CSPB 3202 Final Project - OpenAI Gymnasium Agent

Taylor Larrechea

The University Of Colorado Boulder, College Of Engineering And Applied Science

Abstract

This project focuses on training a reinforcement learning agent that is used for decision-making in environments modeled by Markov Decision Processes (MDPs). A neural network architecture class was created to approximate the Q-value functions to be used to estimate future rewards for each action in a given state. In co-operation with this neural network, a Deep Q-Network (DQN) agent was constructed to learn the optimal policy for the Asteroids-v5 environment in the OpenAI Gymnasium. The strategy for the creation of the DQN agent and the neural network architecture as well as the results of the training process are discussed in this report.

Introduction

The OpenAI Gymnasium is a toolkit for developing and comparing reinforcement learning algorithms. These algorithms are used to train agents to make decisions in environments modeled by Markov Decision Processes (MDPs). Many different environments exist that can be used in the OpenAI Gymnasium, such as Atari games and other similar environments. For this specific project, the Asteroids-v5 environment was evaluated with the use of the neural network and DQN agent that was created. More information on the OpenAI Gymnasium can be found here **OpenAI Gymnasium**.

Reinforcement Learning

For a little bit of a background, reinforcement learning is a type of machine learning that is used to train agents to make decisions in environments. The goal of the training of the agent is to learn the optimal policy for the environment and subsequently maximize the reward that the agent receives. In a generic reinforcement learning problem, the agent interacts with the environment by taking actions and receiving rewards. The agent learns the optimal policy by estimating the value of each action in each state. The value of an action in a state is the expected reward that the agent will receive if it takes that action in that state. The value of an action in a state is estimated using the Q-value function. The Q-value function is the expected reward that the agent will receive if it takes an action in a state and then follows the optimal policy. The optimal policy is the policy that maximizes the expected reward that the agent will receive. The Q-value function is estimated using the Bellman equation.

Bellman Equation The Bellman Equation in a mathematical sense is

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \left(R(s,a) + \gamma \sum_{s'} P(s,a,s') V^{\pi}(s') \right). \tag{1}$$

The parts that make up the Bellman equation are:

- $V^{\pi}(s)$ - The value of state s under policy π.
- $\pi(a|s)$ - The probability of taking action a in state s under policy π.
- $R(s,a)$ - The immediate reward received after taking action a in state s.
- γ - The discount factor that determines the importance of future rewards.
- $P(s,a,s')$ - The probability of transitioning from state s to state s' after taking action a.
- $V^{\pi}(s')$ - The value of state s' under policy π.

The Bellman equation is used to estimate the value of each state under a policy. It provides a recursive decomposition of the value function, expressing the value of a state as the immediate reward plus the discounted value of successor states.

In reinforcement learning, the Bellman equation is a foundational concept for methods such as dynamic programming, temporal difference learning, and Q-learning. The Q-value function, which extends the Bellman equation, estimates the value of each action in each state and is used to update the agent's policy to maximize the expected reward.

A more in depth discussion of what the Bellman equation is and how it is used in reinforcement learning can be found **here**.

Q-Value Function The Q-value function, also known as the action-value function, represents the expected reward that the agent will receive if it takes a specific action in a given state and then follows the

optimal policy thereafter. The Q-value function is an extension of the Bellman equation and is defined as

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a'). \quad (2)$$

The components that make up the Q-value function are:

- $R(s,a)$ - The immediate reward received after taking action a in state s.
- γ - The discount factor that determines the importance of future rewards.
- $P(s'|s,a)$ - The probability of transitioning to state s' after taking action a in state s.
- $\max_{a'} Q(s',a')$ - The maximum Q-value over all possible actions a' in the next state s', representing the optimal future action-value.

The Q-value function is fundamental to many reinforcement learning algorithms, such as Q-learning, where it is used to iteratively update the value of state-action pairs and thereby guide the agent towards the optimal policy. The Q-value function is crucial for the agent to learn the optimal policy and maximize the expected reward. It is essentially the core of the reinforcement learning process.

