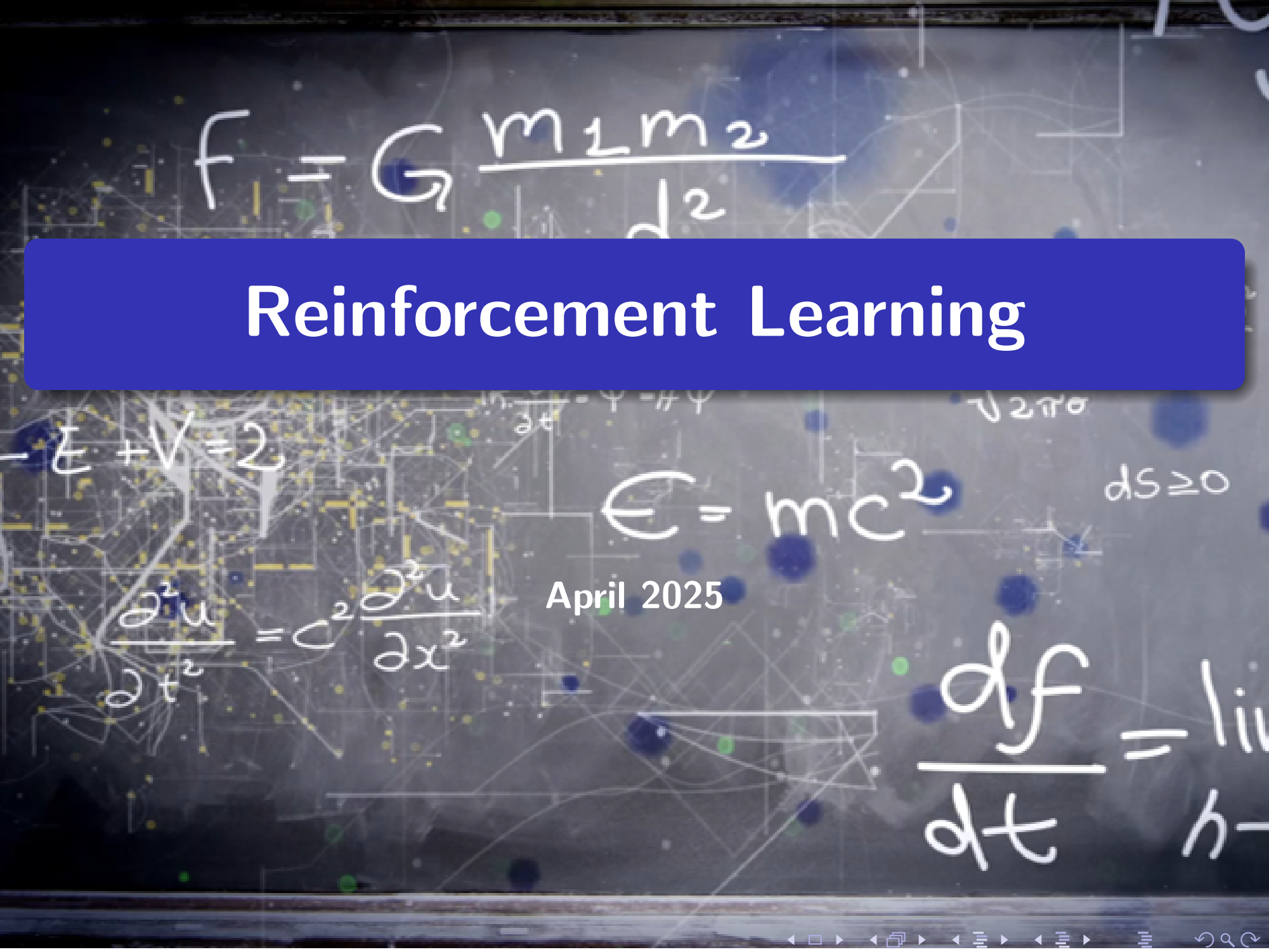


Reinforcement Learning

April 2025

Topics to be discussed

Part I. Introduction

1. Introduction to Artificial intelligence

Part II. Search and optimisation.

2. Search - basic approaches
3. Search - optimisation
4. Two-player deterministic games
5. Evolutionary and genetic algorithms

Part III. Machine learning and data analysis.

6. Regression, classification and clustering (Part I & II)
8. Artificial neural networks
9. Bayesian models
10. Reinforcement Learning

