HUR Group Crash Course

"Introduction to (Deep) Reinforcement Learning (RL)"

Pilwon Hur, PhD

Mechanical and Robotics Engineering, GIST

Aug 21, 2024

Contents

1. Overview
2. Details of RL
3. RL Algorithms
4. Details of Deep RL
5. Deep RL Algorithms
6. Experiments
7. Summary and Further Discussion

Contents

1 Overview

2 Details of RL

3 RL Algorithms

4 Details of Deep RL

5 Deep RL Algorithms

6 Experiments

7 Summary and Further Discussion

Agenda of the Course

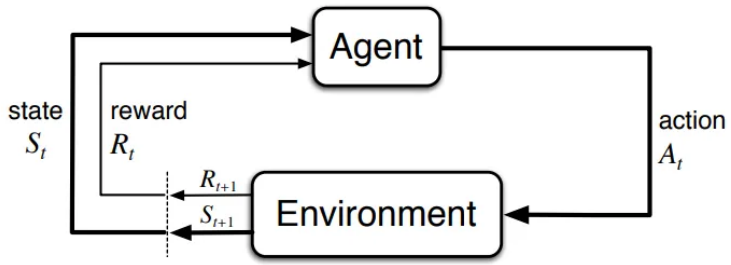
- Overview
  1. Summary of MDP + MDP Challenge Review
  2. Transition from MDP to RL
  3. Examples of RL
- Details of RL
  1. Assumptions and Structures of MDP
     1. State
     2. State Transition Probability
     3. Action
     4. Policy
     5. Reward
     6. Discounting Factor
     7. Value
     8. Bellman Equation
     9. Dynamic Programming
  2. Solution of Bellman Equation
     1. Dynamic Programming: Value Iteration, Policy Iteration
     2. Monte Carlo
     3. Deel Learning
- Experiments

Summary of MDP (I)

A Markov Decision Process (MDP)

MDP is a mathematical framework used to model decision-making where outcomes are partly under the control of the decision-maker (agent) and partly random.

- Components
  - States (S): The set of all possible situations in which the agent can be.
  - Actions (A): The set of all possible actions the agent can take.
  - Transition Probabilities (P): The probability $P(s\prime \mid s, a)$ of moving from state s to state $s\prime$ given action a.
  - Rewards (R): The immediate reward $R(s, a, s\prime)$ received after transitioning from state s to state $s\prime$ due to action a.
  - Discount Factor (γ): A factor ($0 \leq \gamma < 1$) that represents the importance of future rewards compared to immediate rewards.



Summary of MDP (II)

- Goal of MDP
  - To find the a series of actions (i.e., policy) that maximize the cumulative sum of rewards (i.e., return) from the current state.
  - Since the actual rewards cannot be known in advance before the decision is made, the expected sum of rewards (i.e., expected return, or value) is used, instead. $v(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots\right]$
- Bellman Equation
  - The expected return (i.e., value[1]) can be expressed in a recursive form, the Bellman equation. $v(s) = \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1})\right]$
- Strategies to achieve the goal of MDP
  1. To find the maximum value function $v^*(s) = \max_a \left[R_{t+1} + \gamma v^*(S_{t+1})\right]$
  2. To find the policy $\pi^*(s)$ that maximizes the value
- Dynamic Programming[2]
  1. Value iteration $v_{k+1}(s) \leftarrow \max_a \left[R_{t+1} + \gamma v_k(S_{t+1})\right]$
  2. Policy iteration: Policy evaluation $\left(v_{k+1}(s) \leftarrow \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1})\right]\right)$ and Policy improvement (greedy policy)

[1] The collection of all values for all states is called value function

[2] Iteration converges to true value function due to contraction mapping and Banach fixed point theorem

Transition from MDP to RL (I)

- Issues of MDP
  - Agents have complete knowledge of the environment, including:
    - Perfect sensors: set of all state S
    - Set of all available actions A
    - Perfect model I: the transition probabilities $P(s\prime \mid s, a)$
    - Perfect model II: the reward function $R(s, a, s\prime)$
      $\rightarrow$
    - Real-World Challenges: In many real-world scenarios, the transition probabilities and reward functions are unknown and difficult to model.
    - Building an accurate model of the environment is often impractical or impossible.
  - Curse of dimensionality
    - Agents need to know the value function and the policy for ALL states
    - As the complexity of the environment increases, the state and action spaces grow exponentially.
    - Managing and computing optimal values and policies using DP (e.g., VI, PI) becomes computationally infeasible. (doable up to 3-4DOFs)
      $\rightarrow$
    - Need for Efficient Learning: RL algorithms are designed to learn good policies from experience, without requiring an exhaustive computation of the entire state-action space.

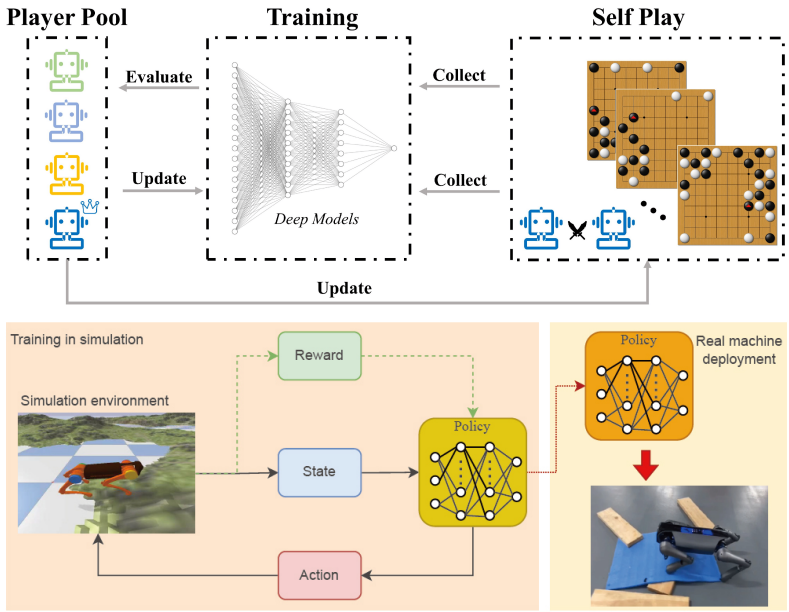
Transition from MDP to RL (II)

- Issues of MDP (continued)
  - Fixed policy
    - MDP-based solutions assume that the agent already knows or can compute the optimal policy from the start.
    - However, the optimal policy may not be effective in a dynamically changing environment.
      $\rightarrow$
    - Learning Through Interaction: In practice, the agent must explore the environment to discover which actions yield the best rewards.
    - RL balances exploration (trying new actions) with exploitation (using known actions) to learn an optimal policy over time.

Transition from MDP to RL (III)

- Benefits of RL
  - No Need for an Explicit Model
    - In many applications, constructing an explicit model (as required by MDP) is difficult or impossible.
    - RL, particularly model-free methods like Q-Learning, allows the agent to learn optimal behaviors directly from interaction with the environment without needing to model the environment explicitly.
  - Continuous and Complex Environments
    - Handling Continuous State and Action Spaces: MDPs are traditionally defined for discrete state and action spaces, but many real-world problems involve continuous states and actions.
    - RL with Function Approximation: RL techniques, especially with function approximation (e.g., neural networks, deep learning), can handle continuous spaces effectively, making them suitable for complex, real-world applications.
  - MDPs Provide a Foundation, But RL is Practical
    - While MDPs provide the theoretical foundation for decision-making under uncertainty, RL is necessary for practical, scalable solutions where the environment model is unknown, large, or complex.

Examples and Applications of RL



- Characteristics
  - Self-play and deep neural networks.
  - Strategic decision-making in complex, high-dimensional spaces.
  - Real-time decision-making.
  - Handling large action spaces and partial observability.
  - Continuous state and action spaces.
  - Learning from real-world interactions.
- AlphaGo, AlphaZero
- Video Games
- Quadruped Robot Walking
  - Training a quadruped to walk and adapt to different terrains.
  - Using deep RL, the robot learns to balance and coordinate its legs to move efficiently.
  - Outcome: The robot can dynamically adjust its gait and recover from disturbances, enabling it to walk on uneven surfaces like sand, rocks, or slopes.
- Healthcare and Personalized Medicine
  - RL applied to optimize treatment strategies for patients, such as dosing in cancer therapy or managing chronic diseases.
  - Adaptation to patient-specific responses.
  - Balancing efficacy and safety.

