Reinforcement learning project 3: Competition And Collaboration

Reza Doobary

September 1, 2019

Contents

1 Introduction

In this short report we provide model implementation details and analysis on the reinforcement learning project entitled 'Competition and collaboration' as part of the Udacity course on deep reinforcement learning.

Whilst full task details of the agent, the environment and success criteria have been outlined in the accompanying README.md, we repeat these for completeness. The agent in this case is a ping pong or tennis paddle and the system is a pair of such agents posed against one another at a ping pong table. The task of the system for each agent to maintain a rally. The state space of the agent is a 24 dimensional real float space and contains the agent's (the racket) position and the velocity of ball and racket. As a pair of states, the system has therefore got a 48 dimensional space. The action space is 2 dimensional float space for each agent and corresponds to moving away and towards the net and moving up and down. The reward structure of the system is that each agent gains a reward of +0.1 each time they hit the ball over the net. They are penalised -0.1 if the ball drops to the table. The score of an episode is the maximum score of both agents. The task is deemed solved if the system gets an average score of +0.5 over 100 consecutive episodes.

NOTE : In this project, some effort has been placed into reporting on the theoretical understanding of the models implemented. To go straight to the actual implementation of the algorithms, please go to section 3. For extremely quick results, with nothing more than the hyper-parameters given please go to section 4.

2 General concepts

In this section, an overview of some of the main concepts used in this project are provided.

2.1 Deep deterministic policy gradient (DDPG)

Given some knowledge about actor-critic methods, the DDPG is the answer to the question: *How can we make the standard DQN apply to continuous actions?*. Recall from [1], the the goal of the DQN is to optimise the zero of the Bellmann equation of the state-action function, namely to minimise:

$$\mathbb{E}\left[\left(Q(s_t, a_t) - \left(R_{t+1} + \gamma \max_a Q\left(s_{t+1}, a\right)\left(1 - \mathbb{I}_{\text{done}}\right)\right)\right)^2\right]. \tag{1}$$

This expression only really makes sense when there is a discrete number of actions, where it would make sense to consider the maximum of a set of finite numbers. For the continuous case, this is essentially an infinite number. We could in principle remedy this with coarse coding [2] , but this is unwieldy and there are better methods.

We can then venture laterally, where we have studied the actor critic architecture. During this course, we studied it in the context of policy gradients methods. The idea is to have an agent to encapsulate two models. Firstly, an actor model for the optimal action (in which we optimise a surrogate of the return function via gradient ascent), and a critic model (in which we optimise the policy loss). In essence, the actor-critic method, tells our agent how to act and then tells the agent how valuable that action was.

The DDPG combines these two basic ideas by targeting the key issue with applying basic DQN to a continuous action, namely to model the action-value function with a critic model. Thus the actor model attempts to find the optimal action which maximises the action-value function.

Let's reword this more concretely in a mathematical formalism. We would like to model the state-value function, $Q(s, a, \theta)$ parametrised by θ. Ultimately, we are interested in the model the minimises the function

$$J_Q\left(\theta, \hat{\theta}\right) = \mathbb{E}\left[\left(Q(s_t, a_t, \theta) - \left(R_{t+1} + \gamma \hat{Q}\left(s_{t+1}, \mu(\psi), \hat{\theta}\right)\left(1 - \mathbb{I}_{\text{done}}\right)\right)\right)^2\right]. \tag{2}$$

in which we now put hats on the fixed target variables to acknowledge that fact that the optimisation problem would not be useful otherwise. This is the critic model. However, we have made the replacement:

$$\hat{Q}\left(s_{t+1}, \mu(\psi), \hat{\theta}\right) = \max_a \hat{Q}\left(s_{t+1}, a, \hat{\theta}\right). \tag{3}$$

We find $\mu(\psi)$ via our critic model, in which we want to maximise:

$$\max_\psi J_\mu\left(\theta, \psi\right) \text{ where } J_\mu\left(\theta, \psi\right) = \mathbb{E}\left[Q(s, \mu(\psi), \theta)\right]. \tag{4}$$

