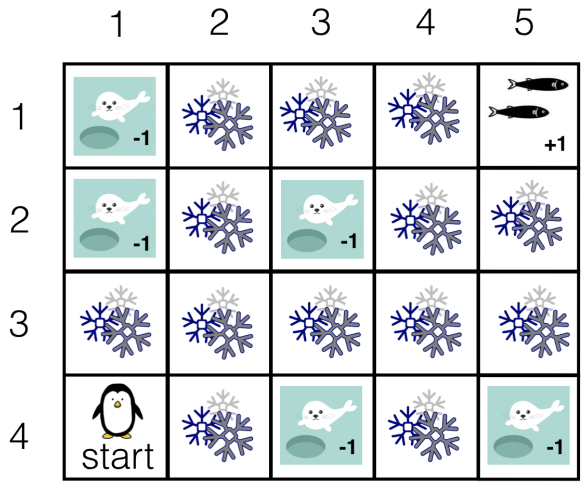
Deep Reinforcement Learning

Dynamic Programming, Monte-Carlo Sarsa, Q-learning

Lecturer: Armin Biess, Ph.D.

Due date: 27.5.21



Frozen Lake: Given is a penguin on a frozen lake, which is described by a 4x5 grid with holes and a goal state (fish), both defining terminal states. For every transition the penguin gets a reward of $r = -0.04$. For transitions to terminal states the penguin gets *in addition* a reward of $+1$ for the goal state and a reward of -1 for the holes. The penguin can take four possible actions $= \{N, E, S, W\}$, but the surface is slippery and only with probability 0.8 the penguin will go into the intended direction and with probability 0.1 to the left and 0.1 to the right of it. It is assumed that the boundaries are reflective, i.e., if the penguin hits the edge of the lake it remains in the same grid location. Assume a discount factor of $\gamma = 0.9$ throughout the exercise.

1. Dynamic Programming [40 pts]

Find the optimal policy for the penguin to reach the fish by solving the corresponding MDP using dynamic programming . You can choose either value or policy iteration. Add to the `env` a new method called `env.value/policy_iteration`. *Hint*: First construct the state-transition model and reward function. Generate two figures showing the optimal state value function and optimal policy.

2. Monte-Carlo RL [20 pts]

Implement a Monte-Carlo GLIE, first-visit, α-constant control algorithm for the frozen lake task. Find a suitable step-size parameter α and ε-decay schedule for GLIE. Add to the `env` a new method called `env.mc_control`. Generate three figures showing the optimal state value function, optimal action value function and optimal policy.

3. Sarsa [20 pts]

Implement SARSA for the frozen lake task. Find a suitable step-size parameter α and ε-decay schedule for GLIE. Add to the `env` a new method called `env.sarsa`. Generate three figures showing the optimal state

value function, optimal action value function and optimal policy.

4. Q-Learning [20 pts]

Implement a Q-learning algorithm for the frozen lake task. Find a suitable step-size parameter α and ε-decay schedule for GLIE. Add to the `env` a new method called `env.q_learning`. Generate three figures showing the optimal state value function, optimal action value function and optimal policy.

The following functions are provided:

A `GridWorld` class with the following methods:

- `render` draws the gridworld and the agent in the gridworld
- `show` displays the gridworld and the agent in the gridworld
- `reset` sets the agent back to the starting state. It can take an argument *"exploring_starts"*, for which the agent starts at a random initial state (`true`) or at the initial state 4 if no argument is provided.
- `step` takes as input an *action* and returns a *next_state*, *reward* and a binary variable *done* indicating whether the episode ended.
- `close` closes the gridworld
- `plot_values` takes as input a vector of values $V(s)$ and plots the values in the gridworld
- `plot_qvalues` takes as input a matrix $Q(s, a)$ and plots the values in the gridworld
- `plot_policy` takes as input a vector $\pi(s)$ or matrix $\pi(a|s)$ and plots the policy

Remarks:

- Label the states as follows:

gridworld

1	5	9	13	17
2	6	10	14	18
3	7	11	15	19
4	8	12	16	20

- Label the actions as follows: $\{N, E, S, W\} = \{0, 1, 2, 3\}$.
- Experiment with the functions by executing main.py:

```
1 from env import World
2 import numpy as np
3
```

```
4
5  if __name__ == "__main__":
6
7      env = World()
8
9      num_episodes = 10
10
11     for n_episode in np.arange(0, num_episodes):
12
13         env.reset("exploring_starts")
14         done = False
15         t = 0
16         env.show()
17         while not done:
18             env.render()
19             action = np.random.randint(1, env.nActions ) # take a random action
20             next_state, reward, done = env.step(action)  # observe next_state and reward
21
22             env.close()
23             t += 1
24             if done:
25                 print("Episode",n_episode + 1, "finished after {} timesteps".format(t ...
                       + 1))
26                 break
```