Contents

1 Overview

2 Details of RL

3 RL Algorithms

4 Details of Deep RL

5 Deep RL Algorithms

6 Experiments

7 Summary and Further Discussion

Introduction to RL

Reinforcement Learning (RL)

RL is a type of machine learning where an agent learns to make decisions by interacting with an environment to maximize cumulative reward (similar to MDP). Unlike supervised learning, where the agent learns from a dataset of correct answers, in RL, the agent learns from the consequences of its actions in a trial-and-error manner.

Components of MDP

An MDP is defined by a tuple (S, A, R, γ):
Agent: The learner or decision-maker.
Environment: What the agent interacts with.
State (S): The current situation the agent is in.
Action (A): Choices available to the agent.
Reward (R): Feedback from the environment.

More on the components of RL (I)

- Agent
  - The learner or decision-maker that interacts with the environment.
  - The goal of the agent is to maximize the cumulative reward over time by learning the optimal policy.
- Environment
  - The world in which the agent operates.
  - The environment provides feedback to the agent in the form of rewards or penalties based on the actions taken by the agent.
- State (S)
  - A representation of the current situation or configuration of the environment as perceived by the agent.
  - States encapsulate all the relevant information needed to make decisions.
- Action (A)
  - A set of all possible moves or decisions the agent can take in a given state.
  - Actions cause transitions from one state to another.

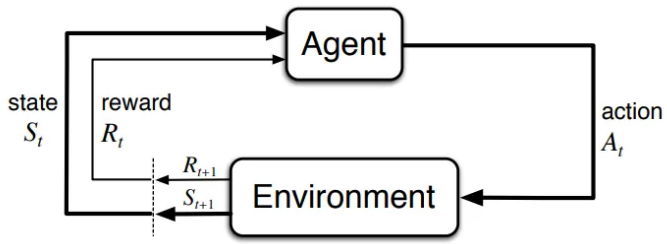
More on the components of RL (II)

- Reward (R)
  - The immediate feedback received by the agent after taking an action in a particular state.
  - Rewards guide the agent toward desirable outcomes by reinforcing actions that lead to positive outcomes.
- Policy (π)
  - The strategy or mapping from states to actions that the agent follows.
  - A policy can be deterministic ($\pi(s) = a$) or stochastic ($\pi(a \mid s)$).
- Value Function ($v(s)$)
  - A function that estimates the expected cumulative reward (return) of being in a state s and following a certain policy.
  - The value function helps the agent assess the long-term benefit of states beyond the immediate reward.
- Action-Value Function ($q(s, a)$)
  - A function that estimates the expected cumulative reward (return) of taking action a in state s and then following a certain policy.
  - The action-value function helps the agent choose the best action to take in a given state.

Common features between MDP and RL

- Sequential Decision-Making: Both MDP and RL are frameworks for modeling sequential decision-making. An agent interacts with an environment over a series of time steps, making decisions (actions) that influence future states and rewards.
- State, Action, and Reward: Both MDP and RL involve the concepts of state (s), action (a), and reward (r). The agent' s goal in both frameworks is to maximize the cumulative reward (or return) over time by choosing actions that lead to favorable outcomes.
- Policy: In both MDP and RL, the agent follows a policy (π), which is a strategy that maps states to actions. The optimal policy (π^*) is the one that maximizes the expected cumulative reward from any given state.
- Value Function: Both frameworks use value functions to evaluate the expected future rewards. The state-value function $V(s)$ represents the expected return starting from state s under a particular policy, while the action-value function $Q(s, a)$ represents the expected return after taking action a in state s.
- Bellman Equations: The Bellman equations, which describe the recursive relationship between the value of a state and the value of subsequent states, are central to both MDP and RL. They are used to compute the optimal value function and policy.

- **Both MDP and RL compute value function and policy.**
- **How value function and policy are computed differs: i) via complete model information (MDP) vs. ii) via interaction with environment (RL)**



Unique features in RL

- Exploration-Exploitation
  - Unlike MDP, RL requires both Exploitation and Exploration.
  - Exploration: Trying new actions to discover their effects.
  - Exploitation: Using known information to maximize reward.
  - Trade-off: Balancing exploration and exploitation is critical in RL.
- Exploration Strategies
  - Epsilon-Greedy: Simple exploration strategy where the agent occasionally chooses a random action.
  - Softmax: Probabilistic action selection based on estimated action values.
  - Upper Confidence Bound (UCB): Balances exploration and exploitation by considering uncertainty in action-value estimates.
- Challenges and Limitations
  - Sample Efficiency: RL can require a large number of interactions with the environment.
  - Stability: Training can be unstable, especially in deep RL.
  - Convergence: Ensuring convergence to an optimal policy is non-trivial in complex environments.

Exploration-Exploitation Matters!

- MDPs Do Not Require Exploration and Exploitation:
  - MDPs assume that the agent has full knowledge of the environment (i.e., transition probabilities and rewards) and can compute the optimal policy beforehand (offline).
  - Since the (pre-computed) optimal policy is known, there is no need for exploration or exploitation during execution.
- RL Requires Exploration and Exploitation:
  - RL, on the other hand, requires the agent to learn from interactions with the environment (i.e., unknown environment dynamics).
  - Consider how humans learn. Humans do not always require complete and perfect information to take actions. Rather, they use imperfect but good-enough information to learn, which is more efficient in real world.
  - The agent must explore to discover the effects of actions and exploit its knowledge to maximize rewards.
  - The balance between exploration and exploitation is critical in RL because the agent starts with incomplete information and must learn the optimal policy over time through trial and error.

Important Algorithms in RL (I)

- Monte Carlo Method:
  - Monte Carlo Policy Evaluation: Estimate value functions based on averaging returns from multiple episodes.
  - Monte Carlo Control: Use policy improvement over time to converge to the optimal policy.
- Temporal Difference Learning:
  - TD(0) Algorithm: Combines ideas from Monte Carlo methods and DP.
    Update Rule: $v(s) \leftarrow v(s) + \alpha[r + \gamma v(s') - v(s)]$
  - Bootstrapping: Using the estimated value of the next state to update the current state.
- SARSA:
  - On-Policy Learning: The agent learns the value of the policy it is actually following.
  - Update Rule: $q(s,a) \leftarrow q(s,a) + \alpha[r + \gamma q(s',a') - q(s,a)]$
- Q-Learning:
  - Off-Policy Learning: The agent learns a policy different[3] from the one it is following.
  - Update Rule: $q(s,a) \leftarrow q(s,a) + \alpha[r + \gamma \max_{a'} q(s',a') - q(s,a)]$

[3] The action different from the policy is due to random exploration

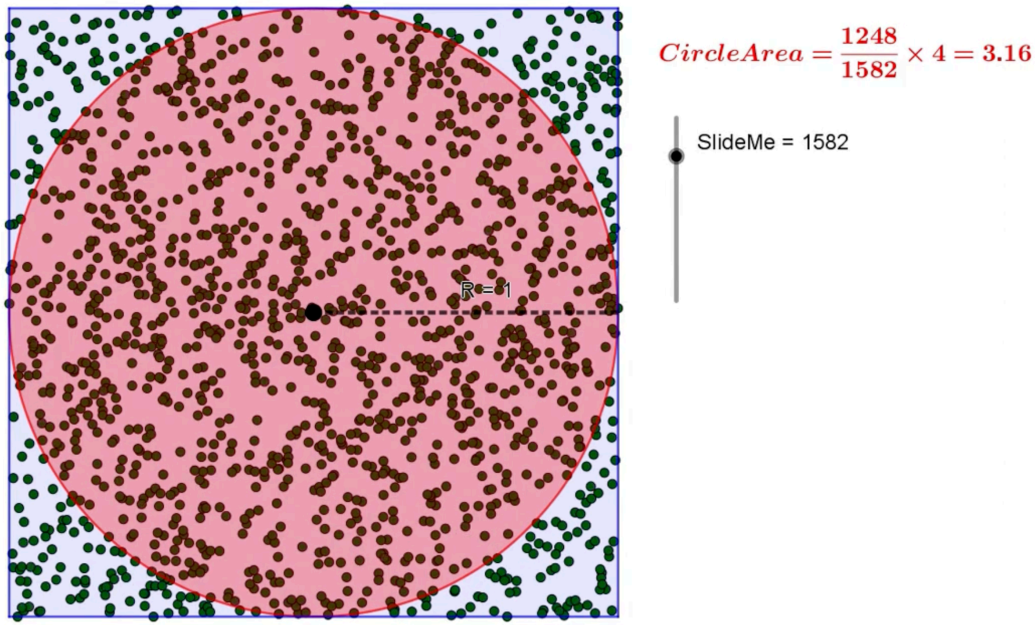
Important Algorithms in RL (II)

