AI 3603 Artificial Intelligence: Principles and Techniques

By: Luoyun Wang (521030910067)

HW#: 2 Reinforcement Learning

November 11, 2023

I. REINFORCEMENT LEARNING IN CLIFF-WALKING ENVIRONMENT

A. Introduction

In Figure 1, there is a maze in the shape of a 12x4 grid. The objective is to find a secure path from the bottom left corner to the bottom right corner. The bottom row of the grid represents treacherous cliffs, where moving incurs a substantial penalty (-100), while the rest of the grid consists of safe grassy areas. Moving in any direction within the grid comes with a minor cost (-1) per step. The goal is to determine the path with the lowest total cost, making it the optimal choice.

The objective is to develop an agent capable of navigating the maze using various algorithms as its learning approach and subsequently evaluate the training outcomes.

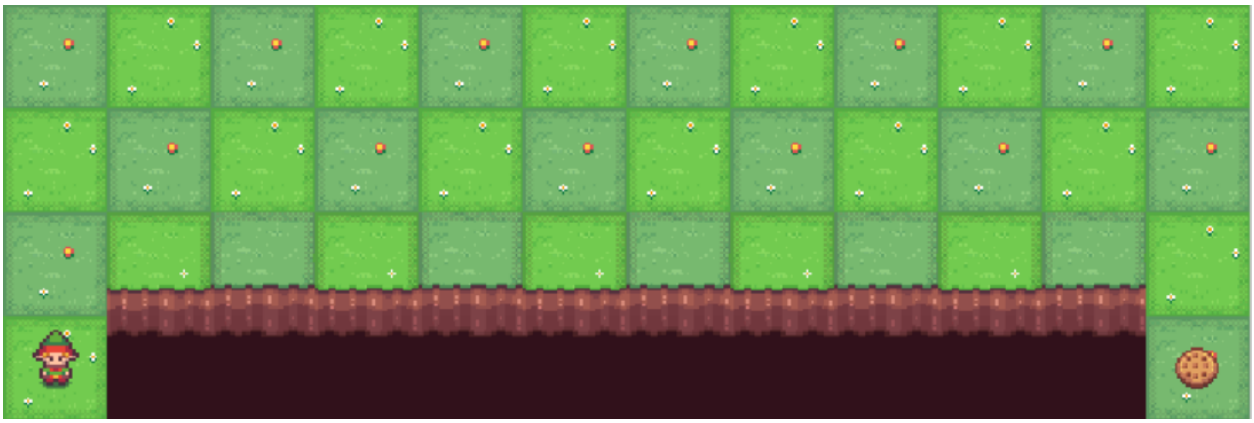


FIG. 1: The cliff-walking environment

B. Q-learning methods description and implementation

1. *Description*

Q-Learning is a reinforcement learning algorithm employed to tackle problems in which an agent must make a series of decisions within an environment to maximize its cumulative rewards. The fundamental concept behind Q-Learning involves the construction of a Q-table, which encapsulates the quality or the anticipated cumulative reward associated with taking a particular action in a specific state. This empowers the agent to make well-informed decisions, opting for actions that are poised to maximize its expected future rewards.

2. *Formulation*

- *State (S)*: The environment is divided into a set of states, each representing the current coordinates (x, y) of the agent.
- *Action (A)*: Within each state, the agent possesses a range of available actions, identified by four integers that denote the four cardinal directions for movement.

