARSA and Q-learning. In both methods, the action-value function (Q-function) is estimated directly rather than the state-value function (V-function) for optimal policy approximation. With the V-function, the agent must compute the V-function for all possible successor states before choosing the next action. However, with the Q-function, the agent only needs to consider action values in the current state and select an action through $\text{argmax}_{a \in \mathcal{A}} Q(s, a)$ without knowing anything about possible successor states and their values [29]. SARSA is an on-policy TD-control method. The term *on-policy* means the policy the agent is learning is the same policy that

determines the agent's next action. The name *SARSA* rose from the elements of the tuple of events $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$ that the algorithm uses to update the action-value function estimate for the state-action pair (S_t, A_t) which is drawn in a similar way to Equation 2.14:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)], \tag{2.12}$$

which occurs after each state transition. If S_{t+1} is the terminal state, then $Q(S_{t+1}, A_{t+1})$ is zero. The full algorithm of SARSA is expressed in Algorithm 2 below.

Algorithm 2 SARSA: On-policy TD Control

Require: Learning rate $\alpha \in (0, 1]$
Require: Discount rate $\gamma \in [0, 1]$
Require: Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, except $Q(terminal, \cdot) = 0$

1: **for** episode $1 : E$ **do**
2: Initialize S
3: Choose A from S using policy derived from Q (e.g. ε-greedy)
4: **for** time step t in $1 : T$ **do**
5: Execute action A, observe R, S'
6: Choose A' from S' using policy derived from Q (e.g. ε-greedy)
7: $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
8: $S \leftarrow S', A \leftarrow A'$
9: **end for**
10: **end for**

In the discrete case, SARSA converges to an optimal policy with probability 1 given that all state-action pairs are visited an infinite number of times, and the policy converges in the limit to the greedy policy [29].

2.1.3 Q-learning: Off-policy TD Control

Q-learning is an off-policy TD learning algorithms proposed by Watkins et al. [32], which can be defined as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{2.13}$$

Here Q is the learned action-value function. Q-learning is *off-policy* because the policy the agent is learning (i.e. target policy) is not the same as its behaviour policy. More specifically, the target policy the agent updates is the absolute greedy policy $\text{argmax}_a Q(s_{t+1}, a)$ at given states and available actions. However, the agent uses a different policy as the behavior policy, such as ε-greedy, to take action in the successor state. This off-policy attribute of Q-learning has an advantage over on-policy methods such as SARSA.

Algorithm 3 Q-learning: Off-policy TD control

Require: Learning rate $\alpha \in (0, 1]$
Require: Discount rate $\gamma \in [0, 1]$
Require: Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, except $Q(terminal, \cdot) = 0$
1: **for** episode $1 : E$ **do**
2: Initialize S
3: **for** time step t in $1 : T$ **do**
4: Choose A from S using policy derived from Q (e.g. ε-greedy)
5: Execute action A, observe R, S'
6: $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{a \in \mathcal{A}} Q(S', a) - Q(S, A)]$
7: $S \leftarrow S', A \leftarrow A'$
8: **end for**
9: **end for**

Like SARSA, for discrete settings it has been proven that the estimated action-value function converges to the optimal as every possible state-action pair (s_t, a_t) is visited infinitely many times during an episode [32].

2.2 Value-function approximation

The value iteration processes described in the above TD learning methods are called *tabular* methods. They create an array to store all possible state-action pairs and corresponding value functions, hence they suffer from the high computational cost (the curse of dimensionality) if there are a large number of possible states and actions. Also, an update on an estimate of a state value (or a state-action value) leaves the estimate values of all other states unchanged [29]. Instead of tabular methods, we can utilize a model such that an induction made on a state should return a function that is close to the target value function. Also, updating the model would not only update the estimated value of one state but also affect the estimated values of all (or many) other states [29]. This estimation gives rise to a process referred to as *function approximation* which is one of the major breakthroughs in TD learning [29]. Here, the updating process can be interpreted as an input-output relationship: the input being the states, and the outputs being the value functions for those states. This means any supervised learning algorithm can conceivably be utilized for learning and producing value function estimates [29]. The supervised learning methods here are called *function approximators*. The function approximators must be trained to minimize the differences between the estimates and the value function computed from the observed rewards. This training is done as the agent interacts with the environment. However, even the most sophisticated supervised learning methods such as artificial neural networks (ANNs) and tree-based methods are trained in batch data settings, and the distribution of the data is

assumed to be static. In reinforcement learning, suitable supervised learning methods need to be able to learn from incrementally acquired data and adapt to distribution shifts of the target functions [29].

There are several studies that utilized different types of function approximators including linear methods such as polynomials [2], Fourier basis [17] and radial basis [24], and non-linear methods including tree-based methods [9] and artificial neural networks (ANNs; [22, 30]). While the linear methods are fast and straightforward to implement and usually come with better convergence guarantees, their performance depends on prior knowledge about the features in the system. Another limitation of the linear approximators is that the interactions between the features cannot be accounted for unless coded separately. For example, the presence of feature i could be good only with the absence of another feature j. Considering possible interactions between the features could be problematic when there are a large number of features to consider. These shortcomings have directed recent focus to non-linear function approximators. Here, we introduce tree-based methods [9], and deep neural networks [22] in the following sections.

2.3 Tree-based Batch Mode Reinforcement Learning

One of the earlier attempts to utilize tree-based supervised learning methods in reinforcement learning includes *fitted Q-iteration* introduced by Ernst et al. in 2005 [9]. For each iteration, the algorithm builds a training set composed of observations obtained by randomly exploring the environment for a certain number of episodes as inputs. The expected reward function is induced by a supervised learning method trained using previous steps as outputs. The model is then re-trained on a training set. Ernst et al. examined various tree-based ensembles and showed that their approach is effective for extracting relevant information from the observations [9]. However, an extensive training set size is required to obtain good approximations, which lead to high computational costs and is not possible in online settings. Furthermore, a tree-based ensemble must be re-built at each iteration, confining the algorithm to batch scenarios [1].

2.3.1 Fitted Q-Iteration

At each iteration of fitted Q-iteration, a training set composed of the full set of four-tuples (S_t, A_t, R_t, S_{t+1}) is compiled. The data is obtained by the agent randomly exploring the environment for a given number of episodes, along with the action-value functions approximated at the previous step. A regression algorithm then uses this training set to update the action-value functions [9]. We provide pseudocode for fitted Q-iteration in Algorithm 4 below.

Algorithm 4 Fitted Q-Iteration

Require: Training set size: N

Require: Initialize $\hat{Q}_n = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

1: Set $n = 0$
2: $i^l = (s_t^l, a_t^l)$, $o^l = r_t^l$
3: **while** Stopping condition not met **do**
4: $\quad n+=1$
5: $\quad$ Build training set($=TS$) $= \{(i^l, o^l), l = 1, ..., N\}$ based on $\hat{Q}_{n-1}$
6: $\quad o^l \leftarrow r_t^l + \gamma \max_{a' \in \mathcal{A}} \hat{Q}_{n-1}(S_{t+1}^l, a')$
7: $\quad$ Induce $\hat{Q}_n = $ Fit(i, o)
8: **end while**

Here i^l indicates the input data, and o^l the output data for training set index l. At the first iteration, input/output pairs composed of the state-action pair and the observed rewards are used to approximate action-value functions. In the subsequent iterations, only the output values of the training set are updated based on the $\hat{Q}$-functions computed at the preceding steps [9]. The authors state that the supervised learning processes in the iterating sequences are independent, and therefore the learned model can reach the best bias/variance trade-off at each step. The user determines the stopping conditions. For example, one option is to stop when the distance between $\hat{Q}_n$ and $\hat{Q}_{n-1}$ falls below a certain value. However, the authors state that for some supervised learning methods, there is no guarantee that $\hat{Q}_n$ would converge. When the iteration ends, the optimal control policy is derived by $\hat{\pi}_n(s) = \text{argmax}_{a \in \mathcal{A}} \hat{Q}_n(s, a)$. In [9] Ernst et al. incorporated a fitted Q-learning framework with various tree-based methods including KD-Tree [3], CART [6], Tree Bagging [4], Extra-Trees [12], and Totally Randomized Trees [12]. They found that while Tree Bagging and Extra-Trees performed better than the other methods, Extra-Tree's performance significantly surpassed others. There are several advantages of the fitted Q-iteration framework combined with the tree-based methods. It is computationally efficient, effective in high dimensional problems and low dimensional problems with small sample size, and is robust to noise [1, 9]. On the other hand, since the framework is a grid-based process like standard Q-learning, it shares similar limitations to standard Q-learning, involving increases in computing time and memory requirements with larger state and action spaces. Also, a large training set size is crucial in order to obtain good approximations, which also adds further to the computational costs. Furthermore, in the fitted Q-iteration framework, a supervised learning model must be built and updated at each iteration, confining the algorithm to the batch scenario [1].

2.3.2 Adapting to different action spaces

In their experiments, Ernst et al. [9] suggested several methods for coping with varying types of action spaces. For discrete action spaces, they suggested splitting the training samples according to each $a \in \mathcal{A}$, and then train regression models based on each training set. Then, the action for the following state is selected by $\text{argmax}_{a \in \mathcal{A}} \hat{Q}_n(s, a)$ [9]. For continuous action space, the authors stated that since $\hat{Q}_n(s, a)$ is a piecewise-constant function over the action a with a fixed state s, it is sufficient for a single regression tree to compute the value of $\hat{Q}_n(s, a)$ for a finite number of values of A, one in each hyperrectangle delimited by the values of split thresholds in the tree [9]. However, for ensembles of trees, the computational intensity for following the same scheme for every tree in the ensembles might become inefficient as the number of the split thresholds might be much higher [9].

2.4 Deep Q-learning

2.4.1 Q-network: Non-linear Q-function approximator

Artificial neural networks (ANNs) have been one of the most widely used non-linear function approximators [28]. Deeply-layered ANNs led breakthroughs in a variety of fields including computer vision [18, 27] and speech recognition [8, 13]. In reinforcement learning, ANNs that approximate Q-functions are refer to as *Q-networks* [22]. Q-networks are trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i [22]:

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \pi(\cdot)}[(y_i - Q(s, a; \theta_i))^2]. \tag{2.14}$$

Here θ is the weights of the network, π is the behaviour policy distribution, and $y_i = \mathbb{E}_{s' \sim \pi}[r + \gamma max_{a' \in A} Q(s', a'; \theta_{i-1})|s, a]$, which is the action-value function obtained from the behaviour distribution. Differentiating the loss function with respect to the weights, we get the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \pi; s'}[(R + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)]. \tag{2.15}$$

which can then be optimized using gradient descent methods. In practice, a ε-greedy policy is often selected as the behaviour distribution which selects a random action with probability ε, and the greedy strategy from the target policy with probability $1 - \epsilon$ [22].

