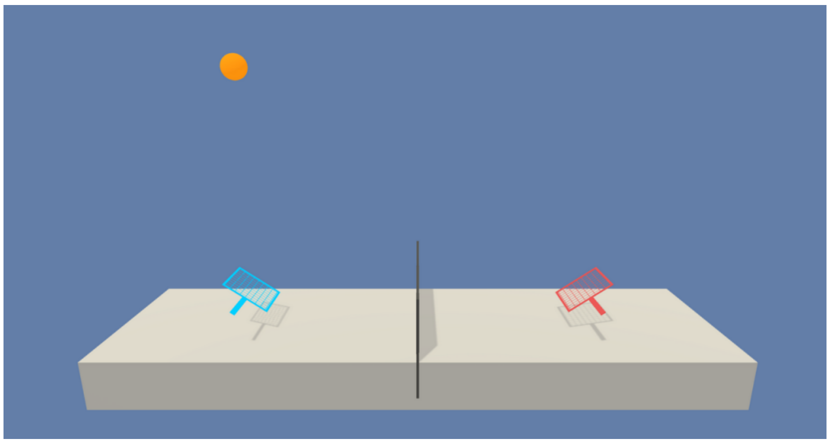
Collaboration and Competition Project

Udacity Deep Reinforcement Learning Nanodegree

António Soares

December 11, 2018



1. Project Description

The current project attempts to solve a task using a value-based model-free and off-policy deep reinforcement learning algorithm that uses a neural network to decide on which actions to take in its environment.

The task consists of two agents that need to learn how to bounce a ball over a net. The environment provides a reward of +0.1 if an agent is able to hit the ball over the net, and a punishment of -0.01 if an agent lets the ball fall into the ground or hits the ball off the playing field.

The goal of each agent is therefore to cooperate and keep the ball at play during the available time-steps in a given episode so as to maximise the total episodic reward.

2. Environment

An adapted version of the Tennis environment available in Unity ML-Agents GitHub page is used in this project. Observation states are composed 3 frames each composed of 8 variables corresponding to the position and velocity of the ball and racket within the environment. The Python Numpy representation of an observation state from both agents as returned by the environment looks like this:

```
[[-1.08997898e+01  8.95357311e-01 -9.53674316e-06 -6.51797473e-01
   6.83172083e+00  2.62143707e+00 -9.53674316e-06 -6.51797473e-01
  -1.08997898e+01  7.71317601e-01 -9.53674316e-06 -1.63279748e+00
   6.83172083e+00  1.75815713e+00 -9.53674316e-06 -1.63279748e+00
  -1.08997898e+01  5.49177825e-01 -9.53674316e-06 -2.61379719e+00
   6.83172083e+00  8.38537276e-01 -9.53674316e-06 -2.61379719e+00]
 [-1.08997898e+01  8.95357311e-01 -9.53674316e-06 -6.51797473e-01
  -6.83172083e+00  2.62143707e+00 -9.53674316e-06 -6.51797473e-01
  -1.08997898e+01  7.71317601e-01 -9.53674316e-06 -1.63279748e+00
  -6.83172083e+00  1.75815713e+00 -9.53674316e-06 -1.63279748e+00
  -1.08997898e+01  5.49177825e-01 -9.53674316e-06 -2.61379719e+00
  -6.83172083e+00  8.38537276e-01 -9.53674316e-06 -2.61379719e+00]]
```

The action space is composed of a vector with 2 continuous variables which correspond to the agent's movement towards/away from the net and jumping. Each variable is a real number between -1 and 1. The Python Numpy representation of an action vector for the two agents looks like this:

```
[[-1.          -0.75461821]
 [-1.          -0.75461577]]
```

Each episode's score is obtained by selecting the maximum score from the set total scores of both agents. The task is considered solved when the average score obtained over 100 consecutive episodes is greater than 0.50.

3. High-level Algorithm

To train our agent and successfully solve environment, we used an episodic task approach where each episode is run for a certain number of time-steps. By default, we run 5000 episodes each running until the environment decides to stop the episode.

At each time-step the agents performs an action, receive a reward, next state and an episode conclusion flag. The agents then use this information to learn a mapping between an observation from the state-space and an instance of the action-space composed of 2 continuous variables.

The learning process uses the Multiple Agent Deep Deterministic Policy Gradient (MADDPG) reinforcement learning algorithm (**Lowe et al., 2018**) which, although not a pure actor-critic algorithm, does use an actor to retrieve actions to use in a given local state and a centralised critic to evaluate such state-action pairs. Both the actor and critic are implemented as neural networks using the Pytorch library (more details about the implementation used can be found later in this document). Each agent has an independent actor and critic network.

In order to have a rich set of experiences from the 20 different environment instances and break any temporal correlations within each environment instance, our approach uses a circular shared replay buffer from which we uniformly sample mini-batches of experiences **(Lin, Long-Ji, 1992)** and use them to train the actor and critic neural networks of each agent. An experience is a tuple composed of an observation state matrix (24 variables x 2 agents), an action vector (2 variables x two agents), a reward (a real number x two agents), the next state matrix (24 variables x two agents) and an episode conclusion flag (a Boolean value x two agents), created at each time-step.

At each time-step the circular replay buffer is updated with new experiences from the agents. This level of experience sampling frequency guarantees that the agents are using a diverse enough set of observation states during learning, which makes it more likely to explore and be able to generalise to unseen observation states which may arise in other environments.

After these new experiences are added to the replay buffer, the agent decides if it is time to learn. If the replay buffer already contains sufficient experiences i.e. the total number of entries is higher that the experiences batch size used for learning (a hyper-parameter), then each agent can retrieve a mini-batch of experiences from the replay buffer and learn from them. In order to also experiment with the number of learning steps i.e. learning only from time to time vs learning at each time-step, a new controlling hyper-parameter called `steps_to_learning` was introduced.