- *Q-Table (Q)*: The Q-table is a two-dimensional matrix, with rows corresponding to states and columns corresponding to actions. It serves as a repository for the expected cumulative rewards associated with each state-action pair. Initially, Q-values are commonly initialized with arbitrary values or set to zero.
- *Q-Learning Algorithm*: Q-Learning is an off-policy reinforcement learning algorithm employed to solve problems in which an agent must make a sequence of decisions in an environment to maximize cumulative rewards. The primary objective of Q-Learning is to establish a policy that equips the agent with the ability to make informed decisions while simultaneously updating state-action values (Q-values) to determine the optimal action for each state. The Q-Learning update rule is as follows:

$$Q(S, A) \longleftarrow Q(S, A) + \alpha * [R + \lambda * Q(S', a) - Q(S, A)]$$

  where:

  - $Q(S, A)$ represents the current Q-value for state S and action A.
  - α (learning rate) governs the extent to which the new information is integrated into the Q-values.
  - R denotes the immediate reward acquired after taking action A in state S and transitioning to state S'.
  - λ (discount factor) reflects the agent's inclination towards immediate rewards versus delayed rewards.
  - $\max(Q(S', a))$ signifies the **highest Q-value** for the next state S' among all possible actions a.
- *Learning Rate (α)*: This parameter controls the weight of newly acquired information in updating Q-values following each interaction with the environment. A smaller α leads to more gradual learning but may result in a more stable policy.
- *Discount Factor (λ)*: The discount factor illustrates the agent's prioritization of immediate rewards over delayed rewards. A lower λ steers the agent towards short-term rewards, while a higher value encourages consideration of long-term rewards.
- *Exploration-Exploitation Strategy (ϵ-greedy)*: Balancing exploration and exploitation is essential for the agent's decision-making. It employs the ϵ-greedy strategy, wherein it explores (selecting a random action with probability ϵ) and exploits (choosing the action with the highest Q-value with a probability of $1 - \epsilon$).

3. Implementation

- *Initialization*: The agent is initialized with a set of parameters, which includes the available actions, a learning rate ($\alpha = 0.3$), a discount factor ($\gamma = 0.9$), an initial value for epsilon ($\epsilon = 1.0$), and a rate of epsilon decay (ϵ-decay factor $= 0.999$). The Q-table is also initialized with all zero values.
- *Choose Action*: This method is responsible for action selection by the agent, adhering to the ϵ-greedy strategy. It chooses a random action from the complete set of actions with a probability of ϵ, while in the remaining cases, it exploits the action with the highest Q-value associated with the current state.
- *Learn*: After taking an action and receiving a reward, the agent updates its Q-values by considering the observed reward and the maximum Q-value for the subsequent state. This is achieved using the Q-Learning update formula, which adjusts the Q-value for the current state-action pair.
- *Exploration-Exploitation Strategy (ϵ-greedy)*: The agent gradually reduces the exploration rate ϵ over time by following an exponential ϵ-decay schedule. This approach promotes increased exploitation (selecting the action with the highest Q-value for the given state) as the learning process advances.

C. Sarsa methods description and implementation

1. Description

Sarsa is a reinforcement learning algorithm utilized to tackle problems where an agent needs to make a series of decisions in an environment to maximize its cumulative reward. Like Q-Learning, Sarsa is designed to learn a policy that enables the agent to make informed decisions. However, in contrast to Q-Learning, Sarsa directly focuses on learning action-values for state-action pairs and takes into account the agent's existing policy when selecting actions.

2. Formulation

- *State (S)*: The environment is divided into a set of states, with each state representing the current coordinates (x, y) of the agent.
- *Action (A)*: Within each state, the agent has a set of available actions, indicated by four integers that correspond to four directional movements.
- *Sarsa Algorithm*: Sarsa is an on-policy algorithm, which means it learns in accordance with the policy it is currently following. It maintains a Q-table, similar to Q-Learning, where expected cumulative rewards for each state-action pair are stored. The algorithm iteratively interacts with the environment and updates the Q-values using the Sarsa Update Rule:
- *Sarsa Update Rule*: After executing an action, observing the next state, receiving a reward, and selecting the subsequent action, the Q-value for the current state-action pair is updated as follows:

$$Q(S, A) \longleftarrow Q(S, A) + \alpha * [R + \lambda * Q(S', A') - Q(S, A)]$$

  Where:

  - $Q(S, A)$ represents the current Q-value for state S and action A.
  - α (learning rate) regulates the extent to which new information is incorporated.
  - R signifies the immediate reward received after taking action A in state S and transitioning to state S'.
  - λ (discount factor) reflects the agent's preference for immediate rewards over delayed rewards.
  - $Q(S', A')$ denotes the **Q-value for the next action** A' and the next state S'.

- *Exploration-Exploitation Strategy (ϵ-greedy)*: Striking a balance between exploration and exploitation is crucial for the agent's decision-making process. The agent employs the ϵ-greedy strategy, which entails exploration (choosing a random action with probability ϵ) and exploitation (selecting the action with the highest Q-value with a probability of $1 - \epsilon$).

3. Implementation

- *Initialization*: The agent is initialized with a set of parameters, which include the available actions, a learning rate ($\alpha = 0.3$), a discount factor ($\lambda = 0.9$), an initial value for epsilon ($\epsilon = 1.0$), and a rate of epsilon decay (ϵ-decay factor $= 0.999$). The Q-table is also initialized with all zero values.
- *Choose Action*: This method is responsible for action selection by the agent, following the ϵ-greedy strategy. It chooses a random action from the complete set of actions with a probability of ϵ, while in other cases, it exploits the action with the highest Q-value associated with the current state.

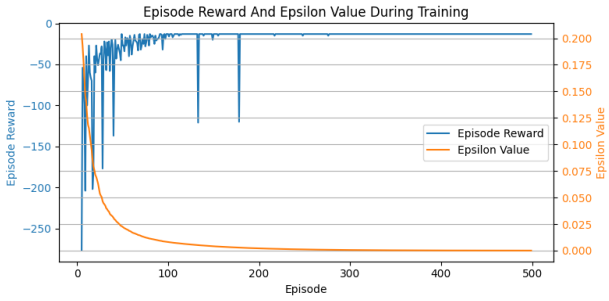
- *Learn*: After executing an action, observing the next state, receiving a reward, and selecting the subsequent action, the agent updates its Q-values using the Sarsa update rule. This update considers both the current state-action pair and the next state-action pair.
- *Exploration-Exploitation Strategy (ϵ-greedy)*: The agent gradually reduces the exploration rate ϵ over time by following an exponential ϵ-decay schedule. This approach encourages a transition towards increased exploitation, where the action with the highest Q-value for the given state is selected, as the learning process progresses.

D. Result visualization and analysis

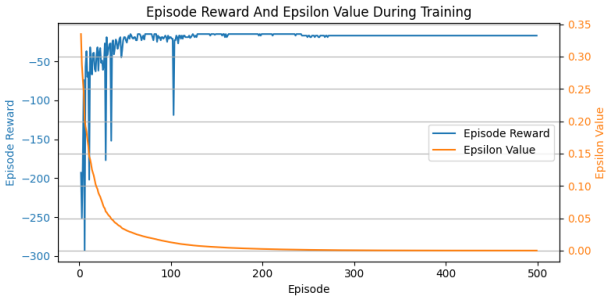
1. *Result visualization*

	Episode reward	Epsilon value
Q-learning	-13	0.000156
Sarsa	-17	0.000084

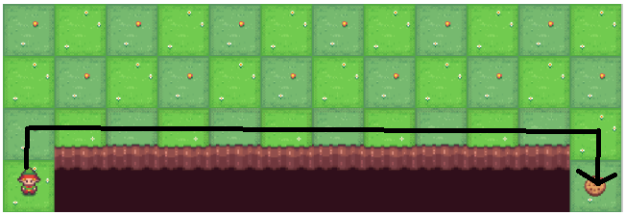
TABLE I: The final values of two methods



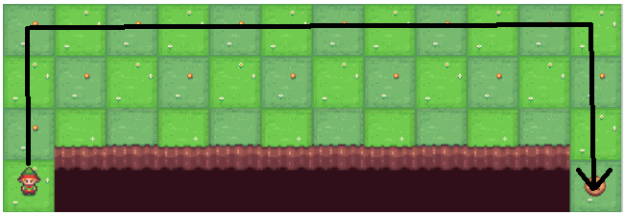
(a) Episode reward and epsilon value of Q-Learning



(b) Episode reward and epsilon value of Sarsa



(c) final paths of Q-Learning



(d) final paths of Sarsa

FIG. 2: Images of values and paths

The videos of final training results about two algorithms are in **videos** folder.

