UNIT-5

TD Gammon

TD-Gammon is a pioneering AI program developed by Gerald Tesauro in the early 1990s to play the board game backgammon using reinforcement learning (RL). It stands as one of the earliest examples of an AI system that achieved high-level human play without being explicitly programmed with expert strategies. Instead, TD-Gammon learned by playing countless games against itself, progressively improving its strategy.

Key Concepts

1. **Temporal-Difference (TD) Learning**: TD learning is a method where the agent updates its value estimates based on the difference between predicted values and the actual rewards received. TD-Gammon uses **TD(λ)**, which blends immediate and delayed rewards to improve learning efficiency.
2. **Neural Networks**: TD-Gammon employs a neural network as a function approximator to estimate the value of different game states. Each state is input into the network, which outputs an estimated value representing the probability of winning from that state.

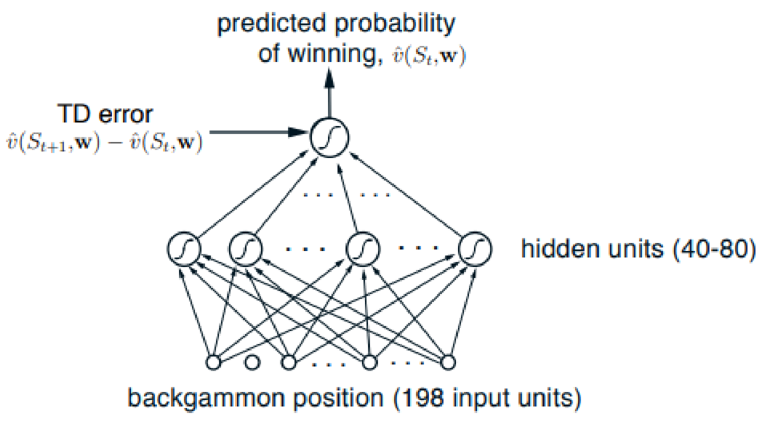

predicted probability of winning, $\hat{v}(S_t,\mathbf{w})$
TD error $\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$
hidden units (40-80)
backgammon position (198 input units)

Fig: The neural Network used in TD Gammon

3. **Self-Play**: By continuously playing against itself, TD-Gammon improves by adjusting its neural network weights based on observed rewards. This concept of self-play helps the model generalize beyond specific human-like strategies and allows it to learn powerful strategies independently.

Algorithm (TD(λ))

1. **Initialize** the neural network with random weights.
2. **Play a game**, moving from state to state based on actions chosen with the current policy.
3. For each time step:
   - **Estimate the value** V(s) of the current state ‘s’ using the neural network.
   - After taking an action and observing the next state (s′), calculate the **TD error**:

$$\delta = r + \gamma V(s') - V(s)$$

where r is the reward for the transition, and γ is the discount factor.

- **Update weights** using gradient descent on the TD error, with the update incorporating past state values weighted by λ, the eligibility trace parameter.

4. **Adjust weights** until the TD error converges, signifying that the model has learned an optimal policy for maximizing rewards.

Example of TD-Gammon in Action

1. **Setup**: TD-Gammon initializes by playing randomly, with no prior knowledge of backgammon strategy.
2. **Early Games**: At first, TD-Gammon plays poorly, but it begins to learn from mistakes, recognizing which moves lead to favorable board positions and higher winning probabilities.
3. **Learning Complex Strategies**: After many games, it starts discovering advanced backgammon strategies such as *back games* and *prime-building* (blocking an opponent's movement). This emerges from learning through self-play and temporal difference updates.
4. **High-Level Play**: Eventually, TD-Gammon reaches a level comparable to top human players, exhibiting creative and sophisticated strategies that hadn't been programmed explicitly.

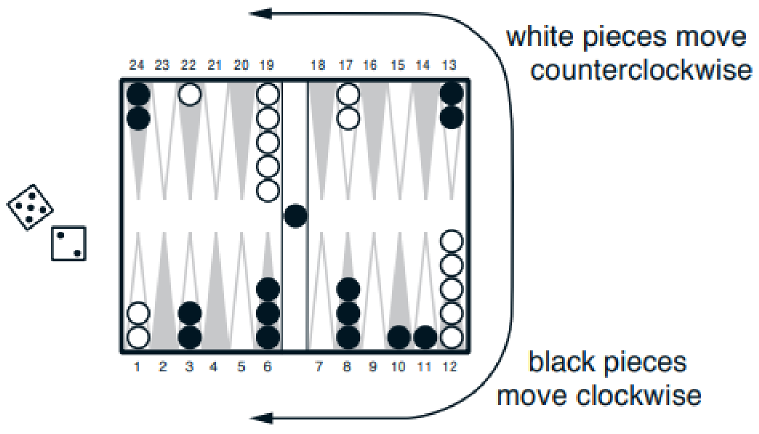

white pieces move counterclockwise
24 23 22 21 20 19
18 17 16 15 14 13
1 2 3 4 5 6
7 8 9 10 11 12
black pieces move clockwise

Fig: A backgammon position

Samuel's Checkers

Arthur Samuel's checkers-playing program, developed in the 1950s, stands as a pioneering achievement in artificial intelligence (AI) and machine learning. It was among the first programs to demonstrate self-learning capabilities, significantly influencing the evolution of AI.

