Reinforcement Learning (RL)

Overview

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment. The agent's goal is to maximize cumulative rewards over time. Unlike supervised learning, which relies on labeled input/output pairs, RL focuses on learning from the consequences of actions through trial and error, making it particularly suitable for tasks where the correct actions are not known beforehand.

Key Concepts

- **Agent**: The learner or decision-maker that interacts with the environment.
- **Environment**: The external system with which the agent interacts, providing states and rewards in response to the agent's actions.
- **State (s)**: A representation of the current situation of the agent within the environment.
- **Action (a)**: A set of all possible moves or decisions the agent can make.
- **Reward (r)**: Feedback from the environment to evaluate the action taken, guiding the agent towards the goal.
- **Policy (π)**: A strategy used by the agent to determine the next action based on the current state.
- **Value Function (V)**: A function that estimates the expected cumulative reward from a given state, guiding the agent in value-based methods.
- **Q-Value (Q)**: A function that estimates the expected cumulative reward from a given state-action pair, often used in action-value methods.

Types of RL

- **Model-Free RL**: The agent learns a policy directly from interactions with the environment without explicitly modeling the environment's dynamics.
  - **Examples**: Q-Learning, SARSA, Deep Q-Network (DQN), Deep Deterministic Policy Gradient (DDPG)
- **Model-Based RL**: The agent builds a model of the environment's dynamics and plans actions using this model.
  - **Examples**: Dynamic Programming, Monte Carlo Methods, Model Predictive Control (MPC)

Deep Deterministic Policy Gradient (DDPG)

Overview

Deep Deterministic Policy Gradient (DDPG) is a model-free, off-policy algorithm used in reinforcement learning for solving continuous action spaces. DDPG combines elements from both Q-Learning and Policy Gradient methods, leveraging neural networks to approximate the policy and value functions.

Core Concepts

Actor-Critic Architecture

- **Actor Network**: This neural network outputs the action to be taken given a state. It represents the policy (π) and is responsible for selecting actions.
- **Critic Network**: This neural network evaluates the action taken by the actor by estimating the Q-value. It represents the value function (Q) and helps in updating the actor.

Target Networks

- To stabilize training, DDPG maintains two additional networks, known as target networks, for both the actor and critic. These target networks are slowly updated to track the learned networks, mitigating the issue of unstable learning that arises from rapidly changing target values.

Experience Replay

- DDPG uses an experience replay buffer to store the agent's experiences (state, action, reward, next state). During training, it samples random batches from this buffer to break the correlation between consecutive samples, leading to more stable and efficient learning.

Detailed Explanation of DDPG

1. **Initialization**:
   - Initialize the actor network (π) and critic network (Q) with random weights.
   - Initialize corresponding target networks (π' and Q') with the same weights as the original networks.
   - Initialize the replay buffer to store experiences.
2. **Interaction with the Environment**:
   - At each time step, the agent selects an action using the actor network. To encourage exploration, noise is added to the action.
   - The environment provides the next state and reward based on the action taken.
   - The experience (state, action, reward, next state) is stored in the replay buffer.
3. **Training the Networks**:
   - Sample a batch of experiences from the replay buffer.
   - Compute the target Q-value using the critic target network and the target actor network.
   - Update the critic network by minimizing the loss between the predicted Q-values and the target Q-values.
   - Update the actor network by maximizing the expected return using the critic network's evaluations.
   - Periodically update the target networks using a soft update mechanism, ensuring they gradually track the learned networks.

Advantages

- **Continuous Action Space Handling**: Unlike traditional Q-learning, DDPG can effectively handle environments with continuous action spaces, making it suitable for robotic control and other complex tasks.
- **Stabilized Learning**: The use of target networks and experience replay buffers helps stabilize the training process, reducing the variance in updates and improving learning efficiency.
- **Efficient Exploration**: By adding noise to the action selection process, DDPG balances exploration and exploitation, allowing the agent to discover better policies over time.

Limitations

- **Hyperparameter Sensitivity**: DDPG's performance can be highly sensitive to the choice of hyperparameters such as learning rates, batch sizes, and the noise process used for exploration.
- **Exploration Challenges**: Despite the added noise, DDPG can still suffer from inadequate exploration, particularly in environments with sparse rewards or deceptive reward structures.
- **Sample Efficiency**: Although more sample-efficient than some policy gradient methods, DDPG still requires a large number of samples to learn effectively, which can be computationally expensive.
- **Overestimation Bias**: Similar to Q-learning, DDPG can suffer from overestimation bias in the Q-value function, potentially leading to suboptimal policies and slower convergence.

DDPG is a powerful and flexible algorithm for solving continuous control tasks in reinforcement learning. By combining the strengths of value-based and policy-based methods, it effectively learns in environments where discrete action spaces are insufficient. However, its performance is contingent on careful tuning and handling of exploration challenges, making it essential to understand and address its limitations for successful application.

Theory Behind DQN

Reinforcement Learning Basics

In RL, an agent interacts with an environment, learning to perform actions to maximize cumulative rewards. The interaction is typically modeled as a Markov Decision Process (MDP), which is defined by:

- A set of states (S)
- A set of actions (A)
- A transition function (P(s'|s,a)) that defines the probability of reaching state (s') from state (s) after taking action (a)
- A reward function (R(s,a)) that provides the reward received after taking action (a) in state (s)
- A discount factor (\gamma \in [0,1]) that models the importance of future rewards

The goal is to find a policy (\pi(a|s)) that maximizes the expected cumulative reward.

Q-Learning

Q-learning is a model-free RL algorithm that learns the value of state-action pairs, represented by the Q-function (Q(s, a)). The Q-function is updated using the Bellman equation:

[Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]]

where (\alpha) is the learning rate, (r) is the reward received, (s') is the next state, and (\gamma) is the discount factor.

