Autonomously Learning Systems WS21/22

Assignment 1
Model-free Reinforcement Learning

Horst Petschenig

Presentation:	25.11.2021 14:00
Info Hour:	02.12.2020 14:00, Cisco WebEx meeting, see TC
Deadline:	**09.12.2020 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+
- 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.

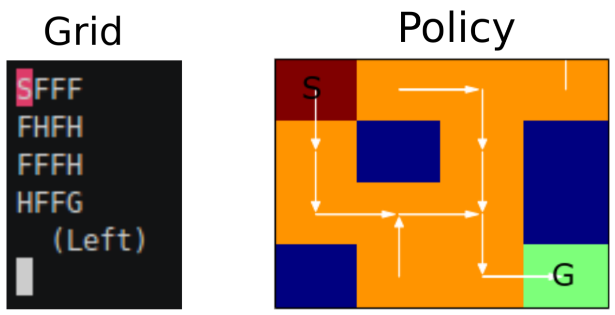
1 RL in a grid world [5 Points]

A grid world is a typical environment with finite action and state spaces. We will solve the FrozenLake environment from the OpenAI Gym package. The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Open the file `ex1_deterministic_SARSA.py` in the code skeleton. This file creates the environment without slippery floor.

a) Fill the TODOs in `ex1_deterministic_SARSA.py` to solves the task using the SARSA algorithm (SARSA: State, Action, Reward, State, Action). After training on 1000 successive episodes, you should easily be able to obtain some reward in more than 70 out of 100 following episodes that we will call test episodes. **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` and implement the off-policy Q learning algorithm.



```
SFFF
FHFH
FFFH
HFFG
  (Left)
```

Figure 1: The **Grid** on the left is the environment as generated by the gym framework. The **Policy** on the right is an example of policy learnt with the SARSA algorithm to navigate into the grid. The official documentation describes the environment as follows: “Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend. The surface is described using the grid shown above. **S:** starting point, safe. **F:** frozen surface, safe. **H:** hole, fall to your doom. **G:** goal, where the frisbee is located. The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise.”

c) Using either 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 $\epsilon \in \{0, 0.01, 0.1, 1\}$. What is the best value for ϵ? Is it still possible to solve the task with $\epsilon = 0$ and why? Fix ϵ and try $\gamma \in \{0, 0.2, 0.7, 1\}$. Describe how it changes the results. 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?

d) Create a copy of `ex1_deterministic_SARSA.py` and `..._Q.py` called `ex1_slippery_SARSA.py` and `..._Q.py`, and set the floor slippery by setting the argument `slippery=True` in the constructor of the FronzenLake object. Set the number of training episodes to 10000, and try $\epsilon = 0.1$ or 0.01 and $\gamma = 1$. It should be much harder now. We now decide to help the agent by providing a more informative reward. The return estimated by the Q table is not the accumulated `reward` anymore, but the accumulation of a `modified_reward` described as:

- `modified_reward=100` at the goal (satisfied when `reward=1` and the task is `done`)
- `modified_reward=-100` in holes (satisfied when `reward=0` and the task is `done`)
- `modified_reward=-1` for taking any other step

Will the algorithm performance change if you use the parameters $\epsilon = 0.1$ or 0.01 and $\gamma = 1$? Explain how changing the reward influences the agent.

e) Change the `map_name` at creation of the FronzenLake object to `map_name=8x8`. Optimize your parameters to solve the task and report your results. You can use the settings from task c (i.e., no reward modification, no slippery floor).

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 fact, in a grid world environment such as the Frozen Lake, if one artificially 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 effectively equivalent to the Q learning-algorithms with 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 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.

[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 $y = r + \gamma \max_{a'} Q(s', a')$