Solving Tennis Environment using Multi Agent DDPG

The algorithm I use is largely based on the DDPG algorithm I used in the continuous control project, which is based on 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

Multiple Agent Deep Deterministic Policy Gradient (MADDPG)

In order to solve the multiple agent problem given by the project, a variation of DDPG called multiple agent DDPG is used for this project. My implementation of MADDPG is largely based on the DDPG code I used to solve the "continuous control" project.

A brief overview of the DDPG algorithm can be found from my "continuous control" project write-up: https://github.com/JoeZhao84/drl/blob/master/drlnd/project-continuous_control/DDPG-report.pdf

Details of the DDPG algorithm can be found from the original DDPG paper: https://arxiv.org/abs/1509.02971

The main design and changes I have done to adapt the DDPG code for MADDPG implementation are:

- There are two agents in the environment. They are sharing the same policy network (Actor) and the value network (Critic), therefore they will have the same target policy and act based on that. Below is the related code change:

```
with torch.no_grad():
     for agent_i, state in enumerate(states):
         action = self.actor_local(state).cpu().data.numpy()
         actions[agent_i] = action
```

- However when the 2 agents act, random noise is added independently to the 2 actions agent are making

- Other adaption is that in my DDPG implementation, the networks are updated 10 times every other 10 timesteps. There seems no strong reason to keep that setting, though it's an option for the later tuning.

Below is the pseudo code for the DDPG algorithm my implementation is based on:

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

Model Architecture

I have largely followed the DDPG model architecture used in the continuous control project. It 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. And I also change the number of neurons to 256 and 128. (This is from the early parameter tweaking attempt, may not be actually important)

Hyperparameters

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

BUFFER_SIZE = int(1e6) # replay buffer size

BATCH_SIZE = 64 # minibatch size

GAMMA = 0.99 # discount factor

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

LR_ACTOR = 2e-4 # learning rate of the actor

LR_CRITIC = 1e-4 # learning rate of the critic changed from 1e-3

WEIGHT_DECAY = 0 # L2 weight decay

Things have tried

So I started with the model structure and the hyperparameters I used for solving the continuous control problem. The time it took to finish one episode was much faster than the continuous control, while it did easily take many episodes (1,000) before you see any improvement of the reward score.

However the most important part for the code adaption was to allow the noise function to generate negative values, instead of only exploring only one direction. I saw this suggestion in the project group chat and got the environment solved immediately after the implementation. The code is like the below:

```
#dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in ra
nge(len(x))])
dx = self.theta * (self.mu - x) + self.sigma * np.random.standard_normal(self.size) #
adapt to allow for negative values to be returned
```

Some other stuff I have tried did really leading to solving the environment, but also contributes to (I think) the final solution. It includes gradient clipping, which can improve the model stability, as the project hints suggests the reward can be very noisy, even the learning has progressed well. The gradient clipping code is as below:

```
self.critic_optimizer.zero_grad()
critic_loss.backward()
torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1)
self.critic_optimizer.step()
```

I have also tweaked the buffer size and learning rate. A smaller buffer size did not work (it may if I tried it with the crucial noise adaptation earlier) and the original learning rate was not working either (1e-3 learning rate, agent almost not learning at all, or stuck at the average reward around 0.1).

And the learning is quite sensitive to the random seed. In the notebook I have shown that, with everything else the same, for some random seeds, it can be solved by less than 2000 episode and for some others, it's not solved even after 10000 episodes. Below is one of the cases that it reaches the solution around 2000 episodes.

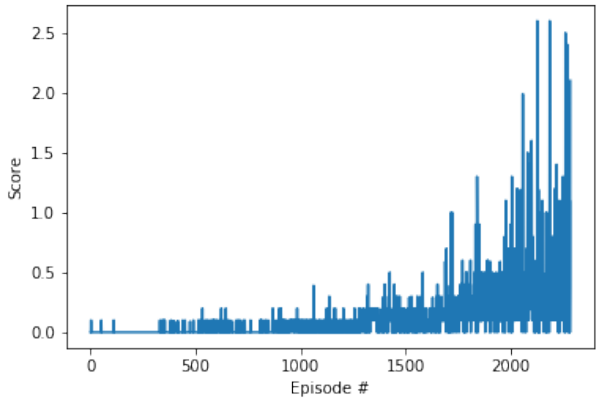

Figure: Average Training Reward

Future Improvement

As suggested by the Unity repo, the benchmark mean reward for the Tennis environment is 2.5. I will definitely try to achieve that under the current model architecture. Right now, I have not thought of any other models to solve it (by reaching 2.5 reward score) or using less episodes, but I'm open to suggestions.

Reference:

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