A more in depth discussion of what the Q-value function is and how it is used in reinforcement learning can be found **here**.

Deep Q-Network

In the construction of this learning agent, a Deep Q-Network (DQN) was constructed to help train the agent to learn the optimal policy for the Asteroids-v5 environment. The DQN is a type of reinforcement learning algorithm that uses a neural network to approximate the Q-value function. The neural network is trained to predict the Q-value of each action in a given state.

The primary purpose of creating this network is to map states to Q-values. The mapping that is produced from this network is used to allow the agent to make informed decisions about which actions should be take to maximize the expected reward.

QNetwork Class

In the context of this project, the QNetwork class was created as a neural network architecture to approximate the Q-value function. The QNetwork class is a feedforward (a neural network that does not form a cycle) neural network that consists of three fully connected layers:

- The input layer, which takes the state as input and flattens it to be processed by the subsequent layers.
- The first hidden layer, which consists of a linear transformation followed by a ReLU activation function. This layer transforms the flattened input state into a 64-dimensional vector.
- The second hidden layer, which also consists of a linear transformation followed by a ReLU activation function. This layer further processes the 64-dimensional vector, maintaining the same dimensionality.
- The output layer, which performs a linear transformation to map the 64-dimensional vector to the Q-values for each possible action. The number of outputs corresponds to the number of actions the agent can take.

The class that was constructed for the neural network architecture consists of the following methods:

- **__init__**: The constructor method that initializes the neural network architecture given the input shape, action size, and a random seed. It sets up the network architecture by initializing the weights and biases.
- **forward**: The forward method that takes the state as input and passes it through the network to produce the Q-values for each action. It returns the Q-values for each action.
- **_init_weights**: A helper method that initializes the weights of the network using the Kaiming Uniform initialization method.

This class is crucial for the agent to learn the optimal policy for the environment. The 'torch' module in Python was used to create the neural network and other aspects of this project. For further reading on the 'torch' module, please refer to **PyTorch**.

The code for this class was largely inspired by the example found in **this** PyTorch example.

DQN Agent

Now that a neural network was created to help approximate the Q-value function, a Deep Q-Network (DQN) agent was then constructed to learn the optimal policy for the Asteroids-v5 environment. This is the agent that was trained and then used to make decisions in the environment.

DQNAgent Class

The DQNAgent class was created to represent a DQN agent that would subsequently be trained to learn the optimal policy for the Asteroids-v5 environment.

The DQNAgent consists of some key components that are essential for the training process:

- **qnetwork_local** - The main network that is used to select actions.
- **qnetwork_target** - The target network that is used to evaluate the Q-values of the next states.
- **optimizer** - An optimizer that is used to update the weights of the local network.
- **batch_size** - The number of experiences that are sampled from the replay buffer for each training iteration.
- **gamma** - The discount factor that determines the importance of future rewards.
- **tau** - The interpolation parameter that is used to update the target network.
- **update_every** - The number of time steps between updating the target network.
- **eps** - The exploration rate that determines the probability of selecting a random action.
- **eps_min** - The minimum exploration rate.
- **eps_decay** - The rate at which the exploration rate decays over time.

With these components, the following methods were created to train the agent:

- **__init__** - The constructor method that initializes the DQN agent given the state size, action size, and a random seed. It sets up the agent by creating the local and target networks,
- **step** - A method that takes the state, action, reward, next state, and done flag as input and stores the experience in the replay buffer. It then samples a batch of experiences from the replay buffer and uses them to train the agent.
- **act** - A method that takes the state as input and returns an action based on the current policy. It uses an epsilon-greedy policy to select actions.
- **sample** - A method that samples a batch of experiences from the replay buffer.
- **learn** - A method that samples a batch of experiences from the replay buffer and uses them to train the agent. Computes the Q-values for the next states, computes the Q-targets using the rewards and Q-values, computes the loss between the expected Q-values and target Q-values. This is all used to update the target Q-network using the local Q-network.
- **soft_update** - A method that updates the target network using a soft update strategy.
- **decay_epsilon** - A method that decays the exploration rate over time.

