Deep Reinforcement Learning

A Practical Programming Handbook

Swagat Kumar

August 3, 2025

Contents

Acronyms

A2C Advantage Actor-Critic.

A3C Asynchronous Advantage Actor-Critic.

AC Actor-Critic.

D3QN Dueling Double Deep Q Network.

DDPG Deep Deterministic Policy Gradient.

DDQN Double Deep Q Network.

DP Dynamic Programming.

DQN Deep Q Network.

DRL Deep Reinforcement Learning.

MC Monte Carlo.

MDP Markov Decision Process.

PER Priority Experience Replay.

PG Policy Gradient.

PPO Proximal Policy Optimization.

RL Reinforcement Learning.

SAC Soft Actor-Critic.

TD Temporal Difference.

TRPO Trust Region Policy Optimization.

List of Figures

List of Algorithms

List of Tables

1. Introduction to Reinforcement Learning

1.1. What is Reinforcement Learning?

Reinforcement Learning (RL) [1] is a type of machine learning algorithm in which an agent learns to behave in an environment by trial and error. The agent receives rewards for taking actions that lead to desired outcomes, and penalties for taking actions that lead to undesired outcomes. Over time, the agent learns to take actions that maximize its expected reward.

RL is a powerful tool for learning complex behaviors in a variety of environments. It has been used to train agents to play games, control robots, and make financial decisions.

1.2. Reinforcement Learning Problem

1.2.1. The Components of a Reinforcement Learning System

The reinforcement learning framework consists of three main components:

The agent: The agent is the entity that learns to behave in the environment. It can be a physical robot, a software program, or even a human being.

The environment: The environment is the world in which the agent lives and acts. It can be physical, such as a game board or a robotic arm, or it can be virtual, such as a computer simulation.

The reward signal: The reward signal is a measure of how well the agent is doing. It is typically given to the agent after it takes an action.

The agent interacts with the environment by taking actions and receiving rewards. The goal of the agent is to learn a policy, which is a mapping from states to actions. The policy tells the agent what action to take in each state in order to maximize its expected reward.

1.2.2. Problem Formulation

The reinforcement learning problem can be formulated as a Markov Decision Process (MDP). An MDP is a probabilistic decision-making model (*Markov Chain* [2]) that solely depends on the current state to predict the next state and not the previous states. In this case, the MDP is represented as a tuple (S, A, P, R, γ), where:

- S is a set of states. Each state represents the condition of the environment at a given time t. It is a vector of variables that fully describe the environment at that time.
- A is a set of actions. Each action is the agent's choice of what to do in a given state. It could be a scalar variable or a vector that represents the agent's decision.
- $P(s|s,a)$ is the probability of transitioning from state s to state s' when taking action a.
- $R(s,a)$ is the reward received from the environment when transitioning from state s to state s' by taking action a. It is a measure of how well the agent's action performed.
- γ is a discount factor, which represents how much the agent cares about future rewards.

In addition, the RL problem includes the following components:

- **Policy:** The policy is a mapping from states to actions. It is a function that tells the agent what action to take in a given state. It is denoted by the symbol $\pi(s)$.
- **Value function:** The value function is a function that maps states to expected rewards. It tells the agent how much reward it can expect to receive from a given state by following a policy π.
- **Action-Value or Q function:** The *action-value function* of a state-action pair (s,a) under policy π is the expected return starting from state s, taking action a, and then following policy π thereafter. It is denoted by $Q^\pi(s,a)$.

The main components of a RL system is shown in the figure 1.1. The goal of the reinforcement learning agent is to find a policy that maximizes the expected return, which is the sum of all future rewards discounted by γ. This can be done by iteratively updating the value function until it converges to the optimal value function. This is accomplished mostly by using the following Bellman equations for value function and Q function respectively:

$$V^\pi(s) = \mathbb{E}_\pi[R_t + \gamma V_\pi(S_{t+1} = s')|S_t = s] \tag{1.1}$$

and

$$Q^\pi(s,a) = \mathbb{E}_\pi[R_t + \gamma Q^\pi(S_{t+1} = s', A_{t+1} = a')|S_t = s, A_t = a] \tag{1.2}$$

The Bellman equations can be used to iteratively improve the value functions and action-value functions. This aspect will be discussed in more detail in the next chapter.

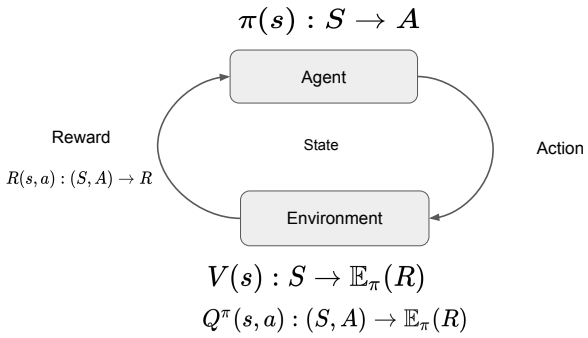


Figure 1.1.: Components of a Reinforcement Learning System. $\mathbb{E}_\pi(R)$ is the expected future reward under a given policy π.

1.3. Types of Reinforcement Learning

The reinforcement learning algorithms can be divided into two types depending how the model information is used. These are:

Model-based RL: In model-based RL, the agent learns a model of the environment. This model can be used to predict the future consequences of actions.

Model-free RL: In model-free RL, the agent does not learn a model of the environment. Instead, it learns a policy directly from its experiences.

Model-based RL is typically more efficient than model-free RL, but it is also more difficult to learn. Model-free RL is typically less efficient than model-based RL, but it is easier to learn.

Similarly, depending on how the optimal policy is learnt, the reinforcement learning algorithms could be divided into the following two types:

Value-based: In this method, the approximate value function, which is a measure of expected reward for a given action, is first learnt and then the optimal policy is derived from this value function. Q-learning, DQN, DDQN are value-based methods. These methods are more sample efficient (require lesser examples) and are more stable.

Policy-based: In these methods, the policy function is directly computed from a state without estimating their value. Policy-based methods converge more easily to a local or a global maximum and they do not suffer from oscillation. They are highly effective in high-dimensional or continuous state spaces. They can learn stochastic policies (give a probability distribution over actions). They, however, take longer time to converge. Some examples include DDPG and PPO. (Need to be better explained.)

The value-based methods that estimate Q function $Q(s, a)$ can further be divided into the following two types:

On-policy: In on-policy algorithms, the Q function $Q(s, a)$ is learned from the actions that is generating using the current policy $\pi(a|s)$.

Off-policy: In off-policy learning, the Q function $Q(s, a)$ is learned from actions generated using a different policy or no policy at all (for example, random actions).

1.4. Types of RL Environment

1.5. Reinforcement Learning Algorithms

There are many different reinforcement learning algorithms. Some of the most popular algorithms include:

- Q-learning: Q-learning is a model-free RL algorithm that learns a table of Q-values. The Q-value for a state-action pair is the expected reward for taking that action in that state.
- SARSA: SARSA is a model-based RL algorithm that learns a table of state-action-reward-state (SARS) values. The SARS value for a state-action-reward-state pair is the expected reward for taking that action in that state, given that the previous state was the given state and the previous reward was the given reward.
- Deep Q-learning: Deep Q-learning is a variant of Q-learning that uses a neural network to represent the Q-values. This allows Deep Q-learning to learn more complex policies than traditional Q-learning.
- Policy gradient methods: Policy gradient methods learn a policy directly, without learning a Q-table. Policy gradient methods are typically more efficient than Q-learning, but they can be more difficult to learn.

1.5.1. Applications of Reinforcement Learning

Reinforcement learning has been used to solve a wide variety of problems, including:

- Playing games: Reinforcement learning has been used to train agents to play games such as Atari games, Go, and Dota 2.
- Controlling robots: Reinforcement learning has been used to train robots to perform tasks such as walking, grasping, and navigation.
- Making financial decisions: Reinforcement learning has been used to make financial decisions such as trading stocks and bonds.

- Medical diagnosis: Reinforcement learning has been used to develop algorithms for medical diagnosis.
- Natural language processing: Reinforcement learning has been used to develop algorithms for natural language processing tasks such as machine translation and speech recognition.

1.6. The history of Reinforcement Learning

Reinforcement learning (RL) [1] is a subfield of machine learning that deals with how agents learn to behave in an environment in order to maximize some notion of cumulative reward. It is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

The origins of reinforcement learning can be traced back to the early work of B. F. Skinner in the 1930s. Skinner was a psychologist who studied the behavior of animals, and he developed a theory of learning called operant conditioning [3]. Operant conditioning is based on the idea that animals learn to associate certain behaviors with rewards or punishments.

In the 1950s, the first reinforcement learning algorithms were developed by computer scientists. One of the most important early algorithms was the Q-learning algorithm, which was developed by John McCain and Paul Werbos [4]. Q-learning is a value-based reinforcement learning algorithm that iteratively updates a table of state-action values.

In the 1970s and 1980s, reinforcement learning research continued to grow, but it was still a relatively niche field. However, in the 1990s, there was a renewed interest in reinforcement learning, due in part to the development of new algorithms and the availability of more powerful computers.

In the 2000s, reinforcement learning saw even more progress, due to the development of deep reinforcement learning algorithms. Deep reinforcement learning algorithms use artificial neural networks to represent the state-action values or policies. This allows them to learn much more complex tasks than traditional reinforcement learning algorithms. Readers can refer to [5] [6] [7] for more details.

In recent years, reinforcement learning has been used to achieve impressive results in a variety of domains, including game playing, robotics, and finance. For example, in 2016, the AlphaGo program developed by Google DeepMind defeated the world champion Go player, Lee Sedol [8]. This was a major breakthrough for reinforcement learning, as Go is a very complex game that was previously thought to be too difficult for computers to master.

Reinforcement learning is a rapidly growing field, and it is still full of challenges. However, the progress that has been made in recent years is very promising, and it is likely that reinforcement learning will continue to play an increasingly important role in artificial intelligence in the years to come.

2. Markov Decision Process and Dynamic Programming

2.1. Markov chain and Markov Decision process

The Markov property states that the future only depends on the present and not on the past. The Markov chain is a probabilistic model that solely depends on the current state to predict the future state and not the previous states. The Markov chain strictly follows the Markov Property. Moving from one state to another is called a **transition** and its probability is called a transition probability.

The Markov decision process (MDP) is an extension of Markov Chain used for modeling decision-making situations. A RL problem can be modeled as an MDP which is represented by a tuple (S, A, P, R, γ) where

- S is a set of states the agent can be in.
- A is a set of actions that can be performed by the agent at any state to move to another state.
- $P(s'|s, a)$ is the transition probability of moving from state s to s'.
- $R(s, a)$ is the reward received by the agent for transitioning from state s to s' by performing the action a.
- γ is the discount factor the controls the importance of immediate and future rewards.

The MDP provides a mathematical framework for solving the reinforcement learning (RL) problem as we will see in this chapter.

2.1.1. Rewards and Returns

In an RL environment, an agent interacts with an environment by performing an action and moves from one state to another. In the process, it receives a reward indicating how good or bad the action is. This reward is obtained by using a function $R(s, a) : (S, A) \to R$ where S, AR represent state (s), action (a) and reward (r) spaces respectively. The agent tries to maximize the total amount of future rewards (cumulative rewards) received from the environment starting from the current state. The total amount of future rewards obtained by the agent is called *returns* and is denoted by G_t. The return for an episodic environment is given by

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{2.1}$$

where $r_{t+1} = R(s_t = s, a_t = a)$ is the reward obtained at the next time instant as the agent transitions from state s_t to s_{t+1} by performing the current action a_t. In case of non-episodic or continuous environments (with no terminal state), the return could be computed as an infinite sum of rewards given by

$$G_t = r_{t+1} + r_{t+2} + \cdots \tag{2.2}$$

Usually, a discount factor is included to decide how much importance should be given to future and immediate rewards. Hence the above equation can be rewritten as:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.3}$$

where γ is the discount factor whose value lies between 0 and 1. A zero value of γ means that the immediate reward is more important while $\gamma = 1$ indicates the future rewards are more important than the immediate rewards.

2.1.2. The policy function

The policy function is a mapping from agent's state space to action space. It is represented as $\pi(s) : S \rightarrow A$. The policy function is used to decide what action to take in each state. The goal of the agent is to learn the optimal policy that would maximize the total reward obtainable for a given task. The policy can be deterministic or stochastic. In deterministic policy, action is defined as a function of state, i.e., $a = \pi(s)$. On the other hand, a stochastic policy is defined as a probability of taking an action given the current state under the policy. This is denoted by the expression: $P_\pi[A = a|S = s] = \pi(a|s)$.

2.1.3. State-Value function

A state-value function or simply a value function specifies how good it is for an agent to be in a particular state with policy π. It is denoted by $V(s)$ and specifies the value of a state following a policy π. Mathematically, it can be defined as the expected return starting from state s according to policy π as given by:

$$V^\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{2.4}$$

By substituting the value from (2.3), the value function may be rewritten as follows:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right] \tag{2.5}$$

It is to be noted that the value function depends on the selected policy. The value of a state will change with varying policy.

2.1.4. State-action value function (Q-function)

A state-action value function (also know as Q function) specifies the value of taking a particular action in a given state with a policy π. Mathematically, it can be written as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[G_t|s_t = s, a_t = a] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\right] \tag{2.6}$$

The difference between value function and Q function is that value function specifies the goodness of a state while the Q-function specifies the goodness of taking an action in that state.

2.1.5. Optimality

As mentioned earlier, value function depends on a policy. The optimal value function is one that yields maximum value compared to other value functions. Mathematically, it can be written as:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.7}$$

In terms of Q functions, the optimal value function may be written as:

$$V^*(s) = \max_{a} \max_{\pi} Q^{\pi}(s,a) = \max_{a} Q^*(s,a) \tag{2.8}$$

The policy which maximizes the value function or Q function is an optimal policy. It is given by

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) = \arg\max_{\pi} Q^{\pi}(s,a) \tag{2.9}$$

2.1.6. Bellman Equation

The Bellman Equation breaks down the value function into two parts: an immediate reward and a discounted future value function. In order to derive the Bellman equation, we will start with the state-value function given by equation (2.5).

$$\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\pi}[G_t|s_t = s] \\
&= \mathbb{E}_{\pi}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots |s_t = s] \\
&= \mathbb{E}_{\pi}[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \cdots)|s_t = s] \\
&= \mathbb{E}_{\pi}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+2+k}|s_t = s] \\
&= \mathbb{E}_{\pi}[r_{t+1} + \gamma G_{t+1}|s_t = s] \\
&= \mathbb{E}_{\pi}[r_{t+1}|s_t = s] + \gamma E_{\pi}[G_{t+1}|s_t = s] \qquad \text{Using Theorem A.1} \\
&= \mathbb{E}_{\pi}[r_{t+1}|s_t = s] + \gamma E_{\pi}[G_{t+1}|s_{t+1} = s'] \qquad \text{Markovian Property: } G_{t+1} \text{ only depends on } s_{t+1} \\
&= \mathbb{E}_{\pi}[r_{t+1}] + \gamma V^{\pi}(s_{t+1}) \\
V^{\pi}(s) &= \mathbb{E}_{\pi}[r_{t+1} + \gamma V^{\pi}(s)|s_t = s]
\end{aligned} \tag{2.10}$$

The above recursive equation is called the Bellman equation for the value function. Similarly, the Bellman equation for action-value function can be written as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1})|s_t = s, a_t = a] \tag{2.11}$$

2.1.7. Deriving Bellman equation

Using (2.5), the value function can be re-written as:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}[G_t|s_t = s] = \mathbb{E}_{\pi}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s] \\ &= \mathbb{E}_{\pi}[r_{t+1}|s_t = s] + \gamma \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s] \\ &= \mathbb{E}_{\pi}[r_{t+1}|s_t = s] + \gamma \mathbb{E}_{\pi}[G_{t+1}|s_t = s] \end{aligned} \tag{2.12}$$

The first term in the above equation which is the expectation of r_{t+1} given that we are in state s can be written as:

$$\mathbb{E}_{\pi}[r_{t+1}|s_t = s] = \sum_{r \in \mathscr{R}} r p(r|s) \tag{2.13}$$

where $p(r|s)$ is the probability of appearance of reward r conditioned on state s. This $p(r|s)$ is a marginal distribution of distribution that also contained action a taken at time t and state s' at time $t+1$ as shown below:

$$p(r|s) = \sum_{s' \in \mathscr{S}} \sum_{a \in \mathscr{A}} p(s', a, r|s) = \sum_{s \in \mathscr{S}} \sum_{a \in \mathscr{A}} \pi(a|s) p(s', r|a, s) \tag{2.14}$$

where $p(a|s) = \pi(a|s)$ is the stochastic policy of the agent. The above equation makes use of the Bayes theorem of conditional probability (A.1) provided in Appendix A. Substituting (2.14) into (2.13), we get the first term of the value function (2.12) as:

$$\mathbb{E}_{\pi}[r_{t+1}|s_t = s] = \sum_{r \in \mathscr{R}} \sum_{s' in \mathscr{S}} \sum_{a \in \mathscr{A}} r \pi(a|s) p(s', r|a, s) \tag{2.15}$$

The expectation in the second term of (2.12) can rewritten as follows by considering G_{t+1} as a random variable that takes a finite number of values $g \in \Gamma$:

$$\mathbb{E}_{\pi}[G_{t+1}|s_t = s] = \sum_{g \in \Gamma} g p(g|s) \tag{2.16}$$

Now, we again "de-marginalise" the probability distribution by using the law of multiplication as follows:

$$\begin{aligned} p(g|s) &= \sum_{r \in \mathscr{R}} \sum_{s' \in \mathscr{S}} \sum_{a \in \mathscr{A}} p(s', a, r, g|s) = \sum_{r \in \mathscr{R}} \sum_{s' \in \mathscr{S}} \sum_{a \in \mathscr{A}} p(g|s', a, r, s) p(s', r, a|s) \\ &= \sum_{r \in \mathscr{R}} \sum_{s' \in \mathscr{S}} \sum_{a \in \mathscr{A}} p(g|s', a, r, s) p(s', r|a, s) \pi(a|s) \qquad (\because p(a|s) = \pi(a|s)) \\ &= \sum_{r \in \mathscr{R}} \sum_{s' \in \mathscr{S}} \sum_{a \in \mathscr{A}} p(g|s') p(s', r|a, s) \pi(a|s) \end{aligned} \tag{2.17}$$

In the last line of the above equations, we have used the Markovian property. Please note that G_{t+1} is the sum of all future discounted rewards that the agent receives after the state s'. So, the future actions and the rewards depend only on the state in which the action is taken, so $p(g|s', a, r, s) = p(g|s')$, by assumption. So, by using (2.16) and (2.17), the second term of the value equation (2.12) can rewritten as:

$$\begin{aligned}\gamma\mathbb{E}_\pi[G_{t+1}|s_t = s] &= \gamma \sum_{g\in\Gamma}\sum_{r\in\mathscr{R}}\sum_{s'\in\mathscr{S}}\sum_{a\in\mathscr{A}} gp(g|s')p(s', r|a, s)\pi(a|s) \\ &= \gamma \sum_{r\in\mathscr{R}}\sum_{s'\in\mathscr{S}}\sum_{a\in\mathscr{A}} \mathbb{E}_\pi[G_{t+1}|s_{t+1} = s']p(s', r|a, s)\pi(a|s) \\ &= \gamma \sum_{r\in\mathscr{R}}\sum_{s'\in\mathscr{S}}\sum_{a\in\mathscr{A}} V_\pi(s')p(s', r|a, s)\pi(a|s) \end{aligned} \tag{2.18}$$

Now substituting the two terms in (2.12) using (2.15) and (2.18), we get the final form of the Bellman equation given as:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] \\ &= \sum_{a\in\mathscr{A}} \pi(a|s) \sum_{r\in\mathscr{R}}\sum_{s'\in\mathscr{S}} p(s', r|a, s)[r + \gamma V^\pi(s')] \end{aligned} \tag{2.19}$$

The Bellman equation for state-action or Q function can be similarly derived as follows (Need to check this!!):

$$\begin{aligned} Q^\pi(s, a) &= \sum_{r\in\mathscr{R}}\sum_{s'\in\mathscr{S}} p(s', r|a, s)[r + \gamma \sum_{a'\in\mathscr{A}} Q(s', a')] \\ &= \sum_{r\in\mathscr{R}}\sum_{s'\in\mathscr{S}} p(s', r|a, s)[r + \gamma V^\pi(s')] \end{aligned} \tag{2.20}$$

2.2. Solving the Bellman Equation

2.2.1. Dynamic Programming

Dynamic programming (Dynamic Programming (DP)) [9] is an algorithmic technique for solving problems by breaking them down into smaller sub-problems and using the solutions to those subproblems to solve the original problem. It is a powerful technique that can be used to solve a wide variety of problems, including many that are difficult to solve with other methods.

Dynamic programming works by storing the results of the subproblems in a table or array. This allows the algorithm to avoid re-computing the same sub-problems multiple times. The table is typically filled in bottom-up, starting with the smallest sub-problems and working up to the largest.

We will solve the above Bellman equation using the following two powerful DP algorithms:

- Value Iteration
- Policy Iteration

2.2.2. Value Iteration

The steps involved in the value iteration algorithm are as follows:

1. Initialize with the random value function with random values for each state.
2. Compute Q function $Q(s, a)$ for all state and action pairs.
3. Update the value function with maximum value of $Q(s, a)$.
4. Repeat the above two steps (2-3) until convergence (change in value function becomes very small.)

The complete python code for value iteration algorithm is provided in the Listing 2.1. The first function `value_iteration` computes the optimal value function by using the Bellman equation (2.19) while the second function `extract_policy` is used to compute the optimal policy from the above value function by using equation (2.9). This will be used to solve the Frozen Lake problem in the next subsection.

```
class ValueIterationAgent():
    def __init__(self, env, gamma=0.99, max_iterations=10000):
        self.env = env
        self.num_states = self.env.observation_space.n
        self.num_actions = self.env.action_space.n
        self.gamma = gamma
        self.max_iterations = max_iterations

    def value_iteration(self, threshold=1e-20):
        value_table = np.zeros(self.num_states)
        for i in range(self.max_iterations):
            updated_value_table = np.copy(value_table)
            for state in range(self.num_states):
                Q_value = []  # Q(s,a)
                for action in range(self.num_actions):
                    Q_value.append(np.sum(
                    [trans_prob * \
                    (reward + self.gamma * updated_value_table[next_state]) \
                    for trans_prob, next_state, reward, _ \
                                        in self.env.P[state][action]]))
                value_table[state] =  max(Q_value)
            if (np.sum(
                np.fabs(updated_value_table - value_table)) <= threshold):
```

```
            print("Value-iteration converged at iteration # %d" % (i+1))
            break
    return value_table

def extract_policy(self, value_table):
    policy = np.zeros(self.num_states)
    for state in range(self.num_states):
        Q_table = np.zeros(self.num_actions)
        for action in range(self.num_actions):
            Q_table[action] = np.sum(
            [trans_prob * \
            (reward + self.gamma * value_table[next_state])\
            for trans_prob, next_state, reward, _ \
                in self.env.P[state][action]])
        policy[state] = np.argmax(Q_table)
    return policy

def train_and_validate(self, max_episodes=10):
    # Compute optimal policy
    optimal_value = self.value_iteration()
    optimal_policy = self.extract_policy(optimal_value)
    ep_rewards, ep_steps = [], []
    done = False
    for i in range(max_episodes):
        rewards = 0
        done = False
        state = self.env.reset()[0]
        step = 0
        while not done:
            step += 1
            action = optimal_policy[state]
            next_state, reward, done, _, _ = env.step(int(action))
            screen = env.render()
            rewards += reward
            state = next_state
        ep_rewards.append(rewards)
        ep_steps.append(step)
    return np.mean(ep_rewards), np.mean(ep_steps)

def __del__(self):
    self.env.close()
```

Listing 2.1: Python class for implementing value iteration algorithm.

2.2.3. Policy Iteration

Policy iteration involves the following steps:

1. Start with a random policy.
2. Find the value function for this policy. Evaluate to see if it is optimal - *Policy Evaluation.*
3. If the policy is not optimal, find a new improved policy - *Policy improvement.*
4. Repeat the above two steps (2-3) until optimal policy is found.

The complete code for policy iteration algorithm is provided in the Listing 2.2. The `policy_iteration` method calls two functions, namely, `evaluate_policy` and `improve_policy`. This code learns a deterministic policy $a = \pi(s)$. It is also possible to learn a stochastic policy $\pi(s, a)$ as shown in the implementation provided in Listing 2.3. In this implementation, the policy function is represented by a matrix representing the probability of an action for a given state. Initially, all actions are assigned equal probability.

```
import gymnasium as gym
import numpy as np

class PolicyIterationAgent():
    def __init__(self, env, gamma=0.99, max_iterations=10000):
        self.env = env
        self.num_states = self.env.observation_space.n
        self.num_actions = self.env.action_space.n
        self.gamma = gamma
        self.max_iterations = max_iterations

    def evaluate_policy(self, policy, threshold=1e-10):
        value_table = np.zeros(self.num_states)
        while True:
            updated_value_table = np.copy(value_table)
            for state in range(self.num_states):
                action = policy[state]
                value_table[state] = np.sum(
                [trans_prob *   \
                (reward + self.gamma * updated_value_table[next_state])\
                for trans_prob, next_state, reward, _ in env.P[state][action]])
            if (np.sum(
                np.fabs(updated_value_table - value_table)) <= threshold):
```

```
            break
        return value_table

    def improve_policy(self, value_table):
        policy = np.zeros(self.num_states)
        for state in range(self.num_states):
            Q_table = np.zeros(self.num_actions)
            for action in range(self.num_actions):
                Q_table[action] = np.sum(
                [trans_prob * \
                (reward + self.gamma * value_table[next_state]) \
                for trans_prob, next_state, reward, _ \
                in self.env.P[state][action]])
            policy[state] = np.argmax(Q_table)
        return policy

    def policy_iteration(self):
        current_policy = np.zeros(self.num_states)
        for i in range(self.max_iterations):
            new_value_function = self.evaluate_policy(current_policy)
            new_policy = self.improve_policy(new_value_function)
            if (np.all(current_policy == new_policy)):
                print('Policy iteration converged at step %d.' %(i+1))
                current_policy = new_policy
                break
            current_policy = new_policy
        return new_policy

    def train_and_validate(self, max_episodes=10):
        # compute optimal policy
        optimal_policy = self.policy_iteration()
        ep_rewards, ep_steps = [], []
        done = False
        for i in range(max_episodes):
            rewards = 0
            done = False
            state = self.env.reset()[0]
            step = 0
            while not done:
                step += 1
                action = optimal_policy[state]
                next_state, reward, done, _, _ = env.step(int(action))
                rewards += reward
                state = next_state
```

```
            ep_rewards.append(rewards)
            ep_steps.append(step)
        return np.mean(ep_rewards), np.mean(ep_steps)

    def __del__(self):   # destructor
        self.env.close()
```

Listing 2.2: Python class for implementing Policy iteration algorithm.

```
import gymnasium as gym
import numpy as np
class PolicyIterationAgent2():
    def __init__(self, env, gamma=0.99, max_iterations=100000, threshold=1e-6):
        self.env = env
        self.num_states = self.env.observation_space.n
        self.num_actions = self.env.action_space.n
        self.gamma = gamma
        self.max_iterations = max_iterations
        self.threshold = threshold

    def policy_evaluation(self, policy):
        value_fn = np.zeros(self.num_states)
        i = 0
        while True:
            i += 1
            prev_value_fn = np.copy(value_fn)
            for state in range(self.num_states):
                outersum = 0
                for action in range(self.num_actions):
                    q_value = np.sum(
                        [trans_prob * \
                        (reward + self.gamma * prev_value_fn[next_state]) \
                        for trans_prob, next_state, reward, _ \
                        in self.env.P[state][action]])
                    outersum += policy[state, action] * q_value
                value_fn[state] = outersum
            if (np.max(np.fabs(prev_value_fn - value_fn)) < self.threshold):
                print('Value convergences in %d iteration' %(i+1))
                break
        return value_fn

    def policy_improvement(self, value_fn):
        q_value = np.zeros((self.num_states, self.num_actions))
```

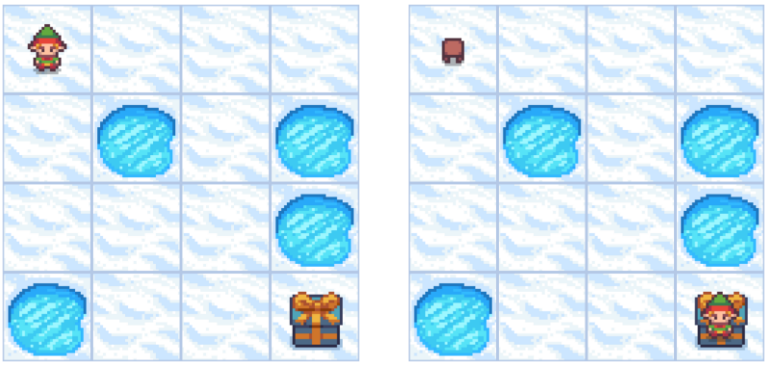
```
        improved_policy = np.zeros((self.num_states, self.num_actions))
        for state in range(self.num_states):
            for action in range(self.num_actions):
                q_value[state, action] = np.sum(
                    [trans_prob * \
                    (reward + self.gamma * value_fn[next_state]) \
                    for trans_prob, next_state, reward, _ \
                    in self.env.P[state][action]])
            best_action_indices = np.where(q_value[state,:] \
                == np.max(q_value[state,:]))[0]
            for index in best_action_indices:
                improved_policy[state, index] = 1/np.size(best_action_indices)
        return improved_policy


    def policy_iteration(self):
        # start with uniform probability for all actions
        initial_policy = (1.0/self.num_actions) * \
            np.ones((self.num_states, self.num_actions))  # \pi(s,a)
        for i in range(self.max_iterations):
            if i == 0:
                current_policy = initial_policy
            current_value = self.policy_evaluation(current_policy)
            improved_policy = self.policy_improvement(current_value)
            if np.allclose(current_policy, improved_policy, \
                rtol=1e-10, atol=1e-15):
                print(f'Policy Iteration converged in {i+1} iterations.' )
                current_policy = improved_policy
                break
            current_policy = improved_policy
        return current_policy


    def train_and_validate(self, max_episodes=10):
        # compute optimal policy
        optimal_policy = self.policy_iteration()
        ep_rewards, ep_steps = [], []
        done = False
        for i in range(max_episodes):
            rewards = 0
            done = False
            state = self.env.reset()[0]
            step = 0
            while not done:
```

```
            step += 1
            action = np.argmax(optimal_policy[state, :])
            next_state, reward, done, _, _ = env.step(int(action))
            rewards += reward
            state = next_state
        ep_rewards.append(rewards)
    ep_steps.append(step)
return np.mean(ep_rewards), np.mean(ep_steps)

def __del__(self): # destructor
    self.env.close()
```

Listing 2.3: Python code for Policy Iteration Algorithm for learning a stochastic policy distribution $\pi(s, a)$.

2.2.4. Solving the Frozen Lake Problem

Frozen Lake is a toy text environment from Gymnasium [10] which involves crossing a frozen lake from start to goal without falling into any holes by walking over the frozen lake. The player may not always move in the intended direction due to the slippery nature of the frozen lake. The game starts with the player at location [0,0] of the frozen lake grid world with the goal located at far extent of the world e.g. [3,3] for the 4x4 environment. The player can be in any of the 16 discrete states and can take one of the four actions (move left, right, top or down). The agent receives a reward of 1.0 only when it reaches the goal. Otherwise it receives a zero reward for all other states. The initial and the final state with success (reward = 1.0) is shown in the Figure 2.1.



(a) Initial state (b) Final successful state

Figure 2.1.: Visualization of the Frozen Lake Environment

The code for solving this problem by using value iteration and policy iteration algorithm is provided in Listing 2.4. It is seen that the algorithms are able to solve the problem at

least 8 times out of 10 episodes indicating a success rate of about 80%.

```
import gymnasium as gym
import numpy as np

# stochastic environment
env = gym.make('FrozenLake-v1', map_name="4x4",
└└└└└└└└└is_slippery=True, render_mode="rgb_array")
state = env.reset()
print('state: ', state)
print("Observation space dimension: ", env.observation_space.n)
print("Action space dimension:", env.action_space.n)
print("Value Iteration Algorithm:")
agent = ValueIterationAgent(env)
mean_rewards, mean_steps = agent.train_and_validate()
print(f'mean episodic reward: {mean_rewards}, \
      average steps per episode: {mean_steps}')
print("---------------------------")
print("Policy Iteration Algorithm:")
p_agent = PolicyIterationAgent(env)
mean_reward, mean_steps = p_agent.train_and_validate(max_episodes=10)
print(f'Mean rewards: {mean_reward}  Mean steps: {mean_steps}.')
```

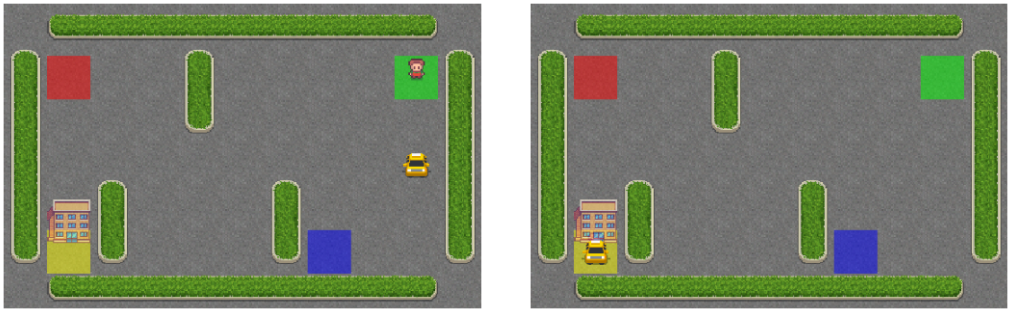
Program output:

```
state: (0, {'prob': 1})
Observation space dimension:  16
Action space dimension: 4
Value Iteration Algorithm:
Value-iteration converged at iteration # 996
mean episodic reward: 0.8, average steps per episode: 68.2
---------------------------------
Policy Iteration Algorithm:
Policy iteration converged at step 7.
Mean rewards: 1.0  Mean steps: 54.2.
```

Listing 2.4: Applying value iteration algorithm to solve the Frozen Lake Problem

2.2.5. Solving the Taxi Problem

The Taxi problem involves navigating to passengers in a grid world, picking them up and dropping them off at one of the four locations. There are four designated pick-up and



(a) Initial state (b) Final successful state

Figure 2.2.: Visualization of The Taxi-v3 environment

drop-off locations (Red, Green, Yellow and Blue) in the 5x5 grid world. The taxi starts off at a random square and the passenger at one of the designated locations.

The goal is move the taxi to the passenger's location, pick up the passenger, move to the passenger's desired destination, and drop off the passenger. Once the passenger is dropped off, the episode ends.

The player receives positive rewards for successfully dropping-off the passenger at the correct location. Negative rewards for incorrect attempts to pick-up/drop-off passenger and for each step where another reward is not received. Two snap-shots of the taxi environment is shown in Figure 2.2. A random policy performs poorly resulting in highly negative average rewards as shown in listing 2.5.

```
import gymnasium as gym
env = gym.make("Taxi-v3", render_mode="rgb_array")
state = env.reset()
print('state:', state)
print('size of state space: ', env.observation_space.n)
print('size of action space: ', env.action_space.n)
print('Scores with random policy:')
ep_rewards, ep_steps = [], []
for i in range(10):
    state = env.reset()[0]
    done = False
    step = 0
    rewards = 0
    while not done:
        step += 1
        action = env.action_space.sample()
        next_state, reward, done, _ ,_= env.step(action)
        rewards += reward
        state = next_state
```

```
    ep_rewards.append(rewards)
    ep_steps.append(step)
print('Mean episodic Reward: ', np.mean(ep_rewards))
print('Mean episodic steps: ', np.mean(ep_steps))
```

```
 Scores with random policy:
 Mean episodic Reward:  -9473.9
Mean episodic steps:  2428.1
```

Listing 2.5: The performance of Random Policy for the Taxi environment

```
env = gym.make("Taxi-v3")
print('Policy Iteration:')
agent = PolicyIterationAgent(env)
mean_rewards, mean_steps  = agent.train_and_validate()
print(f'Avg rewards: {mean_rewards}, Avg steps: {mean_steps}')
print("------------------------")
print('Value Iteration:')
v_agent = ValueIterationAgent(env)
mean_rewards, mean_steps = agent.train_and_validate()
print(f'Avg rewards: {mean_rewards}, Avg steps: {mean_steps}')
```

```
Policy Iteration:
Policy iteration converged at step 17.
Avg rewards: 7.3, Avg steps: 13.7
------------------------------------
Value Iteration:
Value-iteration converged at iteration # 3325
Avg rewards: 7.2, Avg steps: 13.8
```

2.3. Summary

In this chapter, the mathematical framework for solving the reinforcement learning (RL) problem is presented. It is formulated as a Markov Decision Process (MDP) where the

future state depends on the current state. The aim of the RL agent is to maximize the cumulative future reward starting from the current state. This is achieved by solving the Bellman equation. Two algorithms, namely, value-iteration and policy-iteration algorithm are used for solving The Frozen Lake and the Taxi problem.

3. Monte Carlo Simulation for Reinforcement Learning

3.1. Introduction

Monte Carlo (MC) simulation is a powerful tool for reinforcement learning (RL). It can be used to estimate the value of states and actions, and to learn policies that maximize expected rewards. Monte Carlo simulation is particularly well-suited for RL problems where the environment is complex or stochastic, and where it is difficult or impossible to learn a model of the environment's dynamics.

3.2. What is Monte Carlo simulation?

Monte Carlo simulation is a computational method that uses random sampling to solve problems. It works by generating a large number of random samples from a probability distribution, and then performing some operation on those samples to get an estimate of the desired quantity. The intuition behind this approach can be better understood with the examples discussed below.

3.2.1. Tossing a Coin

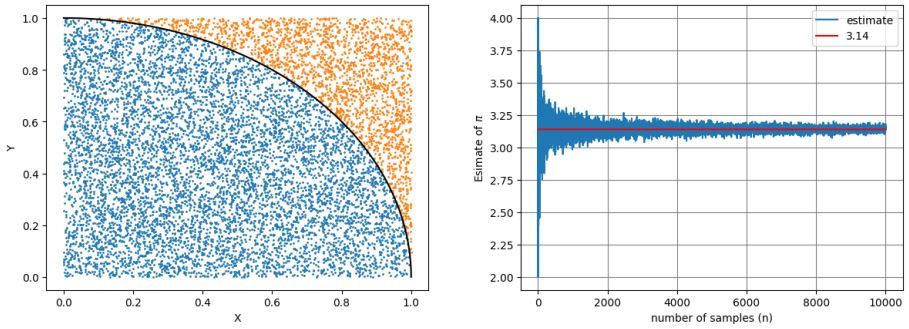
Here is a simple example of how to use a Monte Carlo algorithm to estimate the probability of winning a coin toss game:

- Toss a coin.
- You win if a 'head' is obtained.
- You loose if a 'tail' is obtained.

You can repeat this process a large number of times to get a more accurate estimate of the probability of winning the game. For example, if you repeat the process 1000 times and predict that you will win 500 times, then your estimated probability of winning the game is 50%.

3.2.2. Estimating the value of pi using Monte Carlo

Consider the Figure 3.1(a) where a quadrant of a circle is drawn inside a square. Lets generate random number within the square. Some of these points will lie inside the circle



(a) (b)

Figure 3.1.: Estimating π through Monte-Carlo Simulation. (a) A quadrant of circle inside a square. (b) Estimate of π improves with increasing number of samples

and some will lie outside the circle. The value of π can be calculated as follows:

$$\begin{aligned} \frac{\text{Area of Circle quadrant}}{\text{Area of square}} &= \frac{\frac{\pi r^2}{4}}{a^2} = \frac{\pi}{4} \qquad \because r = a = 1 \\ \pi &= 4 \times \frac{\text{Area of Circle quadrant}}{\text{Area of square}} \\ \pi &= 4 \times \frac{\text{Number of points inside the circle}}{\text{Number of points inside the square}} \end{aligned} \tag{3.1}$$

Where radius r of the circle is same as the length of each side of the square. Hence, the steps to estimate π are as follows:

1. Generate some random numbers inside the square.
2. Calculate the number of points that fall inside the circle by using the equation $x^2 + y^2 \leq r$.
3. Estimate π by using (3.1).
4. Increase the number of samples to improve the estimate.

The resulting estimate of π is shown in Figure 3.1(b). As we can see, the estimation improves with increasing number of samples. The detailed Python code for implementing the above steps is provided in the Listing 3.6.

```
import math
def deg2rad(deg):
    return deg * math.pi / 180

#estimate pi
import matplotlib.pyplot as plt
```

```
import numpy as np
pi_estimate = []
for n in range(1, 10000):
    s = np.random.uniform(0, 1.0, size=(n, 2))  # x,y points
    r = np.sqrt(np.sum(s**2, axis=1))    # r = sqrt(x^2 + y^2)
    c = np.sum(r < 1)                    # count points having r < 1
    pi_estimate.append(4 * c / n)

#plotting sample points
cin = s[r < 1] # points inside the circle
cout = s[r > 1] # points outside the circle
plt.scatter(cin[:,0], cin[:,1], s=1, label='points inside circle')
plt.scatter(cout[:,0], cout[:,1], s=1, label='points outside circle')
plt.xlabel('X')
plt.ylabel('Y')
# draw the circle
theta = np.linspace(0, deg2rad(90), 1000)
radius = 1
x = radius * np.sin(theta)
y = radius * np.cos(theta)
plt.plot(x, y, c='black')

# new plot
plt.figure()
plt.plot(pi_estimate, label='estimate')
plt.plot(3.14*np.ones_like(pi_estimate), c='r', label='3.14')
plt.xlabel('Number of samples (n)')
plt.ylabel('Estimate of $\pi$')
plt.grid(color='gray')
plt.legend(loc='best')
```

Listing 3.6: Python code for estimating π through random sampling.

3.3. Monte Carlo Prediction

In the context of reinforcement learning, Monte Carlo simulation can be used to estimate the value of states and actions by simulating a large number of trajectories from those states and actions. The value of a state or action is then estimated as the average reward of the trajectories. The knowledge of system model (e.g. transition and reward probabilities) is not required for solving the Markov Decision Process (MDP) problem unlike the methods (e.g. value & policy iteration) that we discussed in the last chapter. So, Monte-Carlo methods are *model-free* methods which requires generating sample sequences

of states, actions and rewards. The Monte-Carlo methods is applied only to episodic tasks.