Deep Q-Learning

In large state and action spaces, storing and updating a Q-value for each state-action pair becomes infeasible. DQN addresses this by approximating the Q-function with a neural network, (Q(s, a; \theta)), where (\theta) are the parameters of the network.

Core Concepts of DQN

Neural Network for Q-Function Approximation

DQN uses a neural network to approximate the Q-function. The network takes the state (s) as input and outputs Q-values for all possible actions. The network parameters (\theta) are updated to minimize the loss function:

[L(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right]]

where (\theta^-) are the parameters of a target network, which is a copy of the Q-network updated periodically to stabilize training.

Experience Replay

To break the correlation between consecutive samples and improve sample efficiency, DQN uses an experience replay buffer (\mathcal{D}). The agent's experiences ((s, a, r, s')) are stored in the buffer, and mini-batches of experiences are randomly sampled to update the network.

Target Network

To stabilize training and prevent oscillations, DQN uses a target network with parameters (\theta^-), which are updated less frequently (e.g., every fixed number of steps) than the main network parameters (\theta). The target network is used to compute the target Q-value in the loss function.

Limitations of DQN

1. **Stability and Convergence**: Despite using target networks and experience replay, DQN can still suffer from instability and divergence during training.
2. **Sample Efficiency**: DQN can be sample-inefficient, requiring a large number of experiences to learn effective policies.

3. **Overestimation Bias**: The max operator in the Bellman equation can lead to overestimation of Q-values, especially in environments with stochastic rewards or transitions.

Advantages of DQN

1. **Scalability**: By approximating the Q-function with a neural network, DQN can handle problems with large and high-dimensional state spaces.
2. **Generalization**: The use of deep learning allows DQN to generalize learning across similar states, improving performance in complex environments.
3. **Off-Policy Learning**: DQN is an off-policy algorithm, meaning it can learn from experiences generated by a different policy (e.g., an exploratory policy), making it flexible and efficient in utilizing collected data.

Algorithm Outline

Here is an outline of the DQN algorithm:

1. Initialize the replay buffer (\mathcal{D})
2. Initialize the Q-network with random weights (\theta)
3. Initialize the target network with weights (\theta^- = \theta)
4. For each episode:
   - Initialize the state (s)
   - For each step:
     - With probability (\epsilon), select a random action (a), otherwise select (a = \arg\max_{a} Q(s, a; \theta))
     - Execute action (a), observe reward (r) and next state (s')
     - Store transition ((s, a, r, s')) in replay buffer (\mathcal{D})
     - Sample a random mini-batch of transitions ((s_j, a_j, r_j, s'_j)) from (\mathcal{D})
     - Compute the target Q-value: (y_j = r_j + \gamma \max_{a'} Q(s'_j, a'; \theta^-))
     - Update the Q-network by minimizing the loss: (L(\theta) = \frac{1}{N} \sum_j \left(y_j - Q(s_j, a_j; \theta) \right)^2)
     - Periodically update the target network: (\theta^- = \theta)

DQN represents a significant advancement in RL by effectively combining Q-learning with deep neural networks. It enables agents to learn policies for complex tasks with large state and action spaces. However, it also comes with challenges such as instability and sample inefficiency, which have led to further research and improvements in deep reinforcement learning techniques.

CODE FOR INVERSE PENDULUM PROBLEM USING DQN.

We will be using Deep Q-Netwok algorithm for the inverse pendulum problem.

1. Importing Libraries

We start by importing necessary libraries including Gymnasium for the environment, NumPy for numerical operations, TensorFlow for building neural networks, and Matplotlib for plotting.

```python
import os
import gymnasium as gym
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from PIL import Image
from IPython.display import Image as IPImage, display

# Create directories to save frames
os.makedirs("frames_before_training", exist_ok=True)
os.makedirs("frames_after_training", exist_ok=True)
```

2. Initializing the Environment

We initialize the Pendulum environment and define the state and action spaces. The state space represents the observations from the environment, and we discretize the continuous action space into three discrete actions: -2, 0, and 2.

```python
# Initialize the Pendulum environment with rgb_array rendering mode
env = gym.make("Pendulum-v1", render_mode="rgb_array")

# Get the size of the state space
num_states = env.observation_space.shape[0]
print("Size of State Space ->  {}".format(num_states))

# Discretize the action space to three actions: -2, 0, 2
num_actions = 3
action_space = np.linspace(env.action_space.low[0],
env.action_space.high[0], num_actions)
print("Action Space ->  {}".format(action_space))
```

```
Size of State Space ->  3
Action Space ->  [-2.  0.  2.]
```

Explanation:

Environment Initialization: We create an instance of the Pendulum environment from Gymnasium, specifying "rgb_array" rendering mode for visualization.

State and Action Spaces: We determine the dimensionality of the state space (num_states) and discretize the continuous action space into three actions (num_actions), spanning from the minimum to maximum action values defined by the environment.

