Summer of Science End-Term Report: MDPs and their Applications in AI

Tanay Jha
Student ID: 24B1040
Mentor: Harshitha Inampudi

July 20, 2025

Abstract

This project presents a comprehensive study of Reinforcement Learning (RL), tracing a path from the foundational principles of automata and logic to the practical implementation of advanced, model-free algorithms. The work spans fundamental mathematical constructs, crucial for understanding intelligent systems, to the implementation of algorithms that solve complex sequential decision-making problems under uncertainty. The study begins with an analysis of the exploration-exploitation dilemma using multi-armed bandits, progresses through model-based dynamic programming solutions for known environments, and delves into model-free methods including Monte Carlo, SARSA, and Q-Learning for environments with unknown dynamics. The project culminates in the development of a Deep Q-Network (DQN) from scratch, including a custom neural network library, which is successfully applied to solve the high-dimensional 'LunarLander-v3' environment from Gymnasium. Key results demonstrate the effectiveness of various RL strategies, highlight the critical role of hyperparameter tuning, and validate the power of deep learning for function approximation in complex state spaces. This report documents not only the algorithms and their results but also the practical challenges of implementation and the key theoretical insights gained at each stage of the learning journey.

Contents

1 Introduction

Reinforcement Learning (RL) has emerged as a powerful paradigm within machine learning, enabling agents to learn optimal behaviors in complex and uncertain environments through trial and error. Unlike supervised learning, RL does not rely on labeled datasets but instead on a scalar *reward* signal to guide its learning process. This 'reward signal' methodology, in many ways, is similar to how we humans and other animals learn. It is a form of machine learning where the machine 'learns' more as it grows, much like a child.

I was first introduced to RL through the WiDS 2024 program, where I implemented an RL based agent trying to play the snake game, through various youtube tutorials. This sparked my interest towards RL and broadly ML, and I took a coursera course to understand Machine Learning better. However, both of these were beginner level projects and courses and didnt provide a deep foundational understanding of how RL works, and more importantly, *why* RL works. This project was undertaken to build a systematic, first-principles understanding of RL, progressing from its theoretical underpinnings to the implementation of state-of-the-art algorithms capable of solving challenging problems.

The report is structured as a narrative of this learning journey, reflecting a deliberate progression from simple, abstract concepts to complex, practical applications:

1. Formal Foundations: Logic and Automata Theory: This section revisits formal systems for computation and state representation. These logical structures provide the essential tools for precisely defining states, transitions, and the rules governing system behavior, which are fundamental to any computational model of learning.

2. Stochastic Processes: Markov Chains (MCs): The report then explores sys tems with probabilistic state transitions that adhere to the Markov property. Un derstanding MCs is vital as they form the basis for modeling environments where outcomes are uncertain but dependably linked to the current state.

3. Introduction to Reinforcement Learning: K-Armed Bandits: Before tack ling full state-based RL, this section examines the K-Armed Bandit problem. It serves as a simplified context to explore the critical exploration-exploitation dilemma through the implementation and analysis of various heuristic algorithms.

4. The Core RL Framework: Markov Decision Processes (MDPs):Then the report delves into MDPs, detailing their components, the crucial roles of policies and value functions, the foundational Bellman equations, and the implementation of exact solution algorithms like Value Iteration and Policy Iteration. These methods are exemplified by their application to the classic Grid World problem and the

Gambler's Problem. Later on, Model free algorithms like MC Control, SARSA, and Q learning are also explored along with their applications.

5. Scaling Up: Deep Reinforcement Learning: Finally, we address the "curse of dimensionality" inherent in tabular methods by introducing function approximation. This culminates in the from-scratch implementation of a Deep Q-Network (DQN), complete with a custom neural network library, and its application to solve the high-dimensional, continuous-state 'LunarLander-v3' control problem.

This report aims to not only summarize the theoretical concepts and document the practical implementations [1], but also to candidly discuss the challenges encountered and articulate the key insights gained, providing a holistic view of the theory and practice of modern Reinforcement Learning.

2 Formal Foundations: Logic and Automata

Understanding computation, state representation, and transitions is fundamental to grasp ing how an RL agent perceives its environment and makes decisions. This section revisits core concepts from propositional logic and finite automata, which provide the building blocks for more complex models of sequential decision-making.

2.1 Propositional Logic

Propositional logic is the simplest form of logic and provides the basic tools for formal reasoning about statements that can be definitively determined as either true or false. It is foundational for fields like AI, database theory, and circuit design.

- **Syntax & Semantics:** The syntax defines well-formed formulas (WFFs) using propositional variables (e.g., P, Q, R, representing atomic statements like "It is rain ing") and logical connectives such as ¬ (negation, "not"), (conjunction, "and"), (disjunction, "or"), → (implication, "if...then..."), and (biconditional, "if and only if"). The semantics are given by truth assignments, which map variables to truth values True, False. The truth value of a complex formula is determined compositionally using truth tables for these connectives.

- **Normal Forms:** Any propositional formula can be transformed into equivalent standard or normal forms. These are particularly useful for simplifying formulas, checking equivalences, and for algorithms in automated reasoning (e.g., SAT solvers often require CNF).

  - Conjunctive Normal Form (CNF): A conjunction of one or more clauses, where each clause is a disjunction of literals (a variable or its negation). E.g., (P ¬Q) (¬P R).
  - Disjunctive Normal Form (DNF): A disjunction of one or more terms, where each term is a conjunction of literals. E.g., (P ¬Q) (¬P R).

- **Key Concepts:**
  - Tautology: A formula true for all truth assignments (e.g., P¬P), representing a universally valid logical truth.
  - Contradiction: A formula false for all truth assignments (e.g., P ¬P), rep resenting a logical impossibility.
  - Satisfiability (SAT): A formula is satisfiable if there exists at least one truth assignment making it true. The SAT problem, determining satisfiability, is a central NP-complete problem with wide applications.

2.2 Finite Automata (FA)

Finite Automata are mathematical models of computation that accept or reject strings of symbols based on a predefined set of rules and a finite number of states. They are instrumental in pattern recognition, lexical analysis, and modeling systems with discrete states and transitions.

- **Deterministic Finite Automaton (DFA):** A DFA processes input strings de terministically: for each state and input symbol, the next state is unique. Formally, a DFA M = (Q, , , q0, F) comprises:
  - Q: A finite set of states, representing the machine's memory.
  - : A finite input alphabet.
  - : Q × → Q: The transition function, mapping a state and input symbol to a next state.
  - q0 Q: The designated initial state.
  - F Q: A set of accept or final states.

  A string is accepted if, starting from q0 and following transitions dictated by for each symbol in the string, the DFA ends in a state qf F. DFAs recognize the class of regular languages.

- **Nondeterministic Finite Automaton (NFA):** NFAs generalize DFAs by al lowing multiple possible next states for a given state-input pair and by permitting

-transitions (state changes without consuming an input symbol). The transition function is : $Q \times (\ ) \rightarrow P(Q)$ (the power set of Q). An NFA accepts a string if *any* sequence of valid transitions leads to an accept state. While NFAs can be more concise for describing certain languages, they do not possess more computational power than DFAs.

- **Equivalence of DFA and NFA:** For every NFA, an equivalent DFA (accepting the same language) can be constructed using the "subset construction" algorithm. This DFA's states correspond to sets of NFA states. This equivalence is a cornerstone of automata theory.

- **Regular Expressions (Regex):** A powerful, algebraic way to describe patterns in strings, defining regular languages. They are built from alphabet symbols using operations like concatenation, union, and Kleene star.

- **Kleene's Theorem:** This fundamental theorem states that a language is regular if and only if it can be described by a regular expression, which also means it is accepted by some FA (DFA or NFA). It unifies these three formalisms for describing regular languages.