As per the definition provided in the previous chapter, a value function is the expected return from a state s with a policy π as given by equations (2.4) and (2.5). In Monte-Carlo prediction, this value function is approximated by taking the mean return instead of the expected return. So, the steps involved in the Monte-Carlo prediction of value function is as follows:

1. Initialize a random value to the value function.
2. Create an empty list to store returns.
3. For each state in the episode, calculate the return.
4. Append the return to the return list created above.
5. Take the average of the return list to compute the new value function.

Monte-Carlo prediction algorithms are of two types: (1) First-Visit Monte-Carlo and (2) Every-visit Monte-Carlo

3.3.1. First-Visit Monte-Carlo Algorithm

The first-visit Monte Carlo prediction algorithm is used to estimate the value function of a Markov decision process (MDP) under a given policy. The algorithm works by repeatedly simulating episodes of the MDP and averaging the returns from each episode. The return of an episode is the sum of the rewards received, discounted by a discount factor.

The first-visit Monte Carlo prediction algorithm only considers the first visit to each state in an episode when estimating the value function. This means that the algorithm can be biased towards states that are visited more often. However, the algorithm is relatively simple to implement and converges to the true value function as the number of simulated episodes increases.

3.3.2. Every-visit Monte-Carlo Algorithm

The every-visit Monte Carlo prediction algorithm is similar to the first-visit Monte Carlo prediction algorithm, but it considers every visit to each state in an episode when estimating the value function. This means that the algorithm is less biased than the first-visit algorithm, but it is also more computationally expensive.

The every-visit Monte Carlo prediction algorithm works by keeping track of the number of times each state is visited and the total reward received from each state. The value function for each state is then estimated by averaging the total reward received from the state, divided by the number of times the state has been visited.

The every-visit Monte Carlo prediction algorithm converges to the true value function as the number of simulated episodes increases, regardless of the policy that is being evaluated. This makes it a good choice for evaluating policies that are likely to result in the same states being visited multiple times.

3.3.3. Playing Blackjack Game

Let's play Blackjack game to get a better insight into Monte Carlo prediction algorithms. Blackjack is a card game played between a dealer and one or more players. The goal of the game is to get as close to 21 as possible without going over. If a player or the dealer goes over 21, they "bust" and lose the bet. Blackjack is played with a standard deck of 52 cards. The cards are assigned the following values:

- Ace: 1 or 11
- 2-10: Face value
- Jack, Queen, King: 10

The game begins with each player placing a bet. The dealer then deals two cards to each player which are face up and visible to all. The dealer also takes two cards, one face up and another face down. The players then decide whether to *hit* or *stick*. To hit means to take another card. To stay means to keep the cards that they have.

If the player hit and the sum of cards exceed 21 then it is a *bust* and he looses the game. If he decides to stick, the dealer can reveal with face down card and the sum of dealer's card will be checked. If it does not exceed 21 and higher than player's sum of cards, the dealer wins. Otherwise, the player wins. If the player's and dealer's sum of cards is same, then it is a *draw*.

The rewards of the game can have one of the following values:

- +1 if the player wins.
- -1 if the player loses.
- 0 if it is a draw.

The possible actions are:

- Stick (0): If the player does not need a card.
- Hit (1): If the player takes a card from the deck.

The player has the freedom to decide the value of an ace depending on his current score. If player's sum is 10 and the player receives an ace after a hit, he can consider it as 11 leading to a total sum of 21 (win). But if the player's sum is 17, he/she can consider the ace to 1 to avoid a bust. If player can consider an ace to be 11 without being bust, it is called an *usable ace*. Otherwise, it will be called a *non-usable ace*.

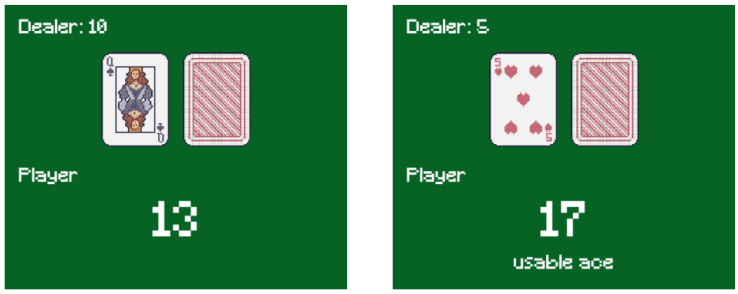
The code for visualizing the state of Gymnasium's Blackjack environment is provided in the Listing 3.7. The visualization of two states are shown in Figure 3.2.

```
import sys
import gymnasium as gym
import matplotlib.pyplot as plt
```

```
env = gym.make('Blackjack-v1', render_mode="rgb_array")
state = env.reset()
print('state:', state)
action = env.action_space.sample()
print('Step function output: ', env.step(action))
print('Observation space: ', env.observation_space.spaces)
print('Action values:', env.action_space.n)
screen = env.render()
plt.imshow(screen)
plt.axis('off')
```

```
Program Output:
state: ((13, 10, 0), {})
Step function output:  ((13, 10, 0), 1.0, True, False, {})
Observation space:  (Discrete(32), Discrete(11), Discrete(2))
Action values: 2
```

Listing 3.7: Python code to visualize Blackjack environment



(a) (b)

Figure 3.2.: Visualization of a state of Blackjack environment. The second image shows a state with an usable ace.

The Python code for implementing first-visit & every-visit Monte Carlo prediction of state-action value (Q) function is provided in the Listing 3.8. As mentioned earlier, the value function V for a given state s is approximated as the average value of (discounted)

future rewards obtainable from this state. This is given by

$$\begin{aligned} V_N &= \frac{1}{N}\sum_{i=1}^{N}\gamma^i R_i = \frac{1}{N}[\sum_{i=1}^{N-1}\gamma^i R_i + \gamma^N R_N] \qquad (3.2)\\ &= \frac{1}{N}[(N-1)V_{N-1} + \gamma^N R_N] \\ &= V_{N-1} + \frac{1}{N}[\gamma^N R_N - V_{N-1}] \\ V_N &= V_{N-1} + \frac{1}{N}[R_N - V_{N-1}] \qquad \text{(assuming } \gamma = 1\text{)} \qquad (3.3) \end{aligned}$$

The class method `mc_predict()` provides option to use either equation (3.2) or (3.3) through the `method` argument for updating the estimate of Q function. The episodes for Monte Carlo simulation is generated using a sample policy that chooses to *stick* with 80% probability if the player sum of cards is greater than 18. Otherwise, it chooses to *hit* with 80% probability. The pseudocode for first-visit Monte Carlo Prediction algorithm is provided in Algorithm 3.1. This listing provides implementation of both 'first visit' and 'every-visit algorithm'. There is not much difference in the performance of these two algorithms for the `Blackjack` environment as each state is not visited more than once in each episode.

Algorithm 3.1 First-Visit Monte Carlo Algorithm (*for state-action values*)

Input: policy π to be evaluated
Output: State-action value function Q
Initialize $N(s,a) = 0$ for all $s \in \mathscr{S}, a \in \mathscr{A}$
Initialize $returns\_sum(s,a) = 0$ for all $s \in \mathscr{S}, a \in \mathscr{A}$
for $i \leftarrow 1$ to $num\_episodes$ **do**
    Generate an episode $\{(s,a,r)_i, i = 1 \ldots T\}$ with policy π
    for each state s in episode **do**
        if (s,a) is a first visit (with return G_t) **then**
            $N(s,a) \leftarrow N(s,a) + 1$
            $returns\_sum(s,a) \leftarrow returns\_sum(s,a) + G_t$
            $Q(s,a) \leftarrow returns\_sum(s,a)/N(s,a)$
        end if
    end for
end for
return Q

The resulting value estimates are shown in Figure 3.3. This plot can be generating by using the python functions provided in Listing 3.9. It can be seen that the states where the player's sum of cards is greater than 18 are more valuable. The values reduce slightly when the ace is not usable.

```
from collections import defaultdict
import numpy as np
```

```
import random
class MCPAgent:
    def __init__(self, env, gamma=0.99):
        self.env = env
        self.n_actions = env.action_space.n
        self.gamma = gamma
        print('Environment: ', self.env.spec.name)

    def sample_policy(self, obs):
        player_sum, dealer_show, usable_ace = obs
        probs = [0.8, 0.2] if player_sum > 18 else [0.2, 0.8]
        action = np.random.choice(np.arange(2), p=probs)
        return action

    def generate_episode(self):
        """ Plays a single episode with a set policy.
        Records the state, action and reward for each step
        returns all the timesteps for the episode.  """
        episode = []
        state = env.reset()[0]
        while True:
            action = self.sample_policy(state)
            next_state, reward, done, info, _ = env.step(action)
            episode.append((state, action, reward))
            state = next_state
            if done:
                break
        return episode

def update_Q_first_visit(self, episode, Q, returns_sum, N, method=1):
        G = 0
        visited_state = set()
        for i in range(len(episode)-1, -1, -1):    # traverse in reverse order
            state, action, reward = episode[i]
            G += self.gamma ** i * reward # returns for (s,a)
            if (state, action) not in visited_state:    # if first visit
                N[state][action] += 1  # first visit count
                if method == 1:
                    returns_sum[state][action] += G  # update returns
                    Q[state][action] = returns_sum[state][action] / N[state][action]
                else:
                    returns_sum[state][action] = G # update returns
                    Q[state][action] += (returns_sum[state][action] -
                                         Q[state][action])/ N[state][action]
```

```
                visited_state.add((state, action))

    def update_Q_every_visit(self, episode, Q, returns_sum, N, method=1):
        G = 0  # return
        for i in range(len(episode)-1, -1, -1):
            s, a, r = episode[i]
            G += self.gamma ** i * r  # returns for (s,a)
            N[s][a] += 1    # every-visit count
            if method == 1:
                returns_sum[s][a] += G
                Q[s][a] = returns_sum[s][a] / N[s][a]
            else:
                returns_sum[s][a] = G
                Q[s][a] += (returns_sum[s][a] - Q[s][a]) / N[s][a]

def mc_predict(self, num_episodes=100000, first_visit=False, method=1):
    """ This plays through several episodes of the game """
    returns_sum = defaultdict(lambda: np.zeros(self.env.action_space.n))
    N = defaultdict(lambda: np.zeros(self.env.action_space.n))
    Q = defaultdict(lambda: np.zeros(self.env.action_space.n))
    for i in range(1, num_episodes+1):
        if i % 1000 == 0:
            print('\rEpisode: {}/{}.'.format(i, num_episodes), end="")
            sys.stdout.flush()
        episode = self.generate_episode()
        if first_visit:
            self.update_Q_first_visit(episode, Q, returns_sum, N, method)
        else:
            self.update_Q_every_visit(episode, Q, returns_sum, N, method)
    return Q

def QtoV(self, Q):
    ''' Converts Q to V '''
    V = dict((k, (k[0]>18) * (np.dot([0.8, 0.2], v)) + \
        (k[0] <= 18) * (np.dot([0.2, 0.8], v))) for k, v in Q.items())
    return V

if __name__ == '__main__':
    agent = MCPAgent(env)
    Q = agent.mc_predict(num_episodes=500000, method=2)
    V = agent.QtoV(Q)
```

Listing 3.8: Python code for estimating value function using First-Visit Monte-Carlo Prediction

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

def plot_blackjack_values(V):
    def get_Z(x, y, usable_ace):
        if (x,y,usable_ace) in V:
            return V[x,y,usable_ace]
        else:
            return 0

    def get_figure(usable_ace, ax):
        x_range = np.arange(11, 22)
        y_range = np.arange(1, 11)
        X, Y = np.meshgrid(x_range, y_range)
        Z = np.array([get_Z(x,y,usable_ace) \
            for x,y in zip(np.ravel(X), np.ravel(Y))]).reshape(X.shape)
        surf = ax.plot_surface(X, Y, Z, rstride=1, \
            cstride=1, cmap=plt.cm.coolwarm, vmin=-1.0, vmax=1.0)
        ax.set_xlabel('Player\'s Current Sum')
        ax.set_ylabel('Dealer\'s Showing Card')
        ax.set_zlabel('State Value')
        ax.view_init(ax.elev, -120)

    fig = plt.figure(figsize=(20, 20))
    ax = fig.add_subplot(121, projection='3d')
    ax.set_title('Usable Ace')
    get_figure(True, ax)
    ax = fig.add_subplot(122, projection='3d')
    ax.set_title('No Usable Ace')
    get_figure(False, ax)
    plt.show()

    # code for plotting policy
def plot_blackjack_policy(policy):

    def get_Z(x, y, usable_ace):
        if (x,y,usable_ace) in policy:
            return policy[x,y,usable_ace]
        else:
```

```
            return 1

    def get_figure(usable_ace, ax):
        x_range = np.arange(11, 22)
        y_range = np.arange(10, 0, -1)
        X, Y = np.meshgrid(x_range, y_range)
        Z = np.array([[get_Z(x,y,usable_ace) for x in x_range] for y in y_range])
        surf = ax.imshow(Z, cmap=plt.get_cmap('Pastel2', 2), \
                         vmin=0, vmax=1, extent=[10.5, 21.5, 0.5, 10.5])
        plt.xticks(x_range)
        plt.yticks(y_range)
        plt.gca().invert_yaxis()
        ax.set_xlabel('Player\'s Current Sum')
        ax.set_ylabel('Dealer\'s Showing Card')
        ax.grid(color='w', linestyle='-', linewidth=1)
        divider = make_axes_locatable(ax)
        cax = divider.append_axes("right", size="5%", pad=0.1)
        cbar = plt.colorbar(surf, ticks=[0,1], cax=cax)
        cbar.ax.set_yticklabels(['0 (STICK)','1 (HIT)'])

    fig = plt.figure(figsize=(15, 15))
    ax = fig.add_subplot(121)
    ax.set_title('Usable Ace')
    get_figure(True, ax)
    ax = fig.add_subplot(122)
    ax.set_title('No Usable Ace')
    get_figure(False, ax)
    plt.show()
```

Listing 3.9: Python code for plotting Value and Policy for Blackjack environment

Another implementation of the first-visit Monte-Carlo algorithm for updating Q is provide in the listing 3.10. It uses Python's list comprehension to provide a more compact code for the same task.

```
  def update_Q_first_visit(self, episode, Q, returns_sum, N, method=1):
    """
    Another implementation of First Visit Monte-Carlo update of Q values.
    """
    for s, a, r in episode:
        first_occurrence_idx = next(i \
            for i, x in enumerate(episode) if x[0] == s)
        G = sum([x[2] * (self.gamma ** i) \
            for i, x in enumerate(episode[first_occurrence_idx:])])
```

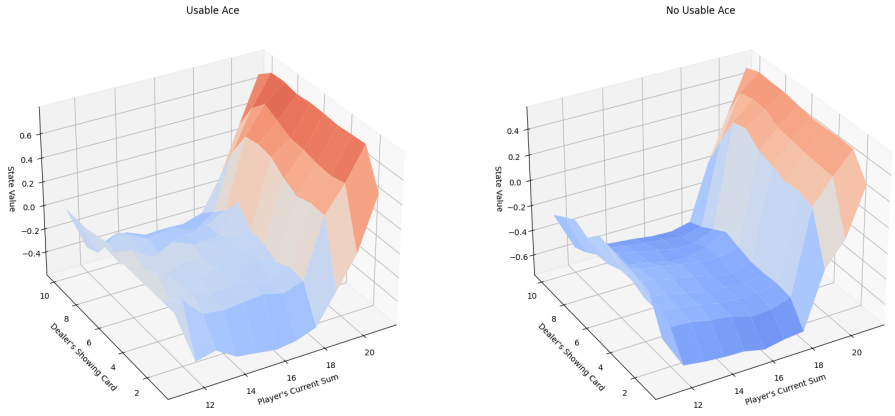


Figure 3.3.: Estimated Value function with and without usable ace

```
            N[s][a] += 1  # first visit count for (s,a)
            if method == 1:
                returns_sum[s][a] += G
                Q[s][a] = returns_sum[s][a] / N[s][a]
            else:
                returns_sum[s][a] = G
                Q[s][a] += (returns_sum[s][a] - Q[s][a]) / N[s][a]
```

Listing 3.10: Another implementation of first visit algorithm for estimating Q values.

3.4. Monte Carlo Control

In the last section, we saw how we can estimate the value function. In this section, we will see how we can optimize the value function to learn the optimal control strategy for the given task. The process involves executing an iterative cycle of *policy evaluation* and *policy improvement* until a convergence is achieved where the policy or the value function do not change any further. Specifically, we will discus a particular version of Monte-Carlo control algorithm known as GLIE-MC Control algorithm. GLIE stands for '*Greedy in the limit with infinite exploration*'. In order to learn the best policy, the agent uses a mix of good moves learnt in the past (*exploitation*) and new moves not tried before (*exploration*). The balance between exploration and exploitation is achieved by using an $\epsilon-$greedy policy where the agent takes random action with probability ϵ and acts greedily with a probability of $1-\epsilon$. A higher ϵ signifies more exploration. A greedy algorithm picks up the best choice available at that moment which may not be optimal for the overall problem. In other words a greedy policy might be locally optimal but globally sub-optimal.

Thus the steps involved in each iterative cycle of Monte-Carlo Control algorithm is as follows:

1. Initialization
2. Exploration
3. Update Policy - *Policy improvement*
4. Generate episode with new policy
5. Update Q values - *Policy evaluation*

The pseudocode of this algorithm is provided in Algorithm 3.2. The Q values are updated using a variant of (3.3) as given below:

$$Q(s,a) = Q(s,a) + \alpha(G_t - Q(s,a)) \tag{3.4}$$

where α is the learning rate which controls the amount of increment in Q values at each step. A small value is preferred for α to avoid instability. G_t is first-visit return for the state-action pair (s,a) in a given episode. The full Python code for this algorithm is provided in Listing 3.11. The resulting plots are shown in Figure 3.4. It can be seen that the agent prefers to hit when the player sum is less than 16 with usable ace. On the other hand, with no usable ace, the agent prefers to hit only when the player sum is less than 12. In others words, it prefers to stick more often without an usable ace. The agent explores more in the beginning starting with $\epsilon = 1$. This explorations decreases over time as the training proceeds and the agent starts exploiting its past experience increasingly over time. This is achieved by decaying ϵ through the iterations.

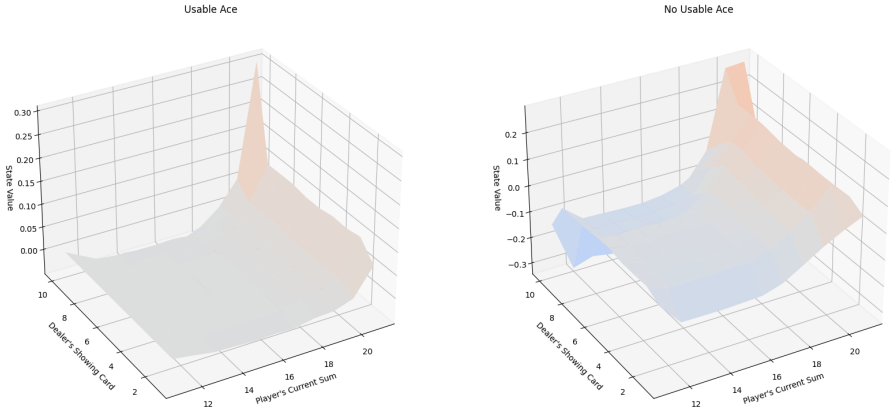
Algorithm 3.2 First-Visit Constant-α GLIE MC Control Algorithm

Input: *num_episodes*, $0 < \alpha < 1$, GLIE ϵ_i
Output: policy π ($\approx \pi^*$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0\ \forall s \in \mathscr{S}$, $a \in \mathscr{A}$)
for $i \leftarrow 1$ **to** *num_episodes* **do**
    $\epsilon \leftarrow \epsilon_i$
    $\pi \leftarrow \epsilon-$Greedy(Q)
    Generate an episode $S_0, A_0, R_1, \cdots, S_T$ using π
    for $t \leftarrow 0$ **to** *num_episodes* **do**
        if (S_t, A_t) is a first visit with return G_t **then**
            $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$
        end if
    end for
end for
return π

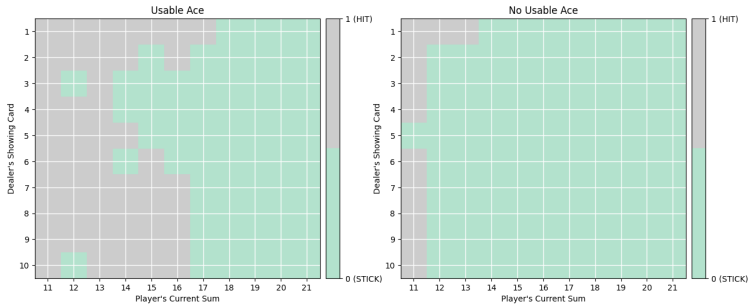
```
import numpy as np
import sys
import random
from collections import defaultdict

class MCAgent():
    def __init__(self, env, alpha=0.0001, gamma=0.99, ep_decay=0.9999):
        self.env = env
        self.n_action = self.env.action_space.n
        self.alpha = alpha
        self.gamma = gamma
        print('Environment name: ', self.env.spec.name)

    def best_policy(self, Q):
        policy = dict((k, np.argmax(v)) for k, v in Q.items())
        return policy

    def epsilon_greedy_policy(self, state, Q, epsilon):
        if random.uniform(0, 1) < epsilon: # explore
            action = self.env.action_space.sample()
        else: # exploit
            action = np.argmax(Q[state,:])
        return action

    def generate_episode(self, Q, epsilon):
        states, actions, rewards = [], [], []
        state = self.env.reset()[0]
        while True:
            states.append(state)
            action = self.epsilon_greedy_policy(state, Q, epsilon)
            actions.append(action)
            next_state, reward, done, info, _ = env.step(action)
            rewards.append(reward)
            state = next_state
            if done:
                break
        return (states, actions, rewards)

    def update_Q(self, episode, Q):
        returns = 0
        states, actions, rewards = episode
        for t in range(len(states) - 1, -1, -1): # traverse in reverse order
            s = states[t]
```

```
            a = actions[t]
            r = rewards[t]
            returns += r * (self.gamma ** t) # discounted rewards
            if s not in states[:t]:   # if S is a first visit (last index is ignore)
                Q[s][a] += self.alpha * (returns - Q[s][a])
        return Q

    def mc_control(self, num_episodes=500000):
        Q = defaultdict(lambda: np.zeros(self.n_action))
        epsilon = 1.0
        eps_min = 0.0001
        decay = 0.9999
        for i in range(num_episodes):
            if i % 1000 == 0:
                print('\rEpisode: {}/{}.'.format(i, num_episodes), end="")
                sys.stdout.flush()

            episode = self.generate_episode(Q, epsilon)
            Q = self.update_Q(episode, Q)
            self.epsilon = max(epsilon * decay, eps_min)
            policy = self.best_policy(Q)
        return policy, Q

    def __delete__(self):
        self.env.close()

if __name__ == '__main__':
    env = gym.make("Blackjack-v1")
    agent = MCAgent(env)

    # Learn optimal policy
    policy, Q = agent.mc_control(num_episodes=500000)

    # Compute value function
    V = dict((k, np.max(v)) for k, v in Q.items())

    # plot Value & Policy
    plot_blackjack_values(V)
    plot_blackjack_policy(policy)
```

Listing 3.11: Python code for implementing First-Visit GLIE Monte-Carlo Control algorithm.



(a) Estimated Values



(b) Estimated Policy

Figure 3.4.: Estimated Value and Policy for Blackjack Environment obtained with Monte-Carlo Control algorithm.

3.4.1. On-Policy & Off-Policy Monte Carlo Algorithms

The policy used to generate actions during interaction with the environment is called the *behaviour* policy. The policy which is being evaluated and improved is called the *target* policy. In on-policy algorithms, the behaviour policy and the target policy are same. The agent learns directly from the experiences generated using its current policy. REINFORCE and SARSA are the examples of on-policy algorithms.

On the other hand, the target policy and the behavior policy are different. The agent learns from experiences generated with a different policy (usually more exploratory). This allows for learning from potentially better actions without having to execute them directly. Q-learning and expected-SARSA are off-policy algorithms.

The on-policy algorithms are comparatively less efficient in learning the optimal policy as they have limited experiences to learn from. They are however more stable due to direct policy updates. On the other hand, the off-policy algorithms are more efficient in learning from diverse experiences. They can be comparatively less stable due to indirect updates.

3.5. challenges of Using Monte Carlo Simulation for Reinforcement Learning

Despite its many advantages, Monte Carlo simulation also has some challenges:

- Sample complexity: Monte Carlo simulation can be slow to learn, especially in environments with a large number of states and actions. This is because Monte Carlo simulation needs to generate a large number of samples in order to get accurate estimates of the value of states and actions.
- Variance: Monte Carlo simulation can be sensitive to the initial policy. If the initial policy is very poor, then it can take a long time for Monte Carlo simulation to learn a good policy.
- Exploration: Monte Carlo simulation can be difficult to use in environments where the agent can only collect a small number of samples. This is because Monte Carlo simulation needs to explore different state-action sequences in order to learn a good policy.
- Monte Carlo methods are generally applied to episodic tasks. This is because of their reliance on the concept of discounted return, which requires a well-defined end to each episode to calculate the return for a state. Hence, Monte Carlo methods are not very suitable for solving continuous or non-episodic tasks.

3.6. Conclusion

Monte Carlo simulation is a powerful and versatile tool for reinforcement learning. It is a model-free approach that relies on sampling to estimate the average value of a given

quantity. It can be used to solve a wide variety of RL problems, including games, robotics, and finance. However, Monte Carlo simulation also has some challenges, such as sample complexity, variance, and exploration. Two types of Monte Carlo algorithms, namely, first-visit and every-visit were discussed to estimate value functions. It is also shown how Monte Carlo algorithms can be used for learning optimal policy.

4. Temporal Difference Learning

4.1. Introduction

In the previous chapter, we studied about Monte Carlo method that can be used for solving the Markov Decision Process (MDP) when the environment model is not known. We showed how Monte Carlo methods can be used for predicting value functions and finding optimal control strategies. One of the limitations of the Monte Carlo methods is that they can be applied only to episodic tasks because of their reliance on the concept of discounted returns which can only computed for an episode with an well-defined end. In this chapter we will talk about another model-free learning approach called *Temporal Difference Learning* that can be used for solving non-episodic tasks. In this process, we will learn two particular TD learning algorithms, namely, Q-learning and SARSA.

4.2. TD Learning

The Temporal Difference (TD) learning, introduced by Sutton [11], uses *bootstrapping* to estimate the value function wherein the current estimates are approximated based on previously learned estimates. In this sense, TD learning takes benefit from both Monte Carlo methods as well as Dynamic Programming. It does not require model information like The MC method and at the same time, it does not wait till the end of the episode to estimate the value function like the DP algorithm. Bellman's Recursive equation (2.10) is a form of Bootstrapping. The temporal difference learning works by updating the current estimate of the state value function $V(s)$ with an error value based on the estimate of the next state. This is given by the following equation:

$$V(s) = V(s) + \alpha[r + \gamma V(s') - V(s)] \tag{4.1}$$

where s is the current state and s' is the next state and r is reward obtained while transitioning from s to s'. α is the learning rate that controls the step size of this incremental update and γ discount rate that determines how much importance is assigned to immediate rewards compared to the future rewards. The difference term $(r + \gamma V(s') - V(s))$ is called the *TD error* as it is difference between the target value estimate and the current value estimate.

The above equation can be used iteratively to predict value function as we did in the case of Monte Carlo methods. The steps involved in the TD-prediction algorithm are as follows:

1. Initialize $V(s)$ to 0 or some arbitrary values.

2. Generate an episode (s, a, r, s') using a policy π.
3. Now update the value function using the TD update rule given by equation (4.1).
4. Repeat last two steps until convergence.

We will now use this understanding to learn optimal policy for a given problem as discussed in the next section.

4.3. TD Control Algorithms

In this section, we will see how we can optimize the value function to learn the optimal control policy. Specifically, we will discuss the following two algorithms: (1) an off-policy learning algorithm called Q learning and (2) an on-policy learning algorithm called SARSA.

4.4. Q Learning

Q-learning is a popular *off-policy* TD control algorithm where the state-action value or Q function is updated using the following equation:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{4.2}$$

The steps involved in Q learning are as follows:

1. Initialize Q function to some arbitrary values.
2. Derive policy from this Q function using an $\epsilon-$greedy policy.
3. Update the Q values by using equation (4.2).
4. Repeat the above two steps until we reach the terminal state.

The Python code for implementing Q learning is provided in the Listing 4.12. Here, an $\epsilon-$ greedy policy is used as the behavioural policy to generate experiences while the Q table is updated using action leading to maximum Q value (greedy policy). In this sense, this is an *off-policy* learning algorithm. The code provides the flexibility to choose between a fixed epsilon value or decaying the epsilon value over time through the constructor argument `fixed_epsilon`. It is also possible to store the training values in a file to generate plots afterwards.

```
import gymnasium as gym
import numpy as np
import sys
import time
```

```
class QLearningAgent():
    def __init__(self, env, alpha=0.3, gamma=0.99, fixed_epsilon=None):
        self.env = env
        self.alpha = alpha
        self.gamma = gamma
        if fixed_epsilon is None:
            self.epsilon = 1.0
            self.eps_min = 0.01
            self.decay_rate = 0.999
            self.decay_flag = True
        else:
            self.epsilon = fixed_epsilon
            self.decay_flag = False

        print('Environment Name: ', self.env.spec.name)
        print('RL Agent: ', 'Q-Learning')

        # initialize Q table
        self.Q = np.zeros((self.env.observation_space.n,
                           self.env.action_space.n))

    def epsilon_greedy_policy(self, state, epsilon):
        randvar = np.random.uniform(0, 1)
        if randvar < epsilon:
            action = self.env.action_space.sample() # explore
        else:
            if np.max(self.Q[state]) > 0:
                action = np.argmax(self.Q[state]) # exploit
            else:
                action = self.env.action_space.sample() # explore
        return action


    def update_q_table(self, s, a, r, s_next):
        self.Q[s][a] += self.alpha * (r + self.gamma * \
                                      np.max(self.Q[s_next]) - self.Q[s][a])

    def train(self, num_episodes=1000, filename=None, freq=100):
        if filename is not None:
            file = open(filename, "w")

        ep_rewards = []
        start = time.time()
        for i in range(num_episodes):
```

```
            ep_reward = 0
            # reset the environment for each episode
            state = self.env.reset()[0]
            while True:
                # select an action
                action = self.epsilon_greedy_policy(state, self.epsilon)
                # obtain rewards
                next_state, reward, done, _, _ = self.env.step(action)
                # update q table
                self.update_q_table(state, action, reward, next_state)
                # accumulate rewards for the episode
                ep_reward += reward
                # prepare for next iteration
                state = next_state
                if done: # end of episode
                    ep_rewards.append(ep_reward)
                    break
            #end of while loop
            if self.decay_flag: # allow epsilon decay
                self.epsilon = max(self.epsilon * self.decay_rate, self.eps_min)

            if filename is not None:
                file.write("{}\t{}\n".format(np.mean(ep_rewards), self.epsilon))
            if i % freq == 0:
                print('\rEpisode: {}/{}, Average episodic Reward:{:.3f}.'\
                      .format(i, num_episodes, np.mean(ep_rewards)), end="")
                sys.stdout.flush()
        #end of for loop
        end = time.time()
        print('\nTraining time (seconds): ', (end - start))
        if filename is not None:
            file.close()

    def validate(self, num_episodes=10):
        ep_rewards = []
        for i in range(num_episodes):
            state = self.env.reset()[0]
            ep_reward = 0
            while True:
                action = self.epsilon_greedy_policy(state, epsilon=0)
                next_state, reward, done, _, _ = self.env.step(action)
                ep_reward += reward
                state = next_state
                if done:
```

```
                ep_rewards.append(ep_reward)
                break
        print('\nTest: Average Episodic Reward: ', np.mean(ep_rewards))

    def display_q_table(self):
        print("\n ------------------ \n")
        print(self.Q)
        print("\n ------------------ \n")

    def __delete__(self):
        self.env.close()
```

Listing 4.12: Python Code for Q Learning Algorithm.

The performance of Q learning algorithm in solving the gym's `FrozenLake-v1` and `Taxi-v3` is shown in the code listing 4.13 and 4.14 respectively. The Frozen Lake is comparatively a simpler problem compared to the taxi problem.

```
if __name__ == '__main__':
    env = gym.make('FrozenLake-v1', is_slippery=False)
    agent = QLearningAgent(env, alpha=0.1)
    agent.train(num_episodes=1000, filename='flake_qlearn.tsv', freq=1000)
    agent.validate()
```

```
Environment Name:  FrozenLake
RL Agent:  Q-Learning
Episode: 900/1000, Average episodic Reward:0.230.
Training time (seconds):  0.24520111083984375
Test: Average Episodic Reward:  1.0
```

Listing 4.13: Outcome of applying Q Learning to solve the Frozen Lake problem

```
if __name__ == '__main__':
    env = gym.make('Taxi-v3', is_slippery=False)
    agent = QLearningAgent(env, alpha=0.1)
    agent.train(num_episodes=20000, filename='taxi_qlearn.tsv', freq=1000)
    agent.validate()
```

```
Environment Name:  Taxi
RL Agent:  Q-Learning
Episode: 19000/20000, Average episodic Reward:-92.656..
Training time (seconds):  31.19220733642578
Test: Average Episodic Reward:  8.1
```

Listing 4.14: Outcome of applying Q Learning to solve the Frozen Lake problem

4.5. SARSA Algorithm

SARSA (State-Action-Reward-State-Action) is an *on-policy* TD algorithm. In SARSA, the state-action value (Q) function is updated using the following equation:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a)) \tag{4.3}$$

where a' is the action selected by the current policy for the next state s'.

The steps involved in SARSA algorithm are as follows:

1. Initialize Q values to some arbitrary values.
2. Select $\epsilon-$greedy policy for state transitions.
3. Update Q values using equation (4.3) where a' is the action selected by the $\epsilon-$greedy policy for the next state s'.
4. Repeat above three states until the terminal state is reached.

The SARSA algorithm uses the same $\epsilon-$greedy policy both for generating its current experiences (behaviour policy) and for updating Q table (learning target policy). The Python code for SARSA algorithm implementation is provided in the Listing 4.15. The code is very similar to that of Q-learning code provided in listing 4.12 except for the `upate_q_table()` function.

```
import gymnasium as gym
import numpy as np
import sys
import time

class AgentSARSA():
    def __init__(self, env, alpha=0.3, gamma=0.99, fixed_epsilon=None):
        self.env = env
```

```
        self.alpha = alpha # learning rate
        self.gamma = gamma # discount factor
        if fixed_epsilon is None:
            self.epsilon = 1.0      # exploration probability
            self.eps_min = 0.01
            self.decay_rate = 0.999
            self.decay_flag = True
        else:
            self.epsilon = fixed_epsilon
            self.decay_flag = False

        print('Environment Name: ', self.env.spec.name)
        print('RL Agent: ', 'SARSA')

        # initialize Q table
        self.Q = np.zeros((self.env.observation_space.n,
                           self.env.action_space.n))


    def epsilon_greedy_policy(self, state, epsilon):
        randvar = np.random.uniform(0, 1)
        if randvar < epsilon:
            action = self.env.action_space.sample() # explore
        else:
            if np.max(self.Q[state]) > 0:
                action = np.argmax(self.Q[state]) # exploit
            else:
                action = self.env.action_space.sample() # explore
        return action


    def update_q_table(self, s, a, r, s_next, a_next):
        self.Q[s][a] += self.alpha * (r + self.gamma * \
                        self.Q[s_next][a_next] - self.Q[s][a])

    def train(self, num_episodes=1000, freq=1000, filename=None):
        if filename is not None:
            file = open(filename, "w")
        ep_rewards = []
        start = time.time()
        for i in range(num_episodes):
            ep_reward = 0
            state = self.env.reset()[0] #reset the environment
            action = self.epsilon_greedy_policy(state, self.epsilon)
```

```
            while True:
                # get next_state and reward
                next_state, reward, done, _, _ = self.env.step(action)

                # get action for next_state
                next_action = self.epsilon_greedy_policy(next_state, self.epsilon)

                # update q table
                self.update_q_table(state, action, reward, \
                                    next_state, next_action)

                # epsodic reward
                ep_reward += reward

                # prepare for next iteration
                state = next_state
                action = next_action
                if done:
                    ep_rewards.append(ep_reward)
                    break
            #end of while loop
            if self.decay_flag: # allow epsilon decay
                self.epsilon = max(self.epsilon * self.decay_rate, self.eps_min)

        if filename is not None:
            file.write("{}\t{}\n".format(np.mean(ep_rewards), self.epsilon))
        if i % freq == 0:
            print('\rEpisode: {}/{}, Average episodic Reward:{:.3f}.'\
                  .format(i, num_episodes, np.mean(ep_rewards)), end="")
            sys.stdout.flush()
    #end of for loop
    end = time.time()
    print('\n Training Time: ', (end-start))
    if filename is not None:
        file.close()

def validate(self, num_episodes=10):
    ep_rewards = []
    for i in range(num_episodes):
        state = self.env.reset()[0]
        ep_reward = 0
        while True:
            action = self.epsilon_greedy_policy(state, epsilon=0)
            next_state, reward, done, _, _ = self.env.step(action)
```

```
            ep_reward += reward
            state = next_state
            if done:
                ep_rewards.append(ep_reward)
                break
    print('\nAverage Episodic Reward: ', np.mean(ep_rewards))

def display_q_table(self):
    print("\n ------------------ \n")
    print(self.Q)
    print("\n ------------------ \n")

def __delete__(self):
    self.env.close()
```

Listing 4.15: Python Code for SARSA Algorithm

The performance of SARSA algorithm in solving the `Taxi` and the `FrozenLake` problem is shown in the code listings 4.16 and 4.17 respectively. Comparing these outcomes with those provided in the code listing 4.14 and 4.13, we find that SARSA takes longer time to train over the same number of episodes in the case of `Taxi` problem with poor performance (lower episodic reward). The performance is almost similar in case of `Frozen Lake` problem. The same is reflected in the plot shown in Figure 4.1. We use decaying epsilon for both the algorithms.

```
if __name__ == '__main__':
    env = gym.make('Taxi-v3')
    agent = AgentSARSA(env, alpha=0.1)
    agent.train(num_episodes=20000, filename='taxi_sarsa.tsv')
    agent.validate()
```

```
Environment Name:  Taxi
RL Agent:  SARSA
Episode: 19000/20000, Average episodic Reward:-1849.457.
Training Time:  261.57872462272644
Test:Average Episodic Reward:  -409.2
```

Listing 4.16: Output of SARSA algorithm when applied to solve the 'Taxi-v3' problem.

