SAYON PALIT

23rd June 2020

Deep Reinforcement Learning

Project 2 :Continuous Control

OVERVIEW

This project aims to use deep reinforcement learning ,more specifically DeepMind's Deep Deterministic Policy Gradient algorithm to solve a Unity based environment where the main objective is to train a robotic arm to move to the target location.

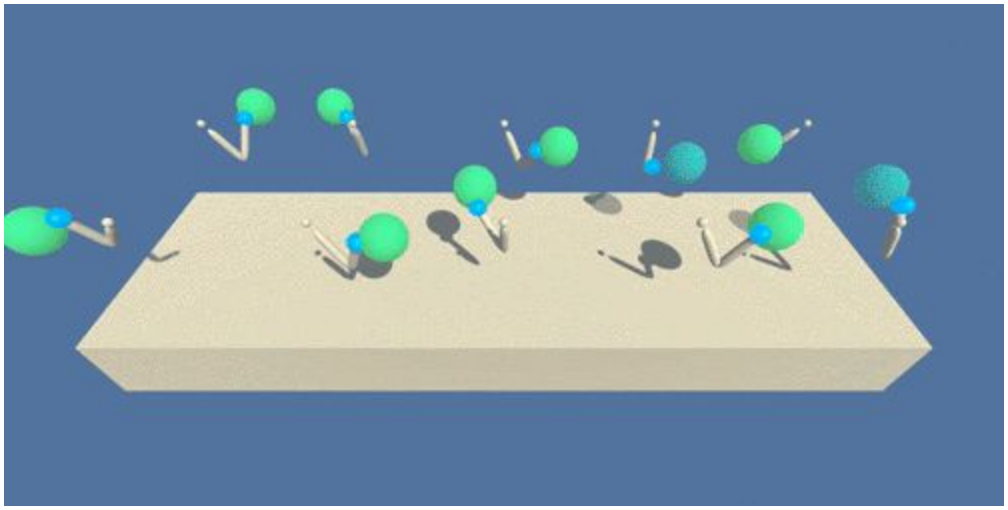


Figure 1 depicts the environment.

- A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.
- The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

- The task is episodic, and in order to solve the environment, your agent must get an average score of +30 over 100 consecutive episodes.

LEARNING ALGORITHM

The learning algorithm used in this project is part of the **Actor-Critic** family of Deep Reinforcement Learning Algorithms.

Why Actor-Critic Models ?

It is not possible for Value-based models like DQN to provide results in continuous action spaces like the one in this current project.Yes, discretization is possible , but after a certain extent the optimization entailed in the Sarsamax algorithm becomes non-trivial as we have to find the maximal Q-value over a vast action space.

Enter into the scenario, Policy-based models which try to directly find the optimal policy function instead of the optimal Value function.Thus they can be used in discrete as well as continuous action spaces as well as stochastic and deterministic.The main issue with policy functions is to find a baseline for evaluating how good our proposed policy is.

Thus the Actor-Critic model uses a hybrid of both these techniques as follows:-

- Actor - The policy based model controls how our agent behaves.
- Critic - The value based model evaluates how good the action taken by the actor is.

Algorithm

1.For our purposes we use the Deep Q Network as the Critic and the Deep Deterministic Policy Gradient Network as our actor.

2. The DDPG algorithm maintains a parameterized actor function $\mu(s|\theta\ \mu)$ which specifies the current policy by deterministically mapping states to a specific action. The critic Q(s,a) is learned using the Bellman equation as in Q-learning.

Thus we use the output of the DQN network to update our policy as follows:-

$$\Delta\theta = \alpha * \nabla_\theta * (log\pi(S_t, A_t, \theta)) * Q(S_t, A_t)$$

Where θ are the weights of our network and ∇ is the gradient with respect to θ.

3.Just like in Deep Q Networks to remove correlation between the experience tuples due to the sequential nature of events in Reinforcement learning environments, we use a **Replay buffer** to store the experience tuples in the form of (st, at, rt, st+1).We sample from these tuples in minibatches and thus the DDPG Algorithm is off-policy.

4.Also the idea of **Fixed Q Targets** to remove the instability caused by updating the targets along with network weights is addressed in a similar manner here with a copy of the network acting as the target network being present for both Actor as well as Critic.Although it is to be noted that the update policy for the target networks is a little different. Here, a new technique was proposed by DeepMind called **Soft Updates ,** where we use another hyperparameter τ which is used to decide the magnitude of updates to the target network as given by below equation:-

theta_target = tau*theta_local + (1-tau)*theta_target

This means that the target values are constrained to change slowly, greatly improving the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist.

5.When learning from low dimensional feature vector observations, the different components of the observation may have different physical units (for example, positions versus velocities) and the ranges may vary across environments. This can make it difficult for the network to learn effectively and may make it difficult to find hyper-parameters which generalise across environments with different scales of state values. One approach to this problem is to manually scale the features so they are in similar ranges across environments and units. We address this issue by adapting a recent technique from deep learning called **batch normalization.**

6.For the purposes of exploration of the state space, we use **Ornstein-Uhlenbeck** process to generate temporally correlated exploration for exploration efficiency where μ` is the new policy and μ the old one.N is the noise generated by Ornstein-Uhlenbeck process.

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

MODEL ARCHITECTURE

The model architecture is an actor and critic neural net with three fully connected layers.

- The actor as well as critic networks take in state size as input and then outputs fc1_units(400)
- Next the leaky ReLU activated first layer output undergoes batch normalization.
- After BatchNorm the output it supplied to the second fully connected layer which outputs FC2_units(300)
- The Leaky ReLU activated output is passed on to the third and final fully connected layer which outputs the tanh activated action values.

```
super(Actor, self).__init__()
self.seed = torch.manual_seed(seed)
self.fc1 = nn.Linear(state_size, fc1_units)
self.bn1 = nn.BatchNorm1d(fc1_units)
self.fc2 = nn.Linear(fc1_units, fc2_units)
self.fc3 = nn.Linear(fc2_units, action_size)
self.leakiness = leakiness
self.reset_parameters()
```

- **Gradient Clipping :** was applied to stabilize the learning as the outsized gradients lead to plateauing or collapse of the training of the agent.
- **Learning Intervals :** Earlier iterations of this project learnt at every step which made training a very time-consuming process without any outright benefits to the agent's performance.So a new hyperparameter learn_every was introduced to to perform learning in every **learn_every timesteps**.

As part of each learning step, the algorithm samples experiences from the buffer and runs the Agent.learn() method learn_num times.

