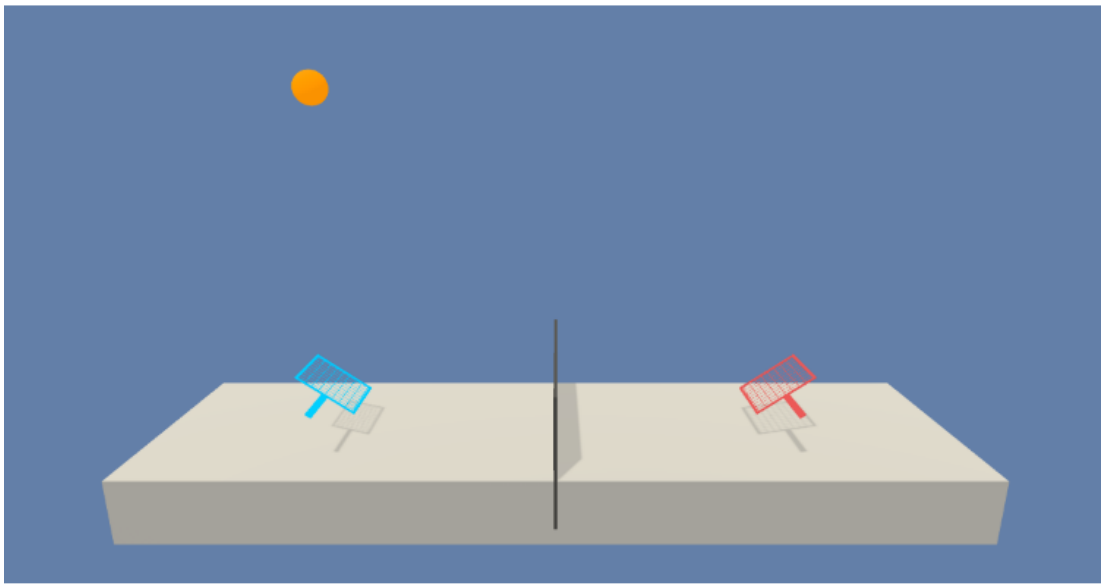
Udacity
Deep Reinforcement Learning
Nanodegree Program

Project: Collaboration and Competition

Developer: Koh Thiam Chye

1. Background



Unity ML-Agents Tennis Environment

For this project, we used the Unity ML-Agents Tennis simulated environment to train two agents to control rackets to bounce a ball over a net. A reward of +0.1 is given to agents for hitting the ball over the net. Conversely, if an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. The state space consists of 24 variables (3 stacked vector observations of size 8) corresponding to the position and velocity of the ball and racket. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. The actions are clipped to between -1 and 1.

The environment is considered solved if an average reward (over 100 episodes) of at least +0.5 can be achieved, which is based on the maximum score for the two agents for each episode.

2. Training Environment Setup

a) Operating Environment: Ubuntu 18.04 LTS

b) Imported Libraries:

i. UnityAgents (ver. 0.4.0)
ii. Numpy
iii. Random
iv. Sys
v. Torch
vi. Matplotlib
vii. Collections
viii. Copy

3. Implementation

Deep Deterministic Policy Gradients (DDPG) with Prioritized Replay Buffer

Hyperparameters:

- n_episodes (int): 3000 # max no. of episode
- max_t (int): 1,000,000 # no. of time-step in each episode
- BUFFER_SIZE = int(1e5) # max size for replay buffer
- BATCH_SIZE = 128 # size of experiences to be sampled
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR_ACTOR = 3e-3 # Actor network learning rate
- LR_CRITIC = 3e-3 # Critic network learning rate

Description of Model Architecture:

Two agents, each with its own Actor and Critic network models (with local and target instances) were created with random seed values.

For a maximum of 3000 episodes, at each time-step, the two agents interacted with the environment to get an independent action. For this, the state values were channeled into the local instance of Actor model to compute corresponding action vector, which was clipped to range [-1,1] and augmented with random noise.

