UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR ROBOTICS
AND COGNITIVE SYSTEMS

Reinforcement Learning (SS2021)
Lecturer: Dr. Christoph Metzner
c.metzner@campus.tu-berlin.de
Teaching Assistant: Honghu Xue
xue@rob.uni-luebeck.de
Submission Deadline: 7. July
Submission Deadline for bonus tasks: 21. July

Reinforcement Learning - Assignment IV (21 pt + 15 pt)

Please fetch the python files for the programming part. The submission should be a zip file of PDF, related python files and a txt file showing the name of the team members in the group and student matriculation number. You do not need to repeat the question in the report. **The zip file is with the name of "RL_Assignment_index number_group leader name". The final PDF should not exceed 6 pages. Any delayed submission will not be counted for the score. Make sure you don't copy any figures or sentences from the books or from other groups, otherwise you won't get points for that. Please note**: For the bonus task, you submit it separately from the non-bonus task, and its deadline is July.21. Good luck!

In this assignment, we transit from discrete state space to a continuous one, whereas the action space still remains discrete. To meet these changes, we resort to function approximation approaches, where the value estimate for $V_\pi(s)$ or $q_\pi(s,a)$ are approximated by $\hat{V}(s,w)$ or $\hat{q}(s,a,w)$, and w are the weights to be learned. In the book of *RL, an introduction*, some conventional approaches are introduced, where the raw state or state-action pair are first transformed into some pre-defined feature space $x(s)$ or $x(s,a)$ and then multiplied by the weight vector w to get the approximated state value or state-action value. However, the approaches of feature encoding introduced in Chapter 9.5 in *RL, an introduction* can only work in low-dimensional state space, as the feature dimension increases rapidly with the raw input state dimension. As an alternative, one can use more expressive function approximators i.e. *Neural Networks* to directly map the raw states to the approximated V-value or q-value. With deep neural networks as function approximators, we move to the field of *Deep Reinforcement Learning*.

Task I: Theoretical understanding on Function Approximation approaches

1. (1) When using function approximation, what are the conditions for convergence? (Hint: Please consider the case of on/off-policy, tabular case/function approximation, bootstrapping/Monte-Carlo estimation) (2) Does the algorithm *Deep Q-Network* (DQN) has convergence guarantee? (2 points)

2. (1) How do *semi-gradient methods* differ from true *gradient descent* when updating the weights w in function approximation? (2) How does semi-gradient TD(0) algorithm differ from gradient Monte Carlo algorithm? Do both of them converge to the global optimum in linear function approximation? Also mention the specific term to which semi-gradient TD(0) algorithm converges. (3) What is the advantage of least squares TD over semi-gradient TD(0) algorithm? (3 points, from the book of RL)

3. (1) What is the benefit of replay buffer in *Deep Q-Network* (DQN)? (2) What is the purpose of the target network in DQN? Mention 2 options on the update of the target network in DQN. (2 points)

4. (1) Explain what is the maximization bias in DQN or q-learning? (2) How to mitigate this problem? Show the formulae for updating the Q-values in the mitigated method. (2 points)

Task II: Programming part on DQN and DDQN

From this assignment on, we need to use **Pytorch** for function approximation methods, a neural network package in Python. Please first check if your (Nvidia) graphics card supports any one of the Cuda verison in `https://pytorch.org/` or not. If yes, then you can install Pytorch locally. Pytorch versions newer than or equal to 1.5 are ok. (See history version in website and the corresponding Cuda version) The installation steps are as follows: (1) Download and install Cuda (2GB), which enables the large training acceleration by using GPU. (2) Copy the command of the correct version from the official website `https://pytorch.org/` to Anaconda Powershell to install Pytorch. Also see assignment I instruction. If your computer graphics card doesn't support the Cuda version that supports the Pytorch, please directly use Google Colab to finish all the rest assignments. Don't forget to set 'Hardware accelerator' to be 'GPU' at Colab's side. You need to copy the .py file content into the code blocks, but no need to install Pytorch on Google Colab as it has Pytorch installed by default.