2. Result analysis

The results obtained for the Q-Learning and Sarsa algorithms, with learning rates approximately around 0.3, indicate that the convergence of the reward typically occurs after approximately 200 episodes and 150 episodes, respectively. Initially, in the experiments, learning rates within the range of 0.8 to 1.0 were used, resulting in relatively fast convergence, but with much less stable convergence curves. As for the value function's curve, it exhibits exponential decay, reaching values of 0.008 and 0.012 within 100 episodes, respectively. Subsequently, throughout the training process, the agent predominantly engages in exploitation rather than exploration, thus achieving policy convergence.

Based on the results of the episode rewards during the training process, it appears that Q-Learning exhibits larger fluctuations in the early rounds compared to the Sarsa algorithm. One possible reason for this is the differences in exploration strategy: In Q-Learning, the deterministic choice of the action with the highest Q-value may lead to the premature selection of a suboptimal action in the early stages, resulting in larger fluctuations. Conversely, Sarsa employs a certain exploration probability to randomly select actions, making it more likely to explore a variety of actions in the initial stages, thus promoting stability, albeit at the potential cost of slower convergence to the optimal policy.

The reasons for these differences can be attributed to:

1. On-Policy or Off-Policy: Sarsa is an on-policy algorithm, which means it learns and updates its Q-values while following the current policy. It takes actions based on its current policy, which includes both exploration and exploitation. In contrast, Q-Learning is an off-policy algorithm, learning the value of the optimal policy while taking actions based on a different policy, often a more exploratory one. This can lead to significant differences in the paths chosen.

2. Stability and Convergence: Q-Learning often converges more quickly to an optimal policy compared to Sarsa, especially in deterministic environments. The faster convergence can affect the paths found during early training stages, as Q-Learning may begin exploiting the optimal path sooner.

3. Exploration Strategies: Sarsa's conservative exploration might lead to more straightforward, less risky paths that steer clear of cliffs, while Q-Learning's more exploratory approach can lead to finding novel routes.

   These insights help shed light on the variations in agent behavior and reward convergence observed during the training process for these two reinforcement learning algorithms.

II. DEEP REINFORCEMENT LEARNING

A. Introduction

The environment depicted in Figure 3 involves the objective of guiding a space-ship to decelerate and safely land between two flags by utilizing thrusters to maneuver in the left, right, or downward directions. Rewards are associated with factors such as oil usage and the accuracy of the landing position.

This task is accomplished through the training of a Deep Q-Network (DQN) using reinforcement learning. The provided code includes the implementation of a DQN and the training methodology. The main task is to comprehend the code, understand its functioning, and make parameter adjustments to enhance the landing outcome.

