Machine Learning using Python : Basic Implementation of Reinforcement Learning

Neelotpal Chakraborty

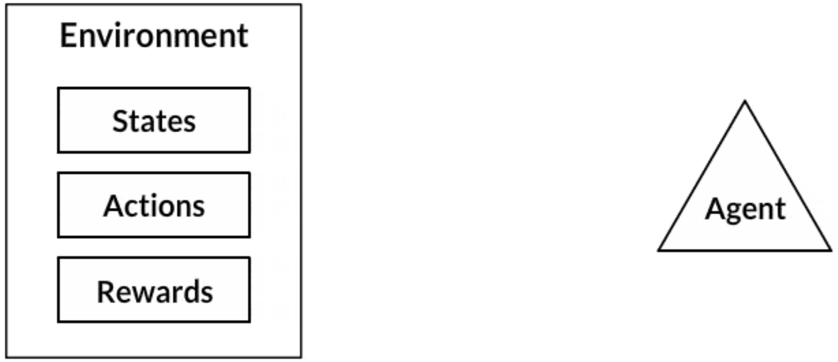
Department of Computer Science and Engineering

Jadavpur University

Reinforcement Learning

A Basic Implementation

Reinforcement Learning – A Basic Implementation



Reinforcement Learning is the science of making optimal decisions using experiences. Breaking it down, the process of Reinforcement Learning involves these simple steps:

1. Observation of the environment
2. Deciding how to act using some strategy
3. Acting accordingly
4. Receiving a reward or penalty
5. Learning from the experiences and refining our strategy
6. Iterate until an optimal strategy is found

Installation of OpenAI GYM

```
C:\Users\admin>py -m pip install gym
Collecting gym
  Downloading gym-0.21.0.tar.gz (1.5 MB)
     |████████████████████████████████| 1.5 MB 467 kB/s
Requirement already satisfied: numpy>=1.18.0 in c:\users\admin\appd
(from gym) (1.19.5)
Collecting cloudpickle>=1.2.0
  Downloading cloudpickle-2.0.0-py3-none-any.whl (25 kB)
Building wheels for collected packages: gym
  Building wheel for gym (setup.py) ... done
  Created wheel for gym: filename=gym-0.21.0-py3-none-any.whl size=
30a3b0f1f2c500586b01a9c133
  Stored in directory: c:\users\admin\appdata\local\pip\cache\wheel
36dbf6d5
Successfully built gym
Installing collected packages: cloudpickle, gym
Successfully installed cloudpickle-2.0.0 gym-0.21.0
WARNING: You are using pip version 21.1.3; however, version 21.3.1
You should consider upgrading via the 'C:\Users\admin\AppData\Local
-upgrade pip' command.

C:\Users\admin>
```

Installation of OpenAI GYM

OR better, install all the gym environments

```
C:\Users\admin>py -m pip install gym[all]
Requirement already satisfied: gym[all] in c:\users\admin\appdata\local\p
.0)
Requirement already satisfied: numpy>=1.18.0 in c:\users\admin\appdata\lo
(from gym[all]) (1.19.5)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\users\admin\appda
ages (from gym[all]) (2.0.0)
Collecting lz4>=3.1.0
  Downloading lz4-3.1.3-cp39-cp39-win_amd64.whl (192 kB)
     |████████████████████████████████| 192 kB 1.6 MB/s
Collecting box2d-py==2.3.5
  Downloading box2d-py-2.3.5.tar.gz (374 kB)
     |████████████████████████████████| 374 kB 1.7 MB/s
Collecting pyglet>=1.4.0
  Downloading pyglet-1.5.21-py3-none-any.whl (1.1 MB)
     |████████████████████████████████| 1.1 MB 726 kB/s
Requirement already satisfied: opencv-python>=3. in c:\users\admin\appdat
ges (from gym[all]) (4.5.3.56)
Collecting ale-py~=0.7.1
  Downloading ale_py-0.7.2-cp39-cp39-win_amd64.whl (926 kB)
     |████████████████████████████████| 926 kB 652 kB/s
Collecting mujoco-py<2.0,>=1.50
  Downloading mujoco-py-1.50.1.68.tar.gz (120 kB)
     |████████████████████████████████| 120 kB 819 kB/s
Requirement already satisfied: scipy>=1.4.1 in c:\users\admin\appdata\loc
from gym[all]) (1.7.1)
```

Installation of IPython

```
C:\Users\admin>py -m pip install IPython
Collecting IPython
  Downloading ipython-7.28.0-py3-none-any.whl (788 kB)
     |████████████████████████████████| 788 kB 1.3 MB/s
Requirement already satisfied: setuptools>=18.5 in c:\users\adr
es (from IPython) (56.0.0)
Collecting jedi>=0.16
  Downloading jedi-0.18.0-py2.py3-none-any.whl (1.4 MB)
     |████████████████████████████████| 1.4 MB 656 kB/s
Collecting pickleshare
  Downloading pickleshare-0.7.5-py2.py3-none-any.whl (6.9 kB)
Collecting colorama
  Downloading colorama-0.4.4-py2.py3-none-any.whl (16 kB)
Collecting backcall
  Downloading backcall-0.2.0-py2.py3-none-any.whl (11 kB)
Collecting decorator
```

Cab/Taxi Example: Code

```
import gym

env = gym.make("Taxi-v3").env

env.reset()
for _ in range(10):
    env.render()
    env.step(env.action_space.sample()) # take a random action
env.close()
```

Cab/Taxi Example: Output

```
+---------+
|R: |▯[43m ▯[0m: :▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
+---------+

+---------+
|R: |▯[43m ▯[0m: :▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
+---------+
  (Dropoff)
+---------+
|R: | :▯[43m ▯[0m:▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
+---------+
  (East)
+---------+
|R: |▯[43m ▯[0m: :▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
```

```
+---------+
  (West)
+---------+
|R: |▯[43m ▯[0m: :▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
+---------+
  (Dropoff)
+---------+
|R: |▯[43m ▯[0m: :▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
+---------+
  (Dropoff)
+---------+
|R: |▯[43m ▯[0m: :▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
+---------+
  (North)
+---------+
|R: | :▯[43m ▯[0m:▯[35mG▯[0m|
| : | : : |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
```

```
+---------+
  (East)
+---------+
|R: | : :▯[35mG▯[0m|
| : | :▯[43m ▯[0m: |
| : : : : |
| | : | : |
|▯[34;1mY▯[0m| : |B: |
+---------+
  (South)
```

Cab/Taxi Example:

```
env.s = 328  # set environment to illustration's state

epochs = 0
penalties, reward = 0, 0

frames = [] # for animation

done = False

while not done:
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)

    if reward == -10:
        penalties += 1

    # Put each rendered frame into dict for animation
    frames.append({
        'frame': env.render(mode='ansi'),
        'state': state,
        'action': action,
        'reward': reward
        }
    )

    epochs += 1


print("Timesteps taken: {}".format(epochs))
print("Penalties incurred: {}".format(penalties))
```

OUTPUT:

```
Timesteps taken: 2764
Penalties incurred: 891
```

Cab/Taxi Example:

```python
from IPython.display import clear_output
from time import sleep

def print_frames(frames):
    for i, frame in enumerate(frames):
        clear_output(wait=True)
        print(frame['frame'])
        print(f"Timestep: {i + 1}")
        print(f"State: {frame['state']}")
        print(f"Action: {frame['action']}")
        print(f"Reward: {frame['reward']}")
        sleep(.1)

print_frames(frames)
```

***Error given by:**

print(frame['frame'].getvalue())

OUTPUT:

```
□[2K□[2K      +---------+
|□[35mR□[0m: | : :G|
| : | : : |
| : : : : |
| | : | : |
|□[42mY□[0m| : |B: |
+---------+
  (East)

Timestep: 251
State: 416
Action: 2
Reward: -1
□[2K□[2K      +---------+
|□[35mR□[0m: | : :G|
| : | : : |
| : : : : |
|□[42m_□[0m| : | : |
|Y| : |B: |
+---------+
  (North)

Timestep: 252
State: 316
Action: 1
Reward: -1
□[2K□[2K      +---------+
|□[35mR□[0m: | : :G|
| : | : : |
|□[42m_□[0m: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)

Timestep: 253
State: 216
Action: 1
Reward: -1
```

Q-learning

From Wikipedia, the free encyclopedia

***Q*-learning** is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.

For any finite Markov decision process (FMDP), *Q*-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state.[1] *Q*-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy.[1] "Q" refers to the function that the algorithm computes – the expected rewards for an action taken in a given state.[2]

Q-Learning Algorithm

After Δt steps into the future the agent will decide some next step. The weight for this step is calculated as $\gamma^{\Delta t}$, where γ (the *discount factor*) is a number between 0 and 1 ($0 \leq \gamma \leq 1$) and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). γ may also be interpreted as the probability to succeed (or survive) at every step Δt.

The algorithm, therefore, has a function that calculates the quality of a state–action combination:

$$Q : S \times A \to \mathbb{R}.$$

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information:

Q-Learning Algorithm

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate $(0 < \alpha \leq 1)$.

Note that $Q^{new}(s_t, a_t)$ is the sum of three factors:

- $(1 - \alpha)Q(s_t, a_t)$: the current value weighted by the learning rate. Values of the learning rate near to 1 make the changes in Q more rapid.
- $\alpha\, r_t$: the reward $r_t = r(s_t, a_t)$ to obtain if action a_t is taken when in state s_t (weighted by learning rate)
- $\alpha\gamma \max_a Q(s_{t+1}, a)$: the maximum reward that can be obtained from state s_{t+1} (weighted by learning rate and discount factor)

An episode of the algorithm ends when state s_{t+1} is a final or *terminal state*. However, Q-learning can also learn in non-episodic tasks (as a result of the property of convergent infinite series). If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states s_f, $Q(s_f, a)$ is never updated, but is set to the reward value r observed for state s_f. In most cases, $Q(s_f, a)$ can be taken to equal zero.

Q Table

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Q-Learning table of states by actions that is initialized to zero, then each cell is updated through training.

Q-Learning Implementation

```
#------------Q-Learning---------------------

import numpy as np
q_table = np.zeros([env.observation_space.n, env.action_space.n])

#%%time
"""Training the agent"""

import random
from IPython.display import clear_output

# Hyperparameters
alpha = 0.1
gamma = 0.6
epsilon = 0.1

# For plotting metrics
all_epochs = []
all_penalties = []
```

Q-Learning Implementation

```
for i in range(1, 1000):
    state = env.reset()

    epochs, penalties, reward, = 0, 0, 0
    done = False

    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Explore action space
        else:
            action = np.argmax(q_table[state]) # Exploit learned values

        next_state, reward, done, info = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])

        new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
        q_table[state, action] = new_value

        if reward == -10:
            penalties += 1

        state = next_state
        epochs += 1

    if i % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")

print("Training finished.\n")
```

OUTPUT:

```
[2K[2KEpisode: 100
[2K[2KEpisode: 200
[2K[2KEpisode: 300
[2K[2KEpisode: 400
[2K[2KEpisode: 500
[2K[2KEpisode: 600
[2K[2KEpisode: 700
[2K[2KEpisode: 800
[2K[2KEpisode: 900
Training finished.
```

Q-Learning Implementation

```
"""Evaluate agent's performance after Q-learning"""

total_epochs, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = False

    while not done:
        action = np.argmax(q_table[state])
        state, reward, done, info = env.step(action)

        if reward == -10:
            penalties += 1

        epochs += 1

    total_penalties += penalties
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

Maybe working for Jupyter

Hyperparameters

The values of `alpha`, `gamma`, and `epsilon` were mostly based on intuition and some "hit and trial", but there are better ways to come up with good values.

Ideally, all three should decrease over time because as the agent continues to learn, it actually builds up more resilient priors;

- α: (the learning rate) should decrease as you continue to gain a larger and larger knowledge base.
- γ: as you get closer and closer to the deadline, your preference for near-term reward should increase, as you won't be around long enough to get the long-term reward, which means your gamma should decrease.
- ϵ: as we develop our strategy, we have less need of exploration and more exploitation to get more utility from our policy, so as trials increase, epsilon should decrease.

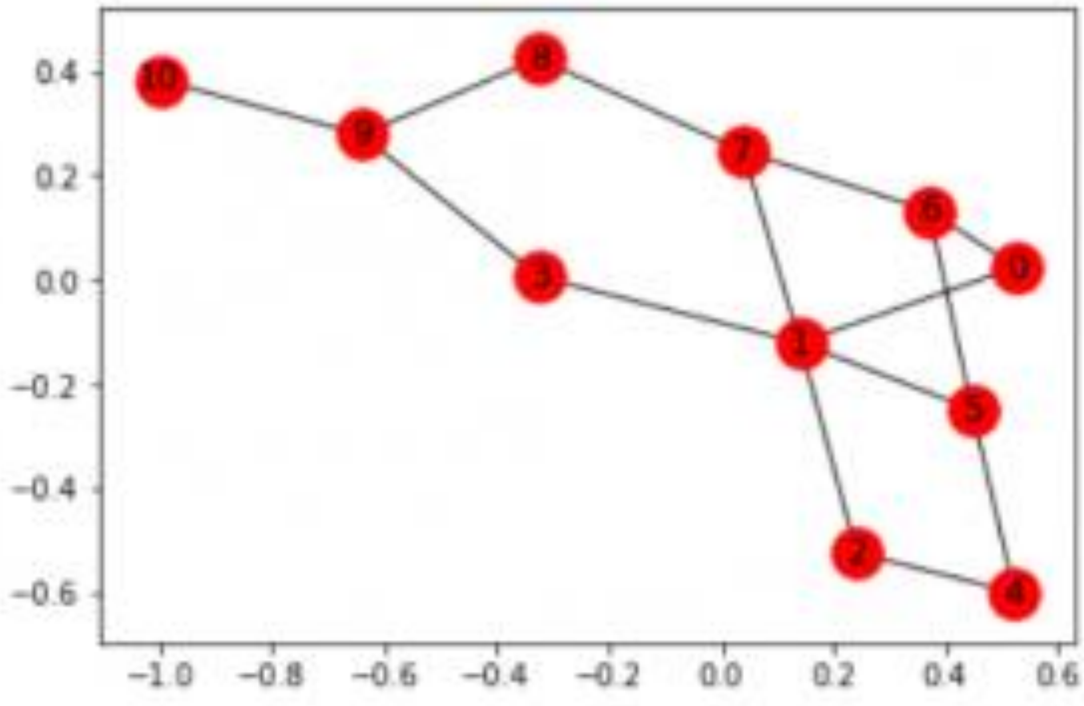
Other Ways of Q-Learning

#Step 1: Importing the required libraries

```
import numpy as np
import pylab as pl
import networkx as nx
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),
         (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),
         (8, 9), (7, 8), (1, 7), (3, 9)]
```

#Step 2: Defining and visualizing the graph

```
goal = 10
G = nx.Graph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
pl.show()
```



Other Ways of Q-Learning

#Step 3: Defining the reward the system for the bot

```
MATRIX_SIZE = 11
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
M *= -1

for point in edges:
        print(point)
        if point[1] == goal:
                M[point] = 100
        else:
                M[point] = 0

        if point[0] == goal:
                M[point[::-1]] = 100
        else:
                M[point[::-1]]= 0
                # reverse of point

