Deep Reinforcement Learning with Smoothed Q-Learning

Smoothed Q-Learning and Smoothed Deep Q-Networks

Author: Xingrui Gu[1]

Degree: Msc Computer Science

Supervisor: Prof. David Barber

Submission date: 18th Sep. 2023

Abstract

In the field of model-free reinforcement learning, traditional algorithms such as Q-Learning and its extension, Deep Q-Networks (DQN), often face performance limitations, particularly in the early stages of learning within complex environments. These limitations arise primarily from the computational challenges associated with refining imprecise estimates of the state-action value function. To mitigate these issues, this work explores the theoretical foundations of Smoothed Q-Learning and extends this framework to deep reinforcement learning, introducing Smoothed Deep Q-Networks (SDQN). Employing the smoothing strategies inherent in Smoothed Q-Learning, SDQN computes a weighted average of Q-values for each possible action, based on their respective probabilities. This approach effectively reduces bias in the estimation of the state-action value function and expedites convergence to optimal policies. Empirical results, obtained from experiments conducted in virtual environments, indicate that both Smoothed Q-Learning and SDQN achieve faster convergence rates and more stable learning trajectories compared to their traditional counterparts.

Acknowledgments

I am profoundly grateful to **Professor David Barber**, whose mentorship has been nothing short of transformative. His incisive guidance, coupled with an extraordinary patience for resolving my myriad queries and confusions, has been instrumental in shaping this academic endeavor. The depth of his expertise and the clarity of his insights have been a constant source of inspiration and have indelibly enriched my understanding of the subject matter.

Equally deserving of my gratitude is **Professor Barber's spouse**. During a particularly challenging evening, her compassionate advice and genuine concern for my well-being provided not just solace but also a renewed sense of purpose. Her kindness served as a poignant reminder of the human elements that underlie the rigors of academic pursuit.

I must also extend my heartfelt appreciation to my parents, who have been my bedrock of support. Their unwavering faith in my abilities and their ceaseless encouragement have fortified me against the challenges I have encountered. Their love has been my sanctuary, and their belief in me, my greatest asset.

Finally, my deepest thanks go to my beloved, **Miss Yiner He**. Her emotional support has been my refuge, and her love, my sustenance. Over the past year, her presence has been a beacon that guided me through the complexities of academic life, providing emotional clarity and spiritual sustenance. Her love has been a wellspring of inspiration and a constant reminder of the joys that life holds beyond the academic sphere.

Contents

Chapter 1

Prologue

1.1 Brief Overview

In this research project, we delve into the intricacies of Reinforcement Learning (RL), a paradigm where agents learn to make decisions through continuous interactions with their environment. While RL holds immense potential for a variety of applications, it is often beset by challenges such as unstable learning trajectories and overconfidence in model predictions. The probabilistic nature of rewards and state transitions further complicates the agent's objective of maximising the total expected reward over time. To navigate this complex landscape, our research proposes a novel amalgamation of Smoothed Q-Learning strategies with Deep Q-Networks (DQN) algorithms. We extend our focus to the statistical underpinnings of RL, analysing the impact of statistical probability distributions on the learning process. Additionally, we conduct an in-depth investigation into the fundamental properties and algorithmic implementations of Smoothed Q-Learning, developing new smoothing strategies in the process. The overarching aim is to stabilise and optimise the agent's decision-making, thereby enhancing the reliability and applicability of RL agents in real-world scenarios, including but not limited to video games, autonomous vehicles, and robotics.

1.2 Goals and Aims

1.2.1 Aims

- **Aim 1: Highlighting the Superiority of Smoothed Q-Learning**: A rigorous comparative analysis will be conducted to evaluate the performance of Q-Learning, Double Q-Learning, and Smoothed Q-Learning across various experimental setups. The focus will be on demonstrating the enhanced convergence to optimal policy and stability of Smoothed Q-Learning, thereby establishing its advantages over traditional methods.
- **Aim 2: Unveiling the Strengths of Smoothed Q-Learning Strategies**: This analysis will delve into the multifaceted nature of Smoothed Q-Learning across various strategies. The study aims to elucidate how these strategies contribute to the superior performance of Smoothed Q-Learning in different scenarios, emphasizing its robustness and adaptability.
- **Aim 3: Integration and Exploration**: This study delves into the nuanced integration of Smoothed Q-Learning with Deep Q-Networks (DQN), focusing on select strategic implementations. This involves understanding the enhancements brought about by such amalgamation

and their implications on stability and performance of the resulting model.

- **Aim 4: Augmenting RL Landscape**: By investigating Smoothed Q-Learning and its potential integration with DQN, this study aspires to illuminate and enrich the field of RL with a novel method. This is aimed at stimulating the development of increasingly sophisticated and efficient learning algorithms and ultimately broadening the horizons of RL applications.

1.2.2 Goals

- **Goal 1: Validation in Complex Environments:** Conduct further experimentation in more complex environments to validate and compare the effectiveness of different strategies of Q-Learning, Double Q-Learning and Smoothed Q-Learning with different strategies. The goal is to gather empirical evidence to better understand the scalability and robustness of these learning algorithms in different contexts.

- **Goal 2: Development of 'Smoothed DQN':** The primary deliverable of this research is the creation of a novel 'Smoothed DQN' model, integrating Smoothed Q-Learning with DQN. This goal includes the successful implementation of the model, as well as preliminary validation of its functioning and performance.

- **Goal 3: Comprehensive Comparative Analysis:** A comprehensive report comparing the performances of Q-Learning, Double Q-Learning and Smoothed Q-Learning in various experimental settings is to be formulated. This goal involves producing clear, detailed, and actionable insights from the comparison.

- **Goal 4: Experimental Results and Insights:** Another crucial deliverable is a set of experimental results generated by applying the 'Smoothed DQN' method to classical games named "Cartpole". This includes detailed analysis and interpretation of the results, demonstrating the practicality and effectiveness of 'Smoothed DQN' in real-world-like scenarios.

1.3 Methodological Progression & Experimental Framework

The research was conducted following a systematic, iterative approach, each stage of which contributed to the depth and breadth of knowledge required for the subsequent steps.

1. **Theoretical Grounding and Literature Review:** The initial phase involved a comprehensive review of key theories and concepts in reinforcement learning, with a particular focus on Q-learning, Double Q-learning, and Smoothed Q-learning. This stage also included an in-depth study of Markov Decision Processes (MDPs) and their applications in any scenarios. Alongside these foundational topics, the research extended to explore the statistical underpinnings of Smoothed Q-Learning in reinforcement learning. This multi-faceted theoretical grounding served as the bedrock for the practical explorations and algorithmic developments that followed.

2. **Replication of Baseline Experiments:** The research began with replicating key experiments from existing literature, specifically Professor Barber's work on Smoothed Q-Learning[1] and Double Q-Learning experiments in Grid World and Roulette[2]. This step

involved implementing the algorithms, setting up the experimental environments, and verifying the results against the original papers. The codebase was designed to be modular for future work. This phase helped deepen the understanding of RL algorithms and refine coding skills through iterative debugging and verification.

3. **Realisation of Policy Algorithms:** Subsequently, three smoothed strategies, integral to Smoothed Q-Learning, were implemented. This stage was pivotal, as it not only cemented a comprehensive understanding of the techniques involved but also laid the groundwork for the further exploration and deployment of Smoothed Q-Learning and its associated policies.

4. **Development of 'Smoothed DQN':** Armed with the insights obtained, the research transitioned into the phase of synthesising Smoothed Q-Learning with Deep Q-Networks (DQN), giving rise to the novel 'Smoothed DQN' model. This methodology incorporated iterative development, testing, and refinement of the newly proposed approach.

5. **Experimental Studies and Analysis:** A fundamental part of this systematic exploration involved conducting experimental studies and performing detailed analysis using the classic CartPole games. These high-dimensional, real-world-like environments served as the testbeds for validating and refining 'Smoothed DQN'. The comparative analysis of various Q-Learning strategies and their performance under diverse conditions was facilitated during this stage.

6. **Validation and Refinement:** The concluding phase involved validation and refinement of the 'Smoothed DQN' model in more complex experimental environments. The findings from these experiments contributed to optimising 'Smoothed DQN', thereby enhancing its scalability and robustness in a multitude of real-world scenarios.

Throughout this process, iterative cycles of implementation, evaluation, and refinement were a constant feature, ensuring the development of a robust and effective 'Smoothed DQN' approach.

1.4 Report Overview

This report is meticulously structured into five principal chapters, each of which is designed to offer an exhaustive understanding of the advancements in Q-learning and its various derivatives. Below is an overview of the remaining content:

- **Chapter 2: Theoretical Foundations of Reinforcement Learning** This chapter rigorously articulates the mathematical foundations underpinning Reinforcement Learning (RL), with an emphasis on canonical algorithms such as Q-Learning and Double Q-Learning. The objective is to establish a robust theoretical framework that serves as a precursor for the formal introduction and justification of Smoothed Q-Learning, which will be expounded in the ensuing chapter.

- **Chapter 3: Smoothed Q-Learning** Building upon the foundational concepts established in Chapter 2, this chapter undertakes a rigorous exploration of Smoothed Q-Learning. The discourse primarily references seminal contributions from Professor David Barber, providing a critical evaluation of the constraints innate to traditional Q-Learning and Double Q-Learning methods. The chapter culminates in an empirical validation of these algorithms across multiple experimental settings of varying complexity, emphasizing the efficacy of different smoothing strategies within Smoothed Q-Learning. Furthermore, the chapter unveils

a novel smoothing policy, Clipped Softmax, and supports its superiority through empirical validation in complex environments.

- **Chapter 4: Smoothed Deep Q-Networks**
  This chapter investigates the integration of statistical methods into deep reinforcement learning, specifically within the framework of Smoothed Q-Learning. The chapter commences with a formal analysis of the mathematical properties and inherent limitations of Deep Q-Networks (DQN) and Double Deep Q-Networks (DDQN). It then proceeds to introduce the Smoothed Deep Q-Networks (SDQN), providing a comprehensive exposition of its algorithmic structure and associated reward function.

- **Chapter 5: Conclusions** The final chapter amalgamates the central contributions of the preceding chapters, offering a synthesized understanding of the advancements made. It also identifies potential areas warranting future scholarly attention, outlining promising avenues for extending and enhancing the current state of knowledge.

Chapter 2

Theoretical Foundations of Reinforcement Learning

This chapter delineates the theoretical underpinnings of Reinforcement Learning (RL) and Q-Learning. It commences by introducing the broader field of RL, proceeds to a focused discourse on Q-Learning, and concludes by setting forth the objectives and methodologies of the research, providing an overview of the subsequent chapters.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a specialized domain within machine learning aimed at optimizing agents' decision-making capabilities via interactions with their environment [3, 4]. Originating from behaviorist psychology [5], RL has diverse applications across domains like robotics, gaming, and autonomous vehicles [6]. The core objective in RL is to maximize an agent's cumulative expected reward over time, considering the stochastic nature of rewards and state transitions [4]. Accordingly, value functions are employed as quantitative metrics to evaluate the merit of states and actions [7].

2.2 Q-Learning

Q-Learning, an off-policy algorithm, excels in optimizing discrete and finite Markov Decision Processes (MDPs) [8]. It has found applications in varied fields such as energy management and transportation systems [9, 10].

2.2.1 Theoretical Foundations

Q-Learning employs Controlled Markov Processes and Bellman equations as its mathematical scaffolding [11, 12].

Markov Decision Process and Policy Definition Given a state s_n from state space $\mathcal{S}$ and an action a_n from action space $\mathcal{A}$ at each step n, a Markov Decision Process models the state transitions as:

$$P^a_{s's} = \text{Prob}[s_{t+1} = s' | s_t = s, A_t = a]. \tag{2.1}$$

Value Functions The Value functions aims to maximize the total discounted expected reward, defined as follows:

$$V(x) = \mathbb{R}_s(\pi(s)) + \gamma \sum_y p_{ss'}[\pi(s)]V(s'). \tag{2.2}$$

Here, $V^*(x)$ represents the value function for an optimal policy π^*:

$$V^*(x) = \max_a(\mathbb{R}_s(a) + \gamma \sum_y p_{ss'}[a]V^*(s')). \tag{2.3}$$

Value Function Approximation In computationally complex settings, value function approximations such as neural networks or linear function approximations are employed [13].

2.2.2 Q-Learning Algorithm

The Q-Learning algorithm, as outlined in Algorithm 1, operationalizes the mathematical constructs into a computational framework for RL [12, 8].

Q-Learning Formalization Q-Learning extends the value function concept to Q values, used for estimating the optimal policy without prior knowledge of the environment's dynamics [13]. The Q-Learning update rule is defined by Equation (2.4):

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)[r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)] \tag{2.4}$$

- $Q_{t+1}(s_t, a_t)$ is the updated value of the action value for action a_t in state s_t at the next time step $t + 1$.
- $Q_t(s_t, a_t)$ represents the current estimate of the action value for action a_t in state s_t at time t.
- α is the learning rate, determining how quickly new information replaces old information.
- r_t is the immediate reward received at time t.
- γ is the discount factor determining the present value of future rewards.
- $\max_a Q_t(s_{t+1}, a)$ is the maximum estimated action value for the next state s_{t+1} at time t.

Algorithm 1 Q-Learning Algorithm

Input: Q-values matrix $Q(s, a)$ for all states s and actions a, starting state s, learning rate α, and discount factor γ
A policy (e.g., ε-greedy) to derive actions from Q

Output: Updated Q-values matrix $Q(s, a)$

Initialization:
    Initialize $Q(s, a)$ for all s, a and starting state s
do
    Choose action a from state s using policy derived from Q
    Take action a, observe reward r and new state s'
    Update Q-value:
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    Update the current state:
        $s \leftarrow s'$
while termination condition is met

2.2.3 Convergence of Q-Learning

The convergence of Q-learning to an optimal policy is a salient feature ensuring its robustness in policy optimization tasks, although convergence is contingent upon specific conditions [14].

Criteria for Convergence For guaranteed convergence, it is imperative that each state-action pair (s, a) is explored infinitely often during the learning process. Typically, an ϵ-greedy exploratory policy with $\epsilon > 0$ suffices for this criterion [8].

Convergence in Tabular Case In finite, discrete state-action spaces, the convergence of Q-learning to the optimal policy π^* is well-established, provided that the sequence of learning rates $\{\alpha_t\}$ satisfies certain summability conditions [8].

Role of the Action-Replay Process The Action-Replay Process (ARP) is instrumental in the proof of convergence, acting as an analytical framework that delineates the temporal interdependencies between action sequences and adaptive learning rates [15, 16, 17].

Asymptotic Convergence Rates While empirical observations often indicate exponential rates of convergence, there exist specific configurations wherein Q-learning exhibits polynomial-time convergence [18, 17].

Mechanism of Convergence to Optimal Policy The convergence of Q-Learning to an optimal policy π^* is underpinned by its iterative update scheme, designed to approximate the optimal action-value function Q^*. This approximation is governed by the Bellman equation for Q^*, which serves as a fixed point in the function space. Provided that each state-action pair (s, a) is visited infinitely often and that the sequence of learning rates $\{\alpha_t\}$ satisfies the summability conditions $\sum_{t=1}^{\infty} \alpha_t(s, a) = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2(s, a) < \infty$, the Q-value estimates asymptotically converge to Q^*. Consequently, the policy π induced by these Q-values also converges to the optimal policy π^* [8, 14].

Implications and Extensions The established convergence properties in the tabular case are not mere theoretical artifacts but serve as foundational elements for the exploration of more complex methods, such as Double Q-learning and Smoothed Q-learning, which will be elaborated upon in subsequent sections.

2.2.4 Challenges in Q-Learning

Sensitivity to Noise and Overestimation Bias in Q-Learning

Q-Learning, while efficacious for policy optimization, is not devoid of limitations, most notably in the stability and precision of its learning dynamics. These limitations are principally evident in two domains: sensitivity to input noise and overestimation bias.

Sensitivity to Input Noise The Q-Learning algorithm's update rule inherently maximizes Q-values, making the algorithm vulnerable to stochastic fluctuations in the Q-value function. This susceptibility results in erratic learning trajectories, thereby compromising both the stability and performance of the learning process [19, 4, 20].

Overestimation Bias Q-Learning's intrinsic noise sensitivity also leads to a systematic overestimation of Q-values, especially in environments characterized by high stochasticity or sparse data. This overestimation bias distorts the action selection process, exacerbating the algorithm's instability [2, 21]. The problem is further magnified in model-free deep reinforcement learning contexts, where the data-hungry nature of deep learning algorithms collides with the often sparse or unlabeled data available in reinforcement learning environments. This discordance amplifies the overestimation bias, as the neural networks employed for function approximation are susceptible to overfitting in data-scarce scenarios. The resultant bias can severely hinder the learning process, culminating in suboptimal policies and unstable learning dynamics.

Implications and Future Work The aforementioned challenges intrinsic to Q-Learning and its deep learning extensions necessitate algorithmic improvements for robust real-world applications. These challenges highlight the imperative for future research focused on algorithmic refinements to mitigate the adverse effects of noise sensitivity and overestimation bias. Further elaboration and formal proofs concerning these challenges are provided in Appendix A.1.1.

2.3 Double Q-Learning

Introduced by Hado van Hasselt in 2010[2], Double Q-Learning aims to ameliorate the overestimation bias observed in standard Q-Learning by utilizing dual Q-value estimators—one for action selection and another for value updating. This section delineates the algorithmic underpinnings and limitations of Double Q-Learning.

2.3.1 Theoretical Framework of Double Q-Learning

Double Q-Learning employs double estimators framework, eschewing the single estimator convention in classical Q-Learning (refer to Appendix A.1.2 for a detailed derivation). Prior analyses indicate that this architecture yields action-value functions less susceptible to overestimation bias, resulting in more conservative value estimates. During each iteration, the algorithm probabilistically selects either Q_A or Q_B for action selection, while the complementary Q-value function

is employed for the value update. Formally, the action a for a given state s is ascertained by $a = \arg\max_a(Q_A(s,a) + Q_B(s,a))$.

2.3.2 The Double Q-Learning Algorithm

The Double Q-Learning update rule modifies the Q-Learning rule by mitigating the over-estimation bias, and it is defined as follows[2]:

$$Q^A_{t+1}(s_t,a_t) = Q^A_t(s_t,a_t) + \alpha_t(s_t,a_t)\left[r_t + \gamma Q^B_t\left(s_{t+1}, \arg\max_a Q^A_t(s_{t+1},a)\right) - Q^A_t(s_t,a_t)\right] \quad (2.5)$$

And vice versa for updating Q^B. In each iteration, either Q^A or Q^B is chosen randomly for updating. Here:

- $Q^A_{t+1}(s_t,a_t)$ is the updated value of the action value for action a_t in state s_t at the next time step $t+1$ using the Q-function Q^A.
- $Q^A_t(s_t,a_t)$ represents the current estimate of the action value for action a_t in state s_t at time t using the Q-function Q^A.
- $\alpha_t(s_t,a_t)$ is the learning rate at time t, determining how quickly new information replaces old information.
- r_t is the immediate reward received at time t.
- γ is the discount factor determining the present value of future rewards.
- $Q^B_t\left(s_{t+1}, \arg\max_a Q^A_t(s_{t+1},a)\right)$ is the maximum estimated action value using Q-function Q^B for the action that maximizes the Q-function Q^A in the next state s_{t+1} at time t.

Algorithm 2 Double Q-Learning Algorithm

Input:
Q-values matrices $Q_A(s,a)$ and $Q_B(s,a)$ for all states s and actions a, starting state s, learning rate α, and discount factor γ
A policy π derived from $Q_A(s,\cdot)$ and $Q_B(s,\cdot)$
Output:
Updated Q-values matrices $Q_A(s,a)$ and $Q_B(s,a)$

Initialization:
    Initialize $Q_A(s,a)$, $Q_B(s,a)$ for all s,a and starting state s
do
    Choose action a based on policy π, observe reward r, and new state s'
    With probability 0.5, set UPDATE = A, otherwise set UPDATE = B
if UPDATE = A **then**
        Define $a^* = \arg\max_a Q_A(s',a)$
        $Q_A(s,a) \leftarrow Q_A(s,a) + \alpha[r + \gamma Q_B(s',a^*) - Q_A(s,a)]$
else
        Define $b^* = \arg\max_a Q_B(s',a)$
        $Q_B(s,a) \leftarrow Q_B(s,a) + \alpha[r + \gamma Q_A(s',b^*) - Q_B(s,a)]$
    Update the current state:
        $s \leftarrow s'$
while termination condition is met

2.3.3 Convergence of Double Q-Learning

Double Q-Learning, an extension of standard Q-Learning, mitigates the overestimation bias inherent in the latter through a dual Q-value estimation mechanism [2]. The algorithm's convergence in finite, discrete state-action spaces ($|S \times A| < \infty$) has been rigorously established under certain conditions.

Prerequisite Conditions for Convergence Convergence criteria are substantiated under the following conditions:

- State-action space finiteness: $|S \times A| < \infty$
- Bounded discount factor: $\gamma \in [0, 1)$
- Suitably chosen learning rates: α

Action-Replay Process The Action-Replay Process (ARP) serves as a theoretical construct that explicates how action sequences and learning rates interact. Through ARP, the proof decouples the Q-value tables Q_A and Q_B, thereby reducing the overestimation bias prevalent in standard Q-Learning.