Development and Objectives

Arthur Lee Samuel, an American computer scientist, initiated the development of a checkers-playing program in 1952. His primary goal was to explore machine learning techniques by enabling a computer to play checkers, a relatively simple yet strategically rich game. Samuel aimed to create a program that could improve its performance over time by learning from experience.

Key Components of Samuel's Checkers Program:

Minimax Search Algorithm: The program employed the minimax algorithm to explore possible future moves in the game. This approach involves simulating all potential moves by both the program and its opponent, aiming to minimize the possible loss in a worst-case scenario. The program would select the move that maximized its minimum gain, effectively anticipating the opponent's best possible counter-move.

Alpha-Beta Pruning: To enhance the efficiency of the minimax search, Samuel implemented alpha-beta pruning. This technique reduces the number of nodes evaluated in the search tree by eliminating branches that cannot influence the final decision, thereby allowing deeper exploration within the same computational constraints.

Evaluation Function: Given the impracticality of searching all the way to terminal game states due to computational limitations, Samuel's program used an evaluation function to estimate the desirability of non-terminal board positions. This function considered various features of the board, such as piece count and positioning, to assign a numerical value representing the likelihood of winning from that position.

Rote Learning: The program incorporated a form of memory-based learning by storing board positions it had encountered, along with their evaluated outcomes. This "rote learning" allowed the program to recall and reuse past evaluations, effectively extending its search depth without additional computation.

Learning by Generalization: Beyond memorizing specific positions, the program adjusted the parameters of its evaluation function based on game outcomes. By playing numerous games against itself, it refined these parameters to improve its assessment of board positions, a process akin to what is now known as temporal-difference learning.

Learning Mechanisms: The program employed two primary learning methods:

- **Rote Learning**: By storing and recalling board positions and their evaluations, the program could recognize and apply knowledge from previously encountered situations.
- **Generalization Learning**: The program adjusted its evaluation function based on outcomes from played games, refining its ability to assess board positions.

Notable Achievements

In 1962, Samuel's program achieved a significant milestone by defeating Robert Nealey, a self-proclaimed checkers master, in a game. This event garnered widespread attention and was considered a substantial advancement in AI at the time.

Legacy and Impact

Samuel's work laid the foundation for future AI research, particularly in game-playing programs. His innovative approaches to machine learning and heuristic search have influenced numerous AI applications beyond gaming, including decision-making systems and pattern recognition.

Significance and Impact:

Samuel's checkers program was among the first to demonstrate that a machine could learn and improve its performance over time without explicit reprogramming. It achieved a level of play that allowed it to compete with and occasionally defeat skilled human players, marking a significant milestone in AI research. The methodologies introduced by Samuel, particularly in heuristic search and machine learning, have influenced a wide range of applications beyond game playing, including optimization problems and decision-making systems.

In summary, Arthur Samuel's checkers-playing program was a landmark in AI history, demonstrating the potential of machines to learn and adapt through experience. It continues to be a reference point in the study of machine learning and artificial intelligence.

The Acrobot Environment

1. Introduction to Acrobot

The **Acrobot** environment is one of the classic control problems in OpenAI’s **Gym** toolkit, which is a collection of environments designed for developing and testing reinforcement learning (RL) algorithms. Acrobot features a two-joint, underactuated robotic arm that must swing its lower link upwards to a predefined goal position. The environment serves as a testbed for reinforcement learning algorithms, challenging agents to learn control policies in dynamic systems with sparse rewards.