- Policy Gradient Method:
  - Directly optimize the policy (rather than the value function) by gradient ascent on the expected return.
  - REINFORCE Algorithm: $\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$
  - Actor-Critic Methods: Combines policy gradient (actor) with value function (critic).
- Deep SARSA:
  - An extension of the SARSA algorithm where the Q-function Q(s, a) is approximated using a deep neural network instead of a table.
- DQN:
  - Use neural networks (deep learning) to approximate the Q-function.
  - For better stability and convergence, Experience Replay and Fixed Q-Target are used.
- Other Algorithms:
  - Proximal Policy Optimization (PPO): An improvement over vanilla policy gradients with a clipped objective function for better stability.
  - Trust Region Policy Optimization (TRPO): Constrains policy updates to ensure they remain within a trust region, preventing large, destabilizing updates.

Monte Carlo Estimation (I)

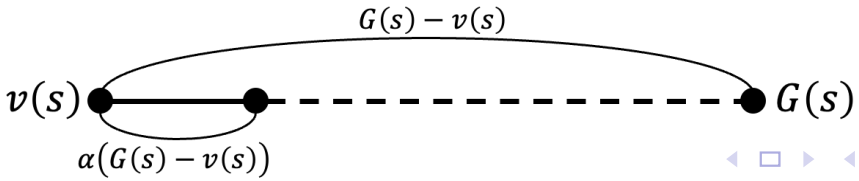
- Approximating the value rather than solving the Bellman equation for an exact value may be more efficient.
- It is especially useful when the models are unknown.
- Ramdom Sampling
  - Random Sampling involves selecting random samples from a probability distribution or dataset to approximate a quantity that might be difficult or impossible to compute exactly.
  - (Try it) Computing the area of a circle (radius $r = 1$) inscribing a square

$$Circle\ Area = \frac{The\ number\ of\ dots\ inside\ the\ Circle}{The\ total\ number\ of\ dots} \times The\ Square\ Area$$



Monte Carlo Estimation (II)

- Monte Carlo Estimation of the Value Function
  - Value Function in MDP
    $v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1})\right] = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a)(R_{t+1} + \gamma v_\pi(s'))$
  - However, in RL, this cannot be computed due to unknown models.
  - Instead, over $N(s)$ number of episodes $(1 \le i \le N(s))$,
    $v_\pi(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} G_t^{(i)}(s)$
    where $G_t^{(i)}(s)$ is the (actual) return of the i-th episode following the policy π.
  - Similar to Bellman equation, the equation can be put in a recursive form.
  - Update rule: $v_\pi(s) \leftarrow v_\pi(s) + \alpha\left(G - v_\pi(s)\right)$
    (proof) $v_{n+1}(s) = \frac{1}{n} \sum_{i=1}^{n} G_i = \frac{1}{n}\left(G_n + \sum_{i=1}^{n-1} G_i\right)$
    $= \frac{1}{n}\left(G_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} G_i\right) = \frac{1}{n}\left(G_n + (n-1)v_n(s)\right)$
    $= \frac{1}{n}\left(G_n + n v_n(s) - v_n(s)\right) = v_n(s) + \frac{1}{n}\left(G_n - v_n(s)\right)$
  - At n-th iteration with the current value, perform the n-th episode to get G_n. The error between G_n and the current estimate of value $(G_n - v_n(s))$ is used to update the next value.
  - In general, α, rather than $1/n$, is used to represent the *learning rate*.



Monte Carlo Estimation (III)

- Update Rule
  - Like the Bellman equation, the update rule is in a recursive form.
  - Unlike the Bellman equation, the update rule is similar to gradient descent.
- Learning Rate
  - Appropriate learning rate (α) should be chosen for stability and convergence.
  - α can be $1/n$ or some constant.
- Convergence
  - There is no convergence proof like in Bellman equation (i.e., contraction mapping).
  - Law of Large Numbers: As the number of episodes increase, the average return used in the Monte Carlo estimation converges to the true expected return, provided the conditions on the learning rate and exploration are met.
- Exploration
  - The agent must continue to explore the environment sufficiently so that every state-action pair is visited infinitely often. This ensures that the value estimates are updated with new information over time.
  - Without sufficient exploration, some state-action pairs might not be updated often enough, leading to biased or incorrect value estimates.

Temporal Difference (I)

- Problems with Monte Carlo Method
  - Monte Carlo methods estimate the value function based on the actual returns observed only after the agent has reached a terminal state.
  - Delayed Updates: Monte Carlo methods require the agent to wait until the end of an episode to update the value function, which can be inefficient, especially in long or continuing tasks where episodes might be long or not well-defined.
  - Ex) You may know if your move is good or not only after your Go game ends.
- Temporal Difference Learning
  - Temporal Difference learning combines ideas from MC and DP. TD learning updates the value function after every step, using the observed reward (e.g., actual experience) and the estimated value (e.g., expected return) of the next state.
  - Immediate Updates: In TD learning, the value function is updated after each action, without needing to wait for the end of the episode. This allows the agent to learn from partial sequences of experiences, making the learning process more efficient.
- Update rule: $v_\pi(s) \leftarrow v_\pi(s) + \alpha\left(R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(s)\right)$
  (proof) From MC, $v_\pi(s) \leftarrow v_\pi(s) + \alpha\left(G - v_\pi(s)\right)$
  From DP, $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$

Temporal Difference (II)

- Bootstrapping
  - Definition: Bootstrapping refers to the idea of updating an estimate based on other estimates. In TD learning, the value of a state is updated using the estimated value of the next state, rather than waiting for the final outcome.
  - Monte Carlo methods do not bootstrap: they use the actual return from the entire episode.
  - TD methods bootstrap: they update the value of a state using the estimated value of the subsequent state.
- Temporal Difference
  - The term $R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$ is called the temporal difference error (TD error).
  - It represents the difference between the predicted value of the state $v(S_t)$ and the actual observed value $R_{t+1} + \gamma v(S_{t+1})$.

$$v_\pi(s) \leftarrow v_\pi(s) + \alpha \left(R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(s) \right)$$

Temporal Difference (III)

- Efficiency
  - TD learning updates the value function after every action, which can make the learning process faster and more responsive to changes in the environment.
- Applicability to Continuing Tasks
  - Since TD learning does not require episodes to terminate, it can be applied to continuing tasks (tasks without a clear end).
- Combines Benefits of DP and MC
  - TD learning combines the bootstrapping of dynamic programming (using estimates to update estimates) with the model-free nature of Monte Carlo methods (learning directly from experience without needing a model).
- Low Variance
  - TD learning typically has lower variance than Monte Carlo methods because it does not rely on the complete return, which can vary widely, but rather on a more immediate reward and the estimate of the next state' s value.
- Extensions of Temporal Difference Learning
  - SARSA: An on-policy TD control algorithm that updates the Q-value based on the action actually taken in the next state.
  - Q-Learning: An off-policy TD control algorithm that updates the Q-value based on the maximum estimated future rewards.