Contraction Mapping and Asymptotic Behavior The mathematical proof validates that the expected contraction of F_t adheres to specific contraction conditions, thus confirming convergence. Because the updates for Q_A and Q_B are mirror images of each other, proving that one converges essentially proves that the other does as well. Additionally, the study confirms that the long-term difference ΔBA_t between Q_A and Q_B eventually shrinks to zero.

Mechanics of Convergence The dual Q-value estimation framework of Double Q-Learning ensures that both Q_A and Q_B asymptotically converge to the optimal Q-values under policy π^*. This is mathematically corroborated by the Bellman equation for Q^*, serving as the fixed point to which both Q_A and Q_B converge. Convergence of the derived policy to π^* is guaranteed, contingent upon the learning rates and exploration policies satisfying specific summability conditions [2].

2.3.4 Challenges in Double Q-Learning

Computational and Memory Overheads in MDP Environment

Despite its advantages in bias correction, Double Q-Learning incurs non-trivial computational and memory costs, particularly when executed in an MDP experimental environment. Empirical data, as presented in Table 2.1, substantiates that the computational latency of Double Q-Learning exceeds that of standard Q-Learning by approximately 1.67 times. Additionally, the memory overhead of Double Q-Learning is notably high, especially during the initial episodes, as delineated in Figure 2.1.

Algorithm	Time (seconds)
Q-Learning	401.194
Double Q-Learning	670.755

Table 2.1: Empirical computational time comparison between Q-Learning and Double Q-Learning, indicating an increased computational latency by a factor of 1.67 in the latter.

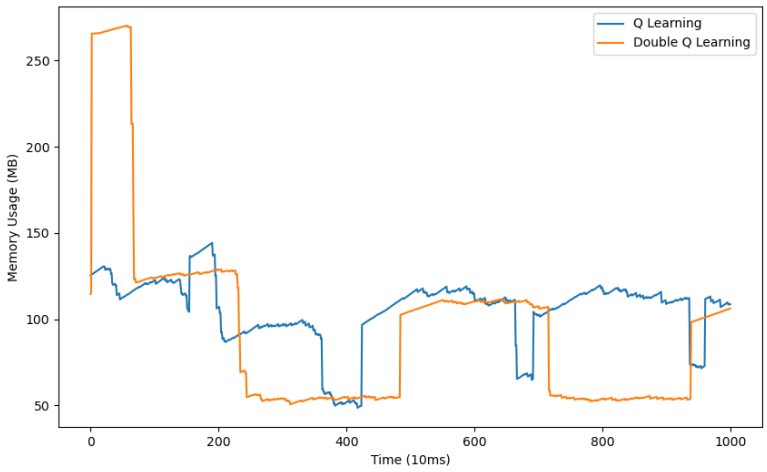


Figure 2.1: Memory utilization of Double Q-Learning, accentuating a sharp rise to around 250MB during the early episodes.

Underestimation Bias in Double Q-Learning

While Double Q-Learning mitigates overestimation bias, it is susceptible to underestimation bias, a phenomenon rigorously examined by Ren et al.[22]. This bias is not merely a theoretical issue but one with significant operational implications. Specifically, it leads to the emergence of multiple non-optimal fixed points, analogous to saddle points in non-convex optimization landscapes. These fixed points serve as barriers to finding the optimal policy and undermine the algorithm's efficiency. Notably, underestimation bias remains prevalent even in popular variants like Clipped Double Q-Learning[23, 24, 25], highlighting its systemic nature. Such bias introduces discontinuities in the learning trajectory, exacerbating issues already prevalent in Temporal Difference learning paradigms.

Chapter 3

Smoothed Q-Learning

In this chapter, we provide a comprehensive analysis of Smoothed Q-Learning, a novel extension of standard Q-Learning that leverages statistical smoothing techniques[1]. Our discourse progresses through multiple dimensions: the historical amalgamation of statistical methods with reinforcement learning, the algorithmic intricacies of Smoothed Q-Learning, and rigorous mathematical proof of its convergence in the Tabular Case. We culminate with empirical evaluations in complex environments to contrast its performance against established methods like standard Q-Learning and Double Q-Learning.

3.1 Smoothed Q-Learning

Addressing the challenges presented, this paper offers an insightful exploration of a novel development known as Smoothed Q-Learning. This groundbreaking concept, introduced by Professor David Barber in March 2023, aims to stabilise the learning process and mitigate the overconfidence prevalent in Q-value estimations[1]. Smoothed Q-Learning revolutionises the field by introducing an innovative action probability distribution, denoted as $q_t(a|s_{t+1})$. This modification significantly deviates from the traditional Q-Learning's Q-value update rules and the action selection mechanism. The design's sophistication lies in facilitating more strategic action selection and a smoother learning curve, steering clear from the typical greedy selection approach[1].

3.1.1 Leveraging Statistical Uncertainty in Reinforcement Learning

The application of statistics in reinforcement learning has been profoundly established in past research. Within the multi-armed bandit problem, the Upper Confidence Bound (UCB) algorithm employs statistical estimates to discern which bandit machine holds the highest potential reward[26, 27]. This strategy does not only take into account the mean return of each machine but also contemplates the uncertainty associated with each of them. In Bayesian reinforcement learning, an agent utilises Bayesian statistics to gauge the unknown dynamics of the environment[28]. Rather than merely estimating the most plausible environment model, it also gauges its inherent uncertainty. Such profound embrace of statistical uncertainty enables reinforcement learning agents to adapt more efficaciously to intricate learning environments. Additionally, in algorithms involving gradient ascent or descent, comprehending the volatility in the return estimates becomes paramount[29, 30]. This involves utilising statistical metrics, such as variance, to measure the dispersion of returns generated by different strategies.

3.1.2 Concept

The idea of Smoothed Q-Learning hinges on the intricate balance between exploration and exploitation, particularly in the Q-value estimation process. Instead of focusing solely on the maximum Q-value, Smoothed Q-Learning involves a more nuanced understanding of the decision-making process. The critical feature of Smoothed Q-Learning lies in its ingenious adoption of an action probability distribution, denoted as $q_t(a|s_{t+1})$, during the Q-value updating phase. This unique approach provides a method for the system to estimate the Q-values in a more 'smoothed' manner, thereby mitigating the overconfidence that typically arises in Q-value estimation. At its core, Smoothed Q-Learning works by spreading the 'weight' or importance across a range of possible actions rather than concentrating solely on the action with the maximum estimated Q-value. This modified approach encourages a more measured and balanced process, thereby allowing the learning algorithm to gradually refine its understanding of the environment. As time progresses, this distribution increasingly focuses on the action that yields the highest Q-value, transitioning smoothly from a broad exploration of the action space towards a more focused exploitation of the optimal actions. This strategy results in more strategic action selection and a smoother learning trajectory, avoiding the pitfalls associated with overly greedy action selection[1].

Statistical View

The quintessence of Smoothed Q-Learning resides in its innovative replacement of the maximum operator with a statistical expectation operator in the Q-value update equation. This shift is not merely cosmetic; it addresses the inherent limitations of traditional Q-Learning algorithms that rely solely on the maximum Q-value, effectively focusing only on the mode of the Q-value distribution. Such a narrow focus can lead to overconfident and potentially misleading estimates. In contrast, Smoothed Q-Learning adopts a more holistic statistical approach. It considers not just the mode but also the expectation and variance of the Q-value distribution. This is grounded in the foundational principles of probability theory, where a probability space is defined by three elements: the sample space, the event set, and the probability function. In the context of reinforcement learning, the sample space includes all possible state-action pairs, the event set comprises all subsets of these pairs, and the probability function assigns probabilities to these events. By assigning a non-zero probability to each potential action via the $q_t(a|s_{t+1})$ distribution, Smoothed Q-Learning captures the inherent uncertainty and variability in the decision-making process. This nuanced approach allows the algorithm to strike a more effective balance between exploration and exploitation, two cornerstone concepts in reinforcement learning. Exploration involves the algorithm taking suboptimal actions occasionally to gather more information about the environment, while exploitation focuses on choosing the action that maximizes the expected reward based on current knowledge. Statistical theorems further bolster the efficacy of this approach. The **Law of Large Numbers** adds that with sufficient sampling, the average reward for a given state-action pair will converge to its true expected value[31]. In high-dimensional spaces, whether probabilistic or Euclidean, the probability mass tends to concentrate in specific regions rather than being uniformly distributed. Smoothed Q-Learning capitalizes on this phenomenon by allowing the algorithm to focus on a range of promising actions over time, rather than fixating prematurely on the action with the highest estimated Q-value.

Smoothed Q-Learning Algorithm

The update equation for Smoothed Q-Learning is represented as:

$$\tilde{Q}_{t+1}(s_t, a_t) = \tilde{Q}_t(s_t, a_t) + \alpha_t(s_t, a_t) \left[r_t + \gamma \sum_a q_t(a|s_{t+1}) \tilde{Q}_t(s_{t+1}, a) - \tilde{Q}_t(s_t, a_t) \right] \tag{3.1}$$

Where:

- $\tilde{Q}_{t+1}(s_t, a_t)$ is the updated value of the action-value for action a_t in state s_t at the next time step $t+1$.
- $\tilde{Q}_t(s_t, a_t)$ represents the current estimate of the action value for action a_t in state s_t at time t.
- $\alpha_t(s_t, a_t)$ is the learning rate at time t, which determines how quickly new information supersedes old information.
- r_t is the immediate reward received at time t.
- γ is the discount factor determining the present value of future rewards.
- $\sum_a q_t(a|s_{t+1}) \tilde{Q}_t(s_{t+1}, a)$ represents the expected action-value over all possible actions in the next state s_{t+1} at time t, weighted by the action probability distribution $q_t(a|s_{t+1})$.

This unique feature of the Smoothed Q-Learning algorithm allows for a smoother and more balanced learning process, gradually increasing the focus on the action that yields the highest Q-value over time, leading to a more strategic action selection and smoother learning trajectory[1].

Algorithm 3 Smoothed Q-Learning Algorithm

Input: Q-values matrix $\tilde{Q}(s, a)$ for all states s and actions a, starting state s, action distribution $q_t(a|s)$, learning rate α, discount factor γ, and exploration probability ϵ.
Output: Updated Q-values matrix $\tilde{Q}(s, a)$

Initialization:
  Initialize $\tilde{Q}(s, a)$ for all s, a and starting state s
do
  Choose action a_t based on ϵ-greedy: Draw u from a uniform distribution in $[0, 1]$
if $u > \epsilon$ **then**
    $a_t = \arg\max_a \tilde{Q}_t(s, a)$ ▷ If there is more than one optimal action, select from them randomly
else
    Select action a_t at random from the action space
  Take action a_t, observe reward r and new state s'
  Update $\tilde{Q}$ value:
    $\tilde{Q}(s, a) \leftarrow \tilde{Q}(s, a) + \alpha \left[r + \gamma \sum_{a'} q_t(a'|s') \tilde{Q}(s', a') - \tilde{Q}(s, a) \right]$
    $s \leftarrow s'$
while termination condition is met

Smoothing Strategies

In Smoothed Q-Learning, the smoothing strategy is encapsulated by the action probability distribution $q_t(a|s_{t+1})$, which serves as a pivotal component for the efficacy of the algorithm[1]. The essence of the smoothing strategy is a probability density function designed to balance exploration and exploitation as time progresses. Specifically, as the learning process unfolds (i.e., as t increases), the probability distribution $q_t(a|s_{t+1})$ increasingly concentrates its mass on the action

$a^* = \arg\max_a \tilde{Q}_t(s_t, a)$ that maximises the estimated action-value. Formally, this distribution places a mass of $1 - \delta_t$ on the optimal action a^*, where δ_t is a parameter that diminishes towards zero as t increases. This property ensures that Smoothed Q-Learning gradually converges to the optimal Q-value[1]. Therefore, the judicious selection of a smoothing strategy not only facilitates more effective action choices but also guarantees the agent's reliable convergence to the optimal Q-value.

1. **Softmax**:
$$q_t(a|s_{t+1}) \propto e^{\beta_t \tilde{Q}_t(s_{t+1},a)}$$
As β_t increases over time, the distribution becomes more concentrated around the action with the maximal Q-value[1].

2. **Clipped Max**: A straightforward methodology is to allocate $1 - \tau_t$ probability mass to the most probable action and distribute the remaining τ_t evenly among the non-maximal actions. Given there are A actions in state s_{t+1}:
$$q_t(a^*|s_{t+1}) = 1 - \tau_t$$
$$q_t(b|s_{t+1}) = \frac{\tau_t}{A-1}$$
where a^* signifies the action that maximises Q and b denotes any alternative action[1].

3. **Clipped Softmax**: The Softmax policy for action selection is traditionally described as:
$$\pi(a|s) = \frac{e^{\beta_t Q(s,a)}}{\sum_{a'} e^{\beta_t Q(s,a')}}$$
Where in:
   - $\pi(a|s)$ characterises the likelihood of opting for action a when in state s.
   - β_t is a positive scalar known as the temperature parameter. It influences the decisiveness of action selection. Larger β values yield a policy with heightened determinism, whereas smaller values encourage exploratory decisions.

   To augment policy stability and efficacy, particularly in expansive action spaces, the Clipped Softmax strategy is introduced. This technique is grounded in the computation of softmax exclusively over the paramount k Q-values associated with a state while relegating the remainder to an inconsequential value, such as $-\infty$.

   The methodology for Clipped Softmax is delineated as:

   (a) Ascertain the top k Q-values for a given state.

   (b) Reassign Q-values not included in the top k to $-\infty$.

   (c) Utilise the conventional softmax function on these adjusted Q-values.

   Mathematically, this can be articulated as:
$$Q_{\text{clipped}}(s,a) = \begin{cases} Q(s,a) & \text{if } a \text{ resides among the top } k \text{ actions for } s \\ -\infty & \text{otherwise} \end{cases} \tag{3.2}$$

Thus, the Clipped Softmax equation becomes:

$$q_t^{\text{clipped}}(a|s_{t+1}) = \frac{e^{\beta_t Q_{\text{clipped}}(s_{t+1},a)}}{\sum_{a'} e^{\beta_t Q_{\text{clipped}}(s_{t+1},a')}} \tag{3.3}$$

By incorporating Clipped Softmax, the scope of actions under evaluation is constricted, yet exploratory tendencies remain intact since the highest Q-value is not selected with absolute certainty.

4. **Future Choices**: Alternative smoothing strategies may encompass the frequency with which a specific state-action pair is visited. Such strategies necessitate rigorous exploration.

Convergence of Smoothed Q-Learning

Smoothed Q-Learning utilizes an update equation Q_t^{upd} to iteratively refine the Q-value function. The update incorporates a stochastic term governed by a time-dependent distribution $q_t(a|s_{t+1})$. This distribution increasingly focuses on actions a maximizing $\tilde{Q}_t(s_t, a)$, modulated by a decaying parameter δ_t that approaches zero as $t \to \infty$[1].

Formal Convergence Theorem The convergence of Smoothed Q-Learning is substantiated by a theorem concerning the asymptotic behavior of $\Delta_t = \tilde{Q}_t(s_t, a) - Q^*(s_t, a)$. The theorem posits that Δ_t converges to zero in expectation under specific conditions: a decaying learning rate α_t and a bounded update function F_t. The proof establishes that these conditions are intrinsically satisfied in the algorithm, ensuring bounded expectation and variance for F_t[1].

Implications in the Tabular Case Under the aforementioned theorem, Smoothed Q-Learning is proven to converge in finite, discrete state-action spaces ($|S \times A| < \infty$). The theorem validates both the theoretical and practical efficacy of the algorithm. Moreover, the decay parameter δ_t is identified as a pivotal factor steering the algorithm towards the optimal policy π^* asymptotically.

3.1.3 Analysis of Smoothing Strategies

Softmax Strategy

Applicability: The Softmax strategy is particularly effective in scenarios where the action space is limited and each action has a discernible impact on the environment. It is also beneficial in complex environments that necessitate a comprehensive exploration of the action space.
Advantages:

- Provides a nuanced balance between exploration and exploitation.
- Considers the Q-values of all possible actions, rather than focusing solely on the maximal Q-value.

Disadvantages:

- In high-dimensional action spaces, the computational burden becomes significant due to the exponential nature of the Softmax function.
- The strategy may lead to excessive exploration, thereby slowing down the learning process.

Clipped Max Strategy

Applicability: The Clipped Max strategy is well-suited for large action spaces where only a subset of actions are meaningful. It offers rapid convergence compared to the Softmax strategy.
Advantages:

- High computational efficiency due to the clipping mechanism.
- Faster convergence to an optimal or near-optimal strategy.

Disadvantages:

- Risk of overlooking valuable sub-optimal actions due to the focus on the maximal Q-value.
- Limited exploratory capabilities owing to its rapid convergence.

Clipped Softmax Strategy

Applicability: The Clipped Softmax strategy is advantageous in large action spaces where only a few actions are significant. It offers a balanced approach to exploration and exploitation.
Advantages:

- Maintains computational efficiency while providing a more balanced exploration.
- Focuses on actions that are most likely to be beneficial, thereby optimizing the learning process.

Disadvantages:

- Parameter tuning can be complex due to the hybrid nature of the strategy.
- Requires extensive experimentation for optimal parameter selection.

3.1.4 Temporal Continuity in Smoothed Q-Learning

Temporal Difference (TD) learning serves as a cornerstone for predictive models in reinforcement learning, emphasizing the importance of temporal continuity for stable learning trajectories [13]. Traditional Q-Learning, a quintessential TD-based algorithm, suffers from overestimation bias. Double Q-Learning, designed to mitigate this bias, introduces a bifurcation in the Q-value update mechanism, inadvertently leading to underestimation and disrupting temporal continuity [22]. Smoothed Q-Learning addresses this issue by refining the Q-value update process. It employs a stochastic component in the update equation, governed by a time-varying distribution $q_t(a|s_{t+1})$, to ensure a more consistent and temporally coherent learning trajectory. This enhancement preserves the temporal continuity essential for the stability and robustness of TD-based learning algorithms. Thus, Smoothed Q-Learning not only adheres to but also refines the foundational principles of TD learning, ensuring a robust and temporally continuous learning process.

3.2 Experiment for comparison with Q-Learning, Double Q-Learning and Smoothed Q-Learning

Under the guidance of Professor David Barber, a series of experiments replicating Q-learning and its extended forms were conducted, which include MDP, Roulette, Grid World. Furthermore, the performance of various learning algorithms was evaluated across environments of differing complexity.

3.2.1 MDP

This experiment stems from replicating David Barber's experiment in his Smoothed Q Learning parper[1].

Description of the experimental environment

In the experiment's setting, the agent is positioned within a simplified Markov Decision Process (MDP). Broadly speaking, this environment encompasses four states: A, B, C and D, paired with several actions. An illustrative representation of this MDP environment is depicted in Figure 3.1.

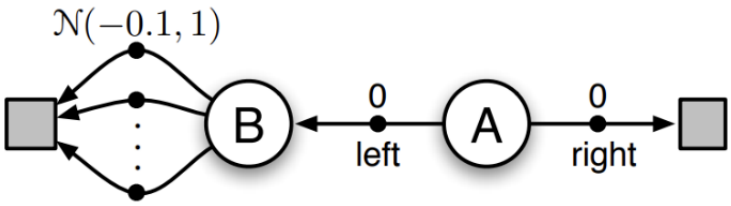


Figure 3.1: Description of MDP Experiment.

Initial State The agent commences in state A. Within state A, the agent is presented with two options:

1. **Move Right:** This action transports the agent to the terminal state C, yielding a reward of 0.
2. **Move Left:** This leads the agent to state B, also conferring a reward of 0.

State B Actions Upon reaching state B, the agent has a repertoire of eight possible actions. Regardless of the action selected, the agent is transitioned to the terminal state D. Here, the reward is drawn from a Gaussian distribution with a mean of -0.1 and a variance of 1.

Experimental Objective and Evaluation

The principal objective of this experiment is to assess and compare various Q-learning methodologies, including the standard Q-learning, double Q-learning, and two distinct strategies of smoothed Q-learning. To evaluate the performance of these methods, we measure two key metrics and plot the corresponding graphs:

1. **Mean Absolute Error (MAE):** This characterises the discrepancy between the predicted values of the Q-matrix and its actual values following each exploration. We calculated the MAE for all experiments and plotted a graph illustrating how this error fluctuates with an increase in the number of experimental trials. The computational method involved determining the absolute difference between the value of the Q-matrix for a specific state 'a' and action 'l' after each exploration and the actual value, followed by averaging all these values.
2. **Mean Action Selection:** This delineates the average number of times an agent selects a particular action given a number of experiments and explorations. For instance, the average number of times the 'left' action is chosen. For each method, the average number of times an

agent chose a specific action across all experiments was calculated and a graph depicting the evolution of this quantity with an increase in experimental trials was plotted. The method comprised determining the number of times an agent selected an action in each experiment, followed by averaging all these values.

Experimental Setup

- `rv`: A reward value set at -0.1. It signifies the negative reward acquired in a particular state-action combination.
- `gamma` (Discount Factor): Set at 0.99, this parameter represents how much importance is given to future rewards. A value close to 1 will give significance to long-term rewards.
- `alpha0` (Learning Rate): Initialized at 0.1, this determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the latest information.
- `eps` (Exploration Rate): Set at 0.1, this rate signifies the agent's probability of choosing a random action over the one that the Q-function suggests. It's essential for the exploration-exploitation trade-off.
- `Exps`: The number of experiments is set at 10,000. It represents the number of trials the algorithm will undergo to learn the Q-values.
- `Episodes`: Represents the number of episodes per experiment, set at 2000.

