A Project to implement vector based DDPG using PyTorch and Unity ML-Agentenvironment

**Environment**

Understanding Environment

Action Space

State Space and Rewards

**Algorithm**

Problem with Policy gradient methods. (Reinforce)

Intuition of DDPG or Policy based methods.

Difference between ON Policy and OFF Policy.

DDPG

Pseudocode

**Implementation:**

GitHub details.

Installation.

Actor network

Critic Network

Local and Target Networks.

Soft Update.

**Results**

Graph

**Future Work**

**References:**

A simple project to show how to implement an actor-critic policy gradient network agent using Python, Pytorch and Unity ML-Agent’s environment.

Environment

This project uses a modified version of Unity ML-Agents Reacher example environment. The environment includes a double-joined arm that can move to target locations. The goal of the agent is to maintain its position at the target location for as many time steps as possible. For the version of the Reacher environment being used in this project, it is considered that the agent has solved the environment when the average score over the last 100 episodes >= +30.0

State Space:

The agent is trained from the vector input data (not the raw pixels as input data). The state space consists of 33 variables corresponding to position, rotation, velocity and angular velocities of the arm.

Action Space:

Since we are training an agent to take actions (that are continuous values), the action here is vector of 4 numbers, corresponding to the torque applicable to two joints. And, every entry in the action vector(consisting of 4 entries) should be a number between -1 and +1.

Reward:

A reward of +0.1 is provided for each step that the agent’s hand is in the goal location.

Training Environment – Versions

For this project, two separate version of the Unity Environment are provided:

1. The first version contains a single agent.
2. The second version contains 20 identical agents, each with its own copy of the environment.
   1. The second version is useful for algorithms like, PPO, A3C, D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

Solving the environment

Option 1: Solve the environment with single agent.

1. The task is episodic, and in order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes.
2. You may refer to train\_agent.py module. 1 agent is assigned with 1 brain, which encapsulates the decision-making process.

Option 2: Solve the environment with 20 identical agents.

1. The barrier for solving the second version of the environment is slightly different, to take into account the presence of many agents. In particular, the agents must get an average score of +30 over 100 consecutive episodes, and over all agents). Specifically, After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. We then take the average of those 20 scores.
2. This yields an average score for each episode (where the average is over all 20 agents).
3. You may refer to train\_agent\_dist.py module. Due to hardware limitations and training time constraint, I have used the strategy to assign 1 brain to 20 agents.

Algorithm

One of the disadvantages of the value-based method is it is suitable only for discrete environments (environments with discrete action space), and we cannot apply value-based methods in continuous environments (environments with a continuous action space). Most of the real-world problems have continuous action space, say, a self-driving car, or a robot learning to walk and more.

Policy gradient method – REINFORCE Intuition:

Policy gradient is one of the most popular algorithms in deep reinforcement learning. As we have learned, policy gradient is a policy-based method by which we can find optimal policy without computing the Q function. It finds the optimal policy by directly parameterizing the policy using some parameter (**theta**)

**The policy gradient method uses a stochastic policy, we select an action based on the probability distribution over the action spaces. Say we have a stochastic policy (PI), then it gives the probability of taking an action a given the state “S”. It can be denoted by PI(a/s). In the policy gradient method, we use a parameterized policy, so we can denote our policy, where theta indicates that our policy is parameterized, where theta is the parameter of the neural network.**

**Say we have a neural network with a parameter theta. First, we feed the state of the environment as an input of the network and it will output the probability of all the actions that can be performed in the state. That is, it outputs a probability distribution over an action space. We have learned that the policy gradient, we use a stochastic policy. So, the stochastic policy selects an action based on the probability distribution given by the neural network. In this way, we can directly compute the policy without the Q function.**

**Paste the gradient equations, and also do mention that REINFORCE is Monty Carlo version**

https://medium.com/intro-to-artificial-intelligence/deep-deterministic-policy-gradient-ddpg-an-off-policy-reinforcement-learning-algorithm-38ca8698131b

Problem with Policy gradient method, REINFORCE:

1. Monty Carlo type, and hence the task has to be episodic.

Deterministic Policy Gradient:

Traditionally, policy gradient algorithms are being used with stochastic policy function. That means policy function is represented as a distributions over actions. For a given state, there will be probability distribution for each action in the action space. In DPG, instead of stochastic policy, pi, deterministic policy mu(./s) is followed. For a given state, s, there will be deterministic decision: a = mu(s) instead of distribution over actions.

The grad objection function of the stochastic Policy gradient algorithm can be written as below:

Q-learning

Q Learning is a value-based off-policy temporal difference (TD) reinforcement learning. Off-policy means as agent follows a behaviour policy for choosing the action to reach the next state, s\_t+1 from state s\_t. From s\_t+1, it uses a policy pi, that is different from behaviour policy. In Q-learning, we take absolute greedy action as policy pi, from the next state s\_t+1.

As we discussed in the action-value function, the above equation indicates how we compute the Q-value for an action a starting from state s in Q learning. It is the sum of immediate reward using a behaviour policy (eps-soft, etc). From state s\_t+1, it takes an absolute greedy action(choose the action that has the maximum Q value over the other actions).

Actor-critic

In simple term, Actor-Critic is a Temporal Difference(TD) version of Policy gradient. It has two networks: Actor and Critic. The actor decides which action should be taken and critic inform the actor how good was the action and how it should adjust it. The learning of the actor is based on policy gradient approach. In comparison, critics evaluate the action produced by the actor by computing the value function.

Deep Deterministic Policy Gradient (DDPG):

DDPG is a model-free off-policy actor-critic algorithm that combines Deep Q learning(DQN) and DPG. Original DQN works in a discrete action space, DPG extends it to the continuous action space while learning a deterministic policy.

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.

DDPG is an actor-critic algorithm; it has two networks: actor and the critic. Technically, the actor produces the action to explore. During the update process of the actor, TD error from a critic is used. The critic network gets updated based on the TD error similar to Q-learning update rule.

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

Github Code Part: