UDACITY Logout

‹ Return to Classroom DISCUSS ON STUDENT HUB

Navigation

REVIEW | CODE REVIEW | HISTORY

Meets Specifications

Congratulations in meeting the specifications of the project! This project had discrete action space. Next, you will work with continuous action control space of the environment.
I hope that you're enjoying this nanodegree and excited to move ahead! Good luck!

Training Code

The repository (or zip file) includes functional, well-documented, and organized code for training the agent.

You can refactor your codes into Python files and run it as a program from the command line terminal. From my experience, jupyter notebooks can be slow! You can also then add hyper-parameters arguments by the use of the argparser in order to easily tune the values of the hyper-parameters through command line. Check out this tutorial on argparse https://www.youtube.com/watch?v=rnatu3xxVQE

Check out this blog https://www.pyimagesearch.com/2018/03/12/python-argparse-command-line-arguments/

The code is written in PyTorch and Python 3.

Pytorch is a dynamic computationally while tensorflow has been static. But recently tensorflow has been updated with eager execution making it easier to use i.e. without building computational graph to run the tensorflow sessions. All the deep learning frameworks are turning into the same page. However Pytorch is still Udacity's favorite because Pytorch performs better than the tensorflow eager execution for rapid prototyping as Pytorch is easier to read and understand than Tensorflow. Check out this article https://medium.com/@yaroslavvb/tensorflow-meets-pytorch-with-eager-mode-714cce161e6c :)

There's a Pytorch Udacity course that you may want to check out https://www.udacity.com/course/deep-learning-pytorch--ud188

The submission includes the saved model weights of the successful agent.

Saved model weights enable to run the agent during inference without re-training.

README

The GitHub (or zip file) submission includes a `README.md` file in the root of the repository.

👍

The README describes the the project environment details (i.e., the state and action spaces, and when the environment is considered solved).

You should state whether the action space is discrete (and not continuous) as it is important to choose the learning algorithm depending on it.

The README has instructions for installing dependencies or downloading needed files.

These steps will help anyone who is interested in running your project. It is also important to document all these steps if you want to re-run your codes again in the later time let's say after a year or so and may forget these steps if these are not documented.

The README describes how to run the code in the repository, to train the agent. For additional resources on creating READMEs or using Markdown, see here and here.

Along with writing the report, I'd also suggest you to write a blog post on this project like on Medium https://medium.com/ which is easier to write on. It will not take you much longer into your write-up and also include the tips given in this review. :D This is a great way to contribute to the AI community by sharing the knowledge.

Report

The submission includes a file in the root of the GitHub repository or zip file (one of `Report.md`, `Report.ipynb`, or `Report.pdf`) that provides a description of the implementation.

Nice work in describing the Q-learning algorithm, how these work and the architectures of DQN where experience replay and separate fixed Q-target network are also being used. Here's another udacity course lesson video where overview of Q-learning is given out. Q-table is also given out

https://www.youtube.com/watch?time_continue=94&v=WQgdnzzhSLM

The report clearly describes the learning algorithm, along with the chosen hyperparameters. It also describes the model architectures for any neural networks.

A plot of rewards per episode is included to illustrate that the agent is able to receive an average reward (over 100 episodes) of at least +13. The submission reports the number of episodes needed to solve the environment.

The reward curve is expected to be noisy. You can further average the rewards over the multiple runs like 100 or 50 to smooth out the fluctuations of the rewards over the episodes. You can also perform moving average technique to smooth out any fluctuations in the reward plot. You can read about it here https://www.dallasfed.org/research/basics/moving.aspx
Here's the python implementation of moving average smoothing https://stackoverflow.com/questions/13728392/moving-average-or-running-mean

The submission has concrete future ideas for improving the agent's performance.

Before implementing rainbow architecture, I'd suggest you to try out Double DQN and Dueling architecture.

You can extend the DQN to Double DQN in which the target network have to choose the greedy action given by local network. Check out the difference between DQN and Double DQN here https://datascience.stackexchange.com/questions/32246/q-learning-target-network-vs-double-dqn and also in the Udacity lesson given in the classroom. Here's the description of Double DQN https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/

Here's the nice description of benefit for using Dueling DQN https://www.quora.com/Why-does-a-dueling-network-for-deep-reinforcement-learning-work The advantage of the dueling architecture lies partly in its ability to learn the state-value function efficiently. With every update of the Q values in the dueling architecture, the value stream V is updated – this contrasts with the updates in a single-stream architecture where only the value for one of the actions is updated, the values for all other actions remain untouched. This more frequent updating of the value stream in our approach allocates more resources to V , and thus allows for better approximation of the state values.

DOWNLOAD PROJECT