The agents then took a step in the environment with the actions generated, which in turn returned the next-state, reward and done status. The agents updated the information to a shared prioritized replay buffer, which was implemented using SumTree. Using a delayed intensive training approach for greater model stability, the agents were trained at interval of 20 time-steps for 10 rounds (and 5 time-steps for 3 rounds) using sampled past experiences (state, action, reward, next_state, done).

To implement reinforcement learning using the Actor-Critic model, the next action was derived by feeding next-state to the target instance of the actor model. The Q-values for next-state is then generated from the target instance of the Critic model by feeding it with both next-state and next action. Following which, the Q-values for next-state is then used, together with other parameters including reward, gamma discount factor and done status, to compute the targeted Q-values for current state. In addition, the expected Q-values for current state is separately computed using the local instance of the Critic model using current state and action.

Next, the mean-square-loss between the targeted Q-values and expected Q-values are used in the backpropagation of the local instance of the Critic model to adjust its weights. Gradient clipping was applied when training the Critic model.

The loss of the Actor model is computed by negating the average of output from the local instance of the Critic model using current state and action predicted by local instance of the actor model.

The weights for the local instances and target instances are then combined based on an interpolation parameter (i.e. tau) and copied to the target instances of the Actor and Critic models for the subsequent rounds of reinforcement learning.

When the environment for any of the two agents is done, the episode ends, and the average cumulative maximum score generated by the agents are computed to assess whether the target reward (in this case +0.5) has been achieved. The training cycle continues with the next episode until the target reward is achieved or maximum episode is reached.

Please refer to Tennis.ipynb and ddpg_agent.py for more details.

Neural Network Model:

The model consists of two fully connected Actor and Critic neural networks.

For the Actor network, the state space of 24 values was embedded onto two input and hidden layers with 24 nodes each. The output layer consists of 2 nodes for the action vector. ReLu activation and batch normalization were implemented for the input and hidden layers of the Actor network, while the output layer was activated using tanh activation.

For the Critic network, the input layer consists of 52 nodes (2 x state_size + 2 x action_size) as it was essential for the Critic network to train on the states and actions of all agents. The corresponding output together with action space were further embedded onto another hidden layer with 52 nodes. ReLu activation was implemented for the input and hidden layers while batch normalization was only applied to the input layer.

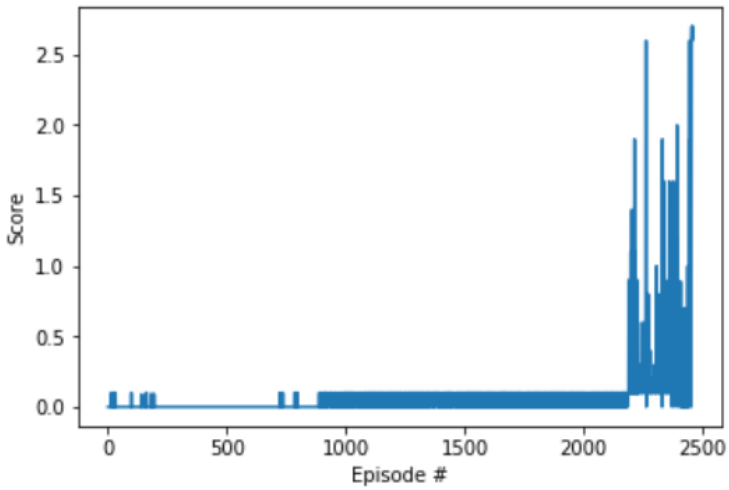
The Actor and Critic networks were also initialized with randomized weights, limited to the inverse square-root of the number of nodes in the respective layers.

Please refer to model.py for more details.

