Mandarory Assignment 2
Proximal Policy Optimization (PPO)

TEK5040/TEK9040
Fall, 2020

Contents

1 Preliminaries

1.1 Introduction

In this assignment we will look at CarRacing environment from OpenAI Gym. We will implement a version of the PPO algorithm, where the pseudocode is given below.

Algorithm 1 PPO, Actor-Critic Style

Initialize value network v_η with random weights.
Initialize policy network π_θ with random weights.
Initialize $\theta_{\text{old}} = \theta$.
for iteration $= 1, 2, \dots$ **do**
  for i $= 1, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$in environment (possibly limit timesteps)
    Compute advantage estimates $\hat{d}_1, \dots, \hat{d}_{\tau^{(i)}}$
  end for
  Set surrogate objective L based on the sampled data.
  Optimize surrogate L wrt. η and θ, for K epochs and
  minibatch size $M \leq \sum_{i=1}^{N} \tau^{(i)}$.
  $\theta_{\text{old}} \leftarrow \theta$.
end for

1.2 Preparation

1.2.1 Installation

We need to have the `gym` python package installed, as well as the box2d environment. The exact installation process will depend on your operating system and setup. You may look at the **requirements.txt** file for python dependencies, using *pip* the dependencies may be installed by

```
pip3 install -r requirements.txt
```

In some cases there may be additional system (non-python) dependencies that are not met. On Ubuntu, if you get error messages about "missing swig", you may install this package with

```
sudo apt install swig
```

1.2.2 Visualize random agent

Test your installation by running 100 steps of a predefined random policy and visualize the results

```python
import gym

env = gym.make('CarRacing-v0')
observation = env.reset()
for t in range(100):
    env.render()

    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    if done:
        print("Episode finished after {} timesteps".format(t+1))
        break
env.close()
```

1.2.3 Try playing yourself!

You may get a feel of the game by trying to play yourself. You first need to locate where the `gym` package has been installed. If you installed using `pip3` on Ubuntu, it should be found at either **/usr/local/lib/python3.x/site-packages/gym** or **$HOME/.local/lib/python3.x/site-packages/gym** depending on whether you used the `--user` flag or not. Here x is e.g. 7 if you use python3.7, and *$HOME* is the path to your home directory. After you have located the package, you may run

```
python3 /path/to/gym/envs/box2d/car_racing.py
```

and use the arrows on your keyboard to control the car.

1.3 Environment

1.3.1 State space

What should our state space be?

After taking an action at time step t, the agent receives a reward $r_{t+1} \in \mathbb{R}$ and a new observation $o_{t+1} \in \mathbb{R}^{96\times96\times3}$, i.e. a color image with height and width of 96. An example observation is given in Figure 1. Note that this observation has much lower resolution then what you get when you use `env.render()`.

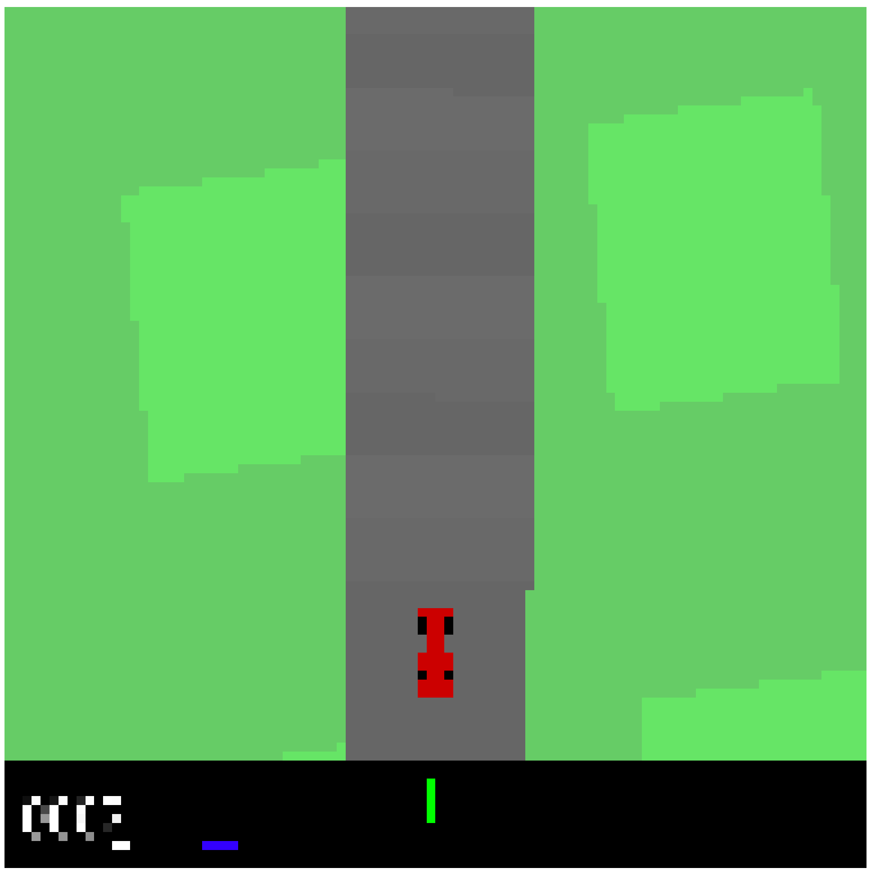

Figure 1: Example observation

We would like to avoid having to use the whole history of observations and rewards to make decisions. We would like to map the history $h_t = (o_0, a_0, r_1, o_1, \ldots, a_{t-1}, r_t, o_t)$ into something more managable, by some function f. The first idea might be to just use the last observation (last frame), such that $f(h_t) = o_t$. Is this sufficient?

From a single frame an agent can figure out its position on the tracks and also the heading of the vehicle. What about e.g. speed though? Clearly when deciding to break or not before a turn, it would be useful to know how fast we are going. Normally one would need at least *two* frames to figure out speed and *three* frames if one would also like to know acceleration (why?). If we however look at the bottom part of the frame, it actually provides some additional information: From left to right the bars are indicating: true speed, four ABS sensors, steering wheel position and gyroscope.

In theory the agent can thus learn about its speed by extracting this information from the leftmost bar in the observation. Whether the agent will actually be able to do this is not clear though. The information has a very different meaning than most other pixels, which gives information about a particular point on the track. Unlike other pixels, the information is actually very high-level, and it would probably have been easier for the agent if we directly gave this as an additional input that could be combined with the extracted information from the image features at a later stage.

One more thing. When we train our car-driving agent, we are going to cut the episodes short, say after a few hundred time steps. If we want to learn a good value function, and only use o_t as input we do have a potential issue. Assume we are going at full speed with a straight lane ahead of us. If you have a fairly decent policy, this would mean that things are looking quite good and you should expect a fair amount of reward. If however, you are actually at the final time step before we cut the episode short, you will actually not get any more reward! Thus it will be very hard to learn a value function without any information about how much time we have left[1]. For the value function, we will thus give an additional input that gives information on how much time there is left before the episode ends.

For this exercise we shall let our state at time t be $s_t = (o_t, \text{time\_remaining}_t)$. The policy network will only use o_t, while the value network will use both.

1.3.2 Action space

The enviroment has a continuous action space : $[-1, 1] \times [0, 1] \times [0, 1]$ which we in this exercise simplify into a discrete action space with 5 actions:

- turn left: $[-1, 0, 0]$

[1]Even if we didn't cut our episodes short, the issue still remains as the episode would still end after we have finished one lap

- straight: $[0, 0, 0]$
- turn right $[1, 0, 0]$
- gas: $[0, 1, 0]$
- break: $[0, 0, 1]$

Note that this limits us to only a subset of the possible actions, which may impact how well we can do. Note that the actions above are also at their extreme values, e.g. if we want to increase the speed, we have to put the gas at full throttle!

1.3.3 Time scale

It is not always clear what the right time scale to operate at is. You would normally like to be able to perform actions just "often enough". If your time scale is too fine-grained, you may just end up running your policy network a lot, just to find that you are going to repeat your last action. If the time-scale is too coarse on the other hand, you may not be able to switch actions often enough to get a good policy. For the *gym* environments, the finest time scale we can get is decided by the environment. We can however get a more coarse-grained time scale by *repeating actions*. If we choose to repeat actions say k times, we get a new environment where the agent only sees every k frames, and where the "immediate" reward is the sum over the four following rewards. One step in the new environment is thus on the form

```
action_repeat = 4
reward = 0
for _ in range(action_repeat):
    observation, r, done, info = env.step(action)
    reward = reward + r
    if done:
        break
```

In this assignment we choose 4 repeats during training.