Part IV. Logic, Inference, Knowledge Representation

11. Propositional logic and predicate logic
12. Knowledge Representation

Part V. AI in Action: Language, Vision

13. AI in Natural language processing
14. AI in Vision and Graphics

Part VI. Summary

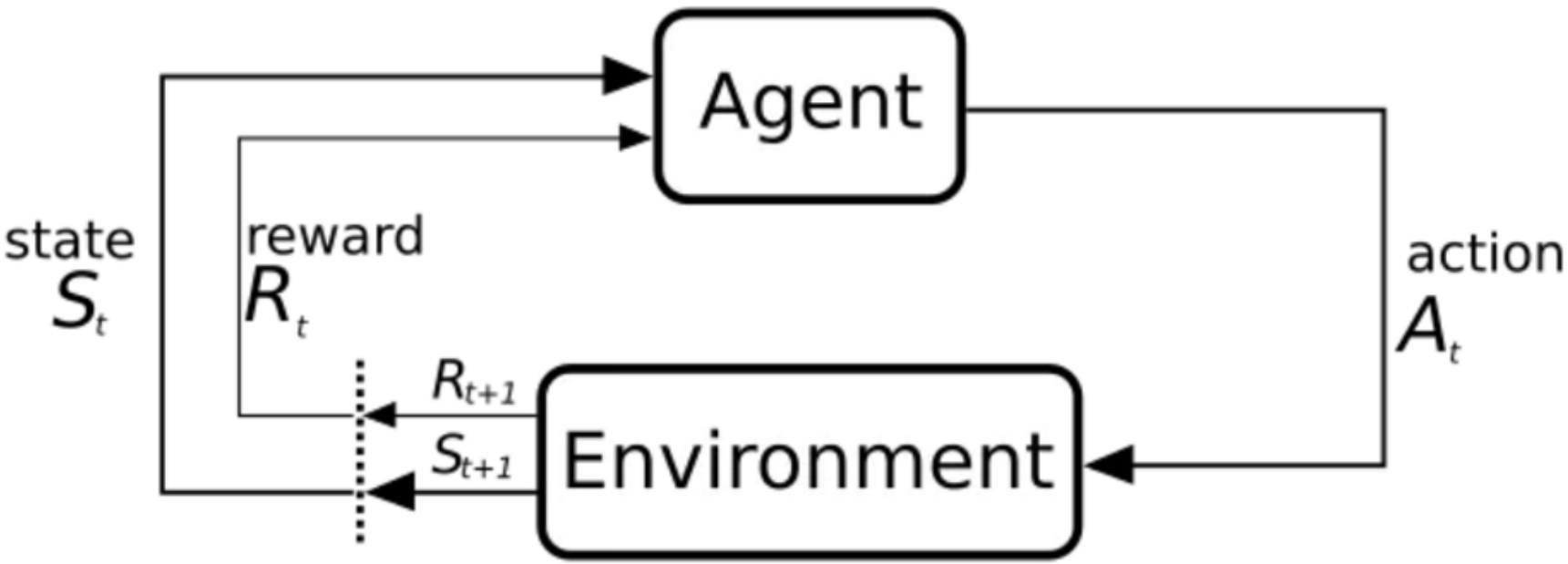
15. AI engineering, Explainable AI, Ethics,

Outline

Introduction to Reinforcement Learning

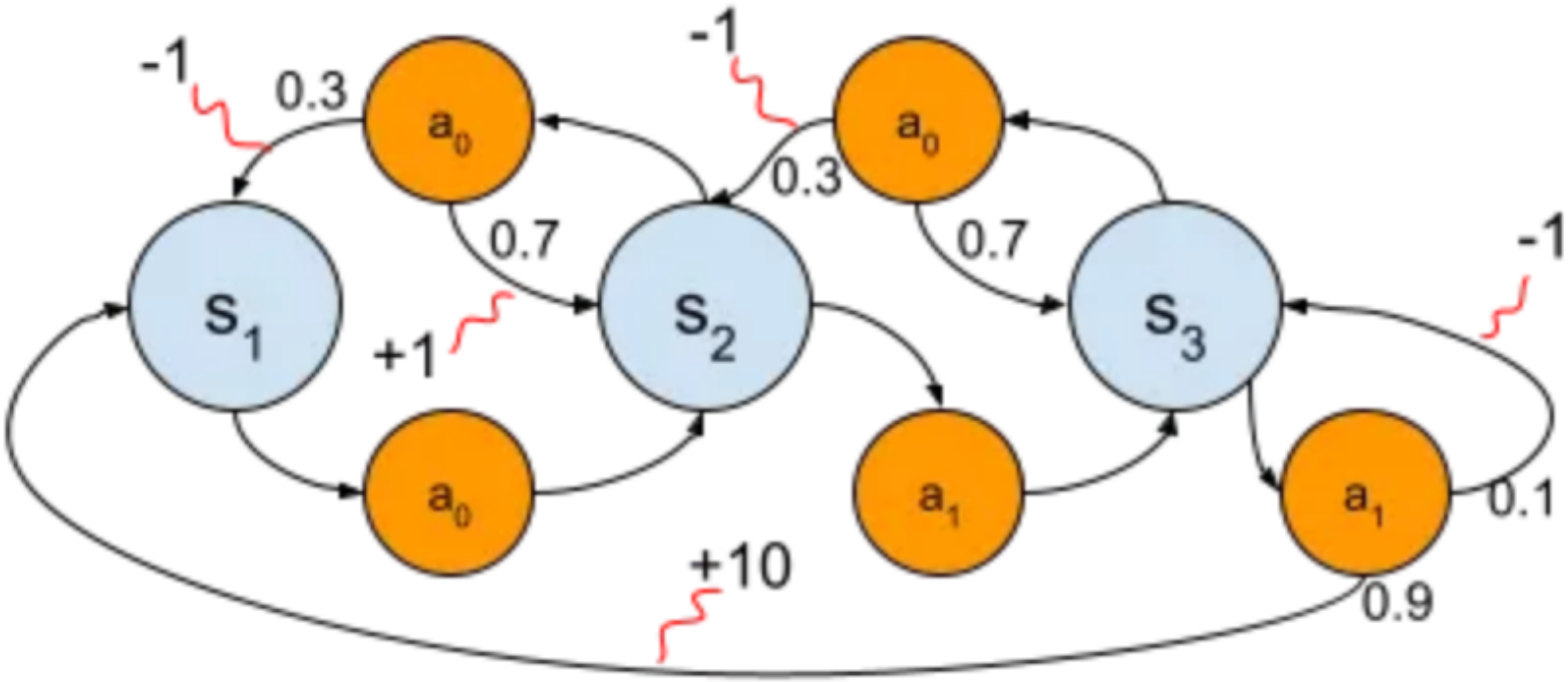
- Definition of Reinforcement Learning
- Comparison with other types of Machine Learning
- Applications of Reinforcement Learning

What is Reinforcement Learning?



https://vn040424.medium.com/understanding-markov-decision-processes-124323136d12

Markov Chain - An Example



https://towardsdatascience.com/reinforcement-learning-intro-markov-decision-process-c73a030f045d

Comparison with other types of Machine Learning

RL differs from other types of Machine Learning, such as Supervised Learning and Unsupervised Learning, in the following ways:

- RL is goal-oriented, whereas Supervised Learning and Unsupervised Learning are not.
- RL relies on trial-and-error experience, whereas Supervised Learning and Unsupervised Learning do not.
- RL requires a reward signal, whereas Supervised Learning and Unsupervised Learning do not.

