Solving Continous Control “Reacher” Environment using 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

In this report, I will have a brief review on the DDPG paper (https://arxiv.org/abs/1509.02971), discuss the model architecture of the implementation and the process of the training and reaching the final solution.

Deep Deterministic Policy Gradient (DDPG)

We have learnt that DQN can solve problems with high-dimensional observation spaces, however it can only handle discrete and low-dimensional action spaces. For continuous tasks like physical control, a policy gradient method would be more suitable. In the DDPG paper, the author presented a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces.

Key elements in the DDPG algorithm:

An actor-critic approach based on the DPG algorithm (Silver et al., 2014):

- 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
- The actor is updated by following the applying the chain rule to the expected return from the start distribution *J* with respect to the actor parameter (see equation (6) in the paper)

Batch learning: a minibatch approach which does not reset the policy at each update

Replay buffer: a finite sized cache storing *(st, at, rt, st+1).* When the replay buffer was full the oldest samples were discarded

Soft target updates: create a copy of the actor and critic networks, $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta\ \mu')$ respectively, that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau << 1$

Batch normalization: normalizes each dimension across the samples in a minibatch to have unit mean and variance.

Exploration: using $\mu'(s_t) = \mu(s_t|\theta^{\mu}_t) + N$ -with N ~ Ornstein-Uhlenbeck process

Below is the pseudo code for the DDPG algorithm:

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

In the DDPG paper, the author also mentioned that all the experiments used substantially fewer steps of experience than was used by DQN learning to find solutions in the Atari domain. This has suggested the potential that, given more simulation time, DDPG may solve even more difficult problems that those considered in the paper.

Model Architecture

I have followed the 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.

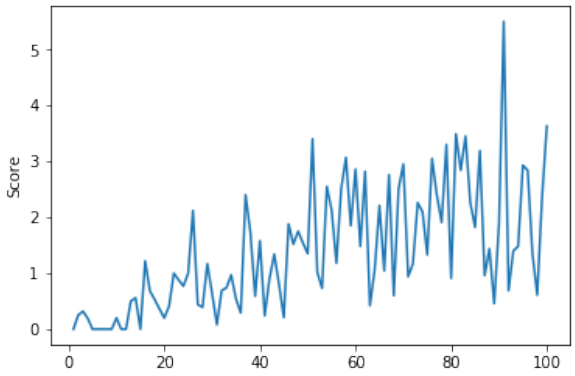

Figure 1: Single Agent Training - with network updating every 5 timesteps

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.

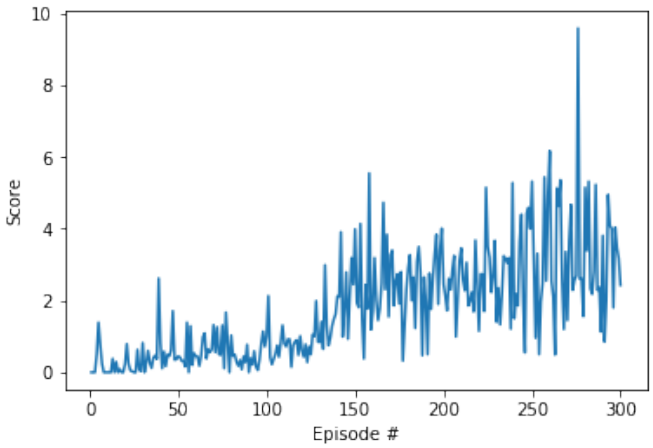

Figure 2: Single Agent Training - with network updating every 5 timesteps

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 afte
r 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.

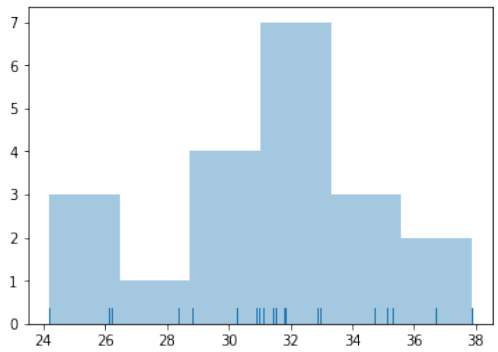

Figure 3:Agents Avg Score in the final episode (Solved)

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 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