One of the earlier successful attempts to incorporate ANNs into RL is TD-gammon by Gerald Tesauro in 1995 [30]. Utilizing a multilayer perceptron with one hidden layer as the value-function approximator for a model-free RL algorithm similar to Q-learning enabled the machine to play backgammon at a super-human level [22]. However, the same schematics did not work well in chess, Go, or checkers. Additionally, it was found that Q-

learning with Q-networks sometimes result in divergence of the network [31]. More recent work to implement deep learning in reinforcement learning includes neural fitted Q-learning algorithms (NFQ) developed by Riedmiller et al. [25]. NFQ updates the Q-networks' weights by optimizing the sequence of loss functions in Equation 2.14 using a RPROP algorithm (resilient backpropagation; [22]). The algorithm successfully solved simple tasks including pole balancing and car-on-the-hill [25]. However, like in Fitted Q-learning [9], this algorithm utilizes a batch of observations to update the Q-network, which has a computational cost proportional to the data size [22]. Mnih et al. adopt stochastic gradient descent, which has a lower cost per iteration and allows lightweight updates on large-scale data [22], a process which they named *deep Q-learning* [22]. In this thesis, we compare our methods with deep Q-learning.

2.4.2 Experience replay

Attempts to applying ANNs directly to RL algorithms encounter several issues [22]. Firstly, many successful ANN architectures assume learning from a large amount of pre-labeled data is relatively balanced. However, for RL, the networks must be able to learn from sparse and noisy data. Also, the training data in RL often comprises of sequences of highly correlated states, and the distribution of the data constantly changes as the agent learns new behaviours [22]. Another breakthrough in deep Q-learning is *experience replay* [19], and this is where the method used in [22] differs from TD-gammon [30] or NFQ [25]. At each time step t, the agent interacts with the environment and outputs a tuple of experience $e_t = (s_t, a_t, r_t, s_{t+1})$, and this is stored in a dataset $\mathbb{D} = e_1, e_2, \ldots, e_N$ referred to as *replay memory* [22]. Replay memory only stores the most recent N observations. When updating the Q-values, we use a fixed-sized minibatch drawn at uniform random $e \sim \mathbb{D}$. Then, the agent chooses the next action to execute by an ε-greedy policy from $Q(s_{t+1}, A)$. Utilization of experience replay has several advantages over learning directly from the most recent experience only. The method is more data-efficient since each observation participates in updating the Q-network multiple times instead of being thrown away after being used only once. Also, using randomly drawn samples mitigates correlation between the consecutive observations, reducing the prediction variance of the Q-network during Q-value updates. Finally, experience replay prevents the behaviour policy from shifting to one side and getting stuck. For instance, without experience replay, if the agent selects an action to move to the left at a state, then the samples gathered from the consecutive time steps would consist mostly of the information about the left side of the environment, and almost no information about the right side of the environment. This phenomenon can cause the parameters of the Q-network to plateau or even diverge [22]. Mnih et al. combined deep Q-learning with experience replay. The combined process is named *deep reinforcement learning*, a full algorithm of which is expressed in 12.

Algorithm 5 Deep reinforcement learning: Deep Q-learning with experience replay

Require: Initialize replay memory D to capacity N

Require: Initialize action-value function Q with random weights

Require: Initialize environment with random beginning state

1: **for** episode in $1 : E$ **do**
2: Initialize sequence s_1 and preprocessed sequence $\phi_1 = \phi(s_1)$
3: **for** t in 1:T **do**
4: Select $a_t = \begin{cases} \text{argmax}_{a \in \mathcal{A}} Q(\phi(s_t), a; \theta) & \text{with probability } 1 - \varepsilon \\ \text{random action} & \text{with probability } \varepsilon \end{cases}$
5: Execute action a_t, observe reward r_t and s_{t+1}
6: Set $s_{t+1} = s_t$, and preprocess $\phi_{t+1} = \phi(s_{t+1})$
7: Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
8: Sample random minibatch of transitions $(\phi_\ell, a_\ell, r_\ell, \phi_{\ell+1})$ from D
9: Set $y_\ell = \begin{cases} r_\ell & \text{for terminal } \phi_{\ell+1} \\ r_\ell + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{\ell+1} \end{cases}$
10: Perform gradient descent on $(y_\ell - Q(\phi_\ell, a_\ell; \theta))^2$
11: **end for**
12: **end for**

Here ϕ is a function that modifies a state into a user-defined format. For example, the authors in [22] converted a given state of size 210x160 with three color channels (red, green, blue) to a greyscale image with a size of 84x84. The authors produced the input data to the Q-function by applying the preprocessing step to the last 4 frames of history and stacking them, making a representation of size 84x84x4. In this thesis, when we compare to the DQN, we use the values of the states directly instead of an image representation of the environment (i.e., without rendering the environment as an image). We describe the details of the state values for each environment in Chapter 4 (Experiments).

2.5 Online random forests

The methods we develop in Chapter 3 make reference to previous work in online random forests. These methods work well when there is a small correlation between the base learners [12]. Online random forests developed by Saffari et al. in 2009 by combining online bagging with extremely randomized forests [26]. The trees in the online random forest start with a single terminal node. When new data is observed, each tree takes in new data according to a random integer drawn from the Poisson distribution. The terminal node of the tree performs splits only when the statistics computed from a series of new data exceeds a certain threshold. Trees are replaced by a new tree based on their out-of-bag errors (OOBEs). In the following subsections, we discuss the key elements of online random forests including

extremely randomized trees [12], online bagging [23] and temporal knowledge weighting [26]. The full algorithm for online random forests by Saffari et al. [26] is shown in Algorithm 6

Algorithm 6 Online Random Forests

Require: Sequential training example $\langle x, y \rangle$
Require: The size of the forest: M
Require: The minimum number of samples that a tree has to observe: η
Require: The minimum gain a test must achieve to split: β
Require: The temporal knowledge weighting rate: φ

```
1:  for each tree m in 1:M do
2:     Draw c ~ Poisson(1)
3:     if c > 0 then
4:        for iin1 : c do
5:           age_m += 1
6:           j = findLeaf(x) # locate the node where the new input x belongs to
7:           updateNode(j, ⟨x, y⟩) # According to Algorithm 8
8:        end for
9:     else
10:       updateOOBE_m(⟨x, y⟩) # According to Algorithm 9
11:    end if
12: end for
13: TemporalKnowledgeWeighting(M) # According to Algorithm 10
```

2.5.1 Extremely randomized trees

Ensemble methods work well when there is a small correlation between the base learners that have low biases [5]. The expected mean squared prediction error (MSPE) of the ensembles can be expressed as:

$$\mathbb{E}(\text{MSPE}) = \sigma^2 \left(\rho + \frac{(1-\rho)}{M} \right). \tag{2.16}$$

Here $\mathbb{E}(\text{MSPE})$ is the expected mean squared prediction error, σ^2 is the variance of the prediction errors, ρ is the correlation between the errors from base models, and $|M|$ is the number of base models in an ensemble M. In the original random forest methods proposed by Breiman et al. (2001) [5], k variables from a total of K explanatory variables are chosen at random with replacement, where usually $k = \sqrt{K}$ for classification, and $k = K/3$ for regression (but tuning is recommended). The cut-point at each variable is selected by examining them with one of the quality measurements such as Gini index or entropy for classification [6]. Each tree is grown using a bootstrapped training sample of size N. The randomness from the bootstrapping reduces the correlation between the trees because the splits in the trees become less similar to each other [5]. In the extremely randomized trees proposed by

Guerts et al. (2006) [12], the authors aim to further decrease the correlation between the learners by injecting more randomness into the splitting process. Along with the random selection of explanatory variables, the cut-points at the variables are also selected at random rather than by optimal quality measurements. To be more specific, each node in each tree creates its own set of $k < K$ randomly generated tests $H = \{(g_1, \theta_1), (g_2, \theta_2), \dots, (g_k, \theta_k)\}$. Here, each g is a randomly selected explanatory variable, and each θ is a split location. Then, each test is evaluated with respect to one of the quality measurement methods such as the Gini index or the entropy:

$$\text{Gini impurity: } L(D_j) = \sum_{i=1}^{K} p_i^j (1 - p_i^j), \text{ Entropy: } L(D_j) = -\sum_{i=1}^{K} p_i^j \log(p_i^j). \tag{2.17}$$

Here D_j denotes a set of data in node j, and p_i^j is the probability of class i in node j, and K is the number of classes. Each node j computes the count of each label i denoted by $\mathbf{p_j} = [p_1^j, p_2^j, \dots, p_K^j]$, and also collects $\mathbf{p_{jrh}} = [p_1^{jrh}, p_2^{jrh}, \dots, p_K^{jrh}]$, and $\mathbf{p_{jlh}} = [p_1^{jlh}, p_2^{jlh}, \dots, p_K^{jlh}]$, where $\{jlh\}$ and $\{jrh\}$ indicate the left (l) and right (r) partitions created by a test h at node j. Thus, the information gain resulting from a test h is:

$$\Delta L(D_j, h) = L(D_j) - \frac{|D_{jlh}|}{|D_j|} L(D_{jlh}) - \frac{|D_{jrh}|}{|D_j|} L(D_{jrh}). \tag{2.18}$$

Here R_{jlh} and R_{jrh} are the left (l) and right (r) child nodes created by a test h at node j, and $|\cdot|$ denotes the number of samples in that node. The higher the gain lower the impurity in the children nodes. Thus, the test with the highest gain is selected as the decisive test for that node. Guerts et al. [12] claim that this further injection of randomness along with the averaging via ensemble contributes to reducing variance in the prediction errors more strongly than the weakly randomized methods.

