RL THEORY

Experiment 02: Implement Policy Iteration in Python

Theory:
Policy iteration is a popular algorithm in reinforcement learning for finding the optimal policy in a Markov decision process (MDP). The algorithm consists of two main steps:
Policy Evaluation: Given a policy, estimate its value function by solving the Bellman equation recursively.
Policy Improvement: Given the value function of the current policy, update the policy to be greedy with respect to the value function.
These two steps are repeated iteratively until convergence to the optimal policy is achieved. In other words, the algorithm alternates between evaluating the current policy and improving it based on the current value function.
The policy evaluation step involves solving the Bellman equation, which expresses the value function of a state as the expected sum of future rewards discounted by a factor called the discount rate. The policy improvement step involves updating the policy to select the action that maximizes the expected sum of discounted rewards, based on the current value function.
Policy iteration has been shown to converge to the optimal policy in a finite number of iterations in MDPs with finite state and action spaces. However, it can be computationally expensive, particularly for large and complex MDPs.
The value function in reinforcement learning represents the expected cumulative reward from a given state, following a particular policy. It can be calculated recursively using the Bellman equation:
where:
V(s) is the value of state s
E[ ] denotes the expected value R_t+1 is the reward received at time t+1

$$V(s) = E\,[R\_t+1 + \gamma V(s\_t+1) \mid S\_t = s]$$

γ is the discount factor (0 <= γ <= 1) that determines the importance of future rewards relative to immediate rewards
V(s_t+1) is the value of the next state s_t+1, which is obtained by applying the current policy

Experiment 03: Implement Markov Decision Model in Python

Theory:

Markov Assumption:

Given a state, an agent takes an action according to a policy. The action leads to a change in state and possibly generates a reward. One brute force way to learn a policy is to actually remember all the possible pairs of state, action and reward. But that is often not feasible. For example, problems such as playing Go, driving a car etc. makes it intractable. Therefore, in the RL problem, we make a Markovian assumption.

The Markov assumption states that, “The current state contains all the necessary information about all the past states the agent was in and all the past actions the agent took”. It assumes that the current state is sufficient for taking the next action.

Markov Decision Process:

A Markov decision process (MDP) refers to a stochastic decision-making process that uses a mathematical framework to model the decision-making of a dynamic system. It is used in scenarios where the results are either random or controlled by a decision maker, which makes sequential decisions over time. MDPs evaluate which actions the decision maker should take considering the current state and environment of the system.

The word ‘Decision’ in MDP takes into account actions taken by the agent in a given Markov state. MDP is the formal name of a sequential decision-making process. All the RL problems set its ground on MDPs, i.e., work on the assumption of the Markov property.

MDP Components:

1. **A set of states:** This represents the possible states that the system can be in.
2. **A set of actions:** This represents the possible decisions or actions that can be taken by the

agent in a given state.

3. **A transition function:** This defines the probability of moving from one state to another

state as a result of taking a particular action.

4. **A reward function:** This defines the reward or cost associated with each state and action,

and is used to evaluate the quality of different decisions.

5. **A discount factor:** This is a scalar value between 0 and 1 that determines the importance

of future rewards compared to current rewards.

Experiment 04: Implement Dynamic Programming in python.

Theory:

Dynamic programming can be used to solve reinforcement learning problems when someone tells us the structure of the MDP (i.e when we know the transition structure, reward structure etc.). Therefore dynamic programming is used for the planning in a MDP either to solve:

Prediction problem (Policy Evaluation):

Given a MDP <S, A, P, R, γ> and a policy π. Find the value function v_π (which tells you how much reward you are going to get in each state). i.e the goal is to find out how good a policy π is.

Control problem (Find the best thing to do in the MDP):

Given a MDP <S, A, P, R, γ>. Find the optimal value function v_π and the optimal policy π*. i.e the goal is to find the policy which gives you the most reward you can receive with the best action to choose.

