CDT Deep Reinforcement Learning Practical 3: Policy Gradients and Actor Critic

Matthew Fellows and Supratik Paul

Overview of Practical

In this practical, we implement actor critic algorithms for continuous domains. The practical is split into two parts. In Part I we use the REINFORCE algorithm with a Gaussian policy and test on a a simple PointEnv environment from rllab where the goal is to control an agent and get it to the target located at $(0, 0)$. At each timestep the agent gets its current location (x,y) as observation, takes an action (dx, dy), and is transitioned to $(x + dx, y + dy)$. We then explore the variance reducing effects of using a baseline to stabilise learning. Finally, we add a discounting factor to rewards.

In Part II, we will then explore deep actor-critic methods, using neural networks for our actor and critic that can be used in more complex domains. We first examine the challenges that these domains present and demonstrate that stochastic policy gradients (SPG) often fail even in low dimensional continuous problems. We then derive two policy gradient methods that overcome these shortcomings: 1) we derive an new update by adding an additional entropy term to our objective to aid exploration and 2) we will then use the reparametrisation trick to further reduce the variance of our gradient updates. We test our algorithms in Part II using OpenAI's continuous Pendulum-V0 task (see here for details).

If this all sounds very daunting, don't worry! We are here to help, so please ask us if you get stuck.

The Codebase

Implementing RL is nasty. Even when everything is coded up perfectly, small things such as neural network initialisations or even using Tensorflow over PyTorch can cause drastic changes in performance (see Deep Reinforcement Learning that Matters (Henderson et al. 2018) for an overview of the many things that can go wrong). As a result, we have given you a very stable implementation of actor-critic, and you won't be alterning much of it, however don't be surprised if there are subtle differences in performance and your agent doesn't learn perfectly every time.

Code Structure

We have used PyTorch for all of the deep learning. If you are not familiar, you may want to have a look at the first two sections of this official guide, paying particular attention to how autograd works. The code contains six python files:

1. `part_i.py` containes the entry point for Part I. It is completely self contained and contains comments to guide you through the implementation.

2. `part_ii.py` contains the entry point for the second part of this practical. You only need to edit line 15, which allows you to switch between `'Score'` to use the score function estimator in Exercise II.1 and `'Reparametrise'` to use the reparametrisation trick in Exercise II.2.

3. `algorithms.py` is where we train and evaluate our agent in Part II. You only need to edit the functions `train_score()` and `train_reparametrisation()` under the section 'Policy (Actor) Loss'.

4. `networks.py` contain the classes for all policy (actor) and value function (critic) networks for Part II. You don't need to edit this file.

5. `storage.py` contains the replay buffer for Part II. You don't need to edit this file either.

6. `plot_all.py` contains code to plot both experiments together in Part II.

