02465 Project: Part 4

Tue Herlau
tuhe@dtu.dk

April 2, 2026

Formalities

- The deadline for this report is **Monday 11th May, 2026** before 23:59.
- Submission of reports happen on DTU learn
- You can work in groups of 1, 2 or 3 students (but not 4)
- Collaboration policy: It is not allowed to collaborate with other groups on this project, except for discussing the text of the project with teachers and students enrolled on the course this semester. It is not allowed to communicate (or make available) solutions or parts of solutions to the project to other people. It is not allowed to use solutions from previous years, or solutions found on the internet or elsewhere.
- You can freely use code from the *exercises* when you solve the project, for instance the dynamical programming algorithm.
- Your overall evaluation will be based on your written answers and your UNITGRADE score. They will be weighted based on an assessment of the required work.

Preparing the hand-in:

Hand in these three files (please do not hand in a `.zip` file as this confuses DTU learn):

A `.tex` file with your written answers: Prepared this by modifying the template in `irlc/project4/Latex/02465project4_handin.tex` . Simply write your answers where it says **YOUR SOLUTION HERE**. I recommend keeping the layout as it is.

A `.pdf` file corresponding to this `.tex` file

A `.token` file containing your python-solutions: Generate this file by running the script `irlc/project4/project4_grade.py` . It is very important you do not modify this file.

In order to hand in the assignment on DTU Learn you must be part of a group. You can join a group on DTU learn from *My Course* → *Groups*.

Contribution table

DTUs exam rules require that each students contribution to the report is clearly specified. Therefore, for each element in the report, specify which student was responsible for it in the table in the template. **A report must contain this documentation to be accepted**. The responsibility assignment must be individualized. This means:

- For reports made by 3 students: Each section must have a student who is 40% or more responsible.
- For reports made by 2 students: Each section must have a student who is 60% or more responsible.

This is an external requirement. Ask me if you have any questions.

Code hand-in:

- Please keep the structure of the `irlc`-folder. All of your code which is specific to this report should be in the `irlc/project4/` directory. Solutions which use code outside the `irlc` folder cannot be verified and therefore cannot be evaluated.
- If you wish to use additional third-party libraries please discuss them with me first to ensure you are on the right track.
- Breaking or tampering with the UNITGRADE framework, for instance by reporting a false number of points or making your solution unverifiable, is potentially cheating. Code which is obfuscated to the point of being unreadable cannot be evaluated.
- This is not a programming course: Strange, long, undocumented, or downright disturbing solutions will be evaluated simply based on whether they work or not.

1 Finding the rebels using UCB-exploration (`rebels.py`)

Things have been tense after the rebels blew up the boss death-star, and finding them has become a top priority. This is done using search-droids, who must explore the various star systems the rebels may be hiding in. Practically, a search droid moves around on a grid, and gets a positive reward depending on how many rebels it finds, or a negative reward for flying into a dangerous zone and being destroyed. The problem, therefore, resembles the standard grid-world problem. You are particularly interested in the two scenarios show in the top row of fig. 1 (*basic* and *bridge*).

The existing exploration-approach (random ϵ-greedy; i.e. what the current Q-learning method does) was designed by the same person who made the curriculum at the marksmanship school for storm troopers, and your job is to see if you can do better – specifically, you want to use an UCB-inspired strategy.

To elaborate on this, it is useful to first refresh how regular Q-learning uses the regular ϵ-greedy bandit exploration method (see [SB18, Equation (2.5)]) when making decisions as explained by [SB18]:

- You have as many bandit-problems as there are states

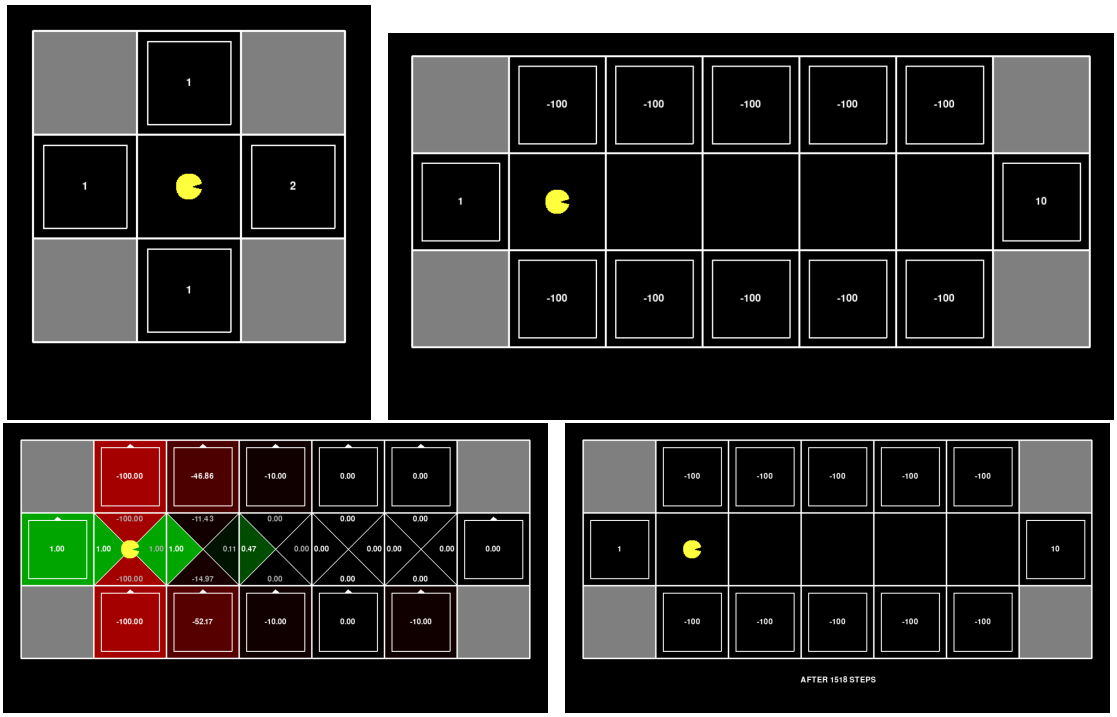

Figure 1: Top row: The basic and bridge rebel-locating scenario. Check the `rebels_demo.py` file to see how they are plotted and try to play in them yourself. Bottom row: The Q-values as obtained using a Q-learning agent for 3000 episodes (!) and an UCB-exploration agent for just 300 episodes.

- The actions for the bandit-problem in state s is $\mathcal{A}(s)$
- When the bandit, in state s, takes an action $a \in \mathcal{A}(s)$, the bandit-algorithm obtains a reward:

$$\tilde{R}_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'). \tag{1}$$

- The Q-learning agent then learns by updating the Q-values using this reward:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(\tilde{R}_t - Q(s,a))$$

  just as in the regular bandit-learning algorithm [SB18, Equation (2.5)].

- Finally, the bandit in state s implements the ϵ-greedy rule: i.e. select the action with the highest Q-value with probability $1-\epsilon$ and otherwise a random action in $\mathcal{A}(s)$

To *use UCB for exploration* simply modify the above as follows:

- You have as many UCB-bandit-problems as there are states
- The actions for the bandit-problem in state s is $\mathcal{A}(s)$
- When the bandit, in state s, takes an action $a \in \mathcal{A}(s)$, the UCB-bandit-algorithm once again obtains a reward of $\tilde{R}_t$ (see eq. (1)).

- The bandit (specific for state s) is trained and selects actions using the UCB-algorithm (see [SB18, Eq. (2.10)]) using the reward $\tilde{R}_t$.
- In this equation, note that Q_t has the same meaning as before (and should therefore be updated the same way), but you need to define N_t and t to be specific for this state. I.e., keep track of how many times we have seen the state t, and how many times we have played a given action $N_t(a)$ in the state.
- The UCB-algorithm in [SB18] has the following corner-cases:
  - If two actions have the same priority (i.e., same value of the UCB-bound according to what is inside the $\arg\max_a$ in [SB18, Eq. (2.10)], we prefer the action with the *lowest* index (i.e. we prefer $a = 1$ over $a = 3$ etc.)
  - If an actions have not been tried before, so that $N(a) = 0$, it has infinitely high priority and will always be selected. I.e. the method will first try all actions once in order: $a = 0, 1, 2, \ldots$.