Contents

1 Overview

2 Details of RL

3 RL Algorithms

4 Details of Deep RL

5 Deep RL Algorithms

6 Experiments

7 Summary and Further Discussion

SARSA (I)

- Recall that Policy Iteration in DP has two parts:
  - Policy Evaluation: Value function is evaluated and updated multiple times until convergence.
  - Policy Improvement: Policy is improved based on the updated value function once using greedy search.
- Generalized Policy Iteration (GPI): In Policy Evaluation, the value function can be evaluated and updated only once regardless of convergence. Then, Policy Improvement is performed.
- Reinforcement Learning uses GPI.
- Try to apply TD. $v_\pi(S_t) \leftarrow v_\pi(S_t) + \alpha\left(R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(S_t)\right)$
  - To use TD update rule, next state S_{t+1} is needed, which requires the action (or policy).
  - Rather than keeping a policy function, ϵ-Greedy policy is more often used.
  - Since value function has action information only implicitly, using Q function is more efficient (note that both are EQUIVALENT!).
- Equivalent TD update rule for Q function:
  $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right)$

SARSA (II)

- TD update rule for Q function:
  $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right)$
- Even if equivalent, using Q function is more efficient than value function:
  - By using the Q-function (i.e., action-value function), the agent can easily select the action that maximizes the expected reward in any given state: $a = \arg\max_a q(s, a)$.
  - This makes the Q-function more practical for determining the optimal policy.
  - The state-value function $v(s)$ only gives the expected cumulative reward for being in state s and following the policy, without directly associating specific actions with their rewards. To make decisions, the agent would still need to consider the rewards of all possible actions separately.
- Exploration
  - RL agents learn about the value only via interaction, whereas MDP agents know the value for all states due to complete model information.
  - In RL, if agents do not visit all states sufficiently many times, value function may not converge to the true value function.
  - If the agent only exploits what it currently knows (usually incomplete information), it may have biases and get stuck in a local optimum—a policy that is good but not the best. Exploration allows the agent to discover potentially better strategies.
- Exploitation
  - Exploitation involves using the agent's current knowledge to select actions that maximize immediate or cumulative rewards.
  - The agent leverages what it has learned to make the best possible decisions based on its current understanding of the environment.

SARSA (III)

- Exploration-Exploitation Trade-off
  - The agent must balance exploration and exploitation throughout the learning process.
  - Too much exploration can lead to suboptimal performance because the agent may waste time trying inferior actions.
  - Too much exploitation can prevent the agent from discovering better strategies.
- Dynamic Balance:
  - Early in the learning process, agents may explore more to gather information.
  - As the agent learns more about the environment, it might shift towards exploitation, using its acquired knowledge to maximize rewards.
- ϵ-greedy Policy:
  - A common balanced method is the ϵ-greedy strategy, where the agent explores with a small probability (ϵ) and exploits with the remaining probability (1-ϵ).
  - This probability can be a constant or decrease over time as the agent becomes more confident in its knowledge.

$$\pi(s) = \begin{cases} \arg\max_{a \in A} q(s,a), & 1-\epsilon \\ \text{random action}, & \epsilon \end{cases}$$

- On-Policy Algorithm:
  - SARSA learns the value of the policy that the agent is currently following.
  - It is generally safer and more stable, as they consider the actual actions.
  - It may converge more slowly than off-policy methods, as they update based on potentially suboptimal actions.

SARSA (IV)

SARSA Algorithm

SARSA stands for State-Action-Reward-State-Action. $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$

- Temporal Difference Learning
- ϵ-greedy policy
- On-Policy Algorithm

1. Initialize $q(s, a)$ arbitrarily;
For example, $q(s, a) = 0$ (or small random numbers) for all s and a ;
2. Start an Episode:;
Begin in an initial state S_0;
Choose an action A_0 based on ϵ-greedy policy;
3. Step Through the Environment:;
while *until episode ends* **do**
    Take action A_t based on ϵ-greedy policy;
    Observe the reward R_{t+1} and the next state S_{t+1}.;
    Choose the next action A_{t+1} based on ϵ-greedy policy;
    Update the Q-value using the SARSA update rule:;
    $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right]$;
    Set $S_t \leftarrow S_{t+1}$ and $A_t \leftarrow A_{t+1}$;
end

Features of SARSA

- Advantages of SARSA
  - Safe Exploration: Q-values is based on the policy's actual actions including exploratory actions
  - Simplicity and Intuition: Relatively simple to implement and understand
  - Convergence: SARSA is guaranteed to converge under certain conditions, such as if all state-action pairs are visited infinitely often, the learning rate decreases appropriately, and the environment is Markovian.
- Limitations of SARSA
  - Slower Convergence: Since Q-values is based on the actual actions taken (which may be exploratory), it can converge more slowly than off-policy methods like Q-learning
  - Exploration Dependency: The quality of the learned policy in SARSA can be heavily dependent on the exploration strategy used. Poor exploration strategies can lead to suboptimal policies.
- Comparison with Q-Learning (Off-Policy):
  - Updates the Q-value using the action that maximizes the Q-value in the next state, regardless of the action actually taken (off-policy).
  - Often converges faster and finds more optimal policies because it always updates towards the best possible action.

Q-Learning (I)

- From SARSA to Q-Learning
  - SARSA is an on-policy as the learning is based on the actual actions.
  - Q-Learning is an off-policy as the learning is based on the optimal actions.
- Q-Learning
  - Off-Policy Algorithm: Updates Q-values based on the maximum possible reward in the next state, independent of the agent' s actual action.
  - Agent's actual action is based on ϵ-greedy policy.
  - Update Rule:
    $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a'} q(S_{t+1}, a') - q(S_t, A_t)\right]$
- Benefits of Q-Learning
  1. Faster Convergence: Q-Learning often converges more quickly because it aggressively seeks out the optimal policy.
  2. Exploration-Insensitive: In environments where exploration strategies are less risky, Q-Learning is preferred because it prioritizes finding the best possible actions.
  3. Learning Robust Policies: Q-Learning is ideal when the goal is to learn the optimal policy, particularly in environments where exploration does not heavily penalize the agent.

Q-Learning (II)

Q-Learning Algorithm

Q-Learning algorithm is almost the same as SARSA except that Q-Learning learns the value based on the optimal actions rather than the actual actions. As a result, the required sample for Q-Learning is $(S_t, A_t, R_{t+1}, S_{t+1})$ rather than $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ for SARSA.

- Temporal Difference Learning
- ϵ-greedy policy
- Off-Policy Algorithm

1. Initialize $q(s, a)$ arbitrarily;

For example, $q(s, a) = 0$ (or small random numbers) for all s and a ;

2. Start an Episode:;

Begin in an initial state S_0;

Choose an action A_0 based on ϵ-greedy policy;

3. Step Through the Environment:;

while *until episode ends* **do**

  Take action A_t based on ϵ-greedy policy;

  Observe the reward R_{t+1} and the next state S_{t+1}.;

  Update the Q-value using the Q-Learing update rule:;

  $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a'} q(S_{t+1}, a') - q(S_t, A_t)\right]$;

  Set $S_t \leftarrow S_{t+1}$;

end

Contents

1 Overview

2 Details of RL

3 RL Algorithms

4 Details of Deep RL

5 Deep RL Algorithms

6 Experiments

7 Summary and Further Discussion

Limitations of Traditional RL

- Scalability Issues:
  - Tabular Methods: Traditional RL methods, like Q-Learning and SARSA, rely on tabular representations of the state-action space, which become impractical as the number of states and actions increases. (grid world: $5 \times 5 \times 5$)
  - Exponential Growth: In environments with large or continuous state spaces, the size of the Q-table grows exponentially, making it computationally infeasible to store and update all possible state-action pairs. (grid world with finer state-action spaces: $100 \times 100 \times 100$ or dangerous objects are moving)
- Inefficiency in Complex Environments:
  - Handcrafted Features: Traditional RL methods often require handcrafted features to represent the state space, which is time-consuming and may not capture all the nuances of complex environments.
  - Limited Generalization: Hard to generalize to new, unseen states because they lack the ability to learn complex patterns directly from raw input data.
- Inability to Handle High-Dimensional Input:
  - Raw Input Processing: Traditional RL cannot effectively process high-dimensional inputs like images, videos, or sensor data, which are common in many real-world applications (e.g., robotics, autonomous driving).