M[goal, goal]= 100
print(M)
# add goal point round trip
```

```
[[ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.   0.   0.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.]
 [ -1.  -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.]
 [ -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.   0.  -1. 100.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0. 100.]]
```

Other Ways of Q-Learning

#Step 4: Defining some utility functions to be used in the training

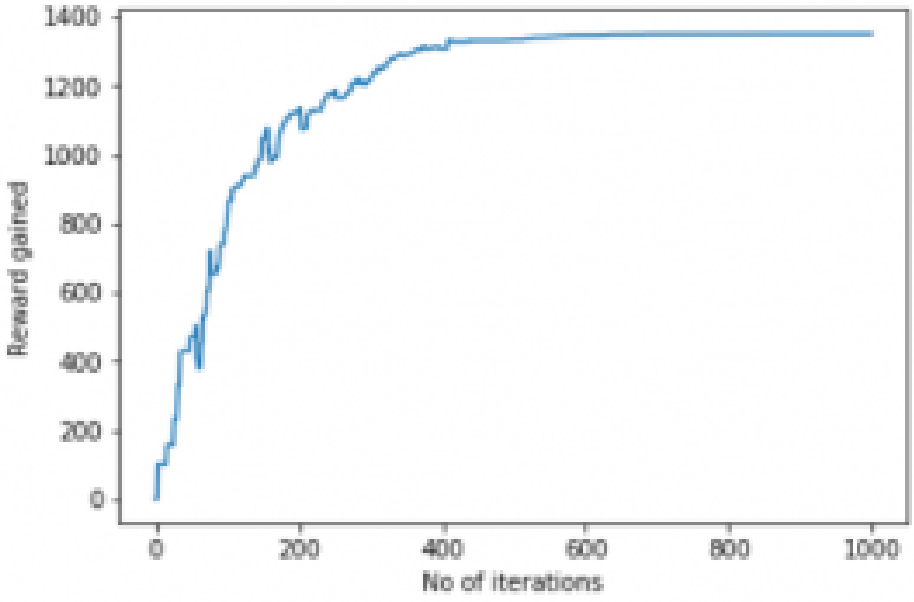
```
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))

gamma = 0.75
# learning parameter
initial_state = 1

# Determines the available actions for a given state
def available_actions(state):
	current_state_row = M[state, ]
	available_action = np.where(current_state_row >= 0)[1]
	return available_action

available_action = available_actions(initial_state)

# Chooses one of the available actions at random
def sample_next_action(available_actions_range):
	next_action = int(np.random.choice(available_action, 1))
	return next_action

action = sample_next_action(available_action)

def update(current_state, action, gamma):

max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
if max_index.shape[0] > 1:
	max_index = int(np.random.choice(max_index, size = 1))
else:
	max_index = int(max_index)
max_value = Q[action, max_index]
Q[current_state, action] = M[current_state, action] + gamma *
max_value
if (np.max(Q) > 0):
	return(np.sum(Q / np.max(Q)*100))
else:
	return (0)

# Updates the Q-Matrix according to the path chosen
update(initial_state, action, gamma)
```

Other Ways of Q-Learning

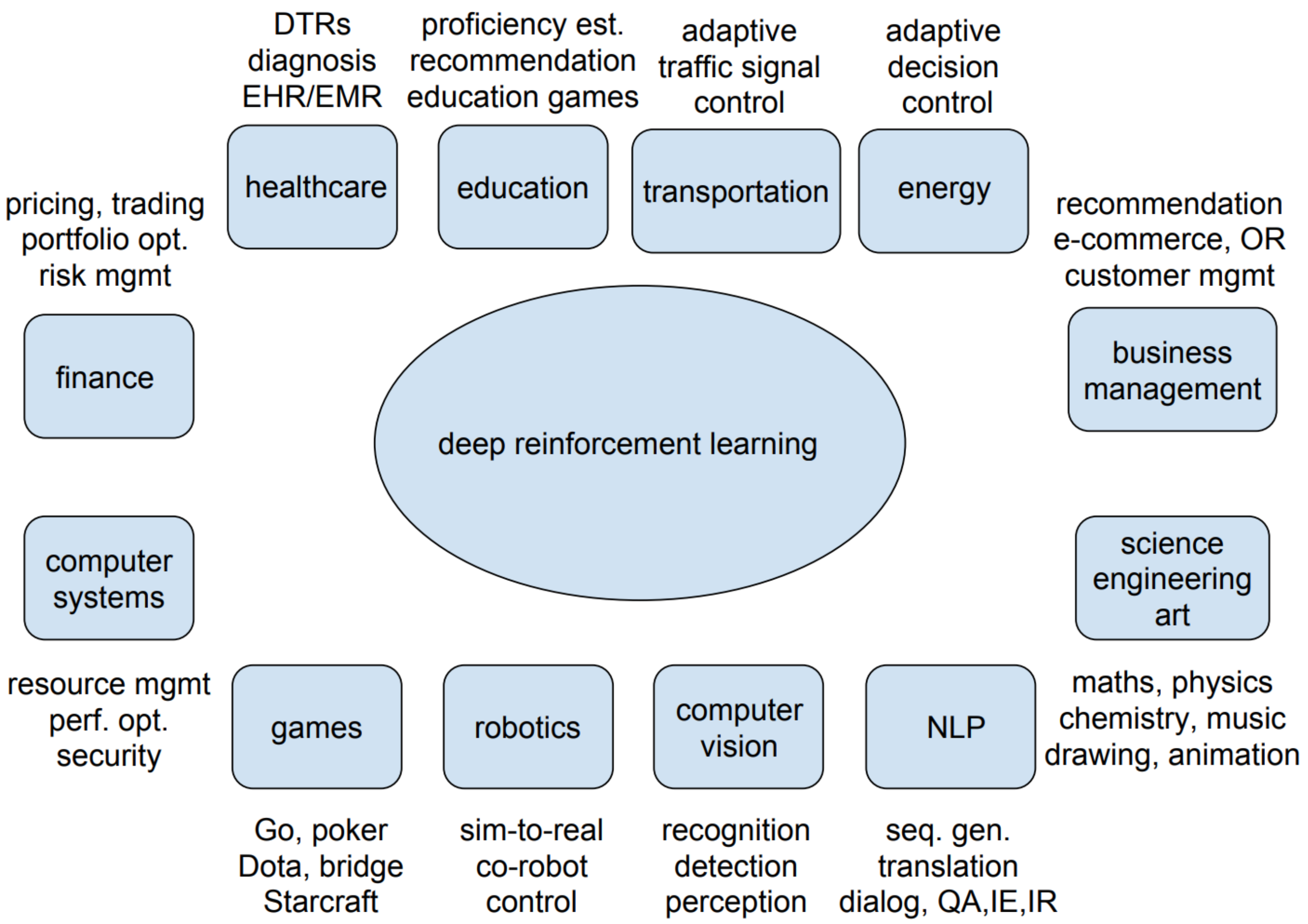
#Step 5: Training and evaluating the bot using the Q-Matrix

```
scores = []
for i in range(1000):
        current_state = np.random.randint(0, int(Q.shape[0]))
        available_action = available_actions(current_state)
        action = sample_next_action(available_action)
        score = update(current_state, action, gamma)
        scores.append(score)

# print("Trained Q matrix:")
# print(Q / np.max(Q)*100)
# You can uncomment the above two lines to view the trained Q matrix

# Testing
current_state = 0
steps = [current_state]

while current_state != 10:

        next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state, ]))[1]
        if next_step_index.shape[0] > 1:
                next_step_index = int(np.random.choice(next_step_index, size = 1))
        else:
                next_step_index = int(next_step_index)
        steps.append(next_step_index)
        current_state = next_step_index

print("Most efficient path:")
print(steps)

pl.plot(scores)
pl.xlabel('No of iterations')
pl.ylabel('Reward gained')
```

```
Most efficient path:
[0, 1, 3, 9, 10]
```



Deep Reinforcement Learning



Deep Reinforcement Learning Applications

Deep Q Network (DQN)

- Core idea: We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates

- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state

- The neural network is trained using mini-batch stochastic gradient updates and experience replay

Required Packages to be Installed

1. keras-rl
2. h5py
3. gym

Implementation

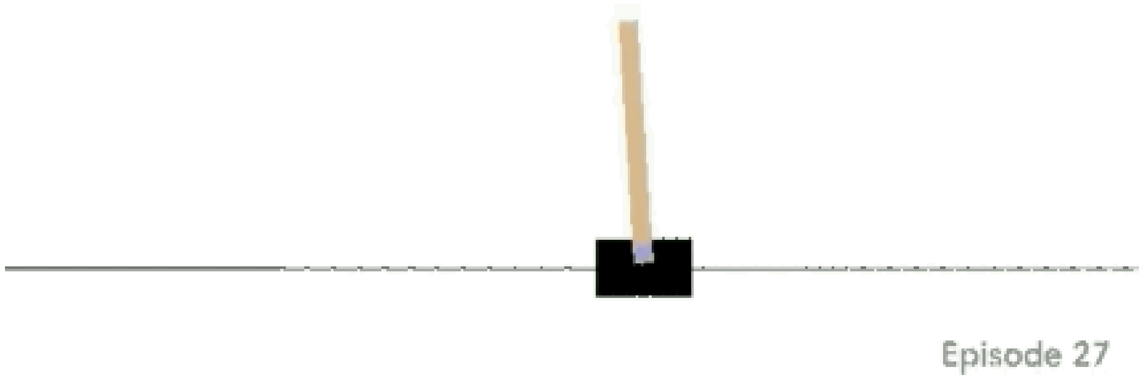
```
import numpy as np
import gym

from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten
from keras.optimizers import Adam

from rl.agents.dqn import DQNAgent
from rl.policy import EpsGreedyQPolicy
from rl.memory import SequentialMemory

ENV_NAME = 'CartPole-v0'

# Get the environment and extract the number of
actions available in the Cartpole problem
env = gym.make(ENV_NAME)
np.random.seed(123)
env.seed(123)
nb_actions = env.action_space.n

model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())

policy = EpsGreedyQPolicy()
memory = SequentialMemory(limit=50000, window_length=1)
dqn = DQNAgent(model=model, nb_actions=nb_actions,
memory=memory, nb_steps_warmup=10,
target_model_update=1e-2, policy=policy)
dqn.compile(Adam(lr=1e-3), metrics=['mae'])

# Okay, now it's time to learn something! We visualize the training
# here for show, but this slows down training quite a lot.
dqn.fit(env, nb_steps=5000, visualize=True, verbose=2)
```

Implementation

#Test our reinforcement learning model:

```
dqn.test(env, nb_episodes=5, visualize=True)
```

OUTPUT:



Reference Links

https://gym.openai.com/docs/#building-from-source

https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

https://www.geeksforgeeks.org/ml-reinforcement-learning-algorithm-python-implementation-using-q-learning/

https://github.com/PacktPublishing/Mastering-Reinforcement-Learning-with-Python

https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

https://www.analyticsvidhya.com/blog/2017/01/introduction-to-reinforcement-learning-implementation/

https://amunategui.github.io/reinforcement-learning/index.html

https://neptune.ai/blog/the-best-tools-for-reinforcement-learning-in-python

https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/

https://github.com/sudharsan13296/Hands-On-Reinforcement-Learning-With-Python