2.3 Relevance to Sequential Decision-Making

Automata theory, particularly its concepts of states and transitions, serves as a conceptual precursor to the more complex framework of MDPs used in RL.

- **State Representation:** States in FAs model the "memory" of past inputs neces sary to decide on future transitions or acceptance. This is analogous to states in RL, which aim to capture all relevant information from the environment's history to make optimal decisions (the Markov property).

- **Transition Dynamics:** Transitions in FAs are driven by input symbols. In MDPs, transitions are driven by agent actions and are often probabilistic, reflecting the stochastic nature of many real-world environments.

- **Goal Achievement:** The notion of an FA "accepting" a string by reaching a final state parallels the RL agent's goal of reaching desirable terminal states or achieving high cumulative rewards through a sequence of actions.

Both formalisms model sequential processes, providing a structured way to think about how systems evolve over time based on events or interventions.

3 Stochastic Processes: Markov Chains

Markov Chains (MCs) are stochastic models describing a sequence of possible events (states) where the probability of transitioning to any future state depends only on the current state, not on the sequence of events that preceded it—this is the defining Markov property. They are essential for modeling systems with inherent randomness and time evolution.

3.1 Definition and The Markov Property

A discrete-time Markov Chain (DTMC) is a sequence of random variables Xt : t = 0, 1, 2, . . . taking values in a countable state space S. Its core characteristic is the Markov Property:

$$\mathbb{P}(X_{t+1} = s_{t+1}|X_t = s_t, Xt1 = s_{t1}, ..., X_0 = s_0) = \mathbb{P}(X_{t+1} = s_{t+1}|X_t = s_t)$$

This "memorylessness" means the future is conditionally independent of the past given the present state. For a time-homogeneous DTMC, the one-step transition probabilities $P_{ij} = \mathbb{P}(X_{t+1} = s_j|X_t = s_i)$ are constant over time and form the transition probability matrix P. Each row of P sums to 1.

3.2 Classification of States

The long-term behavior of an MC is determined by the properties of its states:

- **Accessibility and Communication:** State s_j is accessible from s_i if $P_{ij}^{(n)} > 0$ for some $n \geq 0$. States s_i and s_j communicate if they are accessible from each other. Communication forms equivalence classes, partitioning S. An MC is irreducible if all states form a single communicating class.

- **Recurrence and Transience:** A state s_i is recurrent if, starting from s_i, the probability of eventually returning to s_i is 1; otherwise, it is transient. Recurrence means the process will visit the state infinitely often if the class is re-entered. Transience implies the process eventually leaves the state permanently. This is a class property.

- **Periodicity:** The period $d(s_i)$ of a state s_i is the greatest common divisor (GCD) of all $n \geq 1$ such that $P_{ii}^{(n)} > 0$. If $d(s_i) = 1$, the state is aperiodic. A state being aperiodic is crucial for convergence to a limiting distribution. This is also a class property.

- **Ergodic MC:** An MC is ergodic if it is irreducible, positive recurrent (all states are positive recurrent, meaning expected return time is finite), and aperiodic. Ergodic MCs have strong, predictable long-term behavior.

3.3 Long-Term Behavior and Stationary Distributions

For ergodic MCs, the distribution over states converges to a unique stationary distribu tion π, regardless of the initial state. This distribution satisfies $\pi P = \pi$ and $\sum_j \pi_j = 1$. The component π_j represents the long-run proportion of time the chain spends in state s_j. This predictability is vital for analyzing systems that run for extended periods.

3.4 Absorbing Markov Chains

An absorbing state s_i is one where $P_{ii} = 1$ (once entered, never left). An MC is absorbing if it has at least one absorbing state, and every non-absorbing (transient) state can reach an absorbing state. Analyzing absorbing MCs often involves calculating:

- The probability of being absorbed in a specific absorbing state.
- The expected number of steps until absorption.

These are found using the canonical form of $P = \begin{pmatrix} Q & R \\ 0 & I \end{pmatrix}$ and the fundamental matrix $N = (I - Q)^{-1}$. The matrix $B = NR$ gives absorption probabilities.

4 Introduction to Reinforcement Learning: K-Armed Bandits

The K-Armed Bandit problem serves as a simplified, yet insightful, introduction to rein forcement learning, isolating the critical challenge of balancing exploration of new options against exploitation of known good options.

4.1 Problem Definition

The agent is presented with K choices or "arms" (like slot machines). Each arm a, when selected (or "pulled"), yields a reward drawn from an unknown probability distribution specific to that arm. The agent's objective is to make a sequence of arm selections over time to maximize its total accumulated reward. The true expected reward of arm a is $q_*(a) = \mathbb{E}$[Reward from arm a]. Since $q_*(a)$ is unknown, the agent must estimate it, typically as $Q_t(a)$, the sample average of rewards received from arm a up to time t 1.

4.2 The Exploration-Exploitation Dilemma

At each step, the agent faces a dilemma:

- **Exploitation:** Choose the arm with the highest current estimated value $Q_t(a)$. This maximizes expected immediate reward based on current knowledge but risks missing out on a truly better arm whose value is currently underestimated.
- **Exploration:** Choose an arm that is not currently estimated to be the best. This may yield a lower immediate reward but provides more information, potentially improving future estimates and leading to higher overall reward.

Effective bandit algorithms must intelligently manage this trade-off to minimize long term "regret" (the difference between rewards obtained and rewards that could have been obtained by always picking the best arm).

4.3 Algorithms Implemented

Several algorithms were implemented to address this dilemma:

- **Greedy Algorithm:** Always selects $A_t = \arg\max_a Q_t(a)$. This purely exploits and can easily get stuck with suboptimal arms.
- **ϵ-Greedy Algorithm:** Exploits with probability $1 - \epsilon$ and explores (chooses a random arm) with probability ϵ. This ensures continued exploration. ϵ can be fixed or decay over time.
- **Upper-Confidence-Bound (UCB):** Selects $A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}\right]$. The second term is an "uncertainty bonus" that encourages exploration of less tried arms or arms whose estimates are less certain. $N_t(a)$ is the count of pulls for arm a.
- **Gradient Bandit Algorithm:** Learns preferences $H_t(a)$ for each arm. Actions are selected via a softmax distribution $\pi_t(a) \propto e^{H_t(a)}$. Preferences are updated using stochastic gradient ascent, $H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t))$, where $\bar{R}_t$ is a reward baseline.
- **Optimistic Initial Values:** Initializes all $Q_0(a)$ to a value higher than any possible true reward. This encourages systematic exploration of all arms initially, as pulling an arm tends to reduce its overestimated value.

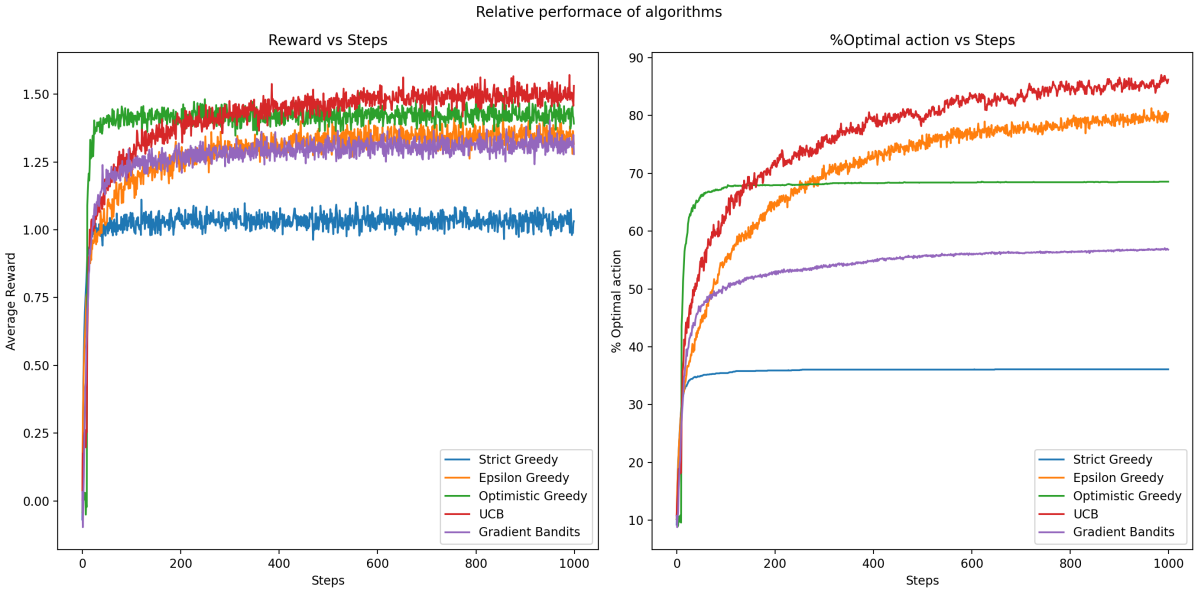

Figure 1: Various Bandit algorithms on the 10-armed testbed. Performance comparison showing average reward and optimal action percentage over steps.

4.4 Key Learnings from Bandit Problems

These implementations were crucial for understanding:

- The fundamental nature of the exploration-exploitation trade-off.
- How different algorithmic strategies (randomness, optimism, uncertainty quantifi cation, preference learning) address this trade-off.
- The sensitivity of algorithm performance to hyperparameters (e.g., ϵ, c, α).
- The bandit problem as a foundational stepping stone to more complex RL problems involving states, where similar value estimation and action selection strategies are employed.

5 The Core RL Framework: Markov Decision Processes

Reinforcement learning uses the formal framework of a Markov Decision Process(MDP, which is precisely defines below) to define the interaction between an *agent* and an *environment* in terms of states, actions and rewards. The agent is an entity who has an explicit goal. This is rigorously formulated in terms of the reward signal. The agent can sense certain aspects of it's environment, and they, along with it's own aspects, form the 'state' of an agent. The action may choose to take an 'action' in the environment to reach

closer to it's gaol which may result in transitioning to another state and a reward signal. Another important element of RL is a *model* of the environment. This is something that mimics the behaviour of the environment, or more generally, allows inferences to be made about how the environment will behave. Models are used for planning. Planning means deciding a particular action for a future state that the agent might encounter, but hasn't encountered yet.

5.1 MDP Components

An MDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$:

- $\mathcal{S}$: A finite set of states.
- $\mathcal{A}$: A finite set of actions.
- $\mathcal{P}(s'|s,a)$: The state transition probability function, $\mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a)$.
- $\mathcal{R}(s,a,s')$: The reward function, giving the immediate reward for the transition.
- $\gamma \in [0,1]$: The discount factor, valuing immediate rewards more than distant future rewards. It ensures bounded returns in infinite horizons and models preference for quicker gratification.

5.2 Policies and Value Functions

- **Policy $\pi(a|s)$:** The agent's strategy, $\mathbb{P}(A_t = a|S_t = s)$. A deterministic policy is a special case.
- **State-Value Function $V^\pi(s)$:** The expected discounted sum of future rewards (return) starting from state s and following policy π : $V^\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$. It quantifies the long-term desirability of state s under π.
- **Action-Value Function $Q^\pi(s,a)$:** The expected return starting from state s, taking action a, and then following policy π : $Q^\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$. It quantifies the desirability of taking action a in state s when following π. The relationship is $V^\pi(s) = \sum_a \pi(a|s) Q^\pi(s,a)$.

5.3 Bellman Equations

Bellman equations provide recursive decompositions of the value functions, forming the basis for most RL algorithms. They relate the value of a state (or state-action pair) to the values of its potential successor states.

5.3.1 Bellman Expectation Equation

For a given policy π, the value functions satisfy:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^{\pi}(s')] \tag{1}$$

$$Q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^{\pi}(s')] \tag{2}$$

Equation 1 states that the value of s is the expected immediate reward plus the dis counted expected value of the next state, averaged over actions taken by π and resulting transitions. Equation 2 has a similar interpretation for taking a specific action a.

5.3.2 Bellman Optimality Equation

For the optimal policy π^*, which maximizes value from all states, the optimal value functions V^* and Q^* satisfy:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^*(s')] \tag{3}$$

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a')] \tag{4}$$

Equation 3 implies $V^*(s)$ is achieved by picking the best action a in state s and then con tinuing optimally. The non-linearity due to the ‘max‘ operator means these are generally solved iteratively.

5.4 Solving MDPs: Exact Methods (Dynamic Programming)

When the MDP model ($\mathcal{P}$ and $\mathcal{R}$) is known, dynamic programming (DP) finds optimal policies.

5.4.1 Value Iteration (VI)

VI iteratively applies the Bellman optimality update (Equation 3) to an estimate $V_k(s)$, starting with an arbitrary V_0. Each iteration involves a "backup" operation for every state:

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V_k(s')]$$

The sequence V_k converges to V^* because the Bellman optimality operator is a γ-contraction mapping. Once V^* is approximated, the optimal policy is extracted greedily.The pseu-docode is shown in Algorithm 1.

Algorithm 1 Value Iteration

Initialize $V_0(s)$ arbitrarily for all $s \in \mathcal{S}$ (e.g., $V_0(s) = 0$), $k \leftarrow 0$
repeat
    $\Delta \leftarrow 0$
    for each state $s \in \mathcal{S}$ **do do**
        $v \leftarrow V_k(s)$
        $V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V_k(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V_{k+1}(s)|)$
    $V_k \leftarrow V_{k+1}$
    $k \leftarrow k + 1$
until $\Delta < \theta$ (convergence threshold)
return V_k (approximation of V^*)
Output policy $\pi^*(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^*(s')]$

5.4.2 Policy Iteration (PI)

PI alternates between two steps until the policy stabilizes:

1. **Policy Evaluation:** For the current policy π_k, compute V^{π_k} by solving the linear system of Bellman expectation equations (Equation 1). This is often done itera tively: $V_{j+1}^{\pi_k}(s) \leftarrow \sum_{s'} \mathcal{P}(s'|s,\pi_k(s))[\mathcal{R}(s,\pi_k(s),s') + \gamma V_j^{\pi_k}(s')]$ until convergence.

2. **Policy Improvement:** Improve π_k by acting greedily with respect to V^{π_k} : $\pi_{k+1}(s) = \arg\max_a Q^{\pi_k}(s,a) = \arg\max_a \sum_{s'} \mathcal{P}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^{\pi_k}(s')]$.

If $\pi_{k+1} \neq \pi_k$, then $V^{\pi_{k+1}}(s) \geq V^{\pi_k}(s)$ for all s (Policy Improvement Theorem). PI converges to π^* and V^*. The pseudocode is in Algorithm 2. While often converging in fewer iterations than VI, each PI iteration (policy evaluation) can be more costly.

