Udacity – Deep Reinforcement Learning Nanodegree

Continuous Control project

Deep reinforcement learning

Marwa Kandil
4-10-2021

Table of Content

1. Introduction

One of the most fascinating areas of AI is the concept of reinforcement learning (RL) as machines learn from their own experiences and optimize their actions based on experience. Agent is rewarded or penalized based on their actions and the actions that get to the optimal outcome are rewarded. Agents of RL managed to learn actions in complex environments but are limited due to intensive computations. Recently, RL is combined with deep neural network resulting deep Q-network (DQN) surpassed all previous algorithms and was able to learn concepts such as object categories directly from raw sensory data [1]. Deep reinforcement learning (DRL) agents, such as DQN, can solve high dimensional observational spaces such as achieving human score in Atari games and chess. However, DQN can only handle discrete and low dimension action spaces. This means that DQN cannot solve physical control tasks as these tasks have continuous and high dimensional action spaces. Actor-critic algorithms are designed to solve continuous control problems by utilizing value-based and policy-based RL methods together to train the agents.

There are many open-source tools train Deep reinforcement learning (DRL) agents to encourage large-scale innovation for multiple fields such as video games. Unity Machine Learning Agents (ML-Agents) is one of the main open-source tools that provide intelligent agents and rich environment for game development. The ML-agents Train and embed intelligent agents by leveraging state-of-the-art deep learning technology [2]. In this project I used Unity's rich environments to design, train, and evaluate actor-critic DRL algorithms.

For this project, I trained an agent to move one doubled-joined arm to target locations. The environment has state space of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. The agent generates an action vector with four numbers, corresponding to torque applicable to two joints. The agent gains a reward of +0.1 for each step that the agent's hand is in the goal location. The aim of the agent is to maximize reward, thus, maintain its position at the target location for as many time steps as possible. The task is solved when the average score is 30 for 100 consecutive episodes. I used two algorithms to solve this problem. The first algorithm used was vanilla Deep Deterministic Policy Gradient (DDPG) and the second algorithm used is a DDPG with prioritized replay buffer.

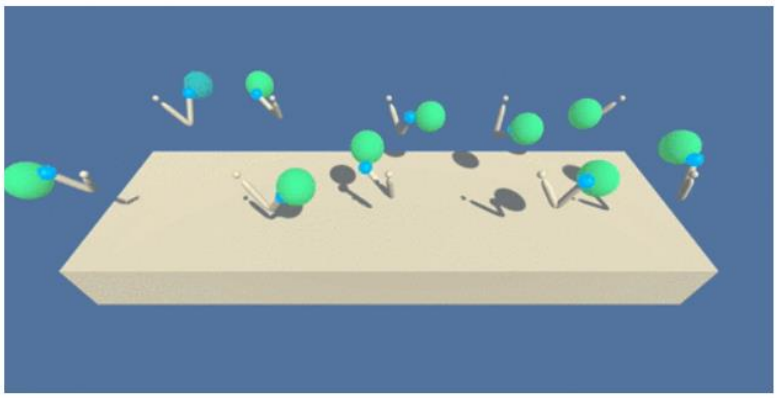

Figure 1: Unity ML-agent Reacher Environment

2. Background

Reinforcement learning framework consists of five main parts: states, actions, environment, agent, and rewards [3]. States are the different situations returned by the environment. Actions are set of possible moves that the agent can take to optimize the reward. The agent is a software that utilizes a policy to determine the next action based on the current state. The agent's role is to evaluate different policies and identify the policy that maximizes reward. Thus, the agent evaluates the expected long-term return of following the agent's policy also known as the value function V(S). The value function uses discounted rewards as the effect of the reward is reduced the further away we move in the future. A better way to assess the policy is to use action-value function Q(S,A), it is the long-term return of an action taken under policy at the current state. The RL agent uses Bellman equation as an iterative update to estimate the action-value function such that:

$$Q^*(s,a) = E_{s'}[r + \gamma max_{a'} Q^*(s',a')|s,a]$$

Where the variables s and a stand for state and action, Q^*(s,a) stands for action-value function for s and a, γ stands for reward discount rate. Any function approximator can be used to estimate the action-value function, such as the non-linear function approximators of neural networks as Deep Q-network (DQN) [1]. The two main methods in RL are value-based and policy-based methods. Value based methods, such as DQN, uses NN to optimize the action-value function. Policy-based methods use NN to approximate policy functions, thus, optimizing the action-value function but suffers from high variance. Actor-critic (AC) algorithms aim to use the value-based techniques to further reduce the variance in policy-based methods. AC agents learn from their actions and uses the critic to identify good and bad actions. Thus, AC are more stable than value-based, and need less samples and time than policy-based.