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
   ```

Continuous Policy Gradient Theorem

Let's examine the policy gradient theorem for continuous domains. Recall the reinforcement learning objective, which we write here in full:

$$J(\theta) = \int p_0(s) \int \pi_\theta(a|s) Q(a,s) da ds \tag{1}$$

Just like in the discrete case (Sutton et al. 2000), we can take derivatives of Eq. (1) with respect to the policy parameters θ, yielding:

$$\nabla_\theta J(\theta) = \int \rho^\pi(s) \int \nabla_\theta \pi_\theta(a|s) Q(a,s) da ds \tag{2}$$

where $\rho^\pi(s)$ is the discounted ergodic occupancy measure. Using the log-derivative trick, we write Eq. (2) as an expectation over actions:

$$\begin{aligned}\nabla_\theta J(\theta) =& \int \rho^\pi(s) \int \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) Q(a,s) da ds \\ =& \mathbb{E}_{s\sim\rho^\pi(s), a\sim\pi_\theta(a|s)} \Big[\nabla_\theta \log \pi_\theta(a|s) Q(a,s) \Big]\end{aligned}$$

We then approximate the expectation by using N samples from the environment under our policy $\pi_\theta(a|s)$, giving:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{t=0}^{N-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q(a_t, s_t)$$

Part I: REINFORCE and Baselines

Using a Monte-Carlo approximator for the action-value function, $Q(a_t, s_t) \approx R_t \triangleq \sum_{i=t}^{T} r(a_i, s_i)$, we recover the REINFORCE (Williams 1992) algorithm:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{t=0}^{N-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R_t$$

Exercise I.1: Implementing REINFORCE

We choose our policy to be a linear Gaussian policy with parameters θ. Given state s, we can define some features $\phi(s)$ and sample an action $a \sim N(\phi(s)^T\theta, \sigma^2)$. Our PointEnv environment is simple enough that we can use $\phi(s) = s$. Note that σ^2 can also depend on s, but we have kept it constant in Part I for simplicity. Policy gradient algorithms use the update rule $\theta' = \theta + \alpha \nabla_\theta J(\pi)$, where α is the learning rate and $J(\pi)$ is the expected return of the policy.

I.1 In the function `get_action_and_grad()` in `part_i.py`, compute a sampled action for $\pi_\theta(a|s)$ and the corresponding value of $\nabla_\theta \log \pi_\theta(a|s)$.

Exercise I.2: Using Baselines to Reduce Variance

We can subtract a baseline $V(s)$ from the action-value function, which does not affect the overall gradient, but will reduce variance in the gradient update. To see why, as $V_\omega(s) \triangleq \mathbb{E}_{a\sim\pi_\theta(a|s)}[Q_\omega(a,s)]$, the quantity $Q_\omega(a,s) - V\omega(s)$ will only be positive for $Q_\omega(a,s) > V_\omega(s)$, hence the gradient updates will therefore only be scaled by a positive value when an action has greater reward than the average return. For these reasons, we call $A_\omega(a,s) \triangleq Q_\omega(a,s) - V_\omega(s)$ the advantage. For REINFORCE, $A_\omega(a_t,s_t) = R_t - V_\omega(s_t)$. Just like we define features $\phi(s)$ for the policy, we can define some features $\psi(s,t)$ for the value function. Our features are going to be $\phi(s,t) = [s, s^2, t, t^2, t^3, 1]$ and we will approximate the value function as a linear function of these features, $V_\omega(s_t) = \omega^T\phi(s,t)$

I.2a Given some sampled trajectories, fitting the value function is a linear regression problem. The targets for our linear regression problem will therefore be the returns. The features have been implemented in the function `baselines()`, your job is to carry out the linear regression. Hint: The function `np.linalg.lstsq` may be of use.

I.2b Now calculate the value for each state in `path` using the newly learnt coefficients and save it to `path['value']`

I.2c Our implementation of the value function baseline has one major issue that can lead to the failure of the policy update. Can you think of what it is? How would you go about fixing it? MF: I can think of 1: More than one return for a given state should be sampled and averaged over w.r.t. actions or else if the value function learns the return very accurately, advantage as implemented will be zero. Another solution would be to bootstrap 2: Bias in the function approximator could cause severe issues here, as it could reverse the sign of the gradient update arbitrarily. 3 Discounting, but this is mentioned below. Are there more? I think we should give more hints to help here.

Exercise I.3: Using Discount Factors

The agent's objective is to maximise the undiscounted return. However, discounting the future rewards in our gradient computation can often lead to faster learning due to the lower variance in the gradient estimates. Refer to (Schulman et al. 16) for more details.

I.3a The discounted return is given by $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$, where $\gamma \in [0,1)$ is the discount factor. Implement this in the function `process_paths()`.

1.3b Now we will run our algorithm and see if we can learn a good policy for our PointEnv task. In our toy problem, we have tried to mitigate the effects of stocasticity by fixing the initial parameters of the policy, and making the environment deterministic. However, the actions taken are still stochastic, and so we will run 10 random starts to see how our algorithm performs. While the algorithm trains, make a start on Part II.

Part II: Deep Actor-Critic

In this part, we use neural network functions approximators with parameters ω and learn a Q-function as well as a value function rather than relying on returns. We therefore write the advantage as $A_\omega(a,s) = Q_\omega(a,s) - V_\omega(s)$. We also learn a neural network with parameters θ to output the state dependent mean and variance of our policy.

Overview of Algorithm

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 $(s_t, a_t, r_t, s_{t+1}, \text{terminal})$ to the repay buffer. The algorithm then trains the networks using either `train_score()` or `train_reparametrisation()`. In both cases, 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 derive and 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 $5 \times$ 200-step episodes. The pseudocode is shown below:

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**
    $a_t \sim \pi^q(a|s;\theta)$
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$
  end for
  for each gradient step **do**
    $\phi \leftarrow \phi - \lambda_V \hat{\nabla}_\phi \mathbb{E}_{s_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\phi(s_t) - \mathbb{E}_{a_t \sim \pi_\theta} \left[Q_\omega(s_t, a_t) \right] \right)^2 \right]$
    $\omega \leftarrow \omega - \lambda_Q \hat{\nabla}_\omega \mathbb{E}_{(h_t, r_t, s_{t+1}) \sim \mathcal{D}} \left[\frac{1}{2} \left(r_t + \gamma V_{\bar{\phi}}(s_{t+1}) - Q_\omega(h_t) \right)^2 \right]$
    if training mode is 'score' **then**
      $\theta \leftarrow \theta - \lambda_{\pi^q} \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}} \Big[\nabla_\theta \Big(\log \pi_\theta(a|s) \left(A_\omega(a,s) - \log \pi_\theta(a|s) \right) \Big) \Big]$
    else
      $\theta \leftarrow \theta - \lambda_{\pi^q} \mathbb{E}_{s_t \sim \mathcal{D}, \epsilon_t \sim p(\epsilon)} \Big[\nabla_\theta (Q_\omega(a_\theta, s) - \log \pi_\theta(a_\theta|s)) \Big]$
    end if
    $\bar{\phi} \leftarrow \tau \bar{\phi} + (1-\tau)\bar{\phi}$
  end for
end for

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 16 to 3 (or less if you are really pushed).

Exercise II.1: Tanh-Gaussian Policies

One problem still remains for continuous domains; our action space is finite, but many useful distributions that we would like to use for our policy (such as the Gaussian distribution) have infinite support. We can either choose to clip actions, however derivatives for hard clipping function are not well defined. Instead, we can use a differentiable squashing function to ensure our distribution has finite support. To achieve this, we first sample a variable $\hat{a}$ from Gaussian distribution with mean $\mu_\theta(s)$ and standard deviation $\sigma_\theta(s)$, $\hat{a} \sim \hat{\pi}_\theta(\hat{a}|s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta(s))$ and then pass the corresponding value through a tanh function, $a = |A| \tanh\left(\frac{\hat{a}}{|A|}\right)$. Without loss of generality, we assume $|A| = 1$. Note that the values of $\mu_\theta(s)$ and $\sigma_\theta(s)$ are the state-dependent outputs of a neural network parametrised by θ. As tanh is a monotonic function, we can use the change of variables formula to derive the overall policy pmf:

$$\begin{aligned}
\pi_\theta(a|s) =& \left| \frac{d}{da} \tanh^{-1}(a) \right| \hat{\pi}_\theta(\tanh^{-1}(a)|s), \\
=& \frac{1}{1-a^2} \hat{\pi}_\theta(\tanh^{-1}(a)|s), \\
=& \frac{1}{(1-a^2)\sqrt{2\pi\sigma_\theta(s)^2}} \exp\left(-\frac{(\tanh^{-1}(a) - \mu_\theta(s))^2}{2\sigma_\theta(s)^2} \right).
\end{aligned}$$

II.1 Find an expression for $\log \pi_\theta(a|s)$ in terms of $\log \hat{\pi}_\theta(\hat{a}|s)$ and a. Check this is correct by comparing to line 170 in the file `networks.py`. Why do we add `self.epsilon`?

Exercise II.2: Exploration using Entropy

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, yielding:

$$J(\theta) = \int p_0(s) \int \pi_\theta(a|s) \left(A_\omega(a, s) - \log \pi_\theta(a|s) \right) dads. \tag{3}$$

II.2a Why does the additional entropy term in Eq. (3) encourages exploration? Hint: What happens when the policy is very confident about a particular action? What about when all actions have equal probability?

Taking derivatives of Eq. (3) using the policy gradient theorem with the log-derivative trick then yields the gradient update:

$$\begin{aligned}\nabla_\theta J(\theta) =& \int \rho^\pi(s) \nabla_\theta \int \pi_\theta(a|s) \left(A_\omega(a,s) - \log \pi_\theta(a|s)\right) dads, \\ =& \mathbb{E}_{s\sim\rho^\pi(s), a\sim\pi_\theta(a|s)} \Big[\nabla_\theta \Big(\log \pi_\theta(a|s) \left(A_\omega(a,s) - \log \pi_\theta(a|s)\right) \Big)\Big]. \end{aligned} \tag{4}$$

II.2b Implement the score function gradient update with entropy as derived in Eq. (4) in the function `train_score()`, uncommenting 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 II.3: Reducing Variance using the Reparametrisation Trick

Estimators like the score function estimator from II.1 often have very high variance for their gradient updates, even when a baseline is used. We now see how performance can be further improved by using the reparametrisation trick to exploit the low variance properties of using an update with the first order derivative $\nabla_a Q_\omega(a,s)$. The results are analogous to those derived for variational auto-encoders (Kingma and Welling, 2013)

Instead of sampling an action directly from our policy $a \sim \pi_\theta(a|s)$, we sample a variable ϵ from a zero-mean Gaussian distribution with standard deviation of unity $\epsilon \sim p(\epsilon) = \mathcal{N}(0,1)$. An action is then constructed using $a_\theta = \tanh\left(\epsilon\sigma_\theta + \mu_\theta\right)$. Although not strictly justified in the case of the classic policy gradient theorem, as we have used function approximators for the value functions, to aid our analysis we can shift the derivative outside of the inner integral:

$$\nabla_\theta J(\theta) = \int \rho^\pi(s) \nabla_\theta \int \pi_\theta(a|s)(A_\omega(a,s) - \log \pi_\theta(a|s)) dads \tag{5}$$

II.3a Using the substitution $a = a_\theta = \tanh\left(\epsilon\sigma_\theta + \mu_\theta\right)$, show that Eq. (5) can be written as

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\pi(s), \epsilon\sim p(\epsilon)} \Big[\nabla_\theta (A_\omega(a_\theta, s) - \log \pi_\theta(a_\theta|s))\Big], \tag{6}$$

II.3b Why can we write the gradient in Eq. (6) as

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\pi(s), \epsilon\sim p(\epsilon)} \Big[\nabla_\theta (Q_\omega(a_\theta, s) - \log \pi_\theta(a_\theta|s))\Big]? \tag{7}$$

II.3c Implement the reparametrisation gradient update with entropy as derived in Eq. (7) in the function `train_reparametrisation()`. If implemented correctly, you should start to see the average test return rise above -1000 at an earlier epoch than when the score function estimator is used. Compare the performance of your new plot against the score function plot from II.1b by running the file 'plot_all.py'.

Hints:

- Don't forget to change `training_mode` to `'reparam'` in the file `part_ii.py` before running.
- `new_actions` and `log_pis` have now been calculated for you using the function `r_sample`, sampling ϵ first from $p(\epsilon)$ and then calculating $a_\theta = \tanh(\epsilon\sigma_\theta + \mu_\theta)$. This means that they have dependency on policy parameters θ.
- `q_new_actions` now also have dependency on policy parameters θ, so gradients will automatically flow through these q-values back to the network parameters when `backward()` is called.

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 before climbing.