What is Reinforcement Learning? (cont.)

Reinforcement Learning (RL) is a type of Machine Learning where an agent interacts with an environment to learn how to make decisions that maximize a cumulative reward. Mathematically, RL is often modeled as a Markov Decision Process (MDP), defined by the tuple (S, A, P, R, γ):

- S is the set of possible states of the environment.
- A is the set of possible actions the agent can take.
- P is the transition probability function, which defines the probability of transitioning from state s to state s' when taking action a.
- R is the reward function, which defines the immediate reward received when taking action a in state s.
- γ is the discount factor, which determines the importance of future rewards.

Applications of Reinforcement Learning

RL has been successfully applied in various domains, including:

- Robotics and Control
- Game AI
- Autonomous Driving
- Recommendation Systems
- Finance
- Healthcare

RL has shown promising results in solving complex problems that are difficult or impossible to solve using traditional methods.

Applications of Reinforcement Learning (Cont.)

Robotics and Control

- Reinforcement learning has shown great potential in robotics and control applications, where the goal is to learn control policies that can perform a task in a dynamic environment.
- RL has been used to train robots to perform a variety of tasks, such as grasping objects, walking, and flying.
- In robotics, RL can be used to learn policies that map sensor readings to actions, enabling robots to adapt to changing environments and perform complex tasks.
- RL can also be used for control applications, such as optimizing the operation of a power plant or controlling the movement of a drone.

Game AI

- Reinforcement learning has been applied to game AI, where the goal is to learn control policies for game agents that can outperform human players.
- RL has been used to train game agents in a variety of games, including chess, Go, and Atari games.
- In game AI, RL can be used to learn policies that maximize a reward function, such as the score in a game, while taking into account the game rules and the opponent's actions.
- RL can also be used to train agents that can play games with imperfect information, such as poker.

Recommendation Systems

- Reinforcement learning has been applied to recommendation systems, where the goal is to learn personalized recommendation policies for users.
- RL can be used to learn policies that maximize user engagement, such as click-through rate or conversion rate, while taking into account user preferences and feedback.
- In recommendation systems, RL can also be used to learn policies that balance exploration and exploitation, enabling the system to recommend both popular and niche items.
- RL can help improve the effectiveness of recommendation systems and provide users with better recommendations.

Key Concepts in Reinforcement Learning

- Agents, Environments, and Rewards
- Markov Decision Process (MDP)
- Policy, Value Functions, and Bellman Equations

Agents, Environments, and Rewards

- An Agent is an entity that interacts with an environment to learn how to make decisions that maximize a cumulative reward.
- An Environment is the external system with which the agent interacts.
- A Reward is a scalar feedback signal provided by the environment to the agent at each time step. The agent's goal is to maximize the cumulative reward over time.

Markov Decision Processes

- A Markov Decision Process (MDP) is a mathematical framework for modeling RL problems.
- An MDP is defined by a tuple (S, A, P, R, γ), where:
  - S is the set of possible states.
  - A is the set of possible actions.
  - $P(s'|s, a)$ is the probability of transitioning to state s' when taking action a in state s.
  - $R(s, a, s')$ is the reward for transitioning to state s' when taking action a in state s.
  - $\gamma \in [0, 1]$ is the discount factor.
- The goal is to find a policy $\pi : S \to A$ that maximizes the expected sum of discounted rewards, defined as follows: $\sum_{t=0}^{\infty} \gamma^t r_{t+1}$, where r_t is the reward obtained at time t.
- Reinforcement learning algorithms aim to learn an optimal policy π^* that maximizes the expected sum of discounted rewards in an MDP.

Policy

A Policy is a function that maps each state to a probability distribution over actions. The policy determines the agent's behavior in the environment.

$$\pi(a|s) = P[\text{take action } a \text{ in state } s]$$

The value of a state under a policy is the expected sum of discounted rewards starting from that state and following the policy. It can be computed as follows:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^{t} r_t \middle| s_0 = s\right]$$

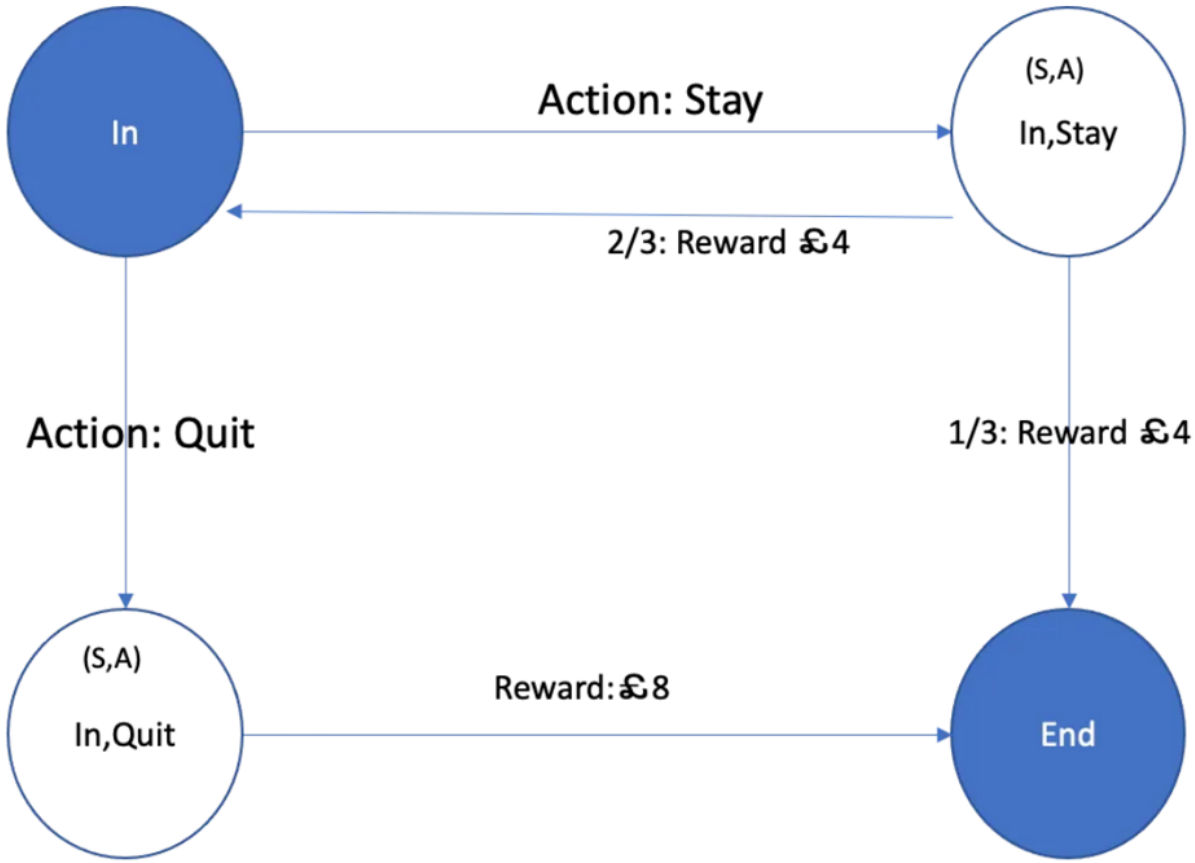
where $\mathbb{E}_{\pi}$ denotes the expected value under policy π, and r_t is the reward obtained at time step t.

Value Functions

Value Functions estimate how good it is for an agent to be in a particular state or to take a particular action. There are two types of Value Functions:

- State Value Function: $V(s)$ estimates the expected cumulative reward starting from state s under a given policy.
- Action Value Function: $Q(s, a)$ estimates the expected cumulative reward starting from state s, taking action a, and then following a given policy.

Markov Chain Example - Dice Game



https://vn040424.medium.com/understanding-markov-decision-processes-124323136d12

Markov Decision Process - Example 1

Consider a simple grid-world environment where an agent can move up, down, left, or right from any state. The agent receives a reward of -1 for each step it takes, and $+10$ for reaching the goal state. The environment is shown below:

$S_{1,1}$	$S_{1,2}$	$\cdots$	$S_{1,n}$
$S_{2,1}$	$S_{2,2}$	$\cdots$	$S_{2,n}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$S_{m,1}$	$S_{m,2}$	$\cdots$	$S_{m,n}$

The agent's actions are probabilistic, meaning that it will move in the desired direction with probability 0.8, and will move in a random direction with probability 0.2. If the agent attempts to move off the grid, it will stay in its current state instead. The agent uses a discount factor of $\gamma = 0.9$.

This environment can be modeled as an MDP with the following components:

- States: $\{S_{1,1}, S_{1,2}, \ldots, S_{m,n}\}$
- Actions: $\{\text{up}, \text{down}, \text{left}, \text{right}\}$
- Transition function: $P(s' \mid s, a)$, which gives the probability of moving to state s' when taking action a in state s.
- Reward function: $R(s, a, s')$, which gives the reward for transitioning from state s to s' under action a.

Markov Decision Process - Example 2

Suppose we have an MDP with three states: s_1, s_2, and s_3. The agent can take two actions in each state: move left or move right. The transition probabilities and rewards for each action in each state are as follows:

- $P(s_1'|s_1, \text{move left}) = 0.8$, $P(s_2'|s_1, \text{move left}) = 0.2$, $R(s_1, \text{move left}, s_1') = 0$
- $P(s_2'|s_1, \text{move right}) = 0.8$, $P(s_1'|s_1, \text{move right}) = 0.2$, $R(s_1, \text{move right}, s_2') = 10$
- $P(s_2'|s_2, \text{move left}) = 0.6$, $P(s_3'|s_2, \text{move left}) = 0.4$, $R(s_2, \text{move left}, s_2') = 0$
- $P(s_3'|s_2, \text{move right}) = 0.6$, $P(s_2'|s_2, \text{move right}) = 0.4$, $R(s_2, \text{move right}, s_3') = 5$
- $P(s_3'|s_3, \text{move left}) = 1$, $R(s_3, \text{move left}, s_3') = 0$
- $P(s_3'|s_3, \text{move right}) = 0$, $R(s_3, \text{move right}, s_3') = 0$

A policy π specifies which action to take in each state. An example policy:

- $\pi(s_1) = \text{move right}$
- $\pi(s_2) = \text{move right}$
- $\pi(s_3) = \text{move left}$

Bellman Equations

Bellman Equations are a set of recursive equations that relate the value of a state or action to the values of its successor states or actions. The Bellman Equations are used to update the value functions during learning.

