Lab 7: DQN and DDPG

August 2023

1 In This Lab

1.1 Topics to Cover

- Deep Q Network
- Deep Deterministic Policy Network

1.2 Requirements

- Python installed on your computer.
- Basic Python programming, Jupyter Notebook.
- Libraries: PyTorch, gym, scikit-learn, numpy, matplotlib, seaborn

1.3 Environment

- OpenAI Gym

2 Deep Q-Network (DQN)

A Deep Q-Network (DQN) is a reinforcement learning algorithm that combines Q-learning with deep learning techniques to solve problems in artificial intelligence and machine learning. Introduced by DeepMind in 2013, DQNs are used to learn optimal policies for agents in environments.

2.1 DQN Components

2.1.1 Reinforcement Learning

Reinforcement learning involves an agent learning to perform actions in an environment to maximize cumulative rewards. The agent interacts with the environment, takes actions, receives rewards, and optimizes actions over time.

2.1.2 Q-Learning

Q-learning is a classic algorithm that learns the optimal action-selection policy for an agent in an environment. The Q-value function $Q(s, a)$ represents expected cumulative reward for action a in state s.

2.1.3 Deep Learning

DQNs use deep neural networks to approximate Q-values. A neural network approximates Q-values for state-action pairs, handling high-dimensional input spaces.

2.1.4 Experience Replay

Experience replay stores agent experiences (state, action, reward, next state) in a buffer. Mini-batches of experiences are sampled during training to improve stability.

2.1.5 Target Network

DQNs use online and target networks. The target network, a copy of the Q-network, is updated less frequently to stabilize training.

2.1.6 Loss Function

Mean squared error (MSE) loss is commonly used in training DQNs, measuring the difference between predicted and target Q-values.

2.1.7 Epsilon-Greedy Exploration

Epsilon-greedy strategy balances exploration and exploitation. With probability ϵ, the agent takes a random action; otherwise, it selects the action with the highest Q-value.

2.2 Why Deep Q-Networks (DQNs)?

Deep Q-Networks (DQNs) emerged as a significant advancement in reinforcement learning due to several key reasons:

1. **Handling High-Dimensional Inputs:** Traditional Q-learning methods struggle with high-dimensional state spaces, making storage and computation impractical. DQNs use neural networks to approximate Q-values, enabling them to manage complex input spaces, such as raw pixel data from images.

2. **End-to-End Learning:** DQNs facilitate end-to-end learning, where agents learn directly from raw sensory input. They eliminate the need for handcrafted features or preprocessing, making them applicable to various tasks without manual feature engineering.

3. **Generalization:** DQNs leverage neural networks' ability to generalize across similar states, which is vital in environments with numerous states and limited experiences.

4. **Autonomous Learning:** DQNs can autonomously learn from interacting with an environment, making them suitable when obtaining an accurate environment model is challenging.

5. **Transferability:** Trained DQNs can be adapted to new tasks with minimal modifications. The learned representations can transfer to different tasks, showcasing their potential for transfer learning.

6. **Learning from Experience:** DQNs use experience replay to break temporal correlations between experiences, stabilizing learning and preventing the agent from getting stuck in local minima.

7. **State-of-the-Art Performance:** DQNs have excelled on various challenging tasks, including superhuman performance in Atari games, robotic control, and simulations. Their success demonstrates their capability in addressing real-world problems.

8. **Advancements and Variants:** The development of DQNs spurred research into enhancements and extensions like Double DQN, Dueling DQN, Prioritized Experience Replay, and Rainbow DQN. These improvements aim to enhance stability, sample efficiency, and overall performance.

2.3 Steps in Deep Q-Network (DQN)

The Deep Q-Network (DQN) algorithm involves the following steps:

1. **Initialize Networks:** Initialize two neural networks - the online Q-network and the target Q-network. The online network is updated iteratively during training, while the target network is updated less frequently to provide stable target Q-values.

2. **Initialize Replay Buffer:** Create a replay buffer to store experiences, each consisting of a tuple of state, action, reward, and next state. This buffer is used for experience replay during training.

3. **Exploration Strategy:** Decide on an exploration strategy, often using epsilon-greedy exploration. With a certain probability ϵ, select a random action; otherwise, choose the action with the highest Q-value.

4. **Interaction with Environment:** Interact with the environment using the chosen exploration strategy. Observe the current state, select an action, receive a reward, and transition to the next state.

5. **Store Experience:** Store the experience tuple (state, action, reward, next state) in the replay buffer.

