Introduction to Reinforcement Learning: Q-learning

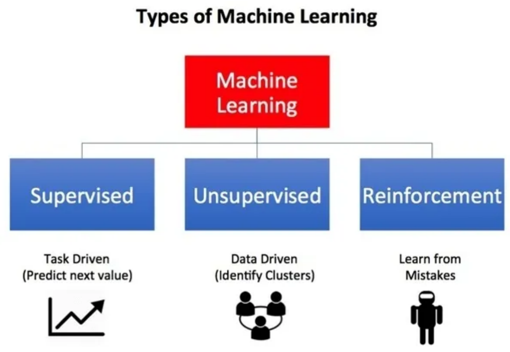
Elsa Ferreira Gomes

2024/2025

(From: https://www.kaggle.com/code/mrhippo/introduction-to-reinforcement-learning-q-learning#Q-learning)

In [1]:
```
from IPython.display import Image
Image("ML_met.png")
```

Out[1]:



Some examples of RL applications

- Self-driving vehicles (cars, helicopters)
- Make a humanoid robot walk
- Manage an investment portfolio
- Control a power station
- Recommendation system for online store
- Find pages relevant to queries
- Defeat the world champion at Backgammon, Go, Chess

  http://www.youtube.com/watch?v=eRwTbRtnT1I

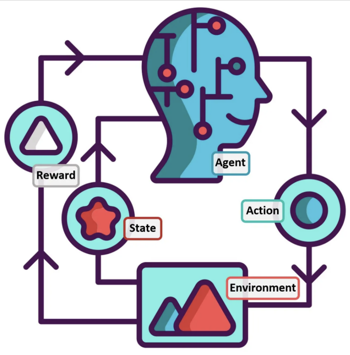
What is Reinforcement Learning?

Reinforcement learning is a machine learning training method based on

- **rewarding** desired behaviors and
- **punishing** undesired ones. In general, a reinforcement learning **agent** (the entity being trained) is able to perceive and interpret its **environment**, take **actions** and learn through trial and error.

In [2]:
```
from IPython.display import Image
Image("RL_imagem.png", width=500)
```

Out[2]:



Reinforcement Learning (image by Flat-Icons on IconScout under license to Chris Mahoney)

Reinforcement Learning (in 1980s, was developed for simulating the behaviors of animals). In order to do that RL uses a reward and punishment system. In nature all animals act like this, they search for food and run away from predators, food means reward (+x points) and wounds, diseases and effects of starvation means punishment (-y points).

Reinforcement Learning

Some key terms that describe the basic elements of an RL problem:

- **Environment**
  - Physical world in which the agent operates
- **State**
  - Current situation of the agent
- **Reward**
  - Feedback from the environment
- **Policy**
  - Method to map agent's state to actions
- **Value**
  - Future reward that an agent would receive by taking an action in a particular state

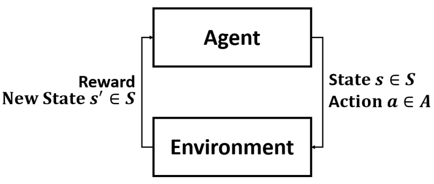
Q-learning

- Q-learning (QL) is the most basic method of RL. Its logic is very simple:
  - we have an **environment** and an **agent**.
  - in environment we have **states** which can be represented as a **S** cluster
  - agent can make actions which can be represented as an **A** cluster.
  - as a consequence of taking an action and being in a state the agent gains a **reward** from the environment.
- The reward can be a negative or positive value.

- By taking reward values from environment our agent will learn what it should do.
- The **agent wants to maximize the reward** since it is a rational being.
  - In order to do that there is a formula for maximizing the reward called **Bellman Equation**.

In [3]:
```
from IPython.display import Image
Image("QLearning.png", width=500)
```

Out[3]:



Bellman Equation

$$V(s) = \max_a(R(s,a) + \gamma V(s')) \tag{1}$$

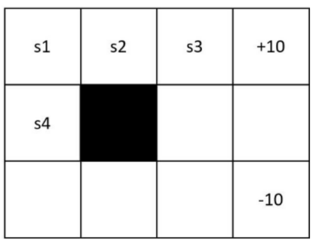
In Bellman Equation we have:

- a **value of s (V(s))** and our agent is trying to maximize it.
- In this equation:
  - **R(s, a)**: means the reward that agent will get by taking action a in state s
  - γ: is the **discount rate**, $0 <= \gamma < 1$ (most of the time its value is 0.9).
  - **V(s')** is the reward in **next state (s')**.

Example: Bellman equation

- Our agent starts at s1. And chooses to go right, lets calculate the reward:
  - R(s, a) = R(s1, right); there is no reward for this action so it is 0.
  - Lets pick gamma as 0.9. V(s') = V(s2) and all states' initial reward is 0 so it is 0 too.
  - V(s1) = 0 + 0.9·0 = 0.
- Now we are at s2.
  - We will do the same calculations as before-> V(s2)=0.
- Now s3:
  - This time we get: R(s, a) = +10 because in the next state we have a reward of +10.
  - Our discount rate is the same as before its 0.9 and V(s') = 0, so our equation is:
  - V(s3) = 10 + 0.9x0 = 10.
- We have reached one of the rewards and our game will start again so our agent is at s1.
  - Supose our agent chose the s2 again our reward will be 0.
  - Now our agent knows the value of s3 it is +10.
  - So it will go to the s3 and end the game by going to +10 reward.
- Our agent starts at s1 again and s2's value is 9 because s3 is 10 if we calculate it:
  - V(s2) = R(s2, right) + 0.9·V(s3) = 0 + 0.9 x 10 = 9.
  - V(s1) = R(s1, right) + 0.9·V(s2) = 0 + 0.9 x 9 = 8.1.

We can use a table that contains states in vertical axis and actions in the horizontal axis. Our agent will look at the table and find the maximum value in its state's row and take the action that coincides with that value.



s1	s2	s3	+10
s4	■		
			-10

Example: Implementing Bellman Equation

- Lets implement the Bellman Equation in a environment.

- Since it is the first example and Bellman Equation is very simple for complex environmnets we will use a ver simple environment.
  - Our environnet is about a squarrel on an 1 dimensional plane searching for a nut.
  - There will be 12 spots in environment and the squarrel will try to find the nut by going left or right (there is no option like no move it has to move).
  - At the 2 ends of the environment there will be predators.
- **Rules**:

  - If the squarrel goes to one of the 2 end it will gain -5 points;
  - If squarrel makes a move it will take -1 points (because we want to finish this as quickly as we can so if our squarrel makes extra moves it means it is a unsuccesful agent) (normally this concept is called living penalty, we will discuss about it later)
  - If squarrel gets to the nut it will gain +10 points and the episode will end

In [4]:

```
import numpy as np # linear algebra and matrix operations
import pandas as pd # data science and tables
import time # for sleep method
import random # random events and probability
import matplotlib.pyplot as plt # data visualization
import warnings
warnings.filterwarnings('ignore')
```

