Lab 3: Policy Gradients and Actor Critic Reinforcement Learning

mingfei.sun@manchester.ac.uk

In this practical, we study policy gradients for continuous domains. The practical is split into two parts. In Part I, we derive the policy gradient theorem from first principles for continuous domains and (optionally!) explore the assumptions required for its derivation. In Part II, we discuss methods of implementing the policy gradient theorem using estimators. We derive the REINFORCE estimator before extending policy gradient methods to actor-critic algorithms. We test our algorithms on continuous Pendulum-V0 task, before exploring differences between value-based and policy-based reinforcement learning algorithms. The questions marked with "*" or BONUS are optional, you should answer them last.

If this all sounds very daunting, don't worry! We don't expect you to finish everything during the practical. The aim is to understand policy gradients and reinforcement learning beyond a superficial algorithmic level. If you get stuck, ask for help, and if you are really stuck, just move on to the next part. We will provide full solutions at the end.

First of all, it is important that you get the coding environment set up:

Installing the Environment and Running the Code

1. Make sure `conda` is installed and updated by opening a terminal running the command:

   ```
   conda update --all
   ```

   If you haven't got `conda` installed, run the command

   ```
   pip install conda
   ```

2. Unzip the file `CDT_Policy_Gradients` and save it somewhere convenient.

3. Open the terminal and navigate to inside the unzipped `CDT_Policy_Gradients` folder. Now build the virtual `conda` environment by running

   ```
   conda env create -f environment.yml
   ```

   This can take up to 10 minutes to build, so you may want to start the first exercise!

4. You should now have a virtual environment called `cdt_policy_gradients` with all the packages required to run the practical. You can activate this environment using the command

   ```
   source activate cdt_policy_gradients
   ```

5. Within the activated environment, you can run each experiment as follows:

   ```
   python part_ii.py
   ```

Run the above code and after a few seconds, you should see the results of the first evaluation. The code should print something like:

```
Epoch:  0, Average Test Return:  -1305.305089977554
```

This should continue for 50 epochs. As the agent is not be learning (yet!), the value of `Average Test Return` will hover in the region -1800 to -1100 depending on the initialisation of the network.

6. At the end of the practical, you can deactivate the virtual environment and remove it from your machine by running

```
source deactivate
conda remove --name cdt_policy_gradients --all
```

Notation and Setting

We can formally define continuous discounted, infinite horizon MDPs as a tuple $\langle S, A, p, r, \gamma \rangle$. Here, S and A are sets of continuous states and actions respectively, p is the state transition distribution $p(s'|s, a) : S \times A \to S$, r is the reward function $r(a, s) : A \times S \to \mathbb{R}$ and $\gamma \in [0, 1)$ is the discount factor. The graphical model for the N-step MDP is shown in Fig. 1; dashed lines are used to emphasise the choice that the agent has in their policy $\pi(a|s)$.

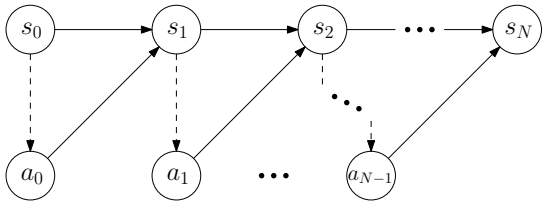

Figure 1: Graphical Model of MDP

We define a trajectory of length N as $\tau_N := (s_0, a_0, s_1, a_1, \cdots s_N)$ and its corresponding distribution as:

$$p_N(\tau) := p_0(s_0) \prod_{i=1}^{N} \pi(a_{i-1}|s_{i-1}) p(s_i|s_{i-1}, a_{i-1}).$$

The aim is to find a policy that maximises long term, expected discounted reward over all possible trajectories, which when using a parametrised policy $\pi_\theta(a|s)$, yields the objective:

$$J(\theta) = \mathbb{E}_{p(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] = \int p_0(s_0) \int \pi_\theta(a_0|s_0) Q^\pi(a_0, s_0) da_0 ds_0, \tag{1}$$

where $p(\tau) := \lim_{N\to\infty} p_N(\tau)$ and we have used the shorthand $r_t := r(a_t, s_t)$. The Q-function for our policy is defined as the total long term, expected discounted reward over all possible trajectories given an arbitrary starting state $s_0 = s$ and action $a_0 = a$:

$$Q^\pi(s, a) := \mathbb{E}_{p(\tau|s,a)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right],$$

where $p(\tau|s, a) := \lim_{N\to\infty} \left(p(s_1|s_0 = s, a_0 = a) \prod_{i=2}^{N} \pi(a_{i-1}|s_{i-1}) p(s_i|s_{i-1}, a_{i-1}) \right)$.

Part I: Deriving the Policy Gradient Theorem

In policy gradient methods, we treat the objective as a function to be maximised through gradient ascent. We therefore need an analytic, convenient closed form of its derivative. To derive this update, we introduce the discounted state distribution and then go on to derive full analytic expression for the policy gradient.

Exercise 1. Discounted State Distribution.

Here we take a closer look into the derivation and interpretation of discounted state distribution, defined as:

$$d^{\pi}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t p_{\theta,t}(s),$$

where $p_{\theta,t}(s)$ is the marginal distribution over states after t timesteps under policy π_θ.

(i) Let $p(s_{i+1}|a_i, s_i)$ be the state-transition function and $\pi_\theta(a_i|s_i)$ be the policy. By deriving an expression for the joint $p_\theta(s_{i+1}, a_i|s_i)$ and marginalising for actions, find the state-conditional distribution $p_\theta(s_{i+1}|s_i)$.

(ii) Find an expression for $p_{\theta,t}(s)$. Hint: Use your solution to previous exercise to find expression for the joint $p_{\theta,t}(s, s_{t-1}, \cdots s_0)$ and then marginalise.

(iii) BONUS. To account for discounting, we multiply by a discount factor γ^t and sum over all time steps:

$$p_{N,\theta}(s) = \sum_{t=0}^{N-1}\gamma^t p_{\theta,t}(s)$$

Note that as presented above, $p_{N,\theta}(s)$ not strictly a probability measure (why is that?). For this reason, it is often referred to as the ergodic occupancy measure when taking the limit $N \to \infty$. A probabilistic interpretation is readily available though if we treat the current timestep t as a random variable, writing $p_{\theta,t}(s) = p_\theta(s|t)$ and introducing a prior distribution $p(t)$. For the undiscounted case, we use a uniform distribution. For the discounted case, we use a geometric prior $p_N(t) = \frac{\gamma^t}{\sum_{t=1}^{N}\gamma^t}$ (what is the interpretation of geometric prior here? What does discounting represent?). We then find the joint distribution as $p_{N,\theta}(s,t) = p_{N,\theta}(s|t)p(t)$ and marginalise out for time:

$$\tilde{p}_{N,\theta} = \frac{1}{\sum_{t=1}^{N}\gamma^t}\sum_{t=0}^{N-1}\gamma^t p_\theta(s|t).$$

(iv) BONUS. For infinite-horizon MDPs, we must take the limit $N \to \infty$. Show that:

$$d^{\pi}(s) \coloneqq \lim_{N\to\infty}\tilde{p}_{N,\theta}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t p_\theta(s|t).$$

We refer to $d^\pi(s)$ as the discounted state distribution, and see that it is related to the ergodic occupancy measure by a scaling of $1-\gamma$. Expanding out $d^\pi(s)$ a few time steps affords some insights into its meaning:

$$d^\pi(s) = (1-\gamma)\left(p_\theta(s|t=0) + \gamma p_\theta(s|t=1) + \gamma^2 p_\theta(s|t=2) + \cdots\right).$$

We see from the above expansions that each $p_\theta(s|t)$ is the distribution over states that our agent will be in at timestep t, and therefore $d^\pi(s)$ can be viewed as an average of these distributions over all timesteps, weighted by the discounting factor γ^t.

Exercise 2. Policy Gradient Theorem

(i) Find the derivative of the reinforcement learning objective in Eq. (1) in terms of $\nabla_\theta \pi_\theta(a_0|s_0)$ and $\nabla_\theta Q^\pi(a_0, s_0)$.

