SAYON PALIT

26th June 2020

Deep Reinforcement Learning

Project 3 :Collaboration and Competition

OVERVIEW

This project aims to use deep reinforcement learning ,more specifically OpenAI's Multi Agent Deep Deterministic Policy Gradient (MADDPG) algorithm to solve a Unity based muti-agent environment where the main objective is to train each player to keep the ball in play for longer durations than the other.

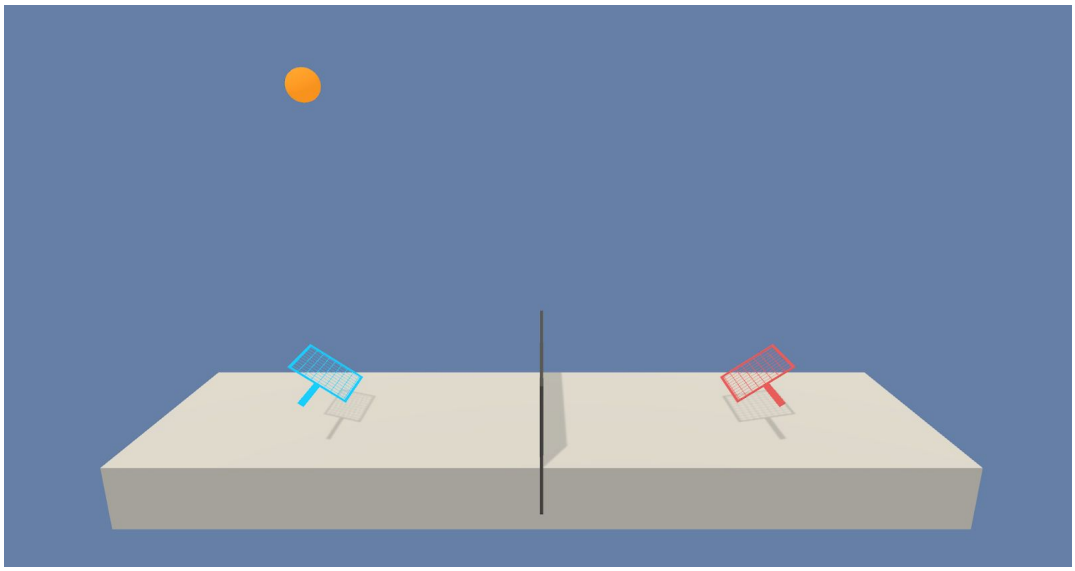


Figure 1 depicts the environment.

- 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 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.

LEARNING ALGORITHM

The learning algorithm used in this project is part of the **Multi Agent Actor-Critic** family of Deep Reinforcement Learning Algorithms.

Why Actor-Critic Models ?

It is not possible for Value-based models like DQN to provide results in continuous action spaces like the one in this current project.Yes, discretization is possible , but after a certain extent the optimization entailed in the Sarsamax algorithm becomes non-trivial as we have to find the maximal Q-value over a vast action space.

Enter into the scenario, Policy-based models which try to directly find the optimal policy function instead of the optimal Value function.Thus they can be used in discrete as well as continuous action spaces as well as stochastic and deterministic.The main issue with policy functions is to find a baseline for evaluating how good our proposed policy is.

Thus the Actor-Critic model uses a hybrid of both these techniques as follows:-

- Actor - The policy based model controls how our agent behaves.
- Critic - The value based model evaluates how good the action taken by the actor is.

Algorithm

The Tennis environment requires two separate agents(Player 1 and 2) to be trained and these two need to collaborate in some situations for instance not letting the ball touch the table and compete with each other at times like trying to achieve a higher score than their opponent.If we just use single agent policies in this environment , convergence to optimal policy will be infeasible as the environment may appear non-stationary to the agents oblivious of each other's existence and interactions with the environment.

For the above reasons as well as taking into mind Udacity's benchmark implementation for solving this environment, I have chosen the off-policy Multi Agent Deep Deterministic Policy Gradient (MADDPG) for solving this environment.

1.The basic notion behind MADDPG is that if all the actions that were taken by all the agents are known to us, the environment is stationary even if the policies for the agents change.

2.In MADDPG, each agent's critic is trained using the observations and actions from all the agents, whereas each agent's actor is trained using just its own observations. This allows the agents to be effectively trained without requiring other agents' observations during inference because the actor is only dependent on its own observations.

3.We inherit the features of the DDPG algorithm like the **Replay Buffer** for sampling experience tuples thus removing correlation inherent to on-policy Reinforcement techniques.We use the **Learning Intervals** method to balance learning and exploration.We use the **Ornstein-Uhlenbeck process** proposed by DeepMind to help with the Exploration vs Exploitation decisions.We use the concept of **Target Networks and Local Networks** along with the **Soft Update technique** to remove instability of the moving targets first seen in the Deep Q-Network project.

Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm below.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)\big|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1 \quad \tau)\theta'_i$$

  end for
end for

Figure 2 MADDPG Algorithm sourced from OpenAI's paper

MODEL ARCHITECTURE

The model architecture uses multiple copies of the actor and critic neural net with three fully connected layers for each agent of the environment given by num_agents.

- The actor as well as critic networks take in state size as input and then outputs fc1_units(400)
- Next the ReLU activated first layer output undergoes batch normalization.
- After BatchNorm the output it supplied to the second fully connected layer which outputs FC2_units(300)
- The ReLU activated output is passed on to the third and final fully connected layer which outputs the tanh activated action values for the actor while no activation is applied to the Critic network final layer.

Actor Forward Pass

```
h1 = self.nonlin(self.fc1(state))
h1 = self.bn1(h1) # Batch Normalization after Activation
h2 = self.nonlin(self.fc2(h1))
return F.tanh(self.fc3(h2))
```

Critic Forward Pass

```
# Modified DDPG architecture
xs = torch.cat((state, action.float()), dim=1)
x = self.nonlin(self.fcs1(xs))
x = self.bn1(x) # Batch Normalization after Activation

x = self.nonlin(self.fc2(x))
return self.fc3(x)
```

- **Gradient Clipping :** was applied to stabilize the learning as the outsized gradients lead to plateauing or collapse of the training of the agent.
- **Learning Intervals :** Earlier iterations of this project learnt at every step which made training a very time-consuming process without any outright benefits to the agent's

performance.So a new hyperparameter update_every was introduced to to perform learning in every **update_every timesteps**.

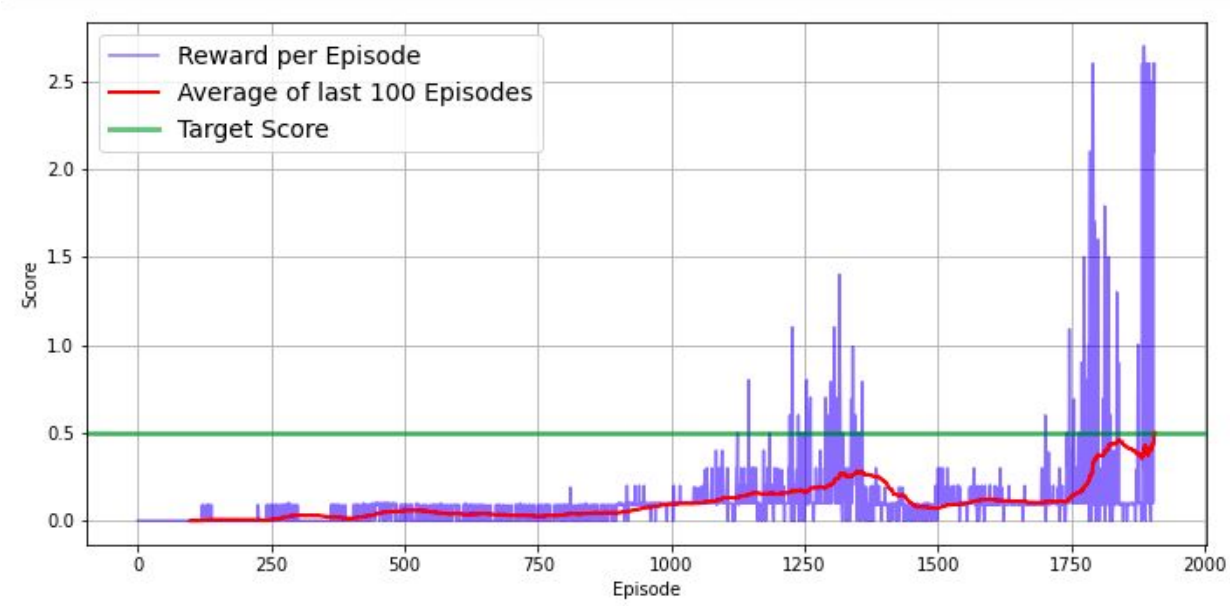
As part of each learning step, the algorithm samples experiences from the buffer and runs the Agent.learn() method learn_num times.

