Project Machine Learning
Task 3: Reinforcement Learning

Philipp Liznerski[1], Saurabh Varshneya[2], and Tobias Michels[3]

[1]liznerski@cs.uni-kl.de, `@liznerski`
[2]varshneya@cs.uni-kl.de, `@varshneya`
[3]t_michels15@cs.uni-kl.de, `@tmichels`

January 4, 2022
Deadline: 07/02/2022

Introduction

Usually, this section would contain a short introduction to *Reinforcement Learning* (RL) in the style of sheets 1 and 2. However, this time we want you to write such an introduction and include it in your report. You can start, for example, with Sutton and Barto [3, Chapter 3] and you should cite any additional resources that you might end up using, but your text should nevertheless explain the topic in your own words (see task 1 a)).

1 Tabular Reinforcement Learning

Gridworld The *Gridworld* is one of the re-occurring examples in Sutton and Barto [3]. Essentially, it takes a top-down view on a world that is made up of grid cells, where each cell can specify a different behavior. The goal is always the same: An agent starts at the designated starting position and must navigate the world to find and reach the designated goal cell. Here, navigating means that the agent can choose to either move left, right, up or down. However, there are different approaches to fit this goal into a reward function as we will explore in one of the upcoming tasks.

Figure 1 shows an example of such a Gridworld. The template code for this task contains code that defines the dynamics of this specific world. The different types of cells are:

Blank cell:	Does nothing special and behaves as you would expect.
S:	Starting point where the agent spawns at the beginning of each episode.
G:	Goal state. If the agent reaches this, the episode ends.
Swamp:	If the agent moves onto a swamp cell, it will get stuck with a probability of 0.5, meaning that the action is not executed.
Arrows:	The agent is moved in the direction of the arrow. Any action that the agent wants to execute is ignored.
Pit:	If the agent moves onto a pit cell, it receives a reward of -1000 and the episode ends.
Wall:	The agent can't move onto a wall cell. Quite self-explanatory.

Unless otherwise specified, the agent receives a reward of -1 on each time-step.

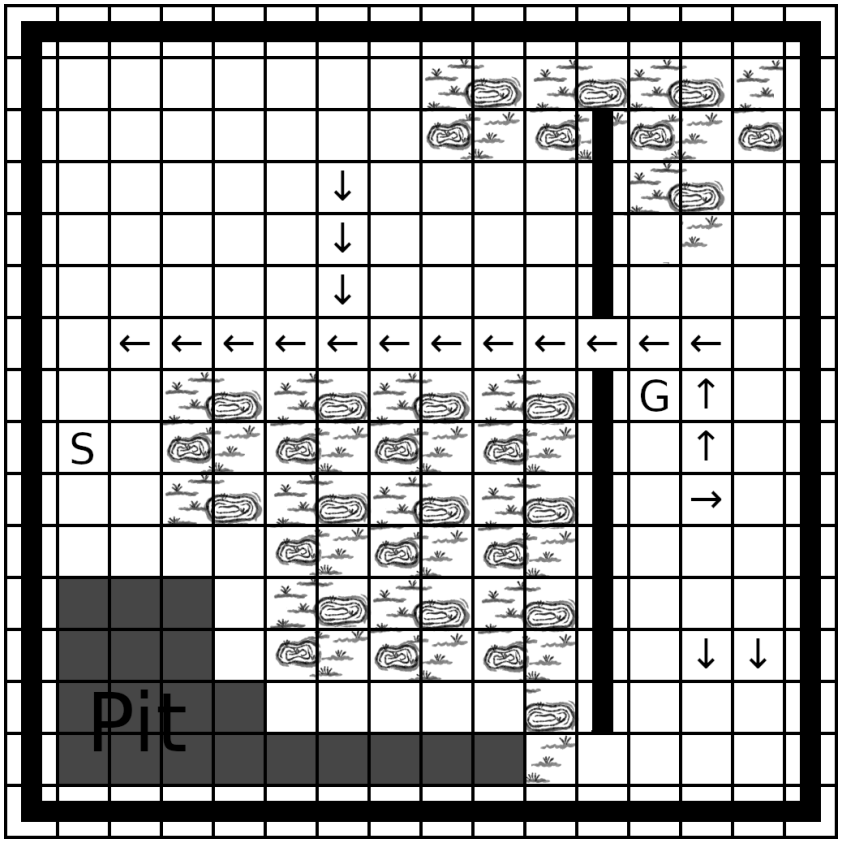

Figure 1: The Gridworld example for task 1.