At the end of each time-step, the rewards obtained by the agents are accounted for. At the end of each episode the maximum score across the 2 agents is computed and added to a circular buffer holding the average scores over the last consecutive 100 episodes. When the average score across the last consecutive 100 episodes is higher than 0.50 the task is considered solved and the training process is halted.

Finally, the system saves the learned models of each agent to disk so that they can later be used during a test phase where the best learned model (the one that yielded the highest score in the lowest number of episodes) is used in an environment in test mode.

4. Acting, Evaluating, Stabilising, Exploring and Learning

As mentioned above, our agent must be able to interact with the different environment instances, collect experiences and learn from a diverse set of interactions, with the objective of building an internal mapping informing the agents about which actions are appropriate for each locally observed state which is the essence of a reinforcement learning algorithm: "map situations to actions" **(Richard S. Sutton and Andrew G. Barto, 2018)**.

Since we are using the MADDPG algorithm for our implementation we will, for each of the agent, be using a local actor neural network and a centralised critic neural network to act and learn about how good our actions were in a given state. The local actor will learn how to act while the centralised critic trained will the states and actions taken by the actors of both agents, will learn to evaluate the actor's actions. The critic will therefore inform the local actor of each agent on how good its action was while the actions of the local actor of each agent will inform the centralised critic on the result of the actions taken.

4.1. Learning to Act: Actor Neural Network

The actor neural network is interested in learning the best action to use in a given observation state for both agents. This corresponds to learning a deterministic policy parametrised by each agents' actor neural networks parameters (weights). Each agent local actor's neural network has the following architecture:

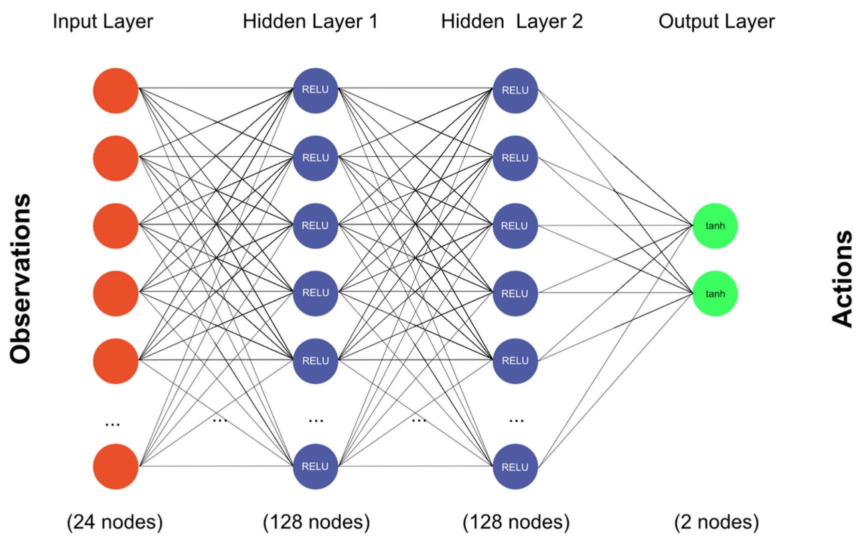


Figure 1 - Actor Neural Network Architecture

- ❖ An input layer which accepts 24 inputs corresponding to an observation state provided by each of the 2 agents.
- ❖ An initial fully-connected hidden layer with 128 nodes followed by a non-linearity which in our case is a Rectified Linear Activation (RELU).
- ❖ A second fully-connected hidden layer also with 128 nodes, followed by a non-linearity which in our case is a Rectified Linear Activation (RELU).
- ❖ Finally, an output layer with 2 nodes one for each of the action-space variables followed by a $\tanh x$ activation function since we are using a continuous action-space with each variable assuming a value between -1 and 1:

$$\tanh(\mathrm{x}) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- ❖ Batch-normalisation layers are included before each hidden layer non-linearity including before the output layer's $\tanh x$ non-linearity.

4.2. Learning to Evaluate State-Actions: Critic Neural Network

The centralised critic's neural network of each agent is similar but has noticeable changes in the input and output layers as its main objective is to learn the Q-value of a state-action pair so that it can evaluate the actor's actions during learning:

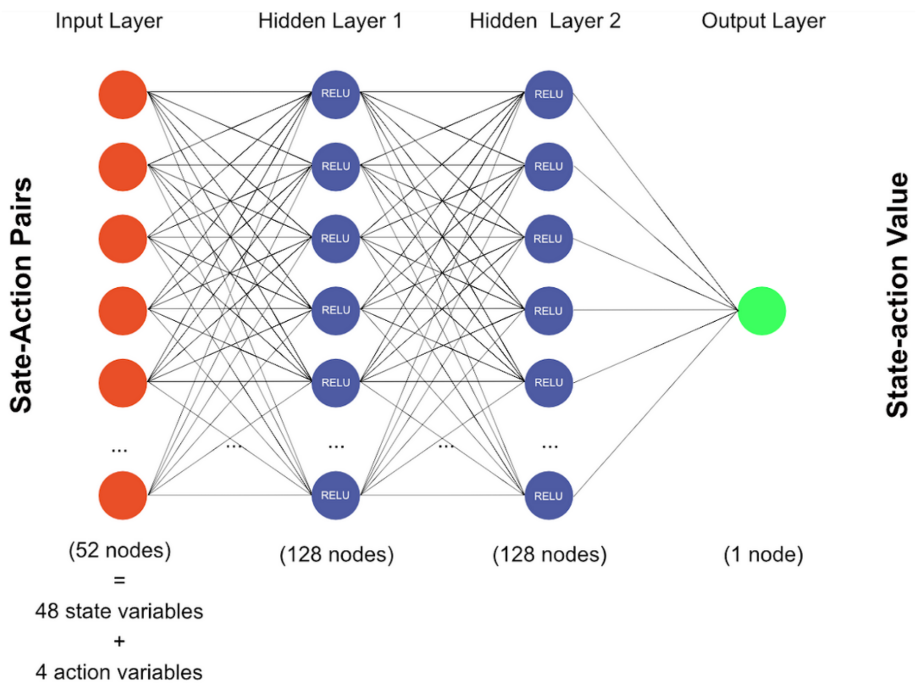


Figure 2 - Critic Neural Network Architecture

- ❖ An input layer which accepts 52 inputs corresponding to an observation state provided by each of the 2 agents (24 x 2 = 48 variables) and the actions taken by each of them (2 x 2 = 4 variables), in this order.
- ❖ An initial fully-connected hidden layer with 128 nodes followed by a non-linearity which in our case is a Rectified Linear Activation (RELU).
- ❖ A second fully-connected hidden layer also with 128 nodes, followed by a non-linearity which in our case is a Rectified Linear Activation (RELU).
- ❖ Finally, an output layer with 1 node is used to output the evaluation of the critic network in regard the inputted observation states and taken action.
- ❖ Batch-normalisation layers are included before each hidden layer non-linearity

4.3. Stabilizing Learning: Actor and Critic Online and Target Networks

As mentioned in **(Mnih, Volodymyr, et al., 2015)**, reinforcement learning using neural networks as non-linear function approximators to state-action functions are known to be difficult to train and stabilise, mainly due to correlations between the observations experienced during training and used during learning. We build on this idea and also introduce a second Q-network named the target network that is used to estimate the target values used in the Bellman equation, which are subsequently used to reduce the error between the estimated values provided by the online Q-network being trained and the values provided by this target Q-network.

Contrary to the online Q-network parameters which are being updated at every time-step during training, the target Q-network parameters are updated only periodically with the old parameters from the online Q-network. We therefore use two hyper-parameters `actor_target_network_parameter_update_steps` and critic_`target_network_parameter_update_steps` to control the number of time-steps between updates of the target networks with the parameters of the online networks of the actor and critic neural networks, respectively, this for each agent.

On top of the number of steps used to update the target network, we also experimented with a hard and soft-update rule. By hard-update we mean that the entire value of a parameter in the online network is copied to the target network, and by soft-update we mean incorporating a portion of the online parameter into the target parameter. We therefore control the level of change to parameter values in the target neural networks of the actor and critic with another hyper-parameter `tau=1e-2`. This parameter determines how much of the value of a parameter in the online network is passed into the same parameter in the target network.

4.4. Handling Co-variate Shift by Using Batch-Normalisation Layers Inside the Neural Networks

From typical machine learning we know that a pre-processing step that normalises inputs values before feeding them to a neural network is a good idea. This idea is important because different dimensions in the input space can have different magnitudes in their values. Imagine a dimension holding temperatures ranging from -10 C to +10 C, and another dimension holding speeds of 0 to 300 km per hour. Simply speaking, normalisation puts the values across all dimensions on the same scale.

The same issue can happen to the weights of a neural network, where large weights propagate through the layers of the network, overwhelming the remaining weights making the network unstable possibly leading to phenomena such as the famous exploding gradients. Normalising the weights of the network seems to be a good idea. Pytorch allows for the inclusion of such batch-normalisation layers directly into the neural network architecture (https://pytorch.org/docs/stable/_modules/torch/nn/modules/batchnorm.html), so this improvement could be fairly simple to implement.

Therefore, to make our model robust to changes in the distribution of data used to train the model in earlier layers we have introduced batch normalisation into our actor and critic networks in each hidden layer and before each non-linearity. This technique is mentioned in this paper **(Ioffe, Sergey and Szegedy, Christian, 2015)**.

Another consequence of batch normalisation is regularisation. This effect happens because the mean and variance values used to normalise the inputs into each layer are extracted from each mini-batch and not from the entire replay buffer used to train the neural network. This introduces some noise to each hidden layer non-linearity functions and hence the mild regularisation effect of batch normalisation.

4.5. Exploring by Adding Noise to Actions

The trade-off between exploration and exploitation is an always present issue in any machine learning algorithm. Reinforcement learning is not an exception and care must be taken to make sure that the agent explores sufficiently at early stages of learning and exploits that gathered knowledge at later stages. By following this strategy we hope to equip the agent with the capability of handling unseen states and generalise well, while at the same time using its "insights" in known state-space regions.

Because we are using MADDPG which is an off-policy algorithm we used the concept of introducing noise in the actions returned by the actors of each agent when interacting with the environment instances. Another reason for this was the fact that we are using a continuous action-space and therefore cannot use ε-greedy approaches.

The solution used was an Ornstein–Uhlenbeck stochastic process which is a Gaussian process that describes the velocity of a Brownian particle under the influence of friction **(Uhlenbeck, George E. and Ornstein, Leonard S., 1930)**.

By using this process, we add noise to the each of the 2 continuous variables of the actions chosen by the actor's deterministic policy being learned, forcing the use of an off-policy strategy for the exploration of the action-space and consequently exploration of the state-space.

