Multi-agent RL

Ali Hummos

1 Overview:

This the third project of the Udacity deep RL nanodegree and it aims to solve a version of the Unity Tennis environment, by achieving a score of ¿0.5 over 100 consecutive episodes, taking the max score achieved by either of the two agents.

The environment models a simple tennis table with two controllable rackets on either side. An agent is rewarded +0.1 if it hits the ball over the net, and is rewarded -0.01 if it lets the ball go out of bounds or drops it. We approach this environment as a multi-agent setup with one agent for each Tennis racket. Each agent receives as an observation an 8 dimensional vector (with the 3 most recent frames stacked). Each agent outputs 2 continuous action values corresponding to moving the racket towards the net, or jumping up.

The environment is episodic with a continuous state space and a continuous action space. It's a generally stationary environment, expect for the policy of the other agent. The action space is continuous, which constrain our choice of algorithm to solve it.

2 Implementation:

The code utilizes the DDPG algorithm [1], and builds on the code provided by the Udacity deep reinforcement learning library [1]. The algorithm can be considered an extension of the Deep Q-Leaning network [3] which enables it to deal with continuous action spaces by adding a policy network (an actor) to output action values, while the Q-network continues to learn an estimation of the value function. To solve this environment, I instantiated two DDPG agents building on the Udacity DDPG implementation, but adapted for multi-agent setting.

Several setups are possible:

1. Centralized learning, centralized acting: One DDPG model takes observations from the two agents, and controls the two agents.
2. Centralized learning, de-centralized acting: Two DDPG models take observations from both agents, but separately control their respective agent
3. De-centralized learning, de-centralized acting: Two DDPG models take only local observations from their agent, and separately control their respective agent
4. MADPG [2]: Two DDPG models, each takes observations from both agents, including the actions of the other agent, but separately control their respective agent.

As in DQN paper, DDPG uses a target and a local version of both policy and value networks, the local version is updated through SGD, while the target version gets soft updates throughout training.

DDPG uses a deterministic policy network, which means that it estimates the actions that would maximize the return or value as estimated by the value network. It directly outputs the action values it

[1] https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-bipedal

predicts would greedily exploit the current states. Which means that exploration has to be factored. In this implementation, we use an Ornstein-Uhlenbeck process to generate white noise (sigma=0.15, theta=0.2) and add it to the predict action vector to encourage exploration.

The DDPG algorithm is summarized in this pseudocode taken from the original paper.

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

Figure 1: DDPG - Pseudocode from original paper [1]

The policy network $\mu(s|\theta^\mu)$ is updated with the gradient of the expected return J with respect to the parameters θ^μ:

$$\nabla_\theta J = \frac{1}{N} \sum_i \nabla_\alpha Q(s, a;\ \theta^Q) \nabla_\theta \mu(s|\theta^\mu) \tag{1}$$

For MADDPG, the equation changes where the value network Q becomes also conditioned on the actions taken by the other agents as follows:

$$\nabla_\theta J = \frac{1}{N} \sum_i \sum_j \nabla_\alpha Q(s, a_i; a_j, \theta^Q) \nabla_\theta \mu(s|\theta^\mu) \tag{2}$$

Accordingly, the policy network (architecture given in table2) attempts to output actions that would maximize the value estimated by the value network. *i.e.* it learns to maximize the output of the value network.

Where as the value network $Q(s, a;\ \theta^Q)$ is updated to minimize the loss L:

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^\mu))^2 \tag{3}$$

Where y_i is the actual reward obtained at episode i and Q is the estimated value of taking action a_i in state s_i, parameterized by a neural network with parameters θ^Q (see table 1). *i.e.* the mean squared error of the value network as it tries to predict the return from the current episode.

Layer	Input	Output	Activation
Layer 1	24+2	256	ReLU
Layer 2	256	128	ReLU
Layer 3	128	1	None

Table 1: Value network architecture

Layer	Input	Output	Activation
Layer 1	24	256	ReLU
Layer 2	256	128	ReLU
Layer 3	128	2	tanh

Table 2: Policy network architecture

3 Hyperparameters:

Here is a direct snippet from the code listing the hyperparameters used. I did little exploration or deviation from most commonly used values in online implementations. The hyperparameters are named in sufficiently expressive terms to indicate their nature in the algorithm.

- state size= env info.vector observations.shape[1]
- action size= brain.vector action space size
- no agents= 1
- device= device
- gradient clip = 1
- episode count = 251
- buffer size = int(1e5)
- batch size = 128
- lr actor = 1e-4
- lr critic = 1e-4
- discount rate = 0.99
- tau = 1e-3
- weight decay = 0

4 Training:

The agent was trained locally on a NVIDIA TITAN RTX graphics card. Training the model takes about 1.5 hours of CPU/GPU time. The model learned quickly and stabilized as shown in the average reward plot 2. It solved the environment at episode 2450 (as seen in the plot, the 100 episodes running average of the score crosses 0.5 at that point).

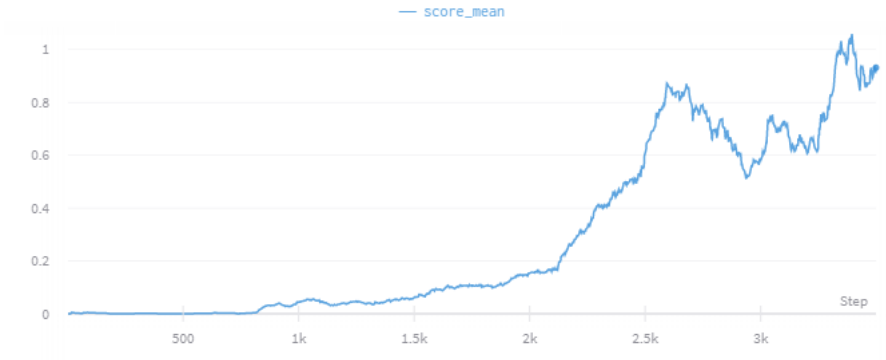

Figure 2: Average rewards over 100 episode

5 Future work:

Future work can include comparing the different setups possible in multi-agent RL, as listed above, learning and acting can both be centralized or decentralized. Additionally, if the direction of the state observations and the actions is flipped, the two agents can be controlled by the same DDPG model, with the same policy and value networks, thereby almost doubling the sample efficiency for each agent. This is possible because the two agents are solving very similar tasks, only flipped horizontally. In addition, considering the model's sensitivity to hyperparameters, a space search technique such as Bayesian optimization, might yield some efficient and possibly more robust regions in the hyperparameters space.

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.org*, September 2015.

[2] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv e-prints*, page arXiv:1706.02275, Jun 2017.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.