Custom DQN trainer for ml-agents toolkit and a multi-agent scenario in Unity

Bruno Ledda
Alma Mater Studiorum - University of Bologna
Department of Computer Engineering
bruno.ledda@studio.unibo.it

June 14, 2024

Abstract

This paper presents the design, implementation, and evaluation of the DQN reinforcement learning algorithm, integrated as a custom plugin with the ML-Agents toolkit and Unity. The goal is to achieve optimal performance during the training of multiple agents, which must learn to interact with a custom Unity environment and cooperate to reach a common objective.

1 Introduction

Nowadays, deep reinforcement learning algorithms are predominantly used to train autonomous agents to solve tasks in complex environments without predefined rules. Next, we will explore how to train multiple agents in a Unity 3D environment through experience and feedback. This involves providing a numerical reward/penalty system and implementing the DQN algorithm, a popular off-policy algorithm based on Q-learning, which uses neural networks to find the best policy for decision-making.

2 Technological background

2.1 Unity 3d

Unity is a popular game engine among independent and professional developers, known for its versatility and ease of use. It enables the creation of applications beyond the gaming field, including business applications, web applications, and virtual reality (VR)

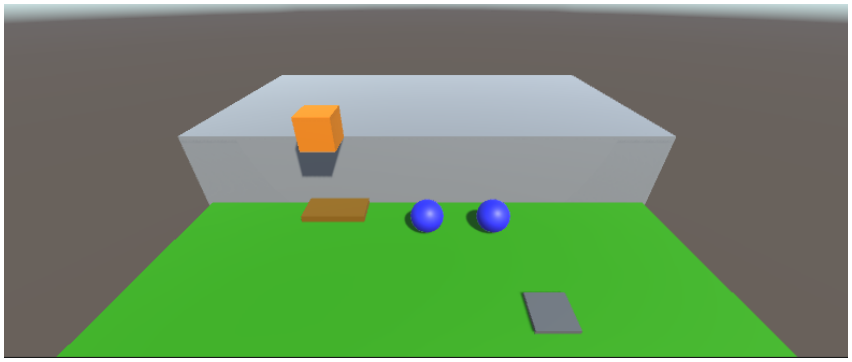

Figure 1: Simulation environment with 2 agents, a target and objects needed to reach it

experiences [1]. For this project, Unity was used to create a simple simulation environment where two agents, represented by blue balls, interact with other objects to reach a target represented by an orange cube. This target is the goal behavior that the agents must learn. The environment includes two additional objects: a movable platform (the brown object) and an activator (the grey object). The platform starts on the ground and can be activated only if an agent-ball is placed on the activator. If an agent falls off the ground or reaches the target, the environment resets to its initial state (as shown in Fig. 1).

Regarding the reward system, this multi-agent scenario required two types of rewards: single agent rewards and group rewards. The single agent reward encourages individual behaviors such as reaching the platform or the activator, while penalties are given to agents that fall off the ground or take too long to reach the goal. The group reward promotes cooperative behavior by rewarding the entire group if one agent is on the platform while the other is on the activator. Here's a summary of the reward system:

- For each step that agent is on the platform, it receive a reward of +0.5
- For each step that agent is on the activator, it receive a reward of +0.5
- If an agent falls from the playable area, it receive a penalty of -0.4 and the group receive a penalty of -0.1
- If one of the agents reach the target position, the group receives a reward of +100
- If both the agents are located respectively on the platform and on the activator, the group receives a reward of +2
- Else if they are not correctly located, for each step the group receives a penalty of -0.2/Max Environment Steps, which means that before that the environment is reset for taking too long time in reaching the cooperative behaviour, a total of -0.2 penalty is assigned to the group.

Unity is the most widely used tool in the world, with a vast and active community of users and an unparalleled wealth of material available for learning and development.

2.2 ML Agents toolkit

The ML-Agents Toolkit is an extension of Unity that enables the use of machine learning and artificial intelligence to create intelligent agents that can interact with their environment. It is based on technologies such as deep learning and reinforcement learning. ML-Agents provides tools for installing and configuring virtual environments and agents, allowing the creation of applications that can learn and improve over time [2]. It also includes a PyTorch backend for training models, which allows experimentation with different built-in algorithms and the possibility to create custom implementations.

For this project, according to the ML-Agents documentation page [3], it was possible to develop a custom trainer plugin for the **Deep Q-Network (DQN)** algorithm, which integrates seamlessly with the Unity environment thanks to the provided Python API. This code collects observations from the environment, as specified in the Unity scripts, and uses a neural network to predict the actions that the agents should take. This involves extending a few classes and overriding some methods to create custom policies and optimizers tailored to the specific task.

During training, a custom policy is created to implement the behavior policy, with an **evaluate** method called at each episode step. The core of the code is the **_process_trajectory** method, which manages interactions with the environment, collects experiences (comprising observations, rewards, and flags for episode completion or interruption), and stores them in a replay buffer. This method then calls a custom optimizer, which updates the target network's weights and estimates values to monitor the training process.

In Unity scripts, in addition to specifying the reward system, you must define the observation vector and the set of continuous or discrete actions that the agents can take. For this multi-agent task, the observation vector consists of 16 observations:

- 3 coordinates for the target position
- 3 coordinates for the agent position
- 3 coordinates for the platform position
- 3 coordinates for the activator position
- 2 values for the agent velocity
- 2 flags indicating whether the agent is located on the platform and the activator

The actions consist of 2 continuous actions for each agent: one for the horizontal axis and one for the vertical axis (agents are not allowed to jump).

2.3 Deep Q-Network algorithm

The Deep Q-Network (DQN) is a reinforcement learning algorithm, a variant of Q-Learning, that uses deep neural networks to learn optimal decision-making in complex environments. It is known for its ability to learn from errors and gradually improve performance. DQN has been successfully applied in various fields, such as Atari games, robot control, and complex system management.

DQN consists of two identical neural networks: the Q-network and the target network. The Q-network is updated at every training step, while the target network is updated periodically. An experience buffer stores experiences, allowing the network to be trained on batches of random data, reducing the correlation between learning experiences.

Similar to the base Q-learning algorithm, DQN uses the Bellman equation to update the Q-values iteratively, minimizing the difference between the predicted Q-values and the target Q-values:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[r + \gamma Q_\pi(s', a') \middle| \, s, a \right]$$

In my implementation, the reward r is the total reward, computed as the sum of the individual reward plus the group reward.

The loss function used for training the Q-network is typically the Mean Squared Error (MSE) between the predicted Q-values and the target Q-values:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

The action selection strategy during learning and evaluation is often epsilon-greedy. In this strategy, a random action is chosen with probability ϵ, while the action with the maximum estimated Q-value by the Q-network is chosen with probability 1 - ϵ. In this project, the epsilon-greedy strategy was implemented to balance the exploration/exploitation trade-off, with the ϵ parameter decreasing at the end of every training step. The frequency of target network updates is another important hyperparameter. Frequent updates can reduce the stability of learning, while infrequent updates can slow down learning.

Tha value choosen for the hyperparameters discussed above are:

- Replay buffer size: 1200
- Discount factor γ: 0.99
- ϵ start value: 0.99
- ϵ end value: 0.3
- ϵ decay value: 0.99995

- Target network update interval: 1000 steps

3 Architecture of Neural Networks

The architecture of neural networks plays a crucial role in the success of the Deep Q-Network (DQN) algorithm. A well-designed neural network can effectively learn the Q-function, which estimates the expected values of future rewards for each state-action pair. In this section, we describe the architecture of the neural networks used in this project, including the main Q-network and the target network. The target network has the same architecture as the main Q-network but its weights are updated less frequently to stabilize learning.

Both networks are feedforward networks with multiple fully connected layers. They take the observation of the current state as input and produce continuous actions as output:

- input dim: vector of 16 elements
- 2 hidden layers with 128 units and ReLU activation function
- output dim: vector of 2 elements

The hyperparameters used for the training are:

- batch size: 64
- learning rate: 1.0e-4

4 Results analysis and conclusion

To analyze the results, it is important to note that with the mentioned hyperparameters and reward system, the desired behavior of reaching the target and winning the game is never fully achieved. However, according to the line charts below, after sufficient training steps, the agents discover that they need to reach the platform and the activator (behavior that maximizes the group reward), even though they do not realize that remaining stationary on these objects could further increase their reward.

The results indicate that the best trade-off between single and group rewards is achieved after 80,000 steps. Beyond this point, approximately corresponding to 4,000 completed episodes, both mean episode rewards decline due to overfitting of the networks.

In conclusion, the agents begin to learn effective strategies after 80,000 steps, but they appear to be stuck in a local maximum. To reinforce the correct behavior, further experimentation with different hyperparameters and reward signals is necessary.

Figure 2: Mean Episode Reward over training steps

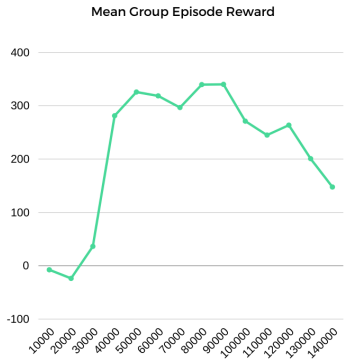

Figure 3: Mean Episode Group Reward over training steps

References

[1] Pietro Polsinelli, *"Why is Unity so popular for videogame development?"*, 05 December 2013, https://designagame.eu/2013/12/unity-popular-videogame-development/.

[2] *"ML-Agents Toolkit Overview"*, last consultation: 12 June 2024, https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/.

[3] ML-agents Documentation, *Custom Trainer Plugin*, last consultation: 12 June 2024, https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Tutorial-Custom-Trainer-Plugin.md/.