We experimented with both a constant and a linear decaying level of noise introduced in the actor actions chosen by each agent. The two different noise signals look as follows:

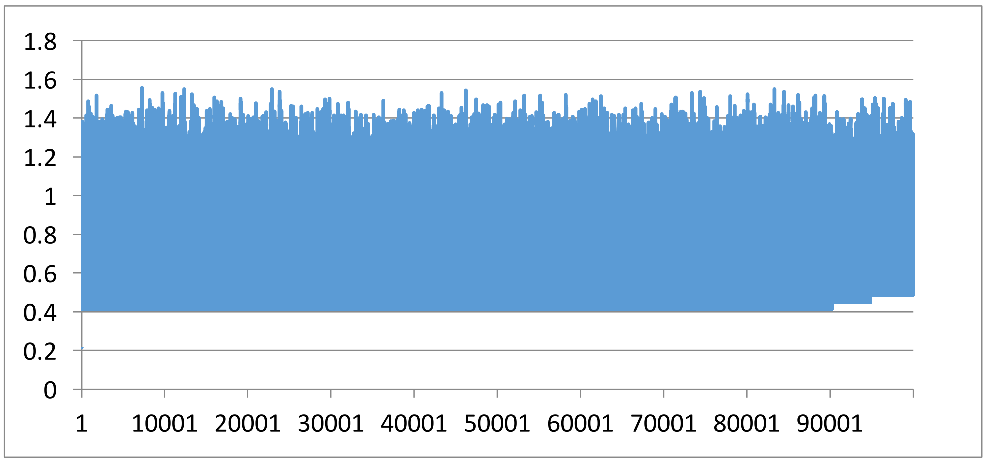


Figure 3 - Constant Noise Signal

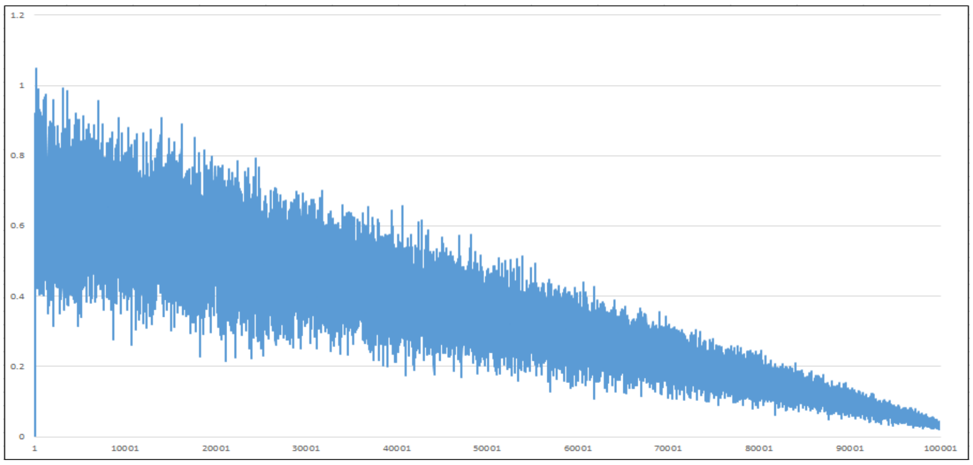


Figure 4 - Linear Decaying Noise Signal

4.6. Improving Learning by Making Sure Neurons Are Alive

To make sure that our neurons are always alive i.e. they output a value different from 0, we have also experimented with leaky ReLU activation functions. The leaky ReLU is implemented by having a small negative slope (e.g. 0.01) so that when the input to the ReLU is negative the neuron still outputs a small value.

4.7. Optimisers, Loss Functions and Hyper-Parameters

To train the Q-learning neural networks we used the Adam optimisation algorithm **(Kingma et. al., 2014)** with learning rates for the actor and critic, `actor_lr=1e-4` and `critic_lr=1e-3`, respectively.

We also used different functions to calculate the loss between the online networks' predictions and target values from the target networks, which we used for the backpropagation step. After being set, the same loss function was used to train both the actor and critic neural networks. We tested our learning algorithm with the following functions:

4.7.1. Mean Squared Error (https://pytorch.org/docs/stable/nn.html#mseloss)

The mean squared loss takes as inputs a Y vector of observations and a Y' of predictions for such observations and outputs the sum of the squared differences of all elements:

$$loss\ (Y, Y') = \frac{1}{n}\sum_{i=1}^{n}(Y_i - Y'_i)^2$$

4.7.2. Smooth L1 Loss (https://pytorch.org/docs/stable/nn.html#smoothl1loss)

To avoid the algorithm's potential exposure to outliers and the problem of exploding gradients, we explored the possibility of using a different loss function called Smooth L1/Huber Loss also available in the Pytorch libraries. This function is defined as follows:

$$Loss(Y, Y') = \frac{1}{n}\sum_{i=1}^{n} z_i$$

$$z_i = \begin{cases} \delta\ (Y_i - Y'_i)^2, & |Y_i - Y'_i| < 1 \\ |Y_i - Y'_i| - \delta, & otherwise \end{cases}$$

This loss function behaves as a mean squared error if the error is below a certain threshold δ (in our case we used $\delta = 0.5$) and as a mean absolute error otherwise. The main advantage of this approach is that large errors provoked by outliers are "toned-down" leading to smoother errors, smoother gradients and therefore smoother learning processes.

5. The Learning Algorithm

The following table puts forward the detailed steps used during training and learning, mapping where relevant the Python code used with the mathematical formulations behind it, together with a brief description of what the code is actually doing.