Moreover, the code utilizes a decreasing α as α `= `α_0 `/ (1 + 0.001 * t)`, where `t` increases with time. This adjustment ensures that the learning rate diminishes over time, allowing the algorithm to stabilize its predictions. The β in Smoothed Q-learning (softmax) is an increasing function of episodes, defined as β `= 0.1 + 0.1 * (episode - 1)`. It tweaks the level of exploration in the action selection process, ensuring a balance between exploration and exploitation. For Smoothed Q-learning(Clipped max), μ is a decreasing function of episodes, defined as μ `= exp(-0.02 * episode)`, which dictates the likelihood of an action being selected over others. The parameter `A`, representing the number of actions, is set dynamically based on the Q-values for the current state. These hyperparameters, together with the algorithms, allow the agent to learn optimal policies while navigating through various state-action pairs in the environment.

Code Structure

The code commences by defining the requisite functions, such as the state transition function `rs` and the reward function `rbar`. Subsequent to this, the Q-matrix is initialised, along with other necessary variables.
For each Q-learning methodology, the code performs the following steps:

1. For each experiment, initialise the Q-matrix and execute 10000 explorations.
2. Within each exploration:
   (a) The agent embarks from state A.
   (b) The exploration concludes upon reaching terminal states C or D.
   (c) Update the Q-matrix based on the action undertaken and the reward procured.
3. Compute and archive metrics pertinent to the experimental objectives.

Evaluation

Figure 3.2 provides a compelling visualization of the convergence trajectories across various Q-learning strategies, including standard Q-Learning, Double Q-Learning, and Smoothed Q-Learning with two different smoothing strategies—softmax and clipped max.

Convergence Rate and Bias Standard Q-Learning is characterized by a notable overestimation bias in the initial phases of training, a feature mitigated in Double Q-Learning. Conversely, Double Q-Learning exhibits a higher convergence point relative to its alternatives, indicating a proclivity for underestimation or other intrinsic limitations. Smoothed Q-Learning, employing softmax and clipped max strategies, outperforms Double Q-Learning in terms of policy optimality. Specifically, the softmax variant demonstrates rapid and smooth convergence to the optimal policy following an initial period of oscillatory exploration. The clipped max strategy, while initially manifesting transient overestimation, swiftly rectifies this, culminating in a stable convergence trajectory.

Comparative Robustness In terms of robustness, Double Q-Learning achieves rapid convergence but not without complications potentially related to underestimation. Smoothed Q-Learning, although slightly slower in converging compared to Double Q-Learning, offers a stable trajectory and avoids the overestimation trap that plagues standard Q-Learning.

Implications These empirical results validate the theoretical underpinnings of Smoothed Q-Learning, confirming its effectiveness in maintaining the balance between convergence rate and bias. They also underscore the algorithm's fidelity to the core principles of temporal continuity, as it presents a consistent and stable learning trajectory over time. Thus, Smoothed Q-Learning emerges as a comprehensive solution that addresses both the overestimation and underestimation biases while retaining temporal continuity, thereby enhancing the reliability and robustness of the Q-learning paradigm.

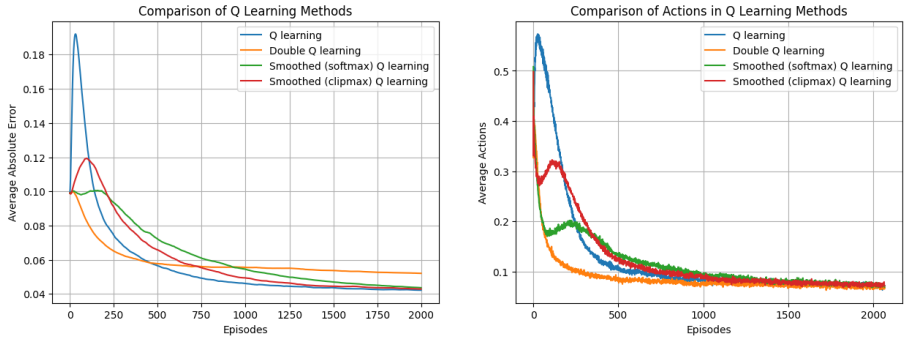


Figure 3.2: Convergence trajectories of different Q-learning strategies.

3.2.2 Roulette

The author then tries to reproduce the experiment on Roulette, which is mentioned in Hado Van Hasselt's article in Double Q Learning[2].

Description of the experimental environment

The RouletteEnv class is a simulation of the classic game of Roulette, tailored to explore reinforcement learning dynamics within this game environment. This environment offers a simplified version of the game of roulette. Each round, a player can place one of several possible bets which have distinct payouts, and then the outcome is determined by a spinning ball that eventually lands in one of the numbered pockets. The environment affords a gamut of 171 discrete actions.

Action Space The environment provides a total of 171 discrete actions, which are categorized as follows:

Table 3.1: Overview of Action Space in the Environment

Action Type	Description	Number of Actions
Individual Numbers	Stake on numbers 0-35	36
Colour	Stake on red or black	2
Parity	Stake on even or odd	2
Range	Stake on low (1-18) or high (19-36)	2
Dozen Intervals	Stake on 1-12, 13-24, or 25-36	3
Columns	Stake on one of three columns	3
Split Bets	Stake on adjacent dual numbers	Variable
Street Bets	Stake on columnar intersections	Variable
Ceasing Play	Termination of engagement	1

Game Mechanics At the commencement of each game round, the player selects an action from the action space, denoting their chosen stake. Subsequently, a winning number is randomly determined, simulating the ball's rotation and its eventual resting position within a pocket. The pay-out ratios are then ascertained based on traditional roulette rules:

- For bets on individual numbers that prove accurate, one stands to win 34 times their stake.
- Bets on red, black, even, odd, low, or high that are successful simply return the original stake, leading to a reward of 0.
- Both dozen-range and column bets yield a return of twice the stake.
- Split and street bets, when successful, offer a return of 16 times the stake.

Experimental Objective and Evaluation

In the following experiment, our main objective is to explore and evaluate the performance of various Q-learning approaches on a specially designed Roulette environment.

Evaluation Metrics

To quantify the performance and progression of our Q-learning models, we employ the following evaluation metrics:

- **Absolute Errors:** It measures the mean absolute difference between successive Q-tables for each episode. This metric helps to discern the learning stability of our models across episodes.

- **Average Action Values:** It tracks the average Q-value of actions over time, which indicates how our model's perception of the potential reward changes as it explores and exploits the environment.

Experimental Setup We conduct our experiments over a series of 100,000 episodes. The hyperparameters utilized in the experiments include:

- **Learning rate:** Determined by a polynomial function with a base of 0.2 and an exponent of 0.5.
- **Epsilon:** Used in epsilon-greedy action selection. It starts at 1.0 and decays to 0.1 with a decay rate of 0.0001.
- **Gamma:** Discount factor set to 0.99, determining how much importance we want to give to future rewards.
- **Delta:** For the clipped max action selection in smoothed Q-learning, begins at 1.0 and decays similarly to epsilon.

Code Structure

This section outlines the code architecture for Smoothed Q-learning, focusing on the integration of Softmax and Clipped max strategies.

- **Environment:** A bespoke environment, `RouletteEnv`, is created, inheriting properties from `gym.Env`. This mimics roulette game dynamics with 171 discrete actions.
- **Q-Learning:** Utilises a Q-table and updates Q-values using the Bellman equation.
- **Double Q-Learning:** An extended form of Q-learning with two Q-tables. This reduces overestimation in Q-values.
- **Smoothed Q-Learning:** The novelty lies in action selection strategies:
  - **Softmax:** Provides a probabilistic distribution over actions based on their Q-values.
  - **Clipped Max:** Prioritizes the maximum Q-value action while assigning a minimal selection probability (δ) to others.
  - **Clipped Softmax:** A variant of Softmax focusing on the top-k actions, reducing the influence of suboptimal actions.
- **Results:** Metrics such as absolute errors and average action values are collected and can be visualised through `matplotlib`.

Evaluation

Experimental Design and Metrics To ensure a robust analysis, we conduct evaluations in the Roulette environment, employing two distinct metrics: Expected Profit and Absolute Error Value. According to established benchmarks, an optimal Expected Profit of 0.947 is documented [2].

Analysis: Expected Profit *Results:* Figure 3.3 shows that traditional Q-Learning and Smoothed Q-Learning with softmax diverge significantly from the optimal Expected Profit. Traditional Q-Learning plateaus at an Expected Profit of approximately 10. *Insights:* Its high Absolute Error value suggests that the algorithm tends to overestimate Q-values, particularly in initial interactions.

Analysis: Smoothed Q-Learning with Softmax *Results:* This strategy shows marginal improvements over traditional Q-Learning, converging to an Expected Profit of about -6 and an Absolute Error around 6. *Insights:* Although it performs better than traditional Q-Learning, it still falls short of optimal performance.

Analysis: Double Q-Learning *Results:* As shown in Figure 3.4, Double Q-Learning stabilizes near an Expected Profit of -4 and an Absolute Error around 4. *Insights:* This suggests a systematic underestimation, making it suboptimal compared to certain Smoothed Q-Learning strategies.

Analysis: Smoothed Q-Learning with Clipped Softmax *Results:* This strategy converges closest to the expected optimal profit and demonstrates rapid convergence. *Insights:* The results validate our hypothesis that Smoothed Q-Learning mitigates both overestimation and underestimation issues found in traditional and Double Q-Learning, respectively.

Concluding Remarks Our empirical analysis substantiates our theoretical insights, establishing the superiority of Smoothed Q-Learning with Clipped Softmax in complex environments like Roulette. It excels particularly in the rate of convergence and minimization of estimation errors.

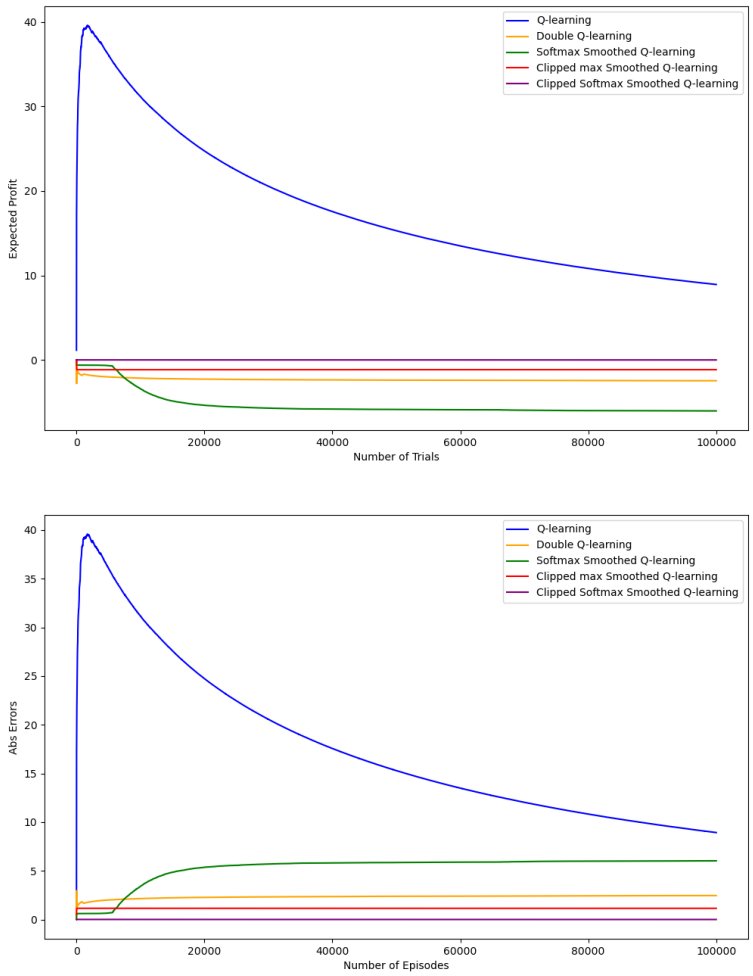


Figure 3.3: Experimental Result in Roulette Environment

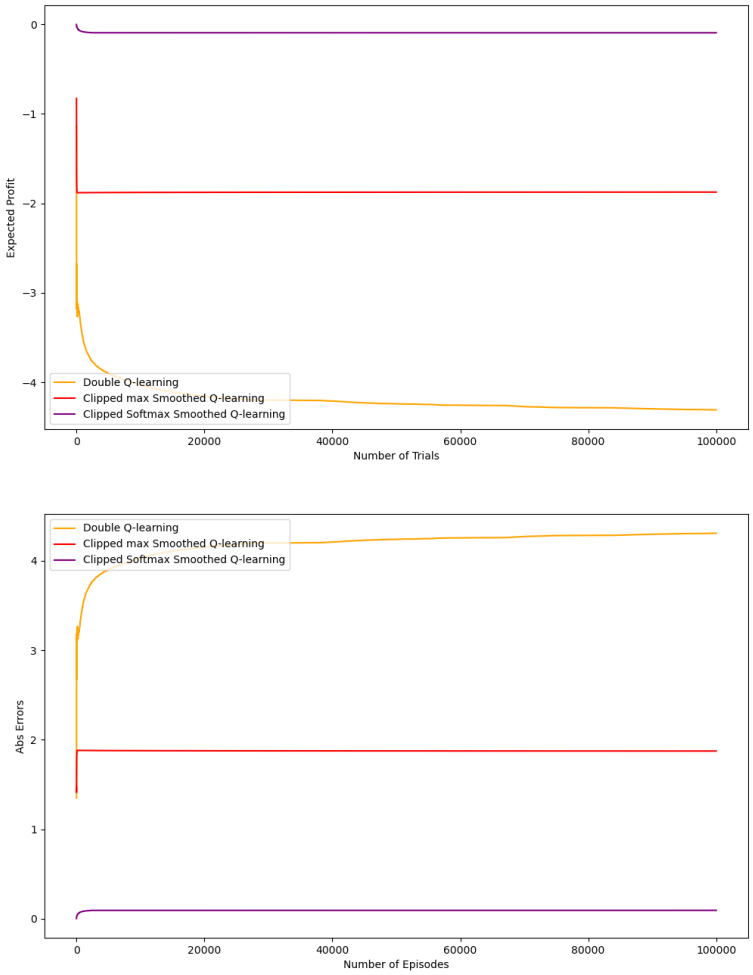


Figure 3.4: Detail of Double Q-learning and Smoothed Q-learning (Clipped max and Clipped Softmax)

3.2.3 Grid World

Gridworld, a quintessential reinforcement learning testbed, has been experimentally replicated by the authors following Hado van Hasselt's article[2].

3.2.4 Description of the experimental environment

The experimental setup is modeled as a *GridWorld* environment. It is essentially a two-dimensional grid where each cell represents a state and the agent can move between these states. Key characteristics of this environment include:

- **Grid Size:** The world is structured as a square grid with dimensions provided by `grid_size`, which defaults to 3×3.

- **Start Position:** The agent commences its journey from the top-left corner, i.e., position (0,0).
- **End Position:** The bottom-right corner, position (`grid_size - 1`, `grid_size - 1`), serves as the goal. Reaching this position rewards the agent with a special `goal_reward`.
- **Actions:** The agent can make four potential moves at each step:
  - Action 0: Move up (if not on the top boundary).
  - Action 1: Move right (if not on the right boundary).
  - Action 2: Move down (if not on the bottom boundary).
  - Action 3: Move left (if not on the left boundary).
- **Rewards and Penalties:** After each action, the agent can receive either a reward or penalty based on the following:
  - If the agent reaches the goal, it is rewarded with `goal_reward` which is set to 5.
  - If the agent exhausts the defined `max_steps` (set to 5) without reaching the goal, the episode terminates with zero reward.
  - Otherwise, at each step, the agent has a 0.5 probability of receiving a `reward` (set to 10) or a `penalty` (set to -12).
- **Adaptive Exploration Based on State Visits:** The exploration rate, ϵ, is dynamically adjusted according to the frequency of visits to a specific state. This ensures that lesser-visited states are explored more intensively. The ϵ for a state is computed as:

$$\epsilon(s) = \frac{1}{\sqrt{n_s + 1}}$$

  where n_s denotes the number of visits to state s.

Experimental Objective and Evaluation

Evaluation Metrics:

- *Average Reward*: Serving as a robust gauge of learning efficacy, the Average Reward metric is computed by aggregating the rewards accrued up to a specific episode and normalizing this sum by the episode count. According to existing literature, an optimal average reward of $+0.2$ per step is indicative of the agent having mastered the optimal strategy[2]. Formally, for any given episode i, the Average Reward is calculated as:

$$\textit{Average Reward}(i) = \frac{\sum_{j=1}^{i} \text{rewards}(j)}{i}$$

  where rewards(j) denotes the reward obtained in episode j. A higher Average Reward value signifies that the learning algorithm is effectively guiding the agent towards accumulating greater rewards on average.
- *Max Action Value at Start State*: This metric provides a perspective on the learned quality of actions from the starting state. It reveals the agent's estimate of the expected future reward for taking the best action from the start state. By the paper of Hado, the highest-valued action's optimal value in the initial state stands at 0.36[2].

Experimental Setup:

- *Environment*: The experiments were carried out in a 3×3 GridWorld environment.
- *Learning Rate (α)*: The learning rate is dynamically adjusted based on the frequency of the state-action pair visitations. It is represented by the equation

$$\alpha = \frac{1}{N_{\text{table}}[\text{state}][\text{action}]}$$

- *Exploration Policy (Epsilon-greedy)*: An epsilon-greedy strategy is employed to balance exploration and exploitation. The value of epsilon decays as

$$\epsilon = \frac{1}{\text{episode} + 1}$$

- *Experiments*: For each of the learning methods, experiments are run for 10000 episodes with a maximum of 5 steps per episode. The entire set of experiments is repeated 1000 times to gather more statistically significant results.

Code Structure

GridWorld Environment A class titled `GridWorld` is defined, which sets the scene for an agent to navigate a square grid.

Q-learning and Double Q-learning Both of these methods employ an epsilon-greedy policy for action selection and dynamically update their Q-values based on rewards received and the current state-action pairs.

Smoothed Q-Learning

1. **Softmax Strategy**:
   - The `softmax(x)` function computes a probability distribution over the Q-values.
   - This distribution facilitates a balanced approach between exploration and exploitation.
2. **Clipped Max Strategy**:
   - The `clipped_max(Q_values, tau)` function allocates a uniform probability to non-maximal actions and a higher probability to the maximal action.
   - This strategy, while simpler, still incorporates a degree of exploration.
3. **Clipped Softmax Strategy**:
   - The `clipped_softmax(Q_values, beta)` function concentrates on the top-k actions, thereby diminishing the influence of suboptimal actions.
   - This strategy is especially beneficial when the action space is large but only a subset of actions are significant.

Evaluation

Experimental Design To ensure statistical rigor, 10,000 experiments are conducted, each comprising 10,000 learning steps. The optimal agent is expected to converge to an average reward of approximately 0.2, based on established benchmarks [2].

Performance Metrics Key performance indicators such as "average reward" and "maximum action values" are computed per learning step for the initial state and averaged across 10,000 experiments.

Analysis: Average Reward *Results:* Smoothed Q-Learning with Clipped Softmax is disqualified due to its suboptimal convergence at an average reward of -4. Standard Q-Learning converges to -3.5, while Smoothed Q-Learning with Softmax reaches -3. *Insights:* Double Q-Learning and Smoothed Q-Learning with Clipped Max both approach -0.5, with the latter surpassing Double Q-Learning at 7,000 steps and approximating the theoretical optimal average reward of 0.2.

Analysis: Maximum Action Values *Results:* Q-Learning and Smoothed Q-Learning variants with Softmax and Clipped Softmax strategies converge to a significantly higher value than the expected 0.36. *Insights:* This likely disrupts the optimal exploration-exploitation trade-off. Smoothed Q-Learning with Clipped Max ascends from an initial value of -3 to zero, while Double Q-Learning declines from 7 to -4.

Discussion *Comparative Analysis:* Double Q-Learning and Smoothed Q-Learning with Clipped Max offer superior convergence properties in the Grid World environment. *Limitations:* While Double Q-Learning corrects over-optimistic estimates inherent in Q-Learning, its tendency to underestimate affects long-term performance.

Concluding Remarks Our analysis corroborates that Smoothed Q-Learning with the Clipped Max strategy emerges as the most effective algorithm in the Grid World setting, substantiating its broader applicability and superiority over other variants.