Key Concepts:

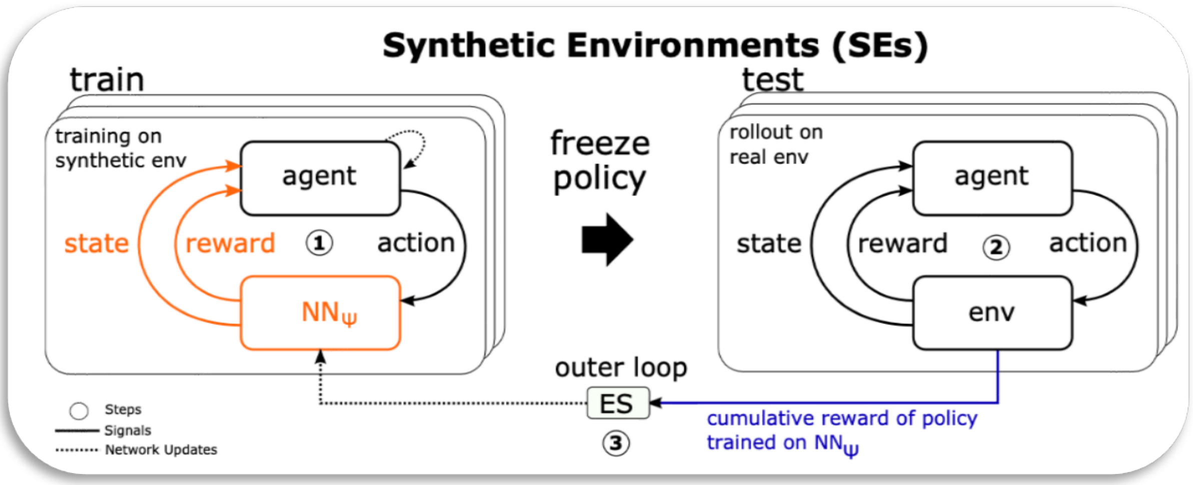
- **Underactuated System**: The system has more degrees of freedom (two joints) than actuators (one motor), which makes the control task more difficult.
- **Discrete Action Space**: The available actions for the agent are discrete. In the case of Acrobot, the agent can apply forces to either or both of the joints to affect the motion of the arm.
- **Stochastic Dynamics**: The environment includes some randomness in the dynamics, making the task more challenging as it introduces uncertainty in how the system behaves from one state to another.

2. Acrobot Mechanics

The Acrobot consists of a two-link arm, each link connected by a joint. The robot is tasked with using its actuators to swing the lower link (Link 2) upwards to a vertical position.

Structure:

- **Link 1**: The first segment of the arm is attached to a fixed base.
- **Link 2**: The second segment, attached to the first link by a hinge joint, is the link that needs to be controlled.
- **Joints**: There are two joints in the system—one between Link 1 and the base, and another between Link 1 and Link 2. The agent can control the torque applied at these joints.



Objective:

The primary objective of the environment is for the agent to swing Link 2 upwards so that it reaches the upright position. The task is considered successful when the second link (Link 2) exceeds a predefined angular threshold from the vertical position.

Action Space:

In most implementations, the agent has a **discrete action space** consisting of three possible actions:

1. **Action 0**: Apply torque to both joints in a specific direction.
2. **Action 1**: Apply torque to the first joint only.
3. **Action 2**: Apply torque to the second joint only.

The agent chooses one of these actions at each timestep, which influences the motion of the arm.

State Space:

The **state space** is continuous and typically includes the following components:

- **θ1**: The angle of the first link relative to the vertical axis.
- **θ2**: The angle of the second link relative to the first link.
- **ω1**: The angular velocity of the first link.
- **ω2**: The angular velocity of the second link.

These variables represent the configuration and motion of the arm at any given time and form the state space used by the agent to make decisions.

Reward Structure:

The reward system in Acrobot is designed to incentivize the agent to reach the goal position. Typical reward structures include:

- **Positive Reward**: A small positive reward is given when Link 2 is closer to the upright position.
- **Negative Reward**: A penalty (small negative reward) is imposed at each timestep to encourage the agent to achieve the goal efficiently, rather than simply taking actions without progressing.

If the agent successfully reaches the goal within a specified number of timesteps, a larger positive reward might be given. If the goal is not achieved, the agent receives a minimal or zero reward.

3. Importance in Reinforcement Learning

Acrobot is widely used in the reinforcement learning community due to its simplicity and the challenges it poses, making it an ideal benchmark environment for testing RL algorithms.

Key Challenges:

- **Underactuation**: Since the system has only one actuator but requires control over two joints, this adds complexity to the task. The agent must utilize momentum and plan movements over time to achieve the goal.
- **Sparse Reward**: The agent typically receives little or no feedback during many timesteps. Significant feedback only occurs when the goal is approached or reached, making it harder for the agent to learn efficiently.
- **Long-Term Planning**: To succeed, the agent needs to make decisions that influence future states over an extended period. This requires the agent to plan ahead and use delayed rewards, a common feature in many real-world control tasks.