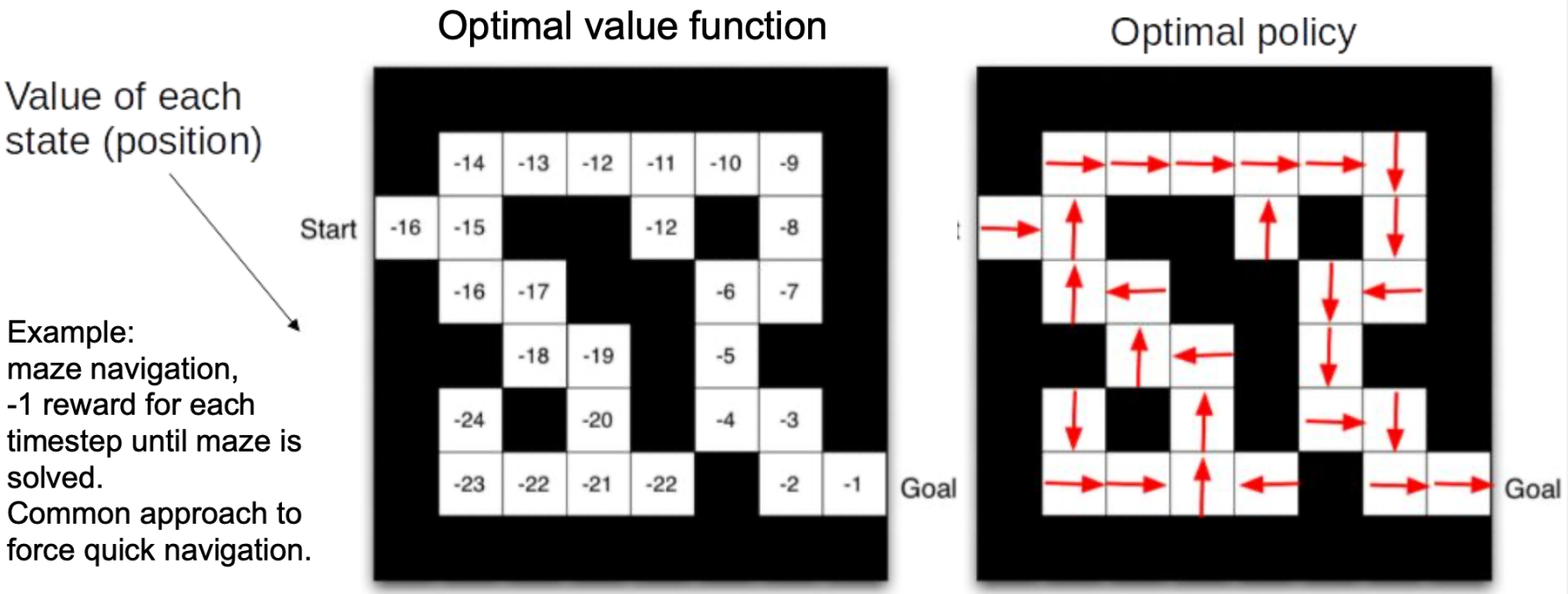
- State Value Function:
  $V(s) = \sum_{a \in A} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$
- Action Value Function:
  $Q(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q(s',a')]$

Value Iteration Algorithm

- Value Iteration Algorithm is a dynamic programming algorithm used to determine the optimal policy for a Markov Decision Process (MDP).
- It relies on the principle of Bellman optimality, which states that the optimal value of a state is equal to the maximum expected value of all possible actions from that state.
- It updates the values of each state iteratively until convergence, taking into account the discounted future rewards.
- The value of each state is based on the maximum expected value of all possible actions from that state.
- Once the value function has converged, the optimal policy can be obtained by selecting the action that maximizes the expected value for each state.
- It can be applied to both finite and infinite horizon problems, and provides an exact solution to the MDP.

Optimal Value and Optimal Policy

The optimal policy



https://www.chegg.com/homework-help/questions-and-answers/reinforcement-learning-problem-base-example-given-example-maze-optimal-values-state-result-q75835182

Value Iteration Algorithm (Cont.)

Algorithm Value Iteration Algorithm

Input: MDP with state space S, action space A, transition function T, reward function R, discount factor γ

Output: Optimal value function V and optimal policy π

for *each state* $s \in S$ **do**
  $V(s) \leftarrow 0$
end
while $\Delta < \epsilon$ **do**
  $\Delta \leftarrow 0$
  for *each state* $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end
end
for *each state* $s \in S$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in A} \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V^{(}s')]$
end

where $V(s)$ is the value of state s, A is the action space, $T(s, a, s')$ is the transition probability from state s to state s' under action a, $R(s, a, s')$ is the reward received when transitioning from state s to state s' under action a, γ is the discount factor, Δ is the maximum change in the value function during an iteration, and ϵ is a small positive value used to determine convergence.

Policy Iteration Algorithm

Algorithm Policy Iteration Algorithm

Initialize π_0 randomly
while $\Delta < \theta$ **do**
    Policy Evaluation:
    $V_{\pi_k}(s) \leftarrow \sum_{a \in A} \pi_k(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_{\pi_k}(s')]$
    Policy Improvement:
    policy-stable $\leftarrow$ true
    for $s \in S$ **do**
        $b \leftarrow \pi_k(s)$
        $\pi_k(s) \leftarrow \underset{a}{\text{argmax}} \sum_{s',r} p(s', r|s, a)[r + \gamma V_{\pi_k}(s')]$
        if $b \neq \pi_k(s)$ **then**
            policy-stable $\leftarrow$ false
        end
    end
    if *policy-stable* **then**
        break
    end
end