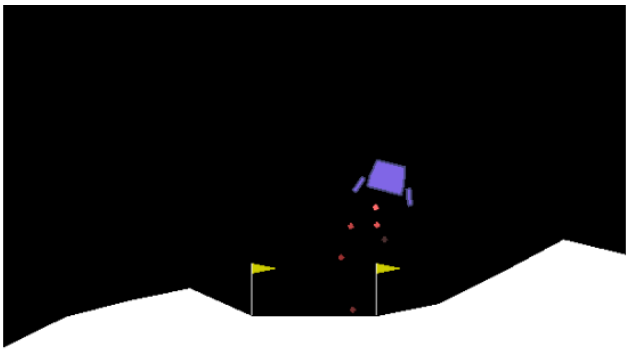


FIG. 3: The cliff-walking environment

B. Code comprehending and commenting

The codes of DQN agent is given and commenting them with one's own comprehension have to be done in this task. The result is shown in this section, and the same code file can be found in **dqn.py**.

```
# -*- coding:utf-8 -*-
import argparse
import os
import random
import time

import gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from stable_baselines3.common.buffers import ReplayBuffer
from torch.utils.tensorboard import SummaryWriter

def parse_args():
    """parse arguments. You can add other arguments if needed."""
    parser = argparse.ArgumentParser()
    parser.add_argument("--exp-name", type=str, default=os.path.basename(__file__).rstrip(".py"),
        help="the name of this experiment")
    parser.add_argument("--seed", type=int, default=42,
        help="seed of the experiment")
    parser.add_argument("--total-timesteps", type=int, default=500000,
        help="total timesteps of the experiments")
    parser.add_argument("--learning-rate", type=float, default=3e-4,
        help="the learning rate of the optimizer")
    parser.add_argument("--buffer-size", type=int, default=10000,
        help="the replay memory buffer size")
    parser.add_argument("--gamma", type=float, default=0.99,
        help="the discount factor gamma")
    parser.add_argument("--target-network-frequency", type=int, default=500,
        help="the timesteps it takes to update the target network")
    parser.add_argument("--batch-size", type=int, default=128,
        help="the batch size of sample from the reply memory")
    parser.add_argument("--start-e", type=float, default=0.5,
        help="the starting epsilon for exploration")
    parser.add_argument("--end-e", type=float, default=0.05,
        help="the ending epsilon for exploration")
    parser.add_argument("--exploration-fraction", type=float, default=0.2,
        help="the fraction of `total-timesteps` it takes from start-e to go end-e")
    parser.add_argument("--learning-starts", type=int, default=10000,
        help="timestep to start learning")
    parser.add_argument("--train-frequency", type=int, default=1,
        help="the frequency of training")
    args = parser.parse_args()
    args.env_id = "LunarLander-v2"
    return args

def make_env(env_id, seed):
    """construct the gym environment"""
    env = gym.make(env_id)
    env = gym.wrappers.RecordEpisodeStatistics(env)
    env.seed(seed=seed)
    env.action_space.seed(seed)
    env.observation_space.seed(seed)
    return env

class QNetwork(nn.Module):
    """comments: the network with input of observation and output of action"""
    def __init__(self, env):
```

```
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(np.array(env.observation_space.shape).prod(), 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, env.action_space.n),
        )

    def forward(self, x):
        return self.network(x)

def linear_schedule(start_e: float, end_e: float, duration: int, t: int):
    """comments: the linear decay schedule of epsilon for exploration"""
    slope = (end_e - start_e) / duration
    return max(slope * t + start_e, end_e)

if __name__ == "__main__":

    """parse the arguments"""
    args = parse_args()
    run_name = f"{args.env_id}__{args.exp_name}__{args.seed}__{int(time.time())}"

    """we utilize tensorboard yo log the training process"""
    writer = SummaryWriter(f"runs/{run_name}")
    writer.add_text(
        "hyperparameters",
        "|param|value|\n|-|-|\n%s" % ("\n".join([f"|{key}|{value}|" for key, value in vars(args).items()])),
    )

    """comments: set the seed for reproducibility and initialize the running environment"""
    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.backends.cudnn.deterministic = True
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    """comments: construct the gym environment"""
    envs = make_env(args.env_id, args.seed)

    """comments: initialize the Qnetwork, optimizer and target network(now same as Qnetwork)"""
    q_network = QNetwork(envs).to(device)
    optimizer = optim.Adam(q_network.parameters(), lr=args.learning_rate)
    target_network = QNetwork(envs).to(device)
    target_network.load_state_dict(q_network.state_dict())

    """comments: initialize the replay memory"""
    rb = ReplayBuffer(
        args.buffer_size,
        envs.observation_space,
        envs.action_space,
        device,
        handle_timeout_termination=False,
    )

    """comments: reset the environment and start training"""
    obs = envs.reset()
    for global_step in range(args.total_timesteps):

        """comments: calculate the epsilon for exploration"""
        epsilon = linear_schedule(args.start_e, args.end_e, args.exploration_fraction * args.total_timesteps,
            global_step)

        """comments: use epsilon-greedy to select action"""
        if random.random() < epsilon:
            actions = envs.action_space.sample()
        else:
            q_values = q_network(torch.Tensor(obs).to(device))
            actions = torch.argmax(q_values, dim=0).cpu().numpy()

        """comments: update the next state, reward, whether done and other information"""
        next_obs, rewards, dones, infos = envs.step(actions)
        # envs.render() # close render during training

        if dones:
            print(f"global_step={global_step},_episodic_return={infos['episode']['r']}")
            writer.add_scalar("charts/episodic_return", infos["episode"]["r"], global_step)
            writer.add_scalar("charts/episodic_length", infos["episode"]["l"], global_step)

        """comments: add the transition to the replay memory"""
        rb.add(obs, next_obs, actions, rewards, dones, infos)

        """comments: update the next observation"""
        obs = next_obs if not dones else envs.reset()

        if global_step > args.learning_starts and global_step % args.train_frequency == 0:

            """comments: sample a batch of transitions from the replay memory"""
            data = rb.sample(args.batch_size)

            """comments: calculate the TD target in target network and q value in Qnetwork, then calculate the loss
                """
            with torch.no_grad():
                target_max, _ = target_network(data.next_observations).max(dim=1)
                td_target = data.rewards.flatten() + args.gamma * target_max * (1 - data.dones.flatten())
            old_val = q_network(data.observations).gather(1, data.actions).squeeze()
            loss = F.mse_loss(td_target, old_val)

            """comments: if global_step is a multiple of 100, log the loss and q value"""
            if global_step % 100 == 0:
                writer.add_scalar("losses/td_loss", loss, global_step)
                writer.add_scalar("losses/q_values", old_val.mean().item(), global_step)
```

