College of Engineering & Applied Sciences

CSPB 3202

Introduction To Artifical Intelligence

Class Notes

UNIVERSITY OF COLORADO

2024

Introduction To Artifical Intelligence - Class Notes

Intro To AI, Search Problems

1.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 1 - Introduction**
- **Artificial Intelligence - A Modern Approach - Chapter 2 - Intelligent Agents**
- **Artificial Intelligence - A Modern Approach - Chapter 3.1 - Problem-Solving Agents**
- **Artificial Intelligence - A Modern Approach - Chapter 3.2 - Example Problems**
- **Artificial Intelligence - A Modern Approach - Chapter 3.3 - Search Algorithms**
- **Artificial Intelligence - A Modern Approach - Chapter 3.4 - Uninformed Search Strategies**

1.0.2 Piazza

Must post at least **three** times this week to Piazza.

1.0.3 Lectures

The lectures for this week are:

- Introduction To AI $\approx$ 23 **min**.
- Intelligent Agent $\approx$ 17 **min**.
- Search Intro $\approx$ 20 **min**.
- Uninformed Search $\approx$ 30 **min**.
- HW1A Helper $\approx$ 17 **min**.

The lecture notes for this week are:

- Intelligent Agent Lecture Notes
- Introduction To AI Lecture Notes
- Search Intro Lecture Notes
- Uninformed Search Lecture Notes

1.0.4 Assignments

The assignment(s) for this week are:

- Assignment 1 - Intro To AI, Search Problems

1.0.5 Quiz

The quiz for this week is:

- Quiz 1 - Intro To AI, Search Problems

1.0.6 Chapter Summary

The first chapter that is covered this week is **Chapter 1: Introduction**.

Chapter 1: Introduction

Overview

This chapter introduces the field of artificial intelligence (AI), discussing its importance, historical context, and various approaches to defining and understanding AI. It outlines the major subfields within AI and emphasizes the ongoing intellectual challenges and opportunities in the field.

What Is AI?

AI can be defined in various ways, including:

- **Human-like intelligence:** Emulating human performance and thought processes.
- **Rationality:** Acting logically to achieve goals.
- **Internal vs. external focus:** Some definitions emphasize internal thought processes, while others focus on observable behavior.

Acting Humanly: The Turing Test Approach

The Turing Test, proposed by Alan Turing, evaluates a machine's ability to exhibit human-like intelligence. To pass, a machine must demonstrate:

- *Natural language processing* to communicate.
- *Knowledge representation* to store information.
- *Automated reasoning* to draw conclusions.
- *Machine learning* to adapt and recognize patterns.

Thinking Humanly: The Cognitive Modeling Approach

To model human thinking, we study cognitive processes through:

- *Introspection:* Observing one's own thoughts.
- *Psychological experiments:* Observing behavior.
- *Brain imaging:* Studying brain activity.

AI programs can be evaluated by comparing their performance and processes to human behavior.

Thinking Rationally: The "Laws of Thought" Approach

This approach is based on formalizing logical reasoning, as initiated by Aristotle's syllogisms. It includes the development of logicist traditions and the incorporation of probabilistic reasoning to handle uncertainty.

Acting Rationally: The Rational Agent Approach

AI systems are viewed as rational agents that act to achieve the best outcome or expected outcome in uncertain situations. This approach emphasizes the construction of agents that do the "right thing," guided by mathematically defined rationality.

Beneficial Machines

The chapter discusses the value alignment problem, which addresses ensuring AI systems pursue human objectives. It emphasizes the need for machines to act cautiously and seek human permission when unsure of objectives.

The Foundations of Artificial Intelligence

The chapter provides a brief history of the disciplines that contributed to AI:

- **Philosophy:** Formal rules, mind-body distinction, and ethics.
- **Mathematics:** Formal logic, probability, and computability.
- **Economics:** Decision theory and game theory.
- **Neuroscience:** Brain function and neural networks.
- **Psychology:** Cognitive science and human-computer interaction.
- **Computer Engineering:** Development of computers and computational theory.
- **Control Theory and Cybernetics:** Self-regulating systems and feedback control.
- **Linguistics:** Language processing and understanding.

The History of Artificial Intelligence

The chapter outlines the milestones in AI history, such as:

- **1943–1956:** Inception of AI with neural networks and Turing's work.
- **1952–1969:** Early enthusiasm with programs like the Logic Theorist and the General Problem Solver.
- **1966–1973:** Realization of the challenges, such as combinatorial explosion and limited problem-solving capabilities.

The next chapter that is being covered this week is **Chapter 2: Intelligent Agents**.

Chapter 2: Intelligent Agents

Overview

This chapter explores the concept of intelligent agents, discussing their nature, the environments they operate in, and various types of agents. It builds on the notion of rational agents, which are central to understanding AI.

Agents and Environments

An agent is an entity that perceives its environment through sensors and acts upon it using actuators. The agent function maps percept sequences to actions. A key example is the vacuum-cleaner agent in a simple environment with two squares, A and B. The agent's actions depend on the percept sequence it has encountered.

Good Behavior: The Concept of Rationality

Rational agents act to maximize their performance measure based on the percept sequence and their built-in knowledge. Rationality is defined by:

- Performance measure
- Prior knowledge
- Actions available
- Percept sequence

Rationality is not omniscience; it involves maximizing expected performance given the available information.

The Nature of Environments

Environments can vary along several dimensions:

- **Fully vs. Partially Observable:** Whether the agent has access to the complete state of the environment.
- **Single-Agent vs. Multiagent:** Whether the environment involves one or multiple agents.
- **Deterministic vs. Nondeterministic:** Whether the next state of the environment is completely determined by the current state and the agent's action.
- **Episodic vs. Sequential:** Whether the agent's experience is divided into atomic episodes.
- **Static vs. Dynamic:** Whether the environment can change while the agent is deliberating.
- **Discrete vs. Continuous:** Whether the environment has a finite number of distinct states.
- **Known vs. Unknown:** Whether the agent knows the rules governing the environment.

The Structure of Agents

Agent programs implement the agent function, mapping percepts to actions. Basic types of agent programs include:

- **Simple Reflex Agents:** Act based on the current percept.
- **Model-Based Reflex Agents:** Maintain internal state to keep track of the world.
- **Goal-Based Agents:** Act to achieve specific goals.
- **Utility-Based Agents:** Maximize a utility function to achieve the best outcome.

Learning Agents

Learning agents can improve their performance by modifying their behavior based on feedback. They consist of:

- **Learning Element:** Responsible for making improvements.
- **Performance Element:** Responsible for selecting actions.
- **Critic:** Provides feedback on the agent's performance.
- **Problem Generator:** Suggests actions to gain new experiences.

Summary

The chapter defines agents, rationality, and environments, laying the foundation for designing intelligent systems. The key points include:

- Agents perceive and act in environments.
- Rational agents maximize expected performance.
- Task environments vary and influence agent design.
- Different types of agent programs offer varying levels of complexity and flexibility.
- Learning agents can adapt and improve over time.

The next chapter that is **Chapter 3: Solving Problems By Searching**. The first section that is covered in this chapter this week is **Section 3.1: Problem-Solving Agents**.

Section 3.1: Problem-Solving Agents

Overview

This chapter introduces problem-solving agents and their use of search algorithms to plan sequences of actions to achieve goals. The focus is on agents operating in simple, fully observable, deterministic, and discrete environments.

Problem-Solving Agents

Problem-solving agents use search to find sequences of actions that lead to goal states. These agents use atomic representations where states are considered as wholes without internal structure. The process involves four phases:

- **Goal formulation:** Defining the goal to organize behavior.
- **Problem formulation:** Describing the states, actions, and transitions needed to reach the goal.
- **Search:** Simulating sequences of actions to find a solution.
- **Execution:** Performing the actions to achieve the goal.

Search Problems and Solutions

A search problem includes:

- **State space:** The set of all possible states.
- **Initial state:** The starting point for the agent.
- **Goal state(s):** The desired state(s) to achieve.
- **Actions:** The set of actions available to the agent.
- **Transition model:** Describes the outcome of actions.
- **Action cost function:** Assigns a cost to each action.

A solution is a sequence of actions leading from the initial state to a goal state. An optimal solution has the lowest path cost among all solutions.

Formulating Problems

Problem formulation involves creating an abstract model by removing irrelevant details. The abstraction must be valid and useful, allowing detailed actions to be carried out without further planning.

Example Problems

Problems can be classified as standardized or real-world. Standardized problems are used for benchmarking algorithms, while real-world problems are practical applications used by people. Examples include:

- **Grid world:** A two-dimensional grid where agents move between cells, potentially containing obstacles or objects.

The next section that is covered from this chapter is **Section 3.2: Example Problems**.

Section 3.2: Example Problems

Overview

This chapter lists various problems that can be addressed using the problem-solving approach, distinguishing between standardized problems for benchmarking and real-world problems with practical applications. The examples provided serve to illustrate the diversity of problems that can be tackled using search algorithms and problem-solving techniques.

Standardized Problems

Standardized problems are designed to illustrate or exercise problem-solving methods and are suitable for comparing the performance of algorithms. These problems have concise and exact descriptions, making them useful benchmarks for researchers.

Grid World

The grid world problem involves a two-dimensional rectangular array of square cells in which agents can move from cell to cell. The simplicity and versatility of this problem make it a popular choice for demonstrating basic search and navigation algorithms.

- **States:** Each cell can contain objects like agents or dirt. In a simple two-cell vacuum world, the agent can be in either of the two cells, and each cell can either contain dirt or not.
- **Initial state:** Any state can be designated as the starting point.
- **Actions:** Movements (e.g., Left, Right, Up, Down) and actions (e.g., Suck). In multi-cell worlds, actions might include moving Upward, Downward, Forward, Backward, or turning.
- **Transition model:** Actions alter the state based on predefined rules, such as removing dirt or moving the agent.
- **Goal states:** States where every cell is clean.
- **Action cost:** Each action costs 1.

Sokoban Puzzle

The Sokoban puzzle requires an agent to push boxes scattered around a grid to designated storage locations. This problem involves both spatial reasoning and strategic planning, making it a more complex variant of the grid world problem.

- **States:** Describes the locations of boxes and the agent. Each cell can contain one box, and the agent moves boxes to storage locations.
- **Initial state:** Any configuration of boxes and the agent.
- **Actions:** Movements that push boxes. If an agent moves into a cell with a box, the box moves to the adjacent cell if it is empty.
- **Transition model:** Moving into a cell pushes the box if the next cell is empty.
- **Goal states:** All boxes in designated storage locations.
- **Action cost:** Each action costs 1.

Sliding-Tile Puzzle

The sliding-tile puzzle involves arranging tiles on a grid to achieve a specific goal state. Popular variants include the 8-puzzle and 15-puzzle. These puzzles are widely used to study search algorithms due to their clear structure and challenging nature.

- **States:** Specifies the location of each tile. For example, the 8-puzzle consists of a 3x3 grid with eight numbered tiles and one blank space.
- **Initial state:** Any arrangement of tiles can be the starting state.
- **Actions:** Move the blank space Left, Right, Up, or Down.
- **Transition model:** Moving the blank space swaps it with the adjacent tile.
- **Goal states:** Tiles arranged in a specific order.
- **Action cost:** Each action costs 1.

Knuth's Conjecture Problem

This mathematical problem, devised by Donald Knuth, explores sequences of operations to reach desired positive integers. It illustrates how infinite state spaces can arise in search problems.

- **States:** Positive real numbers.
- **Initial state:** The number 4.
- **Actions:** Apply square root, floor, or factorial operations.
- **Transition model:** Defined by mathematical operations.
- **Goal states:** The desired positive integer.
- **Action cost:** Each action costs 1.

Real-World Problems

Real-world problems are practical applications used by people and often involve complex and idiosyncratic formulations. These problems illustrate the application of search algorithms in more complicated and dynamic environments.

Route-Finding

Route-finding problems involve finding paths from one location to another. These problems are common in various applications, from GPS navigation to complex logistical planning.

- **States:** Location and time. Each state includes a location (e.g., an airport) and the current time.
- **Initial state:** The user's home airport or starting location.
- **Actions:** Take any flight from the current location or other available transport.
- **Transition model:** The state resulting from taking a flight or transport will have the new location and arrival time.
- **Goal states:** Destination city or specific location.
- **Action cost:** Combination of monetary cost, waiting time, flight time, and other factors.

Touring Problems

Touring problems require visiting multiple locations, rather than just reaching a single destination. These problems are exemplified by the traveling salesperson problem (TSP), which seeks to minimize travel costs while visiting all cities.

- **Traveling Salesperson Problem (TSP):** The goal is to visit every city with the minimal possible cost.

VLSI Layout

VLSI layout problems involve positioning millions of components and connections on a chip to optimize various criteria like area and circuit delays. These problems are crucial in the design and manufacturing of electronic devices.

- **Cell Layout:** Grouping circuit components into cells.
- **Channel Routing:** Routing wires through gaps between cells.

Robot Navigation

Robot navigation problems generalize route-finding by allowing a robot to create its own paths. These problems are more complex due to the continuous nature of the environment and the need to account for sensor errors and dynamic changes.

- **States:** Positions and orientations of the robot.

- **Actions:** Movements and adjustments in the robot's path.

Automatic Assembly Sequencing

Automatic assembly sequencing involves finding feasible and optimized sequences for assembling parts. These problems are critical in industrial manufacturing, aiming to reduce manual labor and enhance efficiency.

- **States:** Partially assembled objects.
- **Actions:** Assembly steps and movements.
- **Goal states:** Fully assembled objects.

Protein Design

Protein design problems involve finding sequences of amino acids that fold into a desired three-dimensional structure with specific properties. These problems are significant in biomedical research and drug development.

- **States:** Sequences of amino acids.
- **Actions:** Addition or modification of amino acids.
- **Goal states:** Desired protein structure.

The next section that is covered in this chapter this week is **Section 3.3: Search Algorithms**

Section 3.3: Search Algorithms

Overview

This section covers search algorithms, which take a search problem as input and return a solution or an indication of failure. Search algorithms construct a search tree over the state-space graph, forming various paths from the initial state and attempting to find a path to a goal state. Each node in the search tree corresponds to a state in the state space, and the edges correspond to actions.

Search Algorithms and Trees

It is essential to distinguish between the state space and the search tree. The state space represents all possible states and the transitions between them. The search tree represents paths between these states, reaching towards the goal. Multiple nodes in the search tree may correspond to the same state, but each node has a unique path back to the root.

Best-First Search

Best-first search is a general approach where the node with the minimum value of some evaluation function $f(n)$ is chosen next. The algorithm returns either a solution or an indication of failure.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node      NODE(STATE = problem.INITIAL)
    frontier      a priority queue ordered by f, with node as an element
    reached      a lookup table, with one entry with key problem.INITIAL and value node

    while not IS-EMPTY(frontier) do
        node      POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node

        for each child in EXPAND(problem, node) do
            s      child.STATE
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s]      child
                add child to frontier

    return failure

function EXPAND(problem, node) yields nodes
```

```
        s      node.STATE
        for each action in problem.ACTIONS(s) do
            s'      problem.RESULT(s, action)
            cost      node.PATH-COST + problem.ACTION-COST(s, action, s')
            yield NODE(STATE = s', PARENT = node, ACTION = action, PATH-COST = cost)

```

Search Data Structures

Search algorithms require specific data structures to keep track of the search tree. A node in the tree is represented by a data structure with four components: the state, the parent node, the action applied, and the path cost.

The frontier, a queue of nodes, supports operations such as checking if it is empty, popping the top node, and adding new nodes. Different types of queues used in search algorithms include priority queues, FIFO queues, and LIFO queues.

Redundant Paths

The search tree may include redundant paths, such as cycles or loopy paths. Eliminating redundant paths can significantly speed up the search process. There are three approaches to handling redundant paths:

- Remember all previously reached states.
- Ignore redundant paths if they are rare or impossible.
- Compromise by checking for cycles without tracking all redundant paths.

Measuring Problem-Solving Performance

The performance of search algorithms is evaluated using four criteria:

- **Completeness:** The algorithm's ability to find a solution if one exists and correctly report failure otherwise.
- **Cost optimality:** Whether the algorithm finds the solution with the lowest path cost.
- **Time complexity:** The duration required to find a solution, measured by the number of states and actions considered.
- **Space complexity:** The memory required to perform the search.

Completeness requires systematic exploration of every state reachable from the initial state. In infinite state spaces, care is necessary to ensure the algorithm systematically covers all reachable states.

The last section that is covered from this chapter this week is **Section 3.4: Uninformed Search Strategies**.

Section 3.4: Uninformed Search Strategies

Overview

This section explores uninformed search algorithms, which are given no clue about how close a state is to the goal(s). These strategies are essential for problems where the agent has no additional information about the domain.

Breadth-First Search

Breadth-first search expands the root node first, then all successors of the root node, and so on. It is a systematic search strategy that is complete even on infinite state spaces.

Breadth-First Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node      NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier      a FIFO queue, with node as an element
    reached     {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node     POP(frontier)
        for each child in EXPAND(problem, node) do
            s     child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
    return failure

```

Breadth-first search finds a solution with a minimal number of actions and is complete and optimal for problems where all actions have the same cost. Its time and space complexity are both $O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution.

Uniform-Cost Search

When actions have different costs, uniform-cost search is used. It expands the node with the lowest path cost from the root.

Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
    return BEST-FIRST-SEARCH(problem, PATH-COST)

```

Uniform-cost search is complete and cost-optimal. Its complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, where C^* is the cost of the optimal solution and ϵ is the smallest positive action cost.

Depth-First Search

Depth-first search expands the deepest node in the frontier first. It is usually implemented as a tree-like search that does not keep a table of reached states.

Depth-First Search

```
function DEPTH-FIRST-SEARCH(problem) returns a solution node or failure
    node     NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier     a LIFO queue, with node as an element
    while not IS-EMPTY(frontier) do
        node     POP(frontier)
        for each child in EXPAND(problem, node) do
            if problem.IS-GOAL(child.STATE) then return child
            add child to frontier
    return failure

```

Depth-first search is not cost-optimal and is incomplete in infinite state spaces. However, it uses less memory, with space complexity of $O(bm)$, where m is the maximum depth of the search tree.

Depth-Limited and Iterative Deepening Search

Depth-limited search imposes a depth limit l and treats nodes at depth l as if they have no successors.

Iterative Deepening Search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
    for depth = 0 to     do
        result     DEPTH-LIMITED-SEARCH(problem, depth)
        if result     cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, l) returns a node, failure, or cutoff
    frontier     a LIFO queue (stack) with NODE(problem.INITIAL) as an element
    result     failure
    while not IS-EMPTY(frontier) do
        node     POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        if DEPTH(node) > l then
            result     cutoff
        else if not IS-CYCLE(node) do
            for each child in EXPAND(problem, node) do
```

```
                add child to frontier
    return result

```

Iterative deepening search combines the benefits of depth-first and breadth-first searches. It is complete and optimal for problems where all actions have the same cost, with time complexity $O(b^d)$ and space complexity $O(bd)$.

Bidirectional Search

Bidirectional search simultaneously searches forward from the initial state and backward from the goal state, hoping that the two searches will meet.

Bidirectional Best-First Search

```
function BIBF-SEARCH(problemF, fF, problemB, fB) returns a solution node, or failure
    nodeF     NODE(problemF.INITIAL)
    nodeB     NODE(problemB.INITIAL)
    frontierF     a priority queue ordered by fF, with nodeF as an element
    frontierB     a priority queue ordered by fB, with nodeB as an element
    reachedF     a lookup table, with one key nodeF.STATE and value nodeF
    reachedB     a lookup table, with one key nodeB.STATE and value nodeB
    solution     failure
    while not TERMINATED(solution, frontierF, frontierB) do
        if fF(TOP(frontierF)) < fB(TOP(frontierB)) then
            solution     PROCEED(F, problemF, frontierF, reachedF, reachedB, solution)
        else
            solution     PROCEED(B, problemB, frontierB, reachedB, reachedF, solution)
    return solution

function PROCEED(dir, problem, frontier, reached, reached2, solution) returns a solution
    node     POP(frontier)
    for each child in EXPAND(problem, node) do
        s     child.STATE
        if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
            reached[s]     child
            add child to frontier
            if s is in reached2 then
                solution2     JOIN-NODES(dir, child, reached2[s])
                if PATH-COST(solution2) < PATH-COST(solution) then
                    solution     solution2
    return solution

```

Bidirectional search is highly efficient, with time and space complexity $O(b^{d/2})$. It is complete and optimal if both search directions use breadth-first or uniform-cost search.

Comparing Uninformed Search Algorithms

The performance of uninformed search algorithms is evaluated based on completeness, optimal cost, time complexity, and space complexity. The comparison is shown in Figure 3.15.

Comparison of Uninformed Search Algorithms

- **Breadth-First Search:** Complete and optimal for uniform-cost problems, time and space complexity $O(b^d)$.
- **Uniform-Cost Search:** Complete and cost-optimal, time and space complexity $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.
- **Depth-First Search:** Not complete or cost-optimal, time complexity $O(b^m)$, space complexity $O(bm)$.
- **Depth-Limited Search:** Not complete or cost-optimal, time and space complexity $O(b^l)$.
- **Iterative Deepening Search:** Complete and optimal for uniform-cost problems, time and space complexity $O(b^d)$.
- **Bidirectional Search:** Complete and optimal, time and space complexity $O(b^{d/2})$.

Informed Search

Informed Search

2.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 3.5 - Informed (Heuristic) Search Strategies**
- **Artificial Intelligence - A Modern Approach - Chapter 3.6 - Heuristic Functions**

2.0.2 Piazza

Must post at least **three** times this week to Piazza.

2.0.3 Lectures

The lectures for this week are:

- Informed Search $\approx$ 22 **min**.
- Heuristics $\approx$ 16 **min**.
- Demo Of Search Algorithms Comparison $\approx$ 19 **min**.

The lecture notes for this week are:

- Heuristics Lecture Notes
- Informed Search Lecture Notes

2.0.4 Assignments

The assignment(s) for this week are:

- Assignment 2 - Informed Search

2.0.5 Quiz

The quiz for this week is:

- Quiz 2 - Informed Search

2.0.6 Chapter Summary

The chapter that is being covered this week is **Chapter 3: Solving Problems By Searching**. The first section that is being covered from this chapter this week is **Section 3.5: Informed (Heuristic) Search Strategies**.

Section 3.5: Informed (Heuristic) Search Strategies

Overview

This section explores informed (heuristic) search strategies, which utilize domain-specific knowledge to guide the search process, making it more efficient compared to uninformed strategies. These strategies are essential for problems where additional information can significantly reduce the search effort.

Heuristics

A heuristic is a technique that helps in solving problems more quickly when classic methods are too slow or fail to find an exact solution. In the context of search algorithms, a heuristic function $h(n)$ provides an estimate of the cost to reach the goal from node n. The better the heuristic, the more efficient the search process.

Heuristic Function

Heuristics guide the search process by estimating how close a given node is to the goal.

- A heuristic function $h(n)$ estimates the cost from node n to the goal.
- Good heuristics reduce the number of nodes expanded during the search.
- Heuristics are problem-specific and must be designed based on domain knowledge.

Greedy Best-First Search

Greedy best-first search expands the node that appears to be closest to the goal, as determined by a heuristic function $h(n)$.

Greedy Best-First Search

This example demonstrates using a heuristic function to prioritize nodes that are closer to the goal.

```
function GREEDY-BEST-FIRST-SEARCH(problem) returns a solution node or failure
    node      NODE(problem.INITIAL)
    frontier      a priority queue ordered by h(n), with node as an element
    while not IS-EMPTY(frontier) do
        node      POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            add child to frontier
    return failure
```

This example shows how the search prioritizes nodes based on their heuristic values.

- Expands nodes that appear to be closest to the goal.
- Complete in finite state spaces but not necessarily optimal.

A* Search

A* search combines the path cost from the start node and the heuristic estimate to the goal to select the node with the lowest combined cost $f(n) = g(n) + h(n)$.

A* Search

This example demonstrates using both path cost and heuristic estimates to guide the search.

```
function A*-SEARCH(problem) returns a solution node or failure
    node      NODE(problem.INITIAL)
    frontier      a priority queue ordered by f(n), with node as an element
    reached      a lookup table with node.STATE as key and node as value
    while not IS-EMPTY(frontier) do
        node      POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s      child.STATE
            if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
                reached[s]      child
                add child to frontier
    return failure
```

This example shows how A* search uses both the path cost and heuristic to find the optimal solution.

- Expands nodes based on combined cost $f(n) = g(n) + h(n)$.
- Complete and optimal with an admissible heuristic.

Memory-Bounded Heuristic Search

Memory-bounded heuristic search algorithms, such as Recursive Best-First Search (RBFS) and Simplified Memory-Bounded A* (SMA*), aim to manage memory usage while maintaining optimality.

Recursive Best-First Search

This example demonstrates RBFS, which uses linear space and backs up values to manage memory.

```
function RBFS(problem) returns a solution node or failure
    return RBFS(problem, NODE(problem.INITIAL),    )

function RBFS(problem, node, f_limit) returns a solution node or failure
    if problem.IS-GOAL(node.STATE) then return node
    successors     EXPAND(problem, node)
    if successors is empty then return failure,
    for each s in successors do
        s.f     max(s.g + s.h, node.f)
    while true do
        best     the node in successors with lowest f value
        if best.f > f_limit then return failure, best.f
        alternative     the second lowest f value among successors
        result, best.f     RBFS(problem, best, min(f_limit, alternative))
        if result     failure then return result

```

This example shows how RBFS maintains linear space complexity by backtracking when necessary.

- Optimal with admissible heuristics.
- Uses linear space, but may re-expand nodes.

Simplified Memory-Bounded A*

SMA* manages memory by expanding the best nodes until memory is full and then dropping the least promising nodes.

- Expands nodes in best-first order until memory is full.
- Drops the worst leaf node when necessary, backing up values.
- Complete if a solution is reachable within memory constraints.
- Optimal if an optimal solution is reachable; otherwise, returns the best reachable solution.

Comparing Informed Search Algorithms

The performance of informed search algorithms is evaluated based on completeness, optimality, time complexity, and space complexity.

Comparison of Informed Search Algorithms

- **Greedy Best-First Search**: Complete in finite spaces, not optimal, time and space complexity $O(|V|)$.
- **A* Search**: Complete and optimal with admissible heuristics, time and space complexity $O(b^d)$.
- **Recursive Best-First Search (RBFS)**: Optimal with admissible heuristics, space complexity $O(bm)$, where m is the maximum depth.
- **Simplified Memory-Bounded A* (SMA*)**: Complete and optimal if sufficient memory, otherwise returns the best reachable solution.

Summary of Key Concepts

Informed (heuristic) search strategies leverage additional knowledge to enhance search efficiency. Here are the key concepts covered:

- **Heuristics**: Guide the search process by providing cost estimates to reach the goal, improving efficiency.
- **Greedy Best-First Search**: Expands nodes closest to the goal based on heuristic values; complete in finite spaces, not necessarily optimal.
- **A* Search**: Combines path cost and heuristic estimate to find the optimal solution; complete and optimal with an admissible heuristic.
- **Recursive Best-First Search (RBFS)**: Uses linear space, backs up values to manage memory, optimal with admissible heuristics.

- **Simplified Memory-Bounded A* (SMA*)**: Expands nodes in best-first order until memory is full, drops least promising nodes, complete and optimal if memory is sufficient.

These strategies make informed search a powerful tool for solving complex problems more efficiently by utilizing domain-specific knowledge.

The last section that is covered from this chapter this week is **Section 3.6: Heuristic Functions**.

Section 3.6: Heuristic Functions

Overview

This section explores heuristic functions, which enhance search performance by providing informed guidance about the likely direction of goal states. We examine the accuracy of heuristics and methods to generate effective heuristics for complex problems. Understanding heuristics is crucial as they can significantly reduce the time and computational resources required to find a solution by making educated guesses about the best paths to follow in a search space.

Heuristics

A heuristic is a method used to estimate the cost from a given node to the goal. It helps in solving problems more efficiently than uninformed strategies by providing domain-specific knowledge. Heuristics are essential in search algorithms because they guide the search process, helping to avoid paths that are unlikely to lead to a solution.

Heuristic Function

Heuristics guide the search process by estimating how close a given node is to the goal. This estimate is not necessarily perfect but should be good enough to make the search more efficient.

- A heuristic function $h(n)$ estimates the cost from node n to the goal.
- Good heuristics reduce the number of nodes expanded during the search, saving time and computational resources.
- Heuristics are problem-specific and must be designed based on domain knowledge, meaning they rely on understanding the problem's structure and constraints.
- For example, in a map navigation problem, the straight-line distance to the destination can serve as a heuristic, indicating how close a location is to the goal.

The Effect of Heuristic Accuracy on Performance

The accuracy of a heuristic significantly impacts the efficiency of the search. An effective heuristic reduces the effective branching factor, making the search faster. The effective branching factor b^* represents the average number of child nodes per parent node in the search tree, and lower values indicate a more efficient search.

Effective Branching Factor

The effective branching factor b^* indicates the quality of a heuristic. A lower b^* value implies a more efficient search, as fewer nodes need to be explored.

- Defined as the branching factor of a uniform tree that generates the same number of nodes as the search.
- Calculated by comparing the total number of nodes generated by the search to the depth of the solution.
- An effective heuristic has a b^* close to 1, meaning it guides the search process very efficiently.
- For instance, if a heuristic reduces the number of nodes explored from thousands to just a few dozen, it

significantly speeds up the search process.

Generating Heuristics from Relaxed Problems

Relaxed problems simplify constraints to make the problem easier to solve, providing admissible heuristics for the original problem. An admissible heuristic never overestimates the true cost to reach the goal, ensuring that the search algorithm remains optimal.

Relaxed Problems

By relaxing the problem's constraints, we can derive admissible heuristics. These simplified problems are easier to solve and give us useful information for the original problem.

- A relaxed problem removes some restrictions, adding edges to the state-space graph, thus making it easier to navigate.
- The cost of an optimal solution to a relaxed problem serves as an admissible heuristic for the original problem because it represents a lower bound on the true cost.
- Examples include the Manhattan distance and misplaced tiles heuristics for the 8-puzzle, where constraints on tile movement are relaxed.
- For example, in the 8-puzzle, if we allow tiles to move anywhere regardless of the blank space, the number of misplaced tiles can serve as a heuristic.

Generating Heuristics from Subproblems: Pattern Databases

Pattern databases store the exact solution costs for subproblems, providing highly accurate admissible heuristics. These databases precompute the optimal solution costs for all possible configurations of a subset of the problem, which can then be used to guide the search in the full problem space.

Pattern Databases

These databases precompute solution costs for subproblems, improving heuristic accuracy by providing exact solution costs for parts of the problem.

- Stores solution costs for every possible configuration of a subset of the problem, making lookups during the search process fast and efficient.
- Lookups in the database provide heuristics for the larger problem, which can drastically reduce the number of nodes that need to be explored.
- Combined heuristics from multiple pattern databases can further enhance performance by using the most accurate estimate available for each state.
- For example, in a sliding-tile puzzle, a pattern database might store the cost of arranging tiles 1, 2, 3, and 4, regardless of the positions of other tiles.

Generating Heuristics with Landmarks

Landmarks provide a way to estimate the cost between nodes by precomputing distances to and from selected landmark points. This method is particularly effective in large graphs, such as maps for navigation.

Landmarks

Using landmarks helps create efficient heuristics for large-scale problems like route-finding by precomputing distances to and from key points.

- Select a few significant points (landmarks) in the graph that are well-distributed across the search space.
- Precompute and store the cost from each vertex to the landmarks, allowing quick lookups during the search.
- Use these precomputed costs to estimate the heuristic for any given node, improving search efficiency.
- For instance, in a city map, landmarks could be major intersections or well-known locations, and the heuristic would estimate travel costs based on these points.

Learning to Search Better

Agents can learn to improve their search strategies over time by using a metalevel state space, which captures the internal state of the search process. This approach allows the agent to learn from past experiences, optimizing future searches.

Metalevel Learning

Learning from experience helps agents to avoid unpromising paths and improve search efficiency by analyzing the search process itself.

- The metalevel state space captures the computational state of the search algorithm, such as which nodes have been expanded and what paths are being considered.
- Metalevel learning algorithms can optimize the search process based on past experiences, identifying patterns that lead to quicker solutions.
- For example, if certain paths frequently lead to dead ends, the agent can learn to avoid those paths in future searches, saving time and resources.
- This approach can be compared to a chess player learning from previous games, refining strategies to avoid losing positions and favor winning ones.

Learning Heuristics from Experience

Heuristics can also be learned from past problem-solving experiences, creating an approximation function that guides the search. This method involves analyzing previously solved problems to derive patterns and features that can predict solution costs.

Learning Heuristics

Learning heuristics from experience involves constructing a function based on solved examples to guide future searches more effectively.

- Use features of the state to predict the heuristic value, such as the number of misplaced tiles in a puzzle or the distance to a goal in a navigation problem.
- Combine multiple features using a linear combination or other methods to create a comprehensive heuristic.
- Balances learning time, search run time, and solution cost, aiming to improve overall efficiency.
- For instance, a learning algorithm might analyze many instances of the 8-puzzle, learning that certain configurations are usually closer to the goal than others, and use this knowledge to speed up future searches.

Summary of Key Concepts

Heuristic functions enhance search efficiency by providing informed estimates of costs to goal states. Here are the key concepts covered:

- **Heuristics**: Guide the search process by providing cost estimates to reach the goal, improving efficiency.
- **Effective Branching Factor**: A measure of the heuristic's efficiency; lower values indicate better performance and faster search processes.
- **Relaxed Problems**: Simplified versions of the original problem used to generate admissible heuristics that provide lower bounds on the cost.
- **Pattern Databases**: Precomputed solution costs for subproblems that provide highly accurate heuristics and significantly reduce search times.
- **Landmarks**: Significant points in the graph used to estimate costs and create heuristics, especially useful in large-scale problems like route-finding.
- **Metalevel Learning**: Learning from the internal states of search algorithms to improve efficiency by avoiding unpromising paths.

- **Learning Heuristics**: Constructing heuristics from past problem-solving experiences to guide future searches, enhancing efficiency and accuracy.

These strategies make heuristic functions a powerful tool for solving complex problems more efficiently by utilizing domain-specific knowledge and learning from experience.

Constraint Satisfaction Problems

3.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 6.1 - Defining Constraint Satisfaction Problems**
- **Artificial Intelligence - A Modern Approach - Chapter 6.2 - Constraint Propagation - Inference In CSPs**
- **Artificial Intelligence - A Modern Approach - Chapter 6.3 - Backtracking Search For CSPs**
- **Artificial Intelligence - A Modern Approach - Chapter 6.4 - Local Search For CSPs**
- **Artificial Intelligence - A Modern Approach - Chapter 6.5 - The Structure Of Problems**

3.0.2 Piazza

Must post at least **three** times this week to Piazza.

3.0.3 Lectures

The lectures for this week are:

- CSP Intro $\approx$ 20 **min**.
- CSP Solving 1: Backtracking & Arc Consistency $\approx$ 21 **min**.
- CSP Ordering & Structure $\approx$ 28 **min**.

The lecture notes for this week are:

- CSP Intro Lecture Notes
- CSP Solving 1 - Backtracking Arc Consistency Lecture Notes
- CSP Solving 2 - Backtracking Arc Consistency Lecture Notes

3.0.4 Assignments

The assignment(s) for this week are:

- Assignment 3 - Constraint Satisfaction Problems

3.0.5 Quiz

The quiz for this week is:

- Quiz 3 - Constraint Satisfaction Problems

3.0.6 Chapter Summary

The chapter that is being covered this week is **Chapter 6: Constraint Satisfaction Problems**. The first section that is covered from this chapter this week is **Section 6.1: Defining Constraint Satisfaction Problems**.

Section 6.1: Defining Constraint Satisfaction Problems

Overview

In this section, we delve into constraint satisfaction problems (CSPs), which are mathematical problems defined by a set of objects whose state must satisfy a number of constraints or limitations. This approach treats states as more than just indivisible entities by using a factored representation, allowing us to use general heuristics to solve complex problems efficiently.

Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three main components: variables, domains, and constraints.

Components of a CSP

- **Variables (X)**: A set of variables $\{X_1, X_2, \ldots, X_n\}$.
- **Domains (D)**: A set of domains $\{D_1, D_2, \ldots, D_n\}$, one for each variable, where each domain D_i consists of a set of allowable values $\{v_1, v_2, \ldots, v_k\}$ for variable X_i.
- **Constraints (C)**: A set of constraints specifying allowable combinations of values. Each constraint C_j consists of a pair $\langle \text{scope}, \text{rel} \rangle$, where scope is a tuple of variables involved in the constraint and rel is a relation defining the permissible values.

CSPs deal with assignments of values to variables. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment assigns values to all variables, and a solution to a CSP is a consistent, complete assignment. Partial assignments leave some variables unassigned, and a partial solution is a consistent partial assignment.

Example Problem: Map Coloring

To illustrate a CSP, consider the problem of coloring a map of Australia such that no two neighboring regions have the same color.

Map Coloring CSP

- **Variables (X)**: The regions $\{WA, NT, Q, NSW, V, SA, T\}$.
- **Domains (D)**: The colors $\{\text{red}, \text{green}, \text{blue}\}$ for each variable.
- **Constraints (C)**: Adjacent regions must have different colors. This results in constraints like $SA \neq WA$, $SA \neq NT$, etc.

Visualizing a CSP as a constraint graph, where nodes represent variables and edges represent constraints, can help understand the problem structure and facilitate efficient problem solving.

Example Problem: Job-Shop Scheduling

In job-shop scheduling, the task is to schedule a series of jobs subject to constraints such as task precedence and resource limitations.

Job-Shop Scheduling CSP

- **Variables (X)**: Tasks involved in assembling a car, such as $\{\text{AxleF}, \text{AxleB}, \text{WheelRF}, \ldots, \text{Inspect}\}$.
- **Domains (D)**: Start times for each task, within a specified range (e.g., 0 to 30 minutes).
- **Constraints (C)**: Precedence constraints (e.g., $\text{AxleF} + 10 \leq \text{WheelRF}$), resource constraints (e.g., only one tool for axle installation), and overall completion time.

CSPs are particularly effective in such scheduling problems due to their ability to prune large portions of the search space by identifying variable assignments that violate constraints.

Variations on the CSP Formalism

CSPs can vary based on the types of variables and constraints involved.

CSP Variations

- **Discrete and Finite Domains**: Variables with a limited set of possible values, such as colors in the map-coloring problem.
- **Infinite Domains**: Variables with an infinite set of possible values, such as start times without a deadline.
- **Continuous Domains**: Variables with continuous values, common in operations research (e.g., scheduling telescope observations).
- **Unary Constraints**: Constraints on a single variable (e.g., $SA \neq$ green).
- **Binary Constraints**: Constraints between two variables (e.g., $SA \neq NSW$).
- **Global Constraints**: Constraints involving multiple variables (e.g., `Alldiff` constraints in Sudoku).
- **Preference Constraints**: Indicating preferred solutions (e.g., scheduling preferences for professors).

Summary of Key Concepts

- **CSPs**: Problems defined by variables, domains, and constraints, where the goal is to find a consistent and complete assignment.
- **Map Coloring Example**: Illustrates CSP with variables as regions, domains as colors, and constraints ensuring adjacent regions have different colors.
- **Job-Shop Scheduling Example**: Demonstrates CSP in scheduling tasks with precedence and resource constraints.
- **CSP Variations**: Covers discrete, finite, infinite, and continuous domains, as well as different types of constraints (unary, binary, global, and preference).

CSPs provide a powerful framework for solving a wide range of problems efficiently by leveraging the structure of the problem to prune the search space.

The next section that is covered from this chapter this week is **Section 6.2: Constraint Propagation - Inference In CSPs**.

Section 6.2: Constraint Propagation - Inference In CSPs

Overview

This section explores constraint propagation, a technique used in constraint satisfaction problems (CSPs) to reduce the number of legal values for variables by enforcing local consistency. Constraint propagation can significantly enhance the efficiency of solving CSPs by reducing the search space and sometimes solving the problem entirely without further search.

