Deep Q-Network

Reinforcement Learning with TensorFlow&OpenAI Gym

Youngho Byeon - July 23, 2017

1. Introduction

Deep Q-Network

Deep Q-Network(DQN) is an algorithm developed by Google DeepMind, which uses deep learning network. The algorithm was published in Nature in February 2015. It's reinforcement learning network that can play Atary games like a human being. The amazing part of this algorithm is simple and powerful. If you just tell a agent that "high score is your goal", the agent will gradually understand game rules and find out a best strategy.

In my country, South Korea, there was a Go challenge match with AlphaGo and Sedol Lee in last year. Go is a very ancient game and it's one of the most complex game. AlphaGo is Go game program that used DQN model developed by Google DeepMind. As you know, the result was a victory of AlphaGo. This was a very important milestone of Artificial Intelligence.

Q-learning

Before we start with DQN, we should check Q-learning algorithm because DQN based in Q-learning. And basic Q-learning algorithm has some problems. So, we will check these problems. And then we will find out how DQN solves the problems. So, we will go through a following process:

Step 1. Q-leaning by Q-table

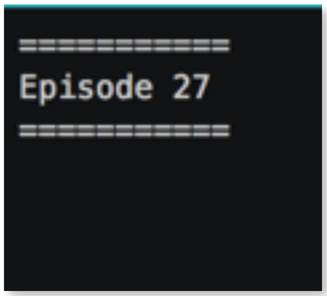
Step 2. Q-leaning by improved Q-table

Step 3. Q-learning by Neural network

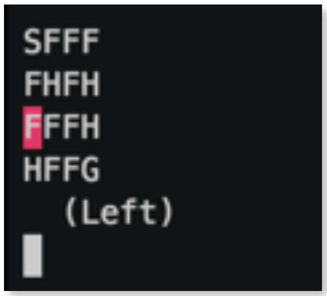
Step 4. Q-learning by DQN

OpenAI Gym

Also we can use OpenAI Gym for this project. It is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Go.[1] Especially, we will build this project by 'FrozenLake' and 'CartPole' games.



```
===========
Episode 27
===========
```



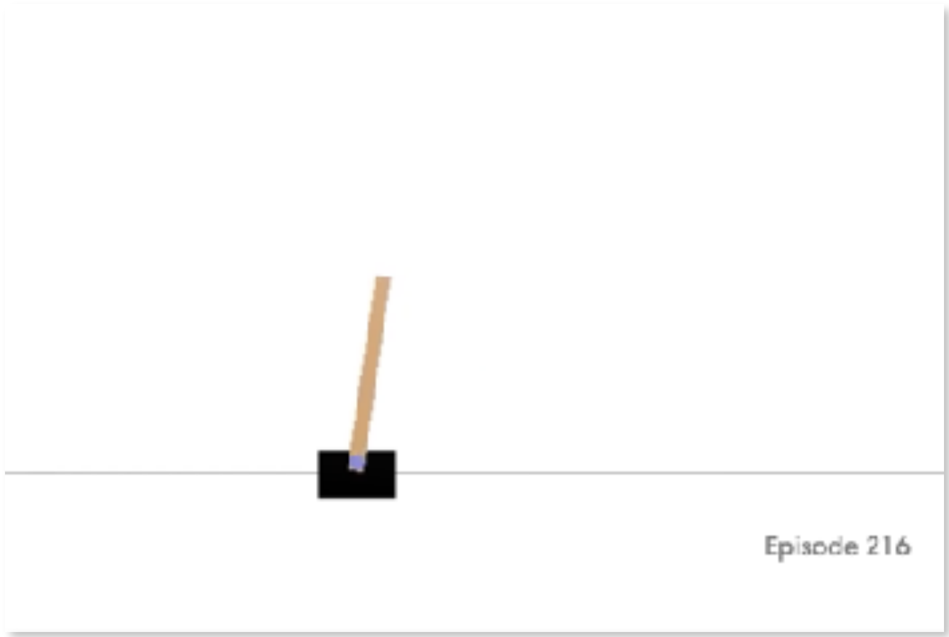
```
SFFF
FHFH
FFFH
HFFG
  (Left)
```

FrozenLake is that a agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.[2] So, in this game,

1 https://gym.openai.com/

2 https://gym.openai.com/envs/FrozenLake-v0

inputs are movement directions of the agent (Up, Down, Left, Right). And, there are 4 states in the grid: Starting point, Frozen surface, Hole, Goal. And in this game, the episode ends when you reach the goal or fall in a hole. The agent receive a reward of 1 if it reach the goal, and zero otherwise. So, we will find a solution that receive the reward of 1.



CartPole is that a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright.[3] For this game, we have four inputs, one for each value in the state, and two outputs. And in this game, the episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. According to CartPole document, it defines "solving" as getting average reward of 195.0 over 100 consecutive trials. So, our goal is same.

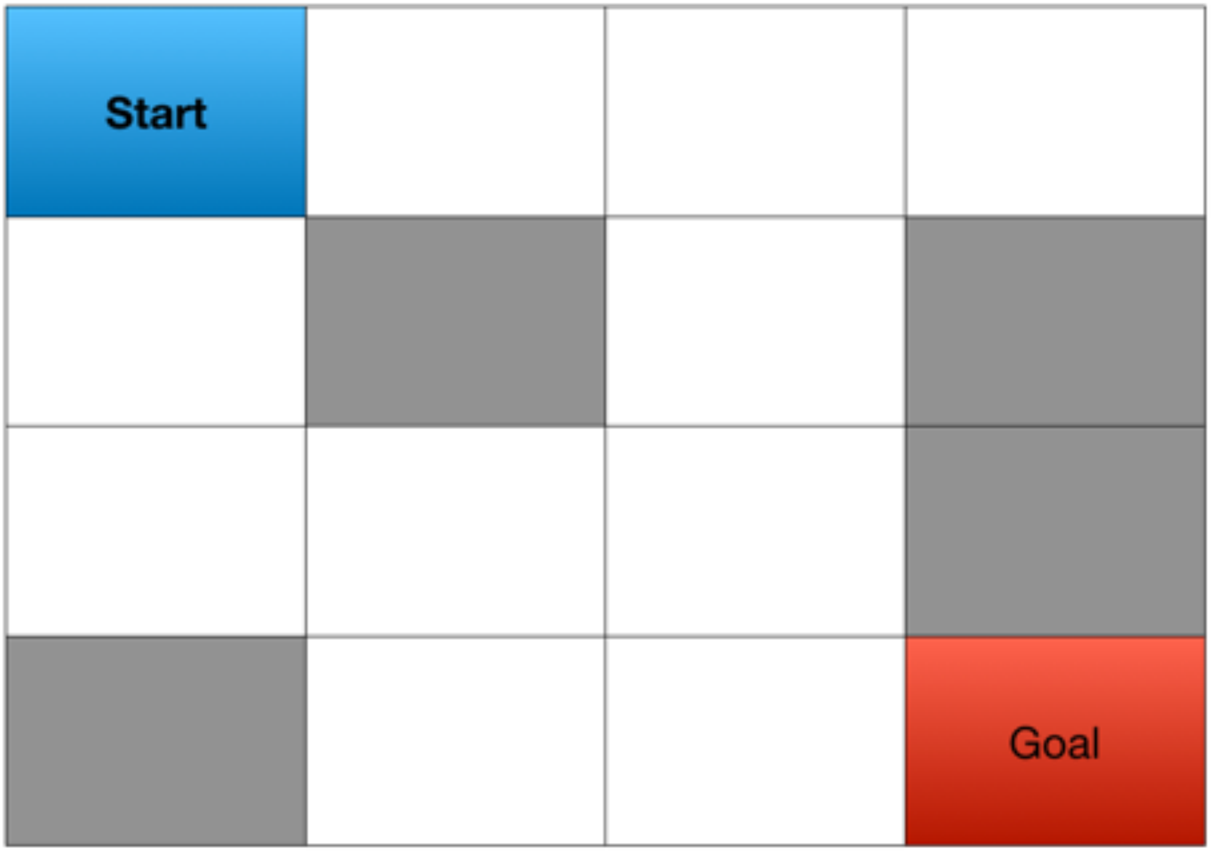
[3] https://gym.openai.com/envs/CartPole-v0

