A simple project to implement Multi Agent Deep Deterministic Policy Gradient (DDPG) network using Python, PyTorch and Unity ML-Agent's Tennis Environment

Understanding Environment

This project uses a modified version of Unity ML-Agents Tennis example environment. The environment. In this environment, two agents control rackets to bounce a ball over a net. The goal of each agent is to keep the ball in play.

Observation space and state space

The agent is trained from vector input data (not the raw pixels as input data). The observation space consists of 8 variables corresponding the position and the velocity of the ball and racket. Each agent receives its own, local observation. However, the agent observes a state with length of 24 dimensions. so, input to the agent would be a tensor/vector of 24 dimensions.

Action space

Since we are training an agent to take actions that can have continuous values, the action here is a vector of 2 values, corresponding to the movement toward (or away from) the net, and the jumping.

Reward

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.

Solving the Environment

The task is episodic, and in order to solve the environment, the 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 the agent. The yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- The yield a single score for each episode.

The environment is considered solved, when the average (over 100 consecutive episodes) of those scores is at least +0.5

Multiple Agents: Cooperation and competition

This environment is quite interesting compared to the single agent environments. It requires the training of two separate agents, and the agents need to collaborate under certain situations (like co-operate, don't let the ball hit the ground) and compete under certain situations (like compete and accumulate as many points as possible).

Need for Stationary environments

Just doing a sample extension of a single agent Reinforcement Learning by independently training the two agents done not work very well because the agents are independently updating their policies as learning progresses. And this causes the environment to appear non-stationary from the viewpoint of any one agent. While we can have non-stationary Markov processes, the convergence guarantees offered by many RL algorithms such as Q-learning requires stationary environments. While there are many different RL algorithms for multi-agent settings, for this project I have chosen to use the Multi Agent Deep Deterministic Policy Gradient (MADDPG) algorithm.

Multi Agent Deep Deterministic Policy Gradient (MADDPG) Algorithm

Since it's also a version of DDPG algorithm, it is also a model-free, off-policy actor-critic algorithm that combines Deep Q learning (DQN) and Deterministic policy gradient (DPG). As it is an off-policy algorithm, it uses two separate policies for the exploration and updates. It uses a stochastic behaviour policy for the exploration and deterministic policy for the target update.

We did learn the fact that the instability issue that can raise in Q Learning with deep neural network as the function approximator. To solve this, experience replay and target networks are being used.

In MADDPG

Critic Network

Each agent's critic is trained using the observation and actions from all the agents.

Actor Network

Each agent's actor network is trained using just its own local observations. This allows the agents to be efficient without requiring other agents' observations during inference (because the agent is only dependent on its own observations).

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:

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

  end for
end for

The pseudocode for Multi-Agent Deep Deterministic Policy Gradient for N agents from [Medium_article]

Implementation

Deep Deterministic Policy Gradient (Actor-Critic Network)

- The "Critic" estimates the value function. This could be the action-value (the Q value) or state-value (the V value)
- The "Actor" updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

And both the Critic and the Actor functions are parameterized with neural networks.

In the off-policy method, we use two policies called the behaviour policy and the target policy (both. As the name suggests, we behave (generate episodes) using the behaviour policy and we try to improve the other policy called the target policy.

As mentioned, the input/vector data is used to learn the policy by the agent. And the action is a continuous value. we will be using behaviour and target networks of same configuration.

Naming convention of behaviour and target networks in the scripts:

- *Behaviour network:* "local_actor" and "local_critic" names are being used to refer actor and critic networks respectively

- *Target network:* "target_actor" and "target_critic" names are being used to refer actor and critic target network respectively.

Actor Network

Input:

The input tensor/vector consists of 24 values (corresponding to position, rotation, velocity and angular velocity of the arm). Hence, the input layer of the Actor Network (both behaviour and target network) consists of 24 neurons.

Output:

The output tensor/vector consists of 2 values (corresponding to the torque applicable to two joints). And every entry in the action vector should be a number between -1 and +1. Hence, the output layer of the Actor Network (both behaviour and target network) consists of 2 neurons, and tanh activation function would be used to limit the output values to -1 and +1.

Neural Network Architecture:

- Input Layer: dimension (24, 256)
- 1st hidden layer: dimension (256, 128)
- 1st batch normalization layer: dimension (256)
- 1 output layer: dimension (128, 2)

```
Actor(
  (fc1): Linear(in_features=24, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=2, bias=True)
  (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

Critic Network

Input:

The critic networks take both the current state and the action (predicted by the actor network) as inputs and concatenates them. However, the concatenation operation is performed after the batch normalization operation and before second hidden layer. Since the input tensor/vector of state consists of 33 entries, the input layer of the critic network has 33 neurons.

Output:

The Critic network estimates the Q value of the current state and the action (predicted by the actor network) and hence the output layer of critic network has only 1 neuron.

Neural Network Architecture:

- 1st input layer: dimension (full_state_size, 256)
- 1st hidden layer: dimension (256+full_action_size, 128)
- 1st batch normalization layer: dimension (256)
- 2nd hidden layer/output layer: dimension (128, 1)

full_state_size: It is the total number of input dimensions observed by all the agents (in our case there are 2 agents). So, 24(input state) * 2(agents) = 48.

full_action_size: It is the total number of actions possible/taken by all the agents (in our case, there are 2 agents). So, 2(possible actions) * 2(agents) = 4. So. 256+4 = 260.

```
Critic(
  (fcs1): Linear(in_features=48, out_features=256, bias=True)
  (fc2): Linear(in_features=260, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
  (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

Hyperparameters:

```
LR_ACTOR = 1e-4                          # learning rate of the actor
LR_CRITIC = 3e-4                         # learning rate of the critic
WEIGHT_DECAY = 0                         # L2 weight decay
NOISE_REDUCTION_RATE = 0.99              # To add noise to output for exploration. However,
we reduce noise to balance exploration-exploitation.
EPISODES_BEFORE_TRAINING = 500           # This is used to reduce the noise, implemented as
follows.
NOISE_START=1.0                          # Initial noise (only exploration)
NOISE_END=0.1                            # Max limit for reduction (to maximize exploitation
```

Multi Agent Deep Deterministic Policy Gradient (MADDPG) Algorithm

The internal working of MADDPG has been implemented in maddpg.py module. So, basically it takes the following input parameters:

- State_size: Dimension of the input state vector/tensor observed by the agent (24).
- Action_size: Dimension of the output action vector/tensor (2).
- Num_agents: Total number of agents required to solve environment (2)
- Random_seed: To facilitate reproducibility of same results (100)

Hyperparameters:

```
Buffer_size: 1e5        # size of replay buffer
batch_size:   256       # minibatch size
gamma:         0.99     # discounting factor
tau:           1e-3     #for soft-update of target parameters
```

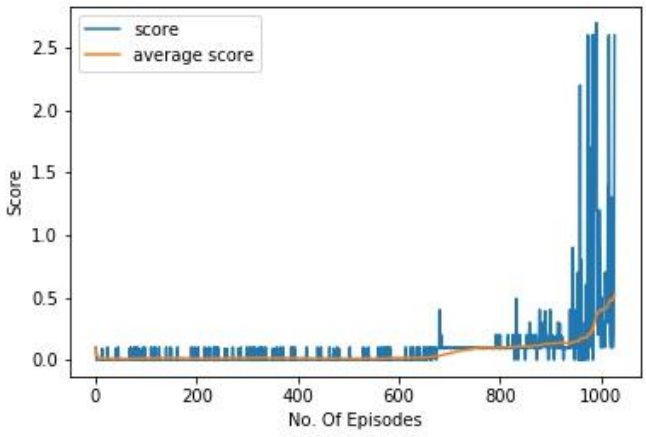
Modules to solve the Tennis environment

- *Ddpg_agent.py:* Source code for DDPG agent
- *Model.py:* Source code for Actor-Critic Network
- *Memory.py:* Source code for Replay buffer to store the experiences.
- *Maddpg.py:* Source code for Multi agent DDPG network/algorithm
- *Noise.py:* The base class for OUNoise process, object of this class will be added to the actor's output (so, dimension of 2) to facilitate the exploration.

- *Utils.py:* This script contains few functions to get the device (CPU/GPU) to train the model.
- *Train_agent.py:* This is the script to run/train 2 agents to solve the Tennis environment.
- *Test_agent.py:* This scrips loads the saved weights of actor-network of MADDPG, to test the agent in the Tennis Environment.

Training Phase and Rewards

MADDPG algorithm for 2 agents has been trained for over 2000 episodes with different hyperparameters setting. Finally, with the above mentioned hyperparameter setting, the 2 agents have been able to solve the environment.



MADDPG agent training performance for solving the Tennis environement (+0.5 rewards for over 100 consecutive episodes). The x-axis in the graph plots the total number of episodes and the y-axis in the graph tells us the average score per episode.

Future Work

- Implement the same using A3C or D4PG algorithm.

References

- Scaling Multi-Agent Reinforcement Learning – The Berkeley Artificial Intelligence Research Blog
- Paper list of multi-agent reinforcement learning (MARL)
- Multi-Agent Actor-Critic
- MADDPG-Medium Article