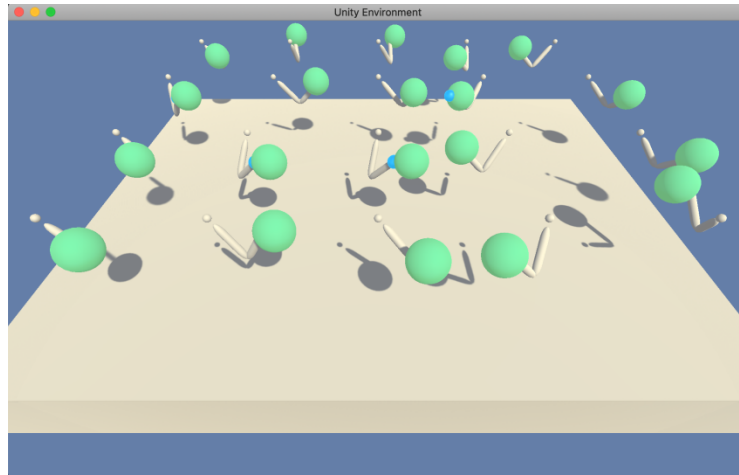


# Continuous Control Project

Udacity Deep Reinforcement Learning Nanodegree

António Soares

November 12, 2018



## 1. Project Description

The current project attempts to solve a continuous learning task using a model-free off-policy deep reinforcement learning algorithm called DDPG – Deep Deterministic Policy Gradient, (**Lillicrap et al., 2016**) that extends the work on Deterministic Policy Gradient algorithm (**Silver et al., 2014**) using recent successes from Deep Q-Learning (**Mnih et al., 2013; 2015**) and uses an actor and a critic neural network to decide on which actions to take and evaluate them, respectively.

The task consists of a double-jointed arm controlled by the agent that must learn how to move the arm to target locations produced by the environment. The environment provides a reward of +0.1 at each time-step if the agent is capable of keeping the arm's hand at a target location selected by the environment.

The goal of the agent is therefore to maintain the arm's hand at the target location at all times during all time-steps during the course of a given episode so as to maximise its total reward.

## 2. Environment

An adapted version of the [Reacher](#) environment is used in the project. Observation states are composed of 33 variables which correspond to physical quantities such as the double-jointed arm's position, rotation, velocity and angular velocities of each joint. Each variable is a number between -1 and 1. The Python Numpy representation of an observation state looks like this:

```
[ 0.00000000e+00 -4.00000000e+00  0.00000000e+00  1.00000000e+00
```

```

-0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
 1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  5.75471878e+00 -1.00000000e+00
 5.55726624e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
-1.68164849e-01 ]

```

The action space is composed of a vector with 4 continuous variables which correspond to the torque applicable to each of the harm's joints. Each variable is a real number between -1 and 1. The Python Numpy representation of an action vector looks like this:

```
[-1.0  1.0 -0.9999121  1.0]
```

In our project we have focused on the 20-agents' version of the adapted [Reacher](#) environment. This environment's version accepts a matrix of 20 different actions and returns 20 different rewards, 20 next observation states and a set of 20 episode-conclusion flags, one for each environment instance respectively.

Using the 20-agents' version, the task is considered solved when the average score obtained across all 20 environment instances is greater than 30 over 100 consecutive episodes.

### 3. High-level Algorithm

To train our agent and successfully solve the 20-agents' environment, we used an episodic task approach where each episode is run for a certain number of time-steps. By default we run 1000 episodes each with 1000 time-steps.

We started by using 20 different agents one for each of the 20 environment instances, but after several experiments decided on training a single agent that uses all 20 environment instances provided by the 20-agents' version of the [Reacher](#) environment, as this approach was faster to train and yielded much better results.

At each time-step the agent performs an action, retrieves a reward, next state and an episode conclusion flag from each of the 20 environments. It then uses this information to learn a mapping between an observation from the state-space and an instance of the action-space composed of 4 continuous variables.

The learning process uses the Deep Deterministic Policy Gradient (DDPG) reinforcement learning algorithm (**Lillicrap et. al., 2016**) which, although not a pure actor-critic algorithm, does use an actor to retrieve actions to use in a given state and a 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).

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. An experience is a tuple composed of an observation state matrix (33 variables), an action vector (4 variables), a reward (a real number), the next state matrix (33 variables) and an episode conclusion flag (a Boolean value), created at each time-step.

At each time-step the circular replay buffer is updated with 20 new experiences from the 20 different environment instances. This level of experience sampling frequency guarantees that the agent is 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. Of course we assume that the 20 environment instances provide experiences that are themselves sufficiently different from each other. Running a training session with the Unity environment graphical interface active did confirm this assumption: the different 20 environment instances are actually locating the targets at different places and moving them at different speeds.

After these 20 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 than the experiences batch size used for learning (a hyper-parameter), then the 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 agent from each of the 20 environment instances are accounted for. At the end of each episode an average score across all 20 environment instances 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 30 the task is considered solved and the training process is halted.