Algorithm 2 Policy Iteration

Initialize policy $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$
repeat
    // 1. Policy Evaluation
    repeat
        $\Delta \leftarrow 0$
        for each state $s \in \mathcal{S}$ **do do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, \pi(s))[\mathcal{R}(s, \pi(s), s') + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta_{eval}$
    // 2. Policy Improvement
    policy_stable $\leftarrow$ true
    for each state $s \in \mathcal{S}$ **do do**
        old_action $\leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a)[\mathcal{R}(s, a, s') + \gamma V(s')]$
        if *old_action* $\neq \pi(s)$ **then**
            policy_stable $\leftarrow$ false
until *policy_stable*
return π and V

5.5 Applications of Dynamic Programming in MDPs

The theoretical DP methods find practical application in solving well-defined MDPs. Two classic examples were explored:

5.5.1 The Grid World Problem (Solved with Policy Iteration)

The Grid World is a standard testbed for illustrating MDP concepts and algorithms. It's a 2D grid where each cell is a state.

- **Setup:** Defined by grid dimensions, cell types (normal, obstacle, terminal goal, terminal penalty), available actions (typically Up, Down, Left, Right), transition probabilities (deterministic or stochastic, e.g., 80% chance of intended move, 10% chance to either perpendicular side), and rewards (e.g., small negative per step, large positive for goal, large negative for penalty).

- **Solution with Policy Iteration: Policy Iteration was implemented to find optimal policies.** This involved defining the MDP components for the Grid World and then applying the PI algorithm. The practical exercise of coding these, debugging transitions (es pecially at boundaries or near obstacles), and observing the propagation of values and convergence of policies was highly instructive. The resulting optimal policy can be visualised as a vector field guiding the agent. The

light blue squares denote the paths, the dark blue squares denote walls and the white square denotes the goal (Code at [1]).

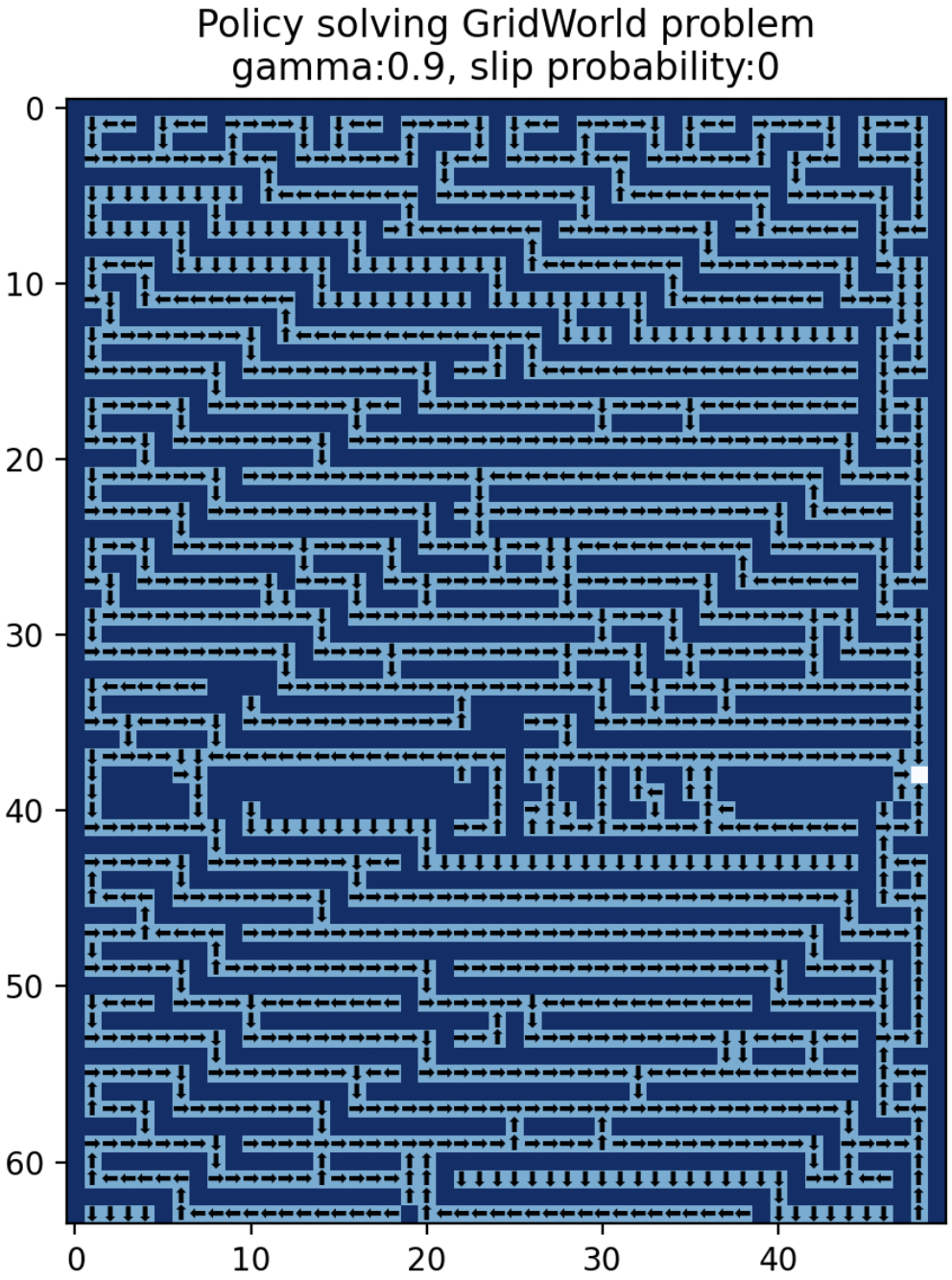

Figure 2: An optimal Grid World policy showing states, actions, and a goal, derived using Policy Iteration

5.5.2 The Gambler's Problem (Solved with Value Iteration)

Another classic MDP, described in Sutton and Barto [2], is the Gambler's Problem.

- **Setup:** A gambler has an initial capital and wants to reach a goal capital (e.g., \$100) by making a series of bets. On each bet, the gambler can stake an integer amount of money. With probability p_h the gambler wins and receives an amount equal to the stake; with probability $1 - p_h$, the gambler loses the stake. The game ends when the gambler reaches the goal capital or runs out of money (capital becomes \$0).

- **MDP Formulation:**

  - States $\mathcal{S}$: The gambler's capital, $\{0, 1, \ldots, \text{GoalCapital}\}$. States 0 and GoalCapital are terminal.
  - Actions $\mathcal{A}_s$: The possible stakes the gambler can bet in state s (current capital), e.g., $\{1, 2, \ldots, \min(s, \text{GoalCapital} - s)\}$.
  - Transition Probabilities $\mathcal{P}$: Determined by p_h. If stake a is bet in state s:
    - Transition to $s + a$ with probability p_h.
    - Transition to $s - a$ with probability $1 - p_h$.
  - Rewards $\mathcal{R}$: Typically, $+1$ upon reaching GoalCapital, and 0 for all other transitions (or small negative cost per bet).

- **Solution with Value Iteration: Value Iteration was implemented to determine the optimal betting policy.** This involves finding the optimal value function $V^*(s)$ (maximum probability of reaching the goal from capital s) and the corresponding optimal stake $\pi^*(s)$ for each capital level. The results show interesting patterns in betting strategy depending on p_h and proximity to the goal or ruin.

