Project 3 Collaboration and Competition (Tennis Environment) Report

Udacity Deep Reinforcement Learning Nanodegree

Algorithm Used

This project uses the Deep Deterministic Policy Gradient (DDPG) algorithm to train an Actor and a Critic to control two tennis rackets – each attempting to hit a ball over a net. The environment state space vector has 24 dimensions – position and velocity of the ball along with position and velocity of the racket, stacked across 3 consecutive frames – and 2 actions corresponding to moving the racket back/forth and up/down. Each racket perceives its own version of these state and action vectors. However, because each racket follows the same rules (i.e., good state→action decisions for one racket apply to both rackets), only one Actor/Critic pair is needed to control both agents.

Good performance was observed using similar neural network structures for the Actor and Critic. The 24-input environment state vector was input to a 128-element hidden layer, followed by a 100-element hidden layer, and finally a 2-element output layer to determine the actions to be taken by the Actor, whereas the Critic outputs 1 element (and incorporates the 2-element action in the first hidden layer). The network structures for the Actor and Critic are depicted in Figures 1 and 2, respectively.

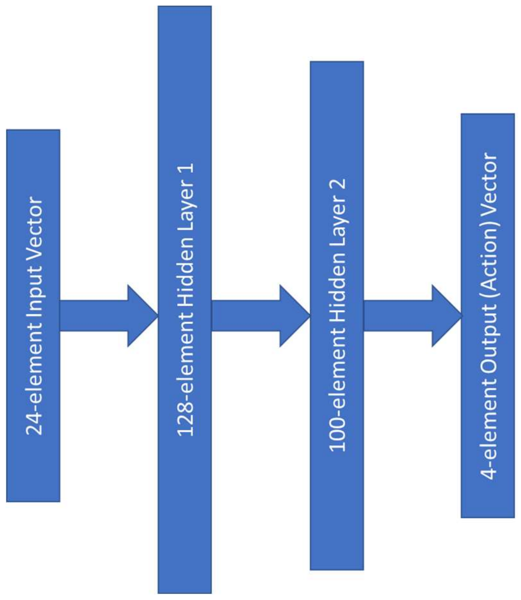
24-element Input Vector
128-element Hidden Layer 1
100-element Hidden Layer 2
4-element Output (Action) Vector

Figure 1. A representation of our Actor neural network

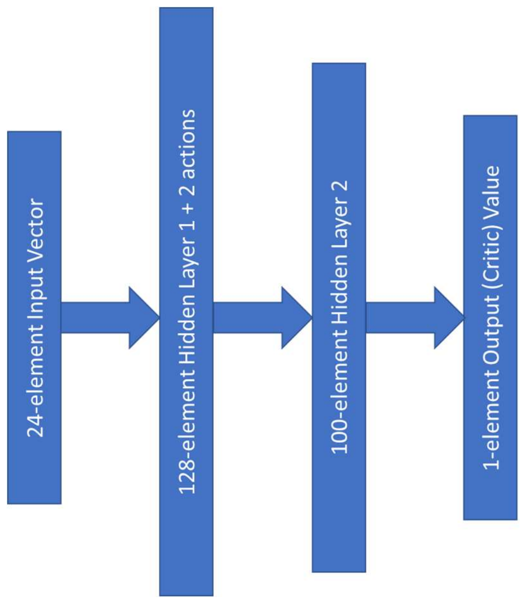
24-element Input Vector
128-element Hidden Layer 1 + 2 actions
100-element Hidden Layer 2
1-element Output (Critic) Value

Figure 2. A representation of our Critic neural network

These networks were trained over a maximum of 3000 episodes while using soft updates with a tau value of 0.001 and minibatches of size 256, each helping to prevent action values from oscillating or diverging), and a modest discount factor of 0.99 seen to be useful in earlier projects. After much experimentation, the learning rates of the Actor and Critic settled to be 0.0002 and 0.0003, respectively. The replay buffer size was also lowered from the previous project from 1,000,000 to 100,000 because timesteps that were much older seemed to hinder training. In this project, success was not seen until the weight decay value for the Adam optimizer was reduced to 0 (previously it was 0.0001).

An "economy" version of Prioritized Experience Replay was seen to be effective by using the following strategy:

- reward value < 0.03 ➔ add experience/reward to replay buffer once
- 0.03 ≤ reward value < 0.1 ➔ add experience/reward to replay buffer reward * 300 times
- 0.1 ≤ reward value ➔ add experience/reward to replay buffer reward * 150 times

Because many early steps resulted in a reward of 0, they were likely to overpopulate the replay buffer. With this strategy, higher rewards are added to the replay buffer multiple times, so the random sampling during the learning process is more likely to include "helpful" observations from which to learn.

Using these parameters, the environment was solved (i.e., achieved a score of at least +0.5 over 100 consecutive episodes after taking the maximum score across both agents) in roughly 1900 episodes. For good measure, I increased my "success" value to 0.6 which was overtaken in 1914 episodes. A plot of the scores over these episodes is shown in Figure 3 below.

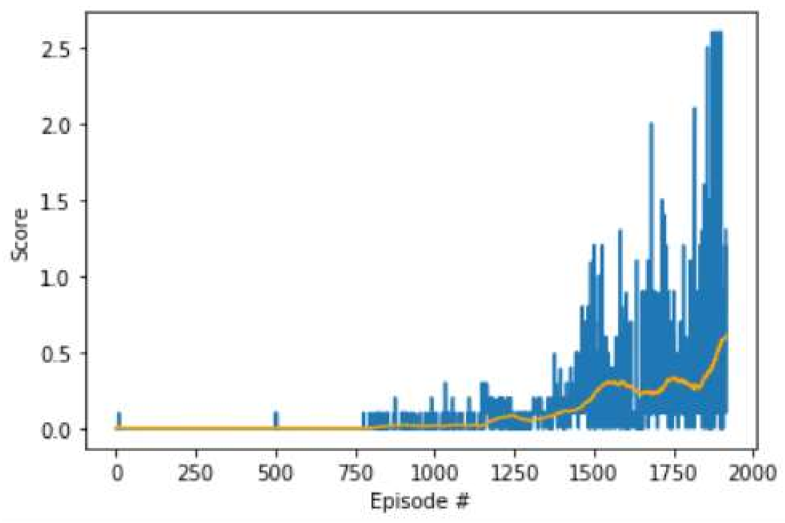

Figure 3. A plot of scores (blue) and moving average score (orange) over training episodes

```
Episode 1700   100-Ep. Avg.: 0.24     Max Ep. Score: 2.00
Episode 1800   100-Ep. Avg.: 0.28     Max Ep. Score: 1.50
Episode 1900   100-Ep. Avg.: 0.58     Max Ep. Score: 2.60
Episode 1914   100-Ep. Avg.: 0.61     Max Ep. Score: 1.30
```

In this case, “100-Ep. Avg.” is the average score over the course of 100 episodes.

Future Work

This algorithm would likely benefit from a less *ad hoc* Prioritized Experience Replay implementation. For example, rather than break up the memory addition process into discrete sections based solely on reward value, it might be more effective to add them using a smooth function based on *relative* reward value; e.g., if the latest reward were statistically much higher than previous rewards, there is likely more to learn from that memory. With the current implementation, high enough values can end up overpopulating the buffer just as low values do in the beginning of the process.

Currently, we transfer weights from the online network to the target network for every update. If instead we set a hyperparameter to decide how often we transfer these weights – for example, after every 2-4 updates – this might result in slightly more stable learning.