Finally, the agent saves the learned model to disk so that it 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 agent about which actions are appropriate for each 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 DDPG algorithm for our implementation, we will be using an actor neural network and a critic neural network to act and learn about how good our actions were in a given state. The actor will learn how to act while the critic will learn to evaluate the actor's actions. The critic will therefore inform the actor on how good its action was while the actor will inform the critic on the result of 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. This corresponds to learning a deterministic policy parametrised by the actor's neural network's parameters (weights). Our actor's network has the following architecture:

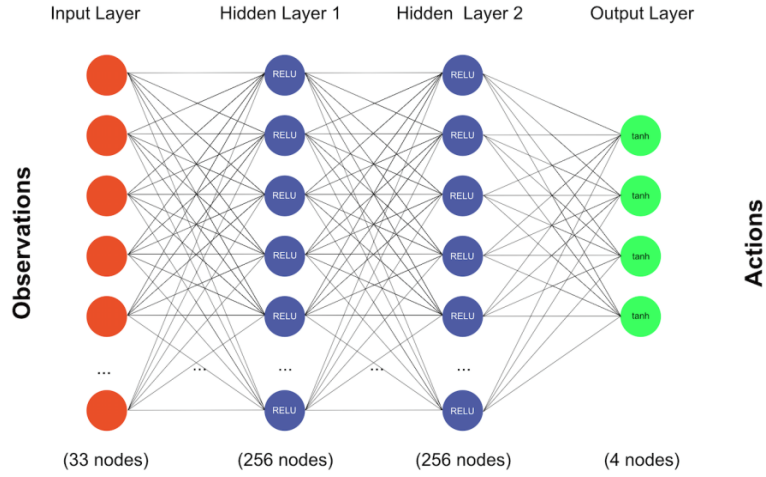


Figure 1 - Actor Neural Network Architecture

- ❖ An input layer which accepts 33 inputs corresponding to an observation state provided by each of the 20 environment instances.
- ❖ An initial fully-connected hidden layer with 256 nodes followed by a non-linearity which in our case is a Rectified Linear Activation (RELU).
- ❖ A second fully-connected hidden layer also with 256 nodes, followed by a non-linearity which in our case is a Rectified Linear Activation (RELU).
- ❖ Finally, an output layer with 4 nodes one for each of the action-space variables followed by a  $\tanh$  activation function since we are using a continuous action-space with each variable assuming a value between -1 and 1:

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

- ❖ No batch-normalisation layers were included in this baseline architecture.

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

The critic's neural network 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:

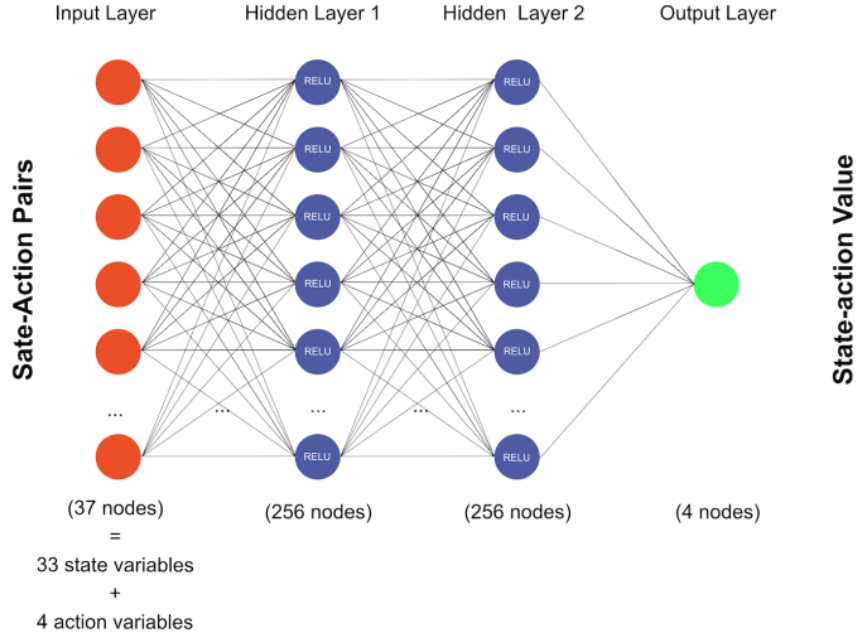


Figure 2 - Critic Neural Network Architecture

- ❖ An input layer which accepts 37 inputs corresponding to an observation state provided by each of the 20 environment instances (33 variables) and the action taken (4 variables).
- ❖ An initial fully-connected hidden layer with 256 nodes followed by a non-linearity which in our case is a Rectified Linear Activation (RELU).
- ❖ A second fully-connected hidden layer also with 256 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 state and action.
- ❖ No batch-normalisation layers were included in this baseline architecture.