Constraint Propagation

Constraint propagation uses the constraints of a CSP to reduce the domains of variables, making it easier to find a solution. This process can be integrated with search or performed as a preprocessing step.

Constraint Propagation

- **Purpose**: Reduce the number of legal values for variables, thereby simplifying the problem.
- **Integration**: Can be intertwined with search or done as preprocessing.
- **Effectiveness**: Sometimes solves the entire problem without further search.

The key idea behind constraint propagation is local consistency. Enforcing local consistency helps eliminate inconsistent values, reducing the domains of variables and making the problem easier to solve.

Node Consistency

A variable is node-consistent if all values in its domain satisfy its unary constraints. Ensuring node consistency involves removing values that violate these constraints.

Node Consistency

- **Definition**: A variable is node-consistent if all values in its domain satisfy its unary constraints.
- **Example**: In the Australia map-coloring problem, if South Australians dislike green, remove green from SA's domain.
- **Implementation**: Reduce the domain of variables with unary constraints at the start of the solving process.

Arc Consistency

A variable is arc-consistent with another variable if every value in its domain satisfies the binary constraint between them. Enforcing arc consistency ensures that all binary constraints are satisfied.

Arc Consistency

- **Definition**: A variable X_i is arc-consistent with respect to X_j if for every value in D_i, there is some value in D_j that satisfies the binary constraint between them.
- **Example**: In a CSP where $Y = X^2$ with domains 0, 1, 2, 3, X is reduced to 0, 1, 2 and Y to 0, 1, 4.
- **Algorithm (AC-3)**: Maintains a queue of arcs and makes each arc consistent, updating domains and rechecking affected arcs.
- **Complexity**: Worst-case time complexity is $O(cd^3)$ for a CSP with n variables, domain size d, and c binary constraints.

Path Consistency

Path consistency extends the idea of arc consistency by considering triples of variables. It ensures that if two variables are consistent, then there exists a consistent value for a third variable.

Path Consistency

- **Definition**: A set $\{X_i, X_j\}$ is path-consistent with X_m if for every consistent assignment $\{X_i = a, X_j = b\}$, there is an assignment to X_m that satisfies all constraints.
- **Example**: In the Australia map-coloring problem with two colors, path consistency shows no solution is possible because three regions touching each other need at least three colors.
- **Purpose**: Helps solve problems that cannot be addressed by arc consistency alone.

K-Consistency

K-consistency generalizes the concept of consistency to any number of variables. A CSP is k-consistent if for any set of $k-1$ variables, a consistent value can always be assigned to the k-th variable.

K-Consistency

- **Definition**: A CSP is k-consistent if for any set of $k-1$ variables, a consistent value can always be assigned to any k-th variable.
- **Levels**:
  - 1-consistency (node consistency)
  - 2-consistency (arc consistency)
  - 3-consistency (path consistency)
- **Strongly K-Consistent**: A CSP is strongly k-consistent if it is k-consistent, (k-1)-consistent, ..., down to 1-consistent.
- **Trade-Off**: Ensuring higher levels of consistency can be computationally expensive and requires more space.

Global Constraints

Global constraints involve multiple variables and are common in real-world CSPs. Special-purpose algorithms can handle these efficiently.

Global Constraints

- **Definition**: Constraints involving an arbitrary number of variables.
- **Examples**:
  - **Alldiff Constraint**: Ensures all variables involved have distinct values.
  - **Resource Constraint (Atmost)**: Limits the total resources used by several tasks.
- **Detection and Enforcement**: Efficient algorithms can detect inconsistencies and enforce constraints, often more effectively than binary constraints.
- **Bounds Propagation**: Manages large integer domains by maintaining and propagating upper and lower bounds.

Sudoku

Sudoku puzzles are a popular example of CSPs, where the objective is to fill a 9x9 grid so that each row, column, and 3x3 box contains the digits 1 to 9 without repetition.

Sudoku

- **Structure**: 81 variables with domains 1, 2, ..., 9 and 27 Alldiff constraints (one for each row, column, and box).
- **Arc Consistency**: Can simplify the puzzle by reducing domains of variables, but often not sufficient alone.
- **Advanced Strategies**: Techniques like "naked triples" enforce stronger consistency and help solve more complex puzzles.

Summary of Key Concepts

- **Constraint Propagation**: Reduces the number of legal values for variables by enforcing local consistency.
- **Node Consistency**: Ensures all values in a variable's domain satisfy unary constraints.
- **Arc Consistency**: Ensures binary constraints are satisfied for all values in variable domains.
- **Path Consistency**: Extends arc consistency to triples of variables.
- **K-Consistency**: Generalizes consistency to any set of k variables, with strong k-consistency ensuring all lower levels of consistency.

- **Global Constraints**: Involves multiple variables, handled by specialized algorithms for efficiency.
- **Sudoku**: A well-known example of CSP demonstrating the application of various consistency techniques.

Constraint propagation techniques enhance the efficiency of solving CSPs by reducing the search space and enforcing various levels of consistency.

The next section that is covered from this chapter this week is **Section 6.3: Backtracking Search For CSPs**.

Section 6.3: Backtracking Search For CSPs

Overview

This section covers backtracking search algorithms for solving constraint satisfaction problems (CSPs). When constraint propagation cannot reduce the domains of all variables to single values, we need to search for a solution. Backtracking search works on partial assignments and is enhanced by various heuristics and inference techniques to improve efficiency and reduce the search space.

Backtracking Search

Backtracking search is a recursive, depth-first search method that extends partial assignments one variable at a time and backtracks when a constraint violation occurs.

Backtracking Search

- **Algorithm**: Extends a partial assignment by selecting an unassigned variable and trying all possible values. If a value leads to a consistent assignment, the search continues; otherwise, it backtracks.
- **Implementation**: Uses a recursive approach similar to depth-first search.
- **Efficiency**: Enhanced by using variable and value ordering heuristics, as well as inference techniques to prune the search space.

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, {})

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var      SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences      INFERENCE(csp, var, assignment)
            if inferences      failure then
                add inferences to csp
                result      BACKTRACK(csp, assignment)
                if result      failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure

```

Variable and Value Ordering

Choosing the right variable and value ordering can significantly reduce the search space and improve efficiency.

Variable and Value Ordering

- **Minimum-Remaining-Values (MRV)**: Selects the variable with the fewest legal values remaining, reducing the likelihood of future conflicts.
- **Degree Heuristic**: Chooses the variable involved in the largest number of constraints with other

unassigned variables, aiming to reduce the branching factor.

- **Least-Constraining-Value**: Prefers the value that leaves the most options open for neighboring variables, maximizing future flexibility.

Interleaving Search and Inference

Combining search with inference can detect inconsistencies earlier and reduce the search space more effectively.

Interleaving Search and Inference

- **Forward Checking**: After assigning a value to a variable, it removes inconsistent values from the domains of neighboring unassigned variables.
- **Maintaining Arc Consistency (MAC)**: Calls AC-3 algorithm during the search to enforce arc consistency and propagate constraints more thoroughly than forward checking.

Intelligent Backtracking: Looking Backward

When a branch fails, intelligent backtracking methods can backtrack to more appropriate points in the search tree, avoiding unnecessary re-evaluations.

Intelligent Backtracking

- **Chronological Backtracking**: Reverts to the most recent variable and tries a different value.
- **Conflict-Directed Backjumping**: Backtracks to the most recent variable in the conflict set that caused the failure, skipping irrelevant variables.
- **Conflict Sets**: Track variables responsible for conflicts, allowing the algorithm to backtrack more effectively.

Constraint Learning

Constraint learning involves identifying and recording sets of assignments that lead to conflicts, known as no-goods, to prevent the algorithm from repeating the same mistakes.

Constraint Learning

- **No-Goods**: Sets of variable assignments that are inconsistent, stored to avoid re-exploring the same conflicting paths.
- **Efficiency**: Reduces redundant searches and improves the overall efficiency of the backtracking algorithm.

Summary of Key Concepts

- **Backtracking Search**: Recursive search method for extending partial assignments and backtracking on conflicts.
- **Variable and Value Ordering**: Heuristics like MRV, degree heuristic, and least-constraining-value to improve search efficiency.
- **Interleaving Search and Inference**: Techniques like forward checking and MAC to enforce consistency during the search.
- **Intelligent Backtracking**: Methods like conflict-directed backjumping to backtrack more effectively based on conflict sets.
- **Constraint Learning**: Recording no-goods to prevent redundant searches and enhance efficiency.

Backtracking search, enhanced by intelligent heuristics and inference techniques, is a powerful method for solving CSPs efficiently by reducing the search space and avoiding redundant work.

The next section that is covered from this chapter this week is **Section 6.4: Local Search For CSPs**.

Section 6.4: Local Search For CSPs

Overview

This section explores local search algorithms for solving constraint satisfaction problems (CSPs). Unlike backtracking search, which works on partial assignments, local search operates on complete assignments and iteratively improves them by making local changes. Local search is particularly effective for large-scale CSPs and can handle problems dynamically as they evolve.

Local Search for CSPs

Local search algorithms begin with a complete assignment of values to variables and attempt to find a solution by iteratively improving the assignment.

Local Search for CSPs

- **Complete-State Formulation**: Starts with a complete assignment, even if it violates constraints.
- **Local Changes**: Iteratively changes the value of one variable at a time to reduce the number of constraint violations.
- **Dynamic Adaptation**: Can handle changes in the problem dynamically, making it suitable for real-time applications.

Min-Conflicts Heuristic

The min-conflicts heuristic is a simple yet powerful technique for local search. It selects a variable with conflicts and assigns it a value that minimizes the number of conflicts.

Min-Conflicts Heuristic

- **Selection**: Choose a variable that is currently in conflict.
- **Assignment**: Assign the variable a value that results in the fewest conflicts with other variables.
- **Effectiveness**: Particularly effective for large CSPs and problems with densely distributed solutions.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
    inputs: csp, a constraint satisfaction problem
            max_steps, the number of steps allowed before giving up
    current     an initial complete assignment for csp
    for i = 1 to max_steps do
        if current is a solution for csp then return current
        var     a randomly chosen conflicted variable from csp.VARIABLES
        value     the value for var that minimizes CONFLICTS(csp, var, value, current)
        set var = value in current
    return failure
```

Example: 8-Queens Problem

The 8-queens problem is a classic example of a CSP that can be effectively solved using local search with the min-conflicts heuristic.

8-Queens Problem

- **Initial State**: Start with a complete assignment where queens are placed on the board, even if they attack each other.
- **Min-Conflicts Heuristic**: Move a queen to a position that minimizes the number of conflicts with

other queens.

- **Efficiency**: The algorithm quickly converges to a solution, even for large instances like the million-queens problem.

Plateau Search and Techniques for Escaping Local Optima

Local search can encounter plateaus where many assignments have the same number of conflicts. Several techniques help escape these local optima.

Escaping Local Optima

- **Plateau Search**: Allows sideways moves to another state with the same score.
- **Tabu Search**: Maintains a list of recently visited states and forbids the algorithm from returning to them.
- **Simulated Annealing**: Uses probabilistic moves to escape local optima by occasionally accepting worse states.
- **Constraint Weighting**: Increases the weight of violated constraints to guide the search towards more critical constraints, adding topography to plateaus and enabling learning over time.

Applications in Dynamic Environments

Local search is particularly well-suited for dynamic environments where the problem constraints can change, such as scheduling problems.

Dynamic Environments

- **Adaptability**: Can quickly adapt to changes in constraints, making minimal adjustments to existing solutions.
- **Example**: Airline scheduling where bad weather disrupts the schedule; local search can reassign flights with minimal disruption.
- **Efficiency**: More efficient than re-running a complete backtracking search from scratch.

Summary of Key Concepts

- **Local Search for CSPs**: Uses a complete-state formulation and iteratively improves assignments.
- **Min-Conflicts Heuristic**: Selects variables in conflict and assigns values to minimize conflicts, effective for large CSPs.
- **Plateau Search and Techniques**: Methods like tabu search, simulated annealing, and constraint weighting help escape local optima.
- **Applications in Dynamic Environments**: Local search's adaptability makes it ideal for real-time and changing CSPs, such as scheduling.

Local search algorithms are powerful tools for solving CSPs, especially in dynamic and large-scale problems, due to their ability to iteratively improve solutions and adapt to changes.

The last section that is covered from this chapter this week is **Section 6.5: The Structure Of Problems**.

Section 6.5: The Structure Of Problems

Overview

This section examines how the structure of a problem, as represented by the constraint graph, can be leveraged to find solutions more efficiently. By understanding and exploiting the structural properties of CSPs, we can often decompose large problems into smaller, more manageable subproblems.

Independent Subproblems

One of the key insights is that many CSPs can be decomposed into independent subproblems, which can be solved separately.

Independent Subproblems

- **Connected Components**: Identifying connected components in the constraint graph allows for decomposition into independent subproblems.
- **Example**: In the Australia map-coloring problem, Tasmania is not connected to the mainland, making it an independent subproblem.
- **Efficiency**: Solving each subproblem separately can drastically reduce the overall complexity from exponential to linear in the number of variables.

Tree-Structured CSPs

Tree-structured CSPs can be solved efficiently using specialized algorithms that take advantage of their hierarchical structure.

Tree-Structured CSPs

- **Definition**: A CSP where the constraint graph forms a tree.
- **Directional Arc Consistency (DAC)**: Ensuring that every variable is arc-consistent with its descendants in a given ordering.
- **Algorithm (TREE-CSP-SOLVER)**: Solves tree-structured CSPs in linear time by first making the tree directed arc-consistent and then assigning values in a topologically sorted order.
- **Complexity**: Time complexity is $O(nd^2)$, where n is the number of variables and d is the domain size.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
    inputs: csp, a CSP with components X, D, C
    n     number of variables in X
    assignment     an empty assignment
    root     any variable in X
    X     TOPOLOGICALSORT(X, root)
    for j = n down to 2 do
        MAKE-ARC-CONSISTENT(PARENT(Xj), Xj)
        if it cannot be made consistent then return failure
    for i = 1 to n do
        assignment[Xi]     any consistent value from Di
        if there is no consistent value then return failure
    return assignment

```

Cutset Conditioning

Cutset conditioning is a technique to transform a general constraint graph into a tree by fixing some variables, turning the remaining graph into a tree.

Cutset Conditioning

- **Cycle Cutset**: A subset of variables whose removal makes the constraint graph a tree.
- **Procedure**: Assign values to variables in the cycle cutset and solve the remaining tree-structured CSP.
- **Complexity**: Time complexity is $O(d^c \cdot (n - c)d^2)$, where c is the size of the cycle cutset.
- **Efficiency**: Effective if the cycle cutset is small, significantly reducing the problem complexity.

Tree Decomposition

Tree decomposition transforms the constraint graph into a tree of clusters, each containing a subset of variables, facilitating efficient problem solving.

Tree Decomposition

- **Definition**: Converts the original graph into a tree where each node represents a cluster of variables.
- **Conditions**:
  - Every variable appears in at least one cluster.
  - If two variables are connected by a constraint, they appear together in at least one cluster.
  - If a variable appears in two clusters, it must appear in all clusters along the path connecting them.
- **Algorithm**: Solve the tree-structured CSP formed by the clusters.
- **Complexity**: Time complexity is $O(nd^w)$, where w is the tree width.
- **Tree Width**: The minimum width among all tree decompositions of the graph.

Value Symmetry

Value symmetry occurs when multiple equivalent solutions exist due to interchangeable values, and breaking this symmetry can reduce the search space.

Value Symmetry

- **Definition**: Permutations of values that result in equivalent solutions.
- **Example**: In the map-coloring problem with d colors, there are $d!$ equivalent solutions for permuting colors.
- **Symmetry-Breaking Constraint**: Introduces constraints to reduce equivalent solutions and thus the search space.

Summary of Key Concepts

- **Independent Subproblems**: Decompose CSPs into smaller subproblems using connected components.
- **Tree-Structured CSPs**: Solved efficiently using directional arc consistency and topological sorting.
- **Cutset Conditioning**: Reduces general graphs to trees by fixing a subset of variables, solving the simplified problem.
- **Tree Decomposition**: Converts the graph into a tree of clusters, simplifying the problem solving process.
- **Value Symmetry**: Reduces the search space by breaking value symmetries with additional constraints.

Understanding the structure of CSPs and leveraging these techniques can greatly enhance the efficiency of finding solutions by exploiting problem decomposition and structural properties.

Games

Games

4.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 5.1 - Game Theory**
- **Artificial Intelligence - A Modern Approach - Chapter 5.2 - Optimal Decisions In Games**
- **Artificial Intelligence - A Modern Approach - Chapter 5.3 - Heuristic Alpha-Beta Tree Search**
- **Artificial Intelligence - A Modern Approach - Chapter 5.4 - Monte Carlo Tree Search**
- **Artificial Intelligence - A Modern Approach - Chapter 5.5 - Partially Observable Games**

4.0.2 Piazza

Must post at least **three** times this week to Piazza.

4.0.3 Lectures

The lectures for this week are:

- Advesarial Search Intro, Minimax Algorith $\approx$ 27 **min**.
- Improving Minimax: Alpha-Beta Pruning And Depth-Limited Approach $\approx$ 22 **min**.
- Expectimax Search $\approx$ 28 **min**.

The lecture notes for this week are:

- Advesarial Search Intro, Minimax Algorith Lecture Notes
- Expectimax Search Lecture Notes
- Improving Minimax - Alpha-Beta Pruning And Depth-Limited Approach Lecture Notes

4.0.4 Assignments

The assignment(s) for this week are:

- Assignment 4 - Games

4.0.5 Quiz

The quiz for this week is:

- Quiz 4 - Games

4.0.6 Chapter Summary

The chapter that is being covered this week is **Chapter 5: Adversarial Search And Games**. The first section that is being covered from this chapter this week is **Section 5.1: Game Theory**.

Section 5.1: Game Theory

Overview

Key topics include two-player zero-sum games, game trees, minimax search, and heuristic evaluation functions. These concepts are crucial for understanding how to model and solve competitive environments where agents have conflicting goals.

Two-Player Zero-Sum Games

Two-player zero-sum games are a fundamental concept in game theory where two players take turns making moves, and one player's gain is the other player's loss. Examples include chess, Go, and tic-tac-toe.

Two-Player Zero-Sum Games

In two-player zero-sum games, each player's goal is to maximize their own payoff while minimizing their opponent's payoff.

- **Perfect Information**: Both players have complete knowledge of the game state at all times.
- **Zero-Sum**: The total payoff to all players is zero, meaning one player's gain is exactly the other player's loss.
- **Game Representation**: The game can be represented using states, actions, results, terminal tests, and utility functions.

Game Trees

A game tree is a theoretical construct that represents all possible moves in a game, starting from the initial state and expanding through all possible sequences of moves to terminal states.

Game Trees

Game trees illustrate the decision-making process in adversarial games by representing all possible game states and actions.

- **Initial State**: The configuration of the game at the beginning.
- **Actions**: The set of legal moves available to the player whose turn it is to move.
- **Result**: The state resulting from a specific action taken in a given state.
- **Terminal Test**: A condition that determines whether the game has ended.
- **Utility Function**: A function that assigns a numeric value to each terminal state, indicating the payoff for the players.

Minimax Search

Minimax search is an algorithm used to determine the optimal strategy in two-player zero-sum games by assuming both players play optimally. It recursively evaluates the minimax value of each state in the game tree.

Minimax Search

Minimax search computes the best move by evaluating the minimax values of states, assuming optimal play by both players.

- **Minimax Value**: The utility of a state assuming both players play optimally.
- **MAX Player**: Seeks to maximize the minimax value.
- **MIN Player**: Seeks to minimize the minimax value.
- **Ply**: One level of moves by each player.

Heuristic Evaluation Functions

Heuristic evaluation functions estimate the value of a game state when the game tree is too large to be fully explored. These functions approximate the minimax value based on features of the state.

Heuristic Evaluation Functions

Heuristic evaluation functions provide an estimate of the game state's value, allowing for effective decision-making without exhaustive search.

- **State Features**: Characteristics of the game state used to compute the heuristic value.
- **Approximation**: Provides an estimate of the true minimax value.
- **Efficiency**: Enables decision-making in complex games where full search is impractical.

Key Concepts

Key Concepts

This section summarizes the key concepts related to game theory and adversarial search, emphasizing their definitions, properties, and applications in AI.

- **Two-Player Zero-Sum Games**
  - **Perfect Information**: Complete knowledge of the game state.
  - **Zero-Sum**: One player's gain is the other player's loss.
  - **Game Representation**: States, actions, results, terminal tests, and utility functions.
- **Game Trees**
  - **Initial State**: Configuration at the game's start.
  - **Actions**: Legal moves available to players.
  - **Result**: Outcome of a specific action.
  - **Terminal Test**: Determines if the game has ended.
  - **Utility Function**: Numeric value assigned to terminal states.
- **Minimax Search**
  - **Minimax Value**: Utility assuming optimal play.
  - **MAX Player**: Maximizes the minimax value.
  - **MIN Player**: Minimizes the minimax value.
  - **Ply**: One level of moves by each player.
- **Heuristic Evaluation Functions**
  - **State Features**: Characteristics used to compute heuristic value.
  - **Approximation**: Estimate of the true minimax value.
  - **Efficiency**: Facilitates decision-making in complex games.

The next section that is being covered from this chapter this week is **Section 5.2: Optimal Decisions In Games**.

Section 5.2: Optimal Decisions In Games

Overview

Key topics include the minimax search algorithm, alpha-beta pruning, and optimal decisions in multiplayer games. These concepts are crucial for understanding how to model and solve competitive environments where agents have conflicting goals.

Minimax Search Algorithm

The minimax search algorithm is used to determine the optimal strategy in two-player zero-sum games by assuming both players play optimally. It evaluates the minimax value of each state in the game tree.

Minimax Search Algorithm

Minimax search computes the best move by evaluating the minimax values of states, assuming optimal play by both players.

- **Minimax Value**: The utility of a state assuming both players play optimally.
- **MAX Player**: Seeks to maximize the minimax value.
- **MIN Player**: Seeks to minimize the minimax value.
- **Ply**: One level of moves by each player.
- **Algorithm**: Recursively explores the game tree and backs up values to determine the optimal move.

Alpha-Beta Pruning

Alpha-beta pruning is a technique used to improve the efficiency of the minimax algorithm by eliminating branches in the game tree that do not influence the final decision.

Alpha-Beta Pruning

Alpha-beta pruning reduces the number of nodes evaluated in the game tree by pruning branches that cannot affect the final decision.

- **Pruning Condition**: Stops evaluation of a move when at least one possibility has been found that proves the move to be worse than a previously examined move.
- **Alpha Value**: The best value that the maximizer currently can guarantee at that level or above.
- **Beta Value**: The best value that the minimizer currently can guarantee at that level or above.
- **Efficiency**: Reduces the effective branching factor, allowing deeper search in the same amount of time.

Optimal Decisions in Multiplayer Games

Extending the minimax approach to multiplayer games involves considering a vector of utilities for each player and recognizing that optimal strategies may involve forming and breaking alliances.

Optimal Decisions in Multiplayer Games

Multiplayer games require a more complex strategy as they involve multiple agents with potentially conflicting goals.

- **Utility Vector**: Represents the utility of a state for each player.
- **Alliances**: Players may form alliances to improve their individual outcomes, although these alliances may be temporary.
- **Non-Zero-Sum**: In non-zero-sum games, collaboration can occur to achieve mutually beneficial outcomes.

Move Ordering

The effectiveness of alpha-beta pruning is highly dependent on the order in which nodes are examined. Good move ordering can significantly enhance the performance of the search.

Move Ordering

Effective move ordering improves the performance of alpha-beta pruning by exploring the most promising moves first.

- **Killer Moves**: Moves that have been successful in the past are tried first.

- **Dynamic Ordering**: Uses iterative deepening to refine move ordering based on earlier searches.
- **Transposition Tables**: Cache heuristic values of previously evaluated states to avoid redundant calculations.

Key Concepts

Key Concepts

This section summarizes the key concepts related to making optimal decisions in games, emphasizing their definitions, properties, and applications in AI.

- **Minimax Search Algorithm**
  - **Minimax Value**: Utility assuming optimal play.
  - **MAX Player**: Maximizes the minimax value.
  - **MIN Player**: Minimizes the minimax value.
  - **Ply**: One level of moves by each player.
  - **Algorithm**: Recursively explores the game tree.
- **Alpha-Beta Pruning**
  - **Pruning Condition**: Stops evaluation when a move is proven worse.
  - **Alpha Value**: Best value maximizer can guarantee.
  - **Beta Value**: Best value minimizer can guarantee.
  - **Efficiency**: Reduces effective branching factor.
- **Optimal Decisions in Multiplayer Games**
  - **Utility Vector**: Represents utility for each player.
  - **Alliances**: Temporary partnerships to improve outcomes.
  - **Non-Zero-Sum**: Collaboration for mutual benefit.
- **Move Ordering**
  - **Killer Moves**: Successful moves tried first.
  - **Dynamic Ordering**: Refines move order through iterative deepening.
  - **Transposition Tables**: Cache heuristic values to avoid redundant calculations.

The next section that is being covered from this chapter this week is **Section 5.3: Heuristic Alpha-Beta Tree Search**.

Section 5.3: Heuristic Alpha-Beta Tree Search

Overview

Key topics include heuristic evaluation functions, cutting off search, quiescence search, the horizon effect, forward pruning, and the use of lookup tables. These concepts are crucial for optimizing search strategies in complex games and making efficient use of computational resources.

Heuristic Evaluation Functions

Heuristic evaluation functions estimate the utility of a game state when the search tree is too large to fully explore. These functions approximate the minimax value based on features of the state.

Heuristic Evaluation Functions

Heuristic evaluation functions provide an estimate of a game state's value, allowing effective decision-making without exhaustive search.

- **EVAL Function**: Replaces the UTILITY function for nonterminal nodes.
- **Features**: State characteristics (e.g., number of pawns, knights) used to compute the evaluation.
- **Weighted Linear Function**: Combines feature values with weights to estimate utility.
- **Correlation**: Should be strongly correlated with the actual chances of winning.

Cutting Off Search

Cutting off search involves stopping the search at a certain depth or when a certain condition is met, and applying the heuristic evaluation function to estimate the state's value.

Cutting Off Search

The search is cut off early to save computation time, and the evaluation function is applied to nonterminal states.

- **Cutoff Test**: Determines when to stop the search based on depth or state properties.
- **Fixed Depth Limit**: A simple approach to control search amount by setting a maximum depth.
- **Iterative Deepening**: Searches incrementally deeper until time runs out, using results to improve move ordering.

Quiescence Search

Quiescence search continues the search until a stable position (quiescent) is reached, avoiding misleading evaluations due to volatile positions.

Quiescence Search

Quiescence search extends the search to stable positions, avoiding drastic evaluation changes due to immediate threats or opportunities.

- **Quiescent Positions**: Stable states where no dramatic changes in evaluation are expected.
- **Extra Search**: Searches beyond the cutoff for non-quiescent positions to resolve uncertainties.

Horizon Effect

The horizon effect occurs when a program cannot see beyond a certain depth, causing it to miss future significant events or threats.

Horizon Effect

The horizon effect arises when the search is unable to see beyond a certain depth, leading to inaccurate evaluations.

- **Delaying Tactics**: Moves that postpone the inevitable, pushing significant events beyond the search horizon.
- **Singular Extensions**: Extending search for clearly superior moves to mitigate the horizon effect.

Forward Pruning

Forward pruning selectively prunes moves that appear to be poor, saving computation time at the risk of missing the best move.

Forward Pruning

Forward pruning saves computation time by pruning moves that are unlikely to be good, at the risk of making errors.

- **Beam Search**: Considers only a "beam" of the best moves at each ply.
- **PROBCUT**: Prunes moves that are probably outside the current window based on statistical estimates.
- **Late Move Reduction**: Reduces the search depth for moves considered less promising.

Search vs. Lookup

Using table lookup for well-known positions in the opening and endgame stages can significantly enhance performance by relying on precomputed solutions.

Search vs. Lookup

Table lookup is used for known positions, especially in the opening and endgame stages, to enhance performance.

- **Opening Books**: Precomputed sequences of optimal moves in the opening stage.
- **Endgame Tables**: Complete solutions for endgames, allowing perfect play by looking up the best move.
- **Retrograde Minimax Search**: Constructs endgame tables by reversing the rules to determine winning moves.

Key Concepts

Key Concepts

This section summarizes the key concepts related to heuristic alpha-beta tree search, emphasizing their definitions, properties, and applications in AI.

- **Heuristic Evaluation Functions**
  - **EVAL Function**: Estimates utility for nonterminal nodes.
  - **Features**: State characteristics used in evaluation.
  - **Weighted Linear Function**: Combines feature values with weights.
  - **Correlation**: Should align with actual chances of winning.
- **Cutting Off Search**
  - **Cutoff Test**: Decides when to stop search.
  - **Fixed Depth Limit**: Simple approach to control search depth.
  - **Iterative Deepening**: Incremental search for better move ordering.
- **Quiescence Search**
  - **Quiescent Positions**: Stable states for reliable evaluation.
  - **Extra Search**: Extends search to avoid volatile evaluations.
- **Horizon Effect**
  - **Delaying Tactics**: Postponing significant events.
  - **Singular Extensions**: Extends search for superior moves.
- **Forward Pruning**
  - **Beam Search**: Considers top moves at each ply.
  - **PROBCUT**: Uses statistics to prune unlikely moves.
  - **Late Move Reduction**: Reduces depth for less promising moves.

- **Search vs. Lookup**
  - **Opening Books**: Precomputed opening sequences.
  - **Endgame Tables**: Perfect play solutions for endgames.
  - **Retrograde Minimax Search**: Constructs endgame tables by reversing rules.

The next section that is being covered from this chapter this week is **Section 5.4: Monte Carlo Tree Search**.

Section 5.4: Monte Carlo Tree Search

Overview

Key topics include the basics of Monte Carlo Tree Search (MCTS), the selection, expansion, simulation, and back-propagation steps, upper confidence bounds applied to trees (UCT), and the comparison of MCTS with alpha-beta search. These concepts are essential for understanding how MCTS can be used to make decisions in complex games with high branching factors and uncertain evaluation functions.

Basics of Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a strategy that estimates the value of a state by averaging the results of numerous simulated games played from that state to termination.

Basics of Monte Carlo Tree Search (MCTS)

MCTS estimates state values through simulations, providing a robust method for decision-making in complex games.

- **Simulation**: Runs complete games from a given state to terminal positions, using random or biased moves.
- **Average Utility**: The state value is estimated by the average utility (win percentage) from simulations.
- **Playout Policy**: Guides move selection during simulations to improve accuracy.

Selection, Expansion, Simulation, and Back-Propagation

MCTS involves four main steps: selection, expansion, simulation, and back-propagation. These steps iteratively build the search tree and update the estimated values.

Selection, Expansion, Simulation, and Back-Propagation

The four steps of MCTS iteratively build the search tree and refine state value estimates.

- **Selection**: Navigates the tree from the root to a leaf node, guided by a selection policy.
- **Expansion**: Adds new child nodes to the tree from the selected node.
- **Simulation**: Runs a playout from the newly added node to a terminal state.
- **Back-Propagation**: Updates the values of nodes along the path from the new node to the root.

Upper Confidence Bounds Applied to Trees (UCT)

UCT is a popular selection policy for MCTS that balances exploration and exploitation using a mathematical formula.

Upper Confidence Bounds Applied to Trees (UCT)

UCT guides the selection step in MCTS by balancing the need to explore new moves and exploit known good moves.

- **Exploitation Term**: $\frac{U(n)}{N(n)}$ represents the average utility of node n.
- **Exploration Term**: $C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$ encourages exploring less-visited nodes.
- **Balancing Constant** C: Adjusts the balance between exploration and exploitation.

Comparison with Alpha-Beta Search

MCTS and alpha-beta search have different strengths and weaknesses, particularly in games with high branching factors and complex evaluation functions.

Comparison with Alpha-Beta Search

MCTS and alpha-beta search each have advantages depending on the game's characteristics and evaluation function accuracy.

- **Branching Factor**: MCTS handles high branching factors better than alpha-beta search.
- **Evaluation Function**: MCTS does not rely on an accurate heuristic evaluation function.
- **Robustness**: MCTS is less susceptible to single-point evaluation errors.
- **Hybrid Approaches**: Combining aspects of MCTS and alpha-beta can leverage the strengths of both.

Key Concepts

Key Concepts

This section summarizes the key concepts related to Monte Carlo Tree Search, emphasizing their definitions, properties, and applications in AI.

- **Basics of Monte Carlo Tree Search (MCTS)**
  - **Simulation**: Runs complete games to terminal positions.
  - **Average Utility**: State value estimated by simulation results.
  - **Playout Policy**: Guides move selection during simulations.
- **Selection, Expansion, Simulation, and Back-Propagation**
  - **Selection**: Navigates the tree to a leaf node.
  - **Expansion**: Adds new child nodes.
  - **Simulation**: Runs a playout to a terminal state.
  - **Back-Propagation**: Updates node values along the path.
- **Upper Confidence Bounds Applied to Trees (UCT)**
  - **Exploitation Term**: Represents average utility.
  - **Exploration Term**: Encourages exploring less-visited nodes.
  - **Balancing Constant** C: Adjusts exploration-exploitation balance.
- **Comparison with Alpha-Beta Search**
  - **Branching Factor**: MCTS handles high branching factors well.
  - **Evaluation Function**: MCTS does not rely on heuristic evaluation.
  - **Robustness**: Less susceptible to single-point errors.
  - **Hybrid Approaches**: Combines MCTS and alpha-beta strengths.

The last section that is being covered from this chapter this week is **Section 5.5: Partially Observable Games**.

Section 5.5: Partially Observable Games

Overview

Key topics include the nature of partially observable games, the example of Kriegspiel (partially observable chess), belief states, and strategies for handling partial observability in games. These concepts are essential for understanding how to model and solve games where players have limited information about the game state.

Nature of Partially Observable Games

Partially observable games are characterized by the players' incomplete information about the game state, leading to uncertainty and the need for strategies that account for this lack of information.

Nature of Partially Observable Games

Partially observable games involve uncertainty due to players' incomplete information about the game state, requiring strategies that manage this uncertainty.

- **Partial Observability**: Players do not have full information about the game state.
- **Information Gathering**: Use of scouts and spies to gather information about the opponent's state.
- **Concealment and Bluffing**: Techniques to hide information and mislead the opponent.

Kriegspiel: Partially Observable Chess

Kriegspiel is a variant of chess where each player can only see their own pieces. A referee, who sees all pieces, adjudicates the game and provides limited information to the players.

Kriegspiel: Partially Observable Chess

Kriegspiel is a partially observable variant of chess where players only see their own pieces, and a referee provides limited information about the opponent's moves.

- **Referee Announcements**: The referee announces information such as captures and checks.
- **Move Proposals**: Players propose moves to the referee, who indicates if the move is legal or not.
- **Information Gained**: Players gain information about the opponent's pieces based on the legality of proposed moves.

Belief States

In partially observable games, belief states represent the set of all possible configurations of the game state that are consistent with the player's observations and actions so far.

Belief States

Belief states encapsulate all possible game states that are consistent with the player's observations and actions, allowing for decision-making under uncertainty.

- **Initial Belief State**: The starting point, often a single known state.
- **Update Mechanism**: Belief states are updated based on new observations and actions.
- **Representation**: Belief states can be represented as a set or a probability distribution over possible states.

Strategies for Partial Observability

Strategies in partially observable games must account for the uncertainty and use information-gathering techniques to improve decision-making.

Strategies for Partial Observability

Effective strategies in partially observable games focus on managing uncertainty and gathering information to improve decision-making.

- **Information Gathering**: Actively seeking information about the opponent's state through probing actions.
- **Conservative Play**: Avoiding high-risk moves that rely on uncertain information.
- **Belief State Management**: Continuously updating and refining belief states based on new information.

Key Concepts

Key Concepts

This section summarizes the key concepts related to partially observable games, emphasizing their definitions, properties, and applications in AI.

- **Nature of Partially Observable Games**
  - **Partial Observability**: Incomplete information about the game state.
  - **Information Gathering**: Use of scouts and spies.
  - **Concealment and Bluffing**: Techniques to mislead the opponent.
- **Kriegspiel: Partially Observable Chess**
  - **Referee Announcements**: Provides limited information about moves.
  - **Move Proposals**: Players propose moves to the referee.
  - **Information Gained**: Based on the legality of proposed moves.
- **Belief States**
  - **Initial Belief State**: Starting point of known states.
  - **Update Mechanism**: Updating beliefs with new observations.
  - **Representation**: Set or probability distribution of states.
- **Strategies for Partial Observability**
  - **Information Gathering**: Probing actions to gather information.
  - **Conservative Play**: Avoiding risky moves.
  - **Belief State Management**: Refining beliefs with new information.

Midterm 1

Midterm 1

5.0.1 Exam

The exam for this week is:

- Exam 1 Notes
- Exam 1

Markov Decision Problem (MDP)

6.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 17.1 - Sequential Decision Problems**
- **Artificial Intelligence - A Modern Approach - Chapter 17.2 - Algorithms For MDPs**
- **Artificial Intelligence - A Modern Approach - Chapter 17.3 - Bandit Problems**
- **Reinforcement Learning - An Introduction - Chapter 3 - Finite Markov Decision Processes**
- **Reinforcement Learning - An Introduction - Chapter 4 - Dynamic Programming**

6.0.2 Piazza

Must post at least **three** times this week to Piazza.

6.0.3 Lectures

The lectures for this week are:

- Markov-Decision Process $\approx$ 43 **min**.
- Solving MDP - Dynamic Programming $\approx$ 62 **min**.

The lecture notes for this week are:

- Markov Decision Process (MDP) Lecture Notes
- Markov Decision Process (MDP) Solved Problems Lecture Notes

6.0.4 Quiz

The quiz for this week is:

- Quiz 5 - Markov Decision Problem (MDP)

6.0.5 Chapter Summary

The reading for this week is from **Artificial Intelligence - A Modern Approach** and **Reinforcement Learning - An Introduction**. The chapter that is being covered from **Artificial Intelligence - A Modern Approach** is **Chapter 17: Making Complex Decisions**. The first section that is being covered from this chapter this week is **Section 17.1: Sequential Decision Problems**.

Section 17.1: Sequential Decision Problems

Overview

This section introduces sequential decision problems, where the agent's utility depends on a sequence of decisions rather than a single decision. These problems incorporate utilities, uncertainty, and sensing, and include search and planning problems as special cases. The complexity arises from the need to make decisions at each step, considering both immediate and future consequences. Sequential decision problems are more realistic and complex compared to single-step decision problems because they reflect the ongoing nature of decision-making in real-world scenarios.

Sequential Decision Problems

In sequential decision problems, an agent must choose actions over time, considering the stochastic nature of the environment and aiming to maximize its cumulative reward. Unlike one-shot decisions, these problems require the agent to plan ahead, anticipating the outcomes of its actions and the subsequent decisions it will need to make.

Sequential Decision Problems

- **Environment**: The agent interacts with a stochastic environment, where actions have uncertain outcomes. This uncertainty can arise from various sources, such as environmental variability or incomplete knowledge.
- **States and Actions**: The agent's state changes based on the actions it takes, with each action leading to a new state with certain probabilities. The set of possible actions and their effects on the state define the transition model.
- **Rewards**: The agent receives rewards for transitions between states, which are used to define the utility of sequences of actions and states. These rewards provide immediate feedback to the agent about the desirability of its actions.

Example: 4x3 Grid Environment

Consider an agent navigating a 4x3 grid where it must reach a goal state while avoiding pitfalls. The environment is stochastic, with actions resulting in intended moves with high probability but sometimes causing unintended moves. This example illustrates the complexities introduced by uncertainty in action outcomes.

4x3 Grid Environment

- **States**: Each cell in the grid represents a state. The agent starts at a specific cell and aims to reach a goal cell.
- **Actions**: Possible actions are Up, Down, Left, and Right. The agent selects an action at each step, intending to move in the chosen direction.
- **Transition Model**: Actions succeed with probability 0.8 and fail with probability 0.2, causing the agent to move at right angles to the intended direction. This model captures the uncertainty and potential for error in executing actions.
- **Rewards**: Transitioning into the goal state yields a reward of +1, while falling into a pit results in a reward of -1. Other transitions have a small negative reward (e.g., -0.04) to encourage reaching the goal quickly. These rewards reflect the agent's objectives and penalties for undesirable outcomes.

Markov Decision Processes (MDPs)

Sequential decision problems in fully observable, stochastic environments with Markovian transitions are formalized as Markov decision processes (MDPs). MDPs provide a mathematical framework for modeling decision-making where outcomes are partly random and partly under the control of the decision-maker.

Markov Decision Processes (MDPs)

- **Components**: An MDP consists of states S, actions A, a transition model $P(s'|s,a)$, and a reward function $R(s,a,s')$. These components define the dynamics of the environment and the agent's interactions with it.
- **Policy**: A policy π specifies the action $\pi(s)$ to take in each state s. An optimal policy π^* maximizes the expected utility over all possible sequences of states and actions, guiding the agent's behavior to achieve the best possible outcomes.
- **Utility of a State**: The expected utility $U(s)$ of a state s under an optimal policy is the sum of the expected rewards from that state onward. This measure reflects the long-term value of being in a particular state.
- **Bellman Equation**: The utility of a state $U(s)$ is given by $U(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')]$, where γ is the discount factor. The Bellman equation captures the recursive nature of decision-making, where the utility of a state depends on the utilities of subsequent states.

Utilities Over Time

The utility function for MDPs can be defined over a finite or infinite horizon, affecting how the agent values future rewards. Different horizons lead to different decision strategies and complexities in solving the MDP.

Utilities Over Time

- **Finite Horizon**: There is a fixed time limit after which future rewards do not matter. This scenario is suitable for tasks with clear endpoints or deadlines.
- **Infinite Horizon**: No fixed time limit; the agent seeks to maximize the sum of discounted rewards over an infinite sequence of actions. This approach models ongoing tasks without a predefined end.
- **Discount Factor** (γ): A value between 0 and 1 that determines the present value of future rewards. A higher γ means future rewards are valued more highly, encouraging long-term planning. The discount factor captures the trade-off between immediate and future rewards, reflecting the agent's preference for immediate versus delayed gratification.

Optimal Policies and State Utilities

Optimal policies maximize the expected utility of the agent, and the utility of each state under the optimal policy can be computed using the Bellman equation. These concepts are central to finding the best strategies in MDPs.

Optimal Policies and State Utilities

- **Optimal Policy** (π^*): The policy that yields the highest expected utility from any given state. It guides the agent's actions to maximize long-term rewards.
- **State Utility** ($U(s)$): The expected sum of discounted rewards from state s under the optimal policy. This value indicates the desirability of being in a particular state, considering future rewards.
- **Action-Utility Function (Q-Function)**: $Q(s, a)$ represents the expected utility of taking action a in state s, following the optimal policy thereafter. The Q-function helps in evaluating the immediate benefit of actions in the context of long-term planning.

Reward Scales and Transformations

The scale of rewards can be transformed without changing the optimal policy, providing flexibility in defining reward functions. This property allows for different representations of rewards while preserving the decision-making framework.

Reward Scales and Transformations

- **Affine Transformation**: Replacing $R(s, a, s')$ with $mR(s, a, s') + b$ (where $m > 0$) does not change the optimal policy. This transformation scales and shifts the rewards uniformly, maintaining the relative preferences between actions.
- **Shaping Theorem**: Adding a potential-based reward $\gamma\Phi(s') - \Phi(s)$ to $R(s, a, s')$ does not alter the optimal policy, but can make the immediate rewards more informative. This technique helps in guiding the agent towards desirable states more effectively by modifying the reward structure.

Representing MDPs

MDPs can be represented using various models, including big tables for small problems or dynamic decision networks (DDNs) for larger, more complex problems. Proper representation is crucial for the efficient solution of MDPs.

Representing MDPs

- **Tabular Representation**: Uses three-dimensional tables to store transition probabilities and rewards. This method is straightforward but can become impractical for large state spaces due to memory constraints.
- **Dynamic Decision Networks (DDNs)**: Extend dynamic Bayesian networks (DBNs) with decision,

reward, and utility nodes to model more complex, real-world problems. DDNs provide a structured and compact representation, capturing dependencies between variables efficiently.

- **Example**: A mobile robot with state variables for location, velocity, charging status, and battery level, and action variables for movement and charging decisions. This example illustrates how DDNs can represent the interdependencies and dynamic nature of the robot's decision-making process.

Summary of Key Concepts

- **Sequential Decision Problems**: Involve making a series of decisions over time in a stochastic environment. These problems are more realistic and complex than single-step decisions, reflecting the ongoing nature of real-world tasks.
- **Markov Decision Processes (MDPs)**: Formalize sequential decision problems with states, actions, transition models, and reward functions. MDPs provide a robust framework for modeling and solving complex decision-making problems under uncertainty.
- **Utilities and Optimal Policies**: Utilities are defined as the expected sum of discounted rewards, and optimal policies maximize these utilities. These concepts are central to identifying the best strategies in MDPs.
- **Utilities Over Time**: Can be finite or infinite horizon, with a discount factor determining the value of future rewards. The choice of horizon and discount factor affects the agent's planning and decision-making strategies.
- **Reward Transformations**: Affine transformations and potential-based rewards can modify reward functions without changing the optimal policy. These transformations offer flexibility in defining rewards while preserving the decision-making framework.
- **Representation of MDPs**: Tabular and dynamic decision networks are used to model MDPs, with DDNs being suitable for complex problems. Proper representation is crucial for efficiently solving MDPs and capturing the dependencies in decision-making processes.

Understanding sequential decision problems and their representation through MDPs is crucial for developing intelligent agents capable of making optimal decisions in uncertain environments. These concepts form the foundation for advanced techniques in decision-making and planning under uncertainty.

The next section that is being covered from this chapter this week is **Section 17.2: Algorithms For MDPs**.

Section 17.2: Algorithms For MDPs

Overview

This section presents four different algorithms for solving Markov Decision Processes (MDPs). These algorithms include value iteration, policy iteration, linear programming, and online approximate algorithms like Monte Carlo planning. Each algorithm offers a different approach to finding optimal policies and utilities in MDPs, addressing various complexities and computational challenges.

Value Iteration

Value iteration is a fundamental algorithm for solving MDPs based on the Bellman equation. It iteratively updates the utility values of states until they converge to the optimal values. This process helps determine the best action to take in each state to maximize cumulative rewards.

Value Iteration

- **Bellman Equation**: The Bellman equation $U(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')]$ is the foundation of value iteration. It recursively defines the utility of a state in terms of the utilities of successor states.

- **Algorithm**: The value iteration algorithm updates the utility values of all states simultaneously using the Bellman update until the utilities converge within a specified threshold.
- **Convergence**: The algorithm converges to the optimal utility values exponentially fast, ensuring the computed policy is optimal. The convergence rate depends on the discount factor γ.