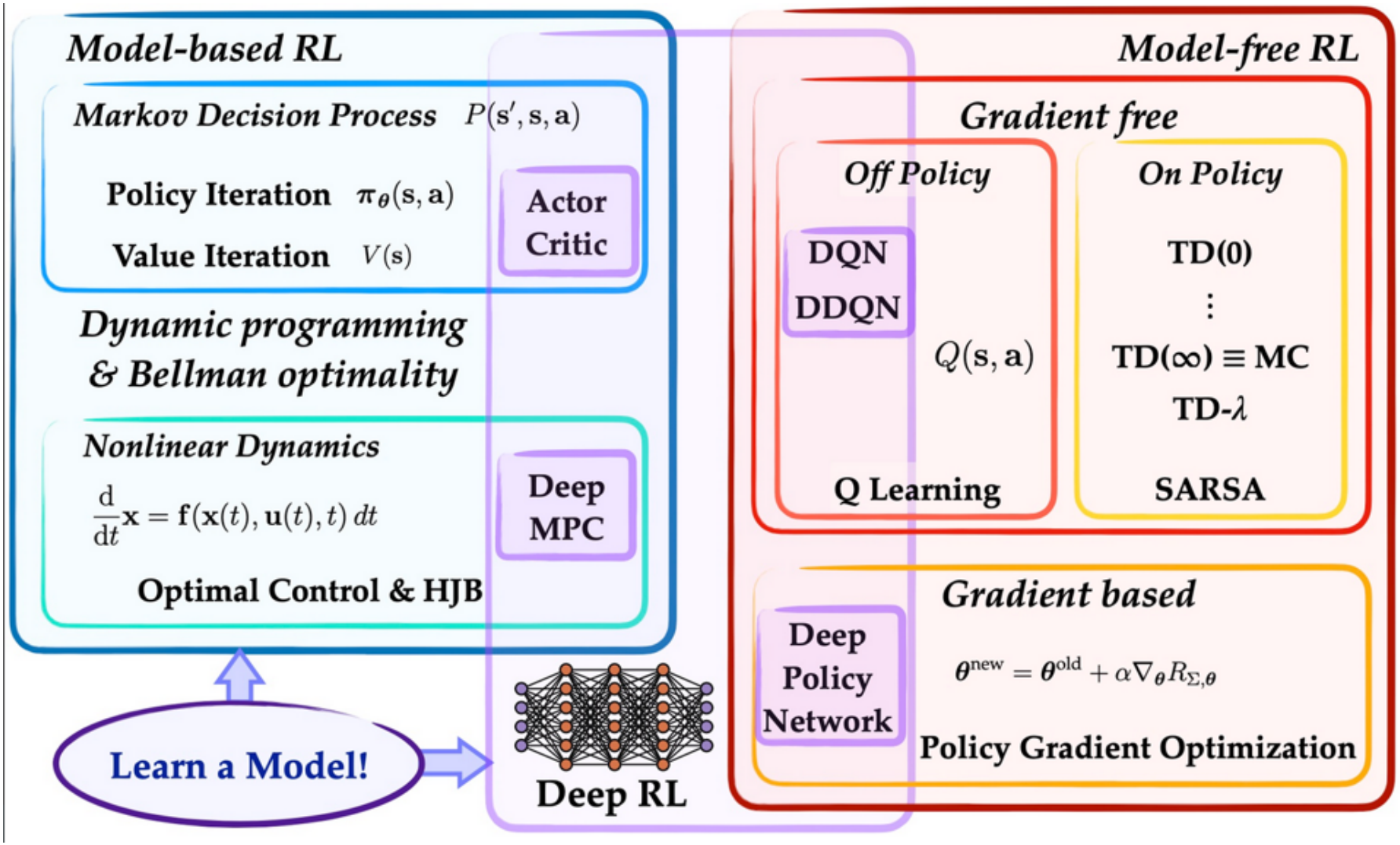
Reinforcement Learning - Recap

- Reinforcement Learning is a type of Machine Learning that focuses on learning to make decisions to maximize a cumulative reward.
- Key concepts in Reinforcement Learning include Agents, Environments, Rewards, Markov Decision Process (MDP), Policy, Value Functions, and Bellman Equations.
- Agents interact with Environments and receive Rewards. MDP is a framework for modeling Reinforcement Learning problems.
- Policy maps each state to a probability distribution over actions.
- Value Functions estimate how good it is to be in a particular state or to take a particular action.
- Bellman Equations relate the value of a state or action to the values of its successor states or actions.
- Value Iteration Algorithm is a dynamic programming algorithm and can be used to compute value of each state and then determine the optimal policy for a Markov Decision Process (MDP).

Reinforcement Learning Algorithms

- Model-based vs. Model-free
- Temporal Difference Learning
- Q-Learning and SARSA
- Policy Gradient Methods
- Deep Reinforcement Learning

Reinforcement Learning Algorithms (cont.)



https://twitter.com/eigensteve/status/1450182390790307841

Policy Search Methods in RL

- In reinforcement learning, the goal is to learn a policy that maximizes the expected cumulative reward.
- Model-free methods, such as Q-learning and SARSA, learn the optimal policy by estimating the action-value function or state-value function.
- Model-based methods, such as value iteration and policy iteration, learn the optimal policy by estimating the transition probabilities and rewards.
- Policy search methods directly optimize the policy parameters to maximize the expected cumulative reward.
- Policy search methods can be divided into two categories:
  - Direct policy search methods: optimize the policy directly.
  - Indirect policy search methods: optimize a surrogate objective function that is related to the policy.
- Policy search methods have some advantages over value-based methods, such as being able to handle continuous action spaces and stochastic policies.
- However, policy search methods can be more computationally expensive than value-based methods, especially for high-dimensional problems.

