Continuous Control Project Report

Yasmeen Khaled
June 6, 2021

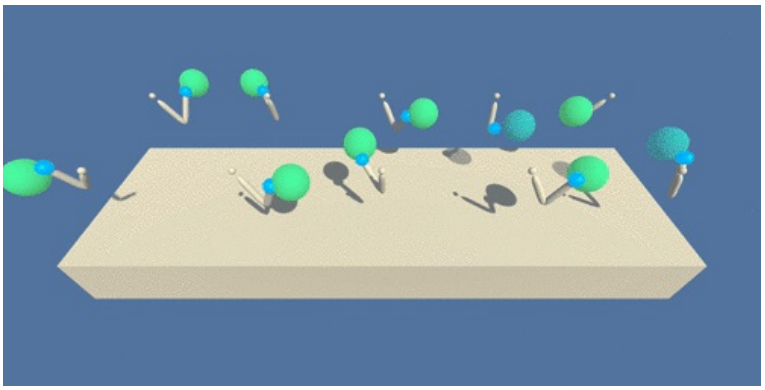

Figure 0.1: Reacher Environment

1 INTRODUCTION

During the last couple of years, with DeepMind introducing DQN which managed to beat human experts in Atari games, Deep Reinforcement Learning became a very hot topic that attracted alot researchers. The goal for any RL agent is to maximize its expected reward upon interaction with an environment. The idea behind the usage of neural networks is that approximating policies and/or value functions can actually output satisfying practical results. The environment used for this project he Udacity version of the Reacher environment. The goal of the agent is to maintain its position at the target location for as many time steps as possible, where the agent is a double-jointed arm can move to target locations. A reward of

+0.1 is provided for each step that the agent's hand is in the goal location.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.

2 ENVIRONMENT

The environment used to solve the task is the second environment in which 20 agents are used and the goal is to get +30 over 100 consecutive episodes among all the agents.

3 LEARNING ALGORITHM

The algorithm used to solve this project is the Deep Deterministic Policy Gradient (DDPG) following this research paper,which is a model free, off-policy actor-critic algorithm that is used to solve environments with continuous action-spaces. The algorithm is explained in the following figure:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 3.1: DDPG Algorithm

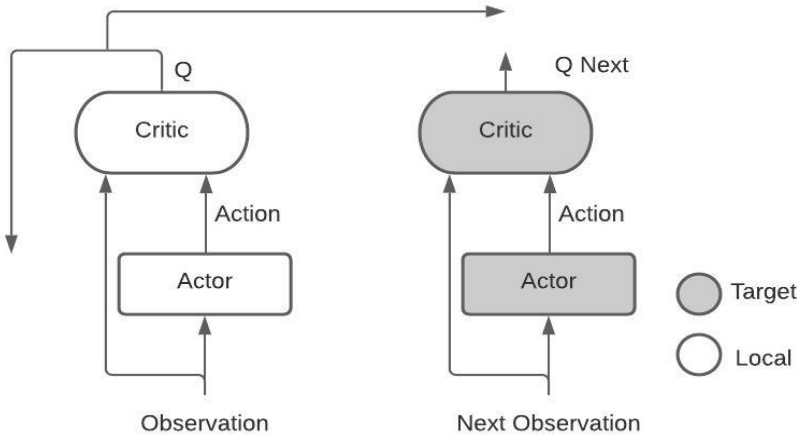

Figure 3.2: DDPG Diagram

Updating the actor is used using the local actor and critic network, we try to maximize the output Q, or in other words, minimize -Q. To train the critic, we use the next observation and pass it to the target actor and critic, then get Qtargetnext. Qtarget = Reward+ discount*Qtargetnext, and now we have Q and Qtargetnext so we want to minimize the difference between them. The hyperparamter values used were as follows:

Hyperparameter	Description	Value
BUFFER_SIZE	replay buffer size	int(1e5)
BATCH_SIZE	minibatch size	128
GAMMA	discount factor	0.99
TAU	for soft update of target parameters	1e-3
LR_ACTOR	Learning rate for the actor	1e-4
LR_CRITIC	Learning rate for the critic	1e-3
UPDATE_EVERY	how often to update the network	20
NUM_UPDATES	how many times update the network	200

The ddpg_agent is implemented as follows:

- **init**: the agent state_size and action_size is passed to the constructor and all the values are initialized. Two critic networks are initialized: local and target and same for the actor.
- **step**: it saves the experience in the replay buffer, and if enough samples are existing in the memory, the agents samples from the buffer and learns every update_every times with num_update times.

- **act**: returns the action vector chosen by actor_local network, trains the actor_local network and adds noise to the action to favor exploration.
- **learn**: it optimizes the losses as explained above and uses soft_update to transfer the local network weights to the target network. Gradient clipping was used when updating local_critic network.
- **soft_update**: the learn method updates the target network using this method which updates the target network based on the values of the local network.

The ddpg_agent.py also contains the ReplayBuffer class which has the following methods:

- **add**: which adds an experience to the replay buffer memory which is initialized as a deque with maxlen of BUFFER_SIZE.
- **sample**: which randomly samples a batch from the replay buffer memory.