2.5.2 Online bagging

Suppose we have training data of size N, and a number of trees M in a forest. In the standard bagging methods developed by Breiman in 1996 [4], each tree m is trained on a bootstrap sample of size N from the original training set. The number of times an observation in the initial training set is included in the bootstrapped sample can be expressed as a binomial distribution parameterized by the probability $\frac{1}{N}$: [26]:

$$P(C = c) = \binom{N}{c} \left(\frac{1}{N}\right)^c \left(1 - \frac{1}{N}\right)^{N-c}. \tag{2.19}$$

As $N \to \infty$, the distribution of C tends to a Poisson distribution with a mean of 1. In [26], online bagging is utilized to grow ExtraTrees in a non-recursive manner. As we observe a new training data $\langle x, y \rangle$, where x is a vector of values of the explanatory variables, and y

is the corresponding response, choose the data c times for each tree, and update the trees accordingly. This process is done independently for each tree.

2.5.3 Online decision trees

In the online setting, the label proportions at each terminal node are collected over time. The node needs to know when to perform a split. For this, two things need to be specified. 1) Sample size that each terminal node needs to observe to produce a robust set of statistics 2) A threshold for the information gain that produces a split that makes a good prediction. Saffari et al. [26] propose two new parameters: The number of samples a terminal node has to observe (η), and the threshold of gain that a split has to achieve (β) before a split proceeds. Let $|D_j|$ be the number of observations that a node j has observed so far. A split proceeds only if $|D_j| > \eta$, and $\exists h \in H : \Delta L(D_j, h) > \beta$. After splitting, $\mathbf{p_{jlh}}$ and $\mathbf{p_{jrh}}$ are passed on to the newly generated left and right children nodes, allowing the new terminal nodes to participate in making predictions immediately using the inherited statistics. The algorithms from Saffari et al. for creating and updating nodes are shown in Algorithms 7 and 8.

Algorithm 7 createChild($\mathbf{p}_{j.h_{split}}$)

Require: Node index: $j + 1$

Require: Statistic from the parent node j: $\mathbf{p}_{j.h}$

Require: User-defined number of tests: k

1: Set $\mathbf{p}_{j+1} = \mathbf{p}_{j.h}$ # Inherit the statistic from the parent node.

2: Select k variables $\{g_1, \ldots, g_k\}$ at random from $\{z_1, \ldots, z_K\}$

3: Select k split points $\{\theta_1, \ldots, \theta_k\}$
where $\theta_i \sim \text{Uniform}(min(g_i), max(g_i)) \forall i = 1, \ldots, k$.

4: Construct a set of tests $H_j = \{h_1, \ldots, h_k\}$ where $h_i = \{g_i, \theta_i\} \forall i = 1, \ldots, k$

Algorithm 8 updateNode(j, $\langle x, y \rangle$)

Require: Node index: j

Require: Training example: $\langle x, y \rangle$

Require: A subset of training dataset in node j: D_j

Require: Number of training samples observed: $|D_j|$

Require: The minimum number of training samples a tree has to observe to split: η

Require: The minimum gain a test must achieve to split: β

Require: A set of randomly created tests in node j: H_j

1: $|D_j| += 1$
2: Update $\mathbf{p}_j = \{p_1, \ldots, p_k\}$ where $p_i =$ (Number of times the label i appeared in j) / $|D_j|$
3: Update $\mathbf{p}_{jlh}$ and $\mathbf{p}_{jrh} \forall h_j \in H_j$
4: Compute $\Delta L(D_j)$
5: **if** $|D_j| > \eta$ and $\exists h_j \in H_j : \Delta L(D_j, h_j) > \beta$ **then**
6: $h_{split} = \text{argmax}_{h \in H} \Delta L(D_j, h)$
7: createChild($\mathbf{p}_{jlh_{split}}$) # create left child node
8: createChild($\mathbf{p}_{jrh_{split}}$) # create right child node
9: **end if**

2.5.4 Temporal knowledge weighting

For trees that are not updated with a new observation $\langle x, y \rangle$, in other words, for trees with c equal to 0, the observation is used for *unlearning* old information. Since c is the number of times observation is included in a bootstrapped sample, an observation with c equal to 0 can be interpreted as out-of-bag data. Temporal knowledge weighting, developed by Saffari et al. [26] uses this to estimate trees' out-of-bag errors (OOBEs), and discards trees with large OOBEs. To prevent trees from being discarded in their early stages of growth, Saffari et al. employ another parameter ϕ; the temporal knowledge weighting rate. Only trees with $\text{age}_m > \frac{1}{\phi}$ are subjected to being discarded. Here, age_t is the number of samples a tree m has observed. The tree to be discarded is then randomly chosen and replaced by a new tree with just one node (a stump). While the influence of discarding one tree in an ensemble is relatively low, continually replacing the trees allows the ensemble to adapt to the changes in the sample distribution throughout time [26]. The algorithms for updating OOBEs of the trees and temporal knowledge are explained in Algorithms 9 and 10. The full algorithm for online random forests is shown in Algorithm 6.

Algorithm 9 updateOOBE($\langle x, y\rangle$)

Require: Tree index: m

Require: Training example: $\langle x, y\rangle$

Require: Number of training samples the tree has observed: age_m

1: $y_{pred} = m(x)$
2: **if** $y \neq y_{pred}$ **then**
3: $\quad error_m += 1$
4: $\quad OOBE_m = error_m / age_m$
5: **end if**

Algorithm 10 TemporalKnowledgeWeighting(M)

Require: Ensemble of on-line trees: M

Require: Temporal knowledge weighting rate: φ

Select a tree m randomly from M that satisfies $\{m | m \in M, age_m > 1/\varphi\}$
if $OOBE_m >$ c $\sim$ Uniform$(0, 1)$ **then**
$\quad m =$ newTree() # replace the tree with a new tree with just one node
end if

2.5.5 Summary

This chapter outlined some of the fundamental ideas of reinforcement learning, including value-functions, value iteration processes, and value-function approximation. Function approximation methods have advantages over tabular methods such as SARSA or Q-learning. They train supervised learning models from the agent's experience that can estimate value functions in a generalized manner without the need to store all possible action-value functions. We pointed out that for Q-learning with function approximation, the approximator needs to be able to update its parameters incrementally. Q-networks used in deep Q-learning along with experience replay have been the most notable example of online function approximators. Finally, we introduced online random forests. They start with one terminal node (a stump), which incrementally splits after observing a user-defined number of samples. Temporal knowledge weighting algorithm allows *un-learning* of trees by replacing a tree with a large out-of-bag error with a new tree. In the following chapter, we discuss how we incorporate these ideas to develop our novel methods for reinforcement learning with online random forests: We extend the online random forest algorithms presented in this chapter, along with experience replay and the action selection methods for discrete action spaces.

Chapter 3

Methods

Our method makes use of the advantages of online random forests. Their ability to learn and unlearn in real-time allows them to easily adapt to changes in the distribution of the agent's experience, making the online random forest a suitable function approximator for Q-learning. Moreover, combined with the method described in the following sections, our function approximator can approximate multiple action-value functions in a regression setting (Section 3.1 and 3.2) and make full use of available information gained from the environment (Section 3.3). Furthermore, we found that starting with a small number of trees and expanding it in a later episode can enhance the agent's performance while saving computational costs. The summarized version of our method is as follows: at each time step, an action is taken by the agent, and the corresponding four-tuple (s_t, a_t, r_t, s_{t+1}) is observed from the environment. This tuple is saved in the replay memory. Replay memory is described in Section 2.4.2 of Chapter 2. The ensembles are used for making action-value estimates. Each tree in the ensemble updates either its terminal node or out-of-bag-error (OOBE) depending on an integer drawn from a Poisson distribution. The OOBE we use is described in the next subsection. When trees' ages reach $1/\varphi$, they become subject to replacement according to their OOBEs. At episode ζ, the tree with the lowest OOBE is duplicated several times so that the ensemble size is expanded to $|M_{\max}|$. The episode terminates when the agent reaches the terminal state of the environment. An overview of our method is given in Figure 3.1. This algorithm makes use of the algorithms *findLeaf, updateNode, updateOOBE, and expandTrees* which are discussed later in this Chapter. The pseudo-code for the overall algorithm is provided in Algorithm 11. Here, ζ and φ are positive real numbers (parameters) for our model. Throughout our experiments, we used $\zeta = 100$ and $\varphi = 1/5,000$.

3.1 Online random forest in regression setting

In the original online random forest in [26], Saffari et al. focus on classification. However, we need online trees with regression in order to specify approximators of action-value functions

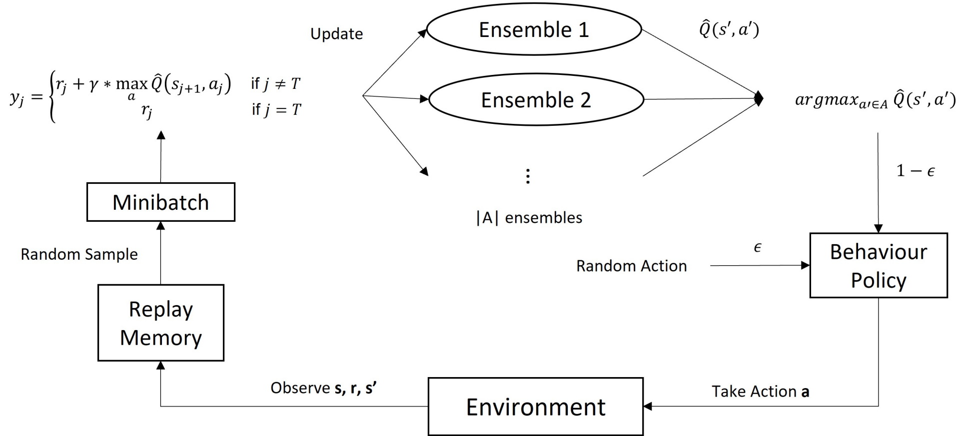


Figure 3.1: Schematics of Q-learning with online trees. Transitions from interactions with the environments are stored in replay memory. Random mini-batches of the transitions sequentially update the online trees, which approximate action-value functions for the succeeding state-action pairs. The behaviour policy chooses the action with the largest Q-function with the probability of $1 - \varepsilon$, or a random action with ε. See Algorithm 11 for detail.