The Power of Deep RL

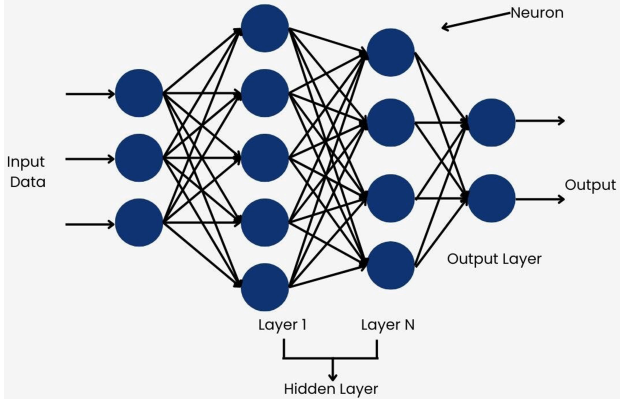
- Function Approximation with Deep Neural Networks (DNNs):
  - Deep Q-Networks (DQN): By integrating DNNs with Q-Learning, Deep RL replaces the Q-table with a neural network that can approximate the Q-function over large or continuous state spaces.
  - Generalization: DNNs can generalize across similar states, allowing agents to learn effective policies in complex and high-dimensional environments.
- Handling High-Dimensional and Raw Input:
  - End-to-End Learning: Deep RL can process raw sensory inputs like images directly, without the need for manual feature extraction, enabling end-to-end learning from perception to action.
  - Complex Task Mastery: This capability allows Deep RL to excel in tasks involving high-dimensional data, such as playing video games from pixel inputs or controlling robots using camera feeds.
- Scalability and Flexibility:
  - Continuous Spaces: Deep RL methods can handle continuous action and state spaces using techniques like policy gradients and actor-critic methods, which are beyond the reach of traditional RL algorithms.
  - Transfer Learning: Deep networks enable transfer learning, where knowledge gained in one task or environment can be leveraged to accelerate learning in a different but related task.

Adoption of Deep Learning

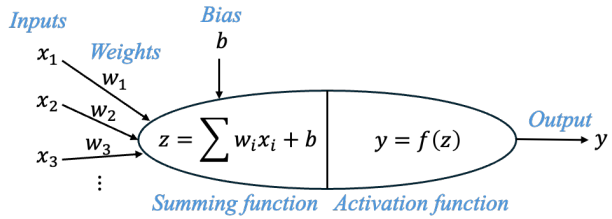
- Deep learning is adopted in RL
  - A subset of machine learning that uses neural networks with many layers to model complex patterns in data. It excels at learning from large datasets, particularly when the data is high-dimensional, such as images, text, or audio.
  - Automatic Feature Extraction: Deep learning models automatically learn to extract relevant features from raw data.
  - Handling Complex and High-Dimensional Data:Deep learning can process and learn from high-dimensional data like images, video, and sound, which are challenging for traditional machine learning methods.
- Why Deep Learning is Essential for Deep RL:
  - Handling Large State and Action Spaces: Deep learning is used as a function approximator to approximate complex functions like the Q-function in Q-Learning or the policy function in policy gradients.
  - Learning from Raw Sensory Inputs (End-to-End Learning): Deep RL agents can learn directly from raw inputs like pixels in images (e.g., in Atari games) or sensor data in robotics, without needing manual feature extraction. The deep network processes these inputs to learn policies that map states to actions.
  - Combining Perception and Decision-Making: Using DL, you can integrate perception (e.g., visual recognition) with decision-making, enabling agents to understand and interact with complex environments in a more human-like way.

Structure of Deep Learning

- Layers of a Neural Network:
  - Input Layer: Receives the input data.
  - Hidden Layers: Intermediate layers that perform feature extraction and transformation. Deep networks have multiple hidden layers.
  - Output Layer: Produces the final prediction or decision, such as class labels in classification tasks.



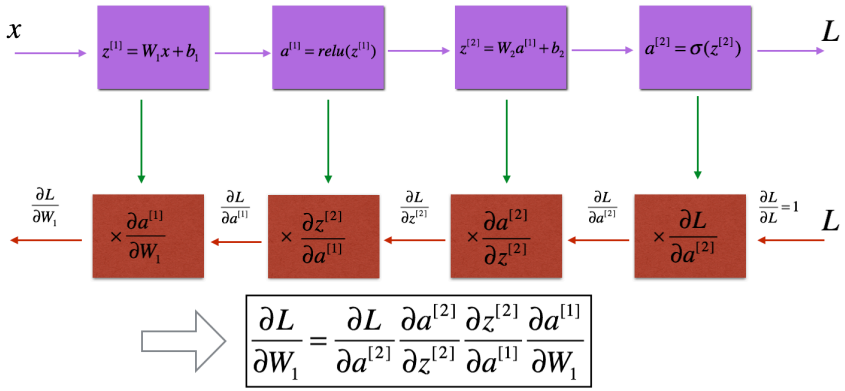
- Neurons and Activation Functions:
  - Neurons: The basic units in a network that take inputs, apply a linear transformation, and pass the result through an activation function.
  - Activation Functions:
    - ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$
    - Leaky ReLU: A variant of ReLU that allows a small, non-zero negative gradient.
    - Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
    - Tanh (Hyperbolic Tangent): $f(x) = \tanh(x)$
    - Softmax: Converts outputs into probabilities $f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$



Structure of a single neuron (or node, cell)

Deep Learning Process

- Deep Learning Process
  - Forward Propagation: Input data is passed through the network layer by layer, with each layer applying its weights and activation functions to transform the data into an output
  - Backward Propagation (Backpropagation): The error (difference between the predicted output and the actual output) is propagated backward through the network to update the weights, minimizing the error using optimization algorithms like gradient descent.

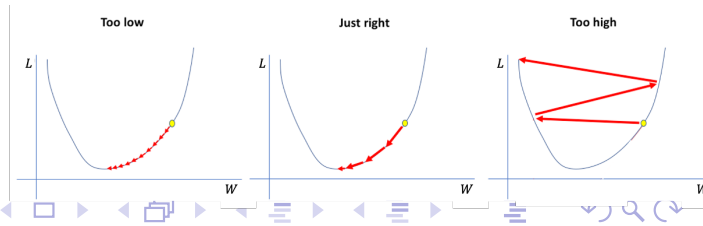


- Training a Neural Network:
  - Loss Function: A function that measures how far the predictions are from the actual targets. Common loss functions include mean squared error for regression tasks and cross-entropy loss for classification tasks.
  - Optimization:
    - Gradient Descent: An optimization algorithm used to minimize the loss function by iteratively adjusting the network's weights.
    - Learning Rate (α): A hyperparameter that controls the step size in the gradient descent process. A smaller learning rate ensures gradual convergence, while a larger one speeds up training but may overshoot the optimal point.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where y_i: the actual target value
$\hat{y}_i$: the predicted value.

$$W_{\text{new}} = W_{\text{old}} - \alpha \frac{\partial L}{\partial W}$$



How Deep Learning is used in Deep RL

- Approximating the Q-Function with Deep Learning
  - Q-Function Approximation: Applicable to SARSA or Q-Learning
  - For example, SARSA update rule is
    $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right)$
  - Target value: $R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$
  - Predicted value: $q(S_t, A_t)$
  - Q-Function Approximatotr: A deep learning model with the following loss function
    Loss $= \left(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right)^2$
- Approximating the Policy with Deep Learning
  - Policy Approximation: Applicable to policy gradient methods and Actor Critic architectures.
  - Deep neural networks are used to directly approximate the policy $\pi(a \mid s)$ without the need to know the values.

Sample Deep Learning using Keras in Python

- Keras: A Deep Learning Library in Python
  - Keras is an open-source deep learning library written in Python. It provides a high-level, user-friendly API for building and training deep learning models.
  - Key Feature: Keras is designed for ease of use, modularity, and quick prototyping, making it accessible for both beginners and experts in machine learning.
  - Keras can run on top of various deep learning backends, including TensorFlow, Theano, and CNTK. This means you can leverage the computational power of different frameworks while using the same Keras code.