With the QNetwork and DQNAgent class now constructed, the training process was now ready to begin. The source code for this agent can be found in '**lib/agent.py**'. Many of the methods took advantage of the 'PyTorch' module where the documentation for this module can be found at **PyTorch Documentation**.

Training

With the QNetwork and DQNAgent classes now constructed, the training process is now possible to begin. The training process for this project used a function called **dqn** that takes in the number of episodes, max time steps per episode, the value of epsilon when it starts, the value of epsilon when it stops, the value of the epsilon decay rate, and the name of the agent that is to be saved by using 'torch'. In the training process, the agent selects an action using the current state and epsilon value, the action is then taken in the environment where the next state, reward, and termination flags are returned. The agent then calls its step method to store the experience in turn training the agent. This process repeats until the game ends for the current episode and then continues until all the episodes have finished running.

After each episode, the agent is saved to a file using the 'torch' module. This is done so that one can evaluate an agent after training it without having to retrain it. Upon running '**lib/training.py**', the user is asked to select a name for the agent. This name is then what the agent is saved as with the '.pth' extension.

Training Results

The training process for these models consisted of changing the number of episodes for each agent. In total, there were twelve agents that were trained with different numbers of episodes for each. The agents and their names are as follows:

- **Agent 10** - 10 episodes.
- **Agent 20** - 20 episodes.
- **Agent 30** - 30 episodes.
- **Agent 40** - 40 episodes.
- **Agent 50** - 50 episodes.
- **Agent 100** - 100 episodes.
- **Agent 200** - 200 episodes.
- **Agent 300** - 300 episodes.
- **Agent 400** - 400 episodes.
- **Agent 500** - 500 episodes.
- **Agent 1000** - 1000 episodes.
- **Agent 2000** - 2000 episodes.

For each agent that was trained, the episode number, score of the current episode, and the running average score of the current training session were recorded and saved to a csv file. The csv files for the

training part of this project can be found from the root directory of this repository in '**Training Results/**'. For every training episode, the initial ϵ value was set to 1.0, the end for the ϵ value was set to 0.01, and the decay rate for ϵ was set to 0.995. The maximum number of time steps per episode was set to 1000. These parameters were all held constant across the training process.

After the training of the agents completed, the results were then analyzed to determine which agent performed best. The metric that was used to determine the best agent was the average score of the agent after all of its episodes concluded. The average score of each agent can be seen in the table below:

Agent	Average	Min	Max
Agent 10	915.0	180.0	1180.0
Agent 20	702.0	240.0	1080.0
Agent 30	742.33	210.0	1180.0
Agent 40	711.0	160.0	1180.0
Agent 50	766.40	210.0	1180.0
Agent 100	740.80	130.0	1440.0
Agent 200	835.80	160.0	1390.0
Agent 300	701.40	130.0	1590.0
Agent 400	525.10	110.0	1280.0
Agent 500	371.0	80.0	1280.0
Agent 1000	439.90	110.0	1610.0
Agent 2000			

The agent that had the highest average score during training was Agent 10, the agent with the lowest score during training happened to be Agent 500, and the agent with the highest score during training was Agent 1000. There really is not a correlation between the number of episodes and the average score of the agent, at least during training that is. In fact, some of the agents that had the higher number of episodes had lower averages in comparison to those of lower episodes.

If you look at the results from Agent 500 and Agent 20 for example, Agent 500 had an average score of just 371.0 while Agent 20 had an average score of 702.0. The minimum score for Agent 500 was smaller than that of Agent 20's but Agent 500 did have a higher single episode score than that of Agent 20. In light of this, we can't really use this data to determine which agent is actually the best. To determine this, we need to evaluate the agents in the environment.