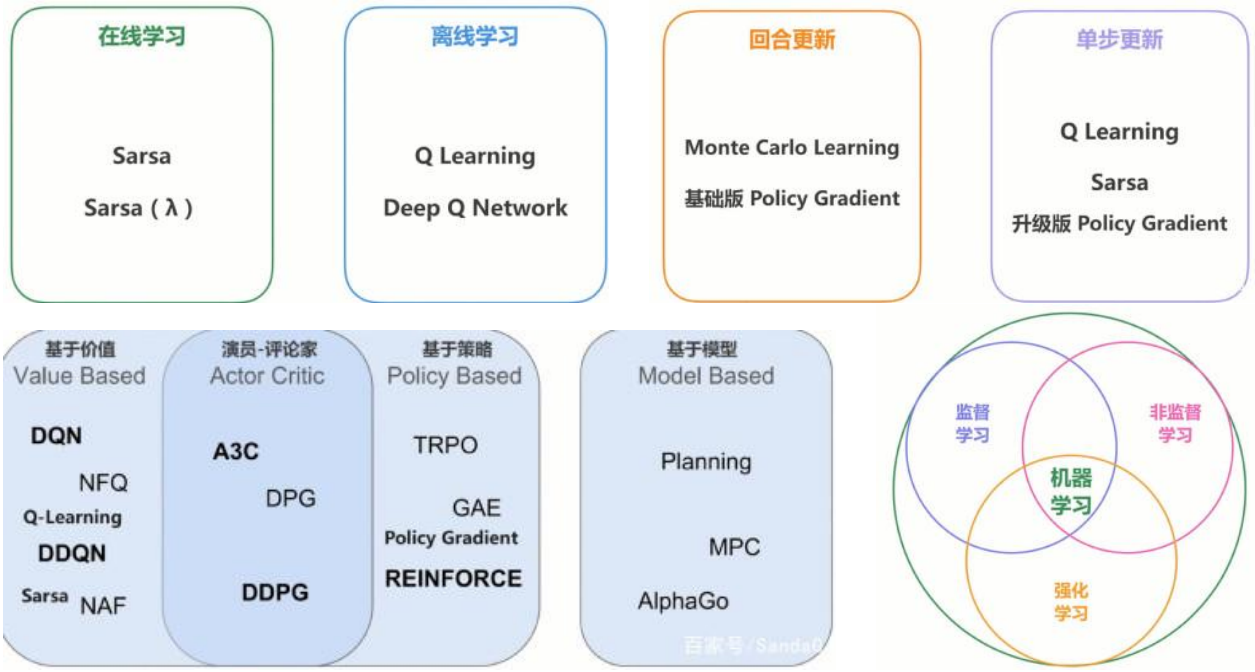
Reinforcement Learning

Contents

1 Overview



2 Introduction to Reinforcement Learning

We only understand a sliver of how the brain works, but we *do* know that it often learns through trial and error. We're rewarded when we do good things and punished when we do the wrong ones; that's how we figure out how to live. Reinforcement Learning puts computational power behind that exact process and lets us model it with software.

2.1 What Is Reinforcement Learning?

The easiest mental model to aid in understanding Reinforcement Learning (RL) is as a video game, which coincidentally is one of the most popular applications of RL algorithms. In a typical video game, you might have:

- An **agent** (the player) who moves around doing stuff
- An **action** that the agent takes (moves upward one space, sells cloak)
- A **reward** that the agent acquires (coins, killing other players, etc.)
- An **environment** that the agent exists in (a map, a room)

- A **state** that the agent currently exists in (on a particular square of a map, part of a room)
- A **goal** of your agent getting as many rewards as possible

These literally are the exact building blocks of Reinforcement Learning (maybe Machine Learning is just a game?). In RL, we guide an **agent** through an **environment**, **state** by **state**, by issuing a **reward** every time the agent does the right thing. If you've heard the term Markov Decision Process thrown around, it pretty much describes this exact setting.

For a visual aid, you can think of a mouse in a maze:

Source: Machine Learning for Humans

If you were navigating the maze yourself and your goal was to collect as many rewards as possible (the water droplets and cheese), what would you do? At each **state** (position in the maze), you'd calculate what steps you need to take to reach the rewards near you. If there are 3 rewards on your right and 1 on your left, you'd go right.

This is how Reinforcement Learning works. At each state, the agent makes an educated calculation about all of the possible **actions** (left, right, up, down, you name it) and takes the action that'll yield the best result. After completing this process a few times, you'd imagine that our mouse might know the maze pretty well.

But how exactly do you decide what the "best result" is?

2.2 Decision Making in Reinforcement Learning

There are 2 broad approaches for how you can teach your agent to make the right decisions in a Reinforcement Learning environment.

Policy Learning Policy learning is best understood as a set of very detailed directions – it tells the agent exactly what to do at each state. A part of a policy might look something like: "if you approach an enemy and the enemy is stronger than you, turn backwards." If you think of a policy as a function, it only has one input: the state. But knowing in advance what your policy should be isn't easy, and requires deep knowledge of the complex function that maps state to goal.

There's been some very interesting research in applying Deep Learning to learn policies for Reinforcement Learning scenarios. Andrej Karpathy implemented a neural net to teach an agent how to play the classic game of pong. This shouldn't surprise us, since neural nets can be very good at approximating complicated functions.

