Project 3: Collaboration and Competition

Kaushik Balakrishnan

November 18, 2019

1 Problem Statement

For the third project, we will attempt to solve the Tennis environment in Unity, a schematic of the environment is shown in Figure 1. In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket, for a total of 24 variables. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. The task is episodic, and in order to solve the environment, the agents must get an average score of +0.5 over 100 consecutive episodes, after taking the maximum over both agents. Specifically, after each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 potentially different scores, and we take the maximum of these 2 scores. This yields a single score for each episode. The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5. We however push the limit even higher and set a goal of +1.0 as the required mean episodic score over 100 episodes.

2 DDPG

For the problem of Tennis, we use the Deep Deterministic Policy Gradient (DDPG) algorithm [1], and this report will summarize the algorithm as well

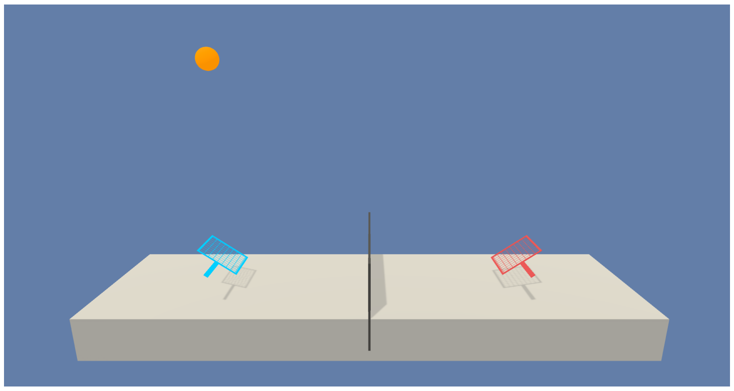

Figure 1: Tennis environment in Unity.

as the different aspects of it. However, the problem we are trying to solve here is a multi-agent RL problem since both agents must collaborate to maximize the score. We thus use a multi-agent DDPG (MADDPG) [2] version of the algorithm where the agents share the policy and training replay buffer.

Reinforcement Learning (RL) is a Markov Decision Process (MDP) [3] where the agent takes an action depending on the immediate state it is in, and does not depend on the past states. RL involves a tradeoff between exploration and exploitation, where exploration is to take random actions to understand the surroundings better, and exploitation is to take a greedy action based on policy or value. In particular, the use of a neural network in RL as a function approximator is called Deep Reinforcement Learning (DRL). Furthermore, a replay buffer is also used to reuse past experiences. Note that since past experiences are used in the training, the algorithm is off-policy.

2.1 Actor-Critic

Actor-Critic algorithms are very popular and enjoy a prestigious history in the annals of Reinforcement Learning literature. Actor-Critic algorithms involve two eponymous networks: an Actor network is trained to learn a policy, i.e., to output an action at a given state. The Critic network is used to evaluate the Actor's performance and output a value function, which is used to train the Actor as a learning signal. The Critic is trained using the Bellman equation to minimize the temporal difference (TD) error [1].

The Actor network consists of fully connected layers with two hidden layers of 128 neurons each, and one output layer that outputs 2 actions. The Relu activation function is used for the hidden layers, and Tanh for the output layer to enforce the actions to be constrained in the [-1, 1] range. In addition, Batchnormalization is used for stability reasons. The Critic network also consists of two hidden layers with 128 neurons each, and an output layer with a single output. Relu activations are used for the hidden layers, and no activation is used for the output layer as the value Q(s,a) output of the critic is not bounded. Batchnormalization is used in the Critic network as well for stability.

The Actor network is updated using the policy gradient, i.e., to maximize the expectation of Q(s,a):

$$\nabla_{\theta^a} J = E\left[\nabla_{\theta^a} Q(s,a)|_{s=s_t, a=\mu(s_t)}\right], \tag{1}$$

where θ^a are the actor's parameters, J is the function that is being maximized, and $\mu(s_t)$ is the deterministic policy of the Actor. The target state-action value function for the Critic network is obtained by using the Bellman equation, with the subsequent state-action value evaluated by bootstrapping:

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})), \tag{2}$$

where r is the immediate reward received and γ is the discount factor. The Critic is updated by minimizing the L2 error between y_t and $Q(s_t, a_t)$:

$$argmin \;\; E[(y_t - Q(s_t, a_t))^2]. \tag{3}$$

The networks are trained using the Adam optimizer [4].

