Collaboration and Competition Project Report

Implementation

The agents are trained using a modified **Deep Deterministic Policy Gradient** (**DDPG**) algorithm to solve the multi-agent **Tennis** environment with two agents.

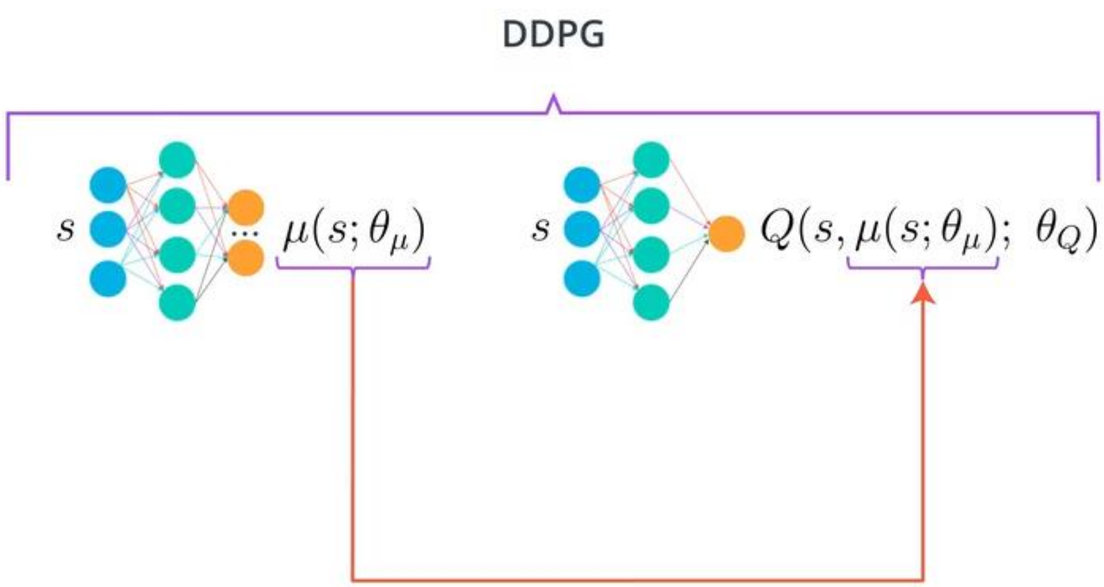
DDPG Model Architecture

The DDPG agent consists of two deep Neural Network:

1) Actor
2) Critic

The **Actor** is used to *approximate* the optimal policy deterministically, by learning $argmax_a Q(s, a)$, which is the best action.

Then the **Critic** learns to evaluate the **optimal value function** by using the Actor's best-believed action.



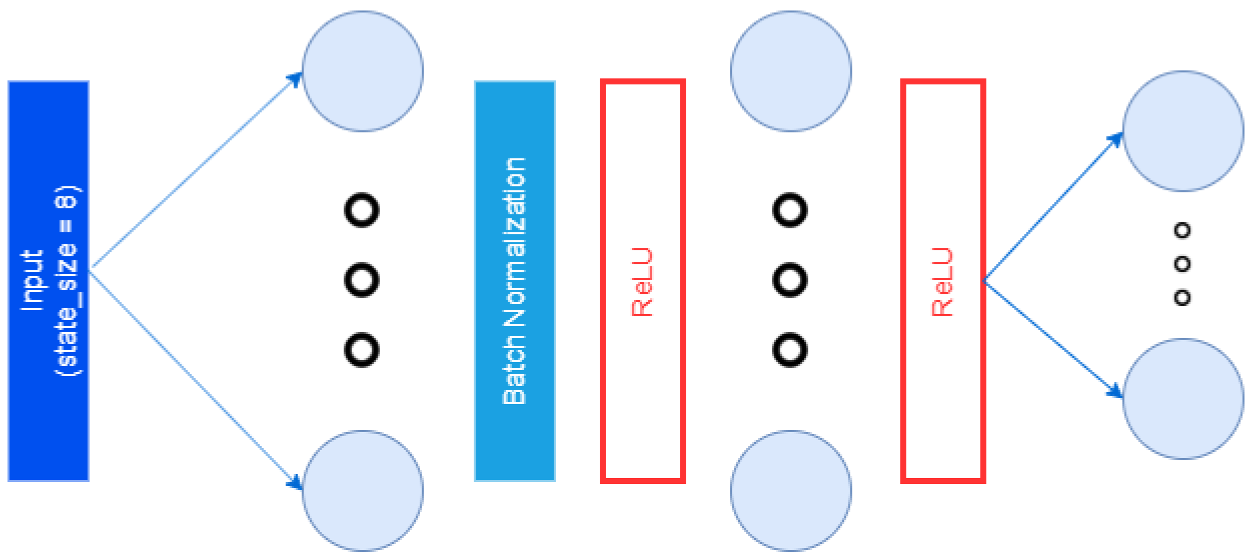
Both networks (Actor and Critic) consist of **three fully connected layers**.

The first hidden layer takes the size of the state space (*state_size = 8*) as input, and outputs *400 units*, which are passed as input to the second hidden layer.

The second hidden layer outputs *300 units*, which are passed as input to the third and final output layer.

In the **Critic**, the action size is added to the input of the second hidden layer, and the final layer outputs a single Q value.

In the **Actor**, the final output from the third output layer is the size of the action space (*action_size = 2*), corresponding to the movement toward (or away from) the net, and jumping.



Batch Normalization is applied to the output of the first fully connected layer to normalize each dimension across the samples in a mini-batch to have unit mean and variance and to minimize covariance shift during training.

The output of the first and second fully connected layers passes through a **ReLU activation function**.

The final output layer of the Actor is a **tanh layer**, to bind the actions.

➲ A single Actor and Critic are used for both agents.

Learning Algorithm

- Randomly initialize critic network (*critic_local*) and actor network (*actor_local*) with random weights θ^{Q} and θ^{π}.
- Initialize target networks (*actor_target*) and (*critic_target*) with weights $\theta^{Q`} \leftarrow \theta^{Q}$, $\theta^{\pi`} \leftarrow \theta^{\pi}$.
- Initialize replay memory (*ReplayBuffer*) with capacity (*BUFFER_SIZE = 10^6*).

- **for** episode (*i_episode* ← *1* to *1000*):
  - Prepare initial state: (*states* = *env_info.vector_observations*)
  - Initialize *scores* ← *0*
  - **for** time step *t* ← *1* to *1000*:
    - Select *actions* A from *states* S according to the current policy and exploration noise (an Ornstein-Uhlenbeck process with $\theta = 0.15$ and $\sigma = 0.2$).
    - Execute *actions* A, observe *rewards* R
    - Prepare next state: (*next_states = env_info.vector_observations*)
    - Store experience tuple (S,A,R,S`) in replay memory (*ReplayBuffer*)
    - states ← next_states
    - Add reward to score

    Every (LEARN_EVERY = *1*) timesteps:

    - Sample a random mini-batch of experience tuples (states, actions, rewards, next_states, dones) from memory (*ReplayBuffer*)
    - Compute Q targets for current states *Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))*
    - Update critic by minimizing the loss and applying Gradient Clipping to avoid exploding gradient problem.
    - Update the actor policy using the sampled policy gradient.
    - Update the target networks with ($\tau = 0.002$) of the local network weights: *θ_target = τ*θ_local + (1 - τ)*θ_target*.

Hyperparameters

Variable Name	Chosen Value	Description
BUFFER_SIZE	10^6	Replay buffer size
BATCH_SIZE	256	Mini-batch size
GAMMA	0.99	Discount Factor
TAU	0.002	Soft target updates value
LR_ACTOR	0.001	Actor's Learning Rate
LR_CRITIC	0.001	Critic's Learning Rate
WEIGHT_DECAY	0	L2 weight decay

LEARN_EVERY	1	Learning timestep interval
LEARN_NUM	1	Number of learning passes
GRAD_CLIPPING	1	Gradient Clipping value
OU_SIGMA	0.01	Sigma for Ornstein-Uhlenbeck noise process
OU_THETA	0.15	Theta for Ornstein-Uhlenbeck noise process

Results

```
Episode 100     Max. Score: 0.10        Avg. Score: 0.01
Episode 200     Max. Score: 0.20        Avg. Score: 0.03
Episode 300     Max. Score: 0.90        Avg. Score: 0.07
Episode 400     Max. Score: 1.70        Avg. Score: 0.16
Episode 440     Max. Score: 2.50        Avg. Score: 0.51
Environment solved in 340 episodes!     Average Score: 0.51
Training complete in 57m 37s
```

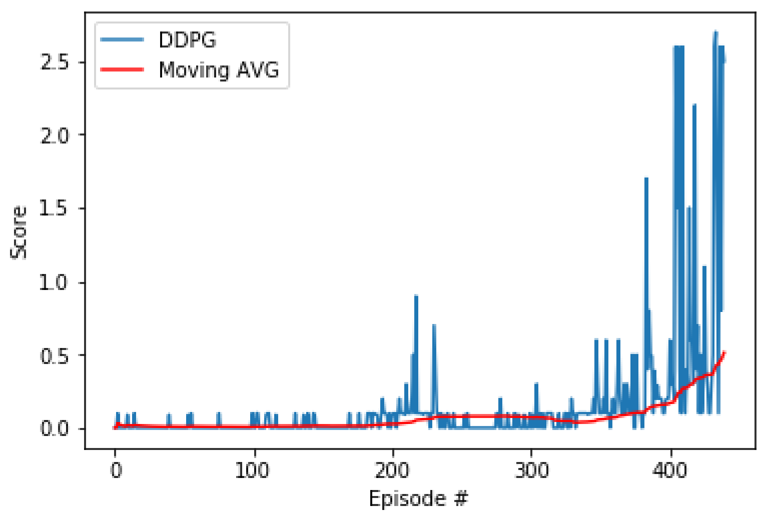


Figure 1 Plot of Rewards

The agents were able to receive an average score of **0.51** (over 100 consecutive episodes, after taking the maximum over both agents) in **340 episodes**.

Future Improvements

Several improvements could be implemented to enhance the agent's performance, including:

- Implementing Prioritized Experience Replay that replays **important** transitions *more frequently*, and therefore learns more efficiently.
- Implementing Multi-Agent DDPG (MADDPG) and compare its performance to simple DDPG.
- Implementing more stable methods to achieve better performance, like: Trust Region Policy Optimization (TRPO), Truncated Natural Policy Gradient (TNPG), Proximal Policy Optimization (PPO) or the more recent Distributed Distributional Deterministic Policy Gradients (D4PG).
- Fine tuning the hyperparameters further to solve the environment.

References

1) Continuous control with deep reinforcement learning

   https://arxiv.org/abs/1509.02971

2) DDPG Pendulum Exercise: Udacity's Deep Reinforcement Learning Nanodegree GitHub Repo

   https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum

3) Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments

   https://arxiv.org/abs/1706.02275

4) Benchmarking Deep Reinforcement Learning for Continuous Control

   https://arxiv.org/abs/1604.06778

5) Prioritized Experience Replay

   https://arxiv.org/abs/1511.05952