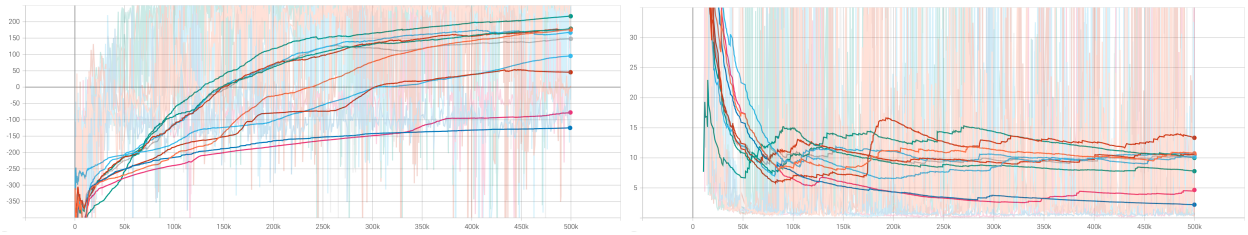
```
                """comments: update the Qnetwork"""
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                """comments: if global_step is a multiple of target_network_frequency, update the target network"""
                if global_step % args.target_network_frequency == 0:
                    target_network.load_state_dict(q_network.state_dict())

    """close the env and tensorboard logger"""
    envs.close()
    writer.close()

    torch.save(q_network.state_dict(), 'trained_model.pth')
```

C. Result of DQN training

In pursuit of improved results, I engaged in parameter tuning for the agent and eventually identified a relatively effective combination. The ensuing tables and graphs illustrate the experimented parameters and their impact by showcasing the reward and loss.