```
if __name__ == '__main__':
    env = gym.make('FrozenLake-v1', is_slippery=False)
```

```
agent = AgentSARSA(env, alpha=0.1)
agent.train(num_episodes=1000, filename='flake_sarsa.tsv')
agent.validate()
```

```
Environment Name:  FrozenLake
RL Agent:  SARSA
Episode: 900/1000, Average episodic Reward:0.238.
Training Time:  0.20873093605041504
Test:Average Episodic Reward:  1.0
```

Listing 4.17: Output of SARSA Algorithm when applied to solve the Frozen Lake problem

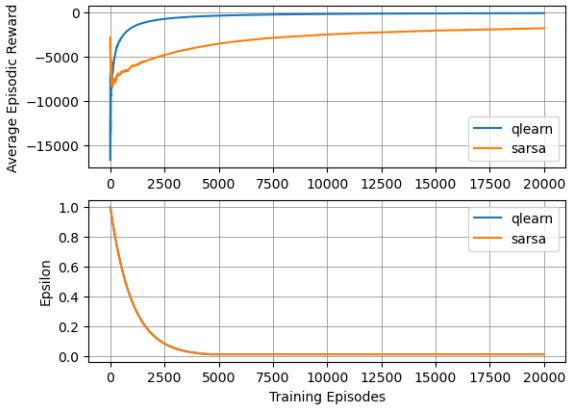


Figure 4.1.: Comparing the training performance of SARSA and Q learning algorithm for `Taxi-v3` environment.

4.6. Difference between Q-Learning and SARSA Algorithms

Q-Learning and SARSA are both popular algorithms in reinforcement learning for finding the optimal policy in an unknown environment. They share many similarities but also have a key difference in how they update their values, leading to distinct advantages and disadvantages.

Here's a breakdown of their key differences:

1. Update Rule:
    - Q-Learning: This algorithm is off-policy, meaning it learns the optimal policy regardless of the policy currently being followed. During updates, it uses the

maximum Q-value for the next state, essentially assuming the best possible action will be taken from that point. This makes it more optimistic and can lead to faster convergence to the optimal policy.

- SARSA: This algorithm is on-policy, meaning it updates its values based on the policy it's currently following (usually an epsilon-greedy policy). It uses the Q-value of the next state and the action actually taken according to the policy. This makes it more realistic and less prone to overestimation, but it can also be slower to converge and potentially get stuck in sub-optimal policies.

2. Exploration-Exploitation Trade-off:
   - Q-Learning: Since it uses the maximum Q-value for the next state, Q-learning implicitly encourages exploration by considering all possible actions even if not currently chosen. This can be helpful in navigating complex environments.
   - SARSA: Because it relies on the actual action taken, SARSA might struggle with exploration if the chosen action leads to areas with low rewards. This can make it prone to getting stuck in local optima.
3. Convergence:
   - Q-Learning: Generally, Q-learning converges faster to the optimal policy due to its optimistic update rule and implicit exploration. However, it may require a larger learning rate and can exhibit some instability during learning.
   - SARSA: While often slower to converge, SARSA's updates are more stable and realistic. It also offers better guarantees of convergence under certain conditions.

To summarize, the choice between these algorithms depends on your specific needs:

- Q-Learning: Choose it if you want faster convergence, implicit exploration, and flexibility in switching policies.
- SARSA: Choose it if you prioritize stability, realistic updates, and guaranteed convergence (in some cases). Ultimately, experimenting with both algorithms in your specific environment can help you determine which one performs better.

4.7. Conclusion

Temporal difference learning approaches use a bootstrapping process to estimate the value functions where in the current estimates are approximated based on past estimates. These are model-free approaches that can be applied non-episodic tasks as well thereby deriving the best of both Monte-Carlo and Dynamic Programming approaches. Specifically, we review two popular TD approaches namely, Q-learning and SARSA algorithms and critically analyzes their advantages and disadvantages.

5. Deep Q Network

5.1. Introduction

Let's recap what we have studied so far about Q function. Q function, also known as state-action value function, is the cumulative future reward that an agent can obtain by taking an action a at state s through a policy π. In other words, Q function specifies how good is an action a at state s. The values of all possible actions for all states are stored in matrix called Q-table. A *greedy* policy selects an action that results in maximum Q value at a given state. In the previous chapter, we studied two algorithms namely, Q-learning and SARSA to estimate the Q function using temporal difference learning. These algorithms are applicable only to problems with discrete state and action spaces and suffer from *curse-of-dimensionality* with exponentially increasing computational cost as the number of states or actions increase.

Deep Q Network (DQN) solves the problem associated with discrete states by using a deep network to estimate the Q function which can take any arbitrary input observation (including continuous, image or text observations). This parametric Q function is called a deep Q network (DQN) and is denoted by $Q(s, a; \theta)$. This network is trained by minimizing a loss function between predicted value $Q(s, a; \theta)$ and a target value as will be discussed in the next section.

5.2. DQN Algorithm

The Q-learning update equation from the last chapter is given by:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{5.1}$$

where $Q(s_t, a_t)$ is the predicted value and $r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ is the target value. This target value can be used to create a loss function that can be minimized to learn the network parameters θ. This loss function for iteration step t is given by:

$$L_t(\theta_t) = \mathbb{E}_{(s,a)\sim P(s,a)}[y_t - Q(s_t, a_t; \theta_t)]^2 \tag{5.2}$$

where $y_t = r + \gamma \max_{a'} Q(s', a'; \theta_t)$ is the target value for iteration step t. For a non-deterministic system, the loss function is computed as the expected value over a batch of experiences $\{(s_t, a_t, r_t, s_{t+1}) \cdots\}$ taken from the state-action probability distribution $P(s, a)$. This is achieved by storing experience tuples $< s_t, a_t, r_t, s_{t+1} >$ in a replay buffer and sampling a batch of experiences for each training step. This form of training is known as *experience replay*. The overall scheme of DQN learning architecture is shown in Figure

5.1. It is shown that *batch training*, where the network parameters are updated only once for each batch of samples, provides faster convergence compared to incremental learning where the parameters are updated after every sample in the batch. The complete DQN algorithm is provided in Algorithm 5.1.

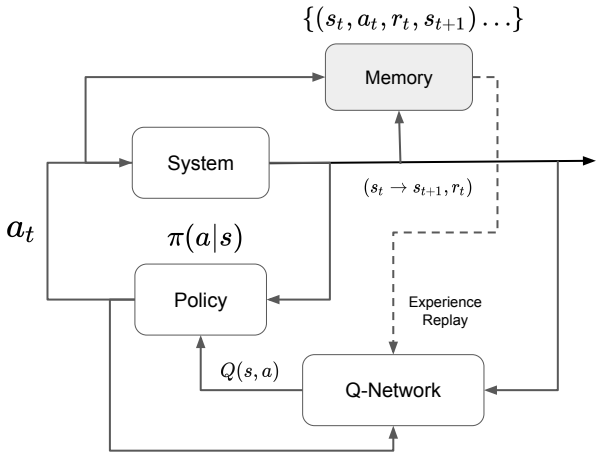


Figure 5.1.: Schematic of DQN Learning Algorithm

5.3. Double DQN Algorithm

The loss function used for training a DQN is given by:

$$L(\theta) = [\underbrace{r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta)}_{\text{target}} - Q(s_t, a_t; \theta)]^2 \tag{5.3}$$

where the target value uses the maximum of estimated values of Q function. This introduces a *maximization bias* as Q learning algorithm uses *bootstrapping* - learning estimates from previous estimates. Such overestimation can be problematic. This overestimation can be solved by double Q learning [12] where two separate Q-value estimators are used to update each other. The modified loss function for double Q learning is given by:

$$L(\theta) = [\underbrace{r + \gamma \max_{a_{t+1}} Q'(s_{t+1}, a_{t+1}; \theta')}_{\text{target}} - Q(s_t, a_t; \theta)] \tag{5.4}$$

where the target uses a different network Q' with parameters θ' which is different from the main value estimator Q with parameters θ. The model Q and Q' share weights at regular interval. One of the models, say, Q is used for *action selection* and the other

Algorithm 5.1 DQN Algorithm

Initialize Replay memory R to capacity N.
Initialize the Q-network $Q(s, a; \theta)$ with random weights θ.
for $i = 1$ to max_iteration **do**
    Observe state s_t
    Select action using *epsilon-greedy* policy
    Transition to next state and collect rewards
    Store (s_i, a_i, r_i, s_{i+1}) into replay buffer R.
    for each update step **do**
        Randomly sample a batch B of transitions from R.
        if done = True **then** ▷ terminal state
            $y_i = r_i$
        else
            $y_i = r_i + \gamma \max_a Q(s_{i+1}, a; \theta_i)$
        end if
        Perform gradient descent to minimize error: $L_i(\theta_i) = \sum_i^{|B|}[y_i - Q(s_i, a_i; \theta_i)]^2$
    end for
end for

model, say, Q' is used for *action evaluation* while computing the target Q value as shown below:

$$Q^*(s_t, a_t) = y_t \approx r_t + \gamma \underbrace{Q'(s_{t+1}, \underbrace{\arg\max_{a_t} Q(s_t, a_t)}_{\text{action selection}})}_{\text{action evaluation}} \tag{5.5}$$

The error arising from the difference between Q and Q' is minimized by slowly copying the parameters of Q to Q' through *Polyak averaging* given by

$$\theta' = \tau\theta + (1 - \tau)\theta' \tag{5.6}$$

where $\tau \in (0, 1)$ is the averaging factor. The steps to implement double DQN (DDQN) algorithm is provided in Algorithm 5.2.

5.4. Python implementation of DQN Algorithm

5.4.1. Replay Buffer

As mentioned before, the experience tuple $(s_t, a_t, r_t, s_{t+1}, d)$ generated using behavioural policy is stored in a buffer. A batch of these experiences are sampled randomly from the buffer during each training step. This form of training is called *experience replay*. A Python class for implementing this replay buffer is provided in the code Listing 5.18. The replay buffer has a fixed size. When the buffer is full, the data is overwritten from the beginning of the buffer by using circular indexing.

Algorithm 5.2 Double DQN (DDQN) Algorithm

Initialize primary network Q_θ and target network $Q_{\theta'}$, replay buffer R and $\tau << 1$.
for each iteration **do**
    Observe the state s_t and take action a_t using epsilon-greedy policy.
    Transition to next state s_{t+1} and collect reward r_t
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer R.
    for each update step **do**
        sample a batch B of episodes from replay buffer R.
        Compute target Q value using equation (5.5).
        Perform gradient descent step to minimize $\sum_{|B|}(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$.
        Update target network parameters using equation (5.6)
    end for
end for

```python
import numpy as np
class ReplayBuffer():
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = np.zeros(self.capacity, dtype=object)
        self.idx = 0
        self.full = False

    def add(self, experience:tuple):
        self.buffer[self.idx] = experience
        self.idx = (self.idx + 1) % self.capacity
        # set this flag if buffer is full
        self.full = self.full or self.idx == 0

    def sample(self, batch_size=24):
        indices = np.random.randint(0, self.capacity \
                                    if self.full else self.idx,
                                    size=batch_size)
        batch = self.buffer[indices]
        return batch

    def __getitem__(self, index):
        if index >= 0 and index < self.capacity \
            if self.full else self.idx: # sanity check
            return self.buffer[index]
        else:
            raise ValueError('Index is out of range')
```

```
    def __len__(self):
        # return the current length of buffer
        return self.capacity if self.full else self.idx
```

Listing 5.18: Python class for implementing Replay Buffer.

5.4.2. The DQN Class

The Python code for implementing a DQN agent class is provided in the code Listing 5.19. It provides an option to select between DQN and Double DQN architecture. It also allows passing a deep network model created externally. It creates a replay buffer memory by instantiating the `ReplayBuffer` class provided in code Listing 5.18. The experiences can be stored using the class method `store_experience()`. The class provides methods to save and load Q network parameters.

```
import sys
import os
import random
import tensorflow as tf
import keras
from keras.layers import Dense
from keras.optimizers import Adam
from keras.models import Sequential

class DQNAgent:
    def __init__(self, obs_shape: tuple, n_actions: int,
                 buffer_size=2000, batch_size=24,
                 ddqn_flag=True, model=None):

        self.obs_shape = obs_shape
        self.action_size = n_actions   # number discrete actions
        self.ddqn = ddqn_flag   # choose between DQN & DDQN

        # hyper-parameters for DQN
        self.gamma = 0.99    # discount factor
        self.epsilon = 1.0    # exploration rate - epsilon-greedy policy
        self.epsilon_decay = 0.999
        self.epsilon_min = 0.01
        self.batch_size = batch_size
        self.buffer_size = buffer_size
        self.train_start = 1000  # minimum buffer size to start training
        self.learning_rate = 0.001 # learning rate for the Deep Network
```

```
        # create a replay buffer to store experiences
        self.memory = ReplayBuffer(self.buffer_size)

        # create main model & target model   - DDQN Architecture
        if model is None:
            self.model = self._build_model()
            self.target_model = self._build_model()
        else:
            self.model = model
            self.target_model = tf.keras.models.clone_model(model)

        # initialize target model
        self.target_model.set_weights(self.model.get_weights())


    def _build_model(self):
        model = keras.Sequential([
            keras.layers.Dense(24, input_shape=self.obs_shape,
                               activation='relu',
                               kernel_initializer='he_uniform'),
            keras.layers.Dense(24, activation='relu',
                               kernel_initializer='he_uniform'),
            keras.layers.Dense(self.action_size, activation='linear',
                               kernel_initializer='he_uniform')
        ])
        model.summary()
        model.compile(loss='mse', optimizer=Adam(
                      learning_rate=self.learning_rate))
        return model

    def update_target_model(self, tau):
        pass

    def get_action(self, state):
        pass

    def store_experience(self, state, action, reward, next_state, done):
        self.memory.add((state, action, reward, next_state, done))

    def get_target_q_value(self, s_next):
        pass

    def experience_replay(self):
        pass
```

```
    def update_epsilon(self):
        pass

    def save_model(self, filename):
        self.model.save_weights(filename)

    def load_model(self, filename):
        self.model.load_weights(filename)
```

Listing 5.19: (Double) DQN agent class template

5.4.3. Epsilon-Greedy Policy

The class method `get_action()` implements the *epsilon-greedy* policy to solve the exploration vs exploitation dilemma during the learning process. The hyperparameter $\epsilon \in (0, 1)$ controls the exploration rate. A uniform random number is generated between 0 and 1. The agent takes random action (explore) if this random number is less than ϵ. Otherwise, it takes a *greedy* action (action resulting in maximum Q value) based on past experience. The training starts with $\epsilon = 1$ resulting in high exploration. The exploration rate is gradually reduced over time by using the class method `update_epsilon()` thereby allowing the agent to exploit the past knowledge. During the testing phase, exploration rate is set to 0 making the agent to take actions based only on a greedy policy. The definitions for the above two methods are provided in the code Listing 5.20.

```
class DQNAgent()
    def get_action(self, state, epsilon=None):
        # epsilon-greedy policy
        if epsilon is None:
            epsilon = self.epsilon   # decaying epsilon
        if np.random.rand() <= epsilon: # explore
            return random.randrange(self.action_size)
        else:
            q_value = self.model.predict(state, verbose=0)
            return np.argmax(q_value[0])
```

Listing 5.20: Code for DQN Class: epsilon-greedy policy

5.4.4. Experience Replay

The code for training a DQN is provided in the code Listing 5.21. The training is carried out by a method called *experience replay* where the experiences are first stored in a

replay buffer and then used during the training phase through random sampling. This is implemented by the class method `experience_replay()`. A batch of experiences is sampled from the replay buffer. Target Q values is computed for this batch using equation (5.5). Gradient descent is applied to minimize the error between the predicted Q values and target Q values during each iteration step. The exploration rate ϵ is reduced after each training step.

```
class DQNAgent():
    def get_target_q_value(self, next_states): # batch input
        q_value_ns = self.model.predict(next_states, verbose=0)  # Q(s', :)
        if self.ddqn: ## DDQN algorithm
            # primary model is used for action selection: a = arg max Q(s,a)
            max_actions = np.argmax(q_value_ns, axis=1)
            # use target model for action evaluation: Q'(s',:)
            target_q_values_ns = self.target_model.predict(
                                                    next_states, verbose=0)
            # Q'(s', argmax(Q(s,a)))
            max_q_values = target_q_values_ns[
                        range(len(target_q_values_ns)), max_actions]
        else: # DQN
            max_q_values = np.amax(q_values_ns, axis=1)
        return max_q_values

    def experience_replay(self):
        if len(self.memory) < self.train_start:
            return
        # sample experiences from replay buffer
        batch_size = min(self.batch_size, len(self.memory))
        mini_batch = self.memory.sample(self.batch_size)

        # unwrapping mini_batch tuple
        states = np.zeros((self.batch_size, *self.obs_shape))
        next_states = np.zeros((self.batch_size, *self.obs_shape))
        actions = np.zeros((self.batch_size, 1))
        rewards = np.zeros((self.batch_size, 1))
        dones = np.zeros((self.batch_size, 1))
        for i in range(len(mini_batch)):
            states[i] = mini_batch[i][0]
            actions[i] = mini_batch[i][1]
            rewards[i] = mini_batch[i][2]
            next_states[i] = mini_batch[i][3]
            dones[i] = mini_batch[i][4]

        q_values_cs = self.model.predict(states, verbose=0)
```

```
        max_q_values_ns = self.get_target_q_value(next_states)

        # compute target q value
        for i in range(len(q_values_cs)):
            action = actions[i].astype(int)[0]
            done = dones[i].astype(bool)[0]
            reward = rewards[i][0]
            if done: # terminal state
                q_values_cs[i][action] = reward
            else:
                q_values_cs[i][action] = reward + \
                                         self.gamma * max_q_values_ns[i]

        # train the Q network
        self.model.fit(np.array(states), np.array(q_values_cs),
                       batch_size=batch_size,
                       epochs=1,
                       verbose=0)

        # decay epsilon over time
        self.update_epsilon()

    def update_epsilon(self):
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

Listing 5.21: Code for DQN class: experience replay class

5.4.5. Training an agent for solving a Gym Problem

The code for training a DQN agent for a given problem environment is provided in the code Listing 5.22. The `train()` function primarily takes two arguments, one environment object and a RL agent. The `train()` method iterates over a number of episodes. For each iterative step, the agent observes the environment (s_t), takes an action a_t based on $\epsilon-$greedy policy and then, transitions to the next state (s_{t+1}) while collecting reward r_t. The end of episode is indicated setting the `done` flag to `True`. The experience tuple $(s_t, a_t, r_t, s_{t+1}, done)$ is stored in the agent's replay buffer. These experiences are then sampled to train the agent during each iterative step. The weights of the main model is copied at regular intervals into the target model. The frequency of training updates and target updates is controlled by the arguments `train_freq` and `copy_freq` respectively. In many case, *reward engineering* must be carried out to allow the agent to solve the problem effectively in a reasonable time. For example, in case of `Cartpole-v0` environment, the agent is penalized when the episode terminates.

```
def train(env, agent, max_episodes=300, train_freq=1,
          copy_freq=1, filename=None):
    if filename is not None:
        file = open(filename, 'w')
    #averaging factor - choose between soft & hard update
    tau = 0.1 if copy_freq < 10 else 1.0
    best_score = 0
    scores = []
    avg_score, avg100_score = [], []
    global_step_cnt = 0
    for e in range(max_episodes):
        state = env.reset()[0]
        state = np.expand_dims(state, axis=0)
        done = False
        ep_reward = 0
        t = 0
        while not done:
            global_step_cnt += 1
            # take action
            action = agent.get_action(state)
            # collect reward & transition to next state
            next_state, reward, done, _, _ = env.step(action)
            next_state = np.expand_dims(next_state, axis=0)

            # reward engineering
            # discourages premature termination
            reward = reward if not done else -100

            #store experience
            agent.store_experience(state, action, \
                                   reward, next_state, done)

            state = next_state
            ep_reward += reward
            t += 1

            # train
            if global_step_cnt % train_freq == 0:
                agent.experience_replay()

            # update the target model
            if global_step_cnt % copy_freq == 0:
                agent.update_target_model(tau=tau)
```

```
            # while loop ends here
            if e > 100 and t > best_score:
                agent.save_model('best_model.weights.h5')
                best_score=t
            scores.append(t)
            avg_score.append(np.mean(scores))
            avg100_score.append(np.mean(scores[-100:]))
            if filename is not None:
                file.write(f'{e}\t{t}\t{np.mean(scores)}\
                                \t{np.mean(scores[-100:])}\n')
                file.flush()
                os.fsync(file.fileno())
            if e % 20 == 0:
                print(f'e:{e}, ep_reward:{t}, avg_ep_reward: \
                                        {np.mean(scores):.2f}')
        # end of for loop
        print('end of training')
        file.close()
```

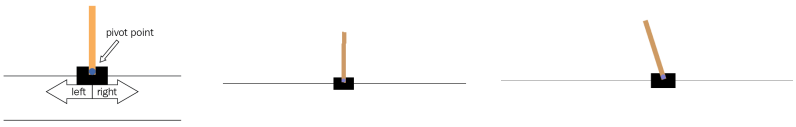
Listing 5.22: Python code for training a DQN for a given Gym Environment

5.4.6. Solving the CartPole problem using DQN algorithm

We will now use the DQN class created above to solve the CartPole-v0 problem. Few snapshots of the simulation environment is shown in Figure 5.2. It consists of a cart (shown in black color) and a vertical bar attached to the cart using passive pivot joint. The cart can move left or right. The problem is to prevent the vertical bar from falling by moving the car left or right. The state vector for this system $\mathbf{x}$ is a four dimensional vector having components $\{x, \dot{x}, \theta, \dot{\theta}\}$. The action has two states: left (0) and right (1). The episode terminates if (1) the pole angle is more than $\pm 12^{\circ}$ from the vertical axis, or (2) the cart position is more than ± 2.4 cm from the centre, or (3) the episode length is greater than 200 (for v0) and 500 (for v1). The agent receives a reward of 1 for every step taken including the termination step. The problem is considered solved, if the average reward is greater than or equal to 195 over 100 consecutive episodes.

The complete code for training a DQN agent to solve Gym's CartPole problem is provided in the code Listing 5.23 along with the program output. The performance of DDQN algorithm is shown in Figure 5.3. As can be seen, the problem is solved in about 100 episodes.

```
    import gymnasium as gym

    if __name__ == '__main__':
    # create gym environment
```



(a) (b) (c)

Figure 5.2.: Visualization of CartPole environment. The task is to balance the pole in the vertical position by controlling the motion of the cart.

```
env = gym.make('CartPole-v0')
obs_shape = env.observation_space.shape
n_actions = env.action_space.n

# create DQN Agent
agent = DQNAgent(obs_shape, n_actions,
buffer_size=2000,
batch_size=24)
# train the agent
train(env, agent, max_episodes=200, copy_freq=100,
filename='cp_dqn.txt')
```

```
e:0, ep reward:24, avg ep reward:24.00
e:20, ep reward:30, avg ep reward:23.05
e:40, ep reward:17, avg ep reward:21.27
e:60, ep reward:29, avg ep reward:21.56
e:80, ep reward:72, avg ep reward:31.91
e:100, ep reward:10, avg ep reward:235.54
e:120, ep reward:210, avg ep reward:210.24
e:140, ep reward:726, avg exp reward:222.46
e:160, ep reward:152, avg ep reward:479.54
e:180, ep reward:479, avg ep reward:458.39
end of training
```

Listing 5.23: Python code for training a DQN agent and its console output

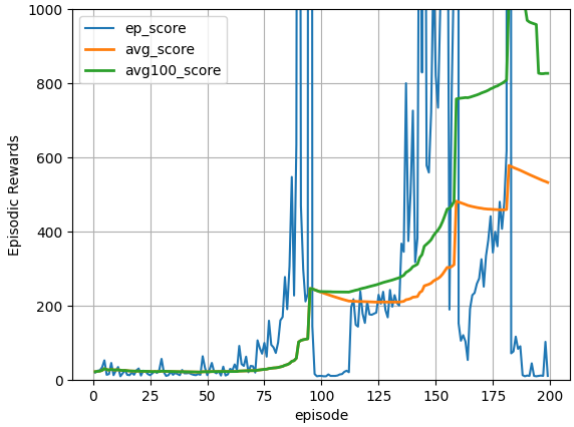


Figure 5.3.: Performance of DDQN algorithm in solving the CartPole problem

5.5. Priority Experience Replay (PER)

In *experience replay*, the experiences are randomly sampled from the replay buffer. All the experiences (transitions) are treated equally during the sampling process. However, some experiences could be more informative for learning than others even when they occur less frequently. Priority experience replay (PER) aims to solve this problem by prioritizing these informative experience tuples (or state transitions). It assigns a priority score to each transition, with higher scores for transitions with larger TD errors. During sample, the transitions (or experience tuples) are chosen with a probability based on their priority scores. The priority score assigned to each sample is given by

$$p_i = |\delta_i| + e \tag{5.7}$$

where δ_i is the magnitude of TD error associated with the sample i and e is a constant value that is added so that no experience is assigned a zero value. So, the experiences along with their priorities are stored in the replay buffer. However, sampling experiences based on priority may lead to *prioritization bias* leading to loss of experience diversity as the high-priority samples will dominate the training process thereby potentially hindering the agent's ability to generalize to unseen situations. This can be solved by using *stochastic prioritization* which introduces a randomization factor during experience sampling to ensure a balance between prioritizing informative transitions and maintaining exploration of the entire replay buffer. This is achieved by selecting the samples with a probability given by:

$$P(i) = \frac{p_i^a}{\sum_k^N p_k^a} \tag{5.8}$$

where a is hyperparameter to introduce randomness in the experience selection for the replay buffer. N is the total number of samples in the replay buffer. If $a = 0$, samples are selected with equal probability (uniform randomness) and if $a = 1$, samples with highest priorities are only selected.

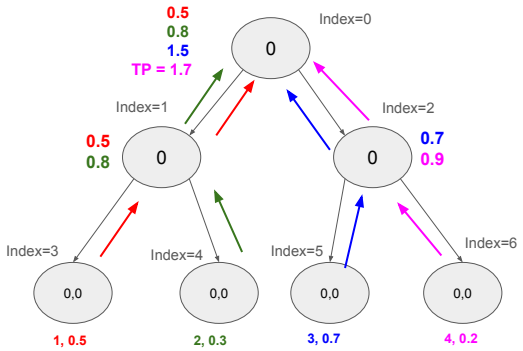


Figure 5.4.: Adding samples with priority to a sum-tree replay buffer

The non-uniform sampling introduced by stochastic prioritization does not completely solve this bias and may still lead to overfitting. To compensate this bias, we use *importance sampling (IS)* weights that are inversely proportional to its sampling probability $P(i)$ given by eqn. (5.8). So, the higher priority experiences will have lower weights. This will effectively reduce their influence on the training. The weights for each sample is given by:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^b \tag{5.9}$$

The role of b is to control how much these importance sampling weights affect the learning. We will use a binary sum tree to store priority experiences providing a $O(\log n)$ time complexity in storing and retrieving samples.

5.5.1. Sum-Tree Data Structure for Priority Replay Buffer

A Sum-tree is a binary tree where the value at each node is the sum of the values of its children. In a binary tree, each node has at most two child nodes. If n is the replay buffer size then, the total number of nodes in the tree will be $2n-1$ which can be considered as the length of tree. The data (or the experiences) are stored in the n leaf nodes of the sum-tree. The index of leaf node is $i+n-1,\ i=0,1,2,\cdots,n-1$. Each leaf node stores both data (experience tuple) and priority while other nodes only store priority. Initially, all data and priorities are set to 0. The root node stores the total priority.

Adding experience to the replay buffer

The process of adding experiences to the sum-tree can be better understood by analysing the Figure 5.4 that shows a sum-tree buffer with a capacity to store only 4 samples

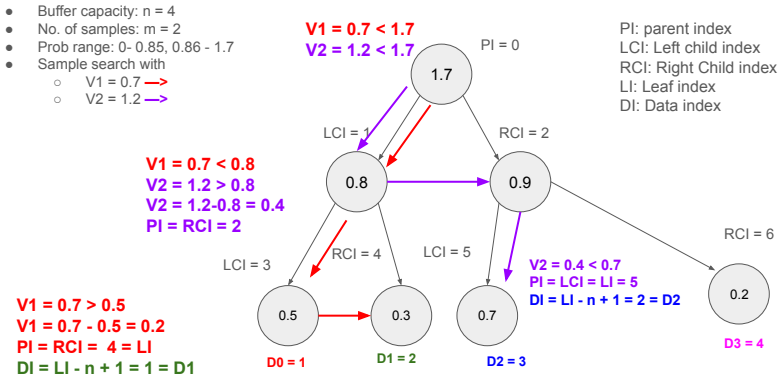


Figure 5.5.: Retrieving samples with priority from a sum-tree replay buffer

($n = 4$). The leaf indices of this tree are 3, 4, 5 and 6 respectively. Whenever a data is added to the leaf node, the change in priority is propagated upwards until the root node. Let's assume that the data and priority being added to each of the leaf nodes are (1,0.5), (2, 0.3), (3, 0.7) and (4, 0.2) respectively. The priority of the non-leaf nodes are 0.8 at node 1, 0.9 at node 2 and 1.7 at node 0. The total priority (TP) in this case is 1.7. The updates occurring at each instance of sample adding is shown in different colors.

Retrieving samples from the buffer

While retrieving m samples ($m \leq n$), we divide the range between 0 and total priority (TP) into m segments and then select one sample uniformly from each segment. The process of retrieving samples with a given priority value v is demonstrated in Figure 5.5. Let us assume that we want to retrieve 2 samples ($m = 2$) having priority values $v_1 = 0.7$ and $v_2 = 1.2$ from the sum-tree created above. The colored arrows show the process flow during the retrieval process starting from the root node. The data points retrieved are $D1 = 2$ and $D2 = 3$ respectively. Let's understand the process by considering the priority value $v_1 = 0.7$ first. It starts by comparing v_1 with the root node value which is the total priority (TP=1.7). The value being less, v_1 is now compared with the left child node with index 1 having a priority value of 0.8. The value being less again, it moves to the next left child node with index 3 whose priority value is 0.5. Here $v_1 > 0.5$ and hence, the right child node with index 4 is selected for the next step. Also, the priority of the left child node is updated to have new priority value = 0.7 - 0.5 = 0.3. The right child node with index 4 being a leaf node, the data point D1=2 is retrieved (shown in green color). The violet arrow shows the steps involved in retrieving the second data sample with priority value of 1.2.

5.5.2. Python Implementation of Sum-Tree class

The Python code for implementing a Sum-tree class is provided in the code Listing 5.24. It uses two arrays - one for storing data and other for storing priorities. The `self.data` buffer has a length n which is the maximum capacity of the replay buffer. The size of the `self.tree` buffer is $2n - 1$. The data buffer is overwritten starting from the beginning when it is full indicated by the `self.full` flag. The priority stored in `self.tree` buffer is updated whenever a data is added at a leaf node. The root node stores the total priority.

```
class SumTree(object):
    # Here we initialize the tree with all nodes = 0,
    # and initialize the data with all values = 0
    def __init__(self, capacity):
    # Number of leaf nodes (final nodes) that contains experiences
    self.capacity = capacity
    self.data_pointer = 0
    self.full = False   # indicates if the buffer is full
    self.tree = np.zeros(2 * capacity - 1)  # contains priorities
    # Contains the experiences (so the size of data is capacity)
    self.data = np.zeros(capacity, dtype=object)

    def add(self, priority, data):
    # data is stored at the leaf of the tree from index: n-1 to 2*n-1
    tree_index = self.data_pointer + self.capacity - 1
    # Update data frame
    self.data[self.data_pointer] = data
    # Update the leaf
    self.update(tree_index, priority)
    # Add 1 to data_pointer
    self.data_pointer += 1
    # If above capacity, go back to first index to overwrite
    if self.data_pointer >= self.capacity:
    self.data_pointer = 0
    self.full = True

    def __len__(self):  # returns the size of data buffer only
    return self.capacity if self.full else self.data_pointer

    def __getitem__(self, index):
    # return data and priority at index i
    if index >= 0 and index < self.capacity \
    if self.full else self.data_pointer:
    tree_idx = index + self.capacity - 1
    return self.data[index], self.tree[tree_idx]
```

```
        else:
            raise ValueError('index out of range')


    def update(self, tree_index, priority):
        # Change = new priority score - former priority score
        change = priority - self.tree[tree_index]
        self.tree[tree_index] = priority

        # then propagate the change through tree
        # this method is faster than the recursive loop
        while tree_index != 0:
            tree_index = (tree_index - 1) // 2
            self.tree[tree_index] += change

    def get_leaf(self, v):
        parent_index = 0
        while True:
            left_child_index = 2 * parent_index + 1
            right_child_index = left_child_index + 1

            # If we reach bottom, end the search
            if left_child_index >= len(self.tree):
                leaf_index = parent_index
                break
            else: # downward search, always search for a higher priority node
                if v <= self.tree[left_child_index]:
                    parent_index = left_child_index
                else:
                    v -= self.tree[left_child_index]
                    parent_index = right_child_index

        data_index = leaf_index - self.capacity + 1
        return leaf_index, self.tree[leaf_index], self.data[data_index]

    @property
    def total_priority(self):
        return self.tree[0] # Returns the root node
```

Listing 5.24: Python implementation of Sum-tree class

5.5.3. Python implementation of Priority Replay Buffer using Sum-Tree

The python code for the replay buffer that uses sum-tree class defined in the previous sub-section is provided in the code Listing 5.25. The hyper-parameter `PER_e` ensures that none of the experiences are assigned zero priority as defined in equation (5.7). The hyper-parameter `PER_a` is used in the stochastic prioritization probability equation (5.8) to balance between selecting high priority experiences and exploring other experiences in the replay buffer. The hyper-parameter `PER_b` is used in equation (5.9) to control how much importance sampling affects the learning process. The class method `STBuffer.add()` updates the priority of tree as the samples are added one by one. The class method `STBuffer.sample()` selects a given number of samples from the replay buffer. The class method `STBuffer.batch_update()` updates the priority of the samples in the replay buffer based on the TD error.

```
    class STBuffer(object):
    # stored as ( state, action, reward, next_state ) in SumTree
    PER_e = 0.01  # avoid some experiences to have 0 probability
    PER_a = 0.6  #  control randomness in stochastic prioritization
    PER_b = 0.4  # importance-sampling, from initial value increasing to 1
    PER_b_increment_per_sampling = 0.001
    absolute_error_upper = 1.  # clipped abs error

    def __init__(self, capacity):
    # Making the tree
    self.tree = SumTree(capacity)

    def add(self, experience):
    # Find the max priority of leaf nodes
    max_priority = np.max(self.tree.tree[-self.tree.capacity:])
    # If the max priority = 0 we can't put priority = 0
    # since this experience will never have a chance to be selected
    # So we use a minimum priority
    if max_priority == 0:
    max_priority = self.absolute_error_upper
    # set the priority for new experience
    self.tree.add(max_priority, experience)

    def sample(self, n):
    # Create a minibatch array that will contains the minibatch
    minibatch = []
    b_idx = np.empty((n,), dtype=np.int32)
    # array to store sample priorities
    priorities = np.empty((n,), dtype=np.float32)
    # Calculate the priority segment
    # we divide the Range[0, ptotal] into n ranges
```

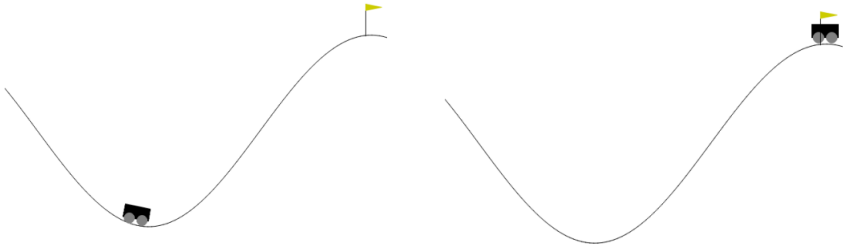
```
        priority_segment = self.tree.total_priority / n  # priority segment
        for i in range(n):
        # A value is uniformly sample from each range
        a, b = priority_segment * i, priority_segment * (i + 1)
        value = np.random.uniform(a, b)
        # Experience that correspond to each value is retrieved
        index, priority, data = self.tree.get_leaf(value)
        b_idx[i]= index
        priorities[i] = priority   # experimental
        minibatch.append([data[0],data[1],data[2],data[3],data[4]])
    return b_idx, minibatch,

    def batch_update(self, tree_idx, abs_errors):
    abs_errors += self.PER_e  # convert to abs and avoid 0
    clipped_errors = np.minimum(abs_errors, self.absolute_error_upper)
    # stochastic prioritization
    ps = np.power(clipped_errors, self.PER_a)  # values between 0 and 1
    # convert priorities into probabilities
    prob = ps / np.sum(ps) # experimental
    # importance sampling weights: iw = [1 / ( N * P)]^b
    is_wts = np.power(len(prob) * prob, -self.PER_b)
    for ti, p, iw in zip(tree_idx, ps, is_wts):
    new_p = p * iw
    self.tree.update(ti, new_p)
    # gradually increase PER_b for more focus on high-error experience
    self.PER_b = min(1.0, self.PER_b + \
    self.PER_b_increment_per_sampling)

    def __len__(self):
    return len(self.tree)

    def __getitem__(self, index):
    return self.tree[index]
```

Listing 5.25: Python implementation of Priority Replay Buffer

5.5.4. Python code for DQN Agent with PER

The Python code for implementing a DQN Agent with PER buffer is provide in the code Listing 5.26. The class `DQNPERAgent` inherits most of its properties from the `DQNAgent` class. Please note that here `STBuffer` is used to create the memory replay buffer called `self.memory` instead of the simple instead of the simple `ReplayBuffer`. This child class overrides the `experience_replay()` function of the parent class.

```
class DQNPERAgent(DQNAgent):
def __init__(self, obs_shape: tuple, n_actions: int,
buffer_size=2000, batch_size=24,
ddqn_flag=True, model=None):
super().__init__(obs_shape, n_actions, buffer_size,
# uses a sumtree Buffer
self.memory = STBuffer(capacity=buffer_size)


def experience_replay(self):
if len(self.memory) < self.train_start:
return
batch_size = min(self.batch_size, len(self.memory))
tree_idx, mini_batch = self.memory.sample(self.batch_size)
states = np.zeros((self.batch_size, *self.obs_shape))
next_states = np.zeros((self.batch_size, *self.obs_shape))
actions = np.zeros((self.batch_size, 1))
rewards = np.zeros((self.batch_size, 1))
dones = np.zeros((self.batch_size, 1))
for i in range(len(mini_batch)):
states[i] = mini_batch[i][0]
actions[i] = mini_batch[i][1]
rewards[i] = mini_batch[i][2]
next_states[i] = mini_batch[i][3]
dones[i]  = mini_batch[i][4]
q_values_cs = self.model.predict(states, verbose=0)
q_values_cs_old = np.array(q_values_cs).copy() # deep copy
max_q_values_ns = self.get_target_q_value(next_states)
# Q-learning updates
for i in range(len(q_values_cs)):
action = actions[i].astype(int)[0] # check
done = dones[i].astype(bool)[0] # check
reward = rewards[i][0] # check
if done:
q_values_cs[i][action] = reward
else:
q_values_cs[i][action] = reward + \
self.gamma * max_q_values_ns[i]
# update experience priorities
indices = np.arange(self.batch_size, dtype=np.int32)
actions = actions[:,0].astype(int)
absolute_errors = np.abs(q_values_cs_old[indices, actions] - \
q_values_cs[indices, actions])
```



(a) Initial random state (b) Desired final state

Figure 5.6.: Gym's Mountain Car Problem. The goal is to reach the top of the hill as soon as possible.

```
        # update sample priorities
        self.memory.batch_update(tree_idx, absolute_errors)
        # train the Q network
        self.model.fit(np.array(states),
        np.array(q_values_cs),
        batch_size = batch_size,
        epochs = 1,
        verbose = 0)
        # decay epsilon over time
        self.update_epsilon()
```

Listing 5.26: Python Code for implementing DQN with PER

5.5.5. Solving MountainCar Problem using DQN with PER

The Mountain Car Problem involves a car moving in 1D environment. It has two states, namely, position (x) and velocity ($\dot{x}$) of the car. The car can take three discrete actions - accelerate left (0), don't accelerate (0) and accelerate right (2). The goal is to reach the flag placed on the top of the right hill ($x \geq 0.5$)) as quickly as possible. The agent receives a reward of -1 every time it fails to reach the goal. Each episode consists of 200 steps leading to a total reward of -200 in the beginning. The initial and desired final state of the environment is shown in Figure 5.6. The problem is considered solved if the car can reach the flag pole in less than 200 steps.

Training the agent

The training function used for solving this problem is provided in the code Listing 5.27. Reward engineering is an important step towards solving this problem. In this case, the agent is given an additional reward of +200 whenever it reaches to the top and reward proportional to its change in position and acceleration applied for other positions. This

encourages the agent to learn faster. The problem is considered solved if the car reaches the flag pole indicated by `done=True` in less than 200 steps.

```
    def train(env, agent, max_episodes=300,
    train_freq=1, copy_freq=10, filename=None, wtfile_prefix=None):
    if filename is not None:
    file = open(filename, 'w')
    if wtfile_prefix is not None:
    wt_filename = wtfile_prefix + '_best_model.weights.h5'
    else:
    wt_filename = 'best_model.weights.h5'

    # choose between soft & hard target update
    tau = 0.1 if copy_freq < 10 else 1.0
    max_steps = 200
    car_positions = []
    scores, avg_scores = [], []
    global_step_cnt = 0
    for e in range(max_episodes):
    # make observation
    state = env.reset()[0]
    state = np.expand_dims(state, axis=0)
    done = False
    ep_reward = 0
    t = 0
    max_pos = -99.0
    while not done:
    global_step_cnt += 1
    # take action using epsilon-greedy policy
    action = agent.get_action(state)
    # transition to next state
    # and collect reward from the environment
    next_state, reward, done, _, _ = env.step(action)
    next_state = np.expand_dims(next_state, axis=0) # (-1, 4)
    # reward engineering - important step
    if next_state[0][0] >= 0.5:
    reward += 200
    else:
    reward = 5*abs(next_state[0][0] - state[0][0])\
    + 3*abs(state[0][1])

    # track maximum car position
    if next_state[0][0] > max_pos:
    max_pos = next_state[0][0]
```

```
            # store experience in replay buffer
            agent.store_experience(state, action, reward, next_state, done)
            state = next_state
            ep_reward += reward
            t += 1

            # train
            if global_step_cnt % train_freq == 0:
                agent.experience_replay()

            # update target model
            if global_step_cnt % copy_freq == 0:
                agent.update_target_model(tau=tau)
            if done and t < max_steps:
                print('\nSuccessfully solved the problem in {} episodes. \
                max_pos:{:.2f}, steps: {}\n'.format(e, max_pos, t))
                agent.save_model(wt_filename)

            if t >= max_steps:
                break
            # episode ends here
            car_positions.append(state[0][0])
            scores.append(ep_reward)
            avg_scores.append(np.mean(scores))
            if filename is not None:
                file.write(f'{e}\t{ep_reward}\t{np.mean(scores)}\
                \t{max_pos}\t{t}\n' )
                file.flush()
                os.fsync(file.fileno()) # write to the file immediately
                #print on console
                print(f'\re:{e}, ep_reward: {ep_reward:.2f}, avg_ep_reward: \
                {np.mean(scores):.2f}, ep_steps: {t}, max_pos: {max_pos:.2f}', end="")
            sys.stdout.flush()
            print('End of training')
            file.close()
```

Listing 5.27: Training function used for solving the Mountain Car Problem.

Main Code for creating and training agent

The main Python code for creating and training an agent is provided in the code Listing 5.28. An instance of gym environment for Mountain Car problem is created. A sequential

deep network is created and passed to the `DQNPERAgent` and then the `train()` function is called. The corresponding console output is also shown in this list. The model training performance is shown in Figure 5.7. It is seen that the average episodic reward increases overtime. The problem is solved for episodes where the episodic reward reaches 200, car position reaches 0.5 in less than 200 steps.

```
    import matplotlib.pyplot as plt
    import gymnasium as gym
    import keras

    # create a gym environment
    env = gym.make('MountainCar-v0', render_mode='rgb_array')
    obs_shape = env.observation_space.shape
    action_shape = env.action_space.shape
    n_actions = env.action_space.n
    print('Observation shape: ', obs_shape)
    print('Action shape: ', action_shape)
    print('Action size: ', n_actions)
    print('Max episodic steps: ', env.spec.max_episode_steps)

    # Create a model
    model = keras.Sequential([
        keras.layers.Dense(30, input_shape=obs_shape, activation='relu'),
        keras.layers.Dense(60, activation='relu'),
        keras.layers.Dense(n_actions, activation='linear')
    ])
    model.compile(loss='mse',  \
    optimizer=keras.optimizers.Adam(learning_rate=0.001))


    # create a DQN PER Agent
    agent = DQNPERAgent(obs_shape, n_actions,
    buffer_size=20000,
    batch_size=64,
    model=model)

    # train the agent
    train(env, agent, max_episodes=200, copy_freq=200, \
    filename='mc_dqn_per.txt')
```

```
    e:10, ep_reward: 25.87, avg_ep_reward: 16.28, ep_steps: 200, max_pos: -0.08
```

```
Successfully solved the problem in 11 episodes. max_pos:0.53, steps: 138

e:11, ep_reward: 223.93, avg_ep_reward: 33.59, ep_steps: 138, max_pos: 0.53
Successfully solved the problem in 12 episodes. max_pos:0.51, steps: 152

e:13, ep_reward: 15.08, avg_ep_reward: 45.95, ep_steps: 200, max_pos: -0.05
Successfully solved the problem in 14 episodes. max_pos:0.50, steps: 174

e:14, ep_reward: 224.50, avg_ep_reward: 57.85, ep_steps: 174, max_pos: 0.50
Successfully solved the problem in 15 episodes. max_pos:0.52, steps: 163

e:16, ep_reward: 31.26, avg_ep_reward: 66.23, ep_steps: 200, max_pos: 0.252
Successfully solved the problem in 17 episodes. max_pos:0.54, steps: 159

e:17, ep_reward: 229.82, avg_ep_reward: 75.32, ep_steps: 159, max_pos: 0.54
Successfully solved the problem in 18 episodes. max_pos:0.51, steps: 136

e:18, ep_reward: 219.73, avg_ep_reward: 82.92, ep_steps: 136, max_pos: 0.51
Successfully solved the problem in 19 episodes. max_pos:0.51, steps: 145

e:19, ep_reward: 219.61, avg_ep_reward: 89.75, ep_steps: 145, max_pos: 0.51
Successfully solved the problem in 20 episodes. max_pos:0.51, steps: 150
```

Listing 5.28: Solving the Mountain Car Problem with DQN and PER

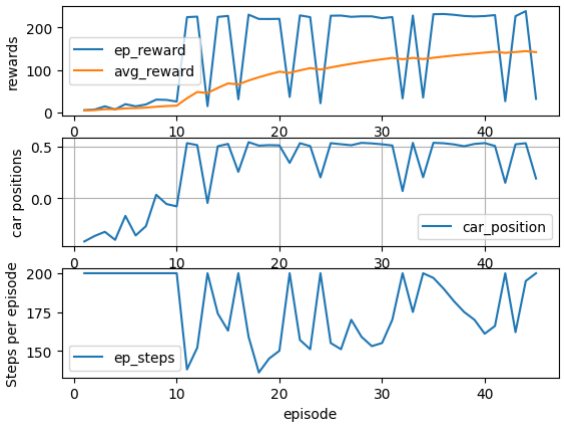


Figure 5.7.: Training performance of the DQN PER Agent used for solving the Mountain Car Problem. The problem is considered solved if the car position reaches 0.5, episodic reward reaches 200 in less than 200 steps.