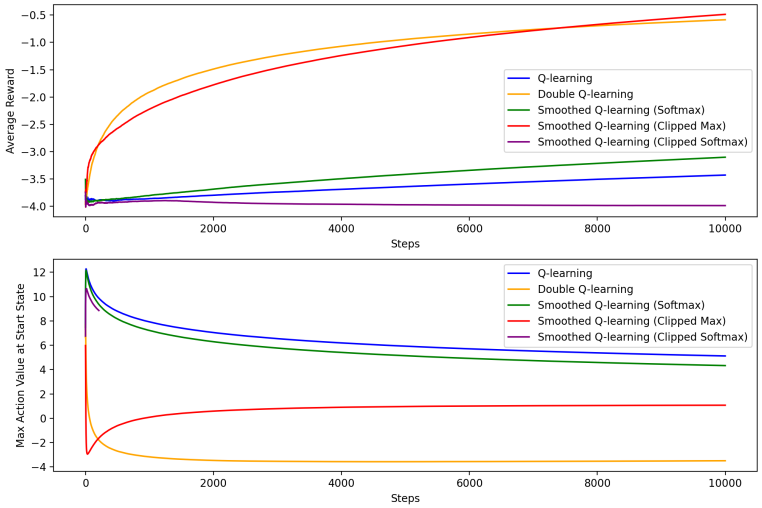


Figure 3.5: Q Learning, Double Q Learning, Smoothed Q Learning (Softmax and Clipped Max)

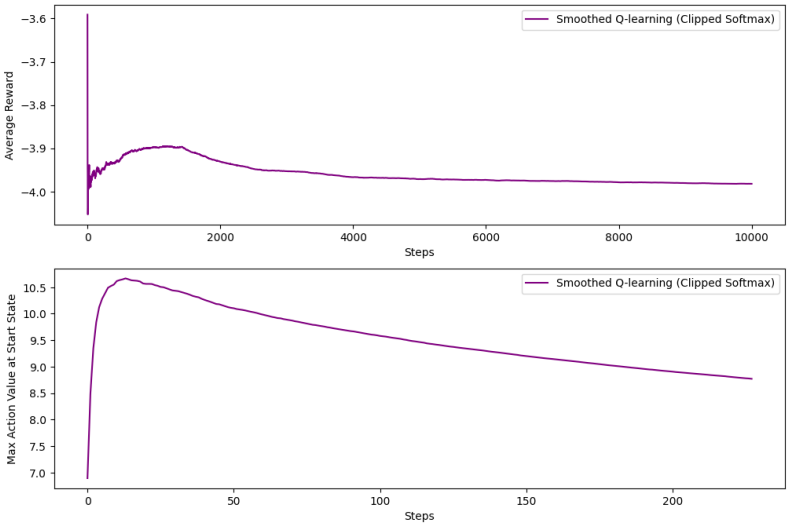


Figure 3.6: Smoothed Q Learning (Clipped Softmax)

3.3 Chapter Summary

Chapter 3 offers an in-depth exploration of Smoothed Q-Learning, a groundbreaking innovation by Professor David Barber. This avant-garde methodology seeks to transcend the constraints of conventional Q-Learning algorithms by integrating statistical uncertainty into the learning equation. Unlike its traditional counterparts, which solely prioritize the maximum Q-value, Smoothed Q-Learning utilizes a specialized action probability distribution, denoted as $q_t(a|s_{t+1})$, for Q-value updates. This innovative approach enables the algorithm to strike a more effective balance between exploration and exploitation. The chapter meticulously unpacks the statistical foundations that underpin this approach. It underscores the algorithm's reliance on seminal statistical principles, such as the Law of Large Numbers, as mechanisms to ensure the robustness and reliability of the

learning process. Additionally, the chapter elucidates the updated equation specific to Smoothed Q-Learning and provides a comparative analysis of various smoothing strategies, namely Softmax, Clipped Max, and Clipped Softmax. Each strategy is evaluated based on its unique merits and limitations, offering readers a comprehensive understanding of their applicability in different scenarios. In conclusion, the chapter validates the efficacy of Smoothed Q-Learning through its proven ability to converge to an optimal policy. This establishes the algorithm as a robust tool for tackling intricate reinforcement learning challenges. The chapter culminates with an empirical comparison between Tabular Case-based Q-Learning, Double Q-Learning, and Smoothed Q-Learning across three distinct experimental environments of varying complexity. Through rigorous experimental validation and a detailed comparative analysis, it is concluded that Smoothed Q-Learning, when equipped with an appropriate smoothing strategy, outperforms the other two algorithms in terms of both efficiency and effectiveness.

Chapter 4

Smoothed Deep Q-Networks

In this chapter, we shall delve further into the theoretical underpinnings and algorithmic intricacies of Smoothed Q-Learning within the realm of Deep Reinforcement Learning. This exploration centres on a hybrid approach that Smoothed Q-Learning with Deep Q-Networks. We aim to integrate the update formula of Smoothed Q-Learning into deep reinforcement learning frameworks to assess its algorithmic efficacy in more complex, high-dimensional spaces. The chapter unfolds against the backdrop of extensive research in Deep Reinforcement Learning, Deep Q-Networks, and Double Deep Q-Networks, thereby setting the stage for a comprehensive discourse on the theory and implementation of Smoothed Deep Q-Networks. In conclusion, we shall empirically validate the performance of this novel algorithm through a rudimentary CartPole experiment.

4.1 Deep Reinforcement Learning

Originating at the confluence of neuroscience, behavioural science, and machine learning, Deep Reinforcement Learning (DRL) has been engineered to enable RL agents to navigate high-dimensional, intricate environments [19]. Capitalising on the strides made in deep learning—which excels in abstracting high-level features from raw sensory data—DRL has catalysed monumental advancements in fields such as computer vision [32, 33, 34] and speech recognition [35, 36, 37]. However, the formidable task of making optimal decisions in environments replete with a plethora of potential states and actions has presented RL with significant challenges [38]. DRL amalgamates the feature extraction prowess of deep learning with the robustness of reinforcement learning in model-free learning from reward signals. Despite its demonstrable efficacy across diverse domains, including gaming, robotic control, and financial applications [39, 40, 41], DRL is fraught with complexities. Specifically, it must contend with sparse, noisy, and temporally delayed scalar reward signals, as well as sequences of highly correlated states—this stands in stark contrast to the abundant, independent, hand-labelled training data typically employed in deep learning [42]. At its nucleus, DRL utilises Deep Neural Networks (DNNs) as function approximators to gauge the value of actions or states within a given environment, thereby underpinning its decision-making paradigm [6]. Nevertheless, the training of DRL algorithms poses a unique set of challenges, such as the exploration-exploitation trade-off, function approximation, policy oscillation, and sample inefficiency, all of which must be judiciously addressed to ensure successful learning [43]. Amongst the myriad of DRL techniques, Q-Learning serves as a linchpin, undergirding numerous DRL algorithms like the Deep Q-Networks (DQN).

4.1.1 Deep Q-Networks(DQN)

Background

In stochastic decision processes, the action-value function $Q^\pi(s,a)$ is pivotal, encapsulating the expected cumulative discounted reward when executing action a in state s under policy π [4]. It is rigorously defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right],$$

where γ is the discount factor constrained to $[0,1]$.
Conventional tabular methods, which maintain explicit representations of $Q(s,a)$ or $V(s)$, are computationally untenable for state-action spaces with high cardinality or continuity [19]. This computational intractability necessitates the use of Deep Q-Networks (DQN) to approximate these functions in complex domains.
To mitigate this computational challenge, function approximation methods are invoked. In this context, $Q(s,a)$ is approximated by $\hat{Q}(s,a;\mathbf{w})$, where $\mathbf{w}$ denotes the parameter vector [19]. Neural networks are the de facto standard for this approximation, given their compatibility with the DQN framework.
The task of updating $Q(s,a)$ is thus reformulated as an optimization problem, targeting the minimization of the Mean Squared Error (MSE) loss function $J(\mathbf{w})$:

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[\left(\hat{Q}(s,a;\mathbf{w}) - Q^\pi(s,a) \right)^2 \right].$$

Gradient-based optimization algorithms are typically employed to solve this problem [44].
This optimization framework serves as the backbone for both the DQN algorithm and its advanced variants, including the Stochastic DQN (SDQN), which are the focal points of this paper. These variants aim to optimize the function approximation and optimization procedures, thereby augmenting the efficacy and stability of the base DQN algorithm.

Concept

Deep Q-Networks (DQN), an innovatory algorithm within the Deep Reinforcement Learning (DRL) domain, capitalises on the superior efficacy of deep learning in approximating the Q-function in spaces characterised by high dimensionality [19]. The distinctive merit of DQN lies in its fusion of Q-Learning, a value-based approach in RL, with deep neural networks. Essentially, DQN employs these networks as function approximators for the assessment of the Q-value of each state-action pair within a given environment. Underpinning Q-Learning is the concept of the Q-value and the Bellman equation. The Q-value estimates the expected return for a particular action selected in a given state following a certain policy, whereas the Bellman equation provides a recursive definition for the optimal Q-function [45]. DQN advances this conceptual framework by utilising a deep neural network to represent the Q-function, hence enabling the generalisation across a state-action space of high dimensionality[42].

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-) \tag{4.1}$$

Algorithm 4 Deep Q-Learning with Experience Replay Algorithm

Input:
Replay memory capacity N, number of episodes M, number of steps T, learning rate α, and discount factor γ.

Output:
Updated Q-values matrix Q with weights θ

Initialise replay memory D to capacity N
Initialise action-value function Q with random weights θ
for *episode* $= 1 \rightarrow M$ **do**
    Initialise sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1 \rightarrow T$ **do**
        if random() $< \varepsilon$ **then**
            Select a random action a_t
        else
            $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        if ϕ_{j+1} is terminal **then**
            $y_j = r_j$
        else
            $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to θ
until termination condition is met

Experience Replay

A salient feature distinguishing Deep Q-Networks (DQNs) from traditional Q-learning is the incorporation of an experience replay mechanism [46]. In high-dimensional or non-stationary environments, online updates based on individual transitions risk catastrophic forgetting [47, 48]. To mitigate this, DQNs employ batch updates using a replay buffer that stores transition tuples (s, a, r, s'), where s is the state, a is the action, r is the reward, and s' is the subsequent state. During training, mini-batches of these stored transitions are sampled uniformly at random for neural network updates. This decouples the temporal correlation between consecutive transitions, thereby enhancing the stability and robustness of the learning algorithm [19]. The architecture and workflow of this mechanism are illustrated in Figure 4.1.

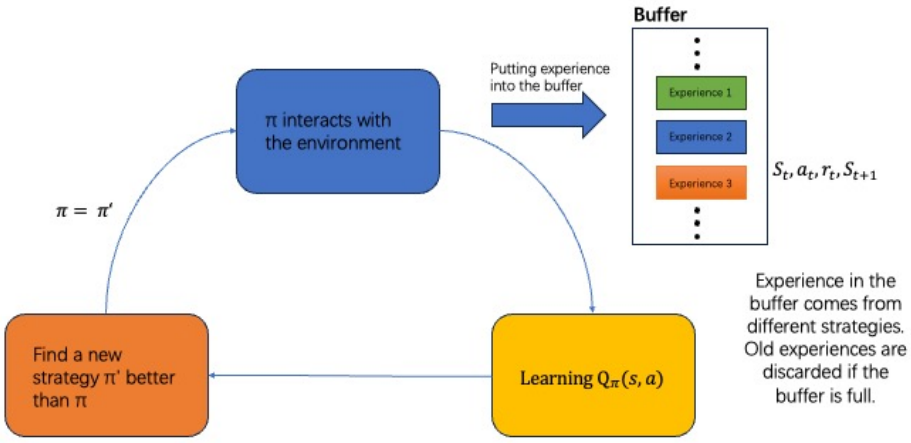


Figure 4.1: Experience Replay

Target Network

A critical innovation in Deep Q-Networks (DQNs) is the introduction of a target network, denoted as θ^-, alongside the primary Q-network θ [19]. The primary Q-network is responsible for approximating the current Q-values, while the target network is employed to generate the temporal-difference (TD) targets for Q-value updates. Specifically, the TD target is set as $r_t + Q_\pi(s_{t+1}, \pi(s_{t+1}))$, where r_t is the immediate reward and $Q_\pi(s_{t+1}, \pi(s_{t+1}))$ is the estimated future value under policy π. This architectural separation mitigates the risk of oscillations and instabilities in the learning dynamics. The target network's parameters are periodically updated to match those of the primary Q-network, reducing the autocorrelation in the TD target and thereby stabilizing the learning process [42]. The target network is defined by the term involving θ^-_{i-1} in the loss function $L_i(\theta_i)$:

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim d}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-_{i-1}) - Q(s, a; \theta_i)\right)^2\right]$$

where θ^-_{i-1} are the parameters of the target network at the previous synchronization point. The architecture and workflow of this mechanism are illustrated in Figure 4.2.

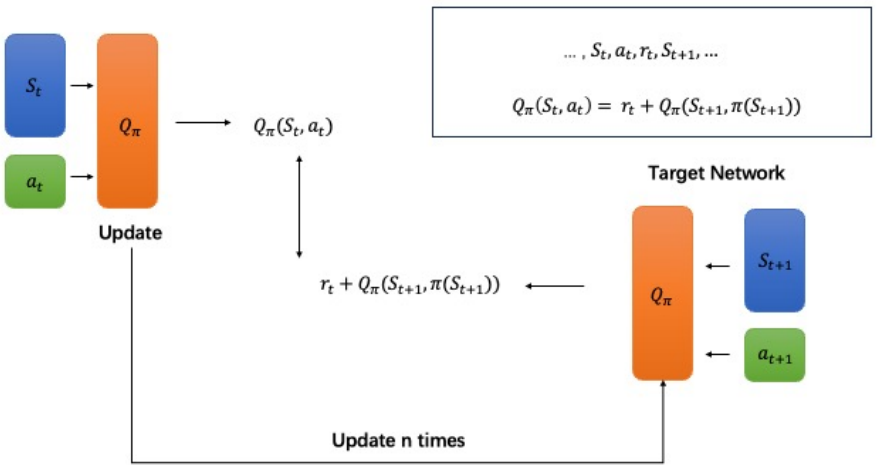


Figure 4.2: Schematic of the Target Network in DQN

4.1.2 Double Deep Q-Networks(DDQN)

Concept

Double Deep Q-Networks(DDQN), a variant of the conventional DQN, explicitly addresses the overestimation bias observed in Q-value approximation [49]. This bias is attributed to the max operator used in the Q-value update equation of standard DQN, which often leads to inflated value estimates.

Decomposition of Max Operation In a departure from DQN, Double DQN disambiguates the max operation into two components: action selection and action evaluation. It employs the primary network Q for the former and the target network Q' for the latter, thus reducing overestimation bias. Mathematically, this is formalized as:

$$Y_t^{\text{DDQN}} = R_{t+1} + \gamma Q' \left(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t^- \right) \tag{4.2}$$

Architectural Bifurcation Unlike DQN, which uses a monolithic network for both action selection and evaluation, Double DQN segregates these tasks across two networks. This architectural bifurcation enhances both the stability and the reliability of the learning process.

Significance of Update Rule The differential update rule is a hallmark of Double DQN. The primary network selects the action that maximizes the current state-action value, while the target network evaluates this action's Q-value. The bifurcation serves to provide a more conservative and unbiased estimate, in accordance with the objective function $J(\theta)$:

$$J(\theta) = \mathbb{E}\left[\left(Y_t^{\text{DDQN}} - Q(S_t, A_t; \theta) \right)^2 \right] \tag{4.3}$$

Algorithm 5 Double Deep Q-Learning with Experience Replay Algorithm

Input:
Replay memory capacity N, number of episodes M, number of steps T, learning rate α, discount factor γ, and reset step C.

Output:
Updated Q-values matrix Q with weights θ

Initialise replay memory D to capacity N
Initialise action-value function Q with random weights θ and θ^-
for *episode* $= 1 \rightarrow M$ **do**
    Initialise sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1 \rightarrow T$ **do**
        if random() $< \varepsilon$ **then**
            Select a random action a_t
        else
            $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        if ϕ_{j+1} is terminal **then**
            $y_j = r_j$
        else
            $y_j = r_j + \gamma Q(\phi_{j+1}, \arg\max_{a'} Q(\phi_{j+1}, a'; \theta); \theta^-)$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to θ
    if *episode* $\mod C == 0$ **then**
        $\theta^- = \theta$
until termination condition is met

4.1.3 Summary

Limitations in DQN and DDQN

Deep Q-Networks (DQN) have advanced the state-of-the-art in various domains but are not devoid of limitations. Primarily, DQN algorithms are susceptible to training instabilities and overestimation biases, complicating the optimization landscape and potentially resulting in suboptimal policy outcomes [49]. Double DQN (DDQN) mitigates overestimation but introduces its own challenges. Specifically, owing to its basis in Double Q-Learning, DDQN is susceptible to underestimation biases due to approximation errors, as elaborated by Ren et al. [22]. Such biases could distort the learning trajectory and converge to non-optimal fixed points, analogously to saddle points in non-convex optimization.

Advancing Deep Reinforcement Learning Through Smoothed Q-Learning Integration

In light of these challenges, our research aims to augment the robustness and versatility of deep reinforcement learning algorithms. We aim to create a model that is generalizable across various domains, including but not limited to video games, autonomous vehicles, and robotics. To this end, we explore the amalgamation of Smoothed Q-Learning techniques into the existing DQN framework. This integration seeks to address the limitations in DQN and DDQN, providing a more stable and unbiased learning algorithm that could potentially excel in complex, real-world scenarios.

4.2 Smoothed Deep Q-Networks (SDQN)

4.2.1 Background and Motivation

This section elucidates the concept and implementation of Smoothed DQN (SDQN), an extension of DQN, aiming to address the limitations of DQN and Double DQN whilst enhancing overall performance. The foundation of SDQN lies in the fusion of DQN with the principles of Smoothed Q-Learning, thereby transforming the Q-value update mechanism from the standard Q-learning to a smoother version[4, 1, 19, 49]. As previously demonstrated, due to overestimation errors in Q-learning, the performance tends to degrade[2, 23]. Double Q-learning tends to underestimate and its update mechanism does not align with the continuity requirements of temporal difference learning[22]. However, Smoothed Q-Learning has been proven to alleviate this situation and conforms to the sensitivity issue regarding temporal continuity in temporal difference learning. In standard Q-learning, the action corresponding to the maximal Q-value is typically selected. The essence of Smoothed Q-learning is to allocate a certain probability weight to all possible actions using $q_t(a|s_{t+1})$, where $q_t(a|s_{t+1})$ represents the probability distribution of action a given the state s_{t+1}. In a continuous action space, this distribution can be viewed as a probability measure defined over a specific measure space. Assigning probability distribution to every potential action ensures a more "smooth" learning process. It also incorporates all actions during updates, not merely the action with the maximal Q-value, leading to a more stable estimation of the Q-values. Given these advantages, it becomes imperative to reconsider the update formula used in Deep Q-Networks (DQNs). The DQN update mechanism, which focuses solely on the action with the maximum Q-value, may not be the most efficient way to explore the action space or to estimate Q-values reliably. Therefore, we propose integrating the Smoothed Q-Learning update methodology into the DQN framework. Through this method, our goal is to enhance the dependability and resilience of the learning process, potentially improving how DQNs perform in complex scenarios.

4.2.2 Core Principles of SDQN

Probabilistic Smoothing in SDQN

SDQN introduces a probabilistic smoothing mechanism to address the limitations of DQN and DDQN, notably overestimation and underestimation biases [2, 49, 22]. The algorithm employs a time-varying probability distribution $q_t(a|s_{t+1})$ to weight the Q-values of different actions during the update. This mechanism serves dual purposes:

1. It mitigates the overestimation bias by providing a weighted average over the Q-values, rather than selecting the maximum.

2. It dynamically adjusts the exploration-exploitation trade-off by modulating the distribution $q_t(a|s_{t+1})$ based on the learned Q-values.

This results in more stable and reliable Q-value estimates, making SDQN a robust choice for complex, high-dimensional state-action spaces.

Dual Neural Network Architecture in SDQN

SDQN adopts a dual neural network architecture similar to DDQN but with a nuanced difference. While DDQN uses the primary network for action selection and the target network for action

evaluation, SDQN integrates the probabilistic smoothing mechanism into the action selection process. Specifically, the online network, parameterized by θ_t, employs the distribution $q_t(a|s_{t+1})$ for action selection. The target network, parameterized by θ_t^-, remains responsible for action evaluation. This separation avoids feedback loops that could amplify estimation errors, a risk present in single-network architectures.
The Q-value update in SDQN is governed by the following equation:

$$Y_t^{\text{SDQN}} = R_{t+1} + \gamma \sum_a q_t(a|s_{t+1}) Q(s_{t+1}, a; \theta_t^-) \tag{4.4}$$

This equation integrates the probabilistic smoothing directly into the Q-value update, thereby enhancing both the stability and reliability of the learning process.

Implementation Insights

SDQN incorporates multiple strategies for computing $q_t(a|s_{t+1})$, each of which can be dynamically adjusted. This adaptability allows for nuanced management of the exploration-exploitation trade-off. Dynamic parameters, such as β_{dynamic} and δ_{dynamic}, enable the algorithm to adapt its behavior over time, thereby enhancing its robustness.

Inheritance of Smoothing Strategies from Smoothed Q-Learning SDQN extends the probabilistic smoothing techniques originally introduced in Smoothed Q-Learning. Unlike its predecessor, which relies on tabular methods, SDQN employs neural networks for Q-value approximation. The algorithm utilizes a distribution $q_t(a|s')$ to guide action selection, similar to Smoothed Q-Learning. This distribution is designed to incrementally concentrate its mass on the action $a^* = \arg\max_a \tilde{Q}_t(s', a)$ as time t progresses. Specifically, the distribution allocates a mass of $1 - \delta_t$ to the maximal action a^*, where δ_t is a decay parameter that diminishes towards zero as t increases. This design ensures that the algorithm progressively narrows its focus onto actions that are likely to be optimal, thereby facilitating convergence to the optimal policy. The algorithm's capacity to employ various smoothing strategies provides the flexibility needed to adapt SDQN to different problem domains while preserving its theoretical convergence properties.

