Mandarory Assignment 2
Proximal Policy Optimization (PPO)
on OpenAI-gym Car Racing environmnet

TEK5040/TEK9040 Autumn, 2023

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 Algorithm 1.

More details of the algorithm and implementation can be found in a later section.

1.2 Preparation

1.2.1 Installation

When you extract the assignment package you will see the following files/folders:

- `assignment.pdf`: This document
- `requirements.txt`: requirement specification for python packages
- `car_race`: Folder containing the following files.
  - `ppo.py` is the ONLY file you are supposed to modify
  - `test_installation.py`: script to test your installation
  - `eval_policy.py` : script to evaluate your policy after training
  - `vis_filters.py`: script to show filter weights of the policy network after training
  - `baselines.py` : source file
  - `common.py`: source file
  - `networks.py` : source file
  - `videofig`: folder containing three files which do not require your attention
- `keras_fix` : folder containing a file called `load.py`, which may fix Tensorflow model reading problems.

We also 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, and using *pip* the dependencies may be installed by

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

In particular, you need the specific `gym` version `0.25.0`. We have tested the provided scripts with `Tensorflow 2.6.0` and `Python 3.6` on an Ubuntu Linux platform.

(If your Tensorflow version is greater than 2.7 and you encounter errors regarding loading saved models, you may need to replace the `$TENSORFLOW/keras/src/saving/legacy/saved_model/load.py` with the provided in `keras_fix/load.py`.

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 Test your installation

Test your installation by running the provided script `test_installation.py`.

```
python3 test_installation.py
```

It will execute ca. 1000 steps of a predefined random policy and shows the graphical environment.

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 (`PATH_TO_GYM`). If you installed using `pip3` on Ubuntu, it should be found at either

- **/usr/local/lib/python3.x/site-packages/gym** or
- **$HOME/.local/lib/python3.x/site-packages/gym** or
- **$TENSORFLOW/lib/python3.x/site-packages/gym**

depending on whether you used the `--user` flag or you installed your system in a virtual environment. Here x is e.g. 6 if you use python3.6. *$HOME* is the path to your home directory or *$TENSORFLOW* is the home of your virtual environment. 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 (up arrow to move forward, down arrow to brake and left/right arrows to turn).

1.3 Environment

This is one of the easiest environments to perform reinforcement learning where the observations are images (array of pixels). The environment provides a top-down view of the track and the racing car (See Figure 1). The track consists of series of tiles and an episode is normally defined as completing a full round along the track (i.e. visiting all the tiles) [1]. The car also can go outside of the PLAYFIELD - that is far off the track, then it will get a negative reward and die. The problem is considered to be solved when the agent gets a reward greater than a predefined threshold.

1.3.1 Observation and State space

At each time step t the observation would be $o_t \in \mathbb{R}^{96\times 96\times 3}$, i.e. a color image with height and width of 96. An example observation is given in Figure 1. This $(o(t))$ includes the control information bar at the bottom, which gives a different types of information than the top-down view of the racing track. It is possible to remove this (control information bar) from the observations, but in our case we do not remove it.

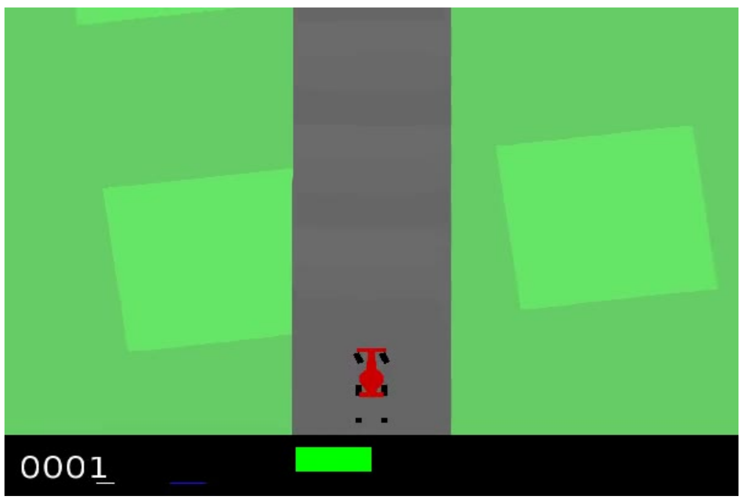

Figure 1: Example observation

For this exercise we make use of two definitions of the state $s(t)$ at time t.