2 Implementation

This section contains information on your implementation tasks. The only file you should need to modify is **ppo.py**. For each of the tasks here, you will find *TODO* comments in this file. Any functions we refer to here are also in that file. The policy network and value networks have already been implemented for you, and your main focus will be on the learning algorithm. As default, both the policy network and value network are *linear* functions. You should have one run with the paramaters already given, but may if you like, also try additional configurations of the networks and/or other hyper-parameters. Note that each iteration takes on the order of one minute on a not too powerful laptop. Thus if you want a "full" run of 500 iterations, you probably want to run it over night. Checkpointing has been implemented, so that you may start and stop training at your convenience. If you *don't* want to continue from a previous checkpoint, you should either delete the corresponding directory or change the `run_name` parameter.

Your may run your program by e.g.

```
python3 ppo.py
```

Note that if when trying to run the script you get the error message

```
ImportError: No module named 'car_race'
```

you need to add the parent directory of the **car_race** directory to your PYTHONPATH environmental variable.

It is recommended to reduce the `num_episodes` and `maxlen_environment` parameters to low values, e.g. 2 and 12, during debugging, as this will greatly reduce the time used on dataset creation.

2.0.1 TensorFlow hints

TensorFlow low-level operations work similar to numpy and are designed to work on tensors/arrays. Furthermore many operators like e.g. $+$, $-$, $*$ and / works elementwise for tensors/arrays. E.g. to take the elementwise difference between two arrays y and v this may be accomplished with

```
diff = y - v
```

A lot of the basic operations in TensorFlow is located in the `tf.math` namespace, see `https://www.tensorflow.org/api_docs/python/tf/math`. Operations that aggregates information are named `tf.math.reduce_*`, e.g. `tf.math.reduce_sum` and `tf.math.reduce_mean` calculates the sum and mean respectively. The `axis` argument may be only used to aggregate over certain dimensions.

2.1 Return

Implement the `calculate_returns` function. Recall that the return at a time step is calculated as

$$g_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where r_t is the reward at time step t and γ is the discount factor.

2.2 Value loss

Implement the `value_loss` function. Given a batch of value predictions $v_1, \ldots v_N$ and corresponding target values $y_1, \ldots, y_N$ we define the loss as

$$\frac{1}{N} \sum_{i=1}^{N} (y_i - v_i)^2$$

where N is the batch size.

2.3 Policy loss

Implement the function `policy_loss`. Given a batch of empirical state-action pairs (s_i, a_i) and estimated advantages $\hat{d}_i = g_i - v_\eta(s_i)$, the policy loss should be calculated as

$$-\frac{1}{N} \sum_{i=1}^{N} \min\left(u_i(\theta)\hat{d}_i, \operatorname{clip}(u_i(\theta), 1-\epsilon, 1+\epsilon)\hat{d}_i\right)$$

where N is the batch size and we have defined

$$u_i(\theta) = \frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)}$$

2.4 Entropy loss

Implement the function `entropy_loss`. You may use the function `entropy` to calculate the entropy for each sample in the batch. The entropy loss should be the average negative entropy over the batch.

Adding an entropy bonus for policy gradient like methods is a common way to encourage the policy not to be too deterministic. This might improve exploration and reduce the potential risk of getting stuck in a local optima.

2.5 Optimization of surrogate objective

For each training *iteration*, optimize the surrogate loss[2] for K epochs, where K is a hyperparamter set to 3 for this exercise, and minibatches of size M, where we here set M to 32.

Note that `policy_network`, `value_network` and `optimizer` has already been initialized for you, and should be used here. To get the logits over actions ("unnormalized probabilties") you may use `policy_network.policy(observation)`. To get predicted returns at timestep t, call `value_network(observation, maxlen-t)`. You may iterate over the dataset for an epoch by

```
for batch in dataset:
    observation, action, advantage, pi_old, value_target, t = batch
```

where each of the tensors above have $M = 32$ elements.

