ASSIGNMENT

CODE, TRAIN AND EVALUATE YOUR OWN CRAFTER AGENT

1 INTRODUCTION

Your task is to implement a Deep Reinforcement Learning (DRL) agent able to play the game of CRAFTER better than the baseline obtained by a coding agent. You are encouraged to focus on developing any of the various challenges associated such as exploration, credit-assignment, optimization, or hierarchical learning. *The aim is for you to have fun while exploring the space of possible architectures and algorithms for solving this environment.*

CRAFTER [9] can be seen as a simpler, 2D version of MINECRAFT, and it shares with it the procedural generation of maps at every episode and some of the complexity of its technological tree, therefore making it a suitable environment for testing the ability of agents to generalize and learn policies requiring long-term credit assignment. Deep Reinforcement Learning (DRL) benchmarks usualy require a lot of time and resources for training agents that perform well, but CRAFTER is designed to be limited at 1M training steps.

1.1 Timetable

- **November 10, 11:59pm** upload all the materials from 2.1 on Moodle.
- **November 11, 04:00pm** half of the teams present their results.
- **November 18, 04:00pm** second half presents their results.

1.2 General rules

- You are strongly encouraged to form teams of two members. Individual submissions are allowed but not encouraged since this is project assumes a fair amount of workload.
- You are not allowed to just run already implemented algorithms from popular DRL frameworks and report the results. Implement the algorithms yourself!
- You are allowed to consult various baselines and available code.
- You are allowed to interact with other teams and consult on your solutions but do not copy each other's solutions.
- You are allowed to implement any existing Reinforcement Learning (RL) algorithm or devise something of your own. You can come up with solutions not involving neural networks if you want to. You can optimize your agent using gradient-based methods or evolutionary algorithms. There

are really no limitations as long as the method you propose is interesting and the performance is above that of a random agent.

2 INSTRUCTIONS

2.1 Files you need to deliver

Only one of the team members needs to upload the following on moodle:

1. An archive with the source files of your agent(s), including the file `train.py`. The `train.py` file should contain the original `argparse.ArgumentParser` with any additional default hyperparameters such that when executing `python train.py` without any arguments the agent starts training with the best hyperparameters you found during your project. Name the archive using one of your team members name such as `surname_name_middlename.zip`. Make sure **not to include** any large files such as checkpoints.
2. A short `pdf` slide-deck named `surname_name_middlename.pdf` that you will use to present your results. It should contain:
   (a) A description of your method, including the objective function of the algorithm and any enhancements you proposed.
   (b) Plots depicting the performance your agents during training and during evaluation or any other metrics you find interesting such as the evolution of the **loss**, **q-values**, etc. **The average episodic reward is mandatory**. You can generate it by executing `python analysis/plot_eval_performance.py --logdir logdir/your-agent` in the provided starter code after you trained your agent. Make sure you include the performance of the random agent for comparison. Bonus if you plot the spectrum of the success rate on various skills, as shown in [9]. The final reported performance of the agent should be the average of two or three training runs with different seeds.
   (c) Any interesting emergent behaviours you observe, see for example `Sec.4.3` in [9].

2.2 Resources

Main resources:

- CRAFTER repository and installation instructions.
- Starter code available on the moodle assignment page. It already configures for you a wrapper over the CRAFTER environment with some basic preprocessing, some basic logging and a way to visualize the evaluation performance of the agent.

Other general resources and ideas you can try out:

- **Solutions to previous labs** containing clean implementations of the following:
  - DQN, Double-DQN
  - Reinforce, Actor-Critic, A2C, A2C-GAE

- **Exploration**: Simple idea such as temporally extended epsilon-greedy, approximate a Bayesian solution [11, 12], make a simple change to provide an intrinsic reward for exploring novel states [5, 14] or make a simple modification to the Replay Buffer [2] to direct the exploration to valuable states.
- **Distributional RL**: [4, 16, 6, 7]. For a clean implementation (although only `pytorch<=1.0`) look here.
- **Credit assignment**: [8, 13, 1, 3].

2.3 Recommendations

- Read the paper accompanying CRAFTER[9] carefully. Focus on the learning curves in the appendix to gain some intuitions about what you should expect when developing your agent.
- Carefully check the CRAFTER github repository, there are many goodies in there. You can adapt the scripts in the `analysis` folder to plot the various performance metrics of the agent at train time. Or you could use the available `/crafter/run_gui.py` script to play the game yourself.
- Start small. Implement an algorithm you already have some experience with, such as Deep Q-Networks (DQN), optimized with ADAM, and only after you convince yourself it's better than random start modding it.
- A well implemented DQN can be enhanced with only minor modifications to Double DQN [10] or Dueling DQN [15] but most likely you will see a large bump in performance by changing the loss function to the one in Categorical DQN [4] or any of the follow-up work in distributional RL: [16, 6, 7].Either of the follow-up works can be easier to implement than Categorical DQN. Another simple modification with possibly large impact is Munchausen-DQN [14].
- You will almost surely want to give n-step returns, or any other method that goes beyond TD(0), a try.
- During development use a smaller number of training steps, around $100 - 200$k so that you can gain some insights faster.
- Run at least two seeds in parallel when developing. RL is high variance and this will help you to cut through the noise. You can do this even on single-gpu systems, you can cram two or three DQN runs on a single gpu (provided the Replay Buffer is kept in system RAM, not on GPU RAM). For example you can parallelize using `bash` like so:
  `for i in $(seq 3); do python train.py --logdir logdir/random-agent/$i & done`.
- Feel free to try out-of-the box ideas such as:
  - Figure out if it's feasible to hard-code some sub-policies which are difficult to learn such as making a wood sword, and use it in conjunction to the learning algorithm.
  - Use the `/crafter/run_gui.py` to save episodes played by a human expert and figure out how to use them for training your agent. For example you could mix the content of the Replay Buffer with human trajectories and agent's experiences.
  - Feel free to hack the environment and provide a richer observation space than the pixels, as long as this doesn't completely defeats the purpose of the assignment.

REFERENCES

[1] Vyacheslav Alipov, Riley Simmons-Edler, Nikita Putintsev, Pavel Kalinin, and Dmitry Vetrov. Towards practical credit assignment for deep reinforcement learning. *CoRR*, abs/2106.04499, 2021.

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[3] Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards, 2018.

[4] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *ICML*, 2017.

[5] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. *CoRR*, abs/1810.12894, 2018.

[6] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. *ArXiv*, abs/1806.06923, 2018.

[7] Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *AAAI*, 2018.

[8] William Fedus, Carles Gelada, Yoshua Bengio, Marc G. Bellemare, and Hugo Larochelle. Hyperbolic discounting and learning over multiple horizons, 2019.

[9] Danijar Hafner. Benchmarking the spectrum of agent capabilities. *ArXiv*, abs/2109.06780, 2021.

[10] H. V. Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *ArXiv*, abs/1509.06461, 2016.

[11] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. *CoRR*, abs/1602.04621, 2016.

[12] Ian Osband, Benjamin Van Roy, Daniel J. Russo, and Zheng Wen. Deep exploration via randomized value functions. *J. Mach. Learn. Res.*, 20:124:1–124:62, 2019.

[13] David Raposo, Sam Ritter, Adam Santoro, Greg Wayne, Theophane Weber, Matt Botvinick, Hado van Hasselt, and Francis Song. Synthetic returns for long-term credit assignment, 2021.

[14] Nino Vieillard, Olivier Pietquin, and Matthieu Geist. Munchausen reinforcement learning. *CoRR*, abs/2007.14430, 2020.

[15] Ziyun Wang, Tom Schaul, Matteo Hessel, H. V. Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *ArXiv*, abs/1511.06581, 2016.

[16] Derek Yang, Li Zhao, Zichuan Lin, Tao Qin, Jiang Bian, and Tie-Yan Liu. Fully parameterized quantile function for distributional reinforcement learning. In *NeurIPS*, 2019.