#### 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.

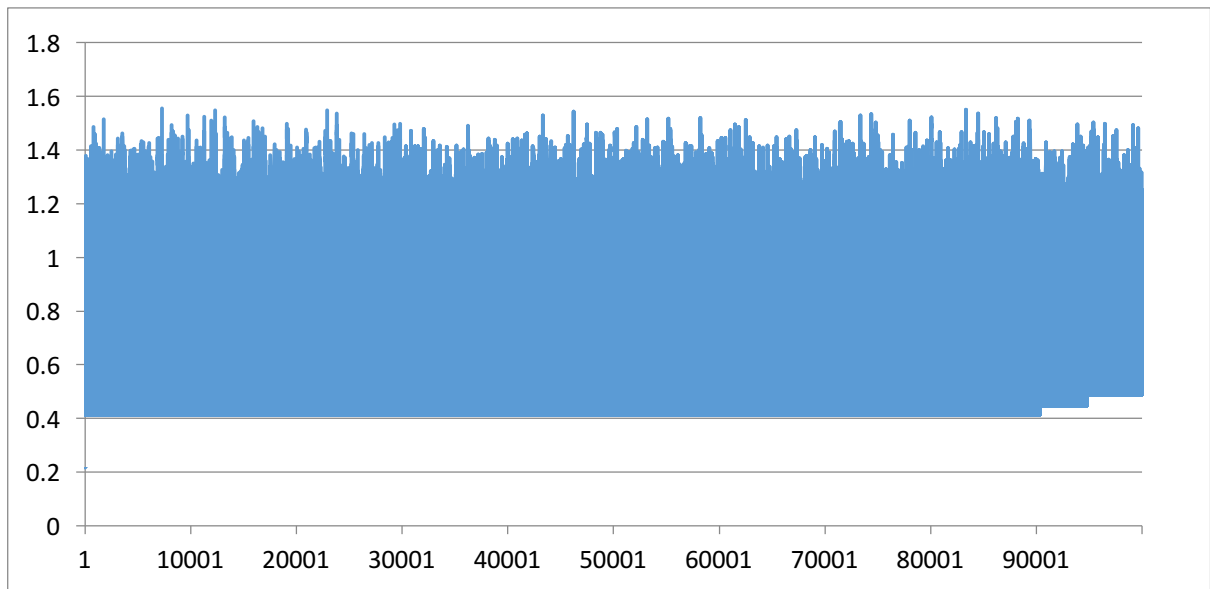
On top of the number of steps used to update the target network, we also experimented with a hard and soft-update rule. By hard-update we mean that the entire value of a parameter in the online network is copied to the target network, and by soft-update we mean incorporating a portion of the online parameter 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. Exploring by Adding Noise

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 DDPG which is an off-policy algorithm we used the concept of introducing noise in the actions returned by the actor 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  $\epsilon$ -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 4 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. The level of noise introduced looks as follows:



#### 4.5. 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:

##### 4.5.1. 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:

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

##### 4.5.2. 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  $\delta$  (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.

Description	Python Code	Mathematical Formulation
At each time-step $t$ select action $a_t$ according to the actor's best policy so far: $\theta^\mu$ .	<code>actions = agents_handler.act(states, step)</code>	$a_t = \mu(s_t   \theta^\mu) + \aleph$
Execute action $a_t$ in the environment and receive reward $r_t$ and next state $s_{t+1}$ .	<code>env_info = env.step(actions)[brain_name]</code> <code>rewards = env_info.rewards</code> <code>next_states = env_info.vector_observations</code>	$r_t, s_{t+1}$
Store this experience in the shared replay buffer.	<code>for state, action, reward, next_state in zip(states, actions, rewards, next_states):</code>  <code>self.replay_buffer.add(state, action, reward, next_state)</code>	$(s_t, a_t, r_t, s_{t+1})$
Sample a random mini-batch of experiences of size <code>batch_size</code> from the shared replay buffer and learn from it.	<code>experiences = self.replay_buffer.sample(self.batch_size)</code>  <code>states, actions, rewards, next_states = experiences</code>	$(s_i, a_i, r_i, s_{i+1})$
The actor's policy encoded in the principal network $\mu$ by its parameters $\theta^{Q\mu}$ is used to provide the best actions $a_i$ for each state $s_i$ in the mini-batch.	<code>actor_actions = self.actor_net(states)</code>	$act_i = \mu(s_i   \theta^\mu)$
The critic then returns its current best evaluations of each state-action pair selected by the actor:	<code>critic_evaluations = self.critic_net(states, actor_actions)</code>	$crit\_eval_i = Q(s_i, act_i   \theta^Q)$



Description	Python Code	Mathematical Formulation
A loss is then calculated by taking the mean of all evaluations returned by the critic	<code>actor_loss = -critic_evaluations.mean()</code>	$\frac{1}{batch\_size} \sum_{i=1}^{batch\_size} critic\_eval_i$
The next actions for the next states are estimated using the actor's target network.	<code>next_actions = self.actor_target_net(next_states)</code>	$a_{i+1} = \mu'(s_{i+1}   \theta^{\mu'})$
The target Q-values for the next states-action pairs indicated by the actor target network, are then evaluated using the critic's target network.	<code>target_values = self.critic_target_net(next_states, next_actions.detach())</code>	$target\_values_i = Q'(s_{i+1}, a_{i+1}   \theta^{Q'})$
Using the Bellman equation, we compute the discounted expected returns for all samples in the mini-batch:	<code>expected_values = rewards + (1.0 - dones) * gamma * target_values</code>	$y_i = r_i + \gamma \cdot target\_values_i$
We obtain the Q-values from the states and actions sampled in this batch of experiences	<code>critic_state_action_values = self.critic_net(states, actions)</code>	$critic\_values_i = Q(s_i, a_i   \theta^Q)$
To train the critic network we then calculate the critic loss between estimated critic values and the previously calculated critic expected values.	<code>critic_loss = self.critic_loss(critic_state_action_values, expected_values.detach())</code>	$\begin{aligned} \mathcal{L}(\theta^Q) \\ = \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} (y_i \\ - critic\_values_i)^2 \end{aligned}$

