DRLND-P2-Report

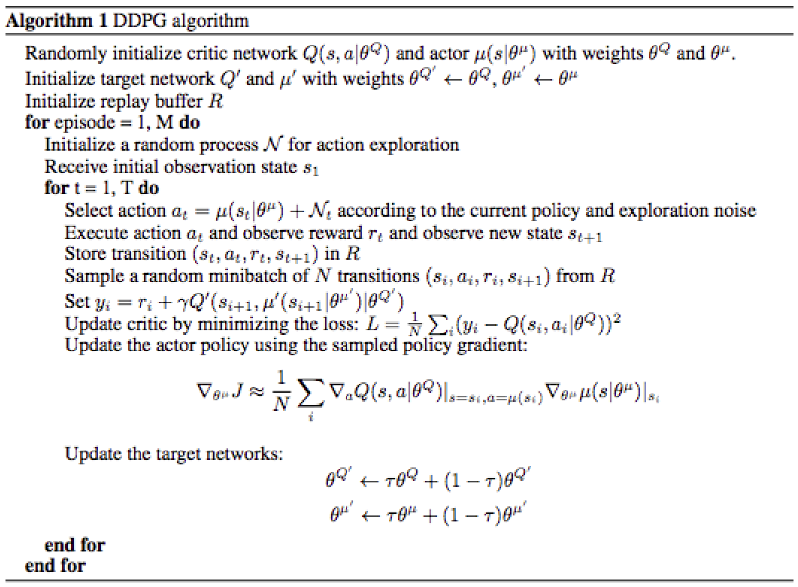
Ali Alizadeh

1 Introduction

In this report, I summarize my final modeling decisions as to succeed in solving the Reacher environment. My approach in this project is to use the Deep Deterministic Policy Gradients (DDPG) algorithm [1] modified to be applicable for distributed training.

2 Description of Learning Algorithm

DDPG algorithm uses actor-critic framework with replay buffer as described in the following pseudo-code [1].

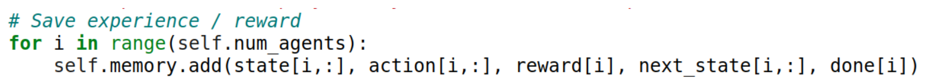


```
Algorithm 1 DDPG algorithm
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay 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
```

My implementation follows the standard DDPG algorithm described above with some modifications to make it multi-agent and suitable for distributed training.

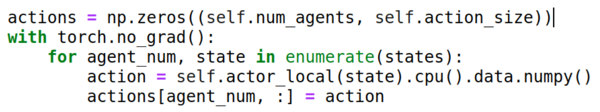
2.1 Multi-agent DDPG

The following modifications are done over the standard DDPG to make it appropriate for distributed training (multi-agent DDPG). Distributed training for multi-agent DDPG is done through using 20 identical agents that act separately in an identical environment. Experience replay buffer in multi-agent DDPG setup for distributed training is fed with large number of agents as defined by the number of agents and shown below.



```
# Save experience / reward
for i in range(self.num_agents):
    self.memory.add(state[i,:], action[i,:], reward[i], next_state[i,:], done[i])
```

Similarly as stated above, each identical agent has its specific interaction with the identical environment as shown below¿



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

3 Hyper-parameters optimization

The following hyper-parameters are being used in my implementation:

- Batch size of 128
- Experience replay buffer size of 100000 randomly samples of actions
- Discount factor of 0.99
- Actor's learning rate of 0.0001
- Critic's learning rate of 0.001
- Soft update parameter of target parameters of 0.001
- Weight decay of 0

3.1 Model Architecture

The actor-critic framework is used in this implementation, where:

- Actor network is a mapping from state to action values. Here I used three layers of fully connected layers with relu activation function. Since we have an action size of four, the output layer of the actor network yields four values with tanh activation function.
- Critic network is a value functions measuring the state-action values of the actions through three fully connected layers with relu activation function.

4 Results and Discussion

The training log of the multi-agent DDPG algorithm applied to 20-agent Reacher algorithm is shown in Fig. 1. I have terminated the learning process once the average score of 100 consecutive episodes reach the goal score of 30. As to give exact number I could achieve the required performance of +30 in 174 episodes which is acceptable.

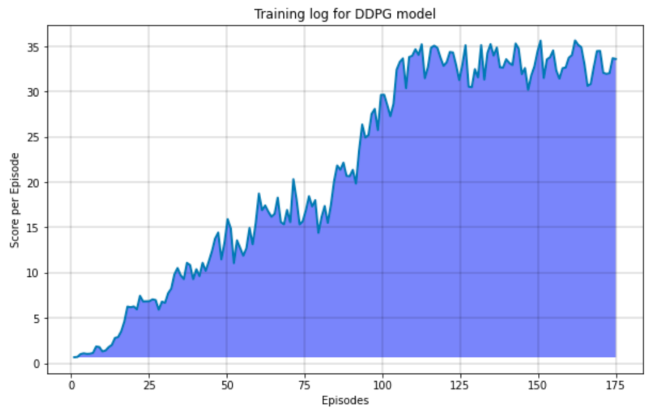


Figure 1: Training log of the agent's performance in Reacher environment

5 Future Work

Hyper-parameter optimization can boost the training progressy, increase the performance and stability of the trained agent, so I will consider it for a systematic and automated search of optimal hyper-parameters. Utilization of prioritized experience replay instead of the standard one would increase the performance of agent and make it more stable. As suggested by the project description, the current distributed training is useful for algorithms such as PPO [1], A3C [2], and D4PG [3] that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

References

[1] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[1]https://arxiv.org/pdf/1707.06347.pdf
[2]https://arxiv.org/pdf/1602.01783.pdf
[3]https://openreview.net/pdf?id=SyZipzbCb