PROJECT 1: POLICY BASED

DEEP REINFORCEMENT LEARNING

CLEMENTE FORTUNA

INTRODUCTION

The solution implements a deep deterministic policy gradient (DDPG) algorithm. Focus areas include setting the step/learning scheme for multiple agents, implementing batch normalization, gradient clipping, network simplification, separating networks from the computation graph, and tuning hyperparameters. The result was solution convergence in 114 episodes.

ALGORITHM OVERVIEW

The figure below captures the basic 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

Action exploration includes noise generation in the selection method; this is in the **OUNoise** class and in **Agent.learn()** line 68. The critic loss minimization was implemented as **pytorch.nn.functional.mse_loss()** in line 99 of **Agent.learn()**. The actor gradient maximization was implemented by computing the negative of the loss, in line 112 of **Agent.learn()**. Soft updates operate on the target networks, in of **Agent.learn()** lines 119-120, and **Agent.soft_update()**.

STEPPING AND LEARNING SCHEDULE

The provided benchmark implementation described 10 updates every 20 time steps. For a single agent this is straightforward. For 20 agents this can have drastically different interpretations. A common approach was at each time step to add all 20 experiences, and to compute 10 updates for each experience addition, if the time step was divisible by 20 and there were sufficient experiences. The pseudocode below captures this.

```
for t in range(max_t):
        act according to agent
        get next_states, rewards, dones
        for i in range(num_agents):
                step()

def step():
        add a single experience
        if multiple of 20 time steps and sufficient experiences:
                do 10 updates
```

This results in 200 updates and 400 additions every 20 time steps. Contrast this with the following pseudocode.

```
for t in range(max_t):
        act according to agent
        get next_states, rewards, dones
        step()

def step():
        add all 20 experiences
        if multiple of 20 time steps and sufficient experiences:
                do 10 updates
```

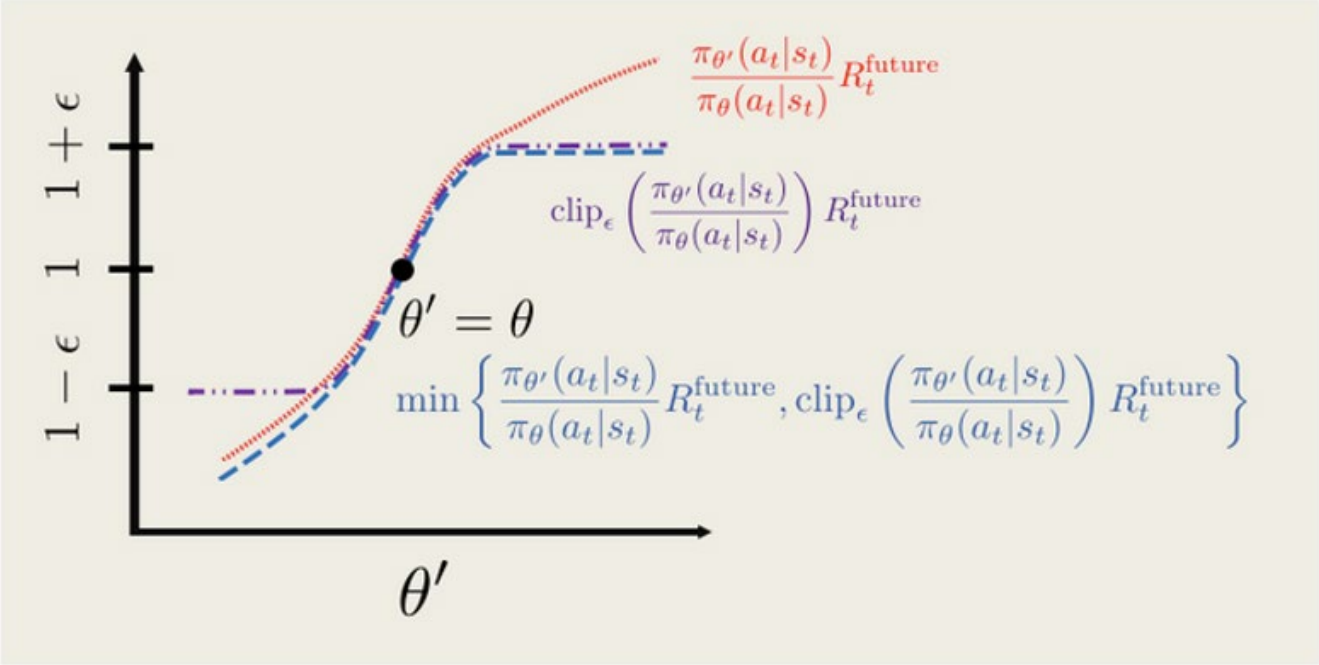
This results in 10 updates and 400 additions every 20 time steps. This is in **ddpg()** line 19, and **Agent.step()** lines 46-57. The result was solution convergence an order of magnitude faster.

