MADDPG for Unity Tennis Multi-player Environment

1. Introduction

Reinforcement learning is rooted in psychological and neurological perspective of animal behavior. In many cases the environment can be too complex that the number of states is not enumerable to a finite set. Animals and humans learn the complex environment around them through a harmonious combination of reinforcement learning and hierarchical sensory processing systems. DQN was one of the ground-breaking use of Deep neural networks in reinforcement learning to learn Atari games. The Deep Q-Network agent surpassed all previous algorithms and achieved levels comparable to professional human players. Most of the success in RL has been in single agent environments where the modeling and predictions the behavior of other actors in the environment is not necessary. Naïve policy gradient methods perform poorly in simple multi-agent settings.

There are many applications of multiplayer games including multi-robot control, discovery of communication and language, analysis of social dilemmas etc.

This implementation to solve the Tennis Multi-player Unity game environment is based on the MADDPG algorithm, The software agents (MADDPG) interacts with the environment thru a sequence of observations (24) actions (4) and rewards. Each observes the actions of other agents also to model its opponents policy.

The goal of the agent is to select an action that maximizes the cumulative future reward. This implementation uses an MLP (Multi-layer perceptron) network to approximate the optimal policy as well as the Q-value functions of both agents. function fig 1.

2. Actor-Critic methods DDPG

2.1 Actor-Critic Methods:

Methods that learn approximations to both policy and value function are often called actor-critic methods where ‘actor’ is a reference to the learned policy and ‘critic’ refers to the learned value function, usually a state-value function fig. 1 (2) . The actor produces an action based on the state of the environment and critic produces a TD error signal based on state and the reward. If the action value function is estimated by the critic, it would need the output of the actor too. Both actor and critic can be represented by Neural Networks.

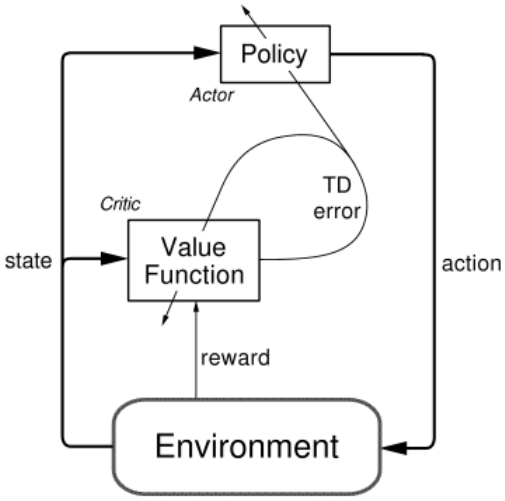

Fig. 1 Actor-Critic Methods (3)

2.2. DDPG Algorithm:

DDPG is a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces. It is based on the DPG algorithm (3). It also uses some of the learning tricks from DQN including replay buffer and local and target networks, one for each actor and critic.

Applying Q-learning to continuous states is not straight forward as finding the greedy policy requires optimization of a_t at every timestep and is too time consuming. Instead DPG algorithm maintains a parametrized actor function $\mu(s|\theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action. The critic $Q(s,a)$ is learned using the Bellman equation as in Q-learning. DPG algorithm by Silver et al (5) showed that the following is the policy gradient, the gradient of the policy's performance. (1)

$$\nabla_{\theta^{\mu}} J \approx E_{s_t \sim \rho^{\beta}} [\nabla_a Q(s,a|\theta^Q)|_{s=s_t\, a=\mu(s_t)} \nabla_{\theta_{\mu}} \mu(s|\theta^{\mu})|_{s=s_t} \quad \text{-------> (3)}$$

It also uses ideas from DQN to make the learning algorithm more stable.

1) ***Replay Buffer***
   Like, DQN, DDPG uses experience replay to make the learning algorithm more stable. The agent's experience at each time step t is stored as a tuple e_i = (s, a, s', r) and these tuples are added to a buffer B = { $e_1, e_2, \ldots \ldots \ldots e_n$} . While learning (in the inner loop) data is sampled in mini batches from this buffer to train the networks. The samples are picked at random. Correlation between consecutive samples affects the stability of the learning process. Randomizing reduces the correlation between consecutive samples thus reducing the variance of updates.