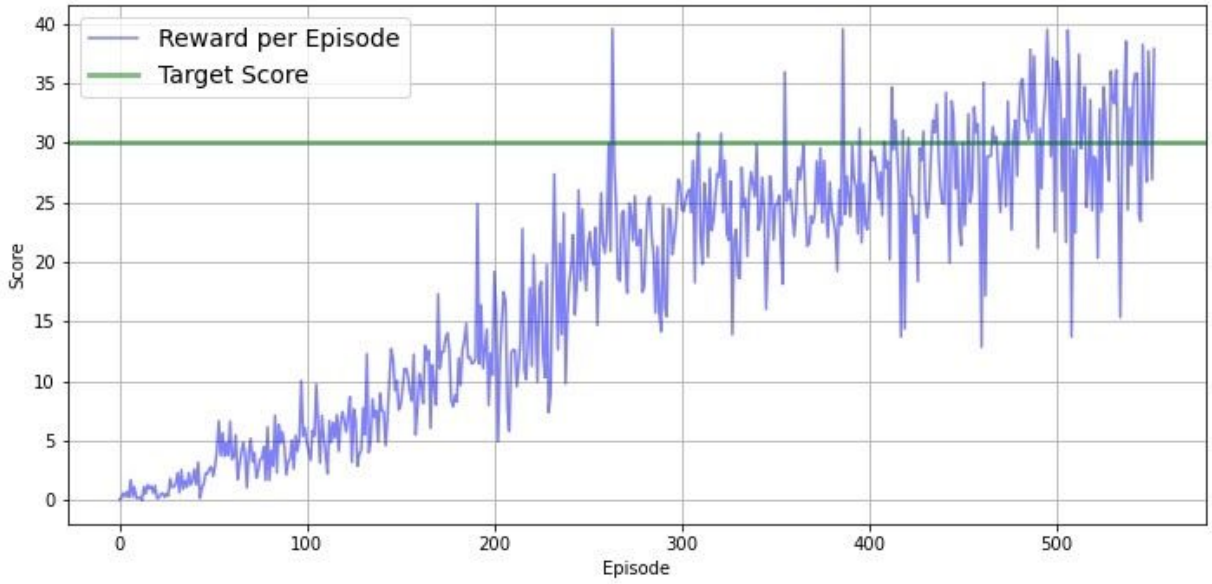
HYPERPARAMETERS

BUFFER_SIZE = int(1e6) - Replay buffer size
BATCH_SIZE = 128 - Mini-batch size
GAMMA = 0.99 - Discount factor
TAU = 1e-3 - Soft-update target parameters
LR_ACTOR = 1e-3 - Learning Rate of Actor
LR_CRITIC = 1e-3 - Learning Rate of Critic
WEIGHT_DECAY= 0 - L2 Weight Decay
LEAK_FACTOR = 0.01 - Leak factor of leaky_relu
LEARN_EVERY = 20 - Learning timestep interval
LEARN_NUM = 10 - Number of learning passes
GRAD_CLIPPING= 1. - Gradient Clipping

# Ornstein-Uhlenbeck noise
OU_SIGMA = 0.2 - Ornstein-Uhlenbeck noise parameter
OU_THETA = 0.15 - Ornstein-Uhlenbeck noise parameter
EPSILON = 1. - for epsilon in noise
EPSILON_DECAY = 1e-6 - decay rate for noise process

RESULTS

```
17%|                                                  | 100/600 [14:10<1:03:50,  7.66s/it]
Episode 100     Average Score: 2.63
33%|                                                  | 200/600 [28:55<56:32,  8.48s/it]
Episode 200     Average Score: 8.89
50%|                                                  | 300/600 [42:57<43:02,  8.61s/it]
Episode 300     Average Score: 18.79
67%|                                                  | 400/600 [57:26<29:15,  8.78s/it]
Episode 400     Average Score: 24.82
83%|                                                  | 500/600 [1:12:51<15:11,  9.11s/it]
Episode 500     Average Score: 28.22
92%|                                                  | 552/600 [1:21:46<07:06,  8.89s/it]
Environment solved in 553 episodes!     Average Score: 30.05
```



As can be seen from the plot above ,the current model solves the environment in 553 episodes with an average score of 30.05.

FUTURE WORK

- **Prioritized experience replay** is an improvement to the experience replay procedure, and the main idea behind it is to accord more interest on experiences that seems to be more important for the learning procedure.
- Other algorithms like **A3C**, **PPO and D4PG** may be more robust in terms of fine tuning hyperparameters and may be worth using in this environment.
- Also the hyperparameters as well as network layer structure can be further tweaked to reduce the solving time from the current 500-600 episodes to even lower.
- I would also like to implement the same model on the 20 Agents version of the Unity Environment.The learning process may be sped up to under 300 episodes with 20 different agents learning from the environment based on a single brain.

REFERENCES

1.ddpg-pendulum - I used this DDPG implementation by our Course Instructor Alexis Cook as the template for my project.

2.[1509.02971] Continuous control with deep reinforcement learning

3.Deep Reinforcement Learning Online Course - Udacity

4.Deep RL Bootcamp Lecture 7 SVG, DDPG, and Stochastic Computation Graphs (John Schulman)