Project 3: Multi-Agent Collaboration and Competition

Train Two agents to play Tennis

Goal

Goal of this project is to train two agents to play Tennis. Main goal of each players is to keep ball in air as long as possible and defeat it`s opponent. Base line criteria is score of 0.5 over last 100 episodes.

Environment

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

Two agents control tennis rackets to bounce a ball over a net. If the agent hits the ball over the net, it gets +0.1 reward. If the agent misses or lets ball fly over the bounds, it will get -0.01. Because of that the goal is to keep ball in play as long as possible.

The observation space consists of 8 variables corresponding to position and velocity of the ball and racket. Each agent gets its own observation vector. There are two continuous actions available:

- *Moving forward/backward*
- *Jumping*

This task is episodic, and because of that agent must get average score of **+0.5 over last 100** episodes (after taking the maximum over both agents).

- After each episode, we add rewards of each agent (without discounting) to get a score for each agent. Then we take maximum of those scores
- Above is score for single episode.

Learning Algorithm: Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

Originally this algorithm is extension of DDPG algorithm, described in below paper by Google Deepmind:

https://arxiv.org/pdf/1509.02971.pdf

In this paper, authors present model-free, off-policy actor-critic algorithm for continuous action spaces. In my implementation I used D4PG algorithm, that I`ve created for previous project. I take that approach, because D4PG shown to be better in mentioned project, than DDPG, so I assumed that it would work here too.

Main change in this algorithm relates to multiple competitive agents in the Tennis environment. I implemented additional components from paper: https://papers.nips.cc/paper/7217-multi-agent-actor-critic-for-mixed-cooperative-competitive-environments.pdf.

Main change relates to actor and critic networks, and interaction between agents.

Actor-Critic

Actor-critic methods uses best features of both policy-based and value-based methods.

- Policy-based approach, the agent (actor) learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent.
- Because of value-based approach, the agent (critic) learns how to estimate the value (i.e., the future cumulative reward) of different state-action pairs.

Actor-critic methods combine these two approaches to accelerate the learning process and because of this whole system is more stable than value-based agent, while requiring fewer training samples than policy-based agent.

In the paper cited above there is decentralized actor with centralized critic approach. Traditional actor-critic methods have a separate critic for each agent, this approach utilizes a single critic that receives as input the actions and state observations from all agents. This extra information makes training easier and allows for centralized training with decentralized execution. Each agent still takes actions based on its own unique observations of the environment.

Logic for that is implemented in

- model.py – networks for actor/critic
- DDPG_agent.py – single agent that gets batch plus actions
- MADDPG_agent.py - wrapper to handle multiple agents/ action choosing etc.

Experience Replay

Experience replay is important part of this algorithm, because it allows us to make data uncorrelated. Experiences are stored in single replay buffer for all agents in MADDPG_agents.py and each time we want to store/get data samples from it we take it from there. Additionally, usage of replay buffer allows us to learn multiple times from given experience. Implementation of replay buffer is in Priority_replay.py and it`s taken from my previous projects.

Exploration vs Exploitation

IN order to address this problem, we`re sing Ornstein-Uhlenbeck process, as mentioned in paper: https://arxiv.org/pdf/1509.02971.pdf. This process adds noise to the action values, at each step we want to choose them. Implementation of this can be found in file: OUNoise.py. In this project I`ve used default values recommended in paper and those values were also used in my previous project that used D4PG algorithm. There is also noise decay used, in order to make actions taken less random each step we`re going forwards with learning.

Single agent

As for single agent, in the paper mentioned at the beginning, there is DDDPG used. IN this project I wanted to reuse algorithm I`ve developed for previous one (D4PG). For details of implementation see: https://github.com/mizzmir/Deep-Reinforcement-Learning/blob/master/reacher/Report.pdf

Main changes in D4PG:

- Prioritized Replay
- Usage of Categorical Distribution
- Usage of n-step return

When I tried to make all those things to work together, I`ve had one problem: My agent didn`t wanted to learn as it should. After using some tensorboard checking it was visible that I`ve had problem of big gradients, and it was the reason of learning being very unstable. Because I`m using few learning per episode, this problem become much worse/bigger than before. To fight this I`ve used gradient clipping: `torch.nn.utils.clip_grad_norm`.

This one line fights big gradients by making upper limit on the size of updates. Because of that there is no big growth in their value when there is few learn steps each iteration.

This can be found in DDPG_agent.py file, before optimizer next step.

Another thing was usage of batch normalization in model, after first layer of NN. This can be found in model.py

As for learning process I`ve joined few things together, that speed up learning a bit:

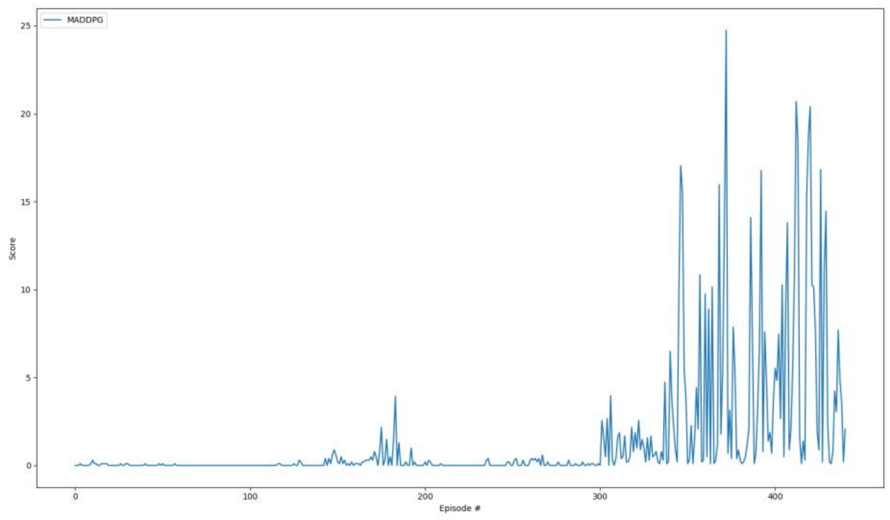
- Learning few times each iteration
- Soft update during each learn step
- Hard update during 100 full learning loops

Results

Below there is one run of algorithm. Other runs are similar, maybe +-10 episodes until environment is solved.

Sample algorithm was solved in episode 341(more detailed values are in results.pdf file):

```
Reached  0.543873473104  in episode  341
```



Future Improvements

- Try to use different hyperparameters
- Try to use pixel learning
- Try to use Gaussian Distribution