Continuous Control with DDPG

In order to solve the continuous control project[1], we have trained a DDPG (Deep Deterministic Policy Gradient) model using Pytorch. In this report, we will elaborate the learning algorithm, training results and the potential for future work.

I. Learning Algorithm

1. DDPG

DDPG is a model free, off-policy actor-critic algorithm that can operate over high-dimensional, continuous action spaces (Lillicrap, Timothy P., et al, 2015). In DDPG, we use two deep neutral networks, a local and a target neural network for actor and critic respectively. The local network is the most up to date network because it is the one that is trained, while the target network is the one used for prediction to stabilize training. The actor is used to approximate the optimal policy deterministically and the critic learns to evaluate the optimal action value function by using the actors best believed action.

The parameterized actor function $\mu(s|\theta^{\mu})$ specifies the current policy by deterministically mapping states to a specific action. The critic $Q(s,a)$ is learned using the Bellman equation.

The actor weights θ^{μ} are updated as follows,

$$\Delta_{\theta^{\mu}} J = -\mathbb{E}[\Delta_{\theta^{\mu}} Q(s,a|\theta^{Q})|_{s=s_t, a=\mu(s_t|\theta^{\mu})}]$$

$$\theta^{\mu} = \theta^{\mu} + \alpha \Delta_{\theta^{\mu}} J$$

where $J = -\mathbb{E}[Q(s,a|\theta^{Q})|_{s=s_t, a=\mu(s_t|\theta^{\mu})}]$ is the loss function and α is the learning rate.

In other words, the action predicted by the actor should be able to maximize the action value function estimated by the critic. Therefore, we use the local critic neural network to evaluate and update the weights of the local actor neural network.

The critic weights θ^{Q} are updated as follows,

$$\Delta_{\theta^{Q}} L = -\mathbb{E}\{[Q(s_t, a_t|\theta^{Q}) - (r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^{Q})]^2\}$$

$$\theta^{Q} = \theta^{Q} + \beta \Delta_{\theta^{Q}} L$$

where $Q(s_t, a_t|\theta^{Q})$ represents the action value for s_t estimated by the local critic neural network with the action a_t outputted by the local actor neural network; And $r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^{Q})$ is the action value for s_t estimated using temporal difference method by the target critic neural network with $\mu(s_{t+1})$ outputted by the target actor neural network. γ is the discount rate and β is the learning rate of the critical network.

[1] The introduction of the project can be found in the README.md file.

In other words, the local critic neural network should learn to converge to the target gradually. In the meantime, the iterative leaning process will push the local actor network to converge to the target actor network.

Additionally, DDPG employs experience replay to disentangle the correlation of consecutive actions. In our case, since we have multiple agents, each agent adds its experience to the replay buffer that is shared by all agents. Afterwards, (N, M) experience tuples (s_t, a_t, r_t, s_{t+1}) are sampled from the replay buffer for training, where N is the minibatch size and M is the number of agents.

Furthermore, a soft update strategy is implemented in DDPG to stabilize the algorithm. This strategy can be described using the following equation:

$$\theta' \longleftarrow \tau\theta + (1-\tau)\theta'$$

where θ' are the target network weights, θ are the local network weights and τ is the update rate.

Intuitively, the soft update strategy slowly blends the local network weights with the target network weights.