```
function VALUE-ITERATION(mdp, e) returns a utility function
    inputs: mdp, an MDP with states S, actions A(s), transition model P(s' |s,a),
            rewards R(s,a,s'), discount g
            e, the maximum error allowed in the utility of any state
    local variables: U, U', vectors of utilities for states in S, initially zero
                    d, the maximum relative change in the utility of any state
    repeat
        U <- U'; d <- 0
        for each state s in S do
            U'[s] <- max_{a E A(s)} Q-VALUE(mdp, s, a, U)
            if |U'[s] - U[s]| > d then d <- |U'[s] - U[s]|
    until d <= e(1-g)/g
    return U

```

Policy Iteration

Policy iteration alternates between policy evaluation and policy improvement to find the optimal policy. It leverages the insight that even with an inaccurate utility function estimate, a good policy can be found if one action is consistently better than others.

Policy Iteration

- **Policy Evaluation**: Given a policy π, calculate the utility of each state if π were executed. This step involves solving a set of linear equations.
- **Policy Improvement**: Update the policy by selecting actions that maximize the expected utility based on the current utility estimates.
- **Algorithm**: The algorithm iteratively evaluates and improves the policy until it converges to the optimal policy, ensuring that the utility function is a fixed point of the Bellman update.

```
function POLICY-ITERATION(mdp) returns a policy
    inputs: mdp, an MDP with states S, actions A(s), transition model P(s' |s,a)
    local variables: U, a vector of utilities for states in S, initially zero
                    p, a policy vector indexed by state, initially random
    repeat
        U <- POLICY-EVALUATION(p, U, mdp)
        unchanged? <- true
        for each state s in S do
            a* <- argmax_{a E A(s)} Q-VALUE(mdp, s, a, U)
            if Q-VALUE(mdp, s, a*, U) > Q-VALUE(mdp, s, p[s], U) then
                p[s] <- a*; unchanged? <- false
    until unchanged?
    return p

```

Linear Programming

Linear programming (LP) is a general approach for formulating and solving constrained optimization problems. In the context of MDPs, LP can be used to find the optimal utilities by expressing the Bellman equations as linear inequalities.

Linear Programming

- **Formulation**: The LP formulation involves minimizing the utilities $U(s)$ for all states s subject to constraints derived from the Bellman equations.
- **Optimization**: The resulting LP problem can be solved using standard LP solvers, which are well-studied and efficient for many applications.
- **Complexity**: Although LP provides a polynomial-time solution, it is often less efficient than dynamic programming methods for large MDPs due to the high dimensionality of the state space.

Online Algorithms for MDPs

Online algorithms for MDPs, such as Monte Carlo planning, perform computation at each decision point rather than precomputing an entire policy. These algorithms are useful for large MDPs where offline computation is infeasible.

Online Algorithms for MDPs

- **Monte Carlo Planning**: Uses random sampling to estimate the value of actions by simulating many possible future scenarios and averaging the results.
- **Expectimax Algorithm**: Builds a tree of alternating max and chance nodes to represent decision points and probabilistic outcomes, respectively.
- **Real-Time Dynamic Programming (RTDP)**: Focuses on the most relevant parts of the state space by updating values based on the agent's current trajectory and immediate needs.

Summary of Key Concepts

- **Value Iteration**: An iterative algorithm based on the Bellman equation, used to find optimal utilities and policies.
- **Policy Iteration**: Alternates between policy evaluation and policy improvement to find the optimal policy.
- **Linear Programming**: Formulates the Bellman equations as linear inequalities to solve for optimal utilities using LP solvers.
- **Online Algorithms**: Include Monte Carlo planning and expectimax algorithms, which perform computation at each decision point for large MDPs.

These algorithms provide a range of tools for solving MDPs, from exact offline methods to approximate online techniques, each suited to different types and sizes of problems.

The last section that is being covered from this chapter this week and from **Artificial Intelligence - A Modern Approach** is **Section 17.3: Bandit Problems**.

Section 17.3: Bandit Problems

Overview

This section delves into bandit problems, a class of sequential decision problems where an agent must choose among multiple options, each with unknown and potentially different reward distributions. These problems exemplify the exploration-exploitation tradeoff, a fundamental challenge in decision-making under uncertainty. The term "bandit problem" originates from the analogy to slot machines (one-armed bandits) in a casino, where each lever represents an arm of the bandit. The goal is to maximize the cumulative reward over time by intelligently balancing exploration (trying new options) and exploitation (leveraging known rewarding options).

Bandit Problems

Bandit problems serve as a simplified yet powerful model for various real-world scenarios where decisions must be made sequentially under uncertainty. The name "bandit problem" reflects the scenario where a gambler interacts with a slot machine, making a sequence of pulls (decisions) to maximize their total winnings.

Bandit Problems

- **Scenario**: The agent can pull one lever at a time, receiving a reward sampled from the lever's unknown distribution. The challenge is to balance exploration (trying new or less frequently used levers) with exploitation (using the lever known to give the highest reward so far).

- **Applications**: Bandit problems model various real-world situations, such as clinical trials (choosing the best treatment), investment decisions (allocating resources to different assets), advertisement placement (selecting ads to show users), and research funding (distributing funds among projects).
- **Historical Context**: During World War II, researchers struggled with bandit problems, highlighting their complexity and the counterintuitive nature of optimal strategies. Early attempts to solve these problems led to significant theoretical advancements and practical algorithms.
- **Fundamental Tradeoff**: The exploration-exploitation tradeoff is central to bandit problems. Exploration involves trying new or less certain options to gain information, while exploitation leverages known information to maximize rewards. Balancing these two aspects is critical for long-term success.

Formulating Bandit Problems

Bandit problems can be formally framed using Markov reward processes (MRPs), where each arm represents an MRP with states, a transition model, and a reward function. The overall problem is modeled as a Markov decision process (MDP).

Formulating Bandit Problems

- **Markov Reward Process (MRP)**: Each arm M_i is an MRP with states S_i, transition model $P_i(s'|s,a_i)$, and reward $R_i(s,a_i,s')$. An MRP models the dynamics of each arm, where actions lead to state transitions and generate rewards.
- **State Space**: The state space of the overall bandit problem is the Cartesian product $S = S_1 \times \cdots \times S_n$, representing all possible combinations of states for the arms.
- **Actions and Transitions**: Actions correspond to selecting an arm to pull, and the transition model updates the state of the chosen arm while leaving others unchanged. This model captures the independence of arms.
- **Discount Factor (γ)**: The discount factor is used to compute the present value of future rewards, balancing immediate and long-term gains. A higher discount factor (γ) places more emphasis on future rewards, encouraging long-term planning.

Example: Simple Deterministic Bandit Problem

Consider a simple bandit problem with two arms, each providing a sequence of rewards. This problem illustrates how switching between arms can yield higher cumulative rewards than sticking to a single arm.

Simple Deterministic Bandit Problem

- **Arms**: Arm M gives rewards [0, 2, 0, 7.2, 0, 0, ...] and arm M_1 gives a constant reward of 1. The sequence of rewards highlights the variability and potential high payouts of arm M compared to the steady rewards of arm M_1.
- **Utilities**: The utility of an arm is the total discounted reward it generates. For arm M:

  $$U(M) = 0 + 0.5 \times 2 + 0 + 0.5^3 \times 7.2 = 1.9$$

  For arm M_1:

  $$U(M_1) = \sum_{t=0}^{\infty} 0.5^t = 2.0$$

- **Optimal Strategy**: Switching from M to M_1 after the fourth pull yields a higher utility $U(S) = 2.025$ than sticking with M_1 or M. This strategy leverages the high reward from arm M before switching to the steady rewards of M_1.

Gittins Index

The Gittins index is a key concept for solving bandit problems. It provides a simple optimal policy: pull the arm with the highest Gittins index. This index balances the expected reward and the uncertainty associated with each arm.

Gittins Index

- **Definition**: The Gittins index for an arm M in state s is the maximum value obtained by balancing the immediate reward with future discounted rewards. It quantifies the potential of an arm in terms of expected utility per unit of discounted time.
- **Calculation**:

$$\lambda = \max_T \frac{E\left[\sum_{t=0}^{T-1} \gamma^t R_t\right]}{E\left[\sum_{t=0}^{T-1} \gamma^t\right]}$$

  This formula defines the Gittins index as the ratio of expected cumulative reward to expected discounted time over the best stopping time T.
- **Optimal Policy**: The optimal policy is to pull the arm with the highest Gittins index. This reduces the bandit problem to a series of decisions, each taking $O(n)$ time initially and $O(1)$ for subsequent decisions, as the indices of unselected arms remain unchanged.

Bernoulli Bandit

The Bernoulli bandit is a well-known variant where each arm produces a reward of 0 or 1 with a fixed but unknown probability. This variant illustrates the exploration-exploitation tradeoff clearly, with the agent needing to balance the uncertainty of rewards.

Bernoulli Bandit

- **States**: Defined by counts of successes s_i and failures f_i for each arm. These counts provide a statistical basis for estimating the probability of success for each arm.
- **Transition Model**: The next reward is 1 with probability $s_i/(s_i+f_i)$ and 0 with probability $f_i/(s_i+f_i)$. This model uses Bayesian updating to refine the estimates of success probabilities as more samples are collected.
- **Exploration Bonus**: Higher payoff probabilities are preferred, but arms tried fewer times have an exploration bonus to encourage sampling. This bonus accounts for the uncertainty in the reward estimates, promoting exploration of less certain arms.

Approximately Optimal Bandit Policies

In practice, exact calculation of the Gittins index can be complex, so approximate methods like the Upper Confidence Bound (UCB) and Thompson sampling are used. These methods provide efficient and practical solutions to bandit problems.

Approximately Optimal Bandit Policies

- **Upper Confidence Bound (UCB)**: Selects the arm with the highest upper confidence bound on its reward estimate. The UCB algorithm balances exploration and exploitation by considering both the mean reward and the uncertainty of the estimate.

$$\text{UCB}(M_i) = \hat{\mu}_i + \frac{g(N)}{\sqrt{N_i}}$$

  where $g(N)$ adjusts the exploration-exploitation balance. This formula captures the tradeoff by adding an exploration term to the estimated mean reward.
- **Thompson Sampling**: Selects arms based on the probability distribution of their being optimal, given the samples so far. It involves sampling from the posterior distributions of the arms and choosing the arm with the highest sample.
- **Regret Bounds**: Both UCB and Thompson sampling have regret bounds that grow logarithmically with the number of samples, making them efficient and practical. Regret measures the difference between the reward obtained by the algorithm and the reward that would have been obtained by an optimal policy.

Non-Indexable Variants

Not all problems fit neatly into the bandit framework. Selection problems, where the goal is to choose the best option as quickly as possible, differ from bandit problems in their mathematical properties and optimal strategies.

Non-Indexable Variants

- **Selection Problems**: Focus on choosing the best option quickly rather than maximizing cumulative reward. Examples include hiring employees or selecting suppliers. These problems often require strategies that prioritize rapid identification of the best option rather than balancing ongoing rewards.
- **Bandit Superprocess (BSP)**: A generalization where each arm is a full MDP. The optimal policy for BSPs may include actions that are locally suboptimal due to the interaction between arms. This generalization reflects more complex decision-making scenarios where each task or project has its own dynamics and reward structure.
- **Opportunity Cost**: Measures the utility lost by not attending to other arms, influencing the strategy for multitasking problems. Opportunity cost considerations lead to strategies that optimize the overall allocation of attention across multiple tasks.

Summary of Key Concepts

- **Bandit Problems**: Involve making sequential decisions to balance exploration and exploitation among multiple options with unknown reward distributions. These problems model real-world decision-making under uncertainty.
- **Formulating Bandit Problems**: Use MRPs and MDPs to model the state space, actions, and transitions of bandit problems. This formalization provides a structured approach to solving these problems.
- **Gittins Index**: Provides a simple and efficient optimal policy for bandit problems by selecting the arm with the highest index. The Gittins index balances immediate and future rewards effectively.
- **Bernoulli Bandit**: A variant where each arm yields binary rewards, highlighting the exploration-exploitation tradeoff. This model simplifies the analysis and solution of bandit problems.
- **Approximate Policies**: Methods like UCB and Thompson sampling offer practical solutions with logarithmic regret bounds. These methods are computationally efficient and robust to uncertainties in reward estimates.
- **Non-Indexable Variants**: Selection problems and BSPs require different approaches due to their unique properties and interactions. These variants reflect more complex and realistic decision-making scenarios.

Understanding bandit problems and their solutions is crucial for making informed decisions in uncertain environments, balancing the need to gather information with the pursuit of immediate rewards. These concepts and methods form the foundation for advanced decision-making algorithms used in various applications, from finance and healthcare to online advertising and project management.

The first chapter that is being covered from **Reinforcement Learning - An Introduction** is **Chapter 3: Finite Markov Decision Processes**.

Chapter 3: Finite Markov Decision Processes

Overview

This section introduces finite Markov Decision Processes (MDPs), a foundational concept in reinforcement learning. MDPs provide a formal framework for modeling decision-making where actions have long-term consequences. This framework includes states, actions, transition models, and rewards, and it is crucial for understanding how agents can learn to optimize their behavior over time.

The Agent-Environment Interface

In an MDP, the agent interacts with the environment over discrete time steps. At each time step, the agent observes the current state, selects an action, receives a reward, and transitions to a new state. This interaction is governed by the dynamics of the environment, which are defined by transition probabilities and reward functions.

The Agent-Environment Interface

- **Agent and Environment**: The agent makes decisions and the environment responds to these decisions. The goal of the agent is to maximize cumulative rewards.
- **States and Actions**: At each time step t, the agent observes a state S_t and selects an action A_t. The environment then transitions to a new state S_{t+1} and provides a reward R_{t+1}.
- **Trajectory**: The sequence of states, actions, and rewards forms a trajectory: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$.
- **Finite Sets**: In a finite MDP, the sets of states S, actions A, and rewards R are finite. The transition model $p(s', r|s, a)$ defines the probability of transitioning to state s' and receiving reward r given the current state s and action a.
- **Markov Property**: The probability of transitioning to a new state depends only on the current state and action, not on previous states or actions. This is known as the Markov property.

Goals and Rewards

The agent's objective is to maximize the cumulative reward it receives over time. Rewards are numerical values that the environment provides in response to the agent's actions. The reward signal formalizes the goal of the agent and guides its behavior.

Goals and Rewards

- **Reward Signal**: At each time step, the agent receives a reward R_t from the environment. The agent's goal is to maximize the expected cumulative reward.
- **Reward Hypothesis**: All goals can be expressed as the maximization of the expected value of the cumulative sum of rewards.
- **Examples**: Rewards can be designed to encourage desired behaviors. For instance, a robot can receive rewards for moving forward, avoiding obstacles, or completing tasks.
- **Designing Rewards**: It is crucial to design reward signals that accurately reflect the desired outcomes. Misaligned rewards can lead to unintended behaviors.

Returns and Episodes

The return is the cumulative reward the agent aims to maximize. Depending on the nature of the task, returns can be defined differently, such as over a finite horizon (episodic tasks) or an infinite horizon with discounting (continuing tasks).

Returns and Episodes

- **Return**: The return G_t is the total accumulated reward from time step t onwards. For episodic tasks, it is the sum of rewards until the end of an episode. For continuing tasks, it is often discounted to ensure finiteness.
- **Episodic Tasks**: Tasks that naturally break into episodes, each ending in a terminal state. The return is the sum of rewards within an episode.
- **Continuing Tasks**: Tasks with no natural end, modeled with discounting to keep the return finite. The return is a weighted sum of future rewards, $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, where $0 \leq \gamma < 1$ is the discount factor.
- **Discount Factor**: Determines the present value of future rewards. A higher discount factor makes the agent more farsighted.

Policies and Value Functions

A policy defines the agent's behavior, mapping states to actions. Value functions estimate the expected return from each state (or state-action pair) under a given policy.

Policies and Value Functions

- **Policy**: A policy $\pi(a|s)$ is a mapping from states to probabilities of selecting each action. It defines the agent's behavior.
- **State-Value Function**: The value function $v_\pi(s)$ is the expected return starting from state s and following policy π. It satisfies the Bellman equation:

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

- **Action-Value Function**: The action-value function $q_\pi(s,a)$ is the expected return starting from state s, taking action a, and following policy π. It also satisfies a Bellman equation:

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

- **Estimating Values**: Value functions can be estimated from experience by averaging observed returns or using more sophisticated methods like temporal-difference learning.

Optimal Policies and Value Functions

Optimal policies maximize the expected return from each state. The optimal state-value function $v_*(s)$ and the optimal action-value function $q_*(s,a)$ define the maximum expected returns.

Optimal Policies and Value Functions

- **Optimal Policy**: A policy π_* is optimal if it yields the highest expected return from each state compared to all other policies.
- **Optimal State-Value Function**: The optimal state-value function $v_*(s)$ is the maximum expected return achievable from state s:

$$v_*(s) = \max_\pi v_\pi(s)$$

- **Optimal Action-Value Function**: The optimal action-value function $q_*(s,a)$ is the maximum expected return achievable from state s and action a:

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

- **Bellman Optimality Equations**: These equations define the relationships for optimal value functions:

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\,[r + \gamma v_*(s')]$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a) \left[r + \gamma \max_{a'} q_*(s',a') \right]$$

- **Solving for Optimal Policies**: Optimal policies can be found by solving the Bellman optimality equations using dynamic programming methods like value iteration or policy iteration.

Summary of Key Concepts

Summary of Key Concepts

- **Finite MDPs**: Framework for sequential decision making with states, actions, transitions, and rewards.
- **Agent-Environment Interaction**: Continuous interaction where the agent selects actions based on states and receives rewards.

- **Goals and Rewards**: Agent aims to maximize cumulative rewards, defined through a reward signal.
- **Returns and Episodes**: Cumulative rewards can be defined over finite or infinite horizons, with discounting for continuing tasks.
- **Policies and Value Functions**: Policies map states to actions, and value functions estimate expected returns under a policy.
- **Optimal Policies and Value Functions**: Optimal policies maximize expected returns, defined by Bellman optimality equations.

Understanding finite MDPs and their components is fundamental to reinforcement learning. These concepts provide the foundation for developing algorithms that enable agents to learn optimal behaviors in uncertain environments.

The last chapter that is being covered from **Reinforcement Learning - An Introduction** is **Chapter 4: Dynamic Programming**.

Chapter 4: Dynamic Programming

Overview

This section delves into Dynamic Programming (DP) methods, which are a collection of algorithms used to compute optimal policies given a perfect model of the environment as a Markov Decision Process (MDP). Despite their computational expense and assumption of a perfect model, DP methods are fundamental in theoretical reinforcement learning and provide a foundation for understanding other methods. DP assumes a finite MDP, making its principles applicable even to problems with continuous state and action spaces through approximation techniques.

Policy Evaluation (Prediction)

Policy evaluation is the process of computing the state-value function v_π for a given policy π. This function gives the expected return starting from a state s and following policy π.

Policy Evaluation

- **State-Value Function**: For any state s, the state-value function under policy π is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

- **Iterative Policy Evaluation**: Iterative methods update estimates of v_π using the Bellman equation:

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

- **Convergence**: The sequence $\{v_k\}$ converges to v_π as $k \to \infty$, assuming either $\gamma < 1$ or eventual termination.

Policy Improvement

Policy improvement involves using the state-value function v_π to derive a better policy π' that is greedy with respect to v_π.

Policy Improvement

- **Improvement Step**: Given a policy π, a new policy π' is derived by:

$$\pi'(s) = \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a]$$

- **Policy Improvement Theorem**: If π' is derived from π using the above step, then π' is at least as good as π.

Policy Iteration

Policy iteration involves alternating between policy evaluation and policy improvement until convergence to an optimal policy.

Policy Iteration

- **Steps**:
  1. **Policy Evaluation**: Compute v_π for the current policy π.
  2. **Policy Improvement**: Derive a new policy π' that is greedy with respect to v_π.
- **Convergence**: This process guarantees convergence to an optimal policy π^* and optimal value function v^*.

Value Iteration

Value iteration combines policy evaluation and improvement into a single step, iteratively updating the value function until convergence.

Value Iteration

- **Update Rule**: The value function is updated using:

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

- **Convergence**: The sequence $\{v_k\}$ converges to v^*, the optimal value function.
- **Optimal Policy**: The optimal policy π^* is derived from v^* by selecting actions that maximize the expected return.

Asynchronous Dynamic Programming

Asynchronous DP methods update the value of states in any order, allowing flexibility and potentially faster convergence, especially in large state spaces.

Asynchronous Dynamic Programming

- **Flexibility**: States can be updated in any sequence, even stochastically.
- **Convergence**: As long as each state is updated infinitely often, the value function converges to v^*.
- **Efficiency**: Particularly useful for large state spaces where sweeping through all states is impractical.

Generalized Policy Iteration (GPI)

GPI describes the interaction between policy evaluation and policy improvement processes, irrespective of their granularity. Both processes work together to find an optimal policy and value function.

Generalized Policy Iteration

- **Interaction**: Policy evaluation and policy improvement processes interact, with each process providing feedback to the other.
- **Convergence**: The interaction typically converges to the optimal policy and value function.

Efficiency of Dynamic Programming

DP methods are efficient for solving MDPs with polynomial time complexity in the number of states and actions, making them exponentially faster than direct policy search methods.

Efficiency of Dynamic Programming

- **Computational Complexity**: DP methods are polynomial in time with respect to the number of states and actions.
- **Comparison**: DP methods are exponentially faster than direct policy search and more practical than linear programming for large MDPs.
- **Asynchronous Methods**: These methods are preferable for very large state spaces due to their flexible updating schemes.

Summary of Key Concepts

- **Dynamic Programming (DP)**: A set of algorithms for computing optimal policies in MDPs with a perfect model of the environment.
- **Policy Evaluation**: Computes the state-value function for a given policy.
- **Policy Improvement**: Derives a better policy from the state-value function.
- **Policy Iteration**: Alternates between policy evaluation and policy improvement until convergence.
- **Value Iteration**: Combines policy evaluation and improvement in a single iterative update.
- **Asynchronous DP**: Updates states in any order, useful for large state spaces.
- **Generalized Policy Iteration (GPI)**: Describes the interaction between evaluation and improvement processes leading to optimal policies.
- **Efficiency**: DP methods are polynomial in time and efficient for large MDPs, especially when using asynchronous methods.

Understanding these concepts is crucial for developing efficient algorithms in reinforcement learning, providing a foundation for solving MDPs and finding optimal policies.

Reinforcement Learning (RL)

Reinforcement Learning (RL)

7.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 22.1 - Learning From Rewards**
- **Artificial Intelligence - A Modern Approach - Chapter 22.2 - Passive Reinforcement Learning**
- **Artificial Intelligence - A Modern Approach - Chapter 22.3 - Active Reinforcement Learning**
- **Artificial Intelligence - A Modern Approach - Chapter 22.4 - Generalization In Reinforcement Learning**
- **Artificial Intelligence - A Modern Approach - Chapter 22.5 - Policy Search**
- **Reinforcement Learning - An Introduction - Chapter 5.1 - Monte Carlo Prediction**
- **Reinforcement Learning - An Introduction - Chapter 5.2 - Monte Carlo Estimation Of Action Values**
- **Reinforcement Learning - An Introduction - Chapter 5.3 - Monte Carlo Control**
- **Reinforcement Learning - An Introduction - Chapter 5.4 - Monte Carlo Control Without Exploring Starts**
- **Reinforcement Learning - An Introduction - Chapter 5.7 - Off-policy Monte Carlo Control**
- **Reinforcement Learning - An Introduction - Chapter 6.1 - TD Prediction**
- **Reinforcement Learning - An Introduction - Chapter 6.2 - Advantages Of TD Prediction Methods**
- **Reinforcement Learning - An Introduction - Chapter 6.4 - Sarsa - On-policy TD Control**
- **Reinforcement Learning - An Introduction - Chapter 6.5 - Q-learning - Off-policy TD Control**

7.0.2 Piazza

Must post at least **three** times this week to Piazza.

7.0.3 Lectures

The lectures for this week are:

- Reinforcement Learning - Intro, Model-Based, Passive RL $\approx$ 65 **min**.
- Reinforcement Learning - Active RL $\approx$ 59 **min**.
- Approximate Reinforcement Learning $\approx$ 73 **min**.

The lecture notes for this week are:

- Reinforcement Learning - Active RL Lecture Notes
- Reinforcement Learning - Intro, Model-Based, Passive RL Lecture Notes
- Approximate Reinforcement Learning Lecture Notes
- Approximate Reinforcement Learning Review Lecture Notes

7.0.4 Assignment

The assignment for this week is:

- Assignment 5 - MDP And RL

7.0.5 Quiz

The quiz for this week is:

- Quiz 6 - Reinforcement Learning
- Quiz 7 - Approximate Reinforcement Learning

7.0.6 Chapter Summary

The reading this week is from **Artificial Intelligence - A Modern Approach** and **Reinforcement Learning - An Introduction**. The chapter that is being covered from **Artificial Intelligence - A Modern Approach** is **Chapter 22: Reinforcement Learning**. The first section that is being covered from this chapter this week is **Section 22.1: Learning From Rewards**.

Section 22.1: Learning From Rewards

Overview

This section introduces reinforcement learning (RL), where agents learn to make decisions through rewards and punishments. Unlike supervised learning, which relies on labeled examples, RL enables agents to learn from their own experiences, optimizing actions based on outcomes. This approach is essential for problems where exhaustive training data is unavailable, allowing agents to generalize from limited feedback.

Reinforcement Learning Fundamentals

In RL, agents interact with the environment and receive rewards, guiding their learning process. The objective is to maximize the cumulative reward over time by balancing exploration and exploitation.

Reinforcement Learning Fundamentals

- **Interaction**: Agents interact with the environment by taking actions and observing the consequences, receiving rewards that indicate success or failure. This feedback loop allows agents to learn optimal behaviors.
- **Rewards**: In games like chess, rewards can be sparse (e.g., 1 for winning, 0.5 for a draw, and 0 for losing). Sparse rewards provide limited feedback, making it challenging to learn effective strategies quickly.
- **Exploration vs. Exploitation**: Agents must explore the environment to discover rewarding strategies while exploiting known strategies to maximize rewards. This tradeoff is central to RL and requires careful balancing.
- **Comparison with Supervised Learning**: Supervised learning uses labeled examples to train agents, but in many real-world scenarios, such data is unavailable. RL provides a way to learn from the environment without explicit examples.

Model-Based vs. Model-Free Learning

RL approaches are categorized into model-based and model-free methods, depending on the agent's knowledge of the environment's dynamics.

Model-Based vs. Model-Free Learning

- **Model-Based RL**: The agent uses a transition model to interpret rewards and make decisions. It may learn this model from interactions or rely on a pre-existing understanding of the environment's rules.

This approach facilitates state estimation and planning.

- **Model-Free RL**: The agent does not rely on a transition model, instead learning directly from experience. Two main approaches are:
  - **Action-Utility Learning (Q-Learning)**: Agents learn a quality function $Q(s, a)$ that estimates the expected rewards from taking action a in state s. The optimal policy is derived by selecting actions with the highest Q values.
  - **Policy Search**: Agents learn a policy $\pi(s)$ that maps states directly to actions, optimizing performance through experience. This approach is akin to reflex agents that respond directly to inputs.

Advantages of Reinforcement Learning

Reinforcement learning offers several advantages for developing intelligent agents, especially in environments where direct supervision is impractical.

Advantages of Reinforcement Learning

- **Scalability**: RL is well-suited for large and complex environments where supervised learning would require vast datasets. Agents can learn from simulations, gaining experience without the need for exhaustive labeled data.
- **Versatility**: RL applies to various domains, including robotics, gaming, and autonomous systems. Agents can learn diverse tasks from playing video games to controlling real-world robots.
- **Intermediate Rewards**: Providing intermediate rewards for progress (e.g., points in games) helps guide agents, facilitating faster learning. This strategy overcomes challenges associated with sparse rewards.
- **Integration with Deep Learning**: Combining RL with deep learning enables agents to process high-dimensional sensory inputs, expanding RL's applicability to more complex tasks like visual perception and language understanding.

Categories of Reinforcement Learning

RL can be divided into several categories based on how agents learn and optimize their behaviors.

Categories of Reinforcement Learning

- **Passive Reinforcement Learning**: Agents follow a fixed policy and learn the utility of states or state-action pairs. The goal is to evaluate the given policy, akin to policy evaluation in Markov Decision Processes (MDPs).
- **Active Reinforcement Learning**: Agents explore the environment and adapt their policies based on accumulated experiences. This approach requires addressing the exploration-exploitation tradeoff effectively.
- **Policy Search Methods**: Agents search for optimal policies using various optimization techniques. These methods can include gradient ascent, evolutionary algorithms, and more, tailored to find the best mapping from states to actions.
- **Apprenticeship Learning**: Agents learn from demonstrations, using observed behaviors to guide their learning. This method leverages expert demonstrations to accelerate the learning process.

Challenges in Reinforcement Learning

Despite its advantages, RL also presents challenges that need to be addressed for successful application in real-world scenarios.

Challenges in Reinforcement Learning

- **Sparse Rewards**: In many environments, rewards are infrequent, making it difficult for agents to learn effective strategies. Providing intermediate rewards or shaping reward functions can mitigate this issue.

- **Exploration**: Balancing exploration and exploitation is crucial. Too much exploration may lead to inefficiency, while excessive exploitation may prevent discovering optimal strategies.
- **Scalability**: As environments become more complex, the state and action spaces grow, posing challenges for RL algorithms in terms of computational efficiency and convergence.
- **Sample Efficiency**: Learning from limited interactions is essential, especially in real-world applications where simulations are not feasible. Sample-efficient algorithms are crucial for practical RL applications.

Summary of Key Concepts

- **Reinforcement Learning**: A learning paradigm where agents learn through interactions with the environment, optimizing actions based on received rewards.
- **Model-Based vs. Model-Free**: Approaches differ based on the agent's knowledge of the environment's dynamics, affecting how policies are learned.
- **Advantages**: RL is scalable, versatile, and integrates well with deep learning, making it suitable for various domains.
- **Challenges**: Sparse rewards, exploration, scalability, and sample efficiency are key challenges that need addressing for effective RL.
- **Categories**: Includes passive and active RL, policy search methods, and apprenticeship learning, each with unique characteristics and applications.

Reinforcement learning provides a robust framework for developing intelligent agents capable of learning from experience, making it an essential area of study in artificial intelligence.

The next section that is being covered from this chapter this week is **Section 22.2: Passive Reinforcement Learning**.

Section 22.2: Passive Reinforcement Learning

Overview

This section explores passive reinforcement learning, where the agent follows a fixed policy and learns the utilities of states. Unlike active learning, where agents explore actions to discover the best policies, passive learning focuses on evaluating an existing policy. This approach is essential in scenarios where the agent must operate within predetermined rules or constraints.

Passive Learning in Reinforcement Learning

In passive reinforcement learning, the agent does not control its actions but instead observes the rewards received from the environment while following a predefined policy. The goal is to learn the utility of each state under this policy.

Passive Learning in Reinforcement Learning

- **Fixed Policy**: The agent adheres to a fixed policy π, which maps states to actions without considering future consequences. The focus is on evaluating the policy rather than improving it.
- **Utility Estimation**: The agent aims to estimate the utility $U(s)$ of each state s under the given policy π. This estimation helps understand the long-term value of being in a state and following the policy thereafter.
- **Learning from Experience**: The agent learns from observed transitions and rewards, updating its estimates of state utilities as it gathers more data.

- **Application Scenarios**: Passive learning is suitable for environments where exploration is risky or undesirable, such as in medical treatment plans or automated trading systems where predefined strategies must be followed.

Policy Evaluation Methods

There are several methods to evaluate the utility of states under a fixed policy, each with its own advantages and limitations.

Policy Evaluation Methods

- **Direct Utility Estimation**: The agent calculates the average reward received from each state over time. This method requires numerous episodes to converge and may suffer from high variance, making it less practical in environments with limited interactions.
- **Temporal-Difference Learning (TD)**: A model-free approach where the agent updates its utility estimates based on the difference between successive state utilities. The update rule is:

$$U(s) \leftarrow U(s) + \alpha(r + \gamma U(s') - U(s))$$

  where r is the reward received, α is the learning rate, and γ is the discount factor.
- **Adaptive Dynamic Programming (ADP)**: A model-based approach that constructs a transition model and reward function based on observed data, using them to update utility estimates through dynamic programming techniques.
- **Monte Carlo Methods**: The agent learns from complete episodes, updating the utility estimates based on the total return from each state. This approach requires episodic tasks and may be computationally expensive due to the need to run complete episodes.

Convergence and Stability

The convergence and stability of utility estimates depend on the method used and the properties of the environment.

Convergence and Stability

- **Temporal-Difference Learning**: TD methods converge to the correct utility values under certain conditions, such as appropriate learning rates and exploration strategies.
- **Monte Carlo Methods**: These methods converge to the true utility values but require a large number of episodes and may suffer from high variance in returns.
- **Adaptive Dynamic Programming**: ADP approaches rely on accurate models, which can be challenging to construct in complex environments. Their convergence depends on the accuracy of the estimated transition models.
- **Stability**: Stability in learning is crucial, ensuring that utility estimates do not oscillate or diverge during the learning process.

Example: Gridworld

Gridworld is a classic example used to illustrate passive reinforcement learning, where an agent navigates a grid following a fixed policy, learning the utility of each state based on the rewards received.

Example: Gridworld

- **Environment**: A grid with states representing positions. Each position has associated rewards, and the agent moves according to a fixed policy.
- **Policy**: The agent follows a predefined policy, such as moving randomly or always heading towards a goal state.
- **Learning Process**: As the agent navigates the grid, it collects rewards and updates the utility estimates for each state based on the observed transitions and rewards.

- **Outcome**: Over time, the agent learns the expected utility of each state under the fixed policy, which can be used to evaluate the policy's effectiveness.

Challenges in Passive Reinforcement Learning

While passive learning simplifies the learning process by following a fixed policy, it still presents challenges that need to be addressed.

Challenges in Passive Reinforcement Learning

- **Exploration**: Since the policy is fixed, the agent's ability to explore the environment is limited, potentially leading to suboptimal utility estimates in unexplored states.
- **Convergence Speed**: Methods like direct utility estimation and Monte Carlo methods may converge slowly, especially in large state spaces with sparse rewards.
- **Model Dependence**: ADP methods rely on accurate models, and inaccuracies can lead to erroneous utility estimates.
- **Variance and Bias**: High variance in returns can affect the stability of learning, while bias in utility estimates can result from insufficient exploration or inaccurate models.

Summary of Key Concepts

- **Passive Reinforcement Learning**: Involves learning the utility of states under a fixed policy without exploring alternative actions.
- **Policy Evaluation**: Various methods exist, including direct utility estimation, temporal-difference learning, adaptive dynamic programming, and Monte Carlo methods.
- **Convergence and Stability**: These depend on the learning method and the environment, with challenges in exploration, convergence speed, and model accuracy.
- **Gridworld Example**: Illustrates passive learning where an agent learns state utilities by navigating a grid following a fixed policy.
- **Challenges**: Include limited exploration, slow convergence, model dependence, and issues with variance and bias in utility estimates.

Passive reinforcement learning provides a foundational understanding of policy evaluation, helping agents learn the value of states within predetermined constraints.

The next section that is being covered from this chapter this week is **Section 22.3: Active Reinforcement Learning**.

Section 22.3: Active Reinforcement Learning

Overview

This section explores active reinforcement learning, where agents have the freedom to select actions rather than follow a fixed policy. This approach allows agents to optimize their behavior by learning which actions yield the highest rewards. Active reinforcement learning emphasizes the balance between exploration and exploitation, which is crucial for maximizing cumulative rewards over time.

Active Learning in Reinforcement Learning

Active learning agents choose their actions based on learned models of the environment. Unlike passive agents, they seek to improve their knowledge and policies through interaction.

Active Learning in Reinforcement Learning

- **Freedom of Action**: Active agents select actions based on their current knowledge, which allows them to explore and learn the utility of different actions in various states.
- **Utility Estimation**: Agents learn utilities defined by the optimal policy, adhering to the Bellman equations:
$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')]$$
where $U(s)$ is the utility of state s, and $R(s,a,s')$ is the reward received when moving from state s to state s' via action a.
- **Exploration vs. Exploitation**: Active agents face the challenge of deciding whether to explore new actions or exploit known rewarding actions. This tradeoff is critical for learning optimal policies.
- **Model Learning**: Active agents need to learn a complete transition model for all actions, not just those specified by a fixed policy.

Exploration Strategies

The success of active reinforcement learning heavily depends on the exploration strategy employed, which determines how the agent balances learning and exploiting.

Exploration Strategies

- **Greedy Agents**: These agents choose actions that currently appear optimal, but they risk missing better strategies because they do not explore sufficiently. Greedy strategies often converge quickly but may not find the optimal policy.
- **GLIE (Greedy in the Limit of Infinite Exploration)**: This approach ensures that every action in each state is explored an unbounded number of times. It prevents the agent from becoming stuck in suboptimal policies by guaranteeing exploration.
- **Exploration Function**: This function $f(u,n)$ balances utility u and novelty n. It favors actions with high utility or low exploration count, effectively encouraging the agent to explore lesser-known state-action pairs:
$$U^+(s) \leftarrow \max_a f\left(\sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U^+(s')], N(s,a)\right)$$
- **Optimistic Initial Values**: Setting high initial utility estimates encourages exploration by making unexplored actions appear attractive, preventing premature convergence to suboptimal policies.

Safe Exploration

In real-world scenarios, exploration can lead to irreversible or costly states. Safe exploration strategies are necessary to prevent negative consequences.

Safe Exploration

- **Irreversible Actions**: Actions that lead to states from which recovery is impossible or costly must be avoided. For instance, a self-driving car must not enter states leading to severe accidents or mechanical failures.
- **Absorbing States**: These are states where no further actions have any effect, often with no reward. Avoiding such states is crucial, as entering them can halt learning and performance.
- **Approaches to Safe Exploration**:
  - **Bayesian Reinforcement Learning**: Incorporates prior knowledge about the environment and updates beliefs based on observations, using posterior probabilities to guide actions.
  - **Robust Control Theory**: Considers worst-case scenarios, optimizing for the best outcome under the least favorable conditions. This approach helps agents avoid risky actions.

  - **Teacher Guidance**: Human intervention or predefined constraints can ensure safety during learning. For example, an autonomous helicopter may have safety constraints overriding exploratory actions in risky states.

Temporal-Difference Q-Learning

Q-learning is a popular model-free algorithm used in active reinforcement learning, focusing on learning an action-value function directly without requiring a transition model.

Temporal-Difference Q-Learning

- **Q-Values**: The function $Q(s, a)$ estimates the expected total reward for taking action a in state s and following an optimal policy thereafter. The update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

