Table of Contents

1. **Section 1** - Lecture #1 - Introduction to RL #1 - May 11, 2024
   (a) **Section 1-1** - Notes from Textbook Readings - Chapter #1: Introduction
   (b) **Section 1-2** - Notes from Textbook Readings - Chapter #2: Multi-armed Bandits

2. **Section 2** - Lecture #2 - Introduction to RL #2 - May 18, 2024
   (a) **Section 2-1** - Notes from Textbook Readings - Chapter #3: Finite Markov Decision Processes

3. **Section 3** - Lecture #3 - Model-based RL: Dynamic Programming - May 25, 2024
   (a) **Section 3-1** - Notes from Textbook Readings - Chapter #4: Dynamic Programming

4. **Section 4** - Lecture #4 - Model-free RL #1: Monte Carlo Methods - June 1, 2024
   (a) **Section 4-1** - Notes from Textbook Readings - Chapter #5: Monte Carlo Methods

5. **Section 5** - Lecture #5 - Model-free RL #2: Temporal-Difference Learning - June 8, 2024
   (a) **Section 5-1** - Notes from Textbook Readings - Chapter #6: Temporal-Difference Learning

6. **Section 6** - Lecture #6 - TD(n) - June 15, 2024
   (a) **Section 6-1** - Notes from Textbook Readings - Chapter #7: n-step Bootstrapping

7. **Section 7** - Lecture #7 - Function Approximation (Supervised Learning) - July 6, 2024
   (a) **Section 7-1** - Notes from Textbook Readings - Chapter #9: On-policy Prediction with Approximation

8. **Section 8** - Lecture #8 - Prediction and Control with F.Approx; DQN - July 13, 2024
   (a) **Section 8-1** - Notes from Textbook Readings - Chapter #10: On-policy Control with Approximation
   (b) **Section 8-2** - Notes from Textbook Readings - Human-level Video Game Play

9. **Section 9** - Lecture #9 - Policy Gradient; RL-Human Feedback; ChatGPT - July 20, 2024
   (a) **Section 9-1** - Notes from Textbook Readings - Chapter #13: Policy Gradient Methods

10. **Section 10** - Lecture #10 - Integrated Model-free and Model-based RL; Planning; Mu-zero - July 27, 2024
    (a) **Section 10-1** - Notes from Textbook Readings - Chapter #8: Planning and Learning with Tabular Methods
    (b) **Section 10-2** - Notes from Textbook Readings - Mastering the Game of Go

Section: 1
Introduction to RL #1

AI, ML and RL; Data Science versus Control; RL problem setup; Agent versus Environment; Applications; Anatomy of an RL solution; Environment models; Terminology

Lecture # 1 Date: May 11, 2024.

AI, ML, and RL; Data Science versus Control

Computer Science:

- Traditional tasks: Sorting, Searching, etc.
- Modern advancements:
  - Incorporates Probability and Models
  - Extends to Neural Networks

Control Theory:

- Model-based: Closed loop Control Systems
- No Model: Stochastic Control
- Dynamic Programming: Extends to Reinforcement Learning (RL)

Machine Learning (ML) Question:

- **What to learn and how to learn?**
- This question often remains unanswered in Supervised and Unsupervised learning methods.

Phases of Reinforcement Learning (RL):

- **Part 1: Fundamentals**
  - Begins with Dynamic Programming and leads to other fundamental methods.
- **Part 2: Integrating Neural Networks (NN)**
  - Combines fundamental techniques with NN to address "what and how to learn."
  - Covers Deep Q-Networks (DQN), Policy Gradient (PG), Reinforcement Learning from Human Feedback (RLHF), and more.
- **Part 3: Hybrid Techniques**
  - Combines traditional computer science techniques with newly learned RL techniques.
  - Example: AlphaZero

Reinforcement Learning (RL) as a subset of AI

- **RL is AI**

Key Terms:

- **Machine (Agent)**: The entity where AI algorithms operate.
- **Environment**: The context or world where the problem is modeled.
- **Actions (A)**: The set of discrete actions the agent can take.
- **State (S)**: The sensory feedback from the environment; represents the state space.
- **Reward**: A real number representing aspects of the state that relate to the goal.
  - Avoid over-designing the reward signal.
  - Example: Winning = 100, Stalemate = 50, Loss = -100
- The machine learns based on how the reward is designed.

$$A : \text{Action} \quad A \in \{A^{(1)}, A^{(2)}, \ldots, A^{(K)}\} = \mathcal{A} \quad \text{(Action Space)} \tag{1}$$

- K: Number of discrete actions

$$S : \text{State Feedback} \quad S \in \{S^{(1)}, S^{(2)}, \ldots, S^{(N)}\} = \mathcal{S} \quad \text{(State Space)} \tag{2}$$

- N: Number of states

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2 \ldots \tag{3}$$

$$S_0(E) \xrightarrow{A_0(M)} R_1(E) \xrightarrow{S_1(E)} \xrightarrow{A_1(M)} \ldots \xrightarrow{S_i} \xrightarrow{A_i} R_{i+1} \xrightarrow{S_{i+1}} \xrightarrow{A_{i+1}} \ldots \tag{4}$$

Temporal Interaction:

- Autonomous systems operate dynamically over time.
- Sequences of state, action, and reward over time are referred to as Episodes, Traces, or Play-outs.

AI Problem: Action Selection

- At time t_i, given state S_i, what action A_i should the agent select from the possible set of actions?
- **The goal in RL: Select actions to maximize the sequence of future rewards (sum of rewards)**.
- Learning involves understanding how to predict and maximize future rewards.

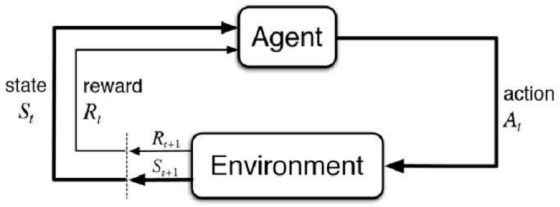

Figure 1: Basic RL setup

Value Functions and Equations:

- Long-term and short-term rewards are managed using a discount factor (γ).
- **Value Function:** Represents the expected return (reward) from a state, following a policy π.
- Equation for return and Value Functions:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{5}$$

- $\gamma \in (0, 1)$: Discount factor

$$V_\pi(s) = E_\pi[G_t \mid S_t = s] \tag{6}$$

$$q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a] \tag{7}$$

- The formulation of the AI control problem involves an agent, M, interacting with an environment, E, by dynamically issuing actions, A_t, based on environment state feedback, S_t, and environment reward, R_t. The agent selects actions. The goal of the agent is to maximize its return, G_t.

Section: 1-1
Notes from Textbook Readings

Chapter #1: Introduction

- Reinforcement learning involves mapping situations to actions to maximize a numerical reward signal.
- The learner discovers which actions yield the most reward through trial-and-error rather than being told which actions to take.
- Actions can affect not only the immediate reward but also subsequent situations and all future rewards, highlighting the importance of trial-and-error and delayed reward.
- Reinforcement learning is simultaneously a problem, a class of solution methods, and a field of study, with a crucial distinction between problems and solution methods.
- The problem of reinforcement learning is formalized using ideas from dynamical systems theory, specifically as the optimal control of incompletely-known Markov decision processes.
- Reinforcement learning differs from supervised learning, which relies on labeled examples provided by an external supervisor, as it requires the agent to learn from its own experience in interactive problems.
- Reinforcement learning differs from unsupervised learning, which focuses on finding hidden structures in unlabeled data.
- Unlike unsupervised learning, reinforcement learning aims to maximize a reward signal rather than uncover hidden structures.
- Reinforcement learning is considered a third machine learning paradigm, alongside supervised and unsupervised learning.
- A unique challenge in reinforcement learning is the trade-off between exploration (trying new actions) and exploitation (using known effective actions).
- The agent must balance exploration and exploitation to maximize rewards, trying various actions and progressively favoring the best ones.
- Reinforcement learning considers the entire problem of a goal-directed agent interacting with an uncertain environment.
- Unlike other approaches that focus on subproblems, reinforcement learning starts with a complete, interactive, goal-seeking agent.
- Reinforcement learning agents have explicit goals, can sense their environments, and choose actions to influence their environments.
- The approach involves planning and real-time action selection, addressing how environment models are acquired and improved.
- A complete, interactive, goal-seeking agent can be part of a larger system, interacting directly with the system and indirectly with the system's environment.
- Modern reinforcement learning integrates with statistics, optimization, psychology, and neuroscience, benefiting from and contributing to these fields.

RL Examples

- Reinforcement learning can be understood through diverse examples such as a master chess player, an adaptive controller for a petroleum refinery, a gazelle calf learning to walk, a mobile robot deciding on tasks, and a person preparing breakfast.
- These examples involve an active decision-making agent interacting with an environment to achieve a goal despite uncertainty.
- The agent's actions affect the future state of the environment, influencing subsequent actions and opportunities, requiring foresight and planning.
- Effects of actions are unpredictable, necessitating frequent monitoring and appropriate reactions from the agent.
- Goals are explicit, allowing the agent to judge progress based on direct sensory feedback (e.g., chess player winning, refinery production, robot battery level).
- Agents can use their experiences to improve performance over time, refining their decision-making and actions based on past outcomes.

RL Elements

- Four main subelements of a reinforcement learning system: policy, reward signal, value function, and optionally, a model of the environment.
- A policy defines the agent's behavior at a given time, mapping perceived states of the environment to actions; it can be a simple function, lookup table, or involve extensive computation.
- Policies can be stochastic, specifying probabilities for each action in given states.
- The reward signal defines the goal, providing immediate feedback to maximize long-term rewards; it is the primary basis for modifying the policy.
- A value function specifies the long-term desirability of states, estimating the total expected reward from each state, which guides decision-making.
- Values predict long-term rewards, unlike immediate rewards which are directly received; values are crucial for making decisions that maximize long-term rewards.
- Value estimation is a core component of reinforcement learning algorithms, as values are harder to determine than immediate rewards and require efficient methods for estimation.
- Some reinforcement learning systems use a model of the environment for planning, predicting future states and rewards based on current states and actions, which helps in decision-making and planning future actions.

Scope and Limitations

- The concept of state is central to reinforcement learning, serving as input to the policy, value function, and model, and representing the environment's status at a given time.
- The state signal is assumed to be produced by a preprocessing system in the agent's environment, and this book focuses on decision-making based on available state signals rather than on designing the state signal.
- Most reinforcement learning methods are structured around estimating value functions, although some methods like genetic algorithms and simulated annealing do not estimate value functions.
- Evolutionary methods apply multiple static policies, selecting and evolving the best-performing ones over generations without estimating value functions.
- Evolutionary methods can be effective if the policy space is small, structured for easy discovery of good policies, or if ample time is available for the search.
- Evolutionary methods are advantageous when the learning agent cannot fully sense the state of its environment.
- Reinforcement learning methods that learn while interacting with the environment can be more efficient than evolutionary methods, as they utilize the structure of the reinforcement learning problem and details of individual interactions.
- Evolutionary methods do not leverage the relationship between states and actions or track the states and actions experienced by individuals, which can lead to less efficient searches compared to methods that do.

Tic-Tac-Toe Example

- The example of tic-tac-toe is used to illustrate and contrast reinforcement learning with other approaches.
- In tic-tac-toe, two players take turns placing Xs and Os on a three-by-three board, aiming to get three in a row.
- Classical techniques like min-max and dynamic programming are not suitable for solving this problem as they require assumptions or complete specifications about the opponent's behavior.
- Reinforcement learning methods estimate an opponent's behavior through experience by playing many games and learning from the outcomes.
- Evolutionary methods directly search the space of possible policies, evaluating each policy's winning probability by playing multiple games against the opponent.
- Value function methods estimate the probability of winning from each state, updating these estimates through temporal-difference learning.
- A reinforcement learning agent for tic-tac-toe would start with initial values for each state, play many games, and update state values based on the outcomes to learn optimal moves.

- The method converges to the true probabilities of winning from each state and learns to make optimal moves against the opponent.
- Reinforcement learning emphasizes learning while interacting with an environment, planning for delayed effects of actions, and can achieve planning-like results without explicit models or search.
- Reinforcement learning is applicable beyond two-person games and episodic problems, extending to continuous-time problems and those with very large or infinite state sets.
- Reinforcement learning can generalize from past experiences using techniques like artificial neural networks, enabling it to handle large state spaces effectively.
- Prior knowledge can be incorporated into reinforcement learning, and the approach can handle partially observable states or states that appear the same to the learner.
- Reinforcement learning can be applied with or without models of the environment, and model-free methods can be advantageous when accurate models are difficult to obtain.

Tic-Tac-Toe Moves Sequence

- **Solid Black Lines:** Represent the moves taken during the game.
- **Dashed Lines:** Indicate moves considered but not made by the reinforcement learning player.
- **Asterisk (*):** Shows the move currently estimated to be the best.
- **Exploratory Move:** Our second move, taken despite another move leading to a higher estimated value. Such moves do not result in learning but help explore the state space.
- **Red Arrows:** Illustrate how estimated values are updated from later game states to earlier ones.

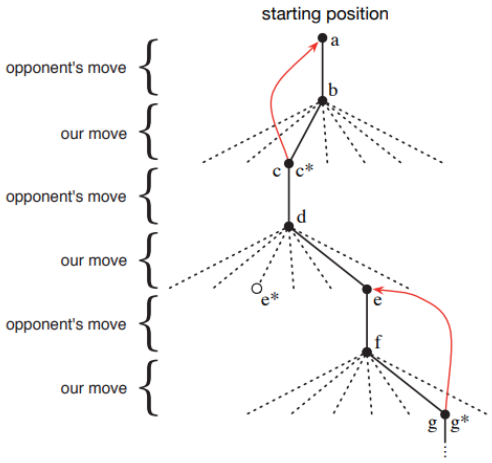

Figure 2: Tic-Tac-Toe Moves Sequence

- The value update rule for a state S_t can be expressed as:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[V(S_{t+1}) - V(S_t) \right] \tag{8}$$

  where:

  - $V(S_t)$ is the estimated value of the state before the move.
  - $V(S_{t+1})$ is the estimated value of the state after the move.
  - α is a small positive fraction called the step-size parameter, which influences the rate of learning.

Section: 1-2
Notes from Textbook Readings

Chapter #2: Multi-armed Bandits

- The key feature that distinguishes reinforcement learning from other types of learning is its use of **evaluative feedback** rather than **instructive feedback**.
  - Evaluative feedback assesses the quality of actions taken but does not specify whether an action was the best or the worst.
  - Instructive feedback, on the other hand, provides explicit instructions on the correct action to take, regardless of the action actually performed.
- This distinction necessitates **active exploration** in reinforcement learning, where an agent actively searches for better behavior through trial and error.
- Pure evaluative feedback, which is the focus of this chapter, evaluates actions based on their outcomes but does not guide the agent towards the best actions directly.
- Pure instructive feedback, central to **supervised learning**, provides direct guidance by indicating the correct actions to take, independent of the agent's actions.
- This chapter will focus on a simplified, nonassociative setting of reinforcement learning to highlight how evaluative feedback differs from, and can be combined with, instructive feedback.
- Simple version of the **k-armed bandit problem** as a case study to introduce fundamental learning methods.
- The chapter concludes by discussing how the k-armed bandit problem can evolve into an **associative** problem, where the best action depends on the situation, thereby bringing us closer to the full reinforcement learning problem.

A Simple Bandit Algorithm

- **Action-Value Methods:** Action-value methods estimate action values as sample averages of observed rewards.
- **Computational Efficiency:** To compute these averages efficiently with constant memory and per-time-step computation, incremental formulas are used.
- **Notation:**
  - Let R_i denote the reward received after the i-th selection of an action.
  - Let Q_n denote the estimate of the action value after $n-1$ selections, simplified to Q_n.
- **Average Computation:**
  - The average action value after n selections is given by:

$$Q_n = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1} \tag{9}$$

A simple bandit algorithm

Initialize, for $a = 1$ to k:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:
$A \leftarrow \begin{cases} \operatorname{argmax}_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$

Figure 3: Simple Bandit Algorithm

  - This approach requires storing all rewards and computing their sum, which increases memory and computation as more rewards are collected.

- **Incremental Update Formula:**
  - To avoid storing all rewards, an incremental update formula is used:

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n] \tag{10}$$

  - This formula allows updating the average with constant memory and computation, where Q_n is the old estimate and R_n is the new reward.
  - For $n = 1$, this simplifies to $Q_2 = R_1$.

- **General Update Rule:**
  - The general update rule for estimates is:

$$\text{NewEstimate} = \text{OldEstimate} + \text{StepSize} \times [\text{Target} - \text{OldEstimate}]$$

  - (Target − OldEstimate) represents the error in the estimate, and the step-size parameter controls the rate of adjustment.
  - In the context of the bandit problem, the target is the most recent reward.

Handling Nonstationary Bandit Problems

- **Nonstationary Bandit Problems:**
  - In nonstationary problems, reward probabilities change over time.
  - It is beneficial to weight recent rewards more heavily than past rewards.

- **Weighted Average with Constant Step-Size:**
  - The update rule for a weighted average is:

$$Q_{n+1} = Q_n + \alpha(R_n - Q_n) \tag{11}$$

    where $\alpha \in (0, 1]$ is a constant step-size parameter.

- The updated estimate Q_{n+1} can be expressed as:

$$Q_{n+1} = Q_n + \alpha \left(R_n - Q_n \right) \tag{12}$$

- This method gives more weight to recent rewards and less weight to past rewards, as follows:

$$Q_{n+1} = \alpha R_n + (1 - \alpha) Q_n \tag{13}$$

- This is known as an *exponential recency-weighted average*.

- **Variable Step-Size Parameter:**
  - Sometimes the step-size parameter $\alpha_n(a)$ varies with each reward:

$$Q_{n+1}(a) = Q_n(a) + \alpha_n(a)(R_n - Q_n(a)) \tag{14}$$

  - For convergence, the following conditions are required:
    - $\sum_{n=1}^{\infty} \alpha_n(a) = \infty$
    - $\sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$
  - These conditions ensure that steps are initially large enough to overcome fluctuations and become small enough for convergence.
  - For the constant step-size parameter case, $\alpha_n(a) = \alpha$:
    - The first condition is satisfied.
    - The second condition is not satisfied, implying estimates never fully converge but continue to adapt to new rewards.

- **Considerations:**
  - Constant step-size methods are often used in nonstationary environments because they allow estimates to adapt continuously.
  - Variable step-size methods, while theoretically useful, may converge slowly or require extensive tuning.

Section: 2
Introduction to RL #2

Recap with further details; Markov Decision and Reward Processes

Lecture # 2 Date: May 18, 2024

AI Problem

- **Agent (Machine) Interaction:**
  - The agent interacts with the environment by taking actions. These actions can vary from simple movements to complex decisions, depending on the nature of the environment and the task.
- **Environment Response:**
  - The environment responds to the agent's actions through various signals, including:
    - **Senses:** Inputs that provide information about the current state of the environment.
    - **States:** Specific configurations or conditions of the environment at a given time.
    - **Feedback:** Responses from the environment that indicate the result of the agent's actions.
    - **Reward:** A scalar signal that measures the desirability of the agent's actions, helping guide the agent toward better behavior.
- **Defining the Reward Signal:**
  - The reward signal is defined by examining the environment and determining how to score the agent's performance. This involves:
    - **Observation:** Analyzing the state of the environment to identify what constitutes good or bad performance.
    - **Scoring System:** Assigning numerical values (rewards) to different states or actions based on their desirability or utility to the agent.
- **Reward as a Real Number:**
  - The reward is typically represented as a real number, quantifying the immediate benefit or cost of the agent's actions. This value is derived from the environment's state and is used to evaluate and guide the agent's behavior.

Reward Examples

- **Goal to Avoid Loss:**
  - In scenarios where the objective is to avoid losing, such as in certain strategic games, the reward structure might be:
    - **Winning:** Positive reward values.
    - **Stalemate:** Neutral or zero reward.

        * **Losing:** Negative reward values.

- **Goal to Win at All Costs:**
  - When the objective is to win regardless of other factors, the reward structure might be:
    * **Winning:** Positive reward values.
    * **Stalemate:** Negative reward values.
    * **Losing:** Negative reward values.
- **Penalization for Excess Time:**
  - In some cases, an additional penalty might be applied for taking too much time to achieve a goal. This could be reflected in the reward structure by subtracting a penalty proportional to the time taken.
- **Reward Representation:**
  - **R = How Good or Bad:** The reward R quantifies how good or bad an action or outcome is, guiding the agent's decision-making process.
  - **R = Engineer Defined:** The specific reward values and structure are defined by engineers based on the problem requirements and objectives.

AI Problem and Reinforcement Learning

- **AI Problem:**
  - At a given time, what action should be selected from a possible set of actions? This is known as the *Action Selection Problem*.
- **Reinforcement Learning Answer:**
  - Select the action that maximizes the expected future rewards.
- **Discount Factor:**
  - The discount factor, denoted by γ, is a constant used to reflect the value of future rewards in reinforcement learning.
  - γ is a hyperparameter that determines how much importance is given to rewards in the future relative to rewards received immediately.
  - It controls how far into the future the solver considers when evaluating the value of an action or state.

Reward with Discount Factor

The reward at time t with a discount factor γ is defined as:

$$R_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{15}$$

where:

- R_t is the total expected reward starting from time t.
- R_{t+k} is the reward received at time $t + k$.
- γ (where $0 \leq \gamma \leq 1$) is the discount factor.

Topic: How to Compute the Means of Action Selection When We Have a Model

- **Mean of Action Selection Problem:**
  - The mean of the action selection problem is represented by the *Policy of the Machine* (π).
  - The policy π dictates the action selection strategy that the machine follows.
- **Model of the Environment:**
  - The model of the environment, denoted by p, provides a representation of how the environment responds to actions.
  - It includes the transition probabilities and the reward function based on actions taken.
- **Deterministic Model:**
  - In a deterministic model, the environment's response is predictable and follows a specific control theory framework.
  - This implies that given a specific action, the outcome is deterministic and can be described precisely by the model.

Policy (π)

- **Definition:**
  - A policy (π) is a property of an agent that defines how the agent selects actions.
  - It represents the action selection mechanism used by the agent.
- **Types of Policies:**
  - **At Best:** In an ideal scenario, the policy is governed by a stochastic model.
  - **Usually:** In practical scenarios, the policy may not have a model.
- **Representation of Policy:**
  - The policy can be represented as $\pi(A|S)$, which is the probability distribution over actions given a state.
  - It can be visualized as a table (data structure) with N rows and K columns, where N represents the states and K represents the actions.
  - The entries in this table are real numbers within the range $[0, 1]$, and each row of the table sums to 1, ensuring it represents a valid probability distribution.

State	Action 1	Action 2	Action 3
A	0.2	0.3	0.5
B	0.4	0.4	0.2
C	0.1	0.2	0.7

Table 1: Policy Table Example

Model

The model of the environment is a property that describes the probabilities of transitions and rewards given a state and action. Specifically, it can be denoted as:

$$p(R, S' \mid S, A) \tag{16}$$

where:

- R is the reward received after taking action A in state S.
- S' is the state that results from taking action A in state S.
- $p(R, S' \mid S, A)$ is the probability distribution over the reward and the next state S', given the current state S and action A.

What Can We Do with p?

- Given a current state S and an action A, use the model p to predict the distribution of possible rewards R and the next state S'.
- Compute the expected reward and the expected next state for a given policy π using the model.
- Use the model to perform planning and optimization by simulating future states and rewards.
- Calculate the expected reward and value of states and actions.
- Generate simulated experiences using the model to train reinforcement learning algorithms

Markovian Property and Value Functions

Markovian Property

The Markovian property asserts that the future state depends only on the present state and action, and not on the sequence of states and actions that preceded it. Formally:

$$P(S_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, \ldots, S_0, A_0) = P(S_{t+1} \mid S_t, A_t) \tag{17}$$

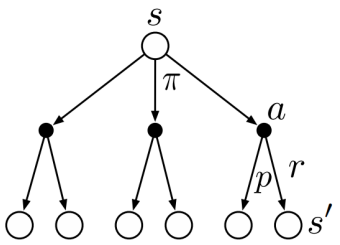

Figure 4: Tree Structure for RL

Tree Structure for Reward Sequences

- Consider a tree-like structure where:
  - **Start with a State:** At the root, you have a current state.
  - **Possible Actions:** From the current state, you branch out to possible actions with probabilities defined by the policy π.
  - **Branch to Possible States and Rewards:** Each action leads to possible next states and rewards with probabilities defined by p.
- This structure helps in finding the sequence of rewards and allows us to compute the expected return of each state.
- By using the value function results, we can determine the best state and pick an action that maximizes the probability p of reaching that state.
- The value function is a crucial quantity in this process.

Value Function and Equations

The value function $V(s)$ of a state s can be computed using the Bellman equation. The Bellman equation for the value function $V(s)$ is given by:

$$V(s) = \sum_{a} \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s') \right] \tag{18}$$

where:

- $\pi(a \mid s)$ is the policy that specifies the probability of taking action a in state s.
- $p(s', r \mid s, a)$ is the probability of transitioning to state s' and receiving reward r, given that action a is taken in state s.
- γ is the discount factor that determines the importance of future rewards.
- The term $r + \gamma V(s')$ represents the immediate reward plus the discounted value of the next state.
- The equation expresses the value of a state s as the expected sum of rewards obtained from taking actions and transitioning to new states.

- It sums over all possible actions a, states s', and rewards r, weighted by their probabilities.
- The value function $V(s)$ provides a measure of the expected return for being in state s and following the policy π.

Section: 2-1
Notes from Textbook Readings

Chapter 3 Finite Markov Decision Processes

Agent & Environment

- MDPs frame the problem of learning from interaction to achieve a goal.
- The learner and decision maker is called the agent.
- The environment comprises everything outside the agent.
- The agent and environment interact continually, with the agent selecting actions and the environment responding.
- The environment provides rewards, numerical values the agent seeks to maximize over time through its actions.
- Interaction specifics:
  - At each time step $t = 0, 1, 2, 3, \ldots$, the agent receives the environment's state $S_t \in S$.
  - Based on S_t, the agent selects an action $A_t \in A(s)$.
  - One time step later, as a consequence of its action, the agent receives a reward $R_{t+1} \in R \subseteq R$ and a new state S_{t+1}.
- Sequence or trajectory:
  - The sequence or trajectory begins:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{19}$$

- Finite MDP:
  - Sets of states, actions, and rewards (S, A, R) have a finite number of elements.
  - Random variables R_t and S_t have well-defined discrete probability distributions dependent on the preceding state and action.
  - For specific values $s' \in S$ and $r \in R$, the probability of these values occurring at time t, given particular preceding state and action, is

$$p(s', r \mid s, a) = \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{20}$$

- Dynamics function p:
  - p defines the MDP dynamics:

$$p : S \times R \times S \times A \to [0, 1] \tag{21}$$

  - The dynamics function $p(s', r \mid s, a)$ specifies a probability distribution for each s and a.

$$\sum_{s' \in S} \sum_{r \in R} p(s', r \mid s, a) = 1 \quad \text{for all } s \in S, a \in A(s) \tag{22}$$

- In an MDP, probabilities given by p fully characterize the environment's dynamics.
- The probability of each possible value for S_t and R_t depends on the immediately preceding state and action, S_{t-1} and A_{t-1}.
- The state must include information about all past agent-environment interactions that affect the future.
- A state with this property is said to have the Markov property.
- From the four-argument dynamics function p, one can compute other useful information:
  - State-transition probabilities:

$$p(s' \mid s, a) = \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r \mid s, a) \tag{23}$$

  - Expected rewards for state-action-next-state triples:

$$r(s, a, s') = E[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \cdot p(s', r \mid s, a) \tag{24}$$

- The MDP framework is abstract and flexible, applicable to various problems:
  - Time steps can refer to arbitrary successive stages of decision-making and acting.
  - Actions can range from low-level controls to high-level decisions.
  - States can be low-level sensor readings or high-level abstract descriptions.
- State information can include memory of past sensations or be entirely mental/subjective.
- Actions can be decisions about what to think about or where to focus attention.
- The boundary between agent and environment is usually closer to the agent than the physical body.
- Anything that cannot be changed arbitrarily by the agent is considered part of the environment.
- The reward computation is external to the agent because it defines the task facing the agent.
- The agent may know everything about its environment but still face a challenging reinforcement learning task.
- The agent-environment boundary represents the limit of the agent's control, not its knowledge.
- The boundary can vary for different purposes; in complex systems, multiple agents may operate simultaneously.
- The MDP framework abstracts goal-directed learning from interaction into three signals:
  - Actions: choices made by the agent.
  - States: basis for making choices.
  - Rewards: define the agent's goal.
- This framework is widely useful and applicable but may not cover all decision-learning problems.
- Representational choices for states and actions can strongly affect performance and are more art than science.

The Reward Signal in Reinforcement Learning

- In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent.
- At each time step, the reward is a simple number, $R_t \in R$.
- The agent's goal is to maximize the cumulative reward in the long run, not just immediate reward.
- The **reward hypothesis** states:

  > All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

- The use of a reward signal to formalize the idea of a goal is a distinctive feature of reinforcement learning.
- Although formulating goals in terms of reward signals might seem limiting, it has proven to be flexible and widely applicable.
- Examples of reward usage:
  - To make a robot learn to walk, provide reward proportional to forward motion.
  - In a maze escape task, give a reward of -1 for every time step before escape to encourage quick escape.
  - For a robot collecting soda cans, give a reward of +1 for each can collected and possibly negative rewards for bumping into objects.
  - For playing checkers or chess, the rewards could be +1 for winning, -1 for losing, and 0 for drawing.
- The agent learns to maximize its reward, so rewards must be set up to align with the desired goals.
- It is critical that rewards truly indicate the desired accomplishments.
- The reward signal is not the place to impart prior knowledge about how to achieve the goals.
- For example, a chess-playing agent should be rewarded only for winning, not for subgoals like taking pieces or controlling the center, to avoid it achieving subgoals at the cost of losing the game.

Formulating the Objective of Learning

- The goal of the agent is to maximize the cumulative reward it receives in the long run.
- The sequence of rewards received after time step t is denoted $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$.
- The return, denoted G_t, is defined as a specific function of the reward sequence.

- In the simplest case, the return is the sum of the rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{25}$$

  where T is a final time step.

- This approach is suitable for applications with a natural notion of a final time step, called episodic tasks.

- Episodic tasks end in a special state called the terminal state, followed by a reset to a standard starting state or a sample from a standard distribution of starting states.

- Tasks without identifiable episodes are called continuing tasks, where the interaction goes on continually without limit.

- For continuing tasks, the simple return formulation is problematic because $T = \infty$ and the return could be infinite.

- To address this, we use the concept of discounting, where the agent selects actions to maximize the sum of discounted rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{26}$$

  where γ is the discount rate and $0 < \gamma < 1$.

- The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth γ^{k-1} times what it would be worth if received immediately.

- If $\gamma = 0$, the agent is concerned only with maximizing immediate rewards (myopic behavior).

- As γ approaches 1, the agent takes future rewards into account more strongly (farsighted behavior).

- Returns at successive time steps are related by:

$$G_t = R_{t+1} + \gamma G_{t+1}. \tag{27}$$

- This works for all time steps $t < T$, even if termination occurs at $t + 1$, provided we define $G_T = 0$.

- The return G_t is finite if the reward is nonzero and constant, and if $\gamma < 1$. For example, if the reward is a constant $+1$, then:

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}. \tag{28}$$

Unified Notation for Episodic and Continuing Tasks

- Two kinds of reinforcement learning tasks: episodic tasks and continuing tasks.

- Episodic tasks naturally break down into a sequence of separate episodes.

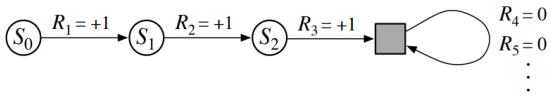

Figure 5: Episodic Task

- Continuing tasks do not break into episodes and go on continually without limit.
- Establishing a notation that covers both cases simultaneously.
- For episodic tasks, consider a series of episodes, each with a finite sequence of time steps.
- Number the time steps of each episode starting anew from zero.
- Use $S_{t,i}$ to refer to the state at time t of episode i (similarly for $A_{t,i}, R_{t,i}, \pi_{t,i}, T_i$, etc.).
- Often drop the explicit reference to the episode number when it is not needed.
- Unified return definition using a special absorbing state for episode termination.
- The solid square in the diagram represents the special absorbing state at the end of an episode.
- Summing the reward sequence gives the same return whether summed over the first T rewards or the full infinite sequence.
- Unified return definition:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{29}$$

  where T could be infinite or $\gamma = 1$ (but not both).

Policy and Value Function

- Almost all reinforcement learning algorithms involve estimating value functions—functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state).
- The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return.
- The rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.
- Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.
- Like p, π is an ordinary function; the "|" in the middle of $\pi(a|s)$ merely reminds us that it defines a probability distribution over $a \in A(s)$ for each $s \in S$.

- Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.
- The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter.
- For MDPs, we can define v_π and q_π formally by:

$$v_\pi(s) \doteq E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\Big|S_t = s, A_t = 1\right] \tag{30}$$

$$q_\pi(s,a) \doteq E_\pi\left[G_t \mid S_t = s, A_t = a\right] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \tag{31}$$

- We call q_π the action-value function for policy π.
- The value functions v_π and q_π can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity.
- If separate averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s,a)$.
- We call estimation methods of this kind Monte Carlo methods because they involve averaging over many random samples of actual returns.
- If there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain v_π and q_π as parameterized functions (with fewer parameters than states) and adjust the parameters to better match the observed returns.
- This can also produce accurate estimates, although much depends on the nature of the parameterized function approximator.
- A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships.
- For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \quad \text{for all } s \in S \tag{32}$$

- It is implicit that the actions, a, are taken from the set $A(s)$, that the next states, s', are taken from the set S (or from S^+ in the case of an episodic problem), and that the rewards, r, are taken from the set R.
- Note how the final expression can be read easily as an expected value. It is a sum over all values of the three variables, a, s', and r. For each triple, we compute its probability, $\pi(a|s)p(s',r|s,a)$, weight the quantity in brackets by that probability, then sum over all possibilities to get an expected value.

- Equation (32) is the Bellman equation for v_π. It expresses a relationship between the value of a state and the values of its successor states.
- The Bellman equation (32) averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.
- The value function v_π is the unique solution to its Bellman equation.
- We call diagrams like Figure 4 backup diagrams because they diagram relationships that form the basis of the update or backup operations that are at the heart of reinforcement learning methods.
- These operations transfer value information back to a state (or a state–action pair) from its successor states (or state–action pairs).

Optimal - Policies and Value Functions

- Solving a reinforcement learning task involves finding a policy that maximizes long-term reward.
- For finite Markov Decision Processes (MDPs):
  - An optimal policy π^* is one that is better than or equal to all other policies.
  - Policies are compared using value functions: $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for all states $s \in S$.
  - Optimal policies share the same state-value function v^*, defined as:

$$v^*(s) = \max_\pi v_\pi(s) \tag{33}$$

  - Optimal policies also share the same optimal action-value function q_*, defined as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{34}$$

  - The action-value function can be expressed in terms of v^*:

$$q_*(s,a) = E[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a] \tag{35}$$

- The Bellman optimality equation for v^* is:

$$v^*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v^*(s')] \tag{36}$$

- The Bellman optimality equation for q_* is:

$$q_*(s,a) = \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{37}$$

- For finite MDPs, the Bellman optimality equation for v^* has a unique solution.
  - The equation system consists of n equations for n states.

  - If the environment dynamics p are known, the system can be solved using methods for nonlinear equations.

- Determining an optimal policy:
  - For each state s, select actions where the maximum is obtained in the Bellman optimality equation.
  - A policy that assigns nonzero probability only to these actions is optimal.
  - Any policy greedy with respect to v^* is optimal.
  - Greedy policies select actions based on short-term consequences but are optimal long-term when evaluated with v^*.

- Using q_* simplifies action selection:
  - For any state s, find an action that maximizes $q_*(s, a)$.
  - The action-value function provides the optimal expected long-term return for state-action pairs.

- Solving the Bellman optimality equation directly is rarely practical due to:
  - Inaccurate knowledge of environment dynamics.
  - Insufficient computational resources.
  - Violation of the Markov property in states.

- Approximate solutions are often used in reinforcement learning:
  - Heuristic search methods approximate v^* at leaf nodes of a search tree.
  - Dynamic programming methods are closely related to the Bellman optimality equation.
  - Many reinforcement learning methods use actual experienced transitions to approximate solutions.

Optimality and Approximation

- Optimal value functions and optimal policies are defined, but learning an optimal policy in practice is rare due to high computational costs.

- Optimality is an ideal that agents can only approximate, guiding the theoretical understanding of learning algorithms.

- Even with a complete and accurate model of the environment's dynamics, computing an optimal policy by solving the Bellman optimality equation is often impractical.
  - Example: Chess requires custom-designed computers to compute optimal moves, which are still not perfect.

- Computational power and memory constraints are critical aspects of the problem facing the agent.
  - Memory is needed to build approximations of value functions, policies, and models.

  - For small, finite state sets, approximations can be formed using arrays or tables (tabular methods).
  - For large state sets, more compact parameterized function representations are required.
- Approximations are necessary due to practical constraints, but offer unique opportunities.
  - Many states are encountered with low probability, so suboptimal actions in these states have little impact on overall reward.
  - Reinforcement learning focuses on making good decisions for frequently encountered states, distinguishing it from other MDP-solving approaches.

MDP Examples

- **Bioreactor**
  - Reinforcement learning is applied to determine moment-by-moment temperatures and stirring rates for a bioreactor.
  - Actions: Target temperatures and target stirring rates passed to lower-level control systems.
  - States: Thermocouple and other sensory readings, possibly filtered and delayed, plus symbolic inputs representing ingredients in the vat and the target chemical.
  - Rewards: Moment-by-moment measures of the rate at which the useful chemical is produced.
  - In this example:
    * Each state is a list (vector) of sensor readings and symbolic inputs.
    * Each action is a vector consisting of a target temperature and a stirring rate.
    * Rewards are always single numbers.
- **Pick-and-Place Robot**
  - Reinforcement learning is used to control the motion of a robot arm in a repetitive pick-and-place task.
  - To achieve fast and smooth movements, the agent must control the motors directly and have low-latency information about current positions and velocities of the mechanical linkages.
  - Actions: Voltages applied to each motor at each joint.
  - States: Latest readings of joint angles and velocities.
  - Rewards:
    * +1 for each object successfully picked up and placed.
    * A small negative reward given at each time step as a function of the moment-to-moment jerkiness of the motion to encourage smooth movements.
- **Recycling Robot**
  - Mobile robot tasked with collecting empty soda cans in an office environment.

- Has sensors for detecting cans, an arm and gripper for picking up cans, and a rechargeable battery.
- Control system includes components for interpreting sensory information, navigating, and controlling the arm and gripper.
- High-level decisions about searching for cans are made by a reinforcement learning agent based on the battery charge level.
- State set $S = \{\text{high, low}\}$ for battery charge levels.
- Actions:
  - $A(\text{high}) = \{\text{search, wait}\}$
  - $A(\text{low}) = \{\text{search, wait, recharge}\}$
- Rewards:
  - Positive when the robot secures a can.
  - Large negative reward if the battery is depleted.
- Searching is the best way to find cans but depletes the battery.
- Probability of battery state after searching:
  - High to low: $1 - \alpha$
  - Low to low: $1 - \beta$
  - Low to high after rescue and recharge.
- Transition probabilities and rewards can be summarized in a transition graph.

- **Pole-Balancing**
  - Objective: Apply forces to a cart to keep a hinged pole from falling over.
  - Failure occurs if the pole falls past a certain angle or if the cart runs off the track.
  - Task can be treated as episodic or continuing.
    - Episodic: Reward of +1 for each time step without failure.
    - Continuing: Reward of -1 at failure, zero otherwise.
  - Goal: Maximize return by balancing the pole for as long as possible.

- **Gridworld**
  - Rectangular grid where each cell represents a state.
  - Possible actions: north, south, east, west.
  - Actions move the agent one cell in the respective direction, unless blocked by the grid edge.
  - Rewards:
    - -1 for actions that move the agent off the grid.
    - 0 for most actions.
    - +10 for actions that move the agent out of special state A to A'.
    - +5 for actions that move the agent out of special state B to B'.
  - Value of states depends on potential rewards and penalties.

- **Golf**
  - Task: Minimize the number of strokes to get the ball into the hole.
  - State: Location of the ball.
  - Actions: Aiming and swinging at the ball, selecting the club.
  - Reward: -1 for each stroke until the ball is in the hole.
  - Value of a state: Negative of the number of strokes to the hole from that location.
  - Example policy: Always use the putter.
  - State-value function $v_{\text{putt}}(s)$ shows values of states for the putter policy.
  - Terminal state (in-the-hole) has a value of 0.
  - Sand traps have a value of $-\infty$ since putting cannot get out of sand traps.

Section: 3
Model-based RL: Dynamic Programming
Lecture # 3 Date: May 25, 2024

Previous Lectures' Summary

- **Design a brain for the agent (M):**
  - The decision-making system or control system for the agent.
- **M coupled with the Environment (E) via actuation (A):**
  - The agent interacts with the environment through actions.
- **Receiving Feedback S (States feedback) and R (Reward) from the E:**
  - The agent receives the current state (S) and reward (R) from the environment.
- **Reward signal related to the goal:**
  - The goal is to maximize the cumulative reward the agent receives.
- **Value Functions:**
  - The agent's knowledge of the environment is contained in:
    1. State Value Function: $v_\pi(s)$
    2. State-Action Value Function: $q_\pi(s, a)$
- **Policy (π):**
  - The policy is the strategy used by the agent to select actions.
  - It is a mapping from states to actions.
- **Action based on Policy:**
  - The agent performs actions based on the policy.
- **Knowledge and Control:**
  - Knowledge: $v_\pi(s)$ and $q_\pi(s, a)$
  - Control: π (policy)
- **Value Functions:**
  - $v_\pi(s)$: How good it is for the environment to be in state s.
  - $q_\pi(s, a)$: How good it is to be in state s and take action a.
- **Environment Dynamics:**
  - Transition probabilities and rewards: $p(s', r|s, a)$

Start off

- **Dynamic Programming (DP): Complete Model E $\sim$ MDP**
- DP represents an ideal, similar to linear systems, and helps in showing a prototype of the general unknown environment.
- If we have a complete model (MDP), is DP the only solution mechanism? **NO**
- DP will be the model for other techniques. It shows what is possible when you have the right information and what the solving structure would look like.

Overview (before going into the details)

- **Prediction:** Given a policy, determine the value function for that policy (policy evaluation).
- **Control:** How do we find the optimal policy for the agent.

Policy Evaluation

- **Policy Evaluation:**
  - Given a known Environment (MDP) and policy $\pi \rightarrow$ how we evaluate the policy.
  - Iteration: We find an estimate of the policy and iteratively refine that estimate until we find the best one.

$$V_\pi(s) = E_\pi[G_t \mid S_t = s] \tag{38}$$

$$V_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V_\pi(s')\right] \tag{39}$$

where G_t is the return (total accumulated reward) starting from state s at time t. $\pi(a \mid s)$ is the policy probability of taking action a in state s, and $p(s', r \mid s, a)$ is the probability of transitioning to state s' and receiving reward r.

Policy Improvement

- **Policy Improvement:**
  - Given we have our value functions based on policy evaluation, can we find a better policy? If $V_{\pi'} > V_\pi$, then $\pi' > \pi$.
  - Notation: $\pi \rightarrow (v, q) \rightarrow \pi'$.
  - Idea: Take new actions that are not part of the current policy and perform value function updates.

Policy Iteration

- **Policy Improvement Process:**
  - Start with an initial policy π_0.
  - Evaluate the policy to obtain the value function v_{π_0}.
  - Improve the policy based on the value function v_{π_0} to obtain a new policy π_1:

$$\pi_0 \xrightarrow{\text{Evaluate}} v_{\pi_0} \xrightarrow{\text{Improve}} \pi_1 \tag{40}$$

  - Iterate this process:

$$\pi_1 \xrightarrow{\text{Evaluate}} v_{\pi_1} \xrightarrow{\text{Improve}} \pi_2 \xrightarrow{\text{Evaluate}} v_{\pi_2} \xrightarrow{\text{Improve}} \pi_3 \dots \tag{41}$$

Fixed Points

- A fixed point is a value that, when substituted into a recurrent relationship or function, returns the same value. It represents an equilibrium point where the value remains unchanged after the function is applied.

Policy Improvement Theorem

- **Policy Improvement Theorem:**
  - If π is a policy and v_π is its value function, then there exists a new policy π' such that:

$$v_{\pi'}(s) \geq v_\pi(s) \tag{42}$$

  - The new policy π' is obtained by choosing actions that maximize the value function $V_\pi(s)$:

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{43}$$

  - Where the action-value function $q_\pi(s, a)$ is defined as:

$$q_\pi(s, a) = \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right] \tag{44}$$

  - Thus, the policy improvement theorem states that by improving the policy based on the current value function, the new policy π' will be at least as good as the old policy π.

Overall Process

- **Environment:** E is a Markov Decision Process (MDP).
- **Steps:**
  1. Evaluate the current policy π:

$$\pi \to v_\pi, q_\pi \tag{45}$$

2. Improve the policy π to obtain a new policy π':

$$\pi \to \pi' \text{ (based on } q_\pi \text{ and } v_\pi) \tag{46}$$

3. Iterate between policy evaluation and policy improvement until convergence to the optimal policy π^*:

$$\text{Evaluation} \leftrightarrow \text{Improvement} \to \pi^* \tag{47}$$

4. This process can be implemented using:
   - **Policy Iteration:** Alternating between policy evaluation and policy improvement.
   - **Value Iteration:** Directly computing the optimal value function and deriving the optimal policy.

Section: 3-1
Notes from Textbook Readings

Chapter 4 Dynamic Programming

- **Dynamic Programming (DP):**
  - Refers to a collection of algorithms for computing optimal policies given a perfect model of the environment as a Markov Decision Process (MDP).
  - Classical DP algorithms assume a perfect model and are computationally expensive, but they are crucial theoretically.
  - DP provides a foundational understanding for reinforcement learning methods.
  - All methods in this book can be viewed as approximations to DP, requiring less computation and not assuming a perfect model.
- **Finite MDP Assumption:**
  - Environment is assumed to be a finite MDP with finite state (S), action (A), and reward (R) sets.
  - Dynamics are defined by probabilities $p(s', r \mid s, a)$ for all $s \in S$, $a \in A(s)$, $r \in R$, and $s' \in S^+$ (where S^+ includes terminal states in episodic problems).
- **Approximate Solutions:**
  - For continuous state and action spaces, exact solutions are rare.
  - Approximate solutions often involve quantizing the state and action spaces and then applying finite-state DP methods.
  - Methods in Part II address continuous problems and extend the finite-state DP approach.
- **Value Functions and Bellman Optimality Equations:**
  - Key idea of DP and reinforcement learning is the use of value functions to find good policies.
  - Optimal value functions (v^* or q_*) satisfy the Bellman optimality equations:

$$v^*(s) = \max_a E[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a] = \max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v^*(s')\right] \tag{48}$$

$$q_*(s,a) = E[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] = \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{49}$$

  - DP algorithms involve converting Bellman equations into update rules for approximating value functions.