2. *Neural Network*

   As shown in Fig1, we have built a 3-layer plain neural network with parallel processing using Pytorch for the actor and the critic respectively[2]. The structure of the local and target network are the same, since the target network is only a copy of the local network at the beginning.

```
Actor network:
 DataParallel(
  (module): Actor(
    (fc1): Linear(in_features=33, out_features=256, bias=True)
    (fc2): Linear(in_features=256, out_features=256, bias=True)
    (fc3): Linear(in_features=256, out_features=4, bias=True)
  )
)

Critic network:
 DataParallel(
  (module): Critic(
    (fcs1): Linear(in_features=33, out_features=256, bias=True)
    (fc2): Linear(in_features=260, out_features=256, bias=True)
    (fc3): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

Fig1 Neural network structures of the actor and the critic

For the actor network, there are 256 units (or nodes) in each of the two hidden layers. We have implemented linear transformations followed by Relu activation functions in the hidden layers. The output layer contains 4 units, corresponding to the action size. Instead of Relu, we

[2] Note that the input layer is not counted. The input size is the state size, which is (33,20) in our case.

employed tanh function after the linear transformation in the output layer, since the action values must be within (-1, 1).

For the critic network, there are also 256 units in each of the two hidden layers. Similar to the actor network, linear transformation and Relu activation function are utilized in the hidden layers. What is worth mentioning is that the input size of the second layer is not equal to the output size of the first layer. This is because we concatenate the output of the first layer together with the action in order to get the state action value at the end. The output layer contains only 1 unit, corresponding to the state action value. And we have used sigmoid activation function after the linear transformation in the output layer, because the Q-value should be non-negative in our project.

Additionally, the weights in the hidden layers are initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$, where f is the fan-in of the layer. The weights of the output layers are initialized uniformly from $[-0.003, 0.003]$.

II. Empirical Results and Technical Details

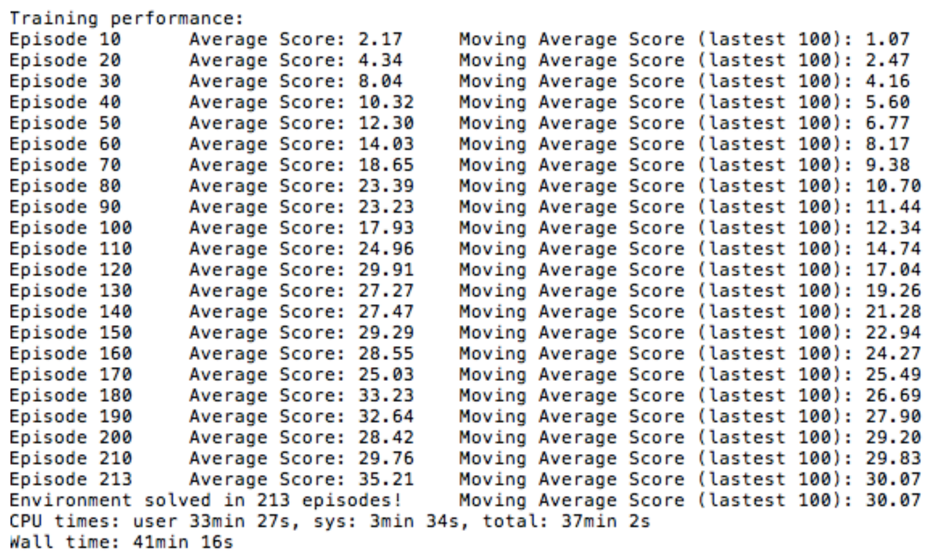
We have solved the project[3] with DDPG and the following set of hyperparameters within 213 episodes as shown in Fig2.

Hyperparameters:

```
'BATCH_SIZE': 128             # minibatch size
'BUFFER_SIZE': 100000         # replay buffer size
'GAMMA': 0.99                 # discount factor
'LR_ACTOR': 0.001             # learning rate of the actor
'LR_CRITIC': 0.001            # learning rate of the critic
'NOISE_DECAY': 0.999          # decay rate of the noise
'TAU': 0.01                   # soft update rate of the target parameters
'UPDATE_EVERY': 5             # how often to update the network
'WEIGHT_DECAY': 1e-07         # L2 weight decay for critic network
```

In our case, to further stabilize the algorithm, we have conducted the soft update in 5 steps instead of every time step and decayed the Ornstein-Uhlenbeck noise gradually with a factor of 0.999.

[3] The criterion to solve the project is to achieve at least 30 average scores over 100 consecutive episodes and over all agents.