Q-Learning / Value Functions Another way of guiding our agent is not by explicitly telling her what to do at each point, but by giving her **a framework** to make her own decisions. Unlike policy learning, Q-Learning takes *two inputs* – state *and* action – and returns a value for each pair. If you're at an intersection, Q-learning will tell you the expected value of each action your agent could take (left, right, etc.).

One of the quirks of Q-Learning is that it doesn't just estimate the *immediate* value of taking an action in a given state: it also adds in all of the potential future value that could be had if you take the specified action. For readers familiar with corporate finance, Q-Learning is sort of like a discounted cash flow analysis – it takes all potential future value into account when determining the current value of an action (or asset). In fact, Q-Learning even uses a *discount-factor* to model the fact that rewards in the future are worth less than rewards now.

Policy Learning and Q-Learning are the two mainstays of how to guide an agent in Reinforcement Learning, but a bunch of new approaches have been using Deep Learning to combine the two or attempt other creative solutions. DeepMind published a paper about using neural nets (called Deep Q Networks) to approximate Q-Learning functions, and achieved impressive results. A few years later, they pioneered a method called A3C that combined Q-Learning and Policy Learning approaches.

Adding neural nets into anything can make it sound complicated. Just remember that all of these learning approaches have a simple goal: to effectively guide your agent through the environment and acquire the most rewards. That's it.

2.3 Practical Applications of Reinforcement Learning

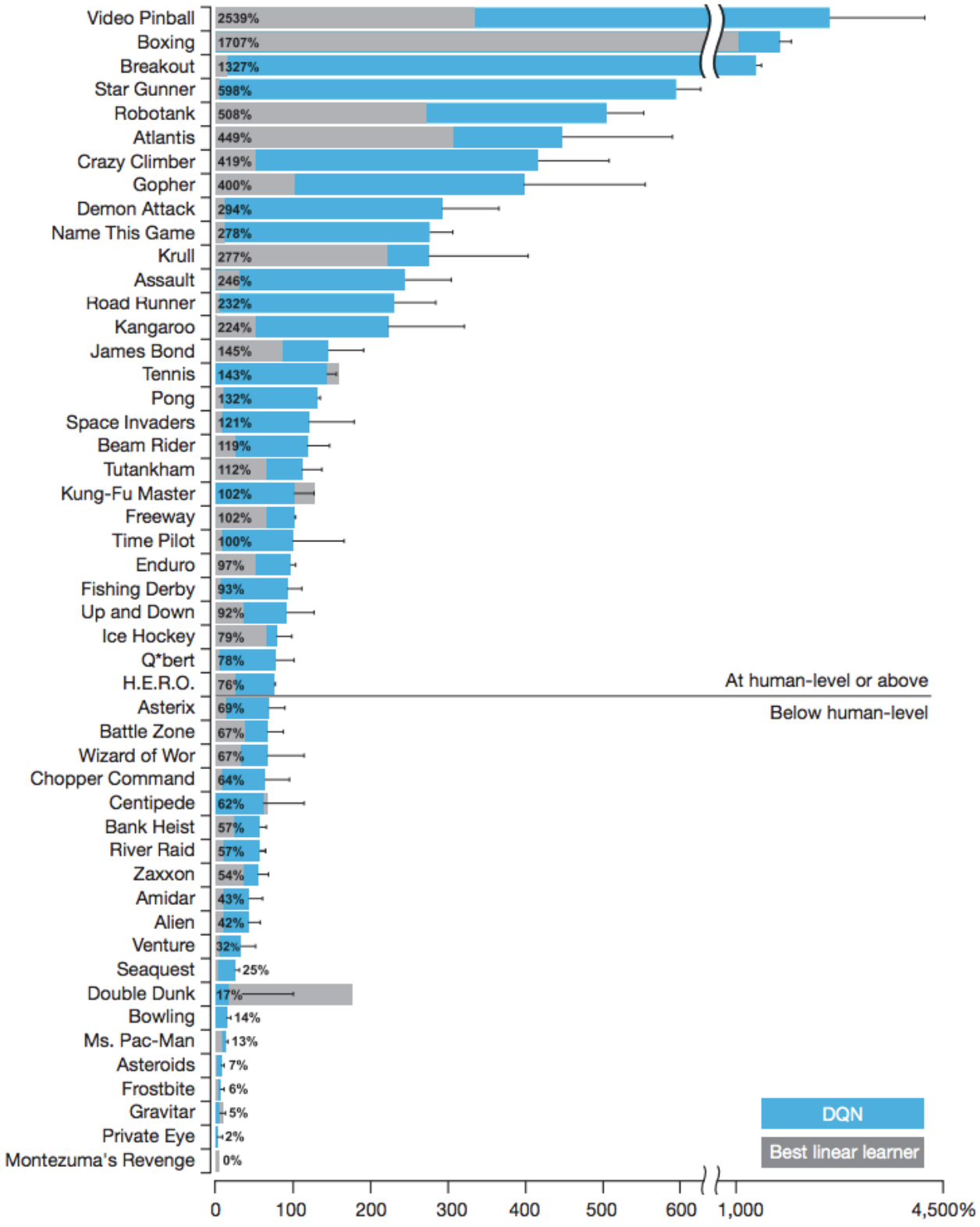
While the concepts supporting Reinforcement Learning have been around for decades, unfortunately it's rarely implemented in practice today in business contexts. There are a number of reasons for that (see the challenges section below), but they all follow a similar thread: Reinforcement Learning struggles to efficiently beat out other algorithms for well defined tasks.

Most of the practical application of Reinforcement Learning in the past decade has been in the realm of video games. Cutting edge Reinforcement Learning algorithms have achieved impressive results in classic and modern games, often beating out their human counterparts by a significant margin.

This graph is from the above mentioned DQN paper by DeepMind. For more than half the games tested, their agent was able to outperform human benchmarks, often by more than double the skill level. For certain games though, their algorithms weren't even close to human performance.

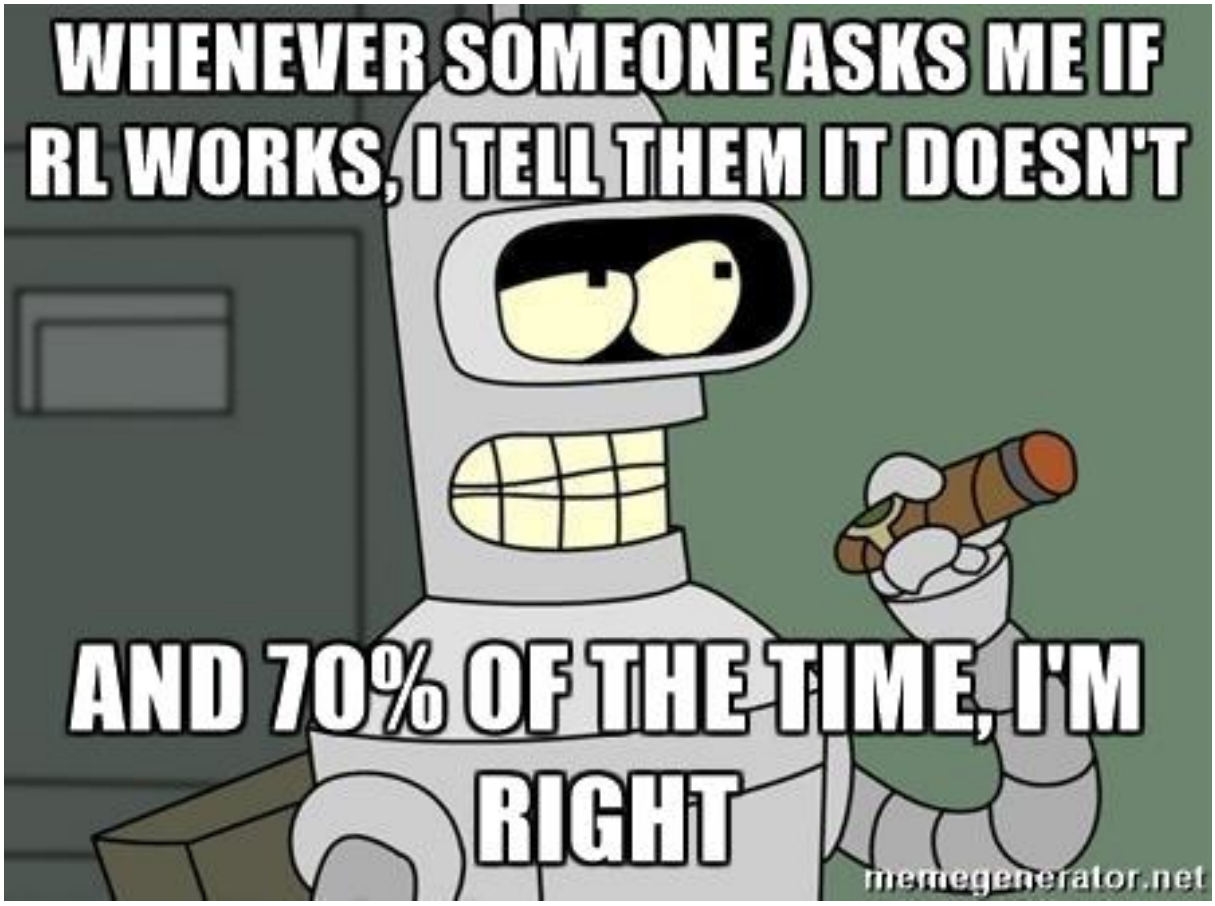
The other major area where RL has seen some practical success is in robotics and industrial automation. Robots can easily be understood as agents in an environment, and Reinforcement Learning has been shown to be a feasible teaching solution. Google has also made progress using Reinforcement Learning to cut down costs in their data centers.

Healthcare and education are also promising areas for Reinforcement Learning, but most of the work is purely academic at this point.


Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 112%
Kung-Fu Master 102%
Freeway 102%
Time Pilot 100%
Enduro 97%
Fishing Derby 93%
Up and Down 92%
Ice Hockey 79%
Q*bert 78%
H.E.R.O. 76%
Asterix 69%
Battle Zone 67%
Wizard of Wor 67%
Chopper Command 64%
Centipede 62%
Bank Heist 57%
River Raid 57%
Zaxxon 54%
Amidar 43%
Alien 42%
Venture 32%
Seaquest 25%
Double Dunk 17%
Bowling 14%
Ms. Pac-Man 13%
Asteroids 7%
Frostbite 6%
Gravitar 5%
Private Eye 2%
Montezuma's Revenge 0%
At human-level or above
Below human-level
DQN
Best linear learner
0
100
200
300
400
500
600
1,000
4,500%

2.4 Challenges With Implementing Reinforcement Learning

While extremely promising, Reinforcement Learning is notoriously difficult to implement in practice.



Source: Alex Irpan

The first issue is data: Reinforcement Learning typically requires a *ton* of training data to reach accuracy levels that other algorithms can get to more efficiently. RainbowDQN (from the most recent DeepMind paper) requires 18 Million frames of Atari gameplay to train properly, or about 83 hours of play time. A human can pick up the game *much*faster than that. This issue seems to hold across disciplines (like learning a running gait).

Another challenge in implementing Reinforcement Learning is the domain-specificity problem. Reinforcement Learning is a general algorithm, in that it should theoretically work for all different types of problems. But most of those problems have a domain-specific solution that will work better than RL, like online trajectory optimization for MuJuCo robots. There's always a tradeoff between scope and intensity.

Finally, the most pressing issue with Reinforcement Learning as it currently stands is

the design of the reward function (recall Q-Learning and Policy Learning). If the algorithm designers are the ones setting up rewards, then model results are extremely subjective to the bias of the designers. Even when set up properly, Reinforcement Learning has this clever way of finding ways around what you want it to do and getting stuck in local optima.

With so much cutting edge research focused on advancing Reinforcement Learning, expect some of these kinks to get ironed out over time.

2.5 Resources

Frameworks and Packages RL-Glue – "*RL-Glue (Reinforcement Learning Glue) provides a standard interface that allows you to connectreinforcement learningagents, environments, and experiment programs together, even if they are written in different languages.*"

Gym (OpenAI) – "*Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything fromwalkingto playing games likePong orPinball.*"

RL4J (DL4J) – "*RL4J is a reinforcement learning framework integrated with deeplearning4j and released under an Apache 2.0 open-source license.*"

TensorForce (Reinforce.io) – "*A TensorFlow library for applied reinforcement learning.*"

Reading Machine Learning for Humans, Part 5: Reinforcement Learning (Machine Learning for Humans) – "*In reinforcement learning (RL) there's no answer key, but your reinforcement learning agent still has to decide how to act to perform its task. In the absence of existing training data, the agent learns from experience. It collects the training examples ("this action was good, that action was bad") through trial-and-error as it attempts its task, with the goal of maximizing long-term reward.*"

An Introduction to Reinforcement Learning (freeCodeCamp) – "*Reinforcement learning is an important type of Machine Learning where an agent learn how to behave in a environment by performing actions and seeing the results. In recent years, we've seen a lot of improvements in this fascinating area of research. In this series of articles, we will focus on learning the different architectures used today to solve Reinforcement Learning problems.*"

A Beginner's Guide to Deep Reinforcement Learning (DL4J) – "*Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps; for example, maximize the points*

won in a game over many moves. They can start from a blank slate, and under the right conditions they achieve superhuman performance. Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement."

Deep Reinforcement Learning (DeepMind) – "*Humans excel at solving a wide variety of challenging problems, from low-level motor control through to high-level cognitive tasks. Our goal at DeepMind is to create artificial agents that can achieve a similar level of performance and generality. Like a human, our agents learn for themselves to achieve successful strategies that lead to the greatest long-term rewards.*"

Lessons Learned Reproducing a Deep Reinforcement Learning Paper (Amid Fish) – "*I've seen a few recommendations that reproducing papers is a good way of levelling up machine learning skills, and I decided this could be an interesting one to try with. It was indeed a super fun project, and I'm happy to have tackled it – but looking back, I realise it wasn't exactly the experience I thought it would be. If you're thinking about reproducing papers too, here are some notes on what surprised me about working with deep RL.*"

Papers Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm (13 authors!) – "*In this paper, we generalise this approach into a single AlphaZero algorithm that can achieve, tabula rasa, superhuman performance in many challenging domains. Starting from random play, and given no domain knowledge except the game rules, AlphaZero achieved within 24 hours a superhuman level of play in the games of chess and shogi (Japanese chess) as well as Go, and convincingly defeated a world-champion program in each case.*"

Deep Reinforcement Learning: An Overview (Li) – "*We give an overview of recent exciting achievements of deep reinforcement learning (RL). We discuss six core elements, six important mechanisms, and twelve applications. We start with background of machine learning, deep learning and reinforcement learning. Next we discuss core RL elements, including value function, in particular, Deep Q-Network (DQN), policy, reward, model, planning, and exploration.*"

Playing Atari with Deep Reinforcement Learning (DeepMind) – "*We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory*

input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them."

Human-Level Control Through Deep Reinforcement Learning (DeepMind) – "*The theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations.*"

Lectures and Videos Reinforcement Learning (Udacity, Georgia Tech) – "*You should take this course if you have an interest in machine learning and the desire to engage with it from a theoretical perspective. Through a combination of classic papers and more recent work, you will explore automated decision-making from a computer-science perspective. You will examine efficient algorithms, where they exist, for single-agent and multi-agent planning as well as approaches to learning near-optimal decisions from experience. At the end of the course, you will replicate a result from a published paper in reinforcement learning.*"

CS234: Reinforcement Learning (Stanford) – "*To realize the dreams and impact of AI requires autonomous systems that learn to make good decisions. Reinforcement learning is one powerful paradigm for doing so, and it is relevant to an enormous range of tasks, including robotics, game playing, consumer modeling and healthcare. This class will provide a solid introduction to the field of reinforcement learning and students will learn about the core challenges and approaches, including generalization and exploration.*"

CS 294: Deep Reinforcement Learning, Fall 2017 (Berkeley) – "*This course will assume some familiarity with reinforcement learning, numerical optimization and machine learning. Students who are not familiar with the concepts below are encouraged to brush up using the references provided right below this list. We'll review this material in class, but it will be rather*

cursory."

Advanced AI: Deep Reinforcement Learning in Python (Udemy) – "*This course is all about the application of deep learning and neural networks to reinforcement learning. If you've taken my first reinforcement learning class, then you know that reinforcement learning is on the bleeding edge of what we can do with AI. Specifically, the combination of deep learning with reinforcement learning has led to AlphaGo beating a world champion in the strategy game Go, it has led to self-driving cars, and it has led to machines that can play video games at a superhuman level.*"

Tutorials Simple Beginner'sGuide to Reinforcement Learning &ItsImplementation (Analytics Vidhya) – "*Today, we will explore Reinforcement Learning – a goal-oriented learning based on interaction with environment. Reinforcement Learning is said to be the hope of true artificial intelligence. And it is rightly said so, because the potential that Reinforcement Learning possesses is immense.*"

Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks (Arthur Juliani) – "*For this tutorial in my Reinforcement Learning series, we are going to be exploring a family of RL algorithms called Q-Learning algorithms. These are a little different than the policy-based algorithms that will be looked at in the the following tutorials (Parts 1–3). Instead of starting with a complex and unwieldy deep neural network, we will begin by implementing a simple lookup-table version of the algorithm, and then show how to implement a neural-network equivalent using Tensorflow.*"

A Tutorial for Reinforcement Learning (Abhijit Gosavi) – "*The tutorial is written for those who would like an introduction to reinforcement learning (RL). The aim is to provide an intuitive presentation of the ideas rather than concentrate on the deeper mathematics underlying the topic.*"

3 Q-learning vs SARSA

Q-learning is an off-policy reinforcement learning algorithm. In Q-learning and related algorithms, an agent tries to learn the optimal policy from its history of interaction with the environment.

We treat this history of interaction as a sequence of experiences, where an experience is a tuple $\langle s, a, r, s' \rangle$ which means that the agent was in state s, it did action a, it received reward r, and it went into state s. These experiences will be the data from which the agent can learn what to do. As in decision-theoretic planning, the aim is for the agent to maximize its value, which is usually the discounted reward.

```
1:  controller Q-learning(S, A, γ, α)
2:      Inputs
3:          S is a set of states
4:          A is a set of actions
5:          γ the discount
6:          α is the step size
7:      Local
8:          real array Q[S, A]
9:          previous state s
10:         previous action a
11:     initialize Q[S, A] arbitrarily
12:     observe current state s
13:     repeat
14:         select and carry out an action a
15:         observe reward r and state s'
16:         Q[s, a] ← Q[s, a] + α (r + γ max_a' Q[s', a'] − Q[s, a])
17:         s ← s'
18:     until termination
```

Figure 11.10: Q-learning controller

Q-learning learns an optimal policy no matter what the agent does, as long as it explores enough. There may be cases where ignoring what the agent actually does is dangerous (there will be large negative rewards). An alternative is to learn the value of the policy the agent is actually carrying out so that it can be iteratively improved. As a result, the learner can take into account the costs associated with exploration.

An **off-policy** learner learns the value of the optimal policy independently of the agent's actions. Q-learning is an off-policy learner. An on-policy learner learns the value of the policy being carried out by the agent, including the exploration steps.

SARSA (so called because it uses state-action-reward-state-action experiences to update the Q-values) is an **on-policy** reinforcement learning algorithm that estimates the value of the policy being followed. An experience in SARSA is of the form $\langle s, a, r, s', a' \rangle$, which means that the agent was in state s, did action a, received reward r, and ended up in state s, from

which it decided to do action a. This provides a new experience to update $Q(s,a)$. The new value that this experience provides is $r + \gamma Q(s',a')$.

SARSA takes into account the current exploration policy which, for example, may be greedy with random steps. It can find a different policy than Qlearning in situations when exploring may incur large penalties. For example, when a robot goes near the top of stairs, even if this is an optimal policy, it may be dangerous for exploration steps. SARSA will discover this and adopt a policy that keeps the robot away from the stairs. It will find a policy that is optimal, taking into account the exploration inherent in the policy.

controller SARSA(S, A, γ, α)
inputs:
    S is a set of states
    A is a set of actions
    γ the discount
    α is the step size
internal state:
    real array $Q[S,A]$
    previous state s
    previous action a
begin
    initialize $Q[S,A]$ arbitrarily
    observe current state s
    select action a using a policy based on Q
    repeat forever:
        carry out an action a
        observe reward r and state s'
        select action a' using a policy based on Q
        $Q[s,a] \leftarrow Q[s,a] + \alpha\left(r + \gamma Q[s',a'] - Q[s,a]\right)$
        $s \leftarrow s'$
        $a \leftarrow a'$
    end-repeat
end

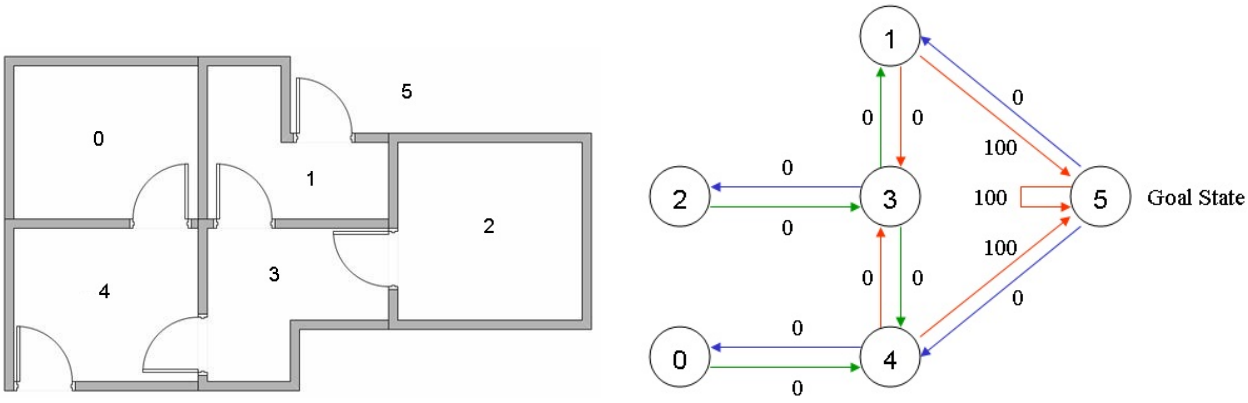
Figure 11.13: SARSA: on-policy reinforcement learning

4 Tutorial

4.1 Step-By-Step Tutorial

Suppose we have 5 rooms in a building connected by doors as shown in the figure below. We'll number each room 0 through 4. The outside of the building can be thought of as one big room (5). Notice that doors 1 and 4 lead into the building from room 5 (outside). We can represent the rooms on a graph, each room as a node, and each door as a link.

For this example, we'd like to put an agent in any room, and from that room, go outside the building (this will be our target room). In other words, the goal room is number 5. To set this room as a goal, we'll associate a reward value to each door (i.e. link between nodes). The doors that lead immediately to the goal have an instant reward of 100. Other doors not directly connected to the target room have zero reward. Because doors are two-way (0 leads to 4, and 4 leads back to 0), two arrows are assigned to each room. Each arrow contains an instant reward value, as shown below:



Of course, Room 5 loops back to itself with a reward of 100, and all other direct connections to the goal room carry a reward of 100. In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an "absorbing goal".

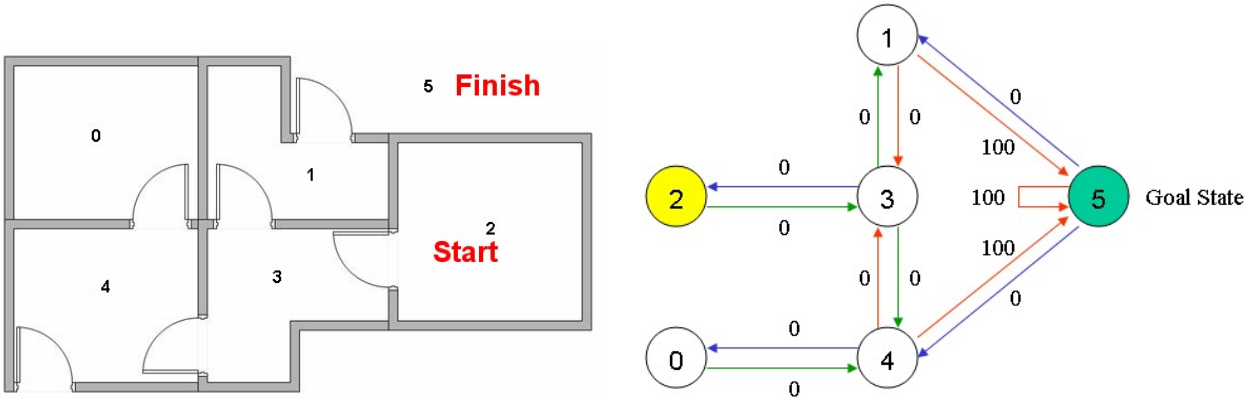
Imagine our agent as a dumb virtual robot that can learn through experience. The agent can pass from one room to another but has no knowledge of the environment, and doesn't know which sequence of doors lead to the outside.

Suppose we want to model some kind of simple evacuation of an agent from any room in the building. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the house (5).

The terminology in Q-Learning includes the terms "state" and "action".

We'll call each room, including outside, a "state", and the agent's movement from one room to another will be an "action". In our diagram, a "state" is depicted as a node, while "action" is represented by the arrows.

Suppose the agent is in state 2. From state 2, it can go to state 3 because state 2 is connected to 3. From state 2, however, the agent cannot directly go to state 1 because there



is no direct door connecting room 1 and 2 (thus, no arrows). From state 3, it can go either to state 1 or 4 or back to 2 (look at all the arrows about state 3). If the agent is in state 4, then the three possible actions are to go to state 0, 5 or 3. If the agent is in state 1, it can go either to state 5 or 3. From state 0, it can only go back to state 4.

We can put the state diagram and the instant reward values into the following reward table, "matrix R".

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} \text{Action} & & & & & \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

The -1's in the table represent null values (i.e.; where there isn't a link between nodes).

Now we'll add a similar matrix, "Q", to the brain of our agent, representing the memory of what the agent has learned through experience. The rows of matrix Q represent the current state of the agent, and the columns represent the possible actions leading to the next state (the links between the nodes).

The agent starts out knowing nothing, the matrix Q is initialized to zero. In this example, for the simplicity of explanation, we assume the number of states is known (to be six). If we didn't know how many states were involved, the matrix Q could start out with only one element. It is a simple task to add more columns and rows in matrix Q if a new state is found.

The transition rule of Q learning is a very simple formula:

$$Q(stte, action) = R(state, action) + \gamma \max[Q(nextstate, allactions)]$$

According to this formula, a value assigned to a specific element of matrix Q, is equal to the sum of the corresponding value in matrix R and the learning parameter γ, multiplied by the maximum value of Q for all possible actions in the next state.

Our virtual agent will learn through experience, without a teacher (this is called unsupervised learning). The agent will explore from state to state until it reaches the goal. We'll call each exploration an episode. Each episode consists of the agent moving from the initial state to the goal state. Each time the agent arrives at the goal state, the program goes to the next episode.

The Q-Learning algorithm goes as follows:

```
1  Set the gamma parameter, and environment rewards in matrix R;
2  Initialize matrix Q to zero;
3  foreach episode do
4  |   Select a random initial state;
5  |   while the goal state hasn't been reached do
6  |   |   Select one among all possible actions for the current state;
7  |   |   Using this possible action, consider going to the next state;
8  |   |   Get maximum Q value for this next state based on all possible actions;
9  |   |   Compute:
   |   |     Q(state, action) = R(state, action) + γ max[Q(nextstate, allactions)];
10 |   |   Set the next state as the current state;
11 |   end
12 end
```

Algorithm 1: The Q-Learning Algorithm

The algorithm above is used by the agent to learn from experience. Each episode is equivalent to one training session. In each training session, the agent explores the environment (represented by matrix R), receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the 'brain' of our agent, represented by matrix Q. More

training results in a more optimized matrix Q. In this case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same rooms, the agent will find the fastest route to the goal state.

The γ parameter has a range of 0 to 1 ($0 \leq \gamma < 1$). If γ is closer to zero, the agent will tend to consider only immediate rewards. If γ is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

To use the matrix Q, the agent simply traces the sequence of states, from the initial state to goal state. The algorithm finds the actions with the highest reward values recorded in matrix Q for current state:

Algorithm to utilize the Q matrix:

1. Set current state = initial state.
2. From current state, find the action with the highest Q value.
3. Set current state = next state.
4. Repeat Steps 2 and 3 until current state = goal state.

The algorithm above will return the sequence of states from the initial state to the goal state.

4.2 Q-learning Example By Hand

To understand how the Q-learning algorithm works, we'll go through a few episodes step by step. The rest of the steps are illustrated in the source code examples.

We'll start by setting the value of the learning parameter $\gamma = 0.8$, and the initial state as Room 1. Initialize matrix Q as a zero matrix. Look at the second row (state 1) of matrix R. There are two possible actions for the current state 1: go to state 3, or go to state 5. By random selection, we select to go to 5 as our action.

Now let's imagine what would happen if our agent were in state 5. Look at the sixth row of the reward matrix R (i.e. state 5). It has 3 possible actions: go to state 1, 4 or 5.

$$Q(state, action) = R(state, action) + \gamma \max[Q(nextstate, allactions)]$$

$$Q(1,5) = R(1,5) + 0.8 \times \max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 \times 0 = 100$$

Since matrix Q is still initialized to zero, Q(5, 1), Q(5, 4), Q(5, 5), are all zero. The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1).

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

The next state, 5, now becomes the current state. Because 5 is the goal state, we've finished one episode. Our agent's brain now contains an updated matrix Q as:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

For the next episode, we start with a randomly chosen initial state. This time, we have state 3 as our initial state.

Look at the fourth row of matrix R; it has 3 possible actions: go to state 1, 2 or 4. By random selection, we select to go to state 1 as our action.

Now we imagine that we are in state 1. Look at the second row of reward matrix R (i.e. state 1). It has 2 possible actions: go to state 3 or state 5. Then, we compute the Q value:

$$Q(state, action) = R(state, action) + \gamma \max[Q(nextstate, allactions)]$$

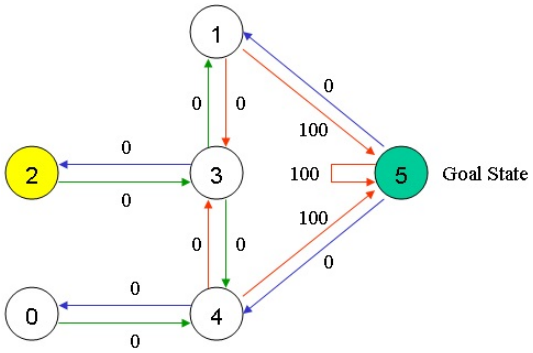
$$Q(1,5) = R(1,5) + 0.8 \times \max[Q(1,2), Q(1,5)] = 0 + 0.8 \times \max(0, 100) = 80$$

We use the updated matrix Q from the last episode. Q(1, 3) = 0 and Q(1, 5) = 100. The result of the computation is Q(3, 1) = 80 because the reward is zero. The matrix Q becomes:

The next state, 1, now becomes the current state. We repeat the inner loop of the Q learning algorithm because state 1 is not the goal state.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

So, starting the new loop with the current state 1, there are two possible actions: go to state 3, or go to state 5. By lucky draw, our action selected is 5.



Now, imaging we're in state 5, there are three possible actions: go to state 1, 4 or 5. We compute the Q value using the maximum value of these possible actions.

$$Q(state, action) = R(state, action) + \gamma \max[Q(nextstate, allactions)]$$

$$Q(1,5) = R(1,5) + 0.8 \times \max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 \times 0 = 100$$

The updated entries of matrix Q, Q(5, 1), Q(5, 4), Q(5, 5), are all zero. The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1). This result does not change the Q matrix.

Because 5 is the goal state, we finish this episode. Our agent's brain now contain updated matrix Q as:

If our agent learns more through further episodes, it will finally reach convergence values in matrix Q like:

This matrix Q, can then be normalized (i.e.; converted to percentage) by dividing all non-zero entries by the highest number (500 in this case):

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix} \end{array}$$

Once the matrix Q gets close enough to a state of convergence, we know our agent has learned the most optimal paths to the goal state. Tracing the best sequences of states is as simple as following the links with the highest values at each state.

For example, from initial State 2, the agent can use the matrix Q as a guide:

From State 2 the maximum Q values suggests the action to go to state 3.

From State 3 the maximum Q values suggest two alternatives: go to state 1 or 4. Suppose we arbitrarily choose to go to 1.

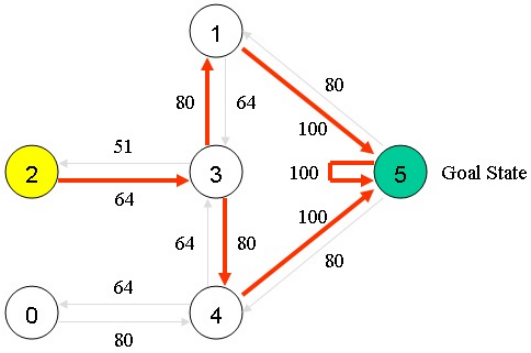
From State 1 the maximum Q values suggests the action to go to state 5.

Thus the sequence is 2 - 3 - 1 - 5.

5 Codes and Results

```
clear;clear all;clc;
global GAMMA; % Reward matrix
global Q; % Q-value matrix
global R; % learning parameter
GAMMA=0.8;% learning rate
Q=zeros(6,6); % all zero
R=[[-1,-1,-1,-1,0,-1],
[-1,-1,-1,0,-1,100],
[-1,-1,-1,0,-1,-1],
[-1,0,0,-1,0,-1],
```

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$




```
[0,-1,-1,0,-1,100],
[-1,0,-1,-1,0,100]];

count=0;
while count<2000 % 2000 iterations
        for i = 1:6
                QLearning(i);
                count=count+1;
                max1=max(Q);
    end
end
disp('Q is :')
disp(Q./max(max(Q))*100);
route=getroute(3,6); % get route from 3 to 6
disp('The result is:');
disp(route);

function [s]=getMaxQ(state) % get the maximum Q at some state
    global Q;
        s=max(Q(state,:));
end

```

```
function QLearning(state)
        curAction=-1;
    global GAMMA;
    global Q;
    action=unidrnd(6);% generate action 1-6 at random
    global R;
        if R(state,action)==-1
                        Q(state,action)=0;
    else
                curAction=action;
        % iteration
        Q(state,action)=R(state,action)+GAMMA*getMaxQ(curAction);
    end
end

function [index]=getIndex(start,num)
        index=1;
    global Q;
        for i = 1:6
                if Q(start,i)==num
                        index=i;
                        break;
        end
    end
end

function [route]=getroute(start,endd)% generate the route
        route=[];
        route=[route,start];
        num=getMaxQ(start);
        next=getIndex(start,num);
        while next ~= endd
                route=[route,next];
                num=getMaxQ(next);
                next=getIndex(next,num);
    end
        route=[route,endd];
```

```
end
```

Note that index of an arrow cannot be 0 in MATLAB, so the route is 3-4-2-6. The result is as following:

```
Q is :
         0         0         0         0   79.9999         0
         0         0         0   63.9997         0  100.0000
         0         0         0   63.9999         0         0
         0   79.9999   51.1992         0   79.9999         0
   63.9999         0         0   63.9990         0  100.0000
         0   79.9999         0         0   80.0000   99.9998

The result is:
     3     4     2     6
```

6 Reference

- https://blog.algorithmia.com/introduction-to-reinforcement-learning/
- https://blog.csdn.net/qq_28168421/article/details/81784563
- http://mnemstudio.org/path-finding-q-learning-tutorial.htm
- https://blog.csdn.net/itplus/article/details/9361915