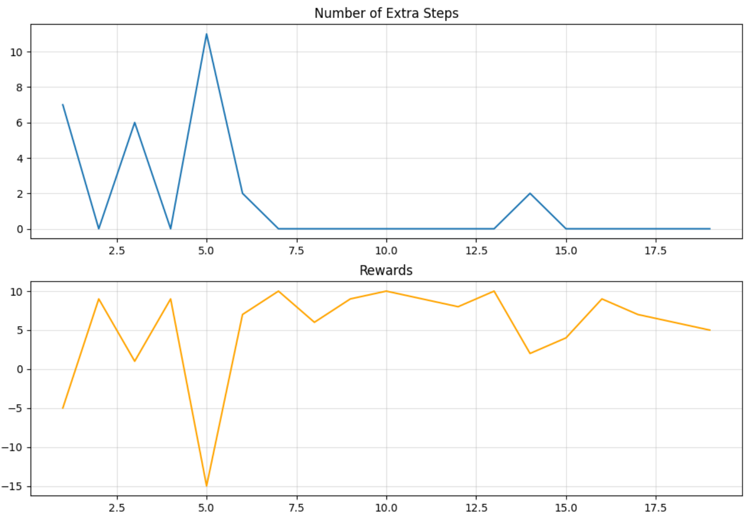
In [5]:

```
# squarrel_pos = state
# variables
action_space = 2
num_states = 12
reward_table = np.zeros([num_states, action_space])
gamma = 0.9 #discount rate
num_of_eps = 20

# environment
env = np.array([-5, -1, -1, -1, 10, -1, -1, -1, -1, -1, -1, -5])
nut_pos = 4
end_pos_1 = 0
end_pos_2 = 11
sleeping = False

# store scores
scores = []

# store number of extra steps
number_of_extra_steps = []

# define step function
def step(pos, value):

        # determine action
        if value == 0:
            next_state = pos - 1 #left

        if value == 1:
            next_state = pos + 1 #right

        # verify action
        if next_state > end_pos_2:
            next_state = end_pos_2

        if next_state < end_pos_1:
            next_state = end_pos_1

        # reward
        reward = env[next_state]

        # return
        return next_state, reward

# training loop
for ep in range(1, num_of_eps):

    # print episode
    ###print("Episode: ", ep)

    # recreate environment
    env = np.array([-5, -1, -1, -1, 10, -1, -1, -1, -1, -1, -1, -5])

    # choose a random position for squarrel
    squarrel_pos = random.choice(range(end_pos_1+1, end_pos_2+1))

    # be sure that squarrel is not on the nut
    while squarrel_pos == nut_pos:
        squarrel_pos = random.choice(range(1, 10))

        if squarrel_pos != nut_pos:
            break

    # find the distance between nut and squarrel
    distance = np.absolute(squarrel_pos - nut_pos)

    # print squarrel position
    ###print("Squarrel Position: ", squarrel_pos)

    # place a zero to represent the squarrel
    env[squarrel_pos] = 0

    # print first state of the environment
    ###print("Environment:", env, "\n----------------------------------------------------")

    # store rewards
    rewards = []

    number_of_steps = 0

    # game loop
    while squarrel_pos != nut_pos:

        # find which number is squarrel on
        if (squarrel_pos == end_pos_1) or (squarrel_pos == end_pos_2):
            last_num = -5

        if (squarrel_pos != end_pos_1) and (squarrel_pos != end_pos_2):
            last_num = -1

        # place back the number that squarrel was blocking
        env[squarrel_pos] = last_num

        # max action in the table
        action = np.argmax(reward_table[squarrel_pos])

        # take action
        next_state, reward = step(squarrel_pos, action)

        # append reward to rewards list
        rewards.append(reward)

        # bounds of the environment
        if squarrel_pos < end_pos_1:
            squarrel_pos = end_pos_1

        if squarrel_pos > end_pos_2:
            squarrel_pos = end_pos_1

        # update reward table with bellman equation
        reward_table[squarrel_pos][action] = reward + gamma * max(reward_table[next_state])

        # update state
        squarrel_pos = next_state

        # show squarrel's position
        env[squarrel_pos] = 0

        # show environment
        ###print(env)

        # pause for 1.2 seconds so we can see what is happening
        if sleeping:
            time.sleep(1.2)

        # increase number_of_steps by 1
        number_of_steps += 1

    # find total score by summing the rewards
    total_score = sum(rewards)

    # append total score to scores list
    scores.append(total_score)

    # print final score of the episode
    ###print("Score:", total_score)

    # find the number of extra steps
    extra_steps = np.absolute(number_of_steps - distance)

    # append number of extra steps to list
    number_of_extra_steps.append(extra_steps)

    # print how many extra steps the agent has took
    ###print("Number of Extra Steps:", extra_steps)

average_score = np.mean(scores)
###print("Training Completed. \nAverage Score: {}".format(average_score))
```

Every time the environment changes the maximum total score agent can make changes too. So it might be a better choice to look at extra steps our agent has took to see its improvement.

In [6]:

```
fig, ax = plt.subplots(2,1, figsize = (12,8))

ax[0].plot(range(1, num_of_eps), number_of_extra_steps)
ax[0].set_title("Number of Extra Steps")
ax[0].grid(True, alpha = 0.4)

ax[1].plot(range(1, num_of_eps), scores, c = "orange")
ax[1].set_title("Rewards")
plt.grid(True, alpha = 0.4)

plt.show()
```



In [7]:

```
table = pd.DataFrame(reward_table)
table
```

Out[7]:

	0	1
0	-5.00000	-1.0000
1	-5.00000	6.2000
2	-1.90000	8.0000
3	-1.00000	10.0000
4	0.00000	0.0000
5	10.00000	0.0000
6	8.00000	-1.0000
7	6.20000	-2.7100
8	4.58000	-2.7100
9	3.12200	-4.0951
10	1.80980	-5.0000
11	-4.68559	-5.0000

Epsilon Greedy

- We learned how to use Bellman Equation but if we had a more complex environment we might meet with a problem:
    - Our agent could find a path and think it is the most optimal way to solve this problem
        - but that can be wrong.
    - There might better ways to solve this problem
        - but our agent could stuck in its own optimal road and cannot find better ones.
    - But if we try to explore every time we might lose time with this process
        - so we have to balance it.
- In order to do that we use **Epsilon Greed**.
    - In this concept we have a hyperparameter represented as ε (epsilon).
        - We define this hyperparameter and then create a random value between 0 and 1 (you can use different numbers if you want to) and see
        - If it is bigger than ε, our agent will look its reward table and take the most efficent action (exploitation).
        - If is less than ε then our agent will take a random action (explore).

Implementing Epsilon Greedy

- To see the benefits of epsilon greedy we will use a more complex environment.
    - Again our squarrel (agent) tries to get to the nut but this time in 2 dimensions and there are 3 predators.
    - We have 2 dimensions so our action space is 4 (up, down, left, right) and we have 25 different states (5 columns 5 rows).
    - We have to give a number to every state;
        - In order to do that we can use this formula: State's Number = Row's Number · 5 + Column's Number.
        - Rules are the same as the first environment.

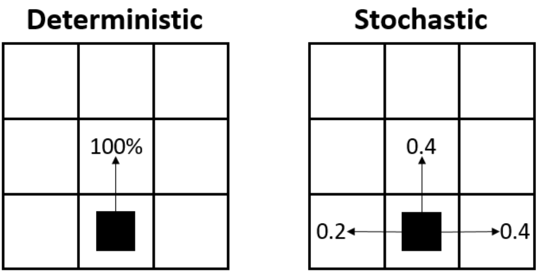
Q-learning

We learned Bellman Equation but there is a problem with it.

- Bellman Equation is only defined for **Deterministic environments** but in real life we might have to deal with **Stochastic environments** too.

Deterministic vs Stochastic environments

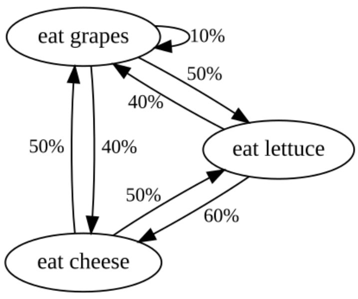
- We can decide to take an action we do it with 100% chance.
- However in Stochastic Environments we decide to take an action but the probability of doing it successfully is not certain.
    - For example if your agent decides to go left it might go other directions too (like 20% left, 30% right, 10% up, 40% down). To deal with stochastic environments we have to learn Markov Decision Process.



Markov Chains

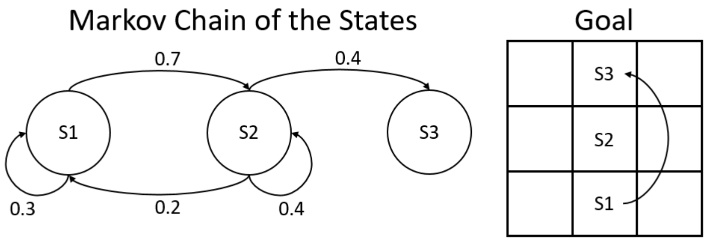
Markov chain is a set of elements that are bounded together with probability.

- For example if we choose eat grapes we have a 50% probability of eating lettuce, 40% probability of eating cheese and 10% probability of eating grapes again in our new meal.
- Remember since this is a probability its sum must be 1. We can do the same thing for lettuce:
    - We have 40% probability of eating grapes, 60% probability of eating cheese and 0% probability of eating lettuce in the next meal (for more information on markov chains: Wikipedia).
        - If we adapt the markov chains to environments we get a stochastic environment.



Markov Chains

- In a stochastic environment if we want to go a state from another state we can think this process as a markov chain.

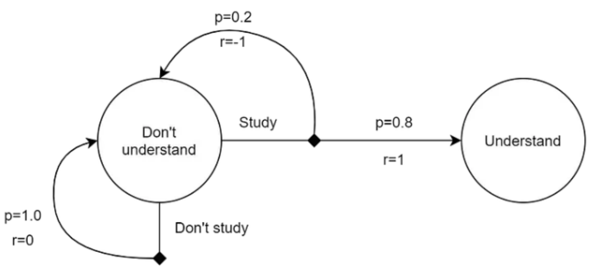


Markov Decision Process (MDP)

A Markov Decision Process (MDP) is an extension of the Markov chain and it is used to model more complex environments.

- In this extension, we add the possibility to make a choice at every state which is called an action.
- We also add a reward which is a feedback from the environment for going from one state to another through an action.

Example:



- In the initial state don't understand, where we have two possible actions: study and don't study.
- For the study action, we may end up in different states according to a probabilistic rule.
- This is what we call a stochastic environment (random), in the sense that for one same action taken in the same state, we might have different results (understand and don't understand).

In reinforcement learning, this is how we model a game or environment, and our goal will be to maximize the reward we get from that environment.

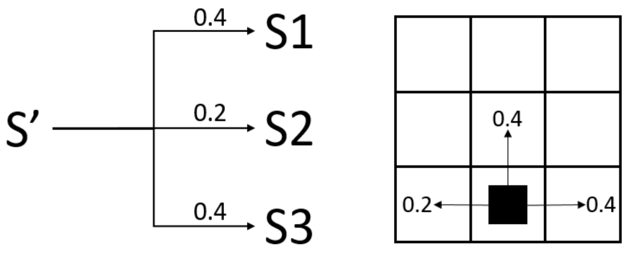
Markov Decision Process (MDP)

In a Markov Decision Process (MDP) we think our stochastic environment as a Markov Chain.

- So every state is bounded together with probabilistic values to other states just like a Markov Chain.
- In this situation our Bellman Equation will not work properly because now we have probabilistic values for s'.
- To get a proper equation for these environments we have to upgrade our Bellman Equation as:

$$V(s) = max_a(R(s,a) + \gamma \sum(P(s,a,s')V(s'))) \tag{2}$$

In this equation we do not know what is P(s, a, s'). Since we are in a stochastic environment our next state is uncertain but there are probabilities for each of them.



In order to find V(s') we have to multiply it with every probability and then add them together.

$0.4 \cdot V(s') + 0.2 \cdot V(s') + 0.4 \cdot V(s')$

Q-functions (Q-learning)

- In deterministic environments we calculate the value of a state.
- In stochastic environments the value of a state is not that significant.
    - Instead of that we have to find the value of an action.

- But we cannot say just V(a), because if we want to go up but if we are not in the same state as the state we were before it would be different things (up in state 1 ≠ up in state 3).
  - So we have to use an expression that contains both of these informations. We do that by using Q-function.

$$Q(s,a) = R(s,a) + \gamma \sum_{s'}(P(s,a,s')max_{a'}(Q(s',a'))) \tag{3}$$

- To understand the Q-function we can look at these equations:

$$max_a(Q(s,a)) = max_a(R(s,a) + \gamma \sum_{s'}(P(s,a,s')V(s'))) \tag{4}$$

$$Q(s,a) = R(s,a) + \gamma \sum(P(s,a,s')V(s')) \tag{5}$$

$$V(s') = max_a'(Q(s',a')) \tag{6}$$

Note: In Q-learning we call the reward table as Q-table.

Temporal Difference (Q-learning)

Normally we know the Q-function as:

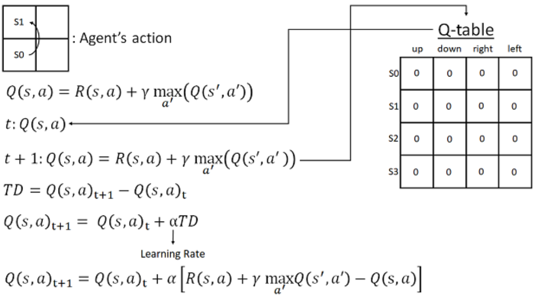
$$Q(s,a) = R(s,a) + \gamma \sum_{s'}(P(s,a,s')max_{a'}(Q(s',a'))) \tag{7}$$

But the part with sigma can be confusing in this subject so, for simplicity, we will write the function without it (but remember it is still in the equation just we do not write it).

Now we are trying to find the best **action**. In order to do that we have to use our Q-table. We are in the t and agent takes a value for its action by looking at he Q-table. When it takes its action our next moment will be t+1 and we will update our Q-table.

Now we will start to calculate **Temporal Difference (TD)**.

- TD equals to q-value in t+1 minus q-value in t becuase there is a time difference so these q-values are not the same.
- We find the q-value in t+1 by reorganizing the equation and we find its q-value in t+1 = (q-value in t) + (TD in t+1).
- But we will add another expression called learning rate (α) and we multiply TD with learning rate. Learning rate changes between 0 and 1. And finally we reach our last equation in the figure, this is Q-function.



$$Q(s,a) = R(s,a) + \gamma \max_{a'}(Q(s',a'))$$

$$t: Q(s,a)$$

$$t+1: Q(s,a) = R(s,a) + \gamma \max_{a'}(Q(s',a'))$$

$$TD = Q(s,a)_{t+1} - Q(s,a)_t$$

$$Q(s,a)_{t+1} = Q(s,a)_t + \alpha TD$$

$$Q(s,a)_{t+1} = Q(s,a)_t + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

	up	down	right	left
S0	0	0	0	0
S1	0	0	0	0
S2	0	0	0	0
S3	0	0	0	0

- Our final equation:

$$Q(s,a)_{t+1} = Q(s,a)_t + \alpha(R(s,a) + \gamma max_{a'}(Q(s',a') - Q(s,a)) \tag{8}$$

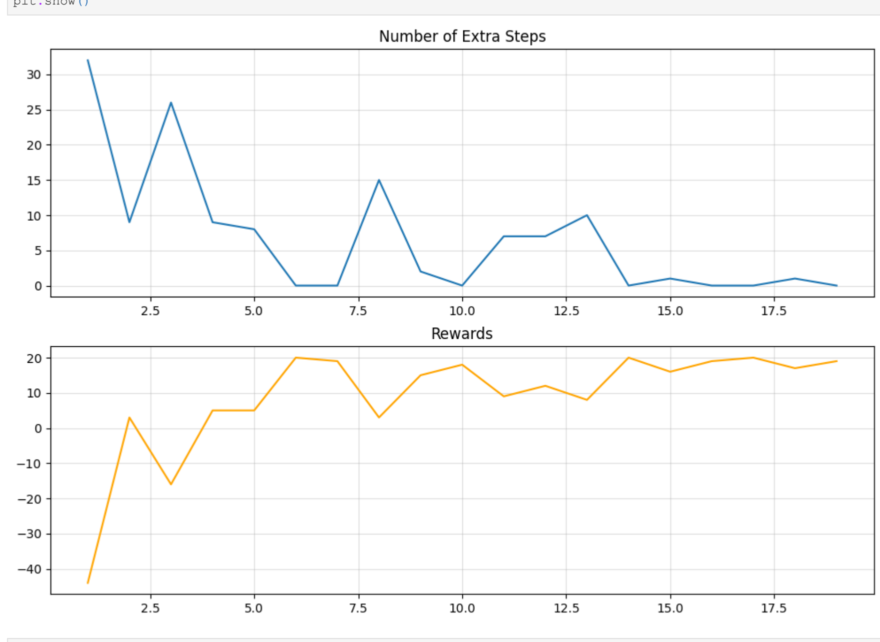
Implementing Q-function

We learned Q-learning now we can implement it.

- This time we will not make our environment ourself we will use the FrozenLake environment of OpenAI Gym.
- In this environment our agent starts at a state and tries to get to goal. But there are holes and surface is slippery (that means our agent's actions' consequences are not certain) so this is a stochastic environment.
- Rules:
  - 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.
  - This environment does not use a living penalty system.
  - Environment has 4 rows and 4 columns.
  - There are 16 different states (4 rows x 4 columns).
  - There are 4 different actions (up, down, left, right).

In [8]:
```python
# parameters
action_space = 4
num_states = 25
reward_table = np.zeros([num_states, action_space])
gamma = 0.9 #discount rate
num_of_eps = 20
epsilon = 0.1

# environment
"""
-5, -1, -1, -1, -1
-1, -1, +20, -1, -1
-1, -1, -1, -5, -1
-1, -5, -1, -1, -1
-1, -1, -1, -1, -1

"""
env = np.array([[-5,-1,-1,-1,-1],
                [-1,-1,+20,-1,-1],
                [-1,-1,-1,-5,-1],
                [-1,-5,-1,-1,-1],
                [-1,-1,-1,-1,-1]])

# parameters
nut_pos = [1, 2]
danger_1_pos = [0,0]
danger_2_pos = [2,3]
danger_3_pos = [3,1]

# environment's bounds
end_column_1 = 0
end_column_2 = 4
end_row_1 = 0
end_row_2 = 4

sleeping = False

# scores
scores = []

number_of_extra_steps = []

# define step function
def step(squarrel_pos_row, squarrel_pos_column, action):

    if action == 0:
        new_pos = [squarrel_pos_row, squarrel_pos_column - 1] # left

    if action == 1:
        new_pos = [squarrel_pos_row, squarrel_pos_column + 1] # right

    if action == 2:
        new_pos = [squarrel_pos_row - 1, squarrel_pos_column] # up

    if action == 3:
        new_pos = [squarrel_pos_row + 1, squarrel_pos_column] # down

    # verify action
    if new_pos[0] < end_row_1:
        new_pos[0] = end_row_1

    if new_pos[0] > end_row_2:
        new_pos[0] = end_row_2

    if new_pos[1] < end_column_1:
        new_pos[1] = end_column_1

    if new_pos[1] > end_column_2:
        new_pos[1] = end_column_2

    # define reward
    reward = env[new_pos[0], new_pos[1]]

    # define next_state
    next_state = new_pos[0]*5 + new_pos[1]

    return next_state, reward, new_pos

# define random action
def random_action():

    # choose a random action
    return random.choice(range(0,4))

# training loop
for ep in range(1, num_of_eps):

    # print episode's number
    ###print("Episode: ", ep)

    # redefine environment
    env = np.array([[-5,-1,-1,-1,-1],
                    [-1,-1,+20,-1,-1],
                    [-1,-1,-1,-5,-1],
                    [-1,-5,-1,-1,-1],
                    [-1,-1,-1,-1,-1]])

    # choose a position for squarrel
    squarrel_pos_column = random.choice(range(end_column_1, end_column_2+1))
    squarrel_pos_row = random.choice(range(end_row_1, end_row_2+1))

    # verify squarrel's position: it should not be on the nut and predators (danger)
    while ([squarrel_pos_row, squarrel_pos_column] == nut_pos) or ([squarrel_pos_row, squarrel_pos_column] == d

        # choose new positions for the squarrel
        squarrel_pos_column = random.choice(range(end_column_1, end_column_2+1))
        squarrel_pos_row = random.choice(range(end_row_1, end_row_2+1))

        # if squarrel's position is decent break the loop
        if (squarrel_pos != nut_pos) and (squarrel_pos != danger_1_pos) and (squarrel_pos != danger_2_pos) and
            break

    squarrel_pos = [squarrel_pos_row, squarrel_pos_column]

    # find distance between squarrel and nut
    distance = np.absolute(squarrel_pos[0] - nut_pos[0]) + np.absolute(squarrel_pos[1] - nut_pos[1])

    ###print("Squarrel Position: ", squarrel_pos)

    # show squarrel's position
    env[squarrel_pos[0], squarrel_pos[1]] = 0

    ###print("Environment:\n", env, "\n---------------------------------------")

    # store rewards
    rewards = []

    # number of total steps that squarrel has took
    number_of_steps = 0

    # game loop
    while squarrel_pos != nut_pos:

        # define state
        state = squarrel_pos[0]*5 + squarrel_pos[1]

        # find which number is squarrel on
        if (squarrel_pos == danger_1_pos) or (squarrel_pos == danger_2_pos) or (squarrel_pos == danger_3_pos):
            last_num = -5

        if (squarrel_pos != danger_1_pos) and (squarrel_pos != danger_2_pos) and (squarrel_pos != danger_3_pos)
            last_num = -1

        # place back the number that squarrel was blocking
        env[squarrel_pos[0], squarrel_pos[1]] = last_num

        # epsilon greedy
        if np.random.uniform() < epsilon:
            action = random_action()
            ###print("Random Action:")

        else:
            # max action in the table
            action = np.argmax(reward_table[state])

        #take action
        next_state, reward, squarrel_pos = step(squarrel_pos[0], squarrel_pos[1], action)

        # store rewards
        rewards.append(reward)

        # verify squarrel's position (bounds of the environment)
        if squarrel_pos[0] < end_row_1:
            squarrel_pos[0] = end_row_1

        if squarrel_pos[0] > end_row_2:
            squarrel_pos[0] = end_row_2

        if squarrel_pos[1] < end_column_1:
            squarrel_pos[1] = end_column_1

        if squarrel_pos[1] > end_column_2:
            squarrel_pos[1] = end_column_2

        # bellman equation
        reward_table[state][action] = reward + gamma * max(reward_table[next_state])

        # show squarrel's position
        env[squarrel_pos[0], squarrel_pos[1]] = 0

        # show environment
        ###print(env)

        if sleeping:
            time.sleep(1.2)

        number_of_steps += 1

    # calculate total score for the episode
    total_score = sum(rewards)

    scores.append(total_score)

    print("Score:", total_score)

    # find extra steps
    extra_steps = np.absolute(number_of_steps - distance)

    number_of_extra_steps.append(extra_steps)

    ###print("Number of Extra Steps:", extra_steps)
```

[illegible]

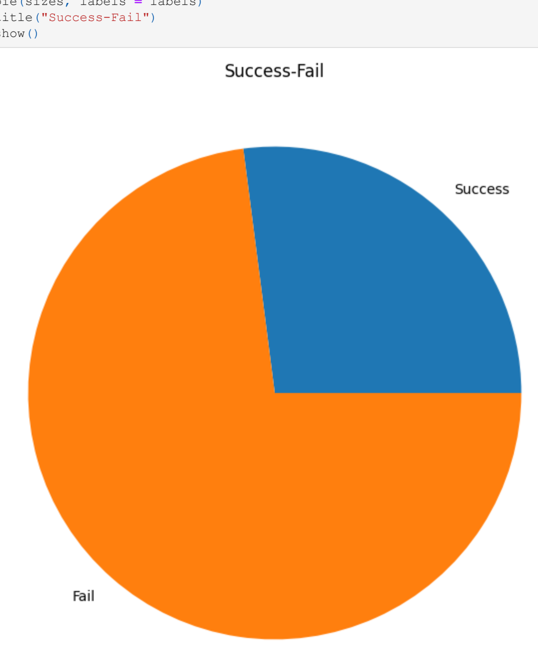


[illegible]

	0	1	2	3
0–12	[illegible]	[illegible]	[illegible]	[illegible]
13	-1.000	-1.00	17.000	0.00
14	-5.000	-2.71	14.300	-1.90
15	-3.439	-5.00	-2.710	-2.71
16	-1.900	-1.00	-1.000	-1.90
17	-5.000	-1.00	17.000	-1.90
18	14.300	-1.90	-5.000	-1.90
19	-2.710	-1.90	-1.900	-1.90
20	-2.710	-2.71	-3.439	-1.90
21	-2.710	11.87	-5.000	-1.90
22	-1.900	-1.90	14.300	-1.00
23	-1.900	-1.90	-1.900	-1.00
24	-1.900	-1.90	-1.900	-1.00

In [11]:
```python
import gym #OpenAI Gym
new_step_api=True
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

env = gym.make("FrozenLake-v1").env # make frozenlake environment
#env = gym.make("FrozenLake-v1").env

q_table = np.zeros([env.observation_space.n, env.action_space.n])
gamma = 0.9 #discount rate
num_of_eps = 75000
epsilon = 0.1
alpha = 0.1 # learning rate (most of the time 0.1 is a good number to start)

# scores
scores = []

success = []
fail = []

# training loop
for ep in range(1, num_of_eps):

    # reset the environment
    ###state = env.reset()
    state = env.reset()

    rewards = []

    # game loop
    while True:

        # explore-exploit (%90 exploit, %10 explore)
        if np.random.uniform() < epsilon:
            action = env.action_space.sample()
        else:
            # max action in the table
            action = np.argmax(q_table[state])

        #print("args  ")
        #print(env.step(action))


        #print("args")
        #print(_)

        #next_state, reward, done, _ = env.step(action) # done = if agent falls on a hole or gets to the goal t
        #next_state, reward, done, truncated, info = env.step(action)
        next_state, reward, done, truncated = env.step(action)

        #q-function
        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])

        next_value = (1-alpha)*old_value + alpha*(reward + gamma*next_max)

        q_table[state, action] = next_value

        # update state
        state = next_state

        rewards.append(reward)

        if done:
            break

    scores.append(sum(rewards))

    # print episode and score
    ###if ep%1000 == 0:
        ###print("Episode: ", ep)
        ###print("Score:", sum(rewards))

    # find number of successful and failed episodes
    if sum(rewards) == 1:
        success.append(True)
    else:
        fail.append(True)

###print("Training Completed \nSuccess: {}\nFail: {}".format(len(success), len(fail)))
```

In [12]:
```python
labels = ["Success", "Fail"]
sizes = [len(success), len(fail)]
fig = plt.figure(figsize = (11,8))
plt.pie(sizes, labels = labels)
plt.title("Success-Fail")
plt.show()
```



In [13]:
```python
table = pd.DataFrame(q_table)
table
```

Out[13]:

	0	1	2	3
0	0.067257	0.061857	0.062108	0.057859
1	0.051199	0.030948	0.042671	0.057917
2	0.068442	0.067430	0.067650	0.061144
3	0.031129	0.041788	0.025430	0.056747
4	0.085753	0.057498	0.054710	0.055688
5	0.000000	0.000000	0.000000	0.000000
6	0.064654	0.058604	0.108401	0.033553
7	0.000000	0.000000	0.000000	0.000000
8	0.083996	0.090327	0.098446	0.111964
9	0.145373	0.220108	0.173158	0.107390
10	0.206178	0.242441	0.145265	0.100878
11	0.000000	0.000000	0.000000	0.000000
12	0.000000	0.000000	0.000000	0.000000
13	0.152000	0.218759	0.285454	0.307569
14	0.413852	0.621143	0.488860	0.477908
15	0.000000	0.000000	0.000000	0.000000

Conclusion

- In this notebook we learned the basics of Q-learning and algorithms.
- If you want to learn more about reinforcement learning you can read this book: https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf

Equations

- Bellman Equation
  - $V(s) = max_a(R(s,a) + \gamma.V(s'))$
- Markov Decision Process (MDP)
  - $V(s) = max_a(R(s,a) + \gamma \sum_{s'}(P(s,a,s')V(s')))$
- Q-function
  - $Q(s,a) = R(s,a) + \gamma \sum_{s'}(P(s,a,s')max_{a'}(Q(s',a')))$
- Q-function final equation
  - $Q(s,a)_{t+1} = Q(s,a)_t + \alpha(R(s,a) + \gamma max_{a'}(Q(s',a') - Q(s,a))$

References:

- https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf
- https://towardsdatascience.com/math-of-q-learning-python-code-5dcbdc49b6f6

In [ ]: