Unifying n-Step Temporal-Difference Action-Value Methods

by

Juan Fernando Hernandez Garcia

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Statistical Machine Learning

Department of Computing Science

University of Alberta

Abstract

Unifying seemingly disparate algorithmic ideas to produce better performing algorithms has been a longstanding goal in reinforcement learning. As a primary example, the TD(λ) algorithm elegantly unifies temporal difference (TD) methods with Monte Carlo methods through the use of eligibility traces and the trace-decay parameter λ. The same type of unification is achievable with n-step algorithms, a simpler version of multi-step TD methods where updates consist of a single backup of length n instead of a geometric average of several backups of different lengths.

In this work, we present a new n-step algorithm named $Q(\sigma)$ that unifies two of the existing n-step algorithms for estimating action-value functions — Sarsa and Tree Backup. The fundamental difference between Sarsa and Tree Backup is that the former samples a single action at every step of the backup, whereas the latter takes an expectation over all the possible actions. We introduce a new parameter, $\sigma \in [0, 1]$, that allows the degree of sampling performed by the algorithm at each step to be continuously varied. This creates a new family of algorithms that span a continuum between Sarsa (full sampling, $\sigma = 1$) and Tree Backup (pure expectation, $\sigma = 0$). Our results show that our algorithm can perform better when using intermediate values of σ instead of any of the extremes. Moreover, if we decay σ over time from one to zero, we obtain an algorithm that outperforms other variants of $Q(\sigma)$ with a fixed σ over a variety of tasks.

This work has three main contributions. First, we introduce our new algo-

rithm, n-step $Q(\sigma)$ and provide empirical evaluations of the algorithm in the tabular case. Second, we extend n-step $Q(\sigma)$ to the linear function approximation case and demonstrate its performance in the environment mountain cliff. Third, we combined n-step $Q(\sigma)$ with the DQN architecture and tested the performance of our new architecture — named the $Q(\sigma)$ network — in the mountain car environment.

Throughout our empirical evaluations, we found that the parameter σ often serves as a trade-off between initial and final performance. Moreover, we found that the decaying σ algorithm performed better than algorithms with fixed values of σ in terms of initial and final performance. We also found that in some domains n-step $Q(\sigma)$ with an intermediate value of σ performed better than either of the extreme values corresponding to n-step Tree Backup and Sarsa. Our results represent a compelling argument for using n-step $Q(\sigma)$ over n-step Sarsa or Tree Backup. n-step $Q(\sigma)$ offers a flexible framework that can be adapted to the specifics of the learning task in order to improve performance.

Preface

The $Q(\sigma)$ algorithm was initially proposed by Precup, Sutton, and Singh in 2000. The first formulation and the name of n-step $Q(\sigma)$ as it is presented in Chapter 3 was first introduced by Sutton and Barto in their online book draft in 2016.

The experiments in Chapter 3 and 4 were published in the 32nd AAAI conference in 2018 in a paper titled "Multi-step reinforcement learning: A unifying algorithm" (De Asis, Hernandez-Garcia, Holland, & Sutton, 2018). Kris De Asis implemented the 19-State Random Walk experiment presented in Chapter 3 and proposed the Decaying σ algorithm. Zach G. Holland implemented the Stochastic Windy Gridworld experiment presented in Chapter 3. I implemented the Mountain Cliff experiment for the AAAI paper. In this thesis in Chapter 4 I present a corrected version of this experiment since there was a bug in the original Mountain Cliff environment. This work and the publication "Multi-step reinforcement learning: A unifying algorithm" was led by professor Richard Sutton at the University of Alberta.

To mom and dad

For their hard work, support, and unconditional love.

Acknowledgements

First and foremost, I would like to thank my supervisor Richard Sutton for his guidance and advice. Over the last two years, he has always provided good judgment and advice on research, writing, and life in general. Second, I would like to thank my co-authors Kris De Asis and Zach G. Holland for helping me get my career as a researcher started. Kris was responsible for the 19-state random walk experiment and Zach worked on the stochastic windy gridworld experiment. However, beyond their collaboration on our paper, they have always provided a good source of support and discussion that have led to many new ideas and projects. Third, I would like to thank Huizhen Yu for helping me verify my proofs, which are not part of this thesis, but helped me develop a lot of the intuition behind the n-step $Q(\sigma)$ algorithm. Finally, I would like to thank the Reinforcement Learning and Artificial Intelligence lab for being there to support me and help me at every step of my master's degree.

Contents

1 Introduction **1**

2 Background on Reinforcement Learning **5**
2.1 Reinforcement Learning and Finite Markov Decision Processes 5
2.2 Returns, Policies, and Value Functions 7
2.2.1 Bellman Equations and Backup Diagrams 8
2.2.2 Optimal and Exploratory Policies 9
2.3 Temporal-Difference Methods 10
2.3.1 Action-Value Methods 12
2.3.2 Off-Policy Methods and Backup Diagrams 13
2.4 n-Step Temporal Difference Methods 15
2.4.1 Multi-Step Methods 17
2.5 Approximate Solution Methods 19
2.5.1 The Value Error Objective and Semi-Gradient Methods 19
2.5.2 Linear Function Approximation 21
2.5.3 Non-Linear Function Approximation: Neural Networks 23

3 The n-Step Q(σ) Algorithm **25**
3.1 One-Step $Q(\sigma)$ 25
3.2 n-Step $Q(\sigma)$ 27
3.3 Bias and Variance Analysis of n-Step $Q(\sigma)$ 31
3.4 Empirical Evaluations of Tabular n-Step $Q(\sigma)$ 34
3.4.1 Tabular n-Step Q(σ) for On-Policy Prediction 34
3.5 Tabular n-Step Q(σ) for On-Policy Control 37

4 n-Step $Q(\sigma)$ with Linear Function Approximation **41**
4.1 Empirical Evaluations of n-Step $Q(\sigma)$ with Linear Function Approximation 44

5 n-Step $Q(\sigma)$ with Non-Linear Function Approximation **50**
5.1 The $Q(\sigma)$ Network 51
5.1.1 The Experience Replay Buffer 51
5.1.2 The Target Network and Loss Function 53
5.1.3 Other Algorithmic Details and Hyper-Parameters . . . 54
5.2 Empirical Evaluations of the Q(σ) Network 58

5.2.1 Off-Policy vs On-Policy Sampling Experiment 60
5.2.2 Effect of the Parameter n on the Performance of the $Q(\sigma)$ Network 63
5.2.3 Effect of the Decay Rate on the Performance of the $Q(\sigma)$ Network 65
5.2.4 Effect of the Target Network on the Performance of the $Q(\sigma)$ Network 69
5.2.5 Effect of σ on the Performance of the $Q(\sigma)$ Network . . 72
5.3 Summary of the Empirical Evaluations 75

6 Conclusion **77**
6.1 Future Work 79
6.2 Summary 81

References **82**

Appendix A Effects of the Target Network Update Frequency **86**

List of Tables

4.1 Experiment Results in the Mountain Cliff Environment in Terms of Initial and Final Performance 47
4.2 Experiment Results in the Mountain Cliff Environment in Terms of Average Return After 500 Episodes 48

5.1 Hyperparameters for the $Q(\sigma)$ Network Architecture 59
5.2 Comparison of the Drop in Performance Experienced by the Decaying σ Algorithm with Linear Decay vs Exponential Decay for $n = 3$ and 5 66
5.3 Comparison of the Drop in Performance Experienced by the Decaying σ Algorithm with Linear Decay vs Exponential Decay for $n = 10$ and 20 67
5.4 Comparison of the Sample Standard Deviation of the Average Return per Episode Over 500 Episodes for Decaying σ Algorithms with Different Values of the Target Network Update Frequency Parameter 70
5.5 Comparison of the Best Performance of the $Q(\sigma)$ Network with Different Settings of σ 73

A.1 Comparison of the Performance of the $Q(\sigma)$ Network with Different Values of σ and the Target Network Update Frequency 87

List of Figures

2.1 Backup Diagrams for Bellman Equations 9
2.2 Backup Diagrams of TD(0), Sarsa, Expected Sarsa, and Q-Learning 14
2.3 Backup Diagrams of 3-Step Sarsa and Tree Backup 18
2.4 Tile Coding 22

3.1 One-Step Q(Sigma) Backup Diagram 27
3.2 Two-Step $Q(\sigma)$ Backup Diagram 28
3.3 Two-Step Q(Sigma) as a Convex Combination 29
3.4 19-State Random Walk 35
3.5 Results of the 19-State Random Walk Experiment 37
3.6 The Windy Gridworld Environment 38
3.7 Results of the Experiment on the Stochastic Windy Gridworld 39

4.1 The Mountain Cliff Environment 45
4.2 The Results of the Mountain Cliff Experiment 49

5.1 Results of the Off-Policy vs On-Policy Experiment 62
5.2 Performance of the $Q(\sigma)$ Network for Different Values of n and σ 64
5.3 Performance of Decaying σ with Linear and Exponential Decay 68
5.4 Performance of Decaying σ with Different Target Network Update Frequencies 71
5.5 Comparison of the $Q(\sigma)$ Network Algorithms with the Best Performance for Different Values of σ 74

Chapter 1

Introduction

Reinforcement learning agents, like humans, learn by trial and error how to interact with their environments. The main goal of an agent is to maximize the amount of total *reward*, a numerical signal, that it gets from the environment. Consequently, an agent has to be able to predict the amount of future reward as a function of the current information available and its own actions. Predicting the amount of future reward is often done by using *value functions*, which represent the expected value of the sum of future rewards. When value functions are not readily available, they have to be estimated by using the rewards sampled from the environment. *Temporal Difference* (TD) is a useful class of solution methods for computing accurate estimates of the value function as the agent is interacting with its environment.

TD methods are a special kind of prediction method where estimates of the value function are learned from previous estimates — a technique known as *bootstrapping*. This is reminiscent of dynamic programming where, given a perfect model of the environment, estimates are learned by bootstrapping on other estimates. TD methods are also capable of learning from raw experiences without using a model of the environment. This is analogous to *Monte Carlo* estimation where estimates can be computed by simulating trajectories of an agent in the environment. TD methods create a flexible framework that combines ideas from dynamic programming and Monte Carlo estimation in order to create powerful algorithms.

Action-value methods are a subset of TD methods concerned with esti-

mating *action-value functions*, which represent the expected reward as a function of states and actions. Several action-value methods have been proposed throughout the years. Q-learning (Watkins, 1989; Watkins & Dayan, 1992) was one of the early breakthroughs in reinforcement learning. Q-learning estimates the action-value function corresponding to the optimal policy by bootstrapping on the maximum of the next available action-value estimate. Q-learning is an example of an *off-policy* method because the decision rule or *policy* generating the behaviour — the *behaviour policy* — is different from the policy whose values are being estimated — the *target policy*. The case where the behaviour and target policies are equal is known as the *on-policy* case.

Q-learning motivated several methods for estimating action-value functions. The classical algorithm Sarsa was originally proposed as an extension to Q-learning (Rummery, 1995; Rummery & Niranjan, 1994; Sutton, 1996). Sarsa learns on-policy by bootstrapping on a sample of the next action-value estimate. Moreover, Sarsa can be extended to the off-policy case by using importance sampling (Precup et al., 2000). The algorithm Expected Sarsa was proposed as a variation of Sarsa and as a generalization of Q-learning (van Hasselt, 2011; van Seijen, van Hasselt, Whiteson, & Wiering, 2009). Instead of sampling, Expected Sarsa bootstraps by taking an expectation over all the next action-value estimates. Additionally, Expected Sarsa is capable of learning off-policy without explicitly using importance sampling. Q-learning can be seen as a special case of Expected Sarsa ; however, there is no clear unification of Sarsa and Expected Sarsa.

Action-value methods can be implemented as *one-step* methods, where updates or *backups* consist of one sample transition of the next reward, state, and action. Alternatively, action-value methods can be extended to the *multi-step* case where the number of transitions used to compute the backup, also known as the backup length, can be greater than one. Multi-step methods span a spectrum of algorithms where at one end we have one-step TD methods and at the other end we have Monte Carlo estimation methods. Thus, multi-step methods unify one-step TD methods and Monte Carlo estimation methods under the same family of algorithms.

The classical example of a multi-step method is the TD(λ) algorithm which, through the use of the trace decay parameter λ, can smoothly shift from one-step TD ($\lambda = 0$) and Monte Carlo estimation ($\lambda = 1$). TD(λ) computes updates using a geometric average of infinite backups of different lengths. In this work we will consider the *n-step* case, a simpler example of a multi-step method where updates are computed using a single backup of length n. The parameter n, just like λ, allows us to shift between one-step TD methods and Monte Carlo methods. Moreover, n has an analogous effect to that of λ: trading off bias for variance as it increases (Kearns & Singh, 2000).

Sarsa naturally generalizes to the n-step setting by sampling more transitions before computing the update. n-step Sarsa can also be extended to the off-policy case through the use of importance sampling (Precup et al., 2000; Sutton & Barto, 2018). For Expected Sarsa, the Tree Backup algorithm seems like a natural generalization since it preserves the most desirable property of the original algorithm: it can learn off-policy without directly using importance sampling (Precup et al., 2000). The fundamental difference between n-step Sarsa and Tree Backup is that Sarsa samples one action at every step of the backup, whereas Tree Backup takes an expectation over all possible actions. n-step Sarsa and Tree Backup share similar convergence guarantees in the tabular case; consequently, it is intuitive to believe that a unification of these two algorithms would inherit the same convergence properties.

The main contribution of this thesis is the introduction of a new algorithm called n-step $Q(\sigma)$ that unifies n-step Sarsa and Tree Backup. n-step $Q(\sigma)$ introduces a new parameter, $\sigma \in [0, 1]$, which allows us to smoothly vary the degree of sampling and expectation. Our algorithm spans a continuum between n-step Sarsa (full sampling, $\sigma = 1$) and n-step Tree Backup (pure expectation, $\sigma = 0$); thus, it unifies both of these algorithms under the same family. In Chapter 2 we introduce the background material necessary for the n-step $Q(\sigma)$ algorithm and in Chapter 3 we present our main algorithm and provide empirical evaluations in the tabular case. Moreover, in Chapter 3 we provide intuition about the benefits of this unification.

Our second contribution is the extension of the n-step $Q(\sigma)$ algorithm to

the linear function approximation case presented in Chapter 4. This extension increases the applicability of n-step $Q(\sigma)$ to more complex domains where tabular representations of the action-value function are infeasible. Additionally, we present empirical evaluations of n-step $Q(\sigma)$ with linear function approximation that show that the effects found in the tabular case carry over to some extent to the linear function approximation case.

Our last contribution is an extension of n-step $Q(\sigma)$ to the non-linear function approximation case. We demonstrate in Chapter 5 how to combine n-step $Q(\sigma)$ with the DQN architecture (Mnih et al., 2015) so that it is capable of using a neural network to approximate the action-value function. We named the resulting architecture the $Q(\sigma)$ network. This extension further increases the applicability of n-step $Q(\sigma)$ because it provides compelling evidence that it can be used in high dimensional environments. We provide empirical evaluations of the $Q(\sigma)$ network and show that many of the effects found in the tabular and linear function approximation cases also carry over to the non-linear function approximation case.

Finally, in Chapter 6, we conclude this work, summarizing the contributions, discussing extensions to the n-step $Q(\sigma)$ algorithm that have been implemented since its introduction, and considering avenues for future work.

Chapter 2

Background on Reinforcement Learning

In this chapter we introduce the notation and background material necessary for the theory of the n-step $Q(\sigma)$ algorithm.

Throughout this work we denote scalar values by lower-case letters (e.g., x), random variables by upper-case letters (e.g., X), and sets by upper-case calligraphic letters (e.g., $\mathcal{X}$). Vectors and matrices are denoted by bold symbols or letters — lower-case for vectors (e.g., $\boldsymbol{x}$) and upper-case for matrices (e.g., $\boldsymbol{X}$). We reserve the use of $\mathbb{E}\{\cdot\}$ for the expected value of a random variable enclosed in between the brackets.

2.1 Reinforcement Learning and Finite Markov Decision Processes

Reinforcement Learning (RL) is learning how to map situations to actions in order to maximize a numerical signal, also known as *reward*. Just like rats learn to navigate mazes to gain access to food in psychological experiments, learners — or *agents* — in the RL framework learn how to act in order to gain access to reward. Similar to the rat in the maze, RL agents are not given exact instructions about how to act. Instead, agents are given a set of actions that they can use in order to interact with the world. Through the process of trial and error, agents learn to map aspects of the world to particular actions in order to maximize the amount of reward they receive.

In this work, we model the sequential decision making problem as a finite *Markov Decision Process* (MDP). In the finite MDP framework there exist two main entities: the agent and the environment. The agent is both a learner and a decision maker. The *environment* is everything that the agent interacts with and is external to the agent. The agent and the environment interact over a sequence of discrete time steps $t \geq 0$. At every time step, the agent receives information about the environment's *state*, $S_t \in \mathcal{S}$, where $\mathcal{S}$ is a finite set of all possible states. On the basis of that information, the agent selects an *action*, A_t, from a finite set of actions, $\mathcal{A}$. After selecting and executing an action, the environment transitions to a new state S_{t+1} and the agent receives a real-valued random reward $R(S_t, A_t) = R_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is a subset of $\mathbb{R}$.

S_{t+1} and R_{t+1} are modeled jointly according to the MDP dynamics probability function

$$P(s', r|s, a) \doteq \mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a), \tag{2.1}$$

where $\mathbb{P}$ is the probability operator. For ease of exposition, we have assumed that the reward is a discrete random variable; nevertheless, the reward can be modeled as a continuous random variable. We define the expected reward with respect to the transition probability function as

$$r(s, a) \doteq \mathbb{E}\{R_{t+1}|S_t = s, A_t = a\}. \tag{2.2}$$

Similarly, we define the *state-transition probability function* as

$$P^a_{ss'} \doteq \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a), \tag{2.3}$$

where $\mathbb{P}(S_{t+1}|S_t, A_t)$ is the marginal probability function of the next state given the current state-action pair.

Note that the dynamics and the state-transition probability functions are independent of the current time step t. The distribution of the values of S_{t+1} and A_{t+1} are only dependent on the preceding state and action, S_t and A_t. This is known as the *Markov property* and is a property already present in our model.

The interaction between the agent and the environment gives rise to a sequence or a *trajectory* of states, actions and rewards of the form

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3.... \tag{2.4}$$

Terminal states are states that transition to themselves and result in zero reward with probability one. Tasks with environments that contain one or more terminal states and where the agent is guaranteed to reach such state with probability one are known as *episodic tasks*. In episodic tasks we let T be a random variable corresponding to the time when the agent reaches the terminal state and the sequence in Equation (2.4) can be considered a finite sequence terminating at state S_T. Alternatively, tasks where an environment does not have a terminal state or where agents cannot reach a terminal state are known as *continuing tasks*.

2.2 Returns, Policies, and Value Functions

The goal of reinforcement learning agents is to maximize the expected sum of rewards given the MDP dynamics. The sum of rewards is also known as the *return*, which is denoted G_t and is defined as

$$G_t = R_{t+1} + R_{t+2} + ... = \sum_{k=0}^{\infty} R_{t+1+k}. \tag{2.5}$$

Since the sum of the rewards can be unbounded when there is no terminal state, we consider instead the *discounted return*

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}, \tag{2.6}$$

where γ is a discount factor in the interval $[0, 1]$ and it can be equal to 1 only when the MDP has a terminal state and the agent is guaranteed to reach such state with probability one. Henceforth, we will assume that the return is always discounted and we will call it return instead of discounted return.

In order to make informed decisions, agents often use *value functions* that represent the expected return given the current information about the environment. The expected return is determined by the agent's actions; consequently,

value functions are determined by the particular decision rule that the agent uses to choose actions. This decision rule is known as a *policy*.

A stationary policy is a mapping from states to a probability distribution over the actions. We often denote policies as π or μ and let $\pi(a|s)$ or $\mu(a|s)$ be the probability of taking action a while in state s. Policies can also depend on the time t; in such case, we denote the policy that the agent is applying at a given time step as π_t or μ_t.

Given a policy, π, the *state-value* function induced by this policy is defined as

$$v_\pi(s) \doteq \mathbb{E}_{\pi,P}\{G_t|S_t = s\} = \mathbb{E}_{\pi,P}\Big\{\sum_{k=0}^{T} \gamma^k R_{t+1+k}|S_t = s\Big\} \;\forall\; s \in \mathcal{S}, \tag{2.7}$$

where the expectation is with respect to the policy π and the MDP dynamics probability function P. Note that the expectation in Equation (2.7) does not depend on the time step t, but we include it because it corresponds to the current time step of the agent.

In a similar fashion, we can define the value function with respect to state-action pairs — the *action-value function* — as

$$q_\pi(s,a) \doteq \mathbb{E}_{\pi,P}\{G_t|S_t = s, A_t = a\} \;\forall\; (s,a) \in \mathcal{S} \times \mathcal{A}. \tag{2.8}$$

It is important to note that there exists a bidirectional relationship between the state-value and the action-value function:

$$\begin{aligned} v_\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a), \\ q_\pi(s,a) &= r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} v_\pi(s'). \end{aligned}$$

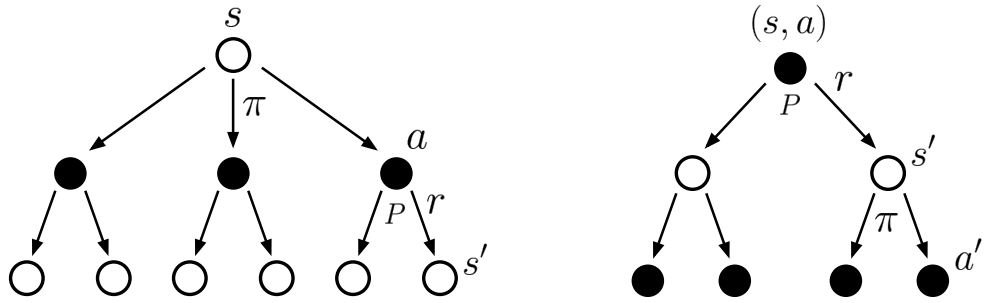
2.2.1 Bellman Equations and Backup Diagrams

The state-value functions can also be defined recursively:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \pi(a|s) P^a_{ss'}[r(s,a) + \gamma v_\pi(s')]. \tag{2.9}$$

The same can be done for action-value functions:

$$q_\pi(s,a) = \sum_{s'\in\mathcal{S}} \sum_{a'\in\mathcal{A}} P^a_{ss'} \pi(a'|s')[r(s,a) + \gamma q_\pi(s',a')]. \tag{2.10}$$



(a) State-Value (b) Action-Value

Figure 2.1: Backup Diagrams for Bellman Equations. (a) Backup diagram for the Bellman equation corresponding to the state-value function. (b) Backup diagram for the Bellman equation corresponding to the action-value function.

