Multi-Agent DDPG Reinforcement Learning using PyTorch and Unity ML-Agents

A simple example of how to implement vector based DDPG for Multi-Agent Reinforcement Learning (MARL) Tasks using PyTorch and an ML-Agents environment.

The example includes the following DDPG related python files:

- **ddpg_agent.py:** the implementation of a DDPG-Agent
- **replay_buffer.py:** the implementation of a DDPG-Agent 's replay buffer (memory)
- **model.py:** example PyTorch Actor and Critic neural networks
- **train.py:** initializes and implements the training processes for a DDPG-agent.
- **test.py**: tests a trained DDPG-agent

The repository also includes links to the Mac/Linux/Windows versions of a simple Unity environment, *Tennis*, for testing. This Unity application and testing environment was developed using ML-Agents Beta v0.4. The version of the Banana environment employed for this project was developed for the Udacity Deep Reinforcement Nanodegree course. For more information about this course visit: https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

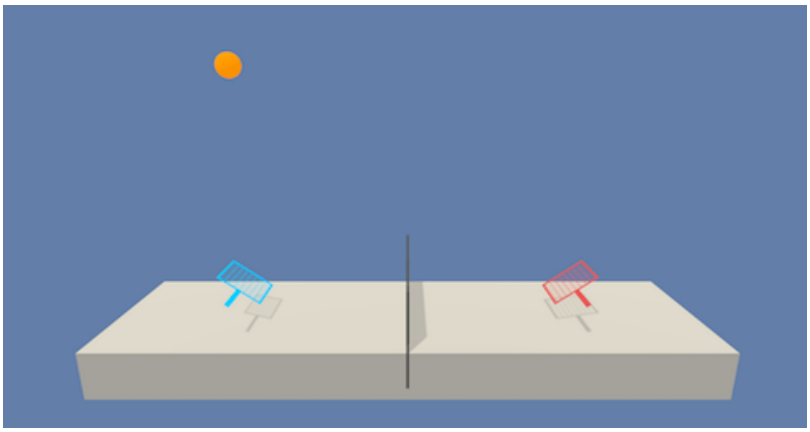
The files in the python/. directory are the ML-Agents toolkit files and dependencies required to run the Tennis environment. For more information about the Unity ML-Agents Toolkit visit: https://github.com/Unity-Technologies/ml-agents

Example Unity Environment – Tennis

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

Multiagent Training:

The Tennis environment contains two unity agents. Each agent needs to collects observations from itself and its co-player. The task is essentially a cooperative task in that both agents maximize reward by hitting the ball back and forth for as long as possible.



Example of the Agents view of the Tennis Environment

State and Action Space

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.

DDPG

DDPG is a (Actor-Critic) policy-based method of deep reinforcement learning than can be employed for both discrete and continuous tasks. The pseudo-code for the DDPG algorithm is provided below (also see below reference for more details).

Algorithm 1 DDPG algorithm

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

Reference:

[Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971.*]

DDPG Implementation

I implemented the ***DDPG*** code I employed for the Reacher project. I adapted and split out the *Replay_Buffer.py* and *UNoise.py code* and *ddpg_agent.py* code and amended the code to received data from multiple unity agents. I then developed the training and testing python scripts (train.py and test.py) for training and testing purposes. The implementation employs two DDPG agents (agent_1 and agent_2). Each agents received the its own observations, as well as the observations of its co-player (24x2=48 state observations).

Hyperparameters

DDPG Agent Parameters

- state_size (int): dimension of each state
- action_size (int): dimension of each action
- replay_buffer size (int): size of the replay memory buffer

- batch_size (int): size of the memory batch used for model updates
- gamma (float): parameter for setting the discount value of future rewards
- tau: for soft update of target parameters
- LR_Actor_rate (float): specifies the rate of Actor model learning
- LR_Critic_rate (float): specifies the rate of Actor model learning
- Weight_Decay (float): decay rate of learning rates.

The Tennis environment is a relative simple environment and, thus standard DDPG hyperparameters are sufficient for timely and robust learning. The recommend hyperparameter settings are as follows:

- state_size: **48**
- action_size: **2**
- replay_memory size: **5e5**
- batch_size: **128** (256 also works well)
- gamma: **0.99**
- tau: **5e-3**
- Actor learning_rate: **1e-3**
- Critic learning rate: **1e-3**
- Weight_Decay: **0.0**

Updated OUNoise Function

I also updated the Ornstein-Uhlenbeck noise process, to include a decayed sigma value (e.g., ***sigma=1.0*** and decays from sigma to ***sigma_min=.05*** by a factor of ***sigma_decay=.99***). I found that having a large amount of noise during the early stages of training, maximized exploration and resulted in overall faster and more robust learning.

Training Parameters

- num_episodes (int): maximum number of training episodes
- scores_average_window (int): the window size employed for calculating the average score (e.g. 100)
- solved_score (float): the average max player score over 100 episodes required for the environment to be considered solved (i.e., average score over 100 episodes > max player score 0.5)

The recommend training settings are as follows:

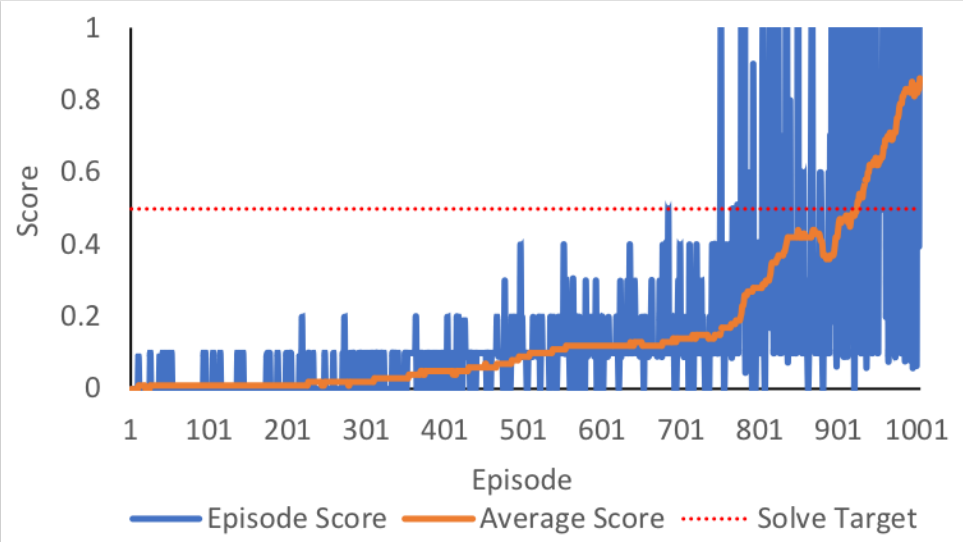
- num_episodes: **500**
- scores_average_window: **100**
- solved_score: I have been setting this to **.85** (rather than min solved score of 0.5) to achieve overall better agent game play performance.

Actor and Critic Neural Networks

Because the agents learn from vector date (not pixel data), the local Actor and Critic networks employed consisted of just 2 hidden, fully connected layers with 256 and 128 nodes, respectively ([256,256] and [512,256] also work, but can result in less stable and/or slower learning).

Training Performance

Using the above hyperparameter and training settings, the agents were able to "solve" the Tennis environment (i.e., reach of average max player score of >0.5 over 100 episodes) in less than 1000 episodes. The lowest recorded number of episodes required to solve the environment over 25 training runs was 764). A prototypical example of DDPG-Agent training performance is illustrated in the below figure. The best average max player score I was able to achieve in less than 1000 episodes was 0.87.



Prototypical Example of DDPG-Agent Training Performance for the Tennis Environment (score per episode)

Future Directions

In the future I plan to implement the following DDPG Extensions

- Test other hyperparameter settings
- Try implementing real MADDPG, using shared critic network.
- Try and implement MA-D4PG, PPO and A2C.
- I am currently also working on implementing prioritized replay.
- You could also make the task "more" competitive by changing the reward structure. That is, adding a negative reward for loosing (i.e., -1) and positive reward for winning (i.e., +1). This would require reducing the 'hit ball' reward to 0.01. Otherwise game play would always be cooperative. I have play around with this using Pong in Unity and it works well (see: https://github.com/AdventuresInUnityMLAgents/AiMULA-Getting-Started-Examples)