2. Implement

Q-table

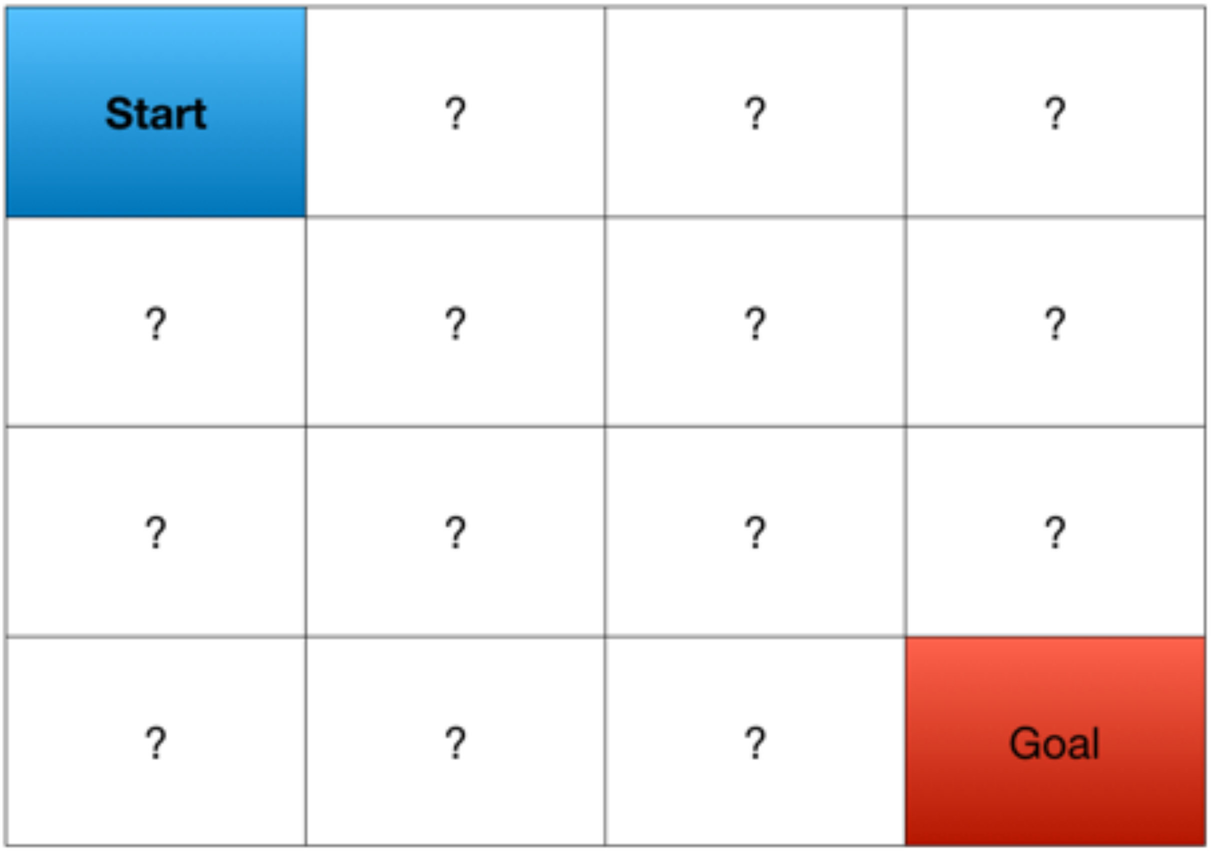
We can simulate this game using OpenAI Gym. FrozenLake is that a agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile. In the FrozenLake game, there are 4 possible actions, moving the agent up, down, left or right. And there are two rewards we can take, encoded as 0 and 1.

Here are the FrozenLake game map.



In this game, the episode ends when you reach the goal or fall in a hole. The agent receive a reward of 1 if it reach the goal, and zero otherwise. So, our final goal is that we get a reward of 1. It looks like very simple game. And we can find a best route easily. But it isn't simple. Because our agent doesn't know where the holes are.

Here is the REAL FrozenLake map for the agent.

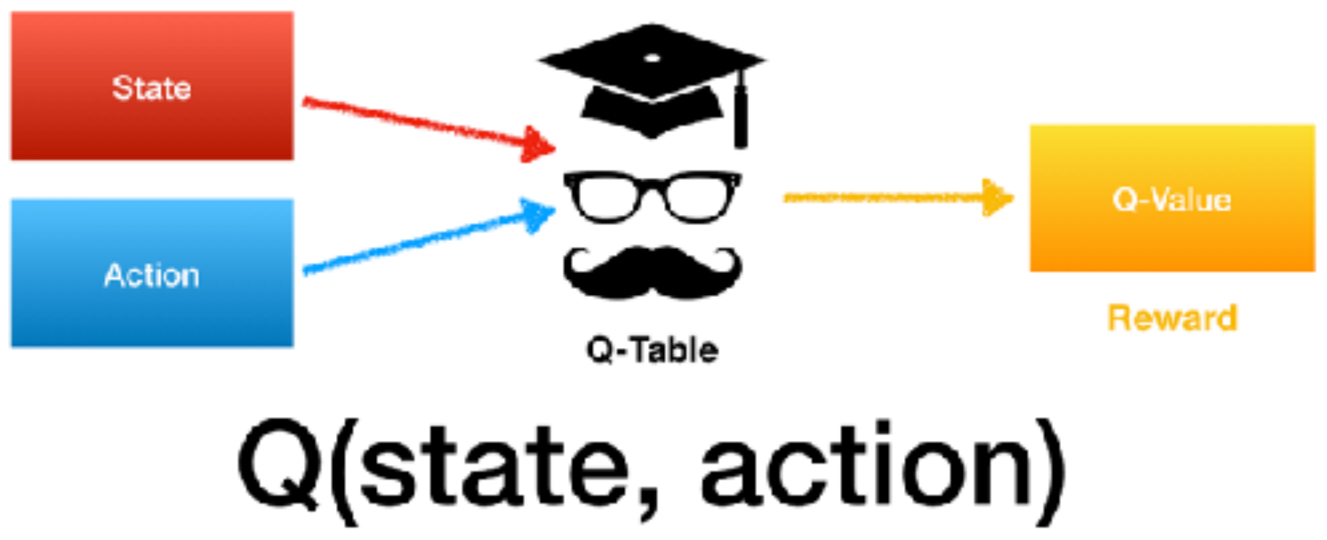


We are stuck. How do we know where are walkable tiles? We can train the agent by random actions. But as you know, it isn't efficient. So, we need other solution. The solution is that:

Even if you know the way, ask one more time. - Korean proverbs

Before we action, the agent should ask to someone. This is Q-learning. Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Additionally, Q-learning can handle problems with stochastic transitions and rewards, without requiring any adaptations. It has been proven that for any finite MDP, Q-learning eventually finds an optimal policy, in the sense that the expected value of the total reward return over all successive steps, starting from the current state, is the maximum achievable.

There is a simple Q-learning function. If we give the state and action as parameters, the function will return optimized value.



Here is an example. Suppose the agent is on the 's1'.

Q-function: Q(state, action)	Reward
Q(s1, Up)	0
Q(s1, Down)	0.3
Q(s1, Left)	0
Q(s1, Right)	0.5

This table shows reward according to each actions on the 's1'. We can choose a action that gives the greatest reward. In this case, it is 'Right'. So, here is a equation:

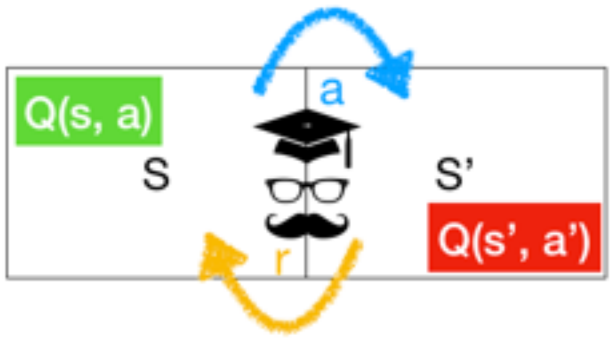
$$\pi^*(s) = \operatorname{argmax} Q(s, a)$$

where s is a state, a is an action, and π is a policy, * is meaning of optimization. So, if r is a reward, we can calculate sum of rewards:

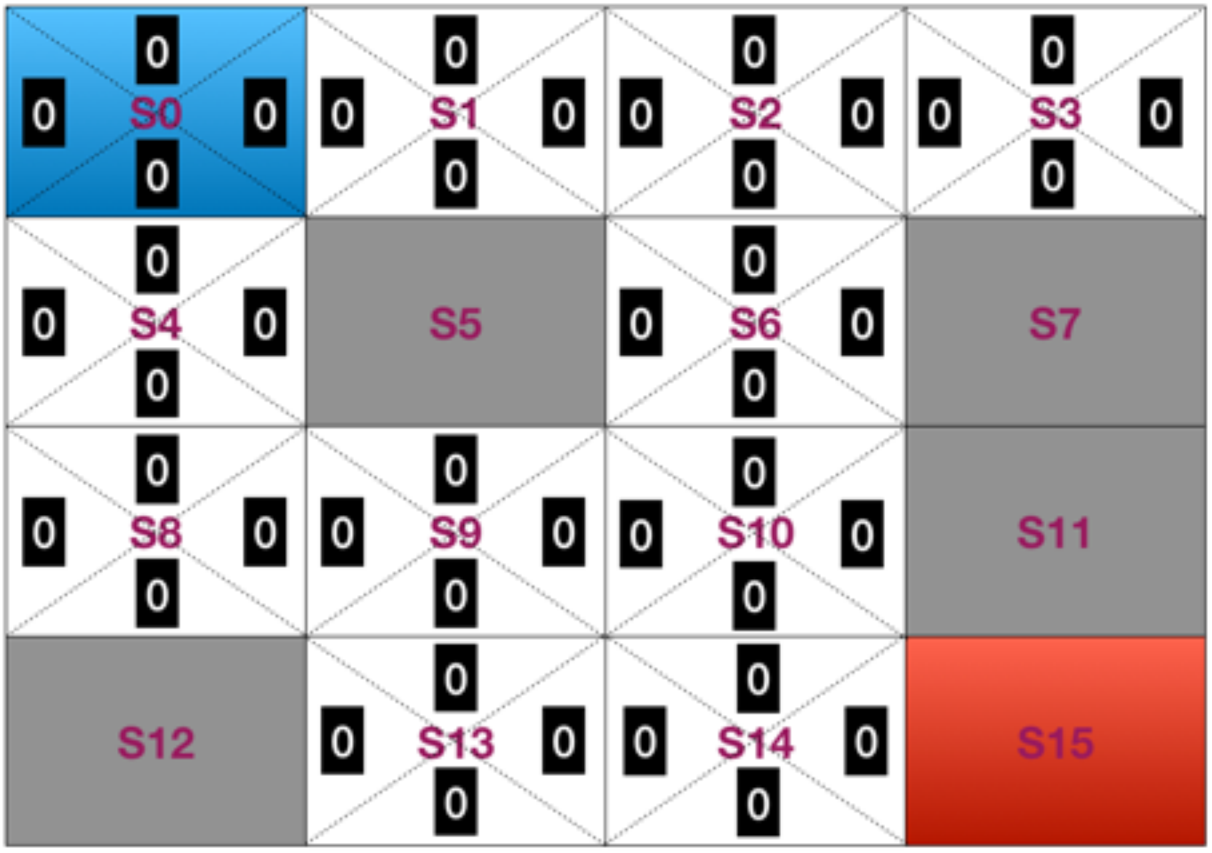
$$R = r_1 + r_2 + r_3 + \ldots + r_n \qquad R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n \qquad R_t = r_t + R_{t+1}$$

When s' is the next state from state s and action a. We should assume $Q(s', a')$ is exist. So, We train our Q-learning agent using the equation:

$$Q(s, a) = r + \max Q(s', a')$$

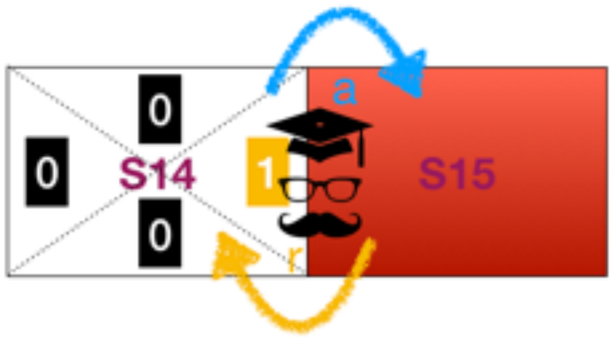


Finally, we get a optimize function $Q(s, a) = r + \max Q(s', a')$. Let's dive in to the deep. First of all, we should initialize by zero.



And we should update our Q-table by many trials. The agent can receive a reward of 1 only when move to the right on s_{14}. So, before the agent arrives on s_{14}, we couldn't get any reward(always get a reward of zero). And return value of Q-function is also zero. So, we shouldn't update any value. But, after many trials, let's assume that the agent is on s_{14}. In this case, finally we can get a reward. because $Q(s_{14}, a_{right})$ will return a reward of 1. And, $\max(Q(s_{15}, a))$ is 0. Because, s_{15} cell is our goal. So, we should update $Q(s_{14}, a_{right}) = 1$.

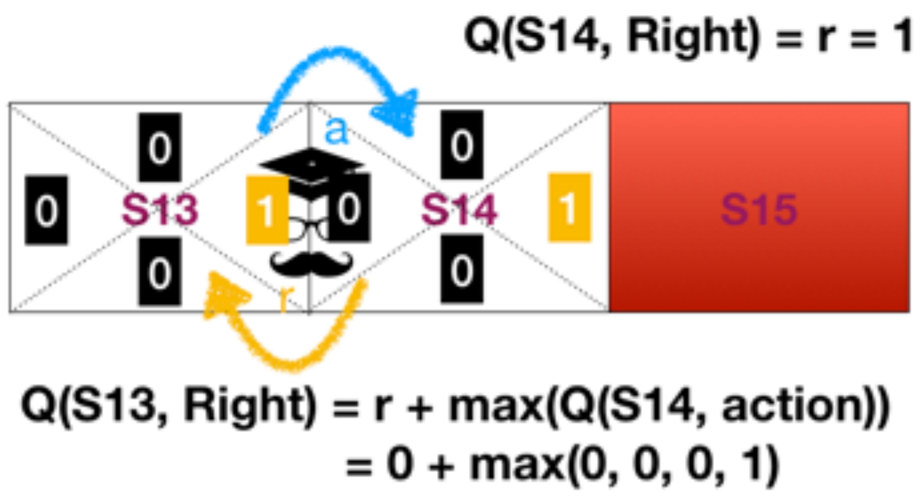
$$Q(s, a) = r + \max Q(s', a')$$
$$Q(s_{14}, a_{right}) = 1 + 0 = 1$$



Q(S14, Right) = r = 1

After s_{14} state was updated, let's assume that the agent is on s_{13} like before example(s_{14}). In this case, $Q(s_{13}, a_{right})$ couldn't get a reward. But, $\max Q(s', a')$ is 1. Because, s' is s_{14} . So, $\max Q(s', a')$ is $Q(s_{14}, a_{right})$. So, we should update $Q(s_{13}, a_{right}) = 1$.

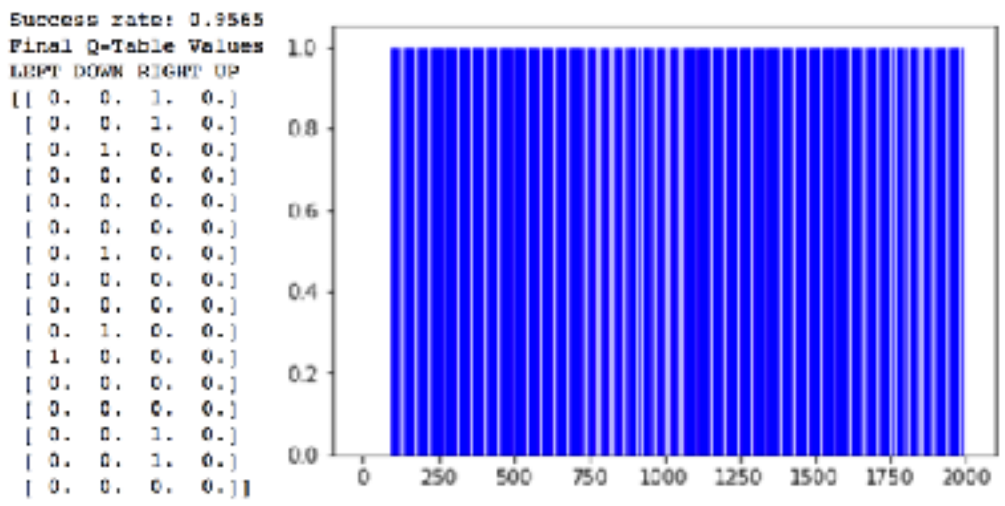
$$Q(s, a) = r + \max Q(s', a')$$
$$Q(s_{13}, a_{right}) = r + \max Q(s_{14}, a') = 0 + Q(s_{14}, a_{right}) = 1$$