Equations (2.9) and (2.10) are called *Bellman equations*. They demonstrate the relationship between the value of a state, or a state-action pair, and the value of its successor states, or successor state-action pairs.

Bellman equations can be represented as *backup diagrams* such as the ones in Figure 2.1. Backup diagrams are useful to provide high level intuition about how information flows from successor states to the current state. This will be particularly useful in the next section when we introduce methods for estimating the state-value and action-value functions.

Figure 2.1a shows the backup diagram of the Bellman equation in Equation (2.9). Empty circles represent states, whereas solid ones represent actions. Branches leading from states to actions are weighted according to the policy, π. Branches connecting actions to states are weighted according to the MDP dynamics probability function. Figure 2.1b shows the backup diagram of the action-value function. In this case, branches and circles have the same meaning, but the starting node of the diagram represents the initial state-action pair.

2.2.2 Optimal and Exploratory Policies

Maximizing the expected return is equivalent to finding a policy π_*, such that the expected return that it generates is greater than or equal to the expected

return of any other policy. We call π_* an *optimal policy*.

Finite MDP's can have several optimal policies. However, all optimal policies share the same value functions. The state-value function induced by an optimal policy π_*, denoted v_*, is called the *optimal state-value function* and is defined as

$$v_*(s) \doteq \max_\pi v_\pi(s). \tag{2.11}$$

The action-value function corresponding to an optimal policy is defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a). \tag{2.12}$$

and is known as the *optimal action-value* function.

If q_* is available, the agent can derive π_* by selecting the action that maximizes q_* at every time step. This type of policies are also known as *greedy policies* with respect to q_*. However, q_* is not generally available; consequently, the agent needs to estimate q_* as it is interacting with the environment. Moreover, the agent needs to collect enough information from every state-action pair to compute accurate estimates of q_*. Consequently, instead of using greedy policies, agents are trained using *exploratory policies* that guarantee that every state-action pair is going to be observed enough times to obtain accurate estimates of q_*.

One example of exploratory policies are *ϵ-greedy policies*. This type of policies choose the greedy action with respect to the current estimate of q_* with probability $1-\epsilon$ and a random action with probability of ϵ. ϵ is an exploration parameter in the interval $[0,1]$ that controls the degree of greediness of the policy.

2.3 Temporal-Difference Methods

One of the main challenges in RL is estimating value functions as the agent is interacting with the environment. *Temporal-difference* (TD) methods are a type of prediction method used to compute new estimates of the value function by using old estimates, a technique known as *bootstrapping*. TD methods

compute value functions using *updates* or *backups* of the form

$$\begin{aligned}\text{New Estimate} &= \text{Old Estimate} + \alpha[\text{Target} - \text{Old Estimate}] \\ &= (1-\alpha)\ \text{Old Estimate} + \alpha\ \text{Target},\end{aligned} \tag{2.13}$$

where α is a learning rate parameter in the half-open interval $(0, 1]$.

In this case, we consider estimating the value function of a stationary policy π — also known as the *prediction setting*. In essence, the update rule above is a stochastic approximation algorithm. As iterations continue and under the appropriate conditions, the expected distance between the target and the old estimate shrinks until converging to a fixed point. The target would ideally be the expected return with respect to π and P given the current state or state-action pair, depending on the type of value function.

Generally, the expected return is not available; instead, we could estimate it by sampling sequences of rewards until an episode terminates. This is analogous to Monte Carlo estimation methods where trajectories are simulated and an estimate is computed by averaging over a large number of trajectories. However, Monte Carlo estimation is infeasible in continuing tasks and, even in episodic tasks, it can be highly inefficient.

TD methods take an alternative approach. Instead of sampling a trajectory of rewards, one can sample one reward and the next value or action-value function. For example, imagine the case where we are interested in estimating $v_\pi(S_t)$. In such case, we could use $R_{t+1} + \gamma v_\pi(S_{t+1})$ to estimate $v_\pi(S_t)$ because they are equal in expectation, as shown by the Bellman equation in Section 2.2.1. However, since we do not have access to $v_\pi(S_t)$, we use our current estimate of the value function $V_t(S_t)$. Thus, when computing the estimate of the state-value function we could use the update rule,

$$V_{t+1}(S_t) = (1-\alpha)V_t(S_t) + \alpha[R_{t+1} + \gamma V_t(S_{t+1})], \tag{2.14}$$

where V_t corresponds to the estimate of v_π at time t, for all the other states $s \neq S_t$ we consider $V_{t+1}(s) = V_t(s)$, and γ satisfies the conditions from Section 2.2. This update rule is a stochastic approximation algorithm that uses the fact that v_π has to satisfy the Bellman equation. If the algorithm converges to a

unique fixed point, such fixed point would also satisfy the Bellman equation and the estimates would have converged to v_π.

The update function in Equation (2.14) corresponds to the algorithm TD(0) (Sutton, 1988) and is used to compute estimates of the state-value function as the agent is interacting with its environment. It has been shown that the estimates computed by the TD(0) algorithm with state and time dependent learning rates (i.e., $\alpha_t(S_t)$) converge to v_π under certain conditions (Bertsekas and Tsitsiklis, 1996).

2.3.1 Action-Value Methods

We can also use the update rule in Equation (2.13) and a truncated estimate of the expected return to compute estimates for the action-value function. In this case, we are going to consider policies that over time converge to the optimal policy π_* — also known as the *control setting*.

For action-value functions, we use $R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})$ as an estimate of the expected return because in expectation it is equal to $q_\pi(S_t, A_t)$, as shown by the Bellman equation for action-value functions. Since we do not have access to q_π, we use the current estimate of the action-value function denoted Q_t. Thus, we obtain the following update rule

$$\begin{aligned} Q_{t+1}(S_t, A_t) &= Q_t(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t)] \\ &= (1 - \alpha)Q_t(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1})] \end{aligned} \tag{2.15}$$

where Q_t is the estimate of the action-value function at time t and we let $Q_{t+1}(s, a) = Q_t(s, a)$ for all the state-action pairs not equal to (S_t, A_t). Once again, the update rule is a stochastic approximation algorithm. As iterations continue and under the appropriate conditions, the expected distance between the target and the old estimate shrinks and the algorithm converges to a fixed point that satisfies the Bellman equation for action-value functions. If this fixed point is unique, then the estimates of the action-value function would have converged to q_π.

The update function in Equation 2.15 corresponds to the classical algorithm Sarsa(0) (Rummery, 1995; Rummery & Niranjan, 1994; Sutton, 1996). Sarsa has been shown to converge under certain conditions to the optimal action-value function q_* for state, action, and time dependent learning rate (i.e., $\alpha_t(S_t, A_t)$) and when the policy π_t becomes greedy in the limit and has infinite exploration (Singh, Jaakkola, Littman, & Szepesvári, 2000).

The update functions of TD methods are characterized by their estimate of the expected return — the quantity in brackets in the second line of Equation (2.15). We will denote this quantity as:

$$\hat{G}_t \doteq R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}). \tag{2.16}$$

Sine $\hat{G}_t$ is the target of the update function, we can obtain different TD algorithms by changing the definition of $\hat{G}_t$. For example, if we let

$$\hat{G}_t^{ES} = R_{t+1} + \gamma \sum_{a \in \mathcal{A}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a), \tag{2.17}$$

and use this quantity as a target, we obtain the update rule of the algorithm Expected Sarsa (Precup et al., 2000; van Seijen et al., 2009). The fixed point of the resulting stochastic approximation algorithm also satisfies a Bellman equation that is equivalent to the Bellman equation corresponding to the Sarsa(0) algorithm. The advantages of using Expected Sarsa is that it has smaller variance than Sarsa while retaining the same bias and convergence guarantees under the same conditions (van Seijen et al., 2009).

2.3.2 Off-Policy Methods and Backup Diagrams

So far we have considered the case where the policy whose value function is being estimated, known as the *target policy* and denoted π, is the same as the policy generating the behaviour, known as the *behaviour policy* and denoted μ. This case is also known as the *on-policy* case. In this section we will consider a more general case where the target and behaviour policies can be different, known as the *off-policy* case.

The main issue when working in the off-policy case is that actions are sampled according to a different policy from the target policy. Since the Bellman

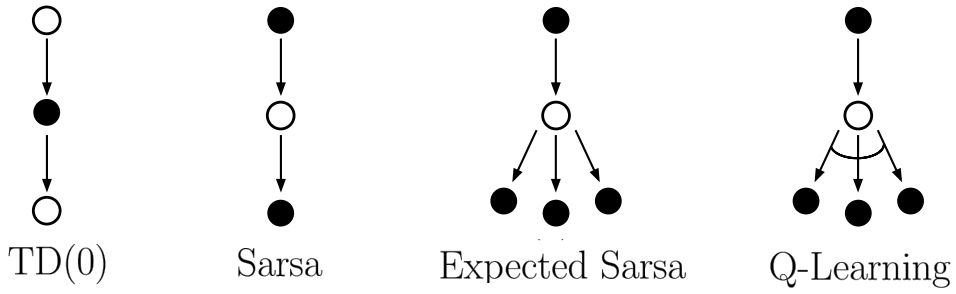


Figure 2.2: Backup diagrams of TD(0), Sarsa, Expected Sarsa, and Q-Learning.

equation assumes that actions are sampled according to the target policy, then the fixed point solution of our algorithms might not satisfy the Bellman equation. In other words, the estimates might converge to a different function other than q_π. Consequently, when defining the estimate of the expected return $\hat{G}_t$ as in the previous section, we need to correct for the difference in the policies so that our estimates converge to the correct action-value function.

Some algorithms are already well suited to handle this kind of scenario. For example, the popular off-policy algorithm Q-learning is able to learn optimal action-value functions by bootstrapping on the maximum over the actions of the next estimate of the action-value function (Watkins, 1989; Watkins & Dayan, 1992). This results in the following update:

$$Q_{t+1}(S_t, A_t) = (1 - \alpha)Q_t(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q_t(S_{t+1}, a')]. \tag{2.18}$$

The update of Q-learning is based on the Bellman optimality equation:

$$q_*(s, a) = \sum_{s' \in \mathcal{S}} P^a_{ss'}[r(s, a) + \max_{a'} q_*(s, a')]. \tag{2.19}$$

Because the Bellman optimality equation is agnostic of the behaviour policy μ, if the Q-learning algorithm converges to a unique fixed point, then it will have converged to the correct action-value function q_* regardless of μ. In fact, the Q-learning algorithm has been proven to converge in multiple settings in RL (Jaakkola, Jordan, & Singh, 1994; Tsitsiklis, 1994; Watkins & Dayan, 1992).

Similar to Q-learning, Expected Sarsa is already capable of learning off-policy because its corresponding Bellman equation is agnostic to μ (Precup

et al., 2000; van Seijen et al., 2009). In contrast, the Sarsa algorithm needs to be extended to use importance sampling in order to be able to converge to the correct action-value function (Precup et al., 2000). In this case, we consider the corrected Bellman equation:

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} P^a_{ss'} \mu(a'|s') \Big[r(s,a) + \gamma \frac{\pi(a'|s')}{\mu(a'|s')} q_\pi(s',a') \Big]. \tag{2.20}$$

The corresponding estimate of the expected return of the stochastic approximation algorithm is:

$$\hat{G}_t = R_{t+1} + \gamma \rho_{t+1} Q_t(S_{t+1}, A_{t+1}), \tag{2.21}$$

$$\rho_{t+1} = \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})}, \tag{2.22}$$

where ρ_{t+1} is known as the *importance sampling ratio*. If such algorithm converges to a unique fixed point, then the fixed point must satisfy the corrected Bellman equation and the estimates of the action-value function would have converged to q_π.

All the TD methods introduced so far can also be represented as backup diagrams. In this case, the backup diagram represents how the information flows from future states and actions to the current estimate of the value function when computing an update. Figure 2.2 shows the backup diagrams of TD(0), Sarsa, Expected Sarsa, and Q-learning, respectively from left to right. Just as before, we let empty circles represent states and solid circles represent actions. For state-value functions, the node at the top of the backup diagram corresponds to the initial state whose value function is being updated. In the case of action-value methods, the starting node corresponds to the initial state-action pair whose value function is being updated. In the backup diagram of Expected Sarsa, the branches corresponding to the actions are weighted by the target policy π. In the case of Q-learning, the arc connecting the branches corresponding to the actions represents the maximum over all the actions.

2.4 n-Step Temporal Difference Methods

In the previous section we introduced the idea of bootstrapping in order to truncate the estimate of the expected return after one time step. However, we

can generalize this idea to longer time intervals where instead of bootstrapping after one time step, we bootstrap at the end of a sequence of rewards of length $n \geq 1$; these type of algorithms are also known as *n-step* methods. n-step methods span a spectrum of algorithms with Monte Carlo estimation on one end and one-step TD methods at the other. In other words, n-step methods unify one-step TD methods and Monte Carlo estimation under the same family of algorithms. The parameter n, also known as the *backup length*, serves as a trade-off between bias and variance. The larger the value of n, the smaller the bias of the estimate of the expected return, but the higher the variance (Kearns & Singh, 2000).

All the algorithms introduced in the previous section can be extended to the n-step case. In this case, the estimates of the expected return are based on unrolled versions of the Bellman equation where q_π is expanded n times. For example, the n-step Sarsa algorithm is based on the n-step Bellman equation:

$$q_\pi(s,a) = \mathbb{E}_{\pi,P}\big\{R_{t+1} + \gamma R_{t+2} + ... + \gamma^n q_\pi(S_{t+n}, A_{t+n}) \big| S_t = s, A_t = a\big\}. \quad (2.23)$$

The corresponding estimate of the expected return is:

$$\hat{G}_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \quad (2.24)$$

where the index $t : t+n$ indicates that the return is unrolled from time t to time $t+n$. The update rule of the stochastic approximation algorithm is:

$$Q_{t+n}(S_t, A_t) = (1-\alpha) Q_{t+n-1}(S_t, A_t) + \alpha \hat{G}_{t:t+n}. \quad (2.25)$$

Once again, if the update rule above converges to a unique fixed point, then the estimates computed by the n-step Sarsa algorithm would have converged to q_π.

For ease of exposition, we will write the estimate of the expected return recursively. In the case of n-step Sarsa, the recursive definition of the return is:

$$\hat{G}_{t:t+n} \doteq R_{t+1} + \gamma \hat{G}_{t+1:t+n} \quad (2.26)$$

$$\hat{G}_{t+n:t+n} \doteq Q_{t+n-1}(S_{t+n}, A_{t+n}). \quad (2.27)$$

Moreover, we can also extend n-step Sarsa to the off-policy case bia importance sampling. The resulting estimate of the expected return is:

$$\hat{G}_{t:t+n} \doteq R_{t+1} + \gamma\rho_{t+1}\hat{G}_{t+1:t+n}, \tag{2.28}$$

where the base case of the recursion is the same as before. This algorithm is also known in the literature as per-decision importance sampling (Precup et al., 2000).

A natural generalization of Expected Sarsa to the n-step case is the Tree Backup algorithm because, like Expected Sarsa, it is capable of computing action-value estimates off-policy without directly using importance sampling (Precup et al., 2000). The n-step estimate of the expected return of the Tree Backup algorithm is

$$\hat{G}^{TB}_{t:t+n} = R_{t+1} + \gamma\pi(A_{t+1}|S_{t+1})\hat{G}^{TB}_{t+1:t+n} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})Q_{t+1}(S_{t+1}, a) \tag{2.29}$$

where $\hat{G}^{TB}_{t+n:t+n}$ is defined as $Q_{t+n-1}(S_{t+n}, A_{t+n})$. The update rule is obtained by replacing $\hat{G}_{t+1:t+n}$ in Equation (2.25) with $\hat{G}^{TB}_{t:t+n}$. If it exists, the unique fixed point of the resulting stochastic approximation algorithm satisfies an n-step Bellman equation equivalent to Equation (2.23). Under certain conditions, the estimates of the action-value function computed by the n-step off-policy Sarsa and Tree Backup algorithms converge to q_π in the prediction setting (Precup et al., 2000).

Just like one-step methods, n-step algorithms can be represented graphically with backup diagrams. Figure 2.3 shows the backup diagrams of 3-step Sarsa and Tree Backup. The backup diagrams clearly illustrate the main difference between n-step Sarsa and Tree Backup: the former samples an action at every step of the backup, whereas the latter takes an expectation over all the possible actions.

2.4.1 Multi-Step Methods

n-step methods are a special case of multi-step methods. *Multi-step* methods are a more general case where updates can be computed using one n-step

3-Step Sarsa 3-Step Tree Backup

Figure 2.3: Backup diagrams of 3-step Sarsa and Tree Backup.

estimate of the expected return or a convex combination of several n-step estimates of different lengths. A classical example of the latter case is the TD(λ) algorithm which estimates state-value functions by taking a geometric average over several n-step estimates of the expected return each weighted by $(1-\lambda)\lambda^n$ (Sutton, 1988). The resulting estimate of the expected return is

$$\hat{G}_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\hat{G}_{t:t+n} \tag{2.30}$$

$$\hat{G}_{t:t+n} = R_{t+1} + \gamma R_{t+2} + .. + \gamma^n V_{t+n-1}(S_{t+n}, A_{t+n}), \tag{2.31}$$

where $\lambda \in [0,1]$. Just like n in the n-step case, the *trace decay* parameter λ allows to shift between one-step TD methods and Monte Carlo estimation. Thus, it is another way of unifying one-step TD and Monte Carlo estimation.

Even though the estimate of the expected return involves infinitely many terms, it is possible to compute updates at every time step that correspond exactly to Equation (2.30) (van Seijen, Mahmood, Pilarski, Machado, & Sutton, 2015). In order to compute updates at every time step, algorithms use an *eligibility trace* vector — a form of short term memory of the most recent state visitations. We will refer to methods that use the trace decay λ as *eligibility trace* methods.

All the action-value methods presented in the previous section can be extended to the eligibility trace case (Precup et al., 2000; Rummery, 1995).

However, we omit these cases because they are beyond the scope of this thesis.

2.5 Approximate Solution Methods

So far, we have considered methods that rely on having one estimate of the action-value function for each state-action pair in $\mathcal{S} \times \mathcal{A}$. These methods are also known as *tabular solution methods*. However, this approach is infeasible when either the state or action spaces are too large or infinite in size. In such case, instead of estimating q_π for each state-action pair, we can approximate it. These methods are called *approximate solution methods* because the resulting estimate of q_π converges to a region close to the true estimate; in other words, it is an approximation to the exact action-value function.

In order to approximate q_π, we use a function $\hat{q}(s, a, \boldsymbol{\theta})$ which is parameterized by the parameter vector $\boldsymbol{\theta} = \big(\theta_1, \theta_2, ..., \theta_d\big)^T \in \mathbb{R}^d$, where $d \ll |\mathcal{S} \times \mathcal{A}|$, θ_i is the i-th element of $\boldsymbol{\theta}$, and T denotes the transpose of the vector. Depending on how $\hat{q}$ is represented in terms of $\boldsymbol{\theta}$, we will distinguish between two special cases of the approximate solution methods: *linear function approximation* methods and *non-linear function approximation* methods. For the non-linear function approximation case, we will only consider the case where the function is produced by a neural network.

2.5.1 The Value Error Objective and Semi-Gradient Methods

In the function approximation case we consider minimizing the *mean squared value error* defined as

$$\overline{\text{VE}}(\boldsymbol{\theta}) \doteq \sum_{s \in \mathcal{S}} \eta(s) \big[v_\pi(s) - \hat{v}(s, \boldsymbol{\theta})\big]^2, \tag{2.32}$$

where $\hat{v}(s, \boldsymbol{\theta})$ is an approximation of the state-value function parameterized by $\boldsymbol{\theta}$ and η the probability distribution of the states. Similarly, for the action-value function we can define the *mean squared action-value error* as

$$\overline{\text{AVE}}(\boldsymbol{\theta}) \doteq \sum_{s \in \mathcal{S}} \eta(s) \sum_{a \in \mathcal{A}} \pi(a|s) \big[q_\pi(s, a) - \hat{q}(s, a, \boldsymbol{\theta})\big]^2. \tag{2.33}$$

The mean squared value and action-value errors measure the difference between the state-value or action-value function and its approximation $\hat{v}$ or $\hat{q}$, respectively. In both cases, η can be thought of as a weighting that represents how much we care about the approximation error for each particular state or state-action pair.

Minimizing the action-value error is equivalent to finding $\boldsymbol{\theta}^*$ such that $\overline{\text{AVE}}(\boldsymbol{\theta}^*) \leq \overline{\text{AVE}}(\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in \mathbb{R}^d$. In some cases, this condition is relaxed to finding a *local optimum* such that $\boldsymbol{\theta}^*$ minimizes the error for all $\boldsymbol{\theta}$ in a neighborhood around $\boldsymbol{\theta}^*$.

When $\hat{q}$ is a differentiable function, we can find $\boldsymbol{\theta}^*$ iteratively by using *stochastic gradient descent* (SGD). This results in the update function

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{1}{2}\alpha\nabla\big[q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\big]^2 \\ &= \boldsymbol{\theta}_t + \alpha\big[q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\big]\nabla\hat{q}(S_t, A_t, \boldsymbol{\theta}_t),\end{aligned} \tag{2.34}$$

where $\boldsymbol{\theta}_t$ is the parameter vector at time t, $\boldsymbol{\theta}_0$ is initialized randomly, and $\nabla\hat{q}$ is the gradient of $\hat{q}$ with respect to $\boldsymbol{\theta}_t$:

$$\nabla\hat{q}(s, a, \boldsymbol{\theta}) \doteq \Big(\frac{\partial\hat{q}(s, a, \boldsymbol{\theta})}{\partial\theta_1}, \frac{\partial\hat{q}(s, a, \boldsymbol{\theta})}{\partial\theta_2}, ..., \frac{\partial\hat{q}(s, a, \boldsymbol{\theta})}{\partial\theta_d}, \Big)^T \tag{2.35}$$

Note that the update function of the SGD algorithm requires the true value of q_π. However, this is not generally available in the reinforcement learning setting. Consequently, just as in the previous section, we use an estimate of this value: $\hat{G}_{t:t+n}$. Hence, we can extend the algorithms presented in the previous sections to the function approximation case by replacing q_π with the corresponding estimate of the expected return, $\hat{G}_{t:t+n}$.

