ERHVERVS AKADEMI AARHUS

Artificial Intelligence & Automation F2022.

Sila. March 25th, 2022.

Mandatory Exam Assignment 1. Spring 2022.

Complete as many of the following exercises as you are able to.
Credit will be given for the efficiency and effectiveness of your solutions and for clear written explanations of the solutions and (important) your own thinking in making the solutions, including the decisions, limitations and interpretations you have made in order to come up with your solution. You must include screen dumps and/or illustrations in your report. So, that it is possible to understand the intended functionality, even if it is (for some reason) difficult to run the handed in scripts. It is your responsibility to find a reasonable level for these illustrations/screen dumps. You must hand-in individuel reports, group hand-ins are not allowed in this exam. Also remember that this is an exam assignment, and it is therefore, as standard for exam assignments, not allowed to make your solution publicly available on the internet, neither before or after the handin Your hand-in can be written in danish or in english. You can do the exercises in any order you want – they are not dependent on each other.
Your report must be handed in on ***Wiseflow on Monday the 4th of April 2022 at 20.00 O'clock*** *at the latest.*
If the report consists of multiple files (report, scripts) then it must be zipped into one file.

Exercise 1.

Earlier, in this course, we have looked at the library OpenCV, https://opencv.org/, and used it for object detection (in Python programs). In this exercise you should make an object detection program that can go through a sequence of detections.

First your program should be able to detect motions (I.e. the programs starts out as a motion detector). After a while in this state, with detected motion, the program should move on, and try to detect a pedestrian. Then, after a while with a successful detection of a pedestrian, it should move on to face detection.
Also: Try to add a color detection to the pedestrian detection, so the program can tell whether a pedestrian is wearing, say, blue or orange (E.g. workwear in different colors). Here, it is not necessary to change state based on this information, but the program should/could output the information.

(You might) Write in the video-frame what state your detector is in. And, remember, as usual, that it is perfectly ok to test your program on printed pictures… The mechanism for going from one state to another can be programmed as sophisticated algorithms. But here it in this prototype, it is perfectly ok just to use some keyboard input, or similar, to switch between the various detection modes. Include screen printouts in your documentation of your program.

Extra (if you have time): Feel free to include (OpenCV) face recognition to your flow, so that the last state in your flow can face-recognize 1 or 2 people. Briefly, in week 2, we mentioned YOLO object detection (with OpenCV in Python). And you can also consider adding this to the flow –

Again, if you think it adds value to your solution, and if you have time for it.

Exercise 2.

Earlier, in this course, we have looked at neural nets as a way to approximate functions.
Specificly, we used tensorflow and keras to approximate the x^2 function in the interval -50 to 50.
In this exercise we want to look at the function:

```
y = x*x*x - 6*x**2 + 4*x + 12
```

in the (x) interval -2 to 6 (Btw. Remember that you can use WolframAlpha to investigate functions).

Now you should:

a) Describe the architecture of a neural net that can approximate that function reasonably well. How many hidden layers, with how many neurons, do you suggest we use?
b) Describe how many points you needed to train the neural net? How many epochs did you use to train the net?
c) Include a plot of your results.
d) Repeat questions a,b and c for the sin(x) function in the interval 0 to 10.

You decide what libraries you want to use to solve the problem. And, of course, what Python environment you want to use. But remember that we used Tensorflow and Keras to solve a very similar problem in this course, week 3.

Exercise 3.a.

You were introduced to the (Gym) Taxi simulator in week 5 of the course.

Here our agent is driving a taxi. There are four locations and our agent has to pick up a passenger at one location and drop the passenger at the another. The agent will receive +20 points as a reward for successful drop off and -1 point for every time step it takes. The agent will also lose -10 points for illegal pickups and drops. So, the goal of our agent is to learn to pick up and drop passengers at the correct location in a short time without boarding any illegal passengers.
We trained the agent using q-learning (See the files Taxi_1.py & Taxi_2.py on Canvas) and saw that it learned to move around properly in the environment.

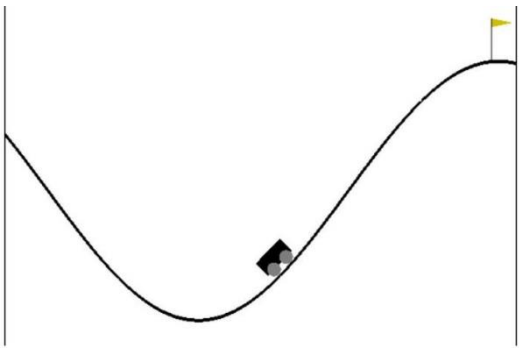
But how good was our q-learning agent actually?

a) Describe what parameter values (alpha, gamma, epsilon) you think we should/could use

(according to your experiments). And what kind of average reward we can (then) expect, if we run our agents for a long time (100.000 steps or similar…), with your settings. Compare that with an agent that only takes random actions (which you should code). And describe how the performance of q-learning (hopefully) gives better results, as we move along, with more episodes. It would be nice to see some plots, or handmade illustrations for that matter, of your results.