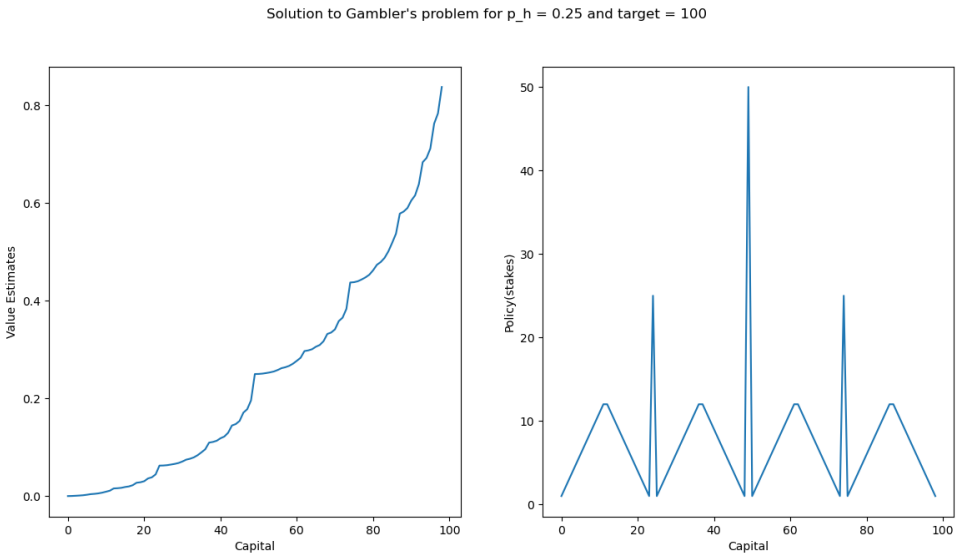

Figure 3: Optimal policy and value function for the Gambler's Problem (e.g., target=\$100, p_h=0.25).

5.6 Model-Free Methods

In most realistic scenarios, the model is unknown. Model-free methods learn optimal behavior directly from experience gathered through interaction.
We first differentiate between on-policy and off-policy methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data.

5.6.1 Monte Carlo (MC) Methods

MC methods learn by averaging the returns from complete episodes. For each state-action pair encountered in an episode, the total discounted return G_t os calculated and averaged into the running estimate of $Q(s, a)$. However, a critical issue arises in MC methods, that of exploration. The fact that we can only learn at the end of an episode, and we take actions in that episode as per our current value estimates, means that not all possible state action pairs might be explored. This is solved by two methods. In some problems, we can guarantee that the starting states in all episodes will be different(or more accurately, all the states are considered starting states at least once, as the number of episodes goes to infinity). The other method is to use ϵ-soft policies, which ensure that each state action pair is encountered at least once as the number of episodes tends to infinity, The ϵ-greedy policy is one of the most simple ϵ-soft policies to ensure sufficient exploration of all state-action pairs. The on-policy first visit MC control pseudocode is shown in algorithm 2

Algorithm 3 On-policy first-visit MC control (for ϵ-soft policies)

Algorithm parameters: small $\epsilon > 0$, discount factor $\gamma \in [0, 1]$

Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
  $Q(s, a) \in \mathbb{R}$ (arbitrarily, e.g., $Q(s, a) = 0$)
  $Returns(s, a) \leftarrow$ an empty list
  $\pi(a|s) \leftarrow$ an arbitrary ϵ-soft policy

loop forever (for each episode)
  Generate an episode following policy π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  // Loop for each step of the episode, processing backwards from $t = T - 1$
  for $t = T - 1$ **down to** 0 **do**
    $G \leftarrow \gamma G + R_{t+1}$
    // If the pair (S_t, A_t) is the first visit in the episode
    if the pair (S_t, A_t) does not appear in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$ **then**
      // Policy Evaluation: Update Q-value by averaging returns
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

      // Policy Improvement: Make the policy ϵ-greedy w.r.t. new Q-value
      $A^* \leftarrow \arg\max_a Q(S_t, a)$ (breaking ties randomly)
      for all $a \in \mathcal{A}(S_t)$ **do**

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

5.6.2 Temporal-Difference (TD) Learning

TD learning combines the bootstrapping of DP with the experience-based learning of MC. It can learn from every step, making it more sample-efficient.

- **SARSA (On-Policy):** This algorithm learns the value of the policy the agent is currently following (the behavior policy). Its update rule uses the five-element tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$:

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

  Because A_{t+1} is chosen by the same exploratory policy, SARSA learns a "safer" or more conservative policy. The pseudocode is given in algorithm 4

Algorithm 4 Sarsa: On-policy TD Control

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$, discount factor $\gamma \in [0, 1]$

Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$.

loop forever (for each episode)
    Initialize S (first state of episode)
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)

    loop for each step of episode
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)

        // Update the Q-value for the state-action pair we just left
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$

        // Move to the next state and action
        $S \leftarrow S'$; $A \leftarrow A'$
        S is terminal

- **Q-Learning (Off-Policy):** This algorithm directly approximates the optimal action-value function, Q^*, independent of the policy being followed. The key difference is that the update target always uses the greedy action:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

  This allows Q-Learning to learn about the optimal path even while taking exploratory actions, which often leads to faster convergence. The pseudocode is given in algorithm 5.

  Q learning suffers from a *maximisation bias*. Since we take always pick the action with maximum Q value, this can lead to over-valuing many of the state-action pairs. A different algorithm, called Double Q learning solves this problem. It uses a separate action-value function, Q' instead of taking the action greedily. This seperate action-value function is updated periodically to keep up with the main function. This reduces maximisation bias.

Algorithm 5 Q-learning: Off-policy TD Control

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$, discount factor $\gamma \in [0, 1]$

Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$.

loop forever (for each episode)
    Initialize S (first state of episode)

    loop for each step of episode
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'

        // Update Q-value using the greedy action from the next state
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

        // Move to the next state
        $S \leftarrow S'$
        S is terminal

5.7 Applications of Q learning and SARSA

Both algorithms were successfully applied to solve Gymnasium's 'CliffWalking-v1'. CliffWalking is an environemnt where the agents goal is to move from start to goal, without falling into the cliff. This is a standard, undiscounted, episodic task, with start and goal states, and the usual actions of a grid world.

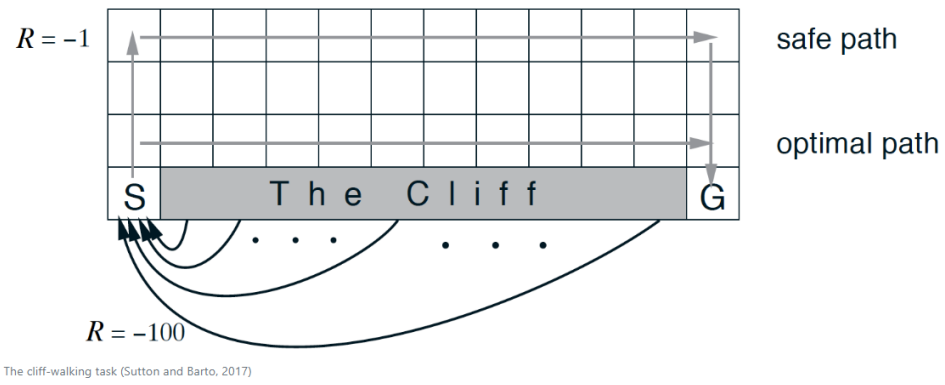

Figure 4: The cliff world environment

Q-learning found the optimal but risky path along the cliff edge, while SARSA found a safer, longer path. It can be seen that Q learning achieves the results faster, but has more variance. However, this variance is due to the exploratory nature of the off policy Q learning. The main policy does know that it should avoid the cliff, and in fact, would outperform SARSA for this particular example. This point is critical

to understand when choosing the correct algorithm. For example, in a "variable" environment case(where the environment might change as time progressing, in the cliffworld say if a piece of rock broke from the cliff) SARSA might be better because of it's inherent exploratory nature. Q learning might learn the better policy however it wont be able to adjust it's q values on the go.