Deep Deterministic Policy Gradient (DDPG) is an actor-critic method where the critic uses a maximizer to approximate the Q-value over the next state [4]. The actor estimates the optimal policy deterministically, by generating the best possible action at any input state. The critic is updated using the loss function between the expected Q-value and the target Q-value as shown below:

$$L = \frac{1}{N}\sum_i \left(y_i - Q\left(s_i, a_i | \theta^Q \right) \right)^2$$

Update the actor policy using the sampled policy gradient as the equation below. DDPG uses replay buffers and preforms soft-update to the target network with ratio of the regular network to allow faster update.

$$\nabla J_{\theta\mu} \approx \frac{1}{N}\sum_i \nabla Q(s,a|\theta^q)\Big|_{s=si,\, a=\mu(si)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{si}$$

Similar to DQN, DDPG uses experience replay buffer as it allows the agent to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of experience. The actor and critic are updated at each timestep by sampling a minibatch uniformly from the buffer. One possible way to improve DDPG implementation is to use prioritized experience replay to improve learning.

Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability [5].

3. Implementation and Results

In this paper I implemented vanilla DDPG and DDPG with prioritized experience replay (DDPG_PER) to solve the 1 agent Reacher problem. The neural network model that I used for the critic consists of one input layer that has same shape as the number of states, one linear hidden layer with number of inputs as 128 + action space size and number of outputs as 128. Finally, I used batch-normalization layer to normalize samples within mini-batches, and an output layer. For training, Adam optimizer was used with learning rate of 2e-4. The neural network model that I used for the actor consists of one input layer that has same shape as the number of states, one linear hidden layer with number of inputs and number of outputs as 128 each. Finally, I used batch-normalization layer to normalize samples within mini-batches, and an output layer with the action space size as the number of outputs. For training, Adam optimizer was used with learning rate of 2e-4.

For both implementations, DDPG and DDPG_PER, I used a replay buffer of size 100,000 unit and 64 mini batch size, discounted factor (ɣ) of value 0.99, and soft-update ratio (τ) of 1e-3. To ensure balanced trade off between exploration and exploitation, Ornstein-Uhlenbeck noise generator for exploration, noise decay rate of 0.99. The scores throughout the episodes were tracked and presented on the plot below every 100 episodes. As shown the DDPG model solved the problem in 181 episodes and the DDPG_PER solved the model in 249 episodes. The DDPG model was faster to reach the solution, but the Figure below shows clearly that there was a sudden jump in the score values. This indicates that the model is not very stable. On the other hand, DDPG_PER too longer to solve the problem, yet the learning trend was much smother. When using prioritized experience replay, initially the sampled experiences are random but as time progresses experiences with higher importance have higher priority. Therefore, learning with DDPG_PER was much more stable than with DDPG.

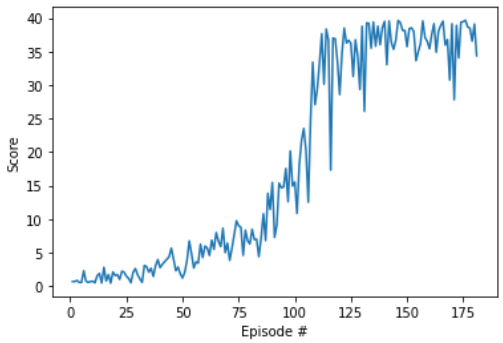

Figure 2: DDPG average score vs training episodes

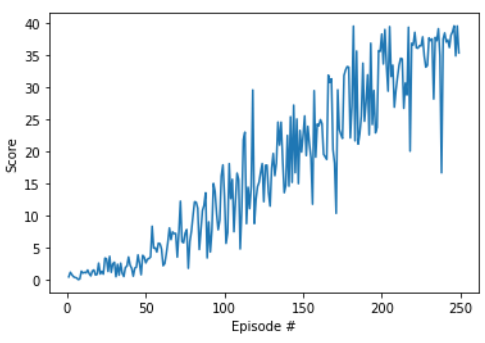

Figure 3: DDPG_PER average score vs training episodes

Algorithm	Total number of episodes	Average Score	Test score
DDPG	181	30.15	39.67
DDPG_PER	249	30.15	37.82

4. Conclusion and Future Work

In this project, I implemented DDPG and DDPG with priority experience replay to train and evaluate a double-joined arm agent to follow target in the unity ML-agent Reacher environment. The results show that using DDPG the agent solved the problem in 181 episodes, on the other hand, DDPG_PER solved the problem in 249 episodes but the leaning process was more stable.

Future work to the current solution would be to experiment with Proximal Policy Optimization (PPO) to determine the appropriate policy with gradient methods. Furthermore, the solution can be modified to solve the 20 agents version of the environment by implementing A3C method where multiple agents who all interact with the environment separately and send their gradients to the global network for optimization in an asynchronous way.

5. References

[1]: V. Mnih et al., "Human-level control through deep reinforcement learning", Nature, vol. 518, no. 7540, pp. 529-533, 2015. Available: 10.1038/nature14236 [Accessed 3 September 2021].

[2]: U. Technologies, "Machine Learning Agents | Unity", Unity, 2021. [Online]. Available: https://unity.com/products/machine-learning-agents. [Accessed: 03- Sep- 2021].

[3]: R. Sutton and A. Barto, Reinforcement Learning, 2nd ed. 2019.

[4]: T. P. Lillicrap et al., "Continuous control with deep reinforcement learning", arXiv, vol. 150902971, 2015. [Accessed 4 October 2021].

[5]: T. Schaul, J. Quan, I. Antonoglou and D. Silver, "Prioritized Experience Replay", arXiv, vol. 151105952, 2016. [Accessed 4 October 2021].