Project 2: Continuous Control

Kaushik Balakrishnan

November 15, 2019

1 Problem Statement

For the second project, we will attempt to solve the Reacher environment in Unity, a schematic of the environment is shown in Figure 1. Note that this environment comes in two versions: either one agent or 20 such parallel agents; the latter is preferred as one can collect 20 times as many data in the same wallclock time, and so can train reinforcement learning agents much faster. In this environment, a double-jointed arm must move and reach out to a prescribed target location in its vicinity. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. The goal of the agent is to maintain its position at the target location for as many time steps as possible so as to accrue high rewards.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. The action vector is continuous and consists of four real values between -1 and 1.

2 DDPG

For the problem of Reacher, we use the Deep Deterministic Policy Gradient (DDPG) algorithm [1], and this report will summarize the algorithm as well as the different aspects of it. Reinforcement Learning (RL) is a Markov Decision Process (MDP) [2] 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 exporation is to take random actions to understand the surroundings better, and exploitation is to

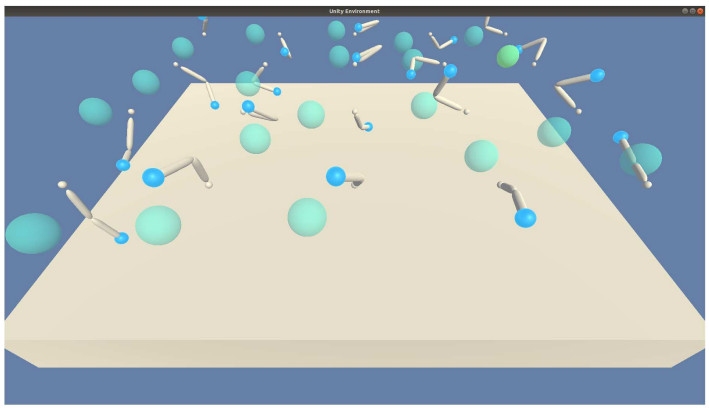

Figure 1: Reacher environment in Unity.

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. In this study we will explore two different kinds of replay buffers, more on this soon.

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 4 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 Critic network is updated by using the Bellman equation with the subsequent state-action value evaluated by bootstrapping, and minimizing the TD error:

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

where r is the immediate reward received and γ is the discount factor.

The networks are trained using the Adam optimizer [3].

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

DDPD 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. Note that by experience, we refer to the tuple: (state, action, reward, new state, done). We consider two replay buffers for this project: (i) a standard replay buffer; and (ii) a prioritized experience replay buffer [4]. In the standard replay buffer, all the samples have equal probability of being sampled. This has the drawback that rarely occurring experiences are sampled with the same probability as frequent experiences, which can slow the learning for some problems. To circumvent this issue, a prioritized replay buffer [4] is also considered in this study, where experiences with high temporal difference (TD) error are sampled with a

higher probability, and so the hard-to-learn samples are used more often in the training. To achieve this, a sumtree data structure is used where the value of each parent node is equal to the values of the two children nodes. Since samples drawn from the prioritized replay buffer have different probabilities, importance sampling weights are also used to compute the expectation. Both replay buffers are of interest in this project and the learning with each of them are compared.

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. [7] for more details.

2.5 Setting up the environment

Download the seven 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 buffers and the functions step(), act(), update_networks(), train_model_s() and train_model_p(), the latter two are for the standard and prioritized replay buffers, respectively.

3. buffer.py This file has the standard and prioritized replay buffer classes, which store and sample past experiences

4. sumtree.py This file has the sumtree data structure required for the prioritized experience replay buffer

5. noise.py This file has the OUNoise class, which computes the Ornstein-Uhlenbeck noise

description	name of hyperparameter	value
discount factor (γ)	GAMMA	0.99
max number of episodes	Nepisodes	1500
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

6. train.py This file has the train_agent() function, which runs Nepisodes number of episodes to train the agent

7. 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 Reacher_Linux folder from the Udacity website to this directory. Set the path to the Reacher.x86_64 file in the env declaration in train.py:

```
env = UnityEnvironment(file_name='Reacher_Linux/Reacher.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> .
```

2.6 Hyperparameters

The hyperparameters used are summarized in Table 1.

2.7 Start the training

To start the training, run the command:

```
python train.py
```

2.8 Saving the model weights

The final trained Actor and Critic weights are saved in ckpt/standard or ckpt/prioritized, depending on the replay buffer type used, and are PyTorch files.

3 Results

As aforementioned we consider two cases with (i) standard and (ii) prioritized experience replay buffers. The Reacher picking problem is considered solved when the average episodic reward for the last 100 episodes is 30. In Figure 2, we present the episodic rewards for both the replay buffers used. As evident, the agent learns very quickly and the episodic rewards are over 30 in just 150 or so episodes, and they stay greater than 30 for the rest of time. Both replay buffers perform very similar, and so using a prioritized replay buffer does not provide any advantage vis-á-vis the standard replay buffer. The agents were also trained multiple times for repeatability, and the number of episodes required to learn, as well as the final performance, was quantitatively similar.

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

```
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 ~ 39 over 10 different 1000-time step episodes.

```
episode:  0  | episode reward:  39.127499125432294
episode:  1  | episode reward:  38.988999128528064
episode:  2  | episode reward:  39.277499122079504
episode:  3  | episode reward:  39.105499125924005
episode:  4  | episode reward:  39.129499125387646
episode:  5  | episode reward:  39.03849912742163
episode:  6  | episode reward:  39.10899912584583
episode:  7  | episode reward:  39.03399912752217
episode:  8  | episode reward:  39.205999123677664
episode:  9  | episode reward:  39.06749912677337
```

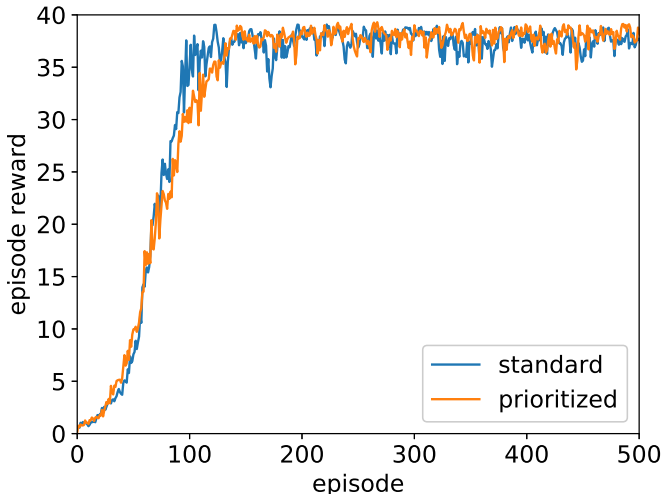

Figure 2: Episodic rewards with standard and prioritized replay buffers.

3.1 Future directions

We can consider PPO [5] and D4PG [6] 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. Sutton, R. S. and Barto, A. G. (2018). Reinforcement Learning: An Introduction, Second Edition, MIT Press, Cambridge, MA

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

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

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

6. Barth-Maron, G., et al. (2018), Distributed Distributional Deterministic Policy Gradients, arXiv: https://arxiv.org/pdf/1804.08617.pdf
7. Ornstein-Uhlenbeck Process, https://en.wikipedia.org/wiki/Ornstein