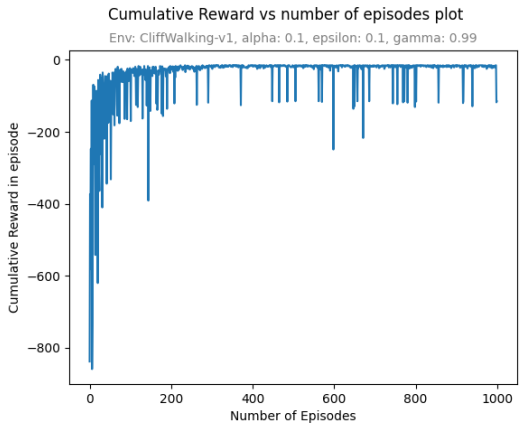

Figure 5: Learning curve for the SARSA agent on 'CliffWalking-v1'. The agent learns to avoid the cliff, converging to a stable, negative reward characteristic of the longer, safer path.

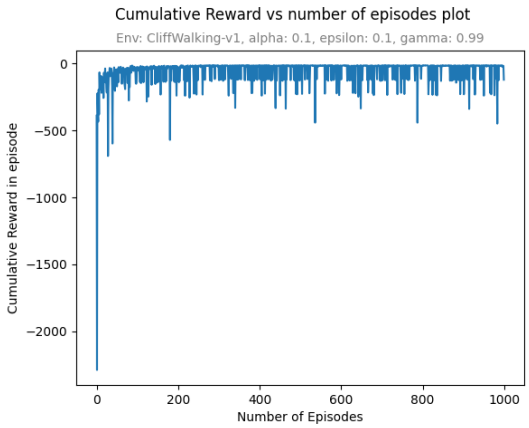

Figure 6: Learning curve for the Q agent agent on 'CliffWalking-v1'. The agent learns the shortest path is right alongside the cliff, even though it may be risky to do so because of exploration in the exploratary(off) policy of Q learning

6 Scaling Up with Function Approximation

The algorithms discussed thus far, from Dynamic Programming to tabular TD-learning, rely on a crucial assumption: the state-action value function, $Q(s, a)$, can be stored in a lookup table. This approach is only feasible for problems with a small, discrete number of states and actions. For environments with large discrete state spaces or, more critically, continuous state spaces, it is impossible to store a distinct value for every possible state. This challenge, often called the "curse of dimensionality," necessitates a more scalable approach: **function approximation**.

The core idea is to generalize from seen states to unseen states by representing the value function with a parameterized function, $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ or $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$, where $\mathbf{w}$ is a weight vector of much lower dimensionality than the number of states. Learning then becomes a task of finding the optimal weights $\mathbf{w}$ using techniques from supervised learning, typically by minimizing the Mean-Squared Error (MSE) between the current estimate and a target value.

6.1 Linear Function Approximation

The simplest form of function approximation uses a linear combination of features extracted from the state. The value function is modeled as:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{n} w_i x_i(s)$$

where $\mathbf{x}(s)$ is a feature vector representing state s. Learning is achieved via stochastic gradient descent (SGD) on the weights:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha(U_t - \hat{v}(S_t, \mathbf{w}_t))\nabla \hat{v}(S_t, \mathbf{w}_t)$$

where U_t is the update target (e.g., the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$) and $\nabla \hat{v}(S_t, \mathbf{w}_t) = \mathbf{x}(S_t)$. The key to success with this method lies in the design of the feature vector $\mathbf{x}(s)$.

- **Coarse Coding:** This technique represents a continuous state space using a set of overlapping receptive fields (often circles or squares). A state is represented by a binary feature vector where each component is 1 if the state falls within the corresponding receptive field, and 0 otherwise. The overlapping nature allows for smooth generalization, as nearby states will activate a similar set of features.
- **Tile Coding:** A highly effective and widely used form of coarse coding, especially for multi-dimensional state spaces. The state space is partitioned multiple times with grids, or "tilings," each offset from the others. A state is represented by the set of tiles it falls into, one from each tiling. This creates a binary feature vector with one '1' for each tiling. By using multiple offset tilings, tile coding achieves fine-grained discrimination while maintaining broad generalization, and it is computationally efficient.

6.2 Non-Linear Function Approximation with Neural Networks

While linear methods are powerful, their effectiveness is fundamentally limited by the quality of the hand-crafted features. For highly complex environments, designing an optimal feature vector $\mathbf{x}(s)$ can be as challenging as solving the RL problem itself. This is where non-linear function approximators, particularly deep neural networks, offer a revolutionary advantage.

A neural network can be viewed as a composition of multiple layers of linear transformations followed by non-linear activation functions (e.g., tanh, ReLU, Sigmoid). This structure allows it to learn a complex, hierarchical representation of the input data automatically. In the context of RL, this means the network can learn the relevant features

directly from the raw state representation (like sensor readings or pixel data), alleviating the need for manual feature engineering.

However, applying neural networks to reinforcement learning presents a unique set of challenges not typically found in supervised learning:

- **Non-Stationary Targets:** The target value for the network's prediction (e.g., the TD target $r + \gamma \max_{a'} Q(s', a'; \mathbf{w})$) is non-stationary because the policy and value estimates are constantly changing as the agent learns. The network is essentially "chasing a moving target."
- **Correlated Data:** The data samples (s_t, a_t, r_t, s_{t+1}) gathered from an episode are highly correlated. The state at time $t+1$ is heavily dependent on the state at time t. This violates the i.i.d. (independent and identically distributed) assumption that underpins many stochastic gradient descent optimization algorithms, leading to inefficient and unstable training.

The **Deep Q-Network (DQN)** algorithm, introduced by Mnih et al. [3], was a landmark achievement because it introduced two key techniques to successfully overcome these stability issues, making deep learning a viable tool for reinforcement learning.

7 Final Project: A Deep Q-Network for Lunar Lander

The culmination of this project was to synthesize the principles of TD-learning and function approximation to tackle a challenging, high-dimensional control problem. I implemented a Deep Q-Network (DQN) agent entirely from scratch, including a custom neural network library, and applied it to solve the 'LunarLander-v3' environment from the Gymnasium suite.

7.1 The Lunar Lander Environment

'LunarLander-v3' is a classic RL benchmark that simulates landing a spacecraft in a 2D world. It presents several challenges that make it a suitable testbed for deep RL:

- **Continuous State Space:** The state is an 8-dimensional continuous vector representing the lander's position, velocity, angle, angular velocity, and whether its legs are in contact with the ground. This high-dimensional, continuous space makes tabular methods completely infeasible.
- **Sparse and Shaped Rewards:** The agent receives small negative rewards for fuel consumption and crashing, and a large positive reward for landing softly and

correctly between the flags. This reward structure requires the agent to learn a long-term strategy.

- **Complex Dynamics:** The physics-based environment requires the agent to learn a nuanced control policy, balancing thruster firings to counteract gravity and control its descent.

7.2 From-Scratch Implementation Details

To gain a first-principles understanding, the entire agent and its components were built using only ‘numpy‘.

7.2.1 Custom Neural Network Library

The core of the agent is a modular neural network. We created distinct classes for each component:

- **Layer:** A class to handle a single dense layer, storing its own weight matrix (initialized with a standard normal distribution) and bias vector. It contains the logic for the forward matrix multiplication.
- **Activation Functions (ReLU, Linear):** Implemented as separate classes, each with a ‘forward‘ method to apply the non-linearity and a ‘backward‘ method to compute the gradient with respect to its input. This design allows for easy extension to other activation types.
- **NeuralNetwork:** A container class that manages the sequence of layers and activations. Its ‘forward‘ method propagates data through the entire network, and its ‘backward‘ method chains the gradients from the output layer back to the input layer using the chain rule, updating the weights and biases of each layer via stochastic gradient descent.