2) *Two Networks*

Like DQN, DDPG also uses a target network and local network to improve the stability of the algorithm. This is done as follows. The network is trained with at target Q network to give consistent targets during temporal difference backups. In DDPG instead of directly copying weights from target, a soft target update is done for the actor-critic.

A copy of the actor-critic networks, $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively, that are used for calculating target values. The weights of these target networks are then updated as below.

$$\theta' = \tau\theta + (1-\tau)\theta' \ with \ \tau < 1 \quad \text{----> (4)}$$

This makes the algorithm more stable compared to online Q-learning as the target values are constrained to change slowly. There is both a target actor: Q' and critic: μ' to train the critic without divergence. This slows the learning process while make the algorithm stable.

3) Batch Normalization

When learning from low dimensional feature vector observations, the different components of the observation may have different physical units (for example, positions, velocities etc.) and the ranges may wary across environments. This makes it difficult to identify hyper-parameters the learn effectively across environments with different scales and state values. To address this issues DDPG uses batch normalization.

4) Exploration

To facilitate exploration in a continuous action space, noise sampled from a noise process N is added to the exploration policy μ'

$$\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + N \quad \text{--------> (5)}$$

Orinstein-Uhlenbeck process is used to generate the noise.

3. MADDPG

Success in multi-agents training will drive systems that productively interact with humans and each other and learn on their own. Traditional RL approaches like Q-Learning & policy gradients are not well suited for multi-agent environments and they do not model the behavior of other agents in the environment. Each agent's policy is changing as training progresses, and the env becomes non-stationary from the perspective of any individual agent. This presents learning stability challenges, and prevents the straightforward use of past experience replay, which is crucial for stabilizing deep Q-learning.

MADDPG uses centralized training with decentralized execution allowing policies to use extra information to ease training, so long as this info is not used while testing/deployed. It is an extension to the actor-critic policy gradient methods, where the critic is augmented with additional information about other agents in the env. In the testing phase, only the actor has access to information and does not get any augmented information about other agents.

Centralized critic in MADDPG uses the decision-making policies of other agents. This helps them learn approximate models of other agents online and effectively use them in their own policy learning procedure. MADDPG also uses an ensemble of policies to stabilize the learning process.

Methods.

Let θ = {θ1, ..., θn }, be policy parameters on N agents in a multi-player game and let π = {π1, ..., πN } be the set of all agent policies. Then the gradient of the expected return for agent i $J(\theta_i) = E[R_i]$ can be defined as

$$\nabla_{\theta_i} J(\theta_i) = E_{s\sim p^{\mu},\, a_i\sim\pi_i} [\nabla_{\theta_i} \log \pi_i(a_i|o_i)\ Q_i^{\pi}(x, a_1, \ldots . a_N)]$$

Where

$a_1, \ldots a_N$ is -----> the actions of the N agents

$Q_i^{\pi}(x, a_1, \ldots . a_N)$ ----> a centralized action-value function that takes as inputs actions of all the N agents, in addition to the state information for the agent and outputs the Q value for agent i.

x ---> could consists of observations of all the agents $x = (o_1 \ldots o_N)$.

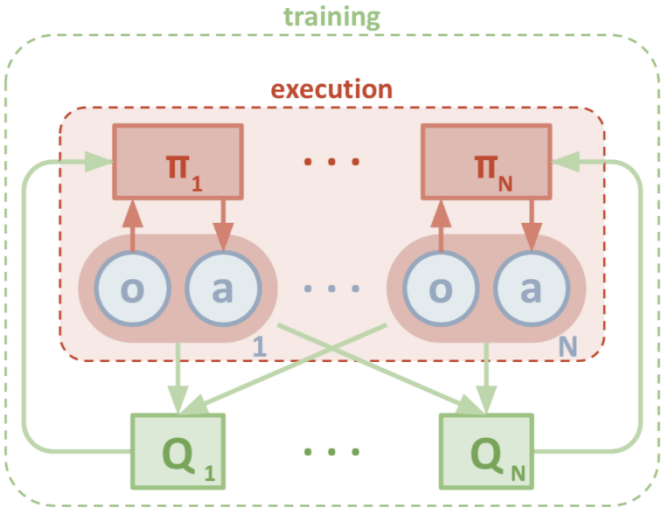

Fig 2. Multi-agent decentralized actor, centralized critic approach (1)

Each agent can have different reward structures as Q_i^{π} is learned separately. For N continuous policies $(\mu_1, \ldots \mu_N)$ with delayed parameters $(\theta_1, \ldots \theta_N)$ this can be written as

$$L(\theta_i) = E_{x,a,r,x'}[(Q_i^{\mu}(x, a_1, \ldots a_N) - y)^2$$

$$y = r_i + \gamma Q_i^{\mu'}(x, a_1, \ldots a_N) \mid_{a'_j = \mu'_j(o_j)}$$

3.2. The Algorithm

The Algorithm from the original DDPG paper is as below.