6. **Sample from Replay Buffer:** Periodically sample a mini-batch of experiences from the replay buffer. This helps break temporal correlations and stabilizes learning.
7. **Compute Target Q-Values:** For each experience in the mini-batch, compute the target Q-value using the target network. The target Q-value is calculated as the immediate reward plus the discounted maximum Q-value of the next state.
8. **Update Online Q-Network:** Update the online Q-network using backpropagation and the computed target Q-values. Minimize the mean squared error between predicted Q-values and target Q-values.
9. **Update Target Network:** Periodically update the target network by copying the weights from the online Q-network. This helps stabilize training and prevents divergence.
10. **Repeat:** Repeat steps 4 to 9 for a predefined number of iterations or until convergence is achieved.
11. **Evaluation:** Periodically evaluate the performance of the trained DQN on the environment to monitor progress. This helps in assessing the agent's learning and generalization.

The iterative execution of these steps allows the DQN algorithm to learn an optimal policy for the agent to perform actions in the environment and maximize cumulative rewards.

2.4 Equations for Deep Q-Network (DQN)

DQN employs several equations to update the Q-values and train the neural networks. Here are the key equations:

2.4.1 Q-Learning Update

The Q-Learning update equation calculates the updated Q-value for a state-action pair using the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

Where:

- $Q(s,a)$ is the Q-value for state s and action a.
- α is the learning rate.
- r is the immediate reward received after taking action a in state s.
- γ is the discount factor that determines the importance of future rewards.
- $\max_{a'} Q(s',a')$ is the maximum Q-value over possible actions a' in the next state s'.

2.4.2 Loss Function for Q-Network

The loss function used to train the Q-network is the mean squared error (MSE) between the predicted Q-values and the target Q-values:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i} \left(Q(s, a; \theta) - (r + \gamma \max_{a'} Q(s', a'; \theta^-))^2 \right)$$

Where:

- θ are the weights of the Q-network.
- N is the mini-batch size.
- $Q(s, a; \theta)$ is the predicted Q-value for state s and action a using the Q-network with weights θ.
- θ^- represents the target network's weights.
- $r + \gamma \max_{a'} Q(s', a'; \theta^-)$ is the target Q-value calculated using the target network.

2.5 Task 1:

1. Follow DQN.ipynb

2.6 Report

1. Do the same steps as in DQN.ipynb for a different discrete action space gym environment

3 Exercise 2: Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning algorithm designed to address problems with continuous action spaces. It combines concepts from deep learning and policy gradients to learn optimal policies for agents in such environments.

3.1 Components

1. **Actor-Critic Architecture:** DDPG utilizes an actor-critic architecture comprising two neural networks:
   - **Actor Network:** Learns a deterministic policy that directly maps states to continuous actions.
   - **Critic Network:** Estimates the expected cumulative reward (Q-value) for a given state-action pair.
2. **Target Networks:** DDPG introduces target networks for both the actor and the critic. These networks are updated gradually using a soft update mechanism to stabilize training.
3. **Experience Replay:** The algorithm employs an experience replay buffer to store tuples of state, action, reward, next state, and done signal. Sampling from this buffer reduces temporal correlations between experiences and enhances learning stability.
4. **Exploration Strategy:** Noise-based exploration is commonly used in DDPG. Noise is added to the actor-selected actions to encourage exploration during training.
5. **Update Process:**
   - The critic network is updated by minimizing the mean squared error between predicted Q-values and target Q-values obtained from the target critic network, similar to the Bellman equation.
   - The actor network is updated by performing policy gradient ascent using Q-value gradients provided by the critic network.
6. **Soft Target Updates:** Target networks are updated periodically using a soft update mechanism. This involves blending the target network weights with the online network weights, promoting stability in training.

DDPG has shown success in various continuous control tasks, though it still faces challenges such as hyperparameter tuning and slow convergence. Extensions and variants like Twin Delayed DDPG (TD3) and Soft Actor-Critic (SAC) have been developed to improve performance and address these limitations.

3.2 Steps in Deep Deterministic Policy Gradient (DDPG)

The Deep Deterministic Policy Gradient (DDPG) algorithm involves the following steps:

1. **Initialize Networks:** Initialize the actor and critic neural networks. The actor network learns the policy, mapping states to actions, while the critic network estimates the Q-value of state-action pairs.
2. **Initialize Target Networks:** Create target networks for both the actor and critic networks. These target networks are periodically updated to provide more stable target Q-value estimates.
3. **Initialize Replay Buffer:** Set up a replay buffer to store experiences, containing tuples of state, action, reward, next state, and whether the episode is done.
4. **Exploration Strategy:** Select an exploration strategy, often using noise added to the chosen action. This encourages exploration of the action space while training.
5. **Interaction with Environment:** Interact with the environment using the chosen exploration strategy. Observe the current state, select an action using the actor network, receive a reward, and transition to the next state.
6. **Store Experience:** Store the experience tuple (state, action, reward, next state, done) in the replay buffer.
7. **Sample from Replay Buffer:** Periodically sample a mini-batch of experiences from the replay buffer. This mini-batch is used for training the actor and critic networks.
8. **Update Critic Network:** Compute the target Q-value using the target actor and target critic networks. Update the critic network by minimizing the mean squared error between predicted Q-values and target Q-values.
9. **Update Actor Network:** Compute the actor's policy gradient using the sampled experiences and the critic network. Update the actor network using the gradient ascent direction.
10. **Update Target Networks:** Periodically update the target networks by applying a soft update. This involves updating the target network weights toward the online network weights.
11. **Repeat:** Repeat steps 4 to 10 for a predefined number of iterations or until convergence is achieved.
12. **Evaluation:** Periodically evaluate the performance of the trained DDPG agent on the environment to assess learning progress and generalization.

By iteratively executing these steps, the DDPG algorithm learns an optimal deterministic policy that guides the agent to select actions in the environment to maximize cumulative rewards.

3.3 Advantages of DDPG over DQN

DDPG algorithm offers several advantages over the DQN algorithm:

1. **Continuous Action Spaces:** DDPG is well-suited for environments with continuous action spaces, which are common in robotics, control systems, and other real-world applications. DQN struggles with such spaces due to the discrete action selection process.

2. **Deterministic Policy:** DDPG learns a deterministic policy that directly maps states to continuous actions. This can lead to more stable and predictable behavior compared to the stochastic policy learned by DQN.

3. **Action Exploration:** DDPG employs noise-based exploration techniques, which can be more effective in continuous action spaces. DQN's epsilon-greedy exploration is better suited for discrete action spaces and may lead to less efficient exploration in continuous settings.

4. **Efficient Learning:** DDPG's use of policy gradients allows for more efficient learning in continuous action spaces. DQN requires adaptations, such as discretizing the action space, which can introduce challenges and inefficiencies.

5. **Better Convergence:** DDPG often converges more smoothly in continuous control tasks due to its deterministic nature and the use of policy gradients. DQN's discrete action choices and Q-value updates may lead to slower or less stable convergence.

6. **Action Value Estimation:** DDPG's critic network estimates action values directly, which can be more suitable for continuous action spaces. DQN's Q-network estimates Q-values for discrete action choices.

7. **Enhanced Generalization:** DDPG's deterministic policy and smoother convergence often lead to better generalization in continuous environments, where small changes in action can have significant effects.

8. **Real-World Applications:** DDPG's strengths make it more applicable to real-world scenarios, such as robotic control and autonomous systems, where precise and continuous actions are crucial.

3.4 Equations for DDPG

DDPG involves several equations to update the actor and critic networks and optimize the policy. Here are the key equations:

3.4.1 Critic Update

The critic network is updated using the mean squared Bellman error:

$$\mathcal{L}_{critic}(\theta^Q) = \frac{1}{N}\sum_i \Big(Q(s,a;\theta^Q) - (r + \gamma Q(s', \pi(s';\theta^{\pi'});\theta^{Q'}))^2\Big)$$

Where:

- θ^Q are the weights of the critic network.
- $Q(s,a;\theta^Q)$ is the Q-value predicted by the critic network for state s and action a.
- r is the immediate reward.
- γ is the discount factor.
- $\pi(s';\theta^{\pi'})$ is the target policy's action for the next state s' using the target actor network's weights $\theta^{\pi'}$.
- $\theta^{Q'}$ are the target critic network's weights.

3.4.2 Actor Update

The actor network is updated using the deterministic policy gradient:

$$\Delta\theta^\pi = \frac{1}{N}\sum_i \nabla_a Q(s,a;\theta^Q)|_{a=\pi(s;\theta^\pi)} \nabla_{\theta^\pi}\pi(s;\theta^\pi)$$

Where:

- θ^π are the weights of the actor network.
- $\nabla_a Q(s,a;\theta^Q)|_{a=\pi(s;\theta^\pi)}$ is the gradient of the critic network with respect to the action evaluated at the actor's chosen action.
- $\nabla_{\theta^\pi}\pi(s;\theta^\pi)$ is the gradient of the actor's policy with respect to its weights.

3.4.3 Target Network Update

The target networks are updated using a soft update mechanism:

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

Where τ is the soft update rate.

3.5 Task 1:

1. Follow DDPG.ipynb

3.6 Report

1. Do the same steps as in DDPG.ipynb for a different continuous action space environment from gym