Mathematical Justifications

- **Relative Underestimation between SDQN and DQN:**
  Let us define a particular state s' and a set of parameters θ. In the DQN framework, the target function is denoted as

  $$Y^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(s', a; \theta_t^-).$$

  For SDQN, the corresponding target function is

  $$Y^{\text{SDQN}} \equiv R_{t+1} + \gamma \sum_a q_t(a|s') Q(s', a; \theta_t^-).$$

  We define Δ as the difference between these two target functions, i.e.,

  $$\Delta = Y^{\text{DQN}} - Y^{\text{SDQN}} = \gamma \left(\max_a Q(s', a; \theta_t^-) - \sum_a q_t(a|s') Q(s', a; \theta_t^-) \right).$$

Given that $\max_a Q(s', a; \theta_t^-)$ represents the supremum of the Q-values over all possible actions a at state s', we have:

$$\max_a Q(s', a; \theta_t^-) \geq Q(s', a; \theta_t^-)$$

$$\sum_a q_t(a|s') \max_a Q(s', a; \theta_t^-) \geq \sum_a q_t(a|s') Q(s', a; \theta_t^-)$$

As $q_t(a \mid s')$ is a probability distribution over actions given state s' and the sum of probabilities must be 1, it implies:

$$\max_a Q(s', a; \theta_t^-) \geq \sum_a q_t(a \mid s') Q(s', a; \theta_t^-)$$

Consequently, this means:

$$Y^{\text{DQN}} \geq Y^{\text{SDQN}}$$

- **Conjecture: Relative Optimism of SDQN and DDQN:**
  Define ΔQ as the difference between the target functions of SDQN and DDQN:

$$\Delta Q = Y^{\text{DDQN}} - Y^{\text{SDQN}}.$$

From the given target functions, we obtain:

$$\Delta Q = \gamma \left(Q(s', \arg\max_a Q(s', a; \theta_t^-), \theta_t^-) - \sum_a q_t(a|s') Q(s', a; \theta_t^-) \right).$$

We hypothesize two critical behaviors:

1. **Temporal Q-Value Evolution:** As $t \to \infty$, we expect both agents to converge towards an optimal policy, yielding elevated Q-values. Therefore, Q-values at time t are likely to be less than those at $t + n$, for $n \in \mathbb{N}$.
2. **Q-Value Underestimation Risk:** The ϵ-greedy strategy causes a progressive shift from exploration to exploitation. During exploration, Q-values are usually underestimated, making Double Q-Learning prone to underestimation if one of the tables is infrequently updated.

Summary The risk of Q-value underestimation, coupled with the temporal Q-value evolution, implies that SDQN could potentially provide a more stable and accurate long-term Q-value estimation than DDQN.

Algorithm 6 Smoothed Deep Q-Learning with Experience Replay (SDQN) Algorithm

Input: Replay memory capacity N, number of episodes M, number of steps T, learning rate α, discount factor γ, exploration probability ε.
Output: Updated Q-values matrix Q with weights θ

Initialise replay memory D to capacity N
Initialise action-value function Q with random weights θ
for *episode* $= 1 \rightarrow M$ **do**
    Initialise sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1 \rightarrow T$ **do**
        Choose action a_t based on ϵ-greedy: Draw u from a uniform distribution in $[0, 1]$
        if $u > \varepsilon$ **then**
            Select action $a_t = \arg\max_a Q_t(a, s_t; \theta)$ ▷ If there is more than one optimal action, select from them randomly
        else
            Select a random action a_t
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        if ϕ_{j+1} is terminal **then**
            $y_j = r_j$
        else
            $y_j = r_j + \gamma \sum_{a'} q_t(a'|s') Q(\phi_{j+1}, a'; \theta)$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to θ, treating y_j as a constant. (Detail proof see in Appendix A.1.3)
until termination condition is met

4.2.3 Implementation Details

The development of SDQN builds upon the foundational principles of DQN but with modifications to accommodate the smoothed Q-value update mechanism. This approach necessitates changes in action selection and Q-value computation during the learning phase. The crucial steps of implementation are as follows:

1. **Action Selection:** Instead of solely relying on the action with the highest Q-value, we incorporate a probability-weighted strategy using a smoothing distribution $q_t(a|s_t)$. This strategy combines the ϵ-greedy approach with a distribution derived from the Q-values, allowing for a more explorative and balanced action selection process.

2. **Q-Value Computation:** During the training, the Q-value for an action is calculated based on an ensemble average using the probabilities $q_t(a|s_t)$, which weights the contribution of each action to the final Q-value.

3. **Target Network Updates:** Similar to DQN, the target network in SDQN is periodically updated from the online network. This ensures stability during training and prevents drastic changes in the Q-value approximations.

4. **Experience Replay:** The SDQN still uses experience replay to break the correlation between consecutive observations, aiding in stable and efficient learning. However, the Q-value update during the replay is modified to incorporate the smoothed Q-values.

4.3 DQN Experiment in CartPole

Due to the substantial computational resources and time required for the Atari 2600, we first implemented all DQN variants using the Cartpole experiment and verified their performance on this benchmark.

4.3.1 Description of the experimental environment

Environment Description

The CartPole task is designed to emulate the challenge of maintaining equilibrium of a pole affixed to a mobile cart. Within any specific time increment, an agent has the capacity for two distinct actions: to shift the cart either leftwards or rightwards. The principal objective is the prevention of the pole's descent. For every time increment wherein the pole remains vertical, the agent is accorded a reward of +1. The episode culminates if the pole inclines excessively, or if the cart's displacement from the centre surpasses 2.4 units. Evidently, a protracted episode duration denotes superior performance by the agent.

Action Space

The action space in the CartPole task is characterised by a one-dimensional array. The two possible determinants within this array are:

- 0: Impelling the cart in a leftward trajectory.
- 1: Directing the cart towards the right.

It is imperative to underscore that the momentum, whether it be an increase or decrease, is not invariable, chiefly due to the external force applied. The pole's angular disposition and its intrinsic centre of gravity significantly influence the energy dynamics required for the locomotion of the cart.

Observation Space

The observation space is represented by a four-dimensional array, each dimension elucidating a specific aspect of the system's state:

1. The cart's linear position, which ranges from -4.8 to 4.8 units.
2. The velocity of the cart, which theoretically spans from negative to positive infinity.
3. The pole's angular position, with an approximate range of -0.418 radians ($-24°$) to 0.418 radians ($24°$).
4. The angular velocity of the pole, with no predefined limits.

While the aforementioned ranges demarcate the potential states within the observation space, it's pertinent to note that in real-time episodes, the cart's linear position is confined within a (-2.4, 2.4) boundary, and the pole's angular orientation is restricted to $\pm 12°$ or (-.2095, .2095) radians.

Reward Mechanism

Every discrete action taken within the task attracts a reward of +1. This includes the actions leading to the termination of an episode. It's salient to mention that while the v1 variant of the task sets a reward benchmark at 500.

Initial State

The inaugural state for all the observations, at the commencement of an episode, lies uniformly distributed within the range (-0.05, 0.05).

Episode Termination Criteria:

- An episode reaches its denouement if the pole's angular deviation extends beyond $\pm 12^{\circ}$.
- Termination is also precipitated if the cart's linear position overshoots the ± 2.4 threshold, marking the cart's central position's alignment with the display's periphery.
- Furthermore, episodes are curtailed if their longevity breaches the 500-step mark.

4.3.2 Experimental Objective and Evaluation

Objective

The primary objective of this experiment is to train a Deep Q-Network (DQN) to efficiently balance a pole on a moving cart, a classic problem in reinforcement learning referred to as the CartPole problem. The learning agent is expected to discover an optimal policy that maximises its cumulative reward over time. The experiment's success can be measured by how well the agent manages to consistently achieve episodes of maximum length.

Deep Q-Network Architecture

The DQN agent leverages a neural network to translate state observations into their respective action values. Our experiment employs the PyTorch framework for building the DQN, benefitting from its intuitive integration of deep learning models with reinforcement learning strategies. The architecture of the neural network is delineated as follows:

- **Input layer:** Receives state observations. Its size is dictated by the dimensionality of the observation space.
- **Hidden layers:** Comprises two layers, each hosting 128 neurons. The ReLU activation function facilitates non-linear transformations within these layers.
- **Output layer:** Designed to accommodate the number of valid actions defined by the action space.

Ultimately, given a state input, this architecture produces a Q-value for every conceivable action.

Training Setup

The training loop utilises several key concepts essential to the DQN algorithm:

- **Epsilon-Greedy Action Selection**: The agent selects actions either based on the Q-values predicted by the DQN or randomly, with the probability of random selection decaying over time.
- **Experience Replay**: A memory buffer is used to store past experiences, and random mini-batches from this memory are utilised to train the network, enhancing data efficiency and breaking harmful temporal correlations.

- **Target Network**: A secondary, target network is employed to compute the Q-values for the next state, stabilising the learning process. The weights of this target network are updated periodically with a soft update strategy.

Hyperparameters

Several hyperparameters guide the learning process:

- Batch Size: 128
- Discount Factor (γ): 0.99
- Starting Epsilon Value (ϵ_{start}): 0.9
- Ending Epsilon Value (ϵ_{end}): 0.05
- Epsilon Decay: 1000
- Target Network Soft Update Rate (τ): 0.005
- Learning Rate: 1×10^{-4}

Evaluation Metric

The primary metric for evaluating the agent's performance is the episode duration, representing how long the agent manages to keep the pole balanced in a given episode. A plot of episode durations, along with a rolling average over the last 100 episodes, provides a visual representation of the agent's progress and proficiency.

4.3.3 Code Structure

In the realm of Deep Q-Learning, the traditional Q-value table, $Q(s, a)$, evolves into a more dynamic function rather than a static lookup table. Drawing parallels between the Q-value update equation and the mechanisms of neural network training, one can discern an evident similarity. The "target" in this context refers to the combination of the immediate reward and the discounted future utilit. $Q(S_t, A_t)$ stands as the model's current perception of the Q-value. The disparity between these two values manifests as the loss. Utilizing a learning rate, the weights of the network undergo iterative updates, refining the predictions over time. When we employ a neural network within this framework, it's fed with states and corresponding target Q-values for training. Consequently, when tasked with predictions, the network outputs Q-values for every action within the action space.

Algorithm Structure

We mainly drew on the Tutorial on the DQN algorithm written by PyTorch[50]. Based on this algorithm, we continued to develop different agents such as DDQN and SDQN.

- **Algorithm Implementation:** All strategies utilize the target network to predict the Q-values for the next state. The divergence lies in the action selection mechanism:

  1. **DQN:**
     - Selects the action with the maximum Q-value.

2. **DDQN:**
   - Employs the policy network to predict the next action.
   - Fetches the Q-value for this action using the target network.
3. **Softmax SDQN:**
   - Applies a softmax operation, parameterized by a dynamic beta.
   - Chooses an action based on the resulting softmax probabilities.
4. **Clipped Max SDQN:**
   - Utilizes a dynamic tau parameter to clip the Q-values.
   - Selects the action with the highest weighted Q-value post-clipping.
5. **Clipped Softmax SDQN:**
   - Employs a dynamic beta in a softmax operation that considers only the top-k Q-values.
   - Chooses the action with the highest weighted Q-value post-softmax.

- **Loss Calculation and Optimization:** After determining the expected Q-values for the current state-action pairs, the code computes the Huber loss between the predicted Q-values and the expected Q-values. It then backpropagates this loss to optimize the weights of the neural network.

4.3.4 Evaluation

Cartpole Experiment Analysis

In Figure 4.3, we present a comparative analysis of five distinct DQN algorithms within the CartPole environment. Our initial experimental design considered averaging results over ten iterations for graphical representation. However, we found this approach unsuitable for highly stochastic environments like CartPole and Atari. Averaging would obscure crucial metrics such as learning stability and convergence time to an optimal strategy. Moreover, the high stochasticity of the environment would introduce significant biases. Consequently, we selected the most representative single iteration for empirical validation after conducting multiple iterations.

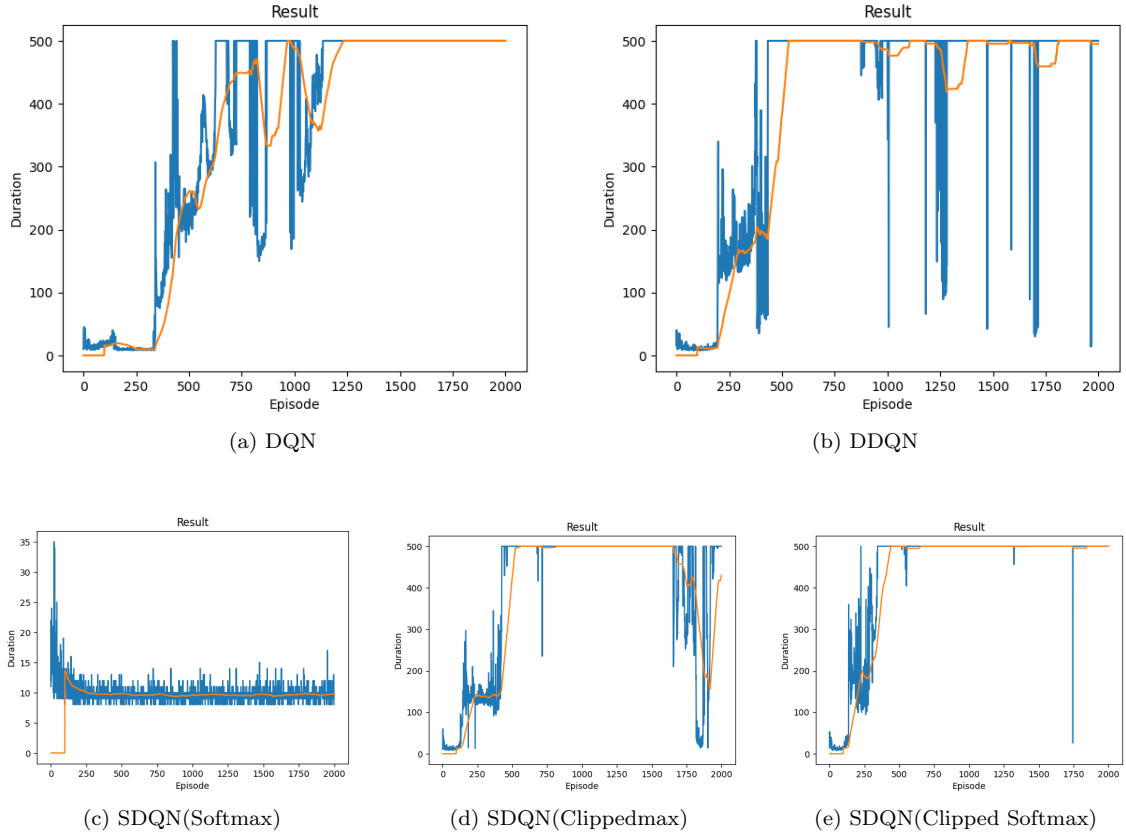
Softmax Strategy in SDQN: SDQN, when employing the Softmax strategy, exhibits suboptimal performance, stagnating at approximately 10 steps. This could be attributed to the undue complexity of the Softmax strategy for the relatively straightforward CartPole environment, causing the algorithm to gravitate towards a local optimum.

Performance of DQN and DDQN: A closer examination of the standard DQN algorithm reveals that it begins to converge to an optimal strategy after approximately 1250 episodes. In contrast, DDQN shows a faster learning rate, approximating the optimal policy within just 500 episodes. This suggests that DDQN effectively mitigates DQN's tendency for reward overestimation, thereby accelerating the learning process.

Clipped-max Strategy: Under the Clipped-max strategy, both SDQN and DDQN display similar learning rates, converging around the 500-episode mark. However, the policy experiences significant fluctuations as the number of learning episodes increases. This instability is likely due to the Clipped-max strategy's resemblance to DQN, leading to undue optimism in policy evaluation.

Clipped Softmax Strategy: Given SDQN's suboptimal performance under the Softmax strategy, we introduce a novel approach termed 'Clipped Softmax,' inspired by Prof Barber's work [1]. For a detailed understanding of the algorithm, readers are referred to `Chapter 3.2.3 Clipped Softmax`. Our analysis indicates that the Clipped Softmax strategy outperforms both DDQN and SDQN (Clipped Max) in terms of stability and learning rate. It converges to the optimal policy before the 500-episode mark and maintains a stable performance metric thereafter.

Summary: In summary, SDQN, particularly when employing the Clipped Softmax strategy, offers a distinct competitive advantage in environments requiring the careful selection of specialized statistical strategies, outperforming its predecessors, DQN and DDQN.



(a) DQN (b) DDQN

(c) SDQN(Softmax) (d) SDQN(Clippedmax) (e) SDQN(Clipped Softmax)

Figure 4.3: Comparison of different Strategies on the CartPole Experiment

Speed analysis for learning optimal strategies

In the CartPole experiment, a rigorous methodology was employed, involving ten iterations to ascertain the number of episodes requisite for each agent to achieve stable convergence to an optimal policy. The average reward over the last 100 episodes was computed, with a threshold of 495 serving as the criterion for having mastered the optimal policy. This metric was deliberately chosen to obviate the distortive effects of transient learning fluctuations.

Data Interpretation: The data thus collated serves as a robust metric for evaluating the temporal efficiency of each agent in mastering a stable optimal policy. It is noteworthy that SDQN,

when operating under the Softmax strategy, was precluded from this comparative analysis due to its failure to converge optimally.

DQN's Inefficiencies: Upon meticulous examination of the tabulated results in Table 4.1, DQN serves not only as a benchmark for comparison with other DQN algorithms but also as a manifestation of the worst-case scenario in this experiment. It registers the highest mean and median values in the original dataset, thereby necessitating a greater number of episodes for convergence. This inefficacy persists even when an outlier is expunged, with DQN continuing to dominate in terms of mean, median, and variance.

DDQN's Variability: Contrastingly, DDQN, despite initially exhibiting the highest variance of 108832.25, sees a substantial attenuation to 15,177.78 upon outlier removal. This suggests that DDQN, although more stable than DQN, still falls short of the efficiency and stability exhibited by SDQN strategies.

SDQN's Superiority: In stark contrast, SDQN, under both the Clipped Max and Clipped Softmax strategies, eclipses DQN and DDQN in terms of mean and median values, thereby indicating a more expedient convergence to the optimal policy. Particularly salient is the precipitous decline in variance for SDQN strategies upon outlier removal, with the variance for the Clipped Softmax variant plummeting to a mere 1,540.22, thereby indicating unparalleled learning stability.

Graphical Insights: To augment these findings, both box plots 4.5a and violin plots 4.5b are presented, both of which are derived from the underlying scatter plot 4.4. The box plot for DQN reveals the most pronounced interquartile range, indicative of an unstable learning trajectory. DDQN, although susceptible to extreme outliers, also manifests a larger interquartile range compared to both SDQN strategies. In contradistinction, SDQN under the Clipped Softmax strategy manifests as the epitome of stability, with the least number of outliers and a box primarily encapsulating the interquartile range, signifying a highly concentrated dataset.

Summary: In summation, the empirical data incontrovertibly substantiates that SDQN, particularly when operating under the Clipped Softmax strategy, converges more efficiently and with remarkable stability to the optimal policy, thereby outperforming both its antecedents, DQN and DDQN, in the CartPole environment.

