Reinforcement Learning WS22/23

Assignment 2
Model-free Reinforcement Learning

Horst Petschenig

Presentation:	25.11.2022 12:45
Info Hour:	02.12.2022 12:45, Cisco WebEx meeting, see TC
Deadline:	**15.12.2022 23:55**
Hand-in procedure:	Use the **cover sheet** from the TeachCenter
	Submit your **Python files and report** on the TeachCenter
Course info:	`https://tc.tugraz.at/main/course/view.php?id=3110`
Group size:	up to two students

General remarks

Your submission will be graded based on:

- Correctness (Is your code doing what it should be doing?)
- The depth of your interpretations (Usually, only a couple of lines are needed)
- The quality of your plots (Is everything clearly visible in the print-out? Are axes labeled? . . .)
- Your submission should run with Python 3.7+ **and gym version 0.17.3**
- Both the submitted report and the code. If some results or plots are not described in the report, they will **not** count towards your points!
- Code in comments will not be executed, inspected or graded. Make sure that your code is runnable.

1 RL in a grid world [5 Points]

A grid world is a typical environment with finite action and state spaces. We will solve the Cliff Walking environment from the OpenAI Gym package. The agent controls the movement of a character in a grid world. In this grid world, there is a start and a goal state and the usual actions causing movement up, down, left, and right. The reward is -1 on all transitions except those into the region marked as the cliff. Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

Open the file `ex1_deterministic_SARSA.py` in the code skeleton.

a) Fill the TODOs in `ex1_deterministic_SARSA.py` to solves the task using the SARSA algorithm (SARSA: State, Action, Reward, State, Action). **Important:** During this test phase, you should choose your actions deterministically, i.e. $\epsilon = 0$!

b) Create a copy of the file `ex1_deterministic_SARSA.py` called `ex1_deterministic_Q.py` and implement the off-policy Q learning algorithm.

c) Create a copy of the file `ex1_deterministic_SARSA.py` called `ex1_expected_SARSA.py` and implement the Expected SARSA algorithm.

d) Using the on-policy or the off-policy algorithms you just implemented, observe the influence of the parameter ϵ. Report the average accumulated reward obtained in the test episodes with $\epsilon \in \{0, 0.1, \ldots, 0.9, 1\}$. What is the best value for ϵ? Is it still possible to solve the task with $\epsilon = 0$ and why? Fix ϵ and try $\gamma \in \{0, 0.1, \ldots, 0.9, 1\}$. Describe how it changes the results.

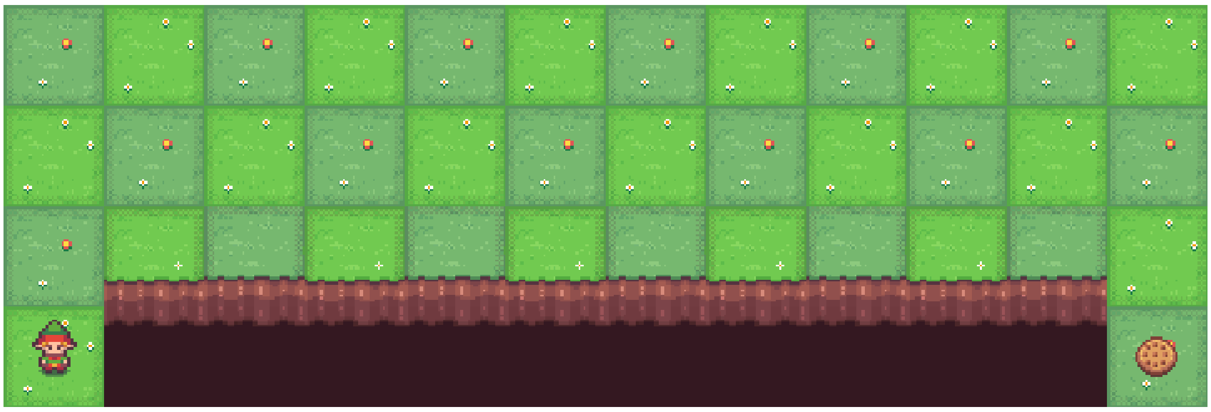

Figure 1: The board is a 4x12 matrix, with (using NumPy matrix indexing): [3, 0] as the start at bottom-left, [3, 11] as the goal at bottom-right, [3, 1..10] as the cliff at bottom-center. If the agent steps on the cliff, it returns to the start. An episode terminates when the agent reaches the goal.

Create plots that show the performance of these methods over the first 200 episodes as a function of

- α and ϵ and
- α and γ

and compare the results! For each hyperparameter search, report the top 3 parameter combinations that worked best (averaged over multiple runs, see code template for details)!

e) In theory, for a continuing MDP, convergence of the policy is only ensured for $\gamma < 1$, but here $\gamma = 1$ works also in practice **and in theory**. Why?

