Endterm Report: Markov Decision Processes and their Applications in Artificial Intelligence

Summer of Science 2025

Aadeshveer Singh
Roll No: 24B0926
Computer Science & Engineering
IIT Bombay
Mentor: Harshita Inampudi

July 20, 2025

Abstract

This report details the progress and final outcomes of the Summer of Science 2025 project focusing on Markov Decision Processes (MDPs) and their applications in Artificial Intelligence. It covers foundational concepts from logic and automata theory, and stochastic processes like Markov Chains. The report then dives deep into the theory and implementation of algorithms for solving MDPs, beginning with Multi-Armed Bandits and Dynamic Programming techniques (Policy Iteration and Value Iteration) applied to custom GridWorld environments and classic problems like Jack's Car Rental and the Gambler's Problem. The work further extends into model-free learning, with successful implementations of Monte Carlo methods for prediction and control in Blackjack and the Racetrack problem. Finally, the project explores Deep Reinforcement Learning, culminating in the successful training of a Deep Q-Network (DQN) agent for Flappy Bird. The report outlines the theoretical understanding gained, the full scope of practical implementations developed, key challenges encountered, and the significant learnings acquired.

Keywords: Reinforcement Learning, Markov Decision Processes, Dynamic Programming, Policy Iteration, Value Iteration, Monte Carlo Methods, Deep Q-Learning (DQN), Multi-Armed Bandits.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Personal Journey

Like everyone, I was introduced to AI through the hype around it being built up for years. Being new to Computer science, I wanted to explore everything, and that was when I got to RL. During the WiDS (Winter in Data Science) after the first semester, I learned what exactly RL was in my "AI snake game" project. Although it was not a theoretically dense program focussing more on learning to code by following tutorials, it sparked my interest. I had tasted the blood, and I needed more. Time passed, and I tried to keep using the same tutorial method. Unthinkingly writing code to do simple stuff. Then came this summer at the end of the first semester, and with that came my first opportunity to take up an SoS project. I had had enough of following someone writing code for me, and It was time for me to learn what I wanted properly. Thus, I began my theoretical journey into RL and MDPs, building my understanding from basic CS logic to MDP algorithms week by week.

1.2 Overview of Reinforcement Learning

> The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning.
>
> — Sutton and Barto [1]

This opening thought from Sutton and Barto's seminal work on reinforcement learning resonates deeply with our intuition about how natural learning occurs. Indeed, when an animal learns, it typically does so through interaction and consequence, without an explicit teacher. Nature has endowed us with an intrinsic learning mechanism: actions taken in an environment often elicit feedback, perhaps akin to hormonal responses in biological systems, signaling the 'goodness' of the outcome. This natural feedback loop is a biological counterpart to the 'reward' signal central to RL.

This concept echoes Alan Turing's proposition in his paper "Computing Machinery and Intelligence", where he suggested that building a "child machine" capable of learning like a human might be a more feasible path to artificial intelligence than attempting to construct a fully formed "adult-like" machine. Reinforcement Learning (RL), a prominent branch of Machine Learning (ML), beautifully formalizes this interactive learning paradigm.

In RL, an **agent** (the learner or decision-maker) interacts with a simulated or real **environment**. The agent takes an **action**, and as a consequence, it perceives **observations** (or transitions to new **states**) from the environment. Following this, the agent receives a scalar **reward** signal, which indicates the immediate desirability of the action taken or the state reached. The agent's objective is to learn a **policy**—a strategy dictating its actions based on its observations—that maximizes the *cumulative reward* it receives over the long run. Typically, the agent begins with a random (or specifically initialized) policy. Through continuous interaction, it evaluates the effectiveness of its current policy and iteratively refines it to achieve its environmental goals more effectively. After sufficient learning, the agent can develop a highly effective policy.

Thus, the core pursuit of RL is to develop computational methods that mimic this natural, trial-and-error learning process, enabling an agent to autonomously improve its behavior and decision-making capabilities within an environment.

1.3 Project Objectives and Scope (First Half)

The first four weeks of this project laid the theoretical groundwork for understanding Reinforcement Learning, progressing through key concepts:

- **Week 1**: Foundational Logic and Automata. This week focused on the syntax and semantics of propositional and predicate logic, which was followed by an emphasis on finite automata as an introduction to state-based systems.
- **Week 2**: Markov Chains. Building upon formal methods, this week delved into the definition, properties, and dynamics of Markov Chains, establishing the foundation for MDPs.
- **Week 3**: MDP Introduction and k-Armed Bandits. This period involved understanding the application of Markov Chains within the context of problem-solving algorithms and marked the beginning of core RL study with the practical exploration of the K-Armed Bandit problem.
- **Week 4**: Algorithm Implementation and Reporting. This culminating week of the first phase focused on translating learned MDP algorithms (such as Policy Iteration and Value Iteration) into code through various projects, alongside compiling this midterm report.

1.4 Report Structure

Chapter 2 entails a summary of my theoretical journey during the project. It is divided into three sections, with section 1 being Logic and Automata Theory. Section 2.2 introduces Markov chains through stochastic processes—finally, Section 2.3 moves on to MDPs and their formal definitions.
Chapter 3 moves on to the first practical aspect of the project. It goes through the theoretical introduction of RL via K-armed Bandit and its implementation.
Chapter 4 involves the bulk of practical work in the project. It begins with an introduction to dynamic programming and its algorithms(Policy Iteration and Value Iteration). Then come 3 case studies: Gridworld(4.4), Jack's Car Rental(4.5) and Gambler's Problem(4.6).

Chapter 7 ends the report with my key challenges and learnings from the project and my future path.

Chapter 2

Theoretical Foundations

2.1 Logic and Automata Theory

Logic is an essential pillar across all Computer Science disciplines, and this foundational role extends directly to Reinforcement Learning (RL). Correctly understanding RL relies on the theoretically rigorous modelling of sequential decision-making problems, for which the Markov Decision Process (MDP) provides a powerful and convenient framework. Therefore, a complete understanding of basic Logic and Automata Theory becomes a essential prerequisite for comprehending the structure and dynamics of MDPs, as these concepts provide methods to describe formally and reason about states, transitions, and system properties.
The primary resources consulted for this section include "Principles of Model Checking" by Baier and Katoen [2], and "Logic in Computer Science" by Huth and Ryan [3].

2.1.1 Propositional Logic

Propositional logic forms the bedrock of logical reasoning by providing a formal language for representing statements that can be either true or false. Key concepts explored include:

- **Syntax and Semantics**: Understanding the well-formed formulas (propositions, connectives like $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$) and how their truth values are determined by truth assignments to atomic propositions using truth tables.

  $\neg$ The **negation** of p is given by $\neg p$

  $\vee$ Representation of **either** p **or** q is given by $p \vee q$.

  $\wedge$ Representation of p **and** q **both** being `True` is given by $p \wedge q$.

  $\rightarrow$ q being the logical consequence of p being true, given by $p \rightarrow q$. In other words **if** p **then** q. It is equivalent to $\neg p \vee q$.

  $\leftrightarrow$ Double implication of p **implies** q **and** q **implies** p, is given by $p \leftrightarrow q$.

- **Logical Equivalence:** Identifying when different formulas have the same truth value under all interpretations (e.g., De Morgan's laws). Some examples

$$p \to q \vdash \neg p \lor q \tag{2.1}$$

$$\neg(p \lor q) \vdash \neg p \land \neg q \qquad \text{De Morgan's law} \tag{2.2}$$

$$\neg(p \land q) \vdash \neg p \lor \neg q \qquad \text{De Morgan's law} \tag{2.3}$$

$$\neg\neg p \vdash p \tag{2.4}$$

- **Satisfiability, Tautologies, Contradictions:** A formula is satisfiable if there's at least one truth assignment making it true; a tautology is true under all assignments; a contradiction is false under all assignments.
  Thus ψ is **satisfiable** iff:

  $$\Phi \vdash \psi \qquad \text{Where,} \Phi = \{\phi_1, \phi_2 \ldots \phi_n\}$$

  For some specific values of atomic propositions $\phi_i \, \forall \, 1 \leq i \leq n$, in other words ψ evaluates to `T` for some valuation.
  A **tautology** can be induced without any premise, as $\top$ is a tautology:

  $$\vdash \top$$

  A **contradiction** is not `True` under any premise. If assumed true can be used to prove anything.

  $$\Phi \nvdash \bot$$

  $$\bot \vdash \psi$$

  Where $\Phi = \{\phi_1, \phi_2 \ldots \phi_n\}$ can be any set of formulas and ψ can be any formula.

- **Logical Consequence (Entailment):** The concept of $\Gamma \vDash \phi$ (semantic entailment), meaning ϕ is true in all models where all formulas in Γ are true.
  It can be easily checked by building a Truth table. Example:

  $$p \to q \vDash \neg p \lor q$$

p	q	$\neg p$	$p \to q$	$\neg p \lor q$
F	F	T	T	T
T	F	F	F	F
F	T	T	T	T
T	T	F	T	T

- **Proof Systems (Natural Deduction):** H&R introduces formal proof systems like Natural Deduction, which provide rules for deriving conclusions from premises syntactically ($\Gamma \vdash \phi$). The goal of such systems is to match semantic entailment. The rules can be found in Appendix A.

- **Soundness and Completeness:** Conceptually, a proof system is sound if everything provable is semantically true ($\vdash$ implies $\models$), and complete if every semantic truth is provable ($\models$ implies $\vdash$). This ensures the reliability and expressive power of the formal deduction system.

- **Normal Forms:** Conjunctive normal forms(CNF) and other normal forms are easier to build proofs using an automated system. Proper algorithms can be formed by first converting all formulas into normal forms and then prove equivalence automatically.

Relevance to RL/MDPs: While not directly modeling stochasticity, propositional logic provides the basic tools for defining conditions and properties that might characterize states or desired outcomes in simpler, deterministic settings. It also lays the groundwork for more expressive logics.

2.1.2 Predicate Logic

Predicate Logic, or First-Order Logic (FOL), significantly extends the expressive power of propositional logic. It allows us to reason about objects, their properties, and the relations between them, which is crucial for describing more complex systems. Key concepts studied include:

Syntax Beyond Propositions FOL introduces:

- **Terms**: Representing objects, which can be constants, variables, or functions applied to terms (e.g., x, Alice, father(x)).
- **Predicates**: Representing properties of objects or relations between them, which evaluate to true or false (e.g., IsRed(x), Loves(Alice, Bob)).
- **Quantifiers**:
  - ◦ **Universal Quantifier** ($\forall$): "For all" - $\forall x P(x)$ means $P(x)$ is true for every object x in the domain.
  - – **Existential Quantifier** ($\exists$): "There exists" - $\exists x P(x)[\circ]$ means $P(x)$ is true for at least one object x in the domain.
- Formulas are constructed using predicates applied to terms, logical connectives ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$), and quantifiers.

Semantics and Interpretations Unlike propositional logic where semantics are given by truth assignments to atoms, FOL semantics require an interpretation (or model). An **interpretation** consists of:

- A non-empty **domain of discourse** (the set of objects we are talking about).
- Assignments of meaning to constant symbols (mapping them to objects in the domain), function symbols (mapping them to functions over the domain), and predicate symbols (mapping them to relations over the domain).
- The truth of a quantified formula depends on whether its inner part holds for all or some assignments of domain objects to its variables.

Validity, Satisfiability, and Entailment in FOL These concepts are analogous to propositional logic but are evaluated with respect to all possible interpretations or within a specific interpretation. A formula is valid if true in all interpretations. An argument $\Gamma \vDash \phi$ holds if ϕ is true in every interpretation where all formulas in Γ are true.

Relevance to RL/MDPs: FOL provides a much richer language for describing states, especially when states have complex internal structures or involve relationships between multiple entities (e.g., in relational RL or AI planning). While core MDPs often use simpler state representations, the principles of FOL are fundamental to more advanced AI reasoning and knowledge representation that can intersect with RL. My work on the Week 1 problem sheet, particularly question 2.7(ii) involving FO-definability of languages, provided practical experience in applying these concepts to describe properties of sequences.

2.1.3 Finite Automata

Finite Automata (FA) serve as fundamental mathematical models for systems that process information sequentially and transition between a finite set of states based on inputs. They are key to understanding regular languages and form a conceptual bridge to more complex state-based models like Markov Chains. My understanding was primarily built from Baier & Katoen [2] and supplemented by Huth & Ryan [3]. Key concepts include:

- **Core Components**:
  - **Deterministic Finite Automata (DFA)**: Defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, representing states, input alphabet, a unique transition function, a start state, and a set of accept states.
  - **Nondeterministic Finite Automata (NFA)**: Similar to DFAs, but the transition function can lead to multiple states (or no state) for a given state-input pair, and can include $\epsilon-$transitions (transitions without consuming an input symbol).
- **Language Recognition**: An FA accepts a string if, starting from the start state, it reaches an accept state after processing the entire string according to its transition rules. The set of all strings accepted by an FA is the language it recognizes.
- **Equivalence of DFAs and NFAs**: A crucial insight is that NFAs, despite their nondeterminism, recognize the same class of languages as DFAs (the regular languages). The subset construction (or powerset construction) provides an algorithm to convert any NFA into an equivalent DFA.
- **Regular Expressions (REs)**: A concise algebraic notation for specifying regular languages, using operations like concatenation, union ($|$ or $+$), and Kleene star ($*$).
- **Kleene's Theorem**: This fundamental theorem establishes the equivalence in expressive power between finite automata and regular expressions: any language definable by an RE can be recognized by an FA, and vice-versa. Conceptual understanding of the constructive proofs (e.g., RE to NFA$-\epsilon$, FA to RE via state elimination or other methods) was a focus.

Relevance to RL/MDPs: The concept of a system defined by a set of states, with transitions between them triggered by events (inputs in FA, actions/probabilities in MDPs), is central to both automata theory and RL. Understanding FAs helps build intuition for state-space diagrams, reachability, and how sequences of events can lead to specific outcomes or "accepting" (goal) states. This paradigm directly informs the way we model environments and agent interactions in MDPs.

2.1.4 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) extends propositional logic to allow reasoning about sequences of events or states over time. It is particularly useful for specifying dynamic properties of systems. My initial exploration covered:

- **Temporal Modalities**: LTL introduces operators to describe how truth values of propositions change along a linear sequence of time points (or states):
  - Xϕ (Ne**X**t): ϕ holds in the immediate next state.
  - Fϕ (Eventually/**F**inally): ϕ holds at some point in the future (or currently).
  - Gϕ (**G**lobally/Always): ϕ holds at all points in the future (including currently).
  - ϕ U ψ (ϕ **U**ntil ψ): ϕ must hold continuously from the current point until ψ becomes true, and ψ must eventually become true.
- **Interpretation over Traces**: LTL formulas are typically interpreted over infinite sequences of states, known as traces, where each state in the trace assigns truth values to atomic propositions.
- **Expressing Properties**: LTL provides a formal way to express common system properties, such as:
  - Safety properties (e.g., G¬(critical_error) – "a critical error never occurs").
  - Liveness properties (e.g., F(request_granted) – "a request will eventually be granted").

Relevance to RL/MDPs: While not always explicitly used in the design of basic RL algorithms, LTL is fundamental in areas like formal verification of AI systems, model checking of MDPs, and specifying complex, temporally extended goals or constraints in RL (e.g., "always avoid unsafe states while eventually reaching the goal"). Understanding LTL provides a richer language for thinking about the desired long-term behavior of an agent.

2.2 Stochastic Processes: Markov Chains

Markov Chains (MCs) are fundamental stochastic models that describe sequences of possible events where the probability of transitioning to a future state depends only on the current state, not on the sequence of events that preceded it (the Markov Property). They form a critical theoretical underpinning for understanding the dynamics of environments in Reinforcement Learning before introducing agent control. My understanding of MCs was developed through solving problems from the provided practice sheet (Practice Sheet 2) and consulting texts such as "Principles of Model Checking" by Baier & Katoen [2] for formal definitions, and "Reinforcement Learning: An Introduction" by Sutton & Barto [1] for their context within RL. Key concepts explored include:

- **Formal Definition**: A Discrete-Time Markov Chain (DTMC) is typically defined by a tuple $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$:
  - S: is a finite or countable set of States.

- $\mathbf{P}$: is a Transition Probability Matrix (TPM)[1] $\mathbf{P}$ where each entry $\mathbf{P}_{ss'} = \mathbf{P}(S_{t+1} = s'|S_t = s)$ represents the probability of moving from state s to state s' in one time step. The rows of $\mathbf{P}$ must sum to 1.
- ι_{init}: Gives the initial distribution of states.
- AP: gives the set of Atomic Propositions which are true in specific states. For now the only case concerning us is one where atomic propositions are of the states itself.
- L: can be defined as a function $S \to 2^{AP}$ such that $L(s)$ gives the set of atomic propositions true for that state s. It can also be defined as a matrix where L_{sp} gives 1 if proposition p is true for state s and 0 if not.

- **The Markov Property**: The conditional probability distribution of future states of the process, given the present state and all past states, depends only upon the present state and not on any past states:

$$P(S_{t+1} = s'|S_t = s, S_{t-1} = s_{t-1} \dots S_0 = s_0) = \mathbf{P}(S_{t+1} = s'|S_t = s)$$

- **N-Step Transition Probabilities**: The probability of transitioning from state s to s' in exactly n steps in given by $\mathbf{P}^{(n)}_{ss'}$, can be found from the n-th power of the TPM $\mathbf{P}^n$. These probabilities satisfy the **Chapman-Kolmogorov equations**.

- **Classification of States**: States in an MC can be classified based on their long-term properties:
  - **Accessibility and Communication**: State s' is *accessible* from s if there is a path from s to s'. States s and s' *communicate* if both of them are accessible from each other. Communicating states form communicating classes.
  - **Recurrence and Transience**: A state is *recurrent* if, starting from that state, the probability of evantually returning to it is 1. Otherwise, it is transient.
  - **Periodicity**: A state s has a period d if any recurrence of state s must occur in time steps that are multiples of d, where d is the largest such number. An MC is *aperiodic* if it has a period of 1.
  - **Irreducibility**: A Markov Chain is *Irreducible* if all the states are accessible from every other state. In other words it must form a single Communicating class.
  - **Absorbing States**: A state s of an MC is called *absorbing* if it is impossible to leave it after arrival i.e. $P_{ss} = 1$.

- **Stationary Distribution**(π): For some MCs $\exists$ a distribution π such that if initialized according to it, it remains in the same distribution π for all following time steps($\pi = \pi\mathbf{P}$). A unique such distribution π exists for every aperiodic and irreducible finite Markov Chain, and independent of initialization the probability of such MC being in state s converges to $\pi(s)$.

[1]Many sources define $\mathbf{P}$ as a Transition Probability *Function* such that $\mathbf{P} : S \times S \to [0, 1]$ and $\mathbf{P}(s, s')$ represents the probability of transitioning from state s to s'.

Relevance to RL/MDPs: Markov Chains model the inherent dynamics of an environment when no actions are taken or when a fixed policy effectively reduces an MDP to an MC (known as a Markov Reward Process if rewards are included). Understanding MC properties like recurrence, transience, and stationary distributions is crucial for analyzing the long-term behavior of policies and the convergence of RL algorithms. The Gambler's Problem and aspects of Jack's Car Rental, for instance, can be analyzed using MC concepts.

2.3 Markov Decision Processes (MDPs)

Markov Decision Processes (MDPs) extend Markov Chains by incorporating an agent that can make decisions (take actions), influencing state transitions and receiving rewards. MDPs provide the standard mathematical formalism for Reinforcement Learning problems involving sequential decision-making under uncertainty. The primary reference for this section is Baier & Katoen [2], Chapter 10, section 10.6 and Sutton & Barto [1], Chapter 3. An MDP is formally defined as a tuple $\mathcal{M} = (S, A, \mathbf{P}, \iota_{\text{init}}, AP, L)$

S is the finite set of states the system can be in.

A is the set of actions the agent can take. It can also be defined as a function $S \to 2^{\mathcal{A}}$ where $\mathcal{A}$ is the set of all actions possible and $A(s)$ gives the actions available at state s.

P $: S \times \mathcal{A} \times S \to [0, 1]$ is known as the **Transition probability function**. For all $s \in S$ and $\alpha \in A(s)$, $\sum\limits_{s' \in S} \mathbf{P}(s', s, a) \in \{0, 1\}$. It defines the dynamics of the problem.

$\boldsymbol{\iota_{\text{init}}}$ $: S \to [0, 1]$ is the initial distribution which defines the probability($\iota(s)$) of MDP starting from a state(s).

AP is the set of atomic propositions that can be true at various states, for present practical considerations we will be assuming AP to represent a set of rewards possible on reaching a reward.

L $: S \to 2^{AP}$ is the labeling function for atomic propositions.

Our agent based MDPs have a goal to maximise the reward it gets not only in present but in all time including present and future. Thus we define a goal G_t as the goal the MDP needs to maximise at time t by chosing some action. G_t can be written as $\mathbb{E}\left[\sum\limits_{i=t+1}^{i\to\infty} R_i\right]$, however it has some problems:

- Defining $G_t = \mathbb{E}\left[\sum\limits_{i=t+1}^{i\to\infty} R_i\right]$ makes all rewards in present and future equally important, thus the agent might have to look too far in future to decide and in case of infinitely long episode the agent will take infinite time to decide its action.
- If some a sequence of actions give the agent a steady reward, over infinity the value of G_t will be undefined as it will be a sum of non converging infinite sequence.

Thus another idea to define the goal introduces the concept of **Discounting**. We assume a constant γ such that $G_t = \mathbb{E}\left[\sum_{i=t+1}^{i\to\infty} \gamma^{i-t-1} R_i\right]$. This directly translates to taking present reward as whole($i = t+1, \gamma^{i-t-1} = 1$) R_{t+1} and future awards are discounted by factor of γ^n where n is larger for in future rewards making future rewards less important in present. This solves both of our problems in most cases(except the ones where R_t diverges faster than the exponent converges, but it is unlikely to occur). At a certain point we can cut off our calculation and oversight as the exponent converges the sequence to make all terms after some terms to make negligibly small.

For practical purposes we will be defining MDPs as a tuple $\mathcal{M} = (S, A, \mathbf{P}, R, \gamma)$, where R is a function $S \times A \to \mathcal{R}$ such that $\mathcal{R}$ is set of all rewards possible and $R(s,a)$ gives the reward for taking action a in state s. Therefore $R(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$. γ gives the discount factor that the agent uses to decide the worth of future action.

Here are some more important frameworks for agent based MDPs:

Policy (π) A policy is agent's behavior function in the system. It maps the states to a probability distribution over actions to take. $\pi(a|s) = Pr\{A_t = a|S_t = s\}$. A policy can be stochastic or non stochastic(where states are mapped to unique actions to take).

Value Functions is a function defined to measure the goodness of certain states and actions. It is of two types:

- **State Value Function** v_π maps states to expected cumulative reward from that state under policy π.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} R_{t+i+1}\gamma^i|S_t = s\right]$$

- **Action State value function** q_π gives the expected cumulative reward for choosing an action a in state s.

$$q_\pi(a,s) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} R_{t+i+1}\gamma^i|S_t = s, A_t = a\right]$$

2.3.1 Bellman Equations

Bellman Equations were given by "Richard E. Bellman". These equations express a recursive relationship between the value of a state (or state-action pair) and the values of its successor states, under a given policy π. They are fundamental for most RL algorithms.

For value function ($v_\pi(s)$), we know

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t=s] && (2.5)\\
&= \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} R_{t+i+1}\gamma^i|S_t=s\right] && (2.6)\\
&= \sum_{a\in A} Pr(A_t=a|S_t=s)\mathbb{E}_\pi[G_t|S_t=s,A_t=a] && (2.7)\\
&= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R} Pr(S_{t+1}=s',R_{t+1}=r|S_t=s,A_t=a)G_t && (2.8)\\
&= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)\mathbb{E}_\pi[G_t|S_t=s,A_t=a,R_{t+1}=r,S_{t+1}=s] && (2.9)\\
&= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)\mathbb{E}_\pi\left[\sum_{i=0}^{\infty} R_{t+i+1}\gamma^i|S_t=s,A_t=a,R_{t+1}=r,S_{t+1}=s\right] && (2.10)\\
&= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)\mathbb{E}_\pi\left[(R_{t+1}+\sum_{i=1}^{\infty} R_{t+i+1}\gamma^i)|S_t=s,A_t=a,R_{t+1}=r,S_{t+1}=s\right] && (2.11)\\
&= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)(r+\mathbb{E}_\pi\left[\gamma\sum_{i=0}^{\infty} R_{t+2+i}\gamma^i|S_t=s,A_t=a,R_{t+1}=r,S_{t+1}=s\right]) && (2.12)\\
&= \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)(r+\gamma v_\pi(s')) && (2.13)
\end{aligned}$$

Therefore the Bellman equation for state value function is

$$v_\pi(s) = \sum_{a\in A}\pi(a|s)\sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)(r+\gamma v_\pi(s'))$$

Similar method can be taken to prove the Bellman equation for state action value function:

$$q_\pi(s,a) = \sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)(r+\gamma\sum_{a'\in A}\pi(a'|s')q_\pi(a',s'))$$

or

$$q_\pi(s,a) = \sum_{s'\in S, r\in R}\mathbf{P}(s',r|s,a)(r+\gamma v_\pi(s'))$$

2.3.2 Optimality

An optimal policy, denoted π_*, is a policy that achieves higher or equal expected return than any other policy from all states. It is the best the agent can do in order to work toward its goal. It maximizes the G_t an agent can get right from initial state.
Following this idea we can define the **Optimal value functions** (v_* and q_*) as:

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s\in S$$

$$q_*(a,s) = \max_{\pi} q_\pi(a,s) \quad \forall a \in A, s \in S$$

With slight variation we can alter Bellman's equations to **Bellman's optimality equations**:

$$v_\pi(s) = \max_{a \in A} \sum_{s' \in S, r \in R} \mathbf{P}(s', r|s, a)(r + \gamma v_\pi(s'))$$

$$q_\pi(s,a) = \sum_{s' \in S, r \in R} \mathbf{P}(s', r|s, a)(r + \gamma \max_{a' \in A} q_\pi(a', s'))$$

Relevance to RL: MDPs are the mathematical framework for almost all problems Reinforcement Learning aims to solve. Understanding this formalism, especially the Bellman equations, is absolutely critical for developing and analyzing RL algorithms, including the Dynamic Programming methods (Policy Iteration, Value Iteration) discussed in subsequent chapters, as well as model-free methods like Q-learning. All my practical implementations (GridWorld, Jack's Car Rental, Gambler's Problem) are direct applications of solving MDPs.

Chapter 3

Exploration and Exploitation: Multi-Armed Bandits

The k-Armed Bandit problem serves as a simplified, yet fundamental, introduction to several core concepts in Reinforcement Learning, most notably the critical trade-off between exploration (trying out actions to learn more about the environment) and exploitation(choosing the action currently believed to give best reward). This chapter details the problem formulation and discusses several algorithms implemented to address it, based on Chapter 2 of Sutton and Barto [1]. My implementations and experiments are available in the associated GitHub repository [4].

