RWTH Aachen University
Institute for Data Science in Mechanical Engineering

Document Version 1.0.0, released 2021-01-29

Bonus Point Assignment II

1 Introduction

This document gives all necessary information concerning the second bonus point assignment for the class CSME II in the winter semester 20/21. This assignment consists of two Jupyter notebooks. In the first (Part A), you will implement Q-table and policy gradient learning. The second Jupyter notebook (Part B) treats deep Q-networks. OpenAI Gym [1] offers a clean environment interface and implementations of example environments often used in reinforcement learning. In this assignment, you will use two of them. As this is an assignment for bonus points, some questions may extend the scope of the lecture.

Formalia With this voluntary assignment, you can earn up to 5% of bonus points for the final exam. The bonus points will only be applied when passing the exam. A failing grade cannot be improved with bonus points. This assignment starts on 29.01.2021 and you have until 26.02.2021 to complete the tasks. The assignment has to be handed in in groups of 3 to 4 students. All group members have to register in the same group on Moodle under *Groups Assignment 2*.
Please hand in your solution as a single zip file including both notebooks as well as the `solution` folder, which contains automatically generated export files. Do not clear the output of your submitted notebooks. Your solution has to be handed in via Moodle before the deadline.

Grading Your implementations will mostly be checked automatically via unit tests, so ensure that your notebooks run correctly from start to finish. Throughout the notebooks, there are short unit tests for many tasks that you can use to validate your implementation. However, note that these are not the unit tests that will be used to grade your assignment.

Jupyter To run the Jupyter Notebooks for this assignment, we recommend installing Jupyter locally, to be able to render the environment. However, you can also use the RWTH JupyterHub again. For this, search for the `CSME2` profile and start the server. This profile was specifically created for this class and automatically comes with PyTorch and the other required dependencies. Once your server is started up, you can upload the notebooks and datasets and start completing the tasks.
Alternatively, you can set up your environment locally. However, please note that we cannot give support in case your environment does not work. First, you should install Anaconda or Miniconda on your system. Then install the required dependencies by running

```
conda install -y pandas seaborn scikit-learn gym pytorch torchvision cpuonly -c pytorch
```

in your **Anaconda prompt**. Next, you can launch Jupyter via the Anaconda GUI or, if you are using miniconda, via executing `jupyter notebook` in a terminal or the Anaconda prompt. When using miniconda, you additionally may have to install Jupyter via `conda install -y jupyter`.

2 Assignment Part A

Question A1 - Q-Tables

The Frozen Lake environment is a discrete 4×4 grid with four different kinds of grid cells:

- S: starting point (safe)
- F: frozen surface (safe)
- H: hole (not safe)
- G: goal

The objective in this environment is to reach the goal from the starting point while avoiding unsafe places. The agent moves around the grid until it reaches the goal (G) or a hole (H). The agent receives a reward of 1 if it reaches the goal and 0 otherwise.

a) Create an instance of the FrozenLake-v0 environment and show the initial state. The current position of the agent is indicated by a red background. Thus, the initial state should look like the following:

S F F F
F H F H
F F F H
H F F G

b) The agent in the Frozen Lake environment has four possible moves: Up, Down, Left and Right. However, the surface of the frozen lake is slippery, so the agent does not always end up in the cell it intended to go to. Find one action sequence that leads to the goal. For this task, the random seed of the environment is fixed, so running the same sequence twice will lead to the same states. Your action sequence does not need to be optimal, it just has to reach the goal. Use `env.step(direction)` method to perform actions.

c) Use a Q-table to learn Q-values to navigate through this environment. The Q-values are stored in the Q-table and are updated during the training process.

Complete the following steps:

- Initialize the Q-table. In this table, rows represent the states and the columns the expected future reward for choosing the corresponding action in that state. There is one state for each cell, denoting the location of the agent.

  Choose sensible default values.

- Implement action sampling. Find a trade-off between exploration (choosing random actions) and exploitation (choosing the action with the highest expected reward).

- Implement the Q-table update. Use the update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big),$$

  where α is the learn rate, and γ is the discount factor for future returns.

You can use the `# Plot Q-table` cell to get insights into your learned Q-table.

When evaluated greedily, your learned Q-table should be able to score an average reward of at least 0.5. To achieve this, you may tweak hyperparameters but do not increase the number of episodes.

Question A2 - Policy Gradients

In this exercise you will program a model-free reinforcement learning algorithm: policy gradient.
Rendering the environment is not supported on the JupytherHub. This means you can still use the JupyterHub to develop your code but then will not be able to render the environment. If you would like to see your environment, run the notebook locally on your computer.
The CartPole environment [2] consists of a pole that is attached by an un-actuated joint to a cart that moves along a frictionless track. The pendulum starts upright. The goal is to keep the pole upright and prevent it from falling over by increasing or reducing the cart's velocity. The system is controlled by applying a force of $+1$ or -1 to the cart. A reward of $+1$ is provided for every timestep that the pole remains upright, including the termination step. Observations in CartPole-v1 lie in a continuous space, defined by four dimensions:

Num	Observation	Min	Max
0	Cart Position (m)	-4.8	4.8
1	Cart Velocity ($\mathrm{m\,s^{-1}}$)	$-\infty$	∞
2	Pole Angle (rad)	-0.418 (-24 deg)	0.418 (24 deg)
3	Pole Velocity at tip ($\mathrm{s^{-1}}$)	$-\infty$	∞

Figure 1: Two states from the CartPoleV1 environment. Left: Pole is in an upright position. Right: The angle of the pole to the vertical got too large and the episode is considered to be failed.

At each timestep, one of two discrete action can be taken:

Num	Action
0	Push cart to the left
1	Push cart to the right

Note: The amount the velocity is reduced or increased is not fixed. It depends on the angle the pole is pointing.
The episode ends when

- the pole is more than 12 degrees from vertical, or
- the cart position is more than 2.4 (center of the cart reaches the edge of or the display), or
- episode length is greater than 500.

Thus, the maximal accumulated reward for one episode is 500.

a) The policy network $\pi(a|s) = P(A_t = a|S_t = s)$ maps states to action probabilities. Therefore the size of the input should match the size of the state space, and the size of the output should match the size of the action space.

Complete the implementations of the `__init__` and `forward` methods. You may choose activation, drop out, batch normalization, weight regularization, as it seems sensible to you.

Note: For this task, a small shallow network is sufficient (e.g., one hidden layer with 128 neurons).

b) The policy returns an action distribution for each given state. Implement the `sample_action` method, which samples one action from this distribution. `sample_action` returns both the action and the log-likelihood of choosing that action given the distribution.

Hint: The log-likelihood is important for the training later on, thus gradients must be able to flow through (i.e. don't use `.detach()`). You may use `torch.distributions.Categorical`.

c) Approximate the return of each state by computing the return over each trajectory.

$$R_t = r_t + \gamma R_{t+1}$$

Compute the expected return based on a series of observed rewards. Use `gamma` to discount future rewards.

To prevent large differences in the magnitude of the return, apply standardization to the expected returns (i.e., bring the mean to zero and the standard deviation to one). Otherwise, a longer trajectory will have a larger return, which can be detrimental to the training of the neural network.

d) Complete the implementation of the training loop.

- Get the action distribution for the current state from the policy. Use the `sample_action` method to sample an action from it and give the corresponding log-likelihood.
- Compute the policy loss `policy_loss`. After each episode, the policy π is trained to minimize

$$\mathcal{L} = \sum_t -\log \pi(a_t|s_t) R_t.$$

After training, your policy should be able to achieve an average reward of at least 487.5 over 100 episode. You may tune hyperparameters and adjust your network architecture. However, do not increase the number of training episodes (though you may lower it, if you're able to successfully train with less).

Hint: If you want to further improve your policy, you can try to train only with unsuccessful episodes.