Table 4.1: Statistical Comparison of DQN, DDQN, SDQN Clipped Max, and SDQN Clipped Softmax

Metric	DQN	DDQN	SDQN Clipped Max	SDQN Clipped Softmax
Original Data				
Mean	713.1	675.5	594.4	**540.5**
Variance	88100.09	108832.25	40768.44	**6216.45**
Median	563.0	543.5	**521.5**	530.0
25% Point	526.0	507.25	**468.25**	491.75
75% Point	776.0	731.25	633.0	**561.75**
After Removing One Outlier Point				
Mean	639.44	572.67	534.89	**517.33**
Variance	43637.58	15177.78	9882.54	**1540.22**
Median	547.0	540.0	**501.0**	521.0
25% Point	526.0	502.0	**468.0**	488.0
75% Point	746.0	573.0	549.0	**549.0**

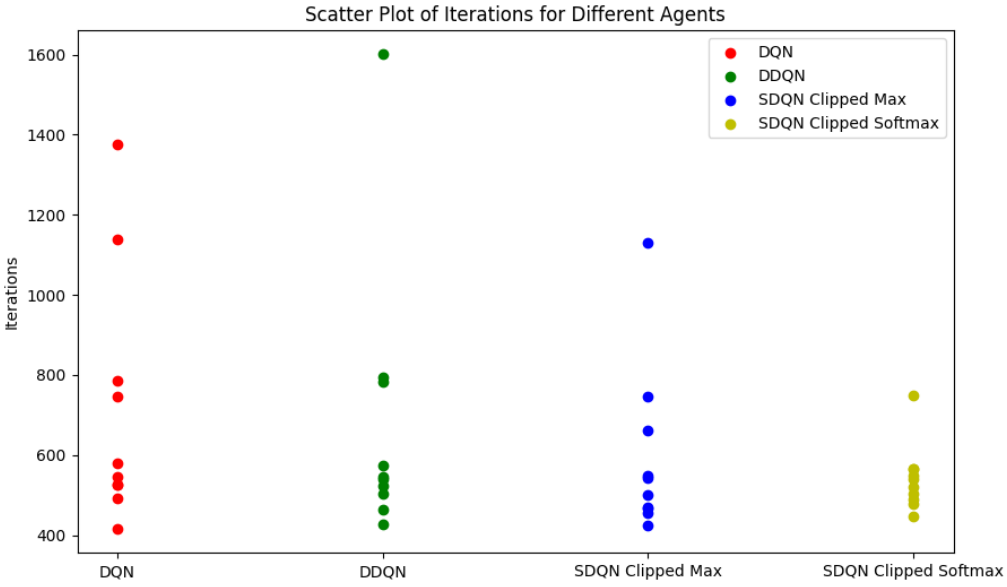
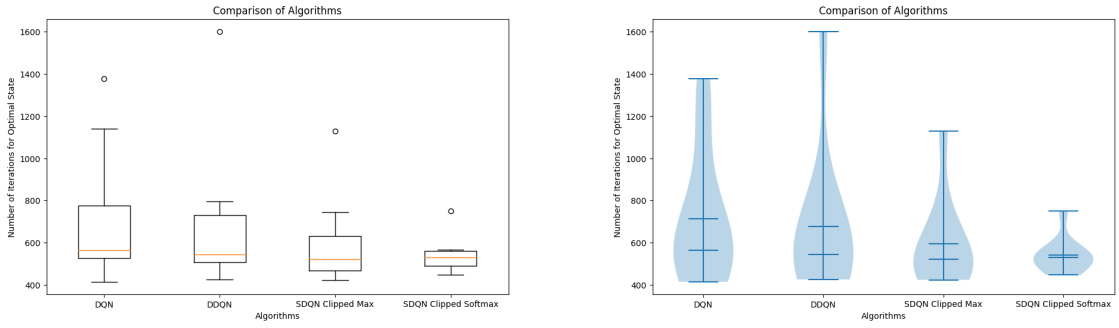


Figure 4.4: Scatter Diagram



(a) Box Diagram

(b) Violin Diagram

Figure 4.5: Number of Episodes required for each agent to learn the optimal policy in ten iterations

Chapter 5

Conclusions

5.1 Achievements

The initial chapter delineated the project objectives. All anticipated goals have been successfully met, as detailed below:

- **Goal 1: Validation in Complex Environments:** Following the completion of MDP experiments as outlined by Professor David Barber, the research was extended to more intricate environments such as Roulette and Grid World. Comparative evaluations among Q-Learning, Double Q-Learning, and multiple Smoothed Q-Learning strategies were conducted. Further details are available in Chapter 3.2.
- **Goal 2: Development of 'Smoothed DQN':** This goal necessitated an in-depth understanding of Smoothed Q-Learning and Deep Q-Networks (DQN) fundamentals. Iterative formulas were examined, and pseudocode for the Smoothed DQN algorithm was developed. Novel smoothing strategies were introduced based on empirical findings, and the theoretical framework was expanded by exploring the interplay between statistics and reinforcement learning.
- **Goal 3: Comprehensive Comparative Analysis:** Various Smoothed Q-Learning algorithms were rigorously implemented for different strategies across multiple environments. A thorough performance comparison among Smoothed Q-Learning, standard Q-Learning, and Double Q-Learning was conducted. The results indicate that an appropriately selected Smoothed Q-Learning strategy can mitigate the overestimation and underestimation issues inherent in standard Q-Learning and Double Q-Learning, respectively.
- **Goal 4: Experimental Results and Insights:** The SDQN algorithm was implemented for three distinct smoothing strategies in the Cartpole experiment. Empirical evidence suggests that a well-chosen Smoothed DQN strategy exhibits superior performance in terms of learning speed and stability compared to both DQN and DDQN.

5.2 Evaluation

In this experiment, I have largely achieved all predefined objectives. It is pertinent to note that the Atari 2600 gaming environment is currently the benchmark for empirical validation in the majority of extant deep reinforcement learning literature. Consequently, to furnish the most rigorous

verification of our Smoothed Deep Q-Networks (SDQN) algorithm, further testing in the complex, high-dimensional action-state spaces of Atari 2600 games is imperative. Given the computational intensity and the substantial resources required for such an endeavour, the research is constrained by both time and computational capacity. However, I am committed to expeditiously conducting this scientifically rigorous experiment as soon as feasible. In terms of critical evaluation, while the project has met its goals to a significant extent, the ultimate validation in the context of Atari 2600 remains pending due to resource limitations. The choice of implementation technology and design practices have been in line with industry standards, but the true test of the application's fitness for purpose will be its performance in the more challenging Atari 2600 environment.

5.3 Future Work

In future, the research agenda aims to empirically validate the Smooth Deep Q Networks (SDQN) algorithm, particularly focusing on its performance in the Atari 2600 gaming environment. Due to computational constraints, a comprehensive evaluation on this platform has yet to be conducted.

Transition to Compiled Language The current implementation, written in Python, faces execution speed limitations inherent to interpreted languages. In light of this, we are contemplating a transition to a compiled language to expedite computational tasks. This shift is motivated by the efficiency gains observed in DeepMind's DQN implementation, which was developed in Lua.

Empirical Validation The primary objective of the forthcoming work diverges from our existing focus on theoretical validation. We aim to conduct extensive experiments to assess the algorithm's performance metrics, thereby providing empirical evidence of its efficacy.

Complex Smoothing Strategies Concurrently, we plan to investigate more sophisticated smoothing strategies to enhance the algorithm's adaptability to complex, high-dimensional environments.

Real-world Applications Ultimately, these extensions will be designed to substantiate the algorithm's applicability and robustness in more intricate real-world scenarios.

5.4 Wrap-up and Final Thoughts

In conclusion, this paper introduces Smoothed Deep Q-Networks (SDQN), a groundbreaking algorithm that ingeniously integrates the update mechanisms of Smoothed Q-Learning into the value iteration function via probabilistic distributions. This innovation represents a significant advancement in reinforcement learning algorithms that operate on non-policy values and are model-free. Empirical results validate that SDQN effectively addresses the overestimation issues inherent in standard DQN and the underestimation biases in Double DQN (DDQN). Moreover, the algorithm demonstrates remarkable generalizability, exhibiting robust performance across both discrete and continuous environments.

This work serves as a comprehensive investigation into the domain of Q-Learning and its smoothed variants. All initial objectives have been successfully achieved, from conducting comparative evaluations in complex environments to the development and theoretical substantiation of the SDQN algorithm. Beyond meeting the immediate goals, this project establishes a robust foundation for

future research in the rapidly evolving field of deep reinforcement learning.
The experience has been profoundly enlightening, and the promising results suggest that Smoothed Q-Learning and SDQN have the potential to significantly enhance the efficiency and effectiveness of deep reinforcement learning algorithms. This contribution is pivotal not only for the field of reinforcement learning but also has far-reaching implications for broader areas of machine learning and artificial intelligence.

Bibliography

[1] D. Barber, "Smoothed q-learning," 2023. 2, 3, 3.1, 3.1.2, 3.1.2, 3.1.2, 1, 2, 3.1.2, 3.1.2, 3.2.1, 4.2.1, 4.3.4

[2] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, 2010. 2, 2.2.4, 2.3, 2.3.2, 2.3.3, 2.3.3, 3.2.2, 3.2.2, 3.2.3, 3.2.4, 3.2.4, 4.2.1, 4.2.2, A.1.1

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016. 2.1

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018. 2.1, 2.2.4, 4.1.1, 4.2.1

[5] B. F. Skinner, *The behavior of organisms: An experimental analysis*. BF Skinner Foundation, 2019. 2.1

[6] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017. 2.1, 4.1

[7] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-dynamic programming: an overview," in *Proceedings of 1995 34th IEEE conference on decision and control*, vol. 1. IEEE, 1995, pp. 560–564. 2.1

[8] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992. 2.2, 2.2.2, 2.2.3, 2.2.3, 2.2.3, A.1.1

[9] Q. Wei, D. Liu, and G. Shi, "A novel dual iterative q-learning method for optimal battery management in smart residential environments," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 4, pp. 2509–2518, 2014. 2.2

[10] R. Crites and A. Barto, "Improving elevator performance using reinforcement learning," *Advances in neural information processing systems*, vol. 8, 1995. 2.2

[11] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990. 2.2.1

[12] C. J. C. H. Watkins, "Learning from delayed rewards," 1989. 2.2.1, 2.2.2

[13] R. S. Sutton, A. G. Barto *et al.*, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 135. 2.2.1, 2.2.2, 3.1.4

[14] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, "Convergence results for single-step on-policy reinforcement-learning algorithms," *Machine learning*, vol. 38, pp. 287–308, 2000. 2.2.3, 2.2.3

[15] P. Dayan, "The convergence of td (λ) for general λ," *Machine learning*, vol. 8, pp. 341–362, 1992. 2.2.3

[16] J. N. Tsitsiklis, "Asynchronous stochastic approximation and q-learning," *Machine learning*, vol. 16, pp. 185–202, 1994. 2.2.3

[17] E. Even-Dar, Y. Mansour, and P. Bartlett, "Learning rates for q-learning." *Journal of machine learning Research*, vol. 5, no. 1, 2003. 2.2.3, 2.2.3

[18] C. Szepesvári, "The asymptotic convergence-rate of q-learning," *Advances in neural information processing systems*, vol. 10, 1997. 2.2.3

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015. 2.2.4, 4.1, 4.1.1, 4.1.1, 4.1.1, 4.1.1, 4.2.1

[20] C. Szepesvári, *Algorithms for reinforcement learning*. Springer Nature, 2022. 2.2.4

[21] J. E. Smith and R. L. Winkler, "The optimizer's curse: Skepticism and postdecision surprise in decision analysis," *Management Science*, vol. 52, no. 3, pp. 311–322, 2006. 2.2.4

[22] Z. Ren, G. Zhu, H. Hu, B. Han, J. Chen, and C. Zhang, "On the estimation bias in double q-learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 10 246–10 259, 2021. 2.3.4, 3.1.4, 4.1.3, 4.2.1, 4.2.2

[23] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596. 2.3.4, 4.2.1

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870. 2.3.4

[25] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke *et al.*, "Scalable deep reinforcement learning for vision-based robotic manipulation," in *Conference on Robot Learning*. PMLR, 2018, pp. 651–673. 2.3.4

[26] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002. 3.1.1

[27] D. Bouneffouf, "Finite-time analysis of the multi-armed bandit problem with known trend," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 2543–2549. 3.1.1

[28] N. Vlassis, M. Ghavamzadeh, S. Mannor, and P. Poupart, "Bayesian reinforcement learning," *Reinforcement Learning: State-of-the-Art*, pp. 359–386, 2012. 3.1.1

[29] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999. 3.1.1

[30] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897. 3.1.1

[31] P.-L. Hsu and H. Robbins, "Complete convergence and the law of large numbers," *Proceedings of the national academy of sciences*, vol. 33, no. 2, pp. 25–31, 1947. 3.1.2

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017. 4.1

[33] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun, "Pedestrian detection with unsupervised multi-stage feature learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 3626–3633. 4.1

[34] V. Mnih, *Machine learning for aerial image labeling*. University of Toronto (Canada), 2013. 4.1

[35] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2011. 4.1

[36] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649. 4.1

[37] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015. 4.1

[38] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," *arXiv preprint arXiv:1904.12901*, 2019. 4.1

[39] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016. 4.1

[40] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International journal of robotics research*, vol. 37, no. 4-5, pp. 421–436, 2018. 4.1

[41] S. Zhou, X. Liu, Y. Xu, and J. Guo, "A deep q-network (dqn) based path planning method for mobile robots," in *2018 IEEE International Conference on Information and Automation (ICIA)*. IEEE, 2018, pp. 366–371. 4.1

[42] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013. 4.1, 4.1.1, 4.1.1

[43] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018. 4.1

[44] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016. 4.1.1

[45] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966. 4.1.1

[46] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, pp. 293–321, 1992. 4.1.1

[47] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*. Elsevier, 1989, vol. 24, pp. 109–165. 4.1.1

[48] R. Ratcliff, "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions." *Psychological review*, vol. 97, no. 2, p. 285, 1990. 4.1.1

[49] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016. 4.1.2, 4.1.3, 4.2.1, 4.2.2

[50] PyTorch, "Pytorch intermediate tutorials: Reinforcement q-learning," 2021, gitHub repository. [Online]. Available: https://github.com/pytorch/tutorials/blob/main/intermediate_source/reinforcement_q_learning.py 4.3.3

Appendix A

Other appendices

A.1 Proof Part

A.1.1 Proof of Q-Learning Overestimation

In the realm of Q-learning, understanding the behavior of estimators is pivotal, especially when approximating the maximum expected value of a set of random variables. This section delves into the intricacies of such estimations and highlights potential biases that can emerge, impacting the efficacy of Q-learning algorithms.
Consider a finite action-state space (V, A) with a set of M random variables, represented as $X = \{X_1, \ldots, X_M\}$[8]. In many practical scenarios, one is often interested in determining the maximum expected value of these variables, denoted as:

$$\max_i \mathbb{E}[X_i]$$

However, without comprehensive knowledge of the underlying distributions of these variables, exact determination becomes elusive. Instead, we often resort to approximations. Let's denote $S = \sum_{i=1}^{M} S_i$ as a set of samples, where each S_i is a subset containing samples for the variable X_i. Assuming these samples are independent and identically distributed (iid), unbiased estimates for the expected values can be derived by computing the sample average for each variable. The crux of the matter lies in the order of operations. While the maximal estimator $\max_i \mu_i(S)$ provides an unbiased estimate for $\mathbb{E}[\max_j \mu_j]$, it introduces a bias when estimating $\max_i \mathbb{E}[X_i]$. This subtle yet significant distinction has profound implications for Q-learning, especially when considering convergence and the potential for overestimation[2].
In the context of Deep Q Networks (DQN), this overestimation bias can further amplify, leading to overly optimistic value estimates. Such optimism can misguide the learning agent, pushing it towards suboptimal policies. Recognizing and addressing this bias is, therefore, paramount for the effective application of Q-learning in complex environments.

A.1.2 Single and Double Estimators: A Comparative Analysis

The Single Estimator

In the single estimator approach, the same set of samples is utilised both for identifying the "optimal" action and for evaluating its value. This methodology is problematic because the act

of selecting the maximum value from a set of noisy estimates inherently introduces a bias towards overestimation. Mathematically, this is captured by the following equation:

$$E\left\{\max_j \mu_j\right\} = \int_{-\infty}^{\infty} x \frac{d}{dx} \prod_{i=1}^{M} F_i^{\mu}(x) dx = \sum_{j=1}^{M} \int_{-\infty}^{\infty} x f_j^{\mu}(s) \prod_{i \neq j} F_i^{\mu}(x) dx \tag{A.1}$$

Equation A.1 essentially tells us that the expected maximum value is a function of the individual probability distributions of the estimators. However, because the same samples are used for both identification and evaluation of this maximum, the method fails to account for the impact of sample noise and randomness on the selection of the maximum value, leading to a biased overestimate.

The Double Estimator

The double estimator method offers an ingenious workaround. It employs two separate sets of estimators, updated with distinct subsets of samples. One set (μ_A) is used to identify the "optimal" action, and the other set (μ_B) is used to evaluate it. This decoupling reduces the inherent bias in the single estimator approach. The key equation here is:

$$\sum_{j=1}^{M} P(j = a^*) E\left\{\mu_j^B\right\} = \sum_{j=1}^{M} E\left\{\mu_j^B\right\} \int_{-\infty}^{\infty} f_j^A(x) \prod_{i \neq j} F_i^A(x) dx \tag{A.2}$$

This equation A.2 shows that the expected value of the approximation by the double estimator is a weighted sum of unbiased expected values, inherently less prone to overestimation.

Comparative Insights

When comparing Equation A.1 and Equation A.2, the key difference lies in how they handle the expected maximum value. The single estimator formula utilises x, which correlates with the monotonically increasing product of the cumulative distribution functions, leading to overestimation. On the other hand, the double estimator employs $E\left\{\mu_j^B\right\}$, which is a weighted sum of unbiased expected values, making it less susceptible to overestimation.

A.1.3 Algorithm 6 SDQN Line 19

In most reinforcement learning algorithms, the action-value function is iteratively updated through the Bellman equation. For Smoothed DQN, a variant of the Deep Q-Network (DQN), the target network is uniquely defined to incorporate a smoothing operation over the action space. Given the definition of the target network in Smoothed DQN, the target y_i is expressed as:

$$y_i = r + \gamma \sum_{a'} q_t(a|s_{t+1}) Q^*(s', a'; \theta) \tag{A.3}$$

From the above definition, the action-value function Q is updated as:

$$Q_{i+1}(s, a) = \mathbb{E}\left[r + \gamma \sum_{a'} q_t(a|s_{t+1}) Q_i(s', a') \mid s, a\right] \tag{A.4}$$

As i grows, Q_i asymptotically approaches Q^*:

$$Q_i \to Q^* \text{ as } i \to \infty \tag{A.5}$$

In DQN, convergence in value iteration is intricate due to the independent nature of each value function approximation. A neural network function approximator, termed the Q-network with weights θ, is used to estimate the action-value function:

$$Q(s, a; \theta) \approx Q^*(s, a) \tag{A.6}$$

The literature suggests that for each iteration in training the Q-network, the loss function $L_i(\theta_i)$ is minimised as:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s, a; \theta_i))^2\right] \tag{A.7}$$

Substituting the target y_i from Equation A.3 into the loss function $L_i(\theta_i)$ from Equation A.7, we get:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \sum_{a'} q_t(a|s_{t+1})Q^*(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2\right] \tag{A.8}$$

During the optimisation of $L_i(\theta_i)$, the parameters θ_{i-1} from the previous iteration are held constant. Differentiating the loss function with respect to the weights, the gradient is:

$$\nabla\theta_i L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[2\left(r + \gamma \sum_{a'} q_t(a|s_{t+1})Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)\nabla\theta_i Q(s, a; \theta_i)\right] \tag{A.9}$$

From Equation A.9, it's clear that the term y_i is treated as a constant during differentiation with respect to θ_i, since it only contains θ_{i-1}. This concludes the derivation.