Thus, we have a meaningful way of finding an optimal action, however contrary to policy gradient methods, this is a deterministic model as opposed to a stochastic one [1]. Putting back the local/target parametrisation, together with a soft update, the pseudo-code is given by algorithm 1.

where

$$\text{SoftUpdate}(\tau, X, Y) = \tau X + (1 - \tau)Y. \tag{5}$$

Finally, although libraries such as tensorflow and PyTorch will handle the gradients for us, it is worth noting that the gradient for equation is reminiscent of the corresponding gradient expression for policy gradients:

$$\nabla_\theta J_\mu\left(\theta, \psi\right) = \mathbb{E}\left[\nabla_\psi \mu(\psi) \nabla_a Q(s, a, \theta)|_{a=\mu(\psi)}\right]. \tag{6}$$

[1] Of course, for the exploration vs. exploitation principle a stochastic element will be added to the determined action.

Algorithm 1 DPG

```
Collect MDP data to replay buffer $\mathcal{B} = (s_t, a_t, R_t, s_{t+1}, d)$
for $i$ in number of updates do
    Sample $B \in \mathcal{B}$
    Compute target $y(R_t, s_{t+1}, d) = R_t + \gamma(1 - \mathbb{I}_{\text{dones}})\hat{Q}(s_{t+1}, \hat{\mu}(\psi), \hat{\theta})$
    Minimise $\mathbb{E}_B \left[ (Q(s_t, a_t, \theta) - y(R_t, s_{t+1}, d))^2 \right]$ for $\theta$
    Maximise $\mathbb{E}_B \left[ Q(s, \mu(\psi), \theta) \right]$ for $\psi$
    Perform the soft update: $\hat{\theta} = \text{SoftUpdate}(\tau, \hat{\theta}, \theta)$ and $\hat{\psi} = \text{SoftUpdate}(\tau, \hat{\psi}, \psi)$
end for
```

2.2 Multiple agents

This project involves the use of multiple agents, and we shall follow the methodology explained in [3]. The main idea explained in correspondence with DDPG is the idea of a centralised critic whilst having a decentralised actor. In addition each agent only knows about observations local to itself, this was studied in [5] and then in [3]. We discuss generalities and then go toward the DDPG algorithm specifically.

Multi-agent problems following a Markov decision process paradigm fall under the definition of Markov games, which is defined to be the tuple:

$$\langle n, S, \{A_i\}_{i=1:n}, \{\mathcal{O}_i\}_{i=1:n}, \{R_i\}_{i=1:n}, \{\pi_i\}_{i=1:n}, T \rangle, \tag{7}$$

where i denotes the agent, n being the number of agents. The countable sets $\{A_i\}_{i=1:n}, \{\mathcal{O}_i\}_{i=1:n}, \{R_i\}_{i=1:n}$ and $\{\pi_i\}_{i=1:n}$ are the actions, observations, rewards and individual policies for each agent. S is a the set of environment states. Finally, each of these sets can be made into a larger vector space by a simple tensor product, thus we write:

$$\begin{aligned} A &: A_1 \times A_2 \times \cdots \times A_n \\ R_i &: S \times A_i \to R \\ \pi_i &: \mathcal{O}_i \to A_i \\ T &: S \times A \to S. \end{aligned} \tag{8}$$

The approach in [3] is to centralise the training whilst decentralising the execution. From the actor-critic method, the execution is encapsulated by the actor whilst the 'training' is encapsulated by the critic. We therefore allow the critic to hold information about the actions and observations of all agents, for DDPG in practice, this means that the state-value function is a function of all actions and states:

$$Q\left(\{\mathcal{O}_i\}_{i=1:n}, \{A_i\}_{i=1:n}, \theta\right). \tag{9}$$

On the other hand the actor (deterministic policy), should take in all observation[2], but only produce an action for the agent it is associated to. Thus we write this as

$$A_i = \mu_i\left(\{\mathcal{O}_i\}_{i=1:n}, \psi\right). \tag{10}$$

The Multi-agent deep deterministic policy gradient (MADDPG) is given by algorithm 2. We shall get into the specifics of how to implement the steps in PyTorch with a neural network, but the main point here is that the algorithm does not materially change too much - we simply have to take into account the handling of multiple agents and their interaction.

[2]so long as all agents exists in the same environment

Algorithm 2 MADDPG

```
1 : Collect Markov games data to replay buffer $\mathcal{B} = (S_t, A_t, R_t, S_{t+1}, D)$
for $i$ in number of updates do
    2 : Sample $B \in \mathcal{B}$
    3 : Compute target $y(R_t, S_{t+1}, d) = R_t + \gamma(1 - \mathbb{I}_{\text{dones}})\hat{Q}(S_{t+1}, \hat{\mu}(\psi), \hat{\theta})$
    4 : Minimise $\mathbb{E}_B\left[(Q(S_t, A_t, \theta) - y(R_t, S_{t+1}, d))^2\right]$ for $\theta$
    5 : Maximise $\mathbb{E}_B\left[Q(S, \mu(\psi), \theta)\right]$ for $\psi$
    6 : Perform the soft update: $\hat{\theta} = \text{SoftUpdate}(\tau, \hat{\theta}, \theta)$ and $\hat{\psi} = \text{SoftUpdate}(\tau, \hat{\psi}, \psi)$
end for
```

3 Implementation and Analysis

In this section, we finally discuss key implementation details and the results that follow. In this project, we have have the MADDPG model, by modelling the state-value function and the deterministic policy with neural networks.

Taking a different approach to previous project, here we will discuss implementation details going operator by operation in algorithm 2, whilst discussing additional steps that we taken, for example, the addition to stochastic noise to the action produced.

3.1 Collecting a sampling data

This following from operations 1 and 2 of algorithm 2. In this specific problem of the two agent system, the algorithm collects states, actions, rewards and terminations of each agent and places them in a memory buffer. In this context, the data comes as a pair of states, actions rewards and terminations. When a sufficient number of data points (in this case 1000000) have been obtained, the replay buffer is sampled from, and the learning algorithm is implemented. This is the standard replay experience method and is implemented in `replaybuffer.py`.

3.2 Minimising $\mathbb{E}_B\left[(Q(S_t, A_t, \theta) - y(R_t, S_{t+1}, d))^2\right]$ for θ

Having obtained the dataset we can minimise the critic model. The critic model is a neural network with two hidden layers of 256 and 128 units respectively. In particularly the layers take the form:

```
super(Critic, self).__init__()
    self.seed = torch.manual_seed(seed)
    self.fcs1 = nn.Linear(state_size, fcs1_units)
    self.fc2 = nn.Linear(fcs1_units+(action_size), fc2_units)
    self.fc3 = nn.Linear(fc2_units, 1)
    self.reset_parameters()
```

which can be found in `model.py` under the `Critic` class. The model is implemented into the agent via the following

```
self.critic_local = Critic(state_size*2, action_size*2, random_seed).to(device)
self.critic_target = Critic(state_size*2, action_size*2, random_seed).to(device)
```

which can be found in `maddpg_agent.py`. From these lines of code we see that this model takes as input the states and actions of both agents. This is how the model implements and notion of centralised training.

The update training is done as follows:

```
# unloading data from the replay buffer called experiences
states, actions, rewards, next_states, dones = experiences

# ---------------------------- update critic ---------------------------- #
# Get predicted next-state actions and Q values from target models
actions_next = self.actor_target(next_states)
# Since the critic takes the actions of both agents we need to update only
# one part of the given action
if agent_number == 0:
    actions_next = torch.cat((actions_next, actions[:,2:]), dim=1)
elif agent_number == 1:
    actions_next = torch.cat((actions[:,:2], actions_next), dim=1)
# Compute Q targets for current states (y_i)
Q_targets_next = self.critic_target(next_states, actions_next)
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
# Compute critic loss
Q_expected = self.critic_local(states, actions)
critic_loss = F.mse_loss(Q_expected, Q_targets)
# Minimize the loss
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()
```

which can be found in `maddpg_agent.py`. We see that the actor model is employed in order to find the actions, which is then used in the critic model. A key point is that the action found using the actor model is for a specific agent denoted `agent_number`, thus the action must be concatenated with the other agents action and plugged into the critic model.