- **Exploration Policy**: Uses an exploration function similar to that in ADP, balancing exploration and exploitation based on observed rewards and action counts.
- **SARSA**: A variant of Q-learning that updates Q-values based on the action actually taken, providing an on-policy learning method. The update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma Q(s',a') - Q(s,a)]$$

- **Comparison of SARSA and Q-Learning**:
  - **SARSA (On-Policy)**: Learns the value of the policy being followed, making it sensitive to the actions actually taken during exploration.
  - **Q-Learning (Off-Policy)**: Learns the value of the optimal policy, independent of the agent's actions, making it more robust to different exploration strategies.

Summary of Key Concepts

- **Active Reinforcement Learning**: Agents choose actions based on learned models, balancing exploration and exploitation to maximize rewards.
- **Exploration Strategies**: Include greedy approaches, GLIE, and optimistic initial values to ensure sufficient exploration of the state space.
- **Safe Exploration**: Strategies that prevent agents from entering irreversible or costly states, using Bayesian methods, robust control, and human guidance.
- **Temporal-Difference Q-Learning**: A model-free method that learns action-value functions without requiring a transition model, with variants like SARSA and Q-learning.
- **Exploration-Exploitation Tradeoff**: Central to active reinforcement learning, this tradeoff determines the balance between exploring new actions and exploiting known rewards.

Active reinforcement learning provides a framework for agents to learn optimal behaviors in dynamic environments, where balancing exploration and exploitation is key to maximizing long-term rewards.

The next section that is being covered from this chapter this week is **Section 22.4: Generalization In Reinforcement Learning**.

Section 22.4: Generalization In Reinforcement Learning

Overview

This section discusses generalization in reinforcement learning (RL), focusing on how agents can learn from limited experiences and apply that knowledge to unvisited states. In real-world environments with vast state spaces, it is impractical to explore all states exhaustively. Generalization allows agents to make informed decisions by approximating the utility of unvisited states based on features and past experiences.

Function Approximation

Function approximation is used to estimate utility functions or Q-functions compactly, making it feasible to handle large state spaces.

Function Approximation

- **Concept**: Instead of maintaining a table of utility values for each state, agents use a parameterized function to approximate utilities:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

  where $f_i(s)$ are features of state s and θ_i are the parameters.

- **Example**: In a 4x3 grid, utilities might be approximated using the coordinates (x, y):

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

  This reduces the number of parameters significantly, allowing generalization across states.

- **Learning Process**: Parameters θ are adjusted using data from state visits, minimizing the difference between predicted and actual rewards. This allows the model to generalize from visited states to unvisited ones.

- **Importance**: Enables agents to perform well in large state spaces, where explicit state enumeration is infeasible, by capturing the underlying structure of the environment.

Direct Utility Estimation with Function Approximation

Direct utility estimation can be enhanced using function approximation, allowing agents to estimate utilities more effectively.

Direct Utility Estimation with Function Approximation

- **Process**: Agents run trials, gathering rewards and state features. Each trial provides training examples, which are used to update the parameterized utility function.

- **Example**: Using a linear function $\hat{U}_\theta(x, y)$ with parameters updated through gradient descent:

$$\theta_i \leftarrow \theta_i + \alpha[u_j(s) - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

  where $u_j(s)$ is the observed total reward from state s.

- **Convergence**: With a proper learning rate α, the parameters converge to values that minimize prediction error, providing a good approximation of true utilities.

- **Challenges**: Care must be taken to avoid high variance in returns and ensure a suitable feature set that accurately represents the environment.

Temporal-Difference Learning with Function Approximation

Temporal-difference (TD) learning can also benefit from function approximation, allowing agents to learn from incremental updates rather than full episodes.

Temporal-Difference Learning with Function Approximation

- **Update Rule**: Parameters are updated based on the temporal difference error:

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s') + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

  where $R(s,a,s')$ is the reward received, and γ is the discount factor.

- **Q-Learning with Function Approximation**:

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s') + \gamma \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)] \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

- **Convergence**: For linear function approximators, convergence is guaranteed under certain conditions. Nonlinear approximators, like neural networks, may present challenges such as instability and catastrophic forgetting.

- **Catastrophic Forgetting**: When new experiences overwrite previously learned information, causing significant degradation in performance. Techniques like experience replay help mitigate this issue by reusing past experiences.

Deep Reinforcement Learning

Deep reinforcement learning uses deep neural networks as function approximators, allowing agents to learn directly from high-dimensional inputs like images.

Deep Reinforcement Learning

- **Concept**: Deep networks approximate utility or Q-functions using many layers of non-linear transformations, automatically extracting features from raw input data.

- **Application**: Successfully applied in various domains, such as playing video games at expert levels, mastering the game of Go, and training robots for complex tasks.

- **Advantages**:
  - **Feature Learning**: Automatically learns relevant features without manual feature engineering.
  - **Scalability**: Capable of handling complex environments with high-dimensional state spaces.

- **Challenges**: Deep RL systems can be unstable, sensitive to hyperparameters, and may not generalize well to environments differing from the training data.

Reward Shaping

Reward shaping involves modifying the reward function to guide learning, addressing the sparse reward problem in complex environments.

Reward Shaping

- **Concept**: Introduces pseudorewards to accelerate learning, providing additional feedback for intermediate steps toward the goal.

- **Example**: In a soccer robot, pseudorewards may be given for ball contact or progressing towards the goal, encouraging beneficial behaviors.

- **Potential Function**: Adjusts the reward function without altering the optimal policy:

$$R'(s,a,s') = R(s,a,s') + \gamma\Phi(s') - \Phi(s)$$

  where $\Phi(s)$ reflects desirable aspects of the state, such as proximity to subgoals.

- **Risks**: Care must be taken to ensure agents do not over-optimize pseudorewards at the expense of true objectives, potentially developing suboptimal strategies.

Hierarchical Reinforcement Learning

Hierarchical reinforcement learning (HRL) breaks down complex tasks into simpler subtasks, facilitating learning in environments with long action sequences.

Hierarchical Reinforcement Learning

- **Concept**: Decomposes tasks into a hierarchy of subtasks, each with its own policies, facilitating more manageable learning.
- **Example**: In a simplified soccer game, tasks may include passing, dribbling, and shooting, each with its sub-level decisions.
- **Partial Programs**: Use structured frameworks, guiding agents while leaving specifics for the agent to learn.
- **Benefits**: Speeds up learning by focusing on subtasks, leveraging previously learned skills, and improving overall policy performance.
- **Joint State Space**: Incorporates both physical and internal states, allowing for detailed behavior modulation based on the agent's context.

Summary of Key Concepts

- **Function Approximation**: Enables agents to generalize from limited experiences, approximating utility and Q-functions using parameterized models.
- **Direct Utility Estimation and TD Learning**: Enhanced through function approximation, allowing for effective utility estimation in large state spaces.
- **Deep Reinforcement Learning**: Utilizes deep networks for automatic feature extraction and complex state-space representation.
- **Reward Shaping**: Introduces additional rewards to guide learning, but must be carefully managed to avoid suboptimal behavior.
- **Hierarchical Reinforcement Learning**: Breaks tasks into subtasks, improving learning efficiency and policy performance in complex environments.

Generalization in reinforcement learning enables agents to efficiently learn and apply knowledge in vast state spaces, paving the way for sophisticated applications in dynamic and complex environments.

The last section that is being covered from this chapter this week is **Section 22.5: Policy Search**.

Section 22.5: Policy Search

Overview

This section discusses policy search, a method in reinforcement learning where the goal is to find an optimal policy by adjusting parameters to maximize performance. Unlike value-based methods that estimate value functions, policy search focuses directly on finding the best policy representation. This approach is useful when value functions are difficult to estimate or when policies are naturally parameterized.

Policy Representation

Policies in reinforcement learning map states to actions. In policy search, these policies are often represented using parameterized functions, significantly reducing the number of parameters compared to explicit state-action mappings.

Policy Representation

- **Parameterized Policies**: Represent policies with a set of parameters θ, such as:

$$\pi_\theta(s) = \arg\max_a \hat{Q}_\theta(s,a)$$

where $\hat{Q}_\theta(s,a)$ is a parameterized Q-function. This could be a linear function or a nonlinear model like a neural network.

- **Stochastic Policies**: Use probabilistic representations such as the softmax function:

$$\pi_\theta(s,a) = \frac{e^{\beta \hat{Q}_\theta(s,a)}}{\sum_{a'} e^{\beta \hat{Q}_\theta(s,a')}}$$

where β controls the randomness. A higher β leads to deterministic choices, while a lower β results in more exploration.

- **Continuous Differentiability**: Stochastic policies ensure that the policy's value changes smoothly with the parameters, facilitating gradient-based optimization.

Policy Search Methods

Policy search methods optimize policy parameters to improve the expected reward. These methods include gradient-based approaches and evolutionary algorithms.

Policy Search Methods

- **Policy Gradient Methods**: Directly optimize the expected return by following the gradient $\nabla_\theta \rho(\theta)$:

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} u_j(s) \frac{\nabla_\theta \pi_\theta(s,a_j)}{\pi_\theta(s,a_j)}$$

where $u_j(s)$ is the total reward from state s onward.

- **REINFORCE Algorithm**: An on-policy method that estimates the gradient from sampled episodes, effectively updating policy parameters based on observed returns.
- **Hill Climbing**: Evaluates changes in policy value for small parameter increments. This method can be slow and unreliable, particularly in noisy environments.
- **Evolutionary Algorithms**: Use population-based search, mutating and selecting policies based on performance. These methods are robust but computationally intensive.

Challenges in Policy Search

Policy search faces unique challenges, including high variance in reward estimation and sensitivity to initial parameter settings.

Challenges in Policy Search

- **Variance in Reward Estimation**: Estimating the expected return can have high variance, especially in stochastic environments. This affects the reliability of gradient estimates.
- **Discontinuous Policies**: When actions are discrete, small changes in parameters can lead to abrupt policy changes, complicating gradient-based optimization.
- **Sample Inefficiency**: Policy search often requires many samples to accurately estimate gradients, making it computationally expensive.
- **Local Optima**: The optimization landscape may contain local optima, where simple hill climbing might get stuck without finding the global best policy.

Techniques for Effective Policy Search

Several techniques enhance the effectiveness of policy search by improving sample efficiency and stability.

Techniques for Effective Policy Search

- **Baseline Subtraction**: Reduces variance by subtracting a baseline from the return, which does not change the gradient's expectation but lowers its variance.
- **Correlated Sampling**: Compares policies using the same sequence of environmental states, reducing variance in performance comparisons. This method is particularly useful in stochastic environments.
- **Actor-Critic Methods**: Combine value function approximation with policy search, where the actor updates policy parameters and the critic evaluates the value function. This structure provides stable updates and reduces variance.
- **Trust Region Policy Optimization (TRPO)**: Ensures updates do not deviate too much from the current policy, maintaining stability by constraining the size of policy updates.

Applications of Policy Search

Policy search methods are applicable in various domains where direct policy representation is advantageous, such as robotics and autonomous systems.

Applications of Policy Search

- **Robotics**: Policy search enables robots to learn complex tasks with continuous action spaces, leveraging parameterized policies for efficient exploration and learning.
- **Autonomous Vehicles**: Vehicles use policy search to navigate and make real-time decisions in dynamic environments, optimizing safety and efficiency.
- **Game Playing**: In games with large action spaces or continuous controls, policy search provides effective strategies, leveraging deep neural networks for high-dimensional inputs.

Summary of Key Concepts

- **Policy Search**: Focuses on optimizing policy parameters directly, suitable for environments where value functions are hard to estimate.
- **Policy Representation**: Utilizes parameterized or stochastic policies, facilitating smooth optimization through gradient-based methods.
- **Policy Search Methods**: Includes policy gradient, hill climbing, and evolutionary algorithms, each with strengths and challenges.
- **Challenges**: High variance, sample inefficiency, and local optima are key challenges in policy search.
- **Techniques for Improvement**: Baseline subtraction, correlated sampling, and actor-critic methods enhance policy search effectiveness.
- **Applications**: Policy search is widely used in robotics, autonomous systems, and game playing, leveraging direct policy optimization for complex tasks.

Policy search provides a robust framework for optimizing agent behavior in complex environments, offering flexibility and effectiveness in scenarios where traditional value-based methods may struggle.

The first chapter that is being covered from **Reinforcement Learning - An Introduction** is **Chapter 5: Monte Carlo Methods**. The first section that is being covered from this chapter this week is **Section 5.1: Monte Carlo Prediction**.

Section 5.1: Monte Carlo Prediction

Overview

This section introduces Monte Carlo methods for predicting state values in reinforcement learning. Monte Carlo methods rely on sampling episodes to estimate the expected return of states under a given policy. Unlike Dynamic Programming (DP), Monte Carlo does not require a model of the environment's dynamics, making it suitable for problems where the environment model is unknown or complex.

Monte Carlo Prediction

Monte Carlo prediction estimates the state-value function $v_\pi(s)$ for a given policy π by averaging the returns obtained from multiple episodes.

Monte Carlo Prediction

- **State-Value Function**: The value of a state s under policy π is the expected cumulative future discounted reward starting from s.
- **First-Visit MC Method**: Estimates $v_\pi(s)$ as the average of returns following the first visit to s in each episode.
- **Every-Visit MC Method**: Averages the returns following all visits to s across episodes. It converges similarly to first-visit MC but may be less biased.
- **Convergence**: Both methods converge to the true value $v_\pi(s)$ as the number of visits increases, leveraging the law of large numbers.

Example: Blackjack

The card game of blackjack illustrates the application of Monte Carlo methods in an episodic task where the player's objective is to maximize the sum of card values without exceeding 21.

Example: Blackjack

- **Game Setup**: The player competes against a dealer, and the state depends on the player's cards, the dealer's visible card, and whether the player holds a usable ace.
- **Policy**: A simple policy might involve sticking on a sum of 20 or 21 and hitting otherwise.
- **Monte Carlo Simulation**: By simulating many games and following the policy, the agent can estimate the state-value function by averaging the returns from each state.
- **Results**: After many episodes, the estimated state-value function approximates the true values, guiding the player on the expected outcomes of different states.

Advantages of Monte Carlo Methods

Monte Carlo methods offer several advantages, particularly when dealing with large or complex environments.

Advantages of Monte Carlo Methods

- **Model-Free Learning**: Unlike DP, Monte Carlo does not require knowledge of the environment's transition probabilities, making it applicable in environments where the model is unknown.
- **Independence of States**: Estimates for each state are independent, allowing targeted evaluation of specific states without needing a complete model of the environment.
- **Sample Efficiency**: Monte Carlo methods can efficiently learn from actual or simulated experience, useful when exact models are unavailable or difficult to derive.

Backup Diagrams in Monte Carlo

Backup diagrams illustrate the difference between Monte Carlo and DP methods, focusing on how states are updated based on episodes.

Backup Diagrams in Monte Carlo

- **Monte Carlo Diagrams**: Show the entire trajectory of transitions in an episode, emphasizing the cumulative reward up to the terminal state.
- **Comparison with DP**: Unlike DP, which updates values based on one-step transitions, Monte Carlo updates use complete episode returns, reflecting sample-based learning.
- **Independence from Bootstrapping**: Monte Carlo methods do not bootstrap; each state's value is updated independently based on the observed returns from episodes.

Applications and Limitations

Monte Carlo methods have practical applications but also face limitations in specific scenarios.

Applications and Limitations

- **Applications**: Effective in tasks like game playing, where episodic interactions provide clear feedback and the model is not readily available.
- **Limitations**:
  - **Long Episodes**: Monte Carlo methods can be computationally expensive if episodes are lengthy, requiring many iterations to converge.
  - **Exploration Requirements**: Effective exploration is necessary to ensure all states are visited; otherwise, the estimates may be biased or incomplete.

Summary of Key Concepts

- **Monte Carlo Prediction**: Uses sampling to estimate state values, relying on episodic returns.
- **First-Visit and Every-Visit Methods**: Two approaches to calculating state-value estimates, differing in how visits are considered.
- **Model-Free Learning**: Monte Carlo methods do not require a model of the environment, making them versatile in complex domains.
- **Backup Diagrams**: Visualize the process of updating state values based on entire episodes rather than single transitions.
- **Advantages and Limitations**: Effective for model-free learning but can be resource-intensive with long episodes and require good exploration strategies.

Monte Carlo methods provide a powerful approach for learning state values in reinforcement learning, especially in environments where model knowledge is limited or unavailable.

The next section that is being covered from this chapter this week is **Section 5.2: Monte Carlo Estimation Of Action Values**.

Section 5.2: Monte Carlo Estimation Of Action Values

Overview

This section focuses on the estimation of action values using Monte Carlo methods in reinforcement learning. When a model of the environment is unavailable, estimating the values of state-action pairs directly becomes essential. Monte Carlo methods offer a way to estimate these action values by sampling episodes and calculating returns, allowing agents to learn optimal policies through experience.

Monte Carlo Estimation of Action Values

Monte Carlo methods estimate the expected return of state-action pairs $q_\pi(s,a)$, which is the expected return starting from state s, taking action a, and following policy π thereafter.

Monte Carlo Estimation of Action Values

- **Action-Value Function**: $q_\pi(s,a)$ represents the expected return when starting in state s, taking action a, and then following policy π.
- **First-Visit MC Method**: Estimates $q_\pi(s,a)$ by averaging the returns following the first visit to the state-action pair (s,a) in each episode.
- **Every-Visit MC Method**: Averages the returns following all visits to the state-action pair (s,a) across episodes, similar to the state-value estimation.
- **Convergence**: Both methods converge to the true values as the number of visits to each state-action pair approaches infinity, leveraging the law of large numbers.

Exploring Starts

To ensure that all state-action pairs are explored adequately, the assumption of exploring starts is introduced, which ensures sufficient exploration of the action space.

Exploring Starts

- **Concept**: Each episode begins with a random state-action pair, ensuring that all pairs have a non-zero probability of being selected.
- **Purpose**: Guarantees that all state-action pairs will be visited infinitely often, facilitating comprehensive learning of action values.
- **Limitations**: While useful in theoretical analysis, exploring starts may not always be practical or possible in real-world scenarios where initial states are constrained.
- **Alternative**: Policies with stochastic components that ensure all actions in each state have non-zero probability, fostering exploration without relying solely on exploring starts.

Challenges in Monte Carlo Estimation of Action Values

While Monte Carlo methods provide a powerful framework for estimating action values, they present specific challenges that need addressing.

Challenges in Monte Carlo Estimation of Action Values

- **Lack of Visits**: Deterministic policies may lead to certain state-action pairs never being visited, resulting in incomplete estimates.
- **Exploration vs. Exploitation**: Balancing exploration of new actions and exploitation of known rewarding actions is crucial for learning effective policies.
- **Dependence on Initial Policy**: The initial policy can heavily influence the learning process, particularly in deterministic setups, potentially biasing the learning outcome.

Comparison with State-Value Estimation

Monte Carlo estimation of action values extends the concepts from state-value estimation, adapting them to focus on state-action pairs.

Comparison with State-Value Estimation

- **Focus on State-Action Pairs**: Unlike state-value methods that estimate $v_\pi(s)$, action-value methods estimate $q_\pi(s,a)$, providing a more granular view of the environment.
- **Policy Dependency**: State-action values are directly tied to the policy being followed, necessitating a comprehensive exploration strategy.

- **Practical Implications**: Provides the basis for deriving policies by selecting actions with the highest estimated values, guiding decision-making in the absence of a model.

Summary of Key Concepts

- **Monte Carlo Estimation of Action Values**: Utilizes sampling to estimate the expected return for state-action pairs under a given policy.
- **Exploring Starts**: Ensures all state-action pairs are visited, promoting comprehensive learning of the action-value function.
- **First-Visit and Every-Visit Methods**: Two approaches to estimating action values, converging to true values with sufficient visits.
- **Challenges**: Include lack of visits, exploration-exploitation balance, and initial policy dependence, affecting the effectiveness of learning.
- **Comparison with State-Value Estimation**: Highlights the focus on state-action pairs and the practical implications for policy derivation.

Monte Carlo methods for action-value estimation provide a powerful tool for learning policies in model-free environments, supporting effective decision-making through direct experience.

The next section that is being covered from this chapter this week is **Section 5.3: Monte Carlo Control**.

Section 5.3: Monte Carlo Control

Overview

This section covers Monte Carlo control, focusing on approximating optimal policies by using Monte Carlo methods. The approach combines policy evaluation and policy improvement to iteratively refine policies based on sampled episodes. This method is model-free, relying solely on experience rather than explicit knowledge of the environment's dynamics.

Monte Carlo Control

Monte Carlo control methods aim to find optimal policies by estimating action-value functions through episodic sampling and iteratively improving policies.

Monte Carlo Control

- **Generalized Policy Iteration (GPI)**: Combines policy evaluation and improvement:

$$\pi \to q_\pi \to \pi'$$

  where q_π is the estimated action-value function for policy π, and π' is a policy improvement based on q_π.

- **Policy Evaluation**: Uses Monte Carlo methods to estimate the action-value function by averaging the returns of episodes starting from each state-action pair.
- **Policy Improvement**: Constructs a new greedy policy π' by selecting actions that maximize the estimated action values:

$$\pi'(s) = \arg\max_a q_\pi(s,a)$$

- **Exploring Starts**: Assumes that all state-action pairs are initially explored, ensuring that all pairs have a non-zero probability of being selected.

Monte Carlo ES (Exploring Starts)

Monte Carlo ES is an algorithm designed to find optimal policies using the concept of exploring starts, ensuring comprehensive exploration of the state-action space.

Monte Carlo ES (Exploring Starts)

- **Initialization**:
  - $\pi(s) \in A(s)$ arbitrarily for all states s.
  - $Q(s,a)$ initialized randomly for all state-action pairs (s,a).
  - Returns lists for state-action pairs initialized as empty.
- **Episode Generation**:
  - Selects a random starting state-action pair, ensuring all pairs have a non-zero probability.
  - Follows policy π to generate an episode, recording the sequence of states, actions, and rewards.
- **Policy Evaluation and Improvement**:
  - For each state-action pair (S_t, A_t) in the episode, updates the estimated return G by summing rewards.
  - Updates the action-value function $Q(S_t, A_t)$ by averaging returns and updates the policy to be greedy with respect to Q.
- **Convergence**: The algorithm converges to the optimal policy and value function, assuming infinite episodes and exploring starts.

Example: Blackjack

The Monte Carlo ES method is applied to the game of blackjack, illustrating its effectiveness in finding optimal policies.

Example: Blackjack

- **Game Setup**: Involves a player against a dealer, where the player's goal is to maximize the sum of card values without exceeding 21.
- **Policy Initialization**: The initial policy may stick on sums of 20 or 21, with the action-value function initialized to zero.
- **Exploration**: Exploring starts ensure that all possible combinations of the player's sum, dealer's cards, and usable ace are covered.
- **Results**: Monte Carlo ES finds an optimal policy that closely matches the known basic strategy for blackjack, with minor differences potentially due to specific game rules.

Challenges and Limitations

Monte Carlo control, while powerful, has certain challenges and limitations, particularly related to assumptions and practical applicability.

Challenges and Limitations

- **Exploring Starts Assumption**: Assumes all state-action pairs are explored, which may not be feasible in real-world scenarios where some states are inaccessible or actions are restricted.
- **Infinite Episodes**: Convergence theoretically requires an infinite number of episodes, which is impractical. Approaches like approximate policy evaluation can mitigate this.
- **Sample Inefficiency**: The method may require many episodes to converge, especially in environments with a vast state-action space.
- **Alternative Strategies**: On-policy and off-policy methods can address the assumption of exploring starts, using ϵ-greedy or soft policies to ensure sufficient exploration.

Summary of Key Concepts

- **Monte Carlo Control**: Focuses on optimizing policies through episodic sampling and policy iteration without needing a model of the environment.
- **Generalized Policy Iteration (GPI)**: Iteratively improves policies by alternating between policy evaluation and improvement.
- **Monte Carlo ES**: A specific algorithm using exploring starts to ensure all state-action pairs are explored, facilitating learning of optimal policies.
- **Challenges**: Include assumptions of exploring starts, the need for infinite episodes, and sample inefficiency in large state-action spaces.
- **Example: Blackjack**: Demonstrates the application of Monte Carlo ES to find optimal strategies in a card game environment.

Monte Carlo control provides a robust framework for learning optimal policies in model-free environments, though practical implementations may require adjustments to address exploration and efficiency challenges.

The next section that is being covered from this chapter this week is **Section 5.4: Monte Carlo Control Without Exploring Starts**.

Section 5.4: Monte Carlo Control Without Exploring Starts

Overview

This section introduces Monte Carlo control methods that do not rely on the unrealistic assumption of exploring starts. Instead, these methods ensure that all actions are selected sufficiently often through soft policies. This approach allows for on-policy control without requiring initial visits to all state-action pairs, making it more applicable in real-world scenarios.

On-Policy Control with ϵ-Greedy Policies

On-policy methods in Monte Carlo control aim to improve the policy that is being used to generate data. These methods use ϵ-greedy policies to balance exploration and exploitation.

On-Policy Control with ϵ-Greedy Policies

- **ϵ-Greedy Policies**: A soft policy where:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|}, & \text{if } a = \arg\max_a q(s,a) \\ \frac{\epsilon}{|A(s)|}, & \text{otherwise} \end{cases}$$

- **Exploration and Exploitation**: Most of the time, the agent selects actions with the highest estimated value, but with probability ϵ, it selects a random action to ensure exploration.
- **Policy Improvement**: The ϵ-greedy policy is gradually shifted towards a deterministic optimal policy by reducing ϵ over time.

On-Policy First-Visit Monte Carlo Control

This algorithm iteratively estimates the action-value function and improves the policy based on ϵ-greedy selection.

On-Policy First-Visit Monte Carlo Control

- **Initialization**:

- Initialize π as an arbitrary ϵ-soft policy.
- Initialize $Q(s, a)$ arbitrarily for all state-action pairs (s, a).
- Create empty lists for returns of each state-action pair.

- **Episode Generation**:
  - Generate episodes using the current policy π.
  - Record the sequence of states, actions, and rewards.
- **Policy Evaluation**:
  - For each state-action pair (S_t, A_t) in the episode, compute the return G from that point.
  - Update $Q(S_t, A_t)$ as the average of the returns.
- **Policy Improvement**:
  - Update the policy π to be ϵ-greedy with respect to the current action-value estimates $Q(s, a)$.
- **Convergence**: The algorithm converges to the optimal policy and action-value function, assuming all actions are sufficiently explored.

Advantages of ϵ-Greedy Policies

Using ϵ-greedy policies addresses some of the limitations of exploring starts, making Monte Carlo control more practical.

Advantages of ϵ-Greedy Policies

- **Avoids Exploring Starts**: Removes the need for the unrealistic assumption that all state-action pairs are initially explored.
- **Ensures Exploration**: ϵ-greedy policies guarantee that all actions have a non-zero probability of being selected, ensuring comprehensive exploration over time.
- **Gradual Policy Improvement**: The policy iteratively approaches optimality while maintaining exploration, facilitating learning in complex environments.

Comparison with Exploring Starts

The transition from exploring starts to ϵ-greedy policies marks a significant shift in Monte Carlo control methods, enhancing their applicability.

Comparison with Exploring Starts

- **Exploring Starts**: Assumes all state-action pairs are visited initially, which is often impractical.
- **ϵ-Greedy Policies**: Enable gradual exploration and learning without needing to visit all state-action pairs at the beginning.
- **Real-World Applicability**: ϵ-greedy policies are more feasible in practical scenarios, especially where some states or actions are initially inaccessible.

Summary of Key Concepts

- **Monte Carlo Control Without Exploring Starts**: Uses ϵ-greedy policies for on-policy control, eliminating the need for initial state-action exploration.
- **On-Policy Control**: Iteratively improves the policy used to generate episodes, balancing exploration and exploitation.
- **ϵ-Greedy Policies**: Soft policies that ensure all actions are selected with non-zero probability, promoting sufficient exploration.
- **Advantages Over Exploring Starts**: More practical and applicable in real-world environments where initial exploration of all state-action pairs is unrealistic.

Monte Carlo control without exploring starts provides a robust framework for learning optimal policies in complex environments, making it a valuable approach in reinforcement learning.

The last section that is being covered from this chapter this week is **Section 5.7: Off-policy Monte Carlo Control**.

Section 5.7: Off-policy Monte Carlo Control

Overview

This section covers off-policy Monte Carlo control, where the policy used to generate behavior (behavior policy) differs from the policy being evaluated and improved (target policy). Off-policy methods allow learning about one policy while following another, providing flexibility and enabling the use of exploratory behavior policies while still learning optimal target policies.

Off-policy Control Basics

Off-policy control involves learning optimal policies by using a behavior policy that explores the state-action space, while the target policy is optimized for performance.

Off-policy Control Basics

- **Behavior Policy (b)**: Used to generate episodes, ensuring exploration by being ϵ-soft, where all actions have non-zero probability.
- **Target Policy (π)**: The policy being evaluated and improved, often greedy with respect to the current action-value estimates.
- **Separation of Policies**: Allows the target policy to be deterministic (e.g., greedy), while the behavior policy can explore different actions.
- **Importance Sampling**: Used to weigh returns from the behavior policy to estimate the value of the target policy accurately.

Importance Sampling

Importance sampling is crucial in off-policy methods, adjusting the returns from the behavior policy to match the target policy.

Importance Sampling

- **Importance Sampling Ratio**: Adjusts the return G_t by the product of the ratios of the probabilities of the actions under the target and behavior policies:

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

- **Weighted Returns**: The action-value function is updated using weighted returns:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + W_t(G_t - Q(S_t, A_t))$$

- **Ensuring Coverage**: The behavior policy must ensure that all actions have non-zero probability, guaranteeing sufficient exploration of the action space.

Off-policy Monte Carlo Control Algorithm

The algorithm combines Monte Carlo methods with off-policy control using weighted importance sampling to evaluate and improve the target policy.

Off-policy Monte Carlo Control Algorithm

- **Initialization**:
  - Initialize $Q(s,a)$ arbitrarily for all state-action pairs (s,a).
  - Initialize $C(s,a)=0$ for all state-action pairs to store cumulative weights.
- **Episode Generation**:
  - Generate episodes using the behavior policy b, ensuring all actions have non-zero probability.
- **Updating Process**:
  - For each episode, initialize $G=0$ and $W=1$.
  - Loop backward through the episode:
    - Update G with the reward:
    $$G \leftarrow G + R_{t+1}$$
    - Update cumulative weights:
    $$C(S_t,A_t) \leftarrow C(S_t,A_t) + W$$
    - Update action-value estimates:
    $$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{W}{C(S_t,A_t)}(G - Q(S_t,A_t))$$
    - Update the policy:
    $$\pi(S_t) \leftarrow \arg\max_a Q(S_t,a)$$
    - If A_t is not equal to $\pi(S_t)$, exit the loop.
    - Update importance sampling ratio:
    $$W \leftarrow W\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$
- **Convergence**: The algorithm converges to the optimal policy and action-value function as the number of episodes increases.

Challenges and Considerations

Off-policy Monte Carlo control presents unique challenges, particularly related to the variance of importance sampling estimates and slow learning rates.

Challenges and Considerations

- **High Variance**: Importance sampling can introduce high variance, especially in long episodes where the product of ratios can become unstable.
- **Slow Learning**: Learning may be slow if the behavior policy frequently chooses actions that differ from the target policy, resulting in fewer effective updates.
- **Tail-only Learning**: Learning primarily occurs at the tails of episodes when the remaining actions are greedy, potentially slowing down convergence.
- **Alternative Approaches**: Techniques such as temporal-difference learning or discounting-aware importance sampling may mitigate some issues by reducing variance and improving convergence rates.

Summary of Key Concepts

- **Off-policy Monte Carlo Control**: Utilizes a behavior policy to explore and a target policy to optimize, allowing separation of exploration and exploitation.

- **Importance Sampling**: Adjusts returns from the behavior policy to evaluate the target policy accurately, ensuring unbiased estimates.
- **Off-policy Algorithm**: Iteratively updates action-value estimates and policies using weighted returns from sampled episodes.
- **Challenges**: High variance, slow learning, and tail-only updates are notable challenges, requiring careful management of exploration strategies.
- **Considerations**: Alternative approaches, such as temporal-difference learning, can address some limitations by improving learning efficiency and reducing variance.

Off-policy Monte Carlo control provides a powerful framework for learning optimal policies in complex environments, offering flexibility and robustness in separating behavior and target policies.

The next chapter that is being covered this week is **Chapter 6: Temporal-Difference Learning**. The first section that is being covered this week is **Section 6.1: TD Prediction**.

Section 6.1: TD Prediction

Overview

This section introduces Temporal-Difference (TD) learning, a central concept in reinforcement learning that combines ideas from both Monte Carlo methods and dynamic programming (DP). TD methods allow agents to learn directly from raw experience without requiring a model of the environment. Unlike Monte Carlo methods, which wait until the end of an episode, TD methods update value estimates based on other learned estimates immediately at each time step.

TD Prediction

TD prediction focuses on estimating the value function v_π for a given policy π. It utilizes bootstrapping, updating estimates using existing value estimates and immediate rewards.

TD Prediction

- **Value Function Update**: TD methods update the value function V incrementally at each step:

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

  where R_{t+1} is the immediate reward, S_{t+1} is the subsequent state, and α is the step-size parameter.
- **TD(0) or One-Step TD**: The simplest form of TD learning that updates the value of the current state based on the next state. It combines sampling and bootstrapping, making it more data-efficient than Monte Carlo methods.
- **TD Error**: The difference between the predicted value and the updated estimate:

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

  This error measures the discrepancy between the estimated value of the current state and the backed-up value.

Comparison with Monte Carlo Methods

TD learning and Monte Carlo methods both use experience to estimate value functions, but they differ in how they handle updates.

Comparison with Monte Carlo Methods

- **Monte Carlo Methods**: Use complete returns G_t from entire episodes to update values, requiring the end of episodes to calculate returns.
- **TD Methods**: Update values at each time step using the immediate reward and the estimated value of the next state, without waiting for episode termination.
- **Bootstrapping**: TD methods update value estimates using existing estimates, unlike Monte Carlo methods which rely solely on sampled returns.
- **Sample Updates vs. Expected Updates**: TD methods rely on sample updates based on a single successor state, whereas DP methods use expected updates over all possible successor states.

Example: Driving Home

This example illustrates the application of TD prediction in estimating travel time while driving home.

Example: Driving Home

- **Scenario**: Predicting travel time based on current observations (e.g., weather, day of the week). Initial estimates are updated as new information becomes available.
- **TD Learning**: Updates travel time predictions incrementally at each decision point (e.g., reaching the car, exiting the highway) based on current and subsequent observations.
- **Outcome**: Predictions are refined continuously as new states and rewards are observed, providing immediate learning and adjustment.

Advantages of TD Methods

TD methods offer several advantages over other reinforcement learning approaches, making them efficient and practical in various scenarios.

Advantages of TD Methods

- **Immediate Learning**: TD methods allow updates at every time step, providing faster learning compared to waiting for episode completion.
- **Bootstrapping**: Combines elements of both Monte Carlo sampling and DP bootstrapping, making it more data-efficient.
- **Flexibility**: Applicable in non-stationary environments and scenarios where complete episodes may be unavailable or impractical.
- **Reduced Variance**: TD methods often exhibit lower variance in estimates compared to Monte Carlo methods, particularly in episodic tasks with long episodes.

Challenges of TD Methods

Despite their advantages, TD methods also face challenges that need careful consideration in practice.

Challenges of TD Methods

- **Bias in Estimates**: Since TD methods rely on current estimates for updates, they may introduce bias, particularly if initial estimates are inaccurate.
- **Step-Size Parameter**: Choosing an appropriate α is crucial for convergence and stability. Too high or too low values can affect learning rates.
- **Non-Stationary Environments**: In environments where dynamics change over time, maintaining up-to-date value estimates becomes challenging.

Summary of Key Concepts

- **Temporal-Difference (TD) Learning**: Combines aspects of Monte Carlo methods and dynamic programming, updating value estimates incrementally at each step.
- **TD(0) Method**: A one-step TD method that uses the next state's value estimate to update the current state's value.
- **TD Error**: The discrepancy between the current state's value estimate and the backed-up estimate, guiding updates.
- **Advantages and Challenges**: TD methods offer immediate learning and lower variance but may introduce bias and depend on careful parameter tuning.

TD learning provides a powerful framework for reinforcement learning, allowing agents to learn from experience efficiently and adaptively in dynamic environments.

The next section that is being covered this week is **Section 6.2: Advantages Of TD Prediction Methods**.

Section 6.2: Advantages Of TD Prediction Methods

Overview

This section discusses the advantages of Temporal-Difference (TD) methods in reinforcement learning, highlighting their strengths over Monte Carlo (MC) and Dynamic Programming (DP) methods. TD methods combine the benefits of both approaches, using bootstrapping to update value estimates incrementally without requiring a model of the environment.

Advantages of TD Methods

TD methods offer several advantages that make them particularly useful in various reinforcement learning scenarios.

Advantages of TD Methods

- **Model-Free Learning**: Unlike DP, TD methods do not require knowledge of the environment's transition probabilities or reward functions, allowing learning from raw experience.
- **Online and Incremental Updates**: TD methods update value estimates at each time step based on the current state and reward, providing continuous learning:

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

  This is in contrast to MC methods, which must wait until the end of an episode to update.
- **Handling Long Episodes and Continuing Tasks**: In tasks with very long episodes or no episodes (continuing tasks), TD methods allow learning without waiting for terminal states, facilitating quicker adaptation.
- **Reduced Susceptibility to Experimental Actions**: TD methods learn from each transition, reducing the impact of exploratory actions on learning speed compared to MC methods, which may need to ignore or discount exploratory episodes.

Convergence of TD Methods

TD methods have been proven to converge under certain conditions, providing robust learning in various environments.

Convergence of TD Methods

- **Convergence Guarantees**: TD(0) converges to the true value function v_π for any fixed policy π under appropriate conditions for the step-size parameter α:

$$\alpha \text{ is sufficiently small or decreases appropriately over time.}$$

- **Applicability to Linear Function Approximation**: Convergence results extend beyond table-based implementations to include cases with general linear function approximators.

Comparison of Learning Speed

TD methods generally converge faster than MC methods, especially in stochastic environments, as demonstrated in the Random Walk example.

Comparison of Learning Speed

- **Empirical Evidence**: In many practical scenarios, TD methods converge faster than constant-α MC methods, making more efficient use of limited data.
- **Example 6.2: Random Walk**:
  - **Setup**: A Markov Reward Process (MRP) with states A through E, starting from C, and moving left or right with equal probability.
  - **Comparison**: TD(0) consistently achieved lower root mean-squared (RMS) error compared to MC, as shown in learning curves for various α values.
  - **Results**: TD(0) demonstrated superior performance in terms of convergence speed and accuracy of value estimates.

Empirical Performance in Random Walk Example

The Random Walk example provides insight into the empirical performance of TD methods compared to MC methods.

Empirical Performance in Random Walk Example

- **Markov Reward Process (MRP)**: No actions involved, focusing purely on state-value prediction. The task was to estimate the probability of reaching the rightmost state from any starting state.
- **RMS Error Analysis**: TD(0) achieved consistently lower RMS error than MC, averaged over multiple episodes and runs, indicating more accurate predictions.
- **Impact of Step-Size (α)**: Varying α affected convergence, with higher α leading to faster convergence but potentially increasing variance in value estimates.

Summary of Key Concepts

- **Advantages of TD Methods**: Model-free learning, online incremental updates, applicability in long or continuous tasks, and reduced sensitivity to exploratory actions.
- **Convergence**: Proven under specific conditions, including for linear function approximators, ensuring robustness across various environments.
- **Learning Speed**: Generally faster convergence compared to MC methods, demonstrated in empirical comparisons such as the Random Walk example.
- **Empirical Performance**: TD methods show lower RMS error and quicker adaptation to true value functions, making them effective in practical applications.

TD methods provide a flexible and efficient approach for value prediction in reinforcement learning, offering significant advantages over traditional methods, especially in complex or unknown environments.

The next section that is being covered this week is **Section 6.4: Sarsa - On-policy TD Control**.

Section 6.4: Sarsa - On-policy TD Control

Overview

This section discusses Sarsa, an on-policy Temporal-Difference (TD) control method for learning the action-value function $q_\pi(s, a)$ for the current policy π. Sarsa is a combination of TD learning and Generalized Policy Iteration (GPI), where the policy is continually improved based on learned action values while balancing exploration and exploitation.

Sarsa: On-policy TD Control

Sarsa estimates the action-value function and uses it to update the policy towards optimality. The method derives its name from the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ used in its update rule.

Sarsa: On-policy TD Control

- **Update Rule**:

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

  where:

  - S_t, A_t: Current state and action.
  - R_{t+1}: Reward received after taking action A_t.
  - S_{t+1}, A_{t+1}: Next state and action.
  - α: Step-size parameter.
  - γ: Discount factor.

- **Policy Improvement**: The policy is updated to be ϵ-greedy with respect to the action-value estimates, ensuring exploration.
- **Terminal States**: If S_{t+1} is terminal, $Q(S_{t+1}, A_{t+1})$ is defined as zero.

Convergence of Sarsa

Sarsa converges to an optimal policy under specific conditions, depending on the policy's characteristics and exploration strategy.

Convergence of Sarsa

- **Convergence Conditions**:
  - All state-action pairs must be visited infinitely often.
  - The policy should converge in the limit to the greedy policy, achievable with ϵ-greedy or ϵ-soft policies.
  - ϵ is typically decreased over time (e.g., $\epsilon = \frac{1}{t}$).
- **Optimality**: Sarsa converges with probability 1 to the optimal policy and action-value function as the number of episodes approaches infinity.

Example: Windy Gridworld

The Windy Gridworld example illustrates the application of Sarsa in an environment with deterministic and stochastic elements.

Example: Windy Gridworld

- **Environment**: A grid with a crosswind that varies by column, affecting the agent's movement.
- **Objective**: Reach the goal state from the start state, minimizing the number of time steps, with constant rewards of -1 per step.
- **Actions**: Move up, down, left, or right, with wind potentially shifting the agent vertically.
- **Results**:
  - Sarsa, with parameters $\epsilon = 0.1$ and $\alpha = 0.5$, successfully learned a near-optimal policy.
  - Over time, the average episode length decreased, showing improvement in reaching the goal quickly.
  - Sarsa adapted to avoid suboptimal policies, unlike Monte Carlo methods, which might struggle with non-terminating episodes.

Advantages and Challenges of Sarsa

Sarsa presents both advantages and challenges, making it suitable for various reinforcement learning tasks with specific considerations.

Advantages and Challenges of Sarsa

- **Advantages**:
  - **On-policy Learning**: Directly learns the action-value function for the policy being followed, ensuring coherence between policy and value estimates.
  - **Exploration-Exploitation Balance**: ϵ-greedy policies provide a practical mechanism for exploration while allowing exploitation of known rewards.
  - **Flexibility**: Can handle environments where policies may not always lead to termination, adapting quickly to poor policies.
- **Challenges**:
  - **Exploration Strategy**: Proper tuning of ϵ is critical to ensure sufficient exploration without sacrificing convergence speed.
  - **Convergence Speed**: Convergence may be slow in environments with large state-action spaces, requiring careful parameter tuning.

Summary of Key Concepts

- **Sarsa Algorithm**: An on-policy TD control method that estimates action-value functions and improves policies iteratively.
- **Update Rule**: Utilizes the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ to update action-value estimates.
- **Convergence Conditions**: Requires sufficient exploration and policy convergence to the greedy policy for optimality.
- **Example - Windy Gridworld**: Demonstrates Sarsa's effectiveness in an environment with dynamic elements, showing its ability to learn optimal policies.
- **Advantages and Challenges**: Highlights the method's on-policy nature, balance of exploration and exploitation, and the need for careful parameter tuning.

Sarsa provides a robust framework for on-policy learning in reinforcement learning, balancing exploration and exploitation while continuously improving the policy based on experience.

The last section that is being covered this week is **Section 6.5: Q-learning - Off-policy TD Control**.

Section 6.5: Q-learning - Off-policy TD Control

Overview

This section explores Q-learning, a widely used off-policy Temporal-Difference (TD) control algorithm in reinforcement learning. Q-learning learns the optimal action-value function q^* independently of the policy being followed, allowing it to improve the target policy even while following a different behavior policy.

Q-learning: Off-policy TD Control

Q-learning updates its estimates of the optimal action-value function using the maximum action value of the next state, making it robust to various exploration strategies.

Q-learning: Off-policy TD Control

- **Update Rule**:

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

  where:

  - S_t, A_t: Current state and action.
  - R_{t+1}: Reward received after taking action A_t.
  - S_{t+1}: Next state.
  - α: Step-size parameter.
  - γ: Discount factor.

- **Off-policy Nature**: The algorithm updates the policy using the maximum future action value, irrespective of the current policy, leading to learning about the optimal policy.
- **Convergence**: Q-learning converges to the optimal action-value function q^* with probability 1, provided that all state-action pairs continue to be updated.

Q-learning Algorithm

The Q-learning algorithm iteratively updates action-value estimates and improves the policy based on the maximum expected future rewards.

Q-learning Algorithm

- **Initialization**:
  - Initialize $Q(s, a)$ arbitrarily for all state-action pairs (s, a).
  - Set $Q(\text{terminal}, \cdot) = 0$.
- **Loop for each episode**:
  - Initialize state S.
  - Loop for each step of the episode:
    * Choose action A from S using a policy derived from Q (e.g., ϵ-greedy).
    * Take action A, observe reward R and next state S'.
    * Update Q value:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$$

    * Update state $S \leftarrow S'$.
- **Until** S is terminal.

Example: Cliff Walking

The Cliff Walking example illustrates the differences between on-policy and off-policy methods, comparing Sarsa and Q-learning.

Example: Cliff Walking

- **Environment**: A gridworld with start and goal states, where moving into the cliff results in a large negative reward (-100) and resets the agent to the start.
- **Reward Structure**: All other transitions have a reward of -1, encouraging the agent to reach the goal quickly while avoiding the cliff.
- **Comparison**:
  - **Sarsa**: Learns a safer path, taking the exploration policy into account, resulting in fewer episodes falling off the cliff.
  - **Q-learning**: Learns the optimal policy that follows the edge of the cliff but is more prone to fall due to exploratory moves.
- **Results**: While Q-learning eventually learns the optimal path, its online performance may be worse due to risky exploratory actions, unlike Sarsa which learns a more conservative policy.

Advantages and Challenges of Q-learning

Q-learning offers significant advantages in flexibility and robustness but also presents challenges that must be managed.

Advantages and Challenges of Q-learning

- **Advantages**:
  - **Off-policy Learning**: Learns about the optimal policy independently of the behavior policy, allowing for flexible exploration strategies.
  - **Optimality**: Converges to the optimal action-value function, making it highly effective for various tasks.
  - **Simplicity**: The algorithm is straightforward to implement and widely used in practice.
- **Challenges**:
  - **Exploration-Exploitation Trade-off**: Balancing exploration and exploitation requires careful tuning of ϵ in ϵ-greedy policies.
  - **Variance in Updates**: The use of maximum estimates in updates can lead to high variance, especially in stochastic environments.
  - **Convergence Speed**: May converge slowly in large state-action spaces, necessitating appropriate parameter tuning and exploration strategies.

Summary of Key Concepts

- **Q-learning Algorithm**: An off-policy TD control method that updates action-value estimates using the maximum action value of the next state.
- **Update Rule**: Uses the maximum of future action values to guide policy improvement towards optimality.
- **Example - Cliff Walking**: Demonstrates the differences between on-policy (Sarsa) and off-policy (Q-learning) approaches in a dynamic environment.
- **Advantages and Challenges**: Highlights Q-learning's robustness and flexibility, while noting challenges in exploration, variance, and convergence speed.

Q-learning provides a powerful framework for off-policy learning, allowing agents to learn optimal policies while balancing exploration and exploitation through experience.

Approximate Reinforcement Learning, Probability Refresher

Approximate Reinforcement Learning, Probability Refresher

8.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 12.1 - Acting Under Uncertainty**
- **Artificial Intelligence - A Modern Approach - Chapter 12.2 - Basic Probability Notation**
- **Artificial Intelligence - A Modern Approach - Chapter 12.3 - Interference Using Full Joint Distributions**
- **Artificial Intelligence - A Modern Approach - Chapter 12.4 - Independence**
- **Artificial Intelligence - A Modern Approach - Chapter 12.5 - Bayes' Rule And Its Use**
- **Artificial Intelligence - A Modern Approach - Chapter 12.6 - Naive Bayes Models**
- **Artificial Intelligence - A Modern Approach - Chapter 12.7 - The Wumpus World Revisited**
- **Artificial Intelligence - A Modern Approach - Chapter 22.4 - Generalization In Reinforcement Learning**

8.0.2 Piazza

Must post at least **three** times this week to Piazza.

8.0.3 Lectures

The lectures for this week are:

- Approximate Reinforcement Learning $\approx$ 73 **min**.
- Probabilistic Reasoning 1 - Probability Review $\approx$ 69 **min**.

The lecture notes for this week are:

- Approximate Reinforcement Learning Lecture Notes
- Approximate Reinforcement Learning Review Lecture Notes
- Probabilistic Reasoning 1 - Probability Review Lecture Notes

8.0.4 Quiz

The quiz for this week is:

- Quiz 8 - Approximate Reinforcement Learning, Probability Refresher

8.0.5 Chapter Summary

The first chapter that is being covered this week is **Chapter 12: Quantifying Uncertainty**. The first section that is being covered from this chapter this week is **Section 12.1: Acting Under Uncertainty**.

Section 12.1: Acting Under Uncertainty

Overview

This section explores how agents can act rationally under uncertainty. It highlights the challenges agents face when dealing with incomplete information, unpredictable environments, and the need for decision-making under such conditions. The section emphasizes the importance of quantifying uncertainty to enable rational decision-making.

Uncertainty in Agent Actions

Agents in real-world environments must handle uncertainty caused by factors such as partial observability, nondeterminism, and adversaries. Traditional logical approaches, which rely on belief states and contingency plans, are often inadequate for dealing with such uncertainty.

Uncertainty in Agent Actions

- **Belief State**: Represents all possible states the agent might be in, given its knowledge and sensor data.
- **Contingency Planning**: Generating plans that handle every possible outcome can be inefficient and infeasible.
- **Example**: An automated taxi planning to reach the airport must consider various uncertainties such as traffic, breakdowns, and accidents.

Summarizing Uncertainty

The logical approach to handling uncertainty, such as using propositional logic for diagnosis, often fails due to practical limitations. Agents need a way to represent and reason with degrees of belief instead of relying solely on logic.

Summarizing Uncertainty

- **Degree of Belief**: Representing uncertainty using numeric values (probabilities) rather than binary true/false logic.
- **Example**: Diagnosing a toothache might involve multiple possible causes, each with a certain probability.
- **Challenges**: Laziness (incomplete rules), theoretical ignorance (lack of complete knowledge), and practical ignorance (incomplete tests).

Uncertainty and Rational Decisions

Making rational decisions under uncertainty involves considering both the likelihood of various outcomes and their relative importance. Utility theory is introduced to represent preferences and make quantitative decisions based on expected utility.

Uncertainty and Rational Decisions

- **Utility Theory**: Assigns a utility value to each possible outcome, representing the agent's preferences.
- **Decision Theory**: Combines probability theory and utility theory to make rational decisions that maximize expected utility.
- **Example**: Choosing between different plans to get to the airport, considering trade-offs between the likelihood of success and the cost (e.g., waiting time).

Agent Architecture for Decision Making

The section concludes with an outline of a decision-theoretic agent architecture, which maintains probabilistic beliefs about the state of the world and selects actions that maximize expected utility.

Agent Architecture for Decision Making

- **Belief State**: Updated based on actions and perceptions.
- **Outcome Probabilities**: Calculated for potential actions.
- **Action Selection**: Chooses the action with the highest expected utility.

Summary of Key Concepts

- **Belief State**: A representation of all possible states an agent might be in.
- **Degree of Belief**: Using probabilities to represent uncertainty.
- **Utility Theory**: Quantifying preferences to guide decision-making.
- **Decision Theory**: Combining probability and utility to make rational decisions.
- **Agent Architecture**: A framework for decision-theoretic agents to act under uncertainty.

Understanding these concepts is crucial for developing agents that can operate effectively in uncertain environments, balancing the need for accurate beliefs with practical decision-making.

The next section from this chapter that is being covered this week is **Section 12.2: Basic Probability Notation**.

Section 12.2: Basic Probability Notation

Overview

This section introduces the basic probability notation necessary for understanding and working with uncertainty in artificial intelligence. It covers foundational concepts such as random variables, probability distributions, and key axioms and properties of probability theory. This notation forms the basis for more advanced probabilistic reasoning techniques.

Random Variables

Random variables are used to represent uncertain quantities in a probabilistic framework. They can take on a range of possible values, each associated with a probability.

Random Variables

- **Definition**: A random variable X is a function that assigns a real number to each outcome in a sample space.
- **Example**: In a coin toss, the random variable X could represent the outcome, with $X = 1$ for heads and $X = 0$ for tails.

Probability Distributions

Probability distributions describe how probabilities are assigned to the values of random variables. They can be defined for both discrete and continuous variables.

Probability Distributions

- **Discrete Random Variables**: The probability distribution is specified by a probability mass function $P(X = x)$, which gives the probability that X takes on the value x.
- **Example**: For a fair die roll, $P(X = x) = \frac{1}{6}$ for $x \in \{1, 2, 3, 4, 5, 6\}$.

- **Continuous Random Variables**: The probability distribution is specified by a probability density function $f_X(x)$, where the probability that X falls within an interval $[a, b]$ is given by:

$$P(a \leq X \leq b) = \int_a^b f_X(x)\, dx$$

Axioms of Probability

The axioms of probability form the foundation of probability theory, providing the rules for how probabilities are assigned and manipulated.

Axioms of Probability

- **Non-negativity**: For any event A, $P(A) \geq 0$.
- **Normalization**: The probability of the sample space Ω is 1, $P(\Omega) = 1$.
- **Additivity**: For any two mutually exclusive events A and B,

$$P(A \cup B) = P(A) + P(B)$$

Conditional Probability

Conditional probability quantifies the probability of one event given that another event has occurred. It is a key concept for reasoning about dependencies between events.

Conditional Probability

- **Definition**: The conditional probability of A given B is defined as:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \quad \text{for } P(B) > 0$$

- **Example**: If A is the event of drawing an ace from a deck of cards and B is the event of drawing a spade, then:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{\frac{1}{52}}{\frac{13}{52}} = \frac{1}{13}$$

Independence

Two events are independent if the occurrence of one does not affect the probability of the other. Independence simplifies the computation of joint probabilities.

Independence

- **Definition**: Events A and B are independent if:

$$P(A \cap B) = P(A)P(B)$$

- **Example**: Rolling two fair dice, the outcome of one die does not affect the outcome of the other.

Summary of Key Concepts

- **Random Variables**: Represent uncertain quantities, taking on various values with associated probabilities.
- **Probability Distributions**: Describe how probabilities are assigned to random variables, through probability mass functions for discrete variables and density functions for continuous variables.
- **Axioms of Probability**: The foundational rules for probability, including non-negativity, normalization, and additivity.
- **Conditional Probability**: The probability of an event given that another event has occurred,

quantifying dependencies.

- **Independence**: Events that do not affect each other's probabilities, simplifying joint probability computations.

Understanding these basic probability notations is essential for modeling and reasoning about uncertainty in intelligent systems, forming the basis for more advanced probabilistic methods and algorithms.

The next section from this chapter that is being covered this week is **Section 12.3: Interference Using Full Joint Distributions**.

Section 12.3: Interference Using Full Joint Distributions

Overview

This section delves into inference using full joint probability distributions, which provide a comprehensive way to represent the probabilities of all possible combinations of variable values in a domain. The full joint distribution serves as the foundation for deriving probabilities of interest through marginalization and conditioning.

Full Joint Probability Distributions

A full joint probability distribution specifies the probability of every possible combination of values for a set of random variables. It is fundamental to probabilistic reasoning but often impractical to use directly due to its size.

Full Joint Probability Distributions

- **Definition**: For a set of random variables $X_1, X_2, \ldots, X_n$, the full joint probability distribution $P(X_1, X_2, \ldots, X_n)$ assigns a probability to each possible combination of values $(x_1, x_2, \ldots, x_n)$.
- **Example**: In a domain with two binary variables, *Rain* and *Sprinkler*, the full joint distribution $P(Rain, Sprinkler)$ includes probabilities for all four combinations of true/false values.

Marginalization

Marginalization involves summing the probabilities of the full joint distribution over the values of one or more variables to obtain the marginal probability of a subset of variables.

Marginalization

- **Definition**: The marginal probability $P(X)$ of a variable X is obtained by summing over the probabilities of all other variables:
$$P(X) = \sum_Y P(X, Y)$$
- **Example**: To find $P(Rain)$ from the joint distribution $P(Rain, Sprinkler)$, sum over all values of *Sprinkler*:
$$P(Rain = true) = P(Rain = true, Sprinkler = true) + P(Rain = true, Sprinkler = false)$$

Conditional Probability and Independence

Conditional probabilities can be derived from the full joint distribution using the definition of conditional probability. Independence assumptions simplify the joint distribution by reducing the number of probabilities that must be specified.

Conditional Probability and Independence

- **Conditional Probability**: The conditional probability $P(X|Y)$ is derived from the joint distribution:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)}$$

- **Independence**: Two variables X and Y are independent if:

$$P(X,Y) = P(X)P(Y)$$

  Independence allows for simplifying the joint distribution into the product of marginal probabilities.

Inference by Enumeration

Inference by enumeration involves computing desired probabilities by summing over the appropriate entries in the full joint distribution. This method, while straightforward, can be computationally expensive for large domains.

Inference by Enumeration

- **Process**: To compute $P(X)$, sum over all entries in the joint distribution that are consistent with X:

$$P(X) = \sum_{y_1,y_2,\dots,y_n} P(X, Y_1 = y_1, Y_2 = y_2, \dots, Y_n = y_n)$$

- **Example**: To compute $P(Rain|Sprinkler = true)$, sum over all values of $Rain$ and $Sprinkler$ consistent with $Sprinkler = true$:

$$\begin{aligned} P(Rain|Sprinkler = true) &= \frac{P(Rain = true, Sprinkler = true)}{P(Sprinkler = true)} \\ &= \frac{P(Rain = true, Sprinkler = true)}{P(Rain = true, Sprinkler = true) + P(Rain = false, Sprinkler = true)} \end{aligned}$$

Summary of Key Concepts

- **Full Joint Probability Distributions**: Represent the probabilities of all possible combinations of variable values.
- **Marginalization**: Summing over the joint distribution to obtain marginal probabilities.
- **Conditional Probability**: Deriving conditional probabilities from the joint distribution.
- **Independence**: Simplifying joint distributions using independence assumptions.
- **Inference by Enumeration**: Computing probabilities by summing entries in the joint distribution, though computationally expensive for large domains.

Understanding full joint distributions and their manipulations is essential for probabilistic reasoning, forming the basis for more sophisticated inference methods in uncertain environments.

The next section from this chapter that is being covered this week is **Section 12.4: Independence**.

Section 12.4: Independence

Overview

This section discusses the concept of independence, which plays a crucial role in simplifying probabilistic models. Independence between variables allows for more efficient representation and computation by reducing the complexity of joint probability distributions.

Types of Independence

There are different types of independence that can be leveraged in probabilistic models: absolute (marginal) independence and conditional independence.

Types of Independence

- **Absolute (Marginal) Independence**: Two variables X and Y are absolutely independent if:
$$P(X,Y) = P(X)P(Y)$$
This implies that knowing the value of one variable provides no information about the other.
- **Conditional Independence**: Two variables X and Y are conditionally independent given a third variable Z if:
$$P(X,Y|Z) = P(X|Z)P(Y|Z)$$
This implies that given Z, knowing X provides no additional information about Y and vice versa.

Benefits of Independence

Independence assumptions simplify the specification and computation of joint probability distributions, making it feasible to handle large and complex probabilistic models.

Benefits of Independence

- **Simplification**: Independence reduces the number of parameters needed to specify a joint distribution. For n variables, the full joint distribution requires 2^n parameters, but with independence assumptions, this number can be significantly reduced.
- **Efficient Computation**: Independence allows for decomposing joint probability distributions into products of smaller, marginal or conditional distributions, enabling more efficient computation.
- **Example**: In a medical diagnosis model, symptoms might be conditionally independent given the disease. This reduces the complexity of the model and makes inference more tractable.

Graphical Models and Independence

Graphical models, such as Bayesian networks, leverage independence assumptions to represent complex joint distributions compactly and efficiently.

Graphical Models and Independence

- **Bayesian Networks**: Use directed acyclic graphs (DAGs) to represent variables and their conditional dependencies. Each node represents a variable, and edges represent direct dependencies.
- **Markov Assumptions**: Each variable is conditionally independent of its non-descendants given its parents in the network.
- **Example**: In a Bayesian network for a medical diagnosis, nodes might represent diseases and symptoms, with edges indicating which symptoms are directly influenced by which diseases.

Summary of Key Concepts

- **Absolute (Marginal) Independence**: Simplifies joint distributions by assuming no direct influence between variables.
- **Conditional Independence**: Further simplifies models by assuming independence given some other variables.
- **Benefits of Independence**: Reduces the number of parameters and computational complexity in probabilistic models.
- **Graphical Models**: Leverage independence assumptions to efficiently represent and compute joint

distributions.

Understanding and utilizing independence is crucial for building scalable and efficient probabilistic models, enabling more effective reasoning under uncertainty.

The next section from this chapter that is being covered this week is **Section 12.5: Bayes' Rule And Its Use**.

Section 12.5: Bayes' Rule And Its Use

Overview

This section introduces Bayes' Rule, a fundamental theorem in probability theory that allows for updating probabilities based on new evidence. Bayes' Rule is crucial for probabilistic reasoning and is extensively used in various AI applications, such as diagnostics, decision-making, and machine learning.

Bayes' Rule

Bayes' Rule provides a way to update the probability of a hypothesis H given new evidence E. It relates the posterior probability $P(H|E)$ to the prior probability $P(H)$, the likelihood $P(E|H)$, and the marginal likelihood $P(E)$.

Bayes' Rule

- **Formula**: Bayes' Rule is expressed as:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

  where:

  - $P(H|E)$ is the posterior probability of the hypothesis H given evidence E.
  - $P(E|H)$ is the likelihood of the evidence E given that the hypothesis H is true.
  - $P(H)$ is the prior probability of the hypothesis H.
  - $P(E)$ is the marginal likelihood of the evidence E, calculated as:

$$P(E) = \sum_i P(E|H_i)P(H_i)$$

    summing over all possible hypotheses.

- **Example**: For medical diagnosis, let H be a disease and E be a symptom. Bayes' Rule updates the probability of the disease given the symptom.

Applications of Bayes' Rule

Bayes' Rule is applied in numerous fields within AI, providing a systematic way to update beliefs and make decisions based on evidence.

Applications of Bayes' Rule

- **Medical Diagnosis**: Updating the probability of a disease based on observed symptoms and test results.
- **Spam Filtering**: Classifying emails as spam or not spam based on the presence of certain words.
- **Machine Learning**: Bayesian inference methods use Bayes' Rule for updating model parameters based on data.

- **Robotics**: Updating the robot's belief about its location based on sensor readings.

Deriving Bayes' Rule

Bayes' Rule can be derived from the definition of conditional probability and the product rule of probability.

Deriving Bayes' Rule

- **Conditional Probability**: The definition of conditional probability is:

$$P(H|E) = \frac{P(H \cap E)}{P(E)} \quad \text{and} \quad P(E|H) = \frac{P(H \cap E)}{P(H)}$$

- **Product Rule**: Rearranging the product rule gives:

$$P(H \cap E) = P(E|H)P(H) = P(H|E)P(E)$$

- **Bayes' Rule**: Solving for $P(H|E)$ gives:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Normalization

The denominator $P(E)$ in Bayes' Rule ensures that the posterior probabilities sum to 1, making it a normalization factor.

Normalization

- **Calculating $P(E)$**: The marginal likelihood $P(E)$ is computed by summing the joint probabilities over all hypotheses:

$$P(E) = \sum_i P(E|H_i)P(H_i)$$

- **Normalization Factor**: Ensures that the posterior probabilities are properly scaled:

$$P(H|E) = \frac{P(E|H)P(H)}{\sum_i P(E|H_i)P(H_i)}$$

Summary of Key Concepts

- **Bayes' Rule**: Provides a method for updating probabilities based on new evidence.
- **Formula**: $P(H|E) = \frac{P(E|H)P(H)}{P(E)}$.
- **Applications**: Used in medical diagnosis, spam filtering, machine learning, and robotics.
- **Derivation**: Derived from the definition of conditional probability and the product rule.
- **Normalization**: Ensures that posterior probabilities sum to 1, calculated using the marginal likelihood.

Understanding Bayes' Rule is essential for probabilistic reasoning and decision-making in uncertain environments, enabling systematic updates to beliefs based on observed evidence.

The next section from this chapter that is being covered this week is **Section 12.6: Naive Bayes Models**.

Section 12.6: Naive Bayes Models

Overview

This section introduces Naive Bayes models, a simple yet powerful probabilistic classifier based on Bayes' Rule. The "naive" assumption is that features are conditionally independent given the class label. Despite this simplification, Naive Bayes models perform well in various real-world applications, particularly in text classification and spam filtering.

The Naive Bayes Assumption

The Naive Bayes assumption simplifies the computation of the joint probability of the features given the class by assuming conditional independence among the features.

The Naive Bayes Assumption

- **Conditional Independence**: Given a class C, the features $X_1, X_2, \dots, X_n$ are conditionally independent:
$$P(X_1, X_2, \dots, X_n|C) = \prod_{i=1}^{n} P(X_i|C)$$
- **Simplification**: This assumption reduces the complexity of calculating the joint probability from exponential to linear in the number of features.
- **Example**: In spam filtering, words in an email are considered independent given the email is spam or not spam.

Classification with Naive Bayes

Naive Bayes classifiers use Bayes' Rule and the Naive Bayes assumption to compute the posterior probability of each class given the observed features and to classify new instances.

Classification with Naive Bayes

- **Posterior Probability**: For a given instance with features $X_1, X_2, \dots, X_n$, the posterior probability of class C_k is:
$$P(C_k|X_1, X_2, \dots, X_n) = \frac{P(C_k)\prod_{i=1}^{n} P(X_i|C_k)}{P(X_1, X_2, \dots, X_n)}$$
- **Classification Rule**: The instance is classified into the class with the highest posterior probability:
$$\hat{C} = \arg\max_{C_k} P(C_k)\prod_{i=1}^{n} P(X_i|C_k)$$
- **Example**: In spam filtering, classify an email as spam if $P(\text{spam}|\text{words}) > P(\text{not spam}|\text{words})$.

Parameter Estimation

The parameters of a Naive Bayes model, namely the prior probabilities of the classes and the conditional probabilities of the features given the classes, can be estimated from training data.

Parameter Estimation

- **Class Priors**: Estimated as the relative frequencies of the classes in the training set:
$$P(C_k) = \frac{N(C_k)}{N}$$
where $N(C_k)$ is the number of instances of class C_k and N is the total number of instances.
- **Conditional Probabilities**: Estimated as the relative frequencies of the feature values given the class:
$$P(X_i = x_i|C_k) = \frac{N(X_i = x_i, C_k)}{N(C_k)}$$
where $N(X_i = x_i, C_k)$ is the number of instances where feature X_i takes value x_i in class C_k.

Applications of Naive Bayes

Naive Bayes models are widely used in various domains due to their simplicity, efficiency, and effectiveness in classification tasks.

Applications of Naive Bayes

- **Text Classification**: Categorizing documents or emails into predefined categories based on word frequencies.
- **Spam Filtering**: Classifying emails as spam or not spam using the presence of certain keywords.
- **Sentiment Analysis**: Determining the sentiment of a text (positive, negative, neutral) based on word usage.
- **Medical Diagnosis**: Diagnosing diseases based on symptoms and patient history.

Summary of Key Concepts

- **Naive Bayes Assumption**: Features are conditionally independent given the class.
- **Classification**: Uses Bayes' Rule to compute posterior probabilities and classify instances into the most probable class.
- **Parameter Estimation**: Class priors and conditional probabilities are estimated from training data.
- **Applications**: Widely used in text classification, spam filtering, sentiment analysis, and medical diagnosis.

Understanding Naive Bayes models provides a foundation for applying probabilistic reasoning to practical classification problems, leveraging simplicity and efficiency to achieve effective results.

The last section from this chapter that is being covered this week is **Section 12.7: The Wumpus World Revisited**.

Section 12.7: The Wumpus World Revisited

Overview

This section revisits the Wumpus World, a classic AI problem, to illustrate how probabilistic reasoning can be applied to complex, uncertain environments. The Wumpus World is a grid-based environment where an agent must navigate to find gold while avoiding pits and the Wumpus, a dangerous creature. This section demonstrates how to use probability to handle the inherent uncertainty in the Wumpus World.

The Wumpus World Environment

The Wumpus World consists of a grid of squares, some of which contain pits, and one contains the Wumpus. The agent must find the gold without falling into a pit or encountering the Wumpus.

The Wumpus World Environment

- **Grid Layout**: The environment is a 4x4 grid with each square potentially containing a pit, the Wumpus, or being empty.
- **Percepts**: The agent receives percepts indicating nearby dangers:
  - **Breeze**: Indicates a pit in an adjacent square.
  - **Stench**: Indicates the Wumpus is in an adjacent square.
  - **Glitter**: Indicates the gold is in the current square.

- **Actions**: The agent can move forward, turn left, turn right, grab the gold, or shoot an arrow to kill the Wumpus.

Using Probabilities in the Wumpus World

Probabilistic reasoning is used to handle the uncertainty in the agent's knowledge about the Wumpus World's state, helping the agent to make informed decisions.

Using Probabilities in the Wumpus World

- **Belief State**: Represents the agent's knowledge about the environment, updated using Bayesian inference based on percepts.
- **Probability Distribution**: The agent maintains a probability distribution over possible states of the world, reflecting the likelihood of pits, the Wumpus, and the gold being in each square.
- **Example**: If the agent perceives a breeze, it updates the probability of pits in adjacent squares.
- **Update Rule**: Bayes' Rule is used to update the belief state:

$$P(X|E) = \frac{P(E|X)P(X)}{P(E)}$$

  where X represents the state of the world and E represents the percepts.

Inference and Decision Making

The agent uses its probabilistic model to infer the most likely state of the world and make decisions that maximize its chances of success.

Inference and Decision Making

- **Risk Assessment**: The agent assesses the risks associated with different actions by considering the probabilities of encountering pits or the Wumpus.
- **Action Selection**: Actions are chosen to maximize expected utility, considering both the likelihood of success and potential dangers.
- **Example**: If the agent perceives a stench but no breeze, it might infer that moving into a square with a high probability of containing the Wumpus is too risky and choose an alternative path.

Performance in the Wumpus World

The probabilistic approach allows the agent to make more informed decisions, improving its performance in navigating the Wumpus World.

Performance in the Wumpus World

- **Improved Decision Making**: The agent uses probabilities to evaluate the safety and potential rewards of different actions, leading to better overall performance.
- **Flexibility**: The probabilistic model can handle new percepts and update the belief state dynamically, allowing the agent to adapt to changing information.
- **Example**: The agent successfully finds the gold while avoiding pits and the Wumpus by continually updating its belief state and making decisions based on the most likely scenarios.

Summary of Key Concepts

- **Wumpus World Environment**: A grid-based environment with pits, the Wumpus, and gold.
- **Probabilistic Reasoning**: Using probabilities to handle uncertainty and update the belief state based on percepts.
- **Inference and Decision Making**: Making decisions that maximize expected utility by assessing risks

and updating beliefs.

- **Performance**: Improved decision-making and flexibility in adapting to new information.

Revisiting the Wumpus World with probabilistic reasoning highlights the power of probability in handling uncertainty and making rational decisions in complex environments.

The next chapter that is being covered this week is **Chapter 22: Reinforcement Learning**. The section that is being covered from this chapter this week is **Section 22.4: Generalization In Reinforcement Learning**.

Section 22.4: Generalization In Reinforcement Learning

Overview

This section addresses generalization in reinforcement learning, highlighting the limitations of tabular representations of utility and Q-functions in large state spaces. It introduces function approximation techniques, which allow reinforcement learning algorithms to generalize from observed states to unvisited states, thereby improving learning efficiency and performance in complex environments.

Function Approximation

Function approximation methods construct compact representations of utility or Q-functions, enabling efficient learning in large state spaces. This is essential for practical applications where the number of states is vast.

Function Approximation

- **Utility Function Approximation**: Approximates the utility function $U(s)$ using a parameterized function $\hat{U}_\theta(s)$, such as a weighted linear combination of features:

$$\hat{U}_\theta(s) = \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

- **Example**: In a 4x3 grid world, the utility function can be approximated using features like the coordinates of the grid squares.

- **Gradient Descent**: Parameters θ are updated using gradient descent to minimize the error between the predicted and actual utilities.

- **Widrow-Hoff Rule**: Also known as the delta rule, used for online least-squares updates:

$$\theta_i \leftarrow \theta_i + \alpha[u_j(s) - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

Approximating Direct Utility Estimation

Direct utility estimation generates trajectories and uses the sum of rewards as training examples for supervised learning algorithms, enabling approximation of utility functions for large state spaces.

Approximating Direct Utility Estimation

- **Linear Function Approximator**: Uses features such as coordinates in a grid world to approximate utility:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

- **Online Learning**: Parameters are updated after each trial based on observed rewards, using gradient descent to reduce prediction error.

- **Example**: If the total reward obtained from state $(1, 1)$ is 0.4 and the current utility estimate is 0.8, the parameters are adjusted to reduce the utility estimate.

Approximating Temporal-Difference Learning

Temporal-Difference (TD) learning methods can also be adapted to use function approximation, adjusting parameters to minimize temporal differences between successive states.

Approximating Temporal-Difference Learning

- **TD Learning Update Rule**: Parameters are updated to reduce the temporal difference:

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s') + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

- **Q-learning Update Rule**: Similar update rule for Q-learning with function approximation:

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s') + \gamma \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)]\frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

- **Challenges**: With nonlinear function approximators like neural networks, parameters may diverge, leading to instability.

Experience Replay

Experience replay addresses catastrophic forgetting, where the agent may forget important experiences as it learns, by reusing past experiences to reinforce learning.

Experience Replay

- **Mechanism**: The agent stores trajectories and periodically replays them to ensure the value function remains accurate for all visited states.
- **Benefit**: Prevents the agent from forgetting important past experiences and maintains stability in learning.
- **Example**: A self-driving car replays experiences where it narrowly avoided obstacles to reinforce safe driving behaviors.

Deep Reinforcement Learning

Deep reinforcement learning combines deep neural networks with reinforcement learning algorithms to handle high-dimensional state spaces and automatically extract useful features.

Deep Reinforcement Learning

- **Deep Neural Networks**: Used as function approximators to learn complex state representations from raw inputs.
- **Backpropagation**: Gradients required for parameter updates are computed using backpropagation.
- **Applications**: Achieved significant results in playing video games, Go, and robotic tasks.
- **Challenges**: Deep RL systems can behave unpredictably in environments that differ from training data.

Reward Shaping

Reward shaping involves providing additional rewards, or pseudorewards, to guide the agent's learning process, addressing the credit assignment problem in environments with sparse rewards.

Reward Shaping

- **Pseudorewards**: Additional rewards for intermediate progress towards the goal, speeding up learning.
- **Risk**: Agent might learn to maximize pseudorewards rather than true rewards.
- **Example**: A soccer robot receives pseudorewards for making contact with the ball or advancing it towards the goal.

Hierarchical Reinforcement Learning

Hierarchical reinforcement learning (HRL) breaks down complex tasks into smaller sub-tasks, simplifying learning by decomposing the overall task hierarchy.

Hierarchical Reinforcement Learning

- **Hierarchical Structure**: Tasks are decomposed into sub-tasks, each with its own policies and value functions.
- **Joint State Space**: Combines physical state and machine state (program counter, arguments, variables) to form a joint state space.
- **Example**: In a soccer game, higher-level actions like passing and shooting are broken down into lower-level motor behaviors.
- **Benefit**: Learning becomes more efficient by focusing on smaller sub-problems within the task hierarchy.

Summary of Key Concepts

- **Function Approximation**: Enables efficient learning by representing utility and Q-functions compactly.
- **Experience Replay**: Prevents forgetting by reusing past experiences.
- **Deep Reinforcement Learning**: Combines deep neural networks with RL for high-dimensional state spaces.
- **Reward Shaping**: Uses pseudorewards to guide learning in sparse reward environments.
- **Hierarchical Reinforcement Learning**: Decomposes tasks into sub-tasks for more efficient learning.

Generalization in reinforcement learning is crucial for handling large state spaces, enabling agents to learn efficiently and perform effectively in complex environments.

Bayes Network

Bayes Network

9.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 13.1 - Representing Knowledge In An Uncertain Domain**
- **Artificial Intelligence - A Modern Approach - Chapter 13.2 - The Semantics Of Bayesian Networks**
- **Artificial Intelligence - A Modern Approach - Chapter 13.3 - Exact Inference In Bayesian Networks**

9.0.2 Piazza

Must post at least **three** times this week to Piazza.

9.0.3 Lectures

The lectures for this week are:

- Probabilistic Reasoning 1 - Probability Review $\approx$ 69 **min**.
- Bayes Net - Representation $\approx$ 34 **min**.
- Bayes Net - Independence $\approx$ 32 **min**.
- Bayes Net - Inference $\approx$ 50 **min**.

The lecture notes for this week are:

- Bayes Net - Independence Lecture Notes
- Bayes Net - Inference Lecture Notes
- Bayes Net - Representation Lecture Notes

9.0.4 Quiz

The quiz for this week is:

- Quiz 9 - Bayes Network

9.0.5 Chapter Summary

This weeks material is from **Chapter 13: Probabilistic Reasoning**. The first section that is covered from this chapter this week is **Section 13.1: Representing Knowledge In An Uncertain Domain**.

Section 13.1: Representing Knowledge In An Uncertain Domain

Overview

This section introduces Bayesian networks as a powerful tool for representing and reasoning with probabilistic knowledge in uncertain domains. It explains the limitations of full joint probability distributions and the advantages of using Bayesian networks to capture conditional independence relationships efficiently.

Bayesian Networks

Bayesian networks are graphical models that use directed acyclic graphs (DAGs) to represent the dependencies among variables. Each node in the network represents a random variable, and the directed edges between nodes represent probabilistic dependencies.

Bayesian Networks

- **Nodes and Edges**: Each node corresponds to a random variable, which can be either discrete or continuous. Directed edges indicate a dependency between the connected nodes.