7.2.2 DQN Agent Architecture

The ‘DQN_agent‘ class integrates the neural network with the key algorithmic components for stable learning:

1. **Experience Replay:** A ‘ReplayBuffer’ was implemented using Python’s ‘collections.deque’ with a capacity of 50,000 transitions. At each timestep, the agent stores the experience tuple $(s_t, a_t, r_{t+1}, s_{t+1}, \text{done})$. For learning, mini-batches of 64 transitions are sampled uniformly from this buffer. This crucial step de-correlates the training data, stabilizing the learning process.

2. **Fixed Target Network:** The agent maintains two neural networks with identical architectures: an *online network* ($Q(s, a; \boldsymbol{\theta})$) and a *target network* ($\hat{Q}(s, a; \boldsymbol{\theta}^-)$). The online network is updated at every learning step. The target network, which is used to generate the TD target for the loss function, has its weights frozen. These weights are only updated periodically (every 1,000 steps in my implementation) by copying the weights from the online network. This provides a stable, consistent target for the online network to learn towards, preventing the destructive oscillations that can occur otherwise.

3. **The Learning Step:** The 'learn' method orchestrates the update. It samples a mini-batch, computes the TD target using the target network ($y_i = r_i + \gamma \max_{a'} \hat{Q}(s'_i, a'; \boldsymbol{\theta}^-)$), calculates the MSE loss against the online network's predictions, and performs back-propagation to update the online network's weights $\boldsymbol{\theta}$.

7.3 Experimental Analysis and Results

Finding a set of hyperparameters that enables successful learning is a critical and empirical part of deep RL. We conducted systematic experiments to tune the agent.

7.3.1 Hyperparameter Tuning

The Learning Rate

The learning rate (α) proved to be the most sensitive hyperparameter. A sweep of values was performed, and the results are shown in Figure 7.

- **High Learning Rates (e.g., 0.01):** Led to extreme instability and divergence. The weight updates were too large, causing the network to overshoot optimal policies and catastrophically forget any learned behaviors.

- **Low Learning Rates (e.g., 0.0001):** Resulted in very stable but slow learning. The agent showed steady progress but converged slowly, taking 3000 episodes for good results.

- **Optimal Learning Rate (around 0.001):** This "Goldilocks" zone provided the best trade-off between learning speed and stability. A rate of 0.01 was ultimately chosen for its slightly more stable convergence to a higher peak performance.

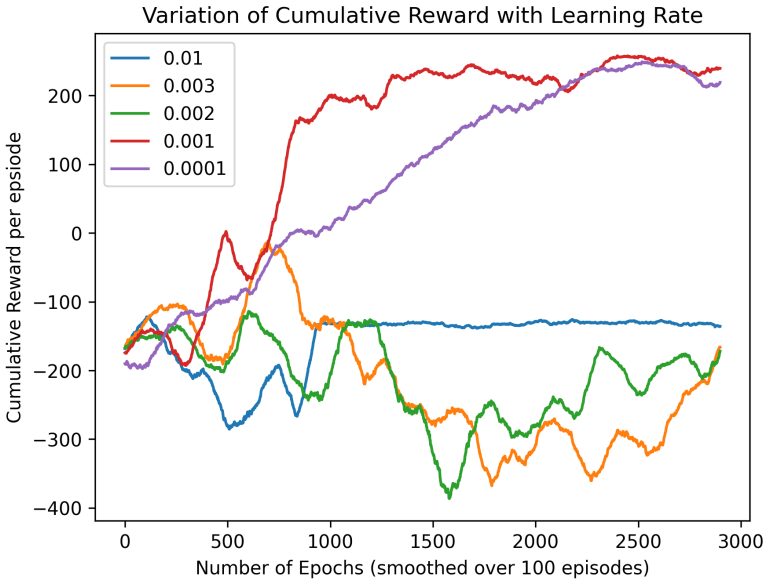

Figure 7: Hyperparameter sweep for learning rate on 'LunarLander-v3'

The Discount Factor

The hyperparameter sweep of gamma, the discount also provides interesting observations. It can be seen that when gamma is relatively less, between 0.9 and 0.99, the agent learns poorely. This is because an episode in Lunar Lander may consist of thousands of steps, and if gamma is less then the agent isn't considering future repercussions of it's actions good enough. For example, an agent might want to go near the goal, however there could be a high chance that a cliff or similar object is in that path. In such scenarious, the agent has to forsake short-term rewards for the long-term ones. Generally, in enviroments with a high number of steps per episodes, a high gamma(greater than 0.95) is preferred.

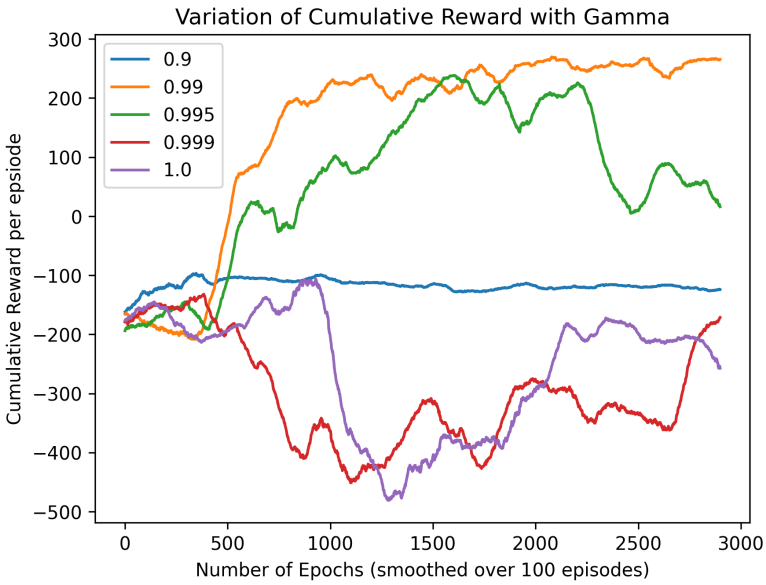

Figure 8: Hyperparameter sweep for discount factor on 'LunarLander-v3'

The Batch Size

The batch size refers to the number of episodes that are used to improve the model in a single pass. Smaller batches lead to more frequent updates but can be noisy and potentially unstable, while larger batches provide more stable gradients but might converge to sharp, less generalizable minima. Moreover, higher batch sizes require more computing memory to process the data. This is also an example of the classic computing power vs memory tradeoff encountered in programming.

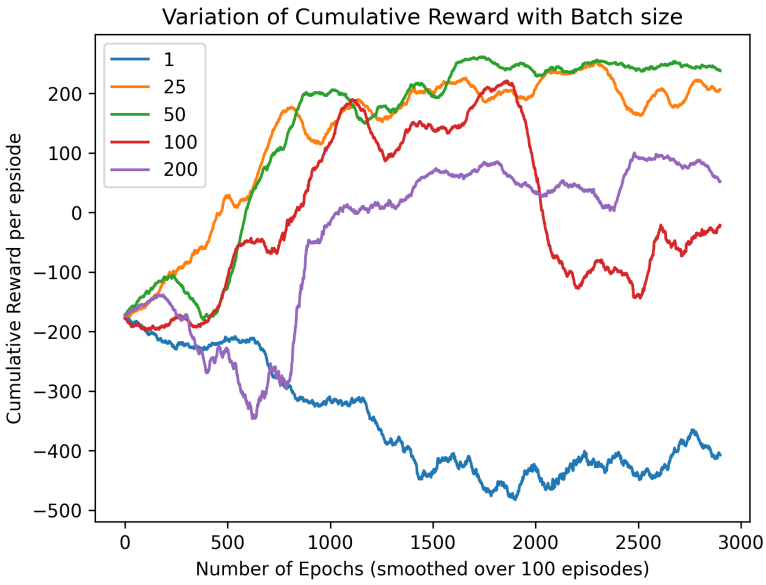

Figure 9: Hyperparameter sweep for batch size on ‘LunarLander-v3‘

The Target Update Frequency

The target update frequency referres to the number of steps the target network is ”behind” the online main network. A less frequent target network update generally leads to more stable learning but may require more training data due to the delayed feedback. Conversely, frequent updates, especially hard updates (replacing the target network entirely), can cause oscillations and instability. Finding the right balance is crucial for optimal performance.

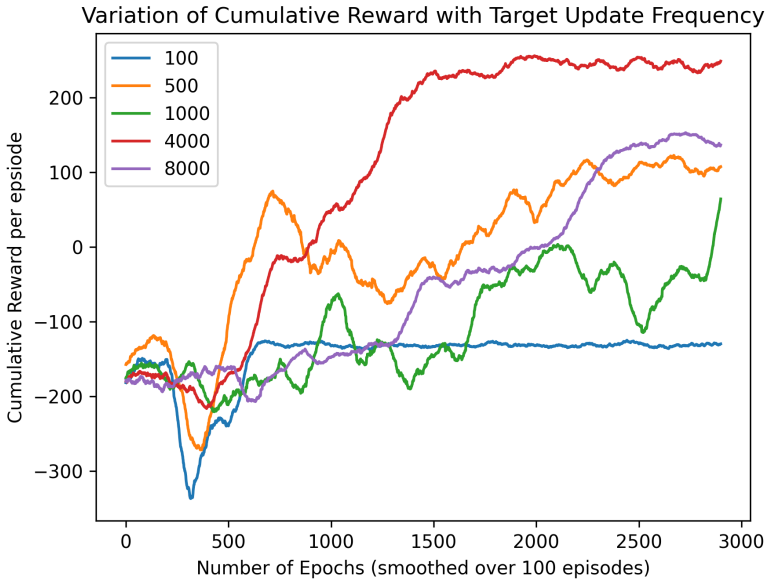

Figure 10: Hyperparameter sweep for target update frequency on ‘LunarLander-v3‘

8 Discussion and Conclusion

8.1 Synthesis of Learnings

This project demonstrated a clear, logical progression of concepts in Reinforcement Learning. The exploration strategies first analyzed in the simple, stateless bandit problems provided direct intuition for designing the ϵ-greedy policy in the complex DQN agent. The Bellman equations, which were solved analytically with DP for known models, formed the conceptual core of the loss function for every subsequent model-free algorithm, from the tabular updates in Q-learning to the gradient-based updates in DQN.

The transition from model-based to model-free learning highlighted the practical necessity of learning directly from interaction when a perfect model is unavailable. The subsequent leap from tabular methods to deep function approximation showed how RL can be scaled to solve problems with high-dimensional, continuous state spaces that were previously intractable. This journey from theory to practice revealed that each advanced algorithm is built firmly upon the principles established by its predecessors.

8.2 Challenges and Insights

The project was not without its challenges, each of which provided valuable learning opportunities.

- **Conceptual Hurdles:** During the first few weeks, we had to go throught the fundamentals of Logic and Automata theory which provided a rigorous mathematical foundation. Reading the texts of Baier and Katoen was a very daunting tasks.

Fortunately, as the project continued, the provided material was far more intuitive to understand, especially Sutton-Barto. The distinction between on-policy (SARSA) and off-policy (Q-Learning) learning was subtle, and the intuition was hard to develop. Implementing them and observing their different behaviors on 'CliffWalking-v1'—where SARSA learns a "safer" path and Q-Learning learns the "optimal" but riskier path—solidified this crucial concept.

- **Implementation Details:** Debugging iterative algorithms like DP required careful attention to loop termination conditions and state-space management. For the DQN, implementing backpropagation correctly through the custom neural network layers was non-trivial but provided an invaluable, deep understanding of how deep learning models are trained.

- **Hyperparameter Sensitivity:** The greatest practical challenge was training the DQN. The process was extremely sensitive to hyperparameters. Initial experiments with a high learning rate or a fast epsilon decay resulted in catastrophic forgetting or policy divergence. This underscored that successful deep RL is as much an empirical science of careful tuning and experimentation as it is a theoretical one. The success of the final agent was a direct result of systematic and patient hyperparameter sweeps. It took a lot of computing time to fully explore the impact of hyperparameters on the final policy and valuye function

8.3 Broader Horizons and Future Directions

This project has been an incredible learning journey. Unlike some other projects I took, where we simply implement algorithms in code, this project help us garner an in-depth understanding of the algorithms and paradigms themselves. The subtle differences between on-policy and off-policy methods, the workings of the "deadly triad" and other aspects of the project greatly clarified my understanding of RL.

While this project focused on a foundational path from dynamic programming to value-based deep RL, the field of reinforcement learning is very vast. The principles learned here provide the necessary background to explore other major paradigms of RL, two of which are particularly noteworthy: the integration of planning and learning, and the shift from value-based to policy-based methods. I studied both of these however could not find the time to implement these algorithms.

8.3.1 Bridging Model-Based and Model-Free Learning: Dyna-Q

A key theme in this report was the distinction between model-based methods (like Dynamic Programming), which are highly sample-efficient but require a model, and model-free methods (like Q-Learning), which are more generally applicable but often require

extensive interaction with the environment. The Dyna-Q architecture, offers an elegant bridge between these two extremes.

Dyna-Q operates on a simple but powerful principle: after each real interaction with the environment, the agent uses the experience (s, a, r, s') to not only update its value function (a model-free "direct RL" update) but also to update an internal, learned model of the environment, $\hat{\mathcal{P}}(s'|s, a)$ and $\hat{\mathcal{R}}(s, a)$. The agent then uses this learned model to perform several "hypothetical" or "simulated" updates. It can sample state-action pairs it has seen before, query its learned model to get a predicted next state and reward, and use this simulated experience to perform additional Q-learning updates.

This allows the agent to "plan" and propagate value information more efficiently without requiring more real-world samples. Implementing Dyna-Q would be a natural next step, exploring how even a simple learned model can dramatically accelerate learning compared to a purely model-free approach.

8.3.2 Directly Learning the Policy: REINFORCE and Policy Gradients

All the algorithms implemented in this project are value-based. They first learn a value function (Q^*) and then derive an implicit policy by acting greedily with respect to that function. An alternative and powerful class of algorithms directly parameterizes the policy itself, $\pi(a|s, \boldsymbol{\theta})$, and learns the optimal weights $\boldsymbol{\theta}$ without an intermediate value function.

The simplest and most fundamental of these policy-gradient methods is REINFORCE. The core idea is to adjust the policy parameters via gradient ascent to maximize the expected return. The REINFORCE update rule is remarkably intuitive:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta}_t)$$

This update pushes the policy in a direction that makes actions leading to high returns (G_t) more likely, and actions leading to low returns less likely. The term $\nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta}_t)$ is the "eligibility vector," which indicates the direction in parameter space that would most increase the probability of taking action A_t in state S_t.

Unlike value-based methods, policy-gradient methods can naturally handle continuous action spaces and can learn stochastic policies, which are sometimes optimal. A future project could implement REINFORCE and its more advanced successors, like Actor-Critic methods (e.g., A2C), which combine the benefits of both policy-based and value-based learning. This would complete the survey of the three main pillars of modern RL: value-based, policy-based, and the integration of planning.

References

[1] Tanay Jha. *Markov-Decision-Processes-and-their-applications-in-AI*. GitHub Repository. https://github.com/Tanay2104/Markov-Decision-Processes-and-their-applications-in-AI

[2] R. S. Sutton, & A. G. Barto (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

[3] V. Mnih, et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529-533.

[4] J. Towers, et al. (2023). *Gymnasium: An API for Reinforcement Learning*.