Exercise 3.b.

We will now look at the "Mountain car" problem.

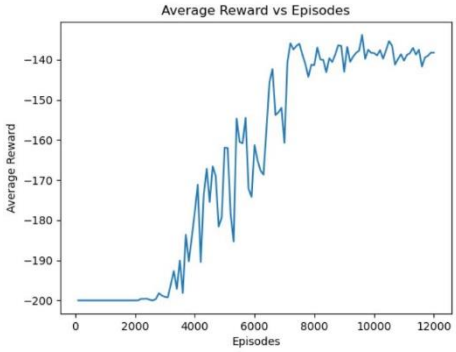


In this problem we get a reward, +1, for reaching the flag with the car. We can push left, stay, or push right (Not that important here, but left is usually coded as an 0 action, 1 means stay still, and 2 means push right).
We could ask the car to go right constantly, but it just doesn't quite have the power to make it. Instead, we need to build momentum to reach the flag. To do that, we'd want to move back and forth on the slopes in order to build up momentum. We could program a function to do this task for us. And yes, you are free to do that, of course.
You can see what happens if you try some random actions, if you run the method "PlayTheGame" in the file MountainCar.py (On Canvas).
A timestep where we don't reach the flag gives us a reward of -1, and the simulator stops an episode after 200 timesteps, if we haven't reached the flag by then. So, if we end up with something better than -200 means that we have reached the flag before the time was up in that episode.

Positions and velocities are continuous numbers. The car can be in position 0.2, 0.3, 0.4 etc. and everything in between (0.22, 0.27 and 0.45677 to mention a few positions…). Just as the car can have velocities from 0 to 0.07, and everything in between.

This is a problem, as we would like to have a finite amount of states (and actions). Why? Because with a finite amount of states, and actions, q-learning might help us to come up with a solution. Without exploration we could reach the flag (starting from some initial state) by taking the action with the highest q-value for a given state, Q(state, action).
So, how could we use q-learning in the Mountain Car problem? I.e.

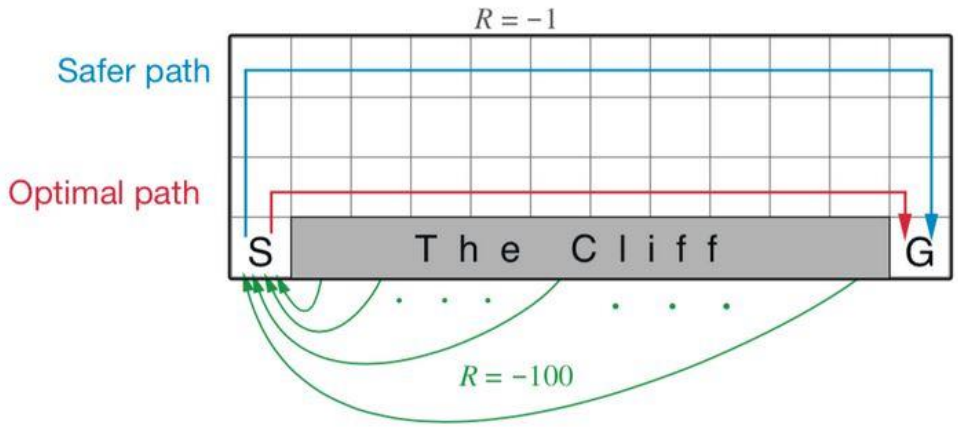
a) How could we fix the "infinite number of states" problem in the Mountain Car scenario? What would you suggest (Take a look in the code, MountainCar.py)? Give a short description of your/the idea (Make print statements in the code in order to understand what the code is doing).



b) Yes, learning is pretty slow in the given code. Can you come up with code changes that makes it learn faster (Number of states you use, learning rate, discount rate etc.). What changes in the code did you try? Did any of your experiments give better results, or faster, than the given code?
Extra: We are here using q-learning. If you have time (☺) you can change the (code) implementation to use Sarsa. And run a few experiments with that code. How did that work?