(as expected rewards are continuous). Transition to regression can be done by replacing the objective of splits from maximizing the information gains to maximizing the change in residual sum of squares (RSS) shown in (3.3), as done in offline tree methods.

$$RSS_j = \sum_{i=1}^{|D_j|} (y_i - \bar{y}_j)^2 \tag{3.1}$$

$$RSS_{jh} = \sum_{i=1}^{|D_{jlh}|} (y_{ilh} - \bar{y}_{jlh})^2 + \sum_{i=1}^{|D_{jrh}|} (y_{irh} - \bar{y}_{jrh})^2 \tag{3.2}$$

$$\Delta L(h) = RSS_j - RSS_{jh}. \tag{3.3}$$

In addition, the computation of out-of-bag errors of the trees must be done differently. In classification, OOBE of a tree m is simply the fraction of the new observation's label y_u for some $u \in k$ in node j:

$$OOBE_m = \frac{\sum_{i=1}^{|D_j|} \mathbb{1}(y_i \neq y_u)}{|D_j|}. \tag{3.4}$$

which falls between 0 and 1. However, for RL problems, OOBEs must be computed in a regression manner. For this, we adopt the mean absolute error to assess OOBEs of the trees that are computed base on λ most recent observations that the terminal nodes have observed.

Algorithm 11 Q-Learning with online trees

Require: Replay memory to capacity N_{mem}
Require: Minibatch size: N_{min}
Require: Temporal knowledge weighting rate: φ
Require: Episode at which ensemble size expansion occurs: ζ
Require: Maximum ensemble size: $|M_{\text{max}}|$

```
1: for episode i in 1 : E do
2:   for time step t in 1 : T do
3:     Select a_t = nextAction(s_t) # According to Algorithm 12.
4:     Execute a_t and obtain tuple e_t = (s_t, a_t, r_t, s_{t+1}), store it in the memory
5:     Randomly sample minibatch of (s_ℓ, a_ℓ, r_ℓ, s_{ℓ+1}) from the memory
6:     Set y_ℓ = { r_ℓ + γ * max_a Q̂(s_{ℓ+1}, a_ℓ)   if s_{ℓ+1} is not terminal
                 { r_ℓ                               if s_{ℓ+1} is terminal
7:     for tree m in 1 : M_{a_j} do
8:       Draw c ~ Poisson(1)
9:       if c > 0 then
10:        for k in 1 : c do
11:          Set age_m+ = 1
12:          Set j = findLeaf(s_ℓ) # Find the terminal node j to which s_ℓ belongs
13:          updateNode(j, ⟨s_ℓ, y_ℓ⟩) # According to Algorithm 13.
14:        end for
15:      else
16:        updateOOBE_m # According to Equation 3.5
17:      end if
18:    end for
19:    # Perform temporal knowledge weighting on ensemble M_{a_ℓ}
20:    Randomly select m from M_{a_ℓ} such that age_m > 1/φ
21:    if OOBE_m > c ~ Uniform(0, 1) then
22:      Replace the tree with a new tree with just one node
23:    end if
24:  end for
25:  if i = ζ then
26:    expandTrees(M, |M_max|) # According to Algorithm 14
27:  end if
28: end for
```

$$\text{OOBE}_{\text{reg.}} = \frac{1}{\lambda}\sum_{i=1}^{\lambda} \min\left(\text{abs}\left(\frac{y_i - m(x_i)}{y_i + \mu}\right), 1\right). \tag{3.5}$$

Here, $m(x)$ is a predicted value from a tree m, and μ is a small arbitrary real number that prevents division by zero. The term in the function $\min(\cdot)$ tells us how much the predicted value is off from the true response. OOBEs computed this way also falls between 0 and 1.

3.2 Computing $\max_{a\in\mathcal{A}}\hat{Q}(S,a)$ when $|A|>1$

In reinforcement learning, there is typically more than one possible action available at any given non-terminal state. In order for agents to determine which action to choose, each action needs its own action-value. This gives rise to a need for function approximators to be able to produce a number of outputs equal to the number of available actions for the iteration process described in Equation 2.12. Deep neural networks naturally solve this issue by having the corresponding number of output nodes in the output layer. However, the online random forest from [26] can approximate only one output at a time. To resolve this, we adopt an idea suggested by Ernst et al. [9] for handling discrete action spaces. In our method, we grow one forest for each action available in the given reinforcement learning environment. Each ensemble starts with just one node and grows independently on sample observations from experience replay. Each forest approximates the corresponding action-value function, and then the largest among them is returned. For a fixed state s and action a at time step t, the Q-learning iteration requires a computation

$$\hat{Q}(s_t,a_t)\leftarrow\hat{Q}(s_t,a_t)+\alpha[r_t+\gamma\max_{a\in\mathcal{A}}\hat{Q}(s_{t+1},a)], \tag{3.6}$$

assuming α=1. Using our method, the term after γ in the brackets can be expressed as:

$$\max_{a\in\mathcal{A}}\hat{Q}(s_{t+1},a)=\max(\hat{Q}_1(s_{t+1},a_1),\hat{Q}_2(s_{t+1},a_2),...,\hat{Q}_{|A|}(s_{t+1},a_{|A|})). \tag{3.7}$$

Here, $\hat{Q}(\cdot)$ is the function approximator, and $\hat{Q}_i(s_{t+1},a_i)=M_i(s_{t+1})\forall i=1,\dots,|A|$ where $M(\cdot)$ denotes prediction by the ensemble. An equivalent method is also applied for action selections: with probability ε, the agent chooses its consecutive action by taking the largest action-value among the approximations induced from the ensembles. This is encoded in Algorithm 12 below.

Algorithm 12 nextAction(s_t)

Require: State at time step $t+1$: s_{t+1}
Require: Probability of taking a random action: ε
Require: Ensemble of trees: M
Require: Number of possible actions: $|A|$
1: Draw $c\sim\text{Uniform}(0,1)$
2: **if** $c<\varepsilon$ **then**
3: **return** $a_{t+1}=$ random action from $\mathcal{A}$
4: **else**
5: **return** $a_{t+1}=\text{argmax}\Big(M_1(s_{t+1}),M_2(s_{t+1}),\dots,M_{|A|}(s_{t+1})\Big)$
6: **end if**

3.3 Partial randomness in split point selection

In reinforcement learning, function approximators must be able to use as much information from the environment as possible. However, In the standard online random forest, each node selects a small subset of features at random [26]. This aspect of online random forest may prevent the function approximators from utilizing important information from the given state, leading to sub-optimal splits. Since the impact of each split is severe on the ensembles' performances, the nodes must utilize as much information from the state as possible. Instead of choosing a small subset, we let all the state features be included in the split tests. In other words, we remove randomness in the split variable selection. However, the split points for each test are still chosen at random. We express the procedure in the following algorithm.

Algorithm 13 createChild($\mathbf{p}_{j \cdot h}$)

Require: Node index of the child node: $j+1$
Require: Statistic from the parent node j: $\mathbf{p}_{j \cdot h}$
Require: The number of explanatory variables: K
Require: State variables: $\{z_1, \ldots, z_K\}$

1: Set $\mathbf{p}_{j+1} = \mathbf{p}_{j \cdot h}$ # Inherit the statistic from the parent node.
   # Apply partial randomness in split point selection.
2: Select K split points $\{\theta_1, \ldots, \theta_K\}$, where $\theta_i \sim \text{Uniform}\left(\min(z_i), \max(z_i)\right) \; \forall i$
3: Construct a set of tests $\text{H} = \{\ h_1, \ldots, h_K\ \}$, where $h_i = (z_i, \theta_i) \; \forall i$

3.4 Expanding ensemble size

Growing a large number of trees in each forest can be computationally intensive. In the early stages of learning, the amount of information learned may be expressed using only a small number of trees. We propose training a small number of trees for a certain period and then duplicating the best tree a large number of times later on in the learning process. These duplicated trees are then further grown and updated as more data come in. Our method begins the learning process with just 100 trees (a default value) in each forest and, when the episode reaches ζ, expands the number of trees up to a value specified by a hyperparameter $|M_{\max}|$ according to the following algorithm.

Algorithm 14 expandTrees(M, $|M_{\max}|$)

Require: Ensemble of trees: M,

Require: Initial size of ensemble: $|M_{\text{init}}|$,

Require: Maximum size of ensemble: $|M_{\max}|$.

1: $m_{\text{best}} = \{m|m \in M, \text{OOBE}_m = \min_{m \in M} \text{OOBE}_m\}$

2: **for** i in $|M_{\text{init}}| + 1 : |M_{\max}|$ **do**

3: $\quad M = \text{append}(M, m_{best})$

4: **end for**

Chapter 4

Experiments and results

We apply the RL-ORF model to the following OpenAI gyms: blackjack (*Blackjack-v1*), inverted pendulum (*CartPole-v1*), and lunar lander (*LunarLander-v2*). OpenAI gyms provide a variety of simulated environments that are widely used as benchmarks in the reinforcement learning literature [7]. In each case, we compare our methods to DQNs (deep Q-networks; [22]). For both DQN and RL-ORF, we used $\gamma = 1.0$, $\varepsilon = 0.5$, and ε-decay rate $= 0.99$ with the minimum $\varepsilon = 0.01$. The replay memory size was fixed at 10,000 in all experiments, and the minibatch size was fixed at 32. For DQN, the neural networks comprised two hidden layers. In each experiment, we compared performances of different hidden layer sizes and learning rates with the Adam optimizer [16]. The input and output layer sizes differed depending on the gym. For instance, there are 4 elements in the state space in the inverted pendulum gym and 2 in the action space, which resulted in 4 input nodes and 2 output nodes. For RL-ORF, we tested different values of η and modulated whether or not to expand the ensemble size using *expandTrees*. All trees were fully grown without pruning, along with default parameters $\beta = 0.01$, $\varphi = 1/5,000$, $|M_{\text{init}}| = 100$, and $|M_{\max}| = 200$. The ensemble size was expanded (if set to do so) at episode 100, meaning $\zeta = 100$. For each experiment, we do 100 random restarts and 1,000 episodes for each run.

In supervised learning, the performance of models can be measured by training the models on the training datasets and assess their accuracy on the validation sets. However, such a paradigm cannot be applied in reinforcement learning since learning and assessing occur simultaneously in an online fashion. Therefore, we used the average total reward per episode that the agent had scored during training to measure performance evaluations. Each epoch consisted of 1,000 episodes, and the total rewards were averaged over 100 epochs.

