Lesson 7: Deep Reinforcement Learning

Deep reinforcement learning is a subfield of machine learning that builds on reinforcement learning by using deep neural networks to approximate the value function or policy of an agent. This enables the agent to learn complex and high-dimensional representations of the environment, allowing it to make decisions based on complex patterns and features. Deep reinforcement learning has been applied to a variety of domains, such as robotics and game playing, and has produced impressive results, including the development of agents that can play complex games such as Go and poker at superhuman levels.

7.1 Introduction to reinforcement learning

Reinforcement learning (RL) is a branch of machine learning that involves an agent interacting with an environment to learn how to make decisions that lead to desired outcomes or rewards. It is a type of learning that focuses on trial-and-error learning through feedback in the form of rewards or penalties.

RL is particularly useful in sequential decision-making problems, where the agent must take a sequence of actions over time to reach a desired goal. For instance, in games like chess, the agent must learn to make the best moves to win the game. In robotics, RL can be used to train robots to perform tasks like grasping objects, walking, and navigating through a space.

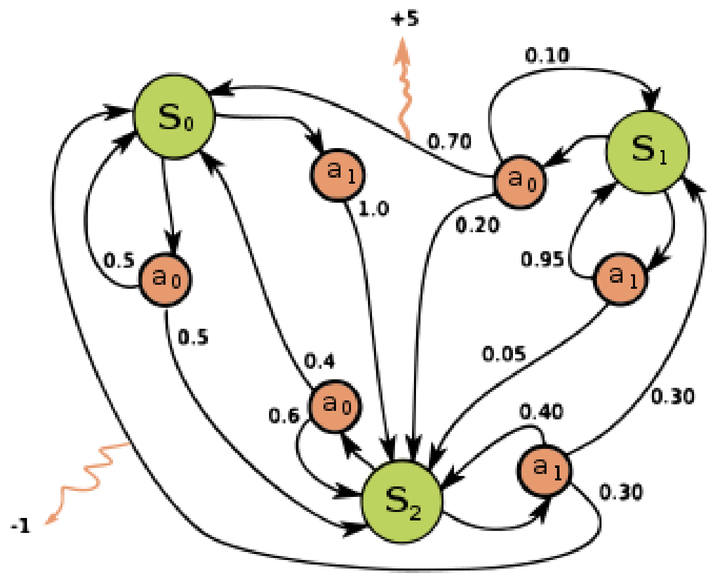
The agent interacts with the environment by taking actions based on the current state of the environment. The environment then transitions to a new state based on the agent's action, and the agent receives a reward or penalty based on the outcome of its action. The goal of the agent is to learn a policy, or a set of actions, that maximizes the cumulative reward over time.

There are various RL algorithms, including Q-learning, SARSA, and policy gradient methods. These algorithms differ in their approach to learning the optimal policy, but they all share the same goal of maximizing the cumulative reward.

Reinforcement learning has seen significant progress in recent years, with notable examples like AlphaGo and AlphaZero, which achieved remarkable performance in complex games without prior knowledge of the rules. RL has also found applications in other areas such as finance, healthcare, and education. Despite its successes, RL is still an area of active research, and challenges such as sample efficiency, exploration-exploitation trade-off, and generalization still need to be addressed.

7.2 Markov decision processes

Markov decision processes (MDPs) are a mathematical framework used to model decision-making in uncertain environments. MDPs represent decision-making problems as a sequence of states and actions where the outcome of each action is uncertain. In an MDP, an agent interacts with the environment over a series of time steps, where at each time step, the agent observes the current state of the environment and takes an action based on that state. The environment then transitions to a new state, and the agent receives a reward or penalty based on its action.



The Markov property is a key assumption in MDPs, which states that the current state of the environment contains all relevant information for predicting the future. This means that the probability of transitioning to a new state depends only on the current state and action, and not on any previous history. This assumption makes the modeling process computationally tractable and allows for the use of dynamic programming, Monte Carlo methods, or temporal difference learning to find an optimal policy.

The goal in MDPs is to learn a policy, which is a mapping from states to actions, that maximizes the expected cumulative reward over time. This policy can be learned using various algorithms, such as value iteration or policy iteration, which iteratively update the values of the state-action pairs to find the optimal policy.

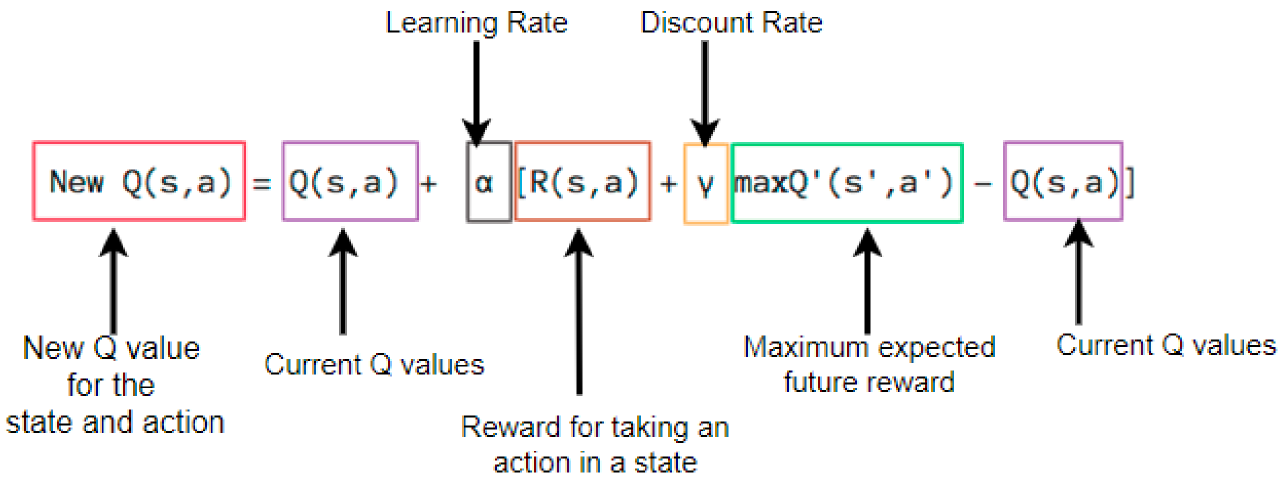
One of the main challenges in using MDPs is determining the optimal policy given the model of the environment. This is known as the planning problem and can be computationally expensive for large state spaces. However, approximate methods such as value iteration and policy iteration can be used to find a close approximation of the optimal policy.

MDPs have numerous applications in various fields, such as robotics, finance, and healthcare. For example, MDPs can be used to model financial decision-making problems such as portfolio optimization, where the goal is to maximize the expected return while minimizing the risk. In healthcare, MDPs can be used to model treatment decisions for patients with chronic diseases, where the goal is to maximize the patient's quality of life while minimizing the cost of treatment.

7.3 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm that has been widely used to solve sequential decision-making problems. The Q-value represents the expected cumulative reward that an agent will receive by taking a particular action in a particular state. Q-Learning algorithm uses the Bellman equation to update the Q-values iteratively. The Bellman equation expresses the expected cumulative reward for a state-action pair as the sum of the immediate reward and the discounted expected cumulative reward of the next state.

Q-Learning algorithm is based on the exploration-exploitation tradeoff, where the agent explores new actions in order to discover better policies, while also exploiting the current knowledge to maximize its rewards. The exploration-exploitation tradeoff is controlled by an epsilon-greedy policy, where the agent selects a random action with probability epsilon and selects the action with the highest Q-value with probability 1-epsilon.


Learning Rate
Discount Rate
New Q(s,a) = Q(s,a) + α [R(s,a) + γ maxQ'(s',a') − Q(s,a)]
New Q value for the state and action
Current Q values
Reward for taking an action in a state
Maximum expected future reward
Current Q values

Q-learning is an off-policy algorithm, which means that it learns a policy that maximizes the expected cumulative reward, even if it explores other actions that may not be optimal in the short term. The algorithm works well in discrete and finite action spaces, but it may not be suitable for problems with continuous state spaces. To address this limitation, deep Q-networks (DQN) have been developed, which use deep neural networks to approximate the Q-values and enable Q-learning in continuous state spaces.

Q-learning has been successfully applied to a variety of problems, such as game playing, robotics, and recommendation systems. It is a simple yet powerful algorithm that can learn optimal policies in large state spaces with high-dimensional inputs. However, it may suffer from the problem of overestimating Q-values, which can lead to suboptimal policies. To address this problem, several modifications to the Q-learning algorithm have been proposed, such as double Q-learning and prioritized experience replay. These modifications improve the performance of the algorithm and make it more robust to noisy and incomplete data.

CODE EXAMPLE

Here is an example code for implementing Q-Learning algorithm in Python using the OpenAI Gym library:

```
import gym
import numpy as np


env = gym.make("FrozenLake-v0")


# Initialize the Q-table with zeros
Q = np.zeros([env.observation_space.n, env.action_space.n])


# Set the hyperparameters
learning_rate = 0.8
```

```
discount_factor = 0.95
num_episodes = 2000


# Run the Q-learning algorithm
for episode in range(num_episodes):
    state = env.reset()
    done = False
    while not done:
        # Choose an action using the epsilon-greedy policy
        if np.random.uniform() < 0.1:
            action = env.action_space.sample()
        else:
            action = np.argmax(Q[state, :])

        # Take the action and observe the new state and reward
        new_state, reward, done, info = env.step(action)

        # Update the Q-table using the Bellman equation
        Q[state, action] = Q[state, action] + learning_rate * (reward
+ discount_factor * np.max(Q[new_state, :]) - Q[state, action])

        # Move to the next state
        state = new_state

    # Print the total reward for the episode
    print("Episode {}: Total reward = {}".format(episode + 1,
np.sum(rewards)))
```

In this code, we first create an instance of the FrozenLake environment using the **gym.make** function. We then initialize the Q-table with zeros, and set the hyperparameters for the Q-learning algorithm: the learning rate, discount factor, and number of episodes.

In each episode, we reset the environment to its initial state and run the Q-learning algorithm until the episode is completed. Within each episode, we use an epsilon-greedy policy to choose an action, and update the Q-table using the Bellman equation. Finally, we print the total reward for the episode.

This code can be modified to work with other OpenAI Gym environments or with custom environments, and can also be extended to implement more advanced Q-learning algorithms such as Deep Q-Networks (DQN).