Training Outcome:

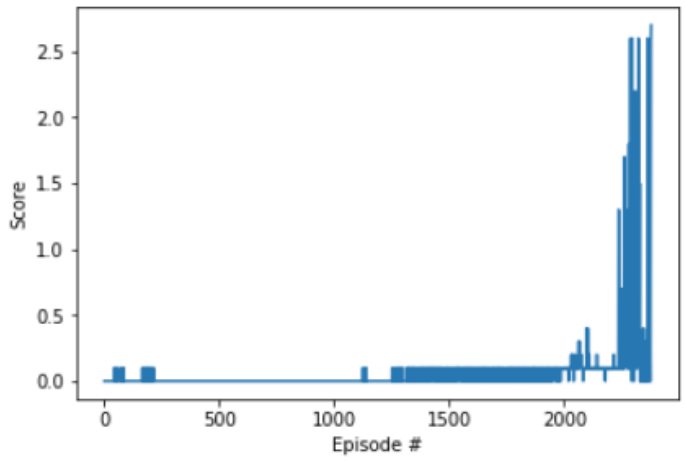
For a target score of at least +0.5, the environment was solved using the above mentioned DDPG model with prioritized replay buffer in 2400-2500 episodes, when configured for 3 rounds of training for every 5 time-steps. Note that the score improved significantly after 2200 episodes.

```
Episode 100     Average Score: 0.01
Episode 200     Average Score: 0.01
Episode 300     Average Score: 0.00
Episode 400     Average Score: 0.00
Episode 500     Average Score: 0.00
Episode 600     Average Score: 0.00
Episode 700     Average Score: 0.00
Episode 800     Average Score: 0.01
Episode 900     Average Score: 0.00
Episode 1000    Average Score: 0.02
Episode 1100    Average Score: 0.03
Episode 1200    Average Score: 0.04
Episode 1300    Average Score: 0.04
Episode 1400    Average Score: 0.05
Episode 1500    Average Score: 0.03
Episode 1600    Average Score: 0.06
Episode 1700    Average Score: 0.05
Episode 1800    Average Score: 0.05
Episode 1900    Average Score: 0.05
Episode 2000    Average Score: 0.05
Episode 2100    Average Score: 0.04
Episode 2200    Average Score: 0.09
Episode 2300    Average Score: 0.27
Episode 2400    Average Score: 0.37
Episode 2457    Average Score: 0.51
Environment solved in 2457 episodes!    Average Score: 0.51
```



With a configuration of 10 rounds of training for every 20 time-steps, the environment was solved in 2377 episodes. Similarly, the training score only improved significantly after 2200 episodes.

```
Episode 100     Average Score: 0.01
Episode 200     Average Score: 0.02
Episode 300     Average Score: 0.01
Episode 400     Average Score: 0.00
Episode 500     Average Score: 0.00
Episode 600     Average Score: 0.00
Episode 700     Average Score: 0.00
Episode 800     Average Score: 0.00
Episode 900     Average Score: 0.00
Episode 1000    Average Score: 0.00
Episode 1100    Average Score: 0.00
Episode 1200    Average Score: 0.00
Episode 1300    Average Score: 0.01
Episode 1400    Average Score: 0.04
Episode 1500    Average Score: 0.06
Episode 1600    Average Score: 0.05
Episode 1700    Average Score: 0.05
Episode 1800    Average Score: 0.05
Episode 1900    Average Score: 0.05
Episode 2000    Average Score: 0.06
Episode 2100    Average Score: 0.11
Episode 2200    Average Score: 0.10
Episode 2300    Average Score: 0.30
Episode 2377    Average Score: 0.50
Environment solved in 2377 episodes!    Average Score: 0.50
```



4. Ideas for Future Work:

The project could be re-implemented using the following methods,

- Distributed Distributional Deterministic Policy Gradients (D4PG)

  Involves modifications of the off-policy actor-critic DDPG to include a distributional critic update, the use of distributed parallel actors, N-step returns for estimating TD errors, and prioritization of the experience replay.

- Asynchronous Advantage Actor-Critic (A3C)

  Overcome the short comings of using experience replay which are memory and computationally intensive per interaction by asynchronously executing multiple agents in parallel across multiple instances of the environment, such that at different time-steps, the agents will be experiencing a variety of different states.

- Proximal Policy Optimization (PPO)

  Focus on alternating between sampling data through interaction with the environment and optimizing a “surrogate” objective function using stochastic gradient ascent. The key difference is while standard policy gradient methods perform one gradient update per data sample, PPO seeks to enable multiple epochs of minibatch updates to achieve stability in training.

Other methods include Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG).