# A robot exploring an unknown terrain
Graded assignment 1 | DTE-2602 | H23

Almaz Ermilov
almaz.ermilov@uit.no

## Table of Contents

## Summary

As far as I understood, the most important thing in this task was to play around with and understand the parameters described below, and also to find the balance between all these parameters for this task ("Fine-Tuning"):

1. **Learning Rate (alpha)**:
- Determines how much new information overrides old information. A high learning rate means the robot learns quickly (but can potentially ignore useful previous information).
- I chose at the end ***alpha=0.9***: during the Q-learning updates, the algorithm will be strongly affected by new information, but it will not completely overwrite the old value, weighted average, with 90% of the new value and 10% of the old value (**too high value for a stable environment maybe, but that's why I don't need so many epochs for training**).

2. **Discount Factor (gamma)**:
- determines the importance of future rewards. A factor close to 1 values future rewards almost as highly as immediate rewards, favouring long-term optimal policies. A lower discount factor makes the agent short-sighted and focuses more on immediate rewards.
- I chose at the end ***gamma=0.9***: quite high, meaning that future rewards are considered almost as important as immediate rewards. This encourages the algorithm to find routes that provide high rewards in the long term, rather than just looking for immediate gains (**too high a value for a stable environment, but that's why I don't need so many epochs for training**).

3. **Exploration vs. exploitation (epsilon)**:
- Fine-tuning epsilon is about balancing exploration and exploitation. Higher epsilon encourages exploration to learn more about the environment.
- I chose ***epsilon=0.5***: With a value of 0.5, the robot chooses to explore (select a random action) half the time and exploit (select the best known action) the other half. This balance can be particularly effective because it allows the algorithm to discover new paths ("exploration") while utilising the knowledge it has already acquired ("exploitation").

4. **Number of Episodes (num_episodes)**:
- The number of episodes affects how much experience the agent gains. Too few episodes may not be enough for the agent to learn effectively, while too many episodes may be wasted or lead to over-adaptation to the environment.
- I chose at the end ***num_episodes=100 (or much less due to large alpha and gamma)***: Although more episodes generally provide better learning, even a lower number like 100 can be sufficient <u>if the learning rate (alpha) is high</u>, because the Q values are updated more aggressively.

5. **Reward**:
I chose at the end: ***self.rewards = { 'M': -300, 'W': -500, 'L': 10, 'E': 1000}***
- 'M': -300 and 'W': -500: The penalty for mountain (M) and water (W) makes these terrains much more undesirable. This causes the robot to avoid these conditions to a greater extent, making it less likely that the route will pass through them.
- 'L': 10: Keeping the regular land reward (L) at 10 provides a small positive reward that encourages the robot to stay on land whenever possible.
- 'E': 1000, End (E): Such a large positive reward gives the algorithm a strong incentive to reach the goal, making the end state very attractive. This can speed up the learning process of the path leading to the goal.

The combination of these parameters and reward settings created a learning environment where the robot could quickly learn <u>a fine balance</u> between exploring new paths and utilising familiar paths, while largely avoiding negative terrain and aiming for the target. The high learning rate allowed the robot to quickly adapt to the increased rewards and penalties, resulting in an efficient route to the target.

## Introduction

In this assignment, I create a simulation of a robot exploring an unknown terrain. The goal

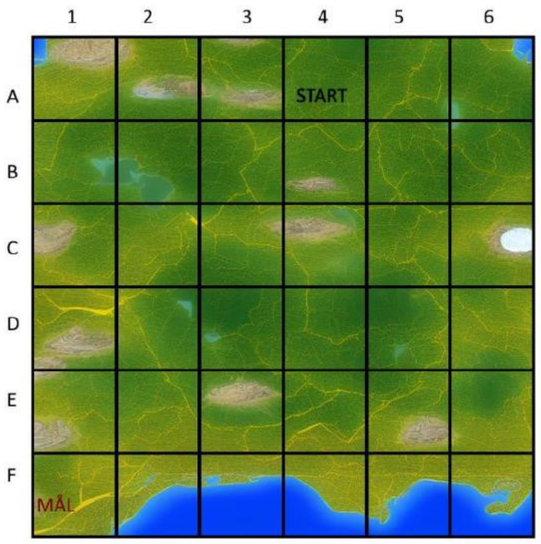
of this assignment: learn about Q-learning.
Q-learning is a model-free reinforcement learning algorithm for learning the value of an action in a particular state (Jansson & Skjelvareid, 2023)

Problem: navigate a robot through a grid-based terrain with varying levels of risk.
The assignment is designed to help us understand how we can apply Q-learning to a practical problem.

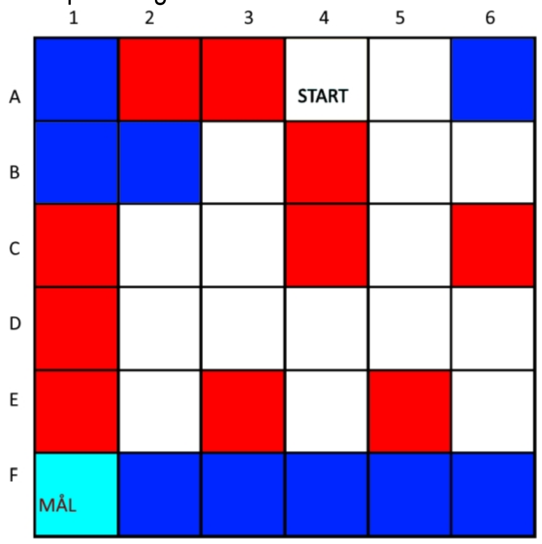
For simplicity, the terrain is divided up as a grid with 36 states. The robot will start in A4 and will try to find a safe path to F1.

About the terrain:

The terrain



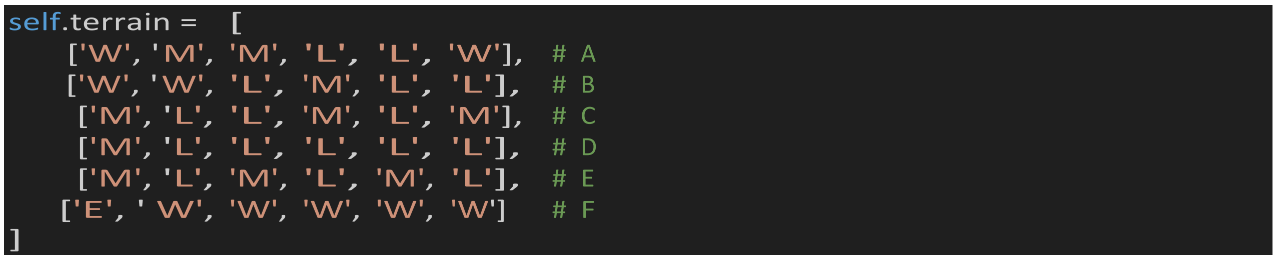
Simplified figure of the terrain



Red routes are mountains and steep terrain, and dark blue routes are water. White routes are associated with low or no risk.

The robot is controlled with an API that allows the following movements: "up", "down", "right" and "left". If the robot is in route "A4" and makes the movement "down", it ends up in route "B4".
The robot cannot move diagonally. The terrain

as an array:



```
self.terrain =  [
    ['W', 'M', 'M', 'L', 'L', 'W'],  # A
    ['W', 'W', 'L', 'M', 'L', 'L'],  # B
     ['M', 'L', 'L', 'M', 'L', 'M'],  # C
     ['M', 'L', 'L', 'L', 'L', 'L'],  # D
     ['M', 'L', 'M', 'L', 'M', 'L'],  # E
   ['E', ' W', 'W', 'W', 'W', 'W']   # F
]
```

Where:
M is red terrain (mountains and steep terrain) W is dark blue terrain (water)
L is white terrain (low or no risk) S is start
E is over

## Task description

The assignment consists of two parts to be submitted via the repository on GitHub Classroom:

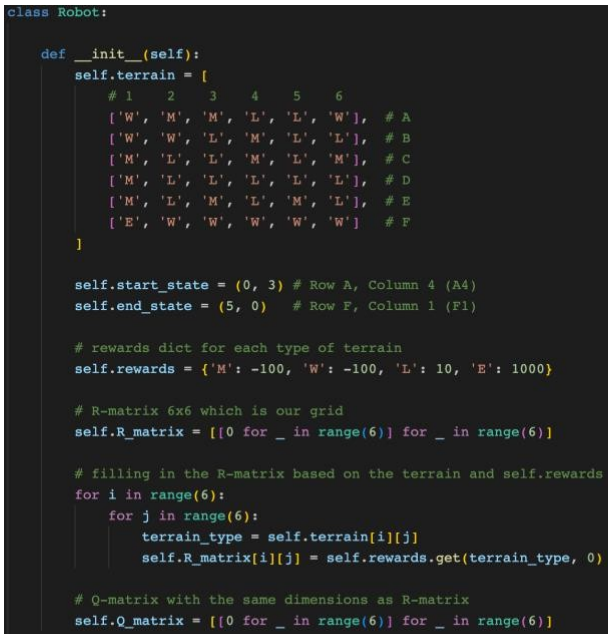
- A Python file with implementation.
- A report in the form of a PDF document (e.g. generated from MS Word or LaTeX) with a description of your own solution. The report should be between 3 and 6 pages (A4, font size 11). The report can be written in Norwegian or English.

The task is divided into subtasks:

1. Implement the "Robot" class and define an R-matrix (reward matrix) for the map provided. Make your own judgements about which values should be used in the matrix. Provide a justification in the report for the choices you make.
2. Implement the method get_next_state_mc() which randomly selects one of four possible actions for the square the robot is in (up, down, right, left). If the robot is at the edge of the grid and tries to leave the map, it will remain in the same square.
3. Implement the monte_carlo_exploration() method, which runs Monte Carlo simulation of the robot moving randomly in the map (as described in task 2). The method should have the number of simulations as an input argument. Each simulation starts with the robot in route A4 and ends when it reaches route F1. The method should calculate the total reward for each simulation (based on the matrix in task 1), taking care of the best route and the highest reward it has found. Run the simulation 100 times and describe the best route the robot finds (preferably with a figure).
4. Define a Q-matrix for the robot and implement the q_learning() method. The method should allow the robot to explore the map and simultaneously update the Q-matrix along the way. The method should have the number of episodes as an input argument. For each episode, place the robot in a random start state, which ends when it reaches F1 - the goal state. Choose one of two policy variants:
   a. Simple variation: The robot chooses 1 of 4 possible movements at random (same as Monte Carlo in task 3)
   b. More complex/efficient variant ("epsilon-greedy"): The robot chooses random movement a proportion of the time (e.g. 80% of the time), and chooses the option that works best (based on the Q-matrix) the rest of the time. Requires alternative method to find the next state (get_next_state_ge() ). The proportion of random choices ("epsilon") should be an input parameter.
5. Implement the greedy_path() method that returns the best route from start to finish (based on the "trained" Q-matrix), with the corresponding total reward.
6. Experiment by running the Q-learning programme with different numbers of episodes (and possibly different variations of the epsilon-greedy policy), and show how good a route the robot finds for each experiment. Is it able to find the optimal route? You may need to adjust the reward matrix (see task 1) to get good results. Feel free to compare with the results in task 3.

## Solution

### Task 1



```python
class Robot:

    def __init__(self):
        self.terrain = [
            # 1    2    3    4    5    6
            ['W', 'M', 'M', 'L', 'L', 'W'],  # A
            ['W', 'W', 'L', 'M', 'L', 'L'],  # B
            ['M', 'L', 'L', 'M', 'L', 'M'],  # C
            ['M', 'L', 'L', 'L', 'L', 'L'],  # D
            ['M', 'L', 'M', 'L', 'M', 'L'],  # E
            ['E', 'W', 'W', 'W', 'W', 'W']   # F
        ]

        self.start_state = (0, 3) # Row A, Column 4 (A4)
        self.end_state = (5, 0)   # Row F, Column 1 (F1)

        # rewards dict for each type of terrain
        self.rewards = {'M': -100, 'W': -100, 'L': 10, 'E': 1000}

        # R-matrix 6x6 which is our grid
        self.R_matrix = [[0 for _ in range(6)] for _ in range(6)]

        # filling in the R-matrix based on the terrain and self.rewards
        for i in range(6):
            for j in range(6):
                terrain_type = self.terrain[i][j]
                self.R_matrix[i][j] = self.rewards.get(terrain_type, 0)

        # Q-matrix with the same dimensions as R-matrix
        self.Q_matrix = [[0 for _ in range(6)] for _ in range(6)]
```

The Robot class is initialised with:

- "*terrain*" which represents the environment the robot can navigate in
- "*start_state*", which is the starting point of the robot
- "*end_state*" which is the destination the robot aims to reach
- The "*rewards*" dict defines the rewards for each terrain type.
- The "*R_matrix*" is created based on the 6x6 dimensions of the grid and contains the rewards for each cell in the grid: it is filled with rewards corresponding to the terrain type for each cell in the grid.
- "*Q_matrix*" is initialised with the same dimensions as the R-matrix and all values are set to zero. It stores the expected rewards for performing an action in a given state.

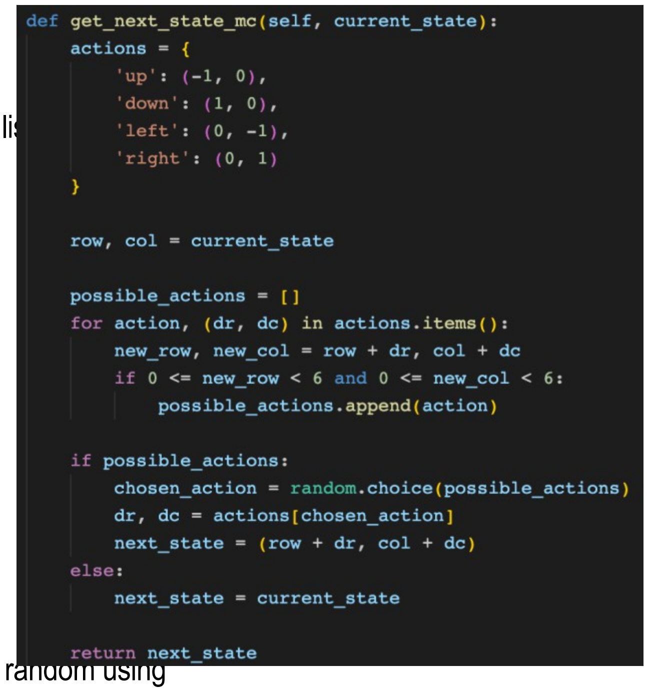
The values chosen for the rewards in the R-matrix follow a logic that is consistent with the navigation goals:

```
self.rewards = { 'M': -100, 'W': -100, 'L': 10, 'E': 1000}
```

***(let's start with that first)***

General justification for the selected values:
- "M" and "W" have the same negative value because both represent poor terrain.
- "L" has a positive value because it is good terrain and should be the preferred path.
- "E" has a higher positive value to motivate the robot to achieve this goal above all else.

### Task 2



```python
def get_next_state_mc(self, current_state):
    actions = {
        'up': (-1, 0),
        'down': (1, 0),
        'left': (0, -1),
        'right': (0, 1)
    }

    row, col = current_state

    possible_actions = []
    for action, (dr, dc) in actions.items():
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 6 and 0 <= new_col < 6:
            possible_actions.append(action)

    if possible_actions:
        chosen_action = random.choice(possible_actions)
        dr, dc = actions[chosen_action]
        next_state = (row + dr, col + dc)
    else:
        next_state = current_state

    return next_state
```

The method defines a dictionary named "*actions*" that assigns action names to corresponding changes in grid position. "*current_state*" is a tuple representing the robot's current position in the grid (row and column). "*possible_actions*" is a li to keep track of which actions can be performed from the current state without moving outside the grid.

The method then iterates over the elements in "*actions*":
- For each action, the new grid position (*new_row, new_col*) is calculated by applying the changes to the current position.
- It checks if the new position is within the grid boundaries (i.e. both the row and column indices are between 0 and 5, inclusive).
- If the new position is valid, the action is added to the list of "*possible_actions*".

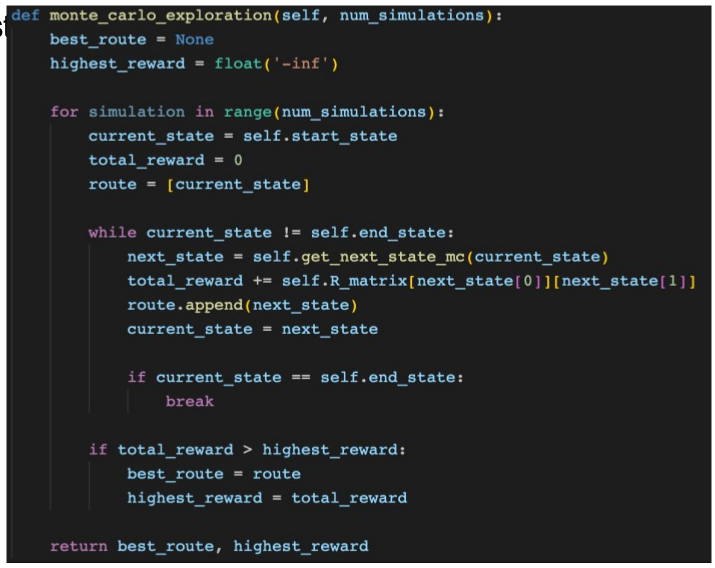
After determining the possible actions:
- If there are any possible actions, the method selects one at random using *random.choice(possible_actions)*, calculates the new state by applying the selected action's changes to the current state and returns this new state as "*next_state*".
- If there are no possible actions, "*next_state*" is set to "*current_state*" (the robot remains in the same position).

The method returns "*next_state*", which is the robot's new position after the move.

## Task 3

The Monte_carlo_exploration *method* performs the following st

- Runs multiple simulations where the robot moves randomly from the start to the end position.
- In each simulation, it tracks the total reward for the selected path.
- Compares the reward for each path to find the one with the highest reward.
- Returns the best path and total reward after all simulations have been completed.

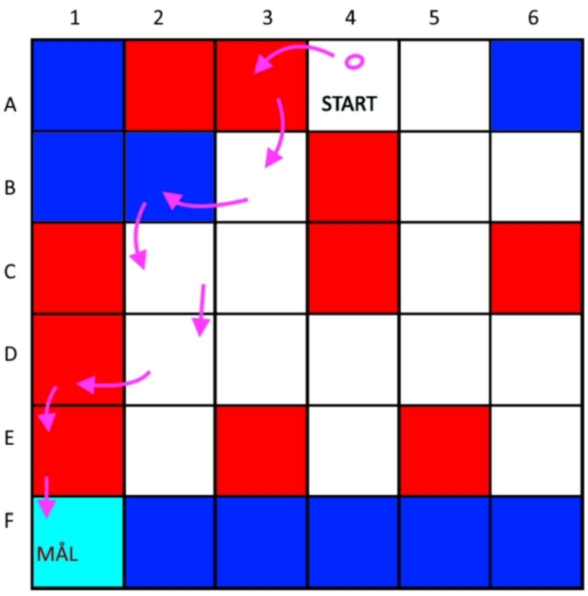


```
def monte_carlo_exploration(self, num_simulations):
    best_route = None
    highest_reward = float('-inf')

    for simulation in range(num_simulations):
        current_state = self.start_state
        total_reward = 0
        route = [current_state]

        while current_state != self.end_state:
            next_state = self.get_next_state_mc(current_state)
            total_reward += self.R_matrix[next_state[0]][next_state[1]]
            route.append(next_state)
            current_state = next_state

            if current_state == self.end_state:
                break

        if total_reward > highest_reward:
            best_route = route
            highest_reward = total_reward

    return best_route, highest_reward
```

I simulated it 100 times and got it
(with self.rewards = {'M': -200, 'W': -100, 'L': 10, 'E': 100}):

*Best Route from Monte Carlo Exploration:*
*[(0, 3), (0, 2), (1, 2), (1, 1), (2, 1), (3, 1), (3, 0), (4, 0), (5, 0)]*
*Highest Reward from Monte Carlo Exploration: -570*

So it was -200 +10 -100 +10 +10 -200 -200 -200 +100, see the figure:



## Task 4b

Changed in *init* :
- Changed the Q matrix (**36x4**, num_states x num_actions) to be updated during the Q learning process.

*q_learning*:
- Conducts episodes of exploration, updating the Q-matrix after each step based on the reward and the estimated future rewards.
- Uses an epsilon-greedy strategy where the robot chooses a random action with probability epsilon (exploration) and the best known action with probability 1-epsilon (exploitation).
- The Q values are updated using the Bellman equation, which takes into account the learning rate (alpha) and the discount factor (gamma).

*get_next_state_eg*:
- Selects the next state during the Q-learning process.
- Selects a random action with probability epsilon.
- With probability 1-epsilon, the action with the highest Q value for the current state is selected.

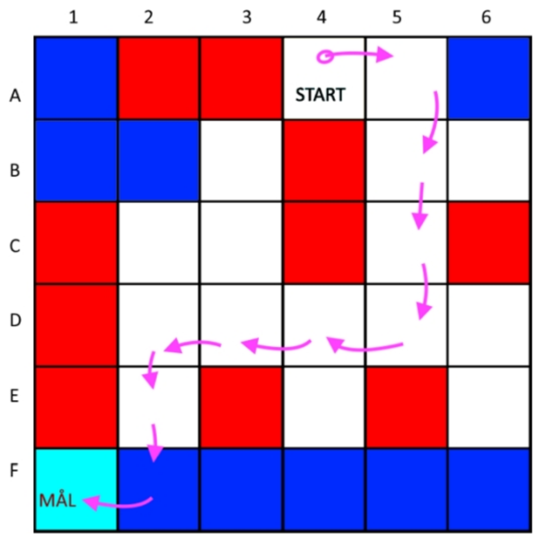
## Tasks 5 and 6

Greedy Path (*greedy_path*):
- After the Q matrix is established, this method finds the best route from the start to the end state based on the highest Q values.
- uses a greedy strategy and always chooses the action with the highest Q value for each step.

Parameters that I used (after doing lots of experiments):

```
num_episodes=100, epsilon=0.5, alpha=0.9, gamma=0.9
self.rewards = { 'M': -300, 'W': -300, 'L': 10, 'E': 1000}
```

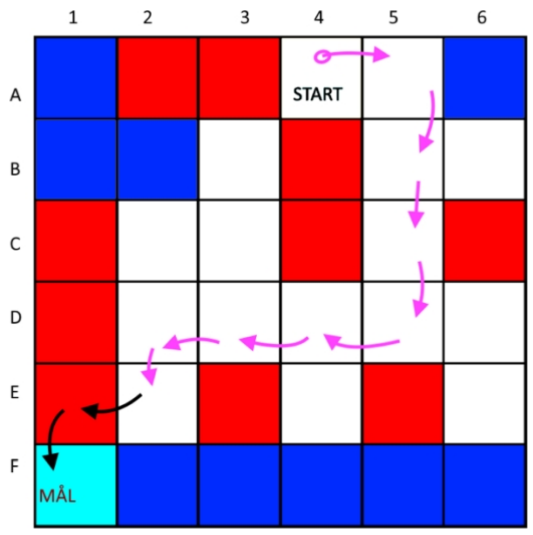
Best Route (one of the runs) that I got was:



```
Best Route from Greedy Path based on Q-Matrix:
[(0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (3, 3), (3, 2), (3, 1), (4, 1), (5, 1), (5, 0)]
Total Reward from Greedy Path: 780
```

The route is shown in the image to the right. I like this route.

There are many parameters to experiment here, it was difficult to get a balance between these parameters (epochs, epsilon, alpha, gamma, rewards dict).



I have tried many different variations and options. Mountains and water have different risks, but it may make sense to give more penalty for water than for mountainous terrain (water requires more energy).

So I changed it for water in
*self.rewards = { 'M': -300, 'W': -500, 'L': 10, 'E': 1000}*

```
Best Route from Greedy Path based on Q-Matrix:
[(0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (3, 3), (3, 2), (3, 1), (4, 1), (4, 0), (5, 0)]
Total Reward from Greedy Path: 780
```

The route is shown in the image to the right. I like this route too.

*num_episodes=100, epsilon=0.5, alpha=0.9, gamma=0.9,*
*self.rewards = {'M': -300, 'W': -500, 'L': 10, 'E': 1000}*

**The combination of these parameters and reward settings created a learning environment where the robot could quickly learn a fine balance between exploring new paths and utilising familiar paths, while largely avoiding negative terrain and aiming for the target. The high learning rate allowed the robot to quickly adapt to the increased rewards and penalties, resulting in an efficient route to the target.**

## Bibliography

Jansson, A. D., & Skjelvareid, M. (2023). *Module 5: Reinforcement learning.* From Canvas: https://uit.instructure.com/courses/30442/modules/items/885185

Amini, A. (2023). *MIT Introduction to Deep Learning 6.S191: Lecture 5* . From YouTube: https://www.youtube.com/watch?v=AhyznRSDjw8

ADL (2018, September 3). *An introduction to Q-Learning: reinforcement learning*. From freeCodeCamp: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/