Keras Code Snippet

```
from keras.layers import Dense
from keras.models import Sequential

x_train = []
y_train = []

model = Sequential()
model.add(Dense(30, input_dim=5, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(5, activation='linear'))
model.compile(loss='mse', optimizer='Adam(lr=0.001)')

model.filt(x_train, y_train, batch_size=32, epoch=1)
```

Contents

1 Overview

2 Details of RL

3 RL Algorithms

4 Details of Deep RL

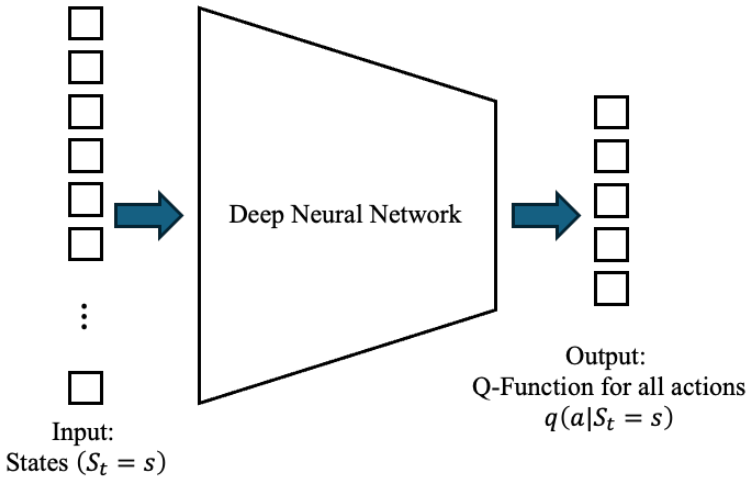
5 Deep RL Algorithms

6 Experiments

7 Summary and Further Discussion

Deep SARSA

- Note the details of SARSA from the previous pages.
- Deep Neural Network for Q-Function Approximation:
  - Instead of a Q-table, Deep SARSA uses a deep neural network to approximate the Q-function, $q(s, a)$.
  - The network takes the state s as input and outputs Q-values for all possible actions a.
  - In other words, the output of the deep learning model is $q(a) = q(a \mid S_t = s)$



Deep Neural Network for Deep SARSA

- Action Selection and Q-Value Update:
  - The action a is selected using a policy derived from the current Q-function (e.g., ϵ-greedy).
  - The Q-value is updated using the Deep SARSA update rule (NOT used explicitly):
    $q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)\right]$
    Target output for the deep learning model: $R_{t+1} + \gamma q(s_{t+1}, a_{t+1})$

Code Snippet for Deep SARSA

Deep SARSA Update Code Snippet

```
def build_model(self):
    model = Sequential()
    model.add(Dense(30, input_dim=self.state_size, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
    return model

def train_model(self, state, action, reward, next_state, next_action, done):
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

    target = model.predict(state)
    discount_factor=self.discount_factor
    target[action] = reward + discount_factor * self.model.predict(next_state)[next_action]

    self.model.fit(state, target, epoch=1, verbose=0)

def get_action(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    else:
        q_values = self.model.predict(state)
        return np.argmax(q_values)
```

DQN

- Limitations of Deep SARSA
  - On-Policy Learning: Deep SARSA, like traditional SARSA, is on-policy, which means it updates its Q-values based on the actions actually taken by the agent. This can result in slower learning in complex environments where exploration is required.
  - Instability in Training: While Deep SARSA uses deep neural networks to approximate Q-values, it can suffer from instability during training, particularly due to correlated updates[4] and non-stationary target values.
- Remedy to the Limitations: DQN
  - Deep Q-Networks (DQN) is an off-policy RL algorithm that combines Q-Learning with deep neural networks to approximate the Q-function.
  - By introducing Experience Replay and Target Network, issues with biased and unstable learning can be fixed.
  - Developed by DeepMind, DQN was the first to demonstrate the capability of deep learning to solve complex reinforcement learning problems, such as playing Atari games at a superhuman level.

[4] Like in Deep SARSA, if values are updated based on inappropriate or biased information, further updates may become worse or get stuck.

Key Innovations in DQN

- Experience Replay:
  - Stores the agent's experiences or sample (state, action, reward, next state) in a replay buffer and randomly samples mini-batches.
  - Breaks Correlation: By randomly sampling experiences from the replay buffer, DQN reduces the correlation between consecutive training samples, leading to more stable learning.
  - Efficient Use of Data: Allows the agent to learn from past experiences multiple times, improving sample efficiency.
- Target Network:
  - Uses a separate target network with fixed parameters for calculating the target Q-value during updates, reducing the risk of divergence and further stabilizing training.
  - Stabilizes Learning: The target network is used to calculate the target Q-values and is updated less frequently. This reduces the risk of unstable updates that can arise from the moving target problem in Q-learning.

DQN Update Rule

- DQN Update Rule
  - Theoretical update rule is given as follows:
    $q(s_t, a_t; \theta) \leftarrow q(s_t, a_t; \theta) + \alpha \left[R_{t+1} + \gamma \max_{a'} q(s_{t+1}, a'; \theta^-) - q(s_t, a_t; \theta)\right]$
    where θ : The parameters of the current network.
    θ^- : The parameters of the target network, which are periodically updated to match θ for stability.
- Deep Learning Implementation
  - The loss function is based on the temporal difference (TD) error:
    Loss $= \left[R_{t+1} + \gamma \max_{a'} q(s_{t+1}, a'; \theta^-) - q(s_t, a_t; \theta)\right]^2$
    where $R_{t+1} + \gamma \max_{a'} q(s_{t+1}, a'; \theta^-)$: target value
    $q(s_t, a_t; \theta)$: predicted value
  - Experience relays in addition to the current sample are used during training as training data.

Code Snippet for DQN (I)

DQN Update Code Snippet

```
def build_model(self):
    model = Sequential()
    model.add(Dense(24, input_dim=self.state_size, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(24, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(self.action_size, activation='linear', kernel_initializer='he_uniform'))
    model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
    return model

# at each move
def train_model(self, state, action, reward, next_state, next_action, done):
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

    mini_batch = random.sample(self.memeory, self.batch_size)

    for i in range(self.batch_size):
        states[i] = mini_batch[i][0]
        actions.append(mini_batch[i][1])
        rewards.append(mini_batch[i][2])
        next_states[i] = mini_batch[i][3]
        dones.append(mini_batch[i][4])

    for i in range(self.batch_size):
        if done[i]:
            target[i][actions[i]] = rewards[i]
        else:
            target[i][actions[i]] = rewards[i] + self.discount_factor * (np.amax(target_val[i]))

    self.model.fit(states, target, batch_size = self.batch_size, epoch=1, verbose=0)
```

Code Snippet for DQN (II)

DQN Update Code Snippet

```
def get_action(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    else:
        q_values = self.model.predict(state)
        return np.argmax(q_values)

def append_sample(self, state, action, reward, next_state, done):
    self.memory.append((state,action,reward,next_state,done))

# right after each episode is done
def update_target_model(self):
    self.target_model.set_weights(self.model.get_weights())
```

Policy Gradient Methods

So far, we have studied value-based RL algorithms where deep learning models approximate the values (e.g., Q functions). However, policy-based RL algorithms are also available.

Policy Gradient Methods

Policy Gradient Methods are a class of RL algorithms that optimize the policy directly. Unlike value-based methods (like Q-Learning), which learn a value function to guide the policy, policy gradient methods learn a policy that maps states directly to actions.

- Direct Policy Optimization:
  - The goal is to find the optimal policy by maximizing the expected cumulative reward (return) directly, typically by adjusting the parameters of a policy function represented by a neural network.
- Advantages:
  - Handling Continuous Action Spaces: Policy gradient methods are well-suited for environments with continuous or high-dimensional action spaces, where value-based methods struggle.
  - Stochastic Policies: These methods naturally support stochastic policies, which can be beneficial in environments where exploration and variability in actions are important.

The Policy Gradient Update Rule

- Objective:
  - The objective in policy gradient methods is to maximize the expected return $J(\theta)$, where θ represents the parameters of the policy $\pi_\theta(a \mid s)$.
- Policy Gradient Theorem (see next page:
  - The gradient of the expected return with respect to the policy parameters θ is given by:
    $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) \cdot q_\pi(s, a) \right]$
- Explanation:
  - This equation indicates that the policy can be improved by adjusting its parameters in the direction that increases the probability of actions that lead to higher returns.
- Optimization:
  - The policy parameters θ are updated using gradient ascent:
    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
    where α is the learning rate.
- Algorithms:
  - REINFORCE: One of the simplest policy gradient algorithms.
  - Actor Critic: combine the benefits of both value-based and policy-based approaches.

The Policy Gradient Theorem

$$
\begin{aligned}
J(\theta) = v_{\pi_\theta}(s) =& \mathbb{E}_{\pi_\theta}[G_t] = \sum_s d_{\pi_\theta}(s) \sum_a \pi_\theta(a \mid s) q_\pi(s,a) \\
\nabla_\theta J(\theta) = \nabla_\theta v_{\pi_\theta}(s) =& \sum_s \nabla_\theta d_{\pi_\theta}(s) \sum_a \pi_\theta(a \mid s) q_\pi(s,a) \\
&+ \sum_s d_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a \mid s) q_\pi(s,a) \\
\approx& \sum_s d_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a \mid s) q_\pi(s,a) \\
=& \sum_s d_{\pi_\theta}(s) \sum_a \pi_\theta(a \mid s) \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} q_\pi(s,a) \\
=& \sum_s d_{\pi_\theta}(s) \sum_a \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) q_\pi(s,a) \\
=& \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a \mid s) q_\pi(s,a)] \\
\therefore \theta_{t+1} =& \theta_t + \alpha \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a \mid s) q_\pi(s,a)]
\end{aligned}
$$

REINFORCE (I)

- Definition
  - REINFORCE is a Monte Carlo policy gradient algorithm used in RL.
  - It directly optimizes the policy by estimating the gradient of the expected return with respect to the policy parameters, using complete episodes of experience.
  - Monte Carlo Approach: REINFORCE relies on complete episodes to estimate the return and does not require a value function.
- Key Concepts of REINFORCE
  - Policy-Based Method: Unlike value-based methods (e.g., Q-learning), REINFORCE focuses on learning the policy directly, mapping states to probabilities of selecting each possible action.
  - Objective: The goal is to maximize the expected return $J(\theta)$, where θ represents the parameters of the policy $\pi_\theta(a \mid s)$.
- Update Rule of Policy Gradient Methods
  $\theta_{t+1} = \theta_t + \alpha \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a \mid s) q_\pi(s, a)]$
- In Policy Gradient Methods Q-function is not used. $q_\pi(s, a)$ can be approximated by the return.
- Expectation can approximated by sampling.
  $\theta_{t+1} = \theta_t + \alpha[\nabla_\theta \log \pi_\theta(a \mid s) G_t]$
  where G_t is the cumulative return from time step t to the end of the episode.

REINFORCE (II)

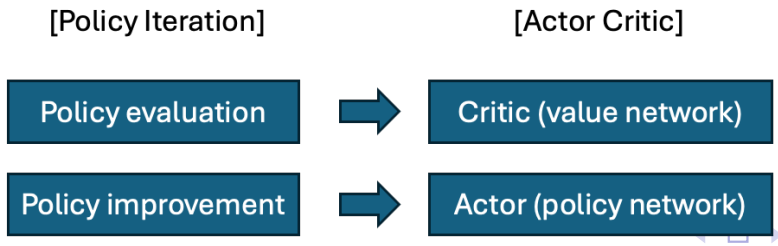
REINFORCE Algorithm

1. Initialization:;
Initialize the policy parameters θ randomly.;
2. Episode Generation:;
while *Policy converging* **do**
    while *During each episode* **do**
        For each episode:;
        2-1. Start from an initial state s_0.;
        2-2. Sample actions according to the policy $\pi_\theta(a_t \mid s_t)$.;
        2-3. Record the sequence of states, actions, and rewards until the episode ends.;
    end
    3. Policy Update:;
    After each episode ends, update the policy parameters:;
    $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) G_t$;
end

- Advantages:
  - Simplicity: REINFORCE is an easy policy gradient method to implement and understand.
  - Effective Exploration: By using entire episodes to update the policy, REINFORCE encourages thorough exploration of the state-action space.
- Limitations:
  - High Variance: The gradient estimates can have high variance ($\rightarrow$ unstable and slow learning).
  - Delayed Updates: The policy is updated only after the entire episode is completed, which can be inefficient in long or complex environments.

Actor Critic (I)

- From REINFORCE to Actor Critic
  - Reducing Variance [Critic's Role]: The Actor-Critic algorithm introduces a critic to estimate the value function (e.g., $v(s)$ or $q(s, a)$), which helps to reduce the variance of the policy gradient updates by providing a more informed and stable feedback signal.
  - Continuous Learning [Step-by-Step Updates]: Unlike REINFORCE, which waits for the entire episode to complete, Actor-Critic can update the policy at each time step. This allows for more efficient learning and quicker convergence.
  - Handling Complex Environments [Applicability]: Actor-Critic methods are more flexible and scalable, making them suitable for complex environments with large or continuous action spaces, where REINFORCE may struggle.
- Features of Actor Critic
  - Hybrid Approach: Actor-Critic combines the best of both worlds: the policy optimization of REINFORCE (actor) and the value estimation of value-based methods (critic).
  - Versatility and Efficiency: This combination leads to more robust and efficient learning, paving the way for more advanced algorithms like A2C, A3C, and PPO.



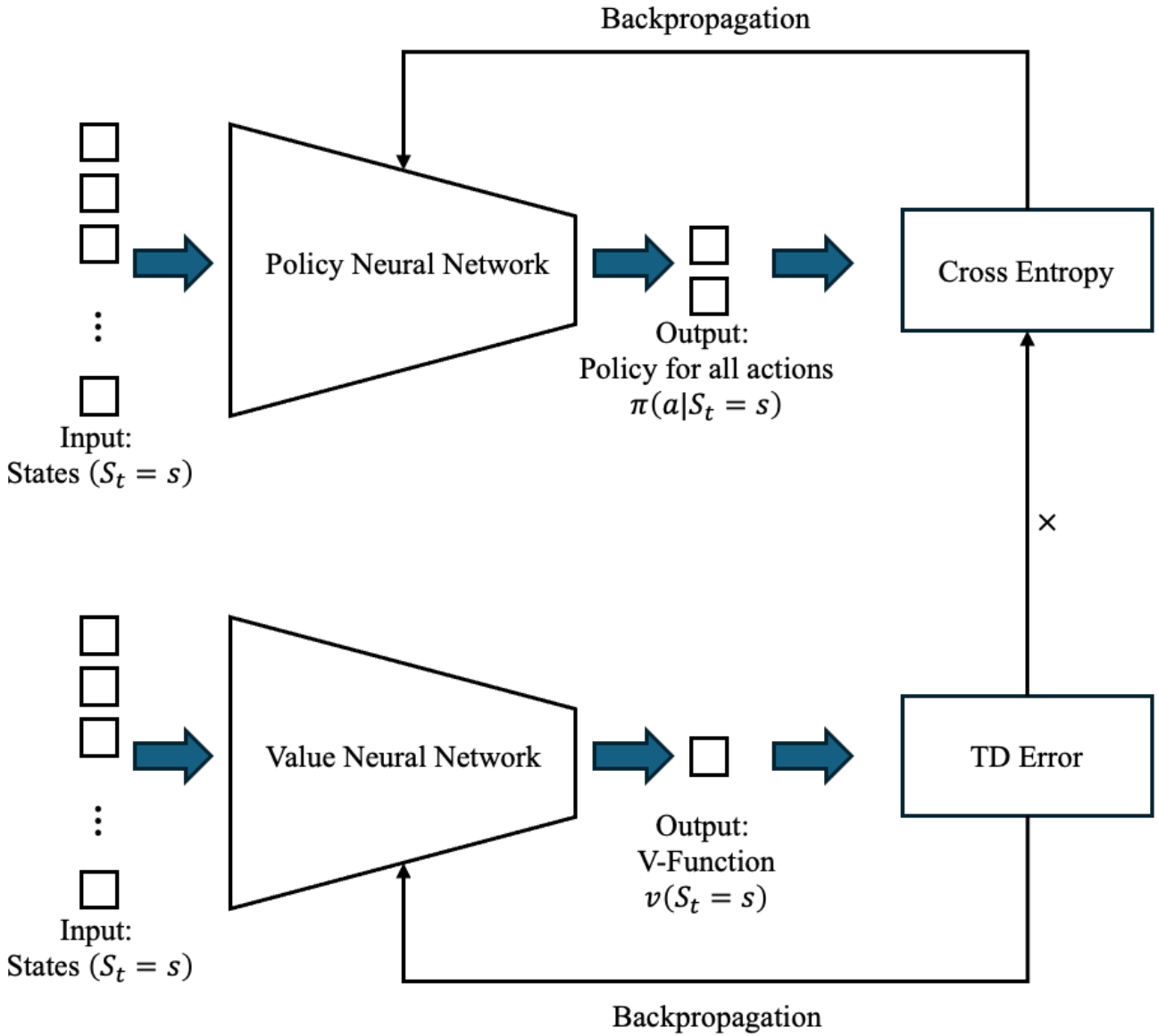
Actor Critic (II)

- Hybrid Policy Gradient Approach
  - Consider the the policy gradient update rule:
    $\theta \leftarrow \theta + \alpha[\nabla_\theta \log \pi_\theta(a \mid s) q_\pi(s, a)]$
    where θ is the weights for the policy network.
  - REINFORCE replaced $q_\pi(s, a)$ with the return G_t
  - Actor-Critic uses another network (i.e., value network) to approximate $q_\pi(s, a)$
- Advantage Function
  - Note that the Q function or the return may cause high variance in the updates due to long excursion of each episode.
  - To reduce the variance, a baseline value can be subtracted, usually the value function:
    $A(s, a) = q_w(s, a) - v_v(s)$
    where w is the weight for the q network and v is the weight for the v network.
  - Since having two value networks to define the advantage function is inefficient, only one value network can be used via TD error.
    $\delta_v = R_{t+1} + \gamma v_v(S_{t+1}) - v_v(S_t)$
    here, TD error is also a advantage function.

Actor Critic (III): Components of Actor-Critic

- Actor (policy network):
  - Role: The actor is responsible for selecting actions according to the policy $\pi_\theta(a \mid s)$.
  - Parameter Update: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot \delta_v$
  - Explanation: The actor's parameters θ are updated to increase the probability of actions that the critic deems advantageous.
- Critic (value network):
  - Role: The critic evaluates the chosen actions by estimating the value function $v_\pi(s)$.
  - Critic's Objective: To minimize the temporal difference (TD) error: $\delta_t = R_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)$
  - Value Function Update: $v(s_t) \leftarrow v(s_t) + \beta \delta_t$
  - Instead of the above update rule, deep learning is used!
  - Explanation: The critic's parameters are updated to reduce the error in estimating the value function.

Actor Critic (IV)



Actor Critic (V)

Actor Critic Algorithm

1. Initialization:;

Initialize the policy parameters θ (actor) and value function parameters v (critic).;

while *policy converging or the learning objective is being met* **do**

2. For each time step t:;
    1. Observe state s_t.;
    2. Actor selects action a_t based on policy $\pi_\theta(a_t \mid s_t)$.;
    3. Execute action a_t, receive reward R_{t+1}, and observe the next state s_{t+1}.;
    4. Critic evaluates the action by computing the TD error:;

    $\delta_t = R_{t+1} + \gamma v(s_{t+1}) - v(s_t)$;

    5. Critic updates the value function parameters v (train the value network) using the TD error.;
    6. Actor updates the policy parameters θ using the TD error as the advantage estimate:;

    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot \delta_t$

end

- Advantages:
  - Lower Variance, Continuous Learning, Applicability to Continuous Action Spaces
- Variants of Actor Critic
  - Advantage Actor Critic (A2C)
  - Asynchronous A2C (A3C)
  - Proximal Policy Optimization (PPO)

Other Algorithms

- Value-based Methods
  - DP: VI, PI
  - Model-free: Q-learning, SARSA, DQN (Double DQN, Dueling DQN), ...
- Policy-based Methods
  - Policy Gradient Methods: REINFORCE, Stochastic Policy Gradient, Deterministic Policy Gradient (DPG)
  - Actor-Critic: A2C, A3C, PPO, Trust Region PO (TRPO), Soft AC (SAC), Deep DPG (DDPG), TD3, Generalized Advantage Estimation (GAE)
- Model-based Methods
  - DP (VI, PI)
  - Dyna-Q, Monte Carlo Tree Search (MCTS), World Models, MPC
- Hybrid Methods
  - Actor Critic and its variants
  - AlphaGo (combines MCTS with deep learning for policy/value networks)
- Distributional RL
  - Categorical DQN, Quantile Regression DQN (QR-DQN), Implicit Quantile Networks (IQN)
- Multi-Agent RL
  - Independent Q-Learning, Multi-Agent DDPG (MADDPG), Cooperative MARL, Competitive MARL, Centralized Training with Decentralized Execution (CTDE)
- Hierarchical RL
  - Option-Critic Architecture, Feudal Networks, Hierarchical AC
- Inverse RL (IRL)
  - MaxEnt IRL (Maximum Entropy IRL), GAIL (Generative Adversarial Imitation Learning), Deep IRL
- Imitation Learning
  - Behavior Cloning, DAgger (Dataset Aggregation), GAIL
- Meta-RL
  - Model-Agnostic Meta-Learning (MAML), RL^2, MetaQ-Learning
- Safe RL
  - Constrained PO (CPO), Safety-Aware RL, Shielded RL
- Off-Policy Methods
  - Q-Learning, DQN, DDPG, TD3, SAC, ...
- On-Policy Methods
  - SARSA, REINFORCE, A2C, PPO, TRPO, ...

Contents

1 Overview

2 Details of RL

3 RL Algorithms

4 Details of Deep RL

5 Deep RL Algorithms

6 Experiments

7 Summary and Further Discussion

Experiments with Toy Examples

- Static Grid World
- Dynamic Grid World
- OpenAI Gym (pendulum, cart-pole, ...)
- No enough time today. I will cover this next time.
- Please understand the whole story of how each algorithms are connected.
- Don't just use packages. Try a few simple examples by yourself from the scratch!

Contents

1 Overview

2 Details of RL

3 RL Algorithms

4 Details of Deep RL

5 Deep RL Algorithms

6 Experiments

7 Summary and Further Discussion

Summary of Reinforcement Learning (RL)

- Reinforcement Learning (RL):
  - A type of machine learning where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards.
  - Unlike supervised learning, where the correct answers are provided, RL learns through trial and error by receiving feedback from the environment.
- Key Components:
  - **Agent**: The decision-maker.
  - **Environment**: The world with which the agent interacts.
  - **States (S)**: The current situation the agent is in.
  - **Actions (A)**: Choices available to the agent.
  - **Rewards (R)**: Feedback from the environment.
  - **Policy (π)**: The strategy the agent follows to make decisions.
  - **Value Function (v)**: Measures the long-term (cumulative) reward for states.
  - **Q-Function (q)**: Measures the long-term (cumulative) reward for state-action pairs.

Summary of Important Algorithms

- Value-Based Methods:
  - SARSA: On-policy method that learns the value of the current policy.
  - Q-Learning: Off-policy method that learns the value of the optimal policy.
- Policy-Based Methods:
  - REINFORCE: Monte Carlo policy gradient method that updates the policy directly based on episodes.
  - Actor-Critic: Combines value-based and policy-based methods to reduce variance in policy updates.
- Deep RL Methods:
  - Deep SARSA: Extends SARSA by using deep neural networks for function approximation.
  - DQN: Deep Q-Networks that use neural networks to approximate the Q-function, introducing techniques like Experience Replay and Target Networks.

Conclusion

- From MDP to RL: RL addresses the limitations of Markov Decision Processes (MDPs) by learning policies directly from interactions with the environment without requiring a model of the environment.
- Value or Policy or Both: It's all about estimating values or policies or the combination.
- Importance of Exploration and Exploitation: Balancing exploration (trying new actions) and exploitation (using known actions) is crucial in RL to discover optimal strategies.
- Deep Reinforcement Learning: Combines RL with deep learning to handle high-dimensional and continuous state-action spaces, enabling applications like robotics, gaming, and healthcare.

Challenges

The same challenge in MDP, this with RL.

Consider a 1-DOF pendulum initially at rest. Using MDP framework, design a control policy to bring the pendulum upright.

- Total point mass of $1kg$, at $1m$ from the hinge
- Availble torques at the hinge are $[-2, -0.5, 0, 0.5, 2](Nm)$
- Discretize state space appropriately (angle and angular velocity). For example,
  - Twenty grids for angle $(-2\pi \leq \theta \leq 2\pi)$(rad)
  - Twenty grids for angular velocity $(-8 \leq \theta \leq 8)$(rad/s)
  - Terminal state$=(0, 0)$
  - Initial state$=(-\pi, 0)$
  - Redefine state space.
- What is the appropriate reward?

Next Topics

- **POMDP**:
  - A POMDP is an extension of an MDP where the agent does not have full observability of the current state. Instead, it receives observations that provide partial information about the state.
  - States (S), Actions (A), Transition Probabilities (P), Rewards (R), Observations (O), Observation Probabilities (Z), Discount Factor (γ)
  - Bayes rule is used to update belief states.

References

- "Reinforcement Learning: An Introduction" by Richard Sutton and Andrew Barto
- "Dynamic Programming and Optimal Control" (two volumes) by Dimitri Bertsekas
- chatGPT
- Several websites and githubs
- `https://github.com/pilwonhur/hurgroup_cc_mdp.git`

Thank you!