All experiments were conducted on Intel i7-8565U 1.8GHz CPU and 16GB RAM with python version 3.7.8. The DQNs were trained using *PyTorch* version 1.8.1. The code for our experiments is available under an open-source license. Portions of our codebase use a modified version of the open-source python code from [20] and [33]. The DQN method is $\sim$103 times faster than our method, and the total compute for all experiments was

approximately 10 CPU years. We report significant results for nominally significant p-values (0.05).

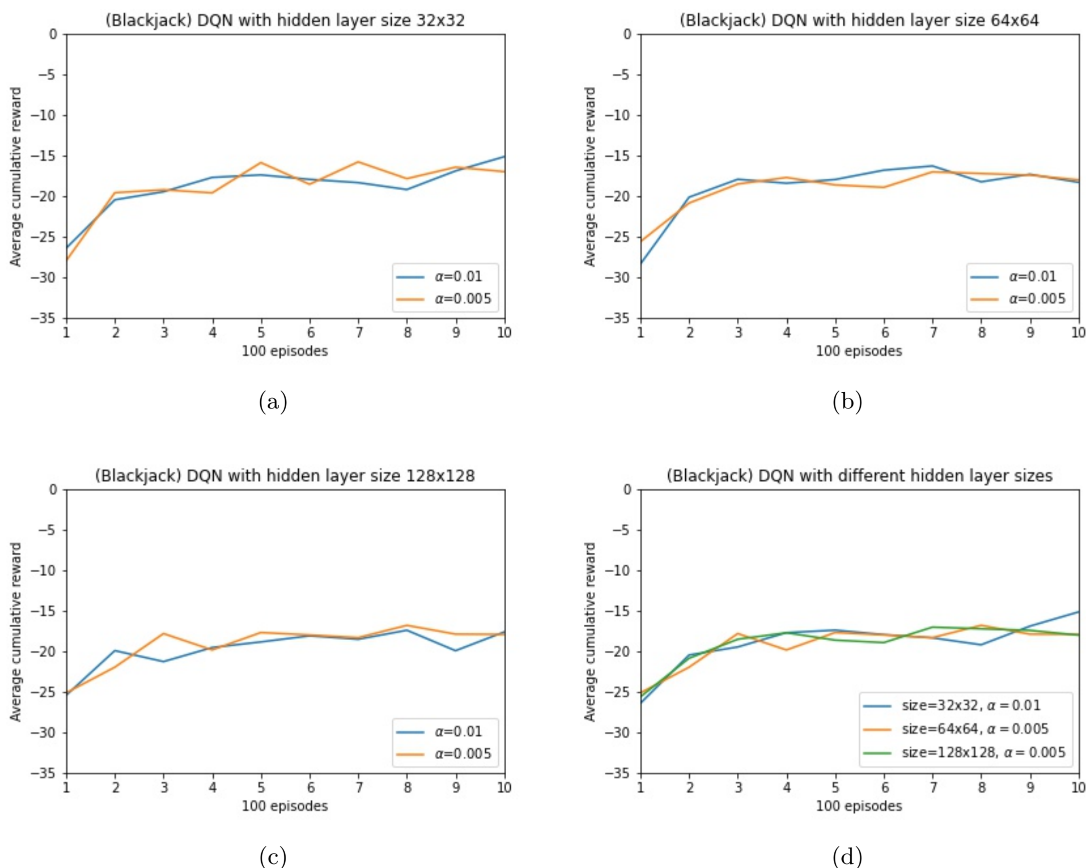
4.1 Blackjack

In OpenAI's blackjack environment, the state is a tuple containing three elements: the agent's hand, the dealer's hand, and whether or not the agent holds an ace. The ace can be treated as either one point or eleven. The agent starts with two cards in hand, whereas the dealer starts with only one. The player draws a card by choosing to *hit*. Suppose the player's hand exceeds 21, the player *busts* and loses the game immediately. The player can choose to end the turn before bust by choosing to *stay*. Subsequently, the dealer begins drawing cards. The dealer keeps drawing until the hand reaches 17 or higher or busts. If the dealer busts, the player wins the game. If neither the player nor the dealer busts, the hands are compared and the one with a higher hand wins. If the hands are equal, the result is a draw, and the game ends. The deck is reshuffled in each game. The reward is awarded given as *win* (+1), *draw* (0), and *lose* (-1). For more details, see [7].

4.1.1 Blackjack: Results

We first illustrate our results on the performances obtained by implementing DQN and RL-ORF as function approximators in Q-learning. For blackjack only, the results are shown in terms of average cumulative reward per 100 episodes. For example, an average cumulative reward of -21.1 at the 200th episode means the agent's total reward won between 101 and 200 episodes was -21.1 (i.e., the agent lost 21.1 more times on average than it won). We discovered that our method outperformed the DQN settings we had carried out (Figure 4.4). Before comparing the methods, we report the performances of DQN and RL-ORF in different parameter settings.

In Figure 4.1, we compared the performance of DQN with different hidden layer sizes and learning rates. We varied the parameters of the DQN extensively but found that the settings did not modulate the performance. Figure4.1 (d) shows that the best performance at episode 1,000 was achieved by implementing DQN with hidden node size 32x32 and the learning rate 0.01.



(a) (b) (c) (d)

Figure 4.1: Blackjack: Average rewards obtained using DQN as the Q-function approximator with different hidden node sizes and learning rates. (d) presents that DQN with hidden node size = 32x32 and $\alpha = 0.01$ performed the best at episode 1,000 among the combinations.

We show RL-ORF results with various η values and modulation of ensemble size expansion in Figure 4.2. As illustrated in Figure 4.2 (a), (b), (c), we observed that varying η had created more recognizable differences in the average cumulative rewards the agent received than the ensemble size expansion. The ensemble size expansion did not boost the performance of the function approximator with a fixed η. The best RL-ORF had settings $\eta = 32$ and ensemble size expansion from 100 to 200 at episode 100 as depicted in (d).

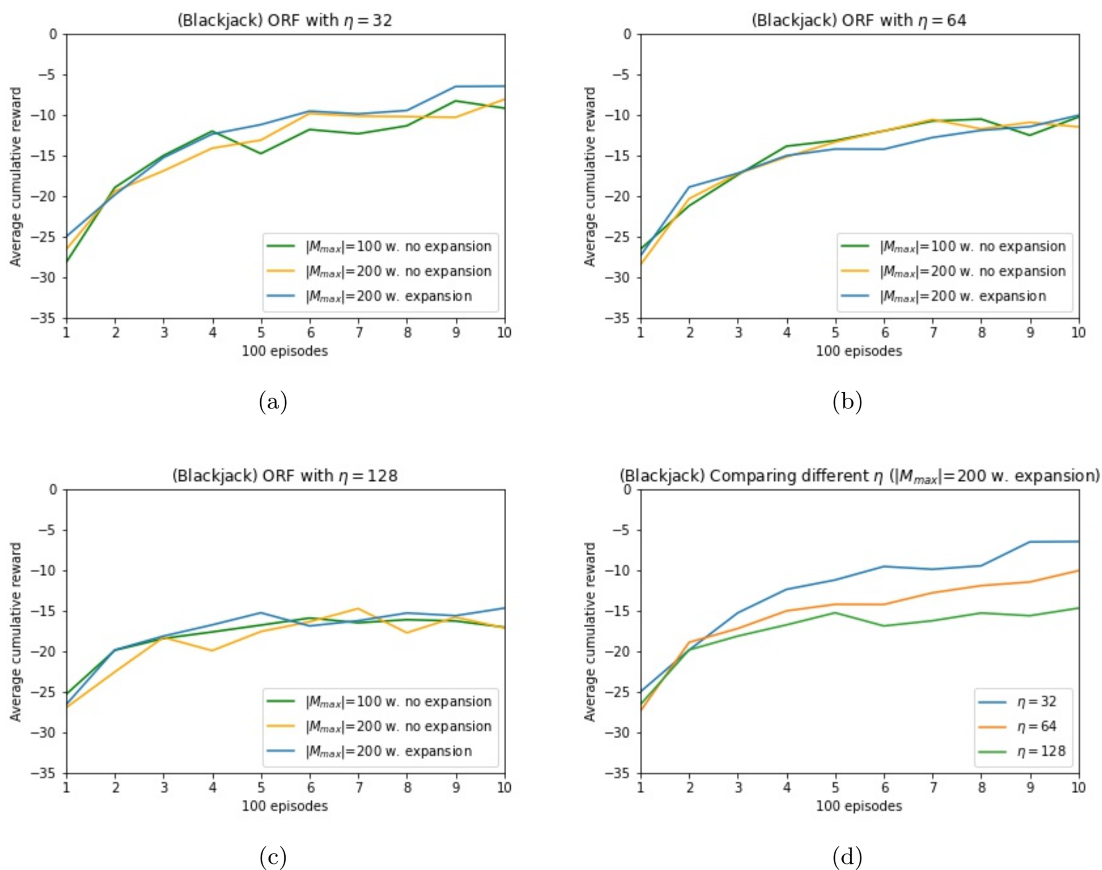
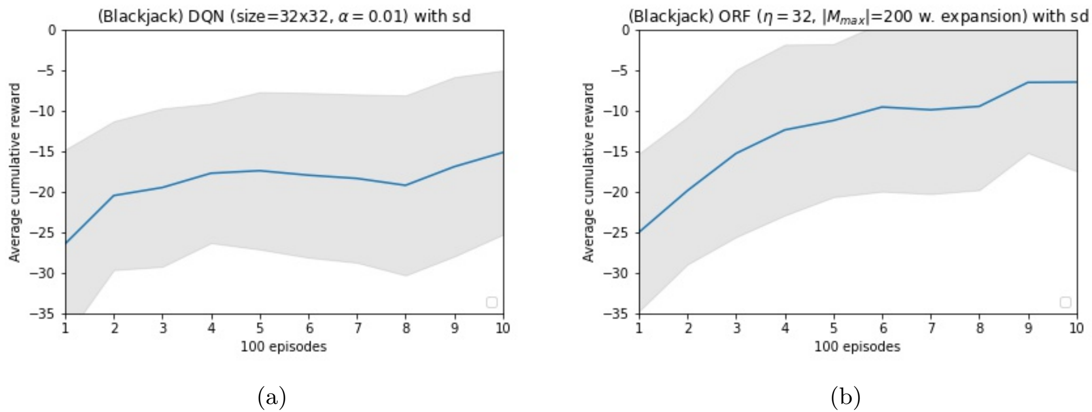


Figure 4.2: Blackjack: Average rewards obtained from ORF with different η and whether ensemble size is expanded at episode 100. (d) shows that RL-ORF with η=32 and the ensemble expansion gives the best performance among the examined parameter sets.