3. Defining the Q-Network

We define the Q-network using TensorFlow's Keras API. The network architecture consists of ten dense hidden layers with ReLU activation functions and an output layer that outputs Q-values for each action.

```python
# Define the Q-network class with increased network capacity
class QNetwork(tf.keras.Model):
    def __init__(self):
        super(QNetwork, self).__init__()
        self.dense1 = layers.Dense(1024, activation='relu')
        self.dense2 = layers.Dense(1024, activation='relu')
        self.dense3 = layers.Dense(1024, activation='relu')
        self.dense4 = layers.Dense(1024, activation='relu')
        self.dense5 = layers.Dense(1024, activation='relu')
        self.dense6 = layers.Dense(1024, activation='relu')
        self.dense7 = layers.Dense(1024, activation='relu')
        self.dense8 = layers.Dense(1024, activation='relu')
        self.dense9 = layers.Dense(1024, activation='relu')
        self.dense10 = layers.Dense(1024, activation='relu')
        self.q_values = layers.Dense(num_actions)

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        x = self.dense3(x)
        x = self.dense4(x)
        x = self.dense5(x)
        x = self.dense6(x)
        x = self.dense7(x)
        x = self.dense8(x)
        x = self.dense9(x)
        x = self.dense10(x)
        return self.q_values(x)
```

Explanation:

QNetwork Class: This class inherits from tf.keras.Model and defines the architecture of the Q-network.

Layers: Ten dense hidden layers with 1024 units each and ReLU activation functions are defined.

Output Layer: q_values layer outputs Q-values for each action in the action space defined by num_actions.

4. Implementing the Replay Buffer

We implement the Replay Buffer to store experiences (state, action, reward, next_state, done) and sample batches for training the Q-network. This helps in breaking the correlation between consecutive experiences and stabilizes training.

```
# Define the Replay Buffer class for experience replay
class ReplayBuffer:
    def __init__(self, buffer_capacity=100000, batch_size=64):
        self.buffer_capacity = buffer_capacity
        self.batch_size = batch_size
        self.buffer_counter = 0

        # Initialize the buffers
        self.state_buffer = np.zeros((self.buffer_capacity,
num_states))
        self.action_buffer = np.zeros((self.buffer_capacity, 1))
        self.reward_buffer = np.zeros((self.buffer_capacity, 1))
        self.next_state_buffer = np.zeros((self.buffer_capacity,
num_states))
        self.done_buffer = np.zeros((self.buffer_capacity, 1))

    def record(self, obs_tuple):
        # Record the experience in the buffer
        index = self.buffer_counter % self.buffer_capacity
        self.state_buffer[index] = obs_tuple[0]
        self.action_buffer[index] = obs_tuple[1]
        self.reward_buffer[index] = obs_tuple[2]
        self.next_state_buffer[index] = obs_tuple[3]
        self.done_buffer[index] = obs_tuple[4]
        self.buffer_counter += 1

    def sample(self):
        # Sample a batch of experiences from the buffer
        record_range = min(self.buffer_counter, self.buffer_capacity)
        batch_indices = np.random.choice(record_range,
self.batch_size)
        state_batch =
tf.convert_to_tensor(self.state_buffer[batch_indices])
        action_batch =
tf.convert_to_tensor(self.action_buffer[batch_indices])
        reward_batch =
tf.convert_to_tensor(self.reward_buffer[batch_indices])
        next_state_batch =
tf.convert_to_tensor(self.next_state_buffer[batch_indices])
        done_batch =
tf.convert_to_tensor(self.done_buffer[batch_indices])
        return state_batch, action_batch, reward_batch,
next_state_batch, done_batch
```

Explanation:

ReplayBuffer Class: This class manages the replay buffer with a specified capacity (buffer_capacity) and batch size (batch_size).

Buffers Initialization: Arrays (state_buffer, action_buffer, reward_buffer, next_state_buffer, done_buffer) are initialized to store states, actions, rewards, next states, and termination flags (done).

Record Method: Stores a tuple (obs_tuple) containing state, action, reward, next state, and termination flag in the buffer.

Sample Method: Randomly samples a batch of experiences from the buffer for training. Converts data to TensorFlow tensors for efficient computation.

5. Updating the Target Network

We define a function to update the target Q-network using soft updates. This helps in stabilizing training by slowly updating the target network towards the Q-network.

```
# Function to update the target Q-network using soft updates
def update_target_network(target_weights, weights, tau):
    for (a, b) in zip(target_weights, weights):
        a.assign(b * tau + a * (1 - tau))
```

Explanation:

update_target_network Function: Performs soft updates to the target Q-network (target_model) using the weights (weights) of the Q-network (model). Soft update mechanism helps in stabilizing training by gradually updating the target network towards the main network.

6. Epsilon-Greedy Policy

We implement the epsilon-greedy policy to balance exploration and exploitation. The agent selects a random action with probability epsilon and the best-known action (based on Q-values) with probability 1 - epsilon.

```
# Epsilon-greedy policy for exploration
def policy(state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.choice(action_space)
    state = tf.convert_to_tensor(state)
    state = tf.expand_dims(state, axis=0)
    q_values = model(state)
    return action_space[np.argmax(q_values.numpy())]
```

Explanation:

policy Function: Implements an epsilon-greedy policy for action selection based on the current state (state) and exploration parameter (epsilon).

Random Exploration: With probability epsilon, selects a random action from the action space (action_space).

Exploitation: Otherwise, selects the action with the highest Q-value (argmax) predicted by the Q-network (model).

```
# Function to visualize the pendulum's movement
def visualize_pendulum(model, folder, episodes=1):
    for ep in range(episodes):
        state, _ = env.reset()
        done = False
        truncated = False
        frame_idx = 0
        while not done and not truncated:
            frame = env.render()
            if frame is not None:

Image.fromarray(frame).save(f"{folder}/frame_{ep}_{frame_idx}.png")
            action = policy(state, epsilon=0)  # Use greedy policy for
visualization
            state, _, done, truncated, _ = env.step([action])
            frame_idx += 1
    env.close()

# Function to create a video from saved frames
def create_video(folder, output_file):
    frames = [Image.open(f"{folder}/{frame}") for frame in
sorted(os.listdir(folder)) if frame.endswith(".png")]
    frames[0].save(output_file, save_all=True,
append_images=frames[1:], duration=50, loop=0)
```