BATCH NORMALIZATION

Page 4 of the DDPG paper describes the use of batch normalization to enable networks to learn when inputs are characteristically different (i.e. different units of measurement). We implemented **nn.BatchNorm1d()** for both the Actor and Critic, between the input layer and the first RELU activation, although some have argued for implementation after the activation function. This is in **Actor.__init__()** line 24, **Actor.forward()** line 39, **Critic.__init_()** line 62, and **Critic.forward()** line 77.

GRADIENT CLIPPING

The figure below depicts gradient clipping, a process for mitigating failure of gradient ascent:



Since we apply gradient ascent on the Critic local network, we apply gradient clipping there. This is in **Agent.learn()** line 104.

NETWORK SHRINKING/SIMPLIFICATION

Reducing network size and complexity reduces computation load. The Actor and Critic fully connected layers were reduced to 128 units. A multiple two was selected to leverage GPU architecture. This is in the **__init__** methods for both the Actor and Critic (lines 10 and 48).

COMPUTATION GRAPH MANAGEMENT

In the DDPG algorithm, as in the DQN, we compute gradients for weight determination for the local network only. The target network is modified only by hard or soft updates. Hence in **Pytorch** both the Actor Target and Critic Target networks should be detached in the learning step. This is in **Agent.learn()** lines 92-93.

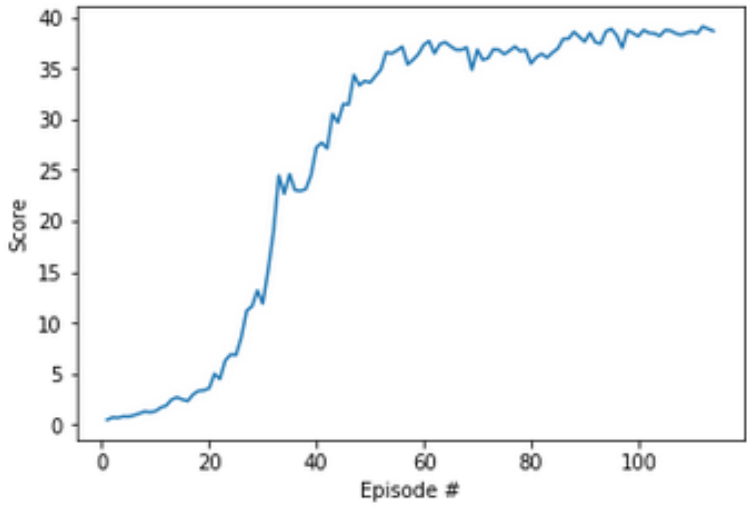
HYPERPARAMETER TUNING AND SELECTION

Hyperparameter selection meant the difference between solution convergence and divergence. The buffer size was *20 * 1e5*, multiplying the size from the DQN project by the number of agents. The batch size was increased to *1024*, to move as much data onto the GPU as possible in multiples of two. Gamma was kept high, since 1000 time steps seemed to allow for optimal learning. A mean of the Actor and Critic learning weights from the DDPG paper was selected, based solely on performance observation. The step/update scheme, discussed previously, was adopted from the provided benchmark implementation. Lastly, a less "noisy" noise process meant more conservative action exploration, which yielded better results; we reduced the noise standard deviation to 0.1 to achieve this.

RESULT AND CONCLUSION

This implementation first had an episode score past 30 at episode 43, and reached a cumulative average above 30 over 100 episodes at episode 114:

```
Environment solved in 114 episodes!  Cumulative100 Average Score: 30.303
```




```
finish time: 7764.258405447006
```

The weights are included as **checkpoint_actor.pth** and **checkpoint_critic.pth**. Performance could be improved by further tuning hyperparameters and experimenting with other network structures. More ambitiously, leveraging **torch.multiprocessing** across the 20 arms or across each update could reduce runtime, thereby allowing room to improve performance. This library may be important for implementing D4PG, for distributional learning.