1. Implement the 'classic' DQN on 'MountainCar-v0' with the version of no episodic length limit. Run the algorithm 3-4 times (one run per group member on average) and **plot the learning curves in 3 subplots**

Reinforcement Learning (SS2021)
Lecturer: Dr. Christoph Metzner
c.metzner@campus.tu-berlin.de
Teaching Assistant: Honghu Xue
xue@rob.uni-luebeck.de
Submission Deadline: 7. July
Submission Deadline for bonus tasks: 21. July

with the title of each subplot to be team member name who does the experiment. (The figure of the estimated q-value $\hat{q}(s_0, a^\star, w)$ and loss on Q-values are not needed in this task, but in Task III). The trained policy should reach the undiscounted episodic reward of around -150. (4 points)

2. We now move to a more interesting gym[box2d] environment 'LunarLander-v2'. Here, you need to implement the Double-DQN (DDQN) on 'LunarLander-v2'. Run the algorithm 3-4 times (one run per team member on average) and **plot the learning curves (loss not needed) in 5 subplots in the form of scatter plot with the title of each subplot to be team member name who does the experiment**. You can expand the code directly in the '.py' file above. Answer the following question: in practice, how are the two q-networks defined in DDQN. For Colab users, you currently can only see the progress of improvement. If your program finally converges to the episodic reward larger than 200, then the training is regarded successful. (Hint: **First test your DDQN algorithm on the easy task of MountainCar to check if it is correct implementation, then on LunarLander**). A successful training requires around 250000 frames, which roughly takes 18 minutes on Colab. (3 points)

Task III: Going Deeper

Here, we go deeper into task II from the the observation of DQN on 'Mountaincar-v0'. For this task, you need to run the algorithm in task II, subtask 1 sufficiently long until the q-loss approaches 0, more than 150000 steps.

1. If you look into the trend of estimated Q-value of the initial state s_0 in the programming task of *DQN* in 'Mountaincar-v0', (1) What can be the lowest possible q-value for $\hat{q}(s_0, a^\star, w)$ in the case of $\gamma = 0.99$? (2) Why or in which case can $\hat{q}(s_0, a^\star, w)$ approach this lowest possible value in DQN/DDQN? (3) Does the final converged value of $\hat{q}(s_0, a^\star, w)$ corresponds roughly to its optimal value, reason that with mathematical computation. (3 points)

2. Again in the task of *DQN* on 'Mountaincar-v0', (1) Provide the figure showing the q-loss function and estimated q-value of the initial state. In this task, one figure per group is sufficient. (2) Explain why q-loss approaches zero at the early phase, and then experience a sudden increase? Mention which kind of sample/experience leads to the first occurence of high q-loss? (2 points)

Bonus Tasks (15 pt)

We now extend the algorithm DQN to raw-pixel image input, which is definitely of higher state dimension than the above tasks. In these tasks, you will train an agent to play some video games. Enjoy! (You could add the code to save the trained models, which is not currently in the code.) Note: you need computer with good GPUs. (recommended: Nvidia : 970, 1060, 1080...), Otherwise, you definitely need to run in Colab. **Your memory buffer should store less than** 0.3 **million experience as Colab has limited RAM.** For local users, instructions on pytorch installation with cuda acceleration is in the Assignment 1.pdf. The folder 'common' needs to be placed in the same path as other '.py' files in case of local run.