- **Directed Acyclic Graph (DAG)**: The structure of the network is a DAG, meaning it has no directed cycles. This hierarchical structure allows for a clear representation of causal relationships.
- **Conditional Probability Table (CPT)**: Each node X_i is associated with a CPT $P(X_i|\text{Parents}(X_i))$, which quantifies the effect of its parents on the node. The CPT provides the probability distribution of the node given its parents' values.
- **Example**: Consider a network with nodes for Burglary (B), Earthquake (E), Alarm (A), JohnCalls (J), and MaryCalls (M). The edges from B and E to A, and from A to J and M, capture the dependencies among these variables.

Constructing Bayesian Networks

Constructing a Bayesian network involves determining the variables, their dependencies, and the corresponding conditional probabilities. The process typically follows these steps:

Constructing Bayesian Networks

- **Identifying Variables**: Determine the set of relevant variables for the domain.
- **Ordering Variables**: Order the variables in a sequence where causes precede effects.
- **Adding Links**: For each variable, identify its parents from the preceding variables in the order. Add directed edges from the parents to the variable.
- **Specifying CPTs**: Define the conditional probability table for each variable given its parents.
- **Example**: In a burglary scenario, the variable "Alarm" might depend on "Burglary" and "Earthquake," with a CPT that specifies the probability of the alarm going off given the occurrence of a burglary or earthquake.

Mathematical Representation

The joint probability distribution of a set of variables in a Bayesian network can be factored into a product of conditional probabilities, leveraging the conditional independence relationships.

Mathematical Representation

- **Chain Rule for Bayesian Networks**: The joint probability $P(X_1, X_2, \ldots, X_n)$ of the variables can be written as:
$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|\text{Parents}(X_i))$$
- **Example**: For the Burglary example, the joint probability is:
$$P(B, E, A, J, M) = P(B)P(E)P(A|B, E)P(J|A)P(M|A)$$

Advantages of Bayesian Networks

Bayesian networks offer several advantages over full joint probability distributions, particularly in handling large and complex domains.

Advantages of Bayesian Networks

- **Compact Representation**: By exploiting conditional independence, Bayesian networks require fewer parameters to represent the joint distribution.
- **Ease of Specification**: Experts can often specify the local relationships (CPTs) more easily than the full joint distribution.
- **Efficient Inference**: Algorithms can leverage the network structure to perform probabilistic inference more efficiently.
- **Scalability**: Suitable for large-scale applications due to their compactness and efficiency.
- **Example**: A network with 30 binary variables, each having up to 5 parents, requires significantly fewer parameters (960) compared to the full joint distribution (over a billion parameters).

Applications of Bayesian Networks

Bayesian networks are widely used in various fields, including medical diagnosis, risk assessment, and decision support systems. They provide a systematic way to model uncertainty and make informed decisions.

Applications of Bayesian Networks

- **Medical Diagnosis**: Used to model the probabilistic relationships between diseases and symptoms, aiding in diagnostic reasoning.
- **Risk Assessment**: Employed to evaluate the likelihood of different risks and their impacts in fields such as finance and engineering.
- **Decision Support**: Assist in making decisions under uncertainty by modeling the dependencies among various factors and their probable outcomes.
- **Example**: A Bayesian network in medical diagnosis might include variables for different diseases, symptoms, and test results, with edges representing causal relationships and CPTs providing the conditional probabilities.

Summary of Key Concepts

- **Bayesian Networks**: Graphical models that represent probabilistic dependencies using directed acyclic graphs (DAGs).
- **Nodes and Edges**: Nodes represent random variables, and directed edges indicate dependencies.
- **Conditional Probability Tables (CPTs)**: Quantify the effect of parent nodes on each variable.
- **Constructing Bayesian Networks**: Involves identifying variables, ordering them, adding links, and specifying CPTs.
- **Mathematical Representation**: The joint probability distribution is factored into a product of conditional probabilities.
- **Advantages**: Offer compact representation, ease of specification, efficient inference, and scalability.
- **Applications**: Used in medical diagnosis, risk assessment, decision support, and more.

Understanding Bayesian networks is crucial for modeling and reasoning under uncertainty, enabling the development of robust and efficient probabilistic models for various applications.

The next section that is covered from this chapter this week is **Section 13.2: The Semantics Of Bayesian Networks**.

Section 13.2: The Semantics Of Bayesian Networks

Overview

This section delves into the semantics of Bayesian networks, explaining how the structure of the network encodes probabilistic relationships among variables. It covers the formal interpretation of nodes and edges, and how to understand the joint probability distribution represented by a Bayesian network.

Conditional Independence

The key idea behind Bayesian networks is the representation of conditional independence among variables. The network structure encodes these dependencies, simplifying the representation of the joint probability distribution.

Conditional Independence

- **Definition**: A variable X_i is conditionally independent of its non-descendants, given its parents.
- **Mathematical Formulation**: For variables X, Y, Z, X is conditionally independent of Y given Z if:

$$P(X|Y, Z) = P(X|Z)$$

- **Example**: In the Burglary network, JohnCalls and MaryCalls are conditionally independent given Alarm.

Local Semantics

The local semantics of a Bayesian network specify that each node is independent of its non-parents given its parents. This local property leads to the global property of the joint distribution.

Local Semantics

- **Node Independence**: Each node is conditionally independent of its non-parents given its parents.
- **Mathematical Expression**: For a node X_i with parents $\text{Parents}(X_i)$:

$$P(X_i|\text{NonParents}(X_i), \text{Parents}(X_i)) = P(X_i|\text{Parents}(X_i))$$

- **Example**: In the Burglary network, $P(\text{JohnCalls}|\text{Burglary}, \text{Earthquake}, \text{Alarm}) = P(\text{JohnCalls}|\text{Alarm})$.

Global Semantics

The global semantics of a Bayesian network describe the joint probability distribution over all variables as the product of the local conditional probabilities defined by the network structure.

Global Semantics

- **Joint Probability Distribution**: The joint probability of all variables is the product of the conditional probabilities of each variable given its parents:

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|\text{Parents}(X_i))$$

- **Example**: For the Burglary network, the joint probability is:

$$P(B, E, A, J, M) = P(B)P(E)P(A|B, E)P(J|A)P(M|A)$$

d-Separation

d-Separation is a graphical criterion used to determine whether a set of variables is conditionally independent given another set of variables. It provides a method for reading conditional independencies directly from the network structure.

d-Separation

- **Definition**: A set of nodes X is d-separated from a set of nodes Y given a set of nodes Z if all paths from any node in X to any node in Y are blocked by Z.
- **Blocking Criteria**: A path is blocked if:
  - It contains a node such that one of its arcs is directed towards the node and neither the node nor any of its descendants are in Z.
  - It contains a collider (a node with two incoming arcs) that is not in Z and has no descendants in Z.
- **Example**: In the Burglary network, JohnCalls is d-separated from Earthquake given Alarm.

Markov Blanket

The Markov blanket of a node in a Bayesian network is the minimal set of nodes that renders the node conditionally independent of the rest of the network. It consists of the node's parents, children, and children's parents.

Markov Blanket

- **Definition**: The Markov blanket of a node X includes:
  - The parents of X.
  - The children of X.
  - The parents of the children of X.
- **Mathematical Formulation**: Given the Markov blanket $\text{MB}(X)$ of X:

$$P(X|\text{All nodes except } X) = P(X|\text{MB}(X))$$

- **Example**: In the Burglary network, the Markov blanket of Alarm includes Burglary, Earthquake, JohnCalls, and MaryCalls.

Summary of Key Concepts

- **Conditional Independence**: Encodes dependencies among variables, reducing complexity.
- **Local Semantics**: Each node is conditionally independent of its non-parents given its parents.
- **Global Semantics**: The joint probability distribution is the product of local conditional probabilities.
- **d-Separation**: A criterion to determine conditional independence from the network structure.
- **Markov Blanket**: The minimal set of nodes that renders a node conditionally independent of the rest of the network.

Understanding the semantics of Bayesian networks is crucial for accurately modeling probabilistic dependencies and performing efficient inference in complex domains.

The last section that is covered from this chapter this week is **Section 13.3: Exact Inference In Bayesian Networks**.

Section 13.3: Exact Inference In Bayesian Networks

Overview

This section covers exact inference methods in Bayesian networks, which involve computing the posterior distribution of a set of query variables given some evidence. It discusses several algorithms for performing exact inference, including enumeration, variable elimination, and belief propagation.

Inference by Enumeration

Inference by enumeration involves summing over the joint probability distribution to compute the desired posterior probabilities. While straightforward, it can be computationally expensive for large networks.

Inference by Enumeration

- **Posterior Probability**: To compute $P(X|e)$ for query variable X and evidence e, sum over all possible values of the hidden variables Y:

$$P(X|e) = \alpha P(X, e) = \alpha \sum_{Y} P(X, Y, e)$$

  where α is a normalization constant.

- **Example**: In the Burglary network, to find $P(\text{Burglary}|\text{JohnCalls} = \text{true})$, sum over all possible values of Earthquake, Alarm, and MaryCalls.

Variable Elimination

Variable elimination is an efficient method for exact inference that systematically eliminates variables by summing out their probabilities, reducing the complexity compared to enumeration.

Variable Elimination

- **Procedure**:
  - Express the joint probability distribution as a product of factors.
  - Eliminate variables one by one by summing them out, combining factors as necessary.
- **Mathematical Formulation**: For a query $P(X|e)$, where Z are the variables to be eliminated:

$$P(X|e) = \alpha \sum_{Z} \prod_{i} f_i$$

- **Example**: In the Burglary network, to compute $P(\text{JohnCalls} = \text{true})$, eliminate variables Earthquake, Burglary, Alarm, and MaryCalls in sequence.

Belief Propagation

Belief propagation (also known as the sum-product algorithm) is an exact inference method for tree-structured Bayesian networks. It passes messages between nodes to compute marginal probabilities.

Belief Propagation

- **Message Passing**: Nodes send messages to their neighbors containing summarized information from the rest of the network.
- **Procedure**:
  - Initialize messages at the leaf nodes.
  - Pass messages inward to a root node.
  - Pass messages outward from the root node to compute marginals.
- **Mathematical Formulation**: For a node X, the message $\mu_{Y \to X}(X)$ from node Y to node X is:

$$\mu_{Y \to X}(X) = \sum_{Y} P(Y|\text{Parents}(Y)) \prod_{Z \in \text{Children}(Y) \setminus X} \mu_{Z \to Y}(Y)$$

- **Example**: In a chain-structured network, belief propagation efficiently computes marginals by passing messages along the chain.

Complexity of Exact Inference

The complexity of exact inference in Bayesian networks depends on the network structure and the chosen method. In general, exact inference is NP-hard, but certain structures allow for more efficient computation.

Complexity of Exact Inference

- **Treewidth**: The complexity of variable elimination and belief propagation is exponential in the treewidth of the network. The treewidth is a measure of how tree-like the network is.
- **Efficient Structures**: For tree-structured networks, belief propagation is efficient. For networks with low treewidth, variable elimination is feasible.
- **Intractable Cases**: For networks with high treewidth, exact inference becomes intractable, necessitating approximate methods.

Summary of Key Concepts

- **Inference by Enumeration**: Summing over the joint probability distribution, straightforward but computationally expensive.
- **Variable Elimination**: Systematically eliminates variables by summing out their probabilities, more efficient than enumeration.
- **Belief Propagation**: An exact inference method for tree-structured networks, using message passing to compute marginals.
- **Complexity of Exact Inference**: Depends on the network's treewidth; exact inference is generally NP-hard but feasible for certain structures.

Exact inference methods in Bayesian networks allow for precise probabilistic reasoning in structured domains, but their feasibility depends on the network's complexity and structure.

Machine Learning And Midterm 2

Machine Learning And Midterm 2

10.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 19.1 - Forms Of Learning**
- **Artificial Intelligence - A Modern Approach - Chapter 19.2 - Supervised Learning**
- **Artificial Intelligence - A Modern Approach - Chapter 19.3 - Learning Decision Trees**
- **Artificial Intelligence - A Modern Approach - Chapter 19.4 - Model Selection And Optimization**
- **Artificial Intelligence - A Modern Approach - Chapter 19.5 - The Theory Of Learning**
- **Artificial Intelligence - A Modern Approach - Chapter 19.6 - Linear Regression And Classification**
- **Artificial Intelligence - A Modern Approach - Chapter 19.7 - Nonparametric Models**
- **Artificial Intelligence - A Modern Approach - Chapter 19.8 - Ensemble Learning**

10.0.2 Piazza

Must post at least **three** times this week to Piazza.

10.0.3 Lectures

The lectures for this week are:

- Machine Learning Intro ≈ 32 **min**.
- Linear Regression Refresher ≈ 37 **min**.
- Logistic Regression (Binary-Class Classification) Refresher ≈ 53 **min**.
- Regularization And Cross Validation ≈ 27 **min**.
- Non-Parametric Models ≈ 64 **min**.

The lecture notes for this week are:

- Improving Training Lecture Notes
- Linear Regression Lecture Notes
- Logistic Regression Lecture Notes
- Machine Learning Intro Lecture Notes
- Non-Parametric Models Lecture Notes

10.0.4 Assignments

The assignment(s) for this week are:

- Assignment 6 - Kaggle Cancer Competition

10.0.5 Quiz

The quiz for this week is:

- Quiz 9 - Basic Machine Learning

10.0.6 Exam

The exam for this week is:

- Exam 2 Notes
- Exam 2

10.0.7 Chapter Summary

The chapter that is being covered this week is **Chapter 19: Learning From Examples**. The first section that is being covered from this chapter this week is **Section 19.1: Forms Of Learning**.

Section 19.1: Forms Of Learning

Overview

This section introduces the various forms of learning that an agent can undergo to improve its performance. Learning involves observing the environment, gathering data, and building models that help predict outcomes or guide actions. It covers the different components of agent programs that can be enhanced through learning, the role of prior knowledge, and the different types of feedback that can guide the learning process.

Components of an Agent that Can Learn

Learning can enhance various components of an agent's architecture, leading to improved decision-making and adaptability.

Components of an Agent that Can Learn

- **Direct Mapping from States to Actions**: The agent can learn condition-action rules, improving its response to specific situations.
- **Perceptual Processing**: The agent can learn to infer relevant properties of the environment from sensory inputs.
- **World Model**: The agent can learn about the effects of its actions and how the world evolves.
- **Utility Information**: Learning can enhance the agent's utility function, refining its understanding of the desirability of different states.
- **Action-Value Information**: The agent can learn the value of different actions in various states, helping to prioritize certain behaviors.
- **Goals and Problem-Solving Capabilities**: The agent can refine its goals and problem-solving strategies through learning.

Prior Knowledge and Learning

The learning process is influenced by the agent's prior knowledge and the type of model it uses to represent this knowledge.

Prior Knowledge and Learning

- **Influence on Model Building**: Prior knowledge can shape the model framework chosen by the agent, facilitating faster and more accurate learning.
- **Learning from Scratch**: The section also discusses scenarios where the agent starts with minimal prior knowledge and builds understanding entirely from observed data.

- **Transfer Learning**: Mentioned as a method where knowledge from one domain is applied to a new domain, enhancing learning efficiency.

Types of Learning

Learning can be categorized based on the type of feedback provided to the agent, leading to different approaches and techniques.

Types of Learning

- **Supervised Learning**: The agent learns a mapping from inputs to outputs using labeled data. This involves observing pairs of inputs and their corresponding outputs and developing a function h that approximates the underlying function f.
- **Unsupervised Learning**: The agent identifies patterns or structures in the input data without explicit output labels. A common task is clustering, where the agent groups similar inputs together.
- **Reinforcement Learning**: The agent learns from a series of reinforcements—rewards and punishments—associated with actions taken. The agent aims to maximize cumulative rewards by discovering which actions yield the best outcomes.

Applications of Learning in Software Systems

Learning technologies are widely applicable in modern software engineering, providing significant improvements in performance and efficiency.

Applications of Learning in Software Systems

- **Image Analysis**: Machine learning models can significantly speed up image analysis tasks, such as those used in astrophysics.
- **Energy Efficiency**: Machine learning can optimize energy usage, such as reducing the cooling costs of data centers.
- **Broad Impact**: The integration of learning algorithms into various software systems marks a transformative phase in computer science and AI, often referred to as the "Golden Age" of machine learning.

Summary of Key Concepts

- **Learning Components**: Various components of an agent, such as direct mapping, perceptual processing, and utility functions, can be improved through learning.
- **Prior Knowledge**: Influences the model-building process and can accelerate learning.
- **Types of Learning**: Supervised, unsupervised, and reinforcement learning cater to different types of feedback and learning objectives.
- **Applications**: Machine learning is broadly applied across different domains, enhancing the capabilities and efficiency of software systems.

Understanding these forms of learning and their applications is essential for developing intelligent systems capable of improving over time through experience.

The next section that is being covered from this chapter this week is **Section 19.2: Supervised Learning**.

Section 19.2: Supervised Learning

Overview

This section provides an in-depth look at supervised learning, a type of machine learning where an agent learns a function that maps inputs to outputs using labeled training data. It discusses the key concepts involved, such as hypothesis spaces, model selection, and the trade-offs between bias and variance.

Supervised Learning Defined

In supervised learning, the agent observes input-output pairs and learns a function that maps inputs to outputs. This function, often called a hypothesis, is trained using a labeled dataset.

Supervised Learning Defined

- **Training Set**: A set of N example input-output pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, where each pair is generated by an unknown function $y = f(x)$.
- **Hypothesis**: The function h learned by the agent, which approximates the true function f.
- **Objective**: Discover a function h such that $h(x) \approx f(x)$ for all x, minimizing the discrepancy between the predicted outputs and the true outputs.
- **Example**: In an image classification task, the input x might be a pixel representation of an image, and the output y a label such as "cat" or "dog".

Hypothesis Space and Model Selection

The hypothesis space is the set of all possible functions h that the learning algorithm can choose from. The choice of hypothesis space affects the learning process and the model's ability to generalize.

Hypothesis Space and Model Selection

- **Hypothesis Space** H: The set of possible functions, e.g., linear functions, polynomials, decision trees.
- **Consistent Hypothesis**: A hypothesis h is consistent if $h(x_i) = y_i$ for all training examples (x_i, y_i).
- **Best-Fit Function**: When outputs are continuous, find a function that minimizes the error $|h(x_i) - y_i|$.
- **Generalization**: The ability of a hypothesis to perform well on unseen data, not just on the training set.

Bias and Variance

Bias and variance are critical factors in supervised learning that describe the behavior of the learned function in relation to the training data and the true function.

Bias and Variance

- **Bias**: The error due to overly simplistic assumptions in the learning algorithm. High bias can lead to underfitting, where the model fails to capture the underlying pattern in the data.
- **Variance**: The error due to the model's sensitivity to small fluctuations in the training set. High variance can lead to overfitting, where the model captures noise in the training data.
- **Bias-Variance Tradeoff**: The balance between bias and variance; finding the right complexity level of the model to minimize total error.
- **Example**: A linear model may have high bias but low variance, while a high-degree polynomial may have low bias but high variance.

Overfitting and Underfitting

Overfitting and underfitting are common issues in supervised learning, affecting the model's performance on new data.

Overfitting and Underfitting

- **Overfitting**: When the model learns the training data too well, including noise, leading to poor performance on unseen data. This is often characterized by a model that fits the training data almost perfectly but generalizes poorly.
- **Underfitting**: When the model is too simple to capture the underlying structure of the data, leading to poor performance on both training and test data.
- **Example**: A degree-12 polynomial may overfit a set of 13 points, capturing all fluctuations in the data, while a simple linear model may underfit the same data by failing to capture important trends.

Model Selection and Evaluation

Choosing the right model involves balancing complexity and generalization. The performance of a model is often evaluated using a separate test set.

Model Selection and Evaluation

- **Test Set**: A separate set of input-output pairs not seen during training, used to evaluate the model's generalization performance.
- **Performance Metrics**: Metrics such as accuracy, precision, recall, and F1 score are used to evaluate classification models, while mean squared error (MSE) and root mean squared error (RMSE) are common for regression models.
- **Cross-Validation**: A technique to assess how the model performs on different subsets of the data, providing a more robust estimate of its generalization ability.
- **Example**: In a classification problem, the accuracy might be measured as the proportion of correctly classified examples in the test set.

Summary of Key Concepts

- **Supervised Learning**: Learning from labeled training data to map inputs to outputs.
- **Hypothesis Space**: The set of possible models the learning algorithm can consider.
- **Bias and Variance**: Key factors affecting the model's performance and generalization.
- **Overfitting and Underfitting**: Common issues where models either learn too much noise or fail to capture the underlying pattern.
- **Model Selection**: Choosing the right model based on its performance on unseen data.

Understanding these concepts is crucial for developing effective supervised learning models that generalize well to new, unseen data, balancing the trade-offs between model complexity and generalization ability.

The next section that is being covered from this chapter this week is **Section 19.3: Learning Decision Trees**.

Section 19.3: Learning Decision Trees

Overview

This section covers the concept and methodology of learning decision trees, a powerful tool for classification tasks. Decision trees represent functions that map a set of attribute values to specific outputs. They are particularly useful for tasks requiring interpretable models and handle both discrete and continuous input data effectively.

Decision Trees Explained

A decision tree reaches a decision by performing a sequence of tests on input attributes, each internal node representing a test, branches representing possible outcomes, and leaf nodes representing the final decision or output.

Decision Trees Explained

- **Structure**: Internal nodes correspond to tests on attributes, branches correspond to attribute values, and leaf nodes represent decisions.
- **Example**: In the restaurant domain, attributes like Patrons, Type, and WaitEstimate are used to decide whether to wait for a table.
- **Boolean Classification**: Decision trees can perform Boolean classification, where the output is either true (positive example) or false (negative example).

Expressiveness of Decision Trees

Decision trees can represent any Boolean function, but their size and complexity depend on the nature of the function being represented.

Expressiveness of Decision Trees

- **Disjunctive Normal Form (DNF)**: A decision tree can represent functions in DNF, where the output is true if at least one conjunction of conditions is met.
- **Example**: The decision tree for the restaurant domain represents a function where the output is "Wait" based on conditions like the number of patrons and estimated waiting time.
- **Limitations**: Some functions, like the majority function or parity function, require large decision trees. Real-valued functions with non-rectangular boundaries are also challenging to represent compactly.

Learning Decision Trees from Examples

The process involves finding a decision tree that fits a set of examples, using a greedy divide-and-conquer strategy to select the most informative attributes.

Learning Decision Trees from Examples

- **Algorithm (LEARN-DECISION-TREE)**:
  1. If all examples have the same classification, return the classification.
  2. If no attributes are left, return the most common classification.
  3. Otherwise, select the attribute with the highest importance and split the examples.
- **Importance Function**: Measures the attribute's effectiveness in classifying examples, often using information gain.
- **Example**: In the restaurant domain, the algorithm might first split on the attribute "Patrons" and then further split on "Hungry".

Choosing Attribute Tests

The choice of attribute for splitting is crucial and is guided by measures like information gain, which is based on the concept of entropy from information theory.

Choosing Attribute Tests

- **Entropy**: A measure of uncertainty or impurity in the data. For a Boolean variable V with probabilities $P(v_k)$:

$$H(V) = -\sum_k P(v_k) \log_2 P(v_k)$$

- **Information Gain**: The reduction in entropy from splitting on an attribute A:

$$\text{Gain}(A) = H(\text{Output}) - \text{Remainder}(A)$$

- **Example**: In the restaurant domain, the attribute "Patrons" provides significant information gain, making it a good choice for the first split.

Generalization and Overfitting

A key challenge in learning decision trees is ensuring the model generalizes well to new data, avoiding overfitting.

Generalization and Overfitting

- **Overfitting**: Occurs when the model captures noise in the training data, leading to poor generalization.
- **Pruning**: A technique to remove parts of the tree that provide little predictive power, thereby reducing complexity and preventing overfitting.
- **Significance Tests**: Used to decide whether an attribute split is meaningful, often employing statistical methods like the chi-squared test.
- **Example**: In the restaurant domain, attributes like "Raining" or "Reservation" might be pruned if they do not significantly improve classification.

Broadening the Applicability of Decision Trees

Decision trees can be adapted to handle missing data, continuous attributes, and even regression tasks, broadening their applicability.

Broadening the Applicability of Decision Trees

- **Handling Missing Data**: Techniques like assigning a probability distribution to missing values or using surrogate splits.
- **Continuous and Multi-Valued Attributes**: Techniques like using split points for continuous data and information gain ratio for attributes with many values.
- **Regression Trees**: Extend decision trees to predict continuous outputs, using methods like linear regression at the leaves.
- **Example**: Predicting house prices using features like square footage and number of bedrooms, where the output is continuous.

Summary of Key Concepts

- **Decision Trees**: A method for classifying inputs based on a sequence of attribute tests.
- **Expressiveness**: Can represent any Boolean function but may require large trees for complex functions.
- **Learning Algorithm**: Uses a greedy approach to build the tree based on the most informative attributes.
- **Information Gain**: A measure used to select the best attribute for splitting.
- **Generalization**: Ensuring the model performs well on unseen data, often using techniques like pruning.
- **Adaptability**: Can handle various data types and tasks, including regression and handling missing data.

Understanding decision trees and their properties is crucial for applying them effectively in various classification and regression tasks, ensuring that they generalize well to new data while remaining interpretable.

The next section that is being covered from this chapter this week is **Section 19.4: Model Selection And Optimization**.

Section 19.4: Model Selection And Optimization

Overview

This section discusses the critical aspects of model selection and optimization in machine learning. It covers the concepts of selecting a suitable hypothesis, measuring performance, dealing with overfitting and underfitting, and the various methods and strategies for tuning models and hyperparameters.

Stationarity and i.i.d. Assumptions

Model selection often relies on the assumption that future examples will resemble past ones, described by the stationarity assumption. This assumption is crucial for predicting the model's performance on unseen data.

Stationarity and i.i.d. Assumptions

- **Stationarity Assumption**: Assumes that the distribution of examples remains consistent over time:

$$P(E_j) = P(E_{j+1}) = \cdots$$

- **Independent and Identically Distributed (i.i.d.)**: Assumes that each example E_j is independent of others:

$$P(E_j) = P(E_j | E_{j-1}, E_{j-2}, \ldots)$$

Error Rate and Hypothesis Testing

The error rate measures the proportion of incorrect predictions made by a model. Model selection involves finding a hypothesis that minimizes this error rate.

Error Rate and Hypothesis Testing

- **Error Rate**: The proportion of times the predicted output $h(x)$ does not match the true output y:

$$\text{Error rate} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(h(x_i) \neq y_i)$$

- **Training, Validation, and Test Sets**: The data is typically split into these sets to prevent overfitting and ensure unbiased evaluation.

Overfitting and Underfitting

Overfitting occurs when a model learns the noise in the training data, while underfitting happens when a model is too simple to capture the underlying pattern.

Overfitting and Underfitting

- **Overfitting**: Characterized by a model that performs well on training data but poorly on unseen data.
- **Underfitting**: A model that fails to capture the underlying trend in the data, resulting in high bias.
- **Example**: A decision tree with too many nodes may overfit, while a simple linear model may underfit complex data.

Cross-Validation

Cross-validation is a technique used to assess the generalization performance of a model, typically involving partitioning the data into subsets.

Cross-Validation

- **K-Fold Cross-Validation**: The data is divided into k subsets, with the model trained on $k-1$ subsets and validated on the remaining subset. This process is repeated k times, each subset serving as the validation set once.
- **Leave-One-Out Cross-Validation (LOOCV)**: A special case of k-fold cross-validation where $k = N$, the number of examples.

Model Complexity and Regularization

Model complexity affects both the fit and the generalization ability of a model. Regularization techniques are employed to control this complexity.

Model Complexity and Regularization

- **Regularization**: Introduces a penalty for complexity, often expressed as a term added to the error function:
$$\text{Cost}(h) = \text{EmpLoss}(h) + \lambda \text{Complexity}(h)$$
- **Regularization Functions**: Common choices include the sum of squares of coefficients for polynomial models.
- **Example**: Lasso (L1 regularization) and Ridge (L2 regularization) are common techniques that penalize the absolute or squared values of model coefficients.

Hyperparameter Tuning

Selecting optimal hyperparameters is crucial for improving model performance. Various techniques are used for tuning, including grid search, random search, and more sophisticated methods like Bayesian optimization.

Hyperparameter Tuning

- **Grid Search**: Exhaustively searches over a specified set of hyperparameter values.
- **Random Search**: Samples hyperparameter values randomly within a specified range, often more efficient than grid search.
- **Bayesian Optimization**: Treats hyperparameter tuning as an optimization problem, balancing exploration and exploitation.

Summary of Key Concepts

- **Model Selection**: Involves choosing the best hypothesis based on performance metrics like error rate.
- **Overfitting and Underfitting**: Critical challenges in machine learning, managed through techniques like cross-validation and regularization.
- **Cross-Validation**: A robust method for assessing a model's generalization ability.
- **Regularization**: Helps control model complexity, preventing overfitting.
- **Hyperparameter Tuning**: Essential for optimizing model performance, with methods ranging from simple grid search to advanced Bayesian optimization.

Understanding these aspects of model selection and optimization is crucial for developing robust, generalizable machine learning models that perform well on unseen data.

The next section that is being covered from this chapter this week is **Section 19.5: The Theory Of Learning**.

Section 19.5: The Theory Of Learning

Overview

This section delves into the theoretical foundations of learning in artificial intelligence, focusing on the conditions under which learned hypotheses can be expected to generalize well to new data. It discusses key concepts such as Probably Approximately Correct (PAC) learning, sample complexity, and the PAC learning model's application to decision lists.

Probably Approximately Correct (PAC) Learning

PAC learning is a framework that provides a probabilistic guarantee on the accuracy of learned hypotheses, assuming the examples are drawn from a stationary distribution.

Probably Approximately Correct (PAC) Learning

- **PAC Learning Model**: A hypothesis h is said to be probably approximately correct if, with high probability, it has a low generalization error:

$$\text{error}(h) = \sum_{x,y} L_{0/1}(y, h(x)) P(x, y)$$

  where $L_{0/1}(y, h(x))$ is the 0/1 loss function, indicating a misclassification.
- **Generalization Error**: The expected error over the distribution of all possible examples, not just the training set.
- **PAC Condition**: A hypothesis h is approximately correct if its error is at most ϵ, and the probability that all consistent hypotheses are approximately correct is at least $1 - \delta$.

Sample Complexity

Sample complexity refers to the number of training examples required to ensure a hypothesis is probably approximately correct.

Sample Complexity

- **Sample Complexity Bound**: The number of samples N needed depends on the desired error bound ϵ and confidence level $1 - \delta$:

$$N \geq \frac{1}{\epsilon}\left(\ln \frac{1}{\delta} + \ln |H|\right)$$

  where $|H|$ is the size of the hypothesis space.
- **Implications**: As the hypothesis space grows, the number of examples needed increases, making it crucial to balance complexity and generalization.

Hypothesis Space and PAC Learning

The hypothesis space H defines the set of all hypotheses the learning algorithm can consider. The complexity of this space impacts the PAC learnability of a problem.

Hypothesis Space and PAC Learning

- **VC Dimension**: A measure of the capacity of a hypothesis space, indicating the largest number of points that can be shattered (classified in all possible ways) by the hypotheses in the space.
- **Relation to Sample Complexity**: The VC dimension helps determine the sample complexity; higher VC dimensions generally require more samples for reliable learning.
- **Example**: The class of linear functions in two dimensions has a VC dimension of 3, as three points can be arranged in such a way that they can be classified in all possible ways by linear functions.

Decision Lists and PAC Learning

Decision lists are a specific type of hypothesis space that is PAC-learnable under certain conditions, offering a practical example of the PAC framework.

Decision Lists and PAC Learning

- **Decision List**: A series of tests (conjunctions of literals) where each test leads to a decision if it matches the input, otherwise proceeding to the next test.
- **Learning Algorithm**: The DECISION-LIST-LEARNING algorithm constructs a decision list by selecting tests that match subsets of training examples, ensuring consistency with the examples.
- **PAC Learnability**: For k-DL, the class of decision lists with up to k literals per test, the sample complexity is polynomial in the number of attributes n and k:

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + O(nk\log_2(nk))\right)$$

Summary of Key Concepts

- **PAC Learning**: Provides a framework for understanding the conditions under which learning algorithms generalize well.
- **Sample Complexity**: The number of training examples needed to achieve a given accuracy and confidence.
- **VC Dimension**: A measure of the capacity of the hypothesis space, influencing sample complexity.
- **Decision Lists**: A practical example of a hypothesis space that can be PAC-learned with a reasonable number of examples.

Understanding these theoretical foundations is crucial for designing learning algorithms that not only fit training data but also generalize well to unseen examples, ensuring robust and reliable performance in real-world applications.

The next section that is being covered from this chapter this week is **Section 19.6: Linear Regression And Classification**.

Section 19.6: Linear Regression And Classification

Overview

This section introduces linear regression and classification, fundamental techniques in machine learning. It covers the basic concepts of univariate and multivariable linear regression, gradient descent, and extends these concepts to linear classifiers, including logistic regression.

Univariate Linear Regression

Univariate linear regression involves fitting a straight line to a set of data points, finding the best line that minimizes the squared errors between the predicted and actual values.

Univariate Linear Regression

- **Model**: The model is represented by $y = w_1x + w_0$, where w_0 and w_1 are the coefficients to be learned.

- **Objective**: Minimize the empirical loss using the squared-error loss function:

$$\text{Loss}(h_w) = \sum_{j=1}^{N}(y_j - h_w(x_j))^2 = \sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2$$

- **Optimal Weights**: The weights w_0 and w_1 that minimize the loss can be found using:

$$w_1 = \frac{N\sum x_j y_j - (\sum x_j)(\sum y_j)}{N\sum x_j^2 - (\sum x_j)^2}$$

$$w_0 = \frac{\sum y_j - w_1 \sum x_j}{N}$$

Gradient Descent

Gradient descent is an optimization technique used to find the minimum of a function. It is particularly useful in linear regression for finding the best-fitting line.

Gradient Descent

- **Process**: Iteratively update weights to reduce the loss:

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i}\text{Loss}(w)$$

- **Learning Rate** (α): Controls the step size during each update. It can be fixed or decay over time.
- **Stochastic Gradient Descent (SGD)**: Uses a small random subset of data (minibatch) to update the weights, making the process faster and more efficient.

Multivariable Linear Regression

Extends univariate linear regression to multiple input variables, where the output is a linear combination of multiple input features.

Multivariable Linear Regression

- **Model**: $h_w(x) = w_0 + \sum_i w_i x_i$ or in vector form $h_w(x) = w^T x$.
- **Optimal Weights**: Determined using:

$$w^* = (X^T X)^{-1} X^T y$$

where X is the matrix of input features and y is the vector of target values.

- **Regularization**: Includes additional terms in the loss function to prevent overfitting, such as L1 (Lasso) or L2 (Ridge) regularization:

$$\text{Cost}(h) = \text{EmpLoss}(h) + \lambda \text{Complexity}(h)$$

Linear Classification with Hard and Soft Thresholds

Linear functions can also be used for classification tasks, distinguishing between classes using a decision boundary.

Linear Classification with Hard and Soft Thresholds

- **Hard Threshold (Perceptron)**: Classifies inputs as 0 or 1 based on whether $w^T x$ is above or below a threshold.
- **Soft Threshold (Logistic Regression)**: Uses the logistic function to output probabilities, providing a more nuanced classification:

$$\text{Logistic}(z) = \frac{1}{1 + e^{-z}}$$

$$h_w(x) = \text{Logistic}(w^T x)$$

- **Gradient Descent in Logistic Regression**: The weights are updated based on the derivative of the logistic function:
$$w_i \leftarrow w_i + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))x_i$$

Summary of Key Concepts

- **Linear Regression**: Involves fitting a linear model to data to predict continuous outcomes.
- **Gradient Descent**: An optimization technique for finding the best-fit parameters in regression models.
- **Multivariable Linear Regression**: Extends univariate regression to multiple input features.
- **Regularization**: Prevents overfitting by adding penalty terms to the loss function.
- **Linear Classification**: Uses linear models for classification tasks, with hard thresholds (Perceptron) and soft thresholds (Logistic Regression).

These concepts form the foundation of many machine learning models, providing tools for both regression and classification tasks, with applications ranging from predictive analytics to decision-making systems.

The next section that is being covered from this chapter this week is **Section 19.7: Nonparametric Models**.

Section 19.7: Nonparametric Models

Overview

This section delves into the theory of learning, focusing on the key concepts of generalization, overfitting, and the theoretical frameworks that guide learning algorithms. It explores the challenges of ensuring that a learned hypothesis performs well not just on training data but also on new, unseen data. The section also introduces key concepts such as Probably Approximately Correct (PAC) learning and discusses methods for balancing model complexity with the need for accurate predictions.

Generalization and Overfitting

Generalization refers to a model's ability to perform well on new, unseen data. Overfitting occurs when a model is too closely tailored to the training data, capturing noise rather than the underlying data distribution.

Generalization and Overfitting

- **Generalization**: The capacity of a model to predict accurately on new data that was not used during training.
- **Overfitting**: When a model learns not only the underlying patterns in the training data but also the noise, leading to poor generalization.
- **Trade-off**: Balancing the complexity of the model with its ability to generalize. Simpler models may underfit, while more complex models may overfit.
- **Example**: A high-degree polynomial may fit the training data perfectly but fail to generalize to new data points.

PAC Learning

PAC (Probably Approximately Correct) learning is a framework that formalizes the conditions under which a learning algorithm can generalize well from a limited sample of training data.

PAC Learning

- **Definition**: A learning algorithm is PAC if, for any hypothesis space, it can find a hypothesis that with high probability (1 -) is approximately correct (error) after seeing a polynomial number of examples.
- **Sample Complexity**: The number of training examples required to ensure that with high probability, the learned hypothesis has a small generalization error. Given by:

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|H|\right)$$

- **Hypothesis Space**: The set of all hypotheses that can be learned by the algorithm. The size and complexity of the hypothesis space directly impact the number of examples needed for learning.

VC Dimension

The Vapnik-Chervonenkis (VC) dimension is a measure of the capacity of a statistical model, defined as the maximum number of points that can be shattered (i.e., correctly classified in all possible ways) by the model.

VC Dimension

- **Definition**: The VC dimension of a model class is the size of the largest set of points that can be shattered by the model.
- **Significance**: A higher VC dimension indicates a more complex model that can capture more intricate patterns but also risks overfitting.
- **Bound on Generalization Error**: The VC dimension provides a bound on the generalization error, helping in the analysis of model performance.
- **Example**: For a linear classifier in a 2D space, the VC dimension is 3, as it can shatter any three points.

Regularization and Model Selection

Regularization techniques are used to prevent overfitting by adding a penalty for more complex models. Model selection involves choosing the best model from a set of candidates based on performance criteria.

Regularization and Model Selection

- **Regularization**: Introduces a penalty term to the loss function to discourage complex models:

$$\text{Cost}(h) = \text{EmpLoss}(h) + \lambda\text{Complexity}(h)$$

  where λ controls the trade-off between fitting the data and model complexity.
- **Model Selection**: The process of selecting a hypothesis from a set of hypotheses that best matches the true function. This involves balancing the trade-off between bias and variance.
- **Cross-Validation**: A technique used to evaluate model performance by dividing the data into training and validation sets. Common methods include k-fold cross-validation.