Bellman equation The underlying problem is easy to solve, since the number of states is small and since the state can be implicitly encoded by the current position of the agent on the grid. That enables us to use so-called *tabular* learning methods, which learn an estimate of the optimal action-value function for each state-action pair. More

formally, the optimal action-value function q_* is the unique solution of the so-called *Bellman optimality equations*:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \tag{1}$$

If you are confused by this notation, you might want to solve task a) before continuing. Once we have q_* (or an approximation), it is trivial to derive an optimal policy.

SARSA One of the most basic algorithms to solve the tabular RL problem is *SARSA*. It maintains an estimate of the value function q_* in a lookup table Q. While the agent is moving around and interacting with the environment, it executes the following algorithm in each time step until the episode ends:

Algorithm 1 SARSA

1: Let S be the current state. With probability ε choose a random action A. Or with probability $1-\varepsilon$ choose the action $A = \arg\max_a Q[S,a]$.
2: Take action A. Receive reward R and new state S' from the environment.
3: Choose a new action A' like in step 1, but with S' as the current state.
4: Update Q according to the equation

$$Q[S,A] = Q[S,A] + \alpha(R + \gamma Q[S',A'] - Q[S,A]),$$

where α is the *step-size* and γ the discount factor, both hyperparameters.
5: Set $A = A', S = S'$ and continue with step 2.

Q-Learning Another widely known algorithm to solve tabular RL problems is *Q-Learning*. It is actually very similar to SARSA in that it also maintains a table of Q values, but the update rule is slightly different. For each time step, we execute the following:

Algorithm 2 Q-Learning

1: Let S be the current state. With probability ε choose a random action A. Or with probability $1-\varepsilon$ choose the action $A = \arg\max_a Q[S,a]$.
2: Take action A. Receive reward R and new state S' from the environment.
3: Update Q according to the equation

$$Q[S,A] = Q[S,A] + \alpha(R + \gamma \max_a Q[S',a] - Q[S,A]),$$

where α is the *step-size* and γ the discount factor, both hyperparameters.
4: Set $S = S'$ and continue with step 1.

1.1 Tasks

Note: You should solve all tasks in this section using only NumPy and the python standard library. You may still use any third-party library for visualizing your results.

a) Write an introduction about reinforcement learning—no longer than two pages—explaining the terms mentioned below. Embed all the explanations into a single continuous text instead of explaining each term separately. In the end, you should also have a solid grasp on the concepts behind the terms and how they are related to each other before continuing with the other tasks.

- Reward function, return
- Markov decision process
- Policy
- Discounting
- Value function, action-value function
- Agent, state, action, environment
- Optimal Policy
- Optimal value function
- Bellman equations

b) Implement SARSA (Algorithm 1) and use it to solve the Gridworld example in Figure 1. The parameter ε controls the frequency of randomly chosen actions and should be a relatively small value. On the other hand, discounting is not necessary here, so we can set $\gamma = 1$.

Visualize the resulting approximation to the optimal policy by drawing one or several possible paths that could be taken under that policy. Also visualize the training progress by making a plot that has all episodes on the x-axis and shows the total return accumulated in that episode on the y-axis.

Note: There is no clear way to tell when and if this algorithm will ever converge in the general case. Your best bet is to run it for a fixed number of steps and decide if you are happy with the outcome.

c) Repeat task b) using different values of ε. Also try setting $\varepsilon = 0$, i.e., the agent deterministically chooses the action with the maximal Q value. Summarize your findings and try to explain the observed behavior.

d) Implement Q-Learning (Algorithm 2), run it on the Gridworld example, and visualize the results as in item b). Compare the results to the ones achieved with SARSA, also for different values of ε.

e) The goal for our Gridworld problem is for the agent to reach the goal state. However, we cannot convey such an explicit goal directly to the agent — in the RL framework, we need to encode it in a reward function that the agent then tries to maximize over time. Of course, the default reward function that we initially

specified for the Gridworld is not the only possible choice. Here we will list it again, together with two alternatives:

1. Receive a reward of -1 in each time step, except for falling into the pits, which incurs a penalty of -1000.
2. The agent receives no reward (i.e., 0) on each time step, except when reaching the goal, which gives a reward of 1. Falling into the pits is still punished with a reward of -1000.
3. Pits remain the same as in the previous two choices, however the agent receives the negative Manhattan distance from its current position to the goal as a reward. More specifically, if the agent moves to position (x_a, y_a) and the goal is at (x_g, y_g), the reward will be

$$R = -|x_g - x_a| - |y_g - y_a|$$

In your report, compare the three alternatives first without actually implementing them: What kind of information is encapsulated in the reward functions? Which one should perform the best? Can you think of any problems you might run into when using one of the alternatives as the reward function? Do some or even all formulations encourage the agent to do something beyond simply reaching the goal?