A.2 Code List

A.2.1 MDP Experiment

```
import numpy as np
from scipy.special import softmax
import matplotlib.pyplot as plt
from copy import deepcopy
import time


def rs(state, action, a, b, c, d, r, l):
    stp = rt = None
    if state == a and action == r:
        stp, rt = c, 0
    elif state == a and action == l:
        stp, rt = b, 0
    elif state == b:
        stp, rt = d, rv + np.random.randn()
    return stp, rt


def rbar(state, action, a, b, r, l, rv):
```

```
    if state == a and (action == r or action == l):
        return 0
    elif state == b:
        return rv
    else:
        return 0

def update(state, action, Q, a, b, c, d, r, l):
    maxQ = np.zeros(4)
    for s in range(4):
        maxQ[s] = np.max(Q[s])
    if state == a and action == r:
        return maxQ[c]
    elif state == a and action == l:
        return maxQ[b]
    elif state == b:
        return maxQ[d]
    else:
        return 0


def zeroQ(Q):
    for i in range(4):
        Q[i].fill(0.0)

def alphat(alpha0, t):
    return alpha0 / (1 + 0.001 * t)

def softprob(x):
    x = x - np.max(x)
    x = np.exp(x)
    return x / np.sum(x)

def argmaxR(x):
    p = softmax(1000 * x)
    return np.random.choice(np.arange(len(x)), p=p)

# define states and actions
a, b, c, d = 0, 1, 2, 3
l, r = 0, 1

rv = -0.1
gamma = 0.99
alpha0 = 0.1
eps = 0.1
Exps = 1000
```

```
Episodes = 2000

# Initialization
Q = [np.zeros((2,)), np.zeros((8,)), np.zeros((1,)), np.zeros((1,))]
zeroQ(Q)

QA = [np.copy(q) for q in Q]
QB = [np.copy(q) for q in Q]

# Get the true solution to the MDP
Qnew = deepcopy(Q)
zeroQ(Qnew)

for _ in range(1000):
    for state in [a, b, c, d]:
        for action in range(len(Q[state])):
            Qnew[state][action] = rbar(state, action, a, b, r, l, rv)
            + gamma * update(state, action, Q, a, b, c, d, r, l)
    Q = deepcopy(Qnew)

QmaxTrue = Q[a][1]

# Initialize other necessary variables
Qend = np.zeros((Exps, Episodes))
dQend = np.zeros((Exps, Episodes))
sQend = np.zeros((Exps, Episodes))
cQend = np.zeros((Exps, Episodes))

# Standard Q learning
start_time = time.time()
actionsQ = []

for experiment in range(Exps):
    zeroQ(Q)
    ct = [0]
    actionsQ_ex = []
    for episode in range(Episodes):
        st = [a]
        at = [0] # dummy initialization
        while True:
            ct[0] += 1
            alpha = alphat(alpha0, ct[0])
            if np.random.rand() > eps: # select action using maxQ
                at[0] = argmaxR(Q[st[0]])
            else:
                at[0] = np.random.choice([0, 1])
```

```
            actionsQ_ex.append(at[0])
            stp, rt = rs(st[0], at[0], a, b, c, d, r, l)
            Q[st[0]][at[0]] += alpha * (rt + gamma * np.max(Q[stp]) -
                                        Q[st[0]][at[0]])
            if stp == d or stp == c:
                break
            st[0] = stp
        Qend[experiment, episode] = Q[a][l]
    actionsQ.append(actionsQ_ex)
print("Standard Q Learning took %s seconds" % (time.time()
        - start_time))

# Double Q learning
start_time = time.time()
actionsdQ = []
for experiment in range(Exps):
    zeroQ(QA)
    zeroQ(QB)
    ct = [0]
    actionsdQ_ex = []
    for episode in range(Episodes):
        st = [a]
        at = [1]
        while True:
            ct[0] += 1
            alpha = alphat(alpha0, ct[0])
            if np.random.rand() > eps: # select action using maxQ
                at[0] = argmaxR(QA[st[0]])
            else:
                at[0] = np.random.choice([0, 1])
            actionsdQ_ex.append(at[0])
            stp, rt = rs(st[0], at[0], a, b, c, d, r, l)
            if np.random.rand() > 0.5: # update A
                astar = argmaxR(QA[stp])
                QA[st[0]][at[0]] += alpha * (rt + gamma *
                            QB[stp][astar] - QA[st[0]][at[0]])
            else: # update B
                bstar = argmaxR(QB[stp])
                QB[st[0]][at[0]] += alpha * (rt + gamma *
                            QA[stp][bstar] - QB[st[0]][at[0]])
            if stp == d or stp == c:
                break
            st[0] = stp
        dQend[experiment, episode] = QA[a][l]
    actionsdQ.append(actionsdQ_ex)
print("Double Q Learning took %s seconds" % (time.time() - start_time))
```

```
158
159 # Smoothed Q learning (softmax)
160 start_time = time.time()
161 actionssQ = []
162 q = deepcopy(Q)
163 for experiment in range(Exps):
164     zeroQ(Q)
165     ct = [0]
166     actionssQ_ex = []
167     for episode in range(Episodes):
168         st = [a]
169         at = [0] # dummy
170         # softmax
171         beta = 0.1 + 0.1 * (episode - 1)
172         while True:
173             for s in range(4):
174                 q[s] = softmax(beta * Q[s])
175             ct[0] += 1
176             alpha = alphat(alpha0, ct[0])
177             if np.random.rand() > eps: # select action using maxQ
178                 at[0] = argmaxR(Q[st[0]])
179             else:
180                 at[0] = np.random.choice([0, 1])
181             actionssQ_ex.append(at[0])
182             stp, rt = rs(st[0], at[0], a, b, c, d, r, l)
183             Q[st[0]][at[0]] += alpha * (rt + gamma * np.sum(q[stp] *
184                                 Q[stp]) - Q[st[0]][at[0]])
185             if stp == d or stp == c:
186                 break
187             st[0] = stp
188         sQend[experiment, episode] = Q[a][1]
189     actionssQ.append(actionssQ_ex)
190 print("Smoothed Q Learning (softmax) took %s seconds" % (time.time() -
191         start_time))
192
193 # Clipped max smoothed Q learning
194 start_time = time.time()
195 actionscQ = []
196 q = deepcopy(Q)
197 for experiment in range(Exps):
198     zeroQ(Q)
199     ct = [0]
200     actionscQ_ex = []
201     for episode in range(Episodes):
202         st = [a]
203         at = [1]
```

```
        mu = np.exp(-0.02 * episode)
        A = 2 # number of actions
        while True:
            for s in range(4):
                A = len(Q[s])
                # q[s] = np.ones(A) * mu / (A - 1)
                q[s] = np.ones(A) * mu / (A - 1) if A != 1
                        else np.ones(A)
                as_ = argmaxR(Q[s])
                q[s][as_] = 1 - mu
                if len(q[s]) == 1:
                    q[s] = 1.0
            ct[0] += 1
            alpha = alphat(alpha0, ct[0])
            if np.random.rand() > eps: # select action using maxQ
                at[0] = argmaxR(Q[st[0]])
            else:
                at[0] = np.random.choice([0, 1])
            actionscQ_ex.append(at[0])
            stp, rt = rs(st[0], at[0], a, b, c, d, r, l)
            Q[st[0]][at[0]] += alpha * (rt + gamma * np.sum(q[stp] *
                            Q[stp]) - Q[st[0]][at[0]])
            if stp == d or stp == c:
                break
            st[0] = stp
        cQend[experiment, episode] = Q[a][l]
    actionscQ.append(actionscQ_ex)
print("Smoothed Q Learning Clipped max took %s seconds" % (time.time()
        - start_time))



avg_abs_error_Q = np.mean(np.abs(Qend - QmaxTrue), axis=0)
avg_abs_error_dQ = np.mean(np.abs(dQend - QmaxTrue), axis=0)
avg_abs_error_sQ = np.mean(np.abs(sQend - QmaxTrue), axis=0)
avg_abs_error_cQ = np.mean(np.abs(cQend - QmaxTrue), axis=0)


min_length = min(min(len(sublist) for sublist in actionsQ),
                 min(len(sublist) for sublist in actionsdQ),
                 min(len(sublist) for sublist in actionssQ),
                 min(len(sublist) for sublist in actionscQ))

actionsQ_trimmed = [sublist[:min_length] for sublist in actionsQ]
actionsdQ_trimmed = [sublist[:min_length] for sublist in actionsdQ]
actionssQ_trimmed = [sublist[:min_length] for sublist in actionssQ]
```

```
actionscQ_trimmed = [sublist[:min_length] for sublist in actionscQ]

avg_action_Q = np.mean([[action == 1 for action in sublist]
for sublist in actionsQ_trimmed], axis=0)
avg_action_dQ = np.mean([[action == 1 for action in sublist]
for sublist in actionsdQ_trimmed], axis=0)
avg_action_sQ = np.mean([[action == 1 for action in sublist]
for sublist in actionssQ_trimmed], axis=0)
avg_action_cQ = np.mean([[action == 1 for action in sublist]
for sublist in actionscQ_trimmed], axis=0)

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

# Plot the average absolute errors for each method
axs[0].plot(avg_abs_error_Q, label='Q learning')
axs[0].plot(avg_abs_error_dQ, label='Double Q learning')
axs[0].plot(avg_abs_error_sQ, label='Smoothed (softmax) Q learning')
axs[0].plot(avg_abs_error_cQ, label='Smoothed (clipmax) Q learning')
axs[0].legend()
axs[0].set_xlabel('Episodes')
axs[0].set_ylabel('Average Absolute Error')
axs[0].set_title('Comparison of Q Learning Methods')
axs[0].grid(True)

# Plot the average actions for each method
axs[1].plot(avg_action_Q, label='Q learning')
axs[1].plot(avg_action_dQ, label='Double Q learning')
axs[1].plot(avg_action_sQ, label='Smoothed (softmax) Q learning')
axs[1].plot(avg_action_cQ, label='Smoothed (clipmax) Q learning')
axs[1].legend()
axs[1].set_xlabel('Episodes')
axs[1].set_ylabel('Average Actions')
axs[1].set_title('Comparison of Actions in Q Learning Methods')
axs[1].grid(True)
# Display the figure
plt.show()
```

A.2.2 Roulette Experiment

Roulette.py

```
# Import necessary libraries
import numpy as np
import gym
from gym import spaces
```

```
import math

# Define a Roulette environment class, inherited from gym.Env
class RouletteEnv(gym.Env):
    def __init__(self):
        # Inherited gym.Env
        super(RouletteEnv, self).__init__()
        # Define a discrete action space with 171 possible actions
        self.action_space = spaces.Discrete(171)
        # Define an observation space with 1 possible state
        self.observation_space = spaces.Discrete(1)
        # Initialize a counter for the current step
        self.current_step = 0

    def step(self, action):
        if action == 170:  # If the action is to stop playing
            return 0, 0, True, {}
        # Choose a random winning number from 0 to 36
        winning_number = np.random.choice(37)
        # Assume player bets $1 each time and initialize the reward as -1
        reward = -1

        # If player bets on a number and wins, they get a reward of 35
        if action < 36 and action == winning_number:
            reward = 35

        # If player bets on red or black and wins, they get a reward of 1
        elif action == 36 or action == 37:
            if (action == 36 and winning_number % 2 == 1)
                or (action == 37 and winning_number % 2 == 0):
                reward = 1

        # Bet on even or odd
        elif action == 38 or action == 39:
            if (action == 38 and winning_number % 2 == 0) or
                (action == 39 and winning_number % 2 == 1):
                reward = 1

        # Bet on low (1-18) or high (19-36)
        elif action == 40 or action == 41:
            if (action == 40 and 1 <= winning_number <= 18) or
                (action == 41 and 19 <= winning_number <= 36):
                reward = 1

        # Bet on first 12, second 12, or third 12
        elif action >= 42 and action <= 44:
```

```
            if (action == 42 and 1 <= winning_number <= 12) or \
                    (action == 43 and 13 <= winning_number <= 24) or \
                    (action == 44 and 25 <= winning_number <= 36):
                reward = 2

        # Bet on column
        elif action >= 45 and action <= 47:
            if (action == 45 and winning_number % 3 == 1) or \
                    (action == 46 and winning_number % 3 == 2) or \
                    (action == 47 and winning_number % 3 == 0):
                reward = 2

        # Bet on two adjacent numbers
        elif action >= 48 and action <= 82:
            if winning_number in [(action - 48), (action - 47)]:
                reward = 17

        # Bet on two numbers at the column intersection
        elif action >= 83 and action <= 94:
            columns = [(0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36),
                       (1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34),
                       (2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35)]
            column_pairs = [(columns[0], columns[1]), (columns[1],
                             columns[2])]
            for pair in column_pairs:
                if winning_number in pair[0] and winning_number
                    in pair[1]:
                    reward = 17

        return 0, reward, False, {}

    def reset(self):
        # Reset the environment to its initial state
        self.current_step = 0
        return 0



# Define a function for selecting an action using epsilon-greedy strategy
def epsilon_greedy(Q, state, epsilon):
    if np.random.uniform(0, 1) < epsilon:
        # Randomly select an action
        action = np.random.choice(Q.shape[1])
    else:
        # Select the action with the highest estimated reward
        action = np.argmax(Q[state])
```

```
    return action

# Define a function for calculating a polynomial learning rate
def polynomial_learning_rate(base, exponent, t):
    t = max(t, 1)  # Ensure t is not zero
    return base / (t ** exponent)

def decay_epsilon(epsilon_start, epsilon_end, decay_rate, i_episode):
    return epsilon_end + (epsilon_start - epsilon_end) *
            math.exp(-decay_rate * i_episode)


# The Q_learning function implements the standard Q-learning algorithm.
# It takes as arguments the environment, the number of episodes,
# the base learning rate, learning rate exponent, epsilon,
# and the discount factor gamma.

def Q_learning(env, num_episodes, base_learning_rate,
    learning_rate_exponent, epsilon_start, epsilon_end,
    decay_rate, gamma):
    Q_table = np.zeros([env.observation_space.n, env.action_space.n])
    n_table = np.zeros_like(Q_table)
    MaxQ_table = np.full([env.observation_space.n, env.action_space.n], 0)
    abs_errors = []
    avg_action_values = []
    for i_episode in range(1, num_episodes + 1):
        state = env.reset()
        done = False
        alpha = polynomial_learning_rate(base_learning_rate,
                learning_rate_exponent, i_episode)
        epsilon = decay_epsilon(epsilon_start, epsilon_end,
                    decay_rate, i_episode)
        actions = []
        while not done:
            action = epsilon_greedy(Q_table, state, epsilon)
            n_table[state, action] += 1
            # alpha = 1 / n_table[state, action]
            actions.append(action)
            next_state, reward, done, info = env.step(action)
            old_value = Q_table[state][action]
            Q_table[state][action] = (1 - alpha) *
                                old_value + alpha *
                                (reward + gamma * np.max(Q_table[next_state]))
            state = next_state
        abs_errors.append(np.mean(np.abs(Q_table - MaxQ_table)))
        avg_action_values.append(np.mean(Q_table[state, :]))
```

```
143     return Q_table, abs_errors, avg_action_values
144
145
146 def double_Q_learning(env, num_episodes, base_learning_rate,
147     learning_rate_exponent, epsilon_start, epsilon_end,
148     decay_rate, gamma):
149     Q1_table = np.zeros([env.observation_space.n, env.action_space.n])
150     Q2_table = np.zeros([env.observation_space.n, env.action_space.n])
151     n1_table = np.zeros_like(Q1_table)
152     n2_table = np.zeros_like(Q2_table)
153     MaxQ_table = np.full([env.observation_space.n, env.action_space.n], 0)
154     abs_errors = []
155     avg_action_values = []
156     for i_episode in range(1, num_episodes + 1):
157         state = env.reset()
158         done = False
159         alpha = polynomial_learning_rate(base_learning_rate,
160                 learning_rate_exponent, i_episode)
161         epsilon = decay_epsilon(epsilon_start, epsilon_end, decay_rate,
162                                 i_episode)
163         actions = []
164         while not done:
165             action = epsilon_greedy(Q1_table + Q2_table, state, epsilon)
166             actions.append(action)
167             next_state, reward, done, info = env.step(action)
168             if np.random.binomial(1, 0.5) == 1:
169                 n1_table[state, action] += 1
170                 # alpha = 1 / n1_table[state, action]
171                 best_next_action = np.argmax(Q1_table[next_state])
172                 Q1_table[state][action] = (1 - alpha) *
173                     Q1_table[state][action] + alpha * (reward +
174                     gamma * Q2_table[next_state][best_next_action])
175             else:
176                 n2_table[state, action] += 1
177                 alpha = 1 / n2_table[state, action]
178                 best_next_action = np.argmax(Q2_table[next_state])
179                 Q2_table[state][action] = (1 - alpha) *
180                     Q2_table[state][action] + alpha * (reward +
181                     gamma * Q1_table[next_state][best_next_action])
182             state = next_state
183         abs_errors.append(np.mean(np.abs((Q1_table + Q2_table) / 2 -
184                         MaxQ_table)))
185         avg_action_values.append(np.mean((Q1_table + Q2_table)
186                         [state, :] / 2))
187     return Q1_table, Q2_table, abs_errors, avg_action_values
188
```

```
189 def smoothed_Q_learning(env, num_episodes, base_learning_rate, learning_rate_exponent
190     Q_table = np.zeros([env.observation_space.n, env.action_space.n])
191     n_table = np.zeros_like(Q_table)
192     MaxQ_table = np.full([env.observation_space.n, env.action_space.n], 0)
193     abs_errors = []
194     avg_action_values = []
195 
196     def softmax(Q_state, beta):
197         Q_normalized = (Q_state - np.mean(Q_state)) / (
198                     # Added epsilon to avoid division by zero
199                     np.std(Q_state) + 1e-10)
200         exps = np.exp(beta * Q_normalized)
201         probs = exps / np.sum(exps)
202         # Ensure probabilities sum to 1
203         probs = probs / np.sum(probs)
204         if np.isnan(probs).any():
205             probs = np.ones_like(Q_state) / len(Q_state)
206         return probs
207 
208     def clipped_softmax(Q_state, beta):
209         Q_clipped = np.ones_like(Q_state) * (-np.inf)
210         top_k_indices = np.argsort(Q_state)[-3:]
211         Q_clipped[top_k_indices] = Q_state[top_k_indices] - np.max(Q_state)
    # Subtract the maximum value
212         exps = np.exp(beta * Q_clipped)
213         return exps / np.sum(exps)
214 
215     def argmaxR(Q_state):
216         max_value_indices = np.argwhere(Q_state == np.amax(Q_state)).flatten().tolist
217         return np.random.choice(max_value_indices)
218 
219     for i_episode in range(num_episodes):
220         state = env.reset()
221         done = False
222         beta = 0.1 + 0.1 * (i_episode + 1)
223         mu = np.exp(-0.02 * i_episode)
224         actions = []
225         epsilon = decay_epsilon(epsilon_start, epsilon_end, decay_rate, i_episode)
226         alpha = polynomial_learning_rate(base_learning_rate, learning_rate_exponent, 
227 
228         while not done:
229             if np.random.rand() > epsilon:
230                 action = argmaxR(Q_table[state])
231             else:
232                 if selection_method == 'softmax':
233                     action = np.random.choice(np.arange(env.action_space.n),
```

```
                    p=softmax(Q_table[state], beta))
                elif selection_method == 'clipped_max':
                    A = len(Q_table[state])
                    action_distribution = np.full(A, mu / (A - 1))
                    action_distribution[argmaxR(Q_table[state])] = 1 - mu
                    action = np.random.choice(np.arange(env.action_space.n),
                    p=action_distribution)
                elif selection_method == 'clipped_softmax':
                    action = np.random.choice(np.arange(env.action_space.n),
                    p=clipped_softmax(Q_table[state], beta))

            actions.append(action)
            n_table[state, action] += 1
            # alpha = 1 / n_table[state, action]
            next_state, reward, done, _ = env.step(action)

            if selection_method == 'softmax':
                action_distribution = softmax(Q_table[next_state], beta)
            elif selection_method == 'clipped_max':
                A = len(Q_table[next_state])
                action_distribution = np.ones(A) * mu / (A - 1)
                                    if A != 1 else np.ones(A)
                as_ = argmaxR(Q_table[next_state])
                action_distribution[as_] = 1 - mu
            elif selection_method == 'clipped_softmax':
                action_distribution = clipped_softmax(Q_table[next_state], beta)

            expected_return = reward + gamma *
                np.sum(Q_table[next_state] * action_distribution)
            Q_table[state][action] = Q_table[state][action] + alpha *
                        (expected_return - Q_table[state][action])
            state = next_state

        abs_errors.append(np.mean(np.abs(Q_table - MaxQ_table)))
        avg_action_values.append(np.mean(Q_table[state, :]))

    return Q_table, abs_errors, avg_action_values
```

run.py

```
from RouletteEnv import *
import matplotlib.pyplot as plt

# Create the environment
env = RouletteEnv()

```

```
# Set the parameters for the experiment
num_episodes = 100000
base_learning_rate = 0.3
learning_rate_exponent = 0.5
epsilon_start = 1.0
epsilon_end = 0.1
decay_rate = 0.0001
gamma = 0.99
delta_start = 1.0

# Run the experiments
_, Q_learning_abs_errors, Q_learning_avg_action_values =
    Q_learning(env, num_episodes, base_learning_rate, learning_rate_exponent,
    epsilon_start, epsilon_end, decay_rate, gamma)
print("Q Learning Completed!")
_, _, double_Q_learning_abs_errors, double_Q_learning_avg_action_values =
    double_Q_learning(env, num_episodes, base_learning_rate, learning_rate_exponent,
    epsilon_start, epsilon_end, decay_rate, gamma)
print("Double Q Learning Completed!")
_, softmax_smoothed_abs_errors, softmax_smoothed_avg_action_values =
smoothed_Q_learning(env, num_episodes, base_learning_rate, learning_rate_exponent,
epsilon_start, epsilon_end, decay_rate, gamma, "softmax")
print("Softmax Smooth Q Learning Completed!")
_, clipped_max_smoothed_abs_errors, clipped_max_smoothed_avg_action_values =
smoothed_Q_learning(env, num_episodes, base_learning_rate, learning_rate_exponent,
epsilon_start, epsilon_end, decay_rate, gamma, "clipped_max")
print("Clipped max Smooth Q Learning Completed!")
_, clipped_softmax_smoothed_abs_errors, clipped_softmax_smoothed_avg_action_values =
smoothed_Q_learning(env, num_episodes, base_learning_rate, learning_rate_exponent,
epsilon_start, epsilon_end, decay_rate, gamma, "clipped_softmax")
print("Clipped Softmax Smooth Q Learning Completed!")


# Create subplots
fig, axs = plt.subplots(2, figsize=(12, 16))

# Plot the results for expected profit
axs[0].plot(range(num_episodes), Q_learning_avg_action_values,
label='Q-learning', color='blue')
axs[0].plot(range(num_episodes), double_Q_learning_avg_action_values,
label='Double Q-learning', color='orange')
axs[0].plot(range(num_episodes), softmax_smoothed_avg_action_values,
label='Softmax Smoothed Q-learning', color='green')
axs[0].plot(range(num_episodes), clipped_max_smoothed_avg_action_values,
label='Clipped max Smoothed Q-learning', color='red')
axs[0].plot(range(num_episodes), clipped_softmax_smoothed_avg_action_values,
```

```
label='Clipped Softmax Smoothed Q-learning', color='purple')
axs[0].set_xlabel('Number of Trials')
axs[0].set_ylabel('Expected Profit')
axs[0].legend()

# Plot the results for abs_errors
axs[1].plot(range(num_episodes), Q_learning_abs_errors,
label='Q-learning', color='blue')
axs[1].plot(range(num_episodes), double_Q_learning_abs_errors,
label='Double Q-learning', color='orange')
axs[1].plot(range(num_episodes), softmax_smoothed_abs_errors,
label='Softmax Smoothed Q-learning', color='green')
axs[1].plot(range(num_episodes), clipped_max_smoothed_abs_errors,
label='Clipped max Smoothed Q-learning', color='red')
axs[1].plot(range(num_episodes), clipped_softmax_smoothed_abs_errors,
label='Clipped Softmax Smoothed Q-learning', color='purple')
axs[1].set_xlabel('Number of Episodes')
axs[1].set_ylabel('Abs Errors')
axs[1].legend()

# Show the plots
plt.show()
```

