Udacity Deep Reinforcement Learning ND

Collaboration and Competition

REPORT

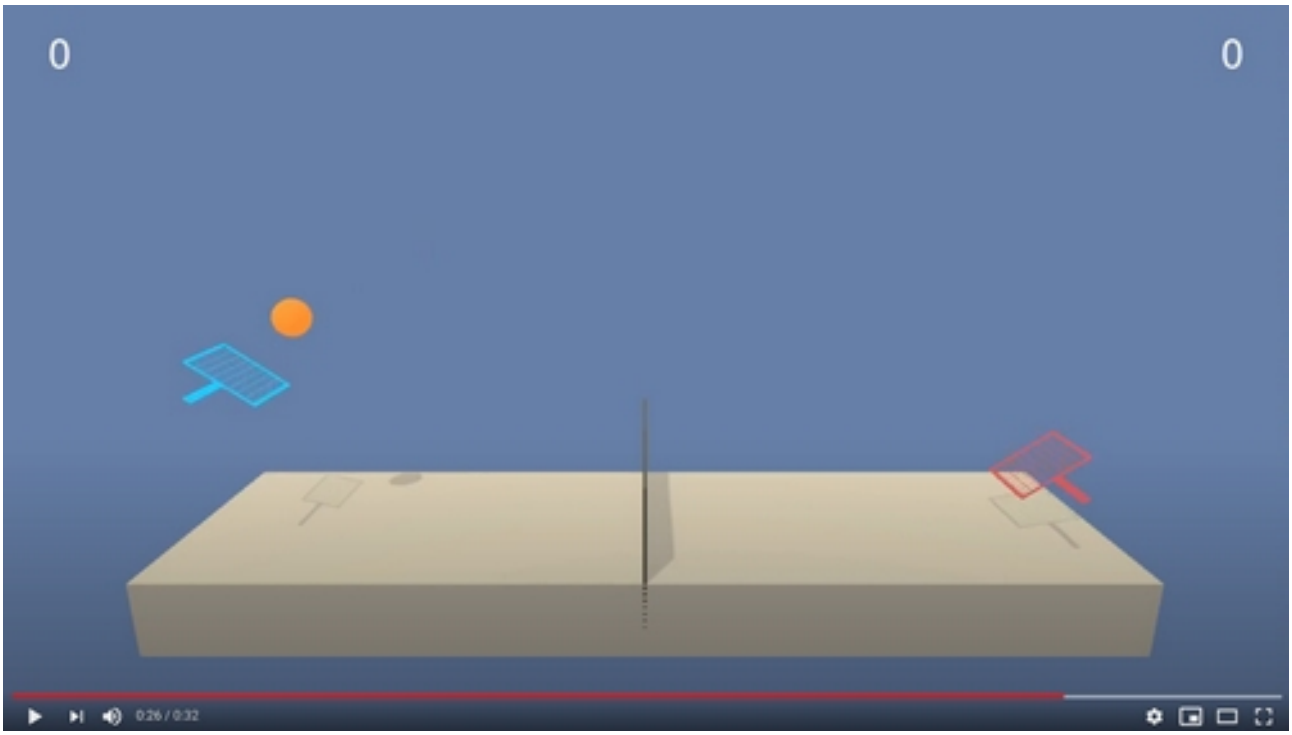


Figure 1: https://www.youtube.com/watch?v=BIa1UGzsWWc

Learning Algorithm

The final agent, was trained using a multi-agent ddpg (deep deterministic policy gradient) algorithm. The key difference between normal ddpg, is that each agents also uses the other agents state and actions to update themselve.

Hyper Parameters

The following hyper-parameters were used , as explained later.

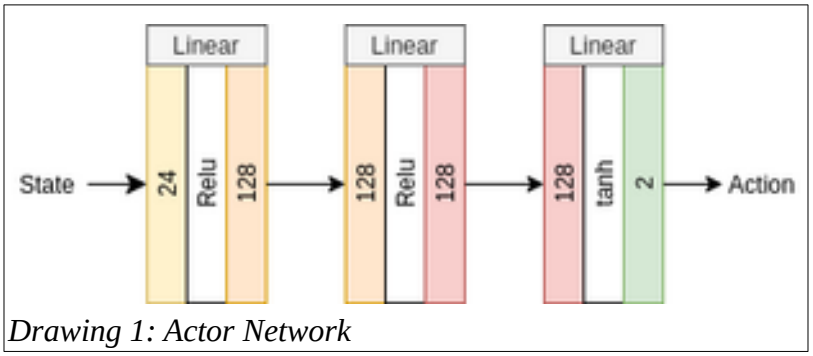
BUFFER_SIZE	100000	Maximum samples
BATCH_SIZE	128	Number of samples to use each step
GAMMA	0.99	Discounted reward
TAU	0.01	Weight update ratio
LR_ACTOR	0.0002	Learning rate for actor

LR_CRITIC	0.0002	Learning rate for critic

Network Architecture

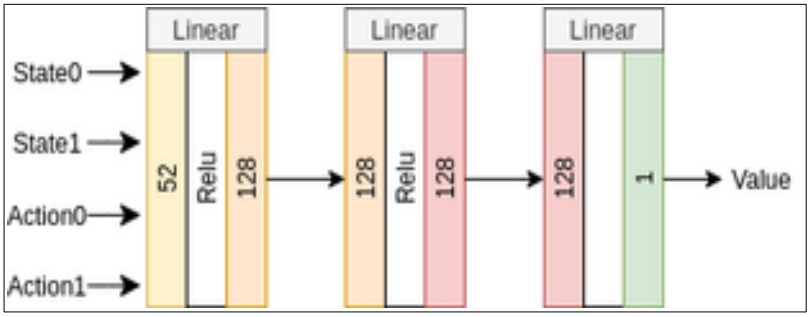
The algorithm uses two ddpg models (one for each agent) . The ddpg model has 4 networks , two actor , and two critic.

The actor network is shown below , where the 24 state inputs , are shown below. The state input (of 24 values) , is passed through 3 linear layers , the final output layer using a tanh() function , and produces two action values.



Drawing 1: Actor Network

The critic network has a similar structure, except that instead of feeding in its own states, it gets all the other agents states and the actions that led to that state. It also does not have any activation function on the final layer.



Drawing 2: Critic Network

Code.

The maddpg algorithm , I have implemented is very basic. The basic idea is that at every step, we feed the current respective state into each agents actor network, it produces an action to try. We add Ornstein Uhlenbeck noise to these actions, and then apply them to the environment. The results are stored in a replay buffer for latter use.

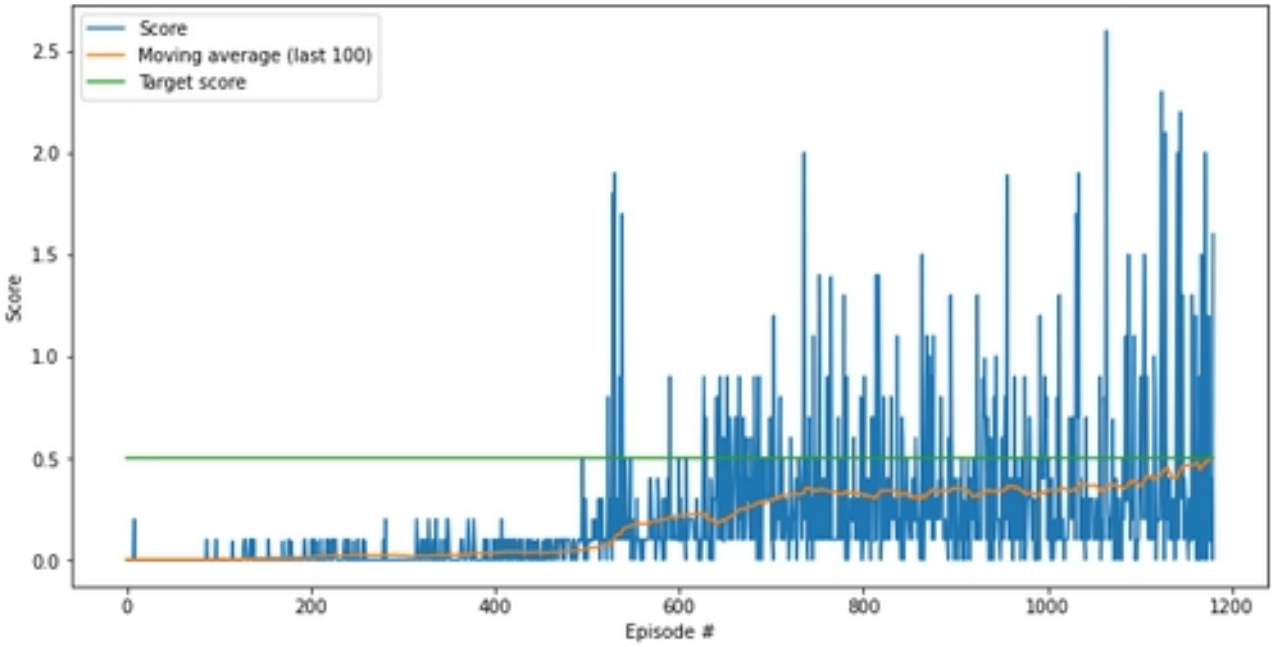
At each step, we sample a small number of these and update each agent's critic , and actors.

We have to be careful when sampling for each agent that we make sure its own states / action's and rewards are presented first, otherwise as I found out one agent may play very well (and we still meet the 0.5 criteria) - but the other agent is actively moving out of the way to concede the point to its competitor.

As we update on every step, the choice of learning rate / tau and batch size (and actor/critical size) seems to be critical. Since we are updating on every step and using a fairly large batch having very low learning rates was crucial (slightly wrong and you waste an hour as it nearly reaches 0.5 before it declines to near 0).

Training Code

Plot of Rewards



The agent, solved this task after the 1081 episode. Its not ideal as its mainly achieved this by having a lot of high scoring rounds.

Ideas for Future Work

For future work, we can probably improve performance of the agent a lot. Apart from optimising the code.

The hyper parameters, would be the first thing, they can be tweaked – but its frustrating as the whole system takes a long time to train – and you are not always guaranteed the solution will converge. Also the better the agents get the longer each episode lasts (we have limited ours to 1000 frames per episode) .

The use of Ornstein Uhlenbeck noise, is apparently not supposed to be the best method of adding noise either. These papers suggest other methods, although just adding a standard gaussian noise , and start delaying its standard deviation should work.

https://arxiv.org/abs/1804.08617

https://arxiv.org/abs/1706.10295

And other methods altogether , there are many to choice from.