In conclusion, we can summarize our first algorithm:

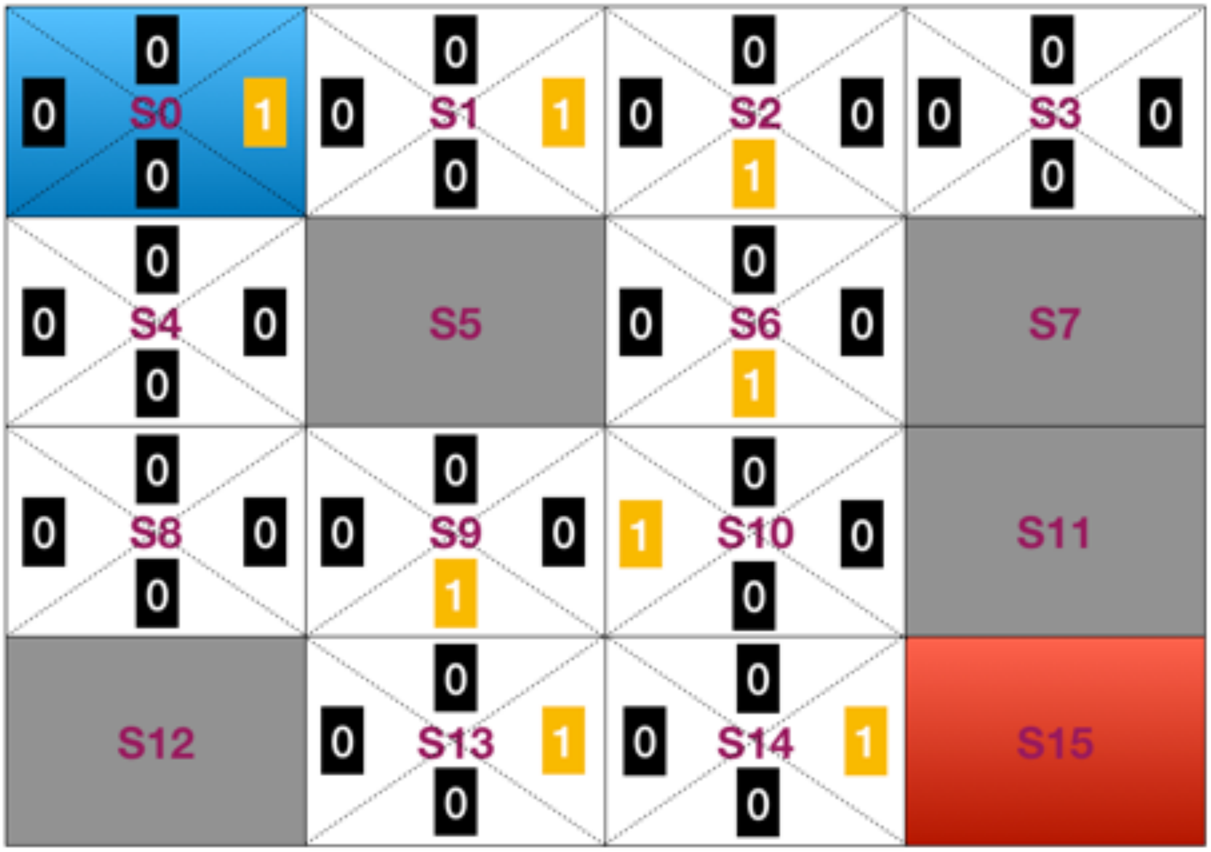
```
For each $s$, $a$ initialize table entry $\hat{Q}(s,a) \leftarrow 0$.
Observe current state $s$
Do forever:
    Select an action $a$ and execute it
    Receive immediate reward $r$
    Observe the new state $s'$
    Update the table entry for $\hat{Q}(s,a)$ as follows:
        $\hat{Q}(s,a) \leftarrow r + \max \hat{Q}(s',a')$
    $s \leftarrow s'$
```

And then, we can implement this algorithm. Here is the result:

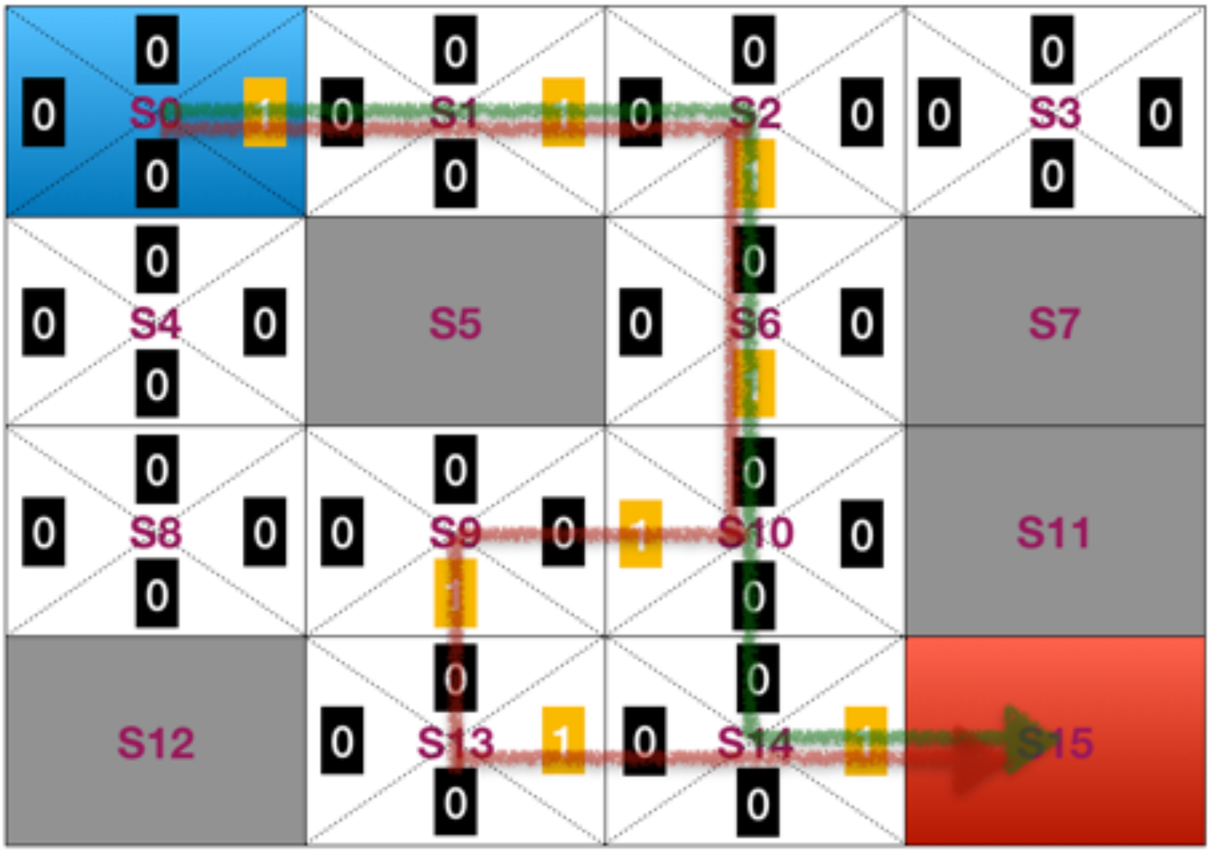


```
Success rate: 0.9565
Final Q-Table Values
LEFT DOWN RIGHT UP
[[ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  0.]]
```

And here is the FrozenLake map that updated all the states.



But, we can find a problem. The problem is that the agent's route isn't optimized.



Our route is red line. But, optimized route is green line. Although we got a correct reward of 1, it isn't efficient route. In our case, to learn about the environment and rules of the game, the agent needs to explore by taking random actions even though these actions haven't optimal values.

Optimized Q-table with Exploit and Exploration

Why isn't the first algorithm optimized? Let's see the algorithm again:

$$\pi^*(s) = \text{argmax} Q(s, a)$$

where s is a state, a is an action, and π is a policy, * is meaning of optimization. According to this equation, the agent should move to next state that has the greatest value. So, although other route has more efficient way, the agent always moves in the same way. This is the problem of our first algorithm.

To improve our algorithm, although other state has small value, sometimes the agent should move to there. Let's call this method as **Exploit VS Exploration**. A simple key of this strategy is that sometimes the agent should move randomly. Here is a example in real life. I'm a big fan of Burger King Whopper. And I like McDonald's Big Mac too. But, I don't like KFC and Popeyes. If I express this as a score, here is my preference of food chains.

KFC
0.3
BURGER KING
0.9
Popeyes CHICKEN & BISCUITS
0.2
0.7

Even though I want Big Mac to eat sometimes, according to our first algorithm, I will always go to Burger King. Because Burger King has the greatest preference value. And, I can't find a new delicious hamburger in other food chains. So, I decided to change my strategy. I will go to Burger King in weekdays, because it is my best food chain **(Exploit)**. But I will go to other food chain in weekend to find a new delicious hamburger **(Exploration)**. There are two ways to improve our strategy.

The first is called an ϵ**-greedy policy**. The main concept of this method is that set a small probability. Let's call this probability ϵ(epsilon). The agent will choose random action with ϵ probability. Contrary, it will choose an action from $Q(s, a)$ with $1 - \epsilon$ probability.

```
e = 0.1
if rand < e:
    a = random
else:
    a = argmax(Q(s, a))
```

So, I will go to Burger King with a 90% chance. And I will go to other food chains with 10% chance. Like this, the agent will follow $Q(s, a)$ with 90% chance. And the agent will find new way with a 10% chance. But, this strategy isn't efficient. Because, as learning progresses, the agent need to find a new way less and less. So, at first, the agent needs to do a lot of exploring. Later when it has learned more, the agent can favor choosing actions based on what it has learned. This is called exploitation. We'll set it up so the agent is more likely to explore early in training, then more likely to exploit later in training. The is called an **decaying ϵ-greedy policy.**

```
for i in range(1000):
    e = 0.1 / (i+1)

    if random(1) < e:
        a = random
    else:
        a = argmax(Q(s, a))
```

According to this, as time passes, ϵ is decreased. So, later when it has learned more, the agent can choose action based on what it has learned than random action.

And the second is called an **add random noise**. The concept of this method is that just add random value to each action.

KFC
0.3 + 0.2
BURGER KING
0.9 + 0.1
Popeyes CHICKEN & BISCUITS
0.2 + 0.4
0.7 + 0.5

$$a = \text{argmax}Q(s,a) \; + \text{RandomValue}$$
$$a = \text{argmax}([0.3,0.9,0.2,0.7] + [0.2,0.1,0.4,0.5])$$

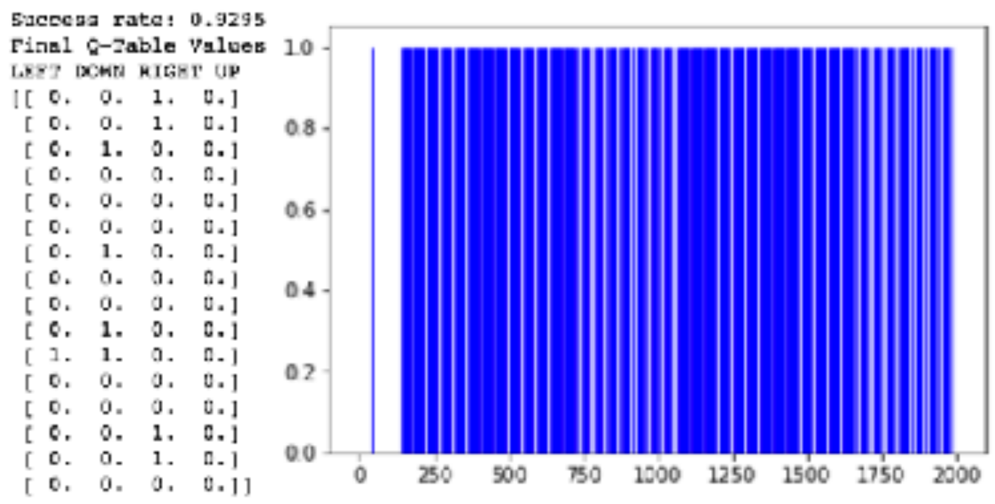
In this case, we will choose McDonald's. And also we can use decaying policy with add random noise.

```
for i in range(1000):
    a = argmax(Q(s, a) + random_value / (i+1))
```

Comparing **ϵ-greedy policy** and **add random noise**, when the agent choose random action in ϵ-greedy policy, it is totally random. But same case in add random noise, since the noise_values are added to the existing $Q(s,a)$ values, it is affected by Q-learning function value even though the agent will choose random action. So, the probability that an action has a high Q-leaning value is selected is higher than ϵ-greedy policy. Let's check our first algorithm again:

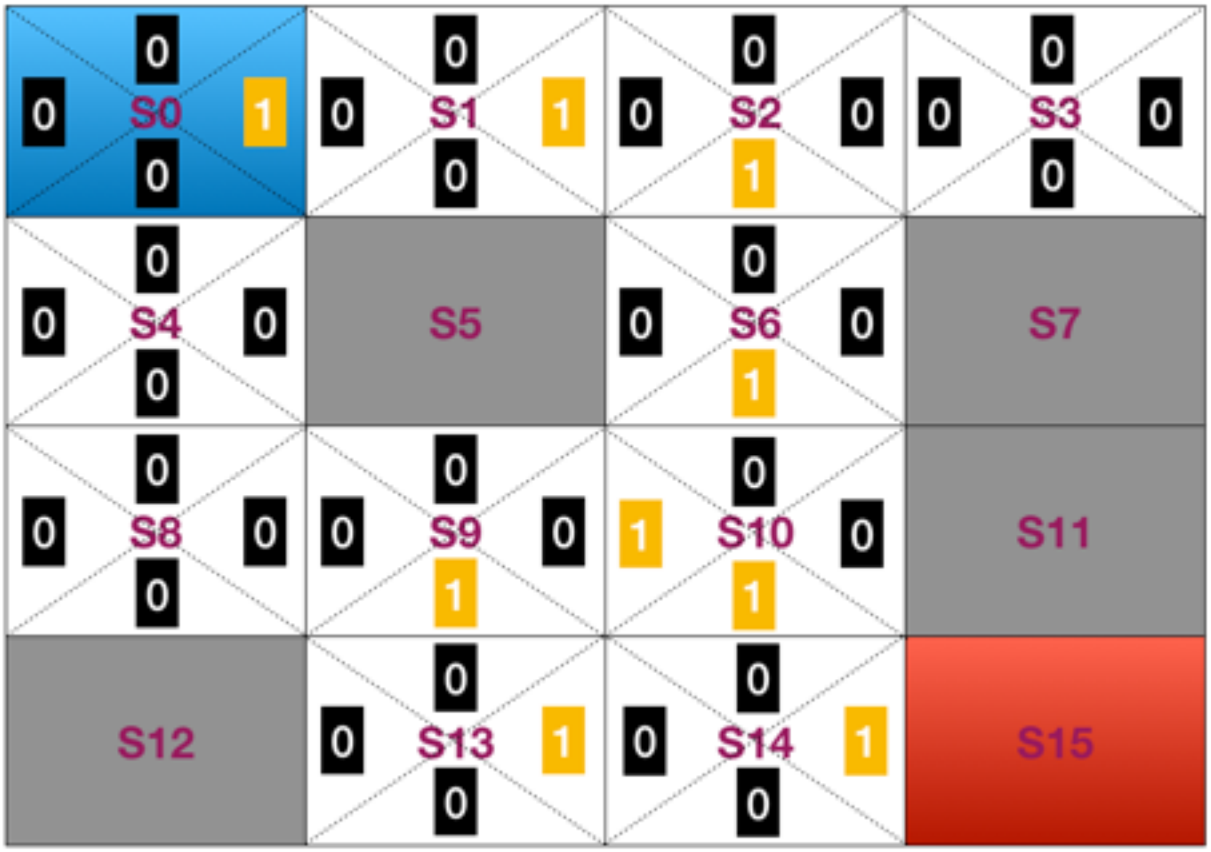
```
For each $s$, $a$ initialize table entry $\hat{Q}(s,a) \leftarrow 0$.
Observe current state $s$
Do forever:
        **Select an action $a$ and execute it**
        Receive immediate reward $r$
        Observe the new state $s'$
        Update the table entry for $\hat{Q}(s,a)$ as follows:
                $\hat{Q}(s,a) \leftarrow r + \max \hat{Q}(s',a')$
        $s \leftarrow s'$
```

In this algorithm, we didn't tell how to select an action and execute it. Now, the agent will select an action with **ϵ-greedy policy** or **add random noise**. And then, we can implement this algorithm. Here is the result:

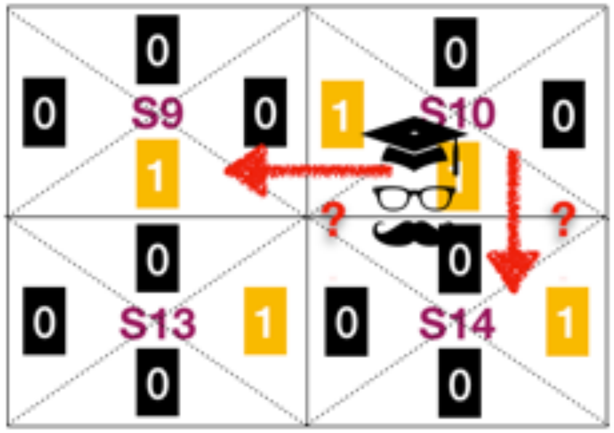


```
Success rate: 0.9295
Final Q-Table Values
LEFT DOWN RIGHT UP
[[ 0.   0.   1.   0.]
 [ 0.   0.   1.   0.]
 [ 0.   1.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   1.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   1.   0.   0.]
 [ 1.   1.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   1.   0.]
 [ 0.   0.   1.   0.]
 [ 0.   0.   0.   0.]]
```

And here is the map that updated all the states.



But, this algorithm has some problems too. If the agent is on s_{10}, where should the agent go?

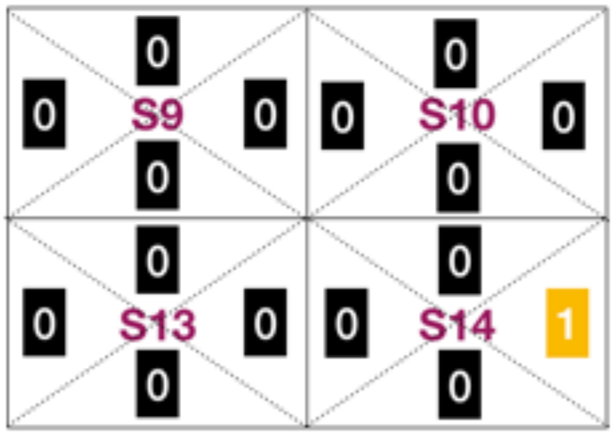


Optimized Q-table with Discounted reward

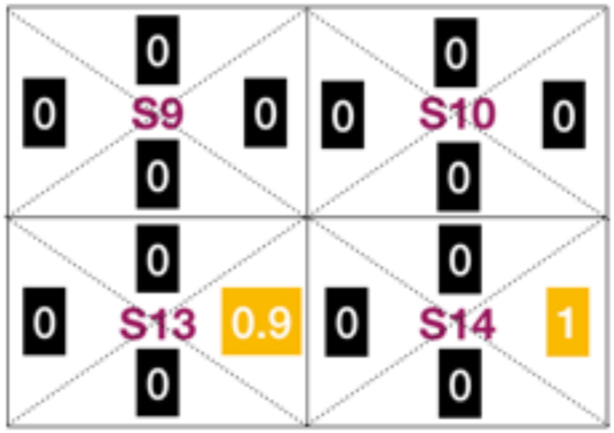
So, we need a solution. The problem with our algorithm is that all the reward is 1. There are two reward in this game: present reward and future reward. However, our algorithm doesn't distinguish between them. So, we should update our algorithm to distinguish current reward and future reward. We will give a weight to the present reward. Getting immediate reward is better than delayed reward. Here is our new Q-learning equation:

$$Q(s, a) = r + \gamma \max Q(s', a')$$

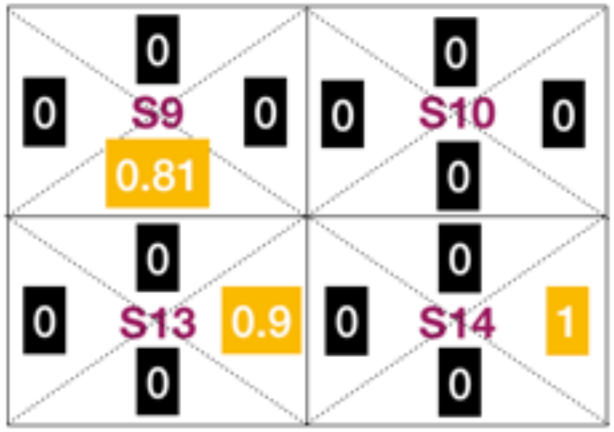
where γ is weight. In this case, r is present reward and $\max Q(s', a')$ is future reward. Because we need reduce future reward, γ should be less than 1. Let's update the map again using this formula when γ is 0.9.



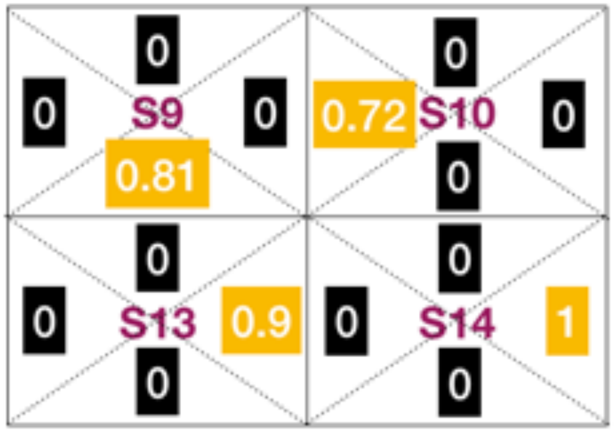
s_{15} is our goal. So, $Q(s_{14}, a_{right})$ is 1.



$$Q(s,a) = r + \gamma \max Q(s',a')$$
$$Q(s_{13},a) = 0 + 0.9 \cdot Q(s_{14},a_{right}) = 0.9$$

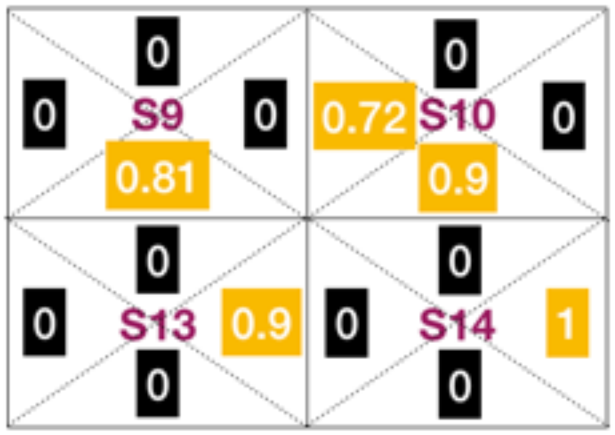


$$Q(s,a) = r + \gamma \max Q(s',a')$$
$$Q(s_9,a) = 0 + 0.9 \cdot Q(s_{13},a_{right}) = 0.9 \cdot 0.9 = 0.81$$



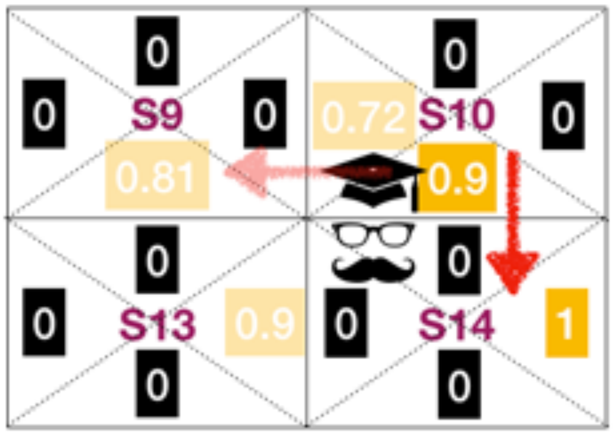
$$Q(s,a) = r + \gamma \max Q(s',a')$$
$$Q(s_{10},a) = 0 + 0.9 \cdot Q(s_9,a_{down}) = 0.9 \cdot 0.9 \cdot 0.9 = 0.72$$

Like this way, we can update $Q(s_{10},a_{down})$.



$$Q(s,a) = r + \gamma \max Q(s',a')$$
$$Q(s_{10},a) = 0 + 0.9 \cdot Q(s_{14},a_{right}) = 0.9$$

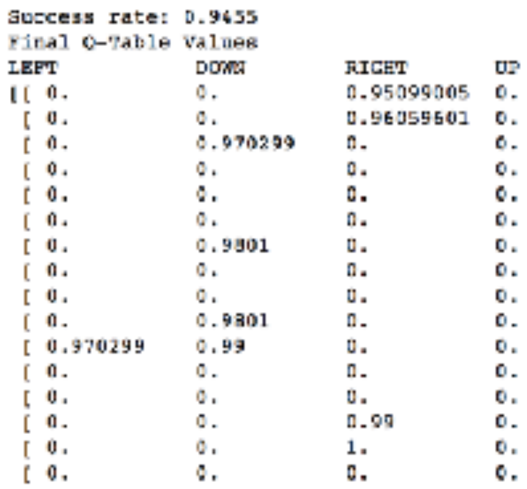
Finally, the agent can distinguish optimized route. On s_{10}, the agent will choose a_{down}.



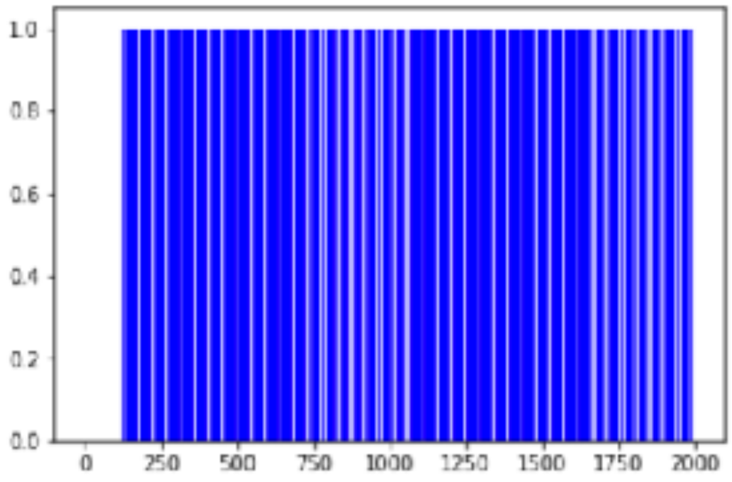
In conclusion, we can summarize our final algorithm:

```
For each $s$, $a$ initialize table entry $\hat{Q}(s,a) \leftarrow 0$.
Observe current state $s$
Do forever:
    Select an action $a$ and execute it
    Receive immediate reward $r$
    Observe the new state $s'$
    Update the table entry for $\hat{Q}(s,a)$ as follows:
        $\hat{Q}(s,a) \leftarrow r + \gamma \max \hat{Q}(s',a')$
    $s \leftarrow s'$
```

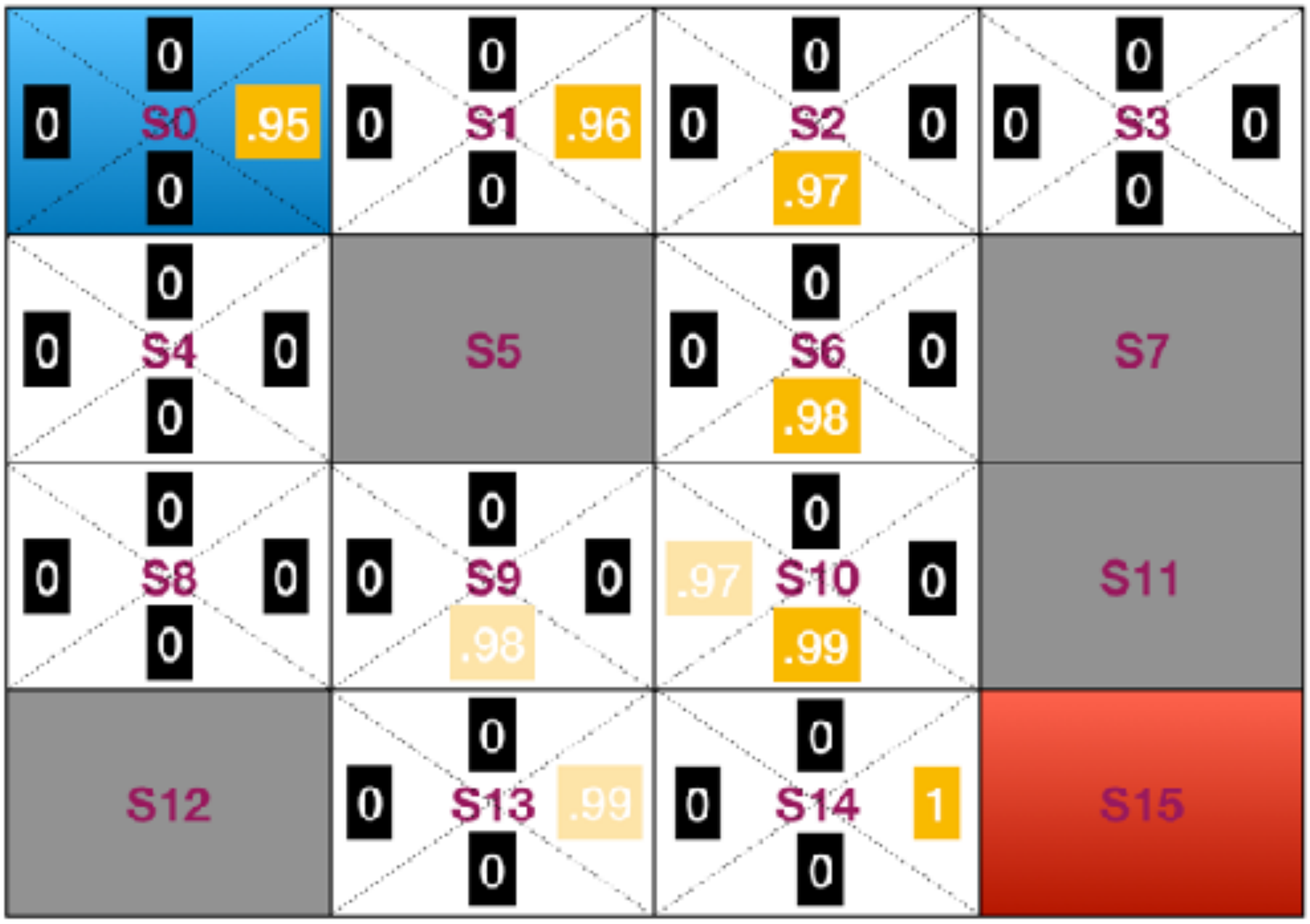
And here is a result:



```
Success rate: 0.9455
Final Q-Table Values
LEFT                 DOWN                 RIGHT                UP
[[ 0.                0.                   0.95099005           0.
 [ 0.                0.                   0.96059601           0.
 [ 0.                0.970299             0.                   0.
 [ 0.                0.                   0.                   0.
 [ 0.                0.                   0.                   0.
 [ 0.                0.                   0.                   0.
 [ 0.                0.9801               0.                   0.
 [ 0.                0.                   0.                   0.
 [ 0.                0.                   0.                   0.
 [ 0.                0.9801               0.                   0.
 [ 0.970299          0.99                 0.                   0.
 [ 0.                0.                   0.                   0.
 [ 0.                0.                   0.                   0.
 [ 0.                0.                   0.99                 0.
 [ 0.                0.                   1.                   0.
 [ 0.                0.                   0.                   0.
```



And here is the reward map.



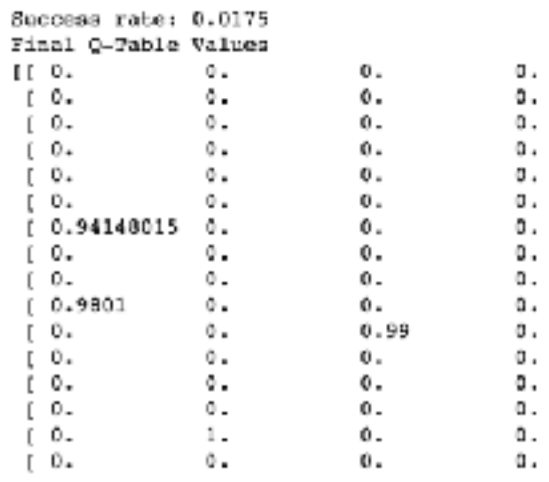
Q-table in real world

Like the name of the game 'Frozenlake', let's think we're on an real frozen lake. The place will be very slippery. So, although we should move carefully, we couldn't move accurately. For instance, we tried to move to right, but we could slip down. In previous environment, the agent could move accurately. Because it is artificial environment. But we can set more realistic environment. We call this environment as **Stochastic model**.

