Udacity Deep Reinforcement Learning Nanodegree

Collaboration and Competition Project Report

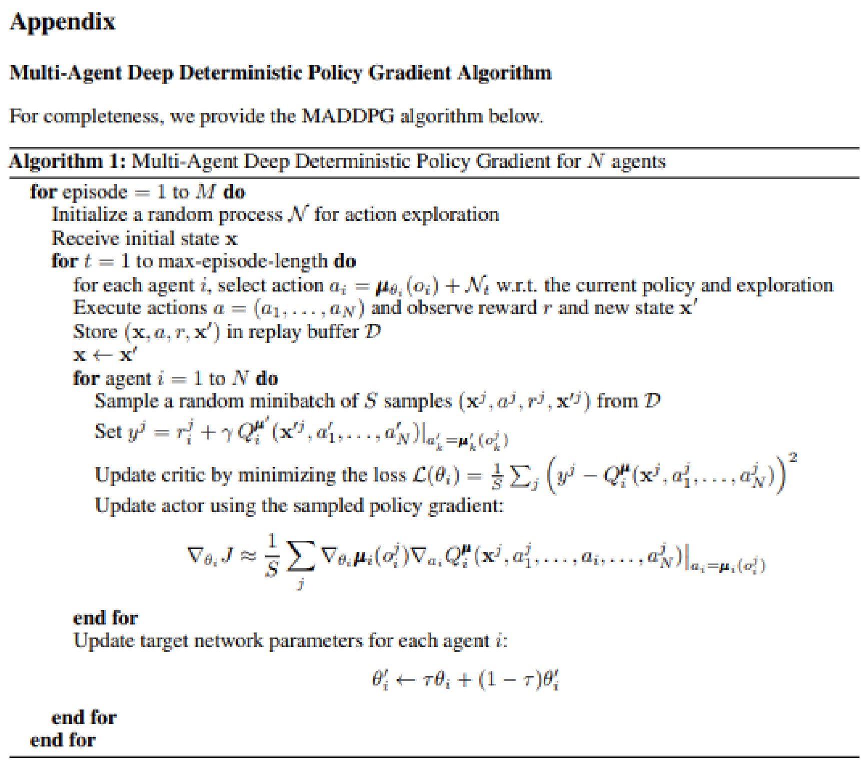
Deborah Fish

Introduction

This report provides a brief description of my project to train two agents to collaborate to play tennis in Udacity's variant of the Unity ML Agents Tennis environment. The environment has a discrete observation space with a state size of 24 for each agent (comprising the position and velocity of the agent, racquet and other agent). Each of the two agents has a continuous action space with two actions in the range -1 to +1 corresponding to the racquet moving left-right or up-down. A reward of +0.1 is earned for every time step in which the agent hits the ball over the net; a reward of -0.01 is incurred if the agent lets the ball hit the ground or pass out of bounds. Success is obtained when the maximum of the two agents' scores exceeds an average of 0.5 over 100 consecutive episodes.

Learning Algorithm

I approached this task by starting with the DDPG (deep deterministic policy gradient) algorithm I used for Project 2 and extending it to work with multiple agents (multi-agent DDPG) as described in Lowe et al., 2017. This required creating a MADDPG_Agent class, in addition to the original Agent class, with functions to carry out actions for each agent and to learn in a multi-agent setting, in accordance with the MADDPG algorithm shown in Figure 1 below. Particular care was needed to process tensors (using tensor.cat, tensor.reshape and tensor.narrow) to ensure they were in the right format to save to the experience replay buffer, to extract the observation made by each agent from the array of state data, and to concatenate information to feed it to the Critic functions.



Appendix

Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm below.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

```
for episode = 1 to $M$ do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length do
    for each agent $i$, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to $N$ do
      Sample a random minibatch of $S$ samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma \, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:
          $$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$
    end for
    Update target network parameters for each agent $i$:
          $$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$
  end for
end for
```

Figure 1. MADDPG algorithm, from Rowe et al., 2017.

Network architecture. My Actor network took the local observations available to that agent only as input, and output the continuous values of the two actions. It included two hidden layers of sizes 400 and 300, each with ReLU activitation, and I also added batch normalisation for the first hidden layer. My Critic networks (one for each agent) concatenated the observations from both agents and the actions from both agents as input, and output a single number for each - the value function. The Critic networks included two fully connected hidden layers of sizes 400 and 300, each with ReLU activation. Again, I used batch normalisation for the first hidden layer.

Optimiser. I used the Adam optimizer to iterate the network weights. I started with learning rates of 0.01, as used by Lowe et al., but found that the agents did not learn. I decreased both learning rates to 1e-3, then decreased the actor and critic learning rates in turn until I reached values that worked (actor learning rate 0.001; critic learning rate 0.0001).

Hyperparameters. I experimented with hyper-parameters in the ranges used by either Lowe et al., used in Project 2 for the DDPG algorithm. I focused on the Actor and Critic learning rates first, as experience with Project 2 showed their importance. The final hyper-parameters used, and the other values tried, are summarised below:

Hyperparameter	**Value used**	**Other values tried**
Batch_size	128	None
Buffer_size (experience replay)	1e5	None
Gamma	0.99	None (standard value)
Tau (for soft updates)	2e-3	1e-3, 2e-3 - higher values learnt more quickly
Actor - Hidden_layers	[400,300]	[256, 128]
Critic - Hidden_layers	[400,300]	[256, 128]
Actor - Learning rate (Adam optimizer)	1e-3	0.01 (as used by Lowe et al.), 1e-3, 2e-4.
Critic - Learning rate (Adam optimizer)	1e-4	0.01 (as used by Lowe et al.) 1e-3, 2e-4.
Update_every (learning takes place once every 4 timesteps)	4	None
Sigma (OU noise parameter)	0.2	None
Theta (OU noise parameter)	0.15	None
Scale applied to OU noise	2, reducing by a factor of 0.999 each episode until a	1 - no discernible difference in

	minimum of 1	performance.

Plot of Rewards

A plot of the rewards achieved during training is shown below. Rewards are plotted as a function of the episode number, showing that after 3557 episodes (for trajectories with a maximum of 1000 timesteps) the agent is obtaining an average score exceeding 0.5. I found it difficult to train the agent, as performance differed when I ran it successive times. After some checking of the random number seeds in the agent, I tried re-running the initial cell in which the agent chooses random moves. Despite using np.random.seed() to seed the sequence of random numbers used, this code gave different answers, so I concluded that the environment was stochastic (or needed seeding). The model weights were saved to files 'checkpoint-agent-0.pth' and 'checkpoint-agent-1.pth' immediately after the agent achieved the goal.

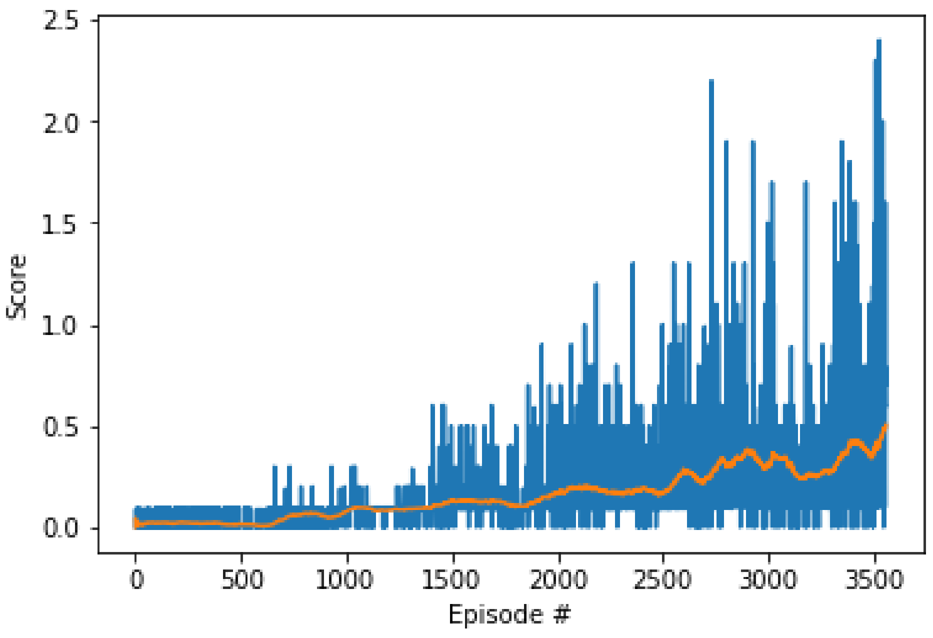


Figure 2. Plot of rewards achieved by the agents. The maximum of the two agents' rewards is shown for each episode in blue, with the running average over 100 episodes shown in orange.

Ideas for Future Work

It might be possible to improve the performance of the MADDPG agent by implementing a prioritised experience replay buffer, instead of sampling the buffer at random. Other hyperparameters could also be tuned further - for example, the length of the experience buffer can be optimised - too short, and it stores too few experiences; too long and it may contain too may experiences from poorer agents.

I would be interested to compare the performance with MADDPG on this task with that of DDPG applied independently to the two agents. My reasoning is that the two agents only collaborate weakly - if both agents independently learn to hit the ball over the net, they should be able to keep a rally going and thus achieve a relatively high score.

References

Lowe, Ryan, Wu, Yi et al., 2017, Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments, 31st Conference on Neural Unformation Processing Systems, NIPS, available at https://arxiv.org/abs/1706.02275.