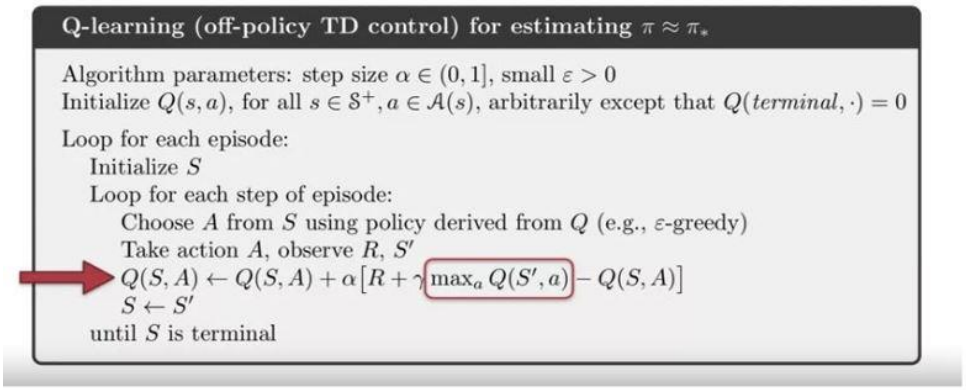
Exercise 4

Cliff walking.



Cliff walking is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked Cliff. Stepping into this region incurs a reward of optimal path -100 and sends the agent instantly back to the start. Indeed, we want to get to the goal as soon as possible.

We will again use Q-Learning to solve the problem.



Source: RL by Sutton and Barto

In the code (Cliffwalking.py on Canvas) we have for the end state:

```
reward = self.cliff.giveReward()
```

And then we back-track where we have been, and update according to q-learning.

```
for s in reversed(self.states):
    pos, action, r = s[0], s[1], s[2]
    current_value = self.state_actions[pos][action]
    reward = current_value + self.lr * (r + reward - current_value)
    self.state_actions[pos][action] = round(reward, 3) # 3 decimals
    # update using the max value of S'
    reward = np.max(list(self.state_actions[pos].values()))  # max
```

Questions:

a) The q-learning algorithm is initially set to 500 rounds of learning. What happens in the program if you set this to say 30 or 50 rounds of learning?

b) Can you improve the learning with other settings for exploration (greediness) and alpha (learning rate) than the preset 0.1 and 0.1? What values?

c) After q-learning with 500 rounds of learning, we then want to write out our best route. A new agent inherits our learned q-values, and we start it out in state S, and see that it finds the goal state. We set the agents exploration rate to 0, but why not 0.2? You might want to try it out by adjusting the code.

Instead of q-learning, we could also have used Sarsa. Which only differs slightly from q-learning.

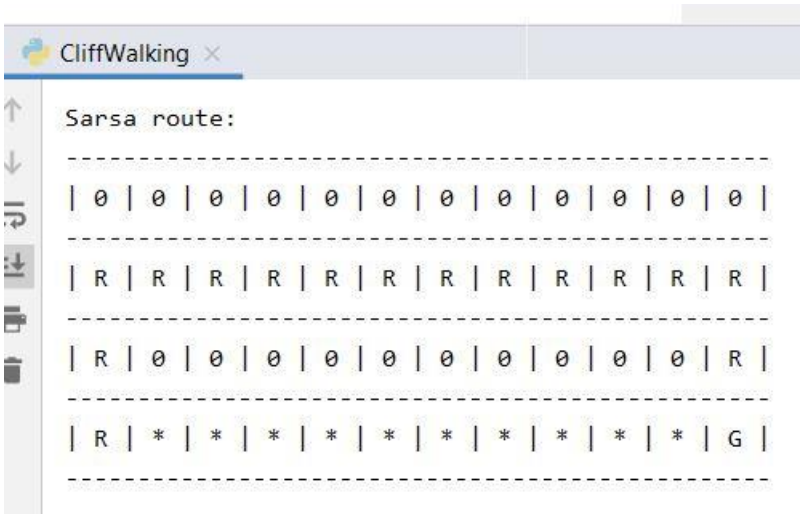
According to Sutton and Barto, in the RL textbook:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Here, we update our q values (state, action) not with the best possible action in the next step, but with the action we should take (in the next step), according to what we have learned.

If we try to run it, we get something like this (remove the exit, where there is remark for exercise d in the code):



d) Based on the algorithms: Explain in your own words why Sarsa (here) finds another route that q-learning. Experiment with the number of rounds. I.e. what are the number of rounds with what settings for exploration (greediness) and alpha (learning rate) you need to get a stable answer?

Exercise 5

Frozenlake and RL.

"*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. Notice, a slippery parameter can be set in such a way 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*".

https://gym.openai.com/envs/FrozenLake-v0/

Notice, FrozenLake-v1 is considered "solved" when the agent obtains an average reward of at least 0.78 over 100 consecutive episodes.

In the notebook Frozenlake_QLearning.ipynb you will find q-learning code that solves the problem (You are free to make your own q-learning code that solves the problem, but using the given code is fine).

a) Describe settings for the parameter values for this q-learning problem. Investigate changes to these parameter settings and report your findings. What values are better or just as good, and what changes do not work (Investigate at least 5-10 changes. And report your findings in a table with these, 5-10, rows).
b) Can the problem be solved with a SARSA or an expected SARSA algorithm? Make an implementation of one of these algorithms for the Frozenlake problem in order to prove your point.