Report

Project 3: Collaboration and Competition

Benji Alwis

Introduction

Being able to scale up multi-agent reinforcement learning algorithms is an important step towards achieving human-robot interaction at scale. However traditional RL algorithms such as Q-learning and policy gradient methods are ill-suited for multi-agent learning. One key challenge is that when the policies of the agents are changed during training, the environment becomes non-stationary from the viewpoint of a single agent which makes the learning process unstable. This prevents the use of replay buffers that is important requirement in making Q-learning stable. On the other hand, policy gradient suffer from high variance in multi-agent environments.

The Multi-Agent Deep Deterministic Policy Gradient (MADDPG) method, adopts centralised learning and decentralised execution by using extra information on other agents to ease training but do not use them at execution time. This is illustrated in figure 1. As shown in green, during training, each agent has access to the policies of the other agent. However, during execution (shown in red), each agent acts on its own.

It is hard to use a similar approach in Q-learning since it generally does not allow different information at training and test phases. Simple extension of actor critic policy gradient networks used in MADDPG allows this. It achieves this by augmenting the critic with extra information about policies of other agents. After training, only the local actors are used in a decentralised manner.

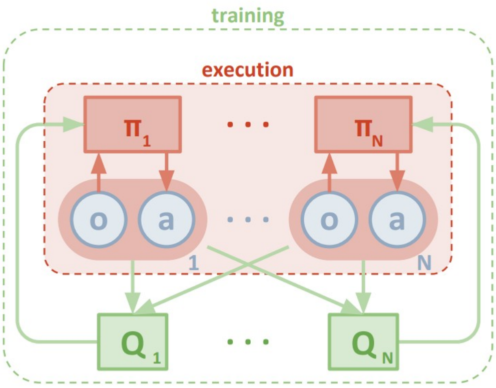
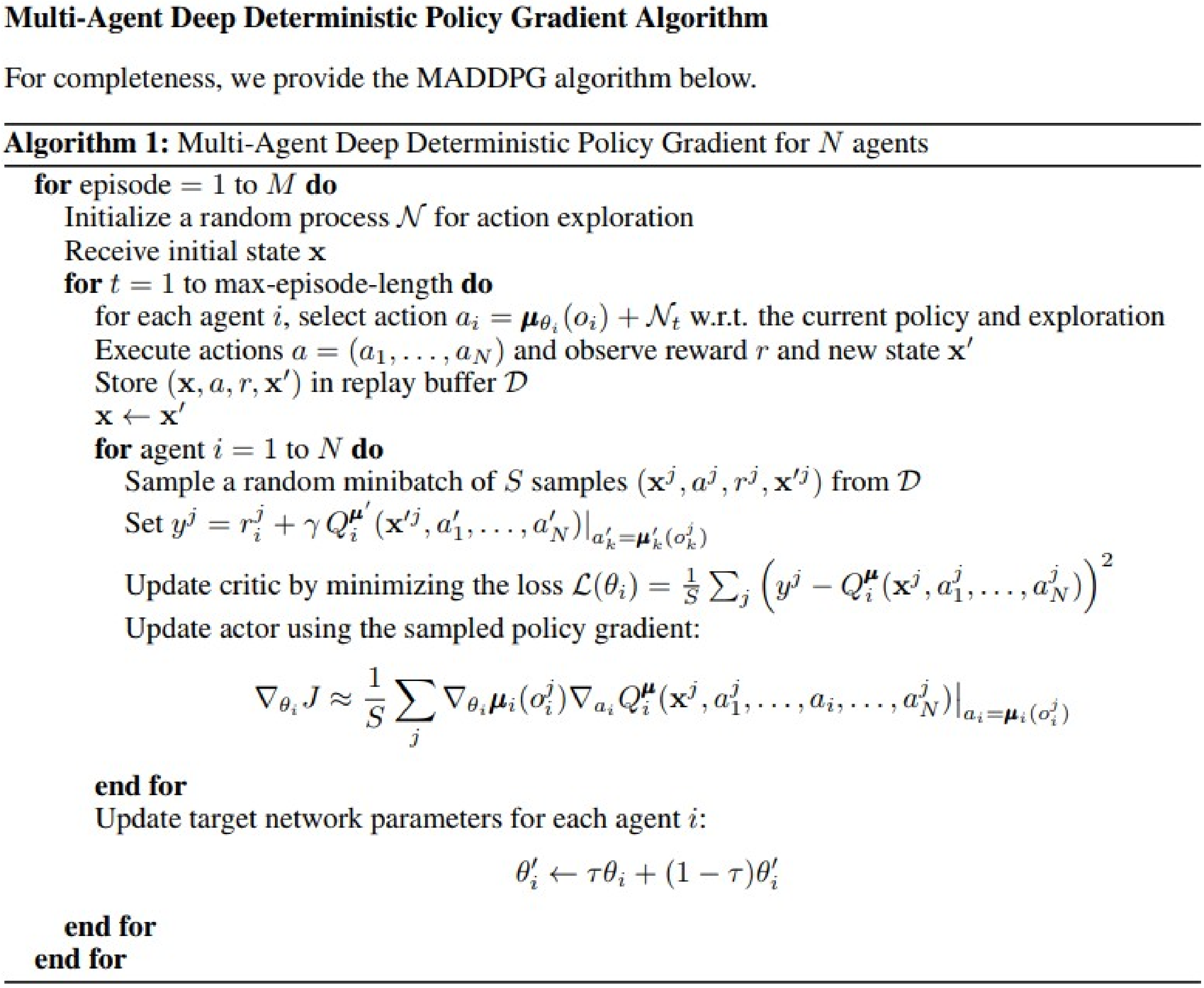


Figure 1 – Overview of the decentralized actor, centralized critic approach used in MADDPG (taken from the original paper).



Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm 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:

        $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)|_{a_i=\boldsymbol{\mu}_i(o_i^j)}$

    end for
    Update target network parameters for each agent $i$:

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

  end for
end for
```

Figure 2 – MADDPG algorithm (taken from the original paper).

Implementation

In this project, we utilise a training agent within a simulated 3D environment. The agent is in charge of two table tennis rackets and their main goal is to act collaboratively and maintain the ball in play for as long as possible, until a certain score is reached.

The code base used in this project was derived from the "DDPG pipedal" tutorial from the [Deep Reinforcement Learning Nanodegree] (https://github.com/udacity/deep-reinforcement-learning/blob/master/p3_collab-compet/Tennis.ipynb). It has been written in Python 3.6 with the use of PyTorch 0.4.0 framework.

The project consist of following files.

`model.py` : Implements the Actor and the Critic classes. Each of these classes implement a target and local network.

__init__() - this is the initialisation function where a fully connected neural network is defined. Actor class implements a network with three hidden layer and a single output node to calculate the mapping between the state and the action. Critic class implements a network of two hiddden layers. It calculates the state action value for a given (state,action) pair.

forward(state) - this function maps states to actions for an actor network and Q(State, Action) for the critic network.

‘ddpg_agent.py`: Basic implementation of a deep determinstic policy gradient (DDPG) agent.

__init__ - Initialisation function in which actor and critic networks are setup.

learn – Implements DDPG learning for the agent

`multi_agent.py`: Implements the multi agent DDPG alorithm. It instanciates a set of DDPG agents and provide the lerarning capability.

__init__() - It instantiates a set of agents.

act() - it returns action to perform for each agent based on policy,

step() - it saves experience in replay memory, and takes a random sample from buffer to learn,

maddpg_learn() - this function initiates the multi agent learning. It supplies a sample taken from the replay buffer and suuply the relevant information to the DDPG agents to perform learning.

- `TennisProject.ipynb` : This Jupyter notebooks allows to instantiate and train both agents.

solved_checkpoint_actor.pth and solved_checkpoint_critic.pth – these files contain the saved DNN’s weights.

Experiments

Experimental Goal

In this experiment, the task assigned for the two agents is to control rackets to keep a ball bouncing 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 -0.1. To reach the goal, the agents should achieve an average score of +0.5 over 100 consecutive episodes. The score for each agent is calculated by accumulating the maximum reward received by an agent for each episode.