Use in Algorithm Development:

Acrobot is often used for evaluating reinforcement learning algorithms such as:

- **Deep Q-Learning (DQN)**: DQN has been applied successfully to the Acrobot task, where deep neural networks approximate the Q-value function. This method enables the agent to learn an optimal control policy even when the environment has large state spaces.
- **Policy Gradient Methods**: Techniques like Proximal Policy Optimization (PPO) and Trust Region Policy Optimization (TRPO) are also tested on Acrobot. These methods focus on directly learning the policy rather than the value function.
- **Actor-Critic Methods**: Algorithms such as A3C or A2C use both value-based and policy- based strategies for solving Acrobot, providing a balanced approach for training the agent.

4. Practical

Considerations

Code Implementation:

The Acrobot environment is part of the **OpenAI Gym** library, which is written in Python and can be easily accessed for experimentation with reinforcement learning algorithms. Gym provides a standard API, which makes it easy to integrate the environment with various RL frameworks.

Acrobot is implemented in a way that allows researchers and practitioners to interact with it via a simple interface that tracks the agent's state, actions, rewards, and success/failure outcomes.

Real-World Applications:

Although Acrobot itself is a simulation, the control principles and algorithms developed through solving it are applicable to real-world problems, especially in robotics. Some of the key real-world applications include:

- **Robotic Arms**: Many robotic systems, including industrial robotic arms, need to perform precise movements to manipulate objects or perform tasks. The principles of underactuation and control learned in Acrobot are directly relevant to such systems.
- **Balancing Systems**: Acrobot's challenges are similar to those faced by balancing robots and systems, such as inverted pendulums or self-balancing scooters, which require precise control and long-term planning to maintain balance.
- **Autonomous Vehicles**: Although more complex, the fundamental control issues in Acrobot (e.g., underactuation, momentum, and planning) are conceptually similar to those faced in autonomous driving and vehicle control systems.

5. Conclusion

The **Acrobot environment** remains a valuable tool in reinforcement learning research. Despite its simplicity, it embodies key control challenges such as underactuation, sparse rewards, and the need for long-term planning. Its use in testing various RL algorithms makes it a crucial benchmark problem for both academic and practical advancements in the field.

Through the study and solution of Acrobot, RL researchers can refine and enhance algorithms that have applications in robotics, autonomous vehicles, and other complex dynamic systems. By leveraging Acrobot as a starting point, RL methods can be extended to solve more difficult and high- dimensional real-world problems.

Elevator Dispatching

Elevator dispatching is a classic real-world application of **Reinforcement Learning (RL)** where intelligent systems are developed to optimize the operation of elevators in multi-story buildings. This problem is part of a broader category of optimization and scheduling tasks. Below is a structured explanation of this concept as related to RL.

---

1. Achieving Elevator Dispatching

Elevator dispatching involves deciding which elevator should respond to a request (call) and in what sequence, with the goal of optimizing performance metrics such as:

- **Minimizing average waiting time** for passengers.
- **Minimizing travel time** or total energy consumption.
- Balancing **load distribution** among elevators.

Key Challenges:

- The **dynamic nature** of requests: elevator calls can occur at any time on any floor.
- **Uncertainty**: Future requests are unknown, and passenger behavior can be unpredictable.
- The **high-dimensional state space**: The system must track the state of multiple elevators and floors simultaneously.

---

2. Reinforcement Learning Approach

Reinforcement Learning provides a framework for tackling the elevator dispatching problem by modeling it as a **Markov Decision Process (MDP)**.

Key Elements of the RL Approach:

1. **State Space**:
    a. Represents the current state of the system.
    b. Example: States can include the position of each elevator, their direction, their current load, and the set of outstanding requests.
2. **Action Space**:
    a. Represents the possible actions the system can take.
    b. Example: Actions include assigning an elevator to a particular request or changing an elevator's direction.
3. **Reward Function**:
    a. Guides the learning process by assigning rewards to actions based on their outcomes.
    b. Example:
        i. Positive reward for minimizing waiting time.
        ii. Negative reward for prolonged waiting or excessive energy use.
4. **Policy**:
    a. Determines the action to take in a given state.
    b. RL learns an optimal policy to maximize cumulative rewards over time.

---

3. Techniques in RL for Elevator Dispatching

Several RL techniques can be used to address this problem:

A. **Q-Learning**:

- A model-free method where an agent learns the value of state-action pairs without knowing the system's dynamics.
- Example: Learn which elevator to dispatch based on historical rewards.

B. **Deep Reinforcement Learning**:

- Combines neural networks with RL algorithms to handle the high-dimensional state and action spaces.
- Example: Use Deep Q-Networks (DQNs) to approximate the Q-value function for complex elevator systems.