HYPERPARAMETERS

```
SEED = 33                           # Random seed

NUM_EPISODES = 5000                 # Max num of episodes
NUM_STEPS = 1000                    # Max timesteps per episodes
UPDATE_EVERY = 20                   # Learning timestep interval
MULTIPLE_LEARN_PER_UPDATE = 10      # Number of learning passes

BUFFER_SIZE = int(1e6)              # replay buffer size
BATCH_SIZE = 128                    # minibatch size

ACTOR_FC1_UNITS = 400               # Number of units for layer 1 in the actor model
ACTOR_FC2_UNITS = 300               # Number of units for layer 2 in the actor model
CRITIC_FCS1_UNITS = 400             # Number of units for layer 1 in the critic model
CRITIC_FC2_UNITS = 300              # Number of units for layer 2 in the critic model
NON_LIN = F.relu                    # Non linearity operator used in the model
LR_ACTOR = 1e-3                     # learning rate of the actor
LR_CRITIC = 1e-3                    # learning rate of the critic
WEIGHT_DECAY = 0                    # L2 weight decay

GAMMA = 0.995                       # Discount factor
TAU = 1e-3                          # For soft update of target parameters
CLIP_CRITIC_GRADIENT = False        # Clip gradient during Critic optimization

# Ornstein-Uhlenbeck noise

ADD_OU_NOISE = True                 # Add Ornstein-Uhlenbeck noise
MU = 0.                             # Ornstein-Uhlenbeck noise parameter
THETA = 0.15                        # Ornstein-Uhlenbeck noise parameter
SIGMA = 0.2                         # Ornstein-Uhlenbeck noise parameter
NOISE = 1.0                         # Initial Noise Amplitude
NOISE_REDUCTION = 1e-6      # Noise amplitude decay ratio
model_dir = 'weight1'
```

RESULTS

```
  4%|▍                                                                | 204/5001 [00:47<25:48,  3.10it/s]
Episode 200     Average Score: 0.01     Max over agents: 0.00(timesteps=2970 )
  8%|▊                                                                | 402/5001 [01:39<37:51,  2.02it/s]
Episode 400     Average Score: 0.01     Max over agents: 0.00(timesteps=6597 )
 12%|▉                                                                | 603/5001 [02:47<35:17,  2.08it/s]
Episode 600     Average Score: 0.04     Max over agents: 0.00(timesteps=11269 )
 16%|▏                                                               | 802/5001 [03:53<1:26:42,  1.24s/it]
Episode 800     Average Score: 0.04     Max over agents: 0.09(timesteps=15363 )
 20%|▏                                                               | 1003/5001 [05:59<1:01:13,  1.09it/s]
Episode 1000    Average Score: 0.10     Max over agents: 0.10(timesteps=21174 )
 24%|▍                                                               | 1201/5001 [08:48<8:55:06,  8.45s/it]
Episode 1200    Average Score: 0.15     Max over agents: 0.29(timesteps=32343 )
 28%|▊                                                               | 1401/5001 [14:36<5:12:30,  5.21s/it]
Episode 1400    Average Score: 0.22     Max over agents: 0.19(timesteps=50037 )
 32%|▏                                                               | 1602/5001 [17:05<56:20,  1.01it/s]
Episode 1600    Average Score: 0.12     Max over agents: 0.09(timesteps=58229 )
 36%|▌                                                               | 1802/5001 [20:47<3:08:25,  3.53s/it]
Episode 1800    Average Score: 0.38     Max over agents: 0.10(timesteps=77194 )
 38%|▊                                                               | 1906/5001 [25:02<40:39,  1.27it/s]
Environment solved in 1906 episodes with an Average Score of 0.50
```



As can be seen from the plot above ,the current model solves the environment in 1906 episodes with an average score of 0.50. I have run the project multiple times and it seems to mysteriously decrease the average score around the 1000-1400 episodes range and then goes back up.If this were not the case, then the environment could be solved in less than 1500 episodes.

FUTURE WORK

- **Prioritized experience replay** is an improvement to the experience replay procedure, and the main idea behind it is to accord more interest on experiences that seems to be more important for the learning procedure.
- I would also like to try out Multi Agent Proximal Policy Optimization(MAPPO) or another algorithm called Twin Delayed DDPG(TD3) which is more robust to hyperparameter tuning and has addressed the issue of overestimated Q functions in the Critic network.
- Also the hyperparameters as well as network layer structure can be further tweaked to reduce the solving time from the current 1900-2000 episodes to even lower.
- I would also like to implement the same model on the Soccer Unity Environment.

REFERENCES

1.ddpg-pendulum - I used this DDPG implementation by our Course Instructor Alexis Cook as the template for my project along with the MADDPG lab tutorial in our course in Udacity..

2.[1509.02971] Continuous control with deep reinforcement learning

3.Deep Reinforcement Learning Online Course - Udacity

4.[1706.02275] Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments

5.Deep RL Bootcamp Lecture 7 SVG, DDPG, and Stochastic Computation Graphs (John Schulman)