Model-based vs. Model-free RL

RL algorithms can be broadly classified into two categories based on how they represent the environment: Model-based and Model-free.

Model-based RL algorithms maintain a model of the environment, which includes a transition function P and a reward function R. They use this model to compute value functions and policies. The main advantage of model-based RL is that it can take advantage of the knowledge of the environment's dynamics to learn more efficiently.

Model-free RL algorithms, on the other hand, do not maintain a model of the environment. Instead, they learn value functions and policies directly from experience. The main advantage of model-free RL is that it can handle complex environments where it may be difficult to construct an accurate model.

Examples of Direct Policy Search Algorithms in RL

- **Hill Climbing**: An optimization algorithm that starts from a random policy and iteratively updates it by making small perturbations to the policy and evaluating the resulting performance. The perturbations are accepted or rejected based on whether they improve the performance.
- **Cross-Entropy Method**: A population-based optimization algorithm that maintains a set of policies and selects the best ones to generate new policies in the next iteration. The policies are selected based on their performance, and the new policies are generated by sampling from a multivariate Gaussian distribution that is fit to the best policies.
- **Policy Gradient**: A family of algorithms that directly optimize the policy parameters using gradient ascent. The gradient is estimated by running rollouts of the policy and computing the average reward weighted by the gradient of the log-probability of the actions.
- **Evolutionary Strategies**: A family of optimization algorithms that maintain a population of policies and update them using a form of gradient-free optimization. The policies are mutated by adding noise to the parameters, and the performance of the resulting policies is evaluated and used to select the best ones for the next generation.

Examples of Indirect Policy Search Algorithms in RL

Indirect policy search algorithms in RL include:

- **Value-based methods**: These algorithms indirectly learn an optimal policy by estimating the value function associated with a given policy. Examples include Q-Learning, SARSA, and Expected SARSA.
- **Policy iteration**: These algorithms iteratively improve a policy by alternating between policy evaluation and policy improvement steps. Examples include Policy Iteration and Value Iteration.
- **Actor-critic methods**: These algorithms combine a policy network (actor) with a value function network (critic) to learn an optimal policy. Examples include A2C, A3C, and DDPG.
- **Trajectory-based methods**: These algorithms learn an optimal policy by directly optimizing the expected return of sampled trajectories. Examples include REINFORCE and PPO.

Indirect policy search algorithms are generally more sample-efficient than direct policy search methods, as they can make use of the additional structure provided by the value function. However, they can be more complex to implement and can suffer from convergence issues in certain scenarios.

Temporal Difference Learning

- Temporal Difference (TD) is an agent learning from an environment through episodes with no prior knowledge of the environment. This means temporal difference takes a model-free RL or unsupervised learning approach. You can consider it learning from trial and error.
- Temporal Difference learning is a model-free RL algorithm that updates value functions based on the difference between estimated values and actual values observed in subsequent time steps.
  The update equation for TD(0) is:

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

  where:
  - $V(S_t)$ is the estimated value of state S_t,
  - R_{t+1} is the observed reward at time $t+1$,
  - S_{t+1} is the next state observed at time $t+1$,
  - α is the learning rate, and
  - γ is the discount factor.

Temporal Difference (TD) Learning

- TD learning is a type of reinforcement learning algorithm that learns to estimate the value function by updating its estimate based on the difference between the predicted and actual reward.
- TD(0): The simplest form of TD learning, also known as one-step TD, updates the value function estimate at every time step based on the difference between the predicted and actual reward plus the estimate of the value function at the next time step.
- TD(λ): TD learning with eligibility traces, also known as TD(lambda), is a more general form of TD learning that updates the value function estimate using a weighted average of the TD(0) updates for each time step, with the weights determined by the eligibility traces.
- TD(∞): The limit of TD(λ) as $\lambda \to \infty$, also known as Monte Carlo TD, updates the value function estimate using the actual total reward obtained from the episode, rather than the predicted reward.

Temporal Difference (TD) Learning Algorithm - Example 1

Algorithm TD Learning algorithm for estimating the state-value function V of a policy π

Input: learning rate α, discount factor γ

Data: state-value function V

Result: approximate $V \approx v_\pi$

for *each episode* **do**

  Initialize s

  while s *is not terminal* **do**

    Choose a from s using policy derived from V

    Take action a, observe r and s'

    $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$

    $s \leftarrow s'$

  end

end

Temporal Difference (TD) Learning Algorithm - Example 2

Algorithm Temporal Difference Learning Algorithm

Data: learning rate α, discount factor γ, initial state s, initial action a

Result: optimal policy π^*

Initialize $Q(s, a)$ arbitrarily

while *not converged* **do**

- Observe current state s
- Select action a based on current policy π
- Take action a, observe reward r and next state s'
- Update $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$
- Update current state and action $s \leftarrow s'$, $a \leftarrow a'$

Q-Learning and SARSA

- Q-Learning
  - Off-policy, model-free TD learning algorithm
  - Learns the optimal action-value function, $Q^(s, a)$
  - Uses the greedy policy with respect to Q to select actions
  - Update rule: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
- SARSA
  - On-policy, model-free TD learning algorithm
  - Learns the action-value function, $Q^{\pi}(s, a)$, under a policy π
  - Selects actions according to the policy π
  - Update rule: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a) \right]$

Q-Learning and SARSA (cont.)