MADDPG		
Description	**Python Code**	**Mathematical Formulation**

During an episode, at each time-step t select action a_i from each agent i according to each agent i observation O_i using the agent's current policy μ_{θ_i} and exploration process $\aleph_i$.	`actions = agents_handler.act(states, step)`
	`agent_actions_tensor = self.actor_net[i](states_tensor[i, :].unsqueeze(0))`
	`agent_actions_noisy_tensor=self.ou_noise.get_action(agent_actions_tensor, step)`
		$\mu_{\theta_i} \rightarrow actor's\ current\ policy$
		$a_i = \mu_{\theta_i}(O_i) + \aleph_i$

Execute the set of actions $a = (a_1, a_2, \dots, a_N)$ from all agents in the environment and collect reward r, next state x' and a dones flag for all agents.	`env_info = env.step(actions)[brain_name]`
	`rewards = np.array(env_info.rewards)`
	`next_states = env_info.vector_observations`
	`dones = np.array(env_info.local_done)`
		$r, x', dones$

Store this experience in the shared replay buffer.	`self.replay_buffer.push(states, actions, rewards, next_states, dones)`	$(x, a, r, x', dones)$

For each agent i sample a mini-batch of `batch_size` samples from the shared replay buffer and learn from it.	`experiences = self.replay_buffer.sample(self.batch_size)`
	`states, actions, rewards, next_states, dones = experiences`
	`states_tensor      = torch.FloatTensor(states).to(device)`
	`actions_tensor     = torch.FloatTensor(actions).to(device)`
	`rewards_tensor     = torch.FloatTensor(rewards).unsqueeze(2).to(device)`
	`next_states_tensor = torch.FloatTensor(next_states).to(device)`
	`dones_tensor                                                      =`
	`torch.FloatTensor(np.float32(dones)).unsqueeze(2).to(device)`
		(x^j, a^j, r^j, x'^j)

For each agent compute the target values y^j of all samples j in the mini-batch, using the critic's target network and passing as input all agents' next states x'^j of this mini-batch and the best actions returned by the different actors' target network. The actor's target network uses the mini-batch's current observation o_k^j from each agent k.	`actor_actions = []`
	`for j in range(self.num_agents):`
	`  actor_action = self.actor_net[j](states_tensor[:, j, :])`
	`  actor_actions.append(actor_action)`
	`critic_evaluations = self.critic_net[i](all_agents_states, actor_actions)`
	`target_next_actions = []`
	`for j in range(self.num_agents):`
	`  target_next_action = self.actor_target_net[j](next_states_tensor[:, j,`
	`:])`
	`  target_next_actions.append(target_next_action)`
	`target_values = self.critic_target_net[i](all_agents_next_states,`
	`target_next_actions.detach())`
		$Q_i^{\mu'} \rightarrow centralised\ critic\ target\ network\ for\ each\ agent\ i$
		$r_i^j \rightarrow reward\ obtained\ from\ agent\ i\ in\ sample\ j$
		$x'^j \rightarrow next\ state\ in\ sample\ j$ for all agents
		$a'_1, \ldots, a'_N \rightarrow next\ actions\ for\ each\ agent\ i$
		$o_k^j \rightarrow$ Observation of each agent in sample j
		$\mu'_k(o_k^j) \rightarrow current\ policy\ of\ each\ agent\ k$
		$y^j = r_i^j + \gamma\, Q_i^{\mu'}(x'^j, a'_1, \ldots, a'_N)\,\|_{a'_k = \mu'_k(o_k^j)}$

Update the <u>critic</u> by minimising loss $\mathcal{L}(\theta_i)$.	`rewards_tensor = rewards_tensor[:, i, :]`
	
	`dones_tensor = dones_tensor[:, i, :]`
	
	`expected_values = rewards_tensor + ((gamma * target_values) * (1 - dones_tensor))`
	
	`critic_state_action_values = self.critic_net[i](all_agents_states, all_agents_actions)`
	
	`critic_loss = self.critic_loss(critic_state_action_values, expected_values.detach())`
	
	`self.critic_optimizer[i].zero_grad()`
	
	`critic_loss.backward()`
	
	`torch.nn.utils.clip_grad_norm_(self.critic_net[i].parameters(), 1)`
	
	`self.critic_optimizer[i].step()`
		$\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\mu}\left(x^j, a_1^j, \dots, a_N^j\right)\right)^2$

Update the actor using the sampled policy gradient.	`critic_evaluations = self.critic_net[i](all_agents_states, actor_actions)`
	
	`actor_loss = -critic_evaluations.mean()`
	
	`self.actor_optimizer[i].zero_grad()`
	
	`actor_loss.backward()`
	
	`torch.nn.utils.clip_grad_norm_(self.actor_net[i].parameters(), 1)`
	
	`self.actor_optimizer[i].step()`
		$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i\left(o_i^j\right) \nabla_{\theta_i} Q_i^{\mu}\left(x^j, a_1^j, \dots, a_i, \dots, a_N^j\right) \big\vert_{a_i=\mu_i\left(o_i^j\right)}$

Update the actor and critic target networks of each agent i at C time-steps within each episode.	`if len(self.replay_buffer) > self.batch_size:`
	
	  `if step % self.critic_target_network_parameter_update_steps == 0:`
	
	    `for i in range(self.num_agents):`
	
	      `soft_update(self.critic_net[i], self.critic_target_net[i])`
	
	  `if step % self.actor_target_network_parameter_update_steps == 0:`
	
	    `for i in range(self.num_agents):`
	
	      `soft_update(self.actor_net[i], self.actor_target_net[i])`
		$\theta'_i = \tau\, \theta_i + (1-\tau)\, \theta'_i, with\ \tau \ll 1$

6. Results

We conducted several experiments using the Unity environment. All experiments were conducted on a MacBook Pro 13" Retina (early 2015).

