PSL432 Theoretical Physiology
Department of Physiology
University of Toronto

ASSIGNMENT 3

Assigned 17 March 2020
Due 31 March 2020

This assignment is worth 35% of your final mark in the course. Please do all parts of both questions. For each question you will submit computer code as an m-file. Please email these m-files to douglas.tweed@utoronto.ca and ali.mojdeh@mail.utoronto.ca by 11:59 p.m. on 31 March. Your code will be graded partly on concision and clarity, so keep it brief and orderly.

1. DDPG

Here you will use the deep deterministic policy gradient method to learn to control a two-joint arm. The arm will have the same dynamics as in Assignment 1 except that the time step `Dt` will now be 0.1 s. And you will implement these state dynamics in a function called `arm_dyn` which you will place at the bottom of your m-file (to see how to define a function at the bottom of an m-file, look at the posted sample code DDPG.m).

Your `arm_dyn` function will take `s` and `a` as inputs, and put out *the state at the next time step* (*not* $\Delta \boldsymbol{s}$ or the rate of change of the state), i.e. `s_next = arm_dyn(s, a)`. Your `arm_dyn` should handle single examples of `s` and `a`, not minibatches. It should also prevent the joints going outside their motion ranges, defined by `q_min = [-1.5; 0]` and `q_max = [1.5; 2.5]`. And it should Euler-update `q` before `q_vel`.

Also at the bottom of your m-file, please insert the function `test_policy` from the sample code DDPG.m, but modify it so it calls your `arm_dyn` function in the appropriate place, and its `ylim` variables are reasonable given the data it plots, and it runs only a single example, not a minibatch. When you call it, make its second argument, `s_test = [-1; 1; 0; 0]`.

Use global variables to pass information other than their inputs to your `arm_dyn` and `test_policy` functions, i.e. write

```
global n_q psi q_min q_max n_steps Dt r s_path a_path
```

in your main m-file right after `clear variables; clc; clf;` and then inside `arm_dyn` and `test_policy`, list the global variables they need (much as I did for `test_policy` in DDPG.m).

Make your reward function $r(\boldsymbol{s}, \boldsymbol{a}) = -10\,(\boldsymbol{s} - \boldsymbol{s}^*)^T(\boldsymbol{s} - \boldsymbol{s}^*) - 0.01\,\boldsymbol{a}^T\boldsymbol{a}$, where $\boldsymbol{s}^*$ is a constant vector — a desired state in which the elbow angle is 0.5 (radians) and all other elements of the state are 0. You will compute these rewards using a function `r(s, a)`, defined inside your main m-file, much as I defined the `r` function in DDPG.m. Use no discount factor, or in other words set $\gamma = 1$.

To make it easier to compare your results with mine and with other students', please seed the random number generator by writing `rng(25)` into your code before you create your networks. Create all your networks using the posted function create_net.m, and make sure that you create the policy network `mu` before you create any other networks, and that you perform no other operations involving random numbers before you make `mu`. Give `mu` four layers, with 100 neurons in each of the hidden layers, and activation function `"relu"`. And initialize its weights by setting the rescale vector (the second input to create_net) equal to `[0; 10; 10; 10]`. Then the first five columns of `mu.W{end}`, before learning begins, should equal

```
    0.0198   -0.0404    0.0080   -0.0463   -0.0404
   -0.0234    0.0477   -0.0263    0.0393   -0.0014
```

Later in your code, when you use `mu` to compute actions, please do not use `tanh` to bound the actions, but instead make `a` equal the output of the network `mu`.

Run 2000 rollouts in all, each with a duration of 3, i.e. 30 times `Dt`. Before rollout 1, and then after every 100 rollouts, call `test_policy` and batch_nse.m, as I did in DDPG.m, and print the resulting return `R` and `Q_nse`.

If your `arm_dyn` and `r` functions and your `mu` network are correct, then you should get `R = -72.564` (to three decimal places) the first time you call `test_policy`. If you program DDPG correctly, you should have `R = -17` or better and `Q_nse` mostly < 0.01 by 2000 rollouts. (I got these results using `eta_Q = 1e-3`, `eta_mu = 5e-6`, `tau = 3e-4`, and `a_sd = 0.1`, and a 4-layer `Q_est` network with relu activation and 20 neurons in each of the hidden layers, initialized with rescale vector `[0; 10; 10; 10]`.)

Name your variables as in these pages, the notes, and the sample code, e.g. `q`, `psi`, `M`, `GAMMA`, `s`, `a`, `Q_est`, `Q_tgt`, `mu_tgt`, `eta_mu`, `eta_Q`, `tau`, etc.

Submit an m-file called Yoursurname_Yourinitial_A3_Q1.

2. Factoring *Q*

Here we will decompose the action-value function Q into a combination of more-basic functions, in the hope that by exploiting its internal structure we can improve learning. For you, this question will be an exercise in making multiple networks operate together, and particularly in figuring out what signals to backpropagate through which networks.

To begin, we will define the *value* function $V^{\boldsymbol{\mu}}(\boldsymbol{s}) = Q^{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{\mu}(\boldsymbol{s}))$, i.e. $V^{\boldsymbol{\mu}}(\boldsymbol{s})$ is simply $Q^{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{a})$ when $\boldsymbol{a}$ is chosen to fit the policy $\boldsymbol{\mu}$, or in other words it is the return we will get from now on if we are currently in state $\boldsymbol{s}$ and we choose all our actions, including the current one, using policy $\boldsymbol{\mu}$.

We will also redefine the state-dynamics function $\boldsymbol{f}$ so it delivers the *next state* rather than the velocity — i.e. $\boldsymbol{s}_{t+\Delta t} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t)$ — just as your function `arm_dyn` did in Question 1.

Next, observe that $Q(\boldsymbol{s}_t, \boldsymbol{a}_t) = r(\boldsymbol{s}_t, \boldsymbol{a}_t)\,\Delta t + V(\boldsymbol{s}_{t+\Delta t}) = r(\boldsymbol{s}_t, \boldsymbol{a}_t)\,\Delta t + V(\boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t))$, i.e. $Q = V \circ \boldsymbol{f} + r\,\Delta t$ (again setting the discount factor $\gamma = 1$, as in Question 1). Your job is to code a variant of DDPG that has no $\langle Q \rangle$ net but instead works with networks $\langle V \rangle$, $\langle \boldsymbol{f} \rangle$, $\langle r \rangle$, and also target networks V^+ and $\boldsymbol{f}^+$, but no Q^+ or $\boldsymbol{\mu}^+$. The control task will be the same as in

Question 1, with the same `arm_dyn` and `r` and `test_policy` functions, the same structure for the replay buffer, and the same `rng(25)`.

Each of your six networks should have four layers and relu activation. $\langle \boldsymbol{f} \rangle$, $\boldsymbol{f}^+$, and $\boldsymbol{\mu}$ should have 100 neurons in each hidden layer whereas $\langle V \rangle$, V^+, and $\langle r \rangle$ should have 20. Initialize all your networks' weights using the same rescale vector as you used for $\boldsymbol{\mu}$ and $\langle Q \rangle$ in Question 1.

You should train $\langle r \rangle$ in the obvious way based on minibatches drawn from the buffer, i.e. using the error $e^r = \langle r \rangle(\boldsymbol{s}\_, \boldsymbol{a}\_) - r\_$ where $\boldsymbol{s}\_$, $\boldsymbol{a}\_$, and $r\_$ are from the buffer. Not so obviously, you should train $\langle \boldsymbol{f} \rangle$ using $e^f = \langle V \rangle(\langle \boldsymbol{f} \rangle(\boldsymbol{s}\_, \boldsymbol{a}\_)) - \langle V \rangle(\boldsymbol{s}_{\text{next}}\_)$, i.e. $\langle \boldsymbol{f} \rangle$ needn't necessarily predict the correct $\boldsymbol{s}_{\text{next}}\_$, but it should predict a new state whose *value* matches that of the true $\boldsymbol{s}_{\text{next}}\_$ (because accurately predicting $\boldsymbol{s}_{\text{next}}\_$ is harder than predicting $\langle V \rangle(\boldsymbol{s}_{\text{next}}\_)$, and doesn't help the policy learn any better, for reasons that should become clearer as you work through the question). Your main challenge is to work out how to train $\langle V \rangle$ — by figuring out what the Bellman error looks like in this setting — and then adjust $\boldsymbol{\mu}$. As in Question 1, you should *not* bound the actions, but instead make $\boldsymbol{a}$ equal the output of the network $\boldsymbol{\mu}$.

Run 2000 rollouts in all. Before learning begins, and then after every 100 rollouts, call `test_policy` using the same `s_test` as in Question 1, and also call batch_nse.m to compute `r_nse`, `f_nse`, and `V_nse` on the most-recent minibatch, and print those 3 variables alongside the `R` from `test_policy`.

If all goes well, `R` should be > –16 and all the nse's < 0.002 by 2000 rollouts (I got good results using `eta_V = 1e-3`, `eta_f = 1e-3`, `eta_r = 1e-3`, `eta_mu = 5e-6`, `tau = 3e-4`, `a_sd = 0.1`).

Submit an m-file called Yoursurname_Yourinitial_A3_Q2.