(ii) The next step is to find a solution for $\nabla_\theta Q^\pi(a_0, s_0)$. By using the Bellman equation $Q^\pi(s_i, a_i) = r_t + \gamma \int p(s_{i+1}|s_i, a_i) \int \pi_\theta(a_{i+1}|s_{i+1}) Q^\pi(s_{i+1}, a_{i+1}) da_{i+1} ds_{i+1}$, derive an equation for the gradient $\nabla_\theta Q^\pi(s_i, a_i)$ in terms of $\nabla_\theta Q^\pi(s_{i+1}, a_{i+1})$. Use this equation to show that:

$$\int p_0(s_0) \int \pi(a_0|s_0) \nabla_\theta Q^\pi(s_0, a_0) da_0 ds_0 = \sum_{t=1}^{\infty} \gamma^t \int p_{\theta,t}(s) \int \nabla_\theta \pi_\theta(a|s) Q^\pi(s, a) ds da.$$

Hint: We can use our first equation to write $\nabla_\theta Q^\pi(s_0, a_0)$ in terms of $\nabla_\theta Q^\pi(s_1, a_1)$, which can be written in terms of $\nabla_\theta Q^\pi(s_2, a_2)$ which can be written in terms of $\nabla_\theta Q^\pi(s_3, a_3)$ etc...

(iii) Finally, use results from parts (i) and (ii) to show that:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{\infty} \gamma^t \int p_{\theta,t}(s) \int \nabla_\theta \pi_\theta(a|s) Q^\pi(s, a) da ds. \tag{2}$$

Exercise 3*. Irreducible, Aperiodic MDPs.

There is another thing that we have glossed over in deriving the continuous policy gradient theorem; for Eq. (2) to converge, the discounted state distribution relies on the existence of $\lim_{t\to\infty} p(s|t)$. For illustrative purposes, we restrict ourselves to finite, discrete MDPs. The stationary distribution can be thought of as the long-term distribution over states, independent of the starting state of the MDP. For more detail about stationary distributions in discrete Markov chains see e.g. Murphy Chapter 17.

A known result, which is a consequence of Perron-Frobenius theorem, is that for every irreducible, aperiodic MDP, $\lim_{t\to\infty} p(s|t)$ exists and converges to the MDP's unique stationary distribution.

(i) An irreducible MDP is one in which we can always (eventually) get back to any state in the MDP. Give an example of a discrete MDP that is irreducible and one that is reducible. Hint: What happens if there is an absorbing state in the MDP?

(ii) The period of a state is the greatest common divider of all time steps required to return to that state, for example, if an agent can return to state s again only every $\{2, 6, 8, 12 \cdots\}$ timesteps, the the period is 2. An aperiodic state has period 1 and an aperiodic MDP consists entriely of aperiodic states. Is the MDP in Fig. 2 periodic? If yes, alter it to make it aperiodic; if no, alter it to make it periodic.

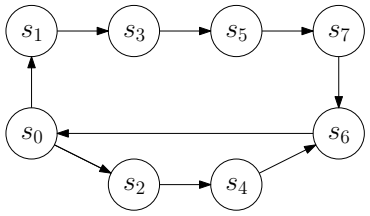

Figure 2: An 8-State Discrete MDP

(iii) If the MDP we are trying to solve is not irreducible and aperiodic, how does that effect our derivation of policy gradient? Is this a problem in practice, and if not, why not?

Part II: Implementation and Actor Critic

Exercise 4. Monte Carlo Estimation.

In reinforcement learning, we often don't have direct access to the Q-function to be able to obtain an analytic solution to Eq. (2), nor do we have an analytic form for the discounted state distribution. A common method for estimating the gradient is to use a Monte Carlo method. One problem remains, however. We need to write Eq. (2) as an expectation over a in order to apply these methods. The log-derivative trick $\nabla_\theta \pi_\theta(a|s) = \pi_\theta(a|s)\nabla_\theta \log \pi_\theta(a|s)$ comes to the rescue:

$$\begin{aligned}
\nabla_\theta J(\theta) &\propto \int d^\pi(s) \int \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) ds da, \\
&= \int d^\pi(s) \int \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a) ds da, \\
&= \mathbb{E}_{d^\pi(s)} \left[\mathbb{E}_{\pi_\theta(a|s)} \left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a) \right] \right].
\end{aligned} \tag{3}$$

To obtain an empirical estimate to the expectation, we sample N trajectories from the environment under our policy π from the distribution $p(\tau)$. We then form our estimate as:

$$\nabla_\theta \hat{J}(\theta) \coloneqq \frac{1}{N} \sum_{n=0}^{N-1} \sum_{t=0}^{\infty} \gamma^t \nabla_\theta \log \pi(a_{t,n}|s_{t,n}) Q^\pi(a_{t,n}, s_{t,n}).$$

This estimator is known commonly as the *score function estimator*. Another way of thinking about this estimator comes from swapping the order of the summations:

$$\nabla_\theta \hat{J}(\theta) \coloneqq \sum_{t=0}^{\infty} \gamma^t \left(\frac{1}{N} \sum_{n=0}^{N-1} \nabla_\theta \log \pi(a_{t,n}|s_{t,n}) Q^\pi(a_{t,n}, s_{t,n}) \right).$$

As we can view sampling from the entire trajectory N times as equivalent to obtaining N samples each from $p(s|t=0)$, $p(s|t=1)$, $\cdots$ $p(s|t=\infty)$, finding $\nabla_\theta \hat{J}(\theta)$ is thus equivalent to finding the γ^t weighted average of all of these samples. Compare this with the definition of $d^\pi(s)$.

(i) BONUS. Is the estimator $\nabla_\theta \hat{J}(\theta)$ biased? Can you prove this?

(ii) BONUS. Two problem remains; firstly we still need to sample from $Q^\pi(a,s)$ to calculate our estimator. An obvious solution to this is to use the actual sampled return $R_i \coloneqq \sum_{j=i}^{\infty} r_j$ as an unbiased estimate of $Q^\pi(a,s)$. The resulting algorithm is known as REINFORCE. Secondly, so far we have assumed that we can sample infinitely long trajectories. As such, we need to use N truncated trajectories instead, each of length T_n. Putting this all together yields the (truncated) REINFORCE estimator:

$$\nabla_\theta \hat{J}_{\text{RE}}(\theta) = \frac{1}{N} \sum_{n=0}^{N-1} \sum_{t=0}^{T_n} \nabla_\theta \log \pi(a_{t,n}|s_{t,n}) R_{t,n}.$$

Is the estimator $\nabla_\theta \hat{J}_{\text{RE}}(\theta)$ biased? Can you prove this?

Exercise 5. Problems with Policy Gradients.

As policy gradient methods render the reinforcement learning problem as an unconstrained gradient optimisation problem, they benefit from all of the convergence properties of stochastic gradient ascent methods provided we can obtain unbiased estimates (see Bertsekas for a comprehensive overview). The stochastic update for our policy parameters is:

$$\theta_{k+1} \leftarrow \theta_k + \alpha_k \nabla_\theta \hat{J}(\theta),$$

where α_k is a step size, often chosen to satisfy the Robbins-Munro conditions $\sum_{k=0}^{\infty} \alpha_k = \infty$ and $\sum_{k=0}^{\infty} \alpha_k^2 < \infty$ for convergence.

Policy gradient methods, however, have two major drawbacks. Firstly, they suffer from high variance; for most RL problems, it is prohibitively costly to sample more than a few trajectories for an update, especially if they are long enough to mitigate the effect of bias. Moreover, they only use a single return for each trajectory; the rewards sampled for $R_{0,n}$ are used not just to estimate $Q^\pi(a_{0,n}, s_{0,n}) \approx R_{0,n}$, but also the Q-function from each further timestep, i.e. $Q^\pi(a_{1,n}, s_{1,n}) \approx R_{1,n}$, $Q^\pi(a_{2,n}, s_{2,n}) \approx R_{2,n}$ etc, so the estimate of $Q^\pi(s,a)$ itself at each timestep will have high variance as a result of only being a single sample. This is further compounded by the fact that we would expect the distribution over states $p_\theta(s|t)$ to have high variance for most MDPs, especially as $t >> 0$, reflecting the fact that more probability mass is spread over more of the state space as time progresses. This is mitigated somewhat by using discount factors, but we must be careful not to make γ too small or else our optimal policies will become myopic - that is, they become more concerned with maximising immediate rather than long term reward. In the following exercises we explore how baselines can be used to reduce this variance.

