Deep Reinforcement Learning- Multi-Agent Collaboration

Udacity- Project 3

Tennis

Introduction

In the report for the third project required for partial fulfillment of requirements pertaining to Udacity's Deep Reinforcement Learning Nanodegree, two Deep Deterministic Policy Gradient based Agents were implemented to solve Unity's Tennis Environment. The grading rubric calls for the report to delve into the Learning Algorithm, Reward Plots and Ideas for Future work. Details about implementation and the environment could be found in the 'README.md' file of the home repository.

Algorithms

Policy gradient methods are a type of reinforcement learning techniques that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. They do not suffer from many of the problems that have been marrying traditional reinforcement learning approaches such as the lack of guarantees of a value function, the intractability problem resulting from uncertain state information and the complexity arising from continuous states & actions.

The general goal of policy optimization in reinforcement learning is to optimize the policy parameters $\theta \in R^K$ so that the expected return,

$$J(\theta) = E\left\{\sum_{k=0}^{H} a_k r_k\right\}$$

is optimized where a_K denotes time-step dependent weighting factors, often set to $a_K = \gamma^K$ for discounted reinforcement learning (where γ is in [0,1]) or $a_K = 1/H$ for the average reward case. For real-world applications, we require that any change to the policy parameterization has to be smooth as drastic changes can be hazardous for the actor as well as useful initializations of the policy based on domain knowledge would otherwise vanish after a single update step. For these reasons, policy gradient methods which follow the steepest descent on the expected return are

the method of choice. These methods update the policy parameterization according to the gradient update rule,

$$\theta_{h+1} = \theta_h + \alpha_h \nabla_\theta J|_{\theta=\theta_h},$$

Where $a^h \in \mathbb{R}^+$ denotes a learning rate and h ∈ {0,1,2,.....} the current update number.

Deep Deterministic Policy Gradient (DDPG)

DDPG uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network. The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, the Actor directly maps states to actions (the output of the network directly the output) instead of outputting the probability distribution across a discrete action space.The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning.

The Pseudo-Code of the algorithm is,

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

Taken from "Continuous Control With Deep Reinforcement Learning" (Lillicrap et al, 2015)

Multi Agent Deep Deterministic Policy Gradient (MADDPG)

MADDPG is an extension of DDPG. It utilizes multiple DDPG agents to solve cooperative, competitive or mixed environments. Similar to single-agent Actor Critic architecture, each agent has its own actor and critic network. The Actor network takes in the current state of agent and output a recommended action for that agent. The Critic part is slightly different from ordinary single-agent DDPG. The Critic network of each agent has *full visibility* on the environment. It takes in the observation and action of that particular agent, and also observations and actions of *all other* agents as well.

Implementation Details

The Actor Network has two hidden layers and an output layer. ReLu activation is used after the first and the second hidden layer. Tanh activation is used with the output layer. The Critic Network also has two hidden layers and an output layer. Similar to the Actor Network, after the and the second hidden layers, ReLu activation is used. The initial input to the Critic Network are the states, actions are later added by concatenating the output from the first ReLu activation layer with the actions and passing the concatenated entity to the next hidden layer. No activation is applied after the output layer.

Hidden layers of size 400 and 300 were used for the Actor Network. Critic Network has hidden layers of size 256 and 128. These values were determined by manually tuning the hyperparameters. The following hyperparameters were used for the agents,

1. BUFFER_SIZE = 1000000
2. BATCH_SIZE = 512
3. GAMMA = 0.99
4. TAU = 1e-3
5. LR_ACTOR = 1e-3
6. LR_CRITIC = 1e-3
7. WEIGHT_DECAY = 0
8. LEARN_EVERY = 20
9. LEARN_NUM = 10
10. OU_SIGMA = 0.2
11. OU_THETA = 0.15
12. EPSILON = 1.0
13. EPSILON_DECAY = 1e-6

Other hyperparameters pertaining to the training process could be found in the ‘Tennis.ipynb’ notebook.

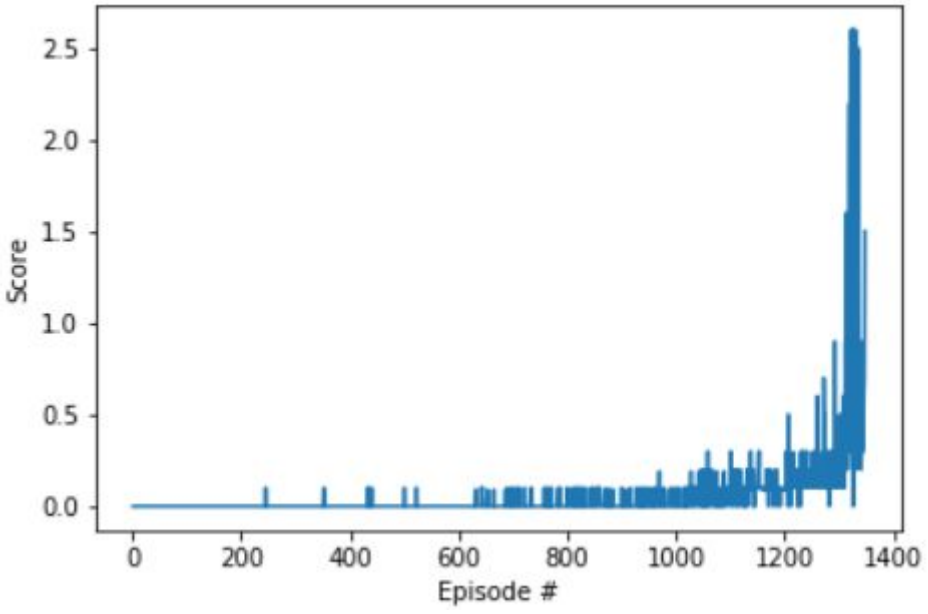
Number of Episodes

The required (by Udacity) threshold of +0.5 for the environment to be considered solved was attained in 1348 episodes.

```
Episode 100     Average Score: 0.00 best_score 0.0
Episode 200     Average Score: 0.00 best_score 0.0
Episode 300     Average Score: 0.00 best_score 0.10000000149011612
Episode 400     Average Score: 0.00 best_score 0.10000000149011612
Episode 500     Average Score: 0.00 best_score 0.10000000149011612
Episode 600     Average Score: 0.00 best_score 0.10000000149011612
Episode 700     Average Score: 0.01 best_score 0.10000000149011612
Episode 800     Average Score: 0.01 best_score 0.10000000149011612
Episode 900     Average Score: 0.02 best_score 0.10000000149011612
Episode 1000    Average Score: 0.02 best_score 0.19000000320374966
Episode 1100    Average Score: 0.07 best_score 0.30000000447034836
Episode 1200    Average Score: 0.10 best_score 0.30000000447034836
Episode 1300    Average Score: 0.19 best_score 0.90000001341104516
Episode 1348    Average Score: 0.51 best_score 2.6000000387430196
Environment solved in 1348 episodes!    Average Score: 0.51
```

Reward Plots

The reward from the training process for a window size of 100 episode is given by the following plot,



Ideas for Future work

1. Prioritization for replay buffer,
2. try Variations of DDPG agents,
3. hyper parameter optimization,
4. experiment with different network architecture and activation layers.