7. Hyperparameters

We define the hyperparameters used for training the DQN algorithm. These include initial and minimum values for epsilon, epsilon decay rate, discount factor (gamma), soft update rate (tau), batch size, replay buffer capacity, learning rate, and total number of training episodes.

```
# Hyperparameters for DQN training
epsilon = 1.0           # Initial epsilon for epsilon-greedy policy
epsilon_min = 0.01      # Minimum epsilon value
epsilon_decay = 0.995   # Decay rate for epsilon
gamma = 0.99            # Discount factor for future rewards
tau = 0.005             # Soft update rate for target network
batch_size = 64         # Size of the batch sampled from the replay
buffer
buffer_capacity = 100000# Capacity of the replay buffer
learning_rate = 0.00025 # Learning rate for the optimizer
total_episodes = 300    # Total number of episodes for training
```

Explanation:

Hyperparameters: These variables control various aspects of the DQN algorithm.

Epsilon: Controls the trade-off between exploration and exploitation.

Epsilon Decay: Rate at which epsilon decreases over time to shift from exploration to exploitation.

Gamma: Discount factor that determines the importance of future rewards.

Tau: Rate at which the target Q-network is updated towards the main Q-network.

Batch Size: Number of experiences sampled from the replay buffer for each training iteration.

Buffer Capacity: Maximum number of experiences stored in the replay buffer.

Learning Rate: Rate at which the optimizer adjusts the weights of the Q-network based on the loss.

Total Episodes: Number of episodes the agent interacts with the environment during training.

8. Initializing Q-Networks, Optimizer, and Replay Buffer

```
# Initialize the Q-network and target Q-network
model = QNetwork()  # Initialize the main Q-network
target_model = QNetwork()  # Initialize the target Q-network
target_model.set_weights(model.get_weights())  # Set initial weights
of target network to match the main network

# Initialize the optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate)

# Initialize the replay buffer
buffer = ReplayBuffer(buffer_capacity, batch_size)

# Lists to store episodic rewards and average rewards
ep_reward_list = []  # List to store rewards for each episode
avg_reward_list = []  # List to store average rewards over the last 40
episodes
```

Explanation:

Q-Network Initialization: Two instances of QNetwork are created — model for the main Q-network and target_model for the target Q-network. The weights of target_model are initially set to match those of model using set_weights.

Optimizer: Adam optimizer is initialized with the specified learning rate (learning_rate) to optimize the Q-network.

Replay Buffer: An instance of ReplayBuffer is initialized with the specified capacity (buffer_capacity) and batch size (batch_size). This buffer stores experiences (state, action, reward, next_state, done) for experience replay during training.

Reward Lists: ep_reward_list stores episodic rewards (total reward accumulated in each episode), and avg_reward_list stores the average episodic reward over the last 40 episodes for monitoring training progress.

```
# Visualize pendulum before training
print("Visualizing pendulum before training...")
visualize_pendulum(model, folder="frames_before_training", episodes=3)
create_video("frames_before_training", "pendulum_before_training.gif")
display(IPImage(filename="pendulum_before_training.gif"))
```

```
Visualizing pendulum before training...
```