(i) Show that for any bounded baseline $b(s) : S \to \mathbb{R}$, we can write the policy gradient from Eq. (3) as:

$$\int d^\pi(s) \int \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) da ds = \int d^\pi(s) \int \nabla_\theta \pi_\theta(a|s) (Q^\pi(s,a) - b(s)) da ds.$$

Analysis by Greensmith shows that using the value function as a baseline $b(s) = V^\pi(s) \coloneqq \mathbb{E}_{\pi(a|s)}[Q^\pi(s,a)]$ yields almost the lowest possible variance. To see why, consider the advantage function $A^\pi(s) \coloneqq Q^\pi(s) - V^\pi(s)$, so called because it gives an indication the advantage - i.e. how much better expected reward is for a given action a relative to the average expected reward in that state. As a result, the policy gradient update

will only be positive for $A^\pi(s) > 0$, increasing the probability of better than average actions being selected in the future. Conversely, when $A^\pi(s) < 0$, our policy gradient is updated in a direction to decrease the probability of worse than average actions.

The second major drawback for policy gradient methods is their sample inefficiency - that is their ability to reuse samples in an efficient way. After we have carried out an update to the parameters, we throw away all the data we have collected to compute that estimate and must re-sample N new trajectories.

A way to tackle both variance and sample inefficiency is to use a critic, giving rise to a family of algorithms known as actor-critic; here we learn a Q-function approximation $Q^\pi(s,a) \approx Q_\omega(s,a)$ and a value function approximation $V^\pi(s) \approx V_\phi(s)$. In actor-critic methods, we carry out successive policy gradient and policy evaluation steps. In the policy evaluation step, we update our value functions using any preferred evaluation method such as TD, LSTD or residual methods.

(ii) Why do actor-critic methods have improved variance and sample efficiency?

Exercise 6. Implementing Actor-Critic.

We now implement an actor-critic algorithm and investigate exploration strategies. We use neural network function approximators for $Q_\omega(s,a)$ and $V_\phi(s)$ to estimate the advantage $A_{\omega,\phi}(s,a) = Q_\omega(s,a) - V_\phi(s)$. We also learn a neural network with parameters θ to output the state dependent mean and variance of our policy, which is a tanh-Gaussian policy - effectively passing the actions sampled from a Gaussian distribution through a tanh squashing function. Importantly, this distribution has support over a finite action space.

Let's go through `algorithms.py` in a bit more detail, as this is the only file where you'll be writing your own code. Firstly, the algorithm samples a step at time t from the Pendulum-v0 environment in the function `env_step()`, saving the tuple $(s_i, a_i, r_t, s_{i+1}, \text{terminal})$ to the repay buffer. The algorithm then trains the networks using `train_score()`; we first sample a random batch of 128 actions, rewards, states, next states and terminal bools from the replay buffer. These take the form of 128×1 tensors and are used to generate the variables required for training. We then generate the critic losses: we learn a value function (`self.vf`) and action-value function (`self.qf`) using the TD-error with a few tricks to stabilise learning (if you want to know more, please ask). We then generate the policy losses, which you will need to implement. After all losses have been generated, we call `backward()` on each respectively and step the optimiser to update the network parameters. There are 200 of these training steps per epoch. Finally, the algorithm evaluates the performance of our policy on the Pendulum-V0 task in the function `evaluate()` at the end of the training epoch, averaging across 5×200-step episodes. The pseudocode is shown below:

The file `part_ii.py` runs the experiment for 5 different random seeds for 30 epochs and saves the data in a .csv file locally. It also plots the mean average test return across the 5 seeds. If you find that you don't have enough time to run 5 trials, change the value of `number_of_trials` on line 18 to 3 (or less if you are really pushed).

In continuous domains, it is often not sufficient to rely solely on the stochastic policy $\pi_\theta(a|s)$ to generate diverse enough samples to learn good policies. As a result, policies become too confident about sub-optimal actions and converge too quickly. Other than using ϵ-greedy exploration, a slightly more sophisticated (and effective) way to encourage exploration is to add an entropy term to the reinforcement learning objective,

Algorithm 1 Actor-Critic:

Initialize parameter vectors ϕ, $\bar{\phi}$, θ, ω, $\mathcal{D} \leftarrow \{\}$
Fill buffer with 1000 initial samples
for each iteration **do**
  for each environment step **do**
    $a_i \sim \pi^q(a|s;\theta)$
    $s_{i+1} \sim p(s_{i+1}|s_i, a_i)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_i, a_i, r(s_i, a_i), s_{i+1})\}$
  end for
  for each gradient step **do**
    $\phi \leftarrow \phi - \lambda_V \hat{\nabla}_\phi \mathbb{E}_{s_i \sim \mathcal{D}} \left[\left(V_\phi(s_i) - \mathbb{E}_{a_i \sim \pi_\theta} \left[Q_\omega(s_i, a_i) \right] \right)^2 \right]$
    $\omega \leftarrow \omega - \lambda_Q \hat{\nabla}_\omega \mathbb{E}_{(h_t, r_t, s_{i+1}) \sim \mathcal{D}} \left[\left(r_t + \gamma V_{\bar{\phi}}(s_{i+1}) - Q_\omega(h_t) \right)^2 \right]$
    $\theta \leftarrow \theta + \lambda_{\pi^q} \mathbb{E}_{(s_i, a_i) \sim \mathcal{D}} \left[\nabla_\theta \Big(\log \pi_\theta(a|s) \left(A_\omega(a, s) - \log \pi_\theta(a|s) \right) \Big) \right]$
    $\bar{\phi} \leftarrow \tau \bar{\phi} + (1 - \tau) \bar{\phi}$
  end for
end for

yielding,

$$J(\theta) = \int p_0(s) \int \pi_\theta(a|s) \left(A_\omega(a, s) - \alpha \log \pi_\theta(a|s) \right) da ds, \tag{4}$$

where α is a constant that controls the degree of entropy added to the objective. Choosing effective values of α is an ongoing question in RL research, and we are often reduced to tuning it using hyperparameter search. We use $\alpha = 1$ for this practical.

Taking derivatives of Eq. (4) using the policy gradient theorem with the log-derivative trick then yields the gradient update:

$$\begin{aligned} \nabla_\theta J(\theta) \propto & \int d^\pi(s) \nabla_\theta \int \pi_\theta(a|s) \left(A_\omega(a, s) - \log \pi_\theta(a|s) \right) da ds, \\ = & \mathbb{E}_{d^\pi(s) \pi_\theta(a|s)} \Big[\nabla_\theta \Big(\log \pi_\theta(a|s) \left(A_\omega(a, s) - \log \pi_\theta(a|s) \right) \Big) \Big]. \end{aligned} \tag{5}$$

(i) Why does the additional entropy term in Eq. (4) encourages exploration? Hint: What happens when the policy is very confident about a particular action? What about when all actions have equal probability?

(ii) Implement the score function gradient update with entropy as derived in Eq. (5) in the function `train_score()`, un-commenting line 173 to allow gradient updates to the policy. If it is implemented correctly, you should start to see the average test return rise above -1000 for a least one epoch by 25 epochs of training. If the agent is training, remove the entropy term and test for 1 trial to show that it doesn't learn to improve it's policy within 30 epochs. Then let it run for the full 5 trials with the entropy term and move on to the next part while it completes (it should take around 20 minutes).

Hints:

- Use `new_q_values` instead of `q_values` for $Q_\omega(a, s)$.

- The variables `log_pis`, `new_q_values` and `values` have already been calculated for you for the current batch.
- PyTorch's `detach()` function stops gradient being passed through variables when `backward()` is called.
- Remember, we are calculating a *loss* that PyTorch will *minimise*.

Exercise 7. Actor-Critic vs. Value Based Methods.

It is important to understand the advantages and disadvantages of actor-critic algorithms compared to critic-only, value based algorithms such as Q-learning. Firstly, there is no inherent way to explore using Q-learning; deterministic policies are always extracted from the Q-function and stochasticity must be constructed as part of the algorithm through a method such as ϵ-greedy exploration. In actor-critic methods, non optimal policies are stochastic and so will induce an exploration strategy for the agent. Secondly, convergence guarantees are limited, even when using simple function approximators for algorithms such as Q-learning. A related issue occurs when the function approximator is biased, which we explore now:

(i) Q-learning has a major disadvantage when using function approximators; explain why finding the deterministic policy $\pi(a|s) = \delta(a = \arg_{a'} \max Q_\omega(s, a))$ may cause issues when the function approximator is biased.

(ii) The optimal Bellman operator presents a further compounding issue for several classes of non-linear function approximators such as arbitrary neural networks in continuous domains. Can you think what it is?

Bonus: Seeing your trained agent!

If you have time at the end, set `number_of_trials` to `1`, `save_data` to `False` and `render_agent` to `True` to see your trained playing with pendulum. See if you can explain why the agent's learning curve always dips first in training steps 0-1000 before climbing.