5.6. Solving Atari Games using DQN

The Atari 2600, also known as the Atari VCS, was a hugely popular game console released in 1977. It helped usher in the home video game revolution[1]. Atari environments are simulated via the Arcade Learning Environment (ALE) through the Stella emulator. You are required to install Atari ROMs separately to make atari environments using gymnasium. Please install the following packages on your system:

```
pip install gymnasium[atari]
pip install gymnasium[accept-rom-license]
```

In atari environments, the observation is available in the form of color or grayscale images. The code listing 5.29 shows how one can make a gym environment for `PacMan` atari game and visualize its observation. The corresponding observation in shown in Figure 5.8.

```
import gymnasium as gym
import matplotlib.pyplot as plt
import numpy as np
# create an atari environment
env = gym.make('ALE/MsPacman-v5', obs_type="grayscale",
                render_mode='rgb_array')
obs_shape = env.observation_space.shape + (1,)
print('shape of action space: ', env.action_space.n)
print('shape of observation space: ', env.observation_space.shape)
x = env.reset()[0]  # initialize environment and make observation
print('shape of x: ', np.shape(x))
plt.imshow(x)   # visualize observation
plt.axis('off')
print('obs_shape: ', obs_shape)
print('Max Environment Steps:', env.spec.max_episode_steps)
print('Observation space low: ', env.observation_space.low[0][0])
print('Observation space high:', env.observation_space.high[0][0])
```

```
A.L.E: Arcade Learning Environment (version 0.8.1+53f58b7)
[Powered by Stella]
shape of action space:  9
shape of observation space:  (210, 160)
shape of x:  (210, 160)
```

[1] https://en.wikipedia.org/wiki/Atari_2600

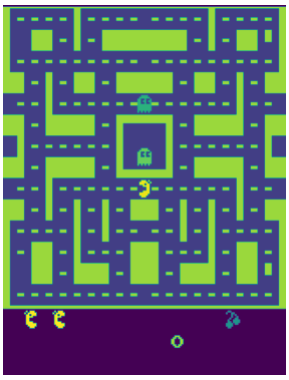


Figure 5.8.: A PacMan Atari Environment Observation

```
obs_shape:  (210, 160, 1)
Max Environment Steps: None
Observation space low:  0
Observation space high: 255
```

Listing 5.29: Python code for creating an atari environment and visualizing state observation.

5.6.1. Image Stacking Wrapper

Observation images are stacked along the depth channel to incorporate temporal information into the training process. The code listing 5.30 provides python code for creating a wrapper class to stack frames in Gym environments.

```
from collections import deque
import gymnasium as gym
from gym.spaces import Box

class FrameStack(gym.Wrapper):
    """
    Wrapper that stacks observations from the environment
    into a single observation. This wrapper keeps a rolling buffer
    of the most recent frames and stacks them
    together as a new observation.
    """
    def __init__(self, env, num_stacked_frames):
        """
```

```
        Args:
          env: The environment to wrap.
          num_stacked_frames: The number of frames to stack.
        """
        super(FrameStack, self).__init__(env)
        self.num_stacked_frames = num_stacked_frames
        self.frames = deque([], maxlen=num_stacked_frames)
        obs_shape = env.observation_space.shape
        if len(obs_shape) == 2: # convert (H, W) to (H, W, D)
            obs_shape = obs_shape + (1,)
        # Modify the observation space to accommodate stacked frames
        self.observation_space = Box(
            low=0, high=255,
            shape=(obs_shape[0], obs_shape[1],
                   obs_shape[2] * self.num_stacked_frames),
            dtype=self.env.observation_space.dtype
        )

    def reset(self):
        """
        Resets the environment and fills the frame buffer
        with initial observations.
        """
        observation = self.env.reset()[0]
        if len(np.shape(observation)) == 2: # convert (H, W) to (H, W, D)
            observation = np.expand_dims(observation, axis=2)
        for _ in range(self.num_stacked_frames):
            self.frames.append(observation)
        return self._stack_frames()

    def step(self, action):
        """ Steps through the environment and stacks
            the new observation with previous ones.
        """
        observation, reward, done, info, _ = self.env.step(action)
        if len(np.shape(observation)) == 2: # convert (H, W) to (H, W, D)
            observation = np.expand_dims(observation, axis=2)
        self.frames.append(observation)
        return self._stack_frames(), reward, done, info

    def _stack_frames(self):
        """ Stacks frames from the buffer into a single observation.
        """
```

```
        return np.concatenate(self.frames, axis=2)
```

Listing 5.30: A wrapper class to stack frames in Gym environment

5.6.2. DQN Agent for Atari Environments

The code listing 5.31 provides the `DQNAtariAgent` class definition for applying DQN algorithm to atari problems. It inherits attributes from both `DQNAgent` and `DQNPERAgent` classes defined earlier. The selection between these two agents is facilitated by the class attribute `self.per_flag`. The class provides `preprocess()` method to resize input frames. It also includes a `train()` function to allow agent training on a given environment.

```
import cv2
import sys

class DQNAtariAgent(DQNPERAgent):
    def __init__(self, obs_shape: tuple, n_actions: int,
                 buffer_size=2000, batch_size=24,
                 ddqn_flag=True, model=None, per_flag=True):
        self.per_flag = per_flag

        if self.per_flag:
            super().__init__(obs_shape, n_actions, buffer_size,
                             batch_size, ddqn_flag, model)
        else:
            DQNAgent.__init__(self, obs_shape, n_actions, buffer_size,
                              batch_size, ddqn_flag, model)

    def experience_replay(self):
        if self.per_flag:
            super().experience_replay()
        else:
            DQNAgent.experience_replay(self)


    def preprocess(self, observation, x_crop=(1, 172), y_crop=None):
        assert len(self.obs_shape) == 3, \
               "Observation must have 3 dimension (H, W, C)"
        output_shape = self.obs_shape[:-1] # all but last (H, W)
        # crop image
        if x_crop is not None and y_crop is not None:
            xlow, xhigh = x_crop
            ylow, yhigh = y_crop
```

```
                observation = observation[xlow:xhigh, ylow:yhigh]
            elif x_crop is not None and y_crop is None:
                xlow, xhigh = x_crop
                observation = observation[xlow:xhigh, :]
            elif x_crop is None and y_crop is not None:
                ylow, yhigh = y_crop
                observation = observation[:, ylow:yhigh]
            else:
                observation = observation

            # resize image
            observation = cv2.resize(observation, output_shape)

            # normalize image
            observation = observation / 255.  # normalize between 0 & 1
            return observation

        def train(self, env, max_episodes=300,
                  train_freq=1, copy_freq=1, filename=None, wtfile_prefix=None):

            if filename is not None:
                file = open(filename, 'w')

            if wtfile_prefix is not None:
                wt_filename = wtfile_prefix + '_best_model.weights.h5'
            else:
                wt_filename = 'best_model.weights.h5'

            tau = 0.01 if copy_freq < 10 else 1.0

            best_score, global_step_cnt = 0, 0
            scores, avg_scores, avg100_scores = [], [], []
            global_step_cnt = 0
            for e in range(max_episodes):
                state = env.reset() # with framestack wrapper
                #state = env.reset()[0] # without framestack wrapper
                state = self.preprocess(state)
                state = np.expand_dims(state, axis=0)
                done = False
                ep_reward = 0
                while not done:
                    global_step_cnt += 1
                    # take action
                    action = self.get_action(state)
```

```
                # collect reward
                next_state, reward, done, _ = env.step(action)
                next_state = self.preprocess(next_state) # (H, W, C)
                next_state = np.expand_dims(next_state, axis=0) # (B, H, W, C)
                # store experiences in eplay buffer
                self.store_experience(state, action, reward,
                                      next_state, done)
                state = next_state
                ep_reward += reward
                # train
                if global_step_cnt % train_freq == 0:
                    self.experience_replay()

                # update target model
                if global_step_cnt % copy_freq == 0:
                    self.update_target_model(tau=tau)
            # end of while-loop
            if ep_reward > best_score:
                self.save_model(wt_filename)
                best_score = ep_reward
            scores.append(ep_reward)
            avg_scores.append(np.mean(scores))
            avg100_scores.append(np.mean(scores[-100:]))
            if filename is not None:
                file.write(f'{e}\t{ep_reward}\t{np.mean(scores)}\
                             \t{np.mean(scores[-100:])}\n')
                file.flush()
                os.fsync(file.fileno())
            print(f'\re:{e}, ep_reward: {ep_reward}, \
                        avg_ep_reward: {np.mean(scores):.2f}', end="")
            sys.stdout.flush()
        # end of for loop
        print('\nEnd of training')
        if filename is not None:
            file.close()
```

Listing 5.31: Python class for applying DQN algorithm to Atari environments. It inherits attributes from DQN and DQNPERAgent

5.6.3. Training agent on Atari PacMan Environment

The main code for creating and training a DQN agent for PacMan environment is provided in the code listing 5.32. The wrapper class `FrameStack` is used to stack observation

frames along the depth channel. A sequential model created externally is passed to the `DQNAtariAgent` for creating the Q network. The performance of DQN and DQN+PER algorithm for this environment is shown in Figure 5.9. Both of these algorithms implement double DQN algorithm by default. It is seen in this figure that PER provides some improvement in training performance over the standard DQN algorithm.

```
# create an instance of gym environment
import gymnasium as gym
env = gym.make('ALE/MsPacman-v5', obs_type="grayscale",
                                  render_mode='rgb_array')

# Stack the frames using Wrapper
env = FrameStack(env, num_stacked_frames=4)
print('observation shape: ', env.observation_space.shape)

n_actions = env.action_space.n
print('Action space dimension: ', n_actions)

# All images will be resized to this size
obs_shape = (84, 84, 4)

# create a sequential model for Q Network
model = keras.Sequential([
    keras.layers.Conv2D(32, kernel_size=8, strides=4, padding='same',
                        activation='relu', kernel_initializer='he_uniform',
                        input_shape=obs_shape),
    keras.layers.MaxPooling2D(pool_size=(2,2)),
    keras.layers.Conv2D(64, kernel_size=2, strides=1, padding='same',
                        activation='relu', kernel_initializer='he_uniform'),
    keras.layers.MaxPooling2D(pool_size=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu',
                       kernel_initializer='he_uniform'),
    keras.layers.Dense(n_actions, activation='linear')
])
model.compile(loss='mse', optimizer="adam")

# Create DQN PER Agent
agent = DQNAtariAgent(obs_shape, n_actions,
                      buffer_size=60000,
                      batch_size=64,
                      model=model, per_flag=True)
# Train the agent
agent.train(env, max_episodes=400, train_freq=5,
```

```
    copy_freq=50, filename='pacman_dqn_per.txt')
```

Listing 5.32: Python code for creating and training a DQN agent for Atari PacMan environment.

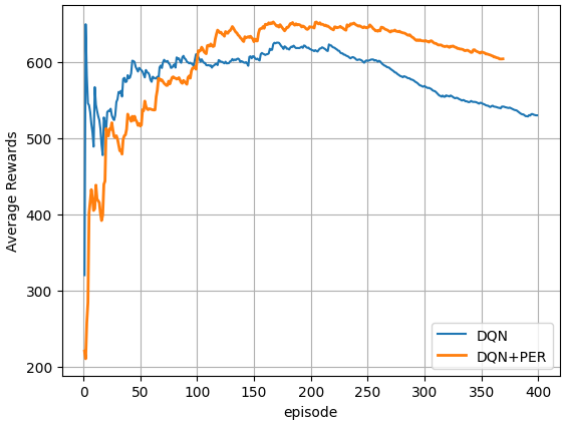


Figure 5.9.: Training performance of DQN and DQN+PER algorithm for PacMan Environment

5.7. Summary

In this chapter, we saw how Q-learning algorithm could be extended to solve complex problems by using a deep network to approximate the Q function. This network can take any arbitrary observation input thereby greatly expanding the capabilities of DQN. Then we talked about several improvements to overcome the limitation of the standard DQN architecture. Some of these improvements include using an additional target Q network to stabilize the learning process and using experience replay to improve the training performance. Finally, we discussed priority experience replay that samples experiences based on their priorities. Finally, we demonstrate the performance of DQN architectures in solving several problems such as `CartPole`, `MountainCar` and Atari problems.

6. Policy Gradient Methods

6.1. Introduction

In the previous chapters, we saw how we can derive policy from the estimated Q-value function. Once the Q function is known, the optimal policy is derived greedily by using the following equation:

$$a^* = \arg\max_a Q(s, a) \tag{6.1}$$

However, this approach can only be applied to problems with discrete action spaces. In this chapter, we will look into policy gradient methods that can learn the optimal policy without estimating the Q function and hence, can be applied to solve problems having continuous action spaces.

6.2. Policy Gradient

We first define the policy function $\pi(a|s)$ as the probability of taking an action a in a given state s. Let us parameterize the policy function via a parameter θ as $\pi(a|s;\theta)$ which will allow us to estimate the optimal policy for a given state. In practice, we use a neural network to approximate the policy function that takes state as input and produces probability of each action as the output. The parameters of this network is now updated in such a way that the actions with higher expected rewards will have a higher probability for a given input state. Hence, the objective function to be maximized in a policy gradient method is the expected cumulative future reward given by

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1}] \tag{6.2}$$

where r_{t+1} is the reward received by performing action a_t in the state s_t and is given by the function

$$r_{t+1} = R(s_t, a_t) \tag{6.3}$$

The parameters of the policy function is now optimized by using *gradient ascent* with the partial derivative of the objective function with respect to the policy parameter θ as shown below:

$$\theta = \theta + \alpha \frac{\partial}{\partial \theta} J(\theta) = \theta + \alpha \nabla_\theta J(\theta) \tag{6.4}$$

where α is the learning rate that controls the amount of update at each iterative step.

Now, we will derive the expression for the gradient term $\nabla_\theta J(\theta)$ which is required to update the policy parameters during the training process.

6.2.1. Computing $\nabla_\theta J(\theta)$

The expected value or mean value of a random variable x is computed by the summation of the product of every value of x and its probability given by:

$$\mathbb{E}[f(x)] = \sum_x P(x)f(x) \tag{6.5}$$

Therefore the objective function in (6.2) can be expanded as given below:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi(\theta)}[\sum_{t=0}^{T-1} r_{t+1}|\pi_\theta] = \sum_\tau \sum_{t=0}^{T-1} P(s_t, a_t|\tau)r_{t+1} \tag{6.6}$$

where $P(s_t, a_t|\tau)$ is the probability of occurrence of (s_t, a_t) given the trajectory τ.

Differentiating both sides of the above equation with respect to policy parameter θ and using the formula $\frac{d}{dx}\log f(x) = \frac{f'(x)}{f(x)}$, we get

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_\tau \sum_{t=0}^{T-1} \nabla_\theta P(s_t, a_t|\tau)r_{t+1} = \sum_\tau \sum_{t=0}^{T-1} P(s_t, a_t|\tau)\frac{\nabla_\theta P(s_t, a_t|\tau)}{P(s_t, a_t|\tau)}r_{t+1} \\
&= \sum_\tau \sum_{t=0}^{T-1} P(s_t, a_t|\tau)\nabla_\theta \log P(s_t, a_t|\tau)r_{t+1} \\
&= \mathbb{E}_{\pi(\theta)\sim\tau}[\sum_{t=0}^{T-1} \nabla_\theta \log P(s_t, a_t|\tau)r_{t+1}]
\end{aligned} \tag{6.7}$$

During the training, random samples of episodes are used instead of computing the expectation. Therefore, we can make use of the following expression:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1} \nabla_\theta \log P(s_t, a_t|\tau)r_{t+1} \tag{6.8}$$

where N represents the number of samples used for each training step.

Probability of being at any state depends on all previous states and actions in a trajectory under a given policy π_θ. Therefore, by definition, we have

$$\begin{aligned}
P(s_t, a_t|\tau) &= P(s_0, a_0, s_1, a_1, \ldots, s_{t-1}, a_{t-1}, s_t, a_t|\pi_\theta) \\
&= P(s_0)\pi_\theta(a_0|s_0)P(s_1|s_0, a_0)\pi_\theta(a_1|s_1)P(s_2|s_1, a_1)\pi_\theta(a_2|s_2) \\
&\quad \ldots P(s_{t-1}|s_{t-2}, a_{t-2})\pi_\theta(a_{t-1}|s_{t-1})P(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_t) \\
&= P(s_0)\pi_\theta(a_0|s_0)\Pi_{i=1}^{t}P(s_i|s_{i-1}, a_{i-1})\pi_\theta(a_i|s_i)
\end{aligned} \tag{6.9}$$

Taking log on both sides, we get

$$\begin{aligned}
\log P(s_t, a_t|\tau) = &\log P(s_0) + \log \pi_\theta(a_0|s_0) + \log P(s_1|s_0, a_0) + \log \pi_\theta(a_1|s_1) \\
&+ \log P(s_2|s_1, a_1) + \log \pi_\theta(a_2|s_2) + \ldots + \log P(s_{t-1}|s_{t-2}, a_{t-2}) \\
&+ \log \pi_\theta(a_{t-1}|s_{t-1}) + \log P(s_t|s_{t-1}, a_{t-1}) + \log \pi_\theta(a_t|s_t)
\end{aligned} \tag{6.10}$$

Differentiating both sides with respect to policy parameter θ, we get

$$\begin{aligned}\nabla_\theta P(s_t, a_t|\tau) &= 0 + \nabla_\theta \log \pi_\theta(a_0|s_0) + 0 + \nabla_\theta \log \pi_\theta(a_1|s_1) + 0 + \nabla_\theta \log \pi_\theta(a_2|s_2) \\ &\quad + \ldots + 0 + \nabla_\theta \log \pi_\theta(a_{t-1}|s_{t-1}) + 0 + \nabla_\theta \log \pi_\theta(a_t|s_t) \\ &= \sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \end{aligned} \tag{6.11}$$

In the above equation, we used the fact that $P(s_t|s_{t-1}, a_{t-1})$ is not dependent on the policy parameter θ and is solely dependent on the environment.

Now substituting equation (6.11) into (6.8), we get

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} r_{t+1} \nabla_\theta P(s_t, a_t|\tau) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} r_{t+1} \left(\sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right) \tag{6.12}$$

The inner term can be expanded as shown below:

$$\begin{aligned}\sum_{t=0}^{T-1} r_{t+1} \left(\sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right) &= r_1 \left(\sum_{t'=0}^{0} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right) + r_2 \left(\sum_{t'=0}^{1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right) + \\ &\quad r_3 \left(\sum_{t'=0}^{2} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right) + \ldots + r_T \left(\sum_{t'=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right) \\ &= r_1 \nabla_\theta \log \pi_\theta(a_0|s_0) + r_2(\nabla_\theta \log \pi_\theta(a_0|s_0) + \nabla_\theta \log \pi_\theta(a_1|s_1)) \\ &\quad + \ldots + r_T(\nabla_\theta \log \pi_\theta(a_0|s_0) + \nabla_\theta \log \pi_\theta(a_1|s_1) + \ldots + \nabla_\theta \log \pi_\theta(a_{T-1}|s_{T-1})) \\ &= \nabla_\theta \log \pi_\theta(a_0|s_0)(r_1 + r_2 + \ldots r_T) + \nabla_\theta \log \pi_\theta(a_1|s_1)(r_2 + r_3 + \ldots r_T) \\ &\quad + \nabla_\theta \log \pi_\theta(a_2|s_2)(r_3 + r_4 + \ldots + r_T) + \ldots + \nabla_\theta \log \pi_\theta(a_{T-1}|s_{T-1}) r_T \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t+1}^{T} r_{t'} \right) \end{aligned} \tag{6.13}$$

Therefore, the policy gradient term can be written as

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t+1}^{T} r_{t'} \right) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \tag{6.14}$$

where $G_t = \sum_{t'=t+1}^{T}$ is the sum of future rewards. By incorporating discounting factory $\gamma \in [0, 1]$ into our objective function, we can rewrite (6.2) as follows:

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} \gamma^t r_{t+1}|\pi_\theta] \tag{6.15}$$

This will lead to the policy gradient as follows:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t+1}^{T} \gamma^{t'-1} r_{t'} \right) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \tag{6.16}$$

where

$$G_t = \sum_{t'=t+1}^{T} \gamma^{t'-1} r_{t'} \tag{6.17}$$

is the sum of discounted future rewards also known as the return. From chapter 2, we know that the expected value of cumulative future reward is also called the Q function $Q(s_t, a_t)$ (see equation (2.6)). Thus, we can rewrite the policy gradient equation (6.16) as:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) Q(s_{i,t}, a_{i,t}) \tag{6.18}$$

We can always subtract a term to the optimization problem as long as the term is not related to θ. So instead of using the total reward, we can subtract it with $V(s)$. This will lead to the following expression for policy gradient in terms of the advantage function:

$$\begin{aligned} \nabla_\theta J(\theta) &\approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) [Q(s_{i,t}, a_{i,t}) - V(s)] \\ &\approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) A(s_{i,t}, a_{i,t}) \end{aligned} \tag{6.19}$$

We will now use the above equations to develop a policy gradient algorithm to learn the optimal policy.

6.3. Monte-Carlo Policy Gradient Algorithm

Monte-Carlo Policy Gradient algorithm, also known as REINFORCE, uses return obtained from each episode of samples to update the policy parameter θ. In other words, it updates the policy parameter after every episode. The policy network π_θ takes states as input and outputs probabilities for all actions. The policy gradient use gradient ascent to adjust the weights. This will encourage the policy network to take better actions. The steps involved are as follows:

1. Initialize the policy parameter θ at random.
2. Generate a trajectory using the policy π_θ:

$$\{\tau : s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots, S_{T-1}\}$$

3. for $t = 0, 1, 2, \cdots T-1$:
   a) Estimate the return G_t by using (6.17).
   b) Update the policy parameter:

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{6.20}$$

6.3.1. Python Code for REINFORCE Algorithm

The implementation involves two steps - first creating a policy network and then implement the Monte-Carlo Policy Gradient algorithm as described above. The code for creating a policy network is provided in Listing 6.33. It takes state as input and outputs probabilities of all actions. The code for implementing REINFORCE Algorithm is provided in Listing 6.34. The method `calculate_return()` computes the cumulative discounted rewards for each episode. The `choose_action` method samples an action based on the action probabilities obtained using the policy network. The `train()` method updates the policy parameters using gradient ascent by using equation (6.20). The buffers for storing transitions are cleared after each training step. The code for solving a Gym problem environment with this agent is provided in the code listing 6.35. It involves generating action using `agent.choose_action()` method and storing transitions in a buffer using `agent.store_transitions()` method. Finally, the agent is trained after each episode using `agent.train()` method.

```
import tensorflow as tf
class PolicyNetwork():
    def __init__(self, obs_shape, n_actions, lr=0.0001,
                 fc1_dim=256, fc2_dim=256):
        self.fc1_dim = fc1_dim
        self.fc2_dim = fc2_dim
        self.n_actions = n_actions
        self.obs_shape = obs_shape
        self.lr = lr
        self.model = self._build_model()
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=self.lr)

    def _build_model(self):
        inputs = tf.keras.layers.Input(shape=self.obs_shape)
        fc1 = tf.keras.layers.Dense(self.fc1_dim,
                                    activation='relu')(inputs)
        fc2 = tf.keras.layers.Dense(self.fc2_dim,
                                    activation='relu')(fc1)
        outputs = tf.keras.layers.Dense(self.n_actions,
                                        activation='softmax')(fc2)
        model = tf.keras.models.Model(inputs, outputs,
                                      name='policy_network')
        model.summary()
        return model

    def __call__(self, state):
        pi = self.model(state)
        return pi
```

Listing 6.33: Python code for creating Policy Network

```
import tensorflow as tf
import tensorflow_probability as tfp
import numpy as np
class REINFORCEAgent:
    def __init__(self, obs_shape, n_actions, alpha=0.0005, gamma=0.99):
        self.gamma = gamma
        self.lr = alpha
        self.n_actions = n_actions
        self.states = []
        self.actions = []
        self.rewards = []
        self.obs_shape = obs_shape
        # create policy network
        self.policy = PolicyNetwork(obs_shape, n_actions, lr=self.alpha)

    def choose_action(self, obs):
        state = tf.convert_to_tensor(obs, dtype=tf.float32)
        probs = self.policy(state)
        action_probs = tfp.distributions.Categorical(probs=probs)
        action = action_probs.sample()
        return action.numpy()[0]

    def store_transitions(self, state, action, reward):
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)

    def calculate_return(self, rewards):
        G = np.zeros_like(rewards)
        for t in range(len(rewards)):
            G_sum = 0
            for k in range(t, len(rewards)):
                G_sum += rewards[k] * self.gamma
            G[t] = G_sum
        return G

    def train(self):
        actions = tf.convert_to_tensor(self.actions, dtype=tf.float32)
        rewards = np.array(self.rewards)
        # compute returns
        G = self.calculate_return(rewards)
        # optimize with gradient ascent
```

```
        with tf.GradientTape() as tape:
            loss = 0
            for idx, (g, state) in enumerate(zip(G, self.states)):
                state = tf.convert_to_tensor(state, dtype=tf.float32)
                probs = self.policy(state)
                action_probs = tfp.distributions.Categorical(probs=probs)
                log_prob = action_probs.log_prob(actions[idx])
                loss += -g * tf.squeeze(log_prob)
        trainable_params = self.policy.model.trainable_variables
        gradient = tape.gradient(loss,  trainable_params)
        self.policy.optimizer.apply_gradients(zip(gradient, \
                                                  trainable_params))
        # empty the buffer
        self.states = []
        self.actions = []
        self.rewards = []
```

Listing 6.34: Python Code for REINFORCE Algorithm.

```
import os
import gymnasium as gym
def train_agent_for_env(env, agent, max_episodes=1000):
    scores, avgscores, avg100scores = [], [], []
    for e in range(max_episodes):
        done = False
        score = 0
        state = env.reset()[0]
        state = np.expand_dims(state, axis=0)
        while not done:
            action = agent.choose_action(state)
            next_state, reward, done, _, _ = env.step(action)
            next_state = np.expand_dims(next_state, axis=0)
            agent.store_transitions(state, action, reward)
            score += reward
            state = next_state
        # end of while loop
        # train the agent at the end of each episode
        agent.train()
        scores.append(score)
        avgscores.append(np.mean(scores))
        avg100scores.append(np.mean(scores[-100:]))
        if e % 100== 0:
            print('episode:{}, score: {:.2f}, avgscore: {:.2f}, \
```

```
                avg100score: {:.2f}'.format(e, score, \
            np.mean(scores), np.mean(scores[-100:])))
        # end of for-loop
    file.close()
```

Listing 6.35: Code for training a monte carlo policy gradient agent on a given Gym environment.

6.3.2. Solving CartPole Problem

The performance of the REINFORCE algorithm in solving the `CartPole-v0` is shown in Figure 6.1. It shows the plot average episodic score and average score of last 100 episodes respectively. It can be seen that the problem is solved in about 700 episodes.

```
import gymnasium as gym
# instantiate a gym environment
env = gym.make('CartPole-v0')
obs_shape = env.observation_space.shape
action_size = env.action_space.n
print('Observation shape: ', obs_shape)
print('Action Size: ', action_size)
print('Max Episode steps: ', env.spec.max_episode_steps)
# create an RL agent
agent = REINFORCEAgent(obs_shape, action_size)
# train the RL agent on
train_agent_for_env(env, agent, max_episodes=2000)
```

```
Output:
episode:0, score: 39.00, avgscore: 39.00, avg100score: 39.00
episode:100, score: 52.00, avgscore: 31.43, avg100score: 31.35
episode:200, score: 19.00, avgscore: 41.33, avg100score: 51.33
episode:300, score: 50.00, avgscore: 63.00, avg100score: 106.57
episode:400, score: 260.00, avgscore: 97.96, avg100score: 203.18
episode:500, score: 368.00, avgscore: 135.60, avg100score: 286.55
episode:600, score: 415.00, avgscore: 191.58, avg100score: 472.00
episode:700, score: 1184.00, avgscore: 360.26, avg100score: 1374.06
```

Listing 6.36: Solving `CartPole-v0` environment using REINFORCE agent.

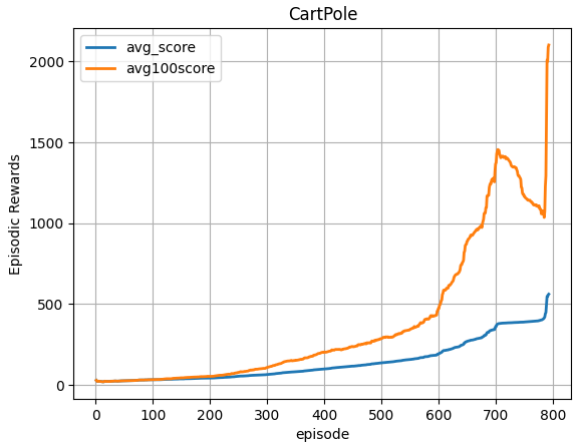
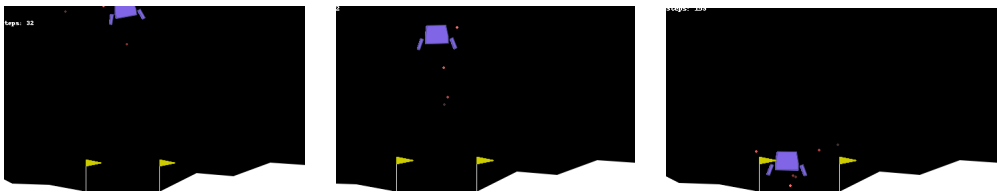


Figure 6.1.: Performance of REINFORCE algorithm on `Cartpole-v0` environment. It shows average episodic score and average score of last 100 episodes as training progresses.



(a) (b) (c)

Figure 6.2.: A few snapshots of Lunar-Lander environment.

6.3.3. Solving Lunar-Lander Problem

This environment is a classic rocket trajectory optimization problem. The problem involves controlling firing of three engines (left, right and the main) to ensure smooth landing on the landing pad. The code listing 6.37 shows how we can apply REINFORCE algorithm to solve the Gym's Lunar-Lander problem. A few snapshots of the problem environment is shown in Figure 6.2. The performance of REINFORCE algorithm on the `LunarLander-v2` environment is shown in Figure 6.3. We see the average score of last 100 episodes increase from -118 to about 100 in about 4000 episodes.

```
import gymnasium as gym
env = gym.make("LunarLander-v2", continuous=False)
obs_shape = env.observation_space.shape
action_size = env.action_space.n
# create a RL agent
agent = REINFORCEAgent2(obs_shape, action_size)
# Train for the environment
```

```
train_agent_for_env(env, agent, max_episodes=4000)
```

```
    Output:
episode:0, score: -118.37, avgscore: -118.37, avg100score: -118.37
episode:100, score: -89.99, avgscore: -158.09, avg100score: -158.49
episode:200, score: -108.09, avgscore: -146.92, avg100score: -135.63
episode:300, score: -103.37, avgscore: -142.65, avg100score: -134.06
episode:400, score: -240.33, avgscore: -140.28, avg100score: -133.16
episode:500, score: -101.36, avgscore: -135.14, avg100score: -114.54
...
...
episode:3300, score: 145.25, avgscore: -27.09, avg100score: 56.75
episode:3400, score: 12.40, avgscore: -24.93, avg100score: 46.65
episode:3500, score: -244.25, avgscore: -22.19, avg100score: 70.81
episode:3600, score: -207.85, avgscore: -19.61, avg100score: 70.74
episode:3700, score: 10.10, avgscore: -17.33, avg100score: 64.76
episode:3800, score: 34.17, avgscore: -15.91, avg100score: 36.53
episode:3900, score: 237.00, avgscore: -12.92, avg100score: 101.05
```

Listing 6.37: Applying REINFORCE agent to solve Lunar-Lander Problem.

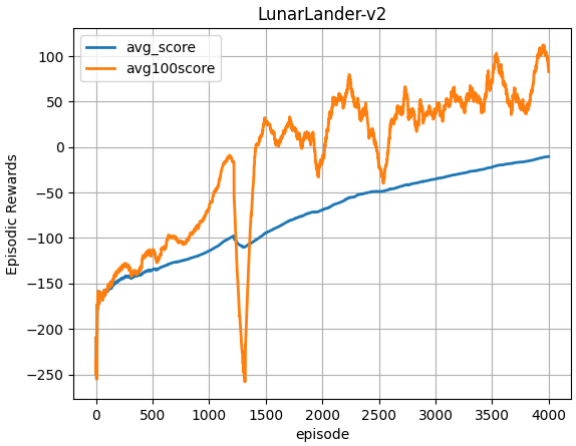


Figure 6.3.: Performance of REINFORCE algorithm on `LunarLander-v2` problem. The problem is considered solved if episodic score exceeds 200.

6.4. Actor-Critic Model

In the Monte-Carlo Policy Gradient Methods, one has to wait till the end of the episode to compute the *return* $G(t)$ required for computing policy parameter update as given by equation (6.20). If we have a high return, all the actions taken for an entire episode are considered good even if some actions were bad. As a consequence, a lot of samples are required to learn the optimal policy.

Rather than waiting till the end of the episode to compute an update, it will be better to update policy parameters at each time step by using an estimation of the Q function $Q(s, a)$ instead of the return $G(t)$ in equation (6.20). This will lead to the following policy parameter update equation:

$$\Delta\theta = \alpha\nabla_\theta(\log \pi_\theta(a|s))Q_\phi(s, a) \tag{6.21}$$

Hence, two different networks are required, one for estimating policy and other for estimating value or Q function. This is known as the actor-critic architecture which combines policy gradient methods and value-based methods that we covered in the previous chapter. In this architecture, an *actor* network is used to learn the policy function $\mu_\theta(s)$ where as a *critic* network is used to learn the state value function $V_\phi(s)$ or Q function $Q_\phi(s, a)$. So, while the actor's role is decide best action for the agent, the critic's role is to evaluate the action produced by the actor. The actor-critic architecture allows one to implement separate training algorithms for the actor and the critic networks and hence, provides greater flexibility compared to other models. The critic is a DQN that learns by minimizing the time-delay (TD) error. The actor network, on the other hand, learns the optimal action by using the update policy gradient method provided in equation (6.21) which essentially tries to maximizing the critic output. A schematic diagram of the actor-critic architecture is shown in Figure 6.4. We will discuss DDPG algorithm that will make use of this architecture in the next section.

6.5. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) algorithm [13] concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q function and then learns a policy by maximizing the Qfunction through a gradient ascent algorithm. This allows DDPG to apply deep Q-learning to continuous action spaces. We will use the actor-critic architecture to implement this algorithm. The actor network represented by $\mu(s; \theta)$ takes state as input and gives out action and θ is the actor network weights. The critic network represented by $Q(s, a; \phi)$ takes state and action as input and returns Q value and ϕ is the critic network weights. Similarly, we define target network for both the actor network and critic network as $\mu(s; \theta_{\text{targ}})$ and $Q(s, a; \phi_{\text{targ}})$ respectively, where θ_{targ} and ϕ_{targ} are target network parameters.

The critic network learns by using Q-learning algorithm where the following mean

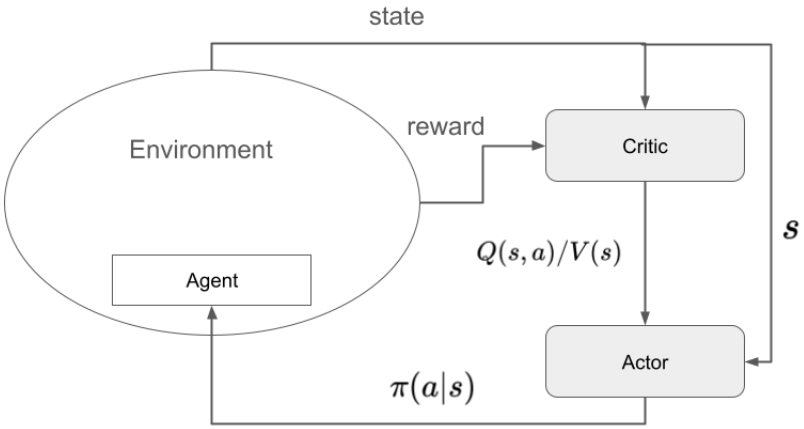


Figure 6.4.: Actor-Critic Architecture

squared bellman error (MBSE) loss is minimized with stochastic gradient descent:

$$\begin{aligned} L(\phi, \mathscr{D}) &= \mathbb{E}_{(s,a,r,s',d)\sim\mathscr{D}}\left[\left(Q_\phi(s,a) - (r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')))\right)^2\right] \\ &= \mathbb{E}_{(s,a,r,s',d)\sim\mathscr{D}}[Q_\phi(s,a) - y(r,s',d)] \end{aligned} \tag{6.22}$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy, $Q_{\phi_{\text{targ}}}$ is the target Q network and $y(r, s', d)$ is the target critic value required for training the Q network. $\mathscr{D}$ denotes the experience replay buffer to store experiences.

The policy learning in DDPG aims at learning a deterministic policy $\mu_\theta(s)$ which gives action that maximizes $Q_\phi(s, a)$. Since the action space is continuous, it is assumed that the Q function is differentiable with respect to action. It is to be noted that the symbol μ_θ is used to represent the *deterministic policy* which corresponds to the mean of the stochastic policy π_θ. The policy parameters θ are updated by performing gradient ascent to solve the following optimization problem:

$$\max_\theta \mathbb{E}_{s\sim\mathscr{D}}[Q_\phi(s, \mu_\theta(s))] \tag{6.23}$$

The parameters of the target networks are updated slowly compared to the main network by using Polyak Averaging [14] as shown below:

$$\begin{aligned} \phi_{\text{targ}} &\leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi \\ \theta_{\text{targ}} &\leftarrow \theta\phi_{\text{targ}} + (1-\rho)\theta \end{aligned}$$

where $0 \leq \rho \leq 1$ is a hyper parameter. It uses a hyper-parameter τ to slow down the update of target network weights compared to the main network. The pseudocode for

DDPG algorithm is provided in the Algorithm 6.1. The experiences to be stored in the replay buffer $\mathscr{D}$ is generated by adding a noise to policy network's output. The DDPG agent samples a batch of experiences from the replay buffer and uses it to train the actor and critic networks. As explained above, the actor implements gradient ascent algorithm by using a negative of the gradient term computed using Tensorflow's '`GradientTape`' utility. The critic, on the other hand, learns by applying gradient descent to minimize the time-delay Q-function error.

Algorithm 6.1 DDPG Algorithm

input: initial policy parameter θ, initial Q-function parameters ϕ, empty replay buffer $\mathscr{D}$.
set target parameters equal to the main parameters: $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$.
repeat
    Observe state s and select action $a = \text{clip}(\pi_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathscr{N}$.
    Execute the action a in the environment and obtain next state s', reward r, done signal d to indicate if it is a terminal state.
    Store (s, a, r, s', d) in the replay buffer $\mathscr{D}$
    if s' is terminal, reset the environment.
    if it's time to update **then**
        for for a certain number of steps **do**
            Randomly sample a batch of transitions $B = (s, a, r, s', d)$ from $\mathscr{D}$.
            Compute targets: $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \pi_{\theta_{\text{target}}}(s'))$
            Update Q function by applying one step of gradient descent using $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B}(Q_\phi(s, a) - y(r, s', d))^2$
            Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \pi_\theta(s))$
            Update target networks using Polyak Averaging given by equations (6.24)
        end for
    end if
until convergence

6.5.1. Python Implementation of DDPG algorithm

The first step is to create a class to generate noise that can be added to the deterministic action generated by the neural network model. This is shown in the code listing 6.38. It uses the Ornstein-Uhlenbeck process [15]. The next step involves creating a buffer class for storing experiences. We will use the same buffer class 5.18 that was used with DQN in the previous chapter. The next two steps involves creating actor and critic classes as will discussed in the following subsections.

```
class OUActionNoise:
    def __init__(self, mean, std_deviation, theta=0.15,
                 dt=1e-2, x_initial=None):
        self.theta = theta
```

```
        self.mean = mean
        self.std_dev = std_deviation
        self.dt = dt
        self.x_initial = x_initial
        self.reset()

    def __call__(self):
        x = (
            self.x_prev
            + self.theta * (self.mean - self.x_prev) * self.dt
            + self.std_dev * np.sqrt(self.dt) * \
                np.random.normal(size=self.mean.shape)
        )
        # Store x into x_prev
        # Makes next noise dependent on current one
        self.x_prev = x
        return x

    def reset(self):
        if self.x_initial is not None:
            self.x_prev = self.x_initial
        else:
            self.x_prev = np.zeros_like(self.mean)
```

Listing 6.38: Python Code for creating class for generating action noise using Ornstein-Uhlenbeck process [15].

