Deep Reinforcement Learning Nanodegree

Project 3 : Multiagent Reinforcement Learning for playing Tennis

Introduction

In this problem, we train a multiagent tennis playing agent in Unity ML agent environment. We use a variation of the previously studied DDPG algorithm, called MADDPG, where two agents act independently to maximize the total reward.

A reward of +0.1 is provided for each step that an agent hits the ball over the net. And -0.01 if ball runs out of bound or hits the ground

Observation space : 24 dimensional vector for each agent, corresponding to position, velocity information of ball and racket

Action Space : Two dimensional vector corresponding to movement towards or away from net

Goal : The goal is to obtain a average reward >= 0.5 over a course of 100 consecutive episodes

Learning Algorithm

We are using a modification to a popular actor-critic based method, **DDPG algorithm** to train the agent in Multiagent Reinforcement Learning (MARL) setting. These algorithms address shortcomings of both policy based and value based methods.

DDPG leverages policy based methods to take best actions that maximize agents' total cumulative reward over a trajectory, while also utilizing the value based methods idea to effectively estimate goodness different action values. And the key thing is, the algorithm is learning policies in high dimensional continuous action spaces.

More into the Modification Involved

Multi-Agent Deep Deterministic Policy Gradient (DDPG):

Our DDPG in this case is modified for the MARL problem. In our Table Tennis problem, two agents need to collaborate with each other to maximize final reward. Traditional approach to RL

involving independent training won't work in this problem or any multiagent problems. The common problem is, the environment will essentially be non stationary from the perspective of just one agent, as that agent won't know how the other agent will behave. MADDPG architecture is interesting due to its simplicity.

To ease training, the both agent's critic network can access information from both agent's actor network (centralized training where critic network can excess policy,observation of every other agent) , while in execution, the actors network of each agent functions independently (decentralized execution).

Algorithm Features accompanying stable learning (similar to DDPG)

Significant problem with Deep RL lies in making the gradient update (both ascent and descent) stable, since we are just working with sample interactions. Due to the high variance of environment signal, we leverage certain techniques to assist in stable learning as well as continual exploration, since, unlike DQN, we don't have privilege to use epsilon greedy over discrete actions.

1) **Common Experience Replay for both agents** (Motivated by DQN to break sequential dependency)
2) **Ornstein-Uhlenbeck noise** (Promotes exploration, while subtly tending to stay in same direction, rather than, negative actions canceling out in long run as sampled with uniform random distribution)
3) **Batch Normalization** (By adding Batch normalization to every layer, motivated stable gradient update.)
4) **Fixed Targets** (Same motivation as in DQN, to chase a stable target)
5) **Soft Updates** (Defined by parameter *tau* that slowly mixes local net parameters into target)

Neural Network Architecture

Both actor and critic are fundamental two layer feed forward neural networks, similar to one as in DDPG algorithm, for past Continuous Control Project.

But key modification is, as training is centralized, each agent's critic has access to state vectors and action vectors from both agents.

After training, we have 4 model checkpoints saved, corresponding to the actor-critic network pair of each agent.

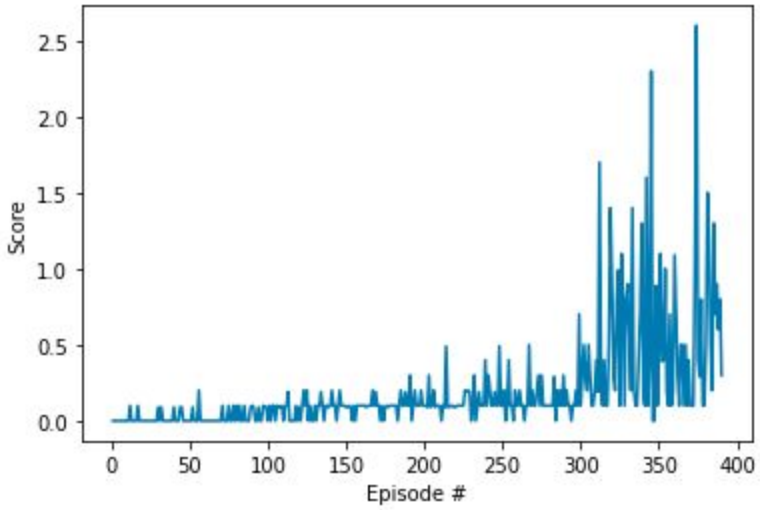
Also, due to multiple updates at each UPDATE_EVERY timestep, training became slow, but, goal state was reached in just 389 episodes. Intuitively, since, useful experiences are sparse, I believed multiple gradient updates will help agents network to effectively utilize reward signals.

Hyperparameters

```
UPDATE_EVERY = 1        # time step gap between updating network
NO_OF_UPDATES = 3       # times of network update at UPDATE_EVERY timestep
INITIAL_NOISE_FOR_OU = 2        # motivation exploration
NOISE_DECAY_AT_EACH_STEP = 0.9999        # motivatie exploitation

BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 0.02              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 3e-4        # learning rate of the critic
WEIGHT_DECAY = 1.e-5        # L2 weight decay
```

Plot of Rewards



Ideas for Future work

- Combining Prioritized Experience Replay to samples better.
- Using PPO with Multiagent setting (MAPPO), since these algorithms are better suited to continuous control settings and maximizing rewards over a long run
- I feel like just optimizing the hyperparameters will give better results for MADDPG as well.