The error regions of the best DQN (hidden layer size=32x32, α=0.01) and RL-ORF ($|M_{\max}| = 200$ with ensemble size expansion) shown in Figure 4.3 illustrates that no one method has noticeably larger or smaller variability than the other. Here, the error regions are $\pm$ 1 standard deviation across the 100 restarts from the mean. The RL-ORF performed significantly better for this gym than the DQN, as seen in Figure 4.4. The best RL-ORF parameters showed a mean cumulative reward of -6.51 with a standard deviation of 10.97 at episode 1,000, and the best DQN had a mean of -15.18 with a standard deviation of 10.10. We provide the results of statistical tests to support our findings in Table 4.1 and Table 4.2.



(a) (b)

Figure 4.3: Blackjack: Average cumulative reward per 100 episodes from the best ones of (a) DQN and (b) RL-ORF with the error regions. Neither of the two methods showed noticeably larger or smaller performance fluctuations compared to the other.

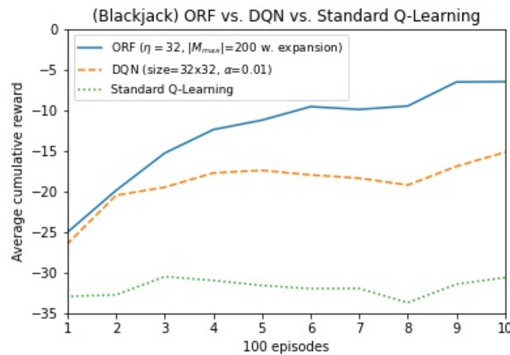


Figure 4.4: Blackjack: Comparison of the average cumulative rewards per 100 episodes between the best DQN, RL-ORF, and standard Q-learning. RL-ORF has a larger average cumulative reward than DQN at episode 1,000. Statistical evidence is provided in Table 4.2.

4.1.2 Blackjack: Statistical tests

We performed various tests to find statistical evidence of whether one method outperforms the other. Before comparing the means, we first tested the normality of the average cumulative rewards from DQN and RL-ORF. Large p-values from Shapiro-Wilk tests (Table 4.1) provided no evidence that the distributions of the performances were not normal with the significance level of 0.05. Thus we proceed to compare the two means using a t-test. One-sided t-tests for the difference in the means at episode 300 yields the p-value 0.665, which does not reject the null hypothesis that the performances were equal. However, at episode 1,000, the was p-value 2.271E-5 rejecting the null hypothesis that the performances were

equal (i.e., the RL-ORF has higher mean performance). Details of these statistical tests are provided in Table 4.2.

Shapiro-Wilk Test					
Approx.	Parameters	Episode	Statistic	P-value	Conclusion
DQN	size=32x32, α=0.01	300	0.970	0.886	Cannot reject H0
DQN	size=32x32, α=0.01	1,000	0.921	0.330	Cannot reject H0
RL-ORF	η=32, $\lvert M_{\max}\rvert$=200, exp	300	0.979	0.113	Cannot reject H0
RL-ORF	η=32, $\lvert M_{\max}\rvert$=200, exp	1,000	0.976	0.060	Cannot reject H0

Table 4.1: Blackjack: Results of Shapiro-Wilk test for the normality of the average rewards data at episodes 300 and 1,000 obtained by DQN and RL-ORF. In all cases, we cannot reject the null hypothesis that the data is normally distributed.

T-Test			
Episode	Statistic	P-value	Conclusion
300	-0.428	0.665	Cannot reject H0
1,000	4.249	$2.271 * 10^{-5}$	Reject H0

Table 4.2: Blackjack: Results of one-sided T-test between the mean average cumulative rewards from DQN and RL-ORF. There is a statistical evidence that the mean from RL-ORF is greater than the mean from DQN at episode 1,000 with p-value of 2.272E-5.

Note that the formulation of blackjack by OpenAI reshuffles the deck after every hand, and ties are not in favour of the agent. This means that the game is stacked against the agent, and it is impossible to achieve an average reward larger than zero (as indicated by Figure 4.4). For the blackjack experiment, there was no evidence of catastrophic forgetting in the DQN. *Catastrophic forgetting*, also known as *catastrophic interference* is a phenomenon that occurs when ANNs lose learned information after learning from new experience [21]. Once the neural network becomes good at solving the problem, the new incoming experience that the agent gets is only good cases, leading to a depletion of unsuccessful cases in the experience memory. The function approximator would start to generate high Q-function values for every state-action pair, which degenerates the accuracy of the network. In Figure 4.1, the DQNs show no sign of catastrophic forgetting as the average rewards do not decay over time. In this experiment, there was no evidence that the *expanding trees* method improved performance of the RL-ORF as depicted in Figure 4.2. Evidence recommending *expanding trees* arises in the following experiment. We also apply standard Q-learning (discrete TD

learning) to the blackjack gym, and this method performed worse than both the DQN and the RL-ORF methods, with a mean reward of -28.93 and a standard deviation of 12.97.

4.2 Inverted pendulum

For inverted pendulum by OpenAI's *CartPole-v1*, the agent's objective is to maintain the pole standing on the cart without falling as long as possible. The state-space tuple consists of 4 elements: cart position, cart velocity, pole angle, and pole angular velocity. There are two possible actions: move the cart to the left (0) or right (1). The agent gets a +1 reward for every step taken, including the termination step. See [7] for more details. We modified the reward function for both DQN and RL-ORF with a heavier penalty to enhance the learning speed. In the altered setting, the agent gets -1,000 points for falling. We tested hidden layer sizes: 32x32, 64x64, 128x128, and learning rates: 0.01, 0.005 for DQN. For RL-ORF, different values of η: 32, 64, 128, and whether to expand the ensemble size from 100 to 200.

4.2.1 Inverted pendulum: Results

We show the average rewards per episode for the varying parameter values of DQN in Figure 4.5. (a), (b), and (c) illustrate that the lower learning rate returned better performances than when α=0.01 in all hidden layer sizes. However, like in Blackjack, a larger number of hidden nodes did not noticeably improve the overall performance throughout the episodes.

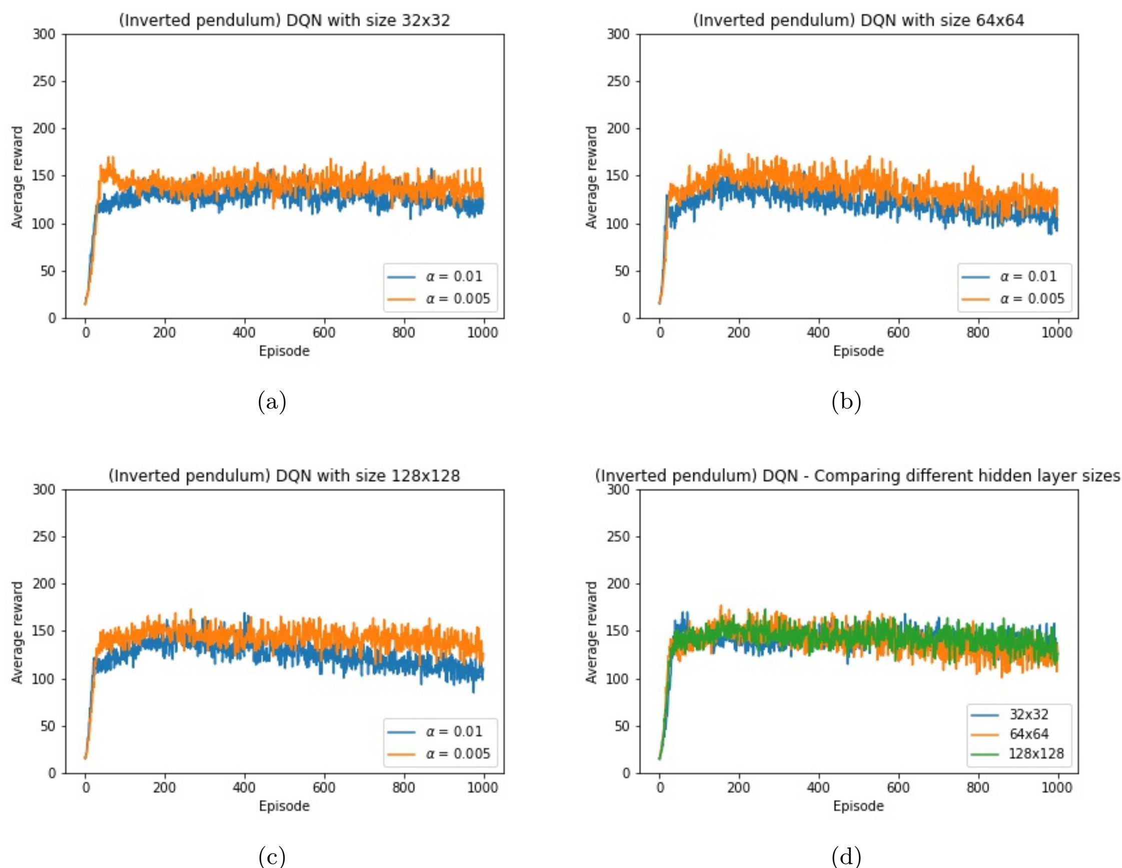


Figure 4.5: Inverted Pendulum: Average rewards per episode obtained by DQN with different hidden node sizes and learning rates. (d) shows that the best DQN has hidden node size 128x128 and $\alpha = 0.005$.

For RL-ORf Figure 4.6 (a), (b) and (c), we noticed that the forest size expansion resulted in better performances in all three schematics. We observed a decrease in the average reward at the point of the expansion, which quickly recovered. The negative impact of the ensemble expansion was more severe with smaller η, although varying the η itself did not show notable differences in the performance of the agent. Figure 4.6 (d) shows that the best performing RL-ORF at episode 1,000 had $\eta = 256$ with ensemble size expansion with the mean average reward of 157.0, outperforming the other two schematics (145.32 for $\eta = 512$, and 139.26 for $\eta = 1,024$).

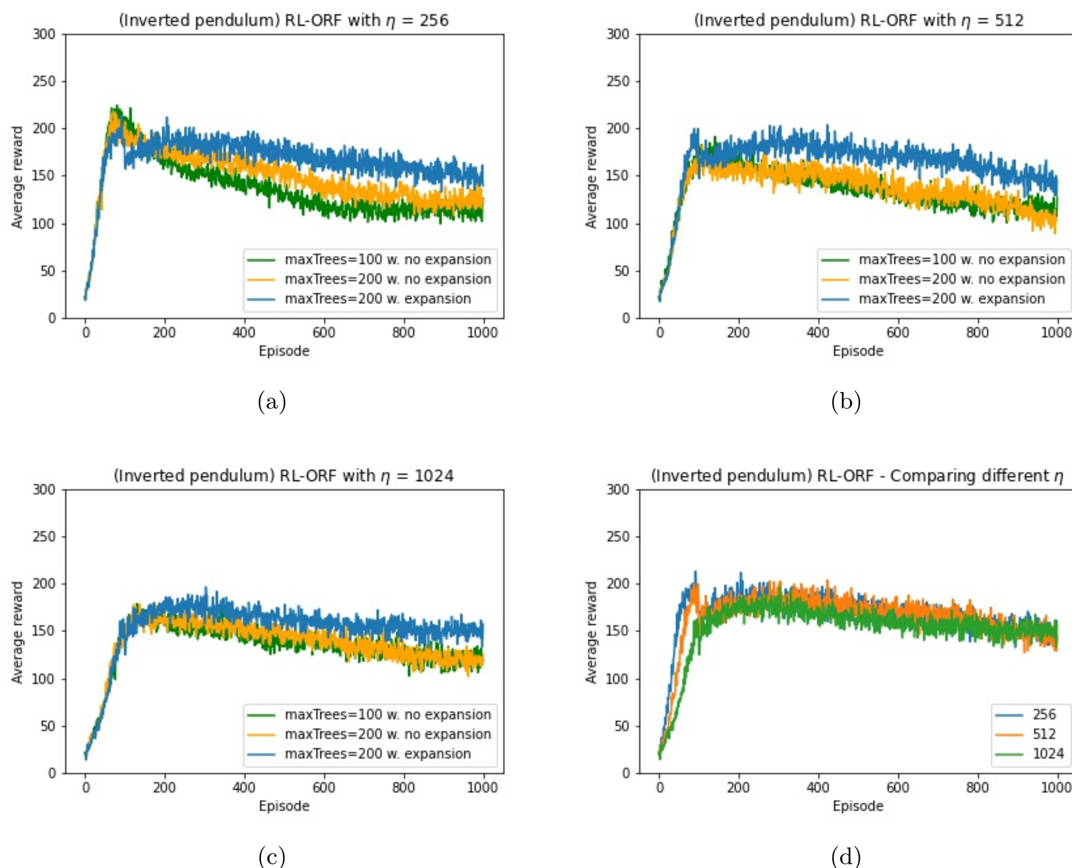


Figure 4.6: Inverted Pendulum: Average rewards per episode obtained by RL-ORF with varying sizes of η and whether ensemble size is expanded. In all (a), (b), and (c), expanding the ensemble size produces the best performance at episode 1,000. The best RL-ORF has $\eta = 256$ and the ensemble size expanded from 100 to 200 at episode 100.

4.2.2 Inverted pendulum: Statistical tests

We performed statistical tests to investigate whether there is any evidence of performance improvements by forest size expansion. The cases compared were: η=256 with forest expansion and η=256 without forest expansion. The results with p-values less than the significance level 0.05 rejected the null hypothesis that the average rewards at episodes 300 and 1,000 in both cases were normally distributed. See Table 4.3 for details. We used the Mann-Whitney U test to compare the two independent data that are not normally distributed. The test showed that at episode 300, there is no evidence that the probability of the mean average reward from RL-ORF with ensemble size expansion exceeds that of RL-ORF without the size expansion. At episode 1,000, however, the p-value was less than 0.05, providing enough evidence to reject the null hypothesis (that the accuracy of RL-ORF without expansion was the same as the accuracy of RL-ORF with expansion; or specifically

that Pr(accuracy of RL-ORF with expansion $>$ accuracy of RL-ORF without expansion) = Pr(accuracy of RL-ORF without expansion $>$ accuracy of RL-ORF with expansion)). Details of the test is reported in Table 4.4.

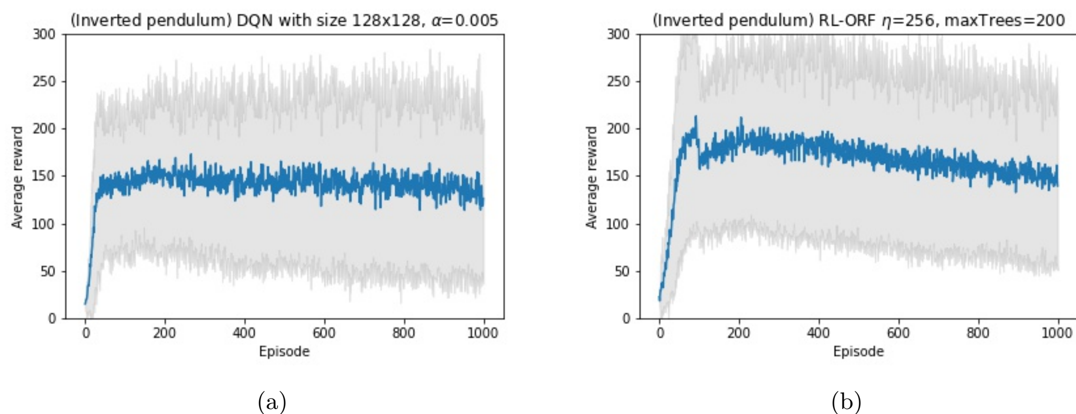
Shapiro-Wilk Test					
Approx.	Parameters	Episode	Statistic	P-value	Conclusion
RL-ORF	η=256, $\|M_{\max}\|$=200, Exp	300	0.803	$2.97 * 10^{-10}$	Reject H0
RL-ORF	η=256, $\|M_{\max}\|$=200, Exp	1,000	0.809	$4.68 * 10^{-10}$	Reject H0
RL-ORF	η=256, $\|M_{\max}\|$=200, no Exp	300	0.794	$1.89 * 10^{-10}$	Reject H0
RL-ORF	η=256, $\|M_{\max}\|$=200, no Exp	1,000	0.788	$1.26 * 10^{-10}$	Reject H0

Table 4.3: Inverted Pendulum: Results from Shapiro-Wilk tests for normality of the average rewards generated by RL-ORF with and without ensemble size expansion. With the p-values smaller than the significance level, we provide statistical evidence that the data is not normally distributed in all cases.

Mann-Whitney U Test			
Episode	Statistic	P-value	Conclusion
300	5,557.0	0.068	Cannot reject H0
1,000	5,654.0	0.042	Reject H0

Table 4.4: Inverted Pendulum: Results on Mann-Whitney U test for comparison between the means of RL-ORF with ensemble size expansion and without expansion. We provide evidence that RL-ORF with ensemble size expansion outperforms the one without the expansion at episode 1,000.

From Figure 4.5 and 4.6, we select the best DQN and RL-ORF to compare the two mean average rewards: DQN with hidden layer size=128x128 and learning rate=0.005, and RL-ORF with $\eta = 256$ and ensemble size expansion. In Figure 4.7, we plot the two cases with their error regions. The error regions are $\pm$ 1 standard deviation from the mean, as in the blackjack case. The error regions indicate that neither of the two methods has noticeably larger performance variability in later episodes. At episode 1,000, the best DQN had a mean average reward of 120.0 and a standard deviation of 92.0, whereas the best RL-ORF had a mean average reward of 157.0 with a standard deviation of 88.8.



(a) (b)

Figure 4.7: Inverted Pendulum: Average rewards per episode from the best of (a) DQN and (b) RL-ORF with error regions. The figures show that neither of the two has recognizably smaller performance fluctuations compared to the other.

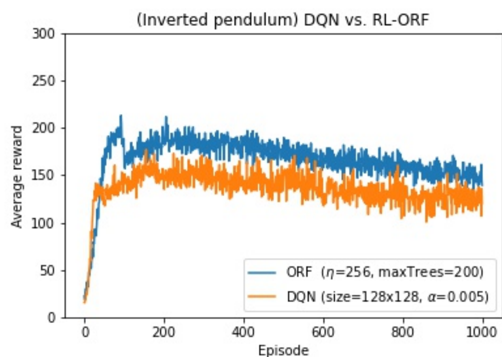


Figure 4.8: Inverted Pendulum: Comparison between the RL-ORF and DQN in average rewards per episode. The statistical test shown in Table 4.5 shows that our method outperforms the DQN.

As depicted in Figure 4.7 and Figure 4.8, we observed that the average rewards for both methods slowly decreases over the episode, indicating that they both suffered from catastrophic forgetting. In Figure 4.8, RL-ORF seems to have a larger mean average reward per episode than DQN. We performed statistical tests to obtain evidence to support our findings. Shapiro-Wilk normality test for the average rewards from DQN and RL-ORF at episodes 300 and 1,000 failed to reject the null hypothesis that each data was normally distributed, as represented in Table 4.5.

Shapiro-Wilk Test					
Approx.	Parameters	Episode	Statistic	P-value	Conclusion
DQN	size=128x128, α=0.005	300	0.813	$6.53 * 10^{-10}$	Reject H0
DQN	size=128x128, α=0.005	1,000	0.877	$1.34 * 10^{-7}$	Reject H0
RL-ORF	η=256, $\lvert M_{\max}\rvert$=200, exp	300	0.803	$2.97 * 10^{-10}$	Reject H0
RL-ORF	η=256, $\lvert M_{\max}\rvert$=200, exp	1,000	0.809	$4.68 * 10^{-10}$	Reject H0

Table 4.5: Inverted Pendulum: Results from the Shapiro-Wilk normality test on the average rewards per episode at episodes 300 and 1,000 generated by the best performing RL-ORF and DQN. The results reports that we have enough evidence to reject the null hypothesis that the data is normally distributed in all four cases.