Actor Model

The python code for creating an actor class is provided in the listing 6.39. The actor class uses a deep network to estimate deterministic actions ($\mu_\theta(s)$) from a given input state. It uses a `'tanh'` activation function to limit the action output in the range of [-1, 1] which is then scaled to the actual output range as required for a given problem. Following our understanding from DDQN architecture discussed in section 5.3, actor class uses a target network to provide training stability. It uses policy gradient method to maximize the critic output which estimates the Q function. The negative sign associated with the loss function indicates that we are applying gradient *ascent* to maximize the objective function. A separate deep network model could also be passed on as an argument to the class constructor if required.

```
class Actor():
    def __init__(self, obs_shape, action_size,
                 learning_rate=0.0003,
                 action_upper_bound=1.0,
```

```
                model=None):
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.lr = learning_rate
        self.action_upper_bound = action_upper_bound
        if model is None:
            self.model = self._build_model()
            self.target = self._build_model()
        else:
            self.model = model
            self.target = tf.keras.models.clone_model(model)
        self.optimizer = tf.keras.optimizers.Adam()
        # target shares same weights as the primary model in the beginning
        self.target.set_weights(self.model.get_weights())

    def _build_model(self):
        initializer = tf.keras.initializers.RandomUniform(
                                    minval=-0.01, maxval=0.01)
        s_input = tf.keras.layers.Input(shape=self.obs_shape)
        fc1 = tf.keras.layers.Dense(256, activation='relu',
                                    kernel_initializer=initializer)(s_input)
        fc2 = tf.keras.layers.Dense(256, activation='relu',
                                    kernel_initializer=initializer)(fc1)
        a_out = tf.keras.layers.Dense(self.action_size, activation='tanh',
                                      kernel_initializer=initializer)(fc2)
        a_out = a_out * self.action_upper_bound
        model = tf.keras.models.Model(s_input, a_out, name='actor')
        model.summary()
        return model

    def __call__(self, states, target=False):
        states
        if not target:
            pi = self.model(states)
        else:
            pi = self.target(states)
        return pi

    def update_target(self, tau=0.01):
        model_weights = self.model.get_weights()
        target_weights = self.target.get_weights()
        # update weights layer-by-layer using Polyak Averaging
        new_weights = []
        for w, w_dash in zip(model_weights, target_weights):
```

```
                new_w = tau * w + (1 - tau) * w_dash
                new_weights.append(new_w)
            self.target.set_weights(new_weights)

        def train(self, states, critic):
            with tf.GradientTape() as tape:
                actor_weights = self.model.trainable_variables
                actions = self.model(states)
                critic_values = critic(states, actions)
                # -ve value is used to maximize the function
                actor_loss = -tf.math.reduce_mean(critic_values)
            actor_grad = tape.gradient(actor_loss, actor_weights)
            self.optimizer.apply_gradients(zip(actor_grad, actor_weights))
            return actor_loss
```

Listing 6.39: Actor class for DDPG algorithm. Actor estimates the policy function. It learns by maximizing critic output.

Critic Class

The python code for creating a critic class is provided in the listing 6.40. The critic uses a deep network to estimate the Q value function for a given state-action pair as input. It is similar to the DQN architecture discussed in the previous chapter. It also uses a target network similar to the DDQN architecture discussed in the previous chapter to provide better training stability. The critic learns by minimizing the TD Q function error provided by equation (6.22). The target value y is computed using both target actor and target critic models. The target network is updated at regular intervals using Polyak Averaging. The value $\tau = 1.0$ indicates a *hard update* where the target parameters are replaced by the primary model parameters. The value $\tau < 1.0$ indicates a *soft update* where the target parameters are replaced by a weighted average of these two network parameters.

```
class Critic:
    def __init__(self, obs_shape, action_size,
                 learning_rate=0.0003,
                 gamma=0.99,
                 model=None):
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.gamma = gamma
        self.lr = learning_rate

        if model is None:
```

```
            self.model = self._build_model()
            self.target = self._build_model()
        else:
            self.model = model
            self.target = tf.keras.models.clone_model(model)
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=self.lr)

        # target shares same weights as the main model in the beginning
        self.target.set_weights(self.model.get_weights())

    def _build_model(self):
        s_input = tf.keras.layers.Input(shape=self.obs_shape)
        s_out = tf.keras.layers.Dense(16, activation='relu')(s_input)
        s_out = tf.keras.layers.Dense(32, activation='relu')(s_out)

        # action as input
        a_input = tf.keras.layers.Input(shape=(action_size, ))
        a_out = tf.keras.layers.Dense(32, activation='relu')(a_input)

        # concat [s, a]
        concat = tf.keras.layers.Concatenate()([s_out, a_out])
        out = tf.keras.layers.Dense(256, activation='relu')(concat)
        out = tf.keras.layers.Dense(256, activation='relu')(out)
        net_out = tf.keras.layers.Dense(1)(out)

        # output is the Q-value output
        model = tf.keras.models.Model(inputs=[s_input, a_input],
                                      outputs=net_out, name='critic')
        model.summary()
        return model

    def __call__(self, states, actions, target=False):
        if not target:
            value = self.model([states, actions])
        else:
            value = self.target([states, actions])
        return value

    def update_target(self, tau=0.01):
        model_weights = self.model.get_weights()
        target_weights = self.target.get_weights()
        # update weights layer-by-layer using Polyak Averaging
        new_weights = []
        for w, w_dash in zip(model_weights, target_weights):
```

```
            new_w = tau * w + (1 - tau) * w_dash
            new_weights.append(new_w)
        self.target.set_weights(new_weights)

    def train(self, states, actions, rewards, next_states, dones, actor):
        with tf.GradientTape() as tape:
            critic_weights = self.model.trainable_variables
            target_actions = actor(states, target=True)
            target_q_values = self.target([next_states, target_actions])
            y = rewards + self.gamma * (1-dones) * target_q_values
            q_values = self.model([states, actions])
            critic_loss = tf.math.reduce_mean(tf.square(y - q_values))
        critic_grads = tape.gradient(critic_loss, critic_weights)
        self.optimizer.apply_gradients(zip(critic_grads, critic_weights))
        return critic_loss
```

Listing 6.40: Critic Class for DDPG algorithm. Critic estimates the Q function and learns by minimizing the TD error.

DDPG Agent

The python code for implement DDPG agent class is provided in Listing 6.41. It uses the actor and critic class defined above to implement the actor-critic architecture for implementing DDPG algorithm. Since it uses a deterministic network to estimate action, a noise is added for exploration during training. During training, a batch of experiences is sampled from the replay buffer and is used for training both actor and critic simultaneously.

```
class DDPGAgent:
    def __init__(self, obs_shape, action_size,
                 batch_size, buffer_capacity,
                 action_upper_bound=1.0,
                 action_lower_bound=-1.0,
                 lr_a=1e-3, lr_c=1e-3, gamma=0.99,
                 noise_std=0.2,
                 actor_model=None,
                 critic_model=None):
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.buffer_capacity = buffer_capacity
        self.batch_size = batch_size
        self.action_upper_bound = action_upper_bound
        self.action_lower_bound = action_lower_bound
        self.gamma = gamma
```

```
        self.lr_a = lr_a
        self.lr_c = lr_c
        self.noise_std = noise_std

        self.actor = Actor(self.obs_shape, self.action_size,
                           learning_rate=self.lr_a,
                           action_upper_bound=self.action_upper_bound,
                           model=actor_model)
        self.critic = Critic(self.obs_shape, self.action_size,
                             learning_rate=self.lr_c,
                             gamma=self.gamma,
                             model=critic_model)
        self.buffer = ReplayBuffer(self.buffer_capacity)
        self.action_noise = OUActionNoise(mean=np.zeros(1),
                                          std_deviation=float(self.noise_std) * np.ones(1))

    def policy(self, state):
        action = tf.squeeze(self.actor(state))
        noise = self.action_noise()
        # add noise to action
        sampled_action = action.numpy() + noise
        # check action bounds
        valid_action = np.clip(sampled_action, self.action_lower_bound,
                               self.action_upper_bound)
        return valid_action

    def experience_replay(self):
        if len(self.buffer) < self.batch_size:
            return
        mini_batch = self.buffer.sample(self.batch_size)
        states = np.zeros((self.batch_size, *self.obs_shape))
        next_states = np.zeros((self.batch_size, *self.obs_shape))
        actions = np.zeros((self.batch_size, self.action_size))
        rewards = np.zeros((self.batch_size, 1))
        dones = np.zeros((self.batch_size, 1))
        for i in range(len(mini_batch)):
            states[i] = mini_batch[i][0]
            actions[i] = mini_batch[i][1]
            rewards[i] = mini_batch[i][2]
            next_states[i] = mini_batch[i][3]
            dones [i] = mini_batch[i][4]
        states = tf.convert_to_tensor(states, dtype=tf.float32)
        actions = tf.convert_to_tensor(actions, dtype=tf.float32)
        rewards = tf.convert_to_tensor(rewards, dtype=tf.float32)
```

```
            next_states = tf.convert_to_tensor(next_states, dtype=tf.float32)
            dones = tf.convert_to_tensor(dones, dtype=tf.float32)
            a_loss = self.actor.train(states, self.critic)
            c_loss = self.critic.train(states, actions, rewards, \
                next_states, dones, self.actor)
        return a_loss, c_loss

    def update_targets(self, tau_a=0.01, tau_c=0.02):
        self.actor.update_target(tau_a)
        self.critic.update_target(tau_c)
```

Listing 6.41: Python code for implementing DDPG Agent. It uses the actor-critic architecture.

6.5.2. Solving Pendulum Problem

Gym's pendulum environment[1] is a classical control problem which is also known as the inverted pendulum swingup problem in control theory. The system consists of a pendulum attached at one end to a fixed point, and the other end being free. The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position, with its center of gravity right above the fixed point. The observation and the action spaces are continuous spaces unlike the environments used in the previous chapters. Some of the states of the pendulum is shown in Figure 6.5. The upright vertical position (d) is the desirable state of the system. The range of values for input, output and reward elements are shown in Table 6.1. Each episode involves 200 iteration steps. The total reward for an episode is the sum of individual rewards obtained at each of these steps. Since, no training is involved in this code, the total reward for each of these episodes will be a large negative number. The problem is considered solved if the average reward for an episode remains above -200 for a considerable amount of time (say, 50 episodes).

Variable Name	shape	Elements	Min	Max
Observation (state), s	(3,1)	$\cos\theta$	-1.0	1.0
		$\sin\theta$	-1.0	1.0
		$\dot{\theta}$	-8.0	8.0
Action, a	(1,)	Joint Effort	-2.0	2.0
Reward, r	(1,)	$-\theta^2 + 0.1\dot{\theta}^2 + 0.001a^2$	-16.273	0

Table 6.1.: Input-Output variables for '`Pendulum-v1`' Gym Simulation Environment.

[1] `https://gymnasium.farama.org/environments/classic_control/pendulum/`

(a) (b) (c) (d)

Figure 6.5.: A few snapshots of `Pendulum-v1` environment states. (d) shows the final successful state of the environment.

Training DDPG Agent

The code for training a DDPG agent to solve the Pendulum problem environment is provided in Listing 6.42. The episode is terminated after 200 iterative steps. The experiences are generated by using a stochastic policy, which is itself obtained by adding stochastic noise to a deterministic action provided by the actor network. The experiences are stored in a replay buffer and are then sampled in batches during training.

```
import gymnasium as gym
import os
import sys
def solve_problem(env, agent, max_episodes=500):
    tau_a, tau_c = 0.01, 0.01
    scores = []
    for e in range(max_episodes):
        state = env.reset()[0]
        ep_score = 0
        done = False
        steps = 0
        while not done:
            tf_state = tf.expand_dims(tf.convert_to_tensor(state), axis=0)
            # take action
            action = agent.policy(tf_state)
            # make transition and receive reward
            next_state, reward, done, _, _ = env.step(action)
            # store experience in the replay buffer
            agent.buffer.add((state, action, reward, next_state, done))
            ep_score += reward
            total_steps += 1
            steps += 1
            state = next_state
            # train the agent
            agent.experience_replay()
```

```
            # update target models
            agent.update_targets(tau_a, tau_c)
            if steps > 200: # terminate the episode
                done = True
        # end of while loop
        scores.append(ep_score)
        if e % 20 == 0:
            print(f'episode: {e}, score: {ep_score:.2f}, \
            avg_score: {np.mean(scores):.2f}, \
            avg50score: {np.mean(scores[-50:]):.2f}')
    # end of for loop
    file.close()
```

Listing 6.42: Python function for solving a Gym problem environment

6.5.3. Main Code to generate output

The main code for solving the pendulum problem using DDPG agent is provided in Listing 6.43. We create an instance for the `Pendulum-v1` environment. Then, we create a DDPG agent and finally call the `solve_problem()` function to train the agent. The listing also shows the output of this program when executed. As one can see, the average episodic reward increases from about -1400 to about -230 in about 500 episodes. The training performance of the DDPG algorithm in solving the pendulum problem is shown graphically in Figure 6.6. The variable `avg50score` represents the average score of last 50 episodes. As we can see, this value reaches above -200 indicating that the problem is solved successfully.

```
import gymnasium as gym
# create an environment
env = gym.make('Pendulum-v1', g=9.81, render_mode="rgb_array")

obs_shape = env.observation_space.shape
action_shape = env.action_space.shape
action_size = action_shape[0]
action_ub = env.action_space.high
action_lb = env.action_space.low
print('Observation shape: ', obs_shape)
print('Action shape: ', action_shape)
print('Max episodic Steps: ', env.spec.max_episode_steps)
print('Action space bounds: ', (action_ub[0], action_lb[0]))

# create an agent
agent = DDPGAgent(obs_shape, action_size,
```

```
                batch_size=128, buffer_capacity=20000,
                action_upper_bound=2.0,
                action_lower_bound=-2.0)

# solve a problem
solve_problem(env, agent, max_episodes=500)
```

```
Output:
Observation shape:  (3,)
Action shape:  (1,)
Max episodic Steps:  200
Action space bounds:  (2., -2)
episode: 0, score: -1392.89, avg_score: -1392.89,  avg50score: -1392.89
episode: 20, score: -249.34, avg_score: -1142.08,  avg50score: -1142.08
episode: 40, score: -124.50, avg_score: -657.28,   avg50score: -657.28
episode: 60, score: -131.54, avg_score: -483.67,   avg50score: -293.25
episode: 80, score: -250.10, avg_score: -417.34,   avg50score: -163.04
...
episode: 400, score: -116.90, avg_score: -240.71, avg50score: -183.42
episode: 420, score: -243.59, avg_score: -238.83, avg50score: -190.46
episode: 440, score: -125.23, avg_score: -239.29, avg50score: -215.33
episode: 460, score: -131.01, avg_score: -236.41, avg50score: -217.93
episode: 480, score: -228.30, avg_score: -235.49, avg50score: -188.83
```

Listing 6.43: Main code for training DDPG agent to solve the `Pendulum-v1` problem.

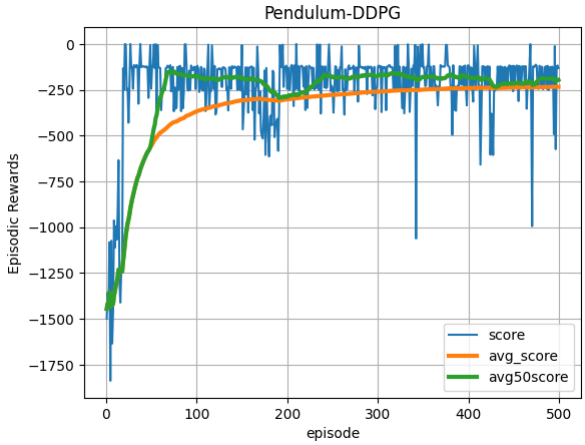


Figure 6.6.: Training performance of DDPG algorithm in solving `Pendulum-v1` problem

6.6. Trust Region Policy Optimization

RL agents learn by trial and error to maximize the cumulative expected rewards for a given task. The agent finds the optimal policy by exploring all kinds of actions and memorizes actions that give good rewards. During exploration, the agent will execute some bad actions as well which could be unsafe or dangerous for an agent operating in a real-world environment. Consider the example of an autonomous vehicle learning to avoid obstacles. It can not afford to make collisions with obstacles just to check if it gives good reward. Therefore, it becomes important in such scenarios to constrain the learning by ensuring that the agent explores in a safe region.

Trust Region Policy Optimization (TRPO) is one such constrained policy optimization method that aims to maximize the expected return of a policy while ensuring that updates to the policy are safe and do not lead to catastrophic performance degradation. It does this by restricting the policy updates to a *trust region*, a neighborhood around the current policy where the expected return is guaranteed to improve. This trust region is defined by imposing a constraint such that the Kullbach-Leibler (KL) divergence between the old policy and the new policy is less than some constant *delta*, also known as trust region constant. KL divergence measures the difference between two probability distributions. It tells us how far a new policy is from the old policy. By imposing this constraint, we ensure that the agent improves the policy leading to higher rewards while avoiding undesirable behaviour.

The detailed mathematical derivation of these constraints are provided in [16]. A part of this derivation is reproduced here for the sake of completion.

The expected discounted reward for a given policy π is given by

$$\eta_\pi = \mathop{\mathbb{E}}_{s_0, a_0, \ldots} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t) \right] \tag{6.24}$$

where $s_0 \sim \rho_0(s_0)$, $a_t \sim \pi(a_t|s_t)$, $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$.

The expected return for a new policy $\tilde{\pi}$ can be written in terms of the advantage over π accumulated over time steps as given by

$$\eta(\tilde{\pi}) = \eta(\pi) + \mathop{\mathbb{E}}_{s_0, a_0, \ldots \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t, a_t) \right] \tag{6.25}$$

where $A_\pi(s_t, a_t)$ is the advantage of the old policy π and $A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$.

We can rewrite the equation (6.25) with a sum over states instead of time steps as follows:

$$\begin{aligned} \eta(\tilde{\pi}) &= \eta(\pi) + \sum_{t=0}^{\infty} \sum_{s} P(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s) \gamma^t A_\pi(s, a) \\ &= \eta(\pi) + \sum_{s} \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s) A_\pi(s, a) \\ &= \eta(\pi) + \sum_{s} \rho_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a|s) A_\pi(s, a) \end{aligned} \tag{6.26}$$

where ρ_π is the discounted visitation frequencies given by

$$\rho_\pi(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \cdots \tag{6.27}$$

The above equation (6.26) implies that any policy update $\pi \rightarrow \tilde{\pi}$ that has a non-negative expected advantage at *every* state s, i.e., $\sum_a \tilde{\pi}(a|s) A_\pi(s,a) \geq 0$, is guaranteed to increase the policy performance η or leave it constant in case the expected advantage is zero everywhere. However, due to approximation and estimation errors, it is not possible to meet this requirement and the expected advantage could become negative for some states. The complex dependency of $\rho_{\tilde{\pi}}$ on $\tilde{pi}$ makes equation (6.26) to optimize directly.

To simplify the problem, we use a local approximation to η:

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a) \tag{6.28}$$

Note that L_π uses the visitation frequency ρ_π rather than $\rho_{\tilde{\pi}}$, ignoring the changes in state visitation density due to changes in policy. If we have a parameterized policy function π_θ, where $\pi_\theta(a|s)$ is differentiable function of the parameter vector θ, then L_π matches to η to the first order for sufficiently small steps. However, it does not give us any guidance on how big of a step to take.

To address this issue, Kakade and Langford [17] proposed a policy update scheme called conservative policy iteration as given below:

$$\pi_{\text{new}}(a|s) = (1-\alpha)\pi_{\text{old}}(a|s) + \alpha\pi'(a|s) \tag{6.29}$$

where $\pi' = \arg\max_{\pi'} L_{\pi_{\text{old}}}(\pi')$. In other words, π' is the policy that maximizes $L_{\pi_{\text{old}}}$. The above equation (6.29) gives rise to the following lower bound on η:

$$\eta(\tilde{\pi}) \geq L_\pi(\tilde{\pi}) - C D_{KL}^{\max}(\pi, \tilde{\pi}) \tag{6.30}$$

where $C = \frac{4\epsilon\gamma}{(1-\gamma)^2}$ is the penalty coefficient with $\epsilon = \max_{s,a} |A_\pi(s,a)|$ and D_{KL}^{max} denotes the maximum KL divergence between the old policy and the new policy over all states given by:

$$D_{KL}^{\max}(\pi, \tilde{\pi}) = \max_s D_{KL}(\pi(.|s), \tilde{\pi}(.|s)) \tag{6.31}$$

This equation shows that by maximizing the right hand side, it can be guaranteed that the objective function η is non-decreasing. To show this, let's denote the right hand side term at any given iteration i with $M_i(\pi)$ given by:

$$M_i(\pi) = L_{\pi_i}(\pi) - C D_{KL}^{\max}(\pi_i, \pi) \tag{6.32}$$

Then we have,

$$\eta(\pi_{i+1}) \geq M_i(\pi_{i+1}); \text{ by using equation (6.30)} \tag{6.33}$$

Now, let's compute $M_i(\pi = \pi_i)$ from equation (6.32) as shown below:

$$\begin{aligned} M_i(\pi_i) &= L_{\pi_i}(\pi_i) - C D_{KL}^{\max}(\pi_i, \pi_i) \\ &= L_{\pi_i}(\pi_i) \quad \because D_{KL}(\pi_i, \pi_i) = 0 \\ &= \eta(\pi_i) \quad \text{by using equation (6.28)} \\ \Rightarrow M_i(\pi_i) &= \eta(\pi_i) \end{aligned} \tag{6.34}$$

Now, subtracting (6.34) from (6.33), we get

$$\eta(\pi_{i+1}) - \eta(\pi_i) \geq M_i(\pi_{i+1}) - M_i(\pi_i) \tag{6.35}$$

This shows that by maximizing M_i at each iteration, we can ensure that the objective function η is non-decreasing. So, for a parametrized policy $\pi_\theta(a|s)$, we can improve the policy by performing the following maximization:

$$\underset{\theta}{\textbf{maximize}}[L_{\theta_{\text{old}}}(\theta) - CD_{KL}^{max}(\theta_{\text{old}}, \theta)] \tag{6.36}$$

In practice, the use of penalty coefficient C in the above equation will lead to very small step size, thereby slowing down the update. This can be remedied by putting a constraint on KL divergence between the new policy and the old policy instead. This is called a trust region constraint. The modified policy optimization problem becomes the following:

$$\begin{aligned} \underset{\theta}{\textbf{maximize}} \quad & L_{\theta_{\text{old}}(\theta)} \\ \text{subject to} \quad & D_{KL}^{\max}(\theta_{\text{old}}, \theta) \leq \delta \end{aligned} \tag{6.37}$$

This problem imposes a constraint that the KL divergence is bounded at every point in the state space. This becomes impractical to solve in practice due to the large number of constraints. Instead, we use a heuristic approximation which considers the average KL divergence:

$$\bar{D}_{KL}^{\rho}(\theta_1, \theta_2) = \mathbb{E}_{s\sim\rho}[D_{KL}(\pi_{\theta_1}(.|s), \pi_{\theta_2}(.|s))] \tag{6.38}$$

Therefore, the final optimization problem becomes:

$$\begin{aligned} \textbf{maximize}_\theta \quad & L_{\theta_{\text{old}}}(\theta) \\ \text{subject to} \quad & \bar{D}_{KL}^{\rho\theta_{\text{old}}}(\theta_{\text{old}}, \theta) \leq \delta \end{aligned} \tag{6.39}$$

Expanding L, we get the following optimization problem:

$$\begin{aligned} \textbf{maximize}_\theta \quad & \textstyle\sum_s \rho\theta_{\text{old}}(s) \sum_a \pi_\theta(a|s) A_{\theta_{\text{old}}}(s,a) \\ \text{subject to} \quad & \bar{D}_{KL}^{\rho\theta_{\text{old}}}(\theta_{\text{old}}, \theta) \leq \delta \end{aligned} \tag{6.40}$$

We now replace the sum over states $\sum_s \rho\theta_{\text{old}}(s)$ by expectation $\mathbb{E}_{s\in\rho\theta_{\text{old}}}[\cdots]$ and replace the sum over actions by an importance sampling estimator. Using q to denote the sampling distribution, the contribution of a single state s_n to the loss function is given by

$$\sum_a \pi_\theta(a|s_n) A_{\theta_{\text{old}}}(s_n, a) = \mathbb{E}_{a\sim q}\left[\frac{\pi_\theta(a|s_n)}{q(a|s_n)} A_{\theta_{\text{old}}}(s_n, a)\right] \tag{6.41}$$

Then we replace the advantage values $A_{\theta_{\text{old}}}$ by Q values $Q_{\theta_{\text{old}}}$, the above constrained optimization problem becomes:

$$\begin{aligned} \textbf{maximize}_\theta \quad & \mathbb{E}_{s\sim\rho\theta_{\text{old}}, a\sim q}\left[\tfrac{\pi_\theta(a|s)}{q(a|s)} Q_{\theta_{\text{old}}}(s,a)\right] \\ \text{subject to} \quad & \mathbb{E}_{s\sim\rho\theta_{\text{old}}}[D_{KL}(\pi_{\theta_{\text{old}}}(.|s)||\pi_\theta(.|s))] \leq \delta \end{aligned} \tag{6.42}$$

The constraint ensures that the new policy does not deviate too much from the current policy.

6.7. Proximal Policy Optimization (PPO) Algorithm

In the previous section, we see that TRPO uses a surrogate objective function for policy optimization while imposing a constraint that the KL divergence between the old and the new policy should be less than δ. This optimization problem can be approximately solved by using the conjugate gradient algorithm, after making a linear approximation to the objective function and a quadratic approximation to the constraint. In general, it is a computationally expensive process. The proximal policy optimization (PPO) [18] simplifies TRPO by using a modified objective function where the constraint is converted into a penalty term. There are two approaches to achieve this as explained in the following two sub-sections.

6.7.1. Clipped Surrogate Objective

Let us denote the probability ratio between new and old policy as $r(\theta)$ as shown below:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \tag{6.43}$$

The objective function to be optimized can be rewritten in terms of r by making substitution in (6.40)

$$J^{\text{CPI}}(\theta) = \mathbb{E}[r(\theta)\hat{A}_{\theta_{\text{old}}}] \tag{6.44}$$

where $\hat{A}$ represents the estimated advantage value and the superscript CPI stands for conservative policy iteration. Maximizing this objective function without TRPO constraint would lead to instability with large parameter updates and big policy ratios. PPO avoids this by forcing $r(\theta)$ to stay within a small interval around 1, given by the range $[1-\epsilon, 1+\epsilon]$ where ϵ is a small hyper-parameter, say $\epsilon = 0.2$. Therefore, the modified cost function to be maximized in PPO is given as follows:

$$J^{\text{CLIP}}(\theta) = \mathbb{E}[\min(r(\theta)\hat{A}_{\text{old}}, \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{\text{old}}})] \tag{6.45}$$

The final objective function is a lower bound on the unclipped objective. The probability ratio will be clipped at $1+\epsilon$ or at $1-\epsilon$ based on two cases as shown in the Figure 6.7. The left figure shows the case where the advantage is positive ($A > 0$) indicating that the new policy is doing better than the old policy. So the probability ratio r will be allowed to increase until it reaches $1+\epsilon$. On the other hand, the right figure shows the case where the advantage is negative ($A < 0$), indicating that the new policy is performing worse. In such a case, r will be reduced until it reaches the lower boundary $1-\epsilon$.

While applying PPO to actor-critic models, the above objective function (6.45) is augmented with an value estimation error term and an entropy term as shown below for better performance:

$$J^{\text{CLIP}+VE+S} = \mathbb{E}[J^{\text{CLIP}}(\theta) - c_1(V_\theta(s) - V_{\text{target}})^2 + c_2 S(s, \pi_\theta(.))] \tag{6.46}$$

The entropy term improves exploration during the training process. The resulting PPO algorithm with a clipped surrogate objective function is provided in the Algorithm listing 6.2

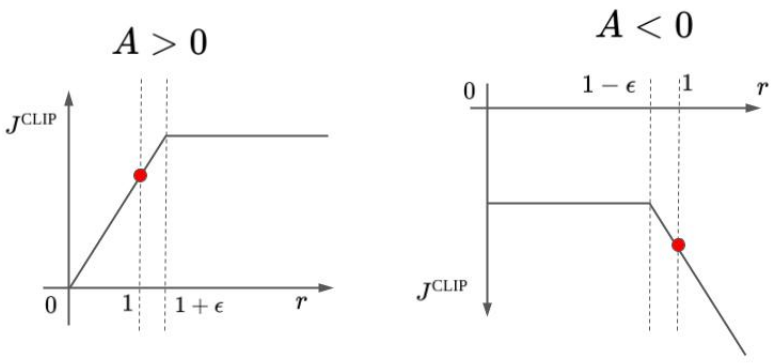


Figure 6.7.: Single time step of surrogate function J^{CLIP} as a function of probability ratio r for positive (left) and negative (right) advantages. The red circle shows the starting point of policy optimization

Algorithm 6.2 PPO-Clip Algorithm

Require: initial policy parameters θ_0, initial value function parameters ϕ_0

1: **for** $k = 0, 1, 2, \ldots$ **do**

2: Collect a set of trajectories $\mathscr{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

3: Compute rewards-to-go $\hat{R}_k$

4: Compute advantage estimates $\hat{A}_k$ based on the current value function $V(\phi_k)$

5: Update the policy by maximizing the PPO-Clip objective via stochastic gradient ascent:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathscr{D}_k|T} \sum_{\tau \in \mathscr{D}_k} \sum_{t=0}^{T} \min \left[r(\theta) A^{\pi_k}(s_t, a_t), g(\epsilon, A^{\pi_k}(s_t, a_t)) \right] \tag{6.47}$$

where

$$g(\epsilon, a) = \begin{cases} (1+\epsilon)A \text{ if } A \geq 0 \\ (1-\epsilon)A \text{ if } A < 0 \end{cases}$$

6: Update value function to minimize the following error function by using gradient descent algorithm:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathscr{D}_k|T} \sum_{\tau \in \mathscr{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2 \tag{6.48}$$

7: **end for**

6.7.2. Adaptive KL Penalty Coefficient

Another approach, as an alternative to the above clipped surrogate objective, is to use a penalty on the KL divergence to convert the TRPO constrained optimization problem given by (6.42) into an unconstrained optimization problem given by

$$\underset{\theta}{\textbf{maximize}}\, \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\hat{A} - \beta D_{KL}(\pi_{\theta_{\text{old}}}, \pi_\theta)\right] \tag{6.49}$$

where β is the penalty coefficient is adapted to achieve some target value of KL divergence d_{targ} after each policy update as per the following rule. Compute $d = D_{KL}(\pi_{\theta_{\text{old}}}, \pi_\theta)$ and,

$$\begin{aligned} &\text{If} \quad d < d_{\text{targ}}/1.5, \quad \beta \leftarrow \beta/2 \\ &\text{If} \quad d > d_{\text{targ}} \times 1.5, \quad \beta \leftarrow \beta \times 2 \end{aligned} \tag{6.50}$$

The corresponding PPO algorithm is provided in the Algorithm listing 6.3.

Algorithm 6.3 PPO with Adaptive KL Penalty

Require: initial policy parameter θ_0, initial value parameter ϕ_0, initial KL penalty β_0, target KL-divergence δ

1: **for** $k = 0, 1, 2, \ldots$ **do**
2: Collect a set of partial trajectories $\mathscr{D}_k$ on policy $\pi = \pi(\theta_k)$.
3: Estimate Advantage $\hat{A}_k$ based on current value function $V(\phi_k)$
4: Compute policy update by taking k steps of mini-batch SGD.

$$\theta_{k+1} = \arg\max_\theta [r_{\theta_k}(\theta)\hat{A}_{\theta_k} - \beta_k \bar{D}_{KL}(\theta||\theta_k)] \tag{6.51}$$

5: for each policy update step k, update the value of β using (6.50)
6: **end for**

6.7.3. Generalized Advantage Estimator

Generalized Advantage Estimator (GAE) is a technique to estimate the advantage function, a crucial component in policy gradient methods. The advantage function measures how much better a particular action is compared to the average action. It's essential for guiding the agent towards actions that lead to higher rewards. The advantage function can be estimated from the value function estimates (details are available in [19]) by using the generalized advantage estimator given by the following equation:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{6.52}$$

where parameter $0 \leq \lambda \leq 1$ controls the trade-off between bias and variance and δ_t represents the time delay error given by

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{6.53}$$

6.7.4. Python implementation of PPO Algorithm

We implement the PPO algorithm using Actor-Critic architecture as described earlier. Actor class learns the policy distribution by maximizing a surrogate objective function. On the other hand, the Critic class learns the value function by minimizing the TD error.

PPO Actor Class

The Python code for implementing a PPO Actor class is provided in the code Listing 6.44. The actor class uses a neural network to estimate the mean and standard deviation of the policy distribution. It has the option of receiving an externally created deep network through constructor argument `model`. The `train` method implements both versions of PPO algorithm, namely, `clip` and `penalty` method as given by equations (6.47) and (6.51) respectively. The negative sign with actor loss indicates that it maximizes the cost function. The `penalty` method updates the entropy coefficient β by using the equation (6.50). Please note that both the probability ratio $r(\theta)$ and the advantage $A(\theta)$ have the same shape.

```
import tensorflow as tf
import numpy as np
import tensorflow_probability as tfp
class PPOActor():
    def __init__(self, obs_shape, action_size,
                 learning_rate=0.0003,
                 action_upper_bound=1.0,
                 epsilon=0.2, lmbda=0.5, kl_target=0.1,
                 beta=0.1, entropy_coeff=0.2,
                 critic_loss_coeff=0.1,
                 grad_clip=10.0,
                 method='clip',
                 model=None):
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.action_ub = action_upper_bound
        self.lr = learning_rate
        self.epsilon = epsilon # clip on ratio
        self.lam = lmbda # required for penalty method
        self.beta = beta # kl penalty coefficient
        self.entropy_coeff = entropy_coeff
        self.c_loss_coeff = critic_loss_coeff
        self.method = method # choose between 'clip' and 'penalty'
        self.kl_target = kl_target
        self.kl_value = 0 # to store most recent kld value
        self.grad_clip = grad_clip # applying gradient clipping
        # create actor model
```

```
        if model is None:
            self.model = self._build_model()
        else:
            self.model = tf.keras.models.clone_model(model)
        self.optimizer = tf.keras.optimizers.Adam(self.lr)
        # additional parameters
        logstd = tf.Variable(np.zeros(shape=(self.action_size, )), \
                             dtype=np.float32)
        self.model.logstd = logstd
        self.model.trainable_variables.append(logstd)

    def _build_model(self):
        last_init = tf.random_uniform_initializer(minval=-0.01, maxval=0.01)
        state_input = tf.keras.layers.Input(shape=self.obs_shape)
        x = tf.keras.layers.Dense(128, activation='relu')(state_input)
        x = tf.keras.layers.Dense(64, activation='relu')(x)
        x = tf.keras.layers.Dense(64, activation='relu')(x)
        net_out = tf.keras.layers.Dense(self.action_size, activation='tanh',
                                        kernel_initializer=last_init)(x)
        net_out = net_out * self.action_ub
        model = tf.keras.models.Model(state_input, net_out, name='actor')
        model.summary()
        return model

    def __call__(self, state):
        # input is a tensor
        mean = tf.squeeze(self.model(state))
        std = tf.squeeze(tf.exp(self.model.logstd))
        return mean, std # return tensor

    def train(self, state_batch, action_batch, advantages, old_pi, c_loss):
        with tf.GradientTape() as tape:
            mean = tf.squeeze(self.model(state_batch))
            std = tf.squeeze(tf.exp(self.model.logstd))
            pi = tfp.distributions.Normal(mean, std)
            # r = pi/pi_old
            ratio = tf.exp(pi.log_prob(tf.squeeze(action_batch)) -
                           old_pi.log_prob(tf.squeeze(action_batch)))
            if ratio.ndim > advantages.ndim: # match shapes
                ratio = tf.reduce_mean(ratio, axis=-1)
            surr_obj = ratio * advantages # surrogate objective function
            # current kl divergence (kld) value
            kld = tfp.distributions.kl_divergence(old_pi, pi)
            if kld.ndim > advantages.ndim:
```

```
            kld = tf.reduce_mean(kld, axis=-1)
            self.kl_value = tf.reduce_mean(kld)
            entropy = tf.reduce_mean(pi.entropy()) # entropy
            if self.method == 'penalty':
                actor_loss = -(tf.reduce_mean(surr_obj - self.beta * kld))
            elif self.method == 'clip':
                l_clip = tf.reduce_mean(
                    tf.minimum(surr_obj, tf.clip_by_value(ratio,
                    1.-self.epsilon, 1.+self.epsilon) * advantages))
                actor_loss = -(l_clip - self.c_loss_coeff * c_loss  + \
                    self.entropy_coeff * entropy)
            else:
                raise ValueError('invalid option for PPO method')
            actor_weights = self.model.trainable_variables
            actor_grad = tape.gradient(actor_loss, actor_weights)
            if self.grad_clip is not None:
                actor_grad = [tf.clip_by_value(grad,\
                    -1 * self.grad_clip, self.grad_clip)\
                    for grad in actor_grad]
        #outside gradient tape
        self.optimizer.apply_gradients(zip(actor_grad, actor_weights))
    return actor_loss.numpy()

def update_beta(self):
    # update beta after each epoch
    if self.kl_value < self.kl_target / 1.5:
        self.beta /= 2.
    elif self.kl_value > self.kl_target * 1.5:
        self.beta *= 2.
```

Listing 6.44: Python code for PPO Actor Class.

PPO Critic Class

The Python code for implementing a PPO Critic network is provided in the code Listing 6.45. The Critic class uses a dense network to approximate value function $V(s)$. It learns by minimizing the TD error which is the difference between the current value and the return for the current batch. Its function is similar to that of a DQN model. The gradient values could be clipped through the `grad_clip` argument of the constructor.

```
class PPOCritic():
    def __init__(self, obs_shape, action_size,
                 learning_rate=0.0003,
```

```
                    gamma=0.99,
                    grad_clip = None,
                    model=None):
        self.lr = learning_rate
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.gamma = gamma
        self.grad_clip = grad_clip
        if model is None:
            self.model = self._build_model()
        else:
            self.model = tf.keras.models.clone_model(model)
        self.optimizer = tf.keras.optimizers.Adam(self.lr)

    def __call__(self, state):
        # input is a tensor
        value = tf.squeeze(self.model(state))
        return value

    def _build_model(self):
        state_input = tf.keras.layers.Input(shape=self.obs_shape)
        out = tf.keras.layers.Dense(64, activation="relu")(state_input)
        out = tf.keras.layers.Dense(64, activation="relu")(out)
        out = tf.keras.layers.Dense(64, activation="relu")(out)
        net_out = tf.keras.layers.Dense(1)(out)
        # Outputs single value for give state-action
        model = tf.keras.models.Model(inputs=state_input, outputs=net_out)
        model.summary()
        return model

    def train(self, state_batch, disc_rewards):
        with tf.GradientTape() as tape:
            critic_weights = self.model.trainable_variables
            critic_value = tf.squeeze(self.model(state_batch))
            critic_loss = tf.math.reduce_mean(
                tf.square(disc_rewards - critic_value))
            critic_grad = tape.gradient(critic_loss, critic_weights)
            if self.grad_clip is not None:
                critic_grad = [tf.clip_by_value(grad, \
                    -1.0 * self.grad_clip, self.grad_clip) \
                    for grad in critic_grad]
            # outside the gradient tape
            self.optimizer.apply_gradients(zip(critic_grad, critic_weights))
```

```
        return critic_loss.numpy()
```

Listing 6.45: Python code for PPO Critic Class

PPO Agent Class

The Python code for implementing PPO Agent class is provided in the code Listing 6.46. The PPO agent class implements actor-critic architecture for implementation of the PPO algorithm. It creates an actor and a critic as an object of the two classes defined above. The `policy` method samples an action from a normal distribution using the mean and standard deviation estimated with the actor network. The `train` functions divides the experience trajectories into batches to compute discounted returns and advantages. The function `compute_advantage` estimates the advantage using the generalized advantage estimator (GAE) equation (6.52). The discounted returns is used by the critic network to compute TD error needed for training. The advantage is used for training the actor network.

```
class PPOAgent:
    def __init__(self, obs_shape, action_size, batch_size,
                 action_upper_bound=1.0,
                 lr_a=1e-3, lr_c=1e-3,
                 gamma=0.99,             # discount factor
                 lmbda=0.5,             # required for GAE
                 beta=0.01,              # KL penalty coefficient
                 epsilon=0.2,            # action clip boundary
                 kl_target=0.01,         # required for KL penalty method
                 entropy_coeff=0.01,     # entropy coefficient
                 c_loss_coeff=0.01,      # critic loss coefficient
                 grad_clip=None,
                 method='clip',          # choose between 'clip' & 'penalty'
                 actor_model=None,
                 critic_model=None):
        self.name='ppo'
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.actor_lr = lr_a
        self.critic_lr = lr_c
        self.batch_size = batch_size
        self.gamma = gamma # discount factor
        self.action_upper_bound = action_upper_bound
        self.epsilon = epsilon # clip boundary for prob ratio
        self.lmbda = lmbda # required for GAE
        self.initial_beta = beta # required for penalty method
```

```
        self.kl_target = kl_target # required for updating beta
        self.method = method # choose between 'clip' & 'penalty'
        self.c_loss_coeff = c_loss_coeff
        self.entropy_coeff = entropy_coeff
        self.grad_clip = grad_clip # apply gradient clipping
        # Actor Model
        self.actor = PPOActor(self.obs_shape, self.action_size,
                              learning_rate=self.actor_lr,
                              action_upper_bound=self.action_upper_bound,
                              epsilon=self.epsilon,
                              lmbda=self.lmbda,
                              kl_target=self.kl_target,
                              beta=self.initial_beta,
                              entropy_coeff=self.entropy_coeff,
                              critic_loss_coeff=self.c_loss_coeff,
                              method=self.method,
                              grad_clip=self.grad_clip,
                              model=actor_model)
        # Critic Model
        self.critic = PPOCritic(self.obs_shape, self.action_size,
                                learning_rate=self.critic_lr,
                                gamma=self.gamma,
                                grad_clip=self.grad_clip,
                                model=critic_model)


    def policy(self, state, greedy=False):
        tf_state = tf.expand_dims(tf.convert_to_tensor(state), axis=0)
        mean, std = self.actor(tf_state)
        if greedy:
            action = mean
        else:
            pi = tfp.distributions.Normal(mean, std)
            action = pi.sample()
            action = tf.reshape(action, shape=(self.action_size, ))
        valid_action = tf.clip_by_value(action, -self.action_upper_bound,
                                        self.action_upper_bound)
        return valid_action.numpy()

    def train(self, states, actions, rewards, next_states, dones, epochs=20):
        states = tf.convert_to_tensor(states, dtype=tf.float32)
        next_states = tf.convert_to_tensor(next_states, dtype=tf.float32)
        actions = tf.convert_to_tensor(actions, dtype=tf.float32)
        rewards = tf.convert_to_tensor(rewards, dtype=tf.float32)
```

```
                dones = tf.convert_to_tensor(dones, dtype=tf.float32)
                # compute advantage & discounted returns
                target_values, advantages = self.compute_advantages(
                    states, rewards, next_states, dones)
                target_values = tf.convert_to_tensor(target_values, dtype=tf.float32)
                advantages = tf.convert_to_tensor(advantages, dtype=tf.float32)
                # current action probability distribution
                mean, std = self.actor(states)
                pi = tfp.distributions.Normal(mean, std)
                n_split = len(rewards) // self.batch_size
                assert n_split > 0, 'buffer length must be greater than batch_size'
                indexes = np.arange(n_split, dtype=int)
                # training
                a_loss_list, c_loss_list, kl_list = [], [], []
                for _ in range(epochs):
                    np.random.shuffle(indexes)
                    for i in indexes:
                        old_pi = pi[i * self.batch_size: (i+1) * self.batch_size]
                        s_split = tf.gather(states, indices=np.arange(
                            i * self.batch_size, (i+1) * self.batch_size), axis=0)
                        a_split = tf.gather(actions, indices=np.arange(
                            i * self.batch_size, (i+1) * self.batch_size), axis=0)
                        tv_split = tf.gather(target_values, indices=np.arange(
                            i * self.batch_size, (i+1) * self.batch_size), axis=0)
                        adv_split = tf.gather(advantages, indices=np.arange(
                            i * self.batch_size, (i+1) * self.batch_size), axis=0)
                        # update critic
                        cl = self.critic.train(s_split, tv_split)
                        c_loss_list.append(cl)
                        # update actor
                        al = self.actor.train(s_split, a_split, \
                            adv_split, old_pi, cl)
                        a_loss_list.append(al)
                        kl_list.append(self.actor.kl_value)
                    # update lambda once in each epoch
                    if self.method == 'penalty':
                        self.actor.update_beta()
                # end of epoch loop
                actor_loss = np.mean(a_loss_list)
                critic_loss = np.mean(c_loss_list)
                kld_mean = np.mean(kl_list)
                return actor_loss, critic_loss, kld_mean

            def compute_advantages(self, states, rewards, next_states, dones):
```

```
        # input/output are tensors
        s_values = self.critic(states)
        ns_values = self.critic(next_states)
        # advantage should have same shape as that of values
        adv = np.zeros_like(s_values)
        returns = np.zeros_like(s_values)
        discount = self.gamma
        lmbda = self.lmbda
        returns_current = ns_values[-1] # last value
        g = 0 # GAE
        for i in reversed(range(len(rewards))):
            gamma = discount * (1. - dones[i])
            td_error = rewards[i] + gamma * ns_values[i] - s_values[i]
            g = td_error + gamma * lmbda * g
            returns_current = rewards[i] + gamma * returns_current
            adv[i] = g
            returns[i] = returns_current
        adv = (adv - np.mean(adv)) / (np.std(adv) + 1e-10)
        return returns, adv

    @property
    def penalty_coefficient(self):
        # returns penalty coefficienty
        return self.actor.beta
```