3 Assignment Part B

Question B1 - Replay Buffer

a) Deep Q-learning commonly uses replay buffers to reuse samples of old episodes and to break high correlation between similar trajectories and neighboring data points. This replay buffer will store $(s_t, a_t, r_t, s_{t+1}, d_t)$-tuples, denoting a transition from state s_t to s_{t+1} via the action a_t while observing the reward r_t. d_t is a binary flag indicating whether the episode terminated with that transition.

Implement the following methods:

`add`: Adds one transition to the buffer and replaces the oldest transition in memory. Compute the index where you want to store the new transition and store the data of the transition at the appropriate position. Remember to increase the `mem_cntr` after adding a transition.

Hint: Use modulo-division to compute the index.

`sample_batch`: Samples one batch from the memory with the given `batch_size`. Return numpy arrays for `states`, `actions`, `rewards`, `next_states` and `is_terminal`.

Hint: You may use `np.random.choice` to choose random indices for the batch.

b) The buffer implemented in a) is initially empty. Fill the replay buffer with transitions sampled from the environment using random actions. Use a loop to generate and add transitions to the buffer until `buffer.is_filled()`. Reset the environment before each episode.

A transition is considered terminal if the pole fell, i.e. `done and env._elapsed_steps < 500`. Otherwise, reaching the time limit of 500 steps would incur a zero return later.

Random actions can be sampled via `env.action_space.sample()`.

Question B2 - Deep Q-Network

a) Create a network architecture for the Q-network. The Q-network models the function

$$s \rightarrow [Q(s,a) \quad \forall a \in \mathcal{A}]^T$$

where k is the size of the action space. Like before, choose a simple architecture with just a few layers. If required, you can always add layers later.

Hint: You can start with a similar architecture as in Question A2.

b) Q-values are not an action distribution from which we could sample. Instead, we will use an ϵ-greedy strategy for action sampling to introduce randomness into the actions. ϵ-greedy chooses a random action with probability ϵ and uses the optimal action otherwise.

$$\pi_\epsilon(s) = \begin{cases} \text{random action } a & \text{with probability } \epsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \end{cases}$$

Implement the `epsilon_greedy` function.

c) Implement the loss function

$$\mathcal{L} = \sum_{(s_t, a_t, r_t, s_{t+1}) \in \Delta} \left(\underbrace{Q(s_t, a_t)}_{\text{Q-value}} - \left(\underbrace{r_t + \gamma \hat{Q}(s_{t+1}, a_{t+1})}_{\text{expected Q-value}} \right) \right)^2$$

where Δ is a batch of transition samples and $Q(s_t, a_t)$ are target Q-values in `q_network` and $\hat{Q}(s_{t+1}, a_{t+1})$ in `target_network`. Essentially, the loss is the difference of the current Q-value and the expected Q-value.

- Compute the Q-values $Q(s_t, a_t)$ for the current state and action in the `q_network`.
- Compute the the best actions

  $$a_{t+1} = \arg\max_a Q(s_{t+1}, a)$$

  under the `q_network` for the next states s_{t+1}.
- Compute the target Q-values $\hat{Q}(s_{t+1}, a_{t+1})$ in `target_network` for the next state and the corresponding best action.
- Zero-out all target Q-values for terminal states
- Compute the expected target Q-values

d) This training loop performs an update step on the Q-network every 4 simulation steps. Each update step samples a batch of transitions from the buffer, computes the loss and performs an update on the Q-net using the optimizer.

Every 2000 steps, the target Q-net and the current Q-net should be synchronized. Implement the synchronization step. In this, `target_network` is updated with the current weights of `q_network`. Make sure they don't accidentally share weights.

Finally, run the training. After training, your policy should be able to achieve an average reward of at least 487.5 over 100 episode. You may tune hyperparameters and adjust your network architecture. However, do not increase the number of training episodes (though you may lower it, if you're able to successfully train with less).

Question B3 - Evaluation

In this assignment, you have seen different approaches to solve the same problem. Which method did work best for you? What changes in network architecture had the most impact?

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[2] A. Barto, R. Sutton, and C. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, pp. 834–846, 1983.