In the first of the two scenarios shown in fig. 1 (basic), the agent can take one of four actions in the starting state, and from there it must take a single extra action to (deterministically) transition to the terminal state with the reward shown as the numbers in the figure. Since there is one relevant state (the starting state), the problem will closely resemble the standard UCB bandit problem. One slight annoyance is that the last (dummy) action in an episode have no effect in the gridworld-environment and we will therefore discard it. You can find an example (see the `rebels_demo.py`)-file which illustrates how you can generate actions using an agent and discard the last (dummy) action:

```
Trajectory 0: States traversed [(1, 1), (1, 2), 'Terminal state'] actions taken [0, 0]
Trajectory 1: States traversed [(1, 1), (0, 1), 'Terminal state'] actions taken [3, 0]
All actions taken in 16 episodes, excluding the terminal (dummy) action [[0], [3], [0], [0], [0], [np.int64(3)], [0], [0], [0], [0], [0],
↪  [0], [0], [0], [0], [0]]
```

The file also contains code for visualizations, and how to turn on keyboard-inputs so you can play with your UCB-agent.

Problem 1 *UCB-based exploration*

Implement the function `get_ucb_actions`. The function should accept a gridworld layout, the parameter α controlling the learning rate in the Q-learning algorithm, the exploration parameter c (the constant which control exploration in the UCB algorithm), and a number of episodes to simulate. You can assume that $\gamma = 1$ in all experiments.

The function should return a list simply consisting of all actions the agent takes but *excluding* the last dummy action. i.e., simply remove the last action in each episode and concatenate the action-sequences together. Example code for doing this is included in the `rebels_demo.py` file. The return value should thus be a list of integers, and in case of the basic grid, it should have the same length as the number of episodes.

Info:

Overall hints:

- The Gridworld-environment does not have the same number of actions available in each state. In fig. 1 (top, left), the starting state will have $|\mathcal{A}(s)| = 4$ actions, but the exit states (those with an white inner border and a number) will have $|\mathcal{A}(s)| = 1$ action. We deal with varying action spaces in the manner prescribed by openai Gym; c.f. the code for the Q-learning agent or the online documentation for part 3 of the project.
- Since the method will be based on Q-learning, the code for the Q-learning agent is probably a good place to start. Note in particular the Q-values are updated the same in the UCB and ϵ-greedy method.
- You can turn the grid layout (a list of lists of strings) into a gym `Env` using the methods from the toolbox. See the `rebels_demo.py` file for more details.
- If you write an agent, you can use the `train`-function to get a list of trajectories, and you can simply concatenate them together to get the list of all actions. See the `rebels_demo.py`-script for details.
- Note that to implement the UCB-algorithm, you must keep track of how many times each state has been visited, and how many times an action has been taken in that state. In other words, you must keep track of the N-values (see [SB18, Eq. (2.10)]). Note this data structure closely resemble Q-values.
- Again, if you look at [SB18, Eq. (2.10)], the t-parameter must actually be equal to: $t = \sum_a N_t(a)$. This can potentially reduce what you need to keep track of.
- Remember that in the basic-environment, with a single state, there is a single UCB-bandit problem and it will therefore be quite simple to reason about/keep track of.

Implementation and self-check hints: UCB is a simple algorithm and you can reason about what the method will do. Two cases are particularly interesting:

- The first four episodes will compute the same action sequence regardless of the rewards. Figure out why and check your result.
- Actually, the first 8 episodes will always be the same independent of the rewards. To explain this, you should consider eq. (1), and that each episode in the basic-environment consists of two actions and three states.
- From here, you can manually run the UCB algorithm and figure out the actions taken.

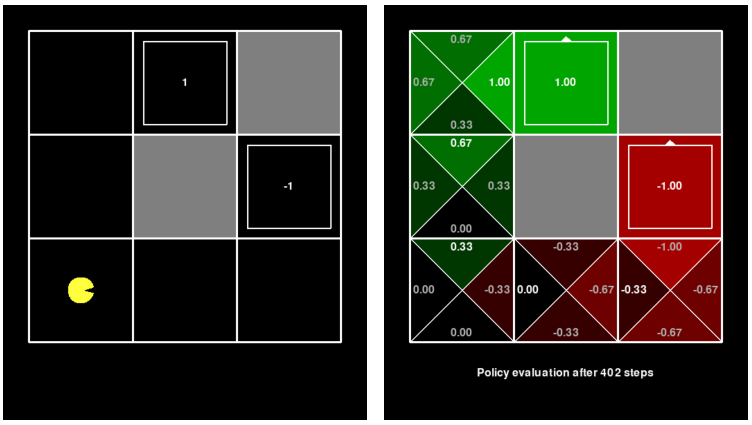

Figure 2: Left: The gridworld map considered in this exercise. Right: The state-action function $Q_\pi(s, a)$) for the random policy computed using policy evaluation.

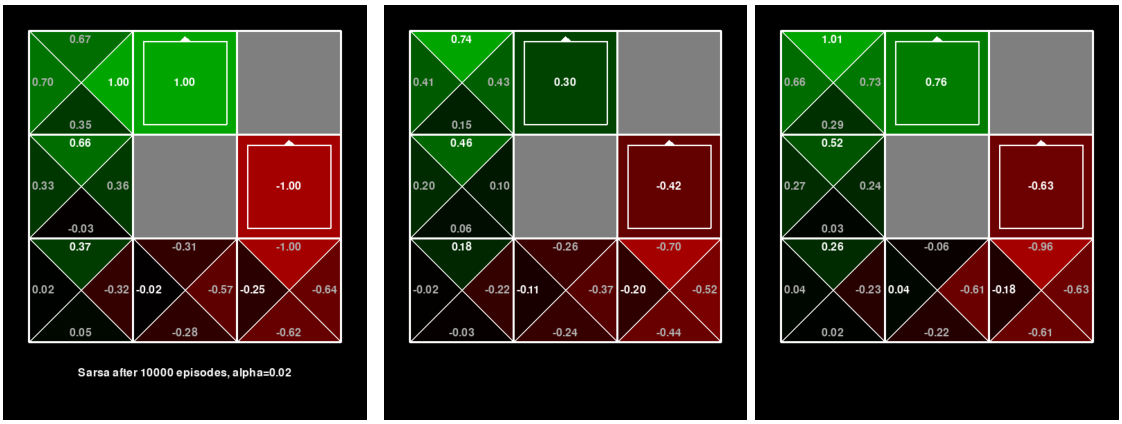

Figure 3: Left: The gridworld map considered in this exercise. Right: The state-action function $Q_\pi(s, a)$) for the random policy computed using policy evaluation.

2 Computing State-action values using random probe-droids (`probe_droids.py`)

In this problem, you a dispatching a large amount of probe-droids in a meteor field shown in fig. 2 (left). Probe droids fly around randomly[1], and by flying a large number n of probe-droids we can approximate the state-action function $Q_\pi(s, a)$ and use it for nefarious purposes.

For reference, fig. 2 (right) shows the (true) action-value function computed using policy evaluation ($\gamma = 1$), however, we have to compute it using samples.

One way to do that is to use standard tabular Sarsa learning, and simulate the random policy by setting $\epsilon = 1$ (thereby guaranteeing random actions). The result of this, using $n = 20000$ episodes and a learning rate of 0.02, can be seen in (left). As can be seen, it fairly closely matches policy evaluation but with some randomness due to the learning rate/finite number of episodes.

[1] i.e., their policy π selects among all available actions with equal probability

This exercise: Your task is to estimate $Q_\pi(s,a)$, for the random policy, using two different linear function approximators. This is equivalent to evaluating Sarsa with a (suitable) linear function approximator using $\epsilon = 1$.