9. Training Loop

```
# Training loop
for ep in range(total_episodes):
    state, _ = env.reset()  # Reset environment and get initial state
    episodic_reward = 0  # Initialize episodic reward to 0
    done = False  # Initialize done flag to False
    truncated = False  # Initialize truncated flag to False

    while not done and not truncated:
        action = policy(state, epsilon)  # Choose action based on
epsilon-greedy policy
        next_state, reward, done, truncated, _ = env.step([action])  #
Take action in the environment
        buffer.record((state, action, reward, next_state, done))  #
Record experience in replay buffer
        episodic_reward += reward  # Accumulate episodic reward

        # Perform training if buffer has enough samples
        if buffer.buffer_counter >= batch_size:
            states, actions, rewards, next_states, dones =
buffer.sample()  # Sample batch from replay buffer
            rewards = tf.cast(rewards, dtype=tf.float32)  # Convert
rewards to float32
            dones = tf.cast(dones, dtype=tf.float32)  # Convert dones
to float32

            # Calculate target Q-values using the target Q-network
            target_q = rewards + gamma *
tf.reduce_max(target_model(next_states), axis=1, keepdims=True) * (1 -
dones)

            # Compute loss and gradients
            with tf.GradientTape() as tape:
                q_values = model(states)  # Predict Q-values using the
main Q-network
                actions_one_hot = tf.one_hot(tf.cast(actions,
tf.int32), num_actions, dtype=tf.float32)  # One-hot encode actions
                q_values = tf.reduce_sum(q_values * actions_one_hot,
axis=1, keepdims=True)  # Calculate Q-values for selected actions
                loss = tf.reduce_mean(tf.square(target_q - q_values))
# Compute MSE loss between predicted and target Q-values
            grads = tape.gradient(loss, model.trainable_variables)  #
Compute gradients of loss w.r.t. Q-network weights
            optimizer.apply_gradients(zip(grads,
model.trainable_variables))  # Apply gradients to update Q-network
weights

            # Update target Q-network using soft update mechanism
            update_target_network(target_model.variables,
model.variables, tau)

        state = next_state  # Update current state to next state
    epsilon = max(epsilon_min, epsilon * epsilon_decay)  # Decay
epsilon for epsilon-greedy policy
    ep_reward_list.append(episodic_reward)  # Append episodic reward
to episodic reward list
    avg_reward = np.mean(ep_reward_list[-40:])  # Compute average
reward over last 40 episodes
    print(f"Episode {ep}, Avg Reward: {avg_reward}")  # Print average
reward for current episode
    avg_reward_list.append(avg_reward)  # Append average reward to
average reward list
```

```
Episode 0, Avg Reward: -1073.7780519535575
Episode 1, Avg Reward: -1284.0425347965215
Episode 2, Avg Reward: -1212.062726151056
Episode 3, Avg Reward: -1289.0129434981013
Episode 4, Avg Reward: -1343.5169207631195
Episode 5, Avg Reward: -1305.1392056380198
Episode 6, Avg Reward: -1257.883526906749
Episode 7, Avg Reward: -1295.8319297543349
Episode 8, Avg Reward: -1271.3478030351876
Episode 9, Avg Reward: -1260.7823226172763
Episode 10, Avg Reward: -1296.7991434697515
Episode 11, Avg Reward: -1262.555189530902
Episode 12, Avg Reward: -1254.3303823986957
Episode 13, Avg Reward: -1228.4580348104992
Episode 14, Avg Reward: -1219.0652461253871
Episode 15, Avg Reward: -1224.8453126599866
Episode 16, Avg Reward: -1250.5983784768123
Episode 17, Avg Reward: -1253.1563640675142
Episode 18, Avg Reward: -1240.485400265098
Episode 19, Avg Reward: -1252.7053453081915
Episode 20, Avg Reward: -1247.1235856966307
Episode 21, Avg Reward: -1255.762313991117
Episode 22, Avg Reward: -1243.306082063467
Episode 23, Avg Reward: -1244.8677342634244
Episode 24, Avg Reward: -1231.4297774817128
Episode 25, Avg Reward: -1223.4090258217636
Episode 26, Avg Reward: -1215.8385305783988
Episode 27, Avg Reward: -1217.9426822170717
Episode 28, Avg Reward: -1211.0830267356423
Episode 29, Avg Reward: -1212.7989491126052
Episode 30, Avg Reward: -1215.23414684996
Episode 31, Avg Reward: -1227.5863274192452
Episode 32, Avg Reward: -1226.118443729792
Episode 33, Avg Reward: -1220.7668054314768
Episode 34, Avg Reward: -1217.141424377471
Episode 35, Avg Reward: -1213.0374358956092
Episode 36, Avg Reward: -1208.1109602415759
Episode 37, Avg Reward: -1218.1465419314436
Episode 38, Avg Reward: -1217.4790380178026
Episode 39, Avg Reward: -1205.6311046000835
Episode 40, Avg Reward: -1216.8204149258913
Episode 41, Avg Reward: -1209.426774022626
Episode 42, Avg Reward: -1208.3124366813481
Episode 43, Avg Reward: -1209.2101582451292
Episode 44, Avg Reward: -1195.563319967685
Episode 45, Avg Reward: -1199.454433286463
Episode 46, Avg Reward: -1218.6631781311442
Episode 47, Avg Reward: -1213.217639199746
Episode 48, Avg Reward: -1208.8551713666486
Episode 49, Avg Reward: -1215.3326959009719
Episode 50, Avg Reward: -1205.8900198616493
Episode 51, Avg Reward: -1210.4579373816755
Episode 52, Avg Reward: -1217.4419477241568
Episode 53, Avg Reward: -1232.2809176721576
Episode 54, Avg Reward: -1240.254158710771
Episode 55, Avg Reward: -1232.6411690555094
Episode 56, Avg Reward: -1217.9345705634482
Episode 57, Avg Reward: -1207.9550292528043
Episode 58, Avg Reward: -1213.5727849348439