C. **Multi-Agent Reinforcement Learning (MARL)**:

- Treats each elevator as an independent agent learning to collaborate with others to optimize the system as a whole.

D. **Policy Gradient Methods**:

- Optimize the policy directly, making them suitable for continuous action spaces.

---

4. Evaluation Metrics

To assess the performance of an RL-based elevator dispatching system, the following metrics are often used:

- **Average Waiting Time**: Time passengers wait for an elevator.
- **Travel Time**: Total time passengers spend in transit.
- **Energy Efficiency**: Total energy consumed by the system.

- **Passenger Satisfaction**: A composite metric combining time and comfort.

---

5. Benefits of RL in Elevator Dispatching

- **Scalability**: Can handle increasing complexity as more elevators and floors are added.
- **Adaptability**: Learns and adjusts to changing traffic patterns in real time.
- **Optimization**: Finds policies that outperform traditional rule-based systems.

---

6. Limitations

- Requires extensive training data and computation.
- May struggle with rare, unforeseen scenarios.

---

7. Example Case Study

One notable implementation involves using RL to manage elevators in a **multi-story office building**:

Scenario:

- Building with 10 floors and 4 elevators.
- Requests are generated randomly, with varying intensities during peak and non-peak hours.

RL Approach:

- Model the system as an MDP.
- Use a Deep Q-Network to learn the optimal dispatching policy.
- Simulate real-world scenarios to train and evaluate the RL model.

Illustration:

1. **State Visualization**:
   a. Floor requests: ([3, 7, 10]).
   b. Elevator positions: ([1, 5, 8, 2]).
2. **Action**:
   a. Assign Elevator 2 to Floor 3 request.
   b. Change Elevator 1's direction to move up.
3. **Reward Computation**:
   a. Reward negatively impacts waiting time for Floor 7 request due to delay.
4. **Learning Process**:
   a. Update ($Q(S_t, A_t)$) using the reward signal and observed future states.

Result

Post-training, the policy adapts to minimize the overall waiting and travel times during peak hours.

By applying RL to elevator dispatching, we leverage advanced learning algorithms to create smarter and more efficient vertical transportation systems, offering a glimpse into the potential of AI in optimizing urban infrastructure.

Job shop scheduling

Job shop scheduling is a classic problem in operations research and optimization that involves scheduling jobs in a production facility with multiple machines. The goal is typically to optimize certain criteria, such as minimizing the total time to complete all jobs (makespan), reducing idle time, or balancing workloads across machines.

Problem Description

1. **Jobs**: A set of jobs, each consisting of a sequence of operations.
2. **Operations**: Each operation requires a specific machine for a specified duration and must be completed before the next operation in the sequence.
3. **Machines**: A set of machines, each capable of processing only one operation at a time.
4. **Constraints**:
    - **Precedence constraints**: Operations in a job must follow a specified order.
    - **Machine availability constraints**: A machine can process only one operation at a time.
    - **Resource constraints**: There may be additional constraints, such as setup times or workforce limits.

Objective Functions

- **Minimize makespan**: Total time required to complete all jobs.
- **Minimize total tardiness**: Total delay of all jobs beyond their due dates.
- **Maximize throughput**: Number of jobs completed in a given time frame.
- **Minimize idle time**: Reduce periods when machines are not in use.

Approaches to Solve

1. **Exact Methods**:
    - **Mixed-Integer Linear Programming (MILP)**: Formulate the problem as a linear program with integer variables.
    - **Branch and Bound**: Systematically explore all possible schedules.
    - **Dynamic Programming**: Break the problem into smaller subproblems.

2. **Heuristic Methods**:
    - **Dispatching Rules**: Simple rules like Shortest Processing Time (SPT), Earliest Due Date (EDD), or First-Come, First-Served (FCFS).
    - **Local Search**: Techniques like hill climbing or simulated annealing to iteratively improve schedules.

3. **Metaheuristic Methods**:
    - **Genetic Algorithms**: Use principles of evolution to explore solutions.
    - **Tabu Search**: Avoid revisiting already-explored solutions.
    - **Ant Colony Optimization**: Mimic behavior of ants finding paths to solutions.

4. **Machine Learning Approaches**:
    - Reinforcement Learning: Train an agent to generate schedules.

    - Neural Networks: Predict optimal schedules or scheduling rules based on historical data.

Applications

- Manufacturing: Allocating jobs in a factory to minimize production time.
- Logistics: Scheduling tasks in warehouses or delivery systems.
- Healthcare: Assigning operating rooms or medical equipment to procedures.
- IT: Managing tasks in computational systems or cloud environments.