We construct the linear function approximator as follows: Suppose the gridworld has a height of $H = 3$ tiles and a width of $W = 3$ tiles.

A state has at most $\mathcal{A}(s) = \{0, 1, 2, 3\}$ actions. Suppose we let $\boldsymbol{x}(s)$ be any function approximator depending on a state s, and let $l = |\boldsymbol{x}(s)|$ be the dimension.

W can then define the full linear function approixmator as a $4l$ long vector $\boldsymbol{x}(s,a)$ defined as:

$$\boldsymbol{x}(s,a=0) = \begin{bmatrix} \boldsymbol{x}(s) \\ \mathbf{0}_{3l} \end{bmatrix}, \boldsymbol{x}(s,a=1) = \begin{bmatrix} \mathbf{0}_{l} \\ \boldsymbol{x}(s) \\ \mathbf{0}_{2l} \end{bmatrix}, \boldsymbol{x}(s,a=2) = \begin{bmatrix} \mathbf{0}_{2l} \\ \boldsymbol{x}(s) \\ \mathbf{0}_{l} \end{bmatrix}, \boldsymbol{x}(s,a=3) = \begin{bmatrix} \mathbf{0}_{3l} \\ \boldsymbol{x}(s) \end{bmatrix},$$

where e.g. $\mathbf{0}_{3l}$ is a vector of $3l$ zeros.

Thus, all that remains is to define $\boldsymbol{x}(s)$.

Testing and evaluation After you have implemented the feature approximator, your task is to evaluate select values of $Q(s,a)$ for specific state/action combinations. For example, the following code shows how this can be done using the Sarsa agent:

```python
# probe_droids.py
agent = SarsaAgent(env, alpha=0.02, gamma=1., epsilon=1)
train(env, agent, num_episodes=10000, verbose=False)

states_and_actions = [((0, 0), 0),
                      ((0, 0), 1),
                      ((2, 0), 3),
                      ((0, 2), 1),
                      ]
q_values = {(s, a): agent.Q[s, a] for s, a in states_and_actions}
for (s, a), q in q_values:
    print(f"In state {s=} and action {a=} we have Q(s,a) = {q}")
```

The linear function approximator For the first feature map, we use the normalized grid coordinates of the state. Let $s = (i, j)$ denote the row and column of the agent, and define

$$\boldsymbol{z}_{\text{lin}}(s) = \begin{bmatrix} 1 \\ u \\ v \end{bmatrix}, \qquad u = \frac{i}{H-1} \text{ and } v = \frac{j}{W-1}. \tag{2}$$

Thus, the state feature dimension is $l = 3$.

Problem 2 *Linear function approximator*

Implement the function `linear_experiment`. The function should accept a number of episodes, the learning rate α, a gridworld layout, and a sequence of states and actions. It should then use the linear function approximator described in eq. (2) together with the action-dependent construction above, and return a dictionary of q-values in the previously described format.

Info:

- One approach is to use the `LinearSemiGradSarsa`, using $\epsilon = 1$, but specify a different feature encoder
- you can find more about feature encoders here: `https://www2.compute.dtu.dk/courses/02465/exercises/ex11.html`
- You can get the H and W values using `env.mdp.height` and `env.mdp.width`.

The quadratic function approximator For the second feature map, we augment the normalized coordinates with quadratic terms and an interaction term. Using the same notation as above, define

$$\boldsymbol{z}_{\text{quad}}(s) = \begin{bmatrix} 1 \\ u \\ v \\ u^2 \\ v^2 \\ uv \end{bmatrix}, \qquad u = \frac{i}{H-1} \text{ and } v = \frac{j}{W-1}. \tag{3}$$

In this case, the state feature dimension is $l = 6$.

Problem 3 *Quadratic function approximator*

Implement the function `quadratic_experiment`. The function should accept a number of episodes, the learning rate α, a gridworld layout, and a sequence of states and actions. It should then use the quadratic function approximator described in eq. (3) together with the action-dependent construction above, and return a dictionary of q-values in the previously described format.

References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (Freely available online).