Episode 59, Avg Reward: -1201.7157421435325
Episode 60, Avg Reward: -1198.591937968621
Episode 61, Avg Reward: -1191.9781549613776
Episode 62, Avg Reward: -1200.6108957076926
Episode 63, Avg Reward: -1193.4907340947732
Episode 64, Avg Reward: -1206.5961689249618
Episode 65, Avg Reward: -1206.2390605716553
Episode 66, Avg Reward: -1207.6462768285255
Episode 67, Avg Reward: -1209.2861688205228
Episode 68, Avg Reward: -1207.9658024680152
Episode 69, Avg Reward: -1215.301103512165
Episode 70, Avg Reward: -1206.456378369766
Episode 71, Avg Reward: -1191.3723171565432
Episode 72, Avg Reward: -1199.161069413254
Episode 73, Avg Reward: -1209.4264222418703
Episode 74, Avg Reward: -1219.0685787936272
Episode 75, Avg Reward: -1225.5184794899105
Episode 76, Avg Reward: -1232.185216697903
Episode 77, Avg Reward: -1223.6316261633572
Episode 78, Avg Reward: -1218.0461102378138
Episode 79, Avg Reward: -1232.8441470880057
Episode 80, Avg Reward: -1231.5222121621766
Episode 81, Avg Reward: -1227.2863212978566
Episode 82, Avg Reward: -1237.6228110990673
Episode 83, Avg Reward: -1225.9911079997942
Episode 84, Avg Reward: -1222.26382387698
Episode 85, Avg Reward: -1230.9161598383457
Episode 86, Avg Reward: -1214.1133099598264
Episode 87, Avg Reward: -1207.4174778339227
Episode 88, Avg Reward: -1222.3892878496613
Episode 89, Avg Reward: -1216.220712837518
Episode 90, Avg Reward: -1203.4251887731375
Episode 91, Avg Reward: -1203.4689811631063
Episode 92, Avg Reward: -1189.540853795403
Episode 93, Avg Reward: -1177.6583660307297
Episode 94, Avg Reward: -1169.8845609288826
Episode 95, Avg Reward: -1168.8273963733077
Episode 96, Avg Reward: -1186.6768618409358
Episode 97, Avg Reward: -1193.7789914455816
Episode 98, Avg Reward: -1195.6779348079576
Episode 99, Avg Reward: -1198.9892801464682
Episode 100, Avg Reward: -1195.8350841939719
Episode 101, Avg Reward: -1189.0118335794373
Episode 102, Avg Reward: -1178.0284105670885
Episode 103, Avg Reward: -1174.1997763798176
Episode 104, Avg Reward: -1173.3196387831008
Episode 105, Avg Reward: -1170.0152789610397
Episode 106, Avg Reward: -1185.3149705147757
Episode 107, Avg Reward: -1186.2161711339872
Episode 108, Avg Reward: -1197.7708300892875
Episode 109, Avg Reward: -1183.8700248424723
Episode 110, Avg Reward: -1194.55642047998
Episode 111, Avg Reward: -1194.4411916988815
Episode 112, Avg Reward: -1191.8547668775798
Episode 113, Avg Reward: -1182.3906028522665
Episode 114, Avg Reward: -1176.8574851051048
Episode 115, Avg Reward: -1175.3864919408795
Episode 116, Avg Reward: -1176.2371817944104
Episode 117, Avg Reward: -1163.9499498102832
Episode 118, Avg Reward: -1160.6723470532888
Episode 119, Avg Reward: -1155.670326436463
Episode 120, Avg Reward: -1159.4854887963925
Episode 121, Avg Reward: -1155.097359988098
Episode 122, Avg Reward: -1138.4468059524272
Episode 123, Avg Reward: -1135.2566601165915
Episode 124, Avg Reward: -1146.4563759735138
Episode 125, Avg Reward: -1132.9914388933207
Episode 126, Avg Reward: -1128.4529393890082
Episode 127, Avg Reward: -1126.2192770509828
Episode 128, Avg Reward: -1113.2000854865425
Episode 129, Avg Reward: -1120.3997577280993
Episode 130, Avg Reward: -1130.4771636181092
Episode 131, Avg Reward: -1127.0363808409336
Episode 132, Avg Reward: -1134.0009052464989
Episode 133, Avg Reward: -1137.2659044279458
Episode 134, Avg Reward: -1154.976724391837
Episode 135, Avg Reward: -1154.9079686050886
Episode 136, Avg Reward: -1156.0503937227827
Episode 137, Avg Reward: -1163.2504981444988
Episode 138, Avg Reward: -1161.9365533334144
Episode 139, Avg Reward: -1160.1629314218555
Episode 140, Avg Reward: -1164.6408827213304
Episode 141, Avg Reward: -1174.1104685806029
Episode 142, Avg Reward: -1177.836830200548
Episode 143, Avg Reward: -1190.3010485680502
Episode 144, Avg Reward: -1197.8160678211768
Episode 145, Avg Reward: -1197.7430058600214
Episode 146, Avg Reward: -1190.4474302231315
Episode 147, Avg Reward: -1190.5582760271293
Episode 148, Avg Reward: -1176.6285216870847
Episode 149, Avg Reward: -1197.1793618875308
Episode 150, Avg Reward: -1205.688132223681
Episode 151, Avg Reward: -1204.8204849890933
Episode 152, Avg Reward: -1191.4673760099527
Episode 153, Avg Reward: -1209.284072229523
Episode 154, Avg Reward: -1215.572767199176
Episode 155, Avg Reward: -1206.236869766642
Episode 156, Avg Reward: -1197.2347630911809
Episode 157, Avg Reward: -1210.2213704921014
Episode 158, Avg Reward: -1229.6065528945626
Episode 159, Avg Reward: -1235.3209444109784
Episode 160, Avg Reward: -1236.3983349579282
Episode 161, Avg Reward: -1243.3438904410018
Episode 162, Avg Reward: -1246.939484683496
Episode 163, Avg Reward: -1264.1617921729269
Episode 164, Avg Reward: -1251.9987279281518
Episode 165, Avg Reward: -1249.1205496069038
Episode 166, Avg Reward: -1258.6126379759164
Episode 167, Avg Reward: -1266.7353497822774
Episode 168, Avg Reward: -1276.4435547853495
Episode 169, Avg Reward: -1279.134681126706
Episode 170, Avg Reward: -1294.488773552181
Episode 171, Avg Reward: -1291.7587910916438
Episode 172, Avg Reward: -1288.9102422735045
Episode 173, Avg Reward: -1281.9609749531976
Episode 174, Avg Reward: -1261.9446319228055
Episode 175, Avg Reward: -1279.53268154454
Episode 176, Avg Reward: -1275.5373111494348
Episode 177, Avg Reward: -1261.034991547786
Episode 178, Avg Reward: -1254.5024332814398
Episode 179, Avg Reward: -1258.7069342353095