- Q-Learning
  - Guarantees convergence to the optimal action-value function, $Q^*(s, a)$, under certain conditions
  - Requires exploration to discover optimal policy
  - Can be slow to converge, especially for large state spaces
  - Can overestimate the value of actions in noisy or stochastic environments
- SARSA
  - Converges to $Q^\pi(s, a)$ for any policy π, under certain conditions
  - Can learn directly from experience under the policy being evaluated
  - More sample efficient than Q-Learning, especially for small state spaces
  - Can be sensitive to the initial policy and may converge to suboptimal solutions

Q-Learning Algorithm

Algorithm Q-Learning Algorithm

Input: Environment E, action-value function Q, learning rate α, discount factor γ, exploration rate ϵ, number of episodes N

Output: Optimal policy π^*

Initialize Q arbitrarily

for *episode* $= 1$ *to* N **do**

  Initialize state s

  while s *is not terminal* **do**

    Choose action a based on ϵ-greedy policy derived from Q

    Take action a and observe next state s' and reward r

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

    Update $s \leftarrow s'$

  end

end

return $\pi^* \leftarrow \text{argmax}_a Q(s, a)$

SARSA Algorithm

Algorithm SARSA Algorithm

Input: Initial state s_0, initial action a_0, step size α, discount factor γ, exploration rate ϵ

for *episode* $= 1, 2, \ldots$ **do**
  Initialize $s = a$, a randomly using ϵ-greedy policy
  for $t = 0, 1, \ldots$ **do**
    Take action a, observe reward r and next state s'
    Choose a' from s' using ϵ-greedy policy
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
    if s' *is terminal* **then**
      break
    end
  end
end

Policy Gradient Theorem

Policy Gradient Theorem is a fundamental result in reinforcement learning (RL) that provides a way to directly learn an optimal policy without using a value function. It enables the optimization of policies that are defined by a parameterized function.

Theorem Statement

Let $\pi_\theta(a|s)$ be a differentiable policy with parameters θ and $J(\theta)$ be the expected return of the policy. The policy gradient theorem states that the gradient of the expected return with respect to the policy parameters is given by:

$$\nabla_\theta J(\theta) \propto \sum_s d^\pi(s) \sum_a q^\pi(s,a) \nabla_\theta \pi_\theta(a|s)$$

where $d^\pi(s)$ is the stationary distribution of the Markov chain induced by policy π and $q^\pi(s,a)$ is the state-action value function.

- The theorem provides a way to update policy parameters directly by gradient ascent.
- This is in contrast to value-based methods where the value function is learned and the policy is derived from the value function.
- Policy Gradient Theorem can be applied to both continuous and discrete action spaces.

Policy Gradient Algorithm

- Policy Gradient Formula
  - Gradient of the expected reward with respect to the policy parameters
  - Enables optimization of the policy using gradient descent
- REINFORCE Algorithm
  - Monte Carlo policy gradient algorithm
  - Samples trajectories from the environment and computes policy gradient
  - Update the policy parameters using gradient descent

REINFORCE Algorithm Pseudocode

Algorithm REINFORCE Algorithm

Input: Policy π_θ, learning rate α

Output: Optimized policy π^*

Initialize policy parameters θ randomly

while *not converged* **do**

Sample a trajectory $\tau = (s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ under π_θ

Compute the returns $G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$

Compute the policy gradient $\nabla_\theta J(\theta) \approx \frac{1}{|\tau|} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$

Update the policy parameters $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

end

Example Algorithm: Deep Q-Networks (DQN)

Algorithm Deep Q-Networks (DQN)

Input: Initial state s, replay memory D, Q-network parameters θ
Output: Optimal action-value function $Q^*(s, a)$
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for *episode* $= 1, M$ **do**
  Initialize state s
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t, otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample a minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from D
    Set target for each minibatch transition i as $y_i = \begin{cases} r_i & \text{for terminal } s_{i+1} \\ r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \theta^-) & \text{for non-terminal } s_{i+1} \end{cases}$
    Perform a gradient descent step on $(y_i - Q(s_i, a_i; \theta))^2$ with respect to the network parameters θ
    Every C steps, reset $\hat{Q} = Q$
  end
end

Example Algorithm: Deep Q-Networks (DQN) (Cont.)

An explanation of the variables used in the Deep Q-Networks (DQN) algorithm:

- S_t: The state at time step t.
- A_t: The action taken at time step t.
- R_t: The reward received at time step t.
- S_{t+1}: The next state at time step $t+1$.
- D: The replay buffer, which stores the experiences (S_t, A_t, R_t, S_{t+1}).
- θ: The parameters of the Q-network, which is used to estimate the Q-value of each action given a state.
- θ^- : The target parameters of the Q-network, which are used to compute the target Q-value.
- ϵ: The probability of selecting a random action (exploration rate).
- ϵ_{min}: The minimum value of the exploration rate.
- ϵ_{decay}: The rate at which the exploration rate decays over time.
- N: The number of steps after which the target Q-network is updated with the parameters of the Q-network.
- γ: The discount factor, which determines the importance of future rewards.

Conclusion and Future Directions

- Reinforcement learning is a type of machine learning that involves an agent interacting with an environment to maximize a reward signal.
- Key concepts in reinforcement learning include agents, environments, rewards, Markov decision processes, policies, value functions, and Bellman equations.
- RL algorithms include model-based and model-free methods, temporal difference learning, Q-learning, SARSA, and policy gradient methods.
- Advanced topics in RL include deep reinforcement learning, multi-agent RL, inverse RL, and hierarchical RL.
- RL has applications in various domains such as robotics and control, game AI, autonomous driving, and recommendation systems.

Summary

References

1. R.S. Sutton, A.G.Barto, "Reinforcement Learning. An Introduction", MIT Press, 2nd ed., 2020
2. S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach", Financial Times Prentice Hall, 2019.
3. A. Geron, "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow", O'Reilly Media, 3rd ed., 2023