Environment

The environment provided by Udacity is based on [Unity ML-agents](https://github.com/Unity-Technologies/ml-agents). It is a two-player game in which agents control rackets to bounce ball over a net. We use the Tennis agent in this environment. Each agent (connected to a single brain called Tennis Brain) has its own, local observation space that consists of 8 variables. They correspond to the position and velocity of the ball and racket. In the Udacity provided environment, 3 observations are stacked (8 *3 = 24 variables). The two continuous actions correspond to movement towards the net, and the jump. Each agent independently takes an action and receives a reward of +0.1 if the ball was hit over net. It receives a reward of -0.1 if it lets ball hit their ground, or hit it out of bounds.

Hyper Parameters

GAMMA = 0.995 # discount factor during reinforcement learning

TAU = 1e-3 # for soft update of target parameters

LR_ACTOR = 1e-4 # learning rate of the actor

LR_CRITIC = 1e-3 # learning rate of the critic

WEIGHT_DECAY = 0. # L2 weight decay

MAX_LAYERS = 4 # maximum number of hidden layers

BATCH_SIZE = 200 # Batch size for replay buffer

BUFFER_SIZE = int(1e5) # replay buffer size

HIDDEN_LAYERS = [400,300,300] # width of the hidden layers

LEAK = 0.1 # leak factor in LEAK_RELU

BACH_NORM = 0 # batch normalisation

APPROACH = 0 # layer selection on where batch normalisation is done

Results

After running many rounds of experiments, I observed that the initial learning phase (roughly about 1000 episodes) is very slow in improving the score. Then it starts improving gradually before making rapid progress after around 2500 episodes. The environment is considered solved when the multi agent setup achieves an average maximum score of +0.5 over the last 100 episodes. I kept all scores in a list and used a sliding window to calculate this. The following figure shows results from one of the trials. The agent solved the environment in this trial in 2796 episodes. This figure was roughly consistent over 30 trials (between 2600 to 2900 episodes).

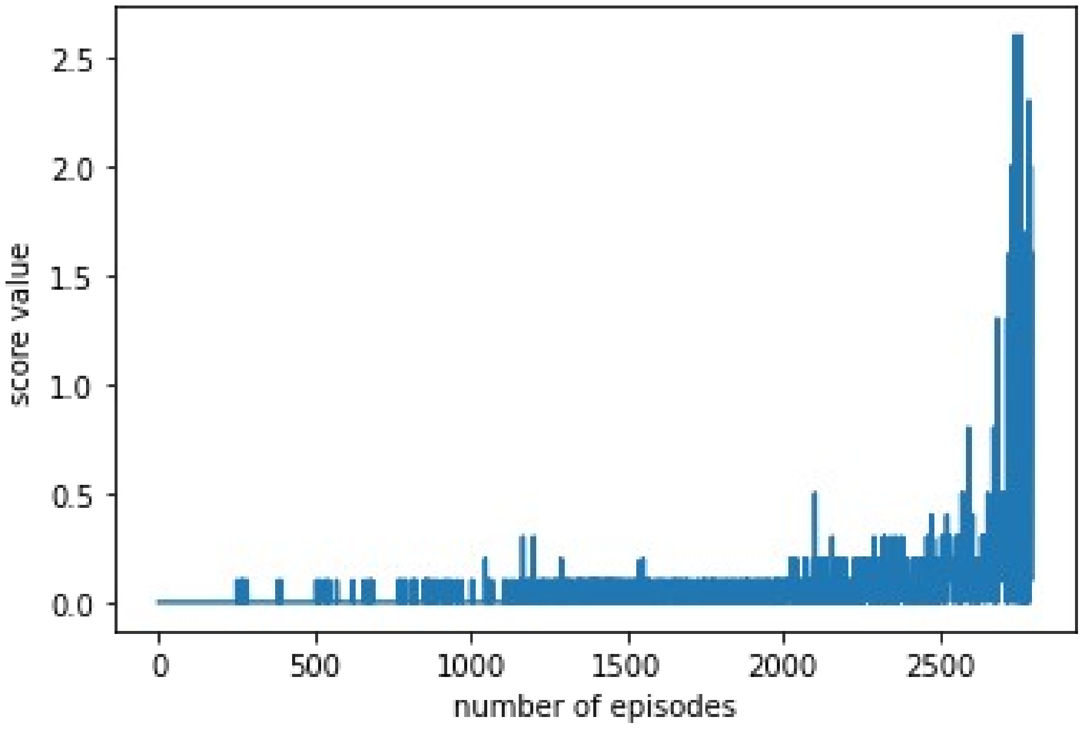


Figure 3 – Scores vs. episodes results from the experiment.

Next Steps

One issue is that DDPG is that it tends to overestimate Q-values. The algorithm tends to become brittle particularly due to its sensitivity for hypeparameters. This can cause “policy breaking”.

Twin Delayed DDPG (TD3) aims to overcome this limitation by using Clipped Double-Q Learning, “Delayed” Policy Updates and Target Policy Smoothing. TD3 aims to learn two Q values instead of one. It updates the policy and target networks less frequently (hence, “delayed updates”). TD3 adds noise in order to minimise adding errors from Q values calculation.

In my future work, I aim to try this extension.