Solving Continous Control “Reacher” Environment using DDPG

#### Deep Deterministic Policy Gradient (DDPG)

#### The algorithm I use is largely based on the DDPG algorithm provided by the Udacity code example for solving the OpenAI Gym's Pendulum. The code is very much readable and provides a clear framework for further fine-tuning to solve other problems using DDPG. The link to the code as below:

#### https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum

#### Compared to DQN (Deep Q Network), DDPG is more suitable to solve continuous tasks, when the action space is continuous, e.g. physical control tasks.

#### https://cdn-images-1.medium.com/max/800/1*qV8STzz6mEYIKjOXyibtrQ.png

#### Model Architecture

#### I have followed suggested benchmark implementation to adjust the model architecture. The original model architecture was the same to the Udacity DDPG-pendulum algorithm. The Actor network is a neural network with 2 full connected layers with size of 400 and 300. Tanh is used as the activation function to map states to actions. The Critic network is similar to the Actor network except that the activation function is a fully connected layer with Relu, which maps states and actions to Q values.

#### Each agent adds its experience to a replay buffer that is shared by all agents. The trajectory has max\_t = 2000, and the networks are updated 10 times every other 10 timesteps. It is implemented as below, with LEARN\_EVERY = 10:

if int (t / LEARN\_EVERY ) % 2 == 1:

self.learn(experiences, GAMMA, t)

#### Hyperparameters

#### Below is the final hyperparameters I used to for getting the solution:

#### BUFFER\_SIZE = int(1e6) # replay buffer size

#### BATCH\_SIZE = 1024 # minibatch size

#### GAMMA = 0.99 # discount factor

#### TAU = 1e-3 # for soft update of target parameters

#### LR\_ACTOR = 1e-3 # learning rate of the actor

#### LR\_CRITIC = 1e-3 # learning rate of the critic changed from 1e-3

#### WEIGHT\_DECAY = 0 # L2 weight decay

#### UPDATE\_EVERY = 1 # update the target network every N timesteps

#### Things have tried

#### I think it’s also very useful to discuss what I have tried during the training (and learning) process. I have almost spent all my 50 GPU hours on trying different things, for example, how often do we update the network or how often do we update the target parameters of the network with the local parameters. Most of the trials were based on intuition and the progress of the agent learning, e.g., the reward scores, but that experience has definitely helped me to reach the final solutions.

#### *With single agent*

#### First, as the benchmark implementation suggested, I was using the single agent environment to test the water. The single agent didn’t learn at all initially and then I realize that I have set the max\_t a bit too low at 300. After I had changed this to 2000, the avg rewards improved to 1, but again stuck at that level. So it makes sense that we need to allow the agent to move away from the initial status to collect more experience to help learn useful information. However, there definitely was more work to be done.

#### So following the benchmark implementation suggestion, I introduced the gradient clipping into the critic network as there’s evidence that it can improve the model stability. However the result didn’t seem to improve significantly.

self.critic\_optimizer.zero\_grad()

critic\_loss.backward()

torch.nn.utils.clip\_grad\_norm(self.critic\_local.parameters(), 1)

self.critic\_optimizer.step()

#### Then I tried to make the local network only update the target network every 20 timesteps. This was to allow the local network to have slightly more ‘freedom’ to explore different states, but this also only improved the score marginally.

#### In addition to that, I had also added batch normalization to the network, according to the “Continuous control with deep reinforcement learning” paper. It makes sense that for solving the pendulum problem, batch normalization was not need as the state space is not that big, but for this Reacher problem with much higher dimension and variation in the state space, it is definitely needed.

#### Then again the benchmark implementation suggested that, “instead of updating the actor and critic networks 20 times at every timestep, we amended the code to update the networks 10 times after every 20 timesteps”. As this was done in the 20-agent environment, so for my single agent environment, I changed my agent to update the network only every 5 timesteps. This actually worked! It gave me a lot of encouragement after I see the below chart. The agent finally started learning something meaning and during the 100 episodes, the avg score stays above 1 most of the time after 40 episodes.

