Project 3: Multi-Agent Collaboration & Competition

Author : Engin Bozkurt

Main Goal Of the Project:

Training two RL agents to play tennis. As in real tennis, the goal of each player is to keep the ball in play.

The properties of environment framework

It is an environment that is similar, but not identical to the Tennis environment on the Unity ML-Agents GitHub page

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. Each agent receives its own, local observation. Two continuous actions are available, corresponding to moves 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).

To recap,

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.

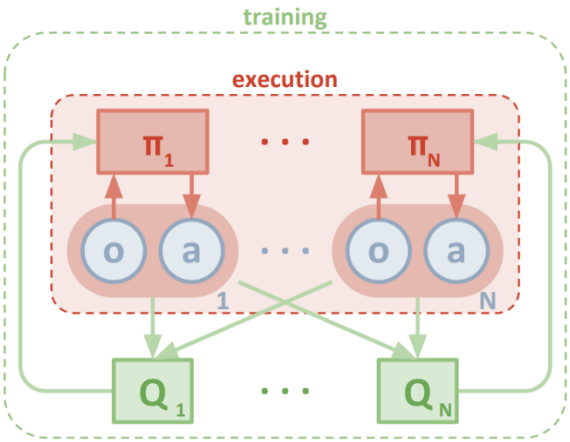
The environment is considered solved when the average (over 100 episodes) of those scores is at least +0.5.

Implementation

Multiagent environments have two useful properties: first, there is a natural curriculum — the difficulty of the environment is determined by the skill of your competitors (and if you're competing against clones of yourself, the environment exactly matches your skill level). Second, a multiagent environment has no stable equilibrium: no matter how smart an agent is, there's always pressure to get smarter. These environments have a very different feel from traditional environments

In this project, we need multiple agents for our goal, so I implemented the MADDPG algorithm, for centralized learning and decentralized execution in multiagent environments, allowing agents to learn to collaborate and compete with each other. **MADDPG extends a reinforcement learning algorithm called DDPG**, taking inspiration **from actor-critic reinforcement learning techniques**; other groups are exploring variations and parallel implementations of these ideas.

We treat each agent in our simulation as an "actor", and each actor gets advice from a "critic" that helps the actor decide what actions to reinforce during training. Traditionally, the critic tries to predict the value (i.e. the reward we expect to get in the future) of an action in a particular state, which is used by the agent - the actor - to update its policy. This is more stable than directly using the reward, which can vary considerably. To make it feasible to train multiple agents that can act in a globally-coordinated way, we enhance our critics so they can access the observations and actions of all the agents, as the following diagram shows.



Exploration vs Exploitation

In Reinforcement Learning, epsilon an important hyperparameter that controls how much the agent should explore and exploit when using epislon-greedy policy

The decaying Epsilon Greedy methods, tries to decrease the percentage dedicated for exploration as time goes by. This can give optimal regret.

Time-Based Decay

The mathematical form of time-based decay is lr = lr0/(1+kt) where lr, k are hyperparameters and t is the iteration number. Looking into the source code of Keras, the SGD optimizer takes decay and lr arguments and update the learning rate by a decreasing factor in each epoch.

I tried to apply a similar approach into decaying Epsilon Greedy here:

```
self.eps_decay = 1/(EPS_EPOSIDE_END*LEARN_NUM)  #set decay rate based on epsilon end target

LEARN_NUM = 4                  # number of learning passes
EPS_EPOSIDE_END = 300          # required episode number to finish the decaying Epsilon
```

I used an **Ornstein-Uhlenbeck process** (Uhlenbeck & Ornstein, 1930) **with θ = 0.15 and σ = 0.2**.

Reference: (pg. 11)

Supplementary Information: Continuous control with deep reinforcement learning research paper

(Published as a conference paper at ICLR 2016)

```
OU_SIGMA = 0.2        # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.15       # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
```

For the aggressive exploration of the action space, EPS_START parameter is set at 6.0 and therefore improved the chances that signal would be detected.

This extra signal seemed to improve learning later in training once the noise decayed to zero.

Learning Interval

In the first few versions of my implementation, the agent only performed a single learning iteration per episode.

In general, I found that performing multiple learning passes per episode yielded faster convergence and higher scores. This did make training slower, but it was a worthwhile trade-off. In the end, I implemented an interval in which the learning step is performed every episode. As part of each learning step, the algorithm then samples experiences from the buffer and runs the ***Agent.learn()*** method.

Hyperparameters

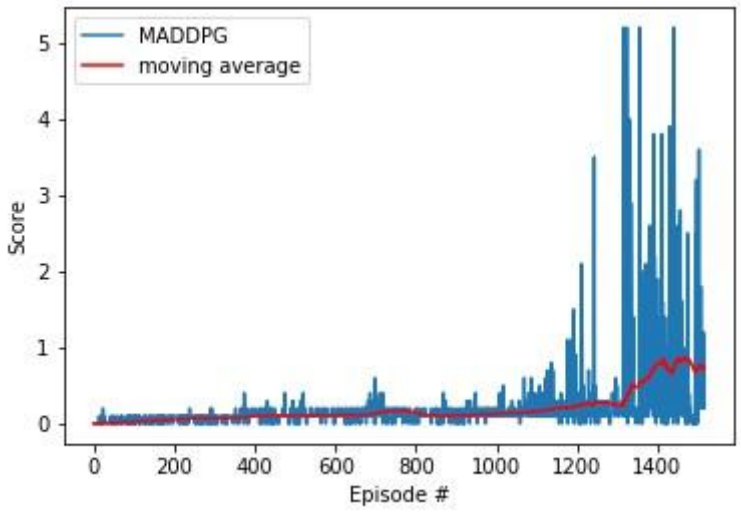
```
BUFFER_SIZE = int(1e6)      # replay buffer size
BATCH_SIZE = 128            # minibatch size
LR_ACTOR = 1e-3             # learning rate of the actor
LR_CRITIC = 1e-3            # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
LEARN_EVERY = 1         # learning timestep interval
LEARN_NUM = 4           # number of learning passes
GAMMA = 0.99            # discount factor
TAU = 7e-2              # for soft update of target parameters
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.15         # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
EPS_START = 6.0         # initial value for epsilon in noise decay process
EPS_EPOSIDE_END = 300   # required episode number to finish the noise decay process
EPS_FINAL = 0           # final value for epsilon after decay
N_EPISODES  = 2000      # number of episodes for MADDPG algorithm
```

Gradient Clipping

To fix the issue of exploding gradient , I implemented gradient clipping using the torch.nn.utils.clip_grad_norm function. I set the function to “clip” the norm of the gradients at 1, therefore placing an upper limit on the size of the parameter updates, and *preventing them from growing exponentially*. Once this change was implemented, my model became much more stable and my agent started learning at a much faster rate.

Results

The graph below shows the final training results.



The agents were able to solve the environment in 1240 episodes, with a top score of 5.2 and a top moving average of 0.507

Future Improvements

1. Memory-driven Communication

A framework for multi-agent training using deep deterministic policy gradients that enables the concurrent, end to- end learning of an explicit communication protocol through a memory device.
(Reference: Improving Coordination in Multi-Agent Deep Reinforcement Learning through Memory-driven Communication research paper)

2. Prioritized experience replay

Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error.

3. Having a modular Q function

The input space of Q grows linearly (depending on what information is contained in x) with the number of agents N. This could be remedied in practice by, for example, having a modular Q function that only considers agents in a certain neighborhood of a given agent.

References:

1. Improving Coordination in Multi-Agent Deep Reinforcement Learning through Memory-driven Communication

https://www.researchgate.net/publication/330383029_Improving_Coordination_in_Multi-Agent_Deep_Reinforcement_Learning_through_Memory-driven_Communication

2. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments

https://arxiv.org/abs/1706.02275

3. Continuous control with deep reinforcement learning

https://arxiv.org/abs/1509.02971

4. Learning to Cooperate, Compete, and Communicate / OpenAI

https://openai.com/blog/learning-to-cooperate-compete-and-communicate/