Machine Learning 2019/2020: Assignment 4 - Reinforcemen

Deadline: Friday 6th of December 2019 9pm

First name: Brian
Last name: Pulfer

About this assignment

In this assignment you will further deepen your understanding of Reinforcement Learning (RL).

Submission instructions

Please write your answers, equations, and code directly in this python notebook and print the final
code has appropriate line breaks such that all code is visible in the final pdf. Also select A3 for the
clipped.

The final pdf must be named name.lastname.pdf and uploaded to the iCorsi website before the de
0 points.

Also share this notebook (top right corner 'Share') with teaching.idsia@gmail.com during submis

Keep your answers brief and respect the sentence limits in each question (answers exceeding th

Learn more about python notebooks and formatting here: https://colab.research.google.com/note

How to get help

We encourage you to use the tutorials to ask questions or to discuss exercises with other students
by others or share your report with others. Violation of that rule will result in 0 points for all student
send an email to louis@idsia.ch

1 Basic probability (6p)

Suppose that a migrating lizard that rests in Ticino can be in four different states: Eating (E), Sleepi
example protecting its territory against other lizards. Each lizard spends 30% of its time sleeping, 4
time mating. A biologist collects a population of lizards and puts them in a cage to study their beha
being caught while eating is 0.1, for a sleeping lizard 0.4, for a fighting lizard 0.8 and for the lizards

Question 1.1 (3p)

What is the relative frequency (probability) for a lizard being caught in the cage?

ANSWER HERE

0.4*0.1 + 0.3*0.4 + 0.2*0.8 + 0.1*0.2 = 0.34

Question 1.2 (3p)

What is the proportion of lizards that are fighting of those that were caught in the cage?

ANSWER HERE

0.2*0.8 / 0.34 ≈ 0.4705

2 Markov Decision Processes (32p)

Suppose a robot is put in a maze with long corridor. The corridor is 1 kilometer long and 5 meters v
moving forward for 1 meter, moving backward for 1 meter, turning left for 90 degrees and turning ri
hits the wall, then it will stay in its position and orientation. The robot's goal is to escape from this r
Note: the answers in the following questions should not exceed 5 sentences.

Question 2.1 (4p)

Assume the robot receives a +1 reward signal for each time step taken in the maze and +1000 for r
corridor). Then you train the robot for a while, but it seems it still does not perform well at all for na
maze. What is happening? Is there something wrong with the reward function?

ANSWER HERE

What is happening is that the biggest reward is given to the robot only after a very long sequence c
robot is random, he will hardly ever find out that there is a much bigger reward of +1'000 at the end
spend all it's time recieving +1 rewards simply staying in the corridor.

The reward function should not incourage the robot staying in the corridor as its doing (by giving +

Question 2.2 (4p)

If there is something wrong with the reward function, how could you fix it? If not, how to resolve the

---

ANSWER HERE

Instead of giving a +1 reward at each timestep (incouraging the robot staying in the corridor), we c
rotates/moves towards the closest exit of the corridor with respect to his position and give a -1 rev
further exit or the wall.

Questions 2.3 (2p)

The discounted return for a non-episodic task is defined as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

where $\gamma \in [0, 1]$ is the discount factor.

Rewrite the above equation such that G_t is on the left hand side and G_{t+1} is on the right hand side

---

ANSWER HERE The above fomula can be rewritten (in terms of G_t and G_{t+1}) as:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Questions 2.4 (2p)

What is the sufficient condition for this infinite series to be a convergent series?

ANSWER HERE

The sufficient condition for the series to converge is:

$$0 \leq \gamma < 1$$

Questions 2.5 (5p)

Suppose this infinite series is a convergent series, and each reward in the series is a constant of +
simple formula for this bound ? Write it down without using summation.

ANSWER HERE

$$bound = 1 + \frac{\gamma}{1 - \gamma}$$

Questions 2.6 (5p)

Let the task be an episodic setting and the robot is running for $T = 5$ time steps. Suppose $\gamma = 0.$
way $R_1 = -1$, $R_2 = -0.5$, $R_3 = 2$, $R_4 = 1$, $R_5 = 6$. What are the values for G_0, G_1, G_2, G_3

ANSWER HERE

$$G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \gamma^4 R_5 = -0.89$$
$$G_1 = R_2 + \gamma R_3 + \gamma^2 R_4 + \gamma^3 R_5 = 0.352$$
$$G_2 = R_3 + \gamma R_4 + \gamma^2 R_5 = 2.84$$
$$G_3 = R_4 + \gamma R_5 = 2.8$$
$$G_4 = R_5 = 6$$
$$G_5 = 0$$

Questions 2.7 (5p)

Suppose each reward in the series is increased by a constant c, i.e. $R_t \leftarrow R_t + c$. Then how does

---

ANSWER HERE

Supposing the episode has T timesteps, G_t was previously defined as:

$$G_t = \sum_{j=0}^{\infty} \gamma^j R_{t+1+j}$$

After adding a constant c to every reward, G_t becomes:

$$G_t = \sum_{j=0}^{\infty} \gamma^j (R_{t+1+j} + c)$$

The total value of G_t increases for any non-negative constant c.

Questions 2.8 (5p)

Now consider episodic tasks, and similar to Question 2.7, we add a constant c to each reward, how

---

ANSWER HERE

Supposing the episode has T timesteps, G_t was previously defined as:

$$G_t = \sum_{j=0}^{T-t} \gamma^j R_{t+1+j}$$

After adding a constant c to every reward, G_t becomes:

$$G_t = \sum_{j=0}^{T-t} \gamma^j (R_{t+1+j} + c)$$

The total value of G_t increases for any non-negative constant c.

3 Dynamic Programming (62p)

Questions 3.1 (5p)

Write down the Bellman optimality equation for the state value function without using expectation instead. Define all variables and probability distributions in bullet points.

ANSWER HERE

$$V^{\pi}(s) = \sum_{a} \pi(s,a)(\sum_{s'} \gamma V^{\pi}(s') \sum_{r'} r' P(s', r' | a, s$$

- π(s,a) = Probability of doing action a in state s according to policy π
- γ = Discount factor ($0 \leq \gamma < 1$)
- $P(s', r' | a, s)$ = Probability of getting reward r' and transitioning to state s' given that we take

Questions 3.2 (5p)

Write down the Bellman optimality equation for the state-action value function without using expec
distributions instead. Define all variables and probability distributions in bullet points.

ANSWER HERE

$$Q^{\pi}(s,a) = \sum_{s'} \gamma V(s') \sum_{r'} P(s',r'|s,a) + \sum_{r'} r' P(s',r$$

- $V(s')$ = Value function for state s

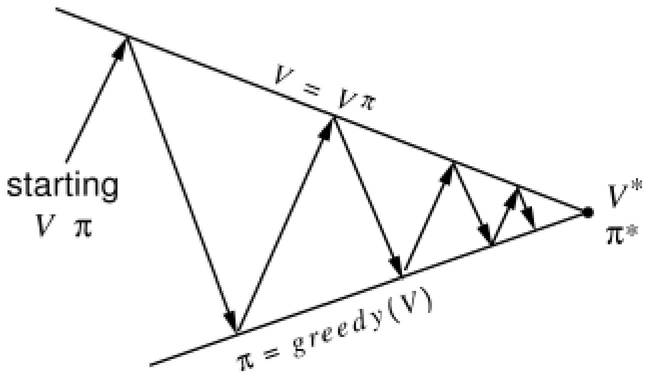
Questions 3.3 (15p)

Consider a 4x4 gridworld depicted in the following table:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The non-terminal states are $S = \{1, 2, \dots, 14\}$ and the terminal states are $\bar{S} = \{0, 15\}$. There a
is $A = \{\text{up, down, left, right}\}$. Assume the state transitions are deterministic and all transitions
termination all rewards are zero). If the agent hits the boundary, then its state will remain unchange
$p(s = 8, r = -1|s = 8, a = \text{left}) = 1$. Note: In this exercise, we assume the policy is a determi

Manually run the policy iteration algorithm (see lecture slide 58) for one iteration. Use the in-place
time policy evaluation with a single pass through the states (16 equations) and one time policy im
all 16 cells are 0.0 and the policy initially always outputs the 'left' action. Write down the equations
updated values of each cell. Use a discount factor $\gamma = 0.5$. Write down the policy after policy impr



Read more about this in Sutton & Barto's book http://www.incompleteideas.net/book/ebook/node4

ANSWER HERE

Policy evaluation

$$V(0) = 0$$

$$
\begin{gathered}
V(1) \leftarrow -1 + \gamma V(0) = -1 + 0 = -1 \\
V(2) \leftarrow -1 + \gamma V(1) = -1.5 \\
V(3) \leftarrow -1 + \gamma V(2) = -1.75 \\
V(4) \leftarrow -1 + \gamma V(4) = -1 \\
V(5) \leftarrow -1 + \gamma V(4) = -1.5 \\
V(6) \leftarrow -1 + \gamma V(5) = -1.75 \\
V(7) \leftarrow -1 + \gamma V(6) = -1.875 \\
V(8) \leftarrow -1 + \gamma V(8) = -1 \\
V(9) \leftarrow -1 + \gamma V(8) = -1.5 \\
V(10) \leftarrow -1 + \gamma V(9) = -1.75 \\
V(11) \leftarrow -1 + \gamma V(10) = -1.875 \\
V(12) \leftarrow -1 + \gamma V(12) = -1 \\
V(13) \leftarrow -1 + \gamma V(12) = -1.5 \\
V(14) \leftarrow -1 + \gamma V(13) = -1.75 \\
V(15) = 0
\end{gathered}
$$

Policy improvement It is enough, for every state, to compute the $argmax(left, up, right, down)$. sum of the reward of transitioning from the current state to summed with the discounted value fro

$$
\begin{gathered}
\pi'(1) = argmax_a(-1 + \gamma V^\pi(0), -1 + \gamma V^\pi(1), -1 + \gamma V^\pi(2), -1 \\
\pi'(2) = argmax_a(-1 + \gamma V^\pi(1), -1 + \gamma V^\pi(2), -1 + \gamma V^\pi(3), -1 \\
\pi'(3) = argmax_a(-1 + \gamma V^\pi(2), -1 + \gamma V^\pi(3), -1 + \gamma V^\pi(3), -1 \\
\pi'(4) = argmax_a(-1 + \gamma V^\pi(4), -1 + \gamma V^\pi(0), -1 + \gamma V^\pi(5), -1 \\
\pi'(5) = argmax_a(-1 + \gamma V^\pi(4), -1 + \gamma V^\pi(1), -1 + \gamma V^\pi(6), -1 + \\
\pi'(6) = argmax_a(-1 + \gamma V^\pi(5), -1 + \gamma V^\pi(2), -1 + \gamma V^\pi(7), -1 + \\
\pi'(7) = argmax_a(-1 + \gamma V^\pi(6), -1 + \gamma V^\pi(3), -1 + \gamma V^\pi(7), -1 + \\
\pi'(8) = argmax_a(-1 + \gamma V^\pi(8), -1 + \gamma V^\pi(4), -1 + \gamma V^\pi(9), -1 \\
\pi'(9) = argmax_a(-1 + \gamma V^\pi(8), -1 + \gamma V^\pi(5), -1 + \gamma V^\pi(10), -1 \\
\pi'(10) = argmax_a(-1 + \gamma V^\pi(9), -1 + \gamma V^\pi(6), -1 + \gamma V^\pi(11), -1 \\
\pi'(11) = argmax_a(-1 + \gamma V^\pi(10), -1 + \gamma V^\pi(7), -1 + \gamma V^\pi(11), -1 \\
\pi'(12) = argmax_a(-1 + \gamma V^\pi(12), -1 + \gamma V^\pi(8), -1 + \gamma V^\pi(13), -1 \\
\pi'(13) = argmax_a(-1 + \gamma V^\pi(12), -1 + \gamma V^\pi(9), -1 + \gamma V^\pi(14), -1 \\
\pi'(14) = argmax_a(-1 + \gamma V^\pi(13), -1 + \gamma V^\pi(10), -1 + \gamma V^\pi(15), -1
\end{gathered}
$$

Questions 3.4 (15p)

Implement the beforementioned environment in the code skeleton below. Come up with your own s third party source.

