Report

May 29, 2020

1 Collaboration and Competition

1.0.1 Introduction

Unity ML-agent library has a lot of simulation evnrionments to test intelligent agent models. This notebook solves one of the envrionnment which simulates a simple **Tennis** game. The task is to move 2 rackets/players appropriately to keep balls within the tennis court. Each time the ball crosses the tennis net towards the opponent side, the player gets a reward of +0.1, and if the ball hits the ground or moves outside of the court, the reward is -0.01. Therefore, a player should play well to pass a ball towards the other player to get a chance of high score. The code uses a Deep Deterministic Policy Gradient model (DDPG) to learn a behavior to gain as many scores as possible. Although there are 2 agents, their situation is symmetric, so DDPG can be applied to both agents simultaneously. All the code is self contained in a notebook for a potential modification of any logic. Each component is separated to its own cell, so it can be easily written to a separate file using "%writefile" if necessary

---

1.0.2 Environment

Requirements

- Linux based OS
- Python (dependency is listed in requirements.txt)
- Linux Tennis environment

The environment spec

- Number of agent: 2
- State space: 24 dimensional vector (for each agent)
- Action space: 2 numbers of range [-1, 1]
- Reward: +0.1 (If the ball crooses the net towards the opponent side), -0.01 (If the ball hits the ground or moves outside of the net)

1.0.3 Implementation

ActorModel

- **Description**
  - Standard fully-connected NN with a Batchnorm layer and ReLU activation. This model is used to model the deterministic policy, so it outputs 2-dimensional continous action vectors.
- **Architecture**
  - 3 Hidden layers (256, 256, 64 nodes)
  - 1 Output layer (2 nodes)
  - Each hidden layer uses ReLU and Batchnorm1d
  - Final output layer is a single linear layer with tanh activation to conform to the action value range
- **Discussion**
  - The initial hidden layer expanded the number of nodes for exploring rich feature space
  - ReLU and Batchnorm layer are used for stable training
  - Final output layer is combined with tanh to generate action of a correct range
  - 2 Hidden layers couldn't simulate a complex playing behavior, so it's increased to 3 layers
  - PyTorch 0.4 linear layer's initialization was too simple, so it's changed for better optimization.

CriticModel

- **Description**
  - A fully-connected NN which combines a state and action to generate Q value for a given state, action pair.
- **Architecture**
  - 3 Hidden layers (512, 256, 128 nodes)
  - 1 Output layer (1 node)
  - Each hidden layer uses ReLU except for action feature embedding
  - Final output layer is a single linear layer without any activation function
- **Discussion**
  - To deal with different characteristics of state and action, they're embedded to the first hidden layer separately
    - The first hidden layer 512 nodes are divided according to the ratio of state and action dimensions (24 to 2)
  - The initial hidden layer expanded the number of nodes for exploring rich feature space
  - Architecture hyperparmeters are decided with experiments
  - Final output layer directly generates an action value

Buffer

- **Description**
  - Replay buffer used in a DDPG algorithm. It is a standard deque with a sampling function
- **Interface**
  - Initialization
    - sz: Replay buffer size
    - pf: Probability function for sampling
  - put(obj): Stored obj in a replay buffer

  - sample(n): Sample n objects from a replay buffer
  - update_pf(pf): Update the probability function pf
- **Discussion**
  - Each agent's experience is stored without any differentiation because agent's situation is symmetric
  - pf can be used for prioritized experience replay
  - It was used to explore the potential benefit of the skew in experience sampling, but was default to uniform sampling

Training

- **Description**
  - Standard DDPG training algorithm
  - Both agents share a same actor and critic network because the learning situation is the same for both agents
- **Detail**
  1. Initialize the models(2 actors, 2 critics) and hyperparameters
  2. The current-actor-model (agent) interacts with the Tennis Unity environment to collect tuples (state, action, reward, next-state)
     1. The actor model decides actions for both agents simultaneously
     2. Exploration in continuous space is implemented using a Gaussian noise. The noise's standard deviation is controlled by an epsilon parameter
     3. Due to the probabilistic nature of the noise, the action is clipped to range [-1, 1]
  3. The collected tuples (experiences) are stored in a replay buffer
  4. The experience is randomly batch-sampled every 20 steps("train_step") and used to train the NN model
     1. There is no agent differentiation on the stored experience because red-player's experience can help blue-player in an exact same way.
     2. The training procedure follows a standard DDPG algorithm
        1. Current-critic-model is trained with TD-update using Target-actor/critic-models
        2. Current-actor model is trained with Q value using Current-critic-model
        3. Target-actor/critic-models are soft-updated with current models using a small mixture ratio(tau)
  5. 2-4 are repeated until the average score(return) reaches the objective score (13)
- **Hyperparameters**

name	value	description
n_episodes	5000	Max number of episodes to explore
max_t	3000	Max number of time steps of each episode
bn	256	Training single batch size
gamma	0.99	Return discounting factor
eps, eps_decay	0.25, 0.995	Parameter to control noise standard deviation (for exploration)

name	value	description
tau, tau_decay, tau_min	0.001, 0.9999, 0.001	Target network soft-update parameters
train_step	20	Frequency of the training
train_cnt	10	Number of gradient-updates in a single training procedure