Once you have answered those questions, implement reward functions 2 and 3 as well and start experimenting. Do the experimental results support the hypotheses you came up with earlier? Report and explain the differing behaviors that you observe.

f) Since the Gridworld only has a very small number of states and actions, we can actually solve the Bellman optimality eq. (1) exactly. In fact, there are a number of algorithms for that, you might want to check Sutton and Barto [3] again for some possibilities.

Compute the optimal action-value function q_* for each of the three reward functions mentioned in item e) and derive the optimal policies from it (the optimal policy is not necessarily unique). Based on those results, evaluate the performance of SARSA and Q-Learning again.

2 Deep Reinforcement Learning

In the previous task we learned about some basic ideas and algorithms that can solve simple, tabular RL problems like the Gridworld. Now we want to extend that knowledge to problems with large or even infinite state space, while still requiring that the action space is small. In theory, we could also conceive problems with large or infinite action spaces, but those are far beyond the scope of this project.

The cart-pole problem More specifically, we want to take a look at the so-called *cart-pole* task. Essentially, the goal is to keep an inverted pendulum upright by moving a cart on its base. The state consists of

1. the cart's position on the x-axis
2. the cart's current velocity
3. the pendulum's current angle
4. and the pendulum's current angular velocity.

At any time, you can only execute one of two actions: move the cart left, or right, by a predefined amount. The reward in each time step is 1, until the episode ends, which is caused either by the pendulum's angle exceeding a certain threshold or the cart moving too far from the center position.

The *Gym* [2] library contains an implementation of the cart-pole problem and several other task and utilities, so please follow the instructions on the website to install it. The code template comes with a driver program that allows you to explore the task by moving around the cart using the arrow keys.

Discretization As you may have feared, all state variables are continuous, so we can no longer store action-values for every combination in a lookup table. One simple work-around that comes to mind is to discretize the state space in some way. For example, we could round the pendulum's angle (in degrees) to the nearest integer and perform a similar strategy for the other variables, allowing us to indeed use the tabular methods from section 1 again.

Deep Q-Learning Instead of discretizing the state space, we can also replace the lookup table Q with a parameterized function $\hat{q}(s, a, \theta)$ that depends only on a finite number of trainable parameters, called θ here. For example, we could choose to use a neural network that takes a state s as the input and outputs the approximate action-value for each action. That doesn't sound like a bad idea at first, especially because neural networks are known as powerful function approximators, but how can we train such a network?

Let's look back at Q-Learning's update rule:

$$Q[S_t, A_t] = Q[S_t, A_t] + \alpha(\underbrace{R_{t+1} + \gamma \max_a Q[S_{t+1}, a]}_{=:G} - Q[S_t, A_t])$$

Essentially, we are trying to update $Q[S_t, A_t]$ in a way that moves its value closer to G, so in a sense we can see $Q[S_t, A_t]$ as the prediction and G as the label for the specific action-value.

The same objective can also be encoded via the squared loss function:

$$\mathcal{L}(Q, S_t, A_t, R_{t+1}, S_{t+1}) = \frac{1}{2}(\underbrace{R_{t+1} + \gamma \max_a Q[S_{t+1}, a]}_{=G} - Q[S_t, A_t])^2.$$

Now replacing the lookup table Q with our approximation $\hat{q}$, we can derive a loss term that allows us to train the neural network:

$$\mathcal{L}(\theta, S_t, A_t, R_{t+1}, S_{t+1}) = \frac{1}{2}(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \theta) - \hat{q}(S_t, A_t, \theta))^2. \tag{2}$$

Note that such a term can be computed for every time step in an episode, and we can use stochastic gradient descent, Adam or any other gradient-based optimization algorithm to train our network on this loss function, just like in the supervised case. We do have to be careful though not to backpropagate gradients when computing the target's term $\max_a \hat{q}(S_{t+1}, a, \theta)$ as we only want to adjust the output of $\hat{q}(S_t, A_t, \theta)$, see `torch.no_grad` for a way to do this.