3.1 Actor-Critic Networks Architectures

3.1.1 Actor Architecture

The network architecture was as follows:

```
Actor(
  (fc1): Linear(in_features=33, out_features=400, bias=True)
  (bn1): BatchNorm1d(400, eps=1e-05, momentum=0.1, affine=True,
   track_running_stats=True)
  (fc2): Linear(in_features=400, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=4, bias=True)
)
```

fc1 is passed to bn1 then RELU activator then the output is passed to fc2 then another RELU activator in the forward pass.
Af first, I tried adding 3 batch normalization layers one for the state size, one for fc1 and one for fc2, but then noticed that the training was showing very low progress per episode. After removing two of the batch normalization layer, the training improved significantly.

3.1.2 Critic Architechture

The network architecture was as follows:

```
Critic(
  (fcs1): Linear(in_features=33, out_features=400, bias=True)
  (bn1): BatchNorm1d(400, eps=1e-05, momentum=0.1, affine=True,
   track_running_stats=True)
  (fc2): Linear(in_features=404, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=1, bias=True)
)
```

fc1 is passed to bn1 then RELU activator then the output is passed to fc2 then another fc2 has extra four inputs which correspond to the 4 values of the action vector.

4 Training and Results

```
Episode 10    Average Score: 2.74
Episode 20    Average Score: 5.09
Episode 30    Average Score: 7.41
Episode 40    Average Score: 12.68
Episode 50    Average Score: 17.16
Episode 60    Average Score: 20.04
Episode 70    Average Score: 22.28
Episode 80    Average Score: 23.76
Episode 90    Average Score: 25.07
Episode 100    Average Score: 26.08
Episode 110    Average Score: 29.33
Episode 113      Average Score: 30.24
Environment solved in 13 episodes!       Average100 Score: 30.24

CPU times: user 3h 24min, sys: 8min 58s, total: 3h 32min 58s
Wall time: 3h 38min 11s
```

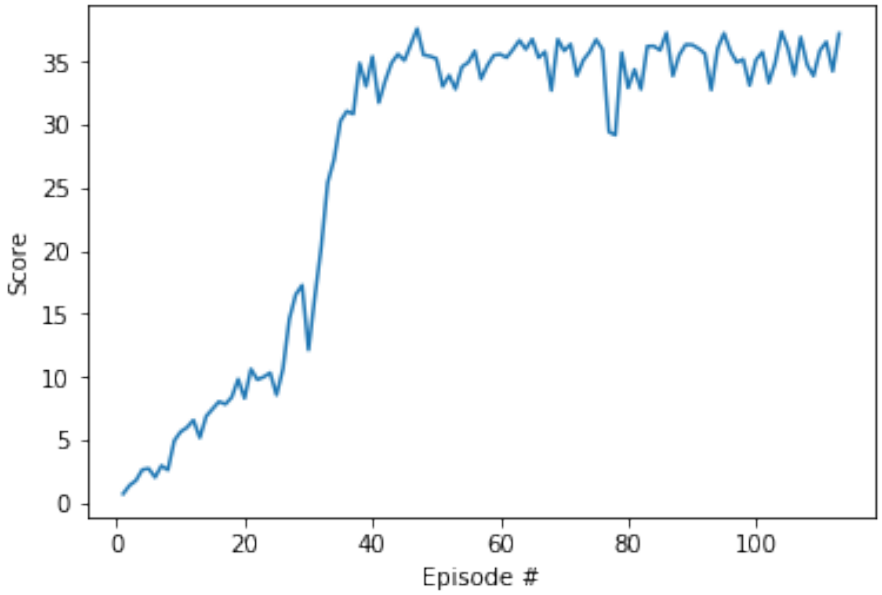

Figure 4.1: DDPG Training

My DDPG agent managed to solve the task in 113 episodes to be able to get a reward of at least +30 over 100 consecutive episodes.

4.1 TESTING

After loading the saved weights and testing the agent for one episode this was the result:

```
Total score of this episode: 25.166999437473713
```

5 FUTURE WORK

To further improve the performance of the model, the current methods are proposed to do in the future:

- **Twin Delayed DDPG (TD3)**: TD3 improves the performance of of DDPG by 3 main modifications which are: Clipped Double Q-learning to reduce positive bias and decouple action selection and evaluation, Delayed Policy updates where the policy is updated less frequently than the Q-function,for example, one policy update for every two Q-function updates ,and lastly, Target poilcy smoothing which adds noise to the target action to make it harder for the policy to exploit Q-function.
- **Hyper-parameter search**: one can try to train multiple agents with different parameters in order to reach the set that significantly improve the performance.
- **Noise Decay Factor**: Adding a noise decay factor which can be multiplied by the noise added to the actions, and this factor shall decrease overtime, this will help the agent converge faster to a better solution after training with enough samples.
- **Distributed Distributional Deterministic Policy Gradients (D4PG)**: one can try to implement this variation of DDPG also.