Deep Reinforcement Learning Nanodegree – Continuous Control

Udacity Nanodegree Program

Prabowo Setiawan

May 22, 2020

Introduction

This report aims to examine and discuss the continuous control project, both the implementation and the results. The environment consists of 20 double-jointed arms. The agent's hand is expected to follow the goal location in order to attain positive rewards. In other words, the agent is rewarded +0.1 for every step the agent's hand is within the goal location.

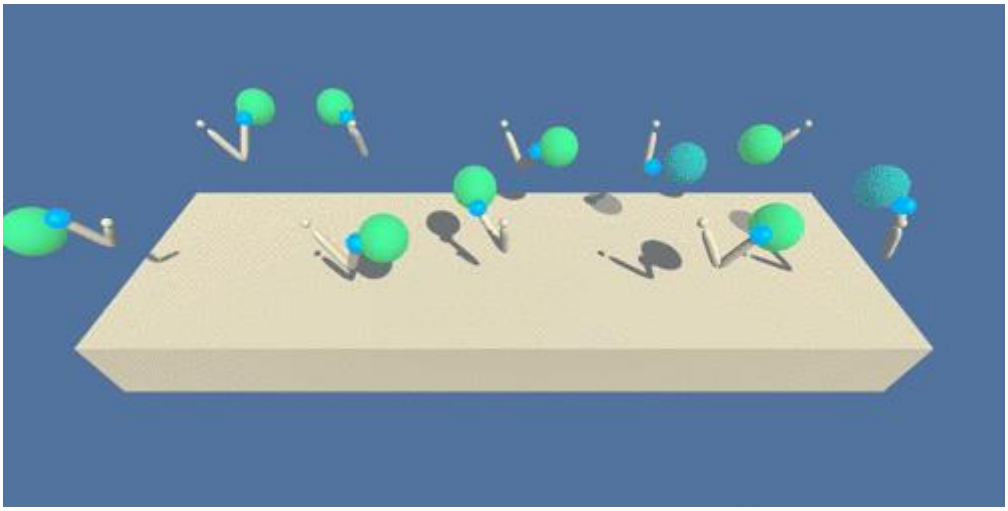

Figure 1: The Unity environment

In each episode, an agent is expected to obtain as many rewards or scores as possible. There are several ways that the agent can achieve this in the environment. The actions are consisted of vectors with four numbers corresponding to the torque applicable to two joints:

- Each entry of the action vector should be a number between -1 and 1 (due to the continuous nature of the problem)

As for the state-space, there are 33 observation spaces available including:

- Arm's position
- Arm's rotation
- Arm's velocity
- Arm's angular velocity

The agent is considered to solve the environment once it attains an average score of at least 30 over the past 100 consecutive episodes.

Project Approach

In order to solve the environment, a policy-based method known as Deep Deterministic Policy Gradient (DDPG) is implemented. DDPG agent works similarly to DQN, with the difference of concentrating more on obtaining the policy rather than Q-values. DDPG also uses several other components which are not commonly implemented in the DQN. One common aspect, however, is that the DDPG still uses replay buffer in stabilizing the learning process of the agent.

DDPG is a model-free approach in which the algorithm learns competitive policies in solving an environment (Lillicrap et al, 2016). In this algorithm, Actor and Critic network is defined using the following architecture:

- Input: size of state space
- Layer 1: 128 layers with ReLU activation function
- Layer 2: 128 layers with ReLU activation function

    - Critic's layer 2 input is a concatenation of the first layer output and action vectors
- Output: size of action vectors
    - Actor's output layer uses tanh activation function while Critic's does not

This configuration is chosen in order to simplify the model (instead of using 256 and bigger layers) to reduce computation time. The next parameter to be determined is the epsilon, ε, for epsilon-greedy choosing actions. There are three different parameters needed to specify ε:

- ε_0 (initial epsilon) = 1.0
- ε_{decay} (decaying factor of epsilon) = 1e-6 (subtractive not multiplicative)
- ε_{min} (minimum value of epsilon) = 0.01

It is noted that the epsilon behaves differently in this method compared to the one implemented in DQN. The epsilon is used in conjunction with the Ornstein-Uhlenbeck (OUNoise) to explore the environment. The epsilon value initially starts with ε_0 to enforce equal amount of exploration within the training environment. As the agent is being trained, the value of epsilon is reduced by the factor until it reaches the minimum value of epsilon.

The next aspect of the approach is the implementation of updateTargetNet. Since the environment has 20 double-jointed arms, it is better to update the network once in awhile rather than for every iteration. This reduces the oscillation of scores and therefore helps in assisting the agent to learn faster and better. However, if the network is only updated once every 20 time steps, the agent would not perform very well. In order to address this problem, the network learns from replay buffer 10 times, every 20 time steps within the episodes. This process prevents oscillations of scores as well as enables the network to learn just enough to make progress. Note that these numbers are determined through trial and error process.

The last but not least is the gradient clipping. Without the gradient clipping, the agent is most likely will experience a downtrend in scores at some point during training. As advised by Udacity in the benchmark implementation of the project, the use of gradient clipping helps tremendously in keeping the agent to learn more consistently for each episode while maintaining consistent scores' growth.

Results

Based on the agent trained using the chosen hyperparameters and random seed number, the DDPG's performance can be observed below.

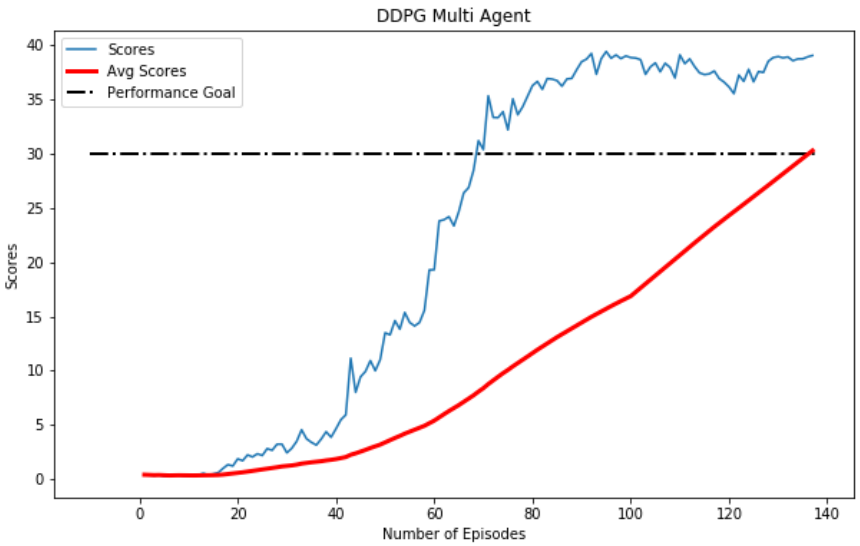

Figure 2: DDPG Scores and Average Scores

It is examined from the plot that DDPG is able to solve the environment in just 137 episodes. It finishes training with score of 30.30. It can be seen that the agent learns consistently with increasing average scores all the time. It is also interesting to see how the average scores of the agent starts to increase linearly starting from episode 100 onwards. This can be seen mainly due to the scores of the 20 agents plateau around 35-40. Nevertheless, the algorithm performs really well by showing consistent improvement in both scores and average scores.

Discussion

The plots available suggest that DDPG has a solid consistency in terms of learning from the environment. Both the scores and average scores seem to be consistently increasing until 65-70. In which after that many episodes, the maximum training score of the agent hovers around 35-40 while the average scores for the past (up to) 100 scores are steadily growing to 30.

In the process of completing the project, numerous combinations of hyperparameters were implemented for DDPG. Among 30 or so combinations, these hyperparameters were found to be the most optimal:

1. epsilon = 1.0
2. eps_min = 0.01
3. eps_decay = 1e-6
4. gamma = 0.99
5. tau = 0.001
6. learning rate
    a. 0.0001 for Actor
    b. 0.0003 for Critic
7. deep network:
    a. hidden_layer_1 = 128, with ReLU
    b. hidden_layer_2 = 128, with ReLU
    c. output_layer
        i. tanh activation function for Actor
        ii. no activation function for Critic
8. targetUpdateNet = 20
9. num_update = 10
10. seed = 0

One limiting factor in exploring all of these hyperparameters is definitely the time it takes to train the agent. Therefore, the model can be improved further by either tweaking the DDPG Agent's hyperparameters or the architecture of the neural network itself, e.g. learning rate, number of hidden layers, number of neurons per hidden layer, activation function (LeakyReLU instead of ReLU), etc.

Future Improvements

Due to the training duration, solving the environment by implementing multiple methods seem undesirable. However, given the computing power to solve the environment, one can explore other methods such as Proximal Policy Optimization (PPO) and Distributed Distributional Deterministic Policy Gradients (D4PG) in solving the Reacher environment. In addition, the behavior of this algorithm towards the singe agent version of the environment can also be studied to further see how robust it is.

Not only the algorithms themselves, one can also implement prioritized experience replay to further improve the learning process. Not to mention that by utilizing GPU further, a grid search optimization of hyperparameters can be studied in optimizing the performance of a specific method.

References

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. 2016. Continuous control with deep reinforcement learning. In: *ICLR 2016*. [Online]. [Accessed 22 May 2020]. Available from: https://arxiv.org/pdf/1509.02971.pdf