Description	Python Code	Mathematical Formulation
The target networks parameters are updated using a soft-update strategy as follows (reference to soft-update paper here)	<pre> if step % self.actor_target_network_parameter_update_steps == 0:     for target_param, param in zip(self.actor_target_net.parameters(), self.actor_net.parameters()):         target_param.data.copy_(target_param.data * (1.0 - tau) + param.data * tau)  if step % self.critic_target_network_parameter_update_steps == 0:     for target_param, param in zip(self.critic_target_net.parameters(), self.critic_net.parameters()):         target_param.data.copy_(target_param.data * (1.0 - tau) + param.data * tau) </pre>	<p>Actor:</p> $\theta^{\mu'} = \tau \theta^Q + (1 - \tau) \theta^{\mu'}, \text{ with } \tau \ll 1$ <p>Critic:</p> $\theta^{Q'} = \tau \theta^Q + (1 - \tau) \theta^{Q'}, \text{ with } \tau \ll 1$
Next we update the actor online network parameters $\theta^Q$ by taking a step on the Adam optimiser. For stability reasons we also normalise the gradients.	<pre> self.policy_optimizer.zero_grad() policy_loss.backward() torch.nn.utils.clip_grad_norm_(self.actor_net.parameters(), 1) self.policy_optimizer.step() </pre>	$\Delta \theta^{\mu}$ $= \alpha \cdot (y_i - \mu(s_i, a_i, \theta^{\mu})) \cdot \nabla_{\theta^{\mu}} \mu(s_i, a_i, \theta^{\mu})$
Finally we update the critic online network parameters $\theta^{\mu}$ by taking a step on the Adam optimiser. For stability reasons we also normalise the gradients	<pre> self.value_optimizer.zero_grad() value_loss.backward() torch.nn.utils.clip_grad_norm_(self.critic_net.parameters(), 1) self.value_optimizer.step() </pre>	$\Delta \theta^Q$ $= \alpha \cdot (y_i - Q(s_i, a_i, \theta^Q)) \cdot \nabla_{\theta^Q} Q(s_i, a_i, \theta^Q)$

## 6. Results

We conducted several experiments using the single and multiple agents Unity environments. Although we started with the single-agent environment, we quickly moved on to the most interesting multiple-agent environment which provided a more diverse set of experiences from which our agent could learn from. 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	1 000 000
The size of the batch size used during the learning step.	batch_size	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-4
The size of the two hidden layers of the critic and actor online and target neural networks.	hidden_dim	256
The number of episodes used to train the agent.	number_of_episodes	1000
The number of steps within each episode used to train the agent.	max_steps_per_episode	1000
Discount rate applied to future rewards.	gamma	0.99
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	critic_target_network_parameter_update_steps	1

from the online network		
Number of steps between updates of the actor's target network with parameters from the online network.	actor_target_network_parameter_update_steps	1
Minimum value used to clamp the expected values computed by the critic's target network.	expected_values_min_clamp	-np.inf
Minimum value used to clamp the expected values computed by the critic's target network.	expected_values_max_clamp	np.inf
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	0.3
Ornstein-Uhlenbeck Minimum Level of Noise.	min_sigma	0.1
Ornstein-Uhlenbeck Number of Steps for Decay Calculations.	decay_period	100 000

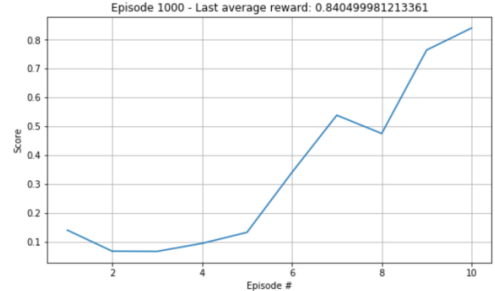
## 6.2. List of Experiments and Hyper-Parameters Used



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.



### 6.2.1. Single Agent Environment



We started by using the single agent environment to try to understand what could and couldn't work in terms of changes in the algorithm (e.g. introducing gradient normalisation) and its hyper-parameters values.

To get a quick view on the resulting scores our graphs show the average score across the last 100 episodes instead of the required score per episode which we will show in the multiple-agents section that follows.



Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S1	<p>This experiment sets up our baseline from which we should improve by changing the algorithm and tweaking the hyper-parameters.</p> <p>This first experiment although fast to run (only 28 minutes) was not successful as the agent was unable to learn and fulfil the task even after 1000 episodes.</p>	<p>number_of_episodes = 1000</p> <p>max_steps_per_episode = 100</p> <p>steps_to_learning = 1</p> <p>actor_target_network_parameter_update_steps = 1</p> <p>critic_target_network_parameter_update_steps = 1</p> <p>critic_loss = nn.MSELoss()</p>	 <p>28.142288637161254</p>

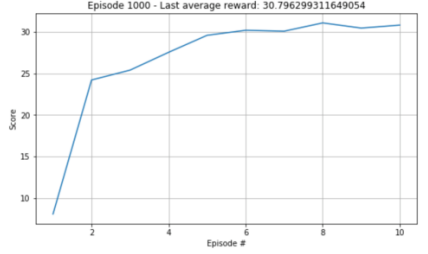
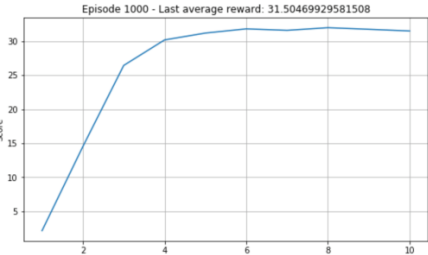
Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S2	<p>Since our agent was not really learning after 1000 episodes, we thought that perhaps not enough time was being given for it to interact with the environment. We therefore increased the number of steps from 100 to 1000 steps per episode.</p> <p>This change yielded interesting results and the agent was able to successfully fulfil the task. It did, however, consume a lot of time for training: 288 minutes.</p>	max_steps_per_episode = 1000	 <p>288.0143241008123</p>
S3	<p>Our previous success obtained by increasing the number of steps had to be put to the test, so in this training session we let the agent learn for a higher number of episodes.</p> <p>The results this time were not successful and after +- 4300 episodes of 1000 steps each, the agent was unable to learn and complete the task. We therefore abandon this approach.</p>	number_of_episodes = 4300	


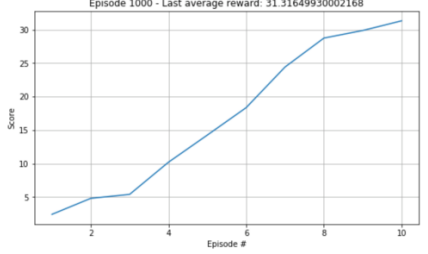
Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S4	<p>Given that experiment #3 had put into question the results of experiment #S2, we wondered if further increasing the number of time-steps per episode would yield better results.</p> <p>This was not the case and even after 1000 episodes of 2000 time-steps each the agent was unable to complete the task.</p>	<p>number_of_episodes = 1000</p> <p>max_steps_per_episode = 2000</p>	 <p>314.61609021822613</p>
S5	<p>In this experiment we start playing with the size of the batch used to train the agent. Perhaps the sampled experiences-set was just not big enough, so we increased it twofold from 128 to 256 experiences.</p> <p>This experiment yielded better results but the agent was still unable to perform the task at an acceptable level.</p> <p>The time needed for training using 1000 episodes also increased to 402 minutes!</p>	<p>max_steps_per_episode = 1000</p> <p>batch_size = 256</p>	 <p>402.0787978331248</p>


Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S6	<p>In this experiment we start focusing on other techniques other than changing the values of hyper-parameters. We go back to a batch size of 128 samples, and start by introducing gradient clipping on the critic network, after the backpropagation step.</p> <p>The resulting scores curve seems much more stable than before and the agent also seems to be learning faster. The agent is, however, still unable to complete the task.</p>	<p>batch_size = 128</p> <p><code>torch.nn.utils.clip_grad_norm_(self.critic_net.parameters(), 1)</code></p>	 <p>281.0037864327431</p>
S7	<p>In this experiment we repeat the previous technique of gradient clipping but this time on the actor. We also removed gradient clipping from the critic in order to understand the sensitivity of this experiment's change.</p> <p>No significant changes were noticed. In fact, there seems to be some instability in the scores. Moreover, the agent took longer to learn during the 1000 episodes and was unable to fulfil the task.</p>	<p>batch_size = 128</p> <p><code>torch.nn.utils.clip_grad_norm_(self.actor_net.parameters(), 1)</code></p>	 <p>322.0661984999975</p>




Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S8	<p>This experiment uses gradient clipping on the actor and critic networks.</p> <p>The obtained final score was higher than the last two experiments, and the agent seems to be learning at a steady rate. Nevertheless, the agent was still unable to complete the task.</p>	<p><code>torch.nn.utils.clip_grad_norm_(self.actor_net.parameters(), 1)</code></p> <p><code>torch.nn.utils.clip_grad_norm_(self.critic_net.parameters(), 1)</code></p>	 <p>293.92822151581447</p>
S9	<p>In this experiment we kept the gradient clipping on the actor and critic networks and started changing the hyper-parameters. This time we focused on the number of steps taken between updates of the actor and critic's target networks with the values from the corresponding online networks.</p> <p>Learning seems to be happening consistently and in stages but the agent is still not able to complete the task. The training time for 1000 episodes has also been reduced which was expected as we are now using less computation steps.</p>	<p><code>actor_target_network_parameter_update_steps = 10</code></p> <p><code>critic_target_network_parameter_update_steps = 10</code></p>	 <p>271.06073388258613</p>

Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S10	<p>This experiment is the same as experiment #S9 but now we double the number of steps between the actor and critic's target networks updates, from 10 to 20 time-steps.</p> <p>The results look similar to the previous experiment, but this time the agent was capable of solving the environment and steadily stay above the threshold of 30 for 100 consecutive episodes as of episode 700.</p>	<p>actor_target_network_parameter_update_steps = 20</p> <p>critic_target_network_parameter_update_steps = 20</p>	 <p>Episode 1000 - Last average reward: 30.796299311649054</p> <p>367.0369551698367</p>
S11	<p>This experiment builds upon the success obtained in experiment #S10 and further increases the number of steps between the actor and critic's target networks updates, from 20 to 30 time-steps.</p> <p>The agent performed even better passing the average score threshold much sooner at around episode 600 and keeping above it from that moment on.</p>	<p>actor_target_network_parameter_update_steps = 30</p> <p>critic_target_network_parameter_update_steps = 30</p>	 <p>Episode 1000 - Last average reward: 31.50469929581508</p> <p>279.73998349905014</p>

Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S12	<p>Now that we had a stable algorithm that successfully solves the task at hand, we turned our attention to the actor and critic network architectures. We wanted to know if we could still solve the problem using simpler network architectures because that would reduce the training time. We therefore reduce the number of nodes in the two actor and critic networks from 256 to 128 nodes.</p> <p>The results were unfortunately not encouraging with the agent failing to solve the task. This architecture did not have a strong enough representational capacity for this specific problem.</p>	hidden_dim = 128	 <p>148.74403164784113</p>
S13	<p>Conversely to experiment #S12, this experiment tests a more complex network in an attempt to have a more detailed representation of the mapping between states and actions and states-actions and their values. We therefore doubled the number of nodes in the hidden layers of the actor and critic networks from 256 to 512.</p> <p>Also here the results were not encouraging with the agent not being able to solve the task, probably due</p>	hidden_dim = 512	 <p>515.0800759156544</p>

Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
	to overfitting, and taking a lot more time to run 1000 episodes due to the added computational effort derived from more complex neural networks architectures.		
S14	<p>Here we test an approach that only learns every few steps. This experiment was selected in an attempt to use a more diverse experiences-set during training hopefully leading the agent to explore more the state space with the given limit of 1000 episodes of 1000 steps maybe even further stabilising the learning process.</p> <p>Although the time needed to run 1000 episodes was dramatically reduced, this approach did not yield a positive result with the agent not being able to successfully solve the task.</p> <p>We conclude that the agent needs to learn at every time-step to keep up with what is happening in the environment. This makes sense as otherwise the agent my “miss out” on significant experiences added to the replay buffer.</p>	steps_to_learning = 10	 <p>Episode 1000 - Last average reward: 20.17209954911843</p> <p>74.08886719942093</p>

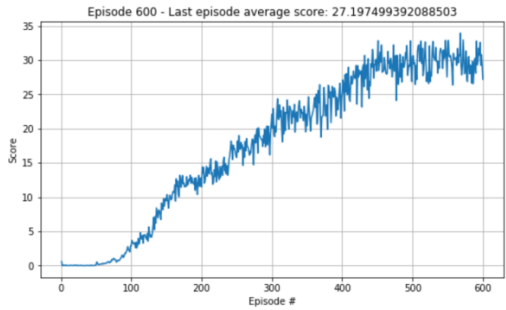
Experiment #	Explanation	Hyper-Parameters Used/Changed	Experiment Results
S15	<p>Finally, we go back to learning at each time step and experiment with a different loss function for the critic, changing from mean squared error to a smooth L1 loss which should be more tolerant to outliers. We expect our model to be more robust with this change and to observe fewer fluctuations in the obtained episodic scores.</p> <p>The results are however inconclusive with no significant gain over the use of a mean-squared-error loss function.</p> <p>The agent does solve the task at around episode 700.</p>	<pre>steps_to_learning = 1 self.critic_loss = nn.SmoothL1Loss()</pre>	 <p>276.3443571011225</p>

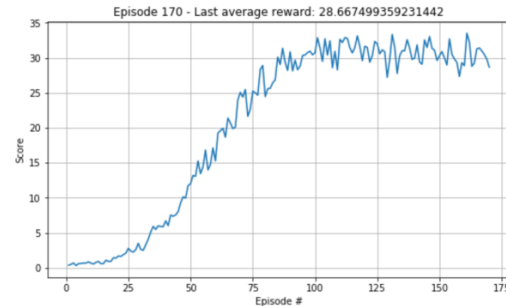
### 6.2.2. Multiple Agents Environment

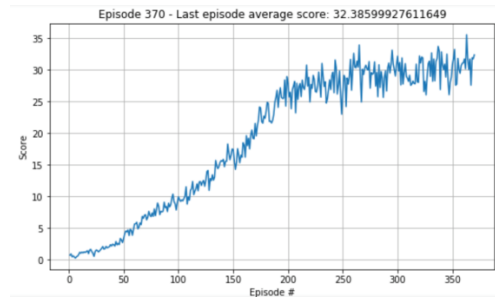
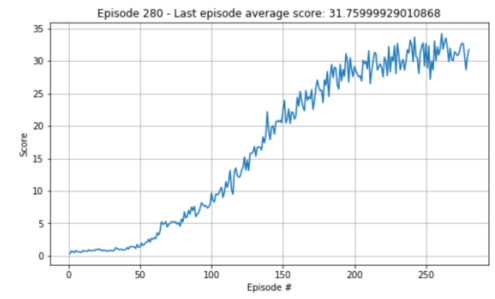
The main difference of our implementation when using the single and multiple agent Unity environments is the storage of experiences from all environment instances in a shared replay buffer. By using this quite small change, implementation-wise, we allow the agent to explore the state space much more as every different environment instance should be experiencing a different region of the state space.

As mentioned earlier we started by training 20 independent agents each linked to a different Unity environment instance with a handler class managing the agents' learning and the storage of experiences from each agent in the shared replay buffer. This approach was insufficient and had several shortcomings, namely the heavy computational burden of 20 agents trying to learn their own model in a CPU (one actor and critic online neural network per agent!) and the fact that an extra layer was needed to orchestrate the agents and their linked environment instances. After a number of training attempts which lasted for several days with no meaningful results on sight, we decided to abandon and not report on this approach.

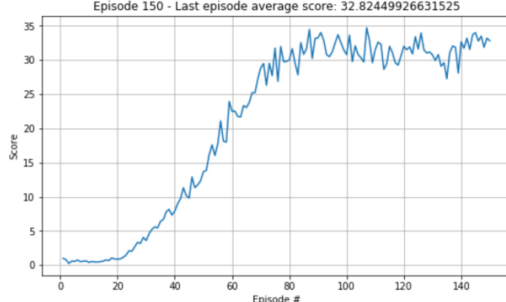
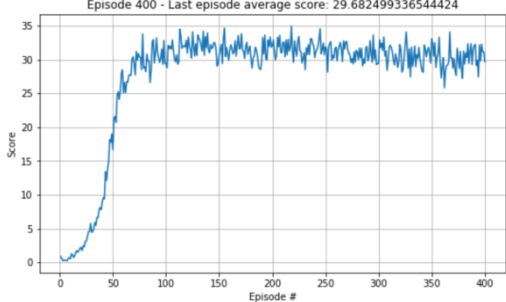
The next solution in the pipeline consisted in training one single agent using experiences from all 20 Unity environment instances still using a shared replay buffer. Right from the start, this approach yielded significant improvements over the previous multiple-agent <-> multiple environment instances approach. The following table puts forward our results in training such an agent together with the episodic scores:

Experiment #	Explanation	Hyper-Parameters Used/Changed	
M1	<p>This experiment sets up our baseline for the multiple-agents environment from which we should improve by changing the algorithm and tweaking the hyper-parameters.</p> <p>We start with a compromise between the number of steps used for learning and number of steps used to update the actor and critic's target networks.</p> <p>Using this set of hyper-parameters the agent</p>	<p>number_of_episodes = 1000</p> <p>max_steps_per_episode = 1000</p> <p>steps_to_learning = 10</p> <p>actor_target_network_parameter_update_steps = 20</p> <p>critic_target_network_parameter_update_steps = 20</p> <p>critic_loss = nn.MSELoss()</p>	 <p>Environment resolved in 605 episodes!</p>

Experiment #	Explanation	Hyper-Parameters Used/Changed	
	is capable of solving the task in 605 episodes. The episodic scores although converging are quite noisy indicating a high-level of variance in the episodic score signal and its underlying reward signal.		71.82740238507589
M2	<p>In this experiment we go back to a successful strategy used in the single-agent environment consisting of learning at every time-step and updating the actor and critic's target networks less often.</p> <p>This strategy paid off immensely when compared with the previous experiment. The agent is capable of solving the task in 173 episodes, while training for only 1 hour! It seems that the shared replay buffer is also helping significantly.</p>	<p>steps_to_learning = 1</p> <p>actor_target_network_parameter_update_steps = 30</p> <p>critic_target_network_parameter_update_steps = 30</p>	 <p>Environment resolved in 173 episodes!</p> <p>63.26565684874853</p>

Experiment #	Explanation	Hyper-Parameters Used/Changed	
M3	<p>This experiment tries to provide information leading to the understanding of which variable had a positive impact on the successful results obtained in experiment #M2. We therefore change the number of steps between each learning step from 1 to 10.</p> <p>Once more the results confirmed the need to learn at every time-step.</p>	<p>steps_to_learning = 10</p> <p>actor_target_network_parameter_update_steps = 30</p> <p>critic_target_network_parameter_update_steps = 30</p>	 <p>Environment resolved in 376 episodes!</p> <p>40.56540506283442</p>
M4	<p>This experiment also tries to provide information leading to the understanding of which variable had a positive impact on the successful results obtained in experiment #M2.</p> <p>This time we changed the number of steps between updates to the actor and critic's target networks from 30 to 40.</p> <p>The results confirm the positive impact in learning and training time of this change.</p>	<p>steps_to_learning = 10</p> <p>actor_target_network_parameter_update_steps = 40</p> <p>critic_target_network_parameter_update_steps = 40</p>	 <p>Environment resolved in 281 episodes!</p> <p>38.51568686564763</p>



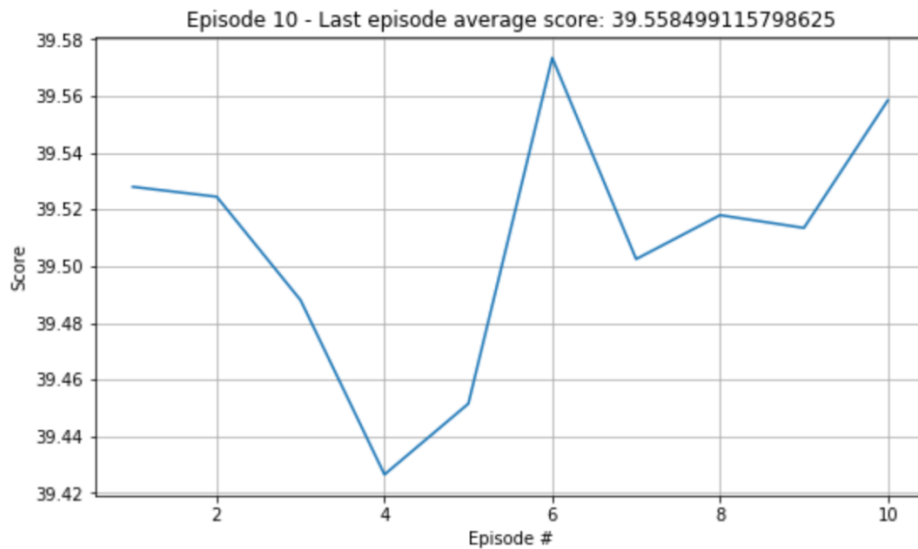
Experiment #	Explanation	Hyper-Parameters Used/Changed	
M5	<p>In this experiment we go back to our most successful agent so far (M2) which learns at every time step and updates the actor and critic's target networks every 30 time-steps, but replace the mean squared error loss function by a Smooth L1 Loss function also known as Huber Loss.</p> <p>This experiment yielded the <u>best model</u> so far with the agent learning the task successfully after only 155 episodes and in under an hour of training!</p> <p>This agent became our best agent.</p>	<pre>steps_to_learning = 1  actor_target_network_parameter_update_steps = 30  critic_target_network_parameter_update_steps = 30  self.critic_loss = nn.SmoothL1Loss()</pre>	 <p>Environment resolved in 155 episodes!</p> <p>55.17994707028071</p>
M6	<p>Finally, and to confirm the best results obtained in experiment #M5, we ran a new training session keeping all hyper-parameters unaltered but not stopping when the agent solves the task so that we could assess if the agent's behaviour became consistent over a long period of time. We therefore ran this modified version for 400 episodes.</p> <p>Indeed, the agent is capable of solving the</p>	<pre>steps_to_learning = 1  actor_target_network_parameter_update_steps = 30  critic_target_network_parameter_update_steps = 30  self.critic_loss = nn.SmoothL1Loss()  number_of_episodes = 400</pre>	 <p>149.27148060003915</p>