2 RL with non-tabular Q function [5 Points]

If the state space is not discrete and possibly infinite, it is not possible to use the state $\times$ action Q table. It is instead necessary to make use of function approximators such as neural networks. Typically, if the state is a vector in $\mathbb{R}^{n_{state}}$, the simplest choice is to assume that Q is linear: $Q_\theta(s,a) = \theta_a^T s$ where θ is a matrix of parameters of size $n_{state} \times n_{action}$ and each of its rows are associated with an action and are written θ_a^T of size n_{state}. Other choices are possible. For instance, it is beneficial to make assumptions of Q functions that are adapted to the structure of the state vectors. In question c), the Q function will be approximated by a neural network.

The algorithms for on-policy and off-policy TD learning have to change. The main difference is that instead of the update $Q(s,a) \leftarrow Q(s,a) + \alpha(y - Q(s,a))$[1] we perform a gradient descent step to minimize the error of prediction of the return $E = \frac{1}{2}(Q(s,a) - y)^2$.

a) Show that, using a linear model $Q_\theta(s,a) = \theta_a^T s$, the update of the Q table is replaced by the parameters updates:

- if y is assumed constant with respect to θ we have $\theta_a \leftarrow \theta_a - \alpha(Q_\theta(s,a) - y)s$,
- if $y = r + \gamma Q_\theta(s',a')$ it becomes $\begin{cases} \theta_a \leftarrow \theta_a - \alpha(Q_\theta(s,a) - y)s, \\ \theta_{a'} \leftarrow \theta_{a'} + \alpha\gamma(Q_\theta(s,a) - y)s', \end{cases}$

For which of the two parameter updates does the following statement becomes true?

"In a grid world environment, if one rewrites the state s as a vector of $\mathbb{R}^{n_{state}}$ with 1 at position s and zero elsewhere (one-hot encoding), this learning rule is equivalent to the Q learning-algorithms with a standard Q table."

b) We will use PyTorch to solve the CartPole environment. Install PyTorch as explained in the documentation at https://pytorch.org/get-started/ (the following code has been tested on the

[1] y is the one-step-ahead prediction of the return based on TD, in SARSA it is written $y = r + \gamma Q(s',a')$ with s', a' the next state/action pair, in Q-Learning it is $y = r + \gamma \max_{a'} Q(s',a')$

latest version of PyTorch with Linux ×64, and the CPU only package – report if you encounter any problem). In the file `ex2_cartpole_linear_sarsa.py` implement SARSA with a linear model. In the current setup, the performance of the algorithm is quite poor (it learns sometimes but for most random seeds it only holds the pole upright for less than 30 time steps even after training) and we would like to improve it by changing the model of the Q function. The main advantage of using PyTorch is that it will compute all the gradients automatically and we do not need to compute them by hand.

Read and understand the code. What is the parameter `eps_decay`? How will it affect the convergence of the algorithm?

c) Create a copy of `ex2_cartpole_linear_sarsa.py` called `ex2_cartpole_non_lin_sarsa.py`, change the Q model to a neural network with one hidden layer as described below. If s is the observed state vector of size n_{state}, and n_{hidden} is your number of hidden neurons:

- The activation of the hidden layer is given by $a_y = W_{hid}s + b_{hid}$ where W_{hid} is a variable matrix of size (n_{hidden}, n_{state}) and b_{hid} is a variable vector of size n_{hidden}.
- The output of the hidden layer is defined as $y = \text{torch.relu}(a_y)$ (`relu` computes the rectified linear function, which is 0 if x is negative and x otherwise). Try different activation functions such as `torch.tanh`.
- The activation of the output layer is given by $a_z = wy + b$ where w is a matrix of size (n_{action}, n_{hidden}) and b is a vector of size (n_{action}).

All of the steps above should be implemented using pre-defined PyTorch modules (`Sequential`, `Linear`, ...) in combination with the Adam optimizer. You should be able to solve the task in less than 1000 episodes with the following parameters: `num_hidden` = 20, `eps` =1., `alpha` = 1e-3, `gamma` = .9, `eps_decay` = .999. Can you find a set of parameters that works better? Find a suitable learning rate for the Adam optimizer (you do not have to tune the other parameters of the optimizer).

d) Create copies of `ex2_cartpole_linear_sarsa.py` and `_non_linear_sarsa.py` called `..._linear_Q` and `..._non_lin_Q`. Replace the SARSA algorithm by Q-Learning (off-policy). You will have to find a new set of parameters. Then choose another gym environment among `'MountainCar-v0'`, `'Pendulum-v0'` or `'Acrobot-v1'`. Use Q learning or SARSA to solve it, report your results. You might have to change the reward and adapt the non-linearity of your Q model. First try a linear model, try to get good performances by changing the parameters and then try a neural network. Compare the results obtained with both algorithms.