Your loss should be of the form:

```
loss = policy_loss(pi_a, pi_old_a, advantage, epsilon) \
       + c1*value_loss(value_target, v) \
       + c2*entropy_loss(pi)
```

2.6 Improvement estimations

Implement the functions `estimate_improvement` and `estimate_improvement_lb`. These functions are only used for summary purposes, and do not have any impact on the optimization. We use them to to calculate estimated improvements policy improvements for the policy iteration, as well as an estimated lower bound. You will later (see Section 3.6) compare these estimates against actual improvements for the policy.

`estimate_improvement` should for a batch return an array with values

$$\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\gamma^t\hat{d}_t$$

where $\hat{d}_t$ is the advantage estimate.

If we define

$$u_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

[2]Optimizers in TensorFlow always tries to *minimize* the objective function.

then `estimate_improvement_lb` should return an array with values

$$\min\left(u_t(\theta)\hat{d}_t, \operatorname{clip}(u_t(\theta), 1-\epsilon, 1+\epsilon)\hat{d}_t\right)\gamma^t$$

3 Report

For the report, you should answer the questions below. Note that the questions do not necessarily have precise answers. They are meant to make you think about e.g. how small changes to the environment might affect the difficulty of the problem and the appropriateness of different models. Try to answer the questions as best you can.

3.1 Linear vs non-linear policy

Assume that the state of the policy is just the last observation o_t (as has been the case for this exercise). Do you expect a linear policy to work? Do you expect the *optimal* policy to be linear? If not, what are some things you expect that the linear policy will have have issues with? If the camera view did not follow the position and direction of the car, but rather was a fixed view over the entire course, do you believe a linear policy would have worked better or worse? Can you think about a simple change to the environment that would make a linear policy not work at all?

3.2 Linear value network

Assume that the state used by the value network is $s_t = (o_t, \text{time\_remaining}_t)$. Do you expect the *true* value function v_π of the optimal policy (or any decent policy) to be linear? If not, what are some effects that a linear value network is not able to capture?

3.3 Convolutional vs fully connected model

What are some common arguments for using *convolutional* neural networks instead of fully connected neural networks? Are the arguments valid for the observations from this environment? Would you expect a convolutional neural network to work better than a fully connected neural network here?

3.4 Visualization of policy network weights

You may visualize the weights of a linear policy network saved at **/path/to/saved/model** with

```
python3 vis_filters.py /path/to/saved/model
```

To visualize the weights for the highest scoring model for a run

```
python3 vis_filters.py train_out/<run-name>/high_score_model
```

where <run-name> is the name you have given the run in **ppo.py** (*ppo_ linear* by default). The weights are organized as in the pattern below.

	gas	
left	straight	right
	break	

Save this figure and put in the report. Are you able to interpret the weights in any way?

3.5 Eval policy

Evaluate your model for the highest scoring iteration (in terms of mean) by

```
python3 eval_policy.py --num_episodes 32 \
                       --policy </path/to/high_score_model> \
                       --action_repeat <N>
```

Report scores for <N> equal to 1, 2, 4 and 8. Report both minimum, median, mean and maximum scores for all cases.

For each evaluation you will also get a video showing your agent for the best episode. How would you judge its performance qualitatively?

3.6 Actual improvement vs "predicted"

With proximal policy, at each iteration we try to maximize[3]

$$L(\pi_\theta) = E_\pi\Big[\sum_t \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\text{old}}}(A_t|S_t)}\gamma^t d_{\pi_{\theta_{\text{old}}}}(S_t, A_t)\Big]$$

[3]Actually we do not include the γ^t factor, in our loss function. A reason for this as that we may not only care about the return from the initial state, but might be interested in getting high returns from later states as well.

which is the policy improvement we would get when we ignore changes in state visitation frequencies. We do this by optimizing an approximation

$$\max_\theta \hat{E}[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\gamma^t \hat{d}_t]$$

over a sampled dataset. We can thus estimate the expected change in policy improvement by measuring this quantity. Compare the *estimated_ improvement* and *estimated_ improvement_ lb* values in TensorBoard, to the estimated change in expected return. Are the estimates accurate? Do they have systematic biases? Note that you need to look at the *accumulated* values over the iterations between each policy iteration.

4 Delivery

You should hand in your assignment on Devilry. It should include:

- Your code (including the files handed out to you)
- TensorBoard log files
- Your report

You should zip everything together into one file.