DDPG Reinforcement Learning using PyTorch and Unity ML-Agents

A simple example of how to implement vector based DDPG 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, *Reacher*, 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

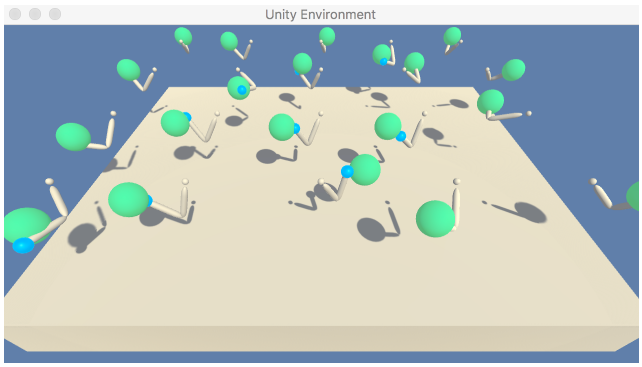
The files in the python/. directory are the ML-Agents toolkit files and dependencies required to run the Reacher environment. For more information about the Unity ML-Agents Toolkit visit: https://github.com/Unity-Technologies/ml-agents

Example Unity Environment – Reacher

The example uses a modified version of the Unity ML-Agents Reacher Example Environment. The environment includes In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible. The environment uses multiple unity agents to increase training time.

Multiagent Training:

The Reacher environment contains multiple unity agents to increase training time. The training agent collects observations and learns from the experiences of all of the unity agents simultaneously. The Reacher environment example employed here has 20 unity agents (i.e., 20 double-jointed arms).



Example of the Agents view of the Reacher Environment

State and Action Space

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

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 algorithm by adapting the example ***ddpg_agent pendulum*** and ***model.py*** code provided in the Udacity GitHub examples. The adapted and split out the *Replay_Buffer.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 various comments and trouble shotting tips by the Udacity DRNLD students on Slack were a great help when it in testing and choosing between various hyperparameters.

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 Reacher 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: **33** (is the non-optional state size employed by the Reacher agent)
- action_size: **4** (is the non-optional action size of the Reacher agent)
- replay_memory size: **1e5**
- batch_size: **128** (64 and 256 will also work)
- gamma: **0.99**
- tau: **1e-3**
- Actor learning_rate: **1e-4**
- Critic learning rate: **1e-4**
- Weight_Decay: **0.0**

Updated OUNoise Function

I also updated the Ornstein-Uhlenbeck noise process, to include a decayed sigma value (e.g., ***sigma=.15*** decays from sigma to ***sigma_min=.05*** by a factor of ***sigma_decay=.95***). This had only a minimal impact on learning time, but did result in a higher and more stable episode score after initial 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 score over 100 episodes required for the environment to be considered solved (i.e., average score over 100 episodes > 13)

The recommend training settings are as follows:

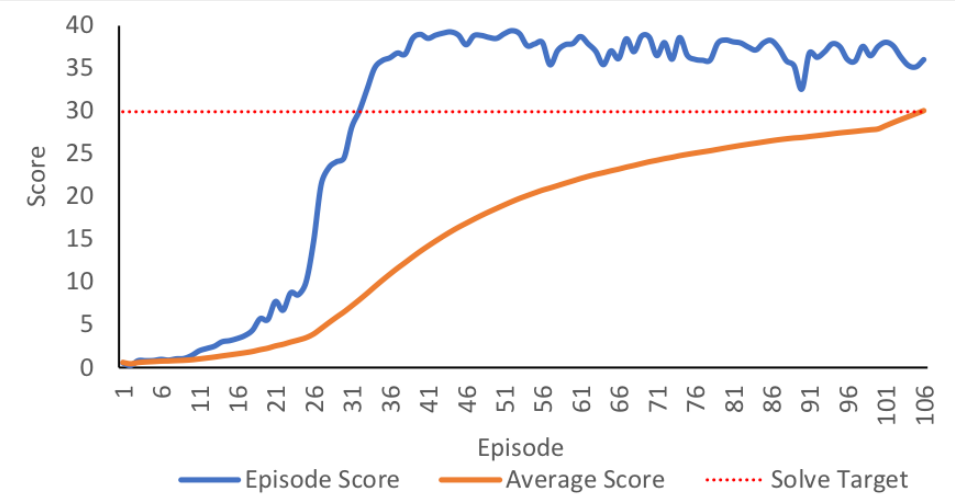
- num_episodes: **500**
- scores_average_window: **100**
- solved_score: **30** (agents easily reach 37+).

Actor and Critic Neural Networks

Because the agent learnings 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 ([128,128] and [256,256] also work, but can result in less stable and/or slower learning).

Training Performance

Using the above hyperparameter and training settings detailed above, the agent is able to "solve" the Reacher environment (i.e., reach of average score of >30 over 100 episodes) in less than 110 episodes. The lowest recorded number of episodes required to solve the environment over 20 training runs was 104 episodes). A prototypical example of DDPG-Agent training performance is illustrated in the below figure.



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

Future Directions

In the future I plan to implement the following DDPG Extensions

- Test other hyperparameter settings
- Try and implement D4PG.