A.2.3 Grid World

gridworld.py

```
import numpy as np

class GridWorld:
    def __init__(self, grid_size=3, gamma=0.9, penalty=-12, reward=10,
    goal_reward=5, max_steps=5):
        self.grid_size = grid_size
        self.gamma = gamma
        self.penalty = penalty
        self.reward = reward
        self.goal_reward = goal_reward
        self.max_steps = max_steps
        self.state_visits = np.zeros((grid_size, grid_size))
        self.reset()
        self.start = (0, 0)  # Define the start position

    def reset(self):
        self.x = 0
        self.y = 0
        self.steps = 0
```

```
        self.state_visits[:] = 0
        return (self.x, self.y)

    def step(self, action):
        self.state_visits[self.x, self.y] += 1

        if action == 0 and self.y != 0:
            self.y -= 1
        elif action == 1 and self.x != self.grid_size-1:
            self.x += 1
        elif action == 2 and self.y != self.grid_size-1:
            self.y += 1
        elif action == 3 and self.x != 0:
            self.x -= 1

        self.steps += 1

        if self.x == self.grid_size - 1 and self.y == self.grid_size - 1:
            return (self.x, self.y), self.goal_reward, True
        elif self.steps >= self.max_steps:
            return (self.x, self.y), 0, True
        else:
            return (self.x, self.y), np.random.choice([self.penalty,
            self.reward], p=[0.5, 0.5]), False

    def get_epsilon(self, state):
        x, y = state
        n_s = self.state_visits[x, y]
        return 1 / np.sqrt(n_s + 1)
```

run.py

```
import numpy as np
import matplotlib.pyplot as plt
from gridworld import GridWorld

def Q_learning(env, num_episodes, max_steps):
    Q_table = np.zeros((env.grid_size, env.grid_size, 4))
    # Keep track of the counts
    N_table = np.zeros((env.grid_size, env.grid_size, 4))
    rewards = np.zeros(num_episodes)
    max_action_values = []

    for episode in range(num_episodes):
        state = env.reset()
        done = False
```

```
        total_reward = 0
        steps = 0
        # alpha = 1 / (episode + 1)  # Decaying learning rate
        while not done and steps < max_steps:
            # epsilon = env.get_epsilon(state)
            epsilon = 1 / np.sqrt(episode + 1)  # Decaying epsilon-greedy
            if np.random.uniform(0, 1) < epsilon:
                action = np.random.randint(0, 4)  # Explore
            else:
                action = np.argmax(Q_table[state])  # Exploit
            next_state, reward, done = env.step(action)
            total_reward += reward
            steps += 1
            N_table[state][action] += 1  # Increase the count
            # Decaying learning rate based on the count
            alpha = 1 / N_table[state][action]
            target = reward + env.gamma * np.max(Q_table[next_state])
            Q_table[state][action] = Q_table[state][action] + alpha *
            (target - Q_table[state][action])
            state = next_state
        rewards[episode] = total_reward
        max_action_values.append(np.max(Q_table[env.start]))
    average_rewards = np.cumsum(rewards) / (np.arange(num_episodes) + 1)
    return average_rewards, max_action_values

def double_Q_learning(env, num_episodes, max_steps):
    Q1_table = np.zeros((env.grid_size, env.grid_size, 4))
    Q2_table = np.zeros((env.grid_size, env.grid_size, 4))
    # Keep track of the counts
    N_table = np.zeros((env.grid_size, env.grid_size, 4))
    rewards = np.zeros(num_episodes)
    max_action_values = []

    for episode in range(num_episodes):
        state = env.reset()
        done = False
        total_reward = 0
        steps = 0

        while not done and steps < max_steps:
            # epsilon = env.get_epsilon(state)
            epsilon = 1 / np.sqrt(episode + 1)  # Decaying epsilon
            if np.random.uniform(0, 1) < epsilon:
                action = np.random.randint(0, 4)  # Explore
            else:
                action = np.argmax(Q1_table[state] + Q2_table[state])  # Exploit
```

```
            next_state, reward, done = env.step(action)
            total_reward += reward
            steps += 1
            N_table[state][action] += 1  # Increase the count
            # Decaying learning rate based on the count
            alpha = 1 / N_table[state][action]
            if np.random.uniform(0, 1) < 0.5:
                target = reward + env.gamma *
                Q2_table[next_state][np.argmax(Q1_table[next_state])]
                Q1_table[state][action] = Q1_table[state][action] + alpha
                * (target - Q1_table[state][action])
            else:
                target = reward + env.gamma * Q1_table[next_state]
                [np.argmax(Q2_table[next_state])]
                Q2_table[state][action] = Q2_table[state][action] +
                alpha * (target - Q2_table[state][action])
            state = next_state
        rewards[episode] = total_reward
        max_action_values.append(np.max(Q1_table[env.start] + Q2_table[env.start]))
    average_rewards = np.cumsum(rewards) / (np.arange(num_episodes) + 1)
    return average_rewards, max_action_values


def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)

def clipped_max(Q_values, tau):
    A = len(Q_values)
    qt = np.ones(A) * tau / (A - 1)
    qt[np.argmax(Q_values)] = 1 - tau
    return qt

def clipped_softmax(Q_values, beta):
    sorted_indices = np.argsort(Q_values)[-3:]
    clipped_Q_values = np.full(Q_values.shape, -np.inf)
    clipped_Q_values[sorted_indices] = Q_values[sorted_indices]
    e_x = np.exp(beta * clipped_Q_values)
    return e_x / e_x.sum()

def smoothed_Q_learning(env, num_episodes, max_steps, smoothing_strategy="softmax"):
    Q_table = np.zeros((env.grid_size, env.grid_size, 4))
     # Keep track of the counts
    N_table = np.zeros((env.grid_size, env.grid_size, 4))
    rewards = np.zeros(num_episodes)
    max_action_values = []
```

```
107
108     for episode in range(num_episodes):
109         state = env.reset()
110         done = False
111         total_reward = 0
112         steps = 0
113
114         beta = 0.1 + 0.1 * episode  # Decaying beta
115         tau = np.exp(-0.02 * episode)  # Decaying tau
116
117         while not done and steps < max_steps:
118             epsilon = 1 / np.sqrt(episode + 1)  # Decaying epsilon
119             if np.random.uniform(0, 1) < epsilon:
120                 action = np.random.randint(0, 4)  # Explore
121             else:
122             #     Big Q Action Selection
123             #     action = np.argmax(Q_table[state])  # Exploit
124             #     Small Q Action Selection
125                 if smoothing_strategy == "softmax":
126                     qt = softmax(Q_table[state])
127                 elif smoothing_strategy == "clipped_softmax":
128                     qt = clipped_softmax(Q_table[state], beta)
129                 elif smoothing_strategy == "clipped_max":
130                     qt = clipped_max(Q_table[state], tau)
131
132                 # Exploit based on max q-value
133                 action = np.argmax(qt * Q_table[state])
134
135             next_state, reward, done = env.step(action)
136             total_reward += reward
137             steps += 1
138             N_table[state][action] += 1  # Increase the count
139             # Decaying learning rate based on the count
140             alpha = 1 / N_table[state][action]
141
142             if smoothing_strategy == "softmax":
143                 qt = softmax(Q_table[next_state])
144             elif smoothing_strategy == "clipped_softmax":
145                 qt = clipped_softmax(Q_table[next_state], beta)
146             elif smoothing_strategy == "clipped_max":
147                 qt = clipped_max(Q_table[next_state], tau)
148
149             target = reward + env.gamma * np.sum(qt * Q_table[next_state])
150             Q_table[state][action] = Q_table[state][action] + alpha *
151             (target - Q_table[state][action])
152
```

```
            state = next_state

        rewards[episode] = total_reward
        max_action_values.append(np.max(Q_table[env.start]))

    average_rewards = np.cumsum(rewards) / (np.arange(num_episodes) + 1)

    return average_rewards, max_action_values



num_experiments = 10000
num_episodes = 10000
max_steps = 5

Q_rewards = []
double_Q_rewards = []
Q_max_action_values = []
double_Q_max_action_values = []
smoothed_Q_softmax_rewards = []
smoothed_Q_softmax_max_action_values = []
smoothed_Q_clipped_max_rewards = []
smoothed_Q_clipped_max_max_action_values = []
smoothed_Q_clipped_softmax_rewards = []
smoothed_Q_clipped_softmax_max_action_values = []


for _ in range(num_experiments):
    env = GridWorld()
    rewards, max_action_values = Q_learning(env, num_episodes, max_steps)
    Q_rewards.append(rewards)
    Q_max_action_values.append(max_action_values)

    env = GridWorld()
    rewards, max_action_values = double_Q_learning(env, num_episodes, max_steps)
    double_Q_rewards.append(rewards)
    double_Q_max_action_values.append(max_action_values)

    env = GridWorld()
    rewards, max_action_values = smoothed_Q_learning(env, num_episodes,
                        max_steps, smoothing_strategy="softmax")
    smoothed_Q_softmax_rewards.append(rewards)
    smoothed_Q_softmax_max_action_values.append(max_action_values)

    env = GridWorld()
    rewards, max_action_values = smoothed_Q_learning(env, num_episodes,
```

```
                            max_steps, smoothing_strategy="clipped_max")
    smoothed_Q_clipped_max_rewards.append(rewards)
    smoothed_Q_clipped_max_max_action_values.append(max_action_values)

    env = GridWorld()
    rewards, max_action_values = smoothed_Q_learning(env, num_episodes,
                            max_steps, smoothing_strategy="clipped_softmax")
    smoothed_Q_clipped_softmax_rewards.append(rewards)
    smoothed_Q_clipped_softmax_max_action_values.append(max_action_values)

Q_rewards = np.mean(np.array(Q_rewards), axis=0)
double_Q_rewards = np.mean(np.array(double_Q_rewards), axis=0)
Q_max_action_values = np.mean(Q_max_action_values, axis=0)
double_Q_max_action_values = np.mean(double_Q_max_action_values, axis=0)
smoothed_Q_softmax_rewards = np.mean(np.array(smoothed_Q_softmax_rewards),
                                axis=0)
smoothed_Q_softmax_max_action_values = np.mean(
                                smoothed_Q_softmax_max_action_values, axis=0)
smoothed_Q_clipped_max_rewards = np.mean(np.array(
                                smoothed_Q_clipped_max_rewards), axis=0)
smoothed_Q_clipped_max_max_action_values =
np.mean(smoothed_Q_clipped_max_max_action_values, axis=0)
smoothed_Q_clipped_softmax_rewards =
np.mean(np.array(smoothed_Q_clipped_softmax_rewards), axis=0)
smoothed_Q_clipped_softmax_max_action_values =
np.mean(np.array(smoothed_Q_clipped_softmax_max_action_values), axis=0)

plt.figure(figsize=(12, 8))

plt.subplot(2, 1, 1)
plt.plot(Q_rewards, label="Q-learning", color="blue")
plt.plot(double_Q_rewards, label="Double Q-learning", color="orange")
plt.plot(smoothed_Q_softmax_rewards,
label="Smoothed Q-learning (Softmax)", color="green")
plt.plot(smoothed_Q_clipped_max_rewards,
label="Smoothed Q-learning (Clipped Max)", color="red")
plt.plot(smoothed_Q_clipped_softmax_rewards,
label="Smoothed Q-learning (Clipped Softmax)", color="purple")
plt.xlabel("Steps")
plt.ylabel("Average Reward")
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(Q_max_action_values, label="Q-learning", color="blue")
plt.plot(double_Q_max_action_values, label="Double Q-learning", color="orange")
plt.plot(smoothed_Q_softmax_max_action_values,
```

```
label="Smoothed Q-learning (Softmax)", color="green")
plt.plot(smoothed_Q_clipped_max_max_action_values,
label="Smoothed Q-learning (Clipped Max)", color="red")
plt.plot(smoothed_Q_clipped_softmax_max_action_values,
label="Smoothed Q-learning (Clipped Softmax)", color="purple")
plt.xlabel("Steps")
plt.ylabel("Max Action Value at Start State")
plt.legend()

plt.show()
```

A.2.4 CartPole

CartPole.py

```
import gymnasium as gym
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F


all_runs_durations = []
# stop_outer_loop = False  # Flag variable

for run in range(10):


    env = gym.make("CartPole-v1")

    episode_durations = []


    # set up matplotlib
    is_ipython = 'inline' in matplotlib.get_backend()
    if is_ipython:
        from IPython import display

    plt.ion()
```

```
32
33  # if GPU is to be used
34  device = torch.device("cuda")
35
36  Transition = namedtuple('Transition',
37                          ('state', 'action', 'next_state', 'reward'))
38
39
40  class ReplayMemory(object):
41
42      def __init__(self, capacity):
43          self.memory = deque([], maxlen=capacity)
44
45      def push(self, *args):
46          """Save a transition"""
47          self.memory.append(Transition(*args))
48
49      def sample(self, batch_size):
50          return random.sample(self.memory, batch_size)
51
52      def __len__(self):
53          return len(self.memory)
54
55
56  class DQN(nn.Module):
57
58      def __init__(self, n_observations, n_actions):
59          super(DQN, self).__init__()
60          self.layer1 = nn.Linear(n_observations, 128)
61          self.layer2 = nn.Linear(128, 128)
62          self.layer3 = nn.Linear(128, n_actions)
63
64      # Called with either one element to determine next action, or a batch
65      # during optimization. Returns tensor([[left0exp,right0exp]...]).
66      def forward(self, x):
67          x = F.relu(self.layer1(x))
68          x = F.relu(self.layer2(x))
69          return self.layer3(x)
70
71
72  BATCH_SIZE = 128
73  GAMMA = 0.99
74  EPS_START = 0.9
75  EPS_END = 0.05
76  EPS_DECAY = 1000
77  TAU = 0.005
```

```
78  LR = 1e-4
79
80  # Get number of actions from gym action space
81  n_actions = env.action_space.n
82  # Get the number of state observations
83  state, info = env.reset()
84  n_observations = len(state)
85
86  policy_net = DQN(n_observations, n_actions).to(device)
87  target_net = DQN(n_observations, n_actions).to(device)
88  target_net.load_state_dict(policy_net.state_dict())
89
90  optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)
91  memory = ReplayMemory(10000)
92
93  steps_done = 0
94
95
96  def select_action(state):
97      global steps_done
98      sample = random.random()
99      eps_threshold = EPS_END + (EPS_START - EPS_END) * \
100                     math.exp(-1. * steps_done / EPS_DECAY)
101     steps_done += 1
102     if sample > eps_threshold:
103         with torch.no_grad():
104             return policy_net(state).max(1)[1].view(1, 1)
105     else:
106         return torch.tensor([[env.action_space.sample()]],
107         device=device, dtype=torch.long)
108
109
110 episode_durations = []
111
112
113 def plot_durations(show_result=False):
114     plt.figure(1)
115     durations_t = torch.tensor(episode_durations, dtype=torch.float)
116     if show_result:
117         plt.title('Result')
118     else:
119         plt.clf()
120         plt.title('Training...')
121     plt.xlabel('Episode')
122     plt.ylabel('Duration')
123     plt.plot(durations_t.numpy())
```

```
        # Take 100 episode averages and plot them too
        if len(durations_t) >= 100:
            means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(99), means))
            plt.plot(means.numpy())

        plt.pause(0.001)  # pause a bit so that plots are updated
        if is_ipython:
            if not show_result:
                display.display(plt.gcf())
                display.clear_output(wait=True)
            else:
                display.display(plt.gcf())

global_step = 0
EPSILON_START = 1.0
EPSILON_END = 0.1
DECAY_RATE = 0.01


def decay_epsilon(epsilon_start, epsilon_end, decay_rate, step):
    return epsilon_end + (epsilon_start - epsilon_end)
            * math.exp(-decay_rate * step)


def optimize_model():
    global global_step

    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                       if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    def softmax(x, beta):
        e_x = torch.exp(beta * x - torch.max(beta * x))
        return e_x / e_x.sum(axis=-1).unsqueeze(-1)

    def clipped_max(x, tau, A):
```

```
            # Get the indices of the actions with the maximum Q-values
            max_action_indices = torch.argmax(x, dim=1)

            # Create a tensor of the same shape as x, set to tau / (A-1)
            clipped_probabilities = torch.full_like(x, tau / (A - 1))

            # Set the probability of the action with the maximum Q-value to 1 - tau
            for i, max_index in enumerate(max_action_indices):
                clipped_probabilities[i, max_index] = 1 - tau

            return clipped_probabilities

        def clipped_softmax(x, beta, k):
            # Step 1: Get the indices of the top-k maximum values for each row
            topk_indices = torch.topk(x, k=k, dim=1).indices
            # Step 2: Create a tensor of the same shape as x,
            % set all to negative infinity
            clipped_x = torch.full_like(x, float('-inf'))
            # Step 3: Place the top-k maximum values back into the tensor
            for i, indices in enumerate(topk_indices):
                clipped_x[i, indices] = x[i, indices]
            # Step 4: Apply the softmax function
            e_x = torch.exp(beta * clipped_x - torch.max(beta * clipped_x, dim=1, keep
            return e_x / e_x.sum(dim=1, keepdim=True)


        # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
        # columns of actions taken. These are the actions which would've been taken
        # for each batch state according to policy_net
        state_action_values = policy_net(state_batch).gather(1, action_batch)

        # Compute V(s_{t+1}) for all next states.
        # Expected values of actions for non_final_next_states are computed based
        # on the "older" target_net; selecting their best reward with max(1)[0].
        # This is merged based on the mask, such that we'll have either the expected
        # state value or 0 in case the state was final.
        next_state_values = torch.zeros(BATCH_SIZE, device=device)

        EPSILON = decay_epsilon(EPSILON_START, EPSILON_END, DECAY_RATE, global_step)

        with torch.no_grad():
            # # DQN
            # next_state_values[non_final_mask] =
            # target_net(non_final_next_states).max(1)[0]

            # # DDQN
```

```
            ## Use policy_net to select next action;
            # then use target_net to compute Q-value of that action.
            # next_actions = policy_net(non_final_next_states).max(1)[1].unsqueeze(1)
            # next_state_values[non_final_mask] =
            # target_net(non_final_next_states).gather(1, next_actions).squeeze()

            ## Softmax SDQN
            ## Obtain Q-values for next states using target network
            # q_values = target_net(non_final_next_states)
            ## Calculate dynamic beta
            # beta_dynamic = 0.1 + 0.1 * global_step
            ## Calculate action probabilities using softmax function
            # action_probabilities = softmax(q_values, beta_dynamic)
            ## Compute the weighted Q-values by multiplying action
            ## probabilities with Q-values
            # weighted_q_values = action_probabilities * q_values
            ## Update next state values with the sum of the weighted Q-values,
            ## aligned with the Smoothed Q-Learning update formula
            # next_state_values[non_final_mask] = torch.sum(weighted_q_values, dim=1)

            ## Clipped max SDQN
            ## Obtain Q-values for next states using target network
            # q_values = target_net(non_final_next_states)
            ## Calculate dynamic delta
            # delta_dynamic = math.exp(-0.02 * global_step)
            ## Calculate action probabilities using clipped max function
            # action_probabilities = clipped_max(q_values,
            #                       delta_dynamic, q_values.shape[1])
            ## Compute the weighted Q-values by multiplying
            ## action probabilities with Q-values
            # weighted_q_values = action_probabilities * q_values
            ## Update next state values with the sum of the weighted Q-values,
            ## aligned with the Smoothed Q-Learning update formula
            # next_state_values[non_final_mask] = torch.sum(weighted_q_values, dim=1)

            # Clipped softmax SDQn
            # Obtain Q-values for next states using target network
            q_values = target_net(non_final_next_states)
            # Calculate dynamic beta
            beta_dynamic = 0.1 + 0.1 * global_step
            # Calculate action probabilities
            # using clipped softmax function
            action_probabilities = clipped_softmax(q_values, beta_dynamic, k=2)
            # Compute the weighted Q-values by multiplying
            # action probabilities with Q-values
            weighted_q_values = action_probabilities * q_values
```

```
            # Update next state values with the sum of the
            # weighted Q-values, aligned with the Smoothed Q-Learning update formula
            next_state_values[non_final_mask] = torch.sum(weighted_q_values, dim=1)

        # Compute the expected Q values
        expected_state_action_values = (next_state_values * GAMMA) + reward_batch

        # Compute Huber loss
        criterion = nn.SmoothL1Loss()
        loss = criterion(state_action_values,
                expected_state_action_values.unsqueeze(1))

        # Optimize the model
        optimizer.zero_grad()
        loss.backward()
        # In-place gradient clipping
        torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
        optimizer.step()
        global_step += 1


    #               500episode
    episode_to_reach_500 = None
    num_episodes = 2000
    i_episode = 0
    stop = False

    # for i_episode in range(num_episodes):
    while i_episode < num_episodes and stop is not True:
        # Initialize the environment and get it's state
        state, info = env.reset()
        state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
        for t in count():
            action = select_action(state)
            observation, reward, terminated, truncated, _ = env.step(action.item())
            reward = torch.tensor([reward], device=device)
            done = terminated or truncated

            if terminated:
                next_state = None
            else:
                next_state = torch.tensor(observation, dtype=torch.float32,
                device=device).unsqueeze(0)

            # Store the transition in memory
            memory.push(state, action, next_state, reward)
```

```
308
309             # Move to the next state
310             state = next_state
311
312             # Perform one step of the optimization (on the policy network)
313             optimize_model()
314
315             # Soft update of the target network's weights
316             #                 + (1       )
317             target_net_state_dict = target_net.state_dict()
318             policy_net_state_dict = policy_net.state_dict()
319             for key in policy_net_state_dict:
320                 target_net_state_dict[key] = policy_net_state_dict[key] *
321                             TAU + target_net_state_dict[key] * (1 - TAU)
322             target_net.load_state_dict(target_net_state_dict)
323
324             if done:
325                 episode_durations.append(t + 1)
326                 plot_durations()
327                 if len(episode_durations) >= 100:
328                     last_100_avg = sum(episode_durations[-100:]) / 100
329                     if last_100_avg >= 495:
330                         reached_500_avg_in_episodes = i_episode
331                         stop_outer_loop = True
332                         stop = True
333                         # break
334                 i_episode += 1
335                 break
```