2.1 Tasks

Note: Every task in this section should be solved using only Gym, NumPy and the python standard library. If a task requires you to implement and/or train a neural network, you may only use PyTorch for that and you can also replace NumPy entirely with PyTorch if you want. Again, for visualizing your results you can also use any other third-party library.

a) Implement a discretization of the cart-pole problem's state space and visualize the results when running it together with SARSA or Q-learning on this problem (the `gym.ObservationWrapper` class might come in handy here). Can those methods actually solve the problem?[1]

b) Implement deep Q-Learning, a version of Q-Learning that works with neural networks, by replacing the original update rule with the NN update in eq. (1). You should also replace the lookup table Q with calls to the prediction function of the

[1] The cart-pole task is considered "solved" once the agent manages to receive a reward of at least 195 for 100 episodes in a row.

neural network. For starters, you might want to try a simple, fully-connected neural network with one or two hidden layers and apply this to the cart-pole problem. As with the other exercises, report and visualize the performance of this algorithm. Can you solve the problem with it?

c) You might find it hard to get deep Q-Learning working properly and find that training it is quite unstable. This comes from the fact that now a single update for a specific action-value-pair S_t, A_t can influence the approximate q values for many other state-vale-pairs, since all of the networks weights may be affected by an update. As subsequent states and actions are highly correlated to each other, the network is under risk to "forget" about values in the regions of the state-action space that were visited earlier.

Therefore, a so-called *Experience Replay* buffer is often used to store tuples $I_t := (S_t, A_t, R_{t+1}, S_{t+1})$ as the agent is exploring the environment. Instead of updating the network in each time step using the currently experienced tuple I_t, we can now completely decouple the update process: Just randomly (and uniformly) sample a batch of tuples $I_{t_1}, I_{t_2}, \ldots, I_{t_k}$ from the replay buffer at any time you want and update the network using these. That means you can still perform an update in every time step, but you could also decide to update the network's weights only every 100 time steps, for example.

Improve your deep Q-Learning implementation so that it uses experience replay. Generally, you want your buffer to be large (say up to 1 million entries or as much as your RAM can handle) and work in a first-in, first-out manner. Re-run this improved version on the cart-pole task and report your results.

2.2 Competition (25 Points)

The competition for this sheet will be about the game *Super Mario Bros.* for the Nintendo Entertainment System (NES). This is as close as we can get to a real life example for Reinforcement Learning without having to put in exorbitant effort. In fact, RL-based systems have been exceptionally successful at playing various types of board[2] and video games[3].

The *gym retro* [1] library builds on top of gym and provides many games on several classic consoles as environments for RL. The documentation describes how to set up the library, download games and how to install them. All games have in common that the observations in each time step are the raw pixel values that would normally be output to the screen, so the RL agents receive essentially the same information as a human player would.

[2] `https://deepmind.com/research/case-studies/alphago-the-story-so-far`
[3] `https://deepmind.com/blog/article/Agent57-Outperforming-the-human-Atari-benchmark`

The library comes with a script that lets you play the games with your keyboard — take a look at the documentation for more details. If you start the game `SuperMarioBros-Nes` using that script, you will see the rewards printed to the console window. Essentially, the return is increased whenever Mario moves to the right into an unseen part of the level and an episode ends with the Game Over screen.

Your goal for this competition will be to achieve the highest return within one episode. The retro library offers a mechanism to record all actions or button presses that the agent performed during an episode to a so-called replay file. Loading and performing those actions again will result in the exact same sequence of states and rewards, since the emulator is deterministic. If you want to participate in the competition, please upload the replay file of your best episode to the repository, along with your trained model and instructions on how to reproduce your result.

Some notes:

- We want to reiterate that participation in the competitions is optional. The competition of this sheet will probably require more time and effort on your side compared to the last two sheets. Therefore, it is absolutely fine and understandable if you decide not to take part in the competition. But on the other hand it can be quite fun to play around with RL on a challenging and interesting task :)
- It is also fine and allowed to take inspiration from other people's code for the competition, as long as you credit them appropriately. In fact, if you start from scratch all by yourselves, you probably won't get too far, as practical RL for those games can be quite difficult to train and a lot of tricks had to be invented to make it work. A good starting point is the linked article about *Agent57* that explains some of the concepts leading to its success and also this blog post[4].
- Don't be discouraged if your agent doesn't learn much at first — it will usually take quite a while until reasonable behavior emerges. Consider using neural networks that were pre-trained on another task. For example, the `torchvision` library comes with several architectures pre-trained on a large corpus of images[5]. However, you are not allowed to use an existing model that was pre-trained on any RL task.

References

[1] *GitHub - openai/retro: Retro Games in Gym.* URL: `https://github.com/openai/retro` (visited on 01/13/2021).

[2] *Gym.* URL: `https://gym.openai.com/` (visited on 01/13/2021).

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018. URL: `http://incompleteideas.net/book/RLbook2020.pdf`.

[4] `https://gsurma.medium.com/atari-reinforcement-learning-in-depth-part-1-ddqn-ceaa762a546f`
[5] `https://pytorch.org/docs/stable/torchvision/models.html`