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 $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      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

4. Architecture

The Architecture of the MADDPG implementation for Unity Environment: Tennis has five main components

1) Agent 2) MADDPG Agent 3) DDPG Actor-Critic Networks 4) Training 5) Saved Model

4.1 Agent

The Agent has five components. MADDPG Agent that holds all the agents (players). MADDPG holds a replay buffer that stores all state and action data for all the agents. Each agent is DDPG. For each agent

four networks are trained: local actor and critic networks to train and the other, target actor and critic networks to verify. Data from the replay buffer is sampled to train the networks.

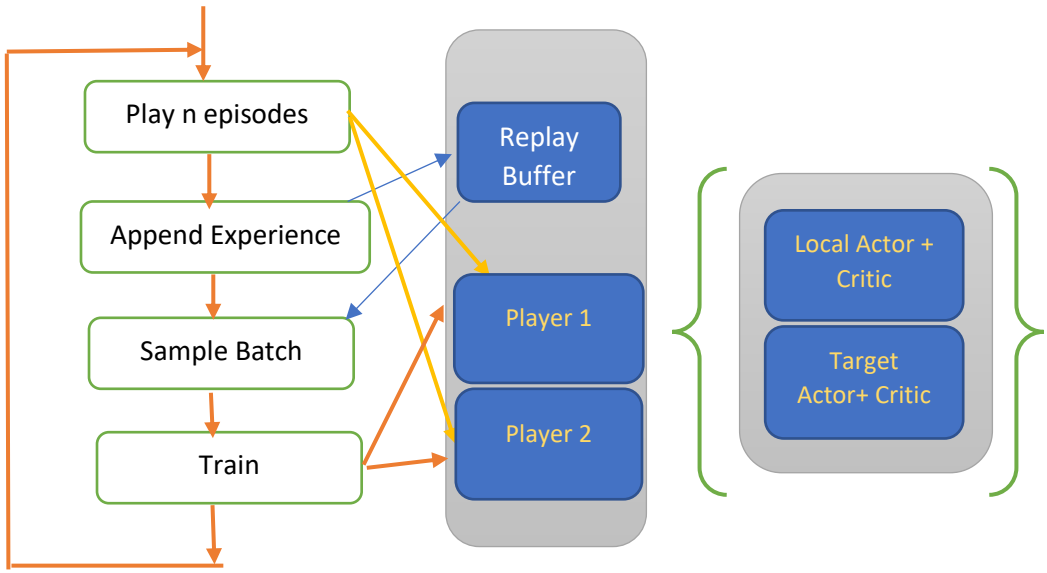

Fig. 3. The Software Architecture

4.2. The Actor-Critic Networks

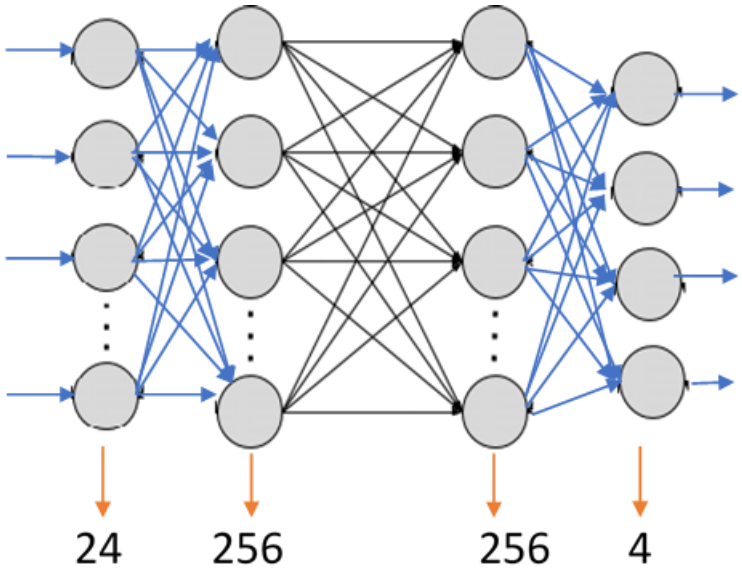

Fig. 5. Actor MLP Network

The Unity Tennis environment has 24 observations for each agent and 2 possible actions (see Environment below). The 24 observations form the input of a MLP network with 2 hidden layers and 4 outputs. The hidden layers have 256 neurons each. Actor and Critic networks have 24 input neurons. Actor networks have 4 outputs while Critic has 1 output. See Fig 4. . The critic networks concatenate the actions of the actor to the second hidden layer as well.

4.3 Training & Testing

The Agent is trained for 9000 episodes. The first 200o episode does not use the network but generates random actions. So the actual network training starts only at 2000+ episodes. With this network and training method, an average score of 0.5+ over 100 episodes was achieved within 7000 episodes. The training can be run by executing

- python Tennis.py.

The parameters used for the model are

Episode count : Maximum : 9000

Learning Rate & TAU :

```
Learning rate Actors : 1e-04
Learning rate Critics: 1e-04

TAU for soft updates : 1e-3

BATCH SIZE : 128
```

Other Training Optimizations used

1) Batch normalization is used in both input to MLP and for the output of first hidden layer.
2) Leaky relu is used in hidden layer outputs.

4.4 Saved Model execution.

The model is saved once a score of 0.5 is reached. This may not consistently perform and show a score of 0.5+ each time. To run saved model

- python run_model.py

The code runs the game 3 times using saved model

5. Environment

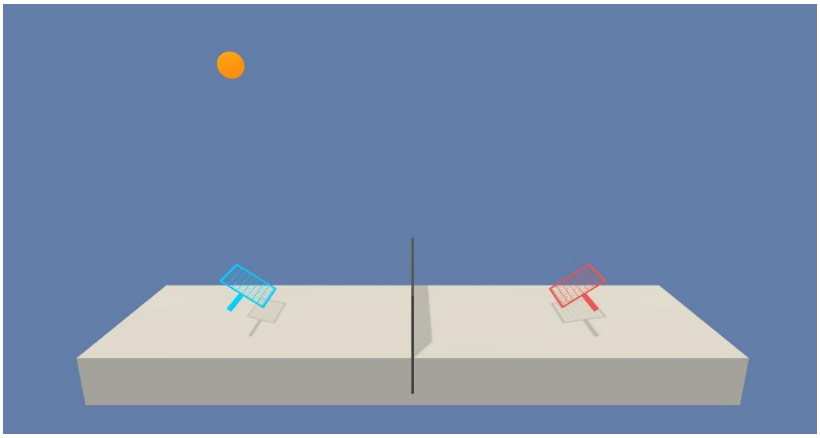

Fig. 6 Unity Tennis

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.

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. The generated actions should be two continuous values, corresponding to movement to & fro from the net, and jumping.

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 each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- This yields a single score for each episode.

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

6. Code

There are 5 files

1. Tennis.py

This file has the code to train the networks. To run training execute this script.

2. maddpg_agent.py

The definition of MADDPG class that holds 2 agents for Unit Tennis game environment.

2. ddpg_agent.py

The Agent and replay buffer is defined in this file. The Agent class instantiates four networks: two actors and two critics. Replay buffer is used to store episodes of game play. Data is randomly retrieved from the replay buffer to train the networks.

3. model.py

Model.py has the MLP definition: both forward and backward pass of actor and critic networks. The critic networks add the actions of the actor to the second layer of the MLP during forward pass.

4. run_model.py

run_model.py : if executed run the saved model. Make sure the Tennis application path is valid before running this.

7. References

1. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments Ryan Lowe et.al
2. Continuous Control with Deep Reinforcement Learning.
3. Reinforcement Learning An Introduction Richard S. Sutton & Andrew G. Barto
4. Deep reinforcement Learning hand-On. Maxim Lapan.
5. Deep Deterministic Policy Gradients in Tensorflow. Patrick Emami
6. Deterministic Policy Gradient Algorithms. Silver et. al