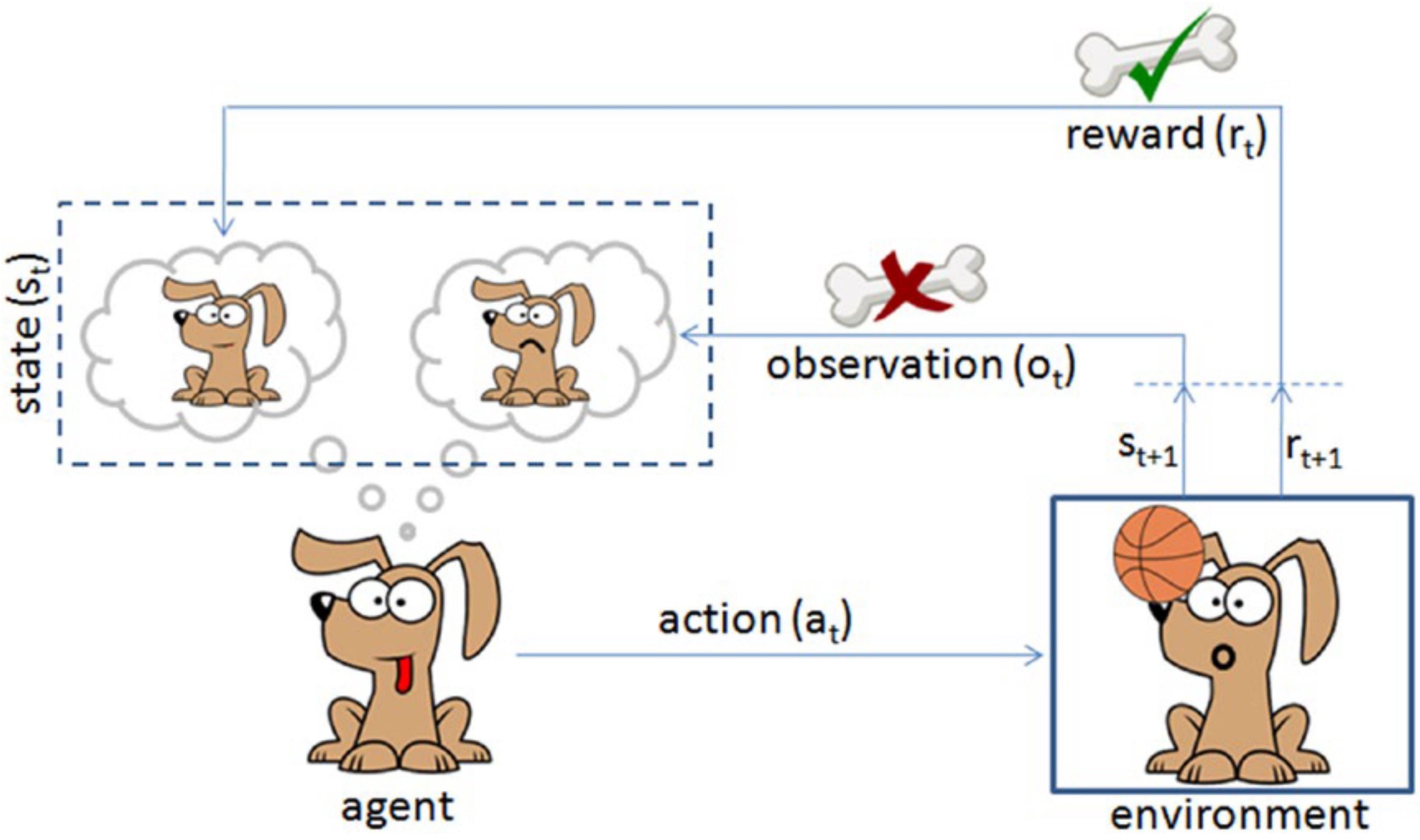
REINFORCEMENT LEARNING

Reinforcement learning is a goal-oriented learning method based on interaction with its environment. The objective is getting an agent to act in an environment in order to maximize its rewards. Here the agent is an intelligent program, and the environment is the external condition.

Reinforcement learning is like teaching your dog a trick. For example, consider teaching a dog a new trick, you cannot tell it what to do, but you can reward or punish it if it does the right or wrong thing respectively. It has to figure out what it did that made it get the reward or punishment, which is known as the credit assignment problem. We can use a similar method to train computers to do many tasks, such as playing chess or any other games, scheduling jobs, and controlling robot limbs.


reward (r_t)
state (s_t)
observation (o_t)
s_{t+1}
r_{t+1}
action (a_t)
agent
environment

How to formulate a basic Reinforcement Learning problem?

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

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

In order to build an optimal policy, the agent faces the dilemma of exploring new states while maximizing its overall reward at the same time. This is called **Exploration vs Exploitation** trade-off. To balance both, the best overall strategy may involve short term sacrifices. Therefore, the agent should collect enough information to make the best overall decision in the future.

Markov Decision Processes (MDPs) are mathematical frameworks to describe an environment in RL and almost all RL problems can be formulated using MDPs. An MDP consists of a set of finite environment states S, a set of possible actions A(s) in each state, a real valued reward function R(s) and a transition model P(s', s | a). However, real world environments are more likely to lack any prior knowledge of environment dynamics. Model-free RL methods come handy in such cases.

Q-learning is a commonly used model-free approach which can be used for building a self-playing PacMan agent (https://www.youtube.com/watch?v=QilHGSYbjDQ). It revolves around the notion of updating Q values which denotes value of performing action *a* in state *s*. The following value update rule is the core of the Q-learning algorithm.

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

According to the paper publish by Deepmind Technologies in 2013, Q-learning rule for updating status is given by:

Q[s,a]new = Q[s,a]prev + α * (r + y*max(s,a) – Q[s,a]prev), where

- α is the learning rate,
- r is reward for latest action,

- ƴ is the discounted factor, and
- max(s,a) is the estimate of new value from best action.

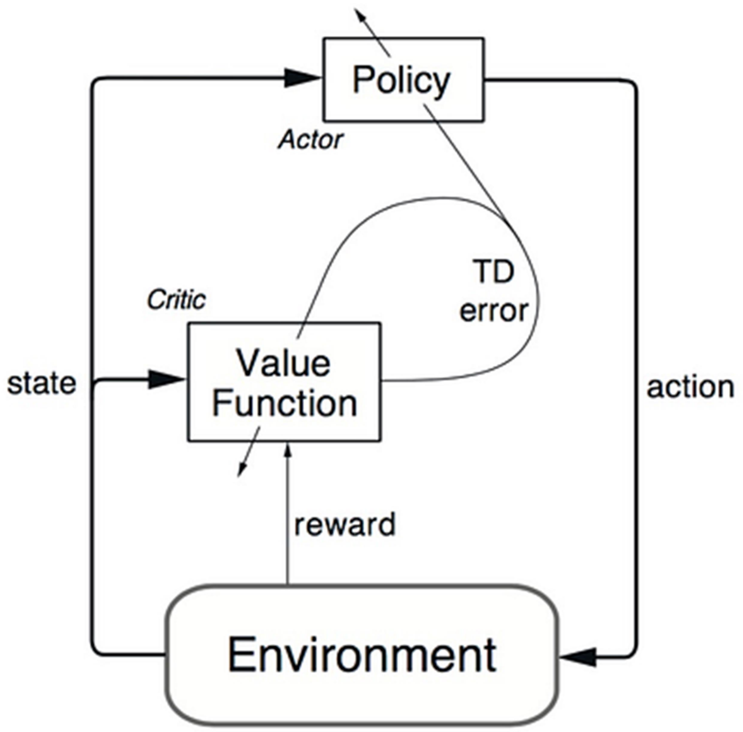
If the optimal value Q[s,a] of the sequence s' at the next time step was known for all possible actions a', then the optimal strategy is to select the action a' maximizing the expected value of r + ƴ*max(s,a) – Q[s,a]prev.

What are some of the most used Reinforcement Learning algorithms?

Q-learning and SARSA (State-Action-Reward-State-Action) are two commonly used model-free RL algorithms. They differ in terms of their exploration strategies while their exploitation strategies are similar. While Q-learning is an off-policy method in which the agent learns the value based on action a* derived from the another policy, SARSA is an on-policy method where it learns the value based on its current action *a* derived from its current policy. These two methods are simple to implement but lack generality as they do not have the ability to estimates values for unseen states.

This can be overcome by more advanced algorithms such as Deep Q-Networks(DQNs) which use Neural Networks to estimate Q-values. But DQNs can only handle discrete, low-dimensional action spaces.

Deep Deterministic Policy Gradient (DDPG) is a model-free, off-policy, actor-critic algorithm that tackles this problem by learning policies in high dimensional, continuous action spaces. The figure below is a representation of actor-critic architecture.



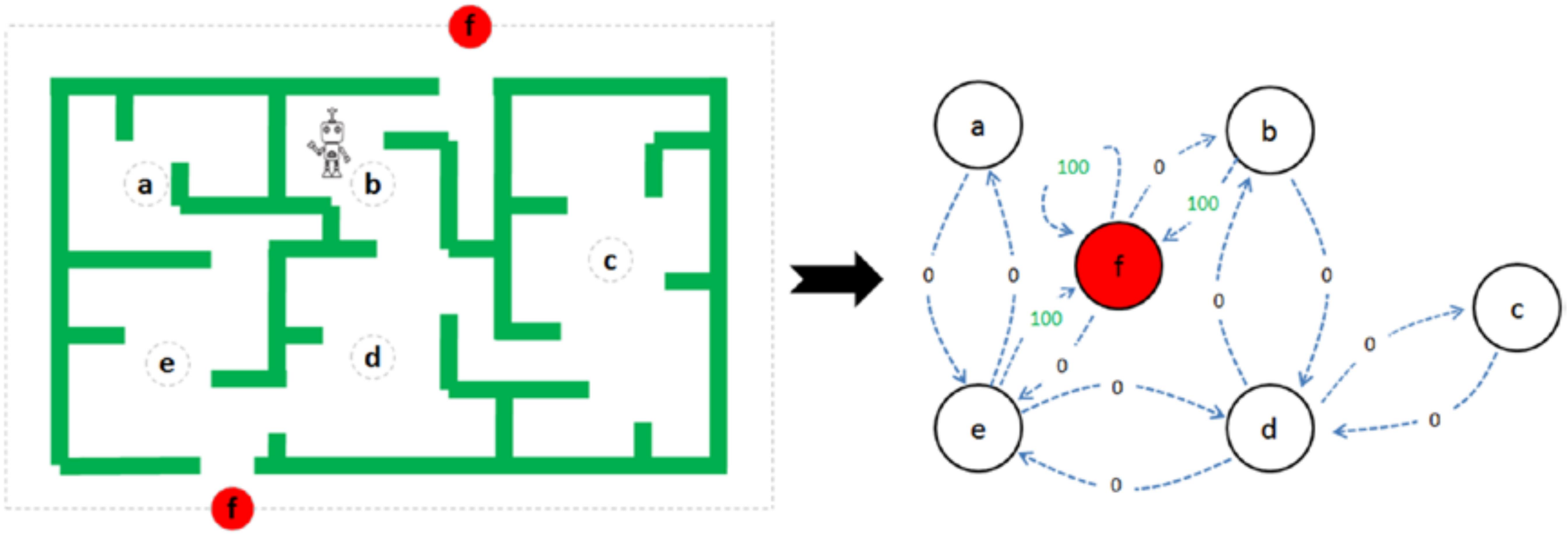
What are the practical applications of Reinforcement Learning?

Since, RL requires a lot of data, therefore it is most applicable in domains where simulated data is readily available like gameplay, robotics.

1. RL is quite widely used in building AI for playing computer games. AlphaGo Zero is the first computer program to defeat a world champion in the ancient Chinese game of Go. Others include ATARI games, Backgammon ,etc
2. In robotics and industrial automation, RL is used to enable the robot to create an efficient adaptive control system for itself which learns from its own experience and behavior. DeepMind's work on Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Policy updates is a good example of the same. Watch this interesting demonstration video (https://www.youtube.com/watch?v=ZhsEKTo7V04).

Example

Let's consider an example where an agent is trying to come out of a maze. It can move one random square or area in any direction, and get a reward if exits. The most common way to formalize a reinforcement problem is to represent it as Markov decision process. Assume the agent is in state b (maze area) and the target is to reach state f. So within one step agent can reach from b to f, let's put a reward of 100 (otherwise 0) for links between nodes that allows agents to reach target state.



Python Code:

```
import numpy as np
from random import randint
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

# defines the reward/link connection graph
R = np.matrix(      [[-1, -1, -1, -1,  0,  -1],
                    [-1, -1, -1,  0, -1, 100],
                    [-1, -1, -1,  0, -1,  -1],
                    [-1,  0,  0, -1,  0,  -1],
                    [ 0, -1, -1,  0, -1, 100],
                    [-1,  0, -1, -1,  0, 100]])
Q = np.zeros_like(R)

###  The -1's in the table means there isn't a link between nodes. For example, State 'a' cannot go to State 'b'.

# learning parameter
gamma = 0.8

# Initialize random_state
initial_state = randint(0,4)
```

```
# This function returns all available actions in the state given as an argument
def available_actions(state):
        current_state_row = R[state,]
        av_act = np.where(current_state_row >= 0)[1]
        return av_act

# This function chooses at random which action to be performed within the range
# of all the available actions.
def sample_next_action(available_actions_range):
        next_action = int(np.random.choice(available_act,1))
        return next_action

# This function updates the Q matrix according to the path selected and the Q learning algorithm
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 learning formula
   Q[current_state, action] = R[current_state, action] + gamma * max_value
```

```
# Get available actions in the current state
available_act = available_actions(initial_state)

# Sample next action to be performed
action = sample_next_action(available_act)
```

### Training

```
# Train over 100 iterations, re-iterate the process above).
for i in range(100):
        current_state = np.random.randint(0, int(Q.shape[0]))
        available_act = available_actions(current_state)
        action = sample_next_action(available_act)
        update(current_state,action,gamma)

# Normalize the "trained" Q matrix
print ("Trained Q matrix: \n", Q/np.max(Q)*100)
```

<u>Output:</u>

```
Trained Q matrix:
 [[  0.          0.          0.          0.         45.9833795        0.       ]
 [  0.          0.          0.         63.71191136   0.            100.       ]
 [  0.          0.          0.         63.71191136   0.              0.      ]
 [  0.         79.77839335  50.96952909   0.         36.28808864         0.       ]
 [ 28.80886427   0.          0.         57.61772853   0.          45.42936288]
 [  0.          79.77839335   0.          0.         45.9833795  100.        ]]
```

```
###  Testing
current_state = 2
steps = [current_state]
while current_state != 5:
    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 selected sequence of steps
print ("Best sequence path: ", steps)
```

Output:

```
Best sequence path:  [2, 3, 1, 5]
```

References:

1. http://incompleteideas.net/book/RLbook2020.pdf
2. https://www.davidsilver.uk/wp-content/uploads/2020/03/intro_RL.pdf
   from https://www.davidsilver.uk/teaching/
3. http://people.eecs.berkeley.edu/~pabbeel/nips-tutorial-policy-optimization-Schulman-Abbeel.pdf
4. http://karpathy.github.io/2016/05/31/rl/
5. https://gist.github.com/karpathy/a4166c7fe253700972fcbc77e4ea32c5
6. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292
7. https://www.youtube.com/watch?v=QilHGSYbjDQ
8. https://www.youtube.com/watch?v=ZhsEKTo7V04
9. https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/
10. Swamynathan, M. (2019). Mastering machine learning with python in six steps: A practical implementation guide to predictive data analytics using python. Apress.