Value Iteration

An alternative approach to control problems is with value iteration using the Bellman Optimality Equation. First we need to define how we can divide an optimal policy into its components using the Principle of Optimality.

Principle of Optimality

Any optimal policy can be subdivided into two components which make the overall behavior optimal:

- An optimal first action A∗
- Followed by an optimal policy from successor state S

Experiment 05: Apply Q-Learning Algorithm on the given dataset.
Theory:
Q-learning is a popular algorithm in reinforcement learning that enables an agent to learn how to make optimal decisions by interacting with an environment. The algorithm works by updating an estimate of the optimal action-value function, which is defined as the expected reward for taking a particular action in a particular state and following the optimal policy thereafter. At each time step, the agent observes the current state of the environment and chooses an action based on its estimate of the optimal action-value function. The action is then executed, and the agent receives a reward and observes the resulting state. The estimate of the optimal action-value function is updated using the observed reward and state transition, based on the Bellman equation, which expresses the relationship between the value of a state and the values of its neighboring states:

$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$ where $Q(s,a)$ is the estimate of the optimal action-value function for state s and action a, r is the reward received for taking action a in state s and transitioning to state s', α is the learning rate that determines the rate at which the estimates are updated, and γ is the discount factor that determines the weight given to future rewards.

The Q-learning algorithm can be used in environments with discrete states and actions or in continuous environments using function approximation techniques such as neural networks. It has been shown to be effective in a wide range of applications, from game playing to robotics and control systems. One of the advantages of Q-learning is that it does not require a model of the environment and can learn directly from experience. However, it can be sensitive to the choice of hyperparameters such as the learning rate and discount factor, and it can be prone to overestimation of the action values in certain cases. There are also several variations of Q-learning, such as double Q-learning and prioritized experience replay, that aim to address some of these issues and improve performance.

Experiment 06: Implement Bellman Equation in Python

Theory:

Bellman Equations are a set of mathematical equations used in dynamic programming to solve optimization problems. They describe the relationship between the value of a current state and the values of its neighbouring states, taking in to account the immediate reward obtained from that state and the expected future reward obtained from the neighbouring states.

There are two types of Bellman equations: The Bellman equation for the value function and the Bellman equation for the Q-Function. The Bellman equation for the value function expresses the value of a state-action pair as the sum immediate reward and the expected future reward, weighted by the probability of transitioning to each neighbouring state and selecting each possible action.

Bellman equations can be used to compare the optimal value function or Q-function, which can be used to determine the optimal policy for an agent to follow in a given environment. The value iteration algorithm is a popular iterative algorithm used to solve Bellman equations and find the optimal value function. In reinforcement learning, agents can learn the optimal policy through trial and error using techniques such as Q-learning or policy gradient methods, which use Bellman equations to update their value function estimates.

- $v^*(s) = \sum_a \pi^*(a|s) q^*(s,a)$
- $q^*(s,a) = \sum_{s'} \sum_r p(s',r|s,a)[r + \gamma v^*(s')]$

Experiment 07: Apply Monte Carlo simulation.

Theory:

The Monte Carlo method for reinforcement learning learns directly from episodes of experience without any prior knowledge of MDP transitions. Here, the random component is the return or reward.

There are two main types of Monte Carlo methods used in reinforcement learning:

Monte Carlo prediction: Monte Carlo prediction is a method for estimating the value function of a policy by averaging the returns observed during multiple episodes of interaction with the environment. The basic idea is to simulate multiple episodes of interaction with the environment using the policy to generate a set of returns, and then compute the average of these returns for each state in the state space. This provides an estimate of the value function for the given policy.

Monte Carlo control: Monte Carlo control is a method for finding the optimal policy by iteratively improving the current policy using the value function estimates obtained through Monte Carlo prediction. The basic idea is to use Monte Carlo prediction to estimate the value function for the current policy, and then update the policy to be greedy with respect to the estimated value function. This process is repeated until the policy converges to the optimal policy.