Policy Evaluation - Prediction

- **Policy Evaluation (Prediction):**
  - **Objective:** Compute the state-value function v_π for a given policy π.
  - **Definition:**

$$v_\pi(s) = E_\pi[G_t \mid S_t = s] = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s')\right] \tag{50}$$

  - **Existence and Uniqueness:**
    - Guaranteed if $r < 1$ or eventual termination is guaranteed from all states under policy π.
  - **Solving the Equations:**
    - If the environment's dynamics are known, it forms a system of $|S|$ simultaneous linear equations.
    - Solution can be tedious; iterative methods are typically used.
  - **Iterative Policy Evaluation:**
    - **Algorithm:** Update value functions iteratively using:

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right] \tag{51}$$

    - **Convergence:** The sequence $\{v_k\}$ converges to v_π under conditions guaranteeing the existence of v_π.
  - **Expected Updates:**
    - Each iteration replaces the old value of s with a new value based on the successor states and rewards.
    - Updates are based on expectations over all possible next states rather than on a sample.
  - **Implementation:**
    - **Two-array Method:** Use separate arrays for old and new values; updates are computed one by one.
    - **In-place Method:** Update values in place, which often converges faster. The order of state updates affects convergence rate.

Policy Improvement

- The goal of computing the value function v_π for a policy π is to find better policies.
- To evaluate whether changing the policy to a new deterministic action $a \neq \pi(s)$ is beneficial, consider the value:

$$q_\pi(s, a) = E[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \tag{52}$$

- Compare $q_\pi(s, a)$ with $v_\pi(s)$. If $q_\pi(s, a) > v_\pi(s)$, then changing the policy to always choose a at state s would improve the policy overall.

Algorithm 1 Iterative Policy Evaluation

Input: Policy π to be evaluated
Algorithm Parameter: A small threshold $\theta > 0$ determining the accuracy of estimation
Initialize $V(s)$ arbitrarily for all $s \in S$
Set $V(\text{terminal})$ to 0

while do
  $\triangle \leftarrow 0$
  for each $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
    $\triangle \leftarrow \max(\triangle, |v - V(s)|)$
  end for
end while

Policy Improvement Theorem:

- Let π and π' be deterministic policies. If for all $s \in S$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{53}$$

  then π' is as good as or better than π, i.e.,

$$v_{\pi'}(s) \geq v_\pi(s) \tag{54}$$

- If strict inequality holds at any state s, then:

$$v_{\pi'}(s) > v_\pi(s) \tag{55}$$

- The policy improvement theorem is applied when considering changes to policies that differ at specific states.
- The greedy policy π' with respect to v_π is defined as:

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{56}$$

- If the new policy π' is as good as but not better than the old policy π, then $v_{\pi'} = v_\pi$ and π' is optimal.
- For stochastic policies, the policy improvement theorem holds similarly, with actions sharing probabilities if multiple actions maximize $q_\pi(s, a)$.
- Example of policy improvement for stochastic policies shows that a new policy π' can be better or equal to the old policy π but not worse.

Policy Iteration

- Policy iteration involves repeatedly improving a policy π using its value function v_π to obtain a better policy π'.
- The process yields a sequence of monotonically improving policies and value functions:

$$\pi \xrightarrow{\text{E}} v_\pi \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \dots$$

  where E denotes policy evaluation, I denotes policy improvement, and $\rightarrow$ denotes the transition to a new policy.
- Each policy in the sequence is guaranteed to be a strict improvement over the previous one, unless the policy is already optimal.
- Given a finite MDP, the process must converge to an optimal policy and its corresponding optimal value function in a finite number of iterations.
- Policy iteration involves iteratively evaluating and improving policies. The evaluation phase starts with the value function of the previous policy, often speeding up convergence.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(\textit{terminal}) \doteq 0$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 6: Policy Iteration

Value Iteration

- A limitation of policy iteration is that policy evaluation can be computationally intensive, often requiring multiple sweeps through the state set.
- Policy evaluation can be truncated without losing convergence guarantees, meaning exact convergence to v_π is not always necessary.

- **Value Iteration** is an approach where policy evaluation is stopped after a single sweep, combining policy improvement and truncated policy evaluation:

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V_k(s') \right] \tag{57}$$

- The sequence $\{V_k\}$ converges to the optimal value function v^* under the same conditions that guarantee its existence.
- Value iteration updates are derived from the Bellman optimality equation and are similar to policy evaluation updates, except they include a maximization over actions.
- **Termination**: Value iteration requires an infinite number of iterations to converge exactly to v^*, but in practice, it stops when changes in the value function are below a small threshold.
- Value iteration effectively performs one sweep of policy evaluation and one sweep of policy improvement in each iteration.
- Faster convergence can be achieved by interposing multiple policy evaluation sweeps between policy improvements.
- Truncated policy iteration algorithms, including value iteration, can be seen as sequences of sweeps that combine policy evaluation and value iteration updates. All these algorithms converge to an optimal policy for discounted finite MDPs.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Figure 7: Value Iteration

Generalized Policy Iteration

- **Policy Iteration:**
  - Consists of two interacting processes: policy evaluation and policy improvement.
  - In policy iteration, these processes alternate, but this is not strictly necessary.
  - **Value Iteration** performs only one iteration of policy evaluation between policy improvements.

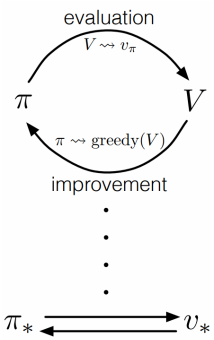

Figure 8: Generalized Policy Iteration

  - **Asynchronous DP Methods** interleave the evaluation and improvement processes at a finer granularity, sometimes updating individual states before switching processes.

- **Generalized Policy Iteration (GPI):**
  - Refers to the general idea of allowing policy-evaluation and policy-improvement processes to interact, regardless of their granularity.
  - Most reinforcement learning methods can be described as GPI, where policies and value functions are identified, and policies are improved based on the value function while the value function is updated towards the policy.
  - **Convergence**: Both evaluation and improvement processes stabilize when the value function and policy are optimal.
  - Evaluation and improvement processes can be viewed as competing and cooperating.
    * Compete: Making the policy greedy with respect to the value function can make the value function incorrect for the new policy.
    * Cooperate: Over time, they interact to find the joint solution of the optimal value function and policy.
  - **Constraints and Goals**:
    * The processes can be thought of as driving toward two constraints or goals, which interact in a non-orthogonal manner.
    * Driving toward one goal may cause movement away from the other goal, but the joint process will approach optimality.

Efficiency of Dynamic Programming

- **Practicality:**
  - DP may not be practical for very large problems.
  - Despite this, DP methods are efficient compared to other methods for solving MDPs.
  - In the worst case, DP methods take polynomial time in the number of states n and actions k.

  - The time complexity of DP methods is less than some polynomial function of n and k.

- **Comparison with Direct Search:**
  - DP methods are exponentially faster than direct search, which would need to exhaustively examine all k^n policies.
  - Linear programming methods can also solve MDPs but become impractical with more states, typically by a factor of about 100.
  - For the largest problems, DP methods are often the only feasible solution.

- **Curse of Dimensionality:**
  - Large state spaces due to the curse of dimensionality pose challenges.
  - These difficulties are inherent to the problem, not to DP itself.
  - DP is comparatively better suited for large state spaces than direct search or linear programming.

- **Practical Usage:**
  - DP methods can solve MDPs with millions of states using modern computers.
  - Both policy iteration and value iteration are widely used, though it is unclear which is better in general.
  - In practice, these methods often converge faster than theoretical worst-case runtimes, especially with good initial values.

- **Asynchronous DP Methods:**
  - Asynchronous methods are preferred for large state spaces.
  - Synchronous methods require computation and memory for every state, which can be impractical.
  - Asynchronous methods and variations of Generalized Policy Iteration (GPI) may find good or optimal policies faster.

Section: 4
Model-free RL #1: Monte Carlo Methods

Lecture # 4 Date: June 1, 2024

Connection Between Value Function Table and Value-Action Table

- There is a connection between the Value function table and the Value-Action function table.
- In the Value function table, the state with the highest probability is selected.
- The following tables help in understanding how to use the value function and model together.

Value Function Table

- S represents the state.
- $E_\pi[G_i|S_i = s]$ denotes the expected return starting from state s under policy π.
- The table below illustrates the values for different states:

State (S)	$E_\pi[G_i\|S_i = s]$
$S(1)$	
...	
$S(i+1)$	
$S(N)$	

Value-Action Function Table

- (s, a) represents a state-action pair.
- The table shows the transitions from a current state s and action a to possible next states $S(k)$ with corresponding actions $a(k)$.

Present	Next	$S(1)$	$S(2)$	... $S(k)$ ... $S(N)$
(s, a)				
$S(1)a(1)$				
$S(1)a(2)$				
$S(i)a(1)$				
$S(i)a(2)$				
$S(i)a(3)$				
...				
$S(i)a(*)$				

Summary of Tables

- We want to find the best state to go to: State with the best value.
- Any algorithm that is called Dynamic Programming has an optimal structure to it that is recursive.

Model Free Reinforcement Learning (RL)

- No model: $V_\pi(s)$ is not useful. In this case, use $q_\pi(s, a)$ - State Action Value.

$$q_\pi(S, a) = E_\pi[G_t \mid S_t = s, A_t = a] \tag{58}$$

- Look at many episodes where we had $(S_t = s, A_t = a)$ and average the return.
- Get experience from episodes. Scan through the trajectory and find the exact pair and get the return in that episode. Average these returns.
- If we take more and more episodes and returns, we approach the true expected value without using P. (Monte Carlo)

Monte Carlo

- Uses repeated random sampling to obtain the likelihood of a range of results occurring.
- Estimates values by averaging over a large number of random samples.
- We may not have key pieces of information like series expansions, but we look for a property for which a ratio exists or some relationship that we can exploit.
- We don't have a model, but we have an agent and an environment, and we can run episodes for a longer time and look at the experiences.
- Scan through episodes and find the exact pair and look for all the rewards onwards.
- No need for P. No need for successive approximations.
- Way of getting to a value function without using anything other than pure experiences.
- This is our first learning algorithm. RL $\rightarrow$ Experience (learning).
- Steps:
  - Scan through episodes.
  - Find the pair of interest.
  - Sum the rewards from that point onwards until the terminal state using the reward summation formula.
  - Get the return.
  - Apply the state-action value formula.
- Requires a lot of time.

Evaluation: Monte Carlo

- Monte Carlo method for learning a value function first observes multiple returns from the same state. Then, it averages those observed returns to estimate the expected return from that state.
- Action values are useful for learning a policy. They allow us to compare different actions in the same state. Then, we can switch to a better action if one is available.
- Initialize $Q(s, a)$ table.
- Initialize Returns (s, a) table.

Input	Output 1	Output 2
(S, a)	Returns{}	$Q(s, a)$
$S(1)A(1)$		
$S(1)A(2)$		
$\cdots$		
$S(2)A(1)$		

Algorithm 2 Monte Carlo Evaluation

Input: Policy π and episodes
Algorithm Parameter: Threshold for convergence (if applicable)
Initialize $Q(s, a)$ table arbitrarily
Initialize Returns (s, a) table to empty lists
while convergence criteria not met **do**
  Acquire an episode using policy π
  for each step (s, a, r) in the episode **do**
    Compute the return G_t from the step (s, a)
    Append G_t to the Returns table for (s, a)
  end for
  for each state-action pair (s, a) **do**
    if Returns for (s, a) has sufficient data **then**
      Average the returns for (s, a)
      Update $Q(s, a)$ with the average return
    end if
  end for
end while
Output: Estimated $Q(s, a)$ values

Exploration

- **Without Model (p):**
  - Use $Q(s, a)$ and experience.
- **Experience and Exploration:**

  - Experience leads to exploration of the environment.

- **Exploratory Starts:**
  - Start the environment in every arbitrary state.
  - Have the agent try every action for each state.

- **Epsilon-Soft Policy:**
  - Define the probability $\pi(a \mid s) > \epsilon > 0$.
  - Higher chances of encountering every possible combination of actions.
  - Epsilon-soft policies are always stochastic.
  - Deterministic policy specifies a single action to take in each state.
  - Stochastic policies specify the probability of taking an action in each state with probability ϵ.
  - All actions have a probability of at least $\frac{\epsilon}{\text{number of actions}}$, ensuring that all actions are eventually tried.

GPI with Monte Carlo

- **Using Monte Carlo Evaluation:**
  - Experience is used to estimate values.
  - Average the returns to evaluate the policy.

- **Policy and Value Function Updates:**
  - The policy iteration with Monte Carlo updates can be described as:

$$\pi_0 \to q_{\pi_0} \to \pi_1 \to q_{\pi_1} \to \ldots \to \pi_* \to q_* \tag{59}$$

- **Policy Evaluation and Improvement:**
  - GPI with Monte Carlo does not perform a complete policy evaluation step before improvement.
  - Instead, it evaluates and improves the policy after each episode.

Section: 4-1
Notes from Textbook Readings

Chapter 5 Monte Carlo Methods

- **Introduction to Learning Methods:**
  - In this chapter, we explore learning methods for estimating value functions and discovering optimal policies without assuming complete knowledge of the environment.
  - Unlike Dynamic Programming (DP), which requires a full model of the environment, Monte Carlo methods rely solely on experience—sample sequences of states, actions, and rewards.
- **Learning from Experience:**
  - **Actual Experience:**
    - No prior knowledge of the environment's dynamics is needed.
    - Optimal behavior can be attained through interaction with the environment.
  - **Simulated Experience:**
    - Requires a model to generate sample transitions, not the complete probability distributions needed for DP.
    - Easier to generate experience sampled according to desired distributions than to obtain the distributions explicitly.
- **Monte Carlo Methods:**
  - Based on averaging sample returns to estimate value functions.
  - Defined for episodic tasks where experience is divided into episodes, and value estimates and policies are updated only after the completion of each episode.
  - Incremental in an episode-by-episode sense, but not in a step-by-step (online) sense.
  - The term "Monte Carlo" specifically refers to methods that average complete returns, as opposed to partial returns.
- **Comparison to Bandit Methods:**
  - Monte Carlo methods are similar to bandit methods but applied in a multi-state context.
  - Each state can be thought of as a separate bandit problem (contextual or associative-search bandit).
  - Returns depend on actions taken in later states, making the problem nonstationary.
- **Handling Nonstationarity:**
  - Adapt the General Policy Iteration (GPI) approach from DP to handle nonstationarity in Monte Carlo methods.
  - Learn value functions from sample returns and interact with policies to attain optimality.
  - Approach involves prediction (computing v_π and q_π for a fixed policy π), policy improvement, and control through GPI.

Monte Carlo Prediction

- **Overview:**
  - Monte Carlo methods estimate the state-value function $v_\pi(s)$ for a given policy π using sample sequences of states, actions, and rewards.
  - The value of a state s is the expected return starting from that state.
  - The core idea is to average the returns observed after visits to the state s. As more returns are observed, the average should converge to the expected value.
- **First-Visit vs. Every-Visit Monte Carlo Methods:**
  - **First-Visit Monte Carlo:**
    - Estimates $v_\pi(s)$ as the average of returns following the first visits to state s.
    - Procedural example: Estimate value by averaging returns from the first visit to s in each episode.
  - **Every-Visit Monte Carlo:**
    - Estimates $v_\pi(s)$ as the average of returns following all visits to state s.
    - Generally extends more naturally to function approximation and eligibility traces.
  - Both methods converge to $v_\pi(s)$ as the number of visits or first visits to s goes to infinity.
  - First-visit MC is well-studied and widely used, while every-visit MC is more suited for advanced topics.
- **Convergence:**
  - **First-Visit MC:**
    - Each return is an independent, identically distributed estimate of $v_\pi(s)$.
    - By the law of large numbers, averages of these estimates converge to the expected value.
    - The standard deviation of the error decreases as $\frac{1}{\sqrt{n}}$, where n is the number of returns averaged.
  - **Every-Visit MC:**
    - Estimates also converge quadratically to $v_\pi(s)$, though this is less straightforward than first-visit MC.
- **Example: Blackjack**
  - The objective is to obtain a hand total as close to 21 as possible without exceeding it.
  - Each game of blackjack is an episode, with rewards of +1, -1, and 0 for winning, losing, and drawing respectively.
  - To estimate the state-value function for a policy, simulate many games and average the returns following each state.
  - With 500,000 games, the value function for states with a usable ace is well approximated.
  - Monte Carlo methods are advantageous here because generating sample games is easier than applying DP, which requires complex probability computations.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 9: First-visit MC prediction

- **Advantages of Monte Carlo Methods:**
  - Estimates for each state are independent; they do not build upon estimates of other states.
  - Computational expense for estimating a single state is independent of the number of states.
  - Useful for cases where only a subset of states or specific states are of interest.

Monte Carlo Estimation of Action Values

- **Overview:**
  - When a model is not available, estimating action values (values of state–action pairs) is crucial.
  - With a model, state values are sufficient to determine a policy. Without a model, both state and action values are needed to suggest a policy.
  - One of the main goals is to estimate q_*, the optimal action values.
- **Policy Evaluation for Action Values:**
  - The goal is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π.
  - Monte Carlo methods for action values are similar to those for state values but focus on state–action pairs.
  - A state–action pair (s, a) is said to be visited in an episode if state s is visited and action a is taken in it.
  - **Every-Visit MC Method:**
    - Estimates $q_\pi(s, a)$ as the average of returns following all visits to the state–action pair (s, a).
  - **First-Visit MC Method:**

    * Estimates $q_\pi(s, a)$ as the average of returns following the first time in each episode that state s is visited and action a is selected.
  - Both methods converge quadratically to the true expected values as the number of visits to each state–action pair approaches infinity.

- **Challenges and Solutions:**
  - Many state–action pairs may never be visited, especially under a deterministic policy where only one action per state is observed.
  - To effectively compare actions, it is essential to estimate the value of all actions from each state.
  - **Exploration Challenges:**
    * For policy evaluation to work, continual exploration is needed.
    * **Exploring Starts:**
      · An approach where episodes start in a state–action pair, and every pair has a nonzero probability of being selected as the start.
      · This ensures all state–action pairs are visited infinitely often with an infinite number of episodes.
    * This approach may not be practical in all situations, especially in actual interaction with an environment.
  - **Stochastic Policies:**
    * An alternative is to use stochastic policies that ensure a nonzero probability of selecting all actions in each state.
    * This approach guarantees exploration of all actions but may be more complex to implement.

Monte Carlo Control

Overview

- Monte Carlo control aims to approximate optimal policies using Monte Carlo estimation.
- The approach follows the Generalized Policy Iteration (GPI) pattern from dynamic programming (DP).
- In GPI, an approximate policy and value function are maintained:
  - The value function is adjusted to better approximate the value function for the current policy.
  - The policy is improved based on the current value function.

Monte Carlo Policy Iteration

- Perform alternating complete steps of policy evaluation and policy improvement.
- Start with an arbitrary policy π_0 and aim to end with the optimal policy and optimal action-value function.

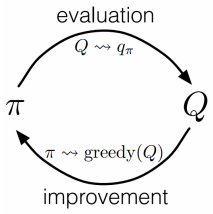

Figure 10: GPI with MC

- Policy evaluation:
  - Experience many episodes with the current policy.
  - The approximate action-value function approaches the true function asymptotically.
- Policy improvement:
  - Construct a greedy policy with respect to the current action-value function.
  - For any action-value function q, the corresponding greedy policy is:
    - $\pi(s) = \arg\max_a q(s, a)$
- The policy improvement theorem ensures that each new policy π_{k+1} is at least as good as the previous policy π_k.
- The overall process converges to the optimal policy and optimal value function.

Practical Considerations

- Theoretical assumptions:
  - Episodes have exploring starts.
  - Policy evaluation is done with an infinite number of episodes.
- In practice, these assumptions are often removed:
  - Approximating q_{π_k} in each policy evaluation.
  - Ensuring sufficient steps during policy evaluation to make errors small.
  - Alternating between evaluation and improvement on an episode-by-episode basis.
- Monte Carlo ES (Exploring Starts):
  - Accumulate and average all returns for each state–action pair, regardless of the policy in force when they were observed.
  - Convergence is ensured only when both policy and value function are optimal.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Figure 11: MC Exploring Start

Monte Carlo Control without Exploring Starts

- To avoid the unrealistic assumption of exploring starts, we need a method to ensure all actions are selected infinitely often.
- Two main approaches:
  - On-policy methods: Evaluate or improve the policy used to make decisions.
  - Off-policy methods: Evaluate or improve a policy different from that used to generate the data.
- On-policy Monte Carlo control can be designed without the assumption of exploring starts.

On-Policy Methods

- In on-policy control methods, the policy is typically soft:
  - $\pi(a|s) > 0$ for all $s \in S$ and all $a \in A(s)$.
  - Gradually shifted closer to a deterministic optimal policy.
- We use ϵ-greedy policies:
  - Most of the time, choose the action with the maximal estimated action value.
  - With probability ϵ, select an action at random.
  - Non-greedy actions get a minimal probability of selection, $\epsilon/|A(s)|$.
  - The remaining probability, $1 - \epsilon + \epsilon/|A(s)|$, is given to the greedy action.
- ϵ-greedy policies are ϵ-soft policies:
  - Defined as policies where $\pi(a|s) \geq \epsilon/|A(s)|$ for all states and actions, for some $\epsilon > 0$.
  - Among ϵ-soft policies, ϵ-greedy policies are closest to greedy.

Monte Carlo Control with ϵ-Greedy Policies

- The overall idea follows Generalized Policy Iteration (GPI):
  - Use first-visit Monte Carlo methods to estimate the action-value function for the current policy.
  - Move the policy toward an ϵ-greedy policy instead of a fully greedy one.
- Policy improvement:
  - An ϵ-greedy policy with respect to Q_π is guaranteed to be at least as good as the current policy π.
  - The policy improvement theorem assures that any ϵ-greedy policy is an improvement over the ϵ-soft policy π.

Analysis and Convergence

- To achieve convergence without exploring starts:
  - The policy iteration works for ϵ-soft policies.
  - Improvement is assured on each step, except when the best policy among the ϵ-soft policies is found.
- New environment for ϵ-soft policies:
  - The environment behaves similarly to the original but with policies required to be ϵ-soft.
  - The best achievable policy in the new environment is equivalent to the best among ϵ-soft policies in the original environment.
- The optimal policy among ϵ-soft policies is the same as the optimal policy in the new environment.

Off-Policy Prediction via Importance Sampling

- Learning control methods face a dilemma:
  - They seek to learn action values conditional on optimal behavior.
  - They need to behave non-optimally to explore all actions.
- On-policy methods: Learn action values for a near-optimal policy that still explores.
- Off-policy methods: Use two policies:
  - Target policy: The policy being learned and optimized.
  - Behavior policy: The policy used to generate behavior (more exploratory).
- Off-policy learning allows learning from data generated by a different policy.

Coverage and Importance Sampling

- To use data from the behavior policy b to estimate values for the target policy π:
  - Every action taken under π must be taken, at least occasionally, under b.
  - This requirement is called the assumption of coverage.
- The behavior policy b must be stochastic where it differs from π.
- Importance sampling:
  - A technique for estimating expected values under one distribution given samples from another.
  - Weight returns by the relative probability of trajectories occurring under the target and behavior policies.
- Importance-sampling ratio: Probability of trajectory under target policy:

$$\Pr\{A_t, S_{t+1}, A_{t+1}, \ldots, S_T \mid S_t, A_t; \pi\} = \prod_{k=t}^{T-1} \pi(A_k \mid S_k) \cdot p(S_{k+1} \mid S_k, A_k) \tag{60}$$

$$\text{Probability under behavior policy: } \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)} \tag{61}$$

- The trajectory probabilities cancel out, leaving the ratio depending only on the policies and the sequence.

Monte Carlo Algorithm

- We aim to estimate $v_\pi(s)$ using returns from episodes following policy b.
- Time steps are numbered across episodes:
  - If an episode ends at time 100, the next begins at time 101.
  - Define the set of all time steps in which state s is visited as $T(s)$.
- For each time step t, let $T(t)$ be the time of termination after t and G_t be the return after t up through $T(t)$.
- Importance sampling:
  - Ordinary importance sampling: Simple average of returns weighted by the importance-sampling ratio.
  - Weighted importance sampling: Weighted average of returns.

Ordinary vs. Weighted Importance Sampling

- Ordinary importance sampling:

$$V(s) = \frac{\sum_{t \in T(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in T(s)} \rho_{t:T(t)-1}} \tag{62}$$

- Weighted importance sampling:

$$V(s) = \frac{\sum_{t \in T(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in T(s)} \rho_{t:T(t)-1}} \tag{63}$$

- Zero if the denominator is zero.

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \in \mathbb{R}$ (arbitrarily)
  $C(s,a) \leftarrow 0$

Loop forever (for each episode):
  $b \leftarrow$ any policy with coverage of π
  Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
    $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

Figure 12: Off policy MC prediction (policy evaluation)

Off-Policy Monte Carlo Control

- Off-policy methods distinguish themselves by separating the policy used for generating behavior from the policy being evaluated and improved.
- The behavior policy can be different from the target policy. The target policy can be deterministic (e.g., greedy), while the behavior policy explores all possible actions.
- Advantages of this separation:
  - The target policy can be greedy, optimizing the value function.
  - The behavior policy ensures exploration of all possible actions.

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Figure 13: Off - Policy MC Control

Algorithm Overview

- Off-policy Monte Carlo control methods use techniques from earlier sections, applying weighted importance sampling for estimating the optimal policy and action-value function.
- The target policy π is updated to be the greedy policy with respect to the current estimate of Q, denoted as Q_π.
- The behavior policy b can be any policy, but it must ensure coverage by having a nonzero probability of selecting all actions that might be selected by the target policy π.
- To ensure convergence of π to the optimal policy, the behavior policy b should be ϵ-soft. This guarantees that all actions are sampled sufficiently.

Importance Sampling

- The behavior policy b must select all possible actions with nonzero probability.
- Coverage requirement: If $\pi(a|s) > 0$, then $b(a|s) > 0$.
- To learn about the target policy π, we use weighted importance sampling:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)} \tag{64}$$

Algorithm Details

- The off-policy Monte Carlo control algorithm based on GPI and weighted importance sampling is outlined below.

- The target policy π converges to optimal as long as all encountered states have infinite returns for each state-action pair, and the behavior policy is ϵ-soft.
- Potential problem: Learning may be slow if nongreedy actions are common or if episodes are long. This issue is particularly significant for states appearing early in long episodes.
- Possible solutions:
  - Incorporate temporal-difference learning to address slow learning.
  - Adjust the discount factor r and consider methods to mitigate the issue.

Example Task

- Apply the Monte Carlo control method to a task where a car needs to navigate a track:
  - Check if the car's projected path intersects the track boundary.
  - If it intersects the finish line, the episode ends. If it intersects elsewhere, the car is reset to the starting line.
  - With probability 0.1 at each time step, velocity increments are zero independently of the intended increments.

Section: 5
Model-free RL #2: Temporal-Difference Learning

Lecture # 5 Date: June 8, 2024

Recap

Dynamic Programming

Dynamic Programming (DP) utilizes the concept of Generalized Policy Iteration (GPI), which involves two main processes:

- **Policy Evaluation:** Given a policy π, compute the value function V_π.
- **Policy Improvement:** Update the policy π to π' based on the value function V_π.

Mathematically:

$$\pi \to V_\pi$$
$$V_\pi \to \pi'$$

Monte Carlo Methods

In contrast to Dynamic Programming, Monte Carlo methods do not require a model of the environment (i.e., transition probabilities P). Instead, they rely on direct experience obtained from episodes. The key aspects of Monte Carlo methods are:

- **Action-Value Function:** We estimate the action-value function Q_π over state-action pairs (s, a).
- **Experience from Episodes:** Monte Carlo methods learn from episodes of experience, which are sequences of states, actions, and rewards.
- **Exploration:** To ensure comprehensive learning, exploration must cover the entire state-action space $S \times A$.

Two foundational pillars of Reinforcement Learning (RL) are:

- **Experience:** Learning from episodes of interaction with the environment.
- **Exploration:** Ensuring all state-action pairs are visited to improve the policy effectively.

The policy π can be derived from the action-value function Q_π by selecting actions that maximize Q_π:

$$\pi(s) = \arg\max_a Q_\pi(s, a) \tag{65}$$

Exploration

Exploration in GPI and Monte Carlo Methods

In both Generalized Policy Iteration (GPI) and Monte Carlo (MC) methods, the goal is to find the optimal policy π^*. Effective exploration ensures that the entire state-action space $S \times A$ is adequately experienced through episodic interactions.

How to Ensure Comprehensive Exploration

To ensure comprehensive exploration of the state-action space, several strategies can be employed:

- **Exploratory Start:** Begin episodes from a variety of initial states to cover different parts of the state space.
- **Epsilon-Soft Policy:** Use an ϵ-soft policy where actions are selected according to an ϵ-greedy strategy. This involves using a Q-value table to decide actions.

Policy Strategies

- **Greedy Policy:** Select actions purely based on the argmax of the action-value function:

$$\pi(s) = \arg\max_a Q_\pi(s, a) \tag{66}$$

- **Epsilon-Greedy Policy:** With probability $1 - \epsilon$, choose the action that maximizes Q_π, and with probability ϵ, select a random action. This can be expressed as:

$$\pi(s) = \begin{cases} \arg\max_a Q_\pi(s, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{67}$$

- **Epsilon Softness:** Ensure that no action is selected with zero probability, thus making the policy ϵ-soft. This ensures that every action has a non-zero chance of being chosen, which helps in better exploration of the state-action space.

Conflict Between GPI and Epsilon Softness

In GPI, the aim is to find the optimal policy π^*. However, incorporating ϵ-softness in policy evaluation introduces a conflict:

- The use of ϵ-soft policies in evaluation does not guarantee the discovery of the optimal policy.
- While ϵ-softness ensures exploration and prevents any action from having zero probability, it may prevent the policy from converging to the optimal policy π^* if ϵ is not sufficiently small.

Thus, while ϵ-softness aids in exploration, achieving the optimal policy π^* requires balancing exploration with the need for precise evaluation, which can be challenging.

On-Policy and Off-Policy Methods

On-Policy Methods

In on-policy methods, the agent learns about the policy that it uses to generate the data. Specifically, this involves learning the value function for the policy that is actively used during control. The key characteristics are:

- The policy being evaluated and improved is the same policy used to generate behavior.
- On-policy methods directly estimate the value function for the current policy while interacting with the environment.

This approach simplifies the learning process as it does not require dealing with the complexity of separating behavior and target policies.

Off-Policy Methods

Off-policy methods, on the other hand, decouple the policy being optimized (the target policy π) from the policy used to generate the data (the behavior policy b). The key characteristics are:

- The agent learns about a policy from data generated by following a different policy.
- For off-policy learning to be effective, the behavior policy b must ensure that all actions possible under the target policy π are also possible under b. This is known as the coverage condition:

$$b(a|s) > 0 \text{ whenever } \pi(a|s) > 0 \tag{68}$$

- When this condition is satisfied, the action-value function Q_π can be approximated from data generated by the behavior policy b by applying a correction factor known as the importance sampling ratio:

$$\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \tag{69}$$

Importance Sampling Ratio

The importance sampling ratio ρ_t adjusts for the discrepancy between the behavior policy b and the target policy π. It ensures that the value estimates are accurate despite the fact that the data was generated under a different policy.

Generalization

Off-policy learning is a strict generalization of on-policy learning:

- When the behavior policy is the same as the target policy ($b = \pi$), off-policy methods reduce to on-policy methods.
- Off-policy methods provide greater flexibility by allowing learning from data generated by different policies, making them applicable in more varied scenarios.

Problems with Monte Carlo Methods

High Latency

- The necessity to experience many episodes means that the learning process can be slow, particularly for complex environments.
- Since each episode is typically required to be complete before updates are made, the method may be inefficient for tasks that require a large number of episodes to cover all possible states and actions.

Stationary Probability Systems

- If the environment is non-stationary, meaning the transition probabilities or reward functions change over time, Monte Carlo methods may struggle to provide accurate estimates of the value functions.
- The requirement for episodes to be long enough to average out to the true return may not hold in environments where dynamics are constantly changing.

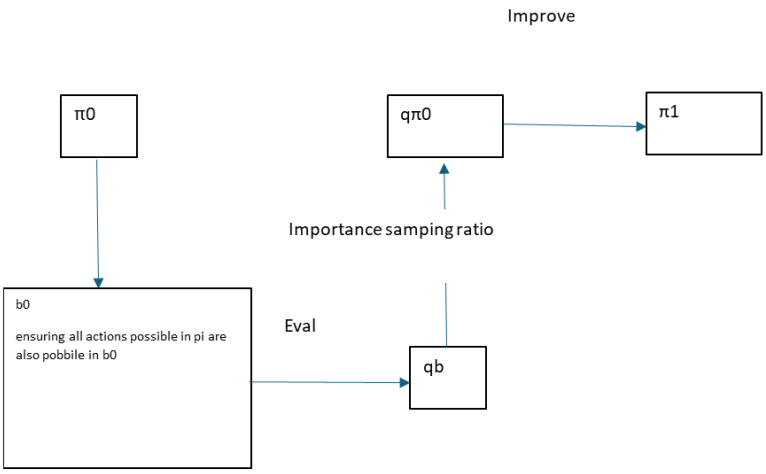

Figure 14: Off Policy technique

Estimating True Average Return

- The estimated return may be influenced by the variability in the episodes, leading to high variance in the return estimates.
- Convergence to the true average return requires a sufficiently large number of episodes, which may be impractical in real-time applications.

Improving Averages

Stationary Situations

In stationary situations, where the statistical properties of the process do not change over time, an incremental formula can be used to update the average. The formula is given by:

$$u_i = u_{i-1} + \frac{1}{i}\left(x_i - u_{i-1}\right) \tag{70}$$

where u_i is the estimate of the process average at step i, and x_i is the new observation at step i. This formula updates the average u_i incrementally with each new observation.

Nonstationary Situations

In nonstationary situations, where the statistical properties of the process may change over time, a different approach is needed to account for recent changes. An exponential moving average (EMA) or Infinite Impulse Response (IIR) filter can be used:

$$u_i = u_{i-1} + \alpha\left(x_i - u_{i-1}\right) \tag{71}$$

where α is a smoothing factor between 0 and 1, and u_i is the updated estimate of the process average at step i.

In this method:

- α controls the weighting of the most recent observation relative to the previous estimate.
- A higher α gives more weight to recent observations, making the estimate more responsive to changes in the process.
- This approach is like an IIR filter, which allows the average to adapt to new data more flexibly.

Temporal Difference TD(0) Technique

- **Requirement:**
  - TD(0) only needs the next state, which can be obtained from the environment without needing a model.
  - Does not require a model (unlike Dynamic Programming).
  - Online and incremental (unlike Monte Carlo methods).
- **Convergence:**
  - TD(0) usually converges faster than Monte Carlo methods.
- **TD(0) Update Rule:**
  - The update for the value function V at time i is:

$$V(S_i) \leftarrow V(S_i) + \alpha \left[R_{i+1} + \gamma V(S_{i+1}) - V(S_i)\right] \tag{72}$$

    where:
    - R_{i+1} is the reward received after taking action in state S_i.
    - γ is the discount factor.
    - α is the learning rate.
- **Modified Monte Carlo:**
  - Uses one true experience with the value function for the remaining reward.
- **Bootstrapping:**
  - Exploits existing information to update the value function.
- **Advantages:**
  - Learn $Q(S, A)$ a lot faster with TD(0).

SARSA

- **Update Rule:**

$$Q(S_i, A_i) \leftarrow Q(S_i, A_i) + \alpha \left[R_{i+1} + \gamma Q(S_{i+1}, A_{i+1}) - Q(S_i, A_i) \right] \tag{73}$$

  where:

  - $Q(S_i, A_i)$ is the current estimate of the action-value function for state S_i and action A_i.
  - R_{i+1} is the reward received after taking action A_i in state S_i.
  - γ is the discount factor.
  - α is the learning rate.
  - $Q(S_{i+1}, A_{i+1})$ is the action-value for the next state S_{i+1} and action A_{i+1}.

- **SARSA Overview:**

  - SARSA stands for State-Action-Reward-State-Action.
  - It updates the action-value function based on the action taken in the next state.
  - It performs General Policy Improvement (GPI) with Temporal Difference (TD) learning.
  - The policy is improved every time step.

Section: 5-1
Notes from Textbook Readings

Chapter 6 Temporal-Difference Learning

- **Temporal-Difference (TD) Learning:**
  - Central and novel idea in reinforcement learning.
  - Combination of Monte Carlo and Dynamic Programming (DP) ideas.
  - **Key Characteristics:**
    - **Like Monte Carlo Methods:**
      - Learn directly from raw experience.
      - No model of the environment's dynamics required.
    - **Like Dynamic Programming:**
      - Update estimates based on other learned estimates.
      - Bootstrap estimates without waiting for a final outcome.
- **Relationships:**
  - TD methods blend ideas from both Monte Carlo and Dynamic Programming.
  - Methods can be combined in various ways.

Focus Areas

- **Policy Evaluation or Prediction Problem:**
  - Estimating the value function v_π for a given policy π.
- **Control Problem (Finding an Optimal Policy):**
  - Uses variations of Generalized Policy Iteration (GPI).
  - Differences in methods are primarily related to approaches to the prediction problem.

Temporal-Difference (TD) Prediction

- **Comparison of TD and Monte Carlo Methods:**
  - Both methods use experience to solve the prediction problem.
  - Given experience following a policy π, both update their estimate V of v_π for nonterminal states S_t.
  - **Monte Carlo Methods:**
    - Wait until the return following the visit is known.
    - Use that return as a target for $V(S_t)$.

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right] \tag{74}$$

    - Here, G_t is the actual return following time t, and α is a constant step-size parameter.
  - **Temporal-Difference (TD) Methods:**
    - Update estimates without waiting for a final outcome (bootstrapping).
    - Need only wait until the next time step to make a useful update.

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \tag{75}$$

    - At time $t+1$, form a target using the observed reward R_{t+1} and the estimate $V(S_{t+1})$.
    - The TD(0) method uses the target $R_{t+1} + \gamma V(S_{t+1})$.

- **Bootstrapping in TD Methods:**
  - TD methods bootstrap, like DP, by updating estimates based on other learned estimates.
  - **Comparison with Monte Carlo and DP:**
    - **Monte Carlo Target:**

$$G_t \text{ (Sample Return)} \tag{76}$$

    - **DP Target:**

$$R_{t+1} + \gamma V(S_{t+1}) \tag{77}$$

    - TD targets are estimates due to sampling and current estimates used instead of true values.

- **TD Error:**
  - The quantity in brackets in the TD(0) update is called the TD error.

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{78}$$

  - TD error measures the difference between the estimated value of S_t and a better estimate $R_{t+1} + \gamma V(S_{t+1})$.

- **Relationship Between Monte Carlo and TD Errors:**
  - If V does not change during the episode (as in Monte Carlo methods), then:

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k \tag{79}$$

  - The identity approximates the Monte Carlo error as a sum of TD errors.
  - Generalizations of this identity are important in the theory and algorithms of TD learning.

Advantages of TD Prediction Methods

- **Bootstrapping:**
  - TD methods update their estimates based in part on other estimates.
  - They learn a guess from a guess—they bootstrap.

- **Advantages over Dynamic Programming (DP) Methods:**

  - TD methods do not require a model of the environment, including its reward and next-state probability distributions.

- **Advantages over Monte Carlo Methods:**
  - TD methods are implemented in an online, fully incremental fashion.
  - Monte Carlo methods require waiting until the end of an episode to determine returns, while TD methods only need to wait one time step.
  - TD methods are useful for applications with long episodes or continuous tasks where episodes do not exist.
  - TD methods can continue learning from each transition regardless of subsequent actions, unlike some Monte Carlo methods which may ignore or discount episodes with experimental actions.

- **Convergence and Efficiency:**
  - TD methods are proven to converge to v_π for a fixed policy π.
  - Convergence is guaranteed in the mean for a constant step-size parameter if it is sufficiently small.
  - With a step-size parameter decreasing according to stochastic approximation conditions, convergence is guaranteed with probability 1.
  - Theoretical results mostly apply to the table-based case but some also apply to general linear function approximation.

- **Practical Considerations:**
  - Although TD and Monte Carlo methods are both asymptotically convergent, it is currently an open question which method converges faster.
  - No mathematical proof currently establishes one method's superiority in convergence speed.
  - Empirically, TD methods have often been found to converge faster than constant-α MC methods on stochastic tasks.

Sarsa: On-policy TD Control

- **Overview:**
  - Sarsa is an on-policy TD control method.
  - It follows the pattern of Generalized Policy Iteration (GPI) but uses TD methods for the evaluation or prediction part.
  - The primary task is to learn an action-value function $Q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a.

- **Learning Process:**
  - Transition from state–action pair to state–action pair is considered, as opposed to state to state.

  - The value of state–action pairs is learned using TD methods.

- **Sarsa Update Rule:**
  - The update rule for Sarsa is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right] \tag{80}$$

  - This update is done after every transition from a nonterminal state S_t.
  - If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero.

- **Algorithm Name:**
  - The algorithm is named Sarsa, based on the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that makes up each transition.

- **On-policy Control Algorithm:**
  - Continually estimate Q_π for the behavior policy π.
  - Simultaneously change π toward greediness with respect to Q_π.

- **Convergence Properties:**
  - Sarsa converges with probability 1 to an optimal policy and action-value function.
  - Convergence is guaranteed under usual conditions on the step sizes.
  - All state–action pairs must be visited an infinite number of times.
  - The policy must converge in the limit to the greedy policy, which can be arranged with ϵ-greedy policies by setting $\epsilon = 1/t$.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  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)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 15: Sarsa (on-policy TD control)

Q-learning: Off-policy TD Control

  - Q-learning is an off-policy TD control algorithm developed by Watkins (1989).
  - It directly approximates the optimal action-value function Q_*, independent of the policy being followed.

- **Q-learning Update Rule:**
  - The update rule for Q-learning is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{81}$$

- **Key Features:**
  - The learned action-value function Q approximates Q_*, the optimal action-value function.
  - The algorithm is independent of the policy being followed.
  - Simplifies analysis and enables early convergence proofs.
- **Policy Impact:**
  - The policy determines which state–action pairs are visited and updated.
  - Correct convergence requires that all state–action pairs continue to be updated.
- **Convergence:**
  - Under the assumption that all state–action pairs are visited and updated, and with appropriate conditions on the step-size parameters, Q converges with probability 1 to Q_*.
- **Backup Diagram:**
  - The update rule affects a state–action pair, so the top node (root of the update) is a small, filled action node.
  - The update is from action nodes, maximizing over all possible actions in the next state.
  - The bottom nodes in the backup diagram are all these action nodes.
  - The maximum of these "next action" nodes is indicated with an arc across them.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
  Initialize S
  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'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 16: Q - Learning

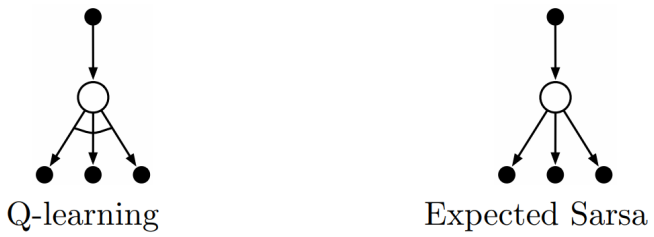

Figure 17: Backup diagrams for Q-learning and Expected Sarsa

Expected Sarsa

  - Expected Sarsa is similar to Q-learning but uses the expected value over next state-action pairs instead of the maximum.

- **Expected Sarsa Update Rule:**
  - The update rule for Expected Sarsa is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma E_\pi [Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right] \tag{82}$$

  - Where:

$$E_\pi [Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] = \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) \tag{83}$$

- **Key Features:**
  - Expected Sarsa moves deterministically in the same direction as Sarsa moves in expectation.
  - It eliminates the variance due to the random selection of A_{t+1}.
  - Generally performs slightly better than Sarsa given the same amount of experience.

- **Performance:**
  - Expected Sarsa retains the significant advantage of Sarsa over Q-learning in tasks like cliff-walking.
  - Shows a significant improvement over Sarsa across a wide range of step-size parameters α.
  - Can set $\alpha = 1$ in deterministic state transitions without degrading asymptotic performance.

- **On-policy and Off-policy Use:**
  - Expected Sarsa can be used on-policy, as in the cliff-walking example.
  - Can also be used off-policy, with a different policy π to generate behavior.
  - When π is the greedy policy and behavior is more exploratory, Expected Sarsa becomes exactly Q-learning.

- **Advantages:**
  - Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa.
  - May completely dominate both Sarsa and Q-learning, except for the small additional computational cost.

Section: 6
TD(n)

Lecture # 6 Date: June 15, 2024

Monte Carlo (MC): $Q(S, A)$

- **Q(S, A):** Represents how good it is to be in a particular state S and take a particular action A. This is also known as the action-value function.
- **Policy Derivation:** From $Q(S, A)$, a policy can always be derived.
- **Greedy Policy:** Always choose the action that maximizes Q for a given state.

$$\pi(s) = \arg\max_a Q(s, a) \tag{84}$$

- ϵ-greedy Policy: Best action with high probability, while exploring other actions with small probability ϵ.

Monte Carlo (MC) Characteristics

- MC requires pure experience and hence, we need to wait until the end of an episode to update values.

Temporal-Difference (TD) with 1-step

- Take one step of the real experience and then exploit the data.
- The return G is calculated as:

$$G = R_{i+1} + \gamma Q(S_{i+1}, A_{i+1}) \tag{85}$$

- Bootstrapping: Taking information from Q, thus exploiting the information.
- Result of our learning: The value function Q.
- Update rule:

$$Q(S, A) \leftarrow Q(S, A) + \alpha\,(\text{Error}) \tag{86}$$

  where Error is derived from new experience.

Algorithm 3 TD with 1-step

1: Take one step of the real experience and then exploit the data
2: $G = R_{i+1} + \gamma Q(S_{i+1}, A_{i+1})$
3: **Bootstrapping:** Taking info from Q exploitation of information.
4: Result of our learning: Value function Q.
5: $Q(S, A) \leftarrow Q(S, A) + \alpha$ (Error) where Error is derived from new experience.