Listing 6.46: Python Class definition for PPO Agent

Training a PPO agent

The function for training a PPO agent on a given Gym environment is provided in the code Listing 6.47. The function `ppo_train` first uses the function `collect_trajectories` to generate trajectories with the agent's current policy. These trajectories constitute a `season` which is repeated for a number of times until desired performance is achieved. The trajectories collected in each season is used for training the PPO agent.

```
def collect_trajectories(env, agent, tmax=1000, max_steps=200):
    states, next_states, actions = [], [], []
    rewards, dones = [], []
    ep_count = 0        # episode count
    state = env.reset()[0]
    step = 0
    for t in range(tmax):
        step += 1
```

```
            action = agent.policy(state)
            next_state, reward, done, _, _ = env.step(action)
            states.append(state)
            actions.append(action)
            next_states.append(next_state)
            rewards.append(reward)
            dones.append(done)
            state = next_state
            if max_steps is not None and step > max_steps:
                done = True
            if done:
                ep_count += 1
                state = env.reset()[0]
                step = 0
    return states, actions, rewards, next_states, dones, ep_count

def ppo_train(env, agent, max_buffer_len=1000, max_seasons=100, epochs=20,
              max_steps=None, stop_score=None):
    print('Environment name: ', env.spec.name)
    print('RL Agent name:', agent.name)
    best_score = -np.inf
    season_scores = []
    total_ep_cnt = 0
    for s in range(max_seasons):
        # collect trajectories
        states, actions, rewards, next_states, dones, ep_count = \
            collect_trajectories(env, agent, tmax=max_buffer_len, \
                                 max_steps=max_steps)
        total_ep_cnt += (ep_count+1)
        # train the agent
        a_loss, c_loss, kld_value = agent.train(states, actions, rewards,
                                                next_states, dones, epochs=epochs)
        season_score = np.sum(rewards, axis=0) / (ep_count + 1)
        season_scores.append(season_score)
        if season_score > best_score:
            best_score = season_score
            agent.save_weights()
        if stop_score is not None and season_score > stop_score:
            print(f'Problem is solved in {s} seasons\
                            or {total_ep_cnt} episodes.' )
            break
        print(f'season: {s}, episodes: {total_ep_cnt}, \
                season_score: {season_score:.2f},\
                avg_ep_reward: {np.mean(season_scores):.2f},\
```

```
                best_score: {best_score:.2f}')
```

Listing 6.47: Python code for training a PPO agent on a given Gym environment

6.7.5. Solving Pendulum environment using PPO

The code for solving `Pendulum-v1` environment is shown in the code Listing 6.48. The pendulum environment has a continuous action space along with a continuous observation space. The problem is solved in about 750 episodes when the average of last 100 episodes exceeds -200. The performance of the algorithm is sensitive to choices of the user-defined parameters. The resulting training performance is shown in Figure 6.8. This plot is generated by using WandB.

```
import gymnasium as gym
env = gym.make('Pendulum-v1')
obs_shape = env.observation_space.shape
action_shape = env.action_space.shape
action_size = action_shape[0]
action_ub = env.action_space.high
action_lb = env.action_space.low
print('Observation shape: ', obs_shape)
print('Action shape: ', action_shape)
print('Max episodic Steps: ', env.spec.max_episode_steps)
print('Action space bounds: ', (action_ub[0], action_lb[0]))

# create a ppo agent
ppo_agent = PPOAgent(obs_shape, action_size,
                     batch_size=200],
                     action_upper_bound=action_ub,
                     entropy_coeff=0.0,
                     c_loss_coeff=0.0,
                     kl_target=0.01,
                     gamma=0.99, beta=0.01,
                     epsilon=0.1, lmbda=0.95,
                    grad_clip=None,
                    method='clip')

# train the agent
ppo_train(env, ppo_agent, max_buffer_len=10000,
          max_seasons=100, epochs=20, max_steps=200, stop_score=200)
```

[1] http://wandb.ai

```
Output:
season: 0, episodes: 50, season_score: -1285.45, avg_ep_reward: -1285.45, best_score: -1285.45
season: 1, episodes: 100, season_score: -1181.49, avg_ep_reward: -1233.47, best_score: -1181.49
season: 2, episodes: 150, season_score: -1110.28, avg_ep_reward: -1192.40, best_score: -1110.28
season: 3, episodes: 200, season_score: -1064.96, avg_ep_reward: -1160.54, best_score: -1064.96
season: 4, episodes: 250, season_score: -1060.86, avg_ep_reward: -1140.61, best_score: -1060.86
season: 5, episodes: 300, season_score: -982.23, avg_ep_reward: -1114.21, best_score: -982.23
season: 6, episodes: 350, season_score: -946.48, avg_ep_reward: -1090.25, best_score: -946.48
season: 7, episodes: 400, season_score: -853.93, avg_ep_reward: -1060.71, best_score: -853.93
season: 8, episodes: 450, season_score: -831.43, avg_ep_reward: -1035.23, best_score: -831.43
season: 9, episodes: 500, season_score: -676.64, avg_ep_reward: -999.38, best_score: -676.64
season: 10, episodes: 550, season_score: -521.00, avg_ep_reward: -955.89, best_score: -521.00
season: 11, episodes: 600, season_score: -379.70, avg_ep_reward: -907.87, best_score: -379.70
season: 12, episodes: 650, season_score: -299.90, avg_ep_reward: -861.10, best_score: -299.90
season: 13, episodes: 700, season_score: -234.21, avg_ep_reward: -816.33, best_score: -234.21
season: 14, episodes: 750, season_score: -224.16, avg_ep_reward: -776.85, best_score: -224.16
Problem is solved in 15 seasons or 800 episodes.
```

Listing 6.48: Python Code for solving `Pendulum-v1` environment by using a PPO agent.

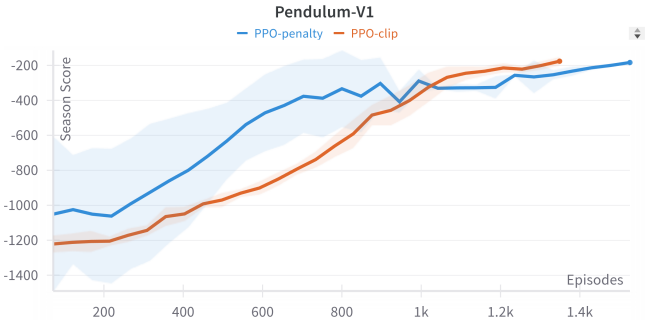


Figure 6.8.: PPO training performance for `Pendulum-v1` environment

6.7.6. Solving LunarLander-v2 Continuous with PPO

```python
import gymnasium as gym
env = gym.make('LunarLander-v2', continuous=True)
obs_shape = env.observation_space.shape
action_shape = env.action_space.shape
action_size = action_shape[0]
action_ub = env.action_space.high
action_lb = env.action_space.low
print('environment name: ', env.spec.name)
print('Observation shape: ', obs_shape)
print('Action shape: ', action_shape)
print('Max episodic Steps: ', env.spec.max_episode_steps)
print('Action space bounds: ', (action_ub, action_lb))
```

```
def create_actor_model(obs_shape, n_actions):
    s_input = tf.keras.layers.Input(shape=obs_shape)
    x = tf.keras.layers.Dense(512, activation='relu')(s_input)
    x = tf.keras.layers.Dense(512, activation='relu')(x)
    a = tf.keras.layers.Dense(n_actions, activation='tanh')(x)
    model = tf.keras.models.Model(s_input, a, name='actor_network')
    model.summary()
    return model

def create_critic_model(obs_shape, n_actions):
    s_input = tf.keras.layers.Input(shape=obs_shape)
    x = tf.keras.layers.Dense(512, activation='relu')(s_input)
    x = tf.keras.layers.Dense(512, activation='relu')(x)
    v = tf.keras.layers.Dense(1, activation=None)(x)
    model = tf.keras.models.Model(s_input, v, name='critic_network')
    model.summary()
    return model

a_model = create_actor_model(obs_shape, action_size)
c_model = create_critic_model(obs_shape, action_size)

CFG = dict(
    batch_size=200,
    entropy_coeff = 0.0,    # required for CLIP method
    c_loss_coeff = 0.0,     # required for CLIP method
    grad_clip = None,
    method = 'clip', # choose between 'clip' or 'penalty'
    kl_target = 0.01, # required for penalty method
    beta = 0.01,  # required for penalty method
    epsilon = 0.3,  # required for clip method
    gamma = 0.99,
    lam = 0.95,      # used for GAE
    buffer_capacity = 20000, # next try with 50000
    lr_a = 1e-3,
    lr_c = 1e-3,
    training_epochs=20,
)

agent = PPOAgent(obs_shape, action_size,
                 batch_size=20000,
                 action_upper_bound=action_ub,
                 entropy_coeff=0.0,
                 c_loss_coeff=0.0,
```

```
                    kl_target=0.01,
                    gamma=0.99, beta=0.01,
                    epsilon=0.3, lmbda=0.95,
                grad_clip=None, method='clip',
                lr_a=1e-3, lr_c=1e-3,
                actor_model=a_model,
                critic_model=c_model)

ppo_train(env, ppo_agent, max_buffer_len=CFG['buffer_capacity'],
          max_seasons=500,
          epochs=CFG['batch_size'],
          stop_score=200, max_steps=200,
          wandb_log=True)
```

Listing 6.49: Solving LunarLander-v2 with PPO

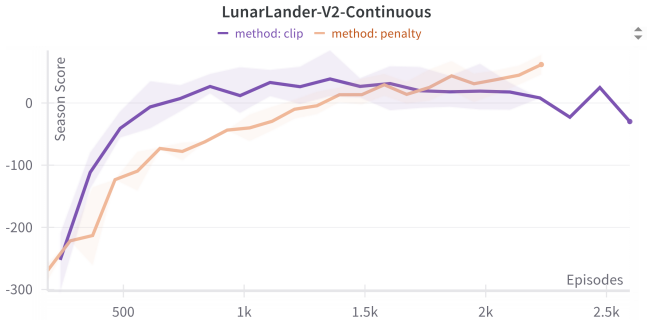


Figure 6.9.: PPO training performance for `LunarLander-v2-Continuous` environment

6.8. Summary

Policy gradient (PG) methods directly learn the policy function by maximizing a given objective function (cumulative reward function) and does not necessarily require learning Q function first as is the case with value-based methods. Usually the policy parameters are updated by using gradient-ascent method that requires computing the gradient of the objective function with respect to the policy parameters. Since, PG methods do not require Q function and use a gradient-based approach, it can be applied to solve problems continuous action spaces which was not possible with value-based methods. Then, we discuss a Monte-Carlo method called REINFORCE that updates the policy parameters only once in each episode by computing the return $G(t)$ function at the end of the episode (see (3.3)). This makes this approach *sample-inefficient*. This can be overcome by using an *actor-critic* approach that uses a separate Q function estimator instead of computing

return thereby allowing parameter update at each step of the episode (see (6.21)) More details of this approach will be discussed in the next chapter. Then, we discuss a popular policy gradient approach called DDPG that uses actor-critic model to concurrently learn a Q function and a policy. The critic network uses off-policy Bellman equation to learn Q function and the actor network learns the optimal policy by maximizing the output of the critic network. In other words DDPG extends Deep Q learning to continuous action spaces. DDPG algorithms can be sometimes unstable as no constraint is put on the value of the gradient being computed. Secondly, in some cases, it is not practical to explore all kinds of actions including bad ones to learn the optimal policy. TRPO methods avoids this by restricting policy updates to a safe region knows as *trust-regions*. This trust region is defined by imposing a constraint on the KL divergence between the old and new policy. TRPO uses a complex approach to optimize policy by using a surrogate objective function while imposing this constraint. This is simplifed in PPO where the constraint is converted into a penalty term. The efficacy of these approaches are demonstrated by solving several problems with continuous spaces, namely, Pendulum and LunarLander.

7. Actor-Critic Models

Actor-Critic (AC) models are a class of reinforcement learning (RL) algorithms that combine the strengths of both policy-based (actor) and value-based (critic) methods. This hybrid approach aims to address the limitations of using each method individually. It has two components, namely, actor and critic. The actor is responsible for selecting actions based on the current state. It learns a policy function, often parameterized by θ, which maps states to actions or a probability distribution over actions. The actor's goal is to improve its policy to maximize the expected cumulative reward. The critic's role is to evaluate the actions taken by the actor. It learns a value function, typically parameterized by ϕ, which estimates the expected future rewards for a given state (state-value function, $V(s)$) or state-action pair (action-value function, $Q(s, a)$). The critic provides feedback to the actor on how "good" its actions were.

Actor-critic models are motivated by the desire to combine the advantages of policy-based and value-based methods while mitigating their individual limitations. Policy-based methods (e.g. REINFORCE) can handle continuous action spaces, learn stochastic policies (which is useful for exploration and in non-deterministic environments), and can directly optimize the policy. However, they suffers from high variance in gradient estimates because they rely on full episode returns. This high variance can lead to unstable training and slow convergence. On the other hand, value-based methods (e.g. Q-learning, SARSA) generally have low variance due to bootstrapping (using estimated values to update other estimated values). They are often more sample-efficient. However, these methods can only be applied to problems with discrete action spaces and typically learn deterministic policies. Actor-critic methods aim to gain the best of both worlds: the ability to handle continuous action spaces and learn stochastic policies (from the actor) combined with the reduced variance and improved stability from bootstrapping with a learned value function (from the critic).

The use of actor-critic model in implementing algorithms such DDPG and PPO have already been discussed in the previous chapter. In this chapter, we will discuss the common architectures in more details.

7.1. Naive Actor-Critic Model

It is a simple version of actor-critic methods where the critic estimates the value function $V(s; w)$, and the actor updates the policy using the critic's feedback, typically via policy gradients. The actor aims to maximize the expected cumulative reward, given by:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_0] \tag{7.1}$$

where G_0 is the total discounted return from the initial state with:

$$G_t = \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right] \tag{7.2}$$

Here, $\gamma \in [0, 1]$ is the discount factor, and R_t is the reward received at time t. The actor updates its policy parameters θ by performing gradient ascent on this objective function. The Policy Gradient Theorem provides the fundamental basis for this update. A practical form of the gradient of $J(\theta)$ used for updating policy parameters is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a)] \tag{7.3}$$

7.1.1. Temporal Difference (TD) Error

In the Naive Actor-Critic algorithm, we approximate $Q^\pi(s, a)$ with the critic's estimate $Q_w(s, a)$ (or $V_w(s)$) along with the immediate reward. The critic learns its value function by minimizing the difference between its current estimate and a more "bootstrapped" estimate of the value. This difference is known as the Temporal Difference (TD) error. For a state-value function $V_w(s)$, the one-step TD error is:

$$\delta_t = R_{t+1} + \gamma V_w(S_{t+1}) - V_w(S_t) \tag{7.4}$$

where S_t is the state at time t, A_t is the action taken, and R_{t+1} is the reward received after taking action A_t and transitioning to S_{t+1}.

If the critic estimates an action-value function $Q_w(s, a)$, the TD error (similar to SARSA) would be:

$$\delta_t = R_{t+1} + \gamma Q_w(S_{t+1}, A_{t+1}) - Q_w(S_t, A_t) \tag{7.5}$$

The TD error serves as the signal for both the critic and the actor.

7.1.2. Advantage Function

To improve the stability and performance of policy gradient methods, the concept of an advantage function is often used. The advantage function $A^\pi(s, a)$ measures how much better an action a is compared to the average value of the state s under policy π:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{7.6}$$

When using the advantage function, the policy gradient update becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) A^\pi(s, a)] \tag{7.7}$$

In the Naive Actor-Critic, the TD error itself can be seen as an estimate of the advantage function. This is because $R_{t+1} + \gamma V_w(S_{t+1})$ can be viewed as a noisy estimate of $Q_w(S_t, A_t)$ for the taken action, and $V_w(S_t)$ is the baseline. So, $\delta_t \approx Q_w(S_t, A_t) - V_w(S_t)$. The pseudocode for naive actor-critic algorithm is provided in Table 7.1.

Algorithm 7.1 Actor-Critic Algorithm

input: Policy parameters θ, value function parameters w. Learning rates α_θ, α_w, discount factor γ.
for each episode **do**
  Initialize s_t
  while episode not done **do**
    Sample action $a_t \sim \pi(a|s;\theta)$
    Take action a_t, obtain reward r_t, transition to new state s_{t+1}
    Compute TD error: $\delta_t = R_{t+1} + \gamma V(S_{t+1};w) - V(s_t;w)$
    Update Critic: $w \leftarrow w + \alpha_w \delta_t \nabla_w V(s_t;w)$ ▷ Critic Update
    Update Actor: $\theta \leftarrow \theta + \alpha_\theta \delta_t \nabla_\theta \log \pi(a|s;\theta)$ ▷ Actor Update
    $s_t \leftarrow s_{t+1}$
  end while
end for

7.1.3. Advantages of Actor-Critic Algorithm

Actor critic methods can provide the following advantages in general:

- Reduced Variance: Compared to pure policy gradient methods (like REINFORCE), the critic's value estimate acts as a learned baseline, significantly reducing the variance of the policy gradient estimate.
- Continuous Action Spaces: Actor-critic methods can easily handle continuous action spaces by having the actor output parameters of a continuous probability distribution.
- On-policy Learning: The critic learns about the policy currently being followed by the actor, providing more relevant feedback.

7.1.4. Limitations of Naive Actor-Critic Algorithm

- Bias from Critic Approximation: If the critic's value estimate is inaccurate, it can introduce bias into the policy gradient, leading to suboptimal policies.
- Convergence Issues: The interaction between the actor and critic can be unstable, sometimes leading to convergence problems.
- Sample Inefficiency: While better than pure policy gradient, it can still be sample-inefficient compared to off-policy methods.

The "naive" actor-critic algorithm generally uses a $Q(s,a)$ or $V(s)$ estimate directly for actor update as given below:

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi(a|s;\theta) V(s_t;w) \tag{7.8}$$

This, however, can be quite unstable and may exhibit high variance. Therefore, TD error is used for updating actor parameters to reduce the variance. The resulting actor update equation is given by

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi(a|s;\theta)\delta_t \tag{7.9}$$

7.1.5. Python Code for implementing Actor-Critic Algorithm

The python code for creating actor and critic classes are provided in the code listing 7.50 and 7.51 respectively. This code can be used to solve problems with discrete action spaces only. The actor and critic classes use a sequential deep network to estimate policy $\pi(s)$ and value function $V(s)$ respectively. The actor network returns the action probabilities for a given input state s. The actor class also provides a method to computer actor loss from the TD error and action log probabilities as explained in Section 7.1.2. The negative sign in the loss function indicates that the actor is optimized by maximizing this loss function. The code for the actor-critic agent is provided in the code listing 7.52. It provides the `train()` method to update both policy and critic parameters after each step of episode.

```
class Actor():
    def __init__(self, obs_shape, action_size, lr=1e-4, model=None):
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.lr = lr
        if model is None:
        self.model = self._build_model()
        else:
        self.model = model
        self.optimizer = tf.keras.optimizers.Adam(
                learning_rate=self.lr)


    def _build_model(self): # outputs action probabilities
        sinput = tf.keras.layers.Input(shape=self.obs_shape)
        x = tf.keras.layers.Dense(512, activation='relu',
           kernel_initializer=tf.keras.initializers.HeUniform())(sinput)
        x = tf.keras.layers.Dense(512, activation='relu',
           kernel_initializer=tf.keras.initializers.HeUniform())(x)
        aout = tf.keras.layers.Dense(self.action_size,
                activation='softmax',
           kernel_initializer=tf.keras.initializers.HeUniform())(x)
        model = tf.keras.models.Model(sinput, aout, name='actor')
        model.summary()
        return model
```

```
    def __call__(self, states):
        # returns action probabilities for each state
        pi = tf.squeeze(self.model(states))
        return pi

    def compute_actor_loss(self, states, actions, td_error):
        pi = self.model(states)
        action_dist = tfp.distributions.Categorical(
└└└└└└└└probs=pi, dtype=tf.float32)
        log_prob = action_dist.log_prob(actions)
        actor_loss = -log_prob * td_error
        return tf.math.reduce_mean(actor_loss)
```

Listing 7.50: Python code for creating Actor Class

```
class Critic():
    def __init__(self, obs_shape,
    └└└└└└└└lr = 1e-4, gamma=0.99, model=None):
        self.obs_shape = obs_shape
        self.gamma = gamma
        self.lr = lr
        if model is None:
        └└└self.model = self._build_model()
        else:
        └└└self.model = model
    └└└└self.optimizer = tf.keras.optimizers.Adam(learning_rate=self.lr)


    def _build_model(self): # returns V(s)
        sinput = tf.keras.layers.Input(shape=self.obs_shape)
        x = tf.keras.layers.Dense(128, activation='relu')(sinput)
        x = tf.keras.layers.Dense(256, activation='relu')(x)
        x = tf.keras.layers.Dense(256, activation='relu')(x)
        vout = tf.keras.layers.Dense(1, activation='relu')(x)
        model = tf.keras.models.Model(inputs= sinput, outputs=vout,
        └└└└└└└└└└└└└└└└└└└└name='critic')
        model.summary()
    └└└└return model

    def __call__(self, states):
        # returns V(s) for each state
        value = tf.squeeze(self.model(states))
    └└└└return value
```

Listing 7.51: Python code for creating Critic Class

```
class ACAgent():
    def __init__(self, obs_shape, action_size,
                 lr_a=1e-4, lr_c=1e-4, gamma=0.99,
                 a_model=None, c_model=None):
        self.obs_shape = obs_shape
        self.action_size = action_size
        self.gamma = gamma
        self.lr_a = lr_a
        self.lr_c = lr_c
        self.name = 'actor-critic'

        # actor model
        self.actor = Actor(self.obs_shape, self.action_size,
                           lr=self.lr_a, model=a_model)
        # critic model
        self.critic = Critic(self.obs_shape, lr=self.lr_c,
                             gamma=self.gamma, model=c_model)

    def policy(self, state):
        state = tf.expand_dims(
            tf.convert_to_tensor(state, dtype=tf.float32), axis=0)
        pi = self.actor(state) # action probabilities
        action_dist = tfp.distributions.Categorical(
                      probs=pi, dtype=tf.float32)
        action = action_dist.sample()
        return int(action.numpy())


    def train(self, state, action, reward, next_state, done):
        state = tf.expand_dims(
            tf.convert_to_tensor(state, dtype=tf.float32), axis=0)
        action= tf.convert_to_tensor(action, dtype=tf.float32)
        reward = tf.convert_to_tensor(reward, dtype=tf.float32)
        next_state = tf.expand_dims(
                     tf.convert_to_tensor(next_state, dtype=tf.float32),
                     axis=0)
        done = tf.convert_to_tensor(done, dtype=tf.float32)
        with tf.GradientTape() as tape1, tf.GradientTape() as tape2:
            value = self.critic(state)
            next_value = self.critic(next_state)
            td_target = reward + self.gamma * next_value * (1 - done)
            td_error = td_target - value
```

```
            a_loss = self.actor.compute_actor_loss(state, action, td_error)
            c_loss = tf.math.reduce_mean(tf.square(td_target - value))

        actor_grads = tape1.gradient(a_loss,
                                     self.actor.model.trainable_variables)
        critic_grads = tape2.gradient(c_loss,
                                      self.critic.model.trainable_variables)
        self.actor.optimizer.apply_gradients(zip(actor_grads,
                                                 self.actor.model.trainable_variables))
        self.critic.optimizer.apply_gradients(zip(critic_grads,
                                                  self.critic.model.trainable_variables))
    return a_loss, c_loss
```

Listing 7.52: Python Code for Actor-Critic Agent Class

Solving CartPole problem using Naive Actor-Critic Algorithm

The Python function for training a Gym environment problem using a naive actor-critic agent is provided in code Listing 7.53. The main function that uses this train function to solve `CartPole-v1` environment is provided in code Listing 7.54 along with the console output. The corresponding training performance plot is shown in Figure 7.1. It can be seen that the episodic reward is steadily increasing with training episodes. This problem is considered solved when the episodic reward crosses 499. So, the problem gets solved in about 400 episodes with best score graph crossing this mark.

```
def train(env, agent, max_episodes=10000, log_freq=50,
          min_score=-200, max_score=200,
          stop_score=200):

    print('Environment name: ', env.spec.id)
    print('RL Agent name:', agent.name)

    assert isinstance(env.action_space, gym.spaces.Discrete),\
           "AC Agent only for discrete action spaces"

    ep_scores = []
    best_score = -np.inf
    for e in range(max_episodes):
        done = False
        state = env.reset()[0]
        ep_score = 0
        a_losses, c_losses = [], []
        while not done:
```

```
            action = agent.policy(state)
            next_state, reward, done, _, _ = env.step(action)
            a_loss, c_loss = agent.train(state, action,
                                         reward, next_state, done)
            a_losses.append(a_loss)
            c_losses.append(c_loss)
            state = next_state
            ep_score += reward
        # while loop ends here
        ep_scores.append(ep_score)

        if e % log_freq == 0:
            print(f'e:{e}, ep_score:{ep_score:.2f},
                  avg_ep_score:{np.mean(ep_scores):.2f},\
                    avg100score:{np.mean(ep_scores[-100:]):.2f}, \
                      best_score:{best_score:.2f}')

        if ep_score > best_score:
            best_score = ep_score
            agent.save_weights()
            print(f'Best Score: {ep_score}, episode: {e}. Model saved.')

        if np.mean(ep_scores[-100:]) > stop_score:
            print('The problem is solved in {} episodes'.format(e))
            break
    # for loop ends here
```

Listing 7.53: Function for training an actor-critic agent for a given Gym environment

```
import gymnasium as gym
from actor_critic import ACAgent
if __name__ == '__main__':
    env = gym.make('CartPole-v1')
    obs_shape = env.observation_space.shape
    action_size = env.action_space.n

    print("Observation shape: ", obs_shape)
    print("Action Size: ", action_size)
    print("Max Episode steps: ", env.spec.max_episode_steps)

    # create an RL agent
    agent = ACAgent(obs_shape, action_size)
```

```
    # train the RL agent on
    train(env, agent, max_episodes=1500, log_freq=100,
          stop_score=499)
```

```
Output:
Best score:67.0, episode:0. Model Saved!
e:0, ep_score:67.00, avg_ep_score:67.00, avg100score:67.00, best_score:67.00
Best score:96.0, episode:29. Model Saved!
e:100, ep_score:38.00, avg_ep_score:34.75, avg100score:34.43, best_score:96.00
Best score:190.0, episode:129. Model Saved!
e:200, ep_score:46.00, avg_ep_score:46.21, avg100score:57.78, best_score:190.00
Best score:199.0, episode:216. Model Saved!
Best score:243.0, episode:234. Model Saved!
Best score:347.0, episode:263. Model Saved!
Best score:387.0, episode:274. Model Saved!
e:300, ep_score:124.00, avg_ep_score:76.65, avg100score:137.85, best_score:387.00
Best score:863.0, episode:316. Model Saved!
e:400, ep_score:134.00, avg_ep_score:102.75, avg100score:181.29, best_score:863.00
e:500, ep_score:201.00, avg_ep_score:125.70, avg100score:217.75, best_score:863.00
e:600, ep_score:130.00, avg_ep_score:141.93, avg100score:223.20, best_score:863.00
e:700, ep_score:200.00, avg_ep_score:159.75, avg100score:266.90, best_score:863.00
```

Listing 7.54: Python code for training an actor-critic agent to solve the `CartPole-v1` problem.

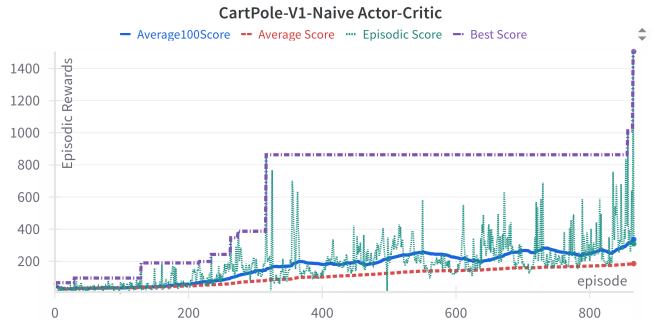


Figure 7.1.: Performance of Naive Actor Critic Algorithm in solving `CartPole-v1` problem.

Solving Lunar Lander Problem using Naive Actor-Critic Algorithm

```
import gymnasium as gym
from actor_critic import ACAgent
if __name__ == '__main__':
```

```
    env = gym.make('LunarLander-v3')
    obs_shape = env.observation_space.shape
    action_size = env.action_space.n

    print("Observation shape: ", obs_shape)
    print("Action Size: ", action_size)
    print("Max Episode steps: ", env.spec.max_episode_steps)

    # create an RL agent
    agent = ACAgent(obs_shape, action_size)

    # train the RL agent on
    train(env, agent, max_episodes=1500, min_score=-500, log_freq=100,
          stop_score=200)
```

```
Output:
provide console output
```

Listing 7.55: Python code for training a actor-critic agent to solve the `LunarLander-v3` problem.

7.2. Advantage Actor-Critic (A2C) Algorithm

In Advantage actor-critic algorithm, Advantage function $A(s, a)$ is used directly to update the actor parameters instead of using raw TD error values which is more noisy. If TD error is used as an estimate of the advantage function as done in the previous section, there is no particular difference between A2C and naive actor-critic algorithm discussed above. To encourage exploration and prevent premature convergence to suboptimal policies, an entropy term is often added to the actor loss given by:

$$H(\pi_\theta) = -\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t) \tag{7.10}$$

Therefore the combined actor loss becomes:

$$L_{actor}(\theta) = -\log \pi_\theta(a_t|s_t).A(s_t, a_t) - \beta H(\pi_\theta) \tag{7.11}$$

Rather than updating the actor and critic parameters after each episode, the training updates can be carried out in a batch mode by collecting trajectories for a fixed number of time steps. The pseudocode of A2C algorithm is provided in Algorithm 7.2.

Algorithm 7.2 Advantage Actor-Critic (A2C) Algorithm

input: Policy parameters θ, value function parameters w. Learning rates α_θ, α_w, discount factor γ and entropy coefficient β.
for each iterative step **do**
  Collect trajectories with transitions $(s_t, a_t, r_t, s_{t+1}, d)$ for $t = 0, 1, \ldots, T$ with $a_t \sim \pi(s_t; \theta)$.
  for each trajectory (or episode) **do**
    Compute Advantage:

$$A(s_t, a_t) = r_{t+1} + \gamma V(s_{t+1}; w) * (1 - d) - V(s_t; w) \tag{7.12}$$

▷ $A(s_t, a_t) \approx \delta_t$, TD error

    Update Critic:

$$w \leftarrow w + \alpha_w \nabla_w V(s_t; w) A(s, a) \tag{7.13}$$

▷ Critic minimises TD error

    Update Actor:

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi(a|s; \theta) A(s, a) + \beta H(\pi_\theta(s_t)) \tag{7.14}$$

▷ maximises actor loss function (7.11)

  end for
end for

7.2.1. Python Implementation of A2C Algorithm

The `A2CAgent` class implements the algorithm shown in the Algorithm Table 7.2. The actor and critic classes remain same as with the previous naive actor-critic implementation. The method `compute_advantages()` uses TD error to estimate the advantage function from the state transition information $\{s_t, a_t, r_t, s_{t+1}, d\}$ collected during exploration as shown in equation (7.12). The method `compute_actor_loss()` computes the combined loss function given by equation (7.11). This function uses `tensorflow_probability` package to compute log probability and entropy values. The `train()` method computes advantages, actor and critic loss functions for each episode and updates the actor and critic parameters using built-in optimizer. The negative sign with the actor loss function indicates that the actor loss is maximized contrast to the critic loss which is minimized. The complete python code for implementing `A2CAgent` class is provided in the listing 7.56.

It is observed that computing TD error as the difference between the discounted return and the current value estimate provides better performance compared to the standard method that requires storing both s_t and s_{t+1}. The method to compute discounted rewards and actor loss function is provided in the code listing 7.57.

The method `a2c_train()` provided in code listing 7.58 shows how one can train an A2C agent on a given gym environment. As we can see in this code, the agent is trained after each episode indicating a batch mode of training unlike the implementation in the previous section that updated the models after each step.

```
import tensorflow_probability as tfp
class A2CAgent:
    def __init__(self, obs_shape, action_size,
                 lr_a=1e-4, lr_c=1e-3, gamma=0.99,
                 entropy_beta=0.01, grad_clip_norm=5.0,
```

```
                a_model=None, c_model=None):
        self.lr_a = lr_a
        self.lr_c = lr_c
        self.entropy_beta = entropy_beta
        self.grad_clip_norm = grad_clip_norm

        self.gamma = gamma
        self.action_size = action_size
        self.obs_shape = obs_shape
        self.name = 'A2C_v2'

        # create actor and critic networks
        self.actor = Actor(obs_shape, action_size, lr=self.lr_a,
                           model=a_model)
        self.critic = Critic(obs_shape, lr=self.lr_c,
                             gamma=self.gamma, model=c_model)

    def policy(self, state):
        state = tf.expand_dims(
            tf.convert_to_tensor(state, dtype=tf.float32),
            axis=0)
        pi = self.actor(state)
        pi_np = pi.numpy()
        action_probs = tfp.distributions.Categorical(probs=pi)
        action = action_probs.sample()
        return action.numpy()

    def compute_advantages(self, rewards, values, next_values, dones):
        advantages = rewards + \
            self.gamma * next_values * (1.0 - dones) - values
        # normalize advantages
        advantages = (advantages - np.mean(advantages))
                     / (np.std(advantages) + 1e-8)
        return advantages

    def compute_actor_loss(self, states, actions, advantages):
        probs = self.actor(states)
        action_dist = tfp.distributions.Categorical(
                      probs=probs, dtype=tf.float32)
        log_probs = action_dist.log_prob(actions)
        actor_loss = -tf.reduce_mean(log_probs * advantages)

        # entropy regularization
        entropy = action_dist.entropy()
```

```
        actor_loss -= self.entropy_beta * tf.reduce_mean(entropy)
        return actor_loss


    def train(self, states, actions, rewards, next_states, dones):
        states = np.array(states, dtype=np.float32)
        actions = np.array(actions, dtype=np.int32)
        rewards = np.array(rewards, dtype=np.float32)
        next_states = np.array(next_states, dtype=np.float32)
        dones = np.array(dones, dtype=np.float32)

        with tf.GradientTape() as tape1, tf.GradientTape() as tape2:
            values = self.critic(states)
            next_values = self.critic(next_states)
            advantages = self.compute_advantages(
                rewards, values, next_values, dones)
            actor_loss = self.compute_actor_loss(states,
                actions,advantages)
            critic_loss = tf.reduce_mean(tf.square(advantages))

        # compute gradients
        actor_grads = tape1.gradient(actor_loss,
            self.actor.model.trainable_variables)
        critic_grads = tape2.gradient(critic_loss,
            self.critic.model.trainable_variables)

        # apply gradients to the models
        self.actor.optimizer.apply_gradients(
            zip(actor_grads, self.actor.model.trainable_variables))
        self.critic.optimizer.apply_gradients(
            zip(critic_grads, self.critic.model.trainable_variables))
        return actor_loss.numpy(), critic_loss.numpy()
```

Listing 7.56: Python Code for implementing A2C agent class

```
class A2CAgent():
    def init(self):
        pass

    def policy(self, state):
        pass

    def compute_discounted_rewards(self, states, actions, rewards):
```

```
                discounted_rewards = []
                sum_rewards = 0
                for r in reversed(rewards):
                    sum_rewards = r + self.gamma * sum_rewards
                    discounted_rewards.insert(0, sum_rewards)
                discounted_rewards = np.array(discounted_rewards)
                states = np.array(states, dtype=np.float32)
                actions = np.array(actions, dtype=np.int32)
                return states, actions, discounted_rewards


    def compute_actor_loss(self, states, actions, td_error):
        probs = self.actor(states)
        action_dist = tfp.distributions.Categorical(
            probs=probs, dtype=tf.float32)
        log_probs = action_dist.log_prob(actions)
        actor_loss = -tf.reduce_mean(log_probs * td_error)
        # entropy regularization
        entropy = action_dist.entropy()
        actor_loss -= self.entropy_beta * tf.reduce_mean(entropy)
        return actor_loss

    def train(self, states, actions, rewards):
        states, actions, discnt_rewards = \
            self.compute_discounted_rewards(states, actions, rewards)
        discnt_rewards = tf.convert_to_tensor(
            discnt_rewards, dtype=tf.float32)

        with tf.GradientTape() as tape1, tf.GradientTape() as tape2:
            values = self.critic(states)
            td_error = tf.math.subtract(discnt_rewards, values)
            actor_loss = self.actor.compute_actor_loss(states,
                                                       actions, td_error)
            critic_loss = tf.reduce_mean(tf.square(td_error))

        actor_grads = tape1.gradient(actor_loss,
                                     self.actor.model.trainable_variables)
        critic_grads = tape2.gradient(critic_loss,
                                      self.critic.model.trainable_variables)

        self.actor.optimizer.apply_gradients(
            zip(actor_grads, self.actor.model.trainable_variables))
        self.critic.optimizer.apply_gradients(
            zip(critic_grads, self.critic.model.trainable_variables))
```

```
        return actor_loss.numpy(), critic_loss.numpy()
```

Listing 7.57: Code for computing TD error from discounted returns

```
def a2c_train(env, agent, max_episodes=10000, log_freq=50,
              max_score=None, min_score=None,
              stop_score=500):
    print('Environment name: ', env.spec.id)
    print('RL Agent name:', agent.name)

    assert isinstance(env.action_space, gym.spaces.Discrete),\
        "A2C Agent only for discrete action spaces"

    ep_scores = []
    best_score = -np.inf
    for e in range(max_episodes):
        states, actions, rewards = [], [], []
        done = False
        state = env.reset()[0]
        ep_score = 0
        while not done:
            action = agent.policy(state)
            next_state, reward, done, _, _ = env.step(action)
            rewards.append(reward)
            states.append(state)
            actions.append(action)
            state = next_state
            ep_score += reward

            if max_score is not None and
                ep_score >= max_score:
                done = True
            if min_score is not None and
                ep_score <= min_score:
                done = True

            if done:
                ep_scores.append(ep_score)
                # train the agent
                a_loss, c_loss = agent.train(states, actions, rewards)

    # while loop ends here
```

```
        if e % log_freq == 0:
            print(f'e:{e}, ep_score:{ep_score:.2f},\
                avg_ep_score:{np.mean(ep_scores):.2f},\
                 avg100score:{np.mean(ep_scores[-100:]):.2f}, \
                  best_score:{best_score:.2f}')

        if ep_score > best_score:
            best_score = ep_score
            agent.save_weights()
            print(f'Best Score: {ep_score}, episode: {e}. Model saved.')

    # for loop ends here
```

Listing 7.58: Training function for an A2C agent

7.2.2. Solving `LunarLander-v3` with A2C algorithm

The main code for training an A2C agent to solve Gymnasium's `LunarLander-v3` problem is provided in the listing 7.59. The actor and critic networks are created externally and passed to the `A2CAgent` object. To speed up training, the episodes are truncated when the total episodic score goes beyond the range $[-300, 500]$. The resulting training performance plot is shown in Figure 7.2. It can be seen that the best score exceeds 200 which is required for solving the problem successfully. However, the running average of last 100 episodes reaches around 0 after about 1500 episode.

```
import gymnasium as gym
# create actor & Critic models
def create_actor_model(obs_shape, n_actions):
    s_input = tf.keras.layers.Input(shape=obs_shape)
    x = tf.keras.layers.Dense(128, activation='relu')(s_input)
    x = tf.keras.layers.Dense(128, activation='relu')(x)
    a = tf.keras.layers.Dense(n_actions, activation='softmax')(x)
    model = tf.keras.models.Model(s_input, a, name='actor_network')
    model.summary()
    return model

def create_critic_model(obs_shape):
    s_input = tf.keras.layers.Input(shape=obs_shape)
    x = tf.keras.layers.Dense(128, activation='relu')(s_input)
    x = tf.keras.layers.Dense(128, activation='relu')(x)
    v = tf.keras.layers.Dense(1, activation=None)(x)
    model = tf.keras.models.Model(s_input, v, name='critic_network')
    model.summary()
```

```
    return model

if __name__ == '__main__':
    # Create Gym environment
    env = gym.make('LunarLander-v3', render_mode='rgb_array')
    obs_shape = env.observation_space.shape
    action_size = env.action_space.n

    print("Observation shape: ", obs_shape)
    print("Action Size: ", action_size)
    print("Max Episode steps: ", env.spec.max_episode_steps)

    actor_net = create_actor_model(obs_shape, action_size)
    critic_net = create_critic_model(obs_shape)

    # create an RL agent
    agent = A2CAgent(obs_shape, action_size,
                     a_model=actor_net,  c_model=critic_net)

    # train the RL agent on
    a2c_train(env, agent, max_episodes=1500, log_freq=100,
              stop_score=200, max_score=500, min_score=-300)
```

```
Best score:-286.7096895045363, episode:0. Model Saved!
e:0, ep_score:-286.71, avg_ep_score:-286.71, avg100score:-286.71, best_score:-286.71
Best score:-278.85662087282384, episode:2. Model Saved!
Best score:7.116470473337458, episode:3. Model Saved!
Best score:23.33927428594974, episode:10. Model Saved!
Best score:100.85838120915994, episode:20. Model Saved!
Best score:166.68768124754476, episode:21. Model Saved!
Best score:167.36705054863853, episode:31. Model Saved!
Best score:211.82154501357178, episode:41. Model Saved!
Best score:221.82705954781403, episode:49. Model Saved!
Best score:236.63119714607453, episode:57. Model Saved!
Best score:249.53318413060276, episode:71. Model Saved!
Best score:255.81425819686874, episode:72. Model Saved!
e:100, ep_score:-133.23, avg_ep_score:-83.00,  avg100score:-80.96, best_score:255.81
e:200, ep_score:-121.24, avg_ep_score:-109.77, avg100score:-136.81, best_score:255.81
e:300, ep_score:-144.85, avg_ep_score:-116.23, avg100score:-129.22, best_score:255.81
e:400, ep_score:-121.21, avg_ep_score:-120.69, avg100score:-134.12, best_score:255.81
e:500, ep_score:-124.12, avg_ep_score:-122.79, avg100score:-131.17, best_score:255.81
e:600, ep_score:-113.23, avg_ep_score:-124.91, avg100score:-135.57, best_score:255.81
e:700, ep_score:-180.10, avg_ep_score:-124.89, avg100score:-124.78, best_score:255.81
e:800, ep_score:-157.82, avg_ep_score:-125.42, avg100score:-129.12, best_score:255.81
e:900, ep_score:-108.00, avg_ep_score:-126.25, avg100score:-132.86, best_score:255.81
```

Listing 7.59: Main function for applying A2C agent to solve `LunarLander-v3` problem

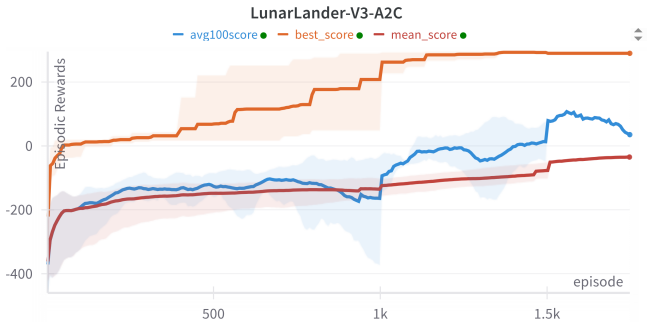


Figure 7.2.: Training performance of A2C algorithm for `Lunarlander-v3` environment

7.3. Asynchronous Advantage Actor-Critic (A3C) Model

In A2C, a single agent interacts with the environment, collects a batch of experience (e.g., a few steps or a full episode), computes the losses for both actor and critic based on this batch, and then updates the agent's network parameters. It is also possible to have multiple workers to collect experiences from different parallel environments and performs a single coordinate update to the shared network parameters of the agent.

Asynchronous Advantage Actor-Critic (A3C) [20] was a breakthrough in deep reinforcement learning, primarily because it enabled highly efficient training on multi-core CPUs without requiring a replay buffer (like in DQN). The "Asynchronous" aspect is key. A3C maintains a global network (actor and critic) and multiple "worker" agents. Each worker agent has its own copy of the network parameters and interacts with its own independent copy of the environment. Each of the multiple parallel worker agents running in separate threads collects its own experience tuples (s, a, r, s') independently. Crucially, each worker computes gradients for its local network parameters based on its collected experience. Instead of waiting for other workers, it asynchronously sends these gradients to the global network to update the global parameters. Periodically, or after a certain number of steps, each worker agent synchronizes its local network parameters with the updated global network parameters. This ensures that all workers are learning from the collective experience without being perfectly in sync. The schematic block diagram of A3C architecture is shown in Figure 7.3.

The mathematical equations for the actor and critic losses are fundamentally the same as in A2C, as they both use the advantage function. The difference lies in how the updates are performed.

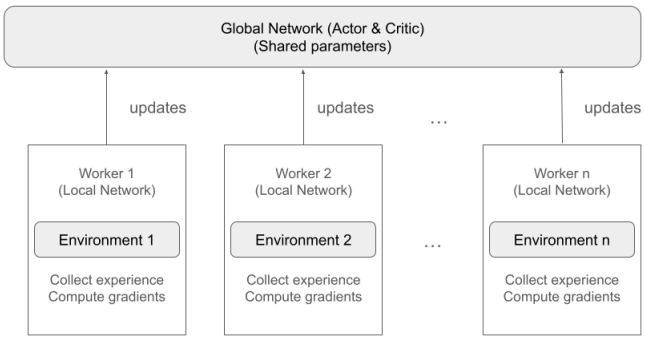


Figure 7.3.: Block Diagram to understand A3C architecture

7.3.1. Python Implementation of A3C Model

The `worker` function

The Python code for implementing a worker is provided in the listing 7.60. The `worker` function creates a local `A2CAgent` referred to as `local_network` using the class definition provided in the previous section. It also creates its own gym environment for interaction. During each episode, the `local_network` starts by copying the weights from the `global_network` and uses these parameters to collects experience tuples (s, a, r, s'). It then computes gradients at the end of the each episode by using this batch of experiences. The computed gradients are pushed to the `gradients_queue` which can then be used for updating parameters of the `global_network`. The parameters between multiple parallel processes are shared by using queues. Several steps are taken to ensure that the queues are not empty leading to deadlock situations. This includes pushing more data into the queue and taking appropriate steps if the queue is empty or full etc. It is also necessary to clear the memory after each thread to prevent running out of memory on the computer. This code uses Python's multiprocessing library to run the code on a multi-core CPU and does not require GPUs for execution.

```
import gymnasium as gym
import numpy as np
import multiprocessing
import time
import gc
import queue
from a2c import A2CAgent
from collections import deque

# Disable GPU for this script
```

```
os.environ['CUDA_VISIBLE_DEVICES'] = '-1'

import tensorflow as tf
import tensorflow_probability as tfp

# Worker function for each process
def worker(worker_id, global_weights_queue,
           gradients_queue, save_request_queue, env_id,
           create_actor_func=None,
           create_critic_func=None,
           max_episodes=1500, max_score = 500,
           min_score = -500, max_steps = None):

    # Set random seed for reproducibility in each process
    tf.random.set_seed(worker_id + 1)
    np.random.seed(worker_id + 1)

    # create environment
    env = gym.make(env_id)
    obs_shape = env.observation_space.shape
    action_size = env.action_space.n

    # Create local network and environment
    if create_actor_func is not None:
        actor = create_actor_func(obs_shape, action_size)
    else:
        actor = None

    if create_critic_func is not None:
        critic = create_critic_func(obs_shape)
    else:
        critic = None
    local_network = A2CAgent(obs_shape, action_size,
                             a_model=actor, c_model=critic)

    # collect experience tuple and compute gradients
    # for the local network
    episode = 0
    ep_scores = []
    best_score = -np.inf
    for episode in range(max_episodes):
        weights_updated = False
        for _ in range(3):  # Retry up to 3 times
            try:
```

```
                        global_weights = global_weights_queue.get_nowait()
                        local_network.set_weights(*global_weights)
                        weights_updated = True
                        break # retrieval success
                    except queue.Empty:
                        time.sleep(0.1)  # Brief pause before retry
                if not weights_updated:
                    continue # Continue with current weights if queue is empty

                state = env.reset()[0]
                episode_reward = 0
                done = False
                step = 0
                states = deque(maxlen=max_steps \
                    if max_steps is not None else 1000)
                actions = deque(maxlen=max_steps \
                    if max_steps is not None else 1000)
                rewards = deque(maxlen=max_steps \
                    if max_steps is not None else 1000)

                # Collect trajectory
                while not done:
                    action = int(local_network.policy(state))
                    next_state, reward, done, truncated, _ = env.step(action)
                    done = done or truncated

                    states.append(state) # type: ignore
                    actions.append(action)
                    rewards.append(reward)

                    state = next_state
                    episode_reward += reward

                step += 1

                    if max_score is not None and episode_reward >= max_score:
                        done = True
                    if min_score is not None and episode_reward <= min_score:
                        done = True
                    if max_steps is not None and step >= max_steps:
                        done = True

                # end of episode
                ep_scores.append(episode_reward)
```

```
                # update the local network
                a_loss, c_loss, actor_grads, critic_grads = \
                    local_network.compute_gradients(
                    states, actions, rewards
                )

                # Update global network
                try:
                    gradients_queue.put_nowait((actor_grads, critic_grads))
                except queue.Full:
                    continue # queue full, skip updated


                if episode_reward > best_score:
                    best_score = episode_reward

                try:
                    save_request_queue.put_nowait(
                        (worker_id, episode, episode_reward))
                except queue.Full:  # skip save request
                    continue

                # free memory
                tf.keras.backend.clear_session()  # Clear TensorFlow session
                states.clear()
                actions.clear()
                rewards.clear()
                gc.collect()  # Collect garbage to free memory
            # end of for-loop
        env.close()
```

Listing 7.60: Worker function that creates an agent to collect experiences from its own individual environment and computes gradient. These gradients are then sent to the global network asynchronously for update

Running multiple workers

The code for running multiple workers on parallel threads is provided in the listing 7.61. The function `run_workers()` uses Python's multiprocessing module to achieve parallel execution of worker function. It creates a global `A2CAgent` referred to as `global_networks` whose parameters are shared with the local network of each worker through the multiprocessing queue called `global_weights_queue`. The gradients computed by the local

networks are received by the global network by using another multiprocessing queue called `gradients_queue`. The gradients received are averaged and then applied to update the parameters of the `global_network`. Adequate steps are taken to deal with situations when multi-processing queues are not empty or full. This prevents deadlock situation where the processes wait indefinitely to receive data from empty queues.

```
def run_workers(env_name, max_num_workers=5, max_episodes=1500,
                wandb_log=True, max_score=500, min_score=-200,
                max_steps=1000,
                create_actor_func=None,
                create_critic_func=None):
    # Set random seed for reproducibility
    tf.random.set_seed(42)
    np.random.seed(42)

    N_WORKERS = min(multiprocessing.cpu_count(), max_num_workers)
    print('N_WORKERS:', N_WORKERS)


    # Initialize environment to get state and action sizes
    env = gym.make(env_name)

    obs_shape = env.observation_space.shape
    action_size = env.action_space.n
    env_id = env.spec.id
    env.close()

    if create_actor_func is not None:
        a_model = create_actor_func(obs_shape, action_size)
    else:
        a_model = None
    if create_critic_func is not None:
        c_model = create_critic_func(obs_shape)
    else:
        c_model = None

    # Initialize global network
    global_network = A2CAgent(obs_shape, action_size,
                              a_model=a_model, c_model=c_model)

    # Create a queue to share weights between processes
    manager = multiprocessing.Manager()
    global_weights_queue = manager.Queue(maxsize=2*N_WORKERS)
    gradients_queue = manager.Queue(maxsize=2*N_WORKERS)
```

```
    save_request_queue = manager.Queue(maxsize=N_WORKERS)
    save_lock = manager.Lock()

    # initial weights for the global network
    for _ in range(2*N_WORKERS):
        # Initialize queue with the global network's weights
        global_weights_queue.put(global_network.get_weights())


    # Create and start worker processes
    processes = []
    for i in range(N_WORKERS):
        p = multiprocessing.Process(
            target=worker,
            args=(i, global_weights_queue, gradients_queue,
                  save_request_queue, env_id,
                  create_actor_func, create_critic_func,
                  max_episodes,
                  max_score, min_score, max_steps, wandb_log)
            )
        p.start()
        processes.append(p)
        print(f'Started worker {p.name}')

    try:
        last_refill_time = time.time()
        best_reward = -np.inf
        while(any(p.is_alive() for p in processes)):
            gradients = []
            for _ in range(N_WORKERS):
                if not gradients_queue.empty():
                    gradients.append(gradients_queue.get())

            if gradients: # process gradients if available
                # Filter out None gradients
                valid_gradients = [
                    (actor_grads, critic_grads)
                    for actor_grads, critic_grads in gradients
                    if actor_grads is not None and critic_grads is not None
                ]
                # Only proceed if there are valid gradients
                if valid_gradients:
                    actor_grads_avg = []
                    critic_grads_avg = []
```

```
                    for actor_grads, critic_grads in valid_gradients:
                        if not actor_grads_avg:
                            actor_grads_avg = [tf.convert_to_tensor(g)\
                                for g in actor_grads]
                            critic_grads_avg = [tf.convert_to_tensor(g)\
                                for g in critic_grads]
                        else:
                            for i, g in enumerate(actor_grads):
                                actor_grads_avg[i] = tf.add(
                                    actor_grads_avg[i], tf.convert_to_tensor(g))
                            for i, g in enumerate(critic_grads):
                                critic_grads_avg[i] = tf.add(
                                    critic_grads_avg[i], tf.convert_to_tensor(g))

                    n = len(valid_gradients)
                    actor_grads_avg = [tf.math.truediv(g, n) \
                        for g in actor_grads_avg]
                    critic_grads_avg = [tf.math.truediv(g, n) \
                        for g in critic_grads_avg]

                    global_network.apply_gradients(actor_grads_avg,
                                                   critic_grads_avg)

                    # Synchronize global weights with all workers
                    updated_weights = global_network.get_weights()
                    for _ in range(N_WORKERS):
                        try:
                            # Synchronize global weights with all workers
                            global_weights_queue.put_nowait(updated_weights)
                        except queue.Full: # skip synchronization
                            while not global_weights_queue.empty():
                                try:
                                    global_weights_queue.get_nowait()
                                except queue.Empty:
                                    break
                            global_weights_queue.put_nowait(updated_weights)

                # periodically refill the global weights queue
                if time.time() - last_refill_time > 5:
                    try:
                        global_weights_queue.put_nowait(
                            global_network.get_weights())
                        last_refill_time = time.time()
                    except queue.Full: # skip refill
```

```
                continue

            # saving weights
            while not save_request_queue.empty():
                try:
                    worker_id, episode, episode_reward = \
                        save_request_queue.get_nowait()
                    print(f"Worker: {worker_id},\
                            episode: {episode},  \
                            reward: {episode_reward:.2f}")
                    if episode_reward > best_reward:
                        best_reward = episode_reward
                        with save_lock:
                            try:
                                global_network.save_weights(
                                    actor_wt_file='a3c_actor.weights.h5',
                                    critic_wt_file='a3c_critic.weights.h5',
                                )
                                print(f"Best Score: {best_reward}. \
                                                Saved weights!")
                            except Exception as e:
                                print(f"Error saving weights: {e}")
                except queue.Empty:
                    break
        except Exception as e:
            print(f"An error occurred: {e}")
        finally:
            print("Shutting down workers...")
            #Wait for all processes to complete
            for p in processes:
                p.join()
                print(f'Worker {p.name} has finished.')
```

Listing 7.61: Python Function to run multiple workers in parallel. The gradients are collected from each worker and used for updating the parameters of the global network asynchronously.

Updating `A2CAgent` to work with multiple workers

The `A2CAgent` class declaration provided in Listing 7.56 is updated to include some additional methods to facilitate working with multiple workers. This is provided in Listing 7.62. This primarily includes separating the gradient computation from the parameter update steps. The method `compute_gradients()` is used by the `local_network`

agent of each worker to compute gradients. The method `apply_gradients()` is used by the `global_network` to update its parameters. The methods `get_weights()` and `set_weights()` allow copying of parameters between the global network and the local networks. Gradient clipping is applied to ensure that the gradients do not grow out of bounds leading to `NaN` values.

```
class A2CAgent:
    def compute_gradients(self, states, actions, rewards):
        states, actions, discnt_rewards = \
            self.compute_discounted_rewards(states, actions, rewards)

        with tf.GradientTape() as tape1, tf.GradientTape() as tape2:
            values = self.critic(states)
            td_error = tf.math.subtract(discnt_rewards, values)
            actor_loss = self.compute_actor_loss(states,\
                                                 actions, td_error)
            critic_loss = tf.reduce_mean(tf.square(td_error))

        # compute gradients
        actor_grads = tape1.gradient(actor_loss,
                                     self.actor.model.trainable_variables)
        critic_grads = tape2.gradient(critic_loss,
                                      self.critic.model.trainable_variables)

        # clip gradients
        actor_grads = [tf.clip_by_norm(grad, self.grad_clip_norm) \
                       if grad is not None else None for grad in actor_grads ]
        critic_grads = [tf.clip_by_norm(grad, self.grad_clip_norm) \
                        if grad is not None else None for grad in critic_grads]

        # Check for NaN gradients
        actor_has_nan = any(tf.reduce_any(tf.math.is_nan(grad)) \
                            for grad in actor_grads if grad is not None)
        critic_has_nan = any(tf.reduce_any(tf.math.is_nan(grad)) \
                             for grad in critic_grads if grad is not None)


        if actor_has_nan or critic_has_nan:
            print(f"NaN gradients detected! "
                  f"Actor NaN: {actor_has_nan}, \
                    Critic NaN: {critic_has_nan}, "
            # Skip sending gradients to avoid corrupting global network
            return None, None, None, None
```

```
            return actor_loss.numpy(), critic_loss.numpy(), \
                   actor_grads, critic_grads

        def apply_gradients(self, actor_grads, critic_grads):
            self.actor.optimizer.apply_gradients(
                zip(actor_grads, self.actor.model.trainable_variables))
            self.critic.optimizer.apply_gradients(
                zip(critic_grads, self.critic.model.trainable_variables))

        def get_weights(self):
            return self.actor.model.get_weights(), \
                   self.critic.model.get_weights()

        def set_weights(self, actor_weights, critic_weights):
            self.actor.model.set_weights(actor_weights)
            self.critic.model.set_weights(critic_weights)
```

Listing 7.62: Additional methods needed to use `A2CAgent` with multiple workers. Specifically, the gradient computation is separated from the updating parameters stage.

7.3.2. Solving `LunarLander-v3` using A3C model

The code provided in Listing 7.63 shows the main code that calls the function `run_workers()` function to train a global `A2CAgent` by running multiple workers in parallel. The training performance of the resulting A3C agent is shown in Figure 7.4. It shows how the average of last 100 episodes (`average100score`) evolves with increasing training episodes. It can be easily seen that A3C clearly outperforms A2C algorithm by achieving higher average episodic score over time. This figure also shows two versions A3C implementation. `A3C_v1` uses discounted return to compute TD error while `A3C_v2` uses the standard Bellman equation to compute TD error from the current s and next states s' information.

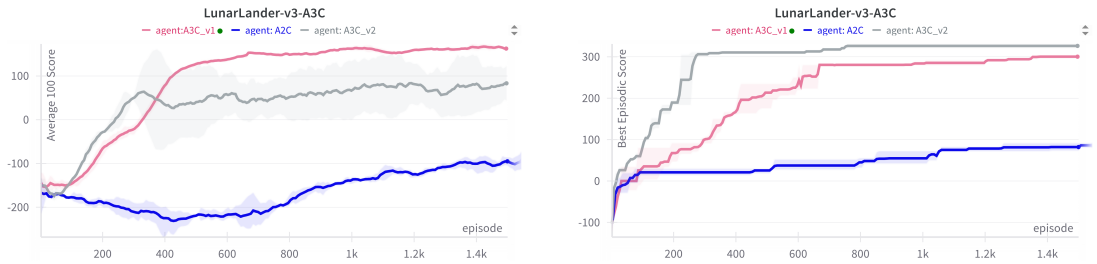
```
from a3c import run_workers
import multiprocessing
import tensorflow as tf

# create actor & Critic models
def create_actor_model(obs_shape, n_actions):
    s_input = tf.keras.layers.Input(shape=obs_shape)
    x = tf.keras.layers.Dense(128, activation='relu')(s_input)
    x = tf.keras.layers.Dense(128, activation='relu')(x)
    a = tf.keras.layers.Dense(n_actions, activation='softmax')(x)
    model = tf.keras.models.Model(s_input, a, name='actor_network')
```

```
    model.summary()
    return model

def create_critic_model(obs_shape):
    s_input = tf.keras.layers.Input(shape=obs_shape)
    x = tf.keras.layers.Dense(128, activation='relu')(s_input)
    x = tf.keras.layers.Dense(128, activation='relu')(x)
    v = tf.keras.layers.Dense(1, activation=None)(x)
    model = tf.keras.models.Model(s_input, v, name='critic_network')
    model.summary()
    return model

if __name__ == '__main__':
    multiprocessing.set_start_method('spawn')
    run_workers(
        env_name='LunarLander-v3',
        max_num_workers=20,
        max_episodes=1500,
        max_score=500,
        min_score=-200,
        max_steps=1000,
        create_actor_func=create_actor_model,
        create_critic_func=create_critic_model,
    )
```

```
Worker: 11, episode: 7, reward: -106.48
Worker: 16, episode: 9, reward: -139.72
Worker: 2, episode: 7, reward: -203.49
Worker: 15, episode: 9, reward: -148.49
Worker: 10, episode: 9, reward: 26.72
Best Score: 26.721404883008873. Saved weights!
Worker: 1, episode: 9, reward: -164.57
Worker: 7, episode: 10, reward: -129.38
Worker: 19, episode: 11, reward: -96.75
Worker: 14, episode: 9, reward: -113.13
Worker: 13, episode: 10, reward: -213.46
Worker: 9, episode: 8, reward: -200.68
.. .
.. .
Worker: 15, episode: 439, reward: 132.81
Worker: 5, episode: 436, reward: 165.65
Worker: 16, episode: 443, reward: 87.35
Worker: 4, episode: 435, reward: 133.58
.. .
.. .
orker: 18, episode: 567, reward: 128.30
Worker: 10, episode: 577, reward: 150.79
Worker: 17, episode: 581, reward: 170.52
```

```
Worker: 7, episode: 592, reward: 181.43
Worker: 11, episode: 601, reward: 135.22
Worker: 2, episode: 593, reward: 184.58
```

Listing 7.63: Main python code for training A3C agent on `LunarLander-v3` problem environment



(a) Average of last 100 episodes

(b) Best Episodic Score

Figure 7.4.: Training performance of A3C algorithm for `LunarLander-v3` environment

7.4. Summary

This chapter explores actor-critic methods, a cornerstone of reinforcement learning that integrates policy-based and value-based approaches. It begins with the foundational actor-critic framework, where the actor learns a policy to select actions and the critic evaluates them by estimating state values, balancing exploration and exploitation. The naive actor-critic approach is introduced, highlighting its simplicity but also its limitations, such as high variance in policy updates due to unscaled rewards. The chapter then delves into advanced variants: A2C (Advantage Actor-Critic) and A3C (Asynchronous Advantage Actor-Critic). A2C improves stability by using the advantage function to guide policy updates and synchronizes experience collection across multiple environments for consistent gradient updates. In contrast, A3C leverages asynchronous parallel workers to update a shared network, enhancing training speed but introducing potential instability from stale gradients. Through mathematical formulations and practical comparisons, the chapter illustrates how these methods address variance and efficiency challenges, making them suitable for complex tasks like game playing and robotics.

8. Soft Actor-Critic: A Maximum Entropy Reinforcement Learning Algorithm

The **Soft Actor-Critic (SAC)** algorithm [21] stands as a cornerstone in modern deep reinforcement learning, particularly for **continuous control tasks**. Unlike traditional reinforcement learning methods that solely focus on maximizing expected cumulative reward, SAC operates within the framework of **maximum entropy reinforcement learning**. This paradigm not only seeks to achieve high rewards but also encourages the policy to be as random as possible, given the reward constraints. This inherent drive for **exploration** makes SAC robust, stable, and highly effective in complex environments.

8.1. The Maximum Entropy Objective

The standard RL objective is to maximize the expected cumulative reward:

$$J(\pi) = \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\left[\sum_t \gamma^t r(s_t,a_t)\right], \tag{8.1}$$

SAC modifies this objective by including an entropy term to encourage exploration. Therefore, SAC's power stems from optimizing a policy that simultaneously maximizes both the expected cumulative reward and the **entropy of the policy**. This objective is formally expressed as:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\left[\gamma^t r(s_t,a_t) + \alpha H(\pi(\cdot|s_t))\right] \tag{8.2}$$

Let's break down the components of this crucial objective:

- $r(s_t, a_t)$ represents the **reward** obtained at time t for executing action a_t in state s_t and $\gamma \in [0, 1)$ is the discount factor.
- ρ_π signifies the **state-action marginal visitation probability distribution** when following policy π. See A.2.1 for more details.
- $H(\pi(\cdot|s_t))$ denotes the **entropy of the policy** at state s_t. Entropy, in information theory, quantifies the uncertainty or randomness of a probability distribution. For a discrete distribution P, it's $H(P) = -\sum_x P(x) \log P(x)$. For continuous distributions, it's the differential entropy, typically $H(P) = \mathbb{E}_{x\sim P}[-\log P(x)]$. A higher entropy value implies a more uncertain or exploratory policy.

- α is the **temperature parameter**. This critical hyperparameter dictates the trade-off between maximizing reward and maximizing entropy. A larger α value places a greater emphasis on exploration and randomness, while a smaller α prioritizes reward accumulation. Interestingly, α can also be adaptively learned during the training process, a common enhancement in SAC implementations.

8.2. Soft Value Functions: The Foundation of SAC

From the maximum entropy objective, SAC defines "soft" versions of the traditional value functions, which explicitly incorporate the entropy term.

8.2.1. The Soft Q-Function

The **soft Q-function**, denoted $Q^\pi(s, a)$, quantifies the expected return from taking action a in state s and subsequently following policy π, with the added entropy bonus. Its Bellman equation is:

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim P, a_{t+1} \sim \pi} \left[Q^\pi(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1}|s_{t+1}) \right] \tag{8.3}$$

Notice the crucial distinction from the standard Bellman equation: an entropy term, $\alpha \log \pi(a_{t+1}|s_{t+1})$, is *subtracted* from the future Q-value. This subtraction effectively incentivizes the policy to favor actions that lead to states with higher subsequent policy entropy, thereby promoting diverse behavior.

8.2.2. The Soft Value Function

Similarly, the **soft value function**, $V^\pi(s)$, represents the expected return from state s under policy π, inherently including the entropy bonus:

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi} \left[Q^\pi(s_t, a_t) - \alpha \log \pi(a_t|s_t) \right] \tag{8.4}$$

This equation explicitly highlights that the value of a state in SAC is not solely based on the expected Q-value but also receives a bonus proportional to the entropy of the policy within that state.

8.3. Architecture and Training Dynamics

SAC typically employs a set of interconnected neural networks and an experience replay buffer to facilitate its learning process.

8.3.1. Key Components:

1. **Actor Network (π_θ)**: Parameterized by θ, this network represents the **stochastic policy**. For continuous action spaces, it commonly outputs the mean and standard deviation of a Gaussian distribution, from which actions are sampled.

2. **Critic Networks (Q-functions Q_{ϕ_1}, Q_{ϕ_2})**: SAC utilizes **two separate Q-function networks**, parameterized by ϕ_1 and ϕ_2. These critics estimate the soft Q-value for a given state-action pair. Employing dual critics is a technique borrowed from **TD3 (Twin Delayed DDPG)**, designed to mitigate the problem of **Q-value overestimation**, a common pitfall in deep Q-learning algorithms.

3. **Target Critic Networks (Q'_{ϕ_1}, Q'_{ϕ_2})**: These are smoothed, delayed copies of the primary critic networks. They provide stable targets for Q-value updates, crucial for learning stability. Their parameters are typically updated using **Polyak averaging** (e.g., $\phi' \leftarrow \rho\phi' + (1-\rho)\phi$, where ρ is a small constant like 0.995) or periodically copied from the main critics.

4. **Replay Buffer**: As an **off-policy algorithm**, SAC extensively uses a **replay buffer** to store past transitions (state, action, reward, next state, done flag). This mechanism allows for efficient reuse of historical data, significantly boosting **sample efficiency** compared to on-policy methods.

8.3.2. Training Objectives and Updates:

The training of SAC involves a simultaneous and iterative update of the policy network, the two Q-function networks, and optionally, the temperature parameter.

1. **Q-Network Update (Critic Update)**: The Q-networks are trained to minimize the **soft Bellman residual**. For each Q-network Q_{ϕ_i}, the loss function is typically a squared error:

$$L(\phi_i) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(Q_{\phi_i}(s,a) - y\right)^2\right] \tag{8.5}$$

   Here, D denotes the experience replay buffer. The **target value** y is computed as:

$$y = r + \gamma\left(\min_{j=1,2} Q'_{\phi_j}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right) \tag{8.6}$$

   In this equation, $\tilde{a}'$ is an action *sampled from the current policy* $\pi_\theta(\cdot|s')$, and Q'_{ϕ_j} are the target Q-networks. The min operator (a feature of clipped double Q-learning) is critical for preventing overestimation bias in the Q-value targets.

2. **Policy Network Update (Actor Update)**: The actor network is updated to maximize the soft Q-value, while simultaneously maintaining a high level of policy entropy. The policy objective can be derived from minimizing the KL divergence between the policy and a "soft-max" distribution of Q-values. More directly, it's often expressed as minimizing:

$$J(\theta) = \mathbb{E}_{s\sim D, \epsilon\sim\mathcal{N}(0,1)}\left[\alpha \log \pi_\theta(\tilde{a}|s) - \min_{j=1,2} Q_{\phi_j}(s,\tilde{a})\right] \tag{8.7}$$

   A crucial technique used here is the **reparameterization trick**. For a continuous policy (e.g., Gaussian), the sampled action $\tilde{a}$ can be expressed as a deterministic

function of the policy parameters and a noise variable ϵ. For instance, if the policy outputs mean $\mu_\theta(s)$ and standard deviation $\sigma_\theta(s)$, then $\tilde{a} = \mu_\theta(s) + \sigma_\theta(s) \odot \epsilon$, where $\epsilon \sim \mathcal{N}(0, 1)$. This trick makes the objective function differentiable with respect to θ, enabling gradient-based optimization.

3. **Temperature Parameter Update (Optional)**: Many SAC implementations include an adaptive learning mechanism for the temperature parameter α. This is typically achieved by minimizing a loss function that drives the policy's entropy towards a predefined **target entropy** H_0:

$$L(\alpha) = \mathbb{E}_{s \sim D, a \sim \pi_\theta} \left[-\alpha (\log \pi_\theta(a|s) + H_0) \right] \tag{8.8}$$

By learning α, the algorithm can automatically adjust the balance between exploration and exploitation throughout the training process, adapting to the specific task's requirements.

8.3.3. Soft Actor-Critic Pseudocode

Algorithm 8.1 provides a summary of the Soft Actor-Critic training process.

8.4. Advantages of Soft Actor-Critic

SAC has gained significant traction in the reinforcement learning community due to several compelling advantages:

- **Exceptional Sample Efficiency**: Thanks to its off-policy nature and reliance on experience replay, SAC can effectively reuse collected data. This is a major benefit in scenarios where data acquisition is costly or time-consuming, such as in robotics.

- **Enhanced Stability**: The explicit **entropy regularization** helps to stabilize the training process. By encouraging diverse actions, SAC is less prone to collapsing into suboptimal deterministic policies or exhibiting brittle behavior due to minor estimation errors in value functions.

- **State-of-the-Art Performance**: SAC has consistently demonstrated top-tier performance across a wide array of continuous control tasks, ranging from simulated locomotion to complex robotic manipulation.

- **Robustness**: The combination of the maximum entropy framework, the use of dual critics, and often, an adaptively learned temperature parameter, contributes to SAC's robustness against varying hyperparameter choices and environmental stochasticity.

Algorithm 8.1 Soft Actor-Critic (SAC)

Require: Initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, target Q-function parameters $\phi_1^{\text{target}}, \phi_2^{\text{target}}$, temperature α (or its initial log value $\log \alpha$), learning rates $\eta_\theta, \eta_\phi, \eta_\alpha$, discount factor γ, Polyak averaging coefficient ρ, replay buffer capacity M, target entropy H_0 (if learning α).

Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$
Initialize $\phi_1^{\text{target}} \leftarrow \phi_1$, $\phi_2^{\text{target}} \leftarrow \phi_2$
for each iteration $k = 1, \ldots, N_{\text{iterations}}$ **do**
    Sample state s_t from environment
    Sample action $a_t \sim \pi_\theta(s_t)$
    Execute a_t in environment, observe $r_t, s_{t+1}, \text{done}_t$
    Store $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$ in $\mathcal{D}$
    for each gradient step $m = 1, \ldots, N_{\text{gradient\_steps}}$ **do**
        Sample a batch of transitions $(s, a, r, s', \text{done})$ from $\mathcal{D}$
        Compute $\tilde{a}' \sim \pi_\theta(s')$ using reparameterization trick
        Compute $\log \pi_\theta(\tilde{a}'|s')$
        Compute target Q-values:
        $y = r + \gamma(1 - \text{done}) \left(\min(Q_{\phi_1}^{\text{target}}(s', \tilde{a}'), Q_{\phi_2}^{\text{target}}(s', \tilde{a}')) - \alpha \log \pi_\theta(\tilde{a}'|s') \right)$
        Update Q-function parameters:
        $\phi_1 \leftarrow \phi_1 - \eta_\phi \nabla_{\phi_1} \left(Q_{\phi_1}(s, a) - y \right)^2$
        $\phi_2 \leftarrow \phi_2 - \eta_\phi \nabla_{\phi_2} \left(Q_{\phi_2}(s, a) - y \right)^2$
        Compute $\tilde{a} \sim \pi_\theta(s)$ using reparameterization trick
        Compute $\log \pi_\theta(\tilde{a}|s)$
        Update policy parameters:
        $\theta \leftarrow \theta - \eta_\theta \nabla_\theta \left(\alpha \log \pi_\theta(\tilde{a}|s) - \min(Q_{\phi_1}(s, \tilde{a}), Q_{\phi_2}(s, \tilde{a})) \right)$
        if learning α **then**
            Update temperature parameter α:
            $\log \alpha \leftarrow \log \alpha - \eta_\alpha \nabla_{\log \alpha} \left(-\alpha (\log \pi_\theta(\tilde{a}|s) + H_0) \right)$
        end if
        Update target Q-function networks (Polyak averaging):
        $\phi_1^{\text{target}} \leftarrow \rho \phi_1^{\text{target}} + (1 - \rho) \phi_1$
        $\phi_2^{\text{target}} \leftarrow \rho \phi_2^{\text{target}} + (1 - \rho) \phi_2$
    end for
end for

8.5. Implementing SAC Algorithm using Python

In this section, we will discuss the details of implementing the SAC algorithm using Tensorflow 2.x and Python. SAC is implemented using an actor-critic architecture where the actor is to used estimate the action policies while a critic is used to evaluate the actor by estimating a Q or value function. The details are discussed in the following subsections.

8.5.1. Actor Network

As stated in section 8.3, the actor learns a stochastic policy. The actor uses a sequential deep network to estimate mean (μ) and log of standard deviation (σ), `log_std`, for a given input state. It is also possible to output standard deviation `std` instead of `log_std`. It is important to note that the `log_std` value is clipped in the range $(-20.0, 2)$ to avoid instability. The method `sample_action()` is used to sample an action from a Gaussian distribution created using the output of the actor network (μ, σ). During the training phase, a reparameterization trick is used that adds noise to a deterministic policy to improve exploration. It is also important to avoid numerical instability (`NaN` values) by ensuring that actor output is quashed by using a tanh() function and scaled to its actual range. Similarly, the values to the `log_prob` function is quashed to a range (0, 1) to avoid zero values or negative values which can result in a NaN value.

```
class Actor():
    def __init__(self, obs_shape, action_shape,
                 learning_rate=0.0001,
                 action_upper_bound=1.0,
                 model=None):
        self.obs_shape = obs_shape
        self.action_size = action_shape[0]
        self.lr = learning_rate
        self.max_action = action_upper_bound

        if model is None:
            self.model = self._build_model()
        else:
            self.model = tf.keras.models.clone_model(model)
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=self.lr)

        # Constraints for log_std to ensure numerical stability
        # and reasonable exploration
        # log_std values are usually clipped to a range like [-20, 2]
        self.max_log_std = tf.constant(2.0, dtype=tf.float32)
        self.min_log_std = tf.constant(-20.0, dtype=tf.float32)

    def _build_model(self):
        x = tf.keras.layers.Input(shape=self.obs_shape)
        f = tf.keras.layers.Dense(256, activation='relu',
```

```
        kernel_initializer=tf.keras.initializers.HeUniform())(x)
        f = tf.keras.layers.Dense(256, activation='relu',
        kernel_initializer=tf.keras.initializers.HeUniform())(f)
        mu = tf.keras.layers.Dense(self.action_size, activation=None,
        kernel_initializer=tf.keras.initializers.HeUniform())(f)
        log_std = tf.keras.layers.Dense(self.action_size, activation=None,
        kernel_initializer=tf.keras.initializers.HeUniform())(f)
        model = tf.keras.models.Model(inputs=x,
        outputs=[mu, log_std], name='actor')
        model.summary()
    return model

def __call__(self, states):
    mean, log_std = self.model(states)
    log_std = tf.clip_by_value(log_std, self.min_log_std,
                               self.max_log_std)
    return mean, log_std

def sample_action(self, state, reparameterize=False):
    mean, log_std = self(state)
    std = tf.exp(log_std)
    dist = tfp.distributions.Normal(mean, std)

    if reparameterize: # Reparameterization trick
        z = mean + std * tf.random.normal(tf.shape(mean))
    else:
        z = dist.sample()

    action = tf.tanh(z) * self.max_action
    squash = 1 - tf.math.pow(tf.tanh(z), 2) # squash values to (0,1)
    squash = tf.clip_by_value(squash, 1e-6, 1.0)  # Avoid division by zero
    log_prob = dist.log_prob(z)
    log_prob -= tf.math.log(squash)
    log_prob = tf.math.reduce_sum(log_prob,  axis=-1, keepdims=True)
    return action, log_prob
```

Listing 8.64: Python code for the SAC Actor Class

8.5.2. Critic Network

The `Critic` class estimates the Q-function $Q(s, a)$ by using a sequential deep network from the input state and action values. The output is a scalar value.

```
class Critic:
    """ Approximates Q(s,a) function """
    def __init__(self, obs_shape, action_shape,
                 learning_rate=1e-3,
                 model=None):
```

```
        self.obs_shape = obs_shape    # shape: (m,)
        self.action_shape = action_shape  # shape: (n, )
        self.lr = learning_rate
        # create NN model
        if model is None:
            self.model = self._build_net()
        else:
            self.model = tf.keras.models.clone_model(model)
        self.optimizer = tf.keras.optimizers.Adam(self.lr)

    def _build_net(self):
        state_input = tf.keras.layers.Input(shape=self.obs_shape)
        action_input = tf.keras.layers.Input(shape=self.action_shape)
        concat = tf.keras.layers.Concatenate()([state_input, action_input])
        out = tf.keras.layers.Dense(256, activation='relu',
            kernel_initializer=tf.keras.initializers.HeUniform())(concat)
        out = tf.keras.layers.Dense(256, activation='relu',
            kernel_initializer=tf.keras.initializers.HeUniform())(out)
        out = tf.keras.layers.Dense(256, activation='relu',
            kernel_initializer=tf.keras.initializers.HeUniform())(out)
        net_out = tf.keras.layers.Dense(1)(out)
        model = tf.keras.Model(inputs=[state_input, action_input],
            outputs=net_out, name='critic')
        model.summary()
        return model

    def __call__(self, states, actions):
        """ Returns Q(s,a) value """
        return self.model([states, actions])
```

Listing 8.65: Python code for the SAC Critic class definition

8.5.3. SAC Agent

The `SACAgent` class creates two critic models `self.critic_1` and `self.critic_2` and the corresponding two target models `self.target_critic_1` and `self.target_critic_2` respectively, one actor model `self.actor` and a replay buffer `self.buffer`. The target critic networks and the primary critic networks share the same weights in the beginning. The target weights are updated slowly over time using Polyak's averaging method. The temperature parameter α is also updated to minimize the loss function given by equation (8.8). The critic parameters are updated to minimize the soft Bellman residual loss functions given by equation (8.5). The actor parameters are updated to maximize the loss function given by equation (8.7) which essentially tries to maximize the soft Q function while maintaining a high level of policy entropy. The performance of SAC algorithm can be improved by having multiple critic update steps for each actor update step. This helps in learning the Q function faster leading to better training stability and improved

performance.

```
class SACAgent:
    def __init__(self, obs_shape, action_shape,
                 lr_a=1e-4, lr_c=3e-4,
                 lr_alpha=3e-4, alpha=0.2,
                 gamma=0.99, polyak=0.999,
                 action_upper_bound=1.0,
                 buffer_size=1000000,
                 batch_size=256,
                 reward_scale=1.0,
                 max_grad_norm=None, # required for gradient clipping
                 actor_model=None,
                 critic_model=None ):

        self.obs_shape = obs_shape
        self.action_shape = action_shape
        self.gamma = gamma  # Discount factor
        self.polyak = polyak # Polyak averaging coefficient
        self.batch_size = batch_size
        self.buffer_size = buffer_size
        self.actor_lr = lr_a
        self.critic_lr = lr_c
        self.alpha_lr = lr_alpha
        self.reward_scale = reward_scale
        self.max_grad_norm = max_grad_norm
        self.name = 'SAC'

        # Initialize actor and critic networks
        self.actor = Actor(obs_shape, action_shape, self.actor_lr,
                           action_upper_bound, model=actor_model)
        self.critic_1 = Critic(obs_shape, action_shape, self.critic_lr,
                               model=critic_model)
        self.critic_2 = Critic(obs_shape, action_shape, self.critic_lr,
                               model=critic_model)

        # Target networks for soft updates
        self.target_critic_1 = Critic(obs_shape, action_shape,
                                      self.critic_lr, model=critic_model)
        self.target_critic_2 = Critic(obs_shape, action_shape,
                                      self.critic_lr, model=critic_model)

        # make alpha a trainable variable
        self.log_alpha = tf.Variable(tf.math.log(alpha),
                                     dtype=tf.float32, trainable=True, name='log_alpha')
        self.alpha_optimizer = tf.keras.optimizers.Adam(
                                                learning_rate=self.alpha_lr)

        # Target entropy for action space
        self.target_entropy = tf.constant(-np.prod(self.action_shape),
                                          dtype=tf.float32)

        # Replay buffer
        self.buffer = ReplayBuffer(self.buffer_size)

        # Initialize target networks with the same weights as the main networks
        self.target_critic_1.model.set_weights(
                                    self.critic_1.model.get_weights())
        self.target_critic_2.model.set_weights(
                                    self.critic_2.model.get_weights())

    def update_target_networks(self):
```

```
        """
        Soft update target networks using Polyak averaging
        """
        for target_var, var in zip(self.target_critic_1.model.trainable_variables,
                                   self.critic_1.model.trainable_variables):
            target_var.assign(self.polyak * target_var + (1 - self.polyak) * var)

        for target_var, var in zip(self.target_critic_2.model.trainable_variables,
                                   self.critic_2.model.trainable_variables):
            target_var.assign(self.polyak * target_var + (1 - self.polyak) * var)

    def choose_action(self, state, evaluate=False):
        """
        Choose an action based on the current state
        """
        state = tf.expand_dims(tf.convert_to_tensor(state, dtype=tf.float32), axis=0)
        action, _ = self.actor.sample_action(state, reparameterize=not evaluate)
        action = tf.squeeze(action, axis=0)  # Remove batch dimension
        return action.numpy()

    def store_transition(self, state, action, reward, next_state, done):
        """
        Store a transition in the replay buffer
        : inputs are numpy arrays
        """
        self.buffer.add((state, action, reward, next_state, done))

    def update_critic(self, states, actions, rewards, next_states, dones):
        """
        Update the critic networks using the sampled transitions
        : inputs are tensors
        """

        with tf.GradientTape(persistent=True) as tape:
            # predict next_actions and log_probs for next states
            next_actions, next_log_probs = self.actor.sample_action(next_states)

            # compute target Q-values using the target critic networks
            target_q1 = self.target_critic_1(next_states, next_actions)
            target_q2 = self.target_critic_2(next_states, next_actions)
            min_target_q_next = tf.minimum(target_q1, target_q2)

            # Soft Bellman backup equation for target Q-value
            # y = r + gamma * (1 - done) * (min_Q_target(s', a') - alpha * log_pi(a'|s'))
            target_q_values = self.reward_scale * rewards + \
                              self.gamma * (tf.ones_like(dones) - dones) * \
                              (min_target_q_next - tf.exp(self.log_alpha) * next_log_probs)

            # compute current Q-values
            current_q1 = self.critic_1(states, actions)
            current_q2 = self.critic_2(states, actions)

            # compute critic losses
            critic_1_loss = tf.reduce_mean(tf.square(current_q1 - target_q_values))
            critic_2_loss = tf.reduce_mean(tf.square(current_q2 - target_q_values))

        # compute gradients for critic networks
        critic_1_grads = tape.gradient(critic_1_loss, self.critic_1.model.trainable_variables)
        critic_2_grads = tape.gradient(critic_2_loss, self.critic_2.model.trainable_variables)

        if self.max_grad_norm is not None:
            # Clip gradients to avoid exploding gradients
```

```
                critic_1_grads, _ = tf.clip_by_global_norm(critic_1_grads, self.max_grad_norm)
                critic_2_grads, _ = tf.clip_by_global_norm(critic_2_grads, self.max_grad_norm)

            # apply gradients to the critic networks if gradients are not None
            if critic_1_grads is not None:
                self.critic_1.optimizer.apply_gradients(zip(critic_1_grads,
                                                            self.critic_1.model.trainable_variables))
            if critic_2_grads is not None:
                self.critic_2.optimizer.apply_gradients(zip(critic_2_grads,
                                                            self.critic_2.model.trainable_variables))

            mean_c_loss = (critic_1_loss + critic_2_loss) / 2.0
            return mean_c_loss

    def update_actor(self, states):
        """
        Update the actor network
        inputs are tensors
        outputs: actor_loss and alpha_loss
        """

            with tf.GradientTape(persistent=True) as tape:
                # Sample actions and log probabilities for current states
                new_actions, log_probs = self.actor.sample_action(states)

                # Compute Q-values for the sampled actions
                q1_new = self.critic_1(states, new_actions)
                q2_new = self.critic_2(states, new_actions)
                min_q = tf.minimum(q1_new, q2_new)

                # Actor loss is the mean of the Q-values minus the entropy term
                # Actor loss (maximize soft Q-value, incorporating entropy)
                # J_pi = E_s,a~pi [alpha * log_pi(a|s) - Q(s,a)] -> minimize -J_pi
                actor_loss = tf.reduce_mean(tf.exp(self.log_alpha) * log_probs - min_q)

                # alpha loss is computed as the mean of the target entropy minus the log probability
                alpha_loss = tf.reduce_mean(tf.negative(self.log_alpha) * \
                                            (log_probs + self.target_entropy))

            # Compute gradients for the actor network
            actor_grads = tape.gradient(actor_loss, self.actor.model.trainable_variables)
            # Compute gradients for alpha
            alpha_grads = tape.gradient(alpha_loss, [self.log_alpha])

            if self.max_grad_norm is not None:
                # Clip gradients to avoid exploding gradients
                actor_grads, _ = tf.clip_by_global_norm(actor_grads, self.max_grad_norm)
                alpha_grads, _ = tf.clip_by_global_norm(alpha_grads, self.max_grad_norm)

            # Apply gradients to the actor network if gradients are not None
            if actor_grads is not None:
                self.actor.optimizer.apply_gradients(zip(actor_grads,
                                                         self.actor.model.trainable_variables))

            # Apply gradients to alpha if gradients are not None
            if alpha_grads is not None:
                self.alpha_optimizer.apply_gradients(zip(alpha_grads, [self.log_alpha]))
            return actor_loss, alpha_loss

    def train(self, update_per_step=1):
        if len(self.buffer) < self.batch_size:
            return 0, 0, 0
```

```
        c_losses, a_losses, alpha_losses = [], [], []
        for _ in range(update_per_step):

            # Sample a batch of transitions from the replay buffer
            states, actions, rewards, next_states, dones = \
                self.buffer.sample_unpacked(self.obs_shape, self.action_shape, self.batch_size)

            states = tf.convert_to_tensor(states, dtype=tf.float32)
            actions = tf.convert_to_tensor(actions, dtype=tf.float32)
            rewards = tf.convert_to_tensor(rewards, dtype=tf.float32)
            next_states = tf.convert_to_tensor(next_states, dtype=tf.float32)
            dones = tf.convert_to_tensor(dones, dtype=tf.float32)

            # Update Critic networks
            critic_loss = self.update_critic(states, actions, rewards, next_states, dones)
            actor_loss, alpha_loss = self.update_actor(states)

            # Update target networks
            self.update_target_networks()

            c_losses.append(critic_loss)
            a_losses.append(actor_loss)
            alpha_losses.append(alpha_loss)
        mean_critic_loss = tf.reduce_mean(c_losses)
        mean_actor_loss = tf.reduce_mean(a_losses)
        mean_alpha_loss = tf.reduce_mean(alpha_losses)
        return mean_critic_loss, mean_actor_loss, mean_alpha_loss
```

Listing 8.66: Python code for implementing SAC agent class

8.5.4. Replay Buffer

As mentioned before, SAC is an off-policy algorithm where the experiences are first stored in a replay buffer and then sampled in batches during training phase. The replay buffer used here is the same one discussed in Section 5.4.1. The Python class for implementing replay buffer is provided in the code listing 5.18. We augment this class by providing an additional method `sample_unpacked()` that unpacks the samples into

```
    def sample_unpacked(self, obs_shape, action_shape, batch_size=24):
        """ Returns a batch of experiences as a tuple of numpy arrays.
        Input:
            batch_size: int
            obs_shape: tuple, shape of the observation space
        returns: (states, actions, rewards, next_states, dones) """

        mini_batch = self.sample(batch_size)
        assert len(mini_batch[0]) == 5, "Each experience tuple must have 5 elements: (s,a,r,s',d)"
        states = np.zeros((batch_size, *obs_shape))
        next_states = np.zeros((batch_size, *obs_shape))
        actions = np.zeros((batch_size, *action_shape))
        rewards = np.zeros((batch_size, 1))
        dones = np.zeros((batch_size, 1))

        for i in range(len(mini_batch)):
            states[i] = mini_batch[i][0]
            actions[i] = mini_batch[i][1]
```

```
            rewards[i] = mini_batch[i][2]
            next_states[i] = mini_batch[i][3]
            dones[i]  = mini_batch[i][4]
        return states, actions, rewards, next_states, dones
```

Listing 8.67: Method to sample experiences from the replay buffer and convert them into numpy arrays

8.5.5. Value Network

The above implementation uses the minimum of two target critics minus the entropy term as the target for training the critic networks. In practice, this target can be noisy or biased early in training when the critics are not yet accurate. This can destabilize the learning process. This can be avoided by using a separate value network to provide a smoother target for critic training. The use a separate value network reduces variances and stabilizes learning. The Python code for creating a value network class is provided in code listing 8.68.

```
class ValueNetwork():
    """  Approximates V(s) function   """
    def __init__(self, obs_shape,
                 learning_rate=1e-3,
                 model=None):
        self.obs_shape = obs_shape
        self.lr = learning_rate
        # create NN model
        if model is None:
            self.model = self._build_net()
        else:
            self.model = tf.keras.models.clone_model(model)
        self.optimizer = tf.keras.optimizers.Adam(self.lr)

    def _build_net(self):
        inp = tf.keras.layers.Input(shape=self.obs_shape)
        out = tf.keras.layers.Dense(256, activation='relu',
                kernel_initializer=tf.keras.initializers.HeUniform())(inp)
        out = tf.keras.layers.Dense(256, activation='relu',
                kernel_initializer=tf.keras.initializers.HeUniform())(out)
        out = tf.keras.layers.Dense(256, activation='relu',
                kernel_initializer=tf.keras.initializers.HeUniform())(out)
        net_out = tf.keras.layers.Dense(1)(out)
        model = tf.keras.Model(inputs=inp, outputs=net_out, name='value_network')
        model.summary()
        return model

    def __call__(self, states):
        """ Returns V(s) value  """
```

```
        v = self.model(states)
    return v
```

Listing 8.68: Python class for creating a value network

8.5.6. SAC Agent with a separate value network

The SAC agent class remains mostly same as before with the following changes. The agent has one actor model, two critic models, a replay buffer and one primary value model and a target value model. The two critics are updated by using value network to compute the q-target values. It reduces variance and enhances the stability of the algorithm. The target value network is updated using Polyak averaging. We call this agent as 'SAC2' agent to differentiate from the previous implementation that does not use a value network.

```
class SACAgent:
    """ Soft Actor-Critic Agent """
    def __init__(self, obs_shape, action_shape,
                 action_upper_bound=1.0, <other args ...>):
        self.obs_shape = obs_shape
        self.action_shape = action_shape
        self.action_size = action_shape[0]
        self.action_upper_bound = action_upper_bound
        self.batch_size = batch_size
        self.name = 'SAC2'
        self.target_entropy = -np.prod(action_shape)  # Target entropy = -|A|
        <snip>

        # Initialize networks
        self.actor = Actor(obs_shape, action_shape, lr_a,
                           action_upper_bound, model=actor_model)
        self.critic_1 = Critic(obs_shape, action_shape,
                               learning_rate=self.actor_lr, model=critic_model)
        self.critic_2 = Critic(obs_shape, action_shape,
                               learning_rate=self.critic_lr, model=critic_model)
        self.value_network = ValueNetwork(obs_shape,
                                          learning_rate=self.critic_lr, model = value_model)

        # create a replay buffer
        self.buffer = ReplayBuffer(self.buffer_size)

        # Target networks for stability
        self.target_value_network = ValueNetwork(obs_shape, learning_rate=self.critic_lr)
        self.target_value_network.model.set_weights(self.value_network.model.get_weights())

        # Initialize alpha (temperature parameter)
        self.log_alpha = tf.Variable(tf.math.log(alpha), trainable=True, dtype=tf.float32)
        self.alpha_optimizer = tf.keras.optimizers.Adam(learning_rate=self.alpha_lr)


    def choose_action(self, state, evaluate=False):
        """ same as before """
        pass

    def store_transition(self, state, action, reward, next_state, done):
        """ same as before """
```

```
        pass

    def update_target_networks(self):
        """ Update only target value network using Polyak averaging   """
        for target_var, var in zip(self.target_value_network.model.trainable_variables,
                                   self.value_network.model.trainable_variables):
            target_var.assign(self.polyak * target_var + (1 - self.polyak) * var)

    def update_value_network(self, states, next_states):
        with tf.GradientTape() as tape:
            next_actions, next_log_probs = self.actor.sample_action(next_states)
            next_q1 = self.critic_1(next_states, next_actions)
            next_q2 = self.critic_2(next_states, next_actions)
            next_q = tf.minimum(next_q1, next_q2)
            value_target = next_q - tf.exp(self.log_alpha) * next_log_probs
            value = self.value_network(states)
            value_loss = tf.reduce_mean(tf.square(value - value_target))

        value_grads = tape.gradient(value_loss, self.value_network.model.trainable_variables)

        if self.max_grad_norm is not None:
            value_grads, _ = tf.clip_by_global_norm(value_grads, self.max_grad_norm)

        if value_grads is not None:
            # Apply gradients to the value network
            self.value_network.optimizer.apply_gradients(zip(value_grads, \
                                                             self.value_network.model.trainable_variables))
        return value_loss

    def update_critic(self, states, actions, rewards, next_states, dones):
        with tf.GradientTape(persistent=True) as tape:
            q1 = self.critic_1(states, actions)
            q2 = self.critic_2(states, actions)
            next_v = self.target_value_network(next_states)
            target_q = self.reward_scale * rewards + self.gamma * (1 - dones) * next_v
            critic_1_loss = tf.reduce_mean(tf.square(target_q - q1))
            critic_2_loss = tf.reduce_mean(tf.square(target_q - q2))

        critic_1_grads = tape.gradient(critic_1_loss, self.critic_1.model.trainable_variables)
        critic_2_grads = tape.gradient(critic_2_loss, self.critic_2.model.trainable_variables)

        if self.max_grad_norm is not None:
            critic_1_grads, _ = tf.clip_by_global_norm(critic_1_grads, self.max_grad_norm)
            critic_2_grads, _ = tf.clip_by_global_norm(critic_2_grads, self.max_grad_norm)

        if critic_1_grads is not None:
            self.critic_1.optimizer.apply_gradients(zip(critic_1_grads, \
                                                        self.critic_1.model.trainable_variables))
        if critic_2_grads is not None:
            self.critic_2.optimizer.apply_gradients(zip(critic_2_grads, \
                                                        self.critic_2.model.trainable_variables))
        return critic_1_loss, critic_2_loss

    def update_actor(self, states):
        """ same as before """
        pass

    def train(self):
        """
        Train the agent using a batch of transitions from the replay buffer
        """
```

```
        if len(self.buffer) < self.batch_size:
            return 0, 0, 0, 0

        # Sample a batch of transitions from the replay buffer
        states, actions, rewards, next_states, dones = \
            self.buffer.sample_unpacked(self.obs_shape,
                                        self.action_shape, self.batch_size)
        # Convert to tensors
        states = tf.convert_to_tensor(states, dtype=tf.float32)
        actions = tf.convert_to_tensor(actions, dtype=tf.float32)
        rewards = tf.convert_to_tensor(rewards, dtype=tf.float32)
        next_states = tf.convert_to_tensor(next_states, dtype=tf.float32)
        dones = tf.convert_to_tensor(dones, dtype=tf.float32)

        # Update value network
        value_loss = self.update_value_network(states, next_states)

        # Update critic networks
        critic_1_loss, critic_2_loss = self.update_critic(states, actions,
                                                          rewards, next_states, dones)

        critic_loss = (critic_1_loss + critic_2_loss)/2.0

        # Update actor network
        actor_loss, alpha_loss = self.update_actor(states)

        # Update target networks
        self.update_target_networks()
        return value_loss, critic_loss, actor_loss, alpha_loss
```

Listing 8.69: SAC agent class with a separate value network

8.5.7. Training a SAC agent

The function for training a SAC agent on a given problem environment is provided in the code listing 8.70. The agent generates action as per its own policy and saves the experiences in a replay buffer.

```
def train_sac_agent(env, agent, num_episodes=1500,
                    stop_score=-200,
                    warmup_steps=1000,
                    update_per_step=1,
                    log_freq=100):

    print('Environment name: ', env.spec.id)
    print('RL Agent name:', agent.name)

    ep_scores = []
    best_score = -np.inf
    total_steps = 0
    a_loss, c_loss, alpha_loss = 0, 0, 0
    if SAC2:
        v_loss = 0
    for e in range(num_episodes):
        done = False
        truncated = False
        state = env.reset()[0]
```

```
        ep_score = 0
        ep_steps = 0
        c_losses, a_losses, alpha_losses = [], [], []
        if SAC2:
            v_losses = []
        while not done and not truncated:

            if total_steps < warmup_steps:
                action = env.action_space.sample()
            else:
                # Use the agent's policy to select an action
                action = agent.choose_action(state)

            next_state, reward, done, truncated, _ = env.step(action)
            agent.store_transition(state, action, reward, next_state, done)

            state = next_state
            ep_score += reward
            ep_steps += 1
            total_steps += 1

            # train the agent
            if total_steps >= warmup_steps:
                c_l, a_l, ap_l = agent.train(update_per_step=update_per_step)
                c_losses.append(c_l)
                a_losses.append(a_l)
                alpha_losses.append(ap_l)
        # while loop ends here - end of episode
        ep_scores.append(ep_score)
        c_loss = np.mean(c_losses)
        a_loss = np.mean(a_losses)
        alpha_loss = np.mean(alpha_losses)

        if e % log_freq == 0:
            print(f'e:{e}, ep_score:{ep_score:.2f}, avg_ep_score:{np.mean(ep_scores):.2f},\
            avg100score:{np.mean(ep_scores[-100:]):.2f}, \
                best_score:{best_score:.2f}')

        if ep_score > best_score:
            best_score = ep_score
            agent.save_weights(filename='ll_sac.weights.h5')
            print(f'Best Score: {ep_score:.2f}, episode: {e}. Model saved.')

        if np.mean(ep_scores[-100:]) > stop_score:
            print('The problem is solved in {} episodes'.format(e))
            break
    # for loop ends here
```

Listing 8.70: Function for training a SAC agent on a given problem environment

8.5.8. Solving `Pendulum-v1` Problem using SAC

The main code for training the SAC agent on `Pendulum-v1` is provided in the code listing 8.71. It is an environment with continuous action space. The training performance of the SAC agent for this environment is shown in Figure 8.1. As one can see, the problem is solved in about 100 episodes when the average of last 100 episodes (`Avg100Score`) exceeds a value of -200. The best episodic score achieved is about -0.6.

```python
import gymnasium as gym
from sac import SACAgent
if __name__ == "__main__":
    # Create the gym environment
    env = gym.make('Pendulum-v1', g=9.81)

    obs_shape = env.observation_space.shape
    action_shape = env.action_space.shape
    print(f"Observation shape: {obs_shape}, Action shape: {action_shape}")
    action_upper_bound = env.action_space.high[0]  # Assuming continuous action space
    print(f"Action upper bound: {action_upper_bound}")
    print(f"Action lower bound: {env.action_space.low}")

    # Initialize the SAC agent
    agent = SACAgent(obs_shape, action_shape,
                     action_upper_bound=action_upper_bound,
                     reward_scale=1.0,
                     buffer_size=1000000,
                     batch_size=256,
                     max_grad_norm=None)


    # train the agent
    train_sac_agent(env, agent, num_episodes=1500,
                    stop_score=-200)
```

```
Output:
Observation shape: (3,), Action shape: (1,)
Action upper bound: 2.0
Action lower bound: [-2.]
Environment name:  Pendulum-v1
RL Agent name: SAC
e:0, ep_score:-773.44, avg_ep_score:-773.44, avg100score:-773.44, best_score:-inf
Best Score: -773.44, episode: 0. Model saved.
Best Score: -662.23, episode: 50. Model saved.
Best Score: -522.63, episode: 55. Model saved.
Best Score: -388.88, episode: 56. Model saved.
Best Score: -261.68, episode: 61. Model saved.
Best Score: -132.35, episode: 62. Model saved.
Best Score: -129.37, episode: 66. Model saved.
Best Score: -129.07, episode: 68. Model saved.
Best Score: -127.75, episode: 74. Model saved.
Best Score: -126.07, episode: 76. Model saved.
Best Score: -1.27, episode: 79. Model saved.
Best Score: -1.17, episode: 83. Model saved.
e:100, ep_score:-120.53, avg_ep_score:-788.01, avg100score:-788.15, best_score:-1.17
Best Score: -1.13, episode: 109. Model saved.
Best Score: -0.98, episode: 116. Model saved.
Best Score: -0.85, episode: 122. Model saved.
The problem is solved in 154 episodes
```

Listing 8.71: Main program for solving `Pendulum-v1` using SAC

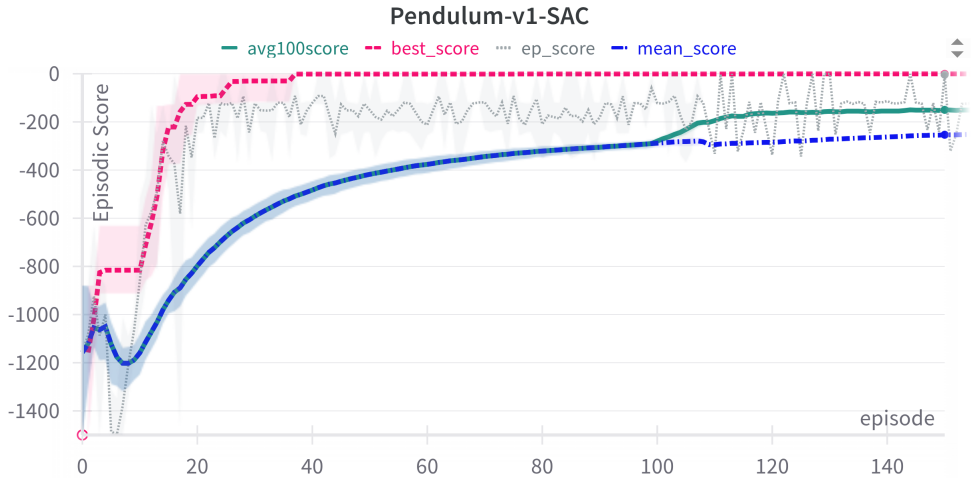


Figure 8.1.: Training performance of SAC2 agent for solving `Pendulum-v1` problem

8.5.9. Solving `LunarLander-v3-Continuous` Problem using SAC

In this version of the environment, both the observation and the action spaces are continuous. The specific details of the environment is provided in Table 8.1. An episode is considered a solution if it scores at least 200 points. The main code for training a SAC agent to solve this problem is provide in the listing 8.72. The training performance of the SAC agent on this environment is shown in Figure 8.2. As one can observe, the agent successfully solves this problem by achieving an `avg100score` of about 200 and best episodic score of about 300 by training over 350 episodes.

Table 8.1.: Parameters for `LunarLanderContinuous-v3` Environment

Variables	Shape	Type	Elements	Min	Max
Observation, state s	(8,)	Box (Continuous)	Position (x, y, θ), and velocities $(\dot{x}, \dot{y}, \dot{\theta})$, two booleans representing leg contact with ground	-10	10
Action, a	(2,)	Box (Continuous)	Throttle of left and right engines	-1	1
Reward, r	()	-	Variable reward is granted for each step	-	-

```
import gymnasium as gym
from sac2 import SACAgent
if __name__ == "__main__":
    # Create the LunarLanderContinuous-v3 environment
    env = gym.make('LunarLander-v3', continuous=True)

    obs_shape = env.observation_space.shape
    action_shape = env.action_space.shape
    print(f"Observation shape: {obs_shape}, Action shape: {action_shape}")
    action_upper_bound = env.action_space.high  # Assuming continuous action space
    print(f"Action upper bound: {action_upper_bound}")
    print(f"Action lower bound: {env.action_space.low}")

    # Initialize the SAC agent
```

```
agent = SACAgent(obs_shape, action_shape,
                 buffer_size=1000000,
                 batch_size=256,
                 action_upper_bound=action_upper_bound,
                 reward_scale=1.0,
                 lr_a=0.001, lr_c=1e-4, lr_alpha=1e-4,
                 polyak=0.995)

# train the agent
train_sac_agent(env, agent, num_episodes=1500,
                max_score=500, min_score=-300,
                stop_score=200,
                update_per_step=1,
                wandb_log=True)
```

```
Environment name:  LunarLander-v3
RL Agent name: SAC2
e:0, ep_score:-304.12, avg_ep_score:-304.12, avg100score:-304.12, best_score:-inf
Best Score: -304.11826608741774, episode: 0. Model saved.
Best Score: -64.66004638633584, episode: 2. Model saved.
Best Score: -58.68014608926175, episode: 17. Model saved.
Best Score: 232.4166849492071, episode: 29. Model saved.
Best Score: 261.54218244093954, episode: 79. Model saved.
e:100, ep_score:-58.37, avg_ep_score:-92.34, avg100score:-90.22, best_score:261.54
Best Score: 276.6843418882439, episode: 110. Model saved.
Best Score: 295.11690047683317, episode: 180. Model saved.
Best Score: 306.79686798603245, episode: 192. Model saved.
e:200, ep_score:262.25, avg_ep_score:-16.02, avg100score:61.06, best_score:306.80
e:300, ep_score:252.19, avg_ep_score:43.81, avg100score:164.06, best_score:306.80
Best Score: 314.70652883744344, episode: 330. Model saved.
The problem is solved in 358 episodes
```

Listing 8.72: Code for training SAC agent to solve `LunarLanderContinous-v3` environment

8.5.10. Solving `FetchReachDense-v3` problem with SAC

This environment was introduced in [22] and is now available with Gymnasium [23]. The task in the environment is for a manipulator to move the end effector to a randomly selected position in the robot's workspace. The robot is a 7-DoF Fetch Mobile Manipulator with a two-fingered parallel gripper. The robot is controlled by small displacements of the gripper in Cartesian coordinates and the inverse kinematics are computed internally by the MuJoCo[1] framework. The task is also continuing which means that the robot has to maintain the end effector's position for an indefinite period of time. The observation space and the action space for this environment are continuous making it a challenging problem to solve. The specific details about the environment is provided in Table 8.2. Few screenshots of the problem environment is shown in Figure 8.3. The

[1] `https://mujoco.org/`

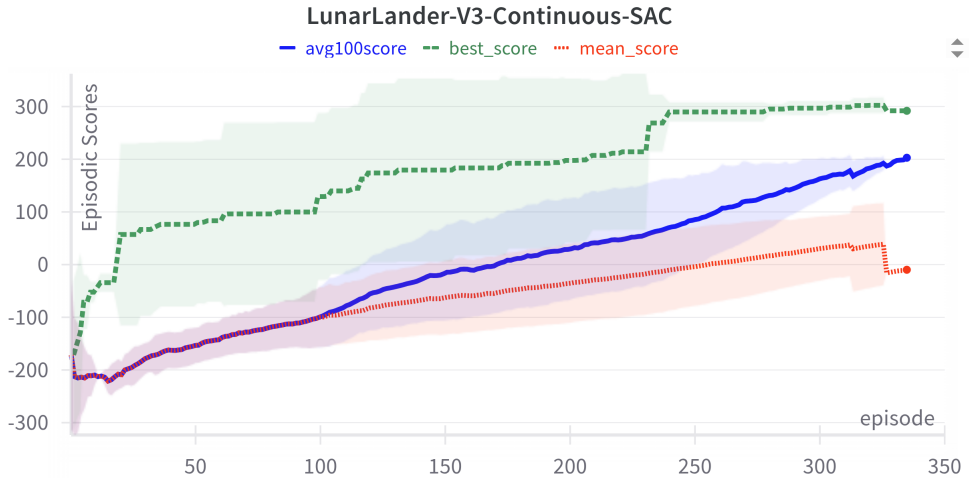


Figure 8.2.: Training performance of SAC agent on `LunarLanderContinous-v3` environment

last picture shows the case when the end-effector successfully reaches the desired goal location (shown as a red dot). The main Python code for training a SAC agent on this environment is provided in Listing 8.73. The training performance of the SAC agent on `FetchReachDense-v3` environment is shown in Figure 8.4. It can be seen that the agent achieves an `average100score` of about -19 and the best score of about -6 after training over 1400 episodes. Ideally, average episodic score should be 0 for successfully solving this problem. The corresponding loss functions of the SAC agent is shown in Figure 8.5. The value and critic losses are minimized and, actor and alpha losses are maximized during training as is being seen in this plot.

Table 8.2.: Parameters for `FetchReachDense-v3` Environment

Variables	Shape	Type	Elements	Min	Max
Observation, state s	(10,)	Box (Continuous)	End-effector position (x, y, z), and velocities $(\dot{x}, \dot{y}, \dot{z})$, left and right gripper finger position and velocities	-	-
Action, a	(4,)	Box (Continuous)	Displacement of end-effector (dx, dy, dz) and last variable controlling the opening and closing of gripper	-1	1
Reward, r	()	Continuous	Negative Euclidean distance between achieved goal and desired goal	$-\infty$	0

```
import gymnasium as gym
from sac2 import SACAgent
if __name__ == "__main__":

    # Create the LunarLanderContinuous-v3 environment
    env = gym.make('FetchReachDense-v3',
                   max_episode_steps=100, render_mode='rgb_array')

    obs_shape = env.observation_space['observation'].shape
    action_shape = env.action_space.shape
```

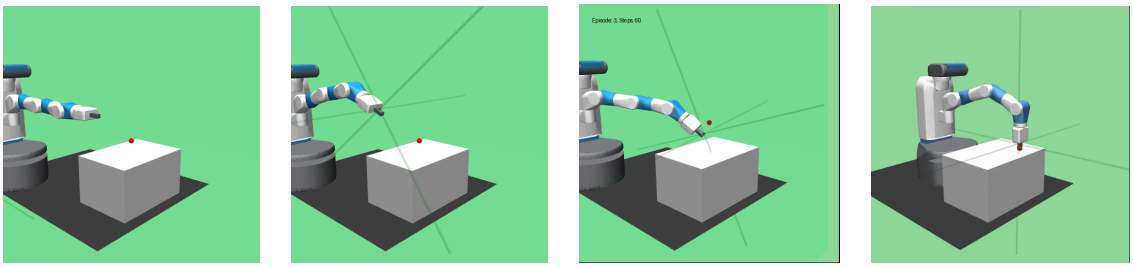


Figure 8.3.: Screenshots of a few observation states of `FetchReachDense-v3` environment. The red dot is the desired goal that the robot end-effector expected to reach for successful completion of task.

```
    print(f"Observation shape: {obs_shape}, Action shape: {action_shape}")
    action_upper_bound = env.action_space.high
    print(f"Action upper bound: {action_upper_bound}")
    print(f"Action lower bound: {env.action_space.low}")

    # Initialize the SAC agent
    agent = SACAgent(obs_shape, action_shape,
                     action_upper_bound=action_upper_bound,
                     reward_scale=2.0,
                     buffer_size=1000000,
                     batch_size=256, )

    # train the agent
    train_sac_agent(env, agent, num_episodes=1500,
                    max_score=None, min_score=None,
                    stop_score=0,
                    ep_max_steps=None,
                    update_per_step=1)
```

```
Output:
Observation shape: (10,), Action shape: (4,)
Action upper bound: [1. 1. 1. 1.]
Action lower bound: [-1. -1. -1. -1.]
Environment name:  FetchReachDense-v3
RL Agent name: SAC2
e:0, ep_score:-42.39, avg_ep_score:-42.39, avg100score:-42.39,  best_score:-inf
Best Score: -42.39, episode: 0. Model saved.
Best Score: -41.00, episode: 1. Model saved.
Best Score: -40.82, episode: 6. Model saved.
Best Score: -30.89, episode: 19. Model saved.
Best Score: -30.21, episode: 34. Model saved.
Best Score: -29.32, episode: 37. Model saved.
Best Score: -23.87, episode: 43. Model saved.
Best Score: -12.88, episode: 84. Model saved.
e:100, ep_score:-15.25, avg_ep_score:-40.83, avg100score:-40.82,  best_score:-12.88
Best Score: -10.64, episode: 124. Model saved.
```

```
Best Score: -10.37, episode: 154. Model saved.
Best Score: -8.45, episode: 179. Model saved.
Best Score: -8.21, episode: 192. Model saved.
e:200, ep_score:-20.71, avg_ep_score:-30.78, avg100score:-20.62,  best_score:-8.21
Best Score: -6.61, episode: 223. Model saved.
e:300, ep_score:-20.45, avg_ep_score:-27.05, avg100score:-19.56,  best_score:-6.61
e:400, ep_score:-10.52, avg_ep_score:-25.32, avg100score:-20.12,  best_score:-6.61
e:500, ep_score:-32.46, avg_ep_score:-24.26, avg100score:-20.01,  best_score:-6.61
e:600, ep_score:-18.83, avg_ep_score:-23.54, avg100score:-19.89,  best_score:-6.61
e:700, ep_score:-14.12, avg_ep_score:-22.76, avg100score:-18.08,  best_score:-6.61
e:800, ep_score:-17.94, avg_ep_score:-22.13, avg100score:-17.70,  best_score:-6.61
Best Score: -5.96, episode: 808. Model saved.
e:900, ep_score:-20.48, avg_ep_score:-21.76, avg100score:-18.84,  best_score:-5.96
e:1000, ep_score:-23.76, avg_ep_score:-21.43, avg100score:-18.42, best_score:-5.96
e:1100, ep_score:-27.59, avg_ep_score:-21.17, avg100score:-18.61, best_score:-5.96
e:1200, ep_score:-23.49, avg_ep_score:-20.97, avg100score:-18.79, best_score:-5.96
e:1300, ep_score:-23.06, avg_ep_score:-20.80, avg100score:-18.67, best_score:-5.96
e:1400, ep_score:-20.86, avg_ep_score:-20.64, avg100score:-18.59, best_score:-5.96
```

Listing 8.73: Main code for training SAC agent on `FetchReachDense-v3` environment

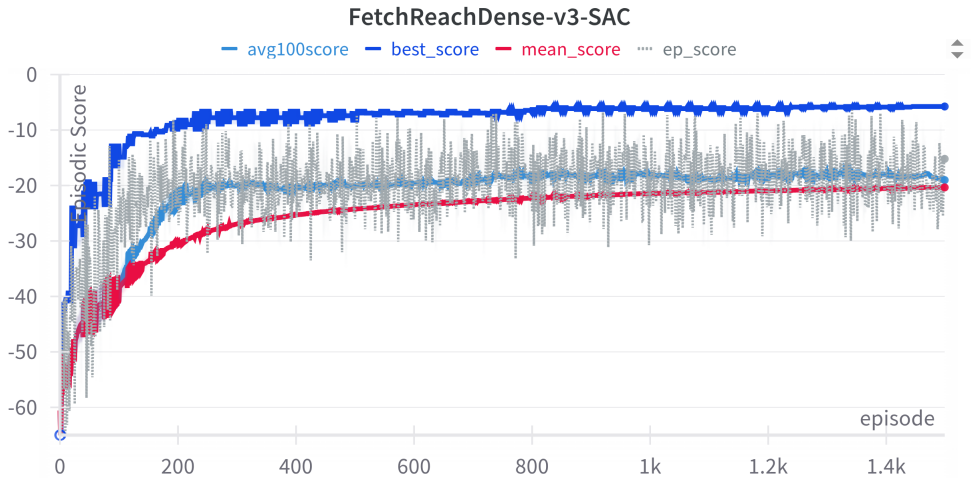


Figure 8.4.: Training performance of SAC algorithm on `FetchReachDense-v3` environment

8.5.11. Comparing SAC vs SAC2

As discussed above, two different implementations of SAC algorithm is presented. The first version, called SAC, uses an actor model, two critic models and two target critic models for training. On the other hand, SAC2 uses a separate value network and a target value network in addition to an actor model and two critic models. In terms of parameters, there is not much difference as both use 5 deep network models. In terms of performances,

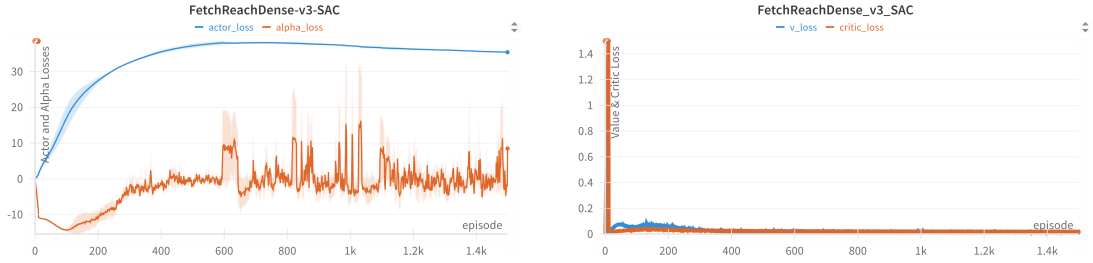


Figure 8.5.: SAC agent's soss functions: (a) Actor & Alpha losses, (b) Value & Critic losses

SAC2 performs slightly better in general as estimating value separately reduces variance and provides better training stability. The performance comparison between these two variants is shown in Figure 8.6. We see that the performance of these variants are mostly same for the `FetchReachDense-v3` environment. However, SAC2 performs better in case of `Pendulum-v1` and `LunarLanderContinuous-v3` environments.

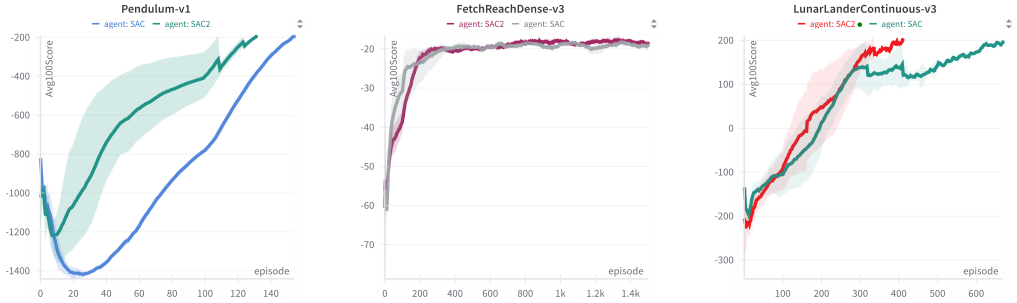


Figure 8.6.: Comparing performances of two implementations: SAC & SAC2

8.6. Conclusion

In conclusion, Soft Actor-Critic represents a sophisticated yet practical approach to reinforcement learning. By seamlessly integrating the pursuit of reward with the encouragement of exploration through entropy maximization, SAC provides a powerful, stable, and sample-efficient algorithm for tackling challenging continuous control problems. It outperforms DDPG due to better exploration and reduced bias, surpasses A2C in sample efficiency and performance in complex tasks, and often outshines PPO in continuous control due to its off-policy nature. However, it can be computationally intensive due to multiple networks and entropy calculations. We discussed two implementations of SAC algorithm - one that uses two target critic networks and the other that uses a separate value network and its corresponding target network. We demonstrate the performance of SAC algorithm in solving three benchmark problems with continuous action spaces,

namely, `Pendulum-v1`, `LunarLanderContinuous-v3` and `FetchReachDense-v3` environments. In general, we see that SAC algorithm provides superior performance compared to the algorithms discussed in previous chapters.

A. Basics of Probability & Statistics

A.1. Theorems & Definitions

Theorem A.1. *If X and Y are two jointly distributed random variables, then*

$$\mathbb{E}(X+Y) = \mathbb{E}(X) + \mathbb{E}(Y)$$

Proof.

$$\begin{aligned}
\mathbb{E}(X+Y) &= \sum_i \sum_j (x_i + y_j) p(x_i, y_j) \\
&= \sum_i \sum_j x_i p(x_i, y_j) + \sum_i \sum_j y_j p(x_i, y_j) \\
&= \sum_i x_i \sum_j p(x_i, y_j) + \sum_j y_j \sum_i p(x_i, y_j) \\
&= \sum_i x_i P(x_i) + \sum_j y_j P(y_j) \\
&= \mathbb{E}(X) + \mathbb{E}(Y)
\end{aligned}$$

■

Definition A.1. ***Bayesian Theorem of Conditional Probability:***

$$P(A,B) = P(A|B)P(B) \tag{A.1}$$

This leads to the following conditional probability relationships:

$$\begin{aligned}
P(A|B) &= P(B|A) * P(A)/P(B) \\
P(B|A) &= P(A|B) * P(B)/P(A)
\end{aligned}$$

A.2. Descriptions & Explanations

A.2.1. State-Action Marginal Visitation Probability Distribution

In the context of reinforcement learning, the **state-action marginal visitation probability distribution**, often denoted as $\rho_\pi(s,a)$, formally defines the likelihood of visiting

a specific state-action pair (s, a) over the entire lifespan of an agent following a particular policy π.

More precisely, for a given policy π and a discount factor $\gamma \in [0, 1)$, it is defined as:

$$\rho_\pi(s, a) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s, a_t = a \mid s_0, \pi) \tag{A.2}$$

Where:

- s_t and a_t are the state and action at time step t, respectively.
- $P(s_t = s, a_t = a \mid s_0, \pi)$ is the probability of being in state s and taking action a at time t, given an initial state s_0 and following policy π. This probability inherently depends on the environment's transition dynamics $P(s_{t+1} \mid s_t, a_t)$ and the policy $\pi(a_t \mid s_t)$.
- The sum over t (time steps) accounts for the long-term visitation.
- The discount factor γ^t gives more weight to state-action pairs visited earlier in an episode, reflecting the standard discounted return objective. In some contexts, particularly for average reward settings, γ^t might be omitted or replaced with a uniform probability over time steps.

In simpler terms, $\rho_\pi(s, a)$ tells you, on average, how frequently (or with what probability density for continuous spaces) the agent will encounter a specific state s and then choose action a when it consistently behaves according to policy π over an extended period. It is a fundamental concept for understanding the expected behavior and performance of a policy.

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] James R Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[3] Burrhus F Skinner. Operant conditioning. *The encyclopedia of education*, 7:29–33, 1971.

[4] Brian Seamus Haney. Applied artificial intelligence in modern warfare and national security policy. *Hastings Sci. & Tech. LJ*, 11:61, 2020.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[6] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[7] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[8] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[9] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[10] Farama Foundation. Gymnasium. `https://gymnasium.farama.org/`.

[11] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[12] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[13] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[14] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992.

[15] Ross A Maller, Gernot Müller, and Alex Szimayer. Ornstein–uhlenbeck processes and extensions. *Handbook of financial time series*, pages 421–437, 2009.

[16] John Schulman. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.

[17] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 267–274, 2002.

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[19] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[20] Mohit Sewak. Actor-critic models and the a3c: The asynchronous advantage actor-critic model. In *Deep reinforcement learning: frontiers of artificial intelligence*, pages 141–152. Springer, 2019.

[21] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[22] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.

[23] Gymnasium-Robotics. Fetch reach gym environment. `https://robotics.farama.org/envs/fetch/reach/`.

Alphabetical Index