Summary of Key Concepts

- **Generalization**: The ability of a model to perform well on new data.
- **Overfitting**: When a model captures noise in the training data, leading to poor performance on new data.
- **PAC Learning**: A theoretical framework that ensures a model's accuracy with high probability after a sufficient number of examples.
- **VC Dimension**: A measure of a model's capacity to classify data points.
- **Regularization**: Techniques to prevent overfitting by adding penalties for complexity.
- **Model Selection**: The process of choosing the most appropriate model based on performance.

Understanding these concepts is crucial for developing robust and accurate learning algorithms that generalize well across different datasets and conditions.

The last section that is being covered from this chapter this week is **Section 19.8: Ensemble Learning**.

Section 19.8: Ensemble Learning

Overview

Ensemble learning involves combining multiple models, called base models, to create a more robust and accurate predictive model. This section explores different ensemble methods, including bagging, random forests, stacking, boosting, and gradient boosting. Ensemble methods can reduce both bias and variance, improving the model's generalization performance.

Bagging

Bagging (Bootstrap Aggregating) involves training multiple versions of a model on different random subsets of the training data, sampled with replacement. The final prediction is made by averaging (prediction) or voting (classification) the outputs of the individual models.

Bagging

- **Process**: Generate K training sets by sampling with replacement from the original dataset. Train K models and combine their predictions.
- **Application**: Particularly useful for models like decision trees that can vary significantly with small changes in the training data.
- **Formula**: For regression, the final output is:

$$h(x) = \frac{1}{K}\sum_{i=1}^{K} h_i(x)$$

Random Forests

Random forests are an extension of bagging applied to decision trees, where additional randomness is introduced by selecting a random subset of features for each split in the decision tree.

Random Forests

- **Randomness in Feature Selection**: At each split, a random subset of features is selected, and the best feature from this subset is chosen.
- **Ensemble of Decision Trees**: Multiple decision trees are grown, each using a random subset of the training data and features.
- **Advantages**: Reduces variance and helps prevent overfitting, as different trees may capture different aspects of the data.

Stacking

Stacking, or stacked generalization, involves training multiple models (potentially of different types) and then training a meta-model to combine their predictions.

Stacking

- **Process**: Train base models on the original data and use their outputs as inputs for the meta-model, which learns to combine these outputs.
- **Meta-Model**: The meta-model can be any machine learning model, including linear regression or neural networks.
- **Benefits**: Can improve performance by leveraging the strengths of different types of models.

Boosting

Boosting combines multiple weak learners to create a strong learner. It focuses on examples that previous models misclassified, adjusting the weights of these examples in subsequent iterations.

Boosting

- **Weighted Training Sets**: Each example is assigned a weight, and weights are adjusted to focus on difficult-to-classify examples.
- **Algorithm (e.g., ADABOOST)**: Train weak learners sequentially, each focusing on the mistakes of the previous ones. The final model is a weighted combination of all weak learners.
- **Formula**: For binary classification, the final prediction is:

  $$h(x) = \text{sign}\left(\sum_{i=1}^{K} \alpha_i h_i(x)\right)$$

  where α_i is the weight of the i-th learner.

Gradient Boosting

Gradient boosting is an advanced form of boosting that optimizes a loss function by adding models in a stage-wise fashion, minimizing the residual errors of the combined model.

Gradient Boosting

- **Optimization**: Uses gradient descent to minimize the loss function of the ensemble.
- **Component Models**: Often uses decision trees as the base models. The new model added at each stage fits to the negative gradient of the loss function.
- **Popular Implementations**: Includes algorithms like XGBoost, which are widely used in practice for their efficiency and performance.

Summary of Key Concepts

- **Bagging**: Reduces variance by averaging over multiple models trained on different data subsets.
- **Random Forests**: Adds feature randomness to bagging, improving robustness and reducing overfitting.
- **Stacking**: Combines different model types using a meta-model to enhance predictive performance.
- **Boosting**: Focuses on difficult examples by adjusting weights, creating a strong learner from weak learners.
- **Gradient Boosting**: Uses gradient descent to optimize an ensemble, particularly effective with decision trees.

Ensemble learning enhances model performance and robustness by combining multiple models, leveraging their collective strengths, and mitigating individual weaknesses.

Computer Vision With Deep Learning

Computer Vision With Deep Learning

11.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Artificial Intelligence - A Modern Approach - Chapter 21.1 - Simple Feedforward Networks**
- **Artificial Intelligence - A Modern Approach - Chapter 21.2 - Computation Graphs For Deep Learning**
- **Artificial Intelligence - A Modern Approach - Chapter 21.3 - Convolutional Networks**
- **Artificial Intelligence - A Modern Approach - Chapter 21.4 - Learning Algorithms**
- **Artificial Intelligence - A Modern Approach - Chapter 21.5 - Generalization**
- **Artificial Intelligence - A Modern Approach - Chapter 25.1 - Introduction**
- **Artificial Intelligence - A Modern Approach - Chapter 25.2 - Image Formation**
- **Artificial Intelligence - A Modern Approach - Chapter 25.3 - Simple Image Features**
- **Artificial Intelligence - A Modern Approach - Chapter 25.4 - Classifying Images**
- **Artificial Intelligence - A Modern Approach - Chapter 25.5 - Detecting Objects**
- **Artificial Intelligence - A Modern Approach - Chapter 25.6 - The 3D World**
- **Artificial Intelligence - A Modern Approach - Chapter 25.7 - Using Computer Vision**

11.0.2 Piazza

Must post at least **three** times this week to Piazza.

11.0.3 Lectures

The lectures for this week are:

- Intro To Deep Learning And MLP $\approx$ 39 **min**.
- Neural Network Training: Backpropagation $\approx$ 29 **min**.
- Nueral Network Training: Optimization And Training Tips $\approx$ 40 **min**.
- Deep Learning Hardware $\approx$ 9 **min**.
- Intro To Keras And Demo $\approx$ 33 **min**.
- Convolutional Neural Network - Intro And Definitions $\approx$ 66 **min**.
- CNN2: Architectures And Training $\approx$ 69 **min**.

The lecture notes for this week are:

- Convolutional Neural Networks 1 Lecture Notes
- Convolutional Neural Networks 2 Lecture Notes
- Intro To Deep Learning Lecture Notes
- Neural Network Training Lecture Notes
- Optimization And Tips For Neural Network Training Lecture Notes

11.0.4 Chapter Summary

The chapters that are being covered this week are **Chapter 21: Deep Learning** and **Chapter 25: Computer Vision**. The first topic that is being covered from **Chapter 21: Deep Learning** is **Section 21.1: Simple Feedforward Networks**.

Section 21.1: Simple Feedforward Networks

Overview

This section introduces simple feedforward networks, a foundational architecture in deep learning. Feedforward networks are characterized by their lack of cycles and one-way connections from input nodes to output nodes. These networks compute functions by passing information from inputs to outputs through a series of transformations at each layer.

Feedforward Networks Explained

Feedforward networks consist of nodes arranged in layers, where each node computes a function of its inputs and passes the result to subsequent layers. The network forms a directed acyclic graph, ensuring a unidirectional flow of information.

Feedforward Networks Explained

- **Architecture**: Comprised of input, hidden, and output layers. Each node (unit) in a layer takes inputs from the previous layer and computes an output.
- **Mathematical Representation**: Each unit computes a weighted sum of its inputs followed by a nonlinear activation function:

$$a_j = g_j\left(\sum_i w_{i,j} a_i\right) \equiv g_j(\text{in}_j)$$

  where $w_{i,j}$ are the weights, a_i are the inputs, and g_j is the activation function of unit j.
- **No Cycles**: Information flows in one direction only, with no cycles, distinguishing feedforward networks from recurrent networks.

Networks as Complex Functions

Each feedforward network can be viewed as a complex function composed of simpler functions computed at each layer. The power of these networks lies in their ability to approximate complex mappings from inputs to outputs.

Networks as Complex Functions

- **Universal Approximation Theorem**: States that a network with one hidden layer can approximate any continuous function given sufficient units and appropriate weights.
- **Activation Functions**: Nonlinear functions applied at each unit to introduce non-linearity. Common functions include:
  - **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$
  - **ReLU (Rectified Linear Unit)**: $\text{ReLU}(x) = \max(0, x)$
  - **Tanh**: $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$
- **Example**: The output $\hat{y}$ of a network with two inputs, one hidden layer of two units, and one output unit can be expressed as:

$$\hat{y} = g_5(w_{0,5} + w_{3,5}g_3(\text{in}_3) + w_{4,5}g_4(\text{in}_4))$$

  where g_3 and g_4 are the activation functions in the hidden layer.

Gradients and Learning

Gradient-based optimization methods, such as gradient descent, are used to train feedforward networks by minimizing a loss function. The gradient of the loss with respect to the weights is computed using backpropagation.

Gradients and Learning

- **Loss Function**: Often the squared loss L_2 for regression tasks:

$$\text{Loss}(h_w) = \sum (y - \hat{y})^2$$

  where $\hat{y}$ is the predicted output and y is the true output.

- **Backpropagation**: A method for calculating the gradient of the loss function with respect to each weight in the network:

$$\Delta w_{i,j} \propto \frac{\partial \text{Loss}}{\partial w_{i,j}}$$

- **Vanishing Gradient Problem**: In deep networks, gradients can diminish as they propagate back through the layers, leading to slow learning or convergence issues.

Summary of Key Concepts

- **Feedforward Networks**: Simple networks with unidirectional information flow and no cycles.
- **Complex Function Representation**: Can approximate complex functions using layers of nonlinear units.
- **Activation Functions**: Key to introducing non-linearity into the network.
- **Gradient-Based Learning**: Uses backpropagation to adjust weights and minimize the loss function.
- **Challenges**: Includes issues like the vanishing gradient problem, which can hinder training in deep networks.

Understanding these foundational aspects of feedforward networks is crucial for building and training deep learning models capable of tackling a wide range of tasks.

The next topic that is being covered from this chapter this week is **Section 21.2: Computation Graphs For Deep Learning**.

Section 21.2: Computation Graphs For Deep Learning

Overview

This section explores the concept of computation graphs in deep learning, a foundational tool for structuring and optimizing neural networks. Computation graphs represent the flow of computations through a network, from input data through intermediate hidden layers to the final output. This framework is crucial for understanding how to construct and train deep learning models efficiently.

Input Encoding

The input layer of a computation graph represents the raw data fed into the network. This data must be encoded appropriately to facilitate effective processing by the network's hidden and output layers.

Input Encoding

- **Factored Data**: For structured data with clear attributes (e.g., numerical or Boolean), each attribute corresponds to an input node. Numeric attributes may be scaled or normalized.

- **Image Data**: Input nodes correspond to pixels or groups of pixels. For RGB images, each pixel may be represented by three values corresponding to red, green, and blue channels.
- **Categorical Attributes**: Use one-hot encoding for attributes with multiple categories. For instance, an attribute with four categories may be encoded with four input nodes, with only one active node representing the present category.

Output Layers and Loss Functions

Output layers are responsible for producing the final predictions of the network. The choice of output layer and loss function depends on the type of task, such as classification or regression.

Output Layers and Loss Functions

- **Sigmoid Output Layer**: Suitable for binary classification tasks. The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ outputs probabilities.
- **Softmax Output Layer**: Used for multiclass classification, where each output node corresponds to a class, and the softmax function ensures the outputs form a probability distribution:

$$\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

- **Linear Output Layer**: Used for regression tasks, where the output is a continuous value. Typically, the output $\hat{y} = z$ is interpreted as the mean of a Gaussian distribution.
- **Loss Functions**: The choice of loss function depends on the task:
  - **Cross-Entropy Loss**: Common for classification, measures the difference between the predicted probability distribution and the true distribution.
  - **Mean Squared Error (MSE)**: Used in regression tasks, it measures the average squared difference between predicted and true values.

Hidden Layers

Hidden layers in a neural network are responsible for transforming the input data into more abstract representations, facilitating the learning of complex patterns and features.

Hidden Layers

- **Nonlinear Transformations**: Each hidden unit applies a nonlinear activation function to a weighted sum of its inputs, allowing the network to capture complex relationships.
- **Common Activation Functions**: Include ReLU ($\text{ReLU}(z) = \max(0, z)$), sigmoid, and tanh ($\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$).
- **Role of Depth**: Deeper networks (more hidden layers) can learn more complex functions but are harder to train due to issues like the vanishing gradient problem.

Summary of Key Concepts

- **Computation Graphs**: Represent the flow of data and computations through a neural network, crucial for both training and inference.
- **Input Encoding**: Proper encoding of input data is essential for effective learning.
- **Output Layers and Loss Functions**: The choice depends on the task (classification vs. regression) and the nature of the output.
- **Hidden Layers**: Transform inputs into higher-level features, enabling the network to learn complex patterns.
- **Activation Functions**: Introduce non-linearity into the network, essential for capturing complex relationships in the data.

Understanding computation graphs and their components is fundamental for designing and training deep learning models, ensuring they can efficiently learn from data and generalize well to new inputs.

The next topic that is being covered from this chapter this week is **Section 21.3: Convolutional Networks**.

Chapter 21.3: Convolutional Networks

Overview

This section introduces convolutional neural networks (CNNs), a class of deep learning models particularly effective for processing data with a known grid-like topology, such as images. CNNs leverage the spatial structure of the data through convolutional layers, which apply filters to local patches of the input to detect features, and pooling layers, which downsample the spatial dimensions.

Convolutional Layers

Convolutional layers are the core building blocks of CNNs. They apply a set of learnable filters (or kernels) to the input data, generating feature maps that capture various aspects of the data, such as edges, textures, or more complex patterns.

Convolutional Layers

- **Filters (Kernels)**: Small, learnable matrices applied across the input to produce a feature map. Each filter detects specific patterns, and the same filter is applied to all positions in the input.
- **Convolution Operation**: The convolution of a filter k with an input x is defined as:

$$z_i = (x * k)_i = \sum_{j=1}^{l} k_j x_{i+j-(l+1)/2}$$

  where l is the size of the filter.
- **Stride and Padding**: Stride controls the step size of the filter as it moves across the input. Padding can be added to control the spatial size of the output, allowing the filter to be applied to the border regions.

Pooling and Downsampling

Pooling layers reduce the spatial dimensions of the feature maps, helping to manage the computational load and control overfitting. They summarize the outputs of local patches by taking the maximum (max-pooling) or average (average-pooling) of the values.

Pooling and Downsampling

- **Max-Pooling**: Takes the maximum value from each patch of the feature map, highlighting the most salient features detected by the convolutional filters.
- **Average-Pooling**: Computes the average value of each patch, providing a smoother downsampled feature map.
- **Downsampling Effect**: Reduces the spatial dimensions of the feature maps, e.g., an input of size $n \times n$ with a stride s results in an output of size $\frac{n}{s} \times \frac{n}{s}$.

Kernels and Convolution Operation

Kernels (or filters) are critical in CNNs for detecting local features. The convolution operation involves sliding these kernels over the input data to produce feature maps.

Kernels and Convolution Operation

- **Convolution Formula**: The feature map z is calculated by:

$$z_i = (x * k)_i = \sum_{j=1}^{l} k_j x_{i+j-(l+1)/2}$$

  where l is the kernel size.

- **Stride and Padding**: Stride defines the step size for moving the filter, and padding allows the filter to cover border regions of the input.
- **Multiple Kernels**: A convolutional layer typically uses multiple kernels to extract different types of features from the input.

Residual Networks

Residual networks (ResNets) address the vanishing gradient problem in very deep networks by introducing shortcut connections that allow gradients to bypass certain layers, facilitating more stable and efficient training.

Residual Networks

- **Residual Blocks**: Introduce an identity shortcut connection that bypasses one or more layers:

$$z^{(i)} = g_r(z^{(i-1)} + f(z^{(i-1)}))$$

  where g_r is the activation function and f represents the residual mapping.

- **Benefits**: Enable the construction of much deeper networks by mitigating the vanishing gradient problem and allowing for better gradient flow through the network.

Summary of Key Concepts

- **Convolutional Layers**: Use learnable filters to detect features in data, with mechanisms like stride and padding controlling output size.
- **Pooling Layers**: Downsample the feature maps, summarizing the presence of features and reducing dimensionality.
- **Kernels and Convolution**: Core operations in CNNs that extract local patterns from the input data.
- **Residual Networks**: Introduce shortcut connections to facilitate training of deeper networks by improving gradient flow.

Convolutional networks are fundamental in processing and extracting hierarchical features from grid-like data, such as images, enabling a wide range of applications in computer vision and beyond.

The next topic that is being covered from this chapter this week is **Section 21.4: Learning Algorithms**.

Section 21.4: Learning Algorithms

Overview

This section discusses the algorithms used for training neural networks, with a focus on optimizing the network's parameters to minimize a loss function. The primary technique discussed is stochastic gradient descent (SGD) and its variants, which are widely used due to their efficiency and effectiveness in handling large datasets and high-dimensional parameter spaces.

Stochastic Gradient Descent (SGD)

SGD is a popular optimization method for training neural networks. Unlike standard gradient descent, which uses the entire training set to compute the gradient, SGD updates the model parameters using a randomly selected subset of the training data (minibatch).

Stochastic Gradient Descent (SGD)

- **Gradient Update Rule**: The parameters w are updated using the gradient of the loss function L with respect to w:
$$w \leftarrow w - \alpha \nabla_w L(w)$$
where α is the learning rate.
- **Minibatch Size**: A small subset m of training examples is used at each step, making the algorithm computationally efficient and helping to escape local minima.
- **Advantages**: SGD is particularly effective for large-scale datasets and can leverage parallelism on hardware like GPUs and TPUs.

Computing Gradients in Computation Graphs

The backpropagation algorithm is used to compute the gradient of the loss function with respect to each parameter in the network, propagating the gradient information from the output layer back through the network.

Computing Gradients in Computation Graphs

- **Backpropagation**: Involves calculating the gradient of the loss L with respect to the output of each node, and then using the chain rule to compute the gradient with respect to the weights:
$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial h_j} + \frac{\partial L}{\partial h_k}$$
where h influences the loss through multiple paths.
- **Node Derivatives**: For a node h with inputs from nodes f and g:
$$\frac{\partial L}{\partial f_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial f_h}$$
$$\frac{\partial L}{\partial g_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial g_h}$$

Batch Normalization

Batch normalization is a technique to improve the training efficiency and stability of neural networks by normalizing the inputs to each layer. It mitigates issues like vanishing and exploding gradients and accelerates convergence.

Batch Normalization

- **Normalization**: The outputs of each layer are normalized using the mean μ and standard deviation σ of the inputs within the minibatch:
$$\hat{z}_i = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
where ϵ is a small constant to prevent division by zero.
- **Learnable Parameters**: Includes scaling factor γ and shift β that are learned during training:
$$z_i^{\text{normalized}} = \gamma \hat{z}_i + \beta$$
- **Benefits**: Helps in maintaining a stable distribution of activations across layers, reducing sensitivity to initialization and accelerating the training process.

Summary of Key Concepts

- **Stochastic Gradient Descent (SGD)**: An efficient optimization algorithm for large-scale datasets, using minibatches to update model parameters.
- **Backpropagation**: A method for computing gradients in neural networks, essential for training via gradient descent.
- **Batch Normalization**: Improves training stability and convergence by normalizing layer inputs.

Understanding these learning algorithms and techniques is crucial for effectively training deep neural networks, ensuring they can learn complex patterns from data and generalize well to new inputs.

The last section that is being covered from this chapter this week is **Section 21.5: Generalization**.

Section 21.5: Generalization

Overview

This section explores the concept of generalization in neural networks, focusing on techniques to improve a model's ability to perform well on unseen data. Generalization is crucial in machine learning, as it determines how well a model trained on a finite dataset will perform on new, previously unseen examples.

Choosing a Network Architecture

Selecting an appropriate network architecture is key to achieving good generalization. Different types of data and tasks may require different architectures, such as convolutional networks for images or recurrent networks for sequential data.

Choosing a Network Architecture

- **Task-Specific Architectures**: Convolutional neural networks (CNNs) are well-suited for image data, while recurrent neural networks (RNNs) are ideal for sequential data like text or speech.
- **Depth and Complexity**: Generally, deeper networks with more layers can capture more complex patterns, but they also risk overfitting if not managed properly.
- **Empirical Findings**: Deeper networks with a similar number of weights typically generalize better than shallower ones. For instance, an eleven-layer network tends to have lower test-set error than a three-layer network for the same task and number of weights.

Weight Decay

Weight decay, a form of regularization, is used to prevent overfitting by penalizing large weights in the network. It helps in controlling the model complexity and thus improves generalization.

Weight Decay

- **Regularization Term**: The weight decay adds a penalty term to the loss function:

$$\text{Cost}(w) = \text{EmpLoss}(w) + \lambda \sum_{i,j} w_{i,j}^2$$

  where λ controls the strength of the penalty.
- **Effect**: Encourages the network to use smaller weights, which helps in avoiding overfitting and keeping the model simpler.
- **Interpretation**: Can be seen as implementing a prior on the weights, favoring smaller values under a Gaussian distribution.

Dropout

Dropout is a technique to prevent overfitting by randomly deactivating a subset of units in the network during training. This forces the network to learn redundant representations, making it more robust.

Dropout

- **Mechanism**: During training, each unit (hidden or input) is kept with a probability p and dropped with a probability $1-p$:
$$\hat{z}_i = \frac{z_i}{p} \text{ if unit } i \text{ is kept, else } 0$$
- **Ensemble Learning Analogy**: Dropout can be thought of as training a large number of "thinned" networks with shared weights, effectively creating an ensemble of networks.
- **Benefits**: Helps in preventing co-adaptation of units, ensuring that the network does not rely too heavily on any single feature.

Neural Architecture Search

Neural architecture search (NAS) involves using algorithms to automate the process of finding the best neural network architecture for a given task. This process explores different configurations, including depth, width, and connectivity, to find architectures that generalize well.

Neural Architecture Search

- **Techniques**: Includes methods like evolutionary algorithms, reinforcement learning, and gradient-based optimization to explore the space of possible architectures.
- **Challenges**: Evaluating each architecture can be computationally expensive, often requiring full training runs, which makes this process resource-intensive.
- **Evaluation Methods**: Approaches to reduce the cost include training on smaller datasets or using proxy models to estimate performance.

Summary of Key Concepts

- **Network Architecture**: Selecting an appropriate architecture is crucial for good generalization.
- **Weight Decay**: Regularizes the network by penalizing large weights, helping to control model complexity.
- **Dropout**: A regularization technique that improves robustness by preventing units from relying too heavily on any single feature.
- **Neural Architecture Search**: Automated methods for finding the best neural network architecture for a specific task.

Understanding these techniques is essential for developing neural networks that generalize well to new data, providing robust and reliable predictions in various applications.

The next chapter that is being covered this week is **Chapter 25: Computer Vision**. The first section that is being covered from this chapter this week is **Section 25.1: Introduction**.

Section 25.1: Introduction

Overview

This section introduces the field of computer vision, which involves connecting computers to the physical world through visual data. Vision, as a perceptual channel, provides valuable information about the environment, enabling agents to make predictions and decisions. The section discusses both passive and active sensing and introduces key concepts in computer vision, including feature extraction, model-based approaches, and the core problems of reconstruction and recognition.

Vision as a Perceptual Channel

Vision allows agents to perceive and interpret their surroundings, offering significant advantages despite the associated costs of maintaining visual systems. Vision enables agents to navigate, identify objects, and predict future events based on visual cues.

Vision as a Perceptual Channel

- **Passive Sensing**: Most vision systems use passive sensing, which relies on ambient light without emitting signals. This includes natural vision systems in animals and standard cameras.
- **Active Sensing**: Involves emitting signals like ultrasound, radar, or light and interpreting the reflections. Examples include echolocation in bats and sonar in dolphins.
- **Feature Extraction**: Simple computations applied to an image to derive useful information, such as edge detection or color segmentation.
- **Examples**: Animals use vision to avoid obstacles, assess threats, find food, and interact with others.

Model-Based Approaches to Vision

Model-based approaches in vision involve using predefined models to interpret visual data. These models can describe the geometric, physical, and statistical properties of objects and scenes.

Model-Based Approaches to Vision

- **Object Models**: Precise geometric models, like those used in computer-aided design (CAD), or general descriptions, such as the typical appearance of faces at low resolution.
- **Rendering Models**: Describe how images are formed based on the interaction of light with objects, accounting for factors like lighting, perspective, and material properties.
- **Ambiguities in Vision**: Visual perception can be ambiguous; for instance, a small object nearby may look similar to a larger distant object, or lighting can alter the perceived color of objects.
- **Managing Ambiguities**: Techniques include leveraging prior knowledge (e.g., there are no real Godzillas) and focusing on relevant details while ignoring insignificant ones.

Core Problems of Computer Vision

The two fundamental problems in computer vision are reconstruction and recognition. Reconstruction involves building a model of the environment from visual data, while recognition entails identifying and distinguishing between objects.

Core Problems of Computer Vision

- **Reconstruction**: Creating a geometric or texture-based model of the scene from one or multiple images. This includes tasks like 3D reconstruction and depth estimation.
- **Recognition**: Involves classifying objects, identifying their state, or understanding their attributes (e.g., determining if an object is animate or inanimate, or identifying specific objects like faces or animals).
- **Examples**: Identifying whether a visual input depicts a real scene or a toy model, or distinguishing between different types of objects in an image.

Summary of Key Concepts

- **Vision Systems**: Enable agents to perceive and interpret their environment, crucial for navigation and interaction.
- **Passive vs. Active Sensing**: Different approaches to capturing visual data, each with its applications

and limitations.

- **Model-Based Vision**: Uses predefined models to interpret visual data, accounting for ambiguities and variations in appearance.
- **Core Problems**: Focus on reconstruction and recognition, essential for applications like autonomous vehicles, robotics, and image analysis.

Understanding these foundational concepts in computer vision is critical for developing systems that can interpret and act upon visual data, enabling a wide range of applications from robotics to medical imaging.

The next section that is being covered from this chapter this week is **Section 25.2: Image Formation**.

Section 25.2: Image Formation

Overview

This section explains the fundamental concepts and physical principles behind image formation, a critical aspect of computer vision. Understanding how images are formed and the factors that influence their appearance is essential for developing models that can accurately interpret visual data. The section covers different imaging systems, geometric distortions, and the role of light and shading in image formation.

Images without Lenses: The Pinhole Camera

A pinhole camera represents the simplest model of image formation, using a small aperture to project light onto an image plane. This system helps in understanding the basic principles of how images are captured.

Images without Lenses: The Pinhole Camera

- **Pinhole Camera**: Consists of a small aperture (pinhole) that allows light from a scene to form an inverted image on the opposite side of a dark box.
- **Geometric Model**: The pinhole camera can be described using a 3D coordinate system with the origin at the pinhole. The projection of a point P with coordinates (X, Y, Z) to the image plane with coordinates (x, y) is given by:
$$x = -f\frac{X}{Z}, \quad y = -f\frac{Y}{Z}$$
where f is the focal length, the distance from the pinhole to the image plane.
- **Perspective Projection**: This process means that the image size decreases with increasing distance from the camera, causing distant objects to appear smaller.

Lens Systems

While pinhole cameras provide a basic understanding, real-world imaging systems typically use lenses to gather more light and focus it more accurately.

Lens Systems

- **Function of Lenses**: Lenses collect light over a larger area than a pinhole, focusing it to a point on the image plane. This increases image brightness and reduces noise.
- **Focal Plane and Depth of Field**: The focal plane is the specific distance at which objects are in sharp focus. The depth of field refers to the range within which objects appear acceptably sharp. It is influenced by the aperture size and lens properties.
- **Aperture and Light Gathering**: A larger aperture allows more light to enter, making the image brighter, but it also reduces the depth of field, requiring more precise focusing.

Scaled Orthographic Projection

In certain cases, perspective distortions are minimal, and a simplified model called scaled orthographic projection can be used. This model is particularly useful for objects at a similar distance from the camera.

Scaled Orthographic Projection

- **Assumption**: This model assumes that the depth Z of all points on an object lies within a small range $Z_0 \pm \Delta Z$ relative to the distance Z_0, allowing the perspective scaling factor f/Z to be approximated as constant.
- **Projection Equations**: The equations for projection from scene coordinates (X, Y, Z) to the image plane become:

$$x = sX, \quad y = sY$$

  where $s = f/Z_0$.
- **Application**: Useful in scenarios where objects have minimal depth variation, making the perspective effects negligible.

Light and Shading

The brightness of an image is influenced by the lighting conditions and the properties of the surfaces in the scene. Understanding these effects is crucial for interpreting image content accurately.

Light and Shading

- **Diffuse Reflection**: Light is scattered evenly across all directions from a surface, with brightness depending on the angle of incidence. Most surfaces in everyday life, such as cloth, wood, and unpolished stones, exhibit diffuse reflection.
- **Specular Reflection**: Light reflects in a specific direction, creating bright spots known as specular highlights. Surfaces like metal, water, and glossy paint exhibit specular reflection.
- **Lambert's Cosine Law**: Describes the brightness I of a diffuse surface as:

$$I = \rho I_0 \cos\theta$$

  where ρ is the diffuse albedo, I_0 is the light source intensity, and θ is the angle between the light direction and the surface normal.

Summary of Key Concepts

- **Image Formation Models**: Include the pinhole camera and lens systems, each with specific characteristics and applications.
- **Perspective and Orthographic Projection**: Different models for projecting 3D scenes onto 2D planes, each with unique distortion properties.
- **Light and Shading**: Crucial for understanding the appearance of objects, influenced by the type of reflection and the lighting conditions.

A thorough understanding of these principles is essential for developing accurate models in computer vision, enabling effective interpretation and analysis of visual data.

The next section that is being covered from this chapter this week is **Section 25.3: Simple Image Features**.

Section 25.3: Simple Image Features

Overview

This section covers the extraction and utilization of simple image features in computer vision. These features include edges, texture, optical flow, and segmentation into regions. Simple image features provide a foundational level of analysis, allowing for the abstraction of complex visual data into manageable and interpretable components.

Edges

Edges are fundamental features in images, representing significant changes in intensity. They are crucial for understanding the structure of a scene and are often the first step in image analysis.

Edges

- **Definition**: Edges are lines or curves in the image where there is a noticeable change in brightness, marking boundaries between different regions.
- **Types of Edges**: Include depth discontinuities, surface orientation changes, reflectance changes, and illumination changes (shadows).
- **Edge Detection**: Involves differentiating the image to find points with large gradients, typically after applying a smoothing function like a Gaussian filter to reduce noise:

$$\text{Gradient} = \nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)$$

- **Noise and Smoothing**: Gaussian smoothing helps in reducing noise and enhancing the true edges:

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

- **Edge Linking**: After detecting edges, the next step is to link these edges to form continuous contours, aiding in the delineation of object boundaries.

Texture

Texture describes the visual patterns on a surface, which can range from regular and repetitive to random and irregular. It is a higher-level feature that helps in recognizing objects and materials.

Texture

- **Definition**: Texture refers to the repeated patterns or statistical properties in an image region, such as the grain on wood or the weave in fabric.
- **Texels**: The fundamental unit of texture, similar to pixels in an image. Texels can represent repetitive elements within a texture.
- **Texture Descriptors**: Methods like histogram of gradient orientations capture the essence of texture, providing invariance to illumination changes:

$$\text{Histogram of Orientations}$$

- **Applications**: Used in object recognition, scene classification, and image segmentation, as textures often differentiate between different objects or surfaces.

Optical Flow

Optical flow refers to the apparent motion of objects in the visual field, caused by the relative motion between the observer and the scene. It is a critical feature for understanding dynamics in a scene.

Optical Flow

- **Definition**: Optical flow describes the motion of pixels in an image sequence, representing the velocity and direction of movement.

- **Computation**: Typically involves tracking the displacement of pixel intensities between frames:

$$(v_x, v_y) = \left(\frac{D_x}{\Delta t}, \frac{D_y}{\Delta t}\right)$$

- **Applications**: Used in motion detection, video stabilization, and understanding scene dynamics. It can indicate object movement, camera motion, or changes in the environment.

Segmentation of Natural Images

Segmentation is the process of partitioning an image into meaningful regions, often corresponding to objects or surfaces. This process simplifies the image analysis by focusing on larger units rather than individual pixels.

Segmentation of Natural Images

- **Definition**: Dividing an image into segments based on criteria like color, brightness, or texture to identify distinct regions.
- **Methods**: Includes edge-based segmentation, region-based segmentation, and clustering methods like k-means.
- **Challenges**: Segmentation must handle noise, overlapping regions, and varying object appearances. Often, both local (pixel-level) and global (object-level) information is used.

Summary of Key Concepts

- **Edges**: Key indicators of structural boundaries in images, essential for shape analysis.
- **Texture**: Provides detailed information about the surface properties of objects.
- **Optical Flow**: Captures motion information, critical for understanding dynamics in video sequences.
- **Segmentation**: Essential for dividing images into meaningful regions, facilitating higher-level analysis.

These simple image features form the building blocks for more advanced computer vision techniques, enabling detailed analysis and interpretation of visual data.

The next section that is being covered from this chapter this week is **Section 25.4: Classifying Images**.

Section 25.4: Classifying Images

Overview

This section discusses the techniques and challenges associated with image classification, a fundamental task in computer vision. Image classification involves assigning a label to an image based on its content, which can include identifying objects, scenes, or specific attributes. The section covers the role of convolutional neural networks (CNNs) in achieving state-of-the-art results and addresses various challenges such as appearance variation and context.

Image Classification Challenges

Classifying images accurately can be challenging due to variations in appearance caused by lighting, foreshortening, aspect changes, occlusion, and deformation. These factors can make the same object appear differently in different images.

Image Classification Challenges

- **Lighting**: Changes in lighting can alter the brightness and color of the image, affecting the appearance of objects.
- **Foreshortening**: When a pattern is viewed at a glancing angle, it becomes distorted, as seen with objects like circular patches appearing elliptical.
- **Aspect Changes**: Different viewing angles can significantly alter the shape of objects, such as a doughnut appearing as an oval from the side and as an annulus from above.
- **Occlusion**: Parts of an object may be hidden by other objects or by itself (self-occlusion), complicating identification.
- **Deformation**: Objects that change shape, like humans or animals moving, add complexity to classification tasks.

Image Classification with Convolutional Neural Networks (CNNs)

CNNs have proven to be highly effective for image classification tasks, largely due to their ability to learn relevant features directly from data. They excel in handling the challenges posed by variations in appearance.

Image Classification with Convolutional Neural Networks (CNNs)

- **Architecture**: CNNs consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Each layer extracts increasingly abstract features from the input.
- **Training on Large Datasets**: The availability of large datasets like ImageNet, containing millions of labeled images, has been crucial for training effective CNNs. These datasets provide diverse examples that help networks learn to generalize well.
- **ImageNet and Performance**: ImageNet's large-scale dataset has facilitated the development of highly accurate classifiers, with CNNs achieving top-5 accuracies of over 98%, surpassing human performance in some categories.
- **Learning Features**: Unlike traditional methods that rely on hand-crafted features, CNNs learn features directly from the data, ensuring they are well-suited for the classification task.

Why CNNs Classify Images Well

CNNs' ability to classify images effectively stems from their architecture, which is adept at capturing spatial hierarchies and local patterns. Data augmentation and the hierarchical structure of CNNs contribute to their robustness and accuracy.

Why CNNs Classify Images Well

- **Hierarchical Feature Learning**: CNNs learn to detect simple patterns in early layers and combine them into complex features in deeper layers. This hierarchical learning allows the network to capture intricate details and contextual relationships.
- **Data Augmentation**: Techniques like random shifting, rotating, and flipping of training images help CNNs become invariant to transformations, improving their robustness.
- **Context Sensitivity**: CNNs can learn to ignore irrelevant background information and focus on discriminative features, even when only a small portion of the image contains the object of interest.

Summary of Key Concepts

- **Image Classification Challenges**: Variations in lighting, perspective, occlusion, and object deformation make classification complex.
- **Convolutional Neural Networks**: CNNs are powerful tools for image classification, learning features directly from large datasets.
- **Hierarchical Learning and Data Augmentation**: Key factors that contribute to the robustness and high performance of CNNs in image classification tasks.

These concepts are fundamental to understanding and developing effective image classification systems, capable of accurately identifying and categorizing objects and scenes in diverse visual data.

The next section that is being covered from this chapter this week is **Section 25.5: Detecting Objects**.

Section 25.5: Detecting Objects

Overview

This section explores object detection in images, a critical task in computer vision that involves not only identifying objects but also localizing them within the image. Object detection systems output bounding boxes around detected objects and classify them into predefined categories. The section covers key concepts such as the sliding window approach, regional proposal networks, and advanced techniques like Faster R-CNN.

Image Classification vs. Object Detection

While image classifiers label an entire image with a single class, object detectors locate and classify multiple objects within the same image. This distinction requires object detectors to manage both classification and localization tasks.

Image Classification vs. Object Detection

- **Bounding Boxes**: Object detectors use bounding boxes to specify the location of each detected object within an image.
- **Sliding Window Approach**: A method where a classifier scans the image with a fixed-size window, classifying the contents of each window. This approach, however, can be computationally intensive due to the large number of possible window positions and sizes.

Regional Proposal Networks (RPNs) and Faster R-CNN

The Faster R-CNN architecture improves object detection efficiency by using a Regional Proposal Network (RPN) to suggest potential object locations, which are then refined and classified.

Regional Proposal Networks (RPNs) and Faster R-CNN

- **Regional Proposal Network (RPN)**: Generates a set of bounding boxes, known as anchor boxes, that likely contain objects. The RPN is trained to distinguish between object and non-object regions.
- **Anchor Boxes**: Predefined boxes of various sizes and aspect ratios that the network evaluates at different positions within the image. Each anchor box is scored based on its "objectness," or likelihood of containing an object.
- **ROI Pooling**: Region of Interest (ROI) pooling is used to standardize the size of feature maps extracted from these boxes, allowing the subsequent classifier to process them.
- **Non-Maximum Suppression**: A technique used to filter out multiple detections of the same object by keeping only the bounding box with the highest score for each object and discarding the others that overlap significantly.