It is important to emphasize that $\hat{G}$ bootstraps on other values of $\hat{q}$ which depend on $\boldsymbol{\theta}_t$; consequently, using $\hat{G}$ in the update does not result in a true SGD update. This is because the update is taking into account the effect that changing the weight vector has on the estimate $\hat{q}$, but ignores the effect it has on the target $\hat{G}_{t:t+n}$. Consequently, this type of methods have been named *semi-gradient methods*. The corresponding semi-gradient update function that

we will be considering is

$$\boldsymbol{\theta}_{t+n} = \boldsymbol{\theta}_{t+n-1} + \alpha\big[\hat{G}_{t:t+n}(\boldsymbol{\theta}_{t+n-1}) - \hat{q}(S_t, A_t, \boldsymbol{\theta}_{t+n-1})\big]\nabla\hat{q}(S_t, A_t, \boldsymbol{\theta}_{t+n-1}), \tag{2.36}$$

where $\hat{G}_{t:t+n}\boldsymbol{\theta}_{t+n-1}$ is the n-step estimate of the expected return of the corresponding algorithm evaluated using the parameter vector at time $t+n-1$.

2.5.2 Linear Function Approximation

In the linear function approximation case $\hat{q}$ is represented as a linear combination of the elements in $\boldsymbol{\theta}_t$. Moreover, instead of working directly with the state-action pairs, we often employ a transformation

$$\mathbf{x}(s,a) \doteq \big(x_1(s,a), x_2(s,a), ..., x_d(s,a)\big)^T \in \mathbb{R}^d. \tag{2.37}$$

$\mathbf{x}(s,a)$ is also called a *feature vector*. Given $\mathbf{x}(s,a)$, we can define $\hat{q}(s,a,\boldsymbol{\theta})$ as a dot product between $\boldsymbol{\theta}$ and $\mathbf{x}(s,a)$:

$$\hat{q}(s,a,\boldsymbol{\theta}) \doteq \boldsymbol{\theta}^T\mathbf{x}(s,a) \tag{2.38}$$

The gradient of $\hat{q}$ with respect to $\boldsymbol{\theta}$ can be easily computed:

$$\nabla\hat{q}(s,a,\boldsymbol{\theta}_t) = \big(x_1(s,a), x_2(s,a), ..., x_d(s,a)\big)^T = \mathbf{x}(s,a). \tag{2.39}$$

Consequently, the semi-gradient update rule in Equation 2.36 reduces to

$$\boldsymbol{\theta}_{t+n} = \boldsymbol{\theta}_{t+n-1} + \alpha\big[\hat{G}_{t:t+n}(\boldsymbol{\theta}_{t+n-1}) - \hat{q}(S_t, A_t, \boldsymbol{\theta}_{t+n-1})\big]\mathbf{x}(S_t, A_t). \tag{2.40}$$

How we design the feature vector $\mathbf{x}$ can have a significant impact in the performance of the algorithm. We will consider a popular feature extractor used in RL called *coarse coding*. The main idea of coarse coding is to partition a continuous state space into several *receptive fields* each corresponding to a binary feature. When an observation is within the area covered by a receptive field, its corresponding binary feature activates — it takes a value of one. The resulting representation is a coarser version of the state space. The resolution of our representation can be improved by increasing the number of receptive fields and by allowing receptive fields to overlap.

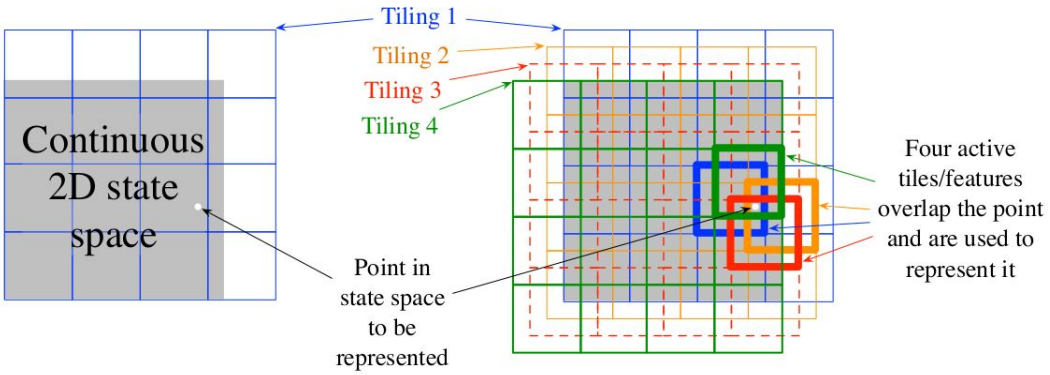


Figure 2.4: Left side: a 2-dimensional state space partitioned into a grid with tiles of equal size. Right side: the same 2-dimensional space partitioned into multiple tilings offset from one another by a uniform amount. Reprinted from *Reinforcement Learning: An Introduction* (p. 217) by Sutton and Barto (2018). Reprinted with permission.

Coarse coding allows learning to generalize across states that share the same features. The size and the shape of the receptive fields have an impact in the way information generalizes across states. In this thesis, we consider a special form of coarse coding known as *tile coding* (Albus, 1971; Albus, 1981; Sutton & Barto, 2018). In tile coding, the receptive fields are grouped into partitions called *tilings*, and each element of the partition is called a *tile*. For example, in a 2-dimensional state space, tilings represent a grid overlaping the state space and each tile in a tiling correspond to each square in the grid such as in the left side of Figure 2.4. To improve the resolution of our representation we can add more tilings, each new tiling offset by some small amount such as in the right side of Figure 2.4. Moreover, for action-value estimates we keep a separate set of tilings for each different action. The number of active features represents a vector of indices each corresponding to an element of the parameter vector $\boldsymbol{\theta}_t$. Hence, computing the inner product in Equation (2.38) corresponds to adding up the elements of $\boldsymbol{\theta}_t$ corresponding to the active features.

2.5.3 Non-Linear Function Approximation: Neural Networks

Artificial Neural Networks (ANN) are one of the most popular methods used for non-linear function approximation. An ANN is a network of several layers of interconnected units, known as *neurons*. Each layer in the network processes the output from the previous layer. The first layer, called the *input layer*, represents the input observation, $\mathbf{x}$. The input layer is followed by one or more *hidden layers*; each hidden layer processes the output information from the previous layer. The last layer, known as the *output layer*, processes the output of the last hidden layer and returns the approximate value of the function at the point $\mathbf{x}$.

The information in the network is processed at every hidden and output layer by multiplying the input of the layer by the weights corresponding to each neuron and then adding a bias term. After adding the bias term, a non-linear function g, also known as *activation function*, is applied individually to each element of the layer and then its output is propagated forward to the next layer. Among the most popular type of activation functions are the ReLu gate, $g(x) = \max(0, x)$, and the sigmoid function, $g(x) = \frac{1}{1+e^{-x}}$. Since each neuron and bias term is represented as a vector of parameters, each layer can be represented as a weight matrix $\mathbf{\Theta}$ with each column corresponding to each neuron, and a bias term $\mathbf{b}$. For example, an ANN with one input layer; one hidden layer with activation function g_1, weight matrix $\mathbf{\Theta}_1$, and bias term $\mathbf{b}_1$; and one output layer with activation function g_2, weight matrix $\mathbf{\Theta}_2$ and bias term $\mathbf{b}_2$, the function f is approximated as

$$\hat{f}(\boldsymbol{x}) = g_2(\mathbf{\Theta}_2^T g_1(\mathbf{\Theta}_1^T \boldsymbol{x} + \mathbf{b}_1) + \mathbf{b}_2). \tag{2.41}$$

Just as in the linear case, all the parameters from the network can be learned using SGD or semi-gradient methods to minimize the mean squared action-value error. The corresponding update function is the same as in Equation (2.36), but in this case the gradient is with respect to each $\mathbf{\Theta}_i$ and $\mathbf{b}_i$ in the network. Moreover, the gradient can be computed using a *mini-batch* of observations instead of a single observation, which provides a more accurate

estimate of the gradient (Goodfellow, Bengio, & Courville, 2016).

Many methods have been proposed for using ANN's for estimating value functions in RL. In this body of work we will focus on the architecture developed by Mnih et al. (2015) called the Deep Q-Network (DQN). DQN was originally design to use the Q-learning target; in chapter 5, we will introduce modifications to DQN in order to adapt it to other action-value methods.

Chapter 3

The n-Step Q(σ) Algorithm

In this chapter, we introduce the n-step $Q(\sigma)$ algorithm — the main algorithm to be studied in this thesis — in the tabular case. The n-step $Q(\sigma)$ algorithm will be the basis for the extensions to the linear function approximation case in Chapter 4 and the non-linear function approximation case in Chapter 5, and for the empirical evaluations in this and the following two chapters.

We will start by introducing the one-step version of $Q(\sigma)$ and its extension to the off-policy case. We then continue to extend one-step $Q(\sigma)$ to the n-step case and show how it can represent the n-step algorithms Sarsa and tree backup — unifying them under the same family of algorithms. We then develop intuition about the benefits of using n-step $Q(\sigma)$ over n-step Sarsa and Tree Backup. We conclude this chapter by providing empirical evaluations of n-step $Q(\sigma)$ in two tabular domains.

3.1 One-Step $Q(\sigma)$

In essence, the one-step $Q(\sigma)$ algorithm is a convex combination of Sarsa and Expected Sarsa. The parameter σ lets us decide whether to use Sarsa and sample the next action-value estimate, or use Expected Sarsa and take an expectation over all the possible next action-value estimates. If we choose σ equal to one, we select Sarsa, whereas with σ equal to zero, we select Expected Sarsa. Nevertheless, it is possible to select values of σ anywhere in the interval $[0, 1]$. In this case, we give Sarsa a weight of σ and Expected Sarsa a weight

of $(1-\sigma)$. The resulting estimate of the expected return is defined as

$$\begin{aligned}\hat{G}_t^{\sigma} &\doteq \sigma\big[R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1})\big] + (1-\sigma)\big[R_{t+1} + \gamma \sum_{a\in\mathcal{A}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a)\big] \\ &= R_{t+1} + \gamma\big[\sigma Q_t(S_{t+1}, A_{t+1}) + (1-\sigma)\sum_{a\in\mathcal{A}} Q_t(S_{t+1}, a)\big]. \end{aligned} \tag{3.1}$$

We can then use the general update rule from the previous chapter to iteratively compute the estimates of the action-value function:

$$Q_{t+1}(S_t, A_t) = (1-\alpha)Q_t(S_t, A_t) + \alpha \hat{G}_t^{\sigma}, \tag{3.2}$$

Just like the Sarsa and Expected Sarsa algorithms, we can represent the $Q(\sigma)$ algorithm using a backup diagram. In this case, we can choose from two different graphical representations. Figure 3.1a, represents the one-step $Q(\sigma)$ algorithm as a convex combination of the backup diagrams corresponding to Sarsa and Expected Sarsa, where Sarsa is weighted by σ and Expected Sarsa is weighted by $(1-\sigma)$. Figure 3.1b represents our algorithm as a single backup diagram with two separate paths: a sampling path weighted by σ and an expectation path weighted by $(1-\sigma)$. The dashed lines in the diagram connecting two solid dots indicate that those actions or state-action pairs are the same in both sides of the diagram. We will demonstrate in the next section that the n-step $Q(\sigma)$ algorithm can always be represented as a convex combination of 2^n different diagrams.

As we can observe in Figure 3.1, the parameters σ allows us to represent a wide variety of algorithms while subsuming the existing one-step algorithms as special cases. Moreover, the parameter σ need not remain constant over time and can be made a function of the state. In such case, we would replace σ with $\sigma_t(S_{t+1})$ in Equation (3.1). Henceforth, we will denote the σ parameter as σ_t to indicate that its value depends on time t, but with the understanding that σ can also be a extended to be a function of the state.

Finally, one-step $Q(\sigma)$ can be extended to the off-policy case by using the importance sampling ratio $\rho_{t+1} = \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})}$, where π is the target policy and μ is the behaviour policy. The resulting estimate of the expected return is

$$\hat{G}_t^{\sigma} \doteq R_{t+1} + \gamma\big[\sigma_t \rho_{t+1} Q(S_{t+1}, A_{t+1}) + (1-\sigma_t)\sum_{a\in\mathcal{A}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a)\big]. \tag{3.3}$$

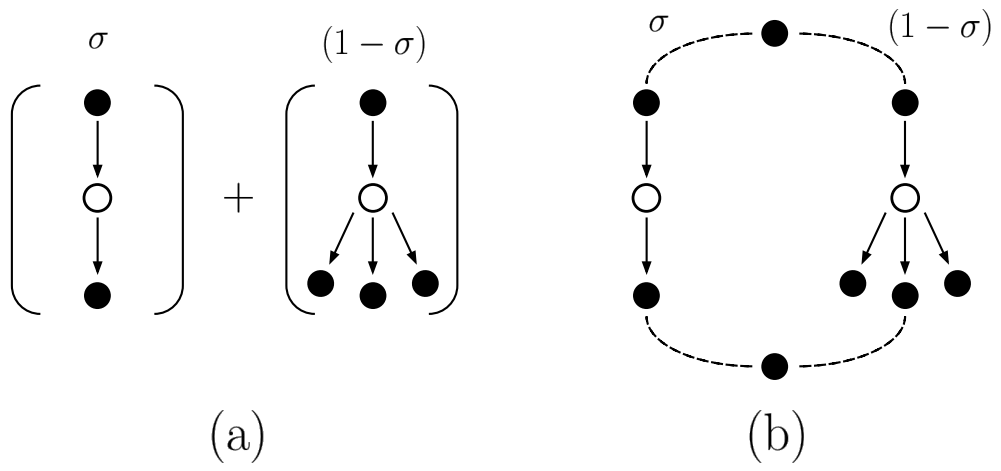


Figure 3.1: Two Graphical Representations of the one-step $Q(\sigma)$ algorithm. Diagram (a) represents $Q(\sigma)$ as a convex combination of Sarsa and Expected Sarsa, each with weights σ and $(1-\sigma)$, respectively. Diagram (b) represents $Q(\sigma)$ as a single diagram with a sampling path weighted by σ and an expectation path weighted by $(1-\sigma)$. The dashed lines connecting two solid dots indicate that both actions or state-action pair are the same in both sides of the backup.

This is equivalent to combining one-step per-decision importance sampling and Expected Sarsa. We can then compute estimates of the action-value by replacing the estimate of the expected return in Equation (3.2) with Equation (3.3). Note that the Expected Sarsa side of the return does not need importance sampling since it is already computing the expected value of the action-value function under the target policy.

3.2 n-Step $Q(\sigma)$

Just like Sarsa and Expected Sarsa, $Q(\sigma)$ can be extended to the n-step case where, instead of looking one step into the future, we look several steps into the future in order to compute an update. In the n-step case, $Q(\sigma)$ unifies the n-step Sarsa and n-step Tree Backup algorithms. In this case, it will be more intuitive to introduce the algorithm graphically using backup diagrams.

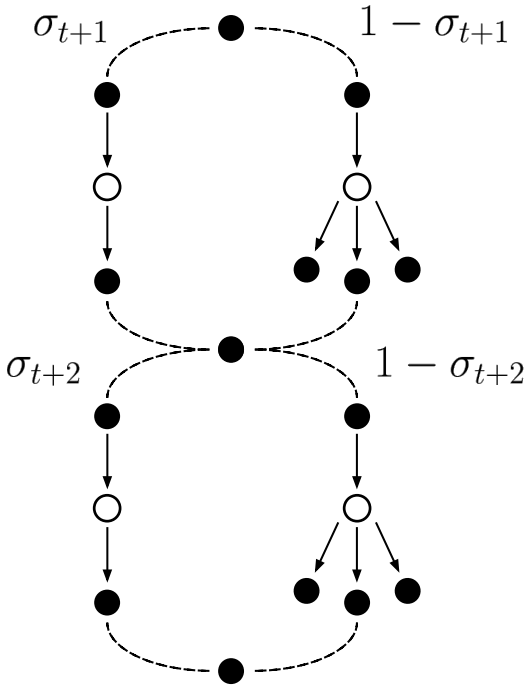


Figure 3.2: Two-Step $Q(\sigma)$ Backup Diagram

Figure 3.2 shows the backup diagram of the two-step $Q(\sigma)$ algorithm. At each step of the diagram, we can can choose between sampling the next state-action pair or taking an expectation over all the possible state-action pairs. At every time step $t + k$, for $k \geq 1$, the sampling path is weighted by σ_{t+k}, whereas the expectation path is weighted by $(1 - \sigma_{t+k})$. Both paths eventually converge to the same state-action pair and then a new step is taken.

The two-step $Q(\sigma)$ algorithm can also be interpreted as a convex combination of several backup diagrams as shown in Figure 3.3. The weights corresponding to each of the backup diagrams are listed above their corresponding diagram. Thus, for a backup length of n, the n-step $Q(\sigma)$ algorithm is a convex combination of 2^n different backup diagrams. Even for a small value of n, it quickly becomes inconvenient to represent the n-step $Q(\sigma)$ algorithm as a convex combination of several different backups. Hence, I advocate for the graphical representation in Figure 3.2 since it is more compact while conveying the same amount of information. It is clear from Figure 3.3 that if we choose a fixed value of σ of one, we recover two-step Sarsa, whereas with a fixed value of σ of zero we recover two-step Tree Backup.

In order to implement the n-step $Q(\sigma)$ algorithm with time-dependent σ,

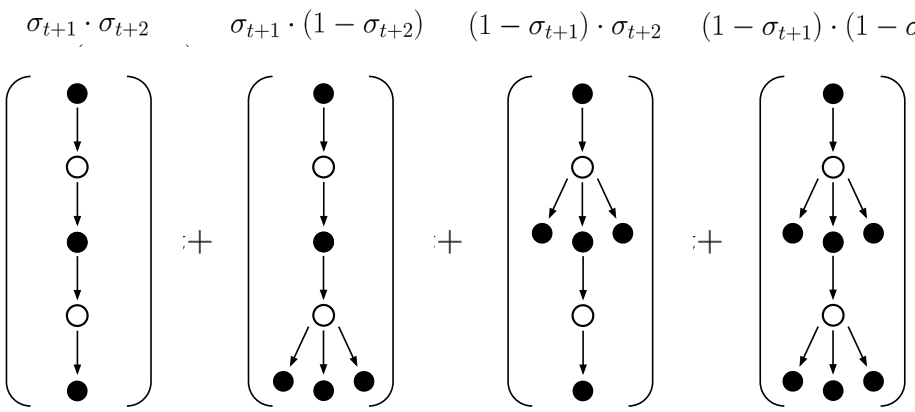


Figure 3.3: Two-step $Q(\sigma)$ backup represented as a convex combination of four different backups. The weight assigned to each backup is written above the corresponding backup diagram.

we use the recursive definition of the estimate of the expected return:

$$\begin{aligned}\hat{G}^{\sigma}_{t:t+n} &\doteq R_{t+1} + \gamma\big[\sigma_{t+1} + (1 - \sigma_{t+1})\pi(A_{t+1}|S_{t+1})\big]\hat{G}^{\sigma}_{t+1:t+n} \\ &\quad + \gamma(1 - \sigma_{t+1}) \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a), \\ \hat{G}^{\sigma}_{t+n:t+n} &\doteq Q_{t+n-1}(S_{t+n}, A_{t+n}). \end{aligned} \tag{3.4}$$

This definition illustrates the idea of choosing between sampling and taking an expectation at every step. At timestep $t + k$, for $n \geq k \geq 1$, we can choose to take an expectation over $\hat{G}^{\sigma}_{t+k:t+n}$ and all the other action-value estimates if we let σ_{t+k} be zero. Otherwise, we can choose to keep unrolling $\hat{G}^{\sigma}_{t+k:t+n}$ by letting σ_{t+k} be equal to one.

Note that the samples needed to compute the estimate of the return are not available until time $t + n$. Consequently, the update for the action-value estimate corresponding to (S_t, A_t) can only be computed at time $t + n$. This results in the n-step update function from the previous chapter

$$Q_{t+n}(S_t, A_t) = (1 - \alpha)Q_{t+n-1}(S_t, A_t) + \alpha\hat{G}^{\sigma}_{t:t+n}. \tag{3.5}$$

As a consequence, the agent does not benefit from any learning during the first n time steps of an episode.

Just like in the one-step case, we can extend the n-step $Q(\sigma)$ algorithm to the off-policy setting via importance sampling. To do so we can modify Equation (3.4) to include the importance sampling term:

$$\begin{aligned}\hat{G}^{\sigma}_{t:t+n} \doteq R_{t+1} + \gamma\big[\sigma_{t+1}\rho_{t+1} + (1-\sigma_{t+1})\pi(A_{t+1}|S_{t+1})\big]\hat{G}^{\sigma}_{t+1:t+n} \\ + \gamma(1-\sigma_{t+1})\sum_{a\neq A_{t+1}}\pi(a|S_{t+1})Q_{t+n-1}(S_{t+1},a),\end{aligned} \tag{3.6}$$

where $\rho_{t+1} = \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})}$ and the base case is the same as for the on-policy case. In the off-policy case, n-step $Q(\sigma)$ unifies the algorithms n-step Tree Backup and n-step per-decision importance sampling (Precup et al., 2000). The update is computed by substituting the estimate of the return in Equation (3.5) with the off-policy version of the estimate of the return.

The algorithm box below shows the pseudocode for the off-policy n-step $Q(\sigma)$. Note that the update is implicitly assuming that $Q_{t+n}(s,a) \doteq Q_t(s,a)$ for all $(s,a) \neq (S_t, A_t)$.

