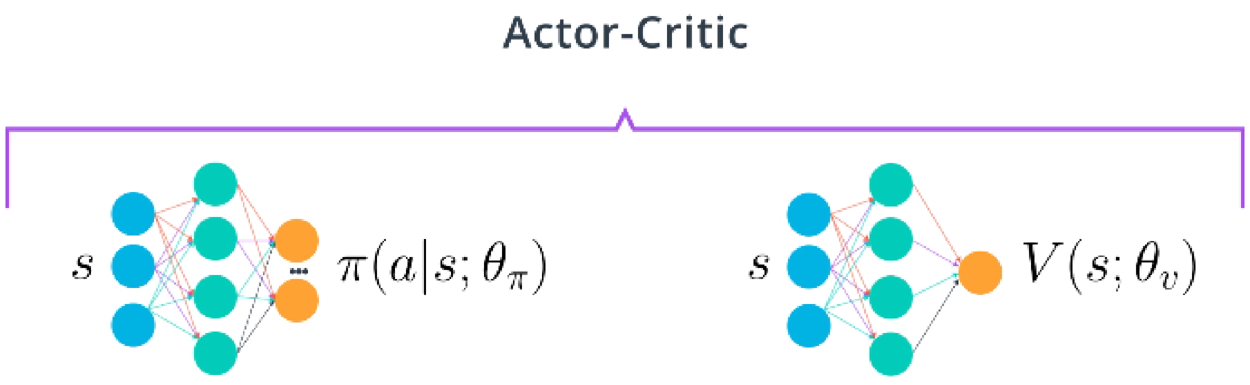
Udacity Deep Reinforcement Learning Nanodegree

Project 3: Collaboration and Competition

Introduction

This report details the implementation of a reinforcement learning algorithm to solve the multiagent tennis game problem in Unity's Tennis environment. The solution presented here uses the Deep Deterministic Policy Gradient (DDPG) algorithm, which is a form of model-free actor-critic reinforcement learning that operates over continuous action spaces, and implements prioritized experience replay (PER) to improve performance.



DDPG is a method that is similar to Deep Q Learning, and uses two neural networks to complete a task – the Actor to determine the best action from the policy deterministically,

$$a^* = argmax_a Q(s,a),$$

and the Critic to determine the action value function,

$$Q(s,\mu(s;\theta_\mu);\theta_Q).$$

The algorithm runs as follows (from Lillicrap et al. 2016):

