Deep Reinforcement Learning – Project 3:

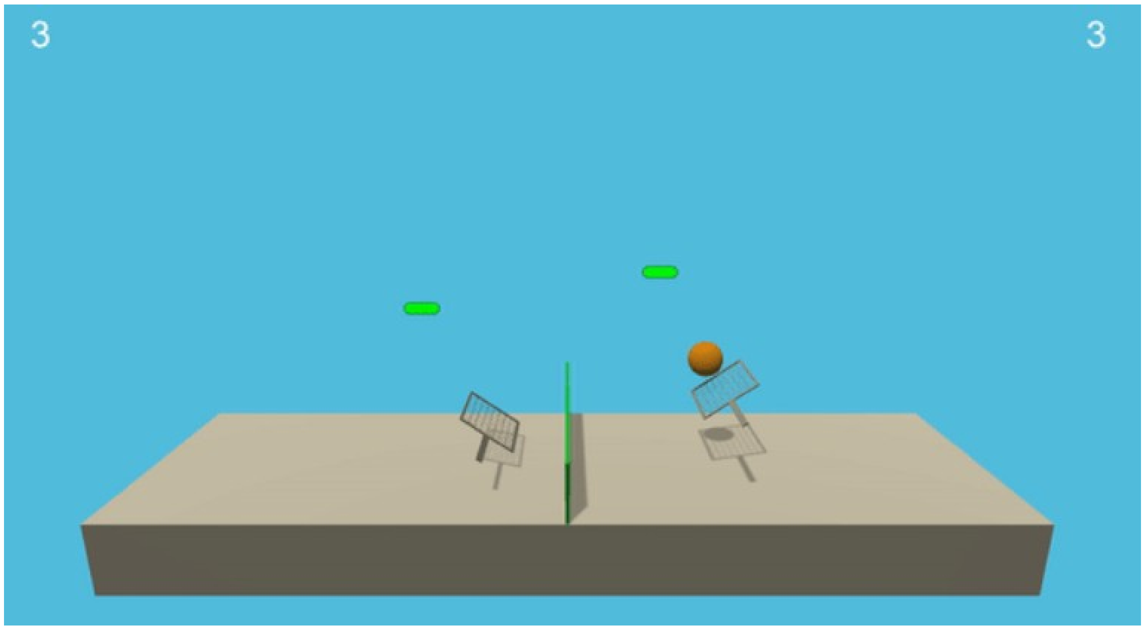
Collaboration and Competition – Tennis

Arpit Savarkar

For this project, we work with the Tennis environment, where two agents control rackets to bounce ball over a net. If an agent hits a ball over net, the agent receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, the agent receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play. The observation space is 24-dimensional consisting of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation.

In this environment, two agents control rackets to bounce a ball 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 -0.01. Thus the goal of each agent is to keep the ball in play.

The observation space consists of 24 variables corresponding to position and velocity of ball and racket. Each action is a vector with two numbers, corresponding to movement toward or away from the net, and jumping. Every entry in the action vector should be a number between -1 and 1. The task is episodic, and in order to solve the environment, the agents must get an average score of +0.5 over 100 consecutive episodes. Specifically, After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores. This yields a signal score for each episode.



2. Deep Deterministic Policy gradient :

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a).$$

DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted *specifically* for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s,a)$.

$$Q^*(s,a) = \underset{s' \sim P}{\mathrm{E}}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution $P(\cdot|s,a)$.

This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d indicates whether state s' is terminal). We can set up a **mean-squared Bellman error (MSBE)** function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}}\left[\left(Q_\phi(s,a) - \left(r + \gamma (1-d) \max_{a'} Q_\phi(s',a')\right)\right)^2\right]$$

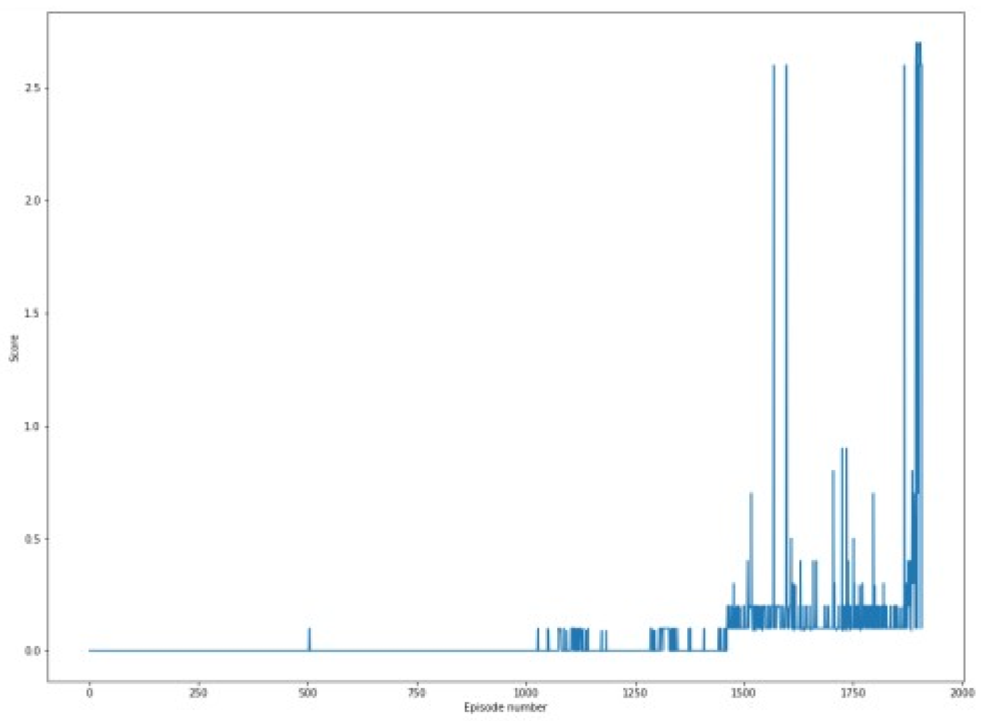
Here, in evaluating $(1-d)$, we've used a Python convention of evaluating `True` to 1 and `False` to zero. Thus, when `d==True`—which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state. (This choice of notation corresponds to what we later implement in code.)

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function. There are two main tricks employed by all of them which are worth describing, and then a specific detail for DDPG.

3. Implementation & Result

There are two models for each agent; the actor and critic that must be trained, it means that we have two set of weights that must be optimized separately. Adam was used for the neural networks with a learning rate of 10 −4 and 10 −3 respectively for the actor and critic for each agent. * For Q, I used a discount factor of γ = 0.99. For the soft target updates, the hyper parameter τ was set to 0.001. The neural networks have 2 hidden layers with 250 and 100 units respectively. For the critic Q, the actions were not included the 1st hidden layer of Q. The final layer weights and biases of both the actor and critic were initialized from a uniform distribution [−3 × 10 −3 , 3 × 10 −3] and [3 × 10 −4 ; 3 × 10 −4] to ensure that the initial outputs for the policy and value estimates were near zero. As for the layers, they were initialized from uniform distribution [− √ 1 f , √ 1 f] where f is the fan-in of the layer. We use the prelu 1 activation function for each layer. The environment is solved in **1963** episodes.

SCORE PLOTS PER EPISODE



Credit

Most of the code is based on the Udacity code for Mupti-agent DDPG.