1. (i) Implement DDQN on one OpenAI Atari Game 'PongNoFrameskip-v4' (with image input), show the learning curve of each run. For this task, one run per team member. **You will also need to install the package 'opencv-python' using 'pip install' in Anaconda Powershell if you run locally.** Be patient with training, since around 500K of frames are needed to see improvements. For debugging purposes: Around 180K frames, there should be an initial but persistent increase in the averaged return over past 100 episodes, although its value is still around -20. Please read the paper in reference list. **Hint: Training can be long, be patient, (might be more than** $12hrs$ **depending on your computer performance), please run it on a good GPU computer or use Colab. In case of Colab, upload 'Atari_on_Colab'.ipynb file, and read the instructions in that file carefully. If running locally, you could hibernate your computer which can continue your program.** For Colab, it roughly

takes 7hrs, and the testing performance will reach 20 at around 3M steps. You can terminate the run as long as the testing phase converges to the performance of 20 or even better, don't forget to save your statistics and show the plot. (5 points)
(ii) Answer the following questions: (a) What does "EpisodicLifeEnv" do in the training phase? Assume you have 3 lives, and you lose one life and perform env.reset(), does the newly-reset environment start with 2 lives or 3 lives in the training phase? (For this, you might need to try other games in Atari ,e.g., Seaquest) (b) How is the immediate reward processed in Atari game? (c) What does "NoopRestEnv" do and what is its purpose? (d) What does "MaxAndSkipEnv" do? (3 points)

2. Another improvement on DQN/DDQN can be using *prioritized experience replay buffer* (PER), where the experience is not uniformly sampled, but in a prioritized manner. Roughy speaking, each sample is with its own sampled probability proportional to its fitted temporal-difference error. Implement DDQN with PER on the OpenAI Atari Game 'PongNoFrameskip-v4' (with image input), show the learning curve. **For this task, one plot/run per team is sufficient.** Please read the second paper and the recommended blog in reference list. **Hint: Training can be long (might be more than** 12hrs**), please run it on a good GPU computer or use Colab.** For Colab, it roughly takes 7hrs, and the testing performance will reach 20 at around 3M steps. (7 points)

Literature

The following reference material is relevant for this assignment. Each reference have some **overlapping content** with the other, so you don't need to go through all of the reference material below, just choose wisely.

- Sergey Levine's online *Deep Reinforcement Learning* lecture 7, 8 (highly-recommended, you can skip the part of Q-learning with continuous actions in lec 8 and 'actor-critic' part which you will learn in Assignment knew in lec 7)
- **Reinforcement Learning - an introduction, second edition** Chapter 9.1-9.4, Chapter 11.1-11.3, Chapter 6.7 (maximization bias)
- Parametric ReLU (PReLU, optional)
`https://medium.com/@shoray.goel/prelu-activation-e294bb21fefa`
- Pytorch basic operations (for programming part)
`https://jhui.github.io/2018/02/09/PyTorch-Basic-operations/`
- Related Paper (only for bonus task)
(i) The paper of DQN on Atari Games: `https://arxiv.org/pdf/1312.5602.pdf`
(ii) The paper of *prioritized experience replay buffer*: `https://arxiv.org/pdf/1511.05952.pdf`
(iii) A good introduction on *prioritized experience replay buffer*: `https://danieltakeshi.github.io/2019/07/14/per/` (**Note: in this blog, there is an error in the statement** "and then further scaled in each minibatch so that $max_i w_i = 1$ for stability reasons", where the $max_i w_i$ isn't necessarily 1 for each batch, i.e.,$max_i w_i$ refers to the max weight in replay buffer, not the one in the current batch.)
(iv) Explanation on Sum Tree: `https://www.fcodelabs.com/2019/03/18/Sum-Tree-Introduction/`
- Explanation for Gym wrappers on Atari Games (only for bonus task) The post-processing of the game env for training: `https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/`
Some explanation on Gym wrappers: `https://books.google.de/books?id=BAlnDwAAQBAJ&pg=PA124&lpg=PA124&dq=NoopResetEnv(gym.Wrapper)&source=bl&ots=zgSwjdNN5E&sig=ACfU3U2QdEnLh3Jd5-Z`

cXUYNW-iccREuGQ&hl=de&sa=X&ved=2ahUKEwia68rN2ILmAhWQy6QKHReODwsQ6AEwAnoECAgQAQ#v=onepage&q=NoopResetEnv(gym.Wrapper)&f=false