Algorithm 1 Off-policy n-step $Q(\sigma)$

Input: a behaviour policy μ such that $\mu(a|s) > 0$ for all $(s, a) \in \mathcal{S} \times \mathcal{A}$
Initialize $Q(s, a)$ arbitrarily for all $(s, a) \in \mathcal{S} \times \mathcal{A}$
Initialize the target policy π as a function of Q or as fixed policy
Algorithm Parameters: learning rate $\alpha \in (0, 1]$, backup length n
for Every Episode **do**
  Initialize S_0, select $A_0 \sim \mu(\cdot|S_0)$
  Store S_0 and A_0
  Initialize $T \leftarrow \infty$ (The time of termination)
  Initialize $t \leftarrow 0$ (The current time step)
  while $t < T + n - 1$ **do**
    if $t < T$ **then**
      Take action A_t; observe and store R_{t+1} and S_{t+1}
      if S_{t+1} is terminal **then**
        $T \leftarrow t + 1$
      else
        Choose and store $A_{t+1}, \mu(A_{t+1}|S_{t+1}), \pi(A_{t+1}|S_{t+1})$ and σ_{t+1}
      end if
    end if
    $\tau \leftarrow t + 1 - n$ (τ is the time whose estimate is being updated)
    if $\tau \geq 0$ **then**
      $l \leftarrow \min(t + 1, T)$ (l is the last time step of the backup)
      if $l \leq T$ **then**
        $\hat{G} \leftarrow 0$
      else
        $\hat{G} \leftarrow Q(S_l, A_l)$
      end if
      for $k = l, l - 1, ..., \tau + 1$ **do**
        $V \leftarrow \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a)$
        $\rho \leftarrow \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$
        $\hat{G} \leftarrow R_k + \gamma\big[\sigma_k \rho + (1 - \sigma_k)\pi(A_k|S_k)\big]\hat{G} + \gamma(1 - \sigma_k)V$
      end for
      $Q(S_\tau, A_\tau) \leftarrow (1 - \alpha)Q(S_\tau, A_\tau) + \alpha\hat{G}$
    end if
    $t \leftarrow t + 1$
  end while
end for

3.3 Bias and Variance Analysis of n-Step $Q(\sigma)$

But why use n-step $Q(\sigma)$? Before demonstrating the performance of n-step $Q(\sigma)$ empirically, we develop some intuition about why it would be beneficial

to use our algorithm. In order to develop this intuition, we look at the bias and the variance of the estimate of the expected return.

First, let us define what we mean by the bias and the variance of the estimate of the expected return. Assume that we are using the estimate of the expected return:

$$\hat{G}_{t:t+n}(S_t, A_t) \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^n q_\pi(S_{t+n}, A_{t+n}),$$

where we have extended the notation by indicating the initial state-action pair of the sequence. We can measure the *mean squared error* (MSE) of our estimate given some initial state-action pair (s, a) as

$$\begin{aligned}
&\text{MSE}(\hat{G}_{t:t+n}(s,a)) \\
&\quad = \mathbb{E}_\pi\big\{ \big(\hat{G}_{t:t+n}(S_t, A_t) - q_\pi(S_t, A_t)\big)^2 \big| S_t = s, A_t = a \big\} \\
&\quad = \mathbb{E}_\pi\big\{ \big(\hat{G}_{t:t+n}(S_t, A_t) - \mathbb{E}_\pi\{\hat{G}_{t:t+n}(S_t, A_t)\} \\
&\qquad\qquad + \mathbb{E}_\pi\{\hat{G}_{t:t+n}(S_t, A_t)\} - q_\pi(S_t, A_t)\big)^2 \big\} \\
&\quad = \mathbb{E}_\pi\big\{ \big(G_{t:t+n} - \mathbb{E}_\pi\{G_{t:t+n}\}\big)^2 \\
&\qquad\qquad + 2(\hat{G}_{t:t+n} - \mathbb{E}_\pi\{\hat{G}_{t:t+n}\})(\mathbb{E}_\pi\{\hat{G}_{t:t+n}\} - q_\pi(S_t, A_t)) \\
&\qquad\qquad + \big(\mathbb{E}_\pi\{\hat{G}_{t:t+n}\} - q_\pi(S_t, A_t)\big)^2 \big\} \\
&\quad = \mathbb{E}_\pi\big\{ \big(\hat{G}_{t:t+n}(S_t, A_t) - \mathbb{E}_\pi\{\hat{G}_{t:t+n}(S_t, A_t)\}\big)^2 \big\} \\
&\qquad\qquad + \big(\mathbb{E}_\pi\{\hat{G}_{t:t+n}(S_t, A_t)\} - q_\pi(S_t, A_t)\big)^2 \\
&\quad = \mathbb{V}_\pi\{\hat{G}_{t:t+n}(S_t, A_t)\} + \text{Bias}(\hat{G}_{t:t+n}(S_t, A_t))^2, \qquad (3.7)
\end{aligned}$$

where we omitted the dependence on s and a in all the expectations after the first equality for concreteness. We will adopt this writing convention henceforth.

Equation (3.7) shows that the MSE of the n-step estimate of the expected return can be decomposed into two quantities: the variance and the bias of $\hat{G}_{t:t+n}(S_t, A_t)$. However, TD methods do not have access to q_π; instead, the estimate of the expected return is modeled after a Bellman equation and we use our current estimates of the action-value function to replace q_π. In this case, we can compute the MSE for a given algorithm by replacing $\hat{G}_{t:t+n}$ in

the last line of Equation (3.7), with the estimate of the expected return of the corresponding algorithm.

For n-step $Q(\sigma)$, the bias of the estimate of the expected return is

$$\begin{aligned}
&\text{Bias}\big(\hat{G}^{\sigma}_{t:t+n}(s,a)\big) \\
&\quad = \mathbb{E}_{\pi}\{\hat{G}^{\sigma}_{t:t+n}(S_t,A_t) - q_{\pi}(S_t,A_t) | S_t = s, A_t = a\} \\
&\quad = \gamma\mathbb{E}_{\pi}\Big\{(1-\sigma_{t+1})\Big[\sum_{a\neq A_{t+1}} \pi(a|S_{t+1})\big(Q(S_{t+1},a) - q_{\pi}(S_{t+1},a)\big) \\
&\qquad\qquad + \pi(A_{t+1}|S_{t+1})\big(\hat{G}^{\sigma}_{t+1:t+n}(S_{t+1},A_{t+1}) - q_{\pi}(S_{t+1},A_{t+1})\big)\Big] \\
&\qquad\qquad + \sigma_{t+1}\Big[\hat{G}^{\sigma}_{t+1:t+n}(S_{t+1},A_{t+1}) - q_{\pi}(S_{t+1},A_{t+1})\Big]\Big\}.
\end{aligned} \tag{3.8}$$

If the branches of the expectation are too inaccurate (i.e. the difference between $Q(S_{t+1},a)$ and $q_{\pi}(S_{t+1},a)$ is too large), one could set $\sigma = 1$ to eliminate that side of the return. On the other hand, if the total bias of the branches are less than the bias of the next estimate of the expected return, then one could set $\sigma = 0$ and assign less weight to next estimate, which would decrease the bias of the overall estimate of the expected return.

The variance of n-step $Q(\sigma)$ is

$$\begin{aligned}
&\mathbb{V}_{\pi}\big\{\hat{G}^{\sigma}_{t:t+n}(S_t,A_t) | S_t = s, A_t = a\big\} \\
&\quad = \mathbb{V}_{\pi}\Big\{R_{t+1} + \gamma\sigma_{t+1}\hat{G}^{\sigma}_{t+1:t+n} \\
&\qquad\qquad + \gamma(1-\sigma_{t+1})\Big[\pi(A_{t+1}|S_{t+1})\hat{G}^{\sigma}_{t+1:t+n} + \sum_{a\neq A_{t+1}} \pi(a|S_{t+1})Q(S_{t+1},a)\Big]\Big\}.
\end{aligned} \tag{3.9}$$

If we select $\sigma_{t+1} = 0$, then the contribution of $\hat{G}^{\sigma}_{t+1:t+n}$ to the total variance is reduced by a factor of $\pi(A_{t+1}|S_{t+1})$. However, this could still increase the MSE if the branches of the expectation are too biased.

An in-depth study of how to choose σ to minimize the MSE is far from the scope of this thesis. We will use a simple heuristic in the next section that benefits from the intuition developed here. In our proposed heuristic, we initialize σ with a value of one and decay its value towards zero over the course of training. During early training the branches of the expectation likely

have large bias since the estimates of the action-value function are randomly initialized. Hence, choosing a value of σ close to one could result in a less biased estimate of the expected return. Once the estimates of the action-value function get closer to q_π, choosing a value of σ close to zero could decrease the bias and the variance of the estimate of the return. We show in the next sections that this heuristic can often outperform other variants of n-step $Q(\sigma)$ that use a fixed value of σ.

3.4 Empirical Evaluations of Tabular n-Step $Q(\sigma)$

We have given an in-depth introduction of the n-step $Q(\sigma)$ algorithm and provided some intuition about how to choose σ to reduce the mean squared error. However, is there a real benefit of using n-step $Q(\sigma)$? Our goal in the following two sections is to go beyond the intuition developed so far and answer this question through empirical evaluations. We implemented Algorithm 1 and demonstrate its performance in two tabular domains: the 19-state random walk and the stochastic windy gridworld.

3.4.1 Tabular n-Step Q(σ) for On-Policy Prediction

In the previous section, we provided intuition about the role of σ on the bias and variance of the estimate of the return. There we argued that, if the estimates of the action-value function are too inaccurate, then using a value of σ close to 0 will result in higher bias than with a value of σ close to 1. The inverse of this effect occurs once the estimates of the action-value function become more accurate. In this experiment we studied this effect empirically. We hypothesized that values of σ close to 1 would perform better during early training — when the estimates of the action-value function are less accurate. On the other hand, values of σ close to 0 would perform better during late training — once the estimates are more accurate. We study this hypothesis several times during this chapter. For this first experiment, we start with a learning task that allows us to gain a simplified and clear view of the behaviour

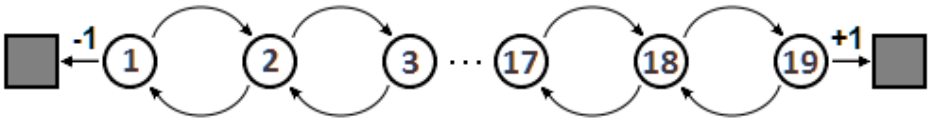


Figure 3.4: The 19-state random walk environment. Reaching the leftmost state results in a reward of -1, whereas the rightmost state results in a reward of $+1$. Reaching either of these two states terminates the episode.

of n-step $Q(\sigma)$.

We used the 19-state random walk environment from Sutton and Barto (1998)— illustrated in figure 3.4. In this environment the agent has two actions available: moving left and moving right. Moving into the rightmost or leftmost states results in a reward of $+1$ and -1, correspondingly, and terminates the episode.

We approached this task as an on-policy prediction problem. At every time step, the agent follows and estimates the action-values corresponding to a fixed policy that with equal probability chooses between the left or right actions. The benefit of using this type of environment is that it is possible to compute analytically the exact action-value function for each state. We can then use this information to assess the accuracy of the action-value functions computed by the n-step $Q(\sigma)$ algorithm at different times during training. The performance measure that we used for this experiment is the root-mean-squared (RMS) error between the true and the estimated action-value function.

We implemented $Q(\sigma)$ agents for each value of σ from 0 to 1 at increments of 0.25 and a value of γ of 0.9. For each of these algorithms, we tested several values for the learning rate α and the backup length n and report the results of the best parameter combination in terms of the average RMS error over 50 episodes of training. For all the algorithms, the best parameter combination was $n = 3$ and $\alpha = 0.4$.

$Q(1)$ (Sarsa) performed the best among all the other algorithms early during training, whereas $Q(0)$ (Tree Backup) performed the best at the end of training. As the value of σ decreased from 1 to 0, the corresponding algorithms performed worse during early training and better during late training.

These results support our initial hypothesis and the intuition developed in the previous section. It seems that using a value of σ close to 1 reduces the bias during early training, whereas a value of σ close to 0 reduces the bias and variance during late training.

Given these conclusions, it seemed possible to devise an algorithm that benefits from the initial performance induced by a value of σ close to 1 and the final performance induced by a value of σ close to 0. We decided to test this idea by extending the experiment to include another n-step $Q(\sigma)$ algorithm. For this instance of the n-step $Q(\sigma)$ algorithm, the parameter σ is initialized at a value of 1 and slowly decays to 0 by a factor $\beta \in [0, 1)$ at the end of each episode. Specifically, we let $\sigma_{k+1} = \beta \cdot \sigma_k$ for $k \geq 1$ and $\sigma_1 = 1$. We originally called this algorithm *Dynamic* σ; however, I will use the name *Decaying* σ since it is more faithful to the true nature of the algorithm. We hypothesized that using this simple heuristic, the algorithm would be able to perform the best during early and late training.

We implemented a decaying σ algorithm with decay rate β of 0.9 and tested several values of the parameters n and α. The best parameter combination for decaying σ was the same as for the other algorithms, $n = 3$ and $\alpha = 0.4$.

Compared to the algorithms from the first part of this experiment, decaying σ performed the best at every stage of learning. Figure 3.5 shows the RMS error computed at the end of each episode for each algorithm. The results are averaged over 100 runs. All the standard errors are less than 0.003, which is narrower than the line width.

It seems that, for this particular domain, the decaying σ algorithm is taking advantage of the initial performance induced by high values of σ and the final performance corresponding to small values of σ. This supports our hypothesis about the decaying σ algorithm and further supports the intuition developed in the previous section. A value of σ close to one makes the estimate of the return have less bias early during training. As training progresses and the estimates of the action-value function become more accurate, a small value of σ results in less bias and smaller variance. We encounter evidence of this effect several times during our empirical evaluations.

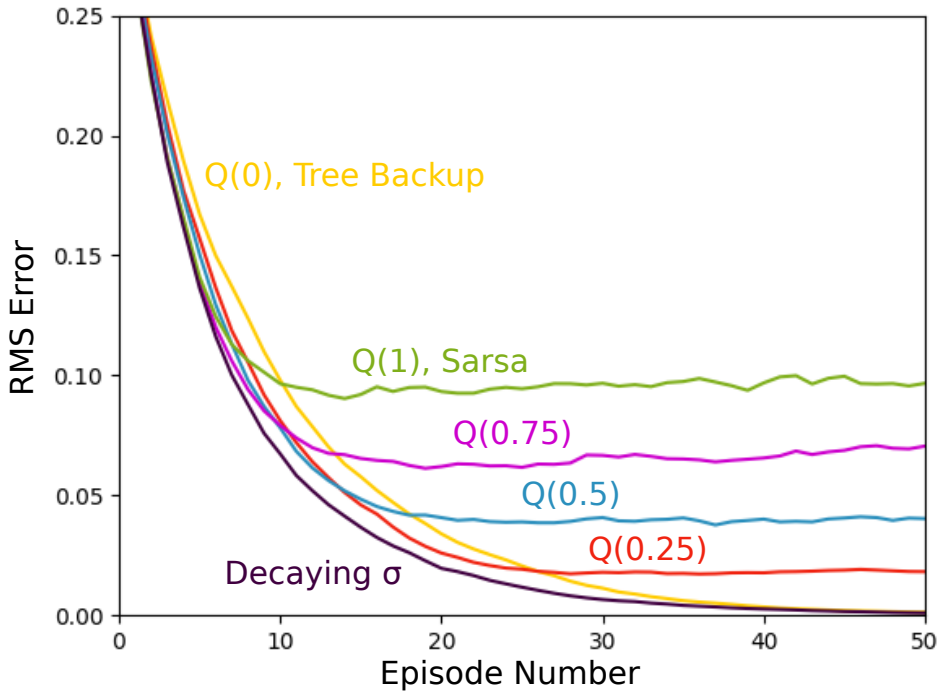


Figure 3.5: Results of the 19-state random walk experiment. Small values of σ resulted in better initial performance in terms of root-mean-squared (RMS) error. Larger values of σ resulted in better final performance. Decaying σ with decay rate of 0.9 outperformed all the other variants of $Q(\sigma)$ with fixed value of σ.

3.5 Tabular n-Step Q(σ) for On-Policy Control

In this experiment, we will increase the complexity of the learning task and study how the parameters σ, n, and α interact with each other and influence the performance of n-step $Q(\sigma)$. Based on the results of the previous experiment, we hypothesized that the decaying σ algorithm would perform the best over a wide variety of settings of the parameters n and α. To test this hypothesis, we set up the learning problem as an on-policy control task in a variant of the windy gridworld environment (Sutton & Barto, 1998; van Seijen et al., 2009).

In the windy gridworld environment (Figrue 3.6), states are represented as discrete (x, y) coordinates indicating the position of the agent in a grid enclosed within four walls. The actions available to the agent are moving

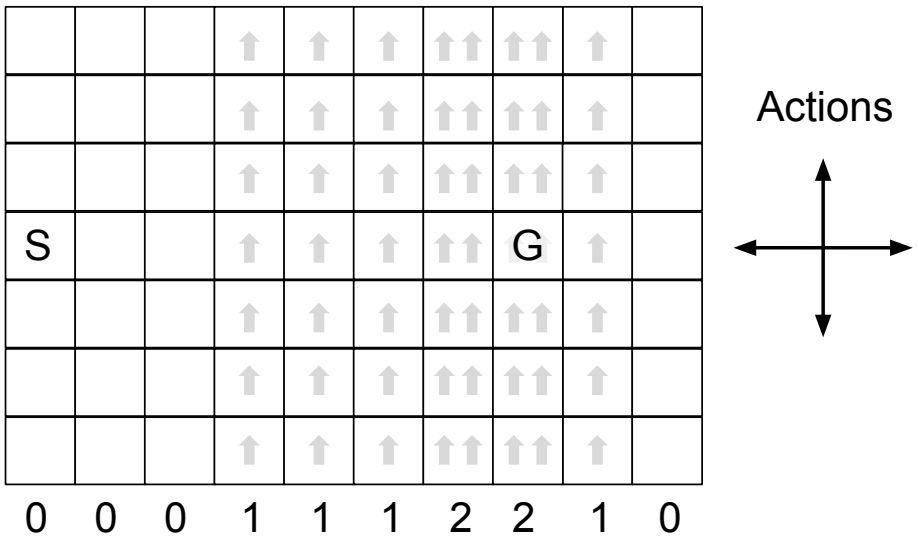


Figure 3.6: The windy gridworld environment. The agent starts at state S and its goal is to reach the terminal state G. The actions available are moving north, east, south, and west. At every column in the grid there is an upward wind that pushes the agent north k number of squares as indicated at the bottom of each column. The reward is -1 at every transition until the agent reaches state G.

north, east, south, or west. At every column in the grid there is an upward wind that pushes the agent north k number of squares after an action has been selected and executed. After every action, the agent receives a reward of -1. When an agent tries to move against a wall it stays in the same location, but executing the action still results in a reward of -1. The agent starts at an initial state S and an episode terminates only when the agent moves into a goal state G. Consequently, in order to maximize reward, the agent has to learn how to navigate the gridworld while accounting for the effects of the wind.

For this experiment, We used a variant of the windy gridworld environment called stochastic windy gridworld. In this variant, all the details of the environment are the same except for the way that actions are executed. In the stochastic windy gridworld, the agent moves to the desired direction only with a probability of 0.9. Otherwise, the agent is moved into one of the eight adjacent squares chosen uniformly at random. The effect of the wind occurs after the action has been selected and executed just as in the original environment.

In order to study the effects of the parameters σ, n, and α, we implemented

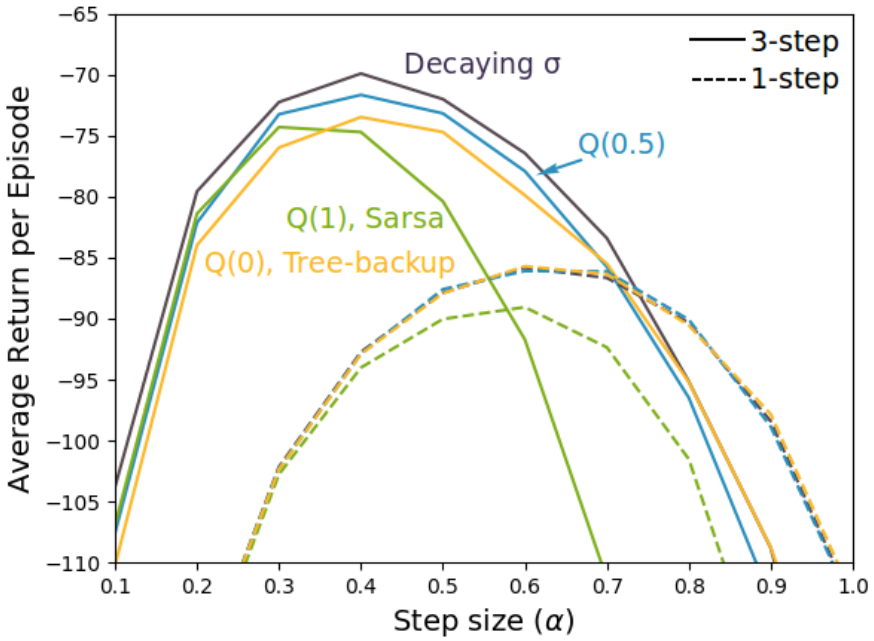


Figure 3.7: Results of the Experiment on the Stochastic Windy Gridworld. The results are averaged over 1000 runs of 100 episodes each. The standard error is narrower than the line width. The best performance for all the algorithms was achieved for a backup length, n, with value of three. For this backup length, Decaying σ and $Q(0.5)$ have a better performance than Sarsa and Tree backup over a wide range of values of α.

eighty different $Q(\sigma)$ agents. For each value of n in $\{1, 3\}$, we implemented four $Q(\sigma)$ agents, one for each value of σ in $\{0, 0.5, 1\}$ and a decaying σ agent with decay rate of 0.95. For each of these settings, we trained an agent for each value of α from 0.1 to 1 at increments of 0.1. All the agents used an ϵ-greedy policy with $\epsilon = 0.1$. The measure of performance we used was the average return per episode over 100 episodes.

For a back up length of 1, $Q(0.5)$, $Q(0)$ and decaying σ performed the same across all values of α, while $Q(1)$'s performance was lower at most values of α. For a backup length of 3, for most of the values of α the resulting performance was higher than for a backup length of 1 across all the settings for σ. Also for $n = 3$, decaying σ performed better than the algorithms with fixed σ for most of the values of α.

The parameter combination that resulted in the best performance was $n =$

3 and $\alpha = 0.4$ for decaying σ, $Q(0.5)$, and $Q(0)$; and $n = 3$ and $\alpha = 0.3$ for $Q(1)$. Comparing only the best parameter combination of each algorithm we observed that decaying σ performed the best with $Q(0.5)$ as a close second.

Figure 3.7 shows the average return per episode over 100 episodes. The results are averaged over 1,000 independent runs of 100 episodes each. The error bars corresponding to the standard error are narrower than the line width.

The results of the experiment partially support the initial hypothesis. Decaying σ performed better than the rest of the algorithms for a wide variety of values of α when the backup length was 3. However, this was not the case for a backup length of 1, in which case decaying σ, $Q(0.5)$, and $Q(0)$ performed the same. Lastly, for $n = 3$, $Q(0.5)$ also managed to outperform $Q(0)$ and $Q(1)$ demonstrating that intermediate values of σ can also perform better than either of the extremes. In the next Chapters, we find similar effects in more complex environments when using function approximation.