Experiment #	Explanation	Hyper-Parameters Used/Changed	
	task and maintain an average score over 100 consecutive episodes above the threshold of 30 points, starting around episode 135 until the end of the 400 episodes. The episodic score signal is still a bit noisy though.		

## 7. Testing Results

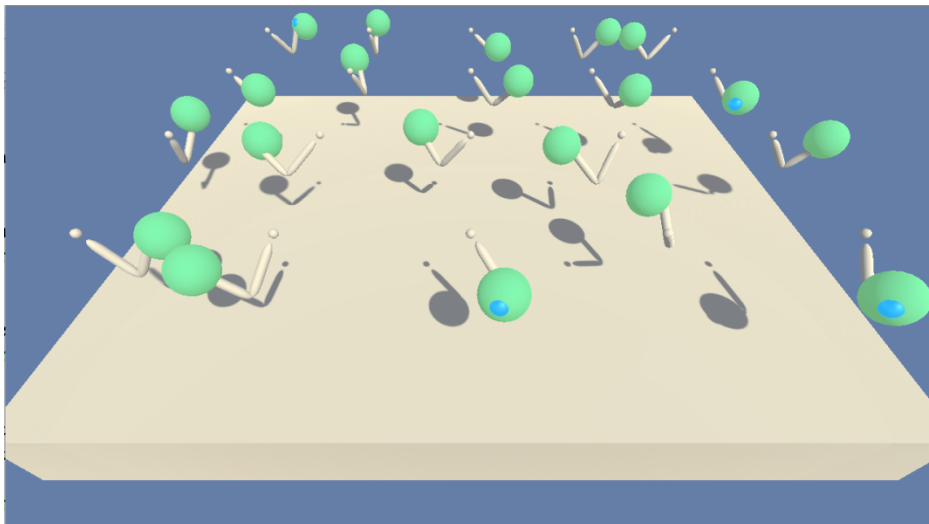
To confirm that our most successful model (M5) obtained during training was indeed fit for purpose, we run a batch of 10 episodes of 1000 steps each, using the 20-agents instance of the Unity environment in test mode and with a different seed from the one used for training.

The results were conclusive with the agent being able to maintain average scores well above the required 30 points right from the first episode.



*Figure 3 - Test Results of the Most Successful Trained Model (M2)*

Further investigating what the agent was actually doing in each of the 20 instances by activating graphics mode in the Unity environment revealed that the agent immediately moves the arms towards each of the 20 targets and stays there for the entire duration of the episode:



*Figure 4 - The Agent Follows Each of the 20 Targets Successfully*

## 8. Ideas for Future Work

### 8.1. Further Enhance the Exploration Strategy

Although the Ornstein–Uhlenbeck stochastic process (**Uhlenbeck, George E. and Ornstein, Leonard S., 1930**) used did help with the exploration of the action-space, we were not completely happy about it. Perhaps experimenting with different hyper-parameters or even using different noise processes for each of the different environment instances could lead to the exploration of unseen action and state spaces and this would yield even better results?

### 8.2. Using Batch-Normalisation Layers Inside the Neural Networks

Another technique that we would like to experiment with, as mentioned in this paper (**Ioffe, Sergey and Szegedy, Christian, 2015**) is batch-normalisation of all parameters in the action and critic 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](https://pytorch.org/docs/stable/_modules/torch/nn/modules/batchnorm.html)), so this improvement could be fairly simple to implement.

We can therefore introduce batch normalisation layers before the input to each of the hidden layers of the actor and critic neural networks as well as before the final out put layer so that the weights of the networks stay within acceptable limits and do not explode.

As a final note, further investigation and tests would have to be conducted though to understand if this technique would indeed work in practice, and perhaps more importantly, how does it play along with other techniques that we are already using namely gradient clipping.

### 8.3. Replace the Shared Replay Buffer by a Shared Prioritised Replay Buffer

Finally, we would 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.

## 9. References

- Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning" Nature 518.7540 (2015): 529-533.
- 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.
- Meire Fortunato et al. "Noisy Networks for Exploration" arXiv:1706.10295 (2018) <https://arxiv.org/abs/1706.10295>.
- 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.