Imports

↳ *1 cell hidden*

Defining the problem

```
class GridWorld:
  UP = 0
  DOWN = 1
  LEFT = 2
  RIGHT = 3

  def __init__(self, side=4):
    self.side = side
    # -------------------------
    # Define integer states, actions, and final states as specified in the problem

    # TODO insert code here
    self.actions = [self.UP, self.DOWN, self.LEFT, self.RIGHT]
    self.states = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
    self.finals = [0, 15]

    # -------------------------
    self.actions_repr = np.array(['↑', '↓', '←', '→'])

  def reward(self, s, s_next, a):
    # -------------------------
    # Return the reward for the given transition as specified in the problem descri

    # TODO insert code here
    if s in self.finals:
        return 0
    return -1
    # -------------------------

  def transition_prob(self, s, s_next, a):
    # -------------------------
    # Return a probability in [0, 1] for the given transition as specified in the p

    # TODO insert code here
    states_matrix = [[0, 1, 2, 3],
                     [4, 5, 6, 7],
                     [8, 9, 10, 11],
                     [12, 13, 14, 15]]

    actual_next = -1

    for row_index in range(len(states_matrix)):
        for col_index in range(len(states_matrix[row_index])):
            if s == states_matrix[row_index][col_index]:
                if a == self.UP:
                    if row_index -1 >= 0:
                        actual_next = states_matrix[row_index-1][col_index]
                    else:
                        actual_next = states_matrix[row_index][col_index]
                elif a == self.DOWN:
                    if row_index +1 < 4:
                        actual_next = states_matrix[row_index+1][col_index]
                    else:
                        actual_next = states_matrix[row_index][col_index]
                        actual_next = states_matrix[row_index][col_index]
                elif a == self.LEFT:
                    if col_index -1 >= 0:
                        actual_next = states_matrix[row_index][col_index-1]
                    else:
                        actual_next = states_matrix[row_index][col_index]
                elif a == self.RIGHT:
                    if col_index +1 < 4:
                        actual_next = states_matrix[row_index][col_index+1]
                    else:
                        actual_next = states_matrix[row_index][col_index]
                else:
                    print("Impossible action specified!")

                if actual_next == s_next:
                    return 1
                else:
                    return 0
    return 0
    # -------------------------

  def print_policy(self, policy):
    P = np.array(policy).reshape(self.side, self.side)
    print(self.actions_repr[P])

  def print_values(self, values):
    V = np.array(values).reshape(self.side, self.side)
    print(V)
```

Questions 3.5 (17p)

Implement policy iteration in the code skeleton below. Come up with your own solution and do not

Run the code multiple times. Do you always end up with the same policy? Why? (max 4 sentences)

ANSWER HERE

No, we don't always end up with the same policy. This is due to the fact that multiple optimal polici behaviour is randomly picked.

Policy iteration

```
def eval_policy(world, policy, values, gamma=0.9, theta=0.01):
  # -------------------------
  # Implement policy evaluation and return the updated value function

  # TODO insert code here
    delta = -1

    while delta > theta or delta == -1:
        for state in range(len(world.states)):
            action = policy[state]
            partial_sum = 0
            delta = 0
            for n_state in range(len(world.states)):
                reward = world.reward(state, n_state, action)
                p_a_ss = world.transition_prob(state, n_state, action)
                partial_sum = partial_sum + p_a_ss * (reward + gamma*values[n_state

            delta = max(delta, abs(values[state]-partial_sum))
            values[state] = partial_sum
    return values
  # -------------------------


def improve_policy(world, policy, values, gamma=0.9):
  # -------------------------
  # Implement policy improvement and return the updated policy

  # TODO insert code here
    stability = True
    new_policy = [-1] * len(policy)

    for state in world.states:
        best_action_value, best_action = -1000000, -1

        for action in world.actions:
            current_action_value = 0

            for n_state in world.states:
                reward = world.reward(state, n_state, action)
                probability = world.transition_prob(state, n_state, action)
                current_value = probability * (reward + gamma*values[n_state])

                current_action_value = current_action_value + current_value

            if current_action_value > best_action_value:
                best_action_value = current_action_value
                best_action = action

        new_policy[state] = best_action

        if new_policy[state] != policy[state]:
            stability = False

    # Policy update
    for index in range(len(policy)):
        policy[index] = new_policy[index]

    return stability
  # -------------------------


def policy_iteration(world, gamma=0.9, theta=0.01):
  # Initialize a random policy
  policy = np.array([np.random.choice(world.actions) for s in world.states])
  print('Initial policy')
```

```
    world.print_policy(policy)
    # Initialize values to zero
    values = np.zeros_like(world.states, dtype=np.float32)

    # Run policy iteration
    stable = False
    for i in itertools.count():
      print(f'Iteration {i}')
      values = eval_policy(world, policy, values, gamma, theta)
      world.print_values(values)
      stable = improve_policy(world, policy, values, gamma)
      world.print_policy(policy)
      if stable:
        break

    return policy, values
```

```
  # Evaluate your code, please include the output in your submission
  world = GridWorld()
  policy, values = policy_iteration(world, gamma=0.5)
```

```
Initial policy
[['→' '↓' '←' '↑']
 ['↓' '←' '↓' '↓']
 ['←' '→' '→' '→']
 ['↑' '→' '↑' '←']]
Iteration 0
[[-0.984 -1.984 -1.992 -1.969]
 [-1.969 -1.984 -1.969 -1.969]
 [-1.969 -1.969 -1.969 -1.969]
 [-1.984 -1.984 -1.984 -0.992]]
[['↑' '←' '↓' '↑']
 ['↑' '↓' '↓' '↑']
 ['↑' '←' '↑' '↓']
 ['↑' '↑' '→' '↓']]
Iteration 1
[[-0.008 -1.004 -2.    -2.   ]
 [-1.004 -1.876 -2.    -2.   ]
 [-1.502 -1.751 -2.    -1.008]
 [-1.751 -1.875 -1.008 -0.008]]
[['↑' '←' '←' '↑']
 ['↑' '↑' '←' '↓']
 ['↑' '←' '↓' '↓']
 ['↑' '→' '→' '↓']]
Iteration 2
[[-0.004 -1.002 -1.501 -2.   ]
 [-1.002 -1.501 -1.75  -1.504]
 [-1.501 -1.75  -1.504 -1.004]
 [-1.75  -1.504 -1.004 -0.004]]
[['↑' '←' '←' '←']
 ['↑' '↑' '↑' '↓']
 ['↑' '↑' '↓' '↓']
 ['↑' '→' '→' '↓']]
Iteration 3
[[-0.002 -1.001 -1.5   -1.75 ]
 [-1.001 -1.5   -1.75  -1.502]
 [-1.5   -1.75  -1.502 -1.002]
 [-1.75  -1.502 -1.002 -0.002]]
[['↑' '←' '←' '←']
 ['↑' '↑' '↑' '↓']
 ['↑' '↑' '↓' '↓']
 ['↑' '→' '→' '↓']]
```

Questions 3.5 (5p)

Let's run policy iteration with $\gamma = 1$. Describe what is happening. Why is this the case? Give an exa
affect policy iteration? (max 8 sentences)

ANSWER HERE

For $\gamma = 1$, the policy evaluation becomes the expected reward that we get taking an action in a sta
get into. This translates into having a policy that simply counts how many squares away (consideri
terminal state.

The series in this case converges because after visiting a final state, the next rewards are all zeroe
transitioning from a final state to any other state is 0).

The trade off with gamma is the following: Higher values of gamma allow to create a behaviour tha
values of gamma we create an agent that cares most about the immediate rewards.

```
policy_iteration(world, gamma=1.0)
```

```
Initial policy
[['←' '↓' '←' '↑']
 ['←' '←' '↓' '→']
 ['↓' '↓' '↓' '←']
 ['←' '↓' '↓' '↓']]
Iteration 0
[[ 0. -1. -2. -1.]
 [-1. -2. -1. -1.]
 [-1. -1. -1. -2.]
 [-1. -1. -1.  0.]]
[['↑' '←' '↓' '↑']
 ['↑' '↑' '↓' '↑']
 ['↑' '↓' '↑' '↓']
 ['↑' '↑' '→' '↓']]
Iteration 1
[[ 0. -1. -2. -2.]
 [-1. -2. -2. -3.]
 [-2. -2. -3. -1.]
 [-3. -3. -1.  0.]]
[['↑' '←' '←' '↑']
 ['↑' '↑' '↑' '↓']
 ['↑' '↑' '↓' '↓']
 ['↑' '→' '→' '↓']]
Iteration 2
[[ 0. -1. -2. -3.]
 [-1. -2. -3. -2.]
 [-2. -3. -2. -1.]
 [-3. -2. -1.  0.]]
[['↑' '←' '←' '↓']
 ['↑' '↑' '↑' '↓']
 ['↑' '↑' '↓' '↓']
 ['↑' '→' '→' '↓']]
Iteration 3
[[ 0. -1. -2. -3.]
 [-1. -2. -3. -2.]
 [-2. -3. -2. -1.]
 [-3. -2. -1.  0.]]
[['↑' '←' '←' '↓']
 ['↑' '↑' '↑' '↓']
 ['↑' '↑' '↓' '↓']
 ['↑' '→' '→' '↓']]
(array([0, 2, 2, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 3, 3, 1]),
 array([ 0., -1., -2., -3., -1., -2., -3., -2., -2., -3., -2., -1., -3., -2.,
```