E-Sarsa (Expected Sarsa)

- Expected Sarsa is similar to Q-learning, but instead of taking the maximum over next state-action pairs, it uses the expected value.
- This algorithm takes into account how likely each action is under the current policy, leading to a smoother learning process.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{87}$$

- R_{t+1} is the reward received at time $t+1$, S_{t+1} is the next state.
- $\pi(a \mid S_{t+1})$ is the probability of taking action a in state S_{t+1} under the current policy π.

$$G = R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) \tag{88}$$

Off-Policy Learning

Exploratory Policy b

- The exploratory policy b is often an ϵ-greedy version of the target policy π.
- ϵ-greedy policy: With high probability, select the action that maximizes $Q(S, A)$. With small probability ϵ, select a random action to ensure exploration.

Optimizing Policy π

- The optimizing policy π is the policy that selects actions based on the maximum action-value function: $\pi = \arg\max_a Q(S, A)$.

Update Law for TD 1-Step

The TD 1-step update rule is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_\pi(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{89}$$

Q-Learning

- In Q-learning, we do not need to convert the exploratory policy b to the target policy π (no importance sampling).
- Q-learning updates the action-value function Q using the maximum reward of the next state-action pair.
- This allows Q-learning to learn the optimal policy independently of the actions taken by the agent.

The Q-learning update rule is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{90}$$

Why Q-Learning is Off-Policy and Doesn't Need Importance Sampling

- Q-learning is off-policy because it learns the value of the optimal policy independently of the agent's actions.
- It uses the maximum estimated future rewards to update the Q-values, which means it does not require importance sampling.
- In Monte Carlo (MC) methods, there is no exploitation of existing information, and each episode is treated fresh. However, in Temporal Difference (TD) methods, we use bootstrapping, allowing us to learn Q values associated with π directly in one-step cases.

TD with n-Steps

Bias in Samples

- Samples from returns (E) are low bias.
- Samples from learned knowledge of Q (M) are high bias.
- By taking more data from real experiences (a low bias source), we reduce any errors or bias in bootstrapping.

n-Step TD Algorithms

On-Policy

The n-step return for on-policy TD is given by:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n}) \tag{91}$$

where $n \geq 0$ and $0 \leq t < T - n$.

Off-Policy

For off-policy n-step TD, we use importance sampling to adjust for the difference between the behavior policy b and the target policy π. The update rule is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \left[G_{t:t+n} - Q(S_t, A_t) \right] \tag{92}$$

Comparison Table

Method	**Bias**	**Bootstrapping**	**Latency**
Monte Carlo (MC)	Low Bias	No Bootstrapping	High Latency
TD n-Step	Allows trade-off between bias and latency	Bootstrapping	Variable (depends on n)
1-Step TD	High Bias	Bootstrapping	1-Step Latency

Table 2: Comparison of Monte Carlo, TD n-Step, and 1-Step TD methods.

Tree Backup

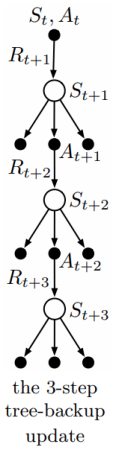

Figure 18: Tree-Backup Update

- **Information in RL is Experience:**
  - In Reinforcement Learning (RL), the core of learning is based on the experience the agent accumulates. This experience includes:
    - **Real-Time Reward:** The immediate feedback or reward received after taking an action in a particular state.
    - **Learned Q Information:** The estimated value of state-action pairs, which reflects the expected future rewards based on prior experiences and updates.
- **Exploiting Information in Q for Actions Not Taken:**
  - Tree Backup involves using the information in the Q-values to evaluate actions that were not actually taken.
  - This method incorporates the potential future rewards of these unchosen actions and integrates them into the learning process.
- **Weighting Actions Taken According to Policy π:**

  - The approach involves weighting the value updates based on the current policy π, which determines the probability of taking each action.
  - By considering the actions that were actually taken and integrating this with the values of actions that were not taken, Tree Backup helps to refine the value function Q_π associated with the policy π.

- **Integrating Information from All Actions:**
  - Tree Backup integrates the information from all possible actions, both those taken and those not taken, to provide a more comprehensive update of the value function.
  - This integration allows the value function Q_π to be more accurate and reflective of the expected rewards under the current policy π.

Section: 6-1
Notes from Textbook Readings

Chapter 7 n-step Bootstrapping

- **Unification of Methods:** n-step TD methods generalize both Monte Carlo (MC) and one-step temporal-difference (TD) methods.
- **Spectrum of Methods:** n-step methods range from MC methods to one-step TD methods, with optimal methods often lying between these extremes.
- **Flexibility in Time Steps:** n-step methods allow bootstrapping over multiple steps, overcoming the limitations of fixed time intervals in one-step TD methods.
- **Faster Action Updates:** They enable quicker action updates while maintaining effective bootstrapping over longer periods.
- **Introduction to Eligibility Traces:** n-step methods introduce the concept of bootstrapping over multiple time intervals, leading to the study of eligibility traces.
- **Application Areas:** Initially applied to prediction problems (estimating v_π), then extended to control methods for action values.

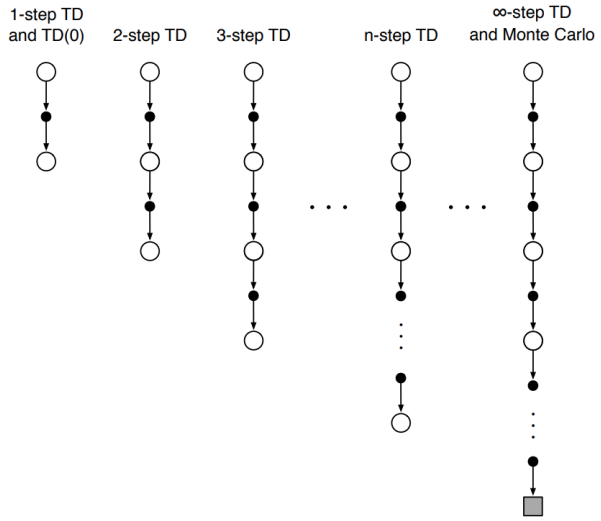

Figure 19: The backup diagrams of n-step methods

n-step TD Prediction

- **Definition of n-step Methods:** n-step TD methods generalize Monte Carlo and one-step TD methods by performing updates based on an intermediate number of rewards (more than one but less than the full sequence until termination).

- **Flexibility in Time Steps:** n-step TD methods provide flexibility by allowing bootstrapping over multiple steps, overcoming the limitation of having a single time step as in one-step TD methods.
- **Intermediate Updates:** A two-step update involves rewards up to two steps later plus the estimated value of the state two steps ahead. Similarly, n-step updates involve rewards up to n steps ahead.
- **Generalization of Update Targets:**

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T \tag{93}$$

  represents the full return in Monte Carlo methods.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{94}$$

  represents the n-step return.
- **n-step TD Update:** The n-step TD update is defined by integrating rewards over n steps and correcting for remaining rewards using the value function V_{t+n-1}.
- **Error Reduction Property:** The n-step return has a guaranteed reduction in error compared to the previous value function, making it a more reliable estimate of v_π.
- **Convergence Guarantee:** n-step TD methods, including one-step TD and Monte Carlo methods, form a family of methods with formal guarantees of convergence under appropriate conditions.
- **Application of n-step Methods:** n-step methods can be used for both prediction and control problems, with the ability to adapt to different scenarios by choosing the appropriate number of steps.

n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$
  Until $\tau = T - 1$

Figure 20: n-step TD

n-step Sarsa

- **On-Policy Control with n-step Methods:** n-step methods can be integrated with Sarsa to produce an on-policy TD control method, which is referred to as n-step Sarsa. This extends the concept of one-step Sarsa (or Sarsa(0)) to handle multiple steps.
- **Action Replacement:** In n-step Sarsa, states are replaced by state-action pairs, and the method employs an ϵ-greedy policy. This approach allows the method to account for the actions taken over multiple steps.
- **Backup Diagrams:** The backup diagrams for n-step Sarsa consist of alternating states and actions, beginning and ending with an action, unlike n-step TD which starts and ends with states.
- **n-step Sarsa Update:** The n-step Sarsa update is given by:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right] \tag{95}$$

  where $Q_{t+n}(S_{t+n}, A_{t+n})$ is the estimated action value at the end of the n-step return.
- **Consistency with One-Step Sarsa:** For states $s \neq S_t$ or actions $a \neq A_t$, the values of all other state-action pairs remain unchanged in the n-step Sarsa update: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$.
- **Expected Sarsa Comparison:** The n-step version of Expected Sarsa, shown in Figure 7.3, involves a linear string of sample actions and states. The final step involves branching over all possible actions, weighted by their probability under the policy π.
- **Expected n-step Sarsa Update:** The update equation for Expected Sarsa is similar to n-step Sarsa but with the n-step return redefined to include the expected value of the actions under the policy:

$$Q_{t:t+n}(S_t, A_t) = R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q_{t+1}(S_{t+1}, a) \tag{96}$$

n-step Off-policy Learning

- Off-policy learning involves learning the value function for one policy, π, while following another policy, b. The policy π is often the greedy policy for the current action-value-function estimate, while b is a more exploratory policy, such as ϵ-greedy.
- In off-policy learning, to use data from b, we must account for the difference between the two policies by using their relative probabilities of taking the actions that were taken.
- For n-step methods, returns are constructed over n steps, so we are interested in the relative probability of those n actions.
- The update for time t (made at time $t+n$) can be weighted by the importance sampling ratio $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1} \left[G_{t:t+n} - V_{t+n-1}(S_t) \right] \tag{97}$$

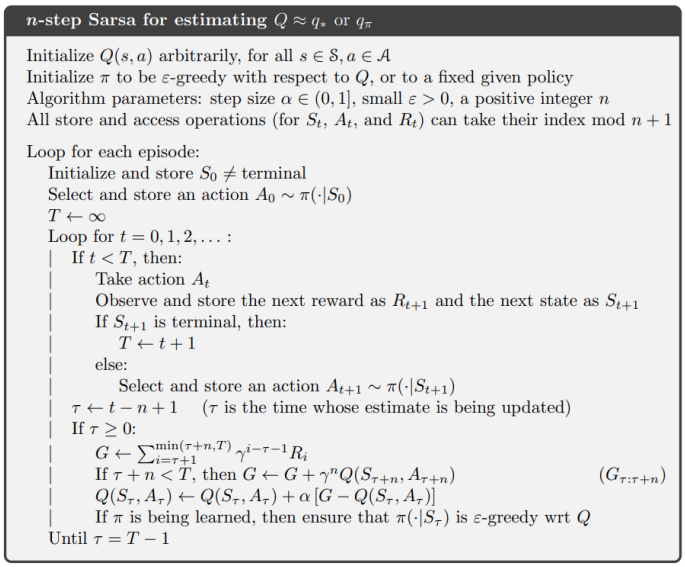

Figure 21: n-step SARSA

where $\rho_{t:t+n-1}$ is the relative probability of taking the n actions from A_t to A_{t+n-1} under the two policies:

$$\rho_{t:t+n-1} = \prod_{k=t}^{t+n-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)} \tag{98}$$

- If any action would never be taken by π (i.e., $\pi(A_k \mid S_k) = 0$), then the n-step return should be given zero weight and ignored. Conversely, if an action is taken that π would take with higher probability than b, it will increase the weight of the return.
- In the on-policy case (where $\pi = b$), the importance sampling ratio is always 1, and thus the update generalizes to the n-step TD update.
- The off-policy version of n-step Expected Sarsa uses the same update as above, but the importance sampling ratio $\rho_{t+1:t+n}$ has one less factor and uses the Expected Sarsa version of the n-step return.

Off-policy Learning Without Importance Sampling

- Off-policy learning can be performed without importance sampling using the n-step Tree Backup algorithm. This method extends concepts from one-step Q-learning and Expected Sarsa to the multi-step case.
- The n-step Tree Backup algorithm uses a tree structure where the central spine represents sampled states and actions, and the branches represent unselected actions. The target for the update includes the estimated values of all actions, not just those sampled.

Off-policy n-step Sarsa for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t+1$
      else:
        Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ ($\rho_{\tau+1:\tau+n}$)
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ ($G_{\tau:\tau+n}$)
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\left[G - Q(S_\tau, A_\tau)\right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

Figure 22: Off policy n-step Sarsa

n-step Tree Backup for estimating $Q \approx q_*$ or q_π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose an action A_0 arbitrarily as a function of S_0; Store A_0
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$:
      Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
      If S_{t+1} is terminal:
        $T \leftarrow t+1$
      else:
        Choose an action A_{t+1} arbitrarily as a function of S_{t+1}; Store A_{t+1}
    $\tau \leftarrow t + 1 - n$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      If $t+1 \geq T$:
        $G \leftarrow R_T$
      else
        $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
      Loop for $k = \min(t, T-1)$ down through $\tau + 1$:
        $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma\pi(A_k|S_k) G$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\left[G - Q(S_\tau, A_\tau)\right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

Figure 23: n-step Tree Backup

- In the tree-backup update, the target combines rewards and estimates from all nodes, including unselected actions, weighted by their probabilities under the target policy π.
- Specifically, the update involves contributions from the leaf nodes of the tree, with weights proportional to their probability of occurrence under the policy π. The weight of each non-selected action is determined by the probability of the selected actions that lead to it.
- The n-step Tree Backup update can be viewed as alternating between sample half-steps (from an action to the next state) and expected half-steps (considering all possible actions from that state, weighted by their probability under π).

$$G_{t:t+n} = Q(S_t, A_t) + \sum_{k=t+1}^{\min(t+n-1,T-1)} \delta_k \prod_{i=t+1}^{k} \gamma\pi(A_i|S_i), \tag{99}$$

$$\delta_k = R_{t+1} + \gamma \bar{V}_t(S_{t+1}) - Q(S_t, A_t) \tag{100}$$

Section: 7
Function Approximation (Supervised Learning)

Lecture # 7 Date: July 6, 2024

Recap (Course Part 1)

- Learning value functions involves obtaining experience by exploring many state-action pairs.
- By analyzing the long-term consequences of these state-action pairs, we observe the sequence of rewards that follow each pair.
- We compute the return, which represents the cumulative reward from a particular state-action pair onward.
- The value function is defined as the expected value of this return.
- The objective of Part 1 is to gather high-quality experience to accurately compute the returns.

Part 2 of the Course

- Addresses challenges posed by large or continuous state spaces.
- Continuous state spaces need to be discretized for practical use in table-based methods.
- Large tables face scalability issues, necessitating more practical data structures and schemes.
- Introduces the concept of function approximation to move away from large tables and towards more scalable solutions.

Function Representation

Closed Form

$$y = x^2 + c \tag{101}$$

- Some methods use a table and interpolate between points to find values not stored in the table.
- Interpolation is a form of function approximation.
- When only a table of values or a small subset of values is available, function approximation can be used.
- Function approximation aims to achieve something close to a closed-form expression.
- Rather than producing a true closed-form expression, it yields an approximate expression that approximates the operational ability of the closed form.
- Function approximation techniques are often used in experimental settings with noisy data to draw a line of best fit, using concepts like slope and intercept.
- Advanced forms of function approximation are learned in university courses, such as Fourier transforms and Laplace transforms.

Function Approximations Instead of Tables

- Tables are not scalable and become impractical for continuous spaces.
- Continuous spaces require discretization, which involves challenges in determining the appropriate granularity.
- In modern contexts, function approximation is often referred to as supervised learning.

Basis: Linear Regression

- Linear regression is the process of finding the line of best fit.
- Hypothesis: There exists a linear function that maps input to output.

$$y = w \cdot x + b \tag{102}$$

- w represents the slope, generalized to the vector case.
- b is the intercept and is a real number.
- The model has N weights and one bias, constituting $N + 1$ parameters.

Goal

- Find w and b such that $y(i) = w \cdot x(i) + b \approx t(i)$ for all i.

$$y(i) - t(i) \tag{103}$$

- We cannot use just the difference due to the potential for positive and negative differences to cancel out.

$$\frac{1}{2}(y(i) - t(i))^2 \tag{104}$$

- We aim to minimize the overall error, not just the individual errors.

$$E = \frac{1}{M} \sum_{i=1}^{M} \frac{1}{2}(y(i) - t(i))^2 \tag{105}$$

$$E = \frac{1}{M} \sum_{i=1}^{M} \frac{1}{2}(w^T x(i) + b - t(i))^2 \tag{106}$$

Algorithm 4 Gradient Descent for Linear Regression

Initialize w and b with w_0 and b_0
while True **do**
  $w \leftarrow w - \alpha \frac{\partial E}{\partial w}$
  $b \leftarrow b - \alpha \frac{\partial E}{\partial b}$
  if $\frac{\partial E}{\partial w} = 0$ and $\frac{\partial E}{\partial b} = 0$ **then**
    break
  end if
  Compute $E = \frac{1}{M} \sum_{i=1}^{M} \frac{1}{2}(y(i) - t(i))^2$
end while

Nonlinear Regression

Polynomial Regression

- Polynomial regression is a form of nonlinear regression.
- General form:

$$y = b_0 + b_1 x_1 + b_2 x_1^2 + \ldots + b_n x_1^n \tag{107}$$

- In polynomial regression, we transform our input X into a series of polynomial basis functions.
- Challenges of polynomial regression:
  - It is not scalable.
  - No clear guidance on the number of polynomial terms to include.
  - Computationally costly and unscalable.
  - Polynomials are not good for handling discontinuities.
- The process involves transforming X into a vector Z and then performing linear regression on Z.
- Two key steps in the transformation:
  - Applying a nonlinear transform to X.
  - Introducing nonlinearity to the model.
- We need a method to transform x to a domain where the transform is nonlinear.

Parametrization

- Adding parameters: weights (w) and biases (b).
- Introducing nonlinearity: Apply a nonlinear function to the output of each function.
- The composition of a nonlinear function with a linear function results in a nonlinear function.
- Common nonlinear functions:
  - ReLU (Rectified Linear Unit)

  - Leaky ReLU
- The training algorithm adjusts the parameters based on gradients. This forms the basis of neural networks.
- Adding more layers to the network increases its depth, leading to deep learning.
- Each additional layer does not require prior information about the actual function.
- Neural networks are the most scalable method for function approximation currently known.
- Issues with neural networks:
  - No definitive way to design the optimal neural network architecture.

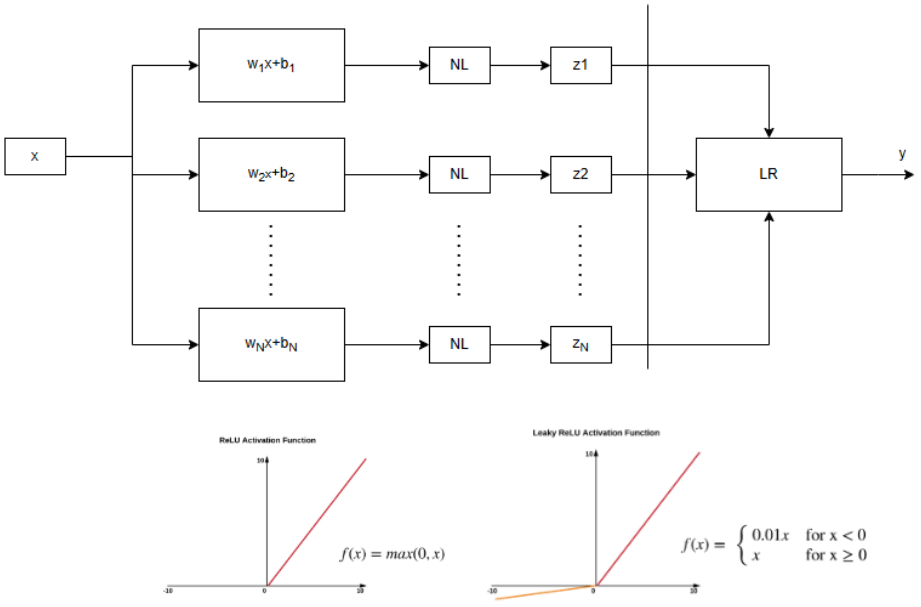

Figure 24: Idea of Neural Networks

Motivation

- Transform the input x through a nonlinear, parameterized, and dimensional transformation to obtain z.
- Apply linear regression on z to produce the output y.

$$x \rightarrow \text{nonlinear, parameterized, dimensional transformation} \rightarrow z$$
$$z \rightarrow \text{Linear Regression} \rightarrow y$$

Section: 7-1
Notes from Textbook Readings

Chapter 9 On-policy Prediction with Approximation

- We study function approximation in reinforcement learning, focusing on estimating the state-value function from on-policy data.
- The approximate value function is represented as a parameterized functional form with weight vector $\mathbf{w} \in R^d$.
- Notation: $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$.
- Examples of $\hat{v}$:
  - Linear function in features of the state, with $\mathbf{w}$ as the vector of feature weights.
  - Multi-layer artificial neural network, with $\mathbf{w}$ as the vector of connection weights.
  - Decision tree, with $\mathbf{w}$ defining the split points and leaf values.
- Typically, the number of weights d is much less than the number of states $|S|$ ($d \ll |S|$).
- Changing one weight can affect the estimated values of many states, leading to generalization.
- Generalization can make learning more powerful but also more challenging.
- Function approximation extends reinforcement learning to partially observable problems.
- If the parameterized function form for $\hat{v}$ does not allow the estimated value to depend on certain aspects of the state, it is as if those aspects are unobservable.
- Theoretical results for methods using function approximation apply equally well to cases of partial observability.
- Function approximation cannot augment the state representation with memories of past observations.

Value-function Approximation

- Prediction methods update an estimated value function by shifting its value at particular states toward a "backed-up value" or update target for that state.
- Individual updates can be represented as $s \to u$, where s is the state being updated and u is the update target.
- Examples of updates:
  - Monte Carlo update: $S_t \to G_t$
  - TD(0) update: $S_t \to R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t)$
  - n-step TD update: $S_t \to G_{t:t+n}$
  - DP (Dynamic Programming) policy-evaluation update: $s \to E_\pi[R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) \mid S_t = s]$

- Each update specifies an example of the desired input-output behavior of the value function.
- Previously, updates were simple: the table entry for s's estimated value was shifted a fraction of the way toward u, leaving estimated values of all other states unchanged.
- Now, more complex methods can implement updates, causing generalization such that the estimated values of many states change.
- Machine learning methods that mimic input-output examples are called supervised learning methods, and when outputs are numbers, the process is often called function approximation.
- Function approximation methods receive examples of desired input-output behavior for the function they aim to approximate.
- These methods are used for value prediction by passing them the $s \to u$ of each update as a training example.
- The approximate function produced is interpreted as the estimated value function.
- Viewing each update as a conventional training example enables the use of various function approximation methods, including artificial neural networks, decision trees, and multivariate regression.
- Not all function approximation methods are equally suitable for reinforcement learning:
  - Sophisticated neural network and statistical methods often assume a static training set over which multiple passes are made.
  - Reinforcement learning requires learning to occur online, while interacting with the environment.
  - Efficient learning from incrementally acquired data is essential.
  - Handling nonstationary target functions (which change over time) is crucial, especially in control methods based on GPI (Generalized Policy Iteration).
- Methods unsuited for handling nonstationarity are less suitable for reinforcement learning.

The Prediction Objective (VE)

- In the tabular case, a continuous measure of prediction quality was not necessary:
  - The learned value function could come to equal the true value function exactly.
  - The learned values at each state were decoupled—an update at one state affected no other.
- With genuine approximation:
  - An update at one state affects many others.
  - It is not possible to get the values of all states exactly correct.
  - We have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate.
- We must specify which states we care most about:

  - Specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$.
  - Representing how much we care about the error in each state s.

- By the error in a state s, it means the square of the difference between the approximate value $\hat{v}(s, w)$ and the true value $v_\pi(s)$:
  - Weighting this over the state space by μ, we obtain the mean square value error (VE):

$$\overline{\text{VE}}(w) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, w)\right]^2$$

- The square root of this measure, the root VE, gives a rough measure of how much the approximate values differ from the true values and is often used in plots.
- Often $\mu(s)$ is chosen to be the fraction of time spent in s:
  - Under on-policy training, this is called the on-policy distribution.
  - In continuing tasks, the on-policy distribution is the stationary distribution under π.
- The on-policy distribution in episodic tasks:
  - The on-policy distribution is different in episodic tasks because it depends on how the initial states of episodes are chosen.
  - Let $h(s)$ denote the probability that an episode begins in state s.
  - Let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode.
  - Time is spent in a state s if episodes start in s, or if transitions are made into s from a preceding state $\bar{s}$ in which time is spent:

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a), \quad \forall s \in S$$

  - This system of equations can be solved for the expected number of visits $\eta(s)$.
  - The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \forall s \in S.$$

- The two cases, continuing and episodic, behave similarly, but with approximation, they must be treated separately in formal analyses, as we will see repeatedly in this part of the book.
- This completes the specification of the learning objective.
- It is not completely clear that the VE is the right performance objective for reinforcement learning:
  - Remember that the ultimate purpose, the reason we are learning a value function is to find a better policy.
  - The best value function for this purpose is not necessarily the best for minimizing VE.
- An ideal goal in terms of VE would be to find a global optimum:

  - A weight vector w^* for which $\overline{\text{VE}}(w^*) \leq \overline{\text{VE}}(w)$ for all possible w.
  - Reaching this goal is sometimes possible for simple function approximators such as linear ones.
  - Rarely possible for complex function approximators such as artificial neural networks and decision trees.
- Complex function approximators may seek to converge to a local optimum:
  - A weight vector w^* for which $\overline{\text{VE}}(w^*) \leq \overline{\text{VE}}(w)$ for all w in some neighborhood of w^*.
  - Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators.
  - Often, it is enough.
- For many cases of interest in reinforcement learning, there is no guarantee of convergence to an optimum, or even to within a bounded distance of an optimum:
  - Some methods may diverge, with their VE approaching infinity in the limit.
- Outlined a framework for combining a wide range of reinforcement learning methods for value prediction with a wide range of function approximation methods:
  - Using the updates of the former to generate training examples for the latter.
- Described a VE performance measure which these methods may aspire to minimize.
- Considered only a few possibilities:
  - Focus on function approximation methods based on gradient principles.
  - Focus on linear gradient-descent methods in particular.
  - These methods are promising, reveal key theoretical issues, and are simple.
  - Space is limited.

Stochastic-gradient and Semi-gradient Methods

- Develop in detail one class of learning methods for function approximation in value prediction, those based on stochastic gradient descent (SGD).
  - SGD methods are among the most widely used of all function approximation methods.
  - They are particularly well suited to online reinforcement learning.
- In gradient-descent methods:
  - The weight vector is a column vector with a fixed number of real-valued components, $\mathbf{w} \equiv (w_1, w_2, \ldots, w_d)^\top$.
  - The approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in S$.
- Updating $\mathbf{w}$ at each of a series of discrete time steps, $t = 0, 1, 2, 3, \ldots$, so there need a notation $\mathbf{w}_t$ for the weight vector at each step.

- Assume that, on each step, we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy.
- These states might be successive states from an interaction with the environment, but for now, we do not assume so.
- Even though we are given the exact, correct values $v_\pi(S_t)$ for each S_t, there is still a difficult problem because:
  - Our function approximator has limited resources and thus limited resolution.
  - There is generally no $\mathbf{w}$ that gets all the states, or even all the examples, exactly correct.
  - We must generalize to all the other states that have not appeared in examples.
- Assume that states appear in examples with the same distribution μ over which we are trying to minimize the VE.
- A good strategy in this case is to try to minimize the error on the observed examples.
- Stochastic gradient-descent (SGD) methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{108}$$

  - Where α is a positive step-size parameter.
  - $\nabla f(\mathbf{w})$ denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \equiv \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^\top \tag{109}$$

- This derivative vector is the gradient of f with respect to $\mathbf{w}$.
- SGD methods are "gradient descent" methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example's squared error.
- This is the direction in which the error falls most rapidly.
- Gradient descent methods are called "stochastic" when the update is done, as here, on only a single example, which might have been selected stochastically.
- Over many examples, making small steps, the overall effect is to minimize an average performance measure such as the VE.
- It may not be immediately apparent why SGD takes only a small step in the direction of the gradient:
  - Could we not move all the way in this direction and completely eliminate the error on the example?
  - In many cases, this could be done, but usually it is not desirable.
  - We do not seek or expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states.

  - If we completely corrected each example in one step, then we would not find such a balance.

- The convergence results for SGD methods assume that α decreases over time:
  - If it decreases in such a way as to satisfy the standard stochastic approximation conditions, then the SGD method is guaranteed to converge to a local optimum.

- Consider the case in which the target output, here (110) denoted $U_t \in R$, of the t-th training example, $S_t \mapsto U_t$, is not the true value, $v_\pi(S_t)$, but some, possibly random, approximation to it:
  - For example, U_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the bootstrapping targets using $\hat{v}$ mentioned in the previous section.
  - In these cases, we cannot perform the exact update because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting U_t in place of $v_\pi(S_t)$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{110}$$

- If U_t is an unbiased estimate, that is, if $E[U_t|S_t = s] = v_\pi(s)$ for each t, then $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions for decreasing α.

- Suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π:
  - Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t \equiv G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$.
  - With this choice, the general SGD method converges to a locally optimal approximation to $v_\pi(S_t)$.
  - The gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution.
  - Pseudocode for a complete algorithm is shown in the box below.

- One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t in:
  - Bootstrapping targets such as n-step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r} \pi(a|S_t)p(s', r|S_t, a)[r + \gamma\hat{v}(s', \mathbf{w}_t)]$ all depend on the current value of the weight vector $\mathbf{w}_t$, which implies that they will be biased and that they will not produce a true gradient-descent method.

- Bootstrapping methods are not in fact instances of true gradient descent:
  - They take into account the effect of changing the weight vector $\mathbf{w}_t$ on the estimate, but ignore its effect on the target.
  - They include only a part of the gradient and, accordingly, we call them semi-gradient methods.

- Although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section:

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

Figure 25: Gradient Monte Carlo and Semi-gradient TD(0)

  - They offer important advantages that make them often clearly preferred.
  - One reason for this is that they typically enable significantly faster learning.
  - Another is that they enable learning to be continual and online, without waiting for the end of an episode.
  - This enables them to be used on continuing problems and provides computational advantages.

- A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \equiv R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t)$ as its target:
  - Complete pseudocode for this method is given in the box below.
- State aggregation is a simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector $\mathbf{w}$) for each group:
  - The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated.
  - State aggregation is a special case of SGD (9.7) in which the gradient, $\nabla\hat{v}(S_t, \mathbf{w}_t)$, is 1 for S_t's group's component and 0 for the other components.

Linear Methods

- **Function Approximation**
  - The approximate value function $\hat{v}(\cdot, w)$ is a linear function of the weight vector w.
  - For each state s, there is a feature vector $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^\top$.
  - The approximate value function can be written as:

$$\hat{v}(s, w) = w^\top \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s) \tag{111}$$

- **Feature Vector**
  - **Definition:** The feature vector $\mathbf{x}(s)$ represents state s.
  - **Components:** Each component $x_i(s)$ is the value of a function $x_i : S \to R$.
  - **Basis Functions:** Features are basis functions forming a linear basis for the set of approximate functions.
- **Stochastic Gradient Descent (SGD) Update**
  - The gradient of the approximate value function with respect to w is:

$$\nabla_w \hat{v}(s, w) = \mathbf{x}(s) \tag{112}$$

  - Therefore, the SGD update rule is:

$$w_{t+1} = w_t + \alpha \left[U_t - \hat{v}(S_t, w_t)\right] \mathbf{x}(S_t) \tag{113}$$

    where α is the step-size parameter.
- **Convergence and Optimality**
  - **Linear Case:** In the linear case, convergence results are favorable and often guaranteed.
  - For gradient Monte Carlo, convergence to the global optimum of the Value Error (VE) is ensured if α decreases according to standard conditions.
  - **Semi-Gradient TD(0):** Converges to a point near the local optimum, not necessarily the global optimum. The update equation is:

$$w_{t+1} = w_t + \alpha \left[R_{t+1}\mathbf{x}_t - \mathbf{x}_t \left(\mathbf{x}_t^\top \left(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\right)\right)\right] \tag{114}$$

- **TD Fixed Point**
  - At the TD fixed point, the Value Error (VE) is bounded by:

$$VE(w_{TD}) \leq \frac{1}{1-\gamma} \times \text{Smallest Possible Error} \tag{115}$$

    where γ is often close to 1.
- **Comparison with Other Methods**
  - **Monte Carlo Methods:** Typically have higher variance compared to TD methods but are unbiased.
  - **TD Methods:** Often have lower variance and can converge faster but might not achieve the global optimum.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t+1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau,\mathbf{w})]\,\nabla \hat{v}(S_\tau,\mathbf{w})$
  Until $\tau = T - 1$

Figure 26: n-step semi-gradient TD

Nonlinear Function Approximation: Artificial Neural Networks

- **Artificial Neural Networks (ANNs)**
  - ANNs are used for nonlinear function approximation.
  - They consist of interconnected units with properties similar to neurons.
  - ANNs have a rich history, and recent advances in deep learning have significantly enhanced their capabilities.
- **Feedforward Neural Networks**
  - A feedforward ANN has no loops; units do not influence their own input.
  - Example architecture:
    - Input layer with four units
    - Two hidden layers
    - Output layer with two units
  - Each link in the network has a real-valued weight.
- **Units and Activation Functions**
  - Units compute a weighted sum of inputs and then apply a nonlinear activation function.
  - Common activation functions:
    - Sigmoid function:
$$f(x) = \frac{1}{1 + e^{-x}} \tag{116}$$
    - Rectifier function:
$$f(x) = \max(0, x) \tag{117}$$

* Step function:

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{118}$$

- **Function Approximation**
  - ANNs with no hidden layers can only approximate simple functions.
  - ANNs with a single hidden layer and sufficient sigmoid units can approximate any continuous function on a compact input space:

$$\hat{v}(s, w) \approx f(\mathbf{x}(s), w) \tag{119}$$

  - Nonlinearity is essential; if all units are linear, the network reduces to a single-layer network.
- **Deep Architectures**
  - Hierarchical composition in deep ANNs provides better function approximation.
  - Deep ANNs automatically create features appropriate for the problem.
  - Deep architectures are essential for complex AI tasks and are discussed in Bengio (2009).
- **Training and Overfitting**
  - ANNs typically use stochastic gradient methods to adjust weights.
  - In reinforcement learning, ANNs can use TD errors or reward maximization.
  - Training involves estimating the gradient of an objective function with respect to each weight:

$$\text{Gradient} = \nabla_w \text{Objective Function} \tag{120}$$

- **Backpropagation Algorithm**
  - Alternates between forward and backward passes to compute gradients.
  - Forward pass calculates activations; backward pass updates weights.
  - Backpropagation works well for shallow networks but may be less effective for very deep networks.
- **Overfitting and Regularization**
  - Overfitting occurs with many degrees of freedom and limited data.
  - Techniques to reduce overfitting include:
    * Cross-validation
    * Regularization
    * Weight sharing
- **Dropout Method**
  - Randomly removes units and their connections during training.
  - Improves generalization by combining results from "thinned" networks.

- **Other Techniques**
  - **Deep Belief Networks:** Use unsupervised learning for pre-training layers.
  - **Batch Normalization:** Normalizes outputs of layers to improve training.
  - **Residual Learning:** Adds residual functions to simplify learning.
- **Deep Convolutional Networks**
  - Specialized for high-dimensional data like images.
  - Example architecture (LeCun et al., 1998):
    * Alternating convolutional and subsampling layers
    * Convolutional layers produce feature maps
    * Subsampling layers reduce spatial resolution and improve invariance

Key Points

- **Parameterized Function Approximation**
  - Policies in RL are often parameterized by a weight vector $\mathbf{w}$.
  - The state space is much larger than the weight vector, leading to approximate solutions.
  - The Mean Square Value Error (VE) is used to measure the error in value estimates $v_\pi^w(s)$ for a weight vector $\mathbf{w}$ under the on-policy distribution μ.
  - VE helps rank different value-function approximations.
- **Stochastic Gradient Descent (SGD)**
  - Popular methods include variations of SGD for finding a good weight vector.
  - Focus is on policy evaluation or prediction with fixed policies.
  - n-step semi-gradient TD is a natural learning algorithm for this case, including:
    * Gradient Monte Carlo (when $n = 1$)
    * Semi-gradient TD(0) (when $n = \infty$)
  - Semi-gradient methods are not true gradient methods; the weight vector appears in the update target but is not accounted for in the gradient computation.
- **Linear Function Approximation**
  - Well-understood and works well in practice with appropriate features.
  - Features can be chosen as polynomials, Fourier basis functions, or through coarse coding techniques like tile coding.
  - Tile coding is computationally efficient and flexible.
  - Radial basis functions are useful for tasks where a smoothly varying response is needed.
  - LSTD (Least-Squares Temporal Difference) is the most data-efficient linear TD method but requires computation proportional to the square of the number of weights.
  - Other methods have complexity linear in the number of weights.

- **Nonlinear Methods**
  - Include artificial neural networks (ANNs) trained by backpropagation and variations of SGD.
  - Popular in recent years as deep reinforcement learning.
- **Convergence and Practical Considerations**
  - Linear semi-gradient n-step TD is guaranteed to converge to a VE within a bound of the optimal error (achieved asymptotically by Monte Carlo methods).
  - The bound is tighter for higher n and approaches zero as $n \to \infty$.
  - Very high n results in slow learning; some degree of bootstrapping ($n < 1$) is usually preferable.

Section: 8
Prediction and Control with F.Approx; DQN

Lecture # 8 Date: July 13, 2024

Overview

- **Supervised Machine Learning**: Method used for approximating functions with neural networks as the dominant contemporary tool. The generic term for this method is function approximation.

Key Concepts

- **Phenomenon P**: Involves inputs X producing outputs T, for example, real numbers in Q-value functions. We conduct experiments to collect data pairs (X_i, T_i), where X is the domain and T is the range.
- **Experiments and Data**: Conduct M experiments to collect data pairs (X_i, T_i). The goal is to approximate the underlying function F_{ref} that maps inputs X_i to outputs T_i.
- **Function Approximation with Neural Networks**: Neural networks are used to approximate F_{ref} by creating a structured model F with parameters (θ) which include all weights and biases.
- **Neural Network Structure**: The network takes input X, processes it through parameter blocks θ, and produces $F(X)$. The complexity of the network can be adjusted by increasing the depth and number of neurons per layer.
- **Training the Neural Network**: The objective is to tune θ to minimize the error, defined as the average squared difference between $F_{\text{ref}}(X_i)$ and $F(X_i)$.

Error Calculation

$$\text{Error} = \frac{1}{2M} \sum_{i=1}^{M} \left(F_{\text{ref}}(x^i) - F'(x^i) \right)^2 \tag{121}$$

- T_i: Actual target value from the dataset.
- $F(\hat{X}_i)$: Predicted output from the neural network.

Steps to Function Approximation

1. **Data Collection**: Gather domain-range pairs (X_i, T_i) through experiments and construct the dataset.
2. **Model Structure**: Define a neural network structure to approximate F_{ref} and aggregate all parameters into θ.
3. **Training**: Define the error as the average squared difference, and use learning algorithms to minimize this error using the dataset.

Important Terms

- **Domain (X)**: Inputs to the function.
- **Range (T)**: Outputs from the function.
- **Parameter Blocks (θ)**: Aggregated weights and biases in the neural network.
- **Adaptation/Learning**: Process of tuning θ to minimize error.

Adaptation Rule for θ

To adapt θ, we use the rule:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \text{Error} \tag{122}$$

where α is a learning rate, and $\nabla_\theta \text{Error}$ is the gradient of the error with respect to θ.

Local Minima and Neural Networks

There's no guarantee of global optimality for a general nonlinear function. If θ converges to θ_{final} where gradients are close to 0 but the error remains high, it suggests a local minimum.

To mitigate local minima:

- Sample initialization points for θ.
- Adjust neural network complexity if stuck at a plateau.

Handling Samples One at a Time

- In reinforcement learning, data is received one sample at a time due to interaction with the environment.
- Unlike batch processing, where we have a complete dataset, we update parameters incrementally with each new sample.
- To update θ based on this error we use (122)
- This incremental update is performed every time a new sample (X_i, T_i) is received from the environment.
- Adjusting α is crucial as it balances the speed of convergence and stability in training, considering the noise introduced by incremental updates.

Stochastic Gradient Descent (SGD)

In reinforcement learning, where data arrives one sample at a time, we employ stochastic gradient descent (SGD) to update model parameters incrementally:

- **Variants of SGD**:
  - **Mini-batch SGD**: Error is computed over a small subset of data samples.

  - **Full-batch Gradient Descent**: Error is computed over the entire dataset in each cycle.

- **Application in Reinforcement Learning**:
  - SGD and its variants are crucial in reinforcement learning due to the incremental nature of data acquisition from the environment.
  - Today, we will explore an application of these concepts, particularly in the context of Deep Q-Networks (DQN).
  - Both mini-batch SGD and full-batch gradient descent find application, influencing training efficiency and convergence in reinforcement learning tasks.

Mapping to Reinforcement Learning

In reinforcement learning, our goal is to approximate the Q-function using a neural network parametrized by θ. Let's map this to our reinforcement learning framework:

- We have X, which corresponds to the state S_t in reinforcement learning.
- $\hat{F}(X)$ represents our prediction $\hat{q}(S_t, A_t; \theta)$, where A_t is the action taken in state S_t.
- θ aggregates all the weights and biases in the neural network.

Our neural network F is denoted as $\hat{q}$, aiming to approximate Q. The learning rule remains consistent across both contexts

Now, let's define the error E_t in the reinforcement learning context:

$$E_t = \frac{1}{2} \left(Q(S_t, A_t) - \hat{q}(S_t, A_t; \theta) \right)^2 \tag{123}$$

To compute $\nabla_\theta E_t$, differentiate with respect to θ:

$$\nabla_\theta E_t = \Big(Q(S_t, A_t) - \hat{Q}(S_t, A_t; \theta) \Big) \nabla_\theta \hat{q}(S_t, A_t; \theta) \tag{124}$$

This gradient guides our learning process in reinforcement learning, where Q is our approximation of the true Q-function, obtained through experience-based techniques such as Monte Carlo or Temporal Difference (TD) learning.

Comparison with Part 1 Techniques

In contrast to Part 1 of the course, where we used tables to store Q-values, Part 2 employs neural networks for function approximation. Here's how the methods differ:

- **Q-Value Computation**:
  - **Part 1 (Table Approach)**: We maintain a table $Q(S, A)$ where each state-action pair has an associated Q-value. This table is updated iteratively as the agent interacts with the environment.
  - **Part 2 (Neural Network Approach)**: Instead of a table, we use a neural network $\hat{Q}(S, A; \theta)$ to approximate the Q-function. The network takes in state S and action A as inputs and outputs a Q-value prediction.

- **Scalability**:
  - **Table Approach**: Requires discretization of the state space S and action space A. This limits scalability, especially for high-dimensional or continuous state-action spaces.
  - **Neural Network Approach**: Handles real-valued, high-dimensional state and action spaces without the need for discretization. Neural networks can scale complexity with the problem size, making them more versatile.
- **Learning and Updating**:
  - **Part 1**: Updates involve directly modifying entries in the Q-table based on experienced rewards and transitions.
  - **Part 2**: Updates are performed by adjusting the neural network parameters θ using gradient descent based on the error between predicted and actual Q-values computed via Monte Carlo or TD methods.
- **Algorithm Integration**:
  - Techniques from Part 1, such as Monte Carlo or TD methods, are directly applicable in Part 2. The main difference lies in how the Q-value updates are applied—directly to a table in Part 1 versus indirectly via parameter updates in a neural network in Part 2.
- **Final State Handling**:
  - In both approaches, when reaching a final state S', the reward R is known (often zero), simplifying the Q-value update.
  - For neural networks, this update modifies θ based on the computed Q-value relative to the neural network's prediction.

Neural Network Approach and Algorithm Structure

Contrasting it with the table-based approach used in Part 1 of the course. Unlike Part 1, where we explicitly stored Q-values in a table, Part 2 utilizes a neural network for function approximation. Let's break down the key aspects and algorithm structure:

- **Neural Network Usage**: In Part 2, instead of maintaining a Q-table, we employ a neural network $\hat{Q}(S, A; \theta)$ that takes the current state S and action A as inputs and outputs a Q-value prediction. This network is updated iteratively based on the agent's interactions with the environment.
- **Q-Value Computation**: The Q-value $Q(S, A)$ is not explicitly computed or stored as in Part 1. Rather, it is implicitly derived from the neural network's output $\hat{Q}(S, A; \theta)$ at each step during the learning process.
- **Algorithm Outline**: After initializing the system and starting an episode, the algorithm proceeds as follows:
  - **Action Selection**: Choose the next action A using an epsilon-greedy strategy with respect to the current $\hat{Q}$-values.
  - **Transition**: Execute the selected action A, observe the reward R and the next state S'.

  - **Update Neural Network**: Adjust the neural network parameters θ using gradient descent. This update is based on the difference between the predicted Q-value $\hat{Q}(S, A; \theta)$ and the actual Q-value, which is either computed via Monte Carlo or Temporal Difference (TD) methods.
  - **State Transition**: Move to the next state S' and repeat the process until the end of the episode.

- **Handling Final States**: When reaching a final state S', typically associated with a terminal condition (e.g., end of the game or episode), the reward R is known (often zero). This simplifies the Q-value update process.
- **Comparison with Part 1**: The neural network approach eliminates the need for discretizing state and action spaces, which was a limitation in the table-based approach. This makes the neural network approach more scalable and applicable to complex, high-dimensional environments.
- **Action Selection with Neural Networks**: To perform an epsilon-greedy selection with a neural network, one queries the network for $\hat{Q}(S, a; \theta)$ for each action a and selects the action with the highest predicted value. In cases where multiple actions have the same highest value, one might apply additional logic.
- **Backpropagation and Training**: The process of updating the neural network θ involves backpropagation, which is simply the application of the chain rule to compute gradients efficiently through the network layers.

Introduction to DQN

- Deep Q-Network (DQN) is a reinforcement learning algorithm that showcases the power of neural networks in learning complex behaviors directly from raw visual input, such as game screens.
- It addresses the challenge of achieving human-level performance in various games without requiring handcrafted features or game-specific knowledge.

Neural Network Architecture

- The term "Deep" in DQN refers to the multi-layered neural network used, which allows for the representation of complex features and relationships within game environments.
- A deep neural network (DNN) typically consists of multiple layers, enabling it to capture intricate patterns that are crucial for learning effective strategies.

Q-Learning

- DQN is based on Q-learning, an off-policy reinforcement learning algorithm. The goal of Q-learning is to learn an optimal action-value function, which estimates the expected future reward when taking action a in state s and following an optimal policy thereafter.

The Q-learning update rule is given by:

$$\theta \leftarrow \theta + \alpha \left[R_{t+1} + \gamma \max_{a'} (\hat{q_\theta}(S_{t+1}, a)) - \hat{q_\theta}(S_t, A) \right] \nabla_\theta \hat{q_\theta}(S_t, A) \tag{125}$$

Making the Problem Markovian

- One of the challenges in applying reinforcement learning to games is dealing with non-Markovian states. Simply using a single game frame as the state does not capture the dynamics and context of the game adequately.
- To address this, DQN employs a technique known as experience replay. It collects a sequence of observations (frames) into a buffer and samples from this buffer to create a more informative state representation. This sequence helps in making the learning problem more Markovian by capturing temporal dependencies and providing richer state information.

Experience Replay

- Experience replay stores tuples of state-action-reward-next state transitions (s, a, r, s') in a replay memory.
- During training, batches of these tuples are sampled randomly to decorrelate the data and provide more stable updates to the neural network parameters.

Target Network

- To further stabilize the training process, DQN introduces a target network. This network is a copy of the main Q-network but with fixed parameters that are updated less frequently.
- It helps to mitigate the issues of moving target and enhances the convergence of the Q-learning process.

Training Stability

DQN addresses the instability issues in training neural networks with Q-learning by:

- Using experience replay to train the network on batches of random transitions.
- Introducing a target network to provide stable targets during learning,
- Clipping the error term to a bounded range to prevent large updates that could destabilize training.

Section: 8-1
Notes from Textbook Readings

Chapter 10 On-policy Control with Approximation

Episodic Semi-gradient Control

- Extend semi-gradient prediction methods to action values.
- Represent the approximate action-value function $\hat{q} \approx q_\pi$ with weight vector $\mathbf{w}$.
- Consider training examples of the form $S_t, A_t \to U_t$.
- The update target U_t can be an approximation of $q_\pi(S_t, A_t)$.
- General gradient-descent update for action-value prediction:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t) \tag{126}$$

- One-step Sarsa update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t) \tag{127}$$

- Method called *episodic semi-gradient one-step Sarsa*.
- Converges like TD(0) with similar error bound for a constant policy.
- Control methods couple action-value prediction with policy improvement and action selection.
- For discrete actions:
  - Compute $\hat{q}(S_{t+1}, a, \mathbf{w}_t)$ for each action a.
  - Find greedy action $A^*_{t+1} = \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$.
  - Use ϵ-greedy policy for policy improvement.
- Actions selected according to the improved policy.
- Pseudocode provided for the complete algorithm.

Semi-gradient n-step Sarsa

- n-step version of episodic semi-gradient Sarsa.
- Use n-step return as the update target in the semi-gradient Sarsa update equation.
- n-step return generalizes from its tabular form to function approximation form:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n}) \quad \text{if } t+n < T \tag{128}$$

$$G_{t:t+n} = G_t \quad \text{if } t+n \geq T \tag{129}$$

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Figure 27: Episodic Semi-gradient Sarsa

- n-step update equation:

$$\mathbf{w}_{t+n} = \mathbf{w}_{t+n-1} + \alpha\left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) \tag{130}$$

- Performance is best with intermediate bootstrapping (n larger than 1).
- Figures show:
  - Faster learning and better asymptotic performance at $n = 8$ than $n = 1$ on the Mountain Car task.
  - Detailed study of the effect of parameters α and n on learning rate for the task.

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t, A_t, and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ε-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take action A_t
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then:
  | $T \leftarrow t + 1$
  | else:
  | Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})\right]\nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$

Figure 28: Episodic semi-gradient n-step Sarsa

Section: 8-2
Notes from Textbook Readings

Human-level Video Game Play

- **Challenge**: Representing and storing value functions and/or policies in reinforcement learning.
- Lookup tables are only feasible for small, finite state sets.
- **Solution**: Use parameterized function approximation schemes.
- Function approximation relies on features that must be:
  - Readily accessible to the learning system.
  - Able to convey necessary information for skilled performance.
- Successful applications often use handcrafted features based on human knowledge and intuition.
- **DeepMind's Contribution**:
  - Developed by Google DeepMind.
  - Used deep multi-layer ANNs to automate the feature design process.
  - Built on the backpropagation algorithm.
  - Coupled reinforcement learning with backpropagation.
- **Examples**:
  - TD-Gammon by Tesauro: Learned to play backgammon at a high level.
  - DQN (Deep Q-Network): Combined Q-learning with a deep convolutional ANN.
- **DQN Details**:
  - Used to play 49 different Atari 2600 video games.
  - Learned different policies for each game with the same raw input, network architecture, and parameters.
  - Achieved human-level performance on many games.
- **Atari 2600 and ALE**:
  - Atari 2600: Classic games, used as a testbed for reinforcement learning.
  - Arcade Learning Environment (ALE): Publicly available platform for using Atari games to study learning and planning algorithms.
- **DQN's Innovations**:
  - Used deep convolutional ANN for processing spatial data like images.
  - Demonstrated learning without game-specific feature sets.

  - Employed techniques such as experience replay and a duplicate target network to improve stability and performance.

- **DQN's Architecture**:
  - Three hidden convolutional layers.
  - One fully connected hidden layer.
  - Output layer representing action values for different actions.
  - Used ϵ-greedy policy for action selection.

- **Training Process**:
  - Preprocessed input frames to reduce dimensions.
  - Used stacks of four most recent frames as input.
  - Applied semi-gradient form of Q-learning for weight updates.
  - Implemented experience replay and gradient clipping to enhance learning stability.

- **Performance Evaluation**:
  - Compared DQN scores with best performing systems, human testers, and random agents.
  - Evaluated over 30 sessions per game.
  - DQN surpassed previous systems and human players in many games.

- **Significance of DQN**:
  - Demonstrated a single agent could learn across diverse tasks.
  - Reduced the need for problem-specific design and tuning.
  - Highlighted the potential of deep learning in reinforcement learning.

- **Limitations**:
  - DQN's performance varied across games.
  - Struggled with games requiring deep planning (e.g., Montezuma's Revenge).
  - Highlighted the need for further advancements in task-independent learning.

Section: 9
Policy Gradient; RL-Human Feedback; ChatGPT

Lecture # 9 Date: July 20, 2024

Recap

- **Model-Based Methods:** Dynamic Programming
- **Model-Free Methods:**
  - Monte Carlo
  - Temporal Difference (TD)
  - Function Approximation based Monte Carlo
  - Function Approximation based TD
  - Deep Q-Network (DQN) - a refinement of Monte Carlo based DQN
- All above are value function methods. They obtain the value function v_π and q_π.
- **Dynamic Programming** computes value functions, while others learn value functions.
- Once we have the value function, we can derive the policy.
- We never handle the policy directly; they are always defined in terms of value functions. That's why they are called value function approaches.
- The primary goal is to obtain the value function v_π or q_π or their approximations. After computing or learning the value function, we can derive the policy from it.
- The policy does not have an independent existence apart from the initialization to some arbitrary policy. It is always tied to the value function.
- **Model-Based** methods involve computation.
- **Model-Free** methods involve learning.
- **Table-Based Methods:** DP, MC, TD
- **Neural Network Based Methods:** FA-MC, FA-TD, DQN

Policy Gradient Methods

- We are not discussing value function-based approaches. Instead, we are focusing on learning the policy directly.
- Policy gradient methods fall within the second part of the course, as they are function approximation-based techniques.
- These methods are also model-free and involve learning.

Method	Model-Based/Model-Free	Table-Based/Neural Network-Based
Dynamic Programming (DP)	Model-Based	Table-Based
Monte Carlo (MC)	Model-Free	Table-Based
Temporal Difference (TD)	Model-Free	Table-Based
Function Approximation based Monte Carlo (FA-MC)	Model-Free	Neural Network-Based
Function Approximation based TD (FA-TD)	Model-Free	Neural Network-Based
Deep Q-Network (DQN)	Model-Free	Neural Network-Based

Table 3: Categorization of Methods

- Unlike previous methods where we learned a value function, we now learn the policy directly.
- Neural networks enable the direct learning of policies, which is why this topic is introduced later in the course.
- This approach is exciting as it differs from prior concepts in a significant way by focusing on policy learning.
- **Why Learn the Policy Directly?**
  - We will explore the advantages of learning the policy directly.
  - We will discuss the limitations of value function-based approaches.

The DQN and Policy Learning - Review

- **Functional Approximation:** DQN enabled practical use of function approximation with TD methods.
- **Q-Learning Application:** It showcased a refined application of Q-learning.
- **Replay Memory:** DQN introduced a replay memory that allowed the network to learn from batches of past experiences, smoothing the learning process by avoiding sample-by-sample learning.

Addressing Non-Markovian Problems

- A significant aspect of DQN was its ability to tackle non-Markovian problems, specifically in the context of learning to play video games.
- In non-Markovian scenarios, a single snapshot of the environment (e.g., a single frame of a video game) does not provide enough context to determine the state of the system effectively. This lack of context makes it difficult to understand the significance of an observation

- **Partial Observations:** For example, in a video game, observing only the current screen does not convey the full dynamics of the game. The state is not fully captured, as you need additional context like positions, velocities, and actions to understand the situation.
- **Historical Context:** To make informed decisions, you require historical context. For example, in chess, knowing only the last move is insufficient without the context of the entire game state.

Transforming Non-Markovian to Markovian

- **Using Multiple Screenshots:** Instead of using a single frame, DQN used multiple screenshots as the state representation. This approach provides more context and transforms a non-Markovian problem into a Markovian one, where the state representation includes historical observations.

Understanding Non-Markovian Problems

Introduction to Non-Markovian Problems

To illustrate a non-Markovian problem, let us consider a simple example involving an agent navigating through a space. The agent's goal is to move from a starting position S to a goal position G.

Example: Navigation Problem

Scenario: - The agent starts at position S and needs to reach position G. - The actual path has two intermediate steps: S_1 and S_2. - The agent is unaware of these intermediate states. It only knows it is not at G and not at S.

Agent's Actions: - The agent can take two actions: R (Right) and L (Left). - In state S:

- Action R moves the agent to the right.
- Action L keeps the agent in state S.

- In the unknown intermediate states (denoted as X):

- Action R moves the agent to the left.
- Action L moves the agent to the right.

- In state G:

- Both actions R and L have no effect, as the goal is reached.

Challenges Faced by the Agent

- **Problem:** - The agent does not know whether it is in state S_1 or S_2 or any intermediate state. It only perceives itself in a non-specific state X.
- The agent's goal is to determine a policy to navigate from S to G, but it lacks information about the exact states in between.

- **Implications:** - If the agent knew the intermediate states (S_1 and S_2), a value function method could easily determine the value of actions in various states.
- Without knowledge of these intermediate states, the agent can only perceive that it is in some unknown state X. This makes it impossible to compute the value function accurately, as the agent does not know the exact state it is in.
- **Example Calculation:** - If the agent takes action R repeatedly, it could end up in S_1 or S_2, or it could think there are many more states between S and G, making the environment appear much larger and more complex.
- The agent may think it has taken many steps and is in a state far from G when it might be much closer or in a different state.
- The core issue with non-Markovian problems is the lack of state information. Without knowing the exact state, it is impossible to compute the value function effectively. - Reinforcement learning methods that rely on value functions struggle in such scenarios because they need accurate state information to compute values and determine the best actions.

Can We Learn Probabilities Directly Without a Value Function?

Optimizing Actions in Unknown States

- In situations where the agent cannot determine its exact state, we can use algebraic and probabilistic methods to maximize long-term rewards. The idea is to adjust the probabilities of taking different actions to achieve the best expected outcome, even when the exact state is unknown.
- **Example Scenario:** - Suppose an agent is in an unknown state X that is neither S nor G. - The agent can take actions R (Right) and L (Left) with certain probabilities.
- **Optimal Probability Distribution:** - According to algebraic and probabilistic analysis, it is possible to maximize the expected reward by adjusting the probabilities of taking actions R and L. - For instance, the textbook suggests taking action R with 52 % probability and action L with 48 % probability when in the unknown state X to maximize long-term return.

Learning Probabilities in Reinforcement Learning

- The key question here is whether we can learn the probabilities of taking actions directly without relying on a value function. This is a fundamental question in reinforcement learning, as it relates to learning policies without explicitly computing value functions.
- **Approach:** - Instead of learning the value function v_π or q_π to derive the optimal policy, we aim to learn the policy $\pi(a|s)$ directly. - The policy $\pi(a|s)$ represents the probability of taking action a in state s. In this case, we would learn how to set these probabilities in the absence of a well-defined state space.
- **Policy Gradient Methods:** - To address this, policy gradient methods can be employed. These methods focus on optimizing the policy directly by adjusting the probabilities of actions.

- Policy gradients do not require the explicit computation of value functions. Instead, they optimize the policy parameters to maximize expected returns.

Learning Policies with Neural Networks

Transition from Value Function to Policy Learning

- To learn an optimal policy directly, we need a different kind of neural network compared to what we've used for learning value functions.

Neural Networks for Value Function

- Previously, we have used neural networks to learn value functions. Value functions are defined on state-action pairs (s, a) and map these pairs to a real number representing the value. This is a typical **regression neural network**, where we learn a mapping from a vector space (state-action pairs) to real numbers.

Neural Networks for Policy Learning

- Now, we aim to learn the policy π, which specifies the probability of taking each action in a given state. Instead of mapping state-action pairs to real numbers, we need to map states to probability distributions over actions.

Requirements for the Neural Network:

- **Output**: Instead of a single real number, the neural network's output should be a vector of probabilities. For an action space of size K, the output is a K-dimensional vector where each component represents the probability of taking a particular action.
- **Constraints**: This vector must satisfy two key constraints:
  - Each probability p_i should be in the range $[0, 1]$.
  - The sum of all probabilities should be equal to 1:

$$\sum_{i=1}^{K} p_i = 1$$

Probabilistic Output vs. Regression Output

- In contrast to regression neural networks that output real numbers directly, neural networks used for policy learning output probabilities. This requires learning a **probability distribution** over actions for each state.
- **Example Scenario:** - In the previously discussed scenario, the agent is in an unknown state X and must determine the probabilities for taking each action. This is a situation where the agent has degraded or incomplete state information but needs to make decisions based on probabilistic distributions.

Examples of Non-Markovian Problems

- We can look at all the examples given throughout the course:
  - In a game of cards, you only know your hand and the limited cards shown to you. You have no idea what most of the deck contains. This is a situation where you know the initial state but not the entire state space.
  - In chess, if you cannot see the chess board and only know the last move made, you have incomplete information. This is another example of a non-Markovian problem.
  - Generating letters or words to produce sentences without knowledge of the state space of the problem. Without a meaningful state model of the language or a model of the listener, generating coherent text is challenging. This problem is approached probabilistically, as seen in technologies like ChatGPT.
  - Video games often present non-Markovian situations. DQN attempts to address this by using multiple snapshots to make the problem approximately Markovian. However, the challenge remains in determining how many snapshots are sufficient.
- Considering a scenario where you can take historical snapshots but face the issue of determining the meaningful number of snapshots:
  - One might ask how to know if the number of snapshots taken is too few or too many.
  - Today's lecture focus is on an alternative approach that does not rely on the Markovian property. We condense all unknown states into a generic state X or Y, which could have many substates.
- Policy learning directly from experience without relying on the Markovian property:
  - In card games, some experts can count cards in their head, which is akin to using historical snapshots. However, this technique may not be effective in all games.
  - The goal is to learn a policy directly from experience without needing to know the exact state.
  - If the state is unknown, value functions cannot be used. Instead, policy approaches based on probabilities can be employed.
- This approach is called **policy gradient** or more generally **policy learning**. This technique utilizes neural networks to compute gradients for policy learning. It addresses how to handle non-Markovian problems by learning a policy stochastically.

Policy Learning and Neural Networks

- To learn a policy, we need a new type of neural network.
- We cannot use a regression neural network because it does not meet the requirements for learning probabilities.
- Last lecture's neural network structure:
  - Inputs: $\mathbf{X}$ (an R^N vector)
  - Internal layers: Transformed into intermediate vectors using various neurons.

  - Output: A real number, obtained by applying a linear regressor.

- The regression neural network structure can be visualized as follows:
  - Input vector $\mathbf{X} = (X_1, X_2, \ldots, X_N)$
  - Transformed through hidden layers to an intermediate vector $\mathbf{H}_1$
  - Further transformation to vector $\mathbf{H}_2$
  - Continue transformations to the final vector $\mathbf{H}$
  - Final step: Linear regressor producing a real number.

- For our new purpose, we will modify this structure:
  - Input vector $\mathbf{X}$ (an R^N vector) is transformed into an intermediate vector $\mathbf{H}$
  - We will replace the regressor with a component to generate probabilities.

- To convert the $\mathbf{H}$ vector into probabilities:
  - Use a function to transform $\mathbf{H}$ into a probability distribution.
  - The function for component Y_i is given by:

$$Y_i = \frac{e^{Z_i}}{\sum_j e^{Z_j}}$$

    where Z is the intermediate vector.
  - Each Y_i will be between 0 and 1, and the sum of all Y_i will be 1.

- This transformation converts an R^K vector into a $[0,1]^K$ vector with components summing to 1.

- This structure is called a classification neural network in supervised machine learning.

- Classification neural networks provide the format needed to learn a policy:
  - The output is a probability distribution over actions.
  - This allows us to handle non-Markovian problems by learning probabilities directly from experience.

Training the Policy Gradient Neural Network

- Simply specifying the neural network is not enough; it must be trained to avoid providing arbitrary outputs.

- Training requires a method to adjust the network's parameters based on performance.

- Previous approaches involved training on reference data, but this is not applicable here because:
  - We do not have a reference policy to guide our training.
  - We cannot use value function approximations or Monte Carlo methods directly due to lack of accurate state information.

- Instead, we need to define a performance criterion for our policy.
- The performance criterion is defined as the value function $J = V_\pi(S_0)$:
  - S_0 is the initial state.
  - $V_\pi(S_0)$ represents the expected return starting from S_0, considering all possible trajectories and their probabilities.
- The value of the initial state S_0 is the expected return over all possible trajectories from that state.
- We aim to maximize this value by tuning our policy.
- The objective is to adjust our policy to maximize the expected return from the initial state.
- To achieve this, we use gradient ascent to update our policy parameters:
  - We will ascend the gradient of J with respect to our policy parameters.
  - Since we want to maximize performance, gradient ascent is used rather than gradient descent.

Determining the Gradient of the Value Function

- The main challenge is determining the gradient of the value function J.
- This is where the **Policy Gradient Theorem** comes into play.
- The gradient of J is proportional to:
  - The sum over all states of the fraction of time spent in a particular state.
  - The sum over all actions of the value of the action multiplied by the gradient of the policy.
- The policy, denoted π, is produced by the neural network.
- The challenge is that we do not have direct access to the value function or the fraction of time spent in each state.
- Researchers refine the gradient calculation using expected values:
  - The percentage of time spent in a state is proportional to the expected value of the sum over all actions.
  - This simplifies to the expected value of the return G.
- By manipulating the expressions, we derive that:
  - The expected value of the quantity can be written in terms of observed rewards and the gradient of the policy.
  - Observed rewards are directly available and can be accumulated to calculate returns.
- The policy gradient algorithm involves:
  - Using observed returns and the gradient of the neural network output to compute the policy gradient.

  - Adding a constant of proportionality α to adjust the gradient computation.

- This leads to the **Policy Gradient Algorithm**, which allows us to learn the policy directly without using the value function.
- This method is related to how chatbots like ChatGPT are trained:
  - The neural network learns to produce action probabilities (e.g., word choices) based on observed data.

Policy Gradient Theorem

- The **objective** in reinforcement learning is to maximize the value function of our policy.
- According to the Policy Gradient Theorem:

$$\nabla J \propto E\left[G_t \cdot \frac{\nabla \pi}{\pi}\right] \tag{131}$$

- To maximize J, we ascend the gradient:

$$\theta \leftarrow \theta + \alpha \cdot G_t \cdot \nabla J \tag{132}$$

- This approach is known as the **REINFORCE** algorithm.

Improving REINFORCE

- **Proximal Policy Optimization (PPO)** is an improvement over REINFORCE.
- PPO introduces techniques such as:
  - Clipping the value of gradients.
  - Restricting how much the policy changes at each update.
- These improvements make PPO more practical for real-world applications by ensuring stable training.

Application to Chatbots

- Modern chatbots, such as Gemini or ChatGPT, use a policy-like neural network for generating responses.
- The core mechanism involves:
  - Generating a probability distribution over possible actions (e.g., words or characters).
  - The neural network maps inputs to these probabilities, much like a policy neural network in reinforcement learning.
- **Training Phase 1:** Supervised learning using a large corpus of text data.

- **Training Phase 2:** Reinforcement Learning with Human Feedback (RLHF):
  - Refines the model by incorporating human feedback into the reward signal.
  - Ensures that the chatbot generates responses that align with human preferences and expectations.

Reinforcement Learning with Human Feedback

- In RLHF, human feedback is used to define the reward function for the policy network.
- While directly providing rewards for each action (e.g., each word or character) would be impractical, feedback is aggregated and used to guide the learning process.
- This process helps in:
  - Improving the relevance and quality of responses.
  - Aligning the chatbot's behavior with human values and expectations.

Application to Robotics

- Similar to chatbots, robots can be trained using a policy network for action generation.
- Human feedback can be provided through video evaluations of the robot's actions (e.g., picking up objects).
- A reward network learns from these evaluations and improves the policy network's performance.
- This approach avoids the impracticality of providing rewards for every possible action or configuration.

Continuous Action Spaces

- Policy gradient methods are particularly useful for continuous action spaces where:
  - Value function methods require an `argmax` operation to select actions, which is not feasible in continuous spaces.
  - Policy gradients directly parameterize the action distribution, making them suitable for continuous action spaces.

Overview

Phase One: Supervised Learning

- The initial phase of training chatbots involves **supervised machine learning**.
- Large Language Models (LLMs) are trained on extensive corpora of text data.
- The goal is to model the probability distribution of generating coherent and contextually relevant text.

Phase Two: Reinforcement Learning with Human Feedback

- In the second phase, **reinforcement learning** is employed to further refine the chatbot's performance.
- Here, a policy network, similar in structure to the output network of an LLM, is trained using reinforcement learning techniques.
- Human feedback plays a critical role in this phase:
  - A **rewards neural network** is introduced to provide reward signals based on human evaluations.
  - The rewards neural network is trained by showing human evaluators various chatbot responses and obtaining their feedback.
  - This feedback is used to adjust the rewards neural network, which in turn guides the policy network's learning.

Integration of Policy Gradient and Reward Networks

- The policy neural network, used in the second phase, has a similar structure to the output neural network of the chatbot.
- The integration involves:
  - Using the rewards neural network to provide feedback on the chatbot's responses.
  - Applying policy gradient techniques to adjust the policy network based on the rewards provided by the rewards neural network.
- This combination leverages both supervised learning (for initial training) and reinforcement learning (for fine-tuning with human feedback) to enhance the chatbot's performance.

Section: 9-1
Notes from Textbook Readings

Chapter 13: Policy Gradient Methods

- **Policy Representation:**
  - Parameterized policy: $\pi(a|s,\theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}$
  - Policy parameter vector: $\theta \in R^{d'}$
- **Policy Gradient Objective:**
  - Performance measure $J(\theta)$
  - Update rule using gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{133}$$

  - Gradient estimate:

$$\nabla_\theta J(\theta_t) \approx E\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(A_t|S_t,\theta) G_t\right] \tag{134}$$

- **Actor-Critic Methods:**
  - Actor: The learned policy $\pi(a|s,\theta)$
  - Critic: The learned value function $v(s,w)$ with parameters $w \in R^d$
- **Episodic Tasks:**
  - Performance defined as the value of the start state under the parameterized policy
  - Gradient of the performance measure for episodic tasks:

$$\nabla_\theta J(\theta) = E\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(A_t|S_t,\theta) G_t\right] \tag{135}$$

  - G_t is the return following time step t
- **Continuing Tasks:**
  - Performance defined as the average reward rate
  - Gradient of the performance measure for continuing tasks:

$$\nabla_\theta J(\theta) = E\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi(A_t|S_t,\theta) R_t\right] \tag{136}$$

  - R_t is the reward at time step t

Policy Approximation

- **Policy Parameterization:**
  - The policy can be parameterized in any differentiable form, denoted as $\pi(a|s,\theta)$.
  - We require that $\pi(a|s,\theta) \in (0,1)$ for all s, a, θ to ensure exploration.
  - For discrete action spaces, parameterized numerical preferences $h(s,a,\theta) \in R$ are common.
- **Soft-max in Action Preferences:**
  - Actions with the highest preferences are given the highest probabilities using an exponential soft-max distribution:

$$\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}} \tag{137}$$

  - $e \approx 2.71828$ is the base of the natural logarithm.
- **Parameterizing Action Preferences:**
  - Preferences $h(s,a,\theta)$ can be computed by:
    - A deep artificial neural network (ANN) where θ represents the connection weights.
    - A linear function of features:

$$h(s,a,\theta) = \theta^T x(s,a) \tag{138}$$

    - Feature vectors $x(s,a) \in R^{d'}$.
- **Advantages of Soft-max Parameterization:**
  - Can approach a deterministic policy, unlike ϵ-greedy action selection.
  - Enables selection of actions with arbitrary probabilities, useful in problems requiring stochastic policies.
  - Example: Card games with imperfect information.
- **Example 13.1 Short Corridor with Switched Actions:**
  - Gridworld with a reward of -1 per step.
  - Actions are reversed in the second state.
  - ϵ-greedy action selection forces a choice between two policies, achieving less optimal values.
  - A specific probability selection method can achieve better performance.
- **Policy vs. Action-Value Function Approximation:**
  - The policy may be simpler to approximate in some problems.
  - Policy-based methods can learn faster and yield superior asymptotic policies when the policy is simpler.
  - Example: Tetris.

- **Prior Knowledge:**
  - Choice of policy parameterization can inject prior knowledge about the desired form of the policy.
  - Often an important reason for using a policy-based learning method.

The Policy Gradient Theorem

- **Advantages of Policy Parameterization:**
  - Action probabilities change smoothly as a function of the learned parameter.
  - Stronger convergence guarantees compared to ϵ-greedy action selection.
  - Allows approximation of gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{139}$$

- **Performance Measure $J(\theta)$:**
  - Defined differently for episodic and continuing cases.
  - Episodic case performance measure:

$$J(\theta) \doteq v_{\pi_\theta}(s_0) \tag{140}$$

  - $v_{\pi_\theta}(s_0)$: True value function for policy π_θ.
  - Assumes no discounting ($\gamma = 1$).

- **Policy Gradient Theorem:**
  - Addresses how to estimate the performance gradient when state distribution is unknown.
  - Provides an analytic expression for the gradient of performance with respect to the policy parameter.
  - For the episodic case:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{141}$$

  - Gradients are column vectors of partial derivatives with respect to components of θ.
  - Proportionality constant:
    - Episodic case: Average length of an episode.
    - Continuing case: 1 (relationship becomes an equality).
  - $\mu(s)$: On-policy distribution under π.

REINFORCE: Monte Carlo Policy Gradient

- **Overall Strategy:**
  - Use stochastic gradient ascent:
$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{142}$$
  - Requires sample gradients proportional to the actual gradient.
  - The constant of proportionality can be absorbed into the step size α.
- **Policy Gradient Theorem:**
  - Provides an exact expression proportional to the gradient:
$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) = E_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] \tag{143}$$
  - Use this to define the update rule:
$$\theta_{t+1} \approx \theta_t + \alpha \sum_a \hat{q}(S_t,a,w) \nabla \pi(a|S_t,\theta) \tag{144}$$
- **REINFORCE Algorithm:**
  - Involves only the action A_t taken at time t.
$$\nabla J(\theta) \propto E_\pi \left[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] \tag{145}$$
  - Update rule:
$$\theta_{t+1} \approx \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{146}$$
  - Intuitive explanation:
    - Increment is proportional to the product of return G_t and a vector.
    - The vector increases the probability of repeating action A_t on future visits to state S_t.
    - The increment is inversely proportional to the action probability to avoid frequent actions dominating.
- **Monte Carlo Nature:**
  - Uses the complete return G_t, including all future rewards until the end of the episode.
  - Well-defined only for the episodic case, with all updates made retrospectively after the episode is completed.
- **Pseudocode Differences:**
  - Uses the compact expression $\nabla \ln \pi(A_t|S_t,\theta_t)$:
$$\nabla \ln \pi(A_t|S_t,\theta_t) = \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{147}$$

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\quad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Figure 29: REINFORCE: Monte-Carlo Policy-Gradient Control

  - Includes a factor of γ^t in the general discounted case.

- **Convergence Properties:**
  - Expected update over an episode is in the same direction as the performance gradient.
  - Assures improvement in expected performance for sufficiently small α.
  - Converges to a local optimum under standard stochastic approximation conditions for decreasing α.
  - As a Monte Carlo method, REINFORCE may have high variance and thus produce slow learning.

Section: 10
Integrated model free and model based RL; Planning; Mu-zero

Lecture # 10 - Date: July 27, 2024

Recap:

- We discussed the role and utility of models in reinforcement learning.
- Covered function approximation techniques.

Understanding Models in the Environment:

- A **model of the environment** allows an agent to predict how the environment will respond to its actions.
- Given a state and an action, the model produces a prediction of:
  - The resultant next state.
  - The next reward.
- **Types of Models:**
  - **Stochastic Models**: Produce several possible next states and rewards, each with a probability.
  - **Distribution Models**: Describe all possibilities and their probabilities.
  - **Sample Models**: Produce just one possibility, sampled according to the probabilities.
  - Distribution models are stronger because they can always produce samples, but sample models are often easier to obtain.

Planning and Simulated Experience:

- **Simulation**: The model is used to simulate the environment and produce simulated experience.
- **Planning**: Refers to any computational process that uses a model to produce or improve a policy for interacting with the modeled environment.
- **Key Ideas in Planning:**
  - All state-space planning methods compute value functions as a key step toward improving the policy.
  - Value functions are computed by updates or backup operations applied to simulated experience.

- The **core of both learning and planning** is the estimation of value functions by backing-up update operations.
- Difference between learning and planning:
  - **Planning**: Uses simulated experience generated by a model.
  - **Learning**: Uses real experience generated by the environment.

Advantages of Incremental Planning:

- Planning in small, incremental steps allows for:
  - Interruption or redirection with minimal wasted computation.
  - Efficient intermixing of planning with acting and learning.
- Even for pure planning problems, small steps may be more efficient if the problem is too large to solve exactly.

Role of Real Experience in a Planning Agent:

- Real experience can be used to:
  - **Improve the model** to better match the real environment (**model-learning**).
  - **Directly improve the value function and policy** using reinforcement learning methods (**direct RL**).

Direct vs. Indirect Methods:

- **Indirect Methods:**
  - Make fuller use of limited experience.
  - Achieve a better policy with fewer environmental interactions.
- **Direct Methods:**
  - Simpler to implement.
  - Not affected by biases in the model design.

Monte Carlo Tree Search (MCTS):

- MCTS is a **rollout algorithm** with modifications:
  - Selects a small set of actions according to its tree policy.
  - Sometimes expands the selection to additional actions.
  - Performs a full rollout to the end of the episode using a rollout policy.
  - The value found at the end is backed up to the current state.
- This process repeats within time/computational constraints, and the agent selects an action based on the best future estimated reward.

Section: 10-1
Notes from Textbook Readings

Chapter 8 Planning and Learning with Tabular Methods

- **Overview:**
  - Development of a unified view of reinforcement learning methods.
  - Model-based methods (e.g., dynamic programming, heuristic search).
  - Model-free methods (e.g., Monte Carlo, temporal-difference methods).
- **Model-Based vs. Model-Free Methods:**
  - Model-based methods:
    - Require a model of the environment.
    - Rely on planning as their primary component.
  - Model-free methods:
    - Do not require a model of the environment.
    - Primarily rely on learning.
- **Commonalities:**
  - Both methods involve computation of value functions.
  - Both look ahead to future events, compute a backed-up value, and use it as an update target.
- **Integration of Methods:**
  - Earlier chapters presented Monte Carlo and temporal-difference methods as distinct alternatives.
  - n-step methods were shown to unify Monte Carlo and temporal-difference methods.
  - Current goal is to integrate model-based and model-free methods.
  - Explore the extent to which they can be intermixed.

Models and Planning

- **Definition of a Model:**
  - A model predicts the environment's response to an agent's actions.
  - Given a state and an action, a model predicts the next state and reward.
  - Stochastic models produce multiple possible outcomes with associated probabilities.
  - Types of models:
    - **Distribution models:** Describe all possibilities and their probabilities.
    - **Sample models:** Produce one possibility sampled according to the probabilities.
- **Example:**

  - Modeling the sum of a dozen dice:
    * **Distribution model:** Produces all possible sums and their probabilities.
    * **Sample model:** Produces a single sum based on the probability distribution.

- **Use of Models:**
  - Models simulate experience by generating possible transitions.
  - **Sample models:** Produce a possible transition given a state and action.
  - **Distribution models:** Generate all possible transitions weighted by their probabilities.
  - Models can simulate entire episodes or generate all possible episodes and their probabilities.

- **Planning:**
  - Planning is any computational process that uses a model to produce or improve a policy.
  - **State-space planning:** Searches through the state space for an optimal policy or path.
  - **Plan-space planning:** Searches through the space of plans.
    * Difficult to apply efficiently to stochastic sequential decision problems.

- **Unified View of Planning Methods:**
  - All state-space planning methods share a common structure:
    1. Compute value functions as an intermediate step toward policy improvement.
    2. Compute value functions by updates or backup operations applied to simulated experience.

- **Common Structure Diagram:**

$$\text{Values} \xrightarrow{\text{backups}} \text{Model simulated experience} \xrightarrow{\text{backups}} \text{Policy updates}$$

- **Dynamic Programming:**
  - Sweeps through the state space, generating distributions of possible transitions.
  - Each distribution is used to compute a backed-up value and update the state's estimated value.

- **Relationship to Learning Methods:**
  - Both planning and learning methods estimate value functions by backing-up update operations.
  - Planning uses simulated experience from a model.
  - Learning uses real experience from the environment.

- **Transfer of Ideas and Algorithms:**
  - Learning algorithms can often be substituted for the key update step of a planning method.
  - Learning methods can be applied to both real and simulated experience.

Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 30: Random-sample one-step tabular Q-planning

- **Example: Random-Sample One-Step Tabular Q-Planning:**
  - Uses one-step tabular Q-learning and random samples from a sample model.
  - Converges to the optimal policy for the model under the same conditions as one-step tabular Q-learning.
- **Planning in Small, Incremental Steps:**
  - Enables planning to be interrupted or redirected at any time with minimal wasted computation.
  - Efficiently intermixes planning with acting and learning of the model.
  - May be the most efficient approach even for pure planning problems if the problem is too large to be solved exactly.

Dyna: Integrated Planning, Acting, and Learning

- **Online Planning and Interaction:**
  - New information from interaction may change the model and interact with planning.
  - Planning may need customization based on current or near-future states or decisions.
  - Computational resources may need to be divided between decision-making and model learning.
- **Roles of Real Experience:**
  - **Model-learning:** Improving the model to better match the real environment.
  - **Direct reinforcement learning (direct RL):** Directly improving the value function and policy using reinforcement learning methods.
- **Indirect Reinforcement Learning:**
  - Experience can improve value functions and policies either directly or indirectly via the model.
  - Indirect methods can make fuller use of limited experience to achieve better policies with fewer interactions.
  - Direct methods are simpler and not affected by biases in model design.

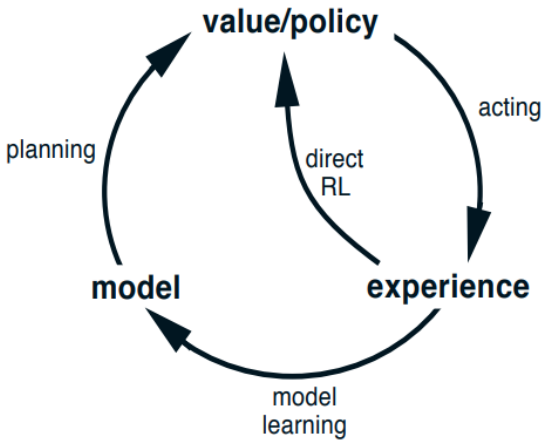

Figure 31: Direct and Indirect RL

- **Dyna-Q Architecture:**
  - Integrates planning, acting, model-learning, and direct RL.
  - **Planning method:** Random-sample one-step tabular Q-planning.
  - **Direct RL method:** One-step tabular Q-learning.
  - **Model-learning method:** Table-based, assuming a deterministic environment.
- **Model Learning:**
  - After each transition $S_t, A_t \to R_{t+1}, S_{t+1}$, the model records R_{t+1}, S_{t+1} for S_t, A_t.
  - The model returns the last-observed next state and next reward for experienced state-action pairs.
  - During planning, the Q-planning algorithm samples from previously experienced state-action pairs.
- **Dyna-Q Algorithm:**
  - Learning and planning use the same reinforcement learning method.
  - Direct RL, model-learning, and planning occur in parallel and share the same machinery.
  - **Serial Implementation:**
    - Assume acting, model-learning, and direct RL require little computation.
    - Remaining time in each step is devoted to planning.
    - Complete n iterations of the Q-planning algorithm per time step.
- **Pseudocode for Dyna-Q:**
  - **Notation:**
    - Model(s, a) denotes the predicted next state and reward for state-action pair (s, a).

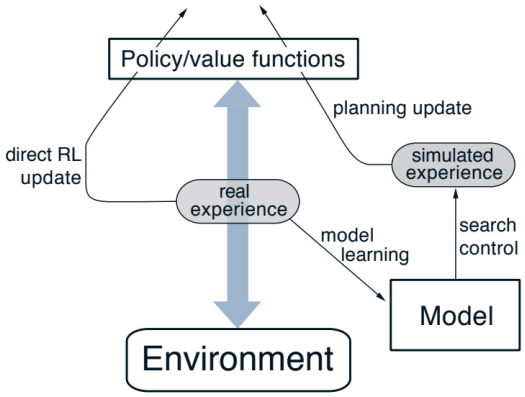

Figure 32: Overall architecture of Dyna agents, illustrating the integration of planning, acting, model-learning, and direct reinforcement learning

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy(S,Q)
(c) Take action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Loop repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 33: Tabular Dyna-Q

  - Steps (d), (e), and (f) correspond to direct RL, model-learning, and planning, respectively.
  - Omitting steps (e) and (f) results in one-step tabular Q-learning.

- **Incremental Planning:**
  - Learning and planning are accomplished by the same algorithm, operating on real and simulated experience.
  - Planning and acting can be intermixed, with the agent being reactive and deliberative simultaneously.
  - The model is updated as new information is gained, influencing the planning process.

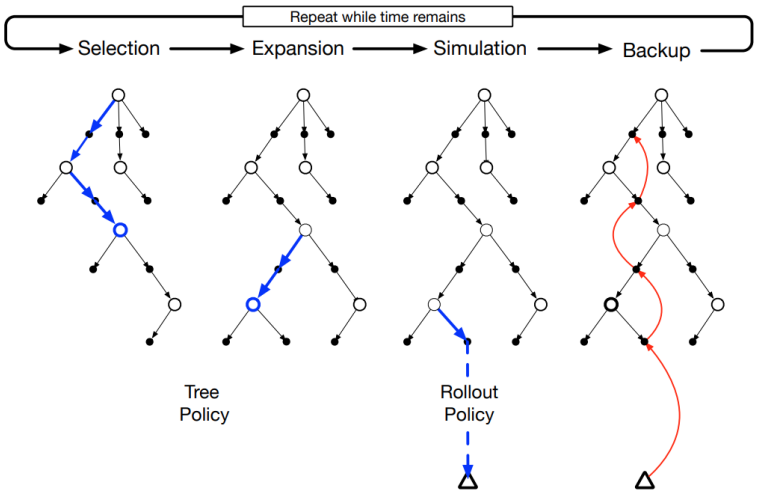

Figure 34: Monte Carlo Tree Search

Monte Carlo Tree Search

- MCTS is a decision-time planning algorithm.
- Originated for games, particularly successful in computer Go and general game playing.
- Can be applied to single-agent sequential decision problems.

Core Idea

- Simulates multiple trajectories from the current state to terminal states.
- Focuses simulations on high-reward trajectories identified in earlier simulations.

Execution

- MCTS is executed at each new state to select the next action.

Process

- **Selection**:
  - Traverse the tree from the root using a tree policy to select a leaf node.
- **Expansion**:
  - Add one or more child nodes to the selected leaf node using unexplored actions.
- **Simulation**:
  - From the selected node or its new child nodes, simulate a complete episode using a rollout policy.

- **Backup**:
  - Update action values from the simulated episode's return, focusing on trajectories with high returns.

Key Notes

- MCTS does not retain approximate value functions or policies from one action selection to the next but can retain useful action values.
- The rollout policy is used for actions outside the tree, while an informed tree policy is used within the tree.
- Action selection at the root is based on accumulated statistics, e.g., selecting the action with the highest value or visit count.
- Often used in two-person competitive games and can be extended with deep neural networks for enhanced performance, as seen in AlphaGo.

Section: 10-2
Notes from Textbook Readings

Mastering the Game of Go

AlphaGo

- **Overview**
  - Developed by DeepMind to play the game of Go.
  - Combines deep artificial neural networks (ANNs), supervised learning, Monte Carlo Tree Search (MCTS), and reinforcement learning.
  - Achieved significant milestones:
    - Defeated European Go champion Fan Hui in October 2015.
    - Defeated 18-time world champion Lee Sedol in March 2016.
- **Training and Techniques**
  - Supervised learning from a database of human expert moves.
  - Reinforcement learning through self-play.
  - Policy network trained on 30 million positions from human games.
  - Value network predicts the winner from a given board position.
  - Rollout policy network used for fast simulations.
  - APV-MCTS (Asynchronous Policy and Value MCTS) integrates policy and value networks.
  - Balance between value network evaluations and rollout simulations.
- **MCTS in AlphaGo**
  - Decision-time planning procedure.
  - Runs many Monte Carlo simulations of entire games to select actions.
  - Incrementally extends a search tree from the current state.
  - Nodes are evaluated using rollouts and value functions.
  - Policy network guides tree expansion.
  - Value network provides evaluations of non-terminal nodes.
  - Final action selected based on the most-visited edge in the search tree.

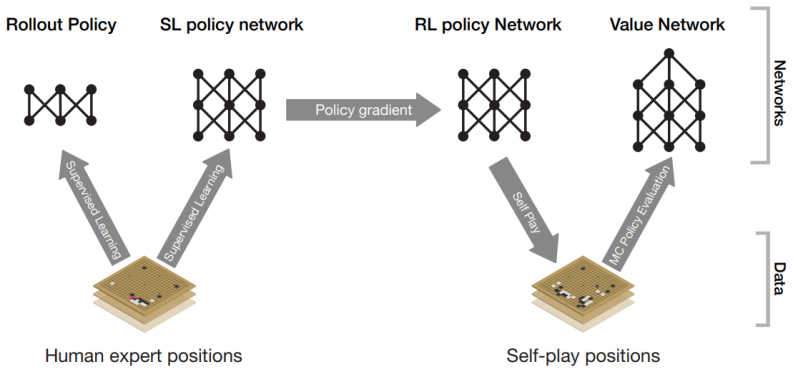

Figure 35: AlphaGo pipeline

AlphaGo Zero

- **Overview**
  - Improved version of AlphaGo, developed by DeepMind.
  - Learns solely from self-play without any human data.
  - Implements a form of policy iteration.
- **Network Architecture**
  - Input: $19 \times 19 \times 17$ image stack of board positions.
  - Two-headed output: move probabilities and win probability.
  - Consists of 41 convolutional layers with batch normalization and skip connections.
- **Training Process**
  - Uses MCTS throughout self-play reinforcement learning.
  - Stochastic gradient descent for network optimization.
  - Sampling from recent self-play games to update the network.
  - Periodic evaluation against the current best policy.
- **Differences from AlphaGo**
  - No rollouts of complete games.
  - Pure reinforcement learning program.
  - Outperforms previous versions and all prior human champion players.
  - Relies on MCTS for both learning and decision-making.
- **Achievements**
  - Demonstrated the power of self-play reinforcement learning.
  - Proved that human data is not necessary to achieve superhuman performance in Go.
  - Became the strongest Go player in the world.

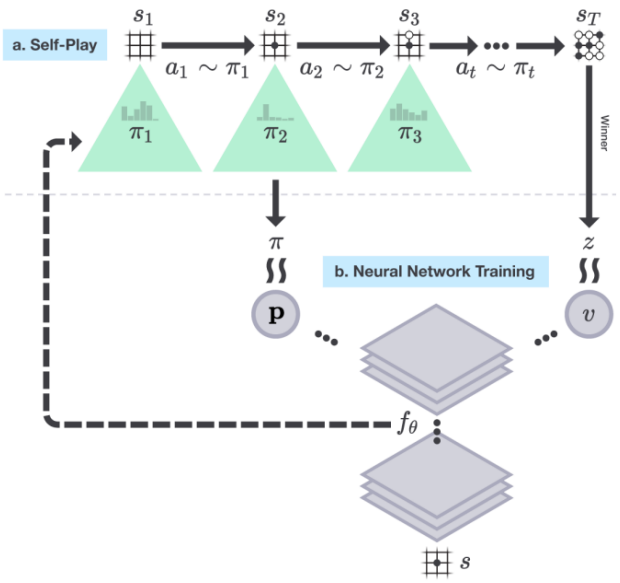

Figure 36: AlphaGo Zero self-play reinforcement learning

AlphaGo Zero Self-Play Reinforcement Learning

- **Self-Play Games**
  - The program played many games against itself.
  - Games represented as sequences of board positions s_i, $i = 1, 2, \ldots, T$ with moves a_i, $i = 1, 2, \ldots, T$.
  - Each game ends with a winner z.
- **Move Selection**
  - Moves a_i determined by action probabilities π_i.
  - Probabilities returned by MCTS executed from root node s_i.
  - MCTS guided by a deep convolutional network f_θ with weights θ.
- **Network Inputs and Outputs**
  - Inputs: raw representations of board positions s_i and several past positions.
  - Outputs: vectors p of move probabilities guiding MCTS forward searches.
  - Outputs: scalar values v estimating the probability of the current player winning from each position s_i.
- **Network Training**
  - Training examples randomly sampled steps from recent self-play games.
  - Weights θ updated to align policy vector p with probabilities π returned by MCTS.
  - Training included the winners z in the estimated win probability v.

Challenges in Go AI

- **Complexity of Go**
  - Search space larger than that of chess.
  - Defining an adequate position evaluation function is difficult.
  - Exhaustive search is infeasible.
- **Advancements with AlphaGo and AlphaGo Zero**
  - MCTS and deep learning as significant advancements.
  - Achieved master-level skill and beyond.