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ 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
```

Implementation Details

As in the original Lillicrap implementation, we also use a replay buffer containing tuples of previous transitions, that we periodically revisit to improve sample efficiency. However, unlike the Continuous Control project, I found here that uniform sampling of the replay buffer was too inefficient. The intuition behind this is that many of the episodes in the buffer are encoding experiences that the neural network has already learned well from, so revisiting them adds little benefit. Instead, we can give focus to the experiences that had greater error. To do this, I implemented prioritized experience replay (PER), which probabilistically samples from the replay buffer with a sample probability based on the TD error δ_t

$$\delta_t = R_{t-1} + \gamma \max_a \hat{q}(S_{t+1}, a, \boldsymbol{w}) - \hat{q}(S_t, A_t, \boldsymbol{w})$$

The priority is then defined as

$$p_t = |\delta_t| + \epsilon$$

where ϵ is a value to ensure that all replays have nonzero probability of being chosen. Then the sampling probability of any given replay i is

$$P(i) = \frac{p_i^a}{\Sigma_{\mathrm{k}} p_k^a}$$

with a modified weight update rule of

$$\Delta \boldsymbol{w} = \alpha \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^b \delta_i \nabla_w \hat{q}(S_i, A_i, \boldsymbol{w})$$

In addition, a soft update is used in order to slowly blend a target and the regular network weights. In this solution, both the actor and the critic are composed of three-layer neural networks. All the layers here were fully-connected layers.

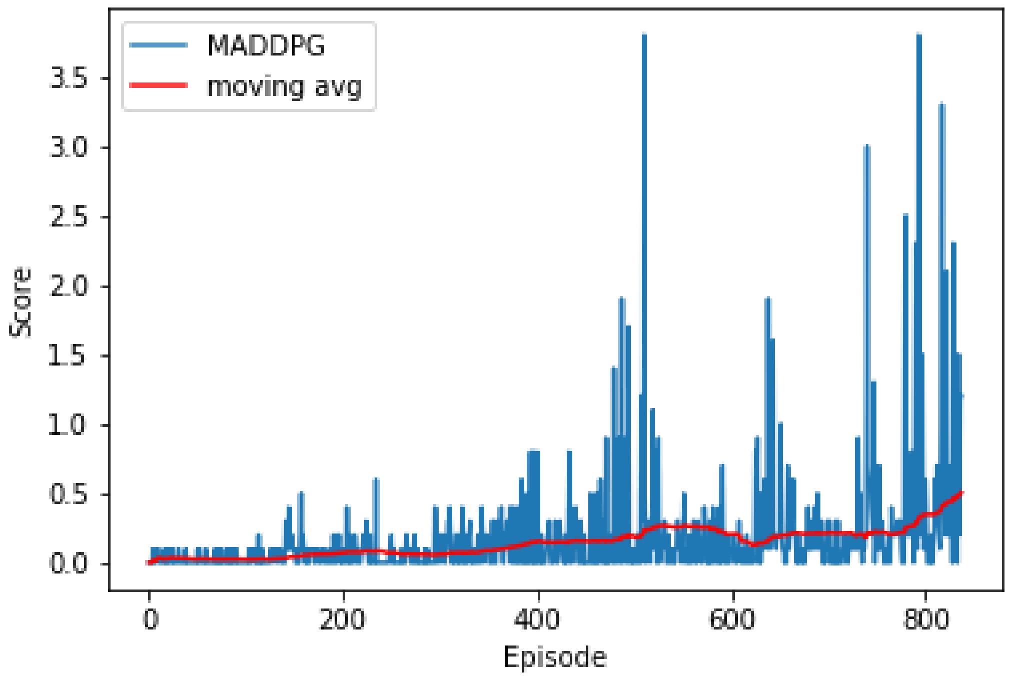
In order to encourage the agent to explore, we impose noise on the actions that are chosen by the actor, so that the actions that are actually taken have an additional term dictated by an Ornstein-Uhlenbeck process. This is a stationary Gauss-Markov process, and is a form of mean-reverting random walk. We also impose a decay factor on the process, to reduce the amount of exploration as time goes on and the agent's policy becomes more feasible. Since the action output can only take values between -1 and 1, the outputs of the actor are clipped to be between -1 and 1.

The chosen hyperparameters are as follows (in the hyperparameter.py file):

```
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_PASSES = 5            # number of learning passes

GAMMA = 0.99                # discount factor
TAU = 1e-2                  # for soft update of target parameters

OU_SIGMA = 0.5              # 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 in Agent.act()
EPS_EP_END = 350            # episode to end the noise decay process
EPS_FINAL = 0               # final value for epsilon after decay

PRIORITY_EPS = 0.01         # small factor to ensure that no experience has zero sample probability
PRIORITY_ALPHA = 0.5        # how much to prioritize replay of high-error experiences
```

Results

The DDPG training solved the environment in 738 episodes (with the window of episodes 738-838 averaging more than +0.5 reward). The plot of the rewards is shown below. Previous attempts to solve this environment without PER were unable to solve even after 2500 episodes, with the highest average scores being no more than ~0.03. It is notable that the average score increased dramatically after approximately 750 episodes, and even before that, a bit after 600 episodes, we see that instances of episodes with a score of zero became much more rare than in previous periods.



Future Work

Bootstrap initial training by imitating a pre-programmed controller. In the environment, the agent is learning to play a game of tennis from scratch. It would not be difficult to program an expert system to play tennis with "classic AI," which can serve as the basis for initializing learning. This might be done by biasing the agent's actions towards what a known controller would do, before gradually decreasing the bias – a form a guided exploration, rather than the purely stochastic Ornstein-Uhlenbeck process used here. This is similar to the way that AlphaGo and AlphaStar initially imitated professional Go and Starcraft players to get their policies up to a reasonable state before training on their own.