```
Training performance:
Episode 10      Average Score: 2.17     Moving Average Score (lastest 100): 1.07
Episode 20      Average Score: 4.34     Moving Average Score (lastest 100): 2.47
Episode 30      Average Score: 8.04     Moving Average Score (lastest 100): 4.16
Episode 40      Average Score: 10.32    Moving Average Score (lastest 100): 5.60
Episode 50      Average Score: 12.30    Moving Average Score (lastest 100): 6.77
Episode 60      Average Score: 14.03    Moving Average Score (lastest 100): 8.17
Episode 70      Average Score: 18.65    Moving Average Score (lastest 100): 9.38
Episode 80      Average Score: 23.39    Moving Average Score (lastest 100): 10.70
Episode 90      Average Score: 23.23    Moving Average Score (lastest 100): 11.44
Episode 100     Average Score: 17.93    Moving Average Score (lastest 100): 12.34
Episode 110     Average Score: 24.96    Moving Average Score (lastest 100): 14.74
Episode 120     Average Score: 29.91    Moving Average Score (lastest 100): 17.04
Episode 130     Average Score: 27.27    Moving Average Score (lastest 100): 19.26
Episode 140     Average Score: 27.47    Moving Average Score (lastest 100): 21.28
Episode 150     Average Score: 29.29    Moving Average Score (lastest 100): 22.94
Episode 160     Average Score: 28.55    Moving Average Score (lastest 100): 24.27
Episode 170     Average Score: 25.03    Moving Average Score (lastest 100): 25.49
Episode 180     Average Score: 33.23    Moving Average Score (lastest 100): 26.69
Episode 190     Average Score: 32.64    Moving Average Score (lastest 100): 27.90
Episode 200     Average Score: 28.42    Moving Average Score (lastest 100): 29.20
Episode 210     Average Score: 29.76    Moving Average Score (lastest 100): 29.83
Episode 213     Average Score: 35.21    Moving Average Score (lastest 100): 30.07
Environment solved in 213 episodes!     Moving Average Score (lastest 100): 30.07
CPU times: user 33min 27s, sys: 3min 34s, total: 37min 2s
Wall time: 41min 16s
```

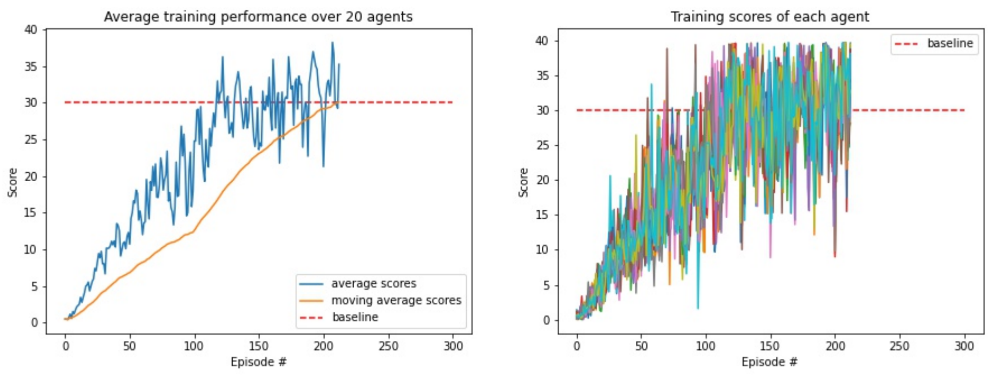


Fig2 Training performance

With 2 GPUs and multiprocessing in Pytorch, we have solved the project within 41 minutes. For users who only use CPU or one GPU, the processing time could be longer. Please note that the average score shown in Fig2 represents the average score over 20 agents per episode and the moving average score stands for the average of the average score over different episodes (max. 100 consecutive episodes). We can observe from the subplot on the right hand side, which shows the training scores for each agent, that the agents share similar learning patterns.

III. Outlook

Proximal Policy Optimization (PPO), Asynchronous Advantage Actor-Critic (A3C) and Deep Distributional Deterministic Policy Gradients(D4PG) are also effective methods that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience. It could be interesting to compare the performance of the three methods with DDPG. Due to the scope of current work, we would like to left it for future research.

Reference

Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971* (2015).