Episode 180, Avg Reward: -1251.2191523773668
Episode 181, Avg Reward: -1251.7017808320784
Episode 182, Avg Reward: -1258.1349177608913
Episode 183, Avg Reward: -1258.414536413912
Episode 184, Avg Reward: -1248.8202899374096
Episode 185, Avg Reward: -1245.9450794425559
Episode 186, Avg Reward: -1237.7105033446617
Episode 187, Avg Reward: -1227.249874546978
Episode 188, Avg Reward: -1226.9069562988007
Episode 189, Avg Reward: -1211.4427537429588
Episode 190, Avg Reward: -1201.0979098490225
Episode 191, Avg Reward: -1198.7621596801982
Episode 192, Avg Reward: -1214.2427846732958
Episode 193, Avg Reward: -1209.234449384428
Episode 194, Avg Reward: -1200.0840092205726
Episode 195, Avg Reward: -1211.2388662845206
Episode 196, Avg Reward: -1216.6012954674995
Episode 197, Avg Reward: -1214.1136088161588
Episode 198, Avg Reward: -1199.759674750839
Episode 199, Avg Reward: -1199.5944203855215
Episode 200, Avg Reward: -1180.113972590542
Episode 201, Avg Reward: -1189.0854974480749
Episode 202, Avg Reward: -1203.9044314937667
Episode 203, Avg Reward: -1207.955099335606
Episode 204, Avg Reward: -1216.468261729304
Episode 205, Avg Reward: -1219.2612135595816
Episode 206, Avg Reward: -1227.3349021671656
Episode 207, Avg Reward: -1237.4122033567367
Episode 208, Avg Reward: -1228.812773558344
Episode 209, Avg Reward: -1219.0630000882081
Episode 210, Avg Reward: -1218.8984709489198
Episode 211, Avg Reward: -1230.9348228174063
Episode 212, Avg Reward: -1234.2621156529563
Episode 213, Avg Reward: -1241.346071520587
Episode 214, Avg Reward: -1237.781105390103
Episode 215, Avg Reward: -1222.5024717055842
Episode 216, Avg Reward: -1205.0633925919287
Episode 217, Avg Reward: -1212.7736145737472
Episode 218, Avg Reward: -1220.865109866097
Episode 219, Avg Reward: -1209.446232145076
Episode 220, Avg Reward: -1222.6264759924857
Episode 221, Avg Reward: -1223.5955251276182
Episode 222, Avg Reward: -1228.3556590457895
Episode 223, Avg Reward: -1218.6325766653765
Episode 224, Avg Reward: -1212.9137349408234
Episode 225, Avg Reward: -1210.0152265154768
Episode 226, Avg Reward: -1209.6866622112248
Episode 227, Avg Reward: -1215.050224582657
Episode 228, Avg Reward: -1215.9312558860825
Episode 229, Avg Reward: -1210.0122792415132
Episode 230, Avg Reward: -1216.2418589006527
Episode 231, Avg Reward: -1221.4862652869263
Episode 232, Avg Reward: -1219.5424147850135
Episode 233, Avg Reward: -1201.7920245822895
Episode 234, Avg Reward: -1209.4553712028005
Episode 235, Avg Reward: -1215.394796820724
Episode 236, Avg Reward: -1230.1273441896515
Episode 237, Avg Reward: -1232.207898490289
Episode 238, Avg Reward: -1247.5871059304616
Episode 239, Avg Reward: -1236.7323338436206
Episode 240, Avg Reward: -1255.9854142793579
Episode 241, Avg Reward: -1260.2845538860365
Episode 242, Avg Reward: -1248.7813410611125
Episode 243, Avg Reward: -1227.404418855389
Episode 244, Avg Reward: -1230.4480861884324
Episode 245, Avg Reward: -1225.41060799995
Episode 246, Avg Reward: -1207.4210695734723
Episode 247, Avg Reward: -1183.0573615578846
Episode 248, Avg Reward: -1191.0196910382979
Episode 249, Avg Reward: -1193.453366390435
Episode 250, Avg Reward: -1175.6902504865882
Episode 251, Avg Reward: -1164.5227985454744
Episode 252, Avg Reward: -1168.03385770803
Episode 253, Avg Reward: -1161.8248063663864
Episode 254, Avg Reward: -1161.7789850053846
Episode 255, Avg Reward: -1171.2860377989066
Episode 256, Avg Reward: -1176.1979684272208
Episode 257, Avg Reward: -1179.4862032826336
Episode 258, Avg Reward: -1175.3986731361597
Episode 259, Avg Reward: -1182.9772150023987
Episode 260, Avg Reward: -1175.0076176163195
Episode 261, Avg Reward: -1163.2113952214745
Episode 262, Avg Reward: -1153.443570888006
Episode 263, Avg Reward: -1150.9951090247848
Episode 264, Avg Reward: -1165.1802528454734
Episode 265, Avg Reward: -1181.2872331142378
Episode 266, Avg Reward: -1180.6870059379423
Episode 267, Avg Reward: -1173.7397313370955
Episode 268, Avg Reward: -1180.7152193276693
Episode 269, Avg Reward: -1183.250410971743
Episode 270, Avg Reward: -1174.5311834637746
Episode 271, Avg Reward: -1174.3903699441635
Episode 272, Avg Reward: -1167.2985407426884
Episode 273, Avg Reward: -1183.3123377659783
Episode 274, Avg Reward: -1185.5277153015154
Episode 275, Avg Reward: -1176.4875900332922
Episode 276, Avg Reward: -1161.386837162619
Episode 277, Avg Reward: -1161.494657295815
Episode 278, Avg Reward: -1158.2781010623717
Episode 279, Avg Reward: -1176.4380154498917
Episode 280, Avg Reward: -1156.5633354437368
Episode 281, Avg Reward: -1136.5222653789717
Episode 282, Avg Reward: -1149.3337114159663
Episode 283, Avg Reward: -1161.0332667695316
Episode 284, Avg Reward: -1166.1520886002368
Episode 285, Avg Reward: -1166.2543221888293
Episode 286, Avg Reward: -1166.8616220631461
Episode 287, Avg Reward: -1189.224434903127
Episode 288, Avg Reward: -1174.464944250079
Episode 289, Avg Reward: -1164.0981350432387
Episode 290, Avg Reward: -1168.4545933164773
Episode 291, Avg Reward: -1189.9017910760076
Episode 292, Avg Reward: -1186.2522636895953
Episode 293, Avg Reward: -1188.0378576703465
Episode 294, Avg Reward: -1205.454901692565
Episode 295, Avg Reward: -1192.9200701877492
Episode 296, Avg Reward: -1185.2389482649544
Episode 297, Avg Reward: -1171.195888356399
Episode 298, Avg Reward: -1163.3495543568674
Episode 299, Avg Reward: -1166.5710196793552
```