- **Bounding Box Regression**: Refines the location and size of the bounding boxes to more accurately match the objects, correcting any initial inaccuracies from the RPN.

Evaluation of Object Detectors

Evaluating the performance of object detectors involves comparing their predictions with ground truth annotations. Key metrics include precision, recall, and intersection over union (IoU).

Evaluation of Object Detectors

- **Ground Truth Annotations**: Human-provided labels that include the category and precise location of objects within the image, used as the standard for evaluating detector performance.
- **Precision and Recall**: Precision measures the proportion of correct positive detections, while recall measures the proportion of true objects detected by the system.
- **Intersection over Union (IoU)**: A metric used to assess the overlap between predicted bounding boxes and ground truth boxes. It is defined as the ratio of the area of intersection to the area of union between the predicted and ground truth boxes.
- **Non-Maximum Suppression (NMS)**: A post-processing step to remove duplicate detections and ensure that each object is reported once, using a threshold on IoU to merge overlapping boxes.

Summary of Key Concepts

- **Object Detection**: Involves both classification and localization of multiple objects within an image.
- **Faster R-CNN and RPNs**: Advanced frameworks for efficient and accurate object detection, utilizing regional proposal networks and ROI pooling.
- **Bounding Box Regression and Non-Maximum Suppression**: Techniques for refining object localization and removing redundant detections.
- **Evaluation Metrics**: Precision, recall, and IoU are essential metrics for assessing the performance of object detectors.

These concepts are crucial for developing robust object detection systems capable of accurately identifying and localizing objects in complex scenes.

The next section that is being covered from this chapter this week is **Section 25.6: The 3D World**.

Section 25.6: The 3D World

Overview

This section delves into how images, which are inherently two-dimensional, can provide rich information about the three-dimensional (3D) world. By leveraging multiple views and various cues within images, it is possible to reconstruct and interpret the 3D structure of scenes. The section explores methods for extracting 3D cues from multiple views, binocular stereopsis, motion, and single images.

3D Cues from Multiple Views

Multiple images of the same scene taken from different viewpoints can be used to reconstruct the 3D geometry. This process involves matching corresponding points across images and using geometric principles to infer depth information.

3D Cues from Multiple Views

- **Correspondence Problem**: Identifying matching points in different views of the same scene. Features such as texture can aid in matching points across images.
- **Reconstruction Techniques**: By triangulating matched points from different views, it is possible to construct a 3D model of the scene. The precision of this reconstruction improves with the number of views and the diversity of viewpoints.
- **Applications**: This technique is fundamental in fields like photogrammetry, where precise measurements of the 3D structure are essential.

Binocular Stereopsis

Binocular stereopsis is the method by which depth is perceived using the slight difference in images between the left and right eyes. This disparity provides crucial information about the relative distance of objects.

Binocular Stereopsis

- **Disparity**: The difference in position of an object's image on the left and right retinas is known as disparity. The size of this disparity is inversely proportional to the object's distance from the observer.
- **Depth Estimation**: Using the formula:

$$\text{disparity} = \frac{b\delta Z}{Z^2}$$

  where b is the baseline (distance between the eyes), δZ is the change in depth, and Z is the depth.
- **Applications**: Useful in both biological vision systems and artificial systems like stereo cameras, aiding in navigation and interaction with the environment.

3D Cues from a Moving Camera

Similar to stereopsis, moving a single camera provides sequential images from different viewpoints. Analyzing the differences between these images, known as optical flow, can reveal the 3D structure of the scene.

3D Cues from a Moving Camera

- **Optical Flow**: The apparent motion of objects in the image as the camera moves. It provides information about the relative motion and depth of objects.
- **Focus of Expansion (FOE)**: The point in the image where the optical flow vectors converge, indicating the direction of motion. This can be used to deduce the camera's trajectory and relative depth of objects.
- **Depth from Motion**: The speed and direction of object motion in the image plane correlate with their distance from the camera, allowing depth estimation.

3D Cues from One View

Even a single image can provide cues about the 3D structure of the scene. These cues include perspective, shading, texture gradients, and occlusion.

3D Cues from One View

- **Perspective**: The way parallel lines converge in an image provides information about depth and distance.
- **Texture Gradients**: Changes in the size, shape, and density of texture elements can indicate the surface orientation and depth.
- **Shading and Lighting**: The distribution of light and shadow on surfaces can reveal the shape and depth of objects.
- **Occlusion**: When one object overlaps another, it provides a powerful cue about their relative positions in depth.

Summary of Key Concepts

- **3D Cues from Multiple Views**: Techniques like triangulation are used to reconstruct 3D models from multiple images.
- **Binocular Stereopsis**: Utilizes the disparity between left and right eye views to gauge depth.
- **3D Cues from Motion**: Optical flow and focus of expansion provide depth information from moving cameras.
- **Single-View Cues**: Perspective, texture, shading, and occlusion cues are crucial for inferring 3D

structure from a single image.

These principles are fundamental to various applications in computer vision, robotics, and augmented reality, where understanding the 3D structure of the environment is critical.

The last section that is being covered from this chapter this week is **Section 25.7: Using Computer Vision**.

Section 25.7: Using Computer Vision

Overview

This section explores various applications of computer vision, highlighting its transformative impact across numerous fields. It covers areas such as understanding human activities, linking pictures with words, 3D reconstruction from multiple views, geometry from a single view, creating pictures, and controlling movement with vision. These applications showcase the versatility and power of computer vision technologies in interpreting and interacting with the visual world.

Understanding What People Are Doing

Computer vision systems can analyze video data to understand human activities, leading to applications in surveillance, human-computer interaction, and more.

Understanding What People Are Doing

- **Activity Recognition**: Systems can now accurately predict the locations of a person's joints and reconstruct 3D body poses from images, enabling applications such as surveillance and gaming.
- **Behavior Classification**: While structured activities like sports are easier to classify, understanding more general behaviors remains challenging due to varying contexts and appearances.
- **Applications**: Includes public safety, ergonomic assessments in workplaces, and interactive entertainment.

Linking Pictures and Words

Image captioning and tagging systems link visual data with descriptive text, enhancing accessibility and searchability of images.

Linking Pictures and Words

- **Image Tagging**: Uses image classification and object detection techniques to label images with keywords, aiding in organizing and retrieving visual content.
- **Image Captioning**: Combines convolutional networks with sequence models like RNNs or transformers to generate descriptive sentences for images. Challenges include generating contextually accurate and detailed descriptions.
- **Visual Question Answering (VQA)**: Systems answer natural language questions about images, testing their comprehension beyond simple descriptions.

Reconstruction from Many Views

Using multiple images from different viewpoints, it is possible to reconstruct detailed 3D models of scenes and objects.

Reconstruction from Many Views

- **Multi-View Geometry**: Involves matching points across different images to build 3D models, with applications in urban modeling, virtual reality, and movie production.
- **Applications**: Includes creating detailed 3D models from tourist photos, integrating CGI characters in live-action footage, and tracking construction progress with drones.

Geometry from a Single View

Single images can also provide cues for 3D reconstruction, using learned models to estimate depth and geometry.

Geometry from a Single View

- **Depth Maps**: Predicting depth from a single image, useful for understanding room layouts and object placement.
- **Object Pose Estimation**: Using known object models to estimate the pose and shape of objects from single images, extending to complete texture mapping.
- **Applications**: Includes virtual try-ons in fashion, enhancing realism in gaming, and interior design visualization.

Making Pictures

Computer vision also involves creating new visual content, such as inserting objects into scenes or transforming images.

Making Pictures

- **Image Synthesis and Style Transfer**: Techniques like GANs can generate realistic images, alter styles, and even create deepfakes, blending real and synthetic elements seamlessly.
- **Applications**: Ranging from entertainment and art to privacy-preserving data synthesis in medical imaging.

Controlling Movement with Vision

Vision-based control is essential for robotics and autonomous systems, enabling navigation and interaction with the environment.

Controlling Movement with Vision

- **Autonomous Vehicles**: Use vision systems for lane detection, obstacle avoidance, and adherence to traffic signals.
- **Mobile Robotics**: Includes SLAM (Simultaneous Localization and Mapping) and path planning, crucial for applications like automated delivery.

Summary of Key Concepts

- **Activity Recognition**: Understanding human actions through video analysis.
- **Image Captioning and VQA**: Linking visual data with textual descriptions and answers.
- **3D Reconstruction**: Techniques for creating 3D models from multiple or single images.
- **Image Synthesis and Transformation**: Generating new visual content and altering existing images.
- **Vision-Based Control**: Applications in autonomous navigation and robotic manipulation.

These diverse applications highlight the versatility of computer vision technologies and their expanding role in various domains, from entertainment to practical automation.

Deep Reinforcement Learning

Deep Reinforcement Learning

12.0.1 Assigned Reading

The reading for this week is from, Artificial Intelligence - A Modern Approach and Reinforcement Learning - An Introduction.

- **Reinforcement Learning - An Introduction - Chapter 10.1 - Episodic Semi-Gradient Control**
- **Reinforcement Learning - An Introduction - Chapter 11.1 - Semi-Gradient Methods**
- **Reinforcement Learning - An Introduction - Chapter 11.2 - Examples Of Off-Policy Divergence**
- **Reinforcement Learning - An Introduction - Chapter 11.3 - The Deadly Triad**
- **Reinforcement Learning - An Introduction - Chapter 13.1 - Policy Approximation And Its Advantages**
- **Reinforcement Learning - An Introduction - Chapter 13.2 - The Policy Gradient Theorem**
- **Reinforcement Learning - An Introduction - Chapter 13.3 - REINFORCE: Monte Carlo Policy Gradient**
- **Reinforcement Learning - An Introduction - Chapter 13.4 - REINFORCE With Baseline**

12.0.2 Piazza

Must post at least **three** times this week to Piazza.

12.0.3 Lectures

The lectures for this week are:

- RL Review / Deep Reinforcement Learning Intro / DQN $\approx$ 40 **min**.
- Policy Gradient Method $\approx$ 33 **min**.

The lecture notes for this week are:

- Deep Reinforcement Learning Intro Lecture Notes
- Policy Gradient Lecture Notes

12.0.4 Project

The final project for this course can be found below:

- Final Project

12.0.5 Exam

The exam for this week is:

- Makeup Exam

12.0.6 Chapter Summary

The chapters that are being covered this week are **Chapter 10: Knowledge Representation**, **Chapter 11: Automated Planning**, and **Chapter 13: Probabilistic Reasoning**. The section that is covered from **Chapter 10: Knowledge Representation** is **Section 10.1: Episodic Semi-Gradient Control**.

Section 10.1: Episodic Semi-Gradient Control

Overview

This section introduces episodic semi-gradient control methods, particularly focusing on the application of the semi-gradient Sarsa algorithm in on-policy control tasks. These methods extend value function approximation techniques to control problems, where the goal is to learn a policy that maximizes expected returns in an episodic setting. The section covers the algorithmic details and provides an example application in the Mountain Car task.

Semi-gradient Sarsa for Control

Semi-gradient Sarsa is an on-policy control algorithm that updates action-value function estimates using semi-gradient methods. The algorithm iteratively improves the policy by coupling action-value predictions with policy improvement techniques, such as -greedy action selection.

Semi-gradient Sarsa for Control

- **Action-value Function Update**: The weights w of the action-value function $\hat{q}(S, A, w)$ are updated at each time step using the following rule:

$$w_{t+1} = w_t + \alpha \left[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t) \right] \nabla_w \hat{q}(S_t, A_t, w_t)$$

  where α is the step size, γ is the discount factor, and $\nabla_w \hat{q}$ denotes the gradient with respect to the weights.

- **Policy Improvement**: The policy is improved by selecting actions that maximize the estimated action-value function. In practice, a soft policy such as -greedy is used to balance exploration and exploitation.

- **Algorithm Outline**: The semi-gradient Sarsa algorithm alternates between evaluating the current policy and improving it by updating the weights based on the observed rewards and state transitions.

Example: Mountain Car Task

The Mountain Car task is a classic continuous control problem where an underpowered car must build up momentum to reach the top of a steep hill. The task demonstrates the challenges of continuous control and the effectiveness of semi-gradient Sarsa in such environments.

Example: Mountain Car Task

- **Task Description**: The car must first move away from the goal to gain enough speed to ascend the hill. The reward is -1 per time step until the car reaches the goal, at which point the episode ends.

- **State and Action Representation**: The state is represented by the car's position and velocity, and the actions include full throttle forward, full throttle reverse, and zero throttle.

- **Function Approximation**: The action-value function is approximated using tile coding, where the continuous state space is discretized into binary features. The function approximation is linear, with weights corresponding to the tiles.

- **Learning Dynamics**: During learning, the agent initially explores the state space extensively, driven by the optimistic initialization of action values. Over time, the agent converges to a policy that efficiently solves the task.

Algorithm Performance

The performance of semi-gradient Sarsa in the Mountain Car task is illustrated through learning curves, showing how the number of steps per episode decreases as the agent learns the optimal policy.

Algorithm Performance

- **Learning Curves**: The curves show the number of steps required per episode as the agent improves its policy. Various step sizes α affect the speed and stability of learning.
- **Exploration Strategy**: The -greedy action selection strategy ensures sufficient exploration during the initial phases of learning, while gradually converging to a near-optimal policy.
- **Optimization**: The choice of function approximation method and hyperparameters such as the step size significantly impacts the algorithm's performance and convergence rate.

Summary of Key Concepts

- **Semi-gradient Sarsa**: An on-policy control algorithm that combines value function approximation with policy improvement techniques.
- **Episodic Control Tasks**: Suitable for tasks where the agent learns a policy to maximize returns over episodes, such as the Mountain Car task.
- **Function Approximation**: Essential for handling continuous state spaces, often implemented using methods like tile coding.
- **Learning Performance**: Influenced by factors such as exploration strategy, step size, and the choice of function approximation.

Understanding these concepts is crucial for developing and applying reinforcement learning algorithms in continuous control tasks, where the state and action spaces are large or continuous.

The first section that is covered from **Chapter 11: Automated Planning** this week is **Section 11.1: Semi-Gradient Methods**.

Section 11.1: Semi-Gradient Methods

Overview

This section introduces semi-gradient methods in the context of off-policy learning with function approximation. Semi-gradient methods extend the off-policy algorithms discussed in earlier chapters to handle function approximation by updating a weight vector rather than a table of values. The section explains how these methods work, their challenges, and the potential for divergence in certain cases.

Off-policy Learning with Function Approximation

Off-policy learning involves learning a target policy different from the behavior policy that generates the data. When combined with function approximation, this approach faces the challenge of unstable updates due to the mismatch between the update distribution and the on-policy distribution.

Off-policy Learning with Function Approximation

- **Update Distribution**: The distribution of updates in off-policy learning does not match the on-policy distribution, which is critical for the stability of semi-gradient methods.
- **Importance Sampling**: One approach to address this issue is to use importance sampling, which re-weights the updates to match the on-policy distribution. However, this can lead to high variance in the updates.
- **True Gradient Methods**: Another approach is to develop true gradient methods that do not rely on the on-policy distribution for stability, though this area remains an active research topic with ongoing developments.

Semi-gradient Methods for Off-policy Learning

Semi-gradient methods extend tabular off-policy algorithms to function approximation. The key idea is to update a weight vector w rather than a value table, using the approximate value function $\hat{v}$ or $\hat{q}$ and their gradients.

Semi-gradient Methods for Off-policy Learning

- **Per-step Importance Sampling**: The per-step importance sampling ratio ρ_t is used to adjust the updates in off-policy learning:
$$\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$
where π is the target policy and b is the behavior policy.
- **Semi-gradient TD(0)**: For state-value functions, the semi-gradient TD(0) update is:
$$w_{t+1} = w_t + \alpha\rho_t\delta_t\nabla_w\hat{v}(S_t, w_t)$$
where δ_t is the TD error:
$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)$$
- **Semi-gradient Expected Sarsa**: For action-value functions, the semi-gradient Expected Sarsa update is:
$$w_{t+1} = w_t + \alpha\delta_t\nabla_w\hat{q}(S_t, A_t, w_t)$$
with the TD error δ_t defined as:
$$\delta_t = R_{t+1} + \gamma\sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)$$

Multi-step Generalizations and Challenges

The section also covers multi-step versions of these algorithms and highlights the challenges involved, such as the potential for divergence when the update distribution is not properly handled.

Multi-step Generalizations and Challenges

- **n-step Semi-gradient Sarsa**: Extends the one-step Sarsa algorithm to consider returns over multiple time steps. The weight update is adjusted using importance sampling over the multi-step return:
$$w_{t+n} = w_{t+n-1} + \alpha\rho_{t+1}\cdots\rho_{t+n-1}[G_{t:t+n} - \hat{q}(S_t, A_t, w_{t+n-1})]\nabla_w\hat{q}(S_t, A_t, w_{t+n-1})$$
where $G_{t:t+n}$ represents the n-step return.
- **Potential Divergence**: Semi-gradient methods can diverge if the update distribution is not properly managed, particularly in off-policy settings with function approximation.
- **Tree-backup Algorithm**: An alternative multi-step algorithm that avoids importance sampling by backing up action values through a tree of possibilities, ensuring stability.

Summary of Key Concepts

- **Off-policy Learning**: Involves learning a target policy from data generated by a different behavior policy.
- **Semi-gradient Methods**: Extend tabular off-policy algorithms to function approximation by updating a weight vector.
- **Importance Sampling**: Used to adjust the updates to match the on-policy distribution, but can lead to high variance.
- **Potential Challenges**: Includes the risk of divergence in off-policy learning with function approximation.

These methods are foundational for extending off-policy reinforcement learning to more complex environments where function approximation is necessary, though careful management of update distributions is crucial to ensure stability and convergence.

The next section that is covered from this chapter this week is **Section 11.2: Examples Of Off-Policy Divergence**.

Section 11.2: Examples Of Off-Policy Divergence

Overview

This section examines the inherent risks of instability and divergence when applying off-policy learning methods with function approximation. By providing concrete counterexamples, it illustrates how even simple algorithms can become unstable and diverge, particularly in off-policy settings. The section emphasizes the mismatch between the distribution of updates and the on-policy distribution as a key factor leading to these issues.

Counterexamples to Off-policy Learning

Off-policy learning with function approximation can lead to divergence under certain conditions. This is demonstrated through specific examples where the use of semi-gradient methods results in the weight vector diverging to infinity.

Counterexamples to Off-policy Learning

- **Simple Divergence Example**: Consider two states with values represented as w and $2w$. If the first state always transitions to the second with a reward of zero, updates can cause w to diverge because the TD error increases with each transition:

$$\delta_t = (2\gamma - 1)w_t$$

  If $\gamma > 0.5$, the system is unstable, and w increases without bound.

- **Key Insight**: Off-policy learning can result in repeated updates to one state without corresponding updates to another, causing instability. This is less likely in on-policy learning where the distribution of updates is more balanced.

Baird's Counterexample

Baird's counterexample provides a complete and practical demonstration of divergence in off-policy learning. It involves a seven-state, two-action Markov Decision Process (MDP) where a behavior policy and a target policy are defined, leading to instability when semi-gradient TD(0) is applied.

Baird's Counterexample

- **MDP Structure**: The MDP consists of six states that transition uniformly to one terminal state under a behavior policy, while the target policy consistently transitions to a different state. The linear function approximation represents state values as combinations of weights:

$$v(s) = 2w_i + w_8 \quad \text{for } i = 1, \dots, 6$$

- **Divergence Behavior**: When semi-gradient TD(0) is applied with any positive step size, the weights diverge to infinity, demonstrating instability. The divergence occurs even when the expected update (as in dynamic programming) is used:

$$w_{k+1} = w_k + \alpha \sum_s \left(E_\pi \left[R_{t+1} + \gamma v(s_{t+1}, w_k) \mid s_t = s \right] - v(s, w_k) \right) \nabla_w v(s, w_k)$$

- **Implications**: The example shows that instability can arise in simple settings when the update distribution is not aligned with the target policy, and that even expected updates cannot guarantee stability in off-policy settings.

Other Divergence Examples

Other examples, like Tsitsiklis and Van Roy's counterexample, further illustrate that linear function approximation can diverge even when using sophisticated update rules like least-squares.

Other Divergence Examples

- **Tsitsiklis and Van Roy's Counterexample**: Extends the simple two-state example by adding a terminal state. Even when using least-squares updates, the system can diverge if the discount factor γ exceeds a certain threshold.
- **Function Approximation Methods**: Stability is guaranteed only for specific types of function approximation methods that do not extrapolate from observed targets, such as nearest neighbor or locally weighted regression.

Summary of Key Concepts

- **Off-policy Divergence**: Demonstrated through examples where semi-gradient methods lead to instability and weight divergence.
- **Distribution Mismatch**: The key issue in off-policy learning, where the update distribution does not match the on-policy distribution, leading to instability.
- **Baird's Counterexample**: A classic demonstration of divergence in off-policy learning with function approximation.
- **Function Approximation Risks**: Certain function approximation methods are more prone to instability in off-policy settings.

These examples underscore the critical importance of understanding the risks of divergence in off-policy learning and the need for careful management of function approximation methods to ensure stability.

The last section that is covered from this chapter this week is **Section 11.3: The Deadly Triad**.

Section 11.3: The Deadly Triad

Overview

This section introduces the concept of the "Deadly Triad," a combination of three elements that can lead to instability and divergence in reinforcement learning when they are used together. These elements are function approximation, bootstrapping, and off-policy training. The section explores why this combination is problematic and discusses potential approaches to mitigate the risks associated with it.

The Deadly Triad Components

The Deadly Triad consists of three components that, when combined, pose significant risks of instability in reinforcement learning algorithms.

The Deadly Triad Components

- **Function Approximation**: Involves using scalable methods to generalize across a large state space. Common techniques include linear function approximation and artificial neural networks (ANNs). While necessary for handling large-scale problems, function approximation can lead to issues if not managed properly.
- **Bootstrapping**: Refers to updating value estimates based on other estimates rather than directly on rewards or complete returns. Bootstrapping is commonly used in dynamic programming and TD methods but introduces potential instability by relying on existing estimates that may be inaccurate.

- **Off-policy Training**: Occurs when learning is based on a distribution of transitions different from that produced by the target policy. Off-policy training is essential for flexibility in learning but can exacerbate the instability caused by the other two components.

Implications of the Deadly Triad

The combination of these three elements leads to significant challenges in reinforcement learning, particularly in maintaining stability and avoiding divergence.

Implications of the Deadly Triad

- **Instability and Divergence**: When function approximation, bootstrapping, and off-policy training are combined, they can interact in ways that cause the learning process to become unstable. This can lead to divergence, where the estimated values grow without bound, ultimately rendering the algorithm ineffective.
- **Function Approximation**: Cannot be eliminated due to its necessity in scaling to large problems. Alternatives like nonparametric methods or state aggregation are either too weak or computationally expensive for large-scale applications.
- **Bootstrapping vs. Monte Carlo**: While avoiding bootstrapping can prevent instability, it comes at the cost of reduced computational and data efficiency. Bootstrapping allows for more efficient updates and faster learning, particularly in environments with significant state re-visitation.
- **Off-policy Training**: Although on-policy methods like Sarsa can be used as an alternative, off-policy learning is essential for certain applications where learning multiple policies in parallel is required, such as in predictive models of the world used in planning.

Mitigating the Risks of the Deadly Triad

The section suggests that while the Deadly Triad poses significant challenges, careful design choices can mitigate the risks associated with it.

Mitigating the Risks of the Deadly Triad

- **Selective Bootstrapping**: Using longer n-step updates or large bootstrapping parameters can reduce reliance on bootstrapping, minimizing its potential to cause instability.
- **Safe Function Approximation**: Choosing function approximation methods that do not extrapolate from observed targets, such as nearest neighbor methods or locally weighted regression, can enhance stability.
- **Pragmatic Off-policy Learning**: While off-policy learning is critical for certain advanced applications, using it judiciously and ensuring a good overlap between the behavior policy and target policy can help maintain stability.

Summary of Key Concepts

- **The Deadly Triad**: The combination of function approximation, bootstrapping, and off-policy training, which can lead to instability in reinforcement learning.
- **Instability Risks**: Divergence and instability arise when all three elements are used together, particularly in large-scale problems with complex state spaces.
- **Mitigation Strategies**: Involves careful selection of function approximation methods, controlled use of bootstrapping, and judicious application of off-policy learning.

Understanding the Deadly Triad is crucial for designing robust reinforcement learning algorithms that can scale effectively without succumbing to instability.

The first section that is covered from **Chapter 13: Probabilistic Reasoning** this week is **Section 13.1: Policy Approximation And Its Advantages**.

Section 13.1: Policy Approximation And Its Advantages

Overview

This section introduces policy gradient methods, focusing on the advantages of parameterizing policies directly. Unlike action-value methods, policy gradient methods optimize the policy itself, which can be particularly beneficial in handling large or continuous action spaces. The section discusses common parameterizations, particularly for discrete action spaces, and highlights the benefits of these approaches over traditional action-value methods.

Policy Parameterization

In policy gradient methods, the policy is parameterized in a way that allows for differentiation with respect to its parameters. This parameterization must ensure that the policy is differentiable and that it never becomes fully deterministic to guarantee exploration.

Policy Parameterization

- **Differentiable Policy**: The policy $\pi(a|s, \theta)$ is parameterized by θ, ensuring that the gradient $\nabla_\theta \pi(a|s, \theta)$ exists and is finite for all states s and actions a.
- **Soft-max in Action Preferences**: A common parameterization for discrete action spaces is the soft-max distribution over action preferences:

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

  where $h(s, a, \theta)$ represents the preference for action a in state s, parameterized by θ.
- **Action Preferences**: These can be parameterized using a linear combination of features $x(s, a)$, such that:

  $$h(s, a, \theta) = \theta^\top x(s, a)$$

Advantages of Policy Gradient Methods

Policy gradient methods offer several advantages over action-value methods, especially in environments requiring stochastic policies or continuous action spaces.

Advantages of Policy Gradient Methods

- **Stochastic Policies**: Policy gradient methods naturally support stochastic policies, which are often optimal in environments with partial information or where mixed strategies are required (e.g., bluffing in Poker).
- **Deterministic Policies**: The soft-max parameterization allows the policy to approach determinism, unlike ϵ-greedy action selection in action-value methods, where there is always a non-zero probability of selecting a suboptimal action.
- **Simpler Policy Approximation**: In some problems, approximating the policy directly is simpler than approximating the action-value function, leading to faster learning and better performance.
- **Incorporating Prior Knowledge**: Policy parameterization can be a powerful way to inject prior knowledge about the desired policy structure into the learning process, guiding the agent towards more effective behaviors.

Example: Short Corridor with Switched Actions

The section provides an example of a small gridworld with reversed actions in one state, demonstrating the limitations of ϵ-greedy action selection and the advantages of a parameterized policy that can learn a specific action probability.

Example: Short Corridor with Switched Actions

- **Gridworld Description**: The gridworld consists of three states, with actions reversed in the second state. Traditional action-value methods struggle due to the identical feature representation for all states.
- **Optimal Policy**: The optimal policy involves selecting the "right" action with a specific probability (approximately 0.59), which is better captured by a parameterized policy compared to ϵ-greedy action selection.
- **Performance Comparison**: The value achieved by the optimal stochastic policy is significantly better than that of policies derived from ϵ-greedy action selection.

Summary of Key Concepts

- **Policy Gradient Methods**: Focus on optimizing the policy directly, providing advantages in stochastic and continuous action spaces.
- **Soft-max Parameterization**: A common approach for discrete action spaces, allowing for flexible and effective policy representation.
- **Advantages over Action-value Methods**: Include better support for stochastic policies, potential for determinism, and the ability to incorporate prior knowledge.
- **Practical Example**: Demonstrates the practical benefits of policy gradient methods in a challenging gridworld task.

Understanding these advantages is crucial for selecting appropriate methods in reinforcement learning tasks, particularly when dealing with complex environments that require sophisticated policies.

The next section that is covered from this chapter this week is **Section 13.2: The Policy Gradient Theorem**.

Section 13.2: The Policy Gradient Theorem

Overview

This section introduces the Policy Gradient Theorem, a foundational result in reinforcement learning that provides an analytical expression for the gradient of the performance measure with respect to the policy parameters. This theorem is critical for developing and understanding policy gradient methods, as it underpins the gradient ascent algorithms used to optimize policies directly.

Policy Gradient Theorem for Episodic Tasks

The Policy Gradient Theorem is first presented in the context of episodic tasks, where the performance measure $J(\theta)$ is defined as the expected return from a specific start state. The theorem provides a way to compute the gradient of this performance measure, crucial for policy optimization.

Policy Gradient Theorem for Episodic Tasks

- **Performance Measure**: For an episodic task, the performance measure $J(\theta)$ is defined as:
$$J(\theta) = v_{\pi_\theta}(s_0)$$
where $v_{\pi_\theta}(s_0)$ is the value of the start state s_0 under the policy π_θ, parameterized by θ.
- **Gradient of the State-value Function**: The gradient of the state-value function can be expressed using the action-value function:
$$\nabla_\theta v_{\pi_\theta}(s) = \sum_a \nabla_\theta \pi_\theta(a|s) q_{\pi_\theta}(s,a)$$

- **Policy Gradient Theorem**: The theorem states that the gradient of the performance measure with respect to the policy parameters is proportional to:

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a \nabla_\theta \pi_\theta(a|s) q_{\pi_\theta}(s, a)$$

where $\mu(s)$ is the on-policy state distribution, representing the frequency with which state s is visited under the policy π_θ.

Advantages of Policy Gradient Methods

The section also highlights the theoretical advantages of policy gradient methods over traditional action-value methods, particularly in terms of the continuity and smoothness of policy updates.

Advantages of Policy Gradient Methods

- **Continuity of Policy Updates**: Unlike ϵ-greedy action selection, where small changes in estimated action values can lead to abrupt changes in action probabilities, policy gradient methods ensure smooth changes in the policy as the parameters are updated.
- **Convergence Guarantees**: The continuity provided by the smooth changes in policy parameters enables stronger convergence guarantees for policy gradient methods compared to action-value methods.
- **Practical Application**: These advantages make policy gradient methods particularly suitable for environments with continuous or large action spaces, where deterministic or ϵ-greedy methods might struggle.

Implications of the Policy Gradient Theorem

The Policy Gradient Theorem simplifies the process of computing gradients for policy optimization, enabling the development of efficient algorithms that can be applied to a wide range of reinforcement learning tasks.

Implications of the Policy Gradient Theorem

- **Gradient Ascent Optimization**: The theorem directly informs the development of gradient ascent algorithms, where the policy parameters are updated using the gradient of the performance measure.
- **Applicability**: The theorem is applicable to both episodic and continuing tasks, although the performance measure and gradients are defined differently in each case.
- **Foundation for Algorithms**: The Policy Gradient Theorem serves as the theoretical foundation for various policy gradient algorithms, including REINFORCE and actor-critic methods.

Summary of Key Concepts

- **Policy Gradient Theorem**: Provides a method for calculating the gradient of the performance measure with respect to policy parameters, crucial for optimizing policies.
- **Episodic Performance Measure**: Defines the expected return from a start state as the performance measure in episodic tasks.
- **Advantages of Policy Gradient Methods**: Include smooth policy updates and stronger convergence guarantees, making them suitable for complex environments.
- **Theorem's Implications**: Underpin the development of various policy optimization algorithms, guiding the design of effective reinforcement learning methods.

Understanding the Policy Gradient Theorem is essential for leveraging policy gradient methods in reinforcement learning, enabling the optimization of policies in both simple and complex environments.

The next section that is covered from this chapter this week is **Section 13.3: REINFORCE: Monte Carlo Policy Gradient**.

Section 13.3: REINFORCE: Monte Carlo Policy Gradient

Overview

This section introduces the REINFORCE algorithm, one of the simplest and most well-known policy gradient methods. REINFORCE uses Monte Carlo sampling to estimate the policy gradient, enabling the optimization of policies in episodic tasks. The section explains the derivation of the REINFORCE update rule and discusses its properties, including its simplicity, convergence behavior, and potential high variance.

Deriving the REINFORCE Algorithm

The REINFORCE algorithm is derived from the Policy Gradient Theorem, which provides an expression for the gradient of the performance measure with respect to the policy parameters. This gradient is then estimated using Monte Carlo methods, where the return is computed based on complete episodes.

Deriving the REINFORCE Algorithm

- **Policy Gradient Theorem**: The gradient of the performance measure $J(\theta)$ with respect to the policy parameters θ is given by:

$$\nabla_\theta J(\theta) = E_\pi\left[\sum_a q_\pi(S_t, a)\nabla_\theta \pi_\theta(a|S_t)\right]$$

  where $q_\pi(S_t, a)$ is the action-value function under the policy π_θ.

- **Monte Carlo Estimation**: The REINFORCE algorithm uses Monte Carlo sampling to estimate this gradient. The key idea is to replace the action-value function $q_\pi(S_t, a)$ with the return G_t, resulting in:

$$\nabla_\theta J(\theta) \approx E_\pi\left[G_t \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\right]$$

- **REINFORCE Update Rule**: The update rule for the policy parameters at each time step t is:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}$$

  where α is the step size, and G_t is the return from time step t to the end of the episode.

Intuition Behind REINFORCE

The REINFORCE update has an intuitive interpretation: it adjusts the policy parameters in a way that increases the likelihood of actions that lead to high returns while reducing the likelihood of actions that lead to low returns.

Intuition Behind REINFORCE

- **Return-weighted Updates**: The update is proportional to both the return G_t and the gradient of the policy. This ensures that actions leading to higher returns have a larger impact on the policy update.
- **Exploration vs. Exploitation**: The gradient term $\nabla_\theta \pi_\theta(A_t|S_t)/\pi_\theta(A_t|S_t)$ adjusts the policy in the direction that increases the probability of selecting action A_t in state S_t, balancing exploration and exploitation.

Properties and Performance of REINFORCE

The section discusses the performance of REINFORCE, highlighting its theoretical convergence properties as well as its practical limitations, such as high variance.

Properties and Performance of REINFORCE

- **Theoretical Convergence**: As a stochastic gradient method, REINFORCE is guaranteed to converge to a local optimum under certain conditions, provided the step size α is sufficiently small.
- **High Variance**: Since REINFORCE relies on complete episode returns, it can suffer from high variance,

leading to slow learning. This is particularly problematic in environments with long episodes or sparse rewards.

- **Monte Carlo Nature**: REINFORCE is well-suited for episodic tasks where updates are made after the completion of an episode. Its Monte Carlo nature makes it less effective for continuing tasks.

Summary of Key Concepts

- **REINFORCE Algorithm**: A Monte Carlo policy gradient method that uses episode returns to update policy parameters.
- **Gradient Estimation**: REINFORCE estimates the policy gradient using the return from each episode, weighted by the gradient of the policy.
- **High Variance**: While simple and theoretically sound, REINFORCE can suffer from high variance, leading to slow convergence in practice.
- **Application Scope**: Best suited for episodic tasks, particularly when episodes are relatively short and rewards are dense.

Understanding the REINFORCE algorithm is essential for applying Monte Carlo policy gradient methods in reinforcement learning, especially in tasks with well-defined episodes and clear returns.

The last section that is covered from this chapter this week is **Section 13.4: REINFORCE With Baseline**.

Section 13.4: REINFORCE With Baseline

Overview

This section introduces an extension to the REINFORCE algorithm by incorporating a baseline into the update rule. The use of a baseline is a common technique in policy gradient methods to reduce the variance of gradient estimates, which in turn can speed up learning. The section explains how the baseline is integrated into the REINFORCE algorithm and discusses its impact on learning efficiency.

Policy Gradient Theorem with Baseline

The Policy Gradient Theorem can be generalized to include a baseline, which is subtracted from the action-value function. The baseline does not affect the expected value of the gradient but can significantly reduce its variance.

Policy Gradient Theorem with Baseline

- **Generalized Theorem**: The policy gradient theorem with baseline is expressed as:

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla_\theta \pi_\theta(a|s)$$

  where $b(s)$ is the baseline, which can be any function that does not depend on the action a.

- **Baseline Subtraction**: The subtraction of the baseline $b(s)$ helps in reducing the variance of the gradient estimate, making the learning process more stable.
- **Expected Update**: The expected update remains the same as in the original REINFORCE algorithm, ensuring that the baseline does not introduce bias into the gradient estimation.

REINFORCE with Baseline Algorithm

Incorporating the baseline into the REINFORCE algorithm leads to a modified update rule. A commonly used baseline is the state-value function $v_\pi(s)$, which is also learned using Monte Carlo methods.

REINFORCE with Baseline Algorithm

- **Update Rule**: The update rule for REINFORCE with baseline is given by:

$$\theta_{t+1} = \theta_t + \alpha \left(G_t - b(St) \right) \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}$$

  where G_t is the return, and $b(St)$ is the baseline, often chosen as an estimate of the state-value function $v_\pi(s)$.

- **Learning the Baseline**: The baseline $b(s)$ is typically learned using the same data generated by the policy. For instance, the state-value function can be learned using a separate Monte Carlo or TD method:

$$w_{t+1} = w_t + \alpha_w \left(G_t - \hat{v}(S_t, w_t) \right) \nabla_w \hat{v}(S_t, w_t)$$

- **Reduction in Variance**: The introduction of the baseline reduces the variance of the gradient estimates, which can result in faster and more stable learning.

Practical Benefits of Using a Baseline

The section discusses the practical benefits of using a baseline in the REINFORCE algorithm, particularly in terms of learning speed and efficiency. It includes a comparison of the REINFORCE algorithm with and without a baseline in a specific example.

Practical Benefits of Using a Baseline

- **Faster Learning**: By reducing the variance of gradient estimates, the baseline can lead to significantly faster learning, especially in complex environments with high variance in returns.
- **Example Comparison**: The section compares the performance of REINFORCE with and without a baseline in the short-corridor gridworld. The results demonstrate that adding a baseline improves convergence speed and reduces the number of steps required to achieve a good policy.
- **Flexibility of Baseline Choice**: While the state-value function is a common choice for the baseline, other functions can be used as long as they do not depend on the action a.

Summary of Key Concepts

- **Baseline in Policy Gradient Methods**: A technique to reduce variance in gradient estimates, leading to more efficient learning.
- **REINFORCE with Baseline**: An extension of the REINFORCE algorithm that incorporates a baseline, typically the state-value function, to stabilize learning.
- **Learning Efficiency**: The addition of a baseline can result in faster convergence and more stable updates, particularly in environments with high return variance.
- **Practical Application**: Demonstrated through examples where the baseline improves the learning speed in reinforcement learning tasks.

Incorporating a baseline into policy gradient methods is a powerful technique that enhances the efficiency and stability of learning, making it a standard practice in advanced reinforcement learning algorithms.