3.3 Maximising $\mathbb{E}_B\left[Q(S, \mu(\psi), \theta)\right]$ for ψ

We can now go onto the maximising for the parameter associated to find the deterministic policy. We should also side step slightly to discuss a major exploration measure added to the model to allow for a higher variance in the actions found early on. The actor model is a neural network with two hidden layers of 256 and 128 units respectively. In particularly the layers take the form:

```
    super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()
```

which can be found in `model.py` under the `Actor` class. The model is implemented into the agent via the following

```
        self.actor_local = Actor(state_size*2, action_size, random_seed).to(device)
        self.actor_target = Actor(state_size*2, action_size, random_seed).to(device)
```

from which we see that the actor take both states as an observation but only produces a single action associated to the agent. This is how the model implements decentralised execution.

The update training of the deterministic policy function is given by:

```
# ---------------------------- update actor ---------------------------- #
# Compute actor loss
```

```
actions_pred = self.actor_local(states)
# Since the critic takes the actions of both agents we need to update only
# one part of the given action
if agent_number == 0:
    actions_pred = torch.cat((actions_pred, actions[:,2:]), dim=1)
elif agent_number == 1:
    actions_pred = torch.cat((actions[:,:2], actions_pred), dim=1)
# Compute actor loss
actor_loss = -self.critic_local(states, actions_pred).mean()
# Minimize the loss
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()
```

which can be found in `maddpg_agent.py`. We see that the actor model is employed in order to find the predicted actions, which is then used in the state value function (the critic model). Once again we see that we need to concatenate the predicted action with the action belonging to the other agent. We do this in this case since we need to make use of the critic model.

3.4 Noise and action

The agent performs an action via the function

```
def act(self, states, add_noise):
    """Returns actions for both agents as per current policy, given their respective
       states."""
    states = torch.from_numpy(states).float().to(device)

    self.actor_local.eval()
    with torch.no_grad():
        actions = self.actor_local(states).cpu().data.numpy()
    self.actor_local.train()
    # add noise to actions
    if add_noise:
        actions += self.eps*self.noise.sample()
    actions = np.clip(actions, -1, 1)
    return actions
```

We see that noise has been added which is the standard Ornstein-Uhlenbeck process, particularly familiar in financial mathematics. The `self.eps` parameter is somewhat new, and is a multiplicative decaying parameter which simply enlarges the variance of the Ornstein-Uhlenbeck process. We started this variable at 3.0 and had the variable decay by a factor of 0.9999 every time the agent learns.

3.4.1 Results

The agents were each made up of two neural networks, one modelling the actor (the policy), the other modelling the critic (the state-value function). Each had two hidden layers with 256 and 128 nodes respectively. The critic model has an input layer of 48 dimensions and an output layer of 1. The actor model has an input layer of 48 dimensions and an output layer of 2 dimensions.

The task was solved in 1930 episodes and the average reward is provided by the plot in figure 1. It's noteworthy that the moving average shows a general trend, however being somewhat stochastic.

