Project 3 Report

This report provides a description of the implementation of the software used to train a pair of deep reinforcement learning agents to play tennis in an environment provided by a unity ml-agent called *Tennis*.

The code is written in Python and PyTorch.

Learning algorithm

The learning algorithm used for the solution is based on the Deep Deterministic Policy Gradient (DDPG) actor-critic agent, originally introduced in this paper.

We train two agents at the same time, based on the intuition that each agent receives its own local observation. Thus, we can easily adapt the DDPG code to simultaneously train both agents through self-play. Each agent uses the same actor network to select actions, and the experience is added to a shared replay buffer.

The DDPG agent is an Actor-Critic which uses two Neural Networks, one for the actor and one for the critic. The actor is the policy function approximator which provides the critic with the best action vector to take at each time step. The critic is the action value function approximator and its aim it to approximate the optimal action value function and therefore the optimal total discounted reward for each state of the MDP.

The actor and the critic are two similar neural networks (NNs) characterised by two fully connected layers. The actor output layer has a number of nodes equivalent to the number of actions which the agent can take to interact with the environment. For the Tennis environment actions must belong to the interval [-1,1], therefore we apply a 'tanh' function to the actor output layer before returning from the forward method of the actor class.

The critic output size is one as the critic network maps state action pairs to corresponding Q-values.

In the Critic we apply drop-out to the layer before the last to minimise the risk of overfitting and speeding up training.

The algorithm uses the following hyperparameters:

- fc1_units = 512: the number of units in the first fully connected layer of both Critic and Actor NNs.

- **fc2_units** = 256: the number of units in the second fully connected layer of both Actor and Critic NNs.

- BUFFER_SIZE = 1e5: Size of the Replay Buffer.

- **BATCH_SIZE** = 512: Number of inputs processed per batch when running Stochastic Gradient Descent.

- **GAMMA** = 0.99: Discount factor of the Q-Learning algorithm

- **TAU** = 2e-1: used to perform soft updates of the target networks' parameters

- **LR_ACTOR** = 1e-4: learning rate provided to the Adam optimiser for the Actor

- **LR_CRITIC** = 1e-3: learning rate provided to the Adam optimiser for the Critic

- **update_every** = 5: number of time steps we wait between consecutive network updates.

- **WEIGHT_DECAY** = 0: weight decay for the Adam optimiser.

- **num_learning_cycles** = 10.0: number of consecutive learning cycles we perform every time we update.

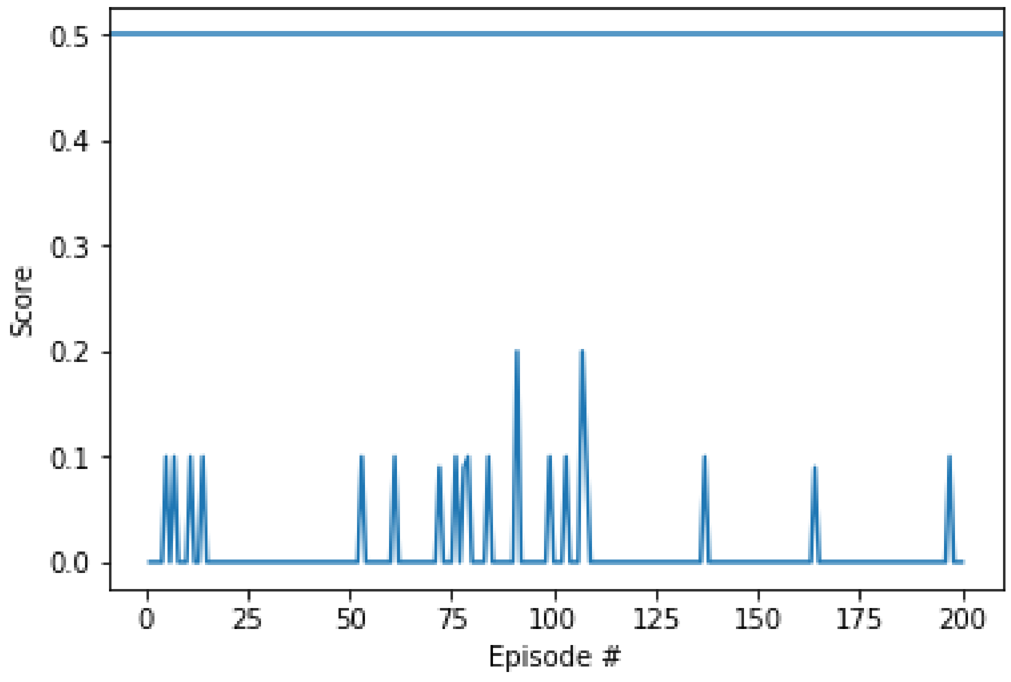
- **drop-out** = 0.2

Plot of Rewards

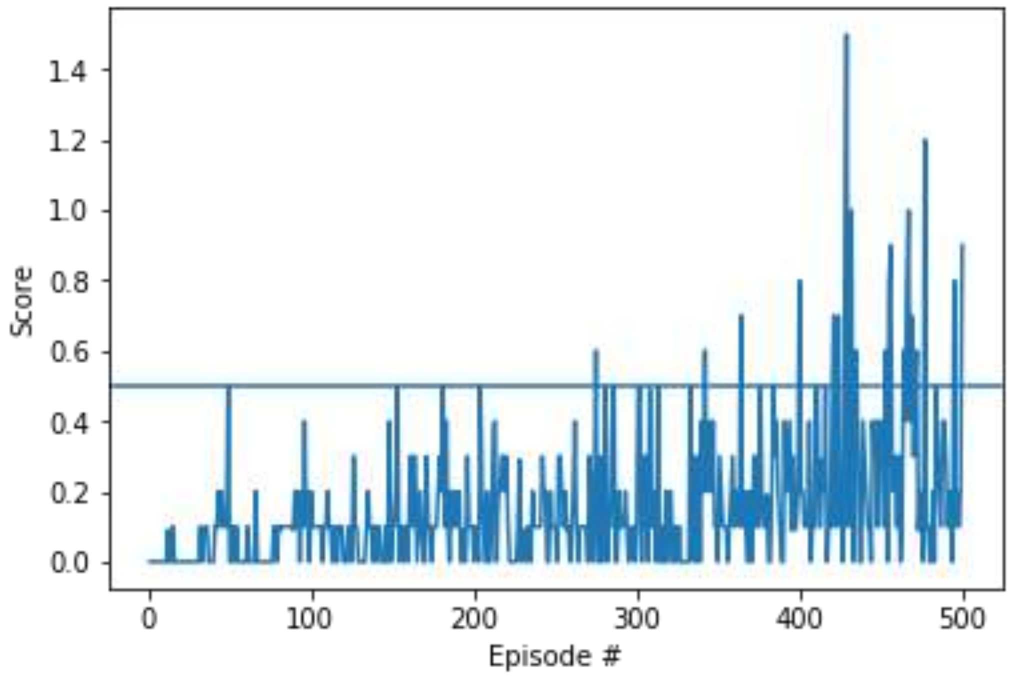
Having decided to pursue the idea of training two agents to self-play based on DDPG, I modified the 'Tennis.ipynb' file to implement the main parallel training loop. The baseline model and agent are those already used in previous projects.

I had already experienced from previous projects that DDPG is quite sensitive to the frequency you update the actor and critic networks. I therefore reduced the number of networks' updates to every 5 time steps, and allowed for 10 consecutive updates each time. I also decided to work on reasonably small networks to start with, by having both actor and critic networks characterised by fully connected layers of 256 and 128 nodes.

As shown in the plot below this did not provide great results.



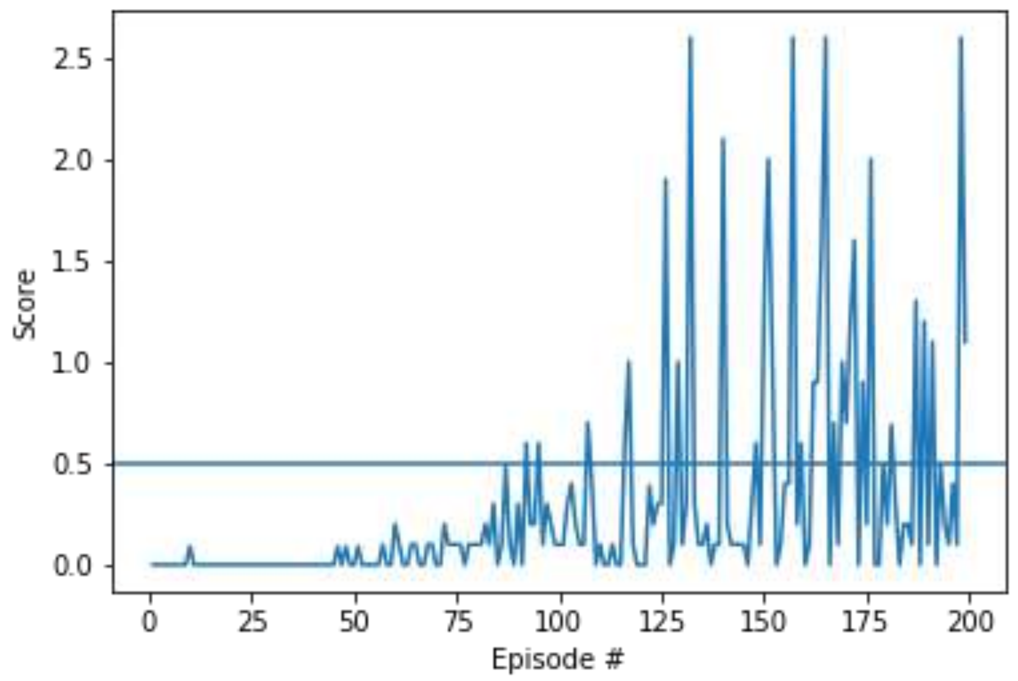
I therefore went back to the baseline actor and critic architectures described in the DDPG original paper, where the two fully connected layers in each network have 400 and 300 units both. I also used a slightly larger batch size of 512. In this case I obtained better results as shown in the plot below, even if the training was still a bit slow.



It looked to me that this was clearly the correct strategy to pursue, but picking the ideal parameters for faster training proved very challenging. Note that in the benchmark implementation discussed in the lectures they took more than 1,500 episodes to solve this environment. I will definitely invest some time in the near future to learn effective ways of making the search automatic via Random Search or Bayesian methods or both.

At this point I invested some time browsing the web to look for other people's experience on solving the tennis environment, in particular their choice of hyperparameters, and found the following blog.

The hyper-parameters used by the author are not too different from mine, but eventually allow for faster training rates. Using similar hyperparameters to the ones used in the blog, allowed me to obtain the best results and resolve the environment in just 199 episodes, as shown in the plot below (better than the blog's author).



Ideas for Future Work

1. Add Prioritised Experience Replay to the Tennis project.
2. Solve the Tennis environment using MADDPG as outlined in the multi agent RL lectures.
3. Create an environment for automatic determination of hyperparameters, by implementing either a Random Search or a Bayesian method or both.
4. Resolve the Soccer environment provided by Udacity.