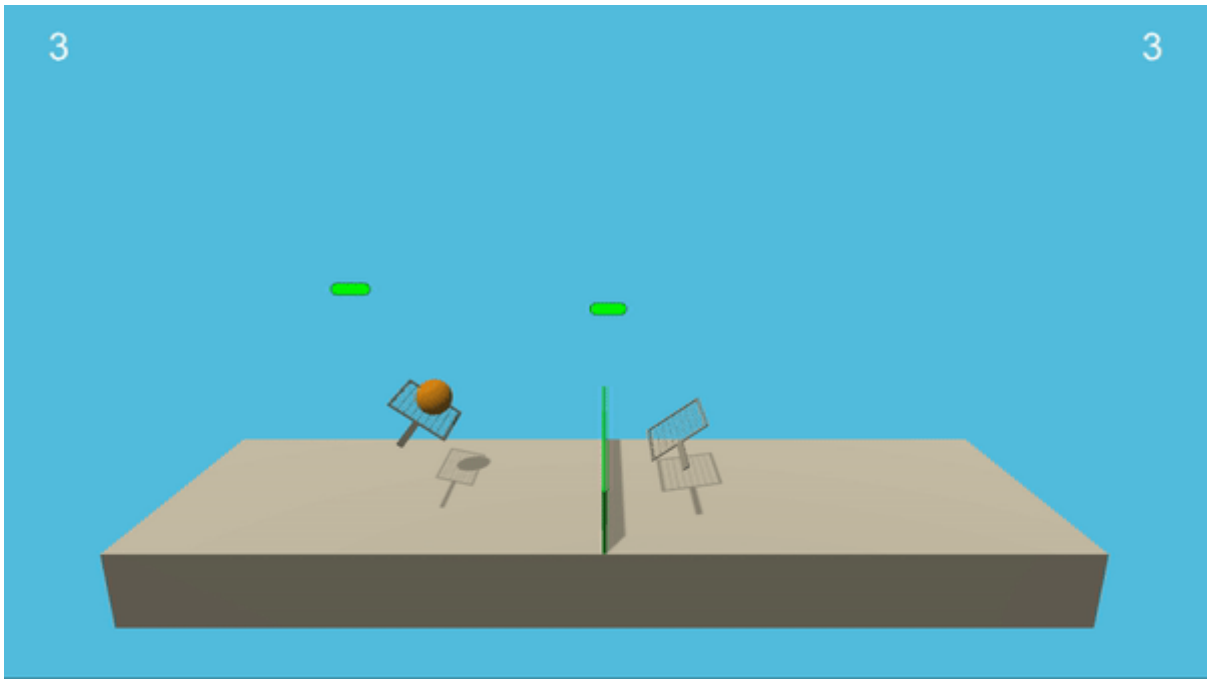
Multi-Agent DDP Algorithm, Project 3
Rens ter Weijde, April 3rd 2019

Introduction to multi-agent systems
For this project we looked at a multi-agent environment. The relevance of this should be obvious; as humans we often live and act in social environments, and so will the intelligent agents we design. Direct applications in the near future might relate to swarms of drones, self-driving groups of vehicles, army convoys, collective trades etc., but I'm sure many more applications will become apparent over time.

Environment
The environment used to showcase the performance of the algorithm was an environment from Unity, called Tennis. Two agents have to bounce a ball over the net and receive a reward of +0.1 if this is done successfully, and -0.1 if not. The primary difficulty in the environment is the fact that it is no longer stationary with another agent present. Note that multi-agent environments can be set up as either *competitive* or *collaborative.*



Example screenshot of the Unity Tennis environment

The goal in the environment is to keep the ball in play, and through such means increase the reward. The task is finished when the reward of 0.500 is reached. In this environment, both agents receive their own local observations (observation space of 8 variables) and has two actions to choose from (towards or away from the net). The task is in 2D to limit the action space.

Algorithm
The algorithm used is called MADDPG (Multi Agent Deep Deterministic Policy Gradient), as also described by Open AI. This algorithm uses centralized learning and decentralized execution with two agents (agent 0 and agent 1); it also uses the actor-critic logic that we

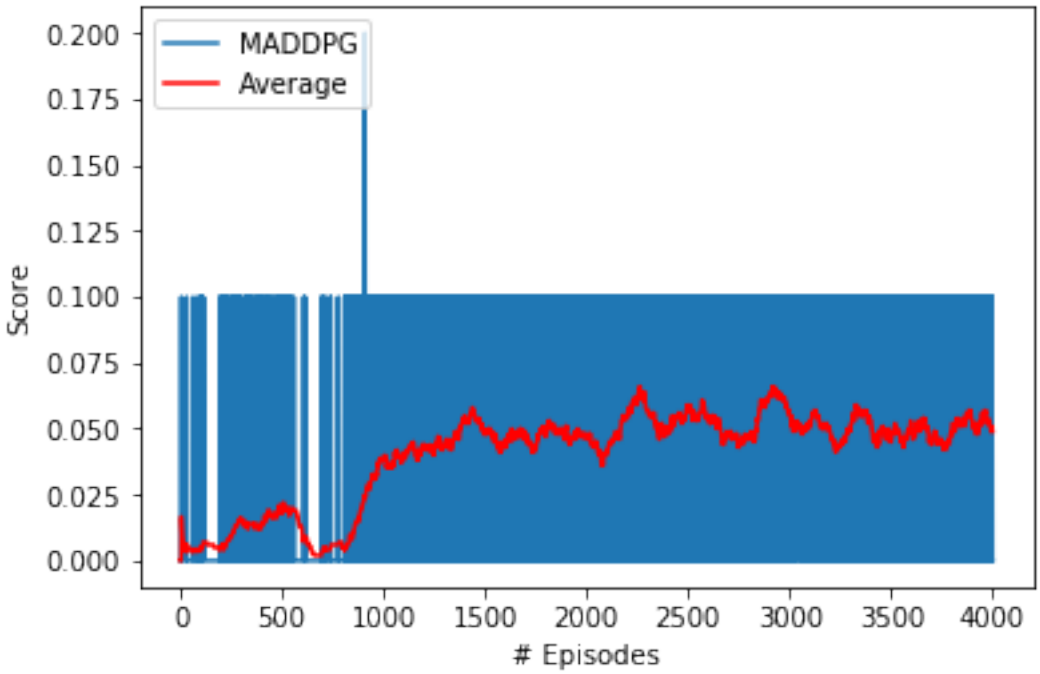
explored in past projects for a more stable learning algorithm. The algorithm is new in the sense that it beats traditional algorithms (DDPG, A2C, Q-learning etc.) in a multi-agent environment. The algorithm furthermore makes use of Experience Replay, similar to past projects, where the experience tuples are sampled from memory. For exploration, the algorithm uses a noise function that adds random noise.

When it comes to activation layers; the networks use simple ReLU activation functions with tanh on the last layer. I tried implementing batchnorm on the first layer but got a size mismatch that I couldn't fix, so in the end trained without it.

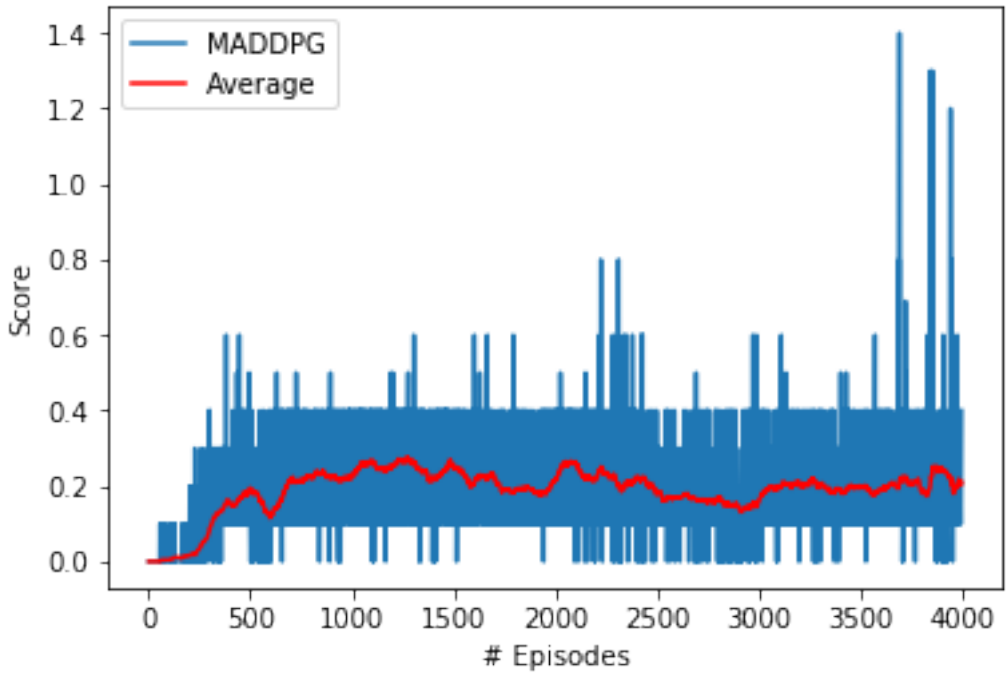
The algorithm uses a lot of parameters, which I have optimized by learning from others' recommendations. The final list of parameters used in this code is below.

BUFFER_SIZE = int(1e6)
BATCH_SIZE = 128 (scaled back from 256, see below)
LR_ACTOR = 1e-3
LR_CRITIC = 3e-4 (changed from 1e-3)
WEIGHT_DECAY = 0
LEARN_EVERY = 1
LEARN_NUM = 1
GAMMA = 0.99
TAU = 6e-2 (lowered from 7e-2)
OU_SIGMA = 0.2
OU_THETA = 0.13
EPS_START = 5.5
EPS_EP_END = 250
EPS_FINAL = 0

Training
For me, the algorithm didn't work right away and took a couple of attempts to get right. My first attempt had a reward that stabilized around 0 and didn't show any improvement after 2,000 episodes. After optimizing the parameters, the model converged better but I realized I still needed more episodes to make it to +0.5. For the third time of training, I increased the number of episodes to 4,000 but changed the parameters for the worse and so got no strong convergence (see below, max reward never got beyond 0.100). In the end, I trained with 5,000 episodes, lowered the Tau value a bit as well as the learning rate for the critic and changed the batch size from 256 to 128 again. The final results are shown in the second graph, where the agent reaches a maximum reward of +1.3.



Third attempt; learning never surpasses 0.05



Fourth attempt; algorithm succeeds in reaching max rewards of 1.300

The stability of the training is another matter; the agents trained through MADDPG are not very stable, there is considerable variation from episode to episode. Sometimes there is even downright regression in the performance shown. I waited for the 'spike in performance' shown in some implementations, but for me that spike never came in the older implementations.

Ideas for future training

- Further hyperparameter tuning. Although I've experimented a bit with the hyperparameters, I'm sure other things could be tried. First things to try would be different batch sizes (256 tried, 128 tried, made a large difference), different noise settings, and a different setting for the learning rates (changed learning rate for critic seemed to matter as well).
- Network depth. The networks are – as in the last project – again relatively simple. It is worth a shot to make them 1-2 layers deeper to check the results. The current layers are 256 -> 128, you could e.g. add another layer with 64 neurons.
- Batch normalization over the input data.
- Other algorithms, like TD3, or the fully decentralized architectures.