3.1 The k-Armed Bandit Problem

A k-armed bandit problem models a scenario where an agent is faced with k different options, or "arms" (like slot machine levers, as slot machine is a one armed bandit). At each time step t, the agent chooses an action $A_t \in \{1, ..., k\}$, and receives a numerical reward R_t drawn from a normal probability distribution(with fix means and unity variance) associated with the chosen arm A_t. The true mean reward for action a, denoted $q_*(a)$, is initially unknown to the agent. The agent's goal is to maximize the total reward obtained over a series of time steps. Another point of view involves calculating regret(Δ_a) which is the difference between the reward received by agent and the best reward it could have received. The point is to minimize the quantity of total regret $\left(\sum_{t=0}^{\infty} \Delta_{A_t}\right)$ i.e. the sum of all regrets. According to Lai and Robbins theorem total regret grows at least logarithmically. Thus the minimum total regret an agent can aim for is logarithmic.

The agent maintains an estimate $Q_t(a)$ of the true action value $q_*(a)$. A common way to estimate this is the sample-average method:

$$Q_t(a) \doteq \frac{\text{sum of rewards when action } a \text{ taken prior to step } t}{\text{number of times action } a \text{ taken prior to step } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{I}(A_i = a)}{\sum_{i=1}^{t-1} \mathbb{I}(A_i = a)} \tag{3.1}$$

where $\mathbb{I}(\cdot)$ is an indicator function. The challenge lies in balancing the desire to exploit arms with high current estimates versus exploring other arms to potentially discover even better options.

3.2 Action-Value Methods Implemented

Several action-value methods were implemented and tested to understand their behavior in balancing exploration and exploitation.

3.2.1 Epsilon-Greedy Action Selection

The ϵ-greedy method is a simple, basic and most used strategy. With a small probability ϵ, the agent chooses an action uniformly at random from all k actions (exploring the environment). With probability $1 - \epsilon$, the agent chooses the action with the highest current estimated value (exploiting it's knowledge).

The performance of ϵ-greedy algorithms was compared for varying values of ϵ, as shown in Figure 3.1. Both non zero values of ϵ(0.1 and 0.01) performs better than pure greedy $\epsilon = 0$. Also higher ϵ allows sooner realization of true mean values but over longer run converge and lag behind lower ϵ as once low ϵ can exploit it's later found knowledge more that high ϵ.

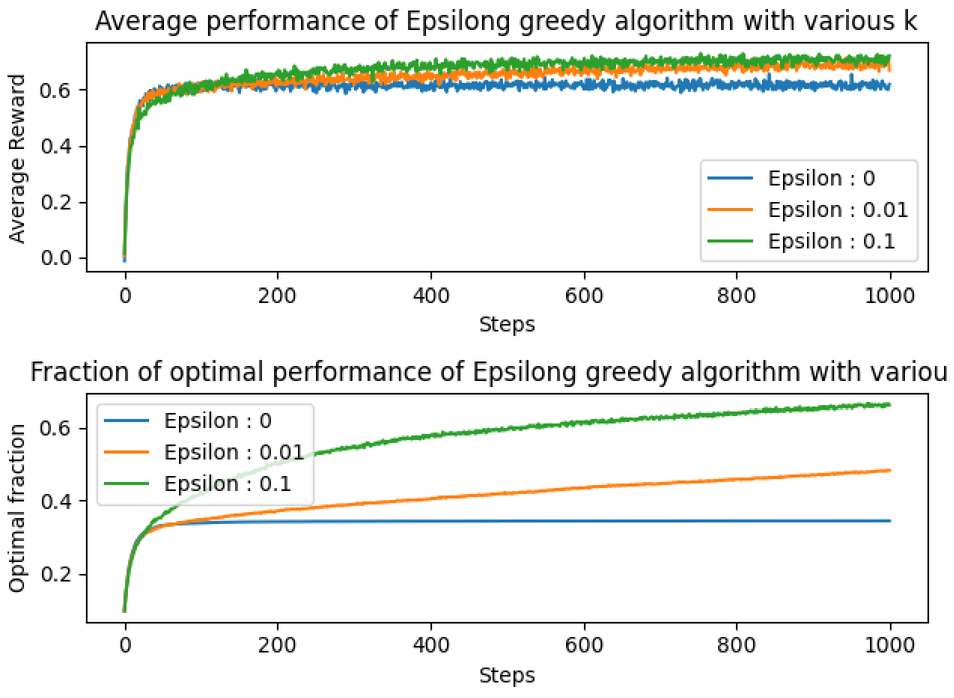


Figure 3.1: Comparison of ϵ-greedy algorithms with varying ϵ values over 1,000 steps, averaged over 10,000 runs.

3.2.2 Handling Non-Stationary Problems

In many real-world scenarios, the reward distributions for actions can change over time (non-stationary problems). The standard sample-average method is not well-suited for this, as it gives equal weight to all past rewards. A more effective approach is to use a constant step-size parameter $\alpha \in (0, 1]$ in the update rule for $Q_{t+1}(A_t)$:

$$Q_{n+1} \doteq Q_n + \alpha[R_n - Q_n] \tag{3.2}$$

This exponentially weighted average gives more weight to recent rewards. Figure 3.2 illustrates the performance difference. It is worth noting that α can also be a function of n. In sample-average case $\alpha = 1/n$.

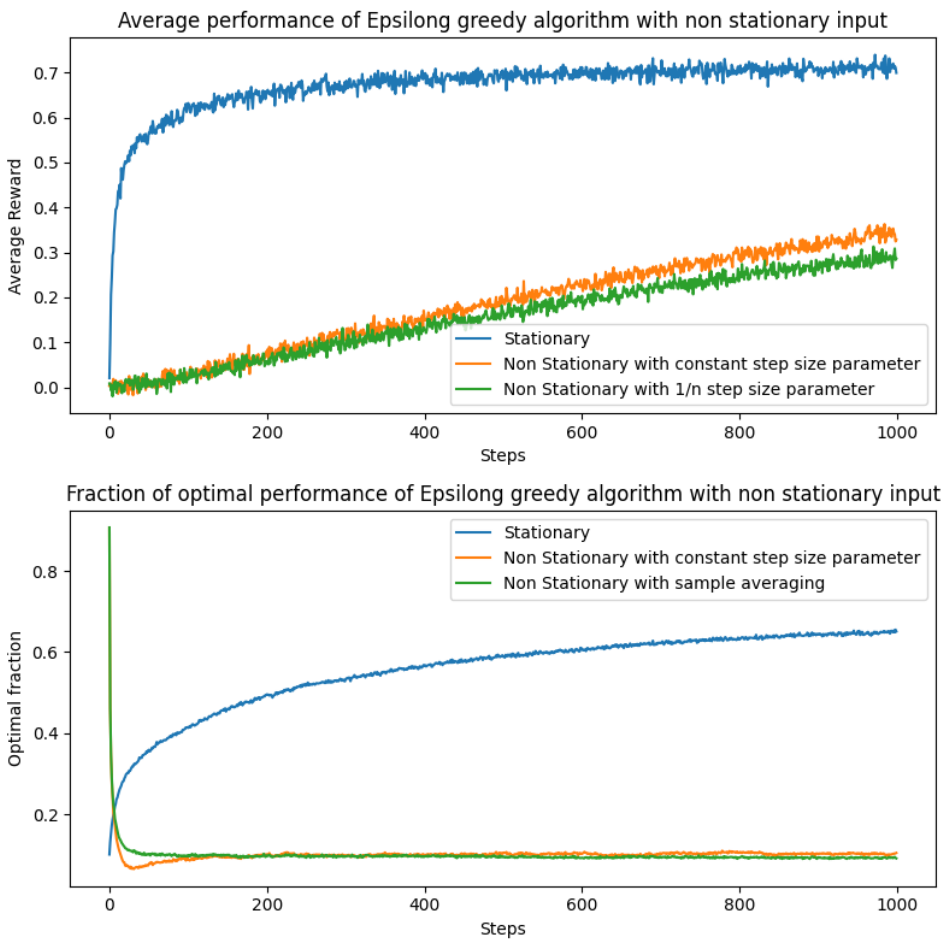


Figure 3.2: Performance comparison on a non-stationary bandit problem: sample-average vs. constant step-size α.

The figure shows that for normal $\epsilon-$greedy stationary case($\epsilon = 0.1$) performing very well. However for a non-stationary input the performance deteriorates with constant step size α case performing better.

3.2.3 Optimistic Initial Values

Initializing action-value estimates $Q_1(a)$ to a high, optimistic value (e.g., higher than any possible true mean reward) can encourage early exploration. When an arm is chosen, if the reward received is less than the initial optimistic value, the estimate for that arm decreases, making other (still optimistically valued) arms more likely to be chosen. Figure 3.3 shows its effect.

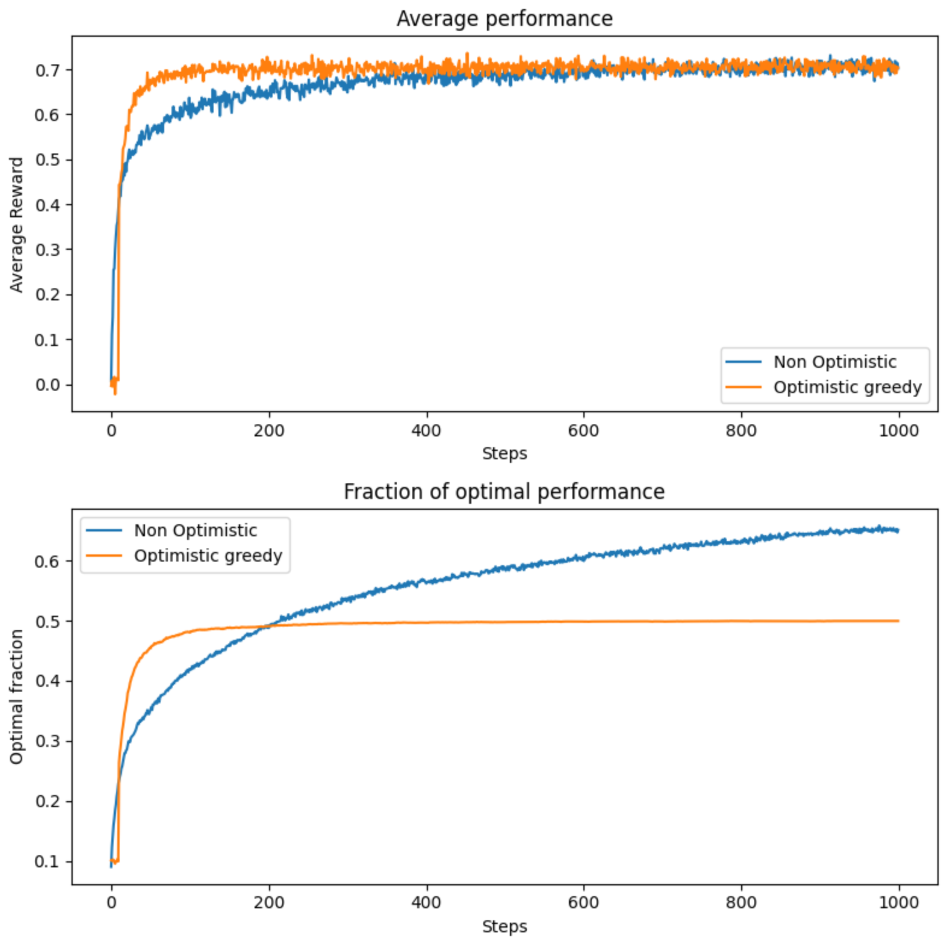


Figure 3.3: Effect of optimistic initial values (e.g., $Q_1(a) = 5, \epsilon = 0$) compared to realistic initial values with ϵ-greedy (e.g., $Q_1(a) = 0, \epsilon = 0.1$).

The graph shows that optimistic greedy policy trains very fast owing to it's early high exploration rate. However because of being completely greed($\epsilon = 0$) sticks to a policy after certain turns and then $\epsilon-$greedy policy outperforms it.

3.2.4 Upper Confidence Bound (UCB) Action Selection

UCB addresses exploration more strategically by selecting actions according to:

$$A_t \doteq \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{3.3}$$

where $N_t(a)$ is the number of times action a has been selected prior to time t, and $c > 0$ controls the degree of exploration. The square-root term is a measure of uncertainty or potential for arm a; UCB favors arms that are either estimated to be good ($Q_t(a)$ is high) or have not been tried often ($N_t(a)$ is low). According to Lai and Robbins theorem the total regret in case of UCB grows logrithmically making it mathematically best algorithm, the results are shown in Figure 3.4.

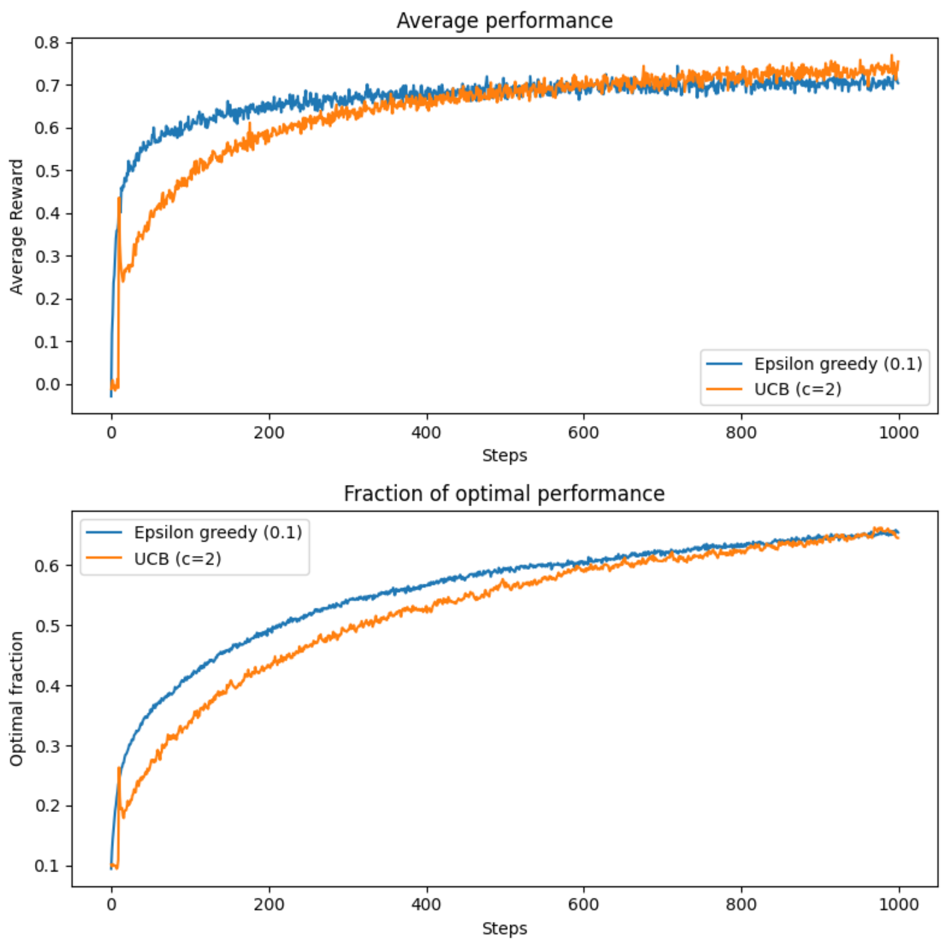


Figure 3.4: Performance of UCB action selection (with $c = 2$) compared to ϵ-greedy.

3.2.5 Gradient Bandit Algorithms

Gradient bandit algorithms learn a numerical preference $H_t(a)$ for each action a. Actions are selected stochastically according to a softmax distribution over these preferences:

$$\pi_t(a) \doteq \Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \tag{3.4}$$

The preferences are updated using stochastic gradient ascent:

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)) \text{ for the chosen action } A_t$$

and

$$H_{t+1}(a) \doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a) \quad \forall a \neq A_t$$

$\bar{R}_t$ is a baseline, often the average of rewards received up to time t. The performance with and without a baseline is compared in Figure 3.5.

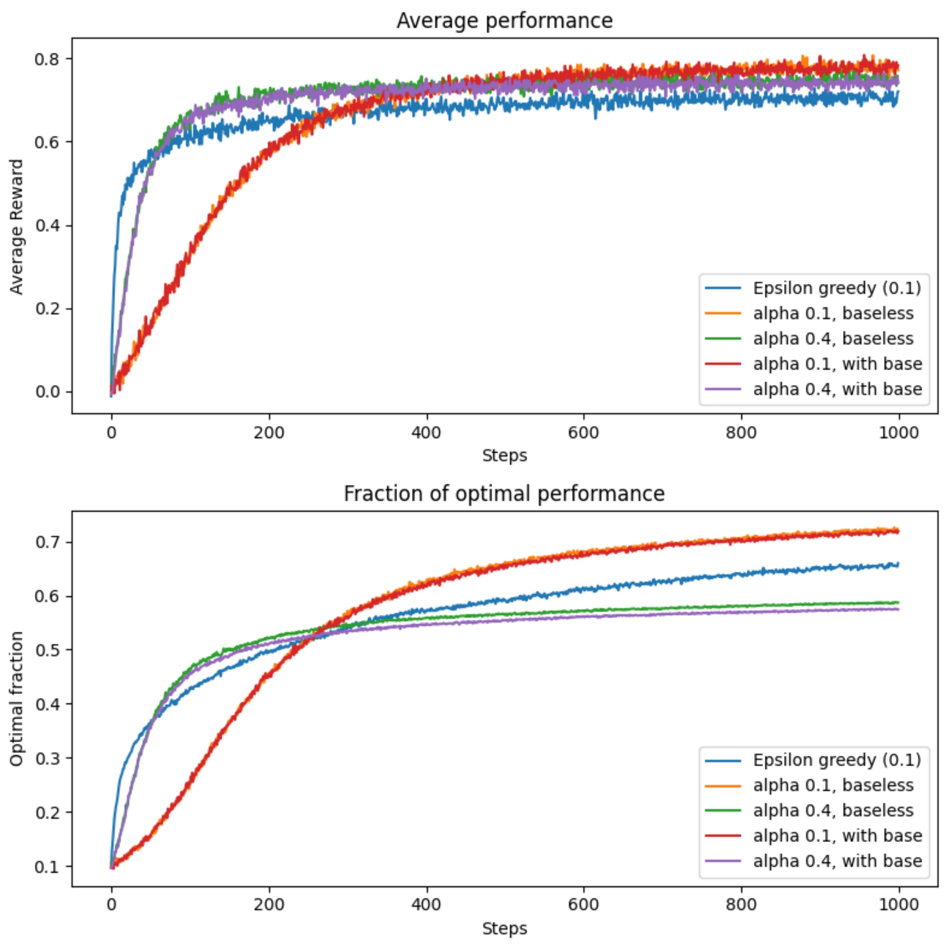


Figure 3.5: Performance of Gradient Bandit algorithms with different α values and with-/without a baseline.

The plots show that average performance of gradient algorithm is better than $\epsilon-$greedy but slower to train. higher α performs much better than other policies.

3.3 Summary of Bandit Experiments

The k-armed bandit problem, despite its simplicity, encapsulates the fundamental exploration-exploitation dilemma in RL. Implementing various strategies like ϵ-greedy, UCB, and gradient bandits provided practical insights into their mechanisms and performance characteristics. Optimistic initialization demonstrated a simple way to encourage early exploration. For non-stationary environments, using a constant step-size for value updates proved crucial for adaptability. Gradient bandit algorithms offer a different approach by learning action preferences and giving an optimal stochastic policy, and the use of a baseline was shown to stabilize learning. These experiments highlight that no single method is universally superior; the best choice often depends on the specifics of the problem, such as stationary and the desired balance between exploration and long-term reward, with mathematically derived methods being slightly better than naive approaches.

Chapter 4

Dynamic Programming for Solving MDPs

Dynamic Programming (DP) refers to a collection of algorithms that can compute optimal policies given a perfect model of the environment as a Markov Decision Process (MDP). While DP methods are often computationally expensive and require a full model (which is not always available in RL), they are theoretically important as they provide a foundation for understanding more advanced RL algorithms. This chapter, based on Sutton & Barto Chapter 4 [1], discusses the core DP algorithms and their application to several case studies.

4.1 Overview of Dynamic Programming in RL

DP algorithms in the context of RL leverage the Bellman equations to iteratively find optimal value functions and policies. Key assumptions include:

- The environment dynamics $p(s', r|s, a)$ are fully known.
- The state and action spaces are finite (though extensions exist).

DP methods work by turning Bellman equations into update rules that are applied repeatedly to improve approximations of the desired value functions or policies until convergence to the optimal solution.

4.2 Policy Iteration

Policy Iteration is an algorithm that alternates between two main steps: policy evaluation and policy improvement, until the policy is stable and optimal. It can be quite computationally expensive but is sure to give an optimal policy.

4.2.1 Policy Evaluation (Prediction)

Given a policy π, policy evaluation (or prediction) is the process of computing the state-value function v_π. This is achieved by iteratively applying the Bellman expectation equation as an update rule:

$$v_{k+1}(s) \doteq E_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \tag{4.1}$$

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{4.2}$$

This update is applied to all states $s \in S$ in each iteration k, until v_k converges to v_π however some tolerance θ needs to be specified.

4.2.2 Policy Improvement

Once v_π for a policy π is known, we can try to improve the policy. For each state s, we consider if selecting an action $a \neq \pi(s)$ and then following π would be better. This involves computing $q_\pi(s, a)$ for all $a \in A(s)$:

$$q_\pi(s, a) \doteq E[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \tag{4.3}$$

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{4.4}$$

A new greedy policy π' is then formed by choosing $a' = \operatorname{argmax}_a q_\pi(s, a)$ for each state s. The Policy Improvement Theorem guarantees that if $\pi' \neq \pi$, then $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in S$. It can be observed as if we take the greedy action at this position it will improve or at least give same result as following policy π, and then on using π will continue to give same results. Thus the new policy is going to an improvement on the previous policy π.

4.2.3 The Policy Iteration Algorithm

Policy Iteration combines these two steps:

1. Initialization: Start with an arbitrary policy π_0 and value function v_0.
2. Policy Evaluation: Compute v_{π_k} using iterative application of Equation 4.1 until convergence.
3. Policy Improvement: Construct a new policy π_{k+1} by acting greedily with respect to v_{π_k}.
4. If $\pi_{k+1} = \pi_k$, then the algorithm has converged to the optimal policy π_* and optimal value function v_*. Otherwise, repeat from step 2 with $k \leftarrow k + 1$.[1]

[1]It is worth noting that the algorithm may not converge if the policies keep switching between two optimal policies. Thus a check may be introduced to prevent this.

4.3 Value Iteration

Value Iteration combines aspects of policy evaluation and policy improvement into a single, simpler update rule. It is much for computationally simple compared to policy iteration. It directly applies the Bellman optimality equation as an update:

$$v_{k+1}(s) \doteq \max_a E[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s, A_t = a] \tag{4.5}$$

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{4.6}$$

This update is applied iteratively for all states s until v_k converges to v_*. Once v_* is found, an optimal deterministic policy can be extracted by choosing actions greedily with respect to v_*. Value Iteration often converges faster than Policy Iteration if policy evaluation steps are computationally expensive.

4.4 Implementation Case Study 1: GridWorld Environment

To practically explore these DP algorithms, a custom GridWorld environment was developed using Pygame.

4.4.1 Environment Description

The GridWorld environment consists of a $N \times M$ grid. States are represented by (row, col) tuples. The agent can take one of four deterministic actions: `UP, DOWN, LEFT, or RIGHT`. Transitions move the agent one cell in the chosen direction unless a `WALL` tile is encountered, in which case the agent remains in its current position getting a reward of -1. Specific tiles offer rewards or penalties: `LAVA` tiles incur a large negative reward(-10) and reset the episode, `TREASURE` tiles provide a large positive reward(+10) and reset, `COIN` tiles offer a smaller positive reward(+1) and reset, and `NORMAL` tiles provide a small negative step cost (or zero reward here). An example map is shown in Figure 4.1.

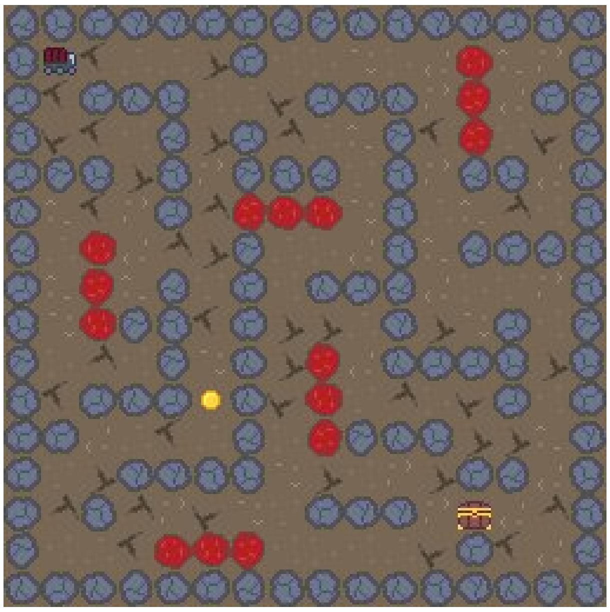


Figure 4.1: A sample 16 × 16 map used in the custom GridWorld environment, illustrating different tile types (e.g., `NORMAL` (brown), `WALL` (grey rocks), `LAVA` (red puddles), `TREASURE` (chest), `COIN` (yellow)).

```python
class TileTypes(enum.Enum):
    NORMAL = 0
    LAVA = -10
    TREASURE = 10
    COIN = 1
    WALL = -1 # Effectively an invalid state to transition into
# Rewards are directly from the map: self.reward = self.mapping[*self.
    truck_pos]
# Transitions are deterministic based on action, with wall-bumping.
```

Listing 4.1: GridWorld Tile Types and Rewards (Conceptual)

4.4.2 Policy Iteration Implementation and Results

Policy Iteration was implemented for the GridWorld. The ‘evaluate_policy‘ method iteratively computed the state values under the current policy, and ‘improve_policy‘ updated the policy to be greedy with respect to these values. This process was repeated until the policy stabilized. The converged value function and the derived optimal policy for the map in Figure 4.1 are shown in Figures 4.2 and 4.3 respectively.

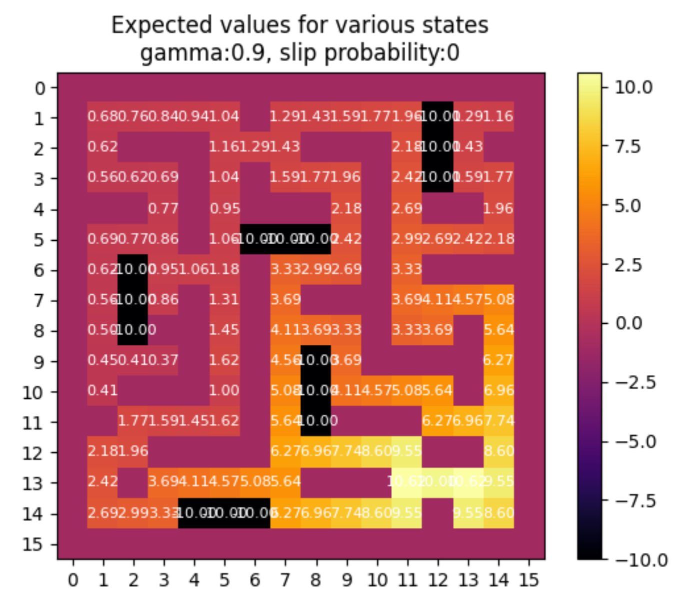


Figure 4.2: Converged state-value function (v_*) for the GridWorld environment obtained using Policy Iteration. Brighter colors indicate higher values.

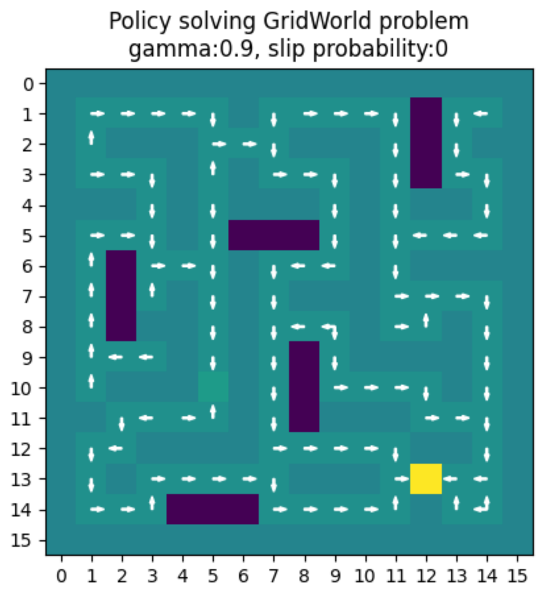


Figure 4.3: Optimal policy for the GridWorld environment derived from Policy Iteration. Arrows indicate the optimal action in each state.

4.4.3 Value Iteration Implementation and Results (If completed)

Value Iteration was also implemented, directly applying the Bellman optimality update (Equation 4.5) until the value function converged. The resulting optimal value function and policy were identical to those found by Policy Iteration, as expected, though convergence characteristics (number of iterations/time) might differ depending on the problem specifics.

4.5 Implementation Case Study 2: Jack's Car Rental

The Jack's Car Rental problem (Sutton & Barto, Example 4.2) is a more complex MDP involving stochastic demand and returns for rental cars at two locations. This was a significantly difficult problem to implement and run with least expected complexity.

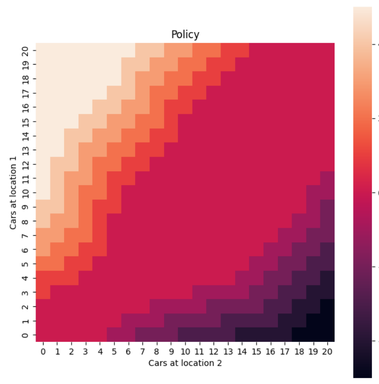
4.5.1 Problem Description

As the name suggests the problems involves Jack who controls two car rental locations for a company. Every car rented gives a 10 credit reward to him. Thus the state is defined by the number of cars at each of two locations at the end of the day, (n_1, n_2), where $0 \leq n_1, n_2 \leq 20$. Jack has the option of moving a number of cars from one location to other overnight, each move costing 2 credits. Hence, actions are the number of cars moved overnight from location 1 to location 2, ranging from -5 (move 5 from loc2 to loc1) to +5 (move 5 from loc1 to loc2). Rewards include income from rentals ($10 per car rented) minus the cost of moving cars ($2 per car moved). Car rental requests and returns at each location follow independent Poisson distributions with specified mean values ($\lambda_{req1} = 3, \lambda_{req2} = 4, \lambda_{ret1} = 3, \lambda_{ret2} = 2$). There's a maximum capacity of 20 cars at each location; excess returns are lost, and requests are capped by availability. There is also an extension to problem which involves dynamics that Jack can move one car from first to second location, and having more than 10 cars resulting in extra parking fees(Sutton & Barto, Exercise 4.7).

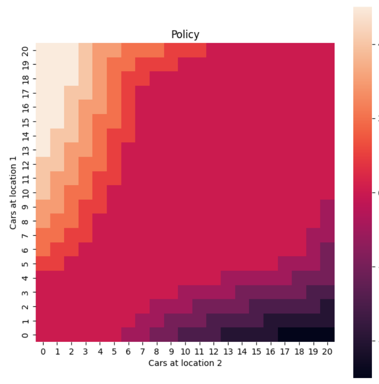
4.5.2 Policy Iteration Implementation Approach

The problem was modeled with an `Agent` class managing the policy and value function, and an `Env` class (specifically, its `simulate_day` method) calculating the expected one-step return $E[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a]$. The core challenge lies in computing this expectation, which involves summing over all possible numbers of rental requests and returns at both locations. A naive implementation required multiple nested loops, which was my first attempt resulting in 4 nested loops within the `simulate_day` function. This resulted in a total of 8 nested loops and the program ran very slowly. Finding the solution with my only computational resource, my laptop(Intel Ultra 9 processes) was going to take too long. It was then that I scraped it to try a new a vectorized approach using NumPy developed within the `simulate_day` method (Listing 4.2). This involves creating a 4-dimensional arrays for Poisson probabilities of requests/returns and broadcasting operations to compute outcomes for all combinations simultaneously, then summing to get the final expectation. The discount factor γ was set to 0.9. This took a significant debugging effort as errors were prone because of difficult imagination of a 4-dimensional array and required some AI help to generate debugging code.

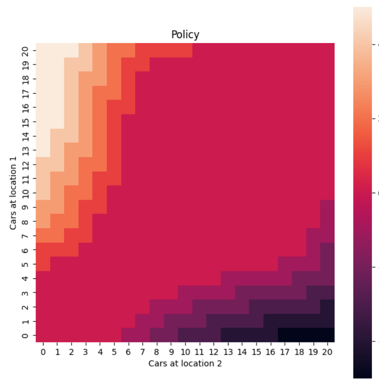
```python
# ... (setup of prob1, prob2, prob3, prob4 4D arrays for Poisson
    probabilities using scipy function) ...
# ... (setup of ncars and credits 4D arrays based on movement, requests
    , returns) ...
ncars = ncars.astype(np.int32) # Ensure integer indices for value array
future_value = values[ncars[0], ncars[1]] # Advanced indexing for V(s')
# Element-wise product of joint probabilities and (immediate rewards +
    discounted future values)
result_matrix = prob1 * prob2 * prob3 * prob4 * (credits + gamma *
    future_value)
```



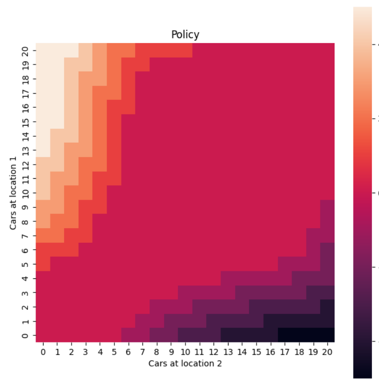
(a) policy after 1st iteration



(b) policy after 2nd iteration



(c) policy after 3rd iteration



(d) converged policy after 4th iteration

Figure 4.4: Jack's Car Rental:Policy iteration. Positive values indicate moving cars from location 1 to 2; negative values indicate movement from 2 to 1.

```
return result_matrix.sum() # Sum over all dimensions of requests/
    returns
```

Listing 4.2: Key Vectorized Expectation Snippet from Jack's Car Rental `simulate_day`

4.5.3 Results and Analysis

Policy Iteration was run until the policy stabilized. Figures 4.4a through 4.4d show the evolution of the policy (average number of cars moved) across iterations. The corresponding converged value function is shown in Figure 4.5. The implementation took approximately one hour to converge on the available hardware. The solution to extended problem is visible in Figure 4.6

It can be seen that the policy learns to equalize the number of cars in both locations in order for maximum cars to rent but still preferring more cars at location with higher mean renting requests.

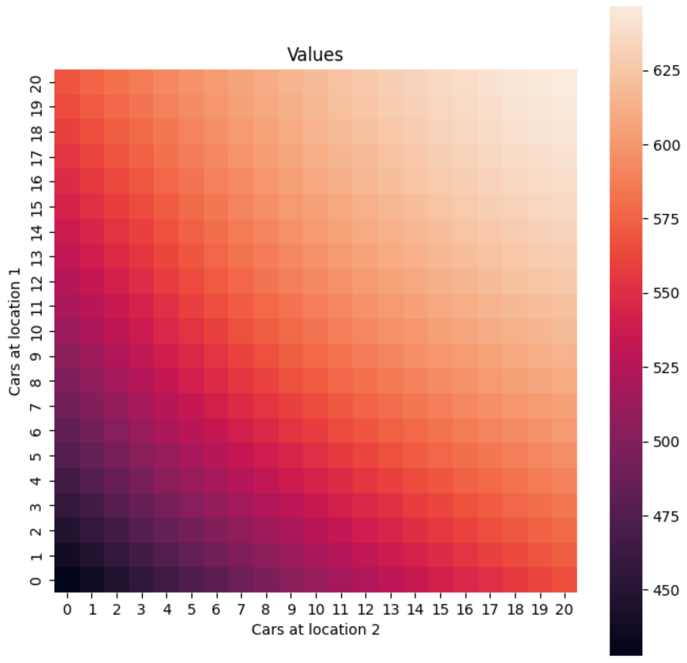
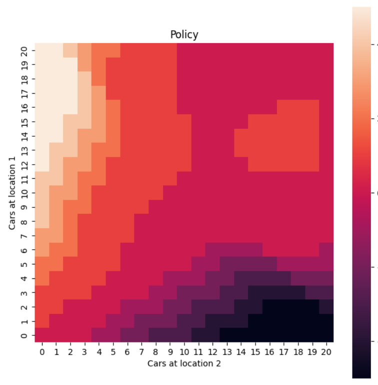
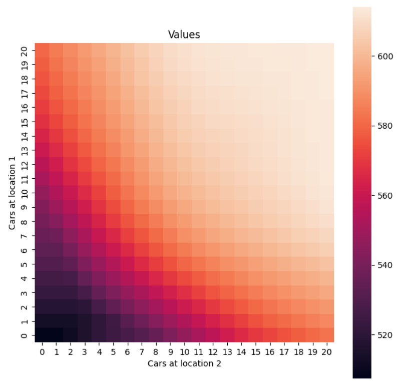


Figure 4.5: Jack's Car Rental: Converged state-value function (v_*).



(a) Extended problem solution policy



(b) Extended problem value function

Figure 4.6: Solution to extended Jack's car rental problem.

4.6 Implementation Case Study 3: Gambler's Problem

The Gambler's Problem (Sutton & Barto, Example 4.3) involves a gambler betting on coin flips to reach a capital goal.

4.6.1 Problem Description

The state $s \in \{1, 2, \cdots, 99\}$ is the gambler's capital. The goal is to reach a capital of \$100. Reaching \$0 means ruin. The action a is the stake, $a \in \{0, 1, 2, \cdots, \min(s, 100 - s)\}$. If the coin flip is heads (probability p_h), the gambler wins the stake; otherwise (probability $1 - p_h$), he loses it. Reaching \$100 yields a reward of $+1$, and all other transitions yield 0 reward. The discount factor $\gamma = 1$.

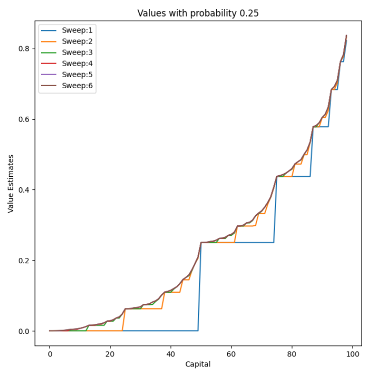
4.6.2 Value Iteration Implementation Approach

Value Iteration was implemented to find the optimal value function and policy. The `iterate_values` method repeatedly sweeps through all states, updating the value of each state s by taking the maximum expected value over all possible valid bets from that state, using the Bellman optimality equation: $v_{k+1}(s) = \max_{a \in A(s)}[p_h(R_{win} + \gamma v_k(s + a)) + (1 - p_h)(R_{loss} + \gamma v_k(s - a))]$ where $v_k(100) = 1$ and $v_k(0) = 0$. The policy is then derived by choosing the bet(s) that achieve this maximum.

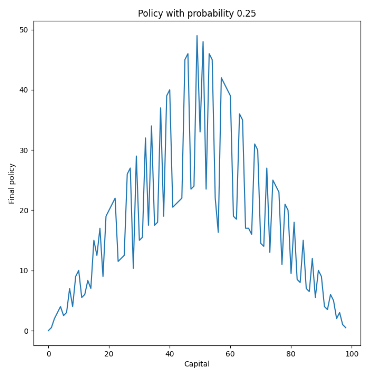
4.6.3 Results and Analysis

The algorithm was run for different probabilities of heads (p_h). Figures 4.7 shows the convergence of the value function and the optimal policies obtained for various values of p_h. For $p_h < 0.5$ (e.g., 0.25, 0.4), the value function is concave and the policy is often to bet boldly, either everything or just enough to reach the goal, especially when capital is low or high. The policy exhibits a characteristic jagged pattern. For ph $>$ 0.5 (e.g., 0.55), the game is favorable, the value function is convex (or near linear), and the optimal policy tends to be more conservative, often betting small amounts (e.g., \$1) to gradually accumulate capital. These results align closely with those presented in Sutton & Barto (Figure 4.3). The optimal policy is non uniform as it is the weighed average of actions with their probabilities as weights. There is no unique optimal policy for this problem hence the non-uniformity.

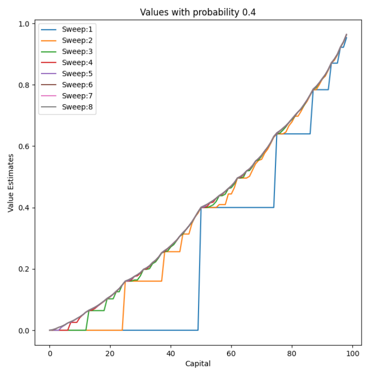
Figure 4.7: Gambler's problem value function sweeps and optimal policies for $p_h \in \{0.25, 0.4, 0.55\}$



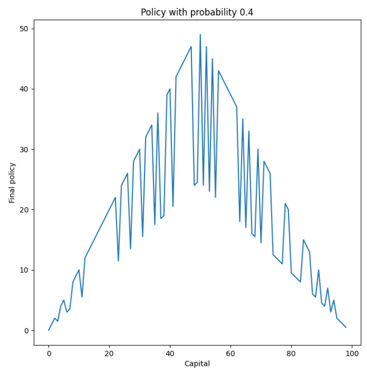
(a) Value function sweeps for $p_h = 0.25$.



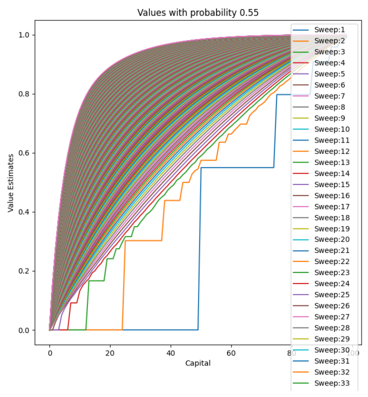
(b) Optimal policy (stake) for $p_h = 0.25$.



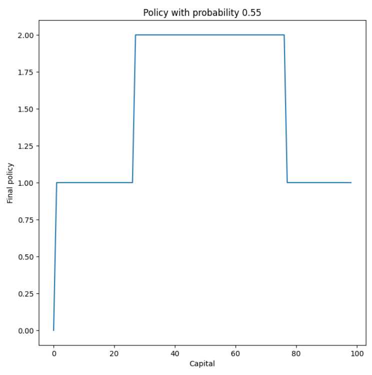
(c) Value function sweeps for $p_h = 0.4$.



(d) Optimal policy (stake) for $p_h = 0.4$.



(e) Value function sweeps for $p_h = 0.55$.



(f) Optimal policy (stake) for $p_h = 0.55$.

Chapter 5

Model-Free Learning via Monte Carlo Methods

While Dynamic Programming provides a powerful toolkit for solving MDPs, it essentially relies on "Bootstrapping". It uses a critical assumption: a perfect model of the environment's dynamics (like the function $p(s', r|s, a)$[1]) must be available. In many real-world problems and games, such a model is unknown or too complex to implement. Model-Free Reinforcement Learning addresses this by enabling an agent to learn directly from experience gathered through interaction with the environment.

This chapter explores Monte Carlo (MC) methods. In statistics Monte Carlo methods refers to an analytical method which involves predicting results based on random sampling from a large data set. In RL it a class of model-free algorithms that learn value functions from complete, episodic experiences. The key idea is to estimate the value of a state or state-action pair by averaging the returns observed following visits to that state or pair. This work is based on Chapter 5 of Sutton & Barto [1].

5.1 Monte Carlo Prediction

Monte Carlo Prediction is used to estimate the state-value function $v_\pi(s)$ for a given policy π. After each episode, the return G_t is calculated for every time step t. An estimate $V(s)$ is then updated by averaging the returns from all visits to state s. As the number of episodes approach infinity the estimate $V(s)$ is bound to converge to $v(s)$.

5.1.1 Case Study: Blackjack State-Value Estimation

To demonstrate MC Prediction, the value function for a simple, fixed policy in a simplified game of Blackjack was estimated. The policy dictates that the player should "hit" until their sum is 20 or 21, and then "stand". The state is defined by the player's current sum (12-21), the dealer's showing card (Ace-10), and whether the player holds a usable Ace.[2]

After simulating 500,000 episodes, the state-value function was computed. The results, shown in Figure 5.1, closely replicate the findings in Sutton & Barto (Figure 5.1), correctly

[1]$p(s', r|s, a)$ is the probability of transition from state s to state s' when action taken is a and the trans

[2]It is assumed that counting cards gives no advantage to player which is un-realistic yet important assumption for the state space to be of manageable size. We can say that we assume an infinite number of card decks are used in game play, thus game with multiple decks will be closer to this value function.

identifying the suboptimal nature of this fixed policy, especially the sharp drop in value for states where the player is forced to hit despite having a high sum. It is worth noting that the probability of reaching a state with usable ace is rare and so even after 500,000 episodes the value function is not smooth but is converging on the true value function.

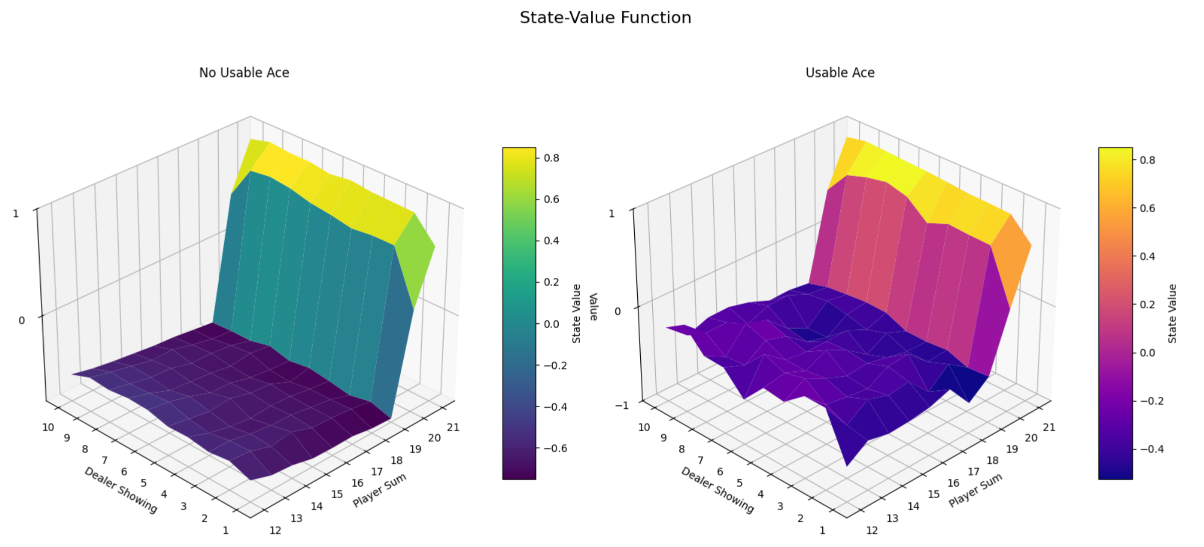


Figure 5.1: Estimated state-value function for Blackjack under a fixed policy (hit until sum is 20 or 21). The value drops significantly for high player sums due to the high probability of busting.

5.2 Monte Carlo Control

MC Control extends prediction to find an optimal policy. This involves estimating action-value function $q_\pi(s, a)$ and improving the policy based greedily based on these estimates.

5.2.1 On-Policy MC Control with Exploring Starts

A common approach is On-Policy First-Visit MC Control with Exploring Starts (ES). To ensure all state-action pairs are visited, each episode is started in a randomly chosen state-action pair. The Q-values are then updated based on the returns, and the policy is improved by making it greedy with respect to the updated Q-values. This can be made more robust by choosing state-action pairs based on the number of times they have been visited to promote exploration of states which have been explored the least.

Case Study: Optimal Policy for Blackjack

The MC with Exploring Starts method was implemented to find the optimal policy for Blackjack. The resulting optimal policy and its corresponding optimal value function are shown in Figure 5.2. These results are a near-perfect replication of Sutton & Barto (Figure 5.2), demonstrating the algorithm's ability to discover well-known optimal strategies for this game, such as when to stand on a "soft 18" versus the dealer's card.

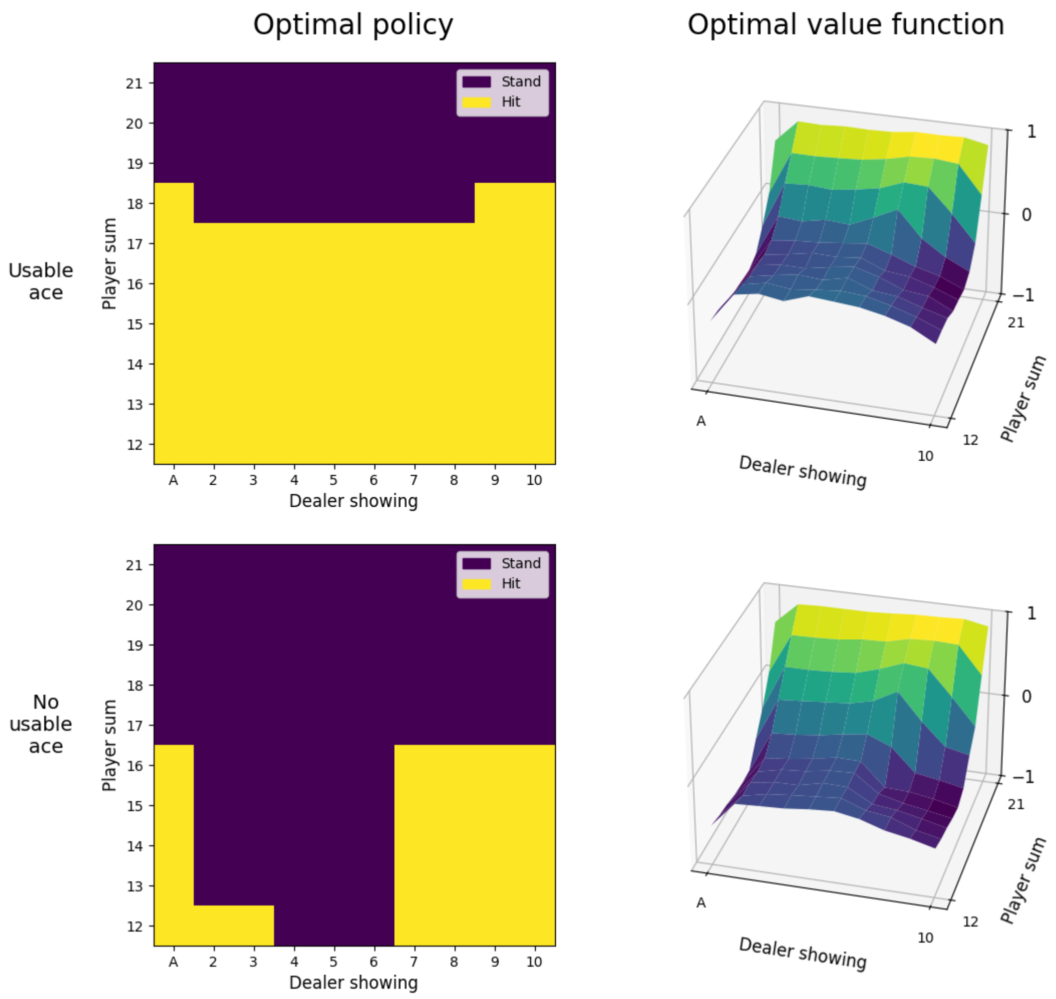


Figure 5.2: The optimal policy and state-value function for Blackjack, found using Monte Carlo with Exploring Starts.

5.2.2 Off-Policy MC Control using Importance Sampling

Off-policy learning is another more generalized method which involves using experience from a behavior policy(e.g., an exploratory ϵ-greedy policy) to learn and control a different target policy (e.g., a purely greedy policy). This is achieved using importance sampling, where returns are weighted by the ratio of probabilities of the trajectory under the target and behavior policies. It is a very general method with on-policy being its special case where the behaviour policy is same as the target policy.

Weights($\rho_{t:T-1}$) for importance sampling are given by the ratio of probability of target policy following an episode to that of behaviour policy.

$$\rho_{t:T-1} = \frac{\Pr(A_t, S_{t+1}, A_{t+1}, \cdots, S_T | S_t, A_{t:T-1} \sim \pi)}{\Pr(A_t, S_{t+1}, A_{t+1}, \cdots, S_T | S_t, A_{t:T-1} \sim b)} \tag{5.1}$$

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) p(S_{k+1}|S_k, A_k)} \tag{5.2}$$

$$\rho_{t:T-1} = \prod_{k=1}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{5.3}$$

Ordinary importance sampling is given by:

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T-1} G_t}{|\mathcal{T}(s)|}$$

where $\mathcal{T}(s)$ gives the time steps at which agent was in state s, and $\rho_{t:T-1}$ are the weights.

Weighed importance sampling also weighs the denominator resulting in:

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T-1}}$$

An essential difference between these two is that the variance of ordinary importance sampling is unbounded and can cause errors. Though both converge to the same answer weighed importance sampling is considred safer.

Case Study: The Racetrack Problem

The Racetrack problem (Sutton & Barto, Exercise 5.12) presents a more complex control challenge with a large state space (position and velocity) and stochastic actions (with some probability, the car's acceleration is zero). An Off-Policy MC Control algorithm was implemented to find an optimal policy for navigating two different tracks.

- **State Representation:** The state was defined by the car's discrete position index and its integer velocity vector (v_x, v_y).
- **Actions:** Nine actions corresponding to changing velocity components by $\{-1, 0, 1\}$.
- **Algorithm:** An off-policy MC control algorithm with weighted importance sampling was used. The agent learns a greedy target policy while behaving according to an ϵ-greedy policy. The implementation correctly breaks from updating an episode's returns as soon as a non-greedy action is encountered, as per the off-policy algorithm.

The agent was trained over a large number of episodes until its performance, measured by the success rate in completing a lap, plateaued at a high level. Figure 5.3 shows a typical learning curve. The resulting learned policies, visualized by plotting sample trajectories, are shown in Figure 5.4. The agent successfully learns an optimal "racing line," effectively managing its velocity to navigate turns. Solving it I encountered a new problem relating to bias of agent with respect to initial policy, All rewards given to the agent were negative and my initial implementation had all value function set to zero. Thus even after many iterations it would run out of the track because the explored states if much safer possessed a negative value making the agent think it is not good. Just changing the initial value function distribution to a very negative value significantly speed up the agent's learning and the results were much better.

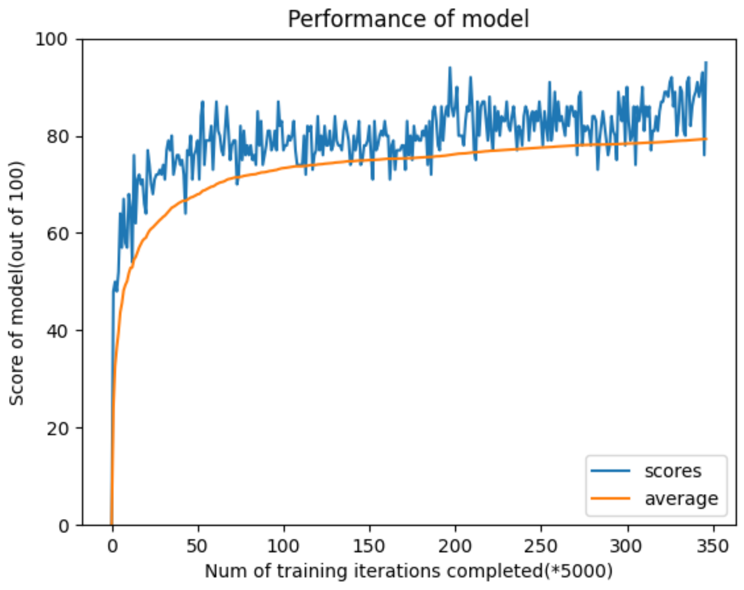


Figure 5.3: Performance curve showing the agent's success rate over training iterations for the Racetrack problem.

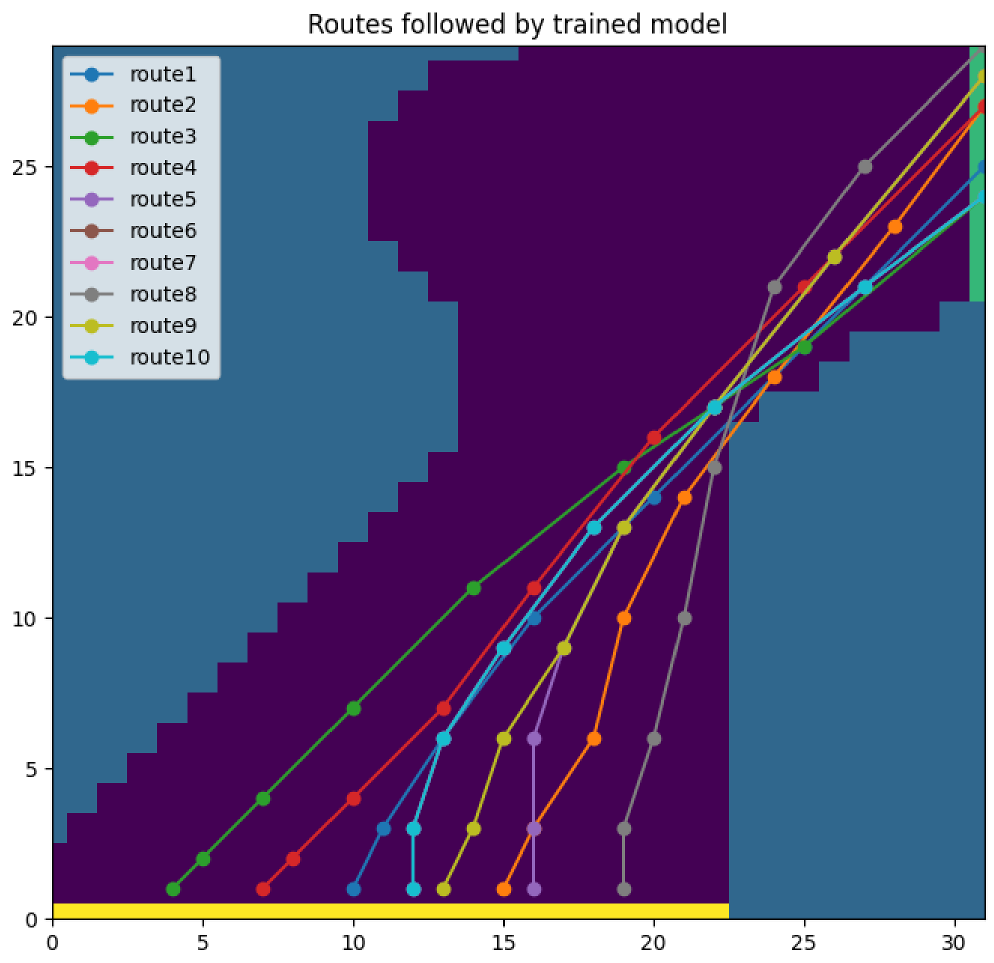


Figure 5.4: Sample trajectories of the trained policy on Racetrack Map 1. The agent learns to follow an efficient racing line.

Chapter 6

Temporal-Difference and Deep Q-Learning

After discussion of the extremes, DP methods that used data to bootstrap to the answer and Monte Carlo methods that predict each value independently, an intermediate of both must be discussed. Temporal-Difference (TD) Learning, a central idea in RL, provides a way to learn from randomly generated incomplete episodes by bootstrapping updating estimates based on other learned estimates. Monte Carlo methods wait until the end of an episode to update value estimates. This chapter introduces TD learning and its extension to large state spaces using neural networks (Deep Q-Learning), based on Sutton & Barto Chapters 6, 9, and 11 [1].

6.1 Temporal-Difference Learning

TD learning methods update value estimates after each time step. The core idea is the TD error, which is the difference between an estimated value and a better, updated estimate. This error is $G_t - V(S_t)$ in case of Monte Carlo where G_t will depend on rest of future episode, however TD uses the goal dependent on the value function itself as in the case of TD(0) the error is $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$.

6.1.1 TD(0) Prediction

The simplest TD method, TD(0), updates the value of a state S_t using the observed reward R_{t+1} and the estimated value of the next state $V(S_{t+1})$:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{6.1}$$

This allows learning to happen online, at every step, without waiting for the final outcome. Thus it presents potential for applications in real life problems.

6.1.2 TD(λ) Prediction

This is a very general TD method. TD(0) is its special case where $\lambda = 0$. It uses λ number of results in future(and one in present). Thus in case fo TD(0) we don't need the future rewards. It's update can be written as:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{\lambda} R_{t+\lambda+1} + \gamma^{\lambda+1} V(S_{t+\lambda+1}) - V(S_t)] \quad (6.2)$$

$$V(S_t) \leftarrow V(S_t) + \alpha \left[\sum_{k=0}^{\lambda} \gamma^k R_{t+k+1} + \gamma^{\lambda+1} V(S_{t+\lambda+1}) - V(S_t) \right] \quad (6.3)$$

6.1.3 SARSA and Q-Learning

For control problems, TD methods are applied to action-values, $Q(s, a)$:

- **SARSA (On-Policy):** SARSA is a wordplay with the the tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ which is used for updating $Q(S_t, A_t)$. Interestingly it also uses the next action A_{t+1} that is actually taken by the policy which is not done in general TD algorithms like TD(λ). SARSA learns the value of the policy it is following, including its exploratory actions.

- **Q-Learning (Off-Policy):** Updates $Q(S_t, A_t)$ by using the maximum Q-value of the next state, regardless of the action actually taken. The update rule is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (6.4)$$

  Q-learning directly learns the optimal action-value function, q_*, independent of the policy being followed. This makes it a powerful and widely used off-policy control algorithm.

6.2 Function Approximation and Deep Q-Networks (DQN)

For problems with vast state spaces, like Flappy Bird or Atari games, maintaining a table of Q-values is infeasible. Function approximation allows us to estimate $q(s, a)$ with a parameterized function $q(s, a, \mathbf{w})$. Deep Q-Networks (DQN) use a deep neural network with weights $\mathbf{w}$ as this function approximator.

To stabilize learning with neural networks, DQN introduced two key innovations:

1. **Experience Replay:** Transitions (s_t, a_t, r_t, s_{t+1}) are stored in a large replay memory buffer. The network is trained on mini-batches of experiences randomly sampled from this buffer, which breaks temporal correlations and improves data efficiency.

2. **Target Network:** A separate, periodically updated "target network" is used to generate the target Q-values $(R + \gamma \max_a Q_{target}(S', a'))$. This provides a more stable target for the main network to learn towards, preventing the "chasing its own tail" problem.

6.3 Case Study: DQN for Flappy Bird

As a final project to explore these advanced concepts, a DQN agent was designed and implemented from scratch to play a custom Flappy Bird game environment built with Pygame.

6.3.1 Initial Attempts and the "Survival Trap"

The initial phase of the implementation focused on a basic DQN agent. The state was represented by a simple 3-element vector: '[vertical distance to pipe gap, bird's vertical velocity, horizontal distance to pipe]'. The reward function was designed to encourage progress: +1 for each frame of survival, +100 for passing a pipe, and −1000 for death.

However, despite extensive hyperparameter tuning (learning rates from 0.01 to 0.00001, various epsilon decay schedules, and different network sizes), the agent failed to learn a successful pipe-navigating policy. The performance, shown in Figure 6.1, remained flat.

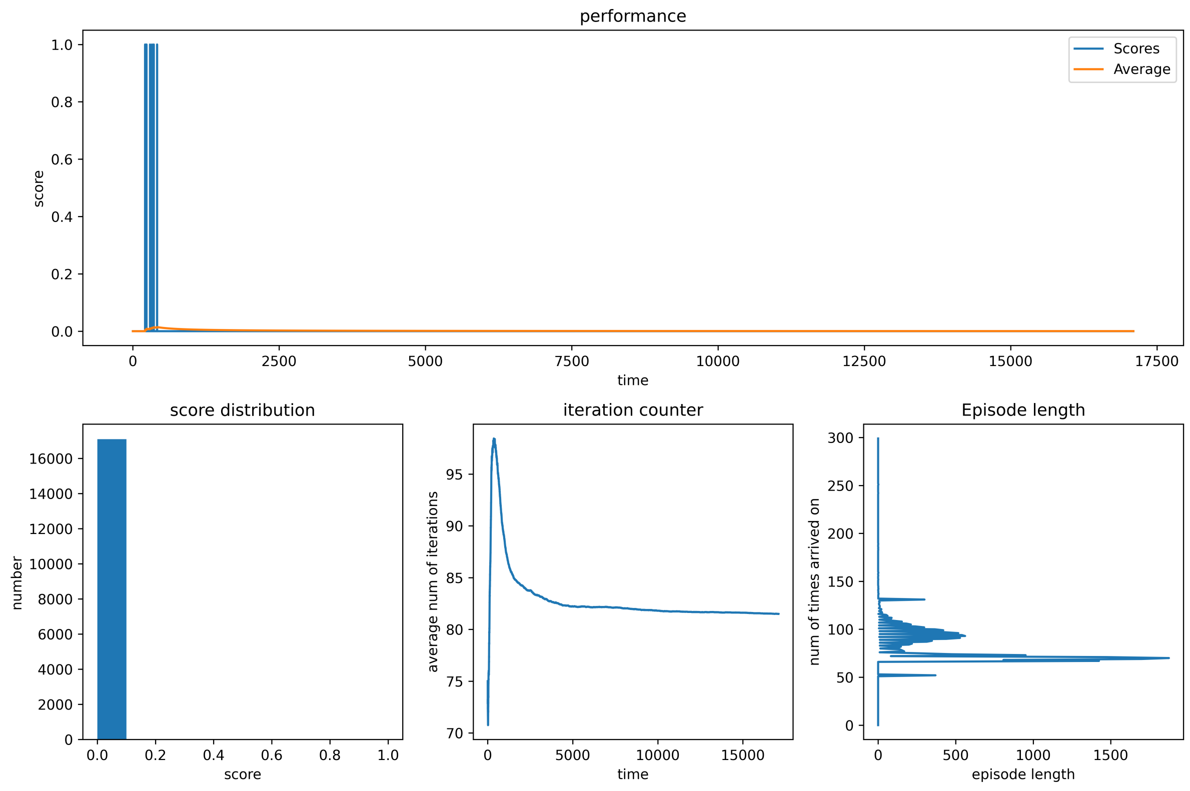


Figure 6.1: Learning curve from an initial training attempt. The agent's average score remains near zero, indicating a failure to learn the primary task of navigating pipes.

Analysis of the agent's behavior revealed it had learned a stable but suboptimal policy. The episode length plots showed that the agent consistently survived for approximately 80-85 frames. It had discovered a local optimum: by flapping minimally in the open space, it could maximize the immediate '+1' survival rewards. From the agent's perspective, the expected return of this "survival trap" was higher than the expected return of attempting a risky pipe passage, which had a high probability of incurring a massive '-1000' penalty. This is a classic challenge in reward shaping and exploration.

6.3.2 A Refined Approach: Stabilizing the Agent

Based on the analysis of the initial failures, a series of systematic improvements were implemented to stabilize training and guide the agent towards the correct goal.

- **Enriched and Normalized State Representation:** The state vector was expanded to 6 elements to provide unambiguous information: '[dist to top of gap, dist to bottom of gap, bird y_pos, dist to screen top, bird y_vel, horiz dist to pipe]‘. All values were normalized to a similar range (e.g., $[0, 1]$ or $[-1, 1]$) to improve neural network training stability.

- **Reward Re-engineering:** The reward function was reshaped to de-incentivize mere survival. The reward for living one frame was reduced to a near-negligible ‘+0.1‘, while the penalty for death was kept high (‘-50‘ to ‘-100‘) and the reward for passing a pipe was made very significant (‘+1000‘). This made pipe passage the only meaningful way for the agent to achieve a high cumulative reward.

- **Advanced DQN Architecture:** Two crucial improvements to the core algorithm were implemented:

  1. **Target Network:** A separate target network was added to provide stable Bellman targets, as described in the original DQN paper. This network's weights are updated to match the policy network's weights every 10 episodes.
  2. **Double DQN:** The training logic was upgraded to implement the Double DQN update (as seen in Listing 6.1). This technique uses the policy network to select the best next action, but uses the target network to evaluate that action's Q-value. This helps to reduce the overestimation of Q-values, a known issue in standard Q-learning.

- **Hyperparameter Optimization and Stabilization:** After experimentation, more robust hyperparameters were chosen (‘LR=0.001‘, ‘HIDDEN_SIZE=128‘, ‘BATCH_SIZE=128‘). Furthermore, "Gradient Clipping" was introduced during the optimization step to prevent excessively large gradient updates, further stabilizing the training process.

```python
# In QLearning.train():
# prediction: Q-values for current states from the main policy network
prediction: torch.Tensor = self.model(state)
target: torch.Tensor = prediction.clone()

with torch.no_grad():
    # 1. Use policy_net to CHOOSE the best next action
    best_next_actions = self.model(new_state).argmax(dim=1).unsqueeze(1)
    # 2. Use target_net to EVALUATE that chosen action's value
    Q_new_next_state = self.target_model(new_state).gather(1, best_next_actions)

# Compute target: R if done, else R + gamma * Q_target(S', argmax_a Q_policy(S',a))
done_bool = done.bool()
Q_target_values = reward.clone()
Q_new_next_state = Q_new_next_state.reshape((-1, 1))
Q_target_values += self.gamma * Q_new_next_state * (~done_bool)

# Update the Q-value for the action that was actually taken
action_indices = action.long().squeeze()
target[torch.arange(target.size(0)), action_indices] = Q_target_values.squeeze()

```

```python
# Compute loss and perform backpropagation
self.optimizer.zero_grad()
loss = self.criterion(target, prediction)
loss.backward()
torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0) #
    Gradient Clipping
self.optimizer.step()
```

Listing 6.1: Vectorized Double DQN Training Logic

6.3.3 Successful Training and Final Results

With the refined approach, the agent was trained for approximately 160,000 episodes. The results, logged and analyzed from this extensive run, demonstrate successful learning.

Figure 6.2 shows the average episode length over time. It illustrates a clear narrative of learning: an initial phase of volatile exploration, a long period of learning a stable but suboptimal survival strategy, followed by a distinct "breakthrough" after approximately 130,000 episodes where the agent discovers the pipe-passing strategy, leading to a dramatic increase in average survival time.

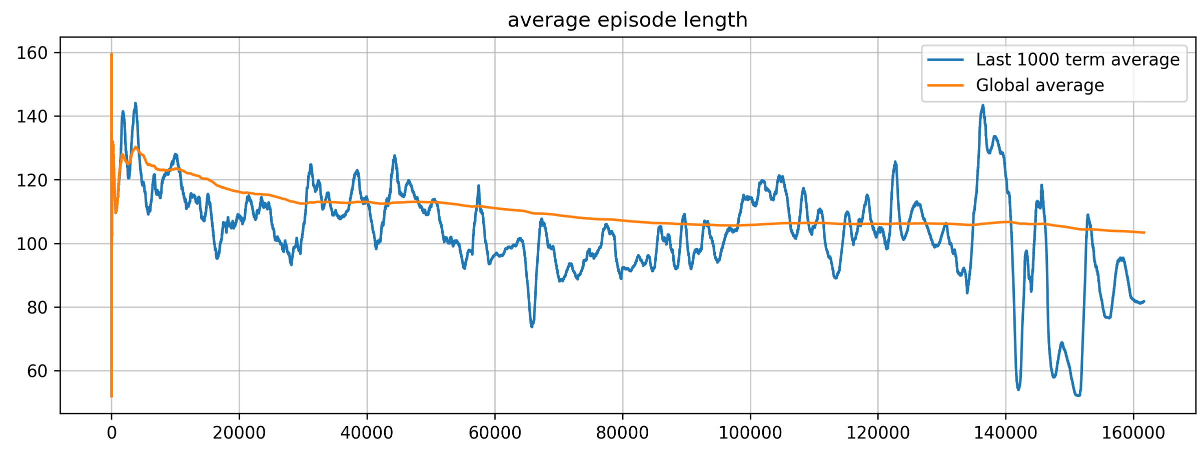


Figure 6.2: The average episode length (moving average of last 1000 episodes and global average) over the course of training. The sharp increase after 130k episodes indicates a breakthrough in learning.

The score per episode, shown in Figure 6.3, confirms this breakthrough. While individual scores remain noisy due to the stochastic nature of the task, a clear emergence of non-zero scores appears in the latter stages of training, pulling the moving average upwards. The episode length distribution (Figure 6.4) further highlights this shift, showing a significant increase in the frequency of longer episodes after the learning breakthrough.

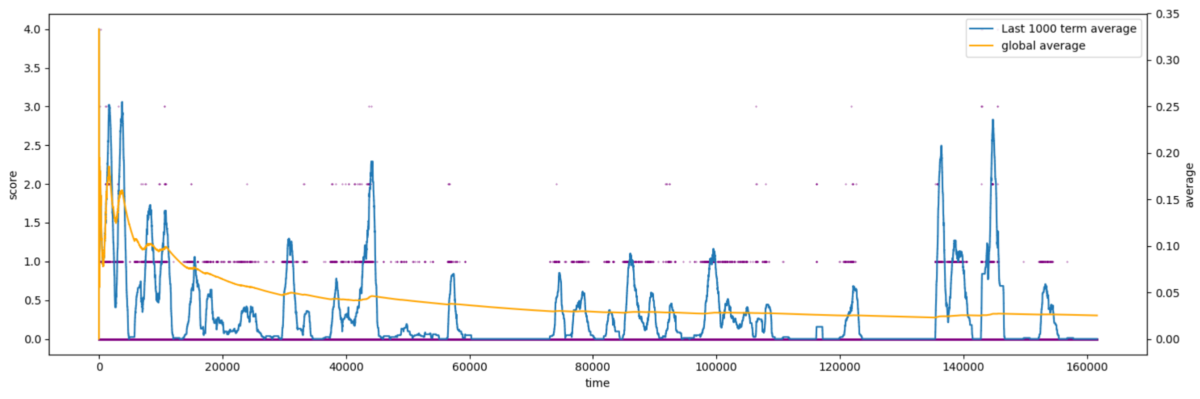


Figure 6.3: Score per episode over time. The purple scatter plot shows individual scores, while the lines show moving and global averages. The emergence of high scores indicates successful learning.

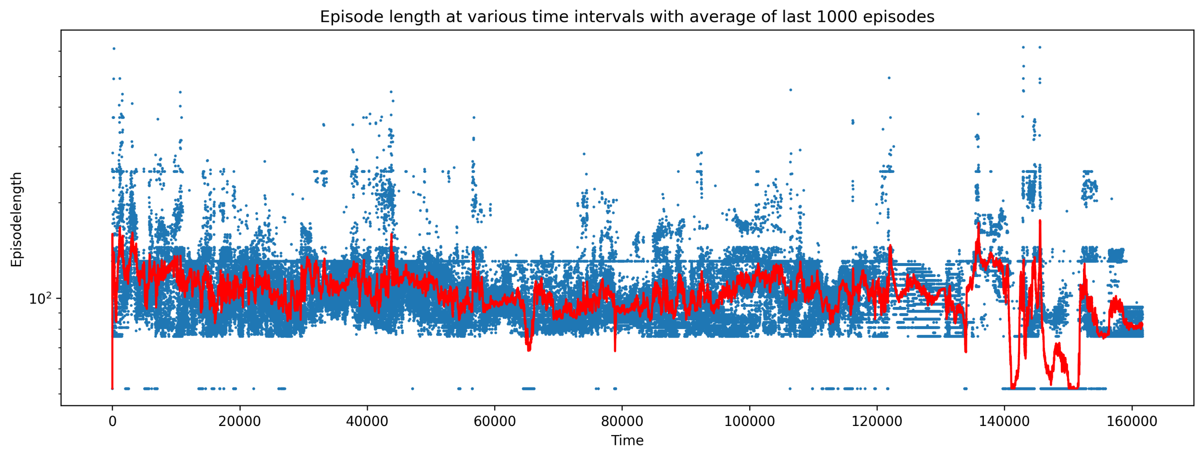


Figure 6.4: Episode length distribution (y-axis is logarithmic). A clear shift towards longer episodes is visible in the later stages of training, corresponding to the agent learning to navigate pipes.

Chapter 7

Challenges, Learnings, and Conclusion

7.1 Challenges Encountered

Throughout this initial phase, several challenges were encountered. Initially, grappling with the dense formalism of texts like Baier & Katoen(Which turned out to be nightmarish during initial period) for certain topics required significant effort, prompting a strategy to supplement with more intuitively explained resources like Sutton & Barto and Huth & Ryan. The implementation of Dynamic Programming algorithms, particularly for Jack's Car Rental, presented considerable debugging challenges. Moving from a conceptual understanding of the nested expectations to an efficient vectorized NumPy implementation was a non-trivial task, involving several iterations of code design to manage the multi-dimensional state and probability arrays. Ensuring correct convergence for both Policy and Value Iteration also required careful attention to termination conditions and update rules. The optimistic horizon problem in the ractrack problem required significant debugging to just identify, and the final result taught how a small change can affect entire learning process of an agent. The Final project held the most challenges and was in a sense not seemingly practical but experimental. Trying many many hyper parameters and debugging might not have made the agent reach a super human performance(or even sub human). I understood many more problems possible in deep Q learning training like exploding gradient, normalized input, dying ReLU, disappearing gradients in sigmoid, instability caused by updating single network without a target network etc. Balancing the demanding theoretical study and implementations for this SoS project alongside commitments to other ongoing projects (Blockchain Arena, unofficial compiler exploration, and DSA Bootcamp) necessitated strict time management and prioritization.

7.2 Key Learnings and Insights

This project has been immensely rewarding in terms of learning. A primary insight is the profound difference between implementing algorithms by following tutorials and truly understanding the underlying theory. Delving into foundational formal logic, automata, and then the formalisms of MCs and MDPs has provided a much deeper appreciation for *why* RL algorithms work. The importance of fine tuning hyperparameters was also evident from the various experiments. Thus the power of Dynamic Programming for solving MDPs with known models became evident through practical implementation; it highlighted the importance of precise problem formulation, including careful definition of states, actions, rewards, and transition dynamics.

The k-Armed Bandit experiments underscored the nuances of the exploration-exploitation trade-off and how same problem can be solved by different and yet same strategies. Implementing Policy and Value Iteration for the GridWorld provided a clear, visual understanding of how value functions and policies evolve. It gave a clear picture of RL algorithm tackling a task scalable at more practical levels. Tackling Jack's Car Rental was particularly instructive in managing larger state spaces and complex stochastic dynamics, and it drove home the practical benefits of vectorization (using NumPy) for performance. It taught the importance of optimizing the approach to same method of solving. The Gambler's Problem illustrated how optimal policies can sometimes be counter-intuitive and highly sensitive to problem parameters. A key recurring theme has been the central role of Bellman equations in structuring solutions.

Moving on from dynamic programming the model free learning introducing a new concept altogether to train an agent without proper knowledge of the system. It came to be the most challenging yet teaching part of the entire project. It included nearly debugging and try to prevent many big problems that are encountered in research related to RL. The importance of initial value function bias was taught by the racetrack, and the biggest take away I got was from the final project: Any small change can cause an avalanche of changes in the learning pattern of an agent.

7.3 Conclusion and Transition to Future Work

The Summer of Science 2025 has been an incredibly intensive and rewarding journey into the theory and practice of Reinforcement Learning. Starting from the formalisms of logic and automata, progressing through Markov Chains and the core principles of MDPs, and culminating in the implementation of algorithms ranging from Dynamic Programming to model-free Monte Carlo and Deep Q-Learning methods, this project has provided a comprehensive, hands-on education. The successful development of solutions for classic problems like Jack's Car Rental and Racetrack, and the construction of a DQN agent for Flappy Bird, have solidified theoretical concepts into practical skills. My future path in this domain will involve exploring more advanced topics such as Proximal Policy Optimization (PPO), as recommended by my mentor, and investigating areas like Safe RL and Multi-Agent systems.

Appendix A

Rules for natural deduction

Natural deduction is a proof system for propositional logic which both sound and complete. It allows us to prove many formulas using a basic set of premise statements. All the defined rules for natural deduction are given in Table A.1. These rules define the possible steps in a natural deduction proof. Here is an example:

Proof of $p \wedge \neg q \to r, \neg r, p \vdash q$

1	$p \wedge \neg q \to r$	premise
2	$\neg r$	premise
3	p	premise
4	$\neg q$	assumption
5	$p \wedge \neg q$	$\wedge i_{3,4}$
6	r	$\to e_{1,5}$
7	$\bot$	$\neg e_{6,2}$
8	$\neg\neg q$	$\neg i_{4-7}$
9	q	$\neg\neg e_8$

Some useful derived rules:

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\text{MT(Modus Tollens)}$$

$$\frac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\text{PBC(Proof by Contradiction)}$$

$$\frac{}{\phi \vee \neg\phi}\text{LEM(law of excluded middle)}$$

Table A.1: Rules for Propositional Deduction

Connective	**Introduction**	**Elimination**
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi} \wedge i$	$\dfrac{\phi \wedge \psi}{\phi} \wedge e_1 \quad \dfrac{\phi \wedge \psi}{\psi} \wedge e_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi} \vee i_1 \quad \dfrac{\psi}{\phi \vee \psi} \vee i_2$	$\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi \\ \vdots \\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi \\ \vdots \\ \chi\end{array}}}{\chi} \vee e$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}}{\phi \rightarrow \psi} \rightarrow i$	$\dfrac{\phi \rightarrow \psi \quad \phi}{\psi} \rightarrow e$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \bot\end{array}}}{\neg\phi} \neg i$	$\dfrac{\neg\phi \quad \phi}{\bot} \neg e$
$\bot$	(no introduction rule for $\bot$)	$\dfrac{\bot}{\phi} \bot e$
$\neg\neg$	$\dfrac{\phi}{\neg\neg\phi} \neg\neg i$	$\dfrac{\neg\neg\phi}{\phi} \neg\neg e$

References

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

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[3] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, second. Cambridge University Press, 2004.

[4] A. Singh, *Mdps and their applications in ai - sos 2025 implementations*, https://github.com/Aadeshveer/MDPs_and_their_applications_in_AI_SOS_2025, 2025.