Tilburg University, Department of Cognitive Science and Artificial Intelligence

Deep Reinforcement Learning Assignment

Spring 2024

Welcome to the mid-term assignment for the Deep Reinforcement Learning course, Spring 2024. This assignment is composed of **two** parts.

You are asked to submit a report outlining your development of the exercises in the assignment, with particular focus on motivating your choices and discussing your results.

You are free to use any software library you prefer, including writing your own implementation of the algorithms (useful activity, but not advised). For example, you can look at implementations of RAINBOW on Github, or use stable-baselines3 [1] or similar libraries. *Please motivate your choice briefly.*

The deadline for submitting your report is **June, 16th** at midnight. Please note that the final exam is currently scheduled for June 10th, so you are advised to not wait until the deadline to work on this assignment. In particular, the code/exercises should not be excessively difficult, but please bear in mind that training DRL agents can be quite time consuming.

1 Software

For this assignment, you are advised to use stable-baselines3 and Mujoco.

```
pip install gymnasium
pip install --upgrade stable-baselines3
pip install mujoco
```

Notes:

- **Please use the official 'mujoco' library instead of the now deprecated 'mujoco-py'!** Please note that the installation guides online are deprecated, since they refer to a very old version of Mujoco. Official and up-to-date information is found at `https://mujoco.org`.

- Make sure you use the very latest version of stable-baselines3, since support for the new Gymnasium library (OpenAI Gym has stopped development, and the new project has been renamed Gymnasium) has only been introduced recently, in April 2023.
- You are strongly advised to use normal .py python code instead of notebooks. Python notebooks are poorly suited for DRL.
- Please install the software as soon as possible and run example code to make sure everything is configured correctly. If you do not manage to solve installation problems within a few days of trying, please let me know.

2 [50 pts] Part 1: generalization of trained policies to different environment dynamics

The objective of the first part of the assignment is to reproduce the first 3 panels of Figure 1 [2] (`https://openreview.net/references/pdf?id=HkU1G1VGe`), included here as Fig. 2.1. You are of course welcome to try and reproduce the last panel, as well, if you wish. ;) Doing so will require modifying the code of the Wrapper below.
The task is based on the Mujoco Hopper environment (`https://www.youtube.com/watch?v=jtXiTP96wow`), which requires a DRL agent to control a jumping agent made up by a torso and a single leg, to make it run as fast as possible.
The task investigates the generalization performance of agents trained in an environment with fixed parameters (e.g., body mass, length of each body part, force of gravity, etc) and evaluated in different environments. In particular, the mass of the torso of the hopping agent will vary in the evaluation environments. You will observe that when the dynamics of the simulation change too much, the performance of the trained agent will degrade significantly. This is a sim2sim demonstration (agent trained in simulation, and tested in a different simulation) of the problems encountered with sim2real transfer (training an agent in simulation, and testing it on a real robot; `https://lilianweng.github.io/posts/2019-05-05-domain-randomization`).
Specifically, you need to:

- (10/50 pts) Train a DRL model of your choice (do justify your choice) on the **Hopper-v4** environment, choosing the rigth algorithm and hyperparameters.
- (10/50 pts) Train 3 different models, each with a different torso mass (3kg, 6kg, and 9kg). Ideally, you should repeat each training run with a few different random seeds (typical values are 3-5 random seeds), if runtime allows it.
- (30/50 pts) Test each trained model over the full range of torso masses (3, 4, 5, 6, 7, 8, 9), to reproduce the figure.

You can use the following code to create 'Hopper-v4' Gymnasium environments with custom torso masses:

```python
import gymnasium as gym

from stable_baselines3.common.vec_env import VecNormalize
from stable_baselines3.common.env_util import make_vec_env

class ChangeMassWrapper(gym.Wrapper):
  def __init__(self, env, torso_mass=6):
  super().__init__(env)

  self.torso_mass = torso_mass
  self.env.model.body_mass[1] = self.torso_mass
```

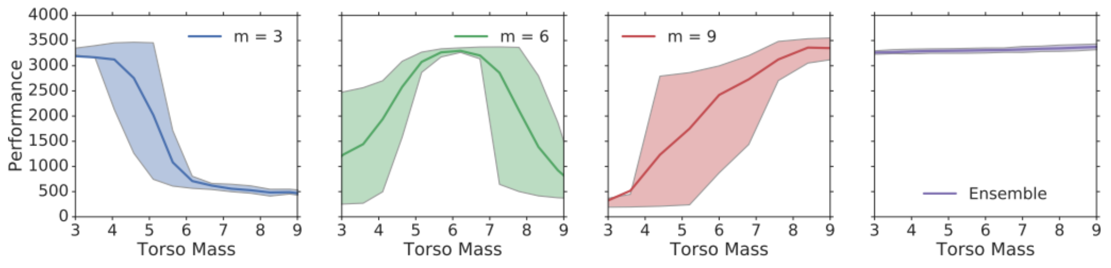

Figure 1: Performance of hopper policies when testing on target domains with different torso masses. The first three plots (blue, green, and red) show the performance of policies trained with TRPO on source domains with torso mass 3, 6, and 9, respectively (denoted by $m =$ in the legend). The rightmost plot shows the performance of EPOpt($\epsilon = 0.1$) trained on a Gaussian source distribution with mean mass $\mu = 6$ and standard deviation $\sigma = 1.5$. The shaded regions show the 10th and 90th percentile of the return distribution. Policies trained using traditional approaches on a single mass value are unstable for even slightly different masses, making the hopper fall over when trying to move forward. In contrast, the EPOpt policy is stable and achieves a high level of performance on the entire range of masses considered. Further, the EPOpt policy does not suffer from degradation in performance as a consequence of adopting a more robust policy.

Figure 2.1: Fig. 1 of [2]. In plain Behavior Cloning, small deviations from expert trajectories at test time lead to compounding errors and a catastrophic drop in the trained agent performance.

```

env = make_vec_env('Hopper-v4', n_envs=[XXXX], seed=[XXXX], vec_env_cls=SubprocVecEnv, wrapper_class=ChangeMassWrapper, wrapper_kwargs=dict(torso_mass=[XXXX]))
env = VecNormalize(env, norm_obs=True, norm_reward=True)
```

Please note that Mujoco environments usually require normalization of observations. An example is shown in the stable-baselines3 documentation, under ‘Examples’ (PyBullet environment). The relevant part is shown in the previous code box. You should save and load normalization statistics with the following code:

```
# After training, you should save the env statistics if you want to use saved models
env.save('Hopper-v4_vecnormalize.pkl')

# Then, when you load a saved model, you can create a new environment and wrap it with VecNormalize as follows:
env = ...
env = VecNormalize.load('Hopper-v4_vecnormalize.pkl', env)
env.training = False
env.norm_reward = False
```

3 [50 pts] Part 2: on-policy vs off-policy algorithms

In Part 2 of this assignment, you will explore the differences between on-policy and off-policy reinforcement learning algorithms.

Using the ***BipedalWalker-v3*** environment from Gymnasium (`https://www.youtube.com/watch?v=14yGAsIG-Rs`), your tasks are:

- (20/50) Choose (and motivate) two RL algorithms, one on-policy and one off-policy, and optimize their hyperparameters to achieve the fastest learning and the highest performance on the environment ‘BipedalWalker-v3’ form Gymnasium. Optimize the two algorithms separately, such that each algorithm gets its ‘best shot’ at solving the problem in as few environment timesteps as possible and as little time as possible, while achieving a final score of 300+.

- (30/50) Investigate (in the way you prefer) the differences, pros and cons, of the two methods. Of particular interest for the comparison are: number of environment steps required to learn to solve the environment (i.e., to reach the full score), wall-clock time (i.e., hours/minutes/seconds it takes for the agent to solve the environment on your computer), maximum score achieved (in the unlikely case your agent does not learn to solve the environment), stability of learning across different random seeds, etc. . .

Following are a few tips and notes that you may find useful:

- The maximum score in the environment is around 300.
- You can set random seeds with

```python
from stable_baselines3.common.utils import set_random_seed
set_random_seed(seed, using_cuda=True)  # using_cuda=True onl2y if you
    are using a GPU
```

- Not all DRL algorithms are directly applicable, and you are not limited to algorithms we covered in class. For example, DQN (apart from being a terrible choice in general ;)) does not directly support continuous actions, which are instead used in this environment. You can see a comparison of a few popular DRL algorithms (implemented in stable-baselines3) at `https://stable-baselines3.readthedocs.io/en/master/guide/algos.html`
- Logging your training using Tensorboard will save you a lot of effort. Please look up tutorials online if you are not sure how to use it. You enable tensorboard logging in stable-baselines3 by adding a tensorboard_log="directory-to-log-to" in the constructor of your drl algorithm (e.g., PPO or DQN; see sb3 documentation for more details), and you can then name your experiments with, e.g., tb_log_name="ppo", as argument to the "learn" method.
- While it should not be a major issue here, you may want to make sure that the comparison between your algorithms is as fair as possible. This may involve making sure that the network sizes and architectures (you can print them by accessing model.policy) are similar, and that you use an appropriate number of parallel environments (you can keep increasing the number until the frames-per-second ‘fps’ stop increasing). Another example is the exploration fraction in DQN (or similar algorithms) or the number of timesteps between updates to the target network: e.g., too much exploration will slow down learning, but too little may prevent your agent from learning, or may get it stuck at a suboptimal policy.
- Be wary of default hyperparameters in stable-baselines3. While they may be good starting point for many algorithms and environments, some algorithms (like DQN) can be very sentitive to changes in their value, and it can be that the default hyperparameters are not suitable for this environment.
- It is not uncommon for DRL agents to take a few million timesteps to learn, especially with on-policy methods. If the rewards plot for your agent keeps steadily increasing, you should continue training for longer.

References

[1] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, “Stable baselines3,” 2019.

[2] A. Rajeswaran, S. Ghotra, B. Ravindran, and S. Levine, “Epopt: Learning robust neural network policies using model ensembles,” *arXiv preprint arXiv:1610.01283*, 2016.