6.1. List of Hyper-parameters

The following hyper-parameters were used during learning:

Hyper-parameter	Python Code Variable Name	Default Value Used
Maximum size of the replay buffer used to store the agent's experiences with the environment.	replay_buffer_size	100 000
The size of the batch size used during the learning step.	batch_size	512
The size of the two hidden layers of the critic and actor online and target neural networks.	hidden_dim	128
The learning rate used in the critic's Adam stochastic optimiser during the update step.	critic_lr	1e-3
The learning rate used in the actor's Adam stochastic optimiser during the update step.	actor_lr	1e-3
The number of episodes used to train the agent.	number_of_episodes	5000
Discount rate applied to future rewards.	gamma	0.95
The factor used to update the actor and critic target neural networks with parameters from the corresponding online networks.	tau	1e-2
Number of steps between training the actor and critic networks.	steps_to_learning	1
Number of steps between updates of the critic's target network with parameters from the online network	critic_target_network_parameter_update_steps	20
Number of steps between updates of the actor's target network with parameters from the online network.	actor_target_network_parameter_update_steps	20

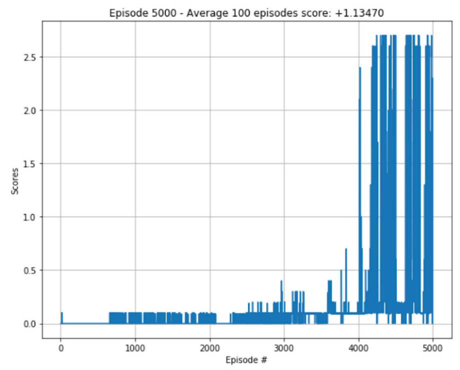
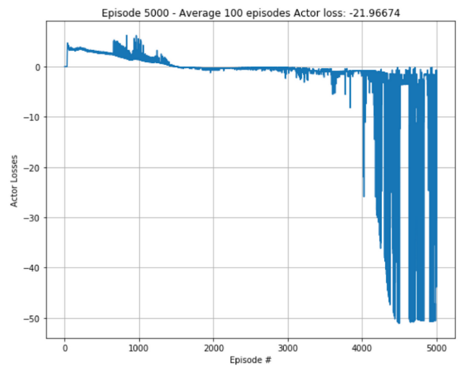
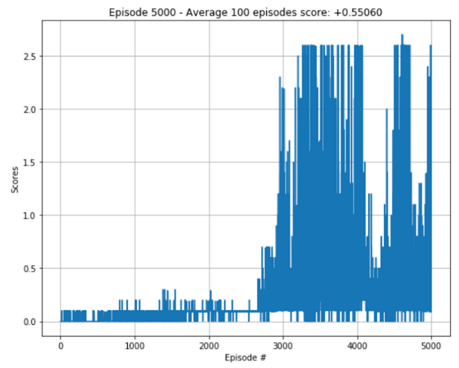
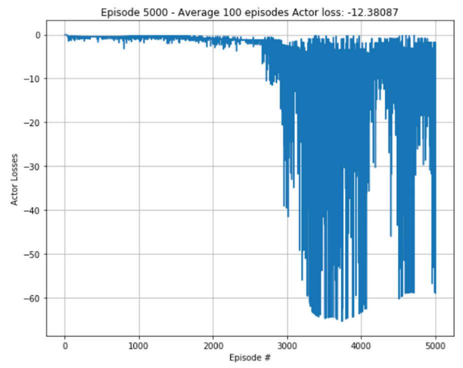
Minimum value used to clamp the expected values computed by the critic's target network.	expected_values_min_clamp	-1
Minimum value used to clamp the expected values computed by the critic's target network.	expected_values_max_clamp	1
Ornstein–Uhlenbeck Long-term Reverting Mean.	mu	0.0
Ornstein–Uhlenbeck Speed of Reversion.	theta	0.15
Ornstein–Uhlenbeck Maximum Level of Noise.	max_sigma	0.2
Ornstein–Uhlenbeck Minimum Level of Noise.	min_sigma	0.01
Ornstein–Uhlenbeck Number of Steps for Decay Calculations.	decay_period	100 000

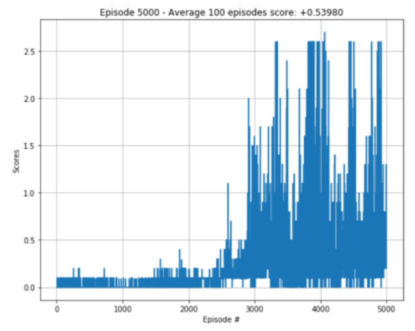
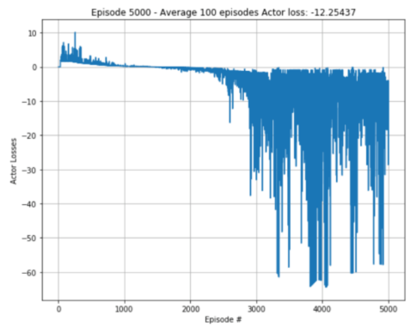
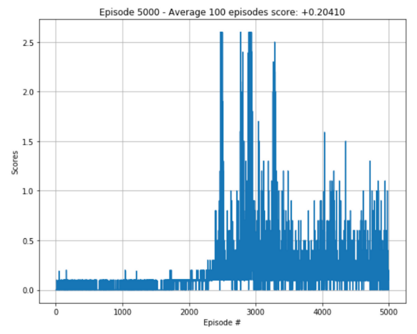
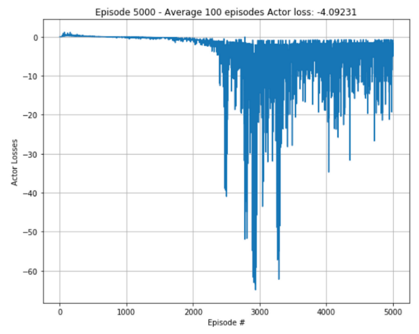
6.2. List of Experiments and Hyper-Parameters Used

During our experiments we used the two environments and several combinations of hyper-parameters. The following sections report on the experiments conducted as well as on the results obtained. The following table puts forward our results in training such an agent together with the episodic scores:

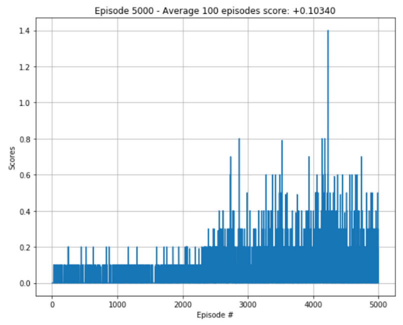
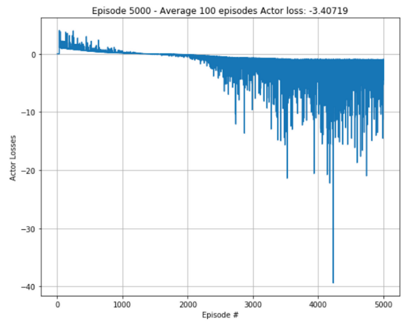
Experiment #	Explanation	Hyper-Parameters Used/Changed

1	This experiment sets up our baseline from which we should improve by changing the algorithm and tweaking the hyper-parameters.	number_of_episodes = 5000
		batch_size = 512
		hidden_dim = 128
		steps_to_learning = 1
		actor_target_network_parameter_update_steps = 20
		max_sigma = 0.2
		min_sigma = 0.01
		ReLU activation units
		MSE Loss Function
		No Normalisation in Hidden Layers
		Independent Critic and Actor Neural Networks

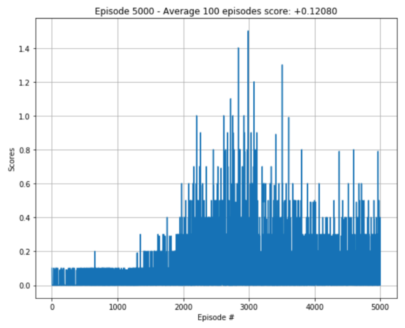
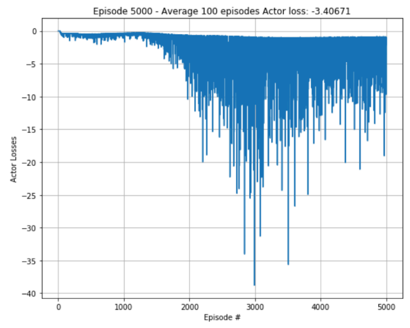
Experiment #	Explanation	Hyper-Parameters Used/Changed
	  `Environment resolved in 5000 episodes!`	
2	In this experiment we introduce normalisation in the hidden layers of the actor and critic neural networks before the non-linearity. The agents seem to be able to maintain higher scores for a larger number of consecutive episodes. A significant improvement.	Normalisation introduced in hidden layers of actor and critic.
	  `Environment resolved in 5000 episodes!`	

3	In this experiment we introduce a different loss function Smooth L1 Loss for the critic. No significant improvements were observed therefore we go back to using mean squared error function for the actor loss.	Smooth L1 Loss replaced MSE as the loss function for the critic.
	  `Environment resolved in 5000 episodes!`	
4	In this experiment we replace the previous ReLU activation function by a Leaky ReLU in an attempt to keep all nodes in the actor and critic networks alive. After 5000 training episodes the agents are not able to solve the environment during the final episodes. We therefore abandon this approach for subsequent training sessions.	Leaky ReLU replaces the previous ReLU activaton function in the actor and critic networks.
	 	

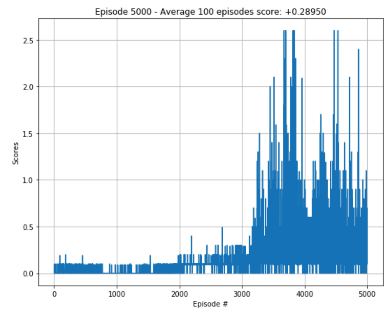
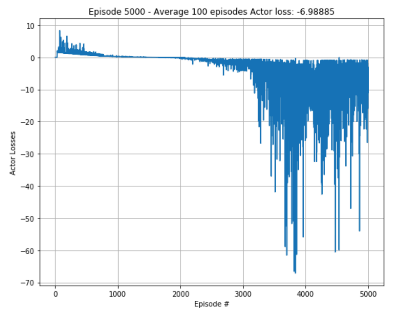
5	In this experiment we change our noise function parameters and do not linearly decay the noise signal in an attempt to have the agent exploring heavily throughout all training episodes.
	
	The results were quite discouraging with the agents not being able to cooperate and obtain high scores.
		theta=0.35
		
		max_sigma = 0.4
		
		min_sigma = 0.4

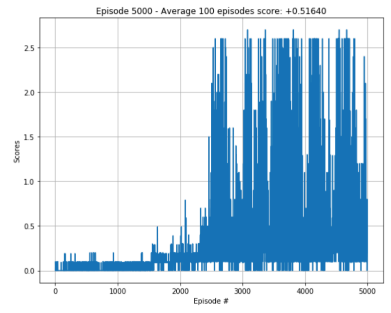
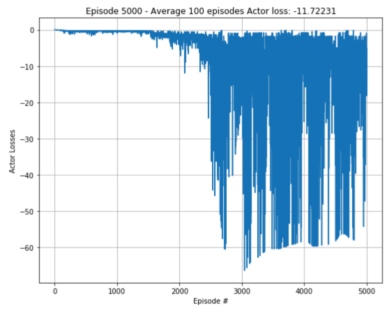
6	Given the weak results from the previous experiment and the need to have the agents explore more in order to find better solutions, we change our noise function parameters and do not linearly decay the noise signal.
	
	The results were definitely not encouraging. It seems that the agents are neither properly exploring at the beginning nor exploiting towards the end of training. Perhaps we are still adding too much noise?
		theta=0.15
		
		max_sigma = 0.3
		
		min_sigma = 0.3

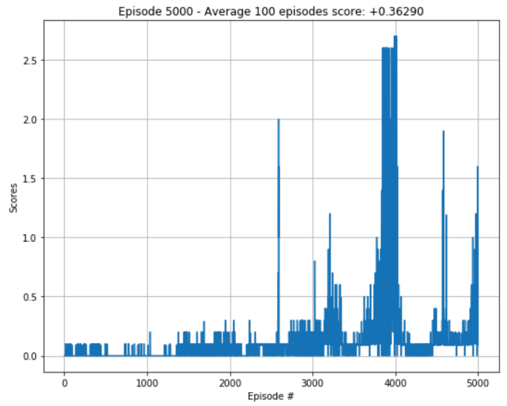
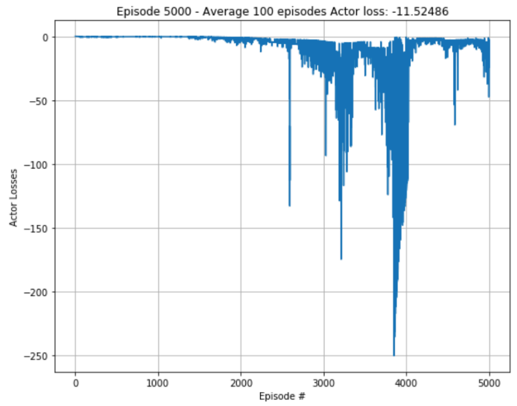
	 	

7	In this experiment, we reduce the level of noise introduced into the actions of the agents during training, without decaying the noise signal.
	
	The results were better than the previous experiment, however the agents were unable to sustain high scores during the last 100 episodes.
		theta=0.15
		
		max_sigma = 0.10
		
		min_sigma = 0.10

8	In this experiment, we further halve the level of noise introduced into the actions of the agents during training without decaying it. We also keep the expected values provided by the critic target neural network between -1 and 1.
	
	This is our best model during training. It achieved several the desired threshold several times during training and also during the last 100 episodes.
		theta=0.15
		
		max_sigma = 0.05
		
		min_sigma = 0.05
		
		expected_values=torch.clamp(expected_values, expected_values_min_clamp, expected_values_max_clamp)

	  `Environment resolved in 5000 episodes!`	

9	Finally, we use a single critic neural network feeding it all experiences from all agents, while at the same time keeping each actor networks independent from each other and therefore keeping the essence of the MADDPG algorithm. This approach did not yield good results and the agents were not able to achieve the threshold score of 0.50 over 100 consecutive episodes.	Single critic network for all agents and independent actor networks per agent.
	 	

7. Ideas for Future Work

7.1. Change the Exploration Implementation

Although the Ornstein–Uhlenbeck stochastic process **(Uhlenbeck, George E. and Ornstein, Leonard S., 1930)** used did help with the exploration of the action-space, the external noise it introduces is not conditioned to the observed state and may lead to inconsistent exploration in our off-policy approach (different actions for the same state), especially if the same state is present in the sample of experiences taken from the replay buffer and used to train our actor neural network.

A technique that addresses the above problem is proposed in **(Matthias Plappert et al., 2018)** and entails introducing noise directly into the actor's neural network parameters affecting the policy being learned. From our tests we can clearly see that the agents do not cooperate when in certain regions of the state space. This may be indicative of a policy that only performs well in certain cases i.e. a sub-optimal policy. Parameter noise may solve this problem by allowing the agents to escape local optima and find a better more flexible policy during training.

7.2. Replace the Shared Replay Buffer by a Shared Prioritised Replay Buffer

Finally, we would have like to further enhance our shared replay buffer with a prioritised replay buffer using ideas from this very interesting paper **(Tom Schaul et al., 2016)**.

In essence a prioritised replay buffer replays important experiences identified during learning by using the TD-error as the criteria to assess how important an experience is for learning, with large errors being deemed more important for learning.

This intuition together with a technique that combines stochastic prioritisation (which introduce some level of randomness in the sampling process) and a bias term duly corrected through the use of annealed importance-sampling weights makes a prioritised replay buffer an interesting candidate to improve the learning process for this specific task.

8. References

- Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning" Nature 518.7540 (2015): 529-533.
- Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, Igor Mordatch "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments" arXiv:1706.02275v3 (2018).
- Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra. "Continuous control with deep reinforcement learning" arXiv:1509.02971 (2016).
- Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, and Riedmiller, Martin. "Deterministic policy gradient algorithms". In ICML, 2014.
- Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, "Prioritised Experience Replay" arXiv:1511.05952v4 (2016).
- Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization" arXiv preprint arXiv:1412.6980 (2014).

- Uhlenbeck, George E. and Ornstein, Leonard S. "On the theory of the brownian motion." Physical review, 36(5):823, 1930.
- Matthias Plappert et al. "Parameter Space Noise for Exploration" arXiv:1706.01905v2 [cs.LG] (2018).
- Richard S. Sutton and Andrew G. Barto, Reinforcement Learning An Introduction (Second Edition), 2018.
- Ioffe, Sergey and Szegedy, Christian. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". arXiv preprint arXiv:1502.03167, 2015.