Chapter 4

n-Step $Q(\sigma)$ with Linear Function Approximation

In Chapter 3 we introduced the n-step $Q(\sigma)$ algorithm — the main subject of study of this thesis — in the tabular case. On one hand, tabular representations of the estimates of the action-value function allow the estimates to converge to the exact action-value function. On the other hand, when the state or action spaces are too large or infinite, it becomes impractical or even impossible to store an estimate of the action-value function for each state-action pair. Even when it is possible to store all the estimates of the action-value function, learning might occur very slowly since there is no generalization across state-action pairs. Function approximation allows for compact representations of the action-value function that can generalize better across states and actions while incurring in some — or none at all in some cases — loss of precision.

The main contribution of this chapter is to extend the n-step $Q(\sigma)$ algorithm to the linear function approximation case. This is an important extension because it improves the applicability of n-step $Q(\sigma)$ to more complex domains. We then provide empirical evaluations of n-step $Q(\sigma)$ and draw parallels to the results obtained in the tabular case.

We will give up trying to exactly estimate q_π for each individual state-action pairs in $\mathcal{S} \times \mathcal{A}$. Instead, we will approximate the action-value function using a parameterized function $\hat{q}(s, a, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is a vector of parameters in $\mathbb{R}^d$. This parameterization reduces the dimensionality of our representation from $|\mathcal{S} \times \mathcal{A}|$ to d, which can be orders of magnitude smaller. Moreover, adjusting

one of the elements in $\boldsymbol{\theta}$ can change the approximation of the action-value function for several states and actions, which improves generalization.

As introduced in Chapter 2, to compute $\hat{q}(s, a, \boldsymbol{\theta})$ we minimize with respect to $\boldsymbol{\theta}$ the mean squared action-value error

$$\overline{\text{AVE}}(\boldsymbol{\theta}) \doteq \sum_{s \in \mathcal{S}} \eta(s) \sum_{a \in \mathcal{A}} \pi(a|s) \big[q_\pi(s, a) - \hat{q}(s, a, \boldsymbol{\theta})\big]^2, \tag{4.1}$$

where η is the distribution of the states such that $\eta(s) \geq 0 \ \forall \ s \in \mathcal{S}$.

To minimize this objective function we use the stochastic gradient descent update

$$\begin{aligned} \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \frac{1}{2}\alpha \nabla \big[q_\pi(s, a) - \hat{q}(s, a, \boldsymbol{\theta}_t)\big]^2 \\ &= \boldsymbol{\theta}_t + \alpha \big[q_\pi(s, a) - \hat{q}(s, a, \boldsymbol{\theta}_t)\big] \nabla \hat{q}(s, a, \boldsymbol{\theta}_t), \end{aligned} \tag{4.2}$$

where $\boldsymbol{\theta}_t$ is the parameter vector at time step t and the operator ∇ represents the gradient with respect to $\boldsymbol{\theta}_t$. We do not have access to q_π; instead, we estimate it using the estimate of the expected return $\hat{G}_{t:t+n}$ corresponding to the n-step $Q(\sigma)$ algorithm. In this case, $\hat{G}_{t:t+n}$ is computed using $\hat{q}(\cdot, \cdot, \boldsymbol{\theta}_t)$ instead of $Q_t(\cdot, \cdot)$. This results in the semi-gradient update

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \big[\hat{G}_{t:t+n} - \hat{q}(s, a, \boldsymbol{\theta}_t)\big] \nabla \hat{q}(s, a, \boldsymbol{\theta}_t). \tag{4.3}$$

In the linear function approximation case, $\hat{q}$ is represented as a linear combination of the elements in $\boldsymbol{\theta}_t$ and a feature vector $\mathbf{x}(s, a)$. Given $\boldsymbol{\theta}_t$ and $\mathbf{x}(s, a)$ in $\mathbb{R}^d$ each with elements $\theta_{t,i}$ and $x_i(s, a)$, respectively, $\hat{q}$ can be represented as a dot product between these two vectors

$$\hat{q}(s, a, \boldsymbol{\theta}_t) \doteq \boldsymbol{\theta}_t^T \mathbf{x}(s, a) \doteq \sum_{i=1}^{d} \theta_{t,i} x_i(s, a). \tag{4.4}$$

In this case, the gradient of $\hat{q}(s, a, \boldsymbol{\theta}_t)$ with respect to $\boldsymbol{\theta}_t$ is simply $\mathbf{x}(s, a)$. Hence, the update function is easily computed by replacing $\nabla \hat{q}(s, a, \boldsymbol{\theta}_t)$ with $\mathbf{x}(s, a)$ in Equation 4.3.

We will consider feature vectors constructed using tile coding as introduced in Chapter 2. The resulting algorithm is reminiscent of Algorithm 1 except that updates are computed for the parameter $\boldsymbol{\theta}_t$ and we use $\hat{q}(S_t, A_t, \boldsymbol{\theta}_t)$ as the

estimate of the action-value function instead of $Q_t(S_t, A_t)$. The pseudocode can be found in the algorithm box below.

Algorithm 2 Off-policy n-step $Q(\sigma)$ with Linear Function Approximation

Input: a behaviour policy μ such that $\mu(a|s) > 0$ for all $(s, a) \in \mathcal{S} \times \mathcal{A}$
Initialize $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
Initialize the target policy π as a function of $\hat{q}$ or as fixed policy
Algorithm Parameters: learning rate $\alpha \in (0, 1]$, backup length n, number of tilings, number of tiles per tiling
for Every Episode **do**
  Initialize S_0, select $A_0 \sim \mu(\cdot|S_0)$
  Store S_0 and A_0
  Initialize $T \leftarrow \infty$ (The time of termination)
  Initialize $t \leftarrow 0$ (The current time step)
  while $t < T + n$ **do**
    if $t < T$ **then**
      Take action A_t; observe and store R_{t+1} and S_{t+1}
      if S_{t+1} is terminal **then**
        $T \leftarrow t + 1$
      else
        Choose and store A_{t+1}, $\mu(A_{t+1}|S_{t+1})$, $\pi(A_{t+1}|S_{t+1})$, and σ_{t+1}
      end if
    end if
    $\tau \leftarrow t + 1 - n$ (τ is the time whose estimate is being updated)
    if $\tau \geq 0$ **then**
      $l \leftarrow \min(t + 1, T)$ (The last time step of the backup)
      if $l = T$ **then**
        $\hat{G} \leftarrow 0$
      else
        $\hat{G} \leftarrow \hat{q}(S_l, A_l, \boldsymbol{\theta})$
      end if
      for $k = l, l - 1, ..., \tau + 1$ **do**
        $V \leftarrow \sum_{a \neq A_k} \pi(a|S_k)\hat{q}(S_k, A_k, \boldsymbol{\theta})$
        $\rho \leftarrow \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$
        $\hat{G} \leftarrow R_k + \gamma\big[\sigma_k\rho + (1 - \sigma_k)\pi(A_k|S_k)\big]\hat{G} + \gamma(1 - \sigma_k)V$
      end for
      $\boldsymbol{\theta} \leftarrow (1 - \alpha)\hat{q}(S_\tau, A_\tau, \boldsymbol{\theta}) + \alpha\hat{G}$
    end if
    $t \leftarrow t + 1$
  end while
end for

4.1 Empirical Evaluations of n-Step $Q(\sigma)$ with Linear Function Approximation

We continue increasing the complexity of the learning task to study how robust are the results we found in the two experiments in the tabular case. In this case, we study the performance of n-step $Q(\sigma)$ with linear function approximation as presented in Algorithm 2. We hypothesized that we would find similar effects as in the tabular case: from the algorithms with fixed σ (1) $Q(1)$ would perform the best during early training, (2) $Q(0)$ would perform the best during late training, (3) there would be an algorithm with intermediate value of σ that performed the best in terms of average performance over all the episodes, and (4) decaying σ would perform better than all the algorithms with fixed σ in terms of average performance over all the episodes. We studied these hypotheses in an on-policy control task in a variant of the mountain car environment (Sutton, 1996; Sutton & Barto, 1998) that we named *mountain cliff*.

In the mountain cliff environment (Figure 4.1) states are represented by a tuple of two continuous numbers that represent the position and the velocity of a car that is located in the valley between two hills. The actions available to the agent are accelerate forward (+1), accelerate backwards (−1), or coast (0). Every action results in a reward of −1 and changes the position x_t and the velocity y_t of the car according to the simplified physics model:

$$\begin{aligned} y_{t+1} &\doteq \mathit{bound}\big[y_t + 0.001A_t - 0.0025\cos 3x_t\big], \\ x_{t+1} &\doteq \mathit{bound}\big[x_t + y_{t+1}\big], \end{aligned} \tag{4.5}$$

where the *bound* operation enforces $x_t \in [-1.2, 0.5]$ and $y_t \in [-0.07, 0.07]$. The initial state of every episode is initialized to a random position in the interval $[-0.6, -0.4)$ and with zero velocity. If an agent ever moves beyond the top of the left hill (i.e., the position of the agent before the *bound* operation is less than −1.2), it falls off a cliff, receives a reward of −100, and is sent to a random state chosen in the same way as the initial state. On the other hand, once the agent reaches the top of the right hill (position 0.5) the episode terminates.

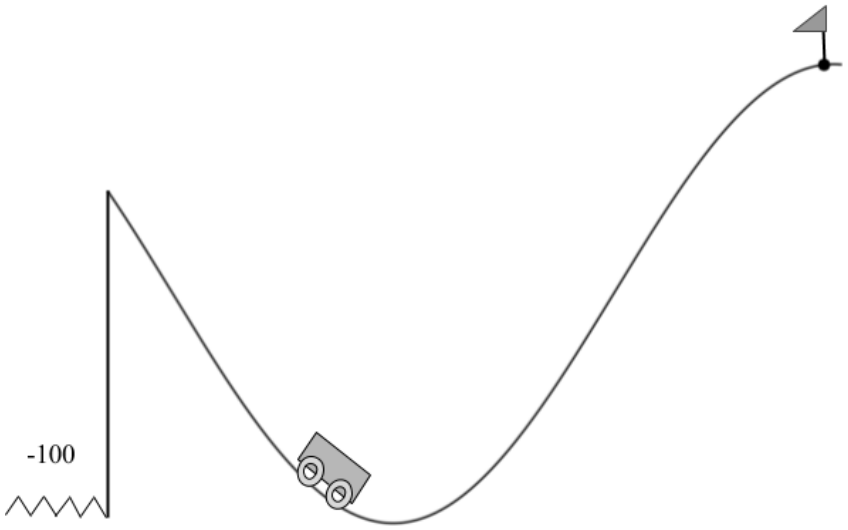


Figure 4.1: The mountain cliff environment. The agent controls a car that is located between two hills. The goal is to reach the top of the right hill. The state is represented as a tuple containing the current position and velocity of the car. The actions are accelerating forward, backwards, or coasting. The agent receives a reward of -1 on every transition and a reward of -100 when it falls off the cliff at the top of the left hill. In the latter case, the agent is returned to a random location in the valley between the two hills with a velocity of zero.

Consequently, maximizing reward implies reaching the top of the right hill as fast as possible. However, the engine of the car is not strong enough to overcome the force of gravity. Hence, the agent has to build momentum by moving back and forth to reach the top of the right hill without falling off the cliff at the end of the left hill.

The only difference between the mountain car and mountain cliff environments is the cliff at the end of the left hill. In mountain car moving beyond the top of the left hill stops the movement of the agent and resets its velocity back to zero (i.e., the position is set to -1.2 and the velocity is set to 0). In comparison, the outcome in this same situation in the mountain cliff environment is a lot more punishing to the agent. Hence, the agent has to be more precise when moving onto the left hill. The consequence of this environment design choice is that the difference in performance between different algorithms is accentuated, which facilitates their comparison. All the agents were trained

for 500 episodes in this environment.

In order to study the effects of the parameter σ on the performance of the n-step $Q(\sigma)$ algorithm we implemented six different agents: one for each value of σ in $\{0, 0.25, 0.5, 0.75, 1\}$ and a decaying σ agent with decay rate of 0.95. All the agents, were trained on-policy using an ϵ-greed policy with $\epsilon = 0.1$. The action-value was approximated using linear function approximation with tile coded features. The tile coder was made up of 32 tilings with asymmetric offset and with each tiling covering 1/8-th of the bounded distance in each dimension. Using this configuration, the total number of parameters used by the function approximator was 6,144.

The performance measure that we used in this task was the average return per episode. To study the initial performance we used the first 50 episodes to compute the average return per episode, whereas for the final performance we used the last 50 episodes of training. To study the overall performance of the algorithm we looked at the average return per episode over 500 episodes. We optimized for the overall performance over the parameters n and α in order to find the best parameter combination for each algorithm. Decaying σ, $Q(0)$, $Q(0.25)$, and $Q(0.5)$ performed the best with $n = 4$ and $\alpha = 1/3$; $Q(0.75)$ performed the best with $n = 3$ and $\alpha = 1/3$; and $Q(1)$ performed the best with $n = 3$ and $\alpha = 1/4$. These values of α correspond to the values before dividing by the number of tilings used in the tile coder.

In terms of initial performance, the decaying σ algorithm performed the best followed by $Q(0.5)$. For algorithms with fixed σ, the initial performance decreased the farther away σ was from 0.5. In terms of final performance, decaying σ and $Q(0)$ performed the best followed closely by $Q(0.25)$. As the value of σ increased from 0 to 1, the final performance of the corresponding algorithms decreased with $Q(1)$ having the worst final performance. Table 4.1 shows the summary of the initial and final performance for all the algorithms averaged over 1,500 independent runs.

In terms of overall performance, decaying σ performed the best among all the algorithms closely followed by $Q(0.25)$. $Q(0)$ (Tree Backup) performed the third best among all the algorithms, while $Q(1)$ (Sarsa) performed the worst.

Table 4.1: Summaries of the average return over the first and last 50 episodes for each algorithm. The standard error, and lower (LB) and upper (UB) 95% confidence interval bounds are provided to validate the results. Decaying σ had the best initial. Decaying σ and Tree Backup tied for the best final performance.

	Average Return of First 50 Episodes			
Algorithm	Mean	Standard Error	LB	UB
$Q(1)$, Sarsa	-403.14	0.53	-402.11	-404.17
$Q(0.75)$	-366.98	0.5	-366.01	-367.95
$Q(0.5)$	-352.05	0.56	-350.96	-353.14
$Q(0.25)$	-357.05	0.55	-355.97	-358.12
$Q(0)$, Tree-backup	-360.77	0.52	-359.76	-361.79
Decaying σ	**-346.15**	0.59	**-345.0**	**-347.3**

	Average Return of Last 50 Episodes			
Algorithm	Mean	Standard Error	LB	UB
$Q(1)$, Sarsa	-134.19	0.2	-133.79	-134.58
$Q(0.75)$	-132.5	0.18	-132.14	-132.85
$Q(0.5)$	-131.86	0.25	-131.38	-132.35
$Q(0.25)$	-128.19	0.19	-127.83	-128.55
$Q(0)$, Tree Backup	**-126.95**	0.16	**-126.63**	**-127.26**
Decaying σ	**-127.03**	0.16	**-126.71**	**-127.35**

The summary of the overall performance for all the algorithms can be seen in Table 4.2.

Figure 4.2 shows the performance of all the algorithms at intervals of 50 episodes. The results are the average over 1,500 independent runs and the error bars correspond to a 95% confidence interval computed using a t-distribution.

The results from Table 4.1 provide evidence against our first hypothesis; values of σ close to one do not necessarily result in better initial performance. On the other hand, the second part of Table 4.1 provides support in favor of our second hypothesis; small values of σ resulted in better final performance. For our third hypothesis, we found inconclusive evidence. The average overall performance of $Q(0.25)$ was better than for $Q(0)$, but the difference is not statistically significant. Finally, the results in Table 4.2 support our the fourth

Table 4.2: Summaries of the performance of each algorithm in terms of average return per episode over 500 episodes. The standard error, and lower (LB) and upper (UB) 95% confidence interval bounds are provided to validate the results. Under this measure of performance, Decaying σ achieved the best performance followed closely by $Q(0.25)$.

	Average Return Over 500 Episodes			
Algorithm	Mean	Standard Error	LB	UB
$Q(1)$, Sarsa	-164.39	2.42	-164.51	-164.26
$Q(0.75)$	-159.03	2.17	-159.14	-158.92
$Q(0.5)$	-155.64	2.69	-155.77	-155.5
$Q(0.25)$	-153.97	2.36	-154.09	-153.85
$Q(0)$, Tree Backup	-154.07	2.23	-154.18	-153.96
Decaying σ	**-152.5**	2.38	**-152.62**	**-152.38**

hypothesis; in this environment, decaying σ had the best overall performance among all the algorithms.

The effect of the parameter σ seems to vary depending on the environment and the representation used for the action-value function. Nevertheless, it still seems possible to improve the performance of the algorithm by allowing σ to change over time — even with our simple heuristic. We now proceed to study these same four hypotheses in the non-linear function approximation case where we observe effects reminiscent of the results in the tabular and linear function approximation case.

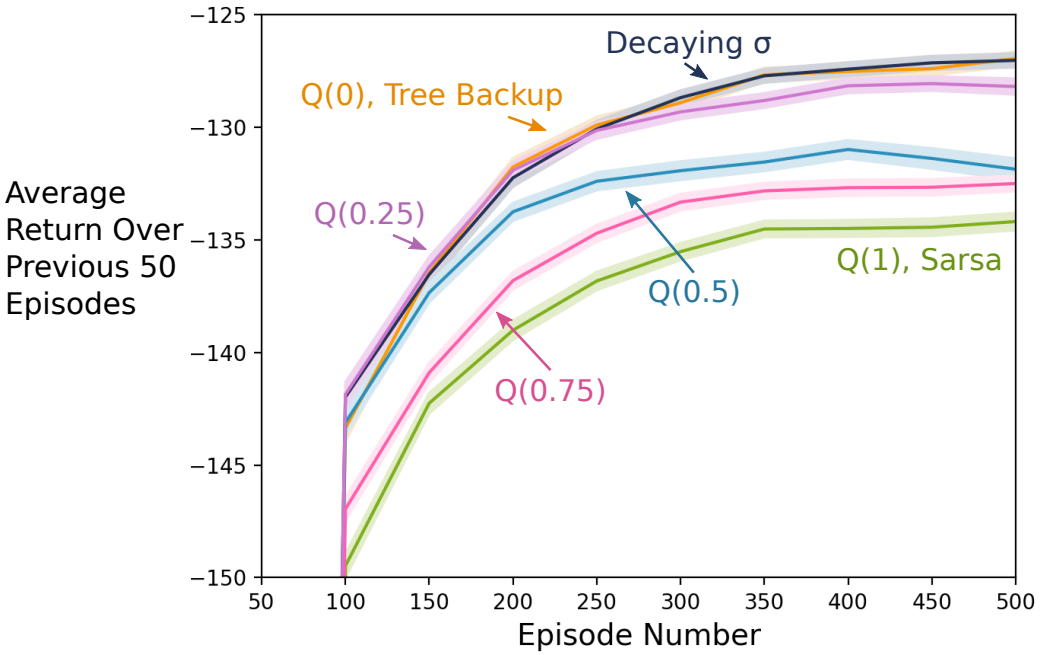


Figure 4.2: The Results of the Mountain Cliff Experiment. The plot shows the average return over the preceding 50 episodes. The shaded area corresponds to a 95% confidence interval. These results correspond to the average over 1,500 independent runs. Decaying σ and Tree Back up had very similar performance under this performance measure. As σ increases, the final performance of the algorithm decreased, which is reminiscent to the 19-state random walk results.

Chapter 5

n-Step $Q(\sigma)$ with Non-Linear Function Approximation

In the previous chapters we introduced n-step $Q(\sigma)$ and its extension to the linear function approximation case. In this chapter we present an extension of n-step $Q(\sigma)$ to the non-linear function approximation case using neural networks. This field of study is also known as *deep reinforcement learning* and has become increasingly relevant over the previous years.

The extension presented in this chapter is relevant to current deep reinforcement learning research because recent state of the art architectures have started adopting multi-step methods. Massively parallel architectures such as Rainbow (Hessel et al., 2017), Reactor (Gruslys, Azar, Bellemare, & Munos, 2017), Impala (Espeholt et al., 2018), and Ape-X (Horgan et al., 2018) use multi-step and n-step versions of the algorithms Retrace(λ) or Q-learning. However, other n-step algorithms remain to be studied in this setting.

We start this chapter by introducing an extension to the DQN architecture (Mnih et al., 2015) to use the off-policy n-step $Q(\sigma)$ return. We have named the resulting architecture the $Q(\sigma)$ network. Then, we provide empirical evaluations of the $Q(\sigma)$ network in the mountain car environment. Our experiments show results that are reminiscent of the ones found in the tabular and linear function approximation case.

5.1 The $Q(\sigma)$ Network

In essence, the DQN architecture is an artificial neural network that takes as input a state from the environment and outputs the action-value estimate of each action. However, there are two algorithmic details from the original DQN architecture that we need to consider and modify in order to implement the $Q(\sigma)$ network architecture. We will go through each of these implementation details in-depth.

5.1.1 The Experience Replay Buffer

The original DQN agent does not learn from the observations sampled directly from the environment. Instead, transitions of the form $(S_k, A_k, R_{k+1}, \mathbb{I}_k)$ are stored in a circular buffer, where k is the time when the transition is collected from the environment and $\mathbb{I}_k$ is a boolean variable indicating if S_k is a terminal state. Then, a mini-batch of transitions is sampled from the buffer in order to perform a stochastic gradient descent step on the weights of the network.

The motivation behind experience replay is to improve the sample efficiency of the learning algorithm and to prevent the agent from forgetting what it has already learned (Lin, 1992). Additionally, using mini-batches of observations instead of a single observation provides a less noisy estimate of the gradient, which induces some stability in the network.

In order to adapt the experience replay buffer to work with the n-step $Q(\sigma)$ algorithm we can simply store the σ_k corresponding to each transition. Then, we would have to sample $n + 1$ transitions in order to compute the n-step estimate of the return. Hence, at every time step we would perform a stochastic gradient descent step on a minibatch of m different trajectories, each trajectory consisting of $n+1$ different transitions. This would be enough if the target policy π was fixed during learning. However, if we allow π_t to change over time, as in the case of ϵ-greedy policies, then there is another issue that we need to address.

When we allow the target policy to change over time, observations are induced by the policy π_k — the policy used at the time the observation was

collected from the environment and stored in the buffer. However, when we sample from the buffer at time t, for $t \geq k$, there is no guarantee that the policies π_t and π_k will be the same. This is not an issue for DQN since the Q-learning algorithm is already an off-policy method and works even if there is a mismatch in the policies. However, for the n-step $Q(\sigma)$ algorithm we need to correct for this difference in policies, otherwise our estimate of the return will be biased. Consequently, along with each transition we will also store $\pi_k(A_k|S_k)$ to be able to compute the importance sampling ratio $\frac{\pi_t(A_k|S_k)}{\pi_k(A_k|S_k)}$, where k is the time when the observation is stored and t is the time when the observation is sampled from the buffer. Then, using this importance sampling ratio, we can correct for the difference in policies and obtain a less biased estimate of the return — note that the estimate is still biased since it uses $\hat{q}$ as an approximation of q_π.

Another similar issue arises as a byproduct of using the experience replay buffer. Since the buffer needs to have a certain amount of observations before training can start, a number of random actions are executed in order to populate the buffer. Consequently, to compensate for this second mismatch in the policies we can store $\pi_k(A_k|S_k) = \frac{1}{|\mathcal{A}|}$ in order to compute the importance sampling ratio for the initial random actions.

It is well known that importance sampling adds variance to the estimate of the return. As a consequence, the decrease in bias might not be able to compensate for the increase in variance introduced by the importance sampling ratio. In such cases, we would be better off not using importance sampling and consider the observations sampled from the buffer as if they had been collected using the current policy π_t. We will test this hypothesis empirically in the next section.

These are all the modifications that we need to employ to adapt the experience replay buffer to work with the n-step $Q(\sigma)$ algorithm. In total, this algorithmic detail adds three more hyper-parameters to our algorithm. First, the size of the experience replay buffer, also known as the replay memory, which is the maximum number of observations that can be stored in the buffer. Second, the replay start size, which is the number of random actions executed before

training in order to populate the buffer. Lastly, the mini-batch size, which is the number of trajectories sampled from the buffer in order to perform an training step.

5.1.2 The Target Network and Loss Function

DQN agents store two copies of its neural network architecture. The first copy, which we call the update network, is updated at every training step and is used to select the actions executed by the agent. The second one, called the target network, is updated less frequently by copying the weights from the update network into the target network every fixed number of training steps. The target network is exclusively used for computing the estimate of the expected return of Q-learning.

The motivation for the target network is reminiscent of the motivation for the experience replay buffer: minimize the interference that new observations have in the learning done so far. Since the representation mechanism of the network is global, changing the value of the weights based on one region of the state-space can have arbitrary effects in another different region of the state-space. Hence, having the network change at every step can interfere with the learning done so far.

Since the Q(σ) network architecture can also suffer from such learning interference, it also requires a target network. No modification is required in order to adapt the target network for the n-step $Q(\sigma)$ algorithm. However, we need to adapt the loss function to use the n-step $Q(\sigma)$ estimate of the expected return.

The loss function used by the original DQN architecture is

$$L(\boldsymbol{\theta}_t) \doteq \left(R_{k+1} + \gamma \max_a \hat{q}(S_{k+1}, a, \boldsymbol{\theta}_t^-) - \hat{q}(S_k, A_k, \boldsymbol{\theta}_t)\right)^2, \tag{5.1}$$

where $\boldsymbol{\theta}_t^-$ is the set of parameters from the target network, t is the time step at which the update is happening, and k is an index from the experience replay buffer.

The expression inside of the squared brackets in Equation 5.1 is the TD-error of the Q-learning algorithm. Adapting this loss function for n-step $Q(\sigma)$

only implies changing the estimate of the return used in the TD-error. The resulting loss function is

$$L(\boldsymbol{\theta}_t) \doteq \left(\hat{G}^{\sigma}_{k:k+n}(\boldsymbol{\theta}_t^-) - \hat{q}(S_k, A_k, \boldsymbol{\theta}_t)\right)^2, \tag{5.2}$$

where $\hat{G}_{k:k+n}(\boldsymbol{\theta}_t^-)$ is the n-step estimate of the expected return of the off-policy n-step $Q(\sigma)$ algorithm from Equation 3.6 evaluated using $\hat{q}(\cdot, \cdot, \boldsymbol{\theta}_t^-)$.

Overall, these modifications add only one hyper-parameter to our algorithm: the target-network update frequency. This concludes all the modifications that we implemented in order to adapt the DQN architecture to work with n-step $Q(\sigma)$.

5.1.3 Other Algorithmic Details and Hyper-Parameters

There is still one implementation detail that the $Q(\sigma)$ network architecture could inherit from DQN: annealing epsilon.

Just like any other reinforcement learning algorithm, the DQN architecture can suffer from a slow learning rate when the state-space is too large due to a poor exploration policy. In order to alleviate this issue, the behaviour policy of the DQN architecture is initialized as an ϵ-greedy policy with ϵ equal to one. Then, over a sequence of time steps, the ϵ of the behaviour policy is annealed linearly towards 0.1.

It is possible to adapt this algorithmic detail to the $Q(\sigma)$ network. In order to do so, one has to store $\mu_k(A_k|S_k)$ in the experience replay buffer to compute the importance sampling ratio $\frac{\pi_t(A_k|S_k)}{\mu_k(A_k|S_k)}$ and correct for the difference in the policies. Note that, once again, k stands for the time when the observation was stored and t denotes the time when the observation was sampled from the buffer. This introduces three more hyper-parameters to the DQN architecture: the initial exploration parameter, the initial value of the parameter ϵ; the final exploration parameter, the final value of ϵ; and the exploration frame, the number of time steps before the initial ϵ converges to the final ϵ. It is important to emphasize that ϵ starts decreasing only after the experience replay buffer has been populated using the initial random actions.

Since we will be testing the $Q(\sigma)$ network architecture in a simple environment where exploration is not often an issue, we will not implement this particular algorithmic detail in our evaluations. Nevertheless, it could be easily added to the $Q(\sigma)$ network architecture.

These are all the algorithmic details that the $Q(\sigma)$ network architecture inherits from DQN. However, We still need to mention several of the hyperparameters and implementation details that the $Q(\sigma)$ network inherits by virtue of being a neural network.

1. **The architecture:** defined by the number of layers, the number of neurons per layer, and the gate function use at each layer. Moreover, any regularization done at every layer such as max pooling, dropout, batch normalization, or other methods (Goodfellow et al., 2016) are also part of the architecture. Nevertheless, we will not use any of such methods in our architecture.

2. **The optimization method:** artificial neural networks are often trained using stochastic gradient descent (SGD). Many modifications have been proposed to the original SGD algorithm, each with its own set of hyperparameters. We will exclusively use the RMSprop optimizer (Tieleman & Hinton, 2012) with gradient momentum of 0.95, squared gradient momentum of 0.95, and minimum squared gradient of 0.01, as in the original DQN paper (Mnih et al., 2015).

3. **The learning rate:** this is simply the learning rate parameter α used by the optimization algorithm and is analogous to the learning rate parameter used in all of the update functions introduced so far.

4. **Weight initialization method:** it is rarely mentioned in the deep reinforcement learning literature, nevertheless, the initialization procedure is often crucial for implementing neural networks (Goodfellow et al., 2016). We will use a variant of Xavier initialization (Glorot & Bengio, 2010) where the weights of each layer are initialized following a normal distribution with zero mean and variance of one over the number of

neurons in the previous layer.

The pseudocode for the $Q(\sigma)$ Network can be seen in the algorithm box below. Note that in the pseudocode the mini-batch size is one and the replay start size is zero. We chose such values in order to facilitate the exposition of the main algorithm. Nevertheless, implementing the algorithm with different parameter values is straightforward.

Algorithm 3 $Q(\sigma)$ Network

Initialize $\boldsymbol{\theta}$ and store a copy $\boldsymbol{\theta}^{-}$ corresponding to the target network
Initialize the target policy π as a function of $\hat{q}(\cdot, \cdot; \boldsymbol{\theta})$
Assume the experience replay buffer D is already populated
Algorithm Parameters: learning rate α, backup length n, target network update frequency C
Update Count $\leftarrow 0$
for Every Episode **do**
  $T \leftarrow \infty$ (The time of termination)
  $t \leftarrow 0$ (The current time step)
  Initialize S_0 and select $A_0 \sim \pi(\cdot|S_0)$
  while $t < T$ **do**
    Take action A_t; observe R_{t+1} and S_{t+1}
    Store transition $(S_t, A_t, R_{t+1}, \mathbb{I}_t, \pi_t(A_t|S_t), \sigma_t)$ in D
    ($\pi_t(A_t|S_t)$ is the probability at the time the observation was stored)
    if S_{t+1} is terminal **then**
      $T \leftarrow t + 1$
    else
      Choose $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    end if
    Sample uniformly at random $n + 1$ consecutive observations from D
    $l \leftarrow$ Buffer index of the first observation
    $\hat{G} \leftarrow \hat{q}(S_{l+n}, A_{l+n}, \boldsymbol{\theta}^{-})$
    for $k = l + n, l + n - 1, ..., l + 1$ **do**
      $V \leftarrow \sum_{a \neq A_k} \pi(a|S_k)\hat{q}(S_k, A_k, \boldsymbol{\theta}^{-})$
      $\rho \leftarrow \frac{\pi(A_k|S_k)}{\pi_k(A_k|S_k)}$
      $\hat{G} \leftarrow R_k + (1 - \mathbb{I}_k)\gamma\Big[\big(\sigma_k\rho + (1 - \sigma_k)\pi(A_k|S_k)\big)\hat{G} + (1 - \sigma_k)V\Big]$
    end for
    Perform gradient descent step on $\big(\hat{G} - \hat{q}(S_l, A_l, \boldsymbol{\theta})\big)^2$ w.r.t. $\boldsymbol{\theta}$
    Update Count $\leftarrow$ Update Count + 1
    if Update Count is a multiple of C **then**
      $\boldsymbol{\theta}^{-} \leftarrow \boldsymbol{\theta}$
    end if
    $t \leftarrow t + 1$
  end while
end for

5.2 Empirical Evaluations of the Q(σ) Network

Our goal in this section is to investigate if the performance of the $Q(\sigma)$ network shows similar effects as the ones found in the tabular and linear function approximation cases. We formulated two main hypotheses based on the previous results: (1) the decaying σ algorithm would perform better than algorithms with fixed σ, and (2) large values of σ would result in better initial performance whereas small values would result in better final performance.

Before studying the two main hypotheses of this section we studied the effects that some of algorithmic details of the $Q(\sigma)$ network have on the performance of an agent. We divide this study into two sets of experiments: (1) experiments studying the effects that algorithmic details from the DQN architecture have on the performance of the $Q(\sigma)$ network, and (2) experiments studying the effects that the parameters σ and n have on the performance of the $Q(\sigma)$ network. Then, we go back to study the main hypotheses of this section.

To run all these experiments we set up the learning problem as a control task in the mountain car environment. The mountain car environment is set up as described in Chapter 4. We decided against using the mountain cliff environment because it added extra noise to the results and took longer to learn.

We imposed a timeout of 5,000 steps in order to prevent episodes from running forever and overpopulating the experience replay buffer with observations that are too similar to each other. It is important to note that timing out is not treated the same way as termination. In the case of termination, all the subsequent rewards and action-value functions after the terminating state are considered zero. In the case of a timeout, the return is computed by bootstrapping off of the last available action-value functions, effectively making the n-step estimate of the return shorter. We found in preliminary experiments that treating time out the same way as termination had catastrophic effects on the performance of the agents. All the agents were trained for 500 episodes.

Hyperparameter	Value	Description
Step-size (α)	0.00025	Step-size used with RMSprop
Gradient momentum	0.95	Gradient momentum used by RMSprop
Squared gradient momentum	0.95	Squared gradient (denominator) momentum used by the RMSprop
Min squared gradient	0.01	Connstant added to the denominator of the RMSprop update
Discount factor (γ)	1.0	Discount factor used in the n-step $Q(\sigma)$ update
Exploration rate (ϵ)	0.1	Probability that a random action will be taken at each time step
Minibatch size	32	Number of samples of the estimate of the return used to compute an update
Replay memory	20,000	Number of observations stored in the experience replay buffer
Replay start size	1,000	A random policy is ran for this many time steps to populate the buffer
Target network update frequency*	1,000	Frequency — measured in number of updates — with which the target network is updated

Table 5.1: The hyperparameters for the $Q(\sigma)$ network architecture. All the parameters remained fixed throughout this set of experiment except for the ones marked with an asterisk (*).

We implemented the $Q(\sigma)$ network architecture as described in the previous section. All the agents in this set of experiments used the same network architecture consisting of an input layer, one hidden layer, and an output layer. The input layer of the network consists of a state tuple containing the position and the velocity of the car at a given time step. This input is passed to a fully-connected hidden layer that consists of 1,000 rectified linear units. The output of the hidden layer is fed into a fully connected linear layer that computes an estimate of the action-value function corresponding to each action. The total number of parameters in the network is 6,003.

The hyper-parameters used by the $Q(\sigma)$ network are listed in Table 5.1. Most of the parameters for the $Q(\sigma)$ network architecture are the same as the ones used in the original DQN architecture. We used the RMSprop optimizer to minimize the loss function as in the original DQN architecture (Mnih et al.,

2015; Tieleman & Hinton, 2012). The discount factor γ was chosen to be one so that agents considered the full sequence of rewards in order to encourage them to find the terminal state before episodes timed out. The replay memory size was selected by testing the performance of a DQN agent with a replay memory size of 100, 50, 20, and 5 thousand observations; this was not an exhaustive search. The target network update frequency and the replay start size were chosen to be 5% of the size of the replay memory buffer. All the hyper-parameters of the network remained fixed for all the experiments except for the target network update frequency. We will make special emphasis when using a different value for the target network update frequency; otherwise, the reader should assume that the value of this hyper-parameter is 1,000.

The performance measure that we used in this task was the average return per episode. We studied the initial performance by computing this measure over the first 50 episodes of training, whereas the final performance is computed in terms of the last 50 episodes of training. The overall performance of the algorithm is computed using the whole training period: 500 episodes.

5.2.1 Off-Policy vs On-Policy Sampling Experiment

We start by studying how sampling from the experience replay buffer affects the performance of the $Q(\sigma)$ network. When sampling from the buffer one can correct for the difference between policy π_k, the policy used at the time a transition was stored in the buffer, and π_t, the policy used at the time a transition is sampled from the buffer. This is equivalent to computing the importance sampling ratio $\frac{\pi_t(A_k|S_k)}{\pi_k(A_k|S_k)}$ in order to correct the sampling side of the $Q(\sigma)$ estimate of the return. The expectation side of the estimate of the return does not have to be corrected since it already works off-policy. Alternatively, one can choose to ignore this difference and hope that the bias of the return will not have a big effect on the performance of the algorithm. We tested both solutions to the sampling problem. Note that one solution corresponds to treating transitions as if they had been collected off-policy, while the second solution corresponds to considering transitions as if they were sampled on-policy. Hence, we will refer to this sampling solutions as sampling off-policy

vs on-policy.

The main goal of this experiment is to test if there is any difference in performance between these two sampling methods. We hypothesized that sampling on-policy would perform better since the variance introduced by the importance sampling ratio would overcome the decrease in bias.

To test this hypothesis, we implemented an on-policy and an off-policy version of three different one-step $Q(\sigma)$ algorithms: $Q(1)$ (Sarsa), $Q(0.5)$, and a decaying σ agent with decay rate of 0.95. We did not test $Q(0)$ (Expected Sarsa) since it works off-policy without using importance sampling.

$Q(1)$ and $Q(0.5)$ had better final and overall performance when sampling on-policy. Across all the algorithms, sampling off-policy resulted in better initial performance. The decaying σ algorithm performed better when sampling off-policy in terms of initial, final, and overall performance. The results of the experiment averaged over 100 independent runs with 95% confidence intervals can be seen in Figure 5.1.

The results of this experiment partially support our hypothesis. For $Q(1)$ and $Q(0.5)$ sampling off-policy resulted in worse performance. Additionally, the performance of these two algorithms consistently had higher variance, as evidenced by the error bars of the plot. This supports the claim that the importance sampling ratio added extra variance to the estimate of the expected return.

In the case of decaying σ we observed the opposite effect: the off-policy algorithm performed considerably better. This could be explained by the high decay rate of the algorithm. Initially, the estimate of the expected return has inherently high variance due to random initialization; hence, adding more variance while reducing the bias of the estimate results in better performance. However, by episode 51, the value of σ is approximately 0.08 meaning that the sampling side of the estimate — the one using importance sampling — has a very small weight assigned to it. Consequently, the performance of the decaying σ algorithm is not affected by the increase in variance induced by the importance sampling ratio.

This second conclusion has an important implication that could be ex-

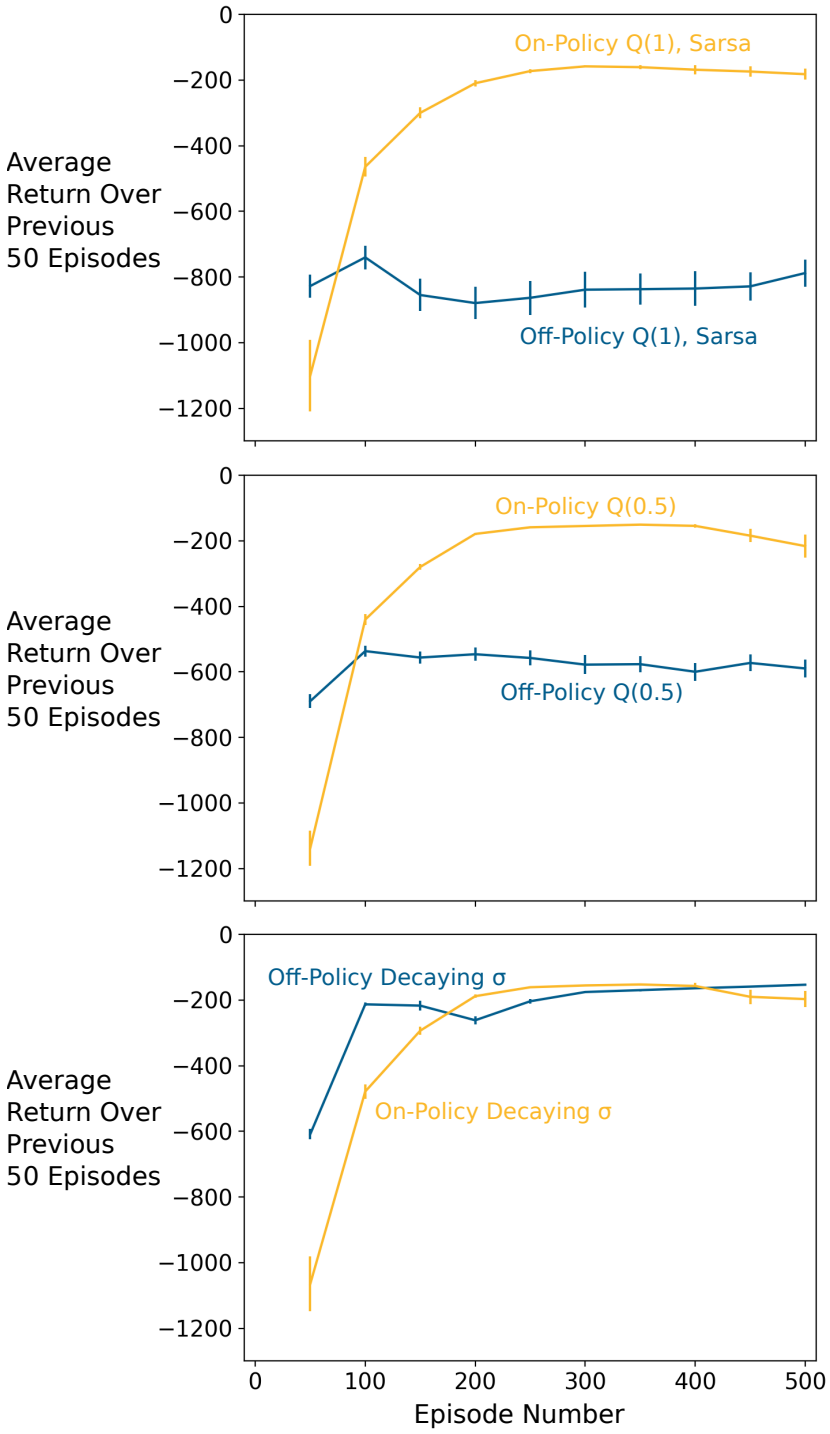


Figure 5.1: Results of the off-policy vs on-policy sampling experiment. The results are averaged over 100 independent runs. The error bars correspond to a 95% confidence interval. Sarsa and $Q(0.5)$ (top and middle plots) have better performance when not using the importanc sampling ratio. On the other hand, dynamic σ (bottom plot) benefited from using importance sampling ratio.

ploited when using n-step $Q(\sigma)$ in the off-policy setting. Analogous to the parameter λ in the ABQ(ζ) algorithm (Mahmood, Yu, & Sutton, 2017), the parameter σ could be used in order to counteract the increase in variance induced by the importance sampling ratio.

Henceforth, we use the on-policy sampling version of all the algorithms that we test since it results in better performance for most of them.

5.2.2 Effect of the Parameter n on the Performance of the $Q(\sigma)$ Network

We move on to study how the parameter n affects the performance of the $Q(\sigma)$ network across several settings of the parameter σ. Given the previous results in the mountain cliff and stochastic windy gridworld environments, we hypothesized that values of $n \geq 1$ would result in better performance, but using too high of a value for n would have an adverse effect in performance. In order to test this hypothesis, we implemented for each value of n in $\{1, 3, 5, 10, 20\}$ four different $Q(\sigma)$ algorithms: $Q(1)$ (Sarsa), $Q(0.5)$, $Q(0)$ (Tree Backup), and decaying σ with decay rate of 0.95.

In terms of initial performance, we found across all the different $Q(\sigma)$ algorithms that increasing the value of the parameter n consistently improved performance. All the algorithms performed the best in terms of initial performance with $n = 20$.

In terms of final performance, we found that there was no statistically significant difference in the performance of $Q(1)$ with $n = 3, 5, 10, 20$ and they all performance better than $Q(1)$ with $n = 1$. For $Q(0.5)$, there was no statistical significance between the final performance of the algorithm with $n =$ 10 and 20, and both of these settings outperformed $Q(0.5)$ with $n = 1, 3$ and 5. $Q(0)$ performed the best with $n = 20$ in terms of final performance. Lastly there was no statistically significant difference between the final performance for decaying σ with $n \geq 5$, and all of those setting performed better than decaying σ with $n = 1, 3$.

Finally, in terms of overall performance, $Q(1)$, $Q(0.5)$, and $Q(0)$ performed the best with $n = 20$. Decaying σ performed the best in terms of overall

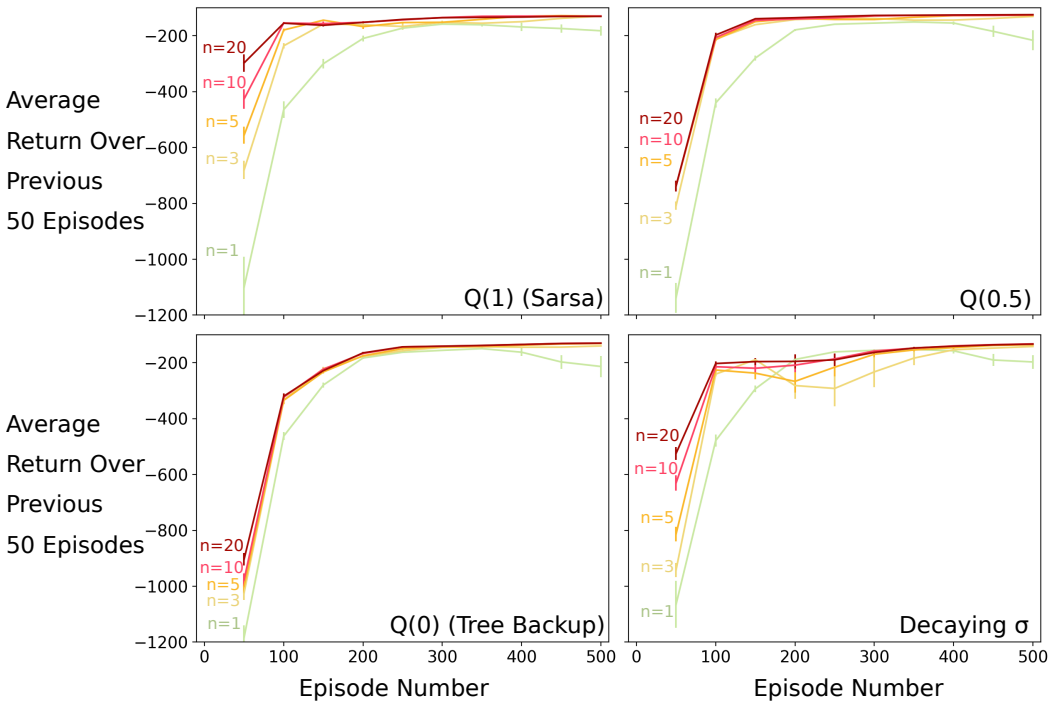


Figure 5.2: Performance of the $Q(\sigma)$ network for different values of n and σ. The plots show the average over 100 independent runs with 95% confidence intervals. All the algorithms benefit from having a larger back up length than one. The decaying σ algorithm had less stable performance as evidenced by its high variance.

performance with $n \geq 10$ with no statistically significant difference between $n = 10$ and 20. Figure 5.2 shows the results corresponding to the average over 100 independent runs. The error bars correspond to a 95% confidence interval.

These results partially support our hypothesis: values of $n > 1$ performed better across all the algorithms, but we could not find a value of n greater than one that had an adverse effect on the performance of the algorithm. Even though the initial performance for all the different $Q(\sigma)$ agents improved as n increased, the magnitude of this effect was bigger the closer σ was to one.

Most of the algorithms did not show a high variance or instability in their performance across different stages of training. However, there were two special cases that stood out: one-step algorithms and decaying σ. For all the one-step methods there was a dip in performance during the last one hundred episodes of training. This did not happened for algorithms with n greater than

one. However, it is not possible to determine from these results whether this dip in performance was completely avoided or whether it was just delayed to a time frame greater than the one we used.

In the case of decaying σ the outstanding pattern is the drop in performance and increase in variance during the middle of training. This happened with a bigger magnitude for smaller values of n than higher values. Moreover, this pattern happened across all values of n except for $n = 1$. The pattern observed in the performance of the decaying σ algorithm motivated the next two experiments.

5.2.3 Effect of the Decay Rate on the Performance of the $Q(\sigma)$ Network

One possible explanation for the drop in performance experienced by the decaying σ algorithm is that the loss function of the neural network is changing too fast making learning unstable. This would also explain why at the end of training the performance stabilizes again. Close to the end, the value of σ is close to 0 and the changes in the value of σ are very small; consequently, the loss function barely changes episode by episode. We hypothesized that if the decay rate was slower, then the drop in performance experienced by the decaying σ algorithm would be smaller.

To test this hypothesis we implemented a second decaying σ algorithm with a linear decay rate instead of an exponential decay rate for each value of n in $\{3, 5, 10, 20\}$. In this case, instead of multiplying σ by a factor of 0.95 at the end of each episode, we subtracted 0.002 from the value of σ after every episode. We compared the performance of the linear and exponential decays and measured the severity of the drop in performance.

In order to gain a more detailed view of the performance of the algorithms in this experiment, we used a different time scale when plotting and analyzing the data. We used the average return per episode as measure of performance; however, instead of looking at intervals of 50 episodes, we decreased the size of the intervals to 10 episodes.

In order to measure the severity of the drop in performance, we first looked

Table 5.2: Comparison of the drop in performance experienced by the decaying σ algorithm with linear decay vs exponential decay for $n = 3$ and 5. The severity of the drop in performance is measured in terms of the average difference between the performance at the start and at the lowest point of the drop within each run. Lower (LB) and upper (UB) 95% confidence interval bounds are provided to validate the results. Decaying σ with linear decay had a less severe drop in performance than with exponential decay.

	Average Drop in Performance			
$n = 3$	Mean	Standard Error	LB	UB
Linear Decay	62.49	8.45	45.9	79.07
Exponential Decay	114.55	10.87	93.21	135.89

	Start	Lowest Point
Linear Decay	Episodes 151 - 160	Episodes 221 - 230
Exponential Decay	Episodes 111 - 120	Episodes 181 - 190

	Average Drop in Performance			
$n = 5$	Mean	Standard Error	LB	UB
Linear Decay	38.37	5.47	27.63	49.12
Exponential Decay	72.64	7.73	57.48	87.81

	Start	Lowest Point
Linear Decay	Episodes 121 - 130	Episodes 181 - 190
Exponential Decay	Episodes 109 - 110	Episodes 161 - 170

at the 10 episode interval before the start of the drop — the start — and the 10 episode interval with the lowest performance during the drop — the lowest point. Then, within each run, we computed the difference in performance between the start of the drop and the lowest point of the drop and computed the average difference along with a 95% confidence interval using a t-distribution. Statistical analysis on the average difference computed in this way is equivalent to a paired t-test where the variable under study is measured for each subject before and after certain treatment or event; the subjects in our experiment would be each individual run. Because of the high variability of the estimates we had to increase the sample size from 100 to 1,200 in order to

Table 5.3: Comparison of the drop in performance experienced by the decaying σ algorithm with linear decay vs exponential decay for $n = 10$ and 20. The severity of the drop in performance is measured in terms of the average difference between the performance at the start and at the lowest point of the drop within each run. Lower (LB) and upper (UB) 95% confidence interval bounds are provided to validate the results. Decaying σ with linear decay had a less severe drop in performance than with exponential decay.

	Average Drop in Performance			
$n = 10$	Mean	Standard Error	LB	UB
Linear Decay	24.42	3.01	18.52	30.32
Exponential Decay	43.46	5.32	33.01	53.9

	Start	Lowest Point
Linear Decay	Episodes 101 - 110	Episodes 201 - 210
Exponential Decay	Episodes 91 - 100	Episodes 121 - 130

	Average Drop in Performance			
$n = 20$	Mean	Standard Error	LB	UB
Linear Decay	9.81	3.06	3.8	15.81
Exponential Decay	46.65	4.83	37.19	56.12

	Start	Lowest Point
Linear Decay	Episodes 91 - 100	Episodes 201 - 210
Exponential Decay	Episodes 101 - 110	Episodes 131 - 140

obtain significant results with 95% confidence.

Our results show that all the decaying σ algorithm with linear decay had a smaller drop in performance in terms of the difference between the performance at the start and at the lowest point of the drop. For every value of n, decaying σ with linear decay performed better than with exponential decay in terms of average return per episode over 500 episodes. In the case of decaying σ with linear decay, using higher values of n consistently resulted in a less severe drop in performance. The summary of these results averaged over 1,200 independent runs and with corresponding 95% confidence intervals can be seen in Tables 5.2 and 5.3. Figure 5.3 shows the training performance of the algorithms.

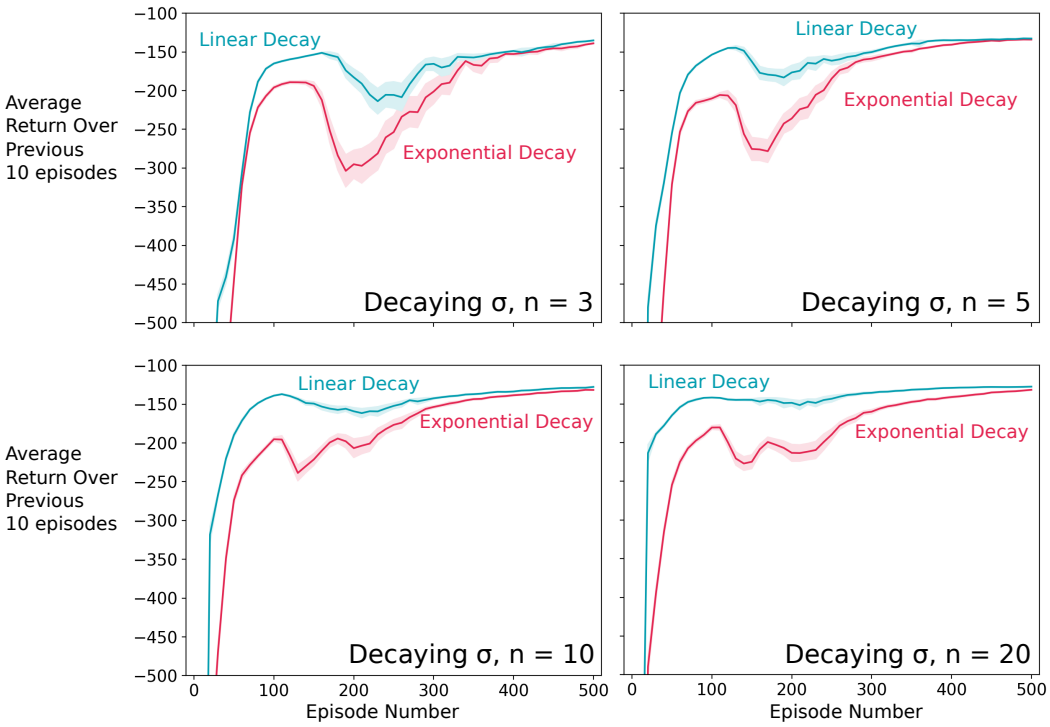


Figure 5.3: Performance of decaying σ with linear and exponential for several values of n. The results correspond to the average of 1,200 independent runs. The error bars correspond to a 95% confidence interval. Using a linear decay decreased the size of the drop in performance. Decaying σ performed better with linear decay than with exponential decay.

The results support our main hypothesis, using a slower decay rate resulted in a smaller drop in performance for decaying σ. It is possible that the cause of the drop in performance is more complex than we previously thought. The severity of the drop in performance seems to be the result of an interaction between the back up length and the decay rate. For instance, for $n = 3$ the drop in performance was 2 times smaller when using linear decay than when using exponential decay. In contrast, for $n = 20$ and linear decay the drop in performance was 4 times smaller than for the exponential decay. Thus, it seems that the parameter n amplifies the effect of the decay rate on the drop in performance.

5.2.4 Effect of the Target Network on the Performance of the $Q(\sigma)$ Network

There is a possibility that there are more factors influencing the drop in performance experienced by the decaying σ algorithm. A possible candidate is the update frequency of the target network. During training and in between updates to the target network, the update network — the network that is updated at every time step — observes and learns from values of σ that the target network has not seen before. Hence, it is possible that the mismatch in learning between the two networks is inducing some instability in the performance of the decaying σ algorithm. We hypothesized that lower update frequencies for the target network would result in more stable performance, whereas increasing the update frequency would result in less stable performance.

To test this hypothesis we implemented decaying σ agents with linear and exponential decay with different target network update frequencies taking values in $\{500, 1000, 2000\}$ and for backup lengths of 3 and 20. This is the only experiment so far that uses a value different from 1,000 for the target network update frequency.

As a measure of instability in the performance of the algorithm we used the sample standard deviation of the average return per episode over 500 episodes — the entire training period. We computed 95% confidence intervals for the sample standard deviation using a chi-squared distribution. In this case, the assumption of the statistical test is that the samples are normally distributed, which is a reasonable assumption in this case since the sample average is normally distributed in the limit.

We found that for decaying σ with linear decay, decreasing the the target network update frequency from 2,000 time steps to 500 steadily reduced the instability of the algorithm. On the other hand, for decaying σ with exponential decay and $n = 3$, the instability of the algorithm was the highest with a target network update frequency of 1,000, the second highest with a frequency of 2,000, and the lowest with a frequency of 500. In the case of decaying σ with exponential decay and $n = 20$ there was no difference in the stability of

Table 5.4: Comparison of the sample standard deviation of the average return per episode over 500 episodes for decaying σ algorithms with different values of the target network update frequency parameter. The results were computed using 100 independent runs. Lower (LB) and upper (UB) 95% confidence interval bounds computed using a chi-squared distribution are provided to validate the results. In the linear decay case, lower frequencies result in lower variance for both values of n.

Decaying σ with Linear Decay				
		Average Return per Episode		
n	Update Frequency	Standard Deviation	LB	UB
3	500	21.9	19.23	25.44
	1,000	52.24	45.87	60.69
	2,000	106.27	93.30	123.45
20	500	19.63	17.24	22.81
	1,000	40.48	35.54	47.02
	2,000	63.38	55.65	73.63

Decaying σ with Exponential Decay				
		Average Return per Episode		
n	Update Frequency	Standard Deviation	LB	UB
3	500	36.12	31.71	41.96
	1,000	85.16	74.77	98.93
	2,000	51.96	45.63	60.37
20	500	58.10	51.01	67.49
	1,000	43.75	38.41	50.82
	2,000	54.51	47.86	63.32

the algorithms for any of the update frequencies. The summary of the results averaged over 100 runs are listed in table 5.4.

Figure 5.4 shows the performance of each algorithm in terms of average return per episode at intervals of 10 episodes. The shaded regions correspond to a 95% confidence interval computed using a t-distribution. For the linear decay algorithms, as the target network update frequency increases, the variability of the performance of the algorithm increases. In the case of the exponential decay, there is no discernible pattern that explains the performance of both values of n.

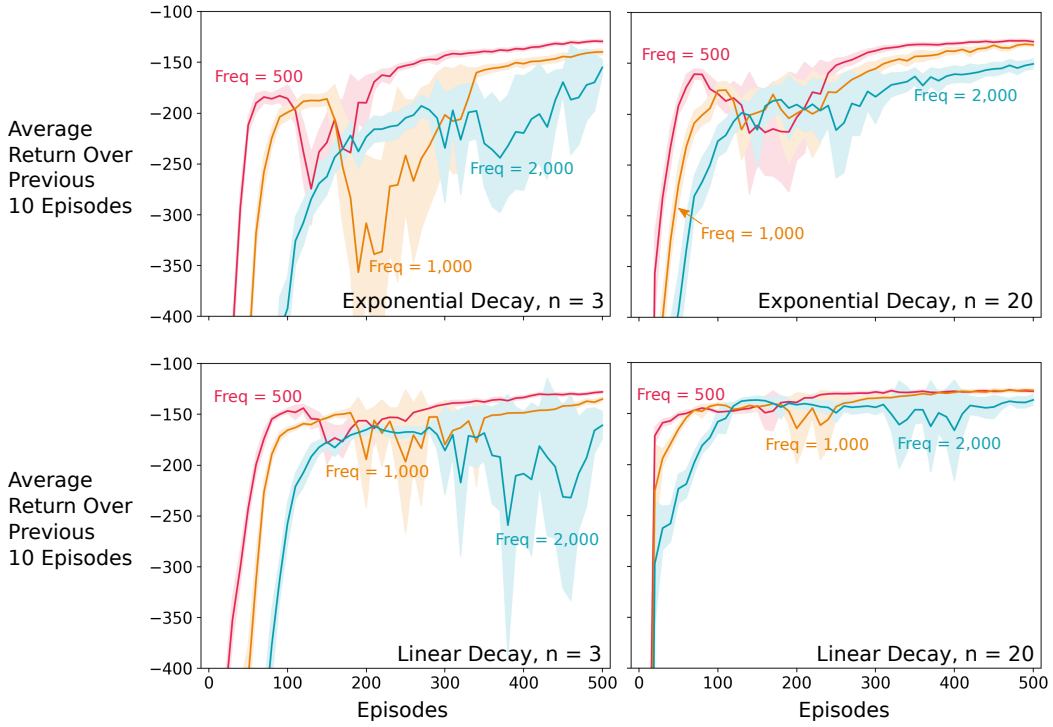


Figure 5.4: Performance of decaying σ with different target network update frequencies. The plot shows the average over 100 independent runs with error bars corresponding to a 95% confidence interval. Smaller update frequencies consistently caused the drop in performance to occur sooner during training.

The results obtained with decaying σ with linear decay support our hypothesis; low target network update frequencies resulted in more stable performance that when using high frequencies. However, the results observed with the exponential decay do not show this effect; there was no statistical significant difference in the stability of the performance of the algorithms when using low or high frequencies. These results, together with the results from the previous experiment, seem to indicate that the drop in performance of the decaying σ algorithm is the result of the interaction between the backup length, the decay rate, and the target network update frequency. The backup length and the decay rate seem to influence the size of the drop in performance, and all three hyper parameters seem to influence the stability of the performance of the algorithm.

5.2.5 Effect of σ on the Performance of the $Q(\sigma)$ Network

Now that we have studied how the algorithmic details of DQN, the parameter n, and the decay rate affect the performance of the $Q(\sigma)$ network, we move on to study the main hypotheses of this section. Remember that our two main hypotheses were: (1) the decaying σ algorithm would perform better than algorithms with fixed σ, and (2) large values of σ would result in better initial performance whereas small values would result in better final performance.

In order to test these hypotheses, we trained four different agents: $Q(0)$, $Q(0.5)$, $Q(1)$, and a decaying σ algorithm with a linear decay of 0.002. The measure of performance that we used in this experiment was the average return per episode. To study the initial performance we computed this measure over the first 50 episodes of training, whereas for the final performance we used the last 50 episodes of training. To study the overall performance, we computed this measure over the entire training period — 500 episodes.

We only compared the 20-step versions of each algorithm since they consistently resulted in the best overall performance in the previous experiments. For each algorithm, we trained 100 agents for values of the target network update frequency of 500, 1,000, and 2,000, then we selected the value that resulted in the best overall performance. $Q(0)$, $Q(0.5)$ and decaying σ with linear decay performed the best with a target network update frequency of 500 time steps. For $Q(1)$, there was no statistically significant difference between the performances of any of the different update frequencies; nevertheless, we selected 1,000 since it had the highest average return. After finding the best target network frequency, we trained 200 more agents for each different algorithm to avoid incurring in maximization bias; all the results presented henceforth correspond to these 200 independent runs.

In terms of initial performance, decaying σ and $Q(1)$ performed the best — with no statistically significant difference between their performances. $Q(0)$ performed the worst in terms of initial performance. In terms of final performance, $Q(0.5)$ performed the best followed by decaying σ and $Q(0)$, and $Q(1)$

Table 5.5: Comparison of the best performance of the $Q(\sigma)$ network with different settings of σ. The standard error, and lower (LB) and upper (UB) 95% confidence interval bounds are provided to validate the results.

	Average Return of First 50 Episodes			
Algorithm	Mean	Standard Error	LB	UB
$Q(1)$, Sarsa	**-342.72**	16.53	**-375.31**	**-310.13**
$Q(0.5)$	-481.46	2.78	-486.94	-475.97
$Q(0)$, Tree-backup	-648.89	3.25	-655.31	-642.47
Decaying σ	**-290.27**	10.47	**-310.91**	**-269.63**

	Average Return of Last 50 Episodes			
Algorithm	Mean	Standard Error	LB	UB
$Q(1)$, Sarsa	-129.5	0.56	-130.61	-128.39
$Q(0.5)$	**-123.92**	0.22	**-124.35**	**-123.49**
$Q(0)$, Tree-backup	-127.13	0.26	-127.65	-126.61
Decaying σ	-128.57	0.36	-129.28	-127.87

	Average Return Over 500 Episodes			
Algorithm	Mean	Standard Error	LB	UB
$Q(1)$, Sarsa	-161.91	2.13	-166.12	-157.71
$Q(0.5)$	-164.04	0.29	-164.6	-163.47
$Q(0)$, Tree-backup	-193.26	0.27	-193.79	-192.74
Decaying σ	**-151.51**	1.7	**-154.86**	**-148.17**

performed the worst. Finally, considering the whole training period, decaying σ performed the best among all the algorithms followed by $Q(1)$ as a close second, then $Q(0.5)$, and at last $Q(0)$. Table 5.5 shows the summaries for the initial, final, and overall performance for all the algorithms. Figure 5.5 shows the performance during training at intervals of 50 episodes for each algorithm. The error bars correspond to a 95% confidence interval.

The results partially support our hypotheses. Decaying σ performed the best in terms of overall performance. Nevertheless, it tied with $Q(1)$ in terms of initial performance and it tied for second place with $Q(0)$ in terms of final performance. $Q(0.5)$ did not have the best performance during early training,

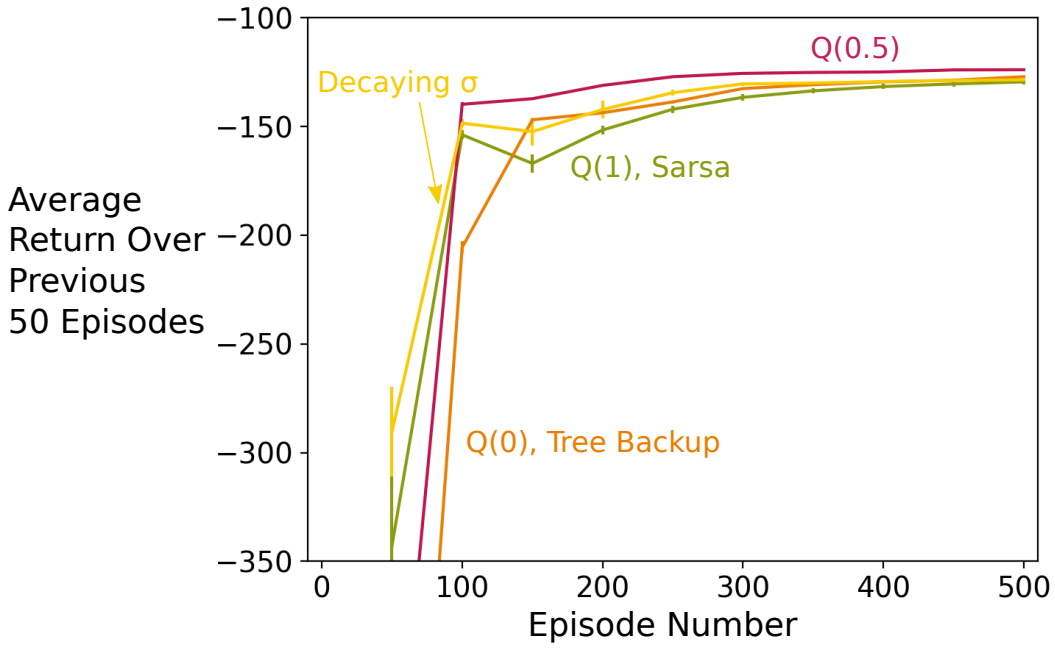


Figure 5.5: Comparison of the $Q(\sigma)$ network algorithms with the best performance for different values of σ. The plot shows the average over 200 independent runs with error bars corresponding to a 95% confidence interval. Sarsa and decaying σ with linear decay had the best initial performance among the four algorithms. $Q(0.5)$ had the best final performance followed closely by decaying σ with linear decay and tree backup.

however, after 100 episodes of training, it consistently had the best performance among all the algorithms. Hence, it seems possible to improve the final performance of decaying σ by letting the final value of σ be 0.5 instead of 0.

For the second hypothesis, increasing the value of σ consistently improved the initial performance. However, decreasing the value of σ did not consistently improve the final performance. The evidence indicates that the best final performance is attained at some intermediate value of σ possibly close to 0.5.

Lastly, one interesting observation from the optimization of these algorithms is that decreasing the target network update frequency improved the performance of $Q(0.5)$ and $Q(0)$. This implies that, similar to decaying σ, the target network update k update frequency can have an adverse effect on the performance of these algorithms. To see the full summary of the these algorithms with different target network update frequencies see appendix A.

5.3 Summary of the Empirical Evaluations

In this chapter and the preceding two chapters we presented several empirical evaluations of the n-step $Q(\sigma)$ algorithm in increasingly complex learning tasks. Now let us summarize our findings and results.

Across all these chapters, we have accumulated increasing evidence to support the claim that there is a benefit in using n-step $Q(\sigma)$ over the n-step algorithms Sarsa and Tree Backup.

We found that decaying σ — a version of n-step $Q(\sigma)$ were σ decays from 1 to 0 over time — often performed better in terms of overall performance than any of the n-step $Q(\sigma)$ algorithms with fixed values of σ. We observed this effect in four different environments with three different representations of the action-value function: 19-state random walk and stochastic windy grid-world, both with tabular representations; mountain cliff with linear function approximation; and mountain car with non-linear function approximation.

The parameter σ seems to influence the performance of the n-step $Q(\sigma)$ algorithm during early and late training. In the 19-state random walk environment, we found that increasing the value of σ from 0 to 1 improved the initial performance but worsen the final performance. This effect seems to be affected by the type of environment and the type of representation used for the action-value function.

In the mountain cliff environment with linear function approximation we found that the best initial performance was achieved with a value of σ of 0.5 and the initial performance worsened the further away σ was from this value. Nevertheless, we still found that as the value of σ decreased from 1 to 0 the resulting algorithms consistently performed better during late training.

In contrast, when using non-linear function approximation in the mountain car environment, we found that the best final performance was achieved with a value of σ of 0.5 and the final performance worsened the further away σ was from this value. Moreover, we found that as the value of σ increased from 0 to 1 the resulting algorithms performed better during early training.

The behaviour of n-step $Q(\sigma)$ algorithms, including $Q(1)$ (Sarsa) and $Q(0)$

(Tree Backup), changed significantly depending on the learning problem, the environment, and the type of representation used for the action-value function. Nevertheless, we were always able to adapt the n-step $Q(\sigma)$ algorithm in order to improve its performance in every single one of the tasks studied in this thesis. This would not have been possible if not for the parameter σ. Therefore, the main benefit of using n-step $Q(\sigma)$ is the flexibility that it provides and that allows it to be adaptable to the particular characteristics of the learning task.

Chapter 6

Conclusion

The main contribution of this work is the in-depth introduction of the n-step $Q(\sigma)$ algorithm, which was first proposed in Precup et al. (2000) and first formulated in Sutton and Barto (2018). Through the introduction of the parameter σ, we have gained the capability of representing a wide family of algorithms opening up another dimension in the space of algorithms that can be studied and implemented. Moreover, we have demonstrated how the n-step $Q(\sigma)$ algorithm can be used to represent the n-step algorithms Sarsa and Tree Backup, effectively unifying them under the same family of algorithms.

We also provided some theoretical intuition about the benefit of using n-step $Q(\sigma)$ over n-step Sarsa and Tree Backup. We showed that the parameter σ can be chosen in order to minimize the mean squared error of the estimate of the expected return. Even though we did not explore methods for selecting the parameter σ, we presented empirical evidence that validate our intuition about the influence of σ on the mean squared error.

The second and third contributions of this thesis are the extensions of n-step $Q(\sigma)$ to the linear and non-linear function approximation cases. In the linear function approximation case, we used n-step $Q(\sigma)$ in combination with tile coding to perform empirical evaluations in the mountain cliff environment. In the non-linear function approximation case, we combined n-step $Q(\sigma)$ with the DQN architecture and named the resulting architecture the $Q(\sigma)$ network. We then proceeded to test our new architecture in the mountain car environment. These two extensions increase the applicability of the n-step

$Q(\sigma)$ algorithm to more complex environments and tasks.

In Chapter 3, 4, and 5 we presented empirical evaluations of n-step $Q(\sigma)$ in the tabular, linear, and non-linear function approximation cases, respectively. We started our evaluations in simple tasks that allowed us to gain a clear view of the performance of the n-step $Q(\sigma)$. Then, we investigated if similar effects to the ones found in simpler tasks could be found under more complex settings. We found that many of the effects found in the simpler tasks carry over to more complex tasks.

In the tabular case, we showed that higher values of the parameter σ resulted in better initial performance, while smaller values resulted in better final performance. We also found that the decaying σ algorithm — an algorithm with an initial value of σ of one that slowly decays to zero — benefits from the early performance induced by high values of σ and the final performance corresponding to small values of σ. Lastly, we also found that intermediate values of σ could outperform either of the extremes.

In the linear function approximation case in the mountain cliff environment, we demonstrated that the effects found in the tabular case also carry over, to some extend, to the linear function approximation case. In this environment we found that the decaying σ algorithm performed the best. However, the effect of the parameter σ was not exactly the same as before: smaller values of σ resulted in better final performance, but larger values of σ not always improved the initial performance. In fact, the value of σ that resulted in the best initial performance was 0.5.

In our last empirical evaluation we tested the $Q(\sigma)$ network — a combination of n-step $Q(\sigma)$ and DQN — in the mountain car environment. We provided an extensive study on how particular algorithmic details of DQN, the parameter n, and the parameter σ influence the performance of the $Q(\sigma)$ network. We found that a variant of the decaying σ algorithm with linear decay performed the best. We found contrasting results about the effect of the parameter σ on the performance of the algorithm. In contrast with the results from the mountain cliff environment, we found that larger values of σ resulted in better initial performance, but smaller values not always resulted

in better final performance. The value of σ that performed the best during late training was 0.5.

This work provides strong support for using n-step $Q(\sigma)$ over the n-step algorithms Sarsa and Tree Backup. Our experiments demonstrated that the performance of the decaying σ algorithm is robust to the learning problem, the type of environment, and the type of representation used for the action-value function. The main benefit of using n-step $Q(\sigma)$ is that it provides a flexible framework that can be adapted to a wide range of learning tasks in order to achieve better performance.

6.1 Future Work

Since its introduction in Sutton and Barto (2018), several extensions have already been made to the n-step $Q(\sigma)$ algorithm. To our knowledge, Dumke (2017) and Harutyunyan, Vrancx, Bacon, Precup, and Nowé (2017) independently presented each an extension of the n-step $Q(\sigma)$ algorithm to use elgibility traces — called $Q(\sigma, \lambda)$. Building upon Dumke's (2017) work, Yang, Shi, Zheng, Meng, and Pan (2018) presented a theoretical analysis on the $Q(\sigma, \lambda)$ algorithm proving its convergence in the on-line control case. De Asis (2018) extended the $Q(\sigma, \lambda)$ to use control variates as in De Asis and Sutton (2018), and showed that in this case the Retrace(λ) algorithm (Munos, Stepleton, Harutyunyan, & Bellemare, 2016) can be represented using $Q(\sigma, \lambda)$.

There has already been work built upon the n-step $Q(\sigma)$ algorithm. Nevertheless, three promising avenues of research remain unexplored: (1) theoretical analysis of the n-step $Q(\sigma)$ algorithm to the linear function approximation case, (2) studying different ways to select the value of σ on a per-decision basis, and (3) providing empirical evaluations of the $Q(\sigma)$ network in more complex environments.

The results presented by Yang et al. (2018) were done for the tabular case. The main challenge in extending the theoretical analysis of n-step $Q(\sigma)$ to the linear function approximation case is avoiding the divergence issues often encountered when combining off-policy sampling, bootstrapping, and function

approximation — a problem that has been eloquently named the *deadly triad* (Sutton & Barto, 2018). In this case, we would have to employ some form of gradient based method analogous to the ones introduced in Sutton, Maei, and Szepesvári (2009) in order to circumvent the deadly triad. Fortunately, such analysis has already being done for the Tree Backup(λ) algorithm (Touati, Bacon, Precup, & Vincent, 2017) and for an algorithm analogous to Sarsa(λ) (Hamid Reza, 2011; Sutton et al., 2009). Therefore, it seems possible that an algorithm that combines both of those gradient based methods would also inherit their convergence guarantees.

The second avenue of research is studying the different methods for selecting the parameter σ on a per-decision basis. In chapter 5 we suggested that, similar to the ABQ(ζ) algorithm (Mahmood et al., 2017), σ could be selected to counteract the negative effects that the importance sampling ratio has in the variance of the algorithm. Another approach would be to use a count-based method — such as in Bellemare et al. (2016) — in order to select σ based on how many times a state-action pair has been observed. The intuition behind this method is that more familiar state-action pairs would have been visited more often and, consequently, would have more accurate action-value estimates. Thus, in this case selecting a value of σ closer to zero would result in an estimate of the return with smaller bias and variance. In this case the count-based method would be a proxy for how accurate are the estimates of the action-value function. Alternatively, one could simply use the TD-error of the update function as a proxy for the accuracy of the estimates.

Finally, it would be valuable to the deep reinforcement learning community to replicate the analysis of the $Q(\sigma)$ network in a more complex environment such as the arcade learning environment (Bellemare, Naddaf, Veness, & Bowling, 2013). Even though the mountain car environment allowed for thorough study of the $Q(\sigma)$ network, it is possible that many of the effects that we observed there would not carry over to more complex environments. After all, the original DQN architecture was designed to work on pixel data which is high dimensional and lacks the topographical structure that we encounter in the mountain car environment. Thus, analyzing the performance of the $Q(\sigma)$

network in a more complex environment and finding similar effects to the ones found in mountain car would provide stronger support for the benefits of using n-step $Q(\sigma)$ with non-linear function approximation.

6.2 Summary

We presented the n-step $Q(\sigma)$ algorithm that unifies the n-step algorithms Sarsa and Tree Backup. We provided extensions of n-step $Q(\sigma)$ algorithm to the off-policy case and the linear and non-linear function approximation cases. For each of the cases introduced — tabular, linear, and non-linear function approximation — we presented empirical evaluations of n-step $Q(\sigma)$ and observed that similar effect can be observed across these settings. Because of the great flexibility that it provides, we were always able to find an instance of the n-step $Q(\sigma)$ algorithm that outperformed the n-step algorithms Sarsa and Tree Backup over a wide variety of settings and environments. The main benefit of using the n-step $Q(\sigma)$ algorithm is that it is capable to adapt to wide variety of learning tasks in order to achieve better performance.

References

Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences*, *10*(1), 25–61. doi:https://doi.org/10.1016/0025-5564(71)90051-4

Albus, J. (1981). *Brains, behavior, and robotics*. Byte books. BYTE Books. Retrieved from https://books.google.ca/books?id=mBJwAAAAIAAJ

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., & Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. *CoRR*, *abs/1606.01868*. arXiv: 1606.01868. Retrieved from http://arxiv.org/abs/1606.01868

Bertsekas, D. P. & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming* (1st). Athena Scientific.

De Asis, K. (2018). *A unified view of multi-step temporal difference learning* (Master's thesis, University of Alberta). Submitted.

De Asis, K., Hernandez-Garcia, J., Holland, G., & Sutton, R. (2018). Multi-step reinforcement learning: A unifying algorithm. Retrieved from https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16294

De Asis, K. & Sutton, R. (2018). Per-decision multi-step temporal difference learning with control variates. *CoRR*, *abs/1807.01830*. arXiv: 1807.01830. Retrieved from https://arxiv.org/abs/1807.01830

Dumke, M. (2017). Double q(σ) and q(σ, λ): Unifying reinforcement learning control algorithms. *CoRR*, *abs/1711.01569*. arXiv: 1711.01569. Retrieved from http://arxiv.org/abs/1711.01569

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., ... Kavukcuoglu, K. (2018). IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, *abs/1802.01561*. arXiv: 1802.01561. Retrieved from http://arxiv.org/abs/1802.01561

Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *In proceedings of the international conference on artificial intelligence and statistics (aistats'10). society for artificial intelligence and statistics*.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. http://www.deeplearningbook.org. MIT Press.

Gruslys, A., Azar, M. G., Bellemare, M. G., & Munos, R. (2017). The reactor: A sample-efficient actor-critic architecture. *CoRR*, *abs/1704.04651*. arXiv: 1704.04651. Retrieved from http://arxiv.org/abs/1704.04651

Hamid Reza, M. (2011). *Gradient temporal-difference learning algorithms* (Doctoral dissertation, University of Alberta).

Harutyunyan, A., Vrancx, P., Bacon, P., Precup, D., & Nowé, A. (2017). Learning with options that terminate off-policy. *CoRR*, *abs/1711.03817*. arXiv: 1711.03817. Retrieved from http://arxiv.org/abs/1711.03817

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, *abs/1710.02298*. arXiv: 1710.02298. Retrieved from http://arxiv.org/abs/1710.02298

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., & Silver, D. (2018). Distributed prioritized experience replay. *CoRR*, *abs/1803.00933*. arXiv: 1803.00933. Retrieved from http://arxiv.org/abs/1803.00933

Jaakkola, T., Jordan, M. I., & Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Comput. 6*(6), 1185–1201. doi:10.1162/neco.1994.6.6.1185

Kearns, M. J. & Singh, S. P. (2000). Bias-variance error bounds for temporal difference updates. In *Proceedings of the thirteenth annual conference on computational learning theory* (pp. 142–147). COLT '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Retrieved from http://dl.acm.org/citation.cfm?id=648299.755183

Lin, L.-J. (1992). *Reinforcement learning for robots using neural networks* (Doctoral dissertation, Pittsburgh, PA, USA). UMI Order No. GAX93-22750.

Mahmood, A. R., Yu, H., & Sutton, R. S. (2017). Multi-step off-policy learning without importance sampling ratios. *CoRR*, *abs/1702.03006*. arXiv: 1702.03006. Retrieved from http://arxiv.org/abs/1702.03006

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. Retrieved from http://dx.doi.org/10.1038/nature14236

Munos, R., Stepleton, T., Harutyunyan, A., & Bellemare, M. G. (2016). Safe and efficient off-policy reinforcement learning. *CoRR*, *abs/1606.02647*. arXiv: 1606.02647. Retrieved from http://arxiv.org/abs/1606.02647

Precup, D., Sutton, R. S., & Singh, S. (2000). Eligibility traces for off-policy policy evaluation. In M. Kaufman (Ed.), *Proceedings of the 17th international conference on machine learning* (pp. 759–766).

Rummery, G. A. (1995). *Problem solving with reinforcement learning* (Doctoral dissertation, Cambridge University).

Rummery, G. A. & Niranjan, M. (1994). *On-line q-learning using connectionist systems*. Cambridge University.

Singh, S., Jaakkola, T., Littman, M. L., & Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, *38*(3), 287–308. doi:10.1023/A:1007678930559

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky & M. E. Hasselmo (Eds.), *Advances in neural information processing systems 8* (pp. 1038–1044). MIT Press.

Sutton, R. S. & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd). Manuscript in preparation.

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

Sutton, R. S. & Barto, A. G. (1998). *Introduction to reinforcement learning* (1st). Cambridge, MA, USA: MIT Press.

Sutton, R. S., Maei, H. R., & Szepesvári, C. (2009). A convergent o(n) temporal-difference algorithm for off-policy learning with linear function approximation. In D. Koller, D. Schuurmans, Y. Bengio, & L. Bottou (Eds.), *Advances in neural information processing systems 21* (pp. 1609–1616). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/3626-a-convergent-on-temporal-difference-algorithm-for-off-policy-learning-with-linear-function-approximation.pdf

Tieleman, T. & Hinton, G. (2012). Lecture 6e—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning.

Touati, A., Bacon, P., Precup, D., & Vincent, P. (2017). Convergent tree-backup and retrace with function approximation. *CoRR*, *abs/1705.09322*. arXiv: 1705.09322. Retrieved from http://arxiv.org/abs/1705.09322

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and q-learning. *Machine Learning*, *16*(3), 185–202. doi:10.1007/BF00993306

van Hasselt, H. (2011). *Insights in reinforcement learning* (Doctoral dissertation, Utrecht University).

van Seijen, H., van Hasselt, H., Whiteson, S., & Wiering, M. (2009). A theoretical and empirical analysis of expected sarsa. In *2009 ieee symposium on adaptive dynamic programming and reinforcement learning* (pp. 177–184). doi:10.1109/ADPRL.2009.4927542

van Seijen, H., Mahmood, A. R., Pilarski, P. M., Machado, M. C., & Sutton, R. S. (2015). True online temporal-difference learning. *CoRR*, *abs/1512.04087*. arXiv: 1512.04087. Retrieved from http://arxiv.org/abs/1512.04087

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation, Cambridge University).

Watkins, C. J. C. H. & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Yang, L., Shi, M., Zheng, Q., Meng, W., & Pan, G. (2018). A unified approach for multi-step temporal-difference learning with eligibility traces

in reinforcement learning. *CoRR*, *abs/1802.03171*. arXiv: 1802.03171. Retrieved from http://arxiv.org/abs/1802.03171

Appendix A

Effects of the Target Network Update Frequency

In this section we present the summaries of the algorithms trained for the experiment in section 5.2.5. In that experiment, we trained 4 different $Q(\sigma)$ agents in the mountain car environment: one agent for each value of σ in $\{0, 0.5, 1\}$ and a decaying σ agent with a linear decay of 0.002. All the agents used the $Q(\sigma)$ network architecture from section 5.2 and were trained with values for the target network update frequency parameter of 500, $1,000$, and $2,000$. The measure of performance was the average return per episode over 500 episodes — the entire training period.

$Q(0)$, $Q(0.5)$, and decaying σ with linear decay performed the best with a target network update frequency of 500 time steps. For all these algorithms, increasing the update frequency consistently resulted in worse performance. There was no statistically significant difference between the performances of $Q(1)$ with different target network update frequencies. Table A.1 shows the results averaged over 100 independent runs and with corresponding 95% confidence interval.

Notice that the only algorithms that benefit from lower update frequencies are those that compute an expectation at every step of the backup. Since the estimate of the return used in the loss function is computed using the target network, it is possible that the mismatch in learning between the two networks — the update and the target network — is the cause of this drop in performance. A possible experiment for testing this hypothesis would be

Table A.1: Comparison of the performance of the $Q(\sigma)$ network with different values of σ and the target network update frequency. The results were computed using 100 independent runs. Lower (LB) and upper (UB) 95% confidence interval bounds computed using a chi-squared distribution are provided to validate the results. $Q(0)$, $Q(0.5)$, and decaying σ with linear decay performed better using a target network update frequency of 500.

$Q(1)$, Sarsa					
		Average Return per Episode			
n	Update Frequency	Mean	Standard Error	LB	UB
20	500	-158.59	1.90	-162.36	-154.81
	1,000	-157.05	2.18	-161.37	-152.73
	2,000	-166.82	2.90	-172.57	-161.07

$Q(0.5)$					
		Average Return per Episode			
n	Update Frequency	Mean	Standard Error	LB	UB
20	500	-164.32	0.37	-165.06	-163.58
	1,000	-196.79	0.77	-198.32	-195.26
	2,000	-257.83	1.39	-260.58	-255.08

$Q(0)$, Tree Backup					
		Average Return per Episode			
n	Update Frequency	Mean	Standard Error	LB	UB
20	500	-193.73	0.41	-194.54	-192.92
	1,000	-243.15	0.76	-244.65	-241.65
	2,000	-338.62	1.55	-341.7	-335.54

Decaying σ with Linear Decay					
		Average Return per Episode			
n	Update Frequency	Mean	Standard Error	LB	UB
20	500	-148.46	1.96	-152.35	-144.56
	1,000	-156.97	4.05	-165.00	-148.94
	2,000	-178.46	6.34	-191.04	-165.88

to implement a version of the target and update network in a tabular domain and see how different frequencies affect the performance of the agents in that domain. We would expect that even for tabular representations of the action-value function higher update frequencies would result in worse performance.

If this hypothesis turned out to be true, it would imply that these algorithms would benefit from not using a target network at all. Nevertheless, more experimentation is needed in more complex environment since the target network was first devised for environments with a high dimensional state space, such as the arcade learning environment.