2.2 Target network

Target networks are used for both the Actor and the Critic to evaluate, as this is more stable. Without the target network, we are forced to update the networks by using them concomittantly to also evaluate them, which is not stable. The target networks are updated using a soft update approach, i.e., using a variable $\tau = 10^{-3}$ and performing a weighted update with weights τ and 1-τ on the current network value and the target network value, respectively. More details can be found in the DDPG paper [1].

2.3 Replay buffer

DDPG is an off-policy algorithm and requires i.i.d. samples to train the Actor and Critic networks, and so to avoid correlated samples, a replay buffer is used where past samples are stored and a mini-batch of experiences is sampled from it at every update. In the current multi-agent setting, both actors share the same replay buffer. Note that by experience, we refer to the tuple: (state, action, reward, new state, done). For this project, a standard replay buffer is considered, although a prioritized experience replay buffer [5] can also be considered in the future as a possible extension. In the standard replay buffer, all the samples have equal probability of being sampled, unlike its prioritized counterpart for which the samples are sampled with a probability that is proportional to its temporal difference (TD) error.

2.4 Ornstein-Uhlenbeck noise

For exploration, we add noise to the Actor's action predictions; specifically, we add Ornstein-Uhlenbeck (OU) noise to the actions. The Ornstein-Uhlenbeck (OU) noise is a stochastic process and is a biased Gaussian noise. It has numerous applications in Physics, Mathematics, and Finance. In DDPG, the OU noise is sometimes preferred over Gaussian noise, as the latter has equal probability of drawing samples on either side of the mean, whereas the OU noise is biased. See Ref. [6] for more details.

2.5 Reload checkpoint files

We found the environment to be highly intermittent, and so the agents were quickly forgetting what they learned. To overcome this problem, we save the recent best performing neural network weights, and reload them whenever performance drops below a threshold, which is also performance-dependent. Specifically, we set a threshold, T = 0.05 at the beginning of the training. Once the agents attain a 100-episode moving average episodic reward greater than this threshold T, the neural network weights are backed-up and T is incremented by 0.05. Subsequently, if the 100-episode moving average episodic reward ever falls below T-0.05, the most recently saved back-up checkpoint files are read from disk and the training continues from there on. Thus, the agent has no way of encountering catastrophic forgetting, as the recent best back-up checkpoint files are restored. This greatly accelerated the overall training.

2.6 Setting up the environment

Download the six Python files from this project:

1. AC.py This file contains the Actor and Critic classes, with their respective neural network architectures
2. ddpg.py This file contains the Agent class, which comprises of the Actor and Critic networks, as well as their respective target networks. In addition, this class consists of the replay buffer and the functions step(), act(), update_networks() and train_model().
3. buffer.py This file has the standard replay buffer class, which stores and samples past experiences
4. noise.py This file has the OUNoise class, which computes the Ornstein-Uhlenbeck noise
5. train.py This file has the train_agent() function, which runs Nepisodes number of episodes to train the agent
6. test.py Once the agent is trained fully, this file is used to test the agent and for this the test_agent() function is called

Copy the Tennis_Linux folder from the Udacity website to this directory. Set the path to the Tennis.x86_64 file in the env declaration in train.py:

```
env = UnityEnvironment(file_name='Tennis_Linux/Tennis.x86_64')
```

Install MLAgents using pip, and also download unityagents and communicator_objects from the Udacity project website, creating symbolic links to them like so:

```
pip install mlagents
ln -s <path to unityagents> .
ln -s <path to communicator_objects> .
```

description	name of hyperparameter	value
discount factor (γ)	GAMMA	0.99
max number of episodes	Nepisodes	5000
batch size	BATCH_SIZE	256
buffer capacity	BUFFER_SIZE	1e6
learning rate for Actor	LR_A	2e-4
learning rate for Critic	LR_C	1e-3
target net update parameter (τ)	TAU	1e-3

Table 1: Table of hyperparameters

2.7 Hyperparameters

The hyperparameters used are summarized in Table 1.

2.8 Start the training

To start the training, run the command:

```
python train.py
```

2.9 Saving the model weights

The final trained Actor and Critic weights are saved in ckpt/ folder and are PyTorch files.

3 Results

The Tennis problem is considered solved when the average episodic reward for the last 100 episodes is 0.5 according to the Udacity project requirement, which we have set to 1.0 in order to aim higher. In Figure 2, we present the episodic rewards as a function of the episode count. As evident, the agent struggles to learn for many of the early episodes, but later the learning picks up rapidly, and thereafter the agent achieves the 100-episode moving average episodic reward of +1.0 in 1674 episodes.

Once the agents are trained, we also test the agents by using the command:

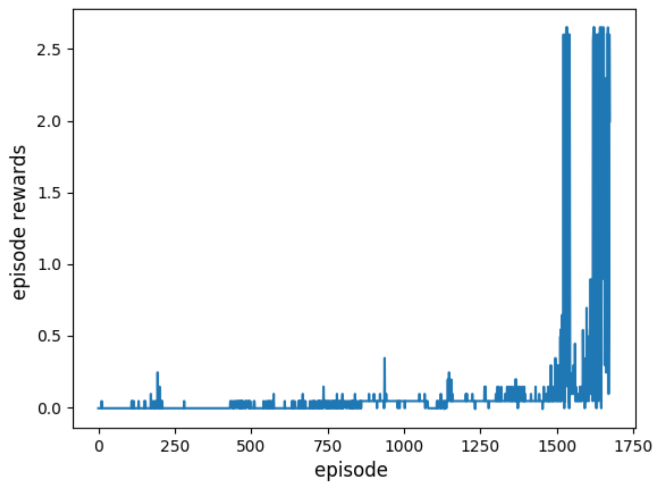

Figure 2: Episodic rewards as a function of episode number.

```
python test.py
```

Specifically, we test the agents for 10 episodes and the episodic rewards at test time are summarized below. The agent is able to successfully maintain an episodic reward of ~ 2.6 over 10 different episodes.

```
episode:  0  | episode reward:  2.600000038743019
episode:  1  | episode reward:  2.600000038743019
episode:  2  | episode reward:  2.600000038743019
episode:  3  | episode reward:  2.650000039488077
episode:  4  | episode reward:  2.600000038743019
episode:  5  | episode reward:  2.650000039488077
episode:  6  | episode reward:  2.600000038743019
episode:  7  | episode reward:  2.600000038743019
episode:  8  | episode reward:  2.650000039488077
episode:  9  | episode reward:  2.600000038743019
```

3.1 Future directions

We can consider PPO [7] for the same problem and evaluate the performance, and compare it to the DDPG performance from this report. We can also change the network architecure to ascertain if that has any effect on the final learning.

4 References

1. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D. (2015). Continuous Control With Deep Reinforcement Learning, arXiv: https://arxiv.org/abs/1509.02971

2. Lowe, R., Wu, Yi., Tamar, A., Harb, J., Abbeel, P. and Mordatch, I. (2017). Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments, arXiv: https://arxiv.org/abs/1706.02275

3. Sutton, R. S. and Barto, A. G. (2018). Reinforcement Learning: An Introduction, Second Edition, MIT Press, Cambridge, MA

4. Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization, arXiv: https://arxiv.org/abs/1412.6980

5. Schaul, T., Quan, J., Antonoglou, I. and Silver, D. (2015). Prioritized Experience Replay, arXiv: https://arxiv.org/abs/1511.05952

6. Ornstein-Uhlenbeck Process, https://en.wikipedia.org/wiki/Ornstein

7. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. (2017). Proximal Policy Optimization Algorithms, arXiv: https://arxiv.org/abs/1707.06347