Project 5: Collaboration and Competition

Udacity Deep Reinforcement Learning Nanodegree Program

Bob Flagg

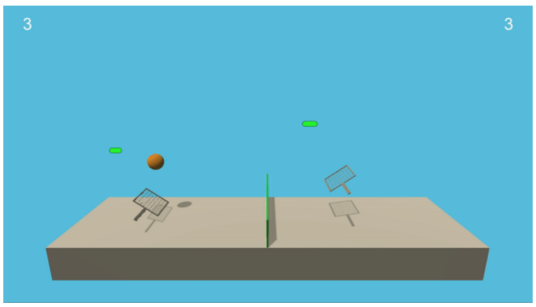

Figure 1

1 INTRODUCTION

In this project I'll implement two solutions to the Tennis environment from the Unity Machine Learning Toolkit. This is a multi-agent environment, where 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. 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, your 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. We then 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.

I'll use *Deep Deterministic Policy Gradient* [2] (DDPG) and *Multi-Agent Deep Deterministic Policy Gradient* [1] (MADDPG) to solve the environment. Source code in Python, using PyTorch, is available in the Collaboration and Competition repo.

2 BACKGROUND

The Tennis environment is a *sequential decision making problem*, in which two agents interact with each other and their environment over discrete time steps and try to find *policies* to maximize the expected *discounted return*:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where $\gamma \in [0, 1]$ is a discount factor that trades-off the importance of immediate and future rewards. See [3] for a general discussion of this sort of problem.

In this project I will use policy-based methods, which try to directly find an *optimal policy*, π^*, each agent can use to decide what actions to take.

Deep Deterministic Policy Gradient

In my first solution, I'll use *Deep Deterministic Policy Gradient* [2] (DDPG), which is motivated by an important connection between the action selected by an optimal policy and the *optimal action-value function*

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a).$$

Namely, if you know the optimal action-value function, then in any given state, s, an optimal action can be found by solving

$$\pi^*(s) = \arg\max_{a} Q^*(s, a).$$

DDPG concurrently learns an approximator to $Q^*(s, a)$ and an approximator to $\pi^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces.

Learning $Q^*(s,a)$. The starting point for approximating the action value function is the **Bellman Equation**:

$$Q^*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \cdot \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a\Big].$$

Suppose we are approximating $Q^*(s,a)$ with a neural network, $Q_\phi(s,a)$, and we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d), then the **mean-squared Bellman error** (MSBE)

$$L(\phi, \mathcal{D}) = \mathbb{E}_{\mathcal{D}}\Big[\Big(Q_\phi(s,a) - (r + \gamma \cdot (1-d) \max_{a'} Q_\phi(s',a'))\Big)^2\Big]$$

tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation and so can serve as the loss function in tuning ϕ.

Learning $\pi^*(s)$. Learning the optimal policy is pretty simple: we want to learn a deterministic policy $\pi^*(s)$ which gives the action that maximizes $Q^*(s,a)$. For this DDPG uses a neural network, $\pi_\theta(s)$, and loss function

$$L(\theta, \mathcal{D}) = -\mathbb{E}_{\mathcal{D}}\Big[Q_\phi(s, \pi_\theta(s))\Big].$$

Multi-Agent Actor-Critic

In my second solution, I'll use *Multi-Agent Deep Deterministic Policy Gradient* [1] (MADDPG).

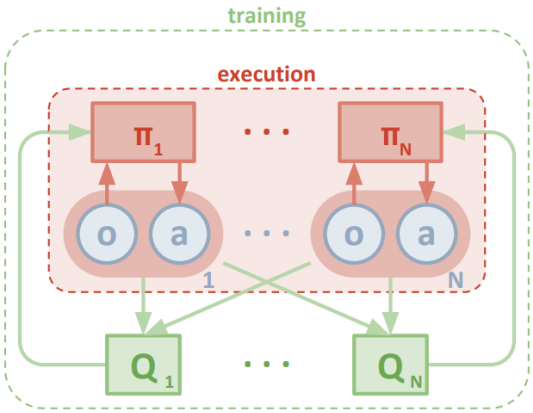

Figure 2: Multi-Agent Actor-Critic [1]

As illustrated in Figure 2, the key idea of MADDPG is to use **centralized critics**, which observe all states and actions, and **decentralized actors**, which only observe their own state.