Since the normality of the data was not guaranteed, we compared the mean average rewards of the best performing DQN and RL-ORF at episodes 300 and 1,000 using the Mann-Whitney U test. The results showed that at 95% significance level, we have enough evidence to reject the null hypothesis that the probability of RL-ORF's mean average rewards is equal to that of the DQN's at episode 300 and 1,000 with p-values 0.0002 and 0.009, respectively. See Table 4.6 for details.

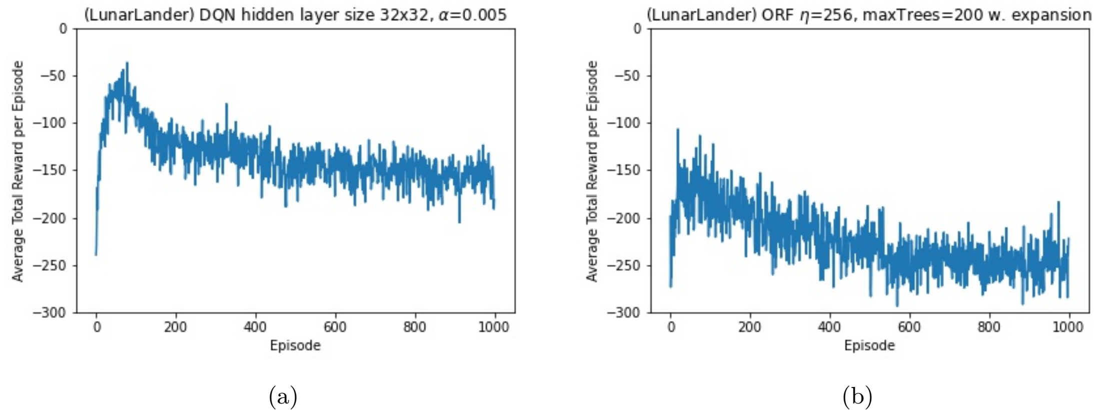
Mann-Whitney U Test			
Episode	Statistic	P-value	Conclusion
300	6,464.5	0.0002	Reject H0
1,000	5,965.0	0.009	Reject H0

Table 4.6: Inverted pendulum: Results from the Mann-Whitney U test comparing the means of DQN and RL-ORF at episodes 300 and 1,000. Small p-values indicate that RL-ORF outperforms DQN in both cases.

4.3 Lunar lander

For the 'lunar lander' gym, the agent tries to land on the landing pad located at coordinates $(0, 0)$. The state-space tuple consists of 8 elements: x-coordinate, y-coordinate, horizontal and vertical velocity, lander angle, angular velocity, right-leg grounded, and left-leg grounded. The agent gets -0.3 points for each frame it fires the main engine, -0.03 for each side engine. If the lander reaches the ground too fast (speed > 0), the lander crashes and receives -100. A successful landing (velocity $= 0$) anywhere awards +100 points, an additional +100 are given for landing on the landing pad. Each leg with ground contact is +10 points. An episode terminates when the lander either crashes or comes to rest [7]. We demonstrate

our results for DQN with hidden layer size 32x32 with learning rate 0.01, and for RL-ORF with $\eta = 256$ and expand ensemble sizes to $|M_{\max}| = 200$.



(a)

(b)

Figure 4.9: Lunar lander: Average rewards per episode obtained by (a) DQN and (b) RL-ORF. Neither one of the function approximators performs well within 1,000 episodes.

As seen in Figure 4.9, neither of the two methods were able to reach the average total reward per episode above 0 within 1,000, with RL-ORF performing significantly worse than DQN. However, this problem is not unsolvable. It was solved in several studies, including [15, 11]. They solved it by implementing a larger network size, lower learning rate, along with an adjustment in the updating frequency [15], and letting the agent learn on a large number of iterations [11]. Both methods used more powerful computational resources than we did. Gadgil et al. [11] mentioned that training 128x128 hidden layers with a learning rate of 0.001 took approximately 8 hours to train, whereas it took around 40 hours for our DQN with size 32x32 and learning rate 0.005 to be trained.

Chapter 5

Discussion

We found in our experiment that the online tree method can be used as an action-value function approximator in Q-learning. By applying our methods in OpenAI's blackjack and inverted pendulum gyms, we discovered that the online tree method could outperform some deep neural networks in terms of average total reward. In the process, we identified several factors that can affect the performance of the online tree method. First, we found that starting the forest size with a small number of trees and then expanding the forest size after the set episode performed better in later episodes than maintaining the initial forest size. This may be because the deterioration of performance, which can inevitably appear in the process of discarding a tree with relatively large out-of-bag error and growing a new tree, has a more significant impact in a 'no expansion' environment. When the forest size is expanded, more than half of the trees temporarily show relatively high performance, which could cancel out the performance degradation due to the correlations between trees and tree re-growth. Furthermore, tree expansion causes a drop in performance at the moment of forest size expansion. However, as shown in our experiment, the impact decreases as η increases and later disappears as the episodes progress. These findings indicate that the ensemble becomes more robust to changes in its structure with larger η. However, online tree methods did not perform well in other environments. In Lunar Lander environment of OpenAI, while the average total reward for the Deep Q-network increased during 1,000 episodes, the online tree method did not show an increasing learning curve. We believe it would be worth investigating what makes it difficult for the online tree methods to solve those problems. Another limitation is that our online tree method has as many user-defined parameters as the deep neural network. In particular, in our experiment, we demonstrated that η has a significant influence on the agent's performance, and optimal η values may not be known a priori. Lastly, Ernst et al. suggest that pruning is necessary even in the ensemble of tree method in a highly stochastic environment [9]. However, in our experiment, all trees were fully grown until they reached a significant difference in RSS without pruning like in [26]. Development of online pruning methods and investigating how it affects the performance of the online tree ensembles could be a subject for future work.

Chapter 6

Conclusion

In our study, we have developed an online random forest method for reinforcement learning. First, we adopted an idea from Ernst et al. [9] to grow an independent ensemble for each action in the discrete action space. In a given state, each ensemble approximates the action-value function for the corresponding state-action pair. The agent selects the action in a greedy manner: choosing the action that returns the largest action-value function. In addition, each tree is subject to termination according to its out-of-bag error after it has gone through a user-defined number of updates. To develop this method, we presented a novel calculation of the out-of-bag error in the regression setting. Next, we proposed ensemble size expansion as a way to save computational costs and boost performance. Lastly, we applied partial-randomness in split point selection to let the ensemble make the most use out of each state. This method is general, and we apply it to gyms without any hand-crafted aspects, without transfer learning, and without building specific representations of the gym. Our experiments demonstrate that we outperform state-of-the-art DQNs and standard TD learning for blackjack, and we outperform DQNs in the inverted pendulum.

In blackjack and the inverted pendulum problem, we found that the number of samples that each terminal node should observe before splitting affects the agent's performance. In addition, rather than fixing the forest size throughout the episodes, we found that expanding the forest size by duplicating the best-performing tree after a particular episode alleviates the performance drop in the long term. Furthermore, we showed that our method could outperform Q-learning with deep neural network if the parameters for our online random forest are appropriately selected.

These gains come at a cost: our method is significantly slower than DQNs (however, our DQN implementation uses *PyTorch*, and we did not attempt to optimize our code with a C implementation matching *PyTorch* optimization techniques). Finally, we found that the lunar lander gym is quite difficult, and neither DQN nor our method performs well within 1,000 episodes for that gym. The lunar lander gym would likely be easier with visual representation and convolution or hand-crafted representations, as is done in standard DQN work. In addition to providing the novel RL-ORF (reinforcement learning online random

forest), our work shows some limitations on the complexity of problems that can be solved with representation-free reinforcement learning.

Bibliography

[1] A. Barreto. Tree-based on-line reinforcement learning. In *Proceedings of the 28th Conference on Artificial Intelligence*, 2014.

[2] A. Barto. *JL Connectionist Learning for Control*. MIT Press, 1995.

[3] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, (4):333–340, 1979.

[4] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

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

[6] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and regression trees*. CRC press, 1984.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. arXiv preprint 1606.01540, 2016.

[8] G. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2011.

[9] D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[10] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau. An introduction to deep reinforcement learning. arXiv preprint 1811.12560, 2018.

[11] S. Gadgil, Y. Xin, and C. Xu. Solving the lunar lander problem under uncertainty using reinforcement learning. arXiv preprint 2011.11850, 2020.

[12] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[13] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[14] K. Gregor, G. Papamakarios, F. Besse, L. Buesing, and T. Weber. Temporal difference variational auto-encoder. arXiv preprint 1806.03107, 2019.

[15] A. Hafiz and G. Bhat. Deep Q-network based multi-agent reinforcement learning with binary action agents. arXiv preprint 2008.04109, 2020.

[16] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. arXiv preprint 1412.6980, 2014.

[17] G. Konidaris, S. Osentoski, and P. Thomas. Value function approximation in reinforcement learning using the Fourier basis. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, 2011.

[18] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. 2012.

[19] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[20] A. Lui. On-line random forests in python. `https://github.com/luiarthur/ORFpy`, 2017. GitHub repository.

[21] M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109–165, 1989.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing Atari with deep reinforcement learning. arXiv preprint 1312.5602, 2013.

[23] N. C. Oza and S. J. Russell. Online bagging and boosting. In *Proceedings of the 8th International Workshop on Artificial Intelligence and Statistics*, 2001.

[24] M. J. D. Powell. *Radial Basis Functions for Multivariable Interpolation: A Review.* Clarendon Press, 1987.

[25] M. Riedmiller. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In *Proceedings of the European Conference on Machine Learning*. Springer, 2005.

[26] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof. On-line random forests. In *Proceedings of the 12th International Conference on Computer Vision*, 2009.

[27] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[28] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[29] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2018.

[30] G. Tesauro. TD-Gammon: A self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[31] J. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[32] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[33] Liu Y. PyTorch 1.x Reinforcement Learning Cookbook. `https://github.com/ Packt Publishing/ PyTorch- 1.x- Reinforcement- Learning- Cookbook`, 2019. GitHub repository.

Appendix A

List of Parameters

α	learning rate
β	minimum gain a split must achieve
γ	discount factor
ε	probability of choosing a random action
ζ	episode at which the number of trees is expanded
η	number of observations a node must observe before splitting
λ	number of observations used to compute OOBE of trees
φ	temporal knowledge weighting rate