Explanation:

Training Loop: The outer loop runs for total_episodes, where each episode interacts with the environment to collect experiences.

Environment Interaction: env.reset() resets the environment and returns the initial state (state). env.step([action]) takes an action (action) based on the epsilon-greedy policy and returns the next state (next_state), reward (reward), and termination flags (done, truncated).

Experience Replay: The experience (state, action, reward, next_state, done) is recorded in the replay buffer (buffer.record()).

Training Condition: If the replay buffer contains enough experiences (buffer.buffer_counter >= batch_size), a batch of experiences is sampled (buffer.sample()).

Q-Value Calculation: Q-values are predicted using the main Q-network (model(states)) and then computed for selected actions using one-hot encoding.

Loss Computation: Mean Squared Error (MSE) loss is calculated between predicted Q-values (q_values) and target Q-values (target_q).

Gradient Descent: Gradients of the loss with respect to Q-network weights (model.trainable_variables) are computed using tf.GradientTape(). These gradients are applied to update Q-network weights using the optimizer (optimizer.apply_gradients()).

Target Network Update: Target Q-network (target_model) is updated using the soft update mechanism (update_target_network).

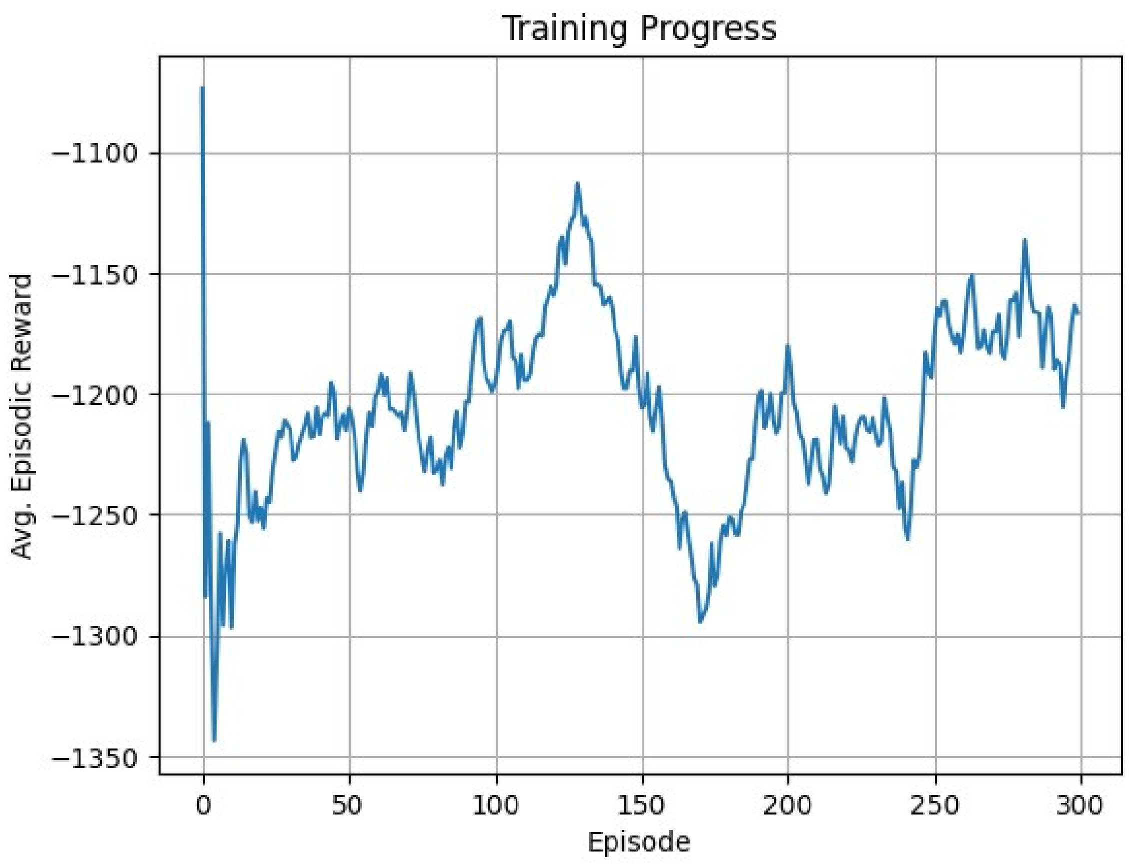
Epsilon Decay: Epsilon (epsilon) decays over time to gradually shift from exploration to exploitation.

Reward Tracking: Episodic reward (episodic_reward) is accumulated and appended to ep_reward_list. Average reward over the last 40 episodes (avg_reward) is computed and appended to avg_reward_list.

Logging: Prints the episode number and average reward for each episode during training.

10. Plotting Results

```
# Plotting the results
plt.plot(avg_reward_list)
plt.xlabel("Episode")
plt.ylabel("Avg. Episodic Reward")
plt.title("Training Progress")
plt.grid(True)
plt.show()
```



Explanation:

Plotting: Matplotlib is used to plot avg_reward_list against episode number to visualize the training progress.

X-axis: Episode number.

Y-axis: Average episodic reward (avg_reward).

Title: Title of the plot ("Training Progress").

Grid: Grid lines are displayed on the plot for better readability.

```python
# Visualize pendulum after training
print("Visualizing pendulum after training...")
visualize_pendulum(model, folder="frames_after_training", episodes=3)
create_video("frames_after_training", "pendulum_after_training.gif")
display(IPImage(filename="pendulum_after_training.gif"))
```

```
Visualizing pendulum after training...
```