3 PLAYING TENNIS WITH DDPG

In the DDPG solution to the Tennis environment I train a single agent that plays against itself. The code is in the Playing Tennis with DDPG notebook.

The key components and hyper-parameter settings are outlined below.

- **Actor Network**: One fully connected linear hidden layer with 256 units. The input size is equal to the environment state-size and the output size is equal to the environment action-size. It uses ReLu activation functions and batch normalization after the hidden layer.
- **Critic Network**: Three fully connected linear hidden layers. The first hidden layer has 256 units and has input size equal to the environment state-size. The second hidden layer has 256 units and input size equal to 256 + the environment action-size. The third hidden layer has 128 units. ReLu activation functions are used throughout and batch normalization is applied after the first layer. The output of this network is one-dimensional.
- **Hyperparameter Settings**: I trained for 1,500 episodes with a batch size of 128, discount factor of $\gamma = 0.99$, soft update weight of $\tau = 0.001$, actor learning rate of 0.0001, and critic learning rate of 0.001. To get DDPG to learn in a multi-agent environment I had to separate updating network parameters from adding samples to the replay buffer so that I could interleave these two steps in an appropriate proportion. After a bit of experimentation, I chose to update the networks 30 times after every 10 timesteps.

With the above settings I achieved an average score of 0.68 for the final 100 episodes and even managed to get an average score above 0.8 briefly during training.

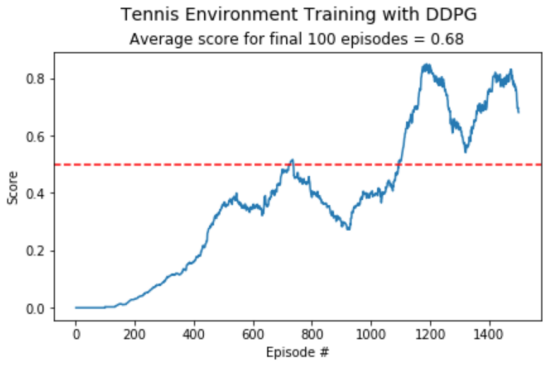

Figure 3: DDPG Training Scores

4 PLAYING TENNIS WITH MADDPG

The solution above is simple, stable and achieves a pretty high average score but it is not very satisfying from a multi-agent reinforcement learning point of view. In general it will not be possible to use the same model instance for all agents since different agents may need to achieve different

goals. In the Playing Tennis with MADDPG notebook I address that shortcoming by adapting DDPG to the multi-agent setting as in the paper *Multi-Agent Deep Deterministic Policy Gradient* [1] (MADDPG). I was able to solve the tennis environment with this approach but the results were disappointing so I won't bother describing hyper-parameter settings here. Figure 3 below shows the scores during training. It's not really surprising that this approach does not do nearly as well as self-playing DDPG because it does not take advantage of special features of the Tennis environment but the average score did reach 0.52 so at least it solved the task.

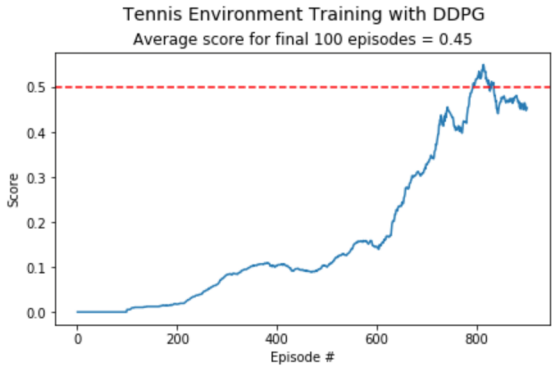

Figure 4: MADDPG Training Scores

5 IMPROVING PERFORMANCE

Deep Deterministic Policy Gradient did well on this task. To improve performance futher, I would do grid search on the hyper-parameters and network architectures for this approach. Another interesting direction would modify the MADDPG agorithm by allowing each critic instance to know which agent it is criticizing; that is, feed the corresponding agent's observation and action into the netword separately from those of the other agents so the critic could focus on the appropriate agent's actions and observtions whild still having access to all observations and actions.

REFERENCES

[1] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *CoRR* abs/1706.02275 (2017). arXiv:1706.02275 http://arxiv.org/abs/1706.02275

[2] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14)*. JMLR.org, I–387–I–395. http://dl.acm.org/citation.cfm?id=3044805.3044850

[3] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning - an introduction*. MIT Press. http://www.worldcat.org/oclc/37293240