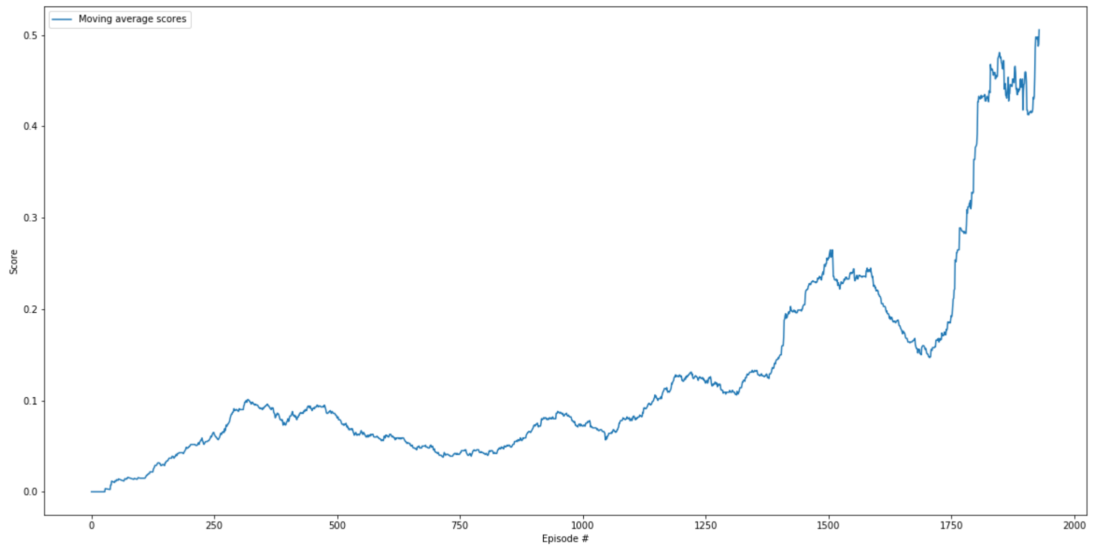

Figure 1: The MADDPG was solved in 1930 episodes.

4 Quick Results

The neural network used was the MADDPG model.

4.1 MADDPG

Parameter	**Value**
MADDPG - Actor - input layer	48
MADDPG - Actor - hidden layers	[256,128]
MADDPG - Actor - output layer	2
MADDPG - Critic - input layer	48
MADDPG - Critic - hidden layers	[256,128]
MADDPG - Critic - output layer	1
Adam optimiser - Actor - learning rate	1e-3
Adam optimiser - Critic - learning rate	1e-3
Discounting	0.99
Epsilon for noise	3.0
Epsilon for noise decay	0.9999

The model was solved in 1930 episodes. The plot of the training can be found in figure 1. The successful model parameters can be found in \results\20190827\run_1.

5 Conclusion and further work

The main conclusion of this project was that the MADDPG algorithm requires 1930 episodes to solve this ping pong task. For further work, it would have been interesting to do the following:

1. To use a prioritized experience replay to store the Markov game data. This is important as the it would attach probabilities to the likelihood of certain impacting TD differences to occur.
2. To employ the use of some policy gradient methods, in particular it would be curious to see how PPO attempts this task given the use of the importance sample.
3. The learning suffered from some instability, thus it would have been wise to play with the hyper-parameters to establish (perhaps slower) but a more stable upward trend. In addition, more experimentation with layering would have been useful here, in particular to understand if a 'number of episodes required vs. stability' issue can be understood.

References

[1] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Rusu, Andrei A. and Veness, Joel and Bellemare, Marc G. and Graves, Alex and Riedmiller, Martin and Fidjeland, Andreas K. and Ostrovski, Georg and Petersen, Stig and Beattie, Charles and Sadik, Amir and Antonoglou, Ioannis and King, Helen and Kumaran, Dharshan and Wierstra, Daan and Legg, Shane and Hassabis, Demis. Human-level control through deep reinforcement learning. Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved. 2015.

[2] Sutton, Richard S. and Barto, Andrew G. .Reinforcement Learning: An Introduction. The MIT Press. 2018.

[3] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel and Igor Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. CoRR. abs/1706.02275.

[4] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra. Continuous control with deep reinforcement learning. abs/1509.02971, 2019.

[5] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas and Shimon Whiteson. Learning to Communicate with Deep Multi-Agent Reinforcement Learning. CoRR. abs/1605.06676.