Learning rate	4e-4	4e-4	4e-4	4e-4	4e-4	4e-4	4e-4	5e-4	3e-4	2.5e-4
Gamma (γ)	0.99	0.9	0.999	0.99	0.99	0.99	0.99	0.99	0.99	0.99
Start-ϵ	0.5	0.5	0.5	0.7	0.5	0.5	0.5	0.5	0.5	0.5
End-ϵ	0.05	0.05	0.05	0.05	0.01	0.05	0.05	0.05	0.05	0.05
Exploration fraction	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.2
Train frequency	5	5	5	5	5	1	1	1	1	1
Reward	175.9	-124.7	45.48	95.33	-77.51	178.5	178.6	166.7	217	147.8
Loss	10.74	2.213	13.33	10.16	4.656	9.994	10.57	10.52	7.788	10.41

TABLE II: Final rewards and loss with different parameters



(a) Rewards curve in training process

(b) Loss curve in training process

FIG. 4: Curve graphs of rewards and loss

As evident from the results, a lower γ or reducing the lower bound of ϵ tends to decrease the loss but also leads to a sharp reduction in rewards. After numerous iterations, the reward was successfully boosted to 217, a highly satisfactory achievement. Consequently, for testing purposes, I employed this well-trained model to guide the space-ship in landing continuously for 15 iterations, yielding positive outcomes. The corresponding video is available in the **video** folder.

D. Improvement of DQN agent

In DQN, a notable drawback is its relatively limited capability to handle highly complex environments and large-scale action spaces. Specifically, when the action space is large, the estimation and update of the Q-value function become complex, leading to reduced training efficiency and even potential instability in the learning process.

To overcome this limitation, various improvement methods have been proposed, with one major enhancement being the use of the Deep Deterministic Policy Gradient (DDPG) algorithm. The Deep Deterministic Policy Gradient (DDPG) algorithm combines deep learning with deterministic policy gradient methods to effectively handle continuous action spaces. Here are the key components of the DDPG algorithm:

1. *Deterministic Policy*: DDPG uses a deterministic policy, where the policy network directly outputs continuous actions rather than sampling from a probability distribution. The deterministic policy $\mu(s;\theta^{\mu})$ is represented by a deep neural network, where s is the state, and θ^{μ} is the parameter of the policy network.

2. *Q-Value Functions*: DDPG maintains two Q-value functions, $Q(s,a;\theta^{Q})$ and $Q'(s',\mu'(s';\theta^{\mu'});\theta^{Q'})$. Here, Q is the Q-value function used for updating the policy, and Q' is the target Q-value function. Both are represented by deep neural networks, with parameters θ^{Q} and $\theta^{Q'}$ respectively.

3. *Experience Replay*: DDPG introduces an experience replay mechanism, storing previous experiences in a replay buffer. During training, small batches of experiences are randomly sampled from the buffer to update the parameters of the neural networks, enhancing sample independence and stability.

4. *Target Networks*: To improve training stability, DDPG employs target networks. The parameters $\theta^{Q'}$ and $\theta^{\mu'}$ of the target networks are gradually updated through a soft update to slow down parameter changes.

The update steps for the DDPG algorithm are as follows:

1. *Update Policy Network*:

$$\nabla_{\theta^{\mu}} J \approx \mathbb{E}_{s_t\sim\rho,a_t\sim\mu(s_t)}\left[\nabla_{\theta^{\mu}} Q(s,a;\theta^{Q})|_{s=s_t,a=\mu(s_t)}\right]\nabla_a \mu(s;\theta^{\mu})|_{s=s_t}$$

2. *Update Q-Value Functions*:

$$\nabla_{\theta^{Q}} J \approx \mathbb{E}_{s_t,a_t\sim\rho,s_{t+1}\sim\mathcal{E}}\left[\Big(Q(s,a;\theta^{Q}) - (r+\gamma Q'(s',\mu'(s';\theta^{\mu'});\theta^{Q'}))\Big)\nabla_{\theta^{Q}} Q(s,a;\theta^{Q})|_{s=s_t,a=a_t}\right]$$

3. *Soft Update Target Networks*:

$$\theta^{Q'} \longleftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \longleftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

Here, J represents the performance metric of the policy, ρ is the experience distribution in the replay buffer, $\mathcal{E}$ is the environment, r is the immediate reward, γ is the discount factor, and τ is the soft update parameter. These formulas summarize the core concepts and update processes of the DDPG algorithm.