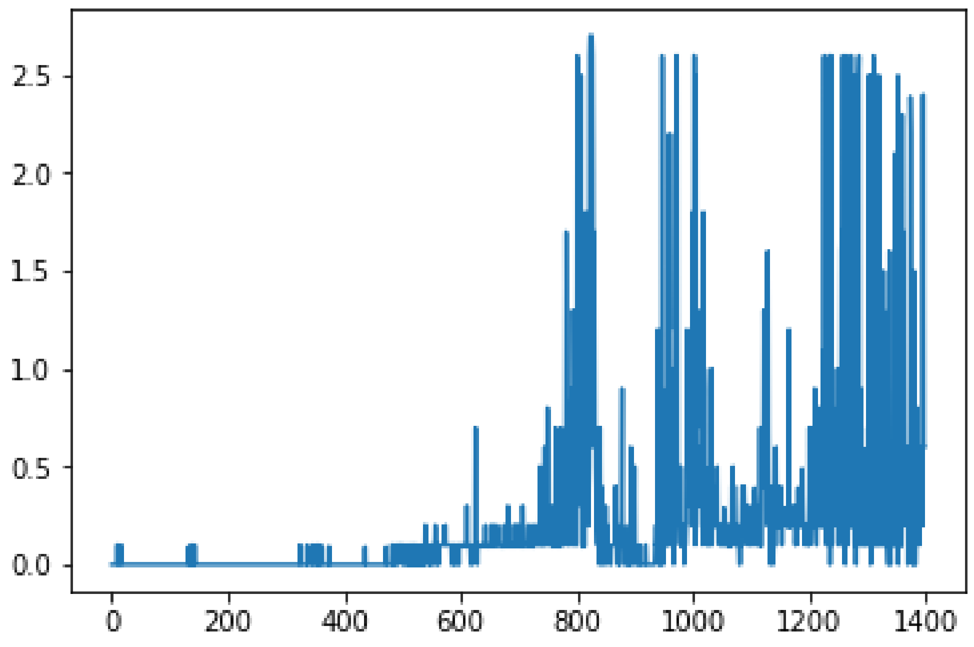
1.0.4 Results

The following 2 charts show the winner's score (max of 2 agents) in each episode and the 100-episode average of winner-scores

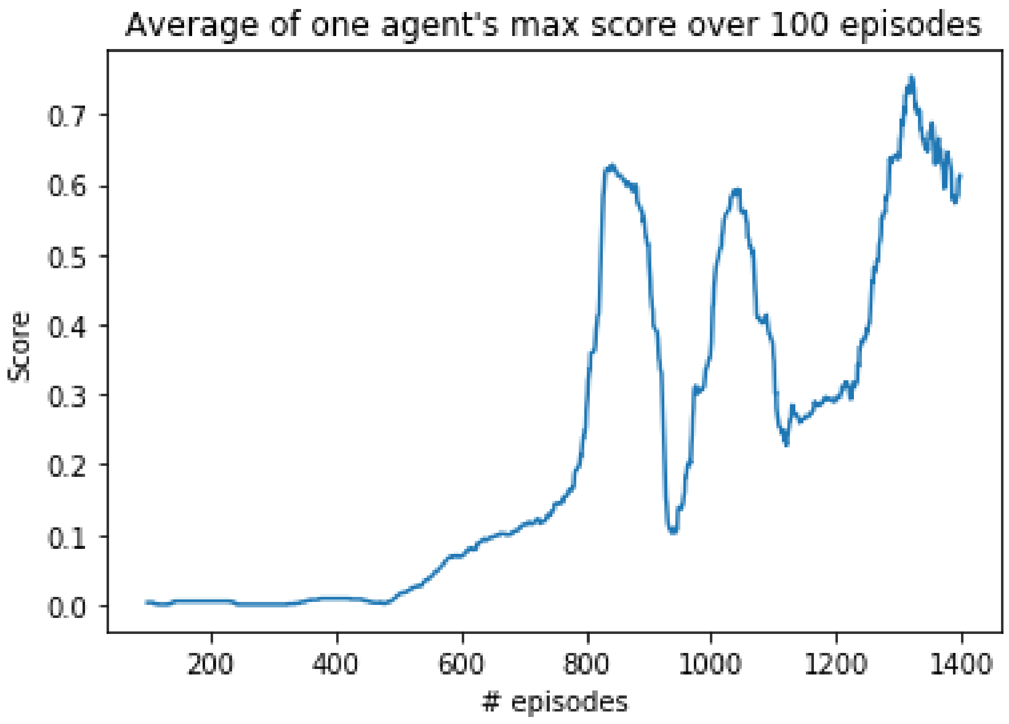
A few interesting observations are

- The agent achieved an objective score of +0.5 at ~800 episodes, but then collapsed and achieved a higher score again within ~1400 episodes
- The score is very variant in each episode. This might mean that the agent didn't understand all aspects of games yet, so it only specialised in a specific game situation.
- Although there is often a collapse in the performance. It recovers and shows a general upward trend as the agents experience more episodes. This could mean that the network architecture and training procedure was appropriate to meet the objective of +0.5 (100-episode average score) and could improve more if it keeps learning.

```
[12]: [<matplotlib.lines.Line2D at 0x7f9c3b740780>]
```



```
[13]: Text(0,0.5,'Score')
```



1.0.5 Future work

1. A lot of experiments on 2-hidden-layer network didn't show a good result and 3-hidden-layer network required a proper selection on its architecture hyperparameters. This can show a potential improvement using a different NN architecture. For example, Sequence Model could be helpful to understand the game history and might improve or stabilise the agent training.
2. In DDPG algorithm, the actor network could be trained with Q-value estimate of critic-current or critic-target network. It seems like critic-target creates a slower but more smooth training, but this can be validated further to improve the training procedure.
3. It's unclear yet what triggers the collapse of the scores. Some Hypotheses are:
   - NN training failure. New experience completely wiped NN's knowledge
   - The game situation could be adverse against what NN specialised in

   The situation can be debugged in more detail, which will help pinpoint the potential pitfall of training and can help avoid such a collapse during training