- $s_t = (o_t)$: Our policy network will use this definition of state.
- $s_t = (o_t, \text{time\_remaining}_t)$: Our value network will use this definition, where time_remaining_t is time remaining until the end of the episode. i.e. $\text{time\_remaining}_t = T - t$, where T is the total number of time steps in an episode.

[1] However, in this exercise an episode is defined as a predefined number of time steps, *maxlen*

1.3.2 Action space

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

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

The reward is -0.1 every frame and $+1000/N$ for every track tile visited, where N is the total number of tiles in the track. For example, if you have finished visiting all the tiles in a track in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points. The game is considered solved when the agent consistently gets 900+ points. The generated track is random in every episode. The car also can go outside of the PLAYFIELD - that is far off the track, then it will get -100 and die.

1.3.4 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 Task

This section contains information on your implementation tasks based on the PPO algorithm listed in Algorithm 1.

Algorithm 1 PPO, Actor-Critic Style

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

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 hyperparameters. 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
```

If there are any empty windows that pops up, just ignore these, you should not expect any visualization here. If 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 Implementation hints

When compared to the pseudo code given in Algorithm 1, the implementation task corresponds to lines 9, 10 and 11. That is you just need to compose the loss function and optimize it using the usual `tf.tape.gradient`. The core of implementation task is therefore to form the loss function.

Lines 5-10 in the pseudo code creates the data samples which you can use in calculation of the loss. This part (creation of data samples) is implemented in lines 311-318 in `ppo.py`. Each data sample consists of the following components:

- observation o_t. This is also the same as state in this task
- action a_t
- advantage $\hat{d}_t$. This is evaluated as the difference between the return g_t and the value function v_t (i.e. $\hat{d}_t = g_t - v_t$). The value function is predicted using the value network which takes $(o_t, T-t)$ as input, i.e. $v_t = v_\eta(o_t, T-t)$ where v_η is the value network.
- The old policy π_{old}. More specifically it gives the probability of action a_t output by the previous(old) policy network.
- value_target y_t. This is just the return g_t calculated using the old policy
- time step t

2.2 Implementation tasks

2.2.1 Return

Implement the `calculate_returns` function (line 90 in `ppo.py`). Recall that the return at a time step is calculated as

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

where r_t is the reward at time step t, γ is the discount factor and T is the episode length.

2.2.2 Value loss

Implement the `value_loss` function (line 107 in `ppo.py`). Given a batch of value predictions $v_1, \dots v_N$ and corresponding target values $y_1, \dots, y_N$ we define the loss as

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

where N is the batch size.

2.2.3 Policy loss

Implement the function `policy_loss` (line 121 in `ppo.py`). 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

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

where N is the batch size and we have defined

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

2.2.4 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. (start with line 330 in `ppo.py`)

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)
```

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 Actor-critic

This task implements an actor-critic architecture. Identify the actor and critic in this system. Name an advantage of having an actor.

3.2 Observation and action sampling

Write down the number of the line (in `ppo.py`) where

- the action probabilities are calculated
- an action is sampled based on the probabilities above
- new state (same as an observation in this task) is generated based on the action

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

3.3 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). Can you give an example state with which a linear policy can work satisfactorily? Do you expect the *optimal* policy to be linear?

3.4 Value function example

Give an example of a state, i.e. (o_t, time_remaining) which has a high value function (high average return). Can you think of a state having a low value function.

3.5 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 break	right

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

3.6 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?

4 Delivery

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

- The completed `ppo.py` file
- Your report

You should zip everything together into one file. Please do NOT submit Tensorflow log files or any other files which are not specified above.