#### D:\Road to be Data Scientist\UdacityDeepReinforcementLearning\single agent-100episode.png

Figure 1: Single Agent Training - with network updating every 5 timesteps

#### Then I tried to execute the algorithm for longer time with GPU. I have got the result like below. It seemed the result did improve steadily, however it looked a bit unstable to me as it still can drop back to 1 at any point even after 250 episodes. Therefore, I have decided to switch to the multi-agent environment as the benchmark implementation solved the task with multi-agent. I hoped the multi-agent learning would provide more stability, but when I look at it retrospectively, I could have stuck to the single agent environment. Based on the discussion with fellow students, they have solved it with single agent, probably without using too many GPU hours.

#### D:\Road to be Data Scientist\UdacityDeepReinforcementLearning\single agent-300episode.png

Figure 2: Single Agent Training - with network updating every 5 timesteps

#### *With multiple agents*

#### Firstly I amended the code for the multi-agent environment and made the algorithm to learn 10 times for every 20 timesteps. The learning was quite slow in the Udacity Workspace even with GPU enabled, and I never found the way to keep the workspace alive without being timed out and lost all the progress. So I had to save the training progress using the checkpoint, however when I restart the training, the avg reward in the initial training went down a lot, as a lot of useful experience previously saved in the replay buffer was lot.

#### Below are the reward score I have got during my 4 attempts before I solved it. In total it was solved after 600 episodes, however I had been able to train the network continuously, it could be solved with about 400 episodes. You can find that, it always needs 50 – 100 episodes in the next attempt, just to reach the average score at the end of the last attempt. And in the first 2 attempts, I didn’t record the time spent for each episode. You may find in the last attempt, the time spent was more than the third, that’s because I ran out of GPU hours and had to train with CPU. Luckily, at that point of time, the agent was already quite intelligent, and it didn’t spend ‘too long’ to reach the goal score.

#### *First attempt:*

Episode 10 Average Score: 0.73

Episode 20 Average Score: 3.01

Episode 30 Average Score: 12.98

Episode 40 Average Score: 20.48

Episode 50 Average Score: 23.28

Episode 60 Average Score: 20.48

Episode 70 Average Score: 19.80

Episode 80 Average Score: 21.21

Episode 90 Average Score: 20.59

Episode 100 Average Score: 17.18

#### *Second attempt:*

Episode 25 Average Score: 14.59

Episode 50 Average Score: 17.26

Episode 75 Average Score: 19.06

Episode 100 Average Score: 20.64

Episode 125 Average Score: 23.17

Episode 150 Average Score: 24.20

Episode 175 Average Score: 24.69

Episode 200 Average Score: 24.46

#### *Third attempt:*

Episode 25 Average Score: 21.97 Time Spent: 57.69

Episode 50 Average Score: 21.75 Time Spent: 107.31

Episode 75 Average Score: 22.22 Time Spent: 107.88

Episode 100 Average Score: 22.77 Time Spent: 108.35

Episode 125 Average Score: 23.97 Time Spent: 108.01

Episode 150 Average Score: 25.29 Time Spent: 110.61

Episode 175 Average Score: 25.93 Time Spent: 109.02

Episode 200 Average Score: 26.92 Time Spent: 111.47

#### *Fourth attempt:*

Episode 25 Average Score: 26.93 Time Spent: 99.13

Episode 50 Average Score: 26.87 Time Spent: 111.34

Episode 75 Average Score: 27.34 Time Spent: 124.63

Episode 100 Average Score: 28.57 Time Spent: 149.13

Episode 125 Average Score: 29.33 Time Spent: 178.52

Episode 140 Average Score: 30.02 Time Spent: 180.38Problem Solved after 140 episodes!! Total Average score: 30.02

#### Another downside for having to separate the training into multiple attempts was that, I couldn’t report the continuous reward line chart. It would be nice but basically you could get an idea from the avg scores reported above. Below I reported the 20 agents’ average score in the final episode. Out of 20 agents, 16 of them were above 30 points.

#### D:\Road to be Data Scientist\UdacityDeepReinforcementLearning\final_20agents_scores.png

Figure 3:Agents Avg Score in the final episode (Solved)

#### Future Improvement

#### As suggested by the project instruction, Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG) should achieve better performance for this task. More details can be found in this paper: <https://arxiv.org/abs/1604.06778>. And I can try to write the implementation of Proximal Policy Optimization (PPO), which has also [demonstrated good performance](https://blog.openai.com/openai-baselines-ppo/) with continuous control tasks.

#### Reference:

#### Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG): <https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287>

#### Udacity DDPG-pendulum algorithm: <https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum>

#### Continuous control with deep reinforcement learning <https://arxiv.org/abs/1509.02971>