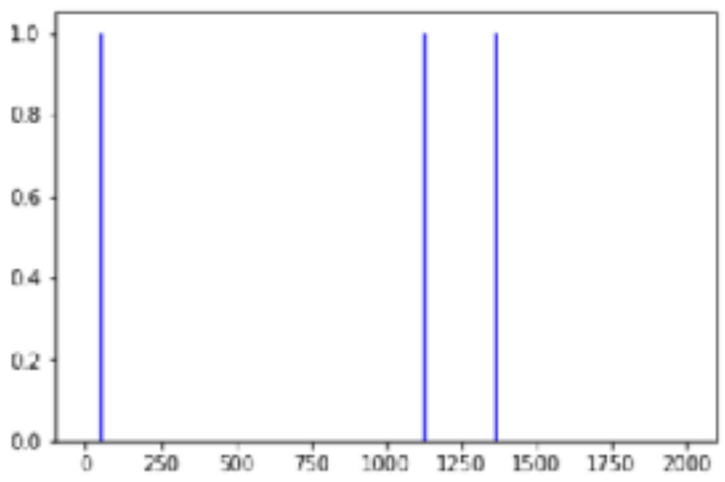
Deterministic models : the output of the model is fully determined by the parameter values and the initial conditions initial conditions.

Stochastic models : possess some inherent randomness. The same set of parameter values and initial conditions will lead to an ensemble of different outputs.

In this model, the agent doesn't move as we expected. Here is a result that apply our previous algorithm.



```
Success rate: 0.0175
Final Q-Table Values
[[ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.94148015  0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.9801      0.          0.          0.
 [ 0.          0.          0.99        0.
 [ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.          0.          0.          0.
 [ 0.          1.          0.          0.
 [ 0.          0.          0.          0.
```



Totally failed. We need new solution. The solution is simple:

Two heads are better than one

Like our life mentors. Don't just listen and follow one mentor. we need our own solutions. In previous model, we just accept Q-function completely. So, our new algorithm is that listen to $Q(s')$ just little bit(mentor's solution) and update $Q(s)$ little bit(our own solution). So, we will update $Q(s)$ by learning rate.

There is our previous equation:

$$Q(s,a) \leftarrow r + \gamma \max Q(s',a')$$

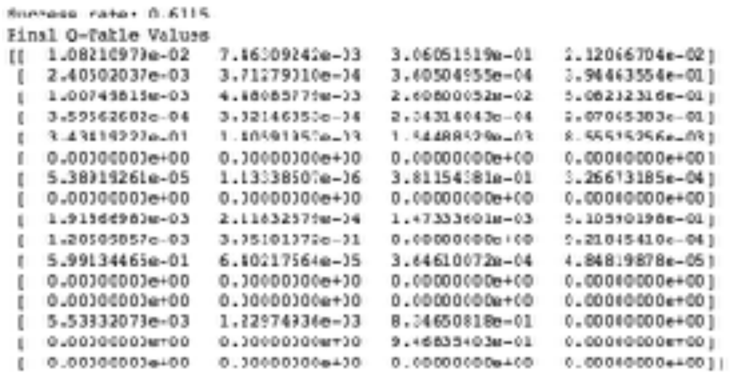
And Here is a new equation:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max Q(s',a')]$$
$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$$

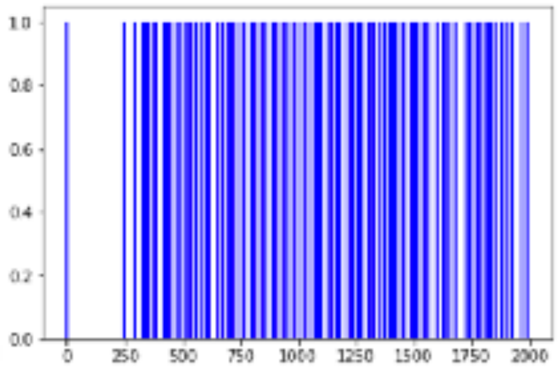
where α is learning rate. And, let's update our algorithm:

```
For each $s$, $a$ initialize table entry $\hat{Q}(s,a) \leftarrow 0$.
Observe current state $s$
Do forever:
    Select an action $a$ and execute it
    Receive immediate reward $r$
    Observe the new state $s'$
    Update the table entry for $\hat{Q}(s,a)$ as follows:
        $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max Q(s',a')]$
    $s \leftarrow s'$
```

And here is a result:



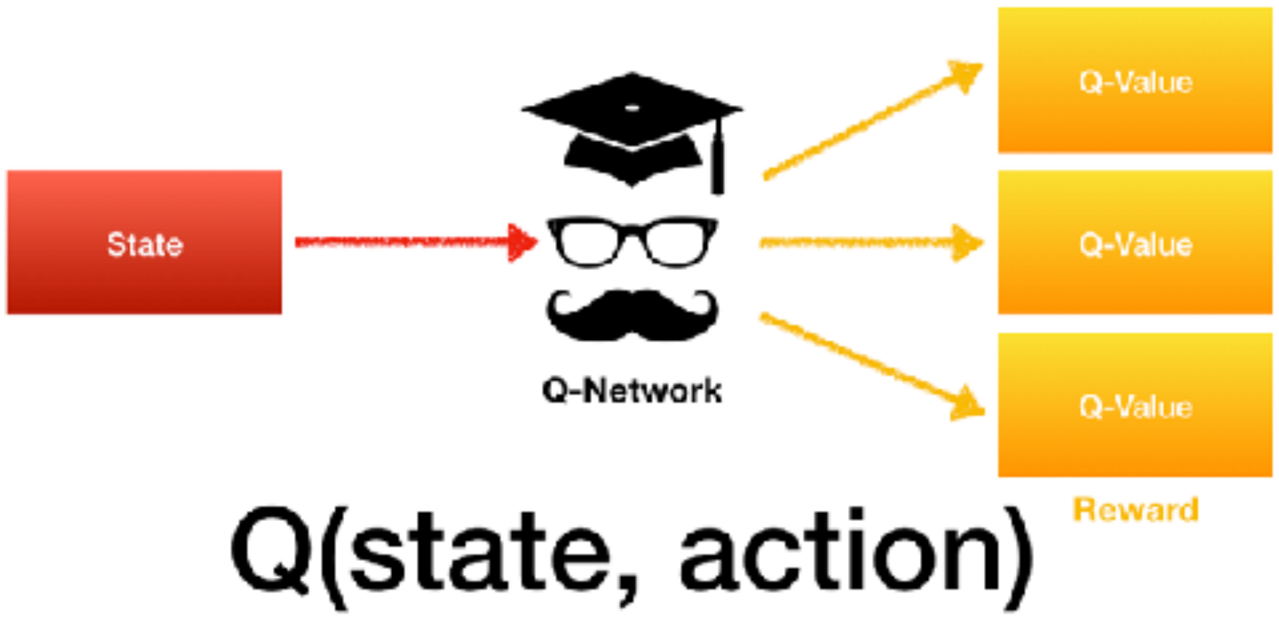
```
Success rate: 0.6115
Final Q-Table Values
[[  1.08210973e-02   7.16309242e-03   3.06051519e-01   1.12066704e-02]
 [  2.40502037e-03   3.71279310e-04   3.60504555e-04   3.94463554e-01]
 [  1.00749815e-03   4.48065777e-03   2.60800052e-02   5.08212316e-01]
 [  3.59562602e-04   3.32146350e-04   2.34314043e-04   2.07065303e-01]
 [  3.43819229e-01   1.10591357e-03   1.54488529e-03   8.55515256e-03]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  5.38019261e-05   1.13338507e-06   3.81154381e-01   3.26613185e-04]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  1.91546963e-03   2.11632577e-04   1.47333601e-03   5.10590198e-01]
 [  1.20505057e-03   3.35101372e-01   0.00000000e+00   5.21045410e-04]
 [  5.99134465e-01   6.10217564e-05   3.64610072e-04   4.84819878e-05]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  5.53332073e-03   1.22974934e-03   8.34650818e-01   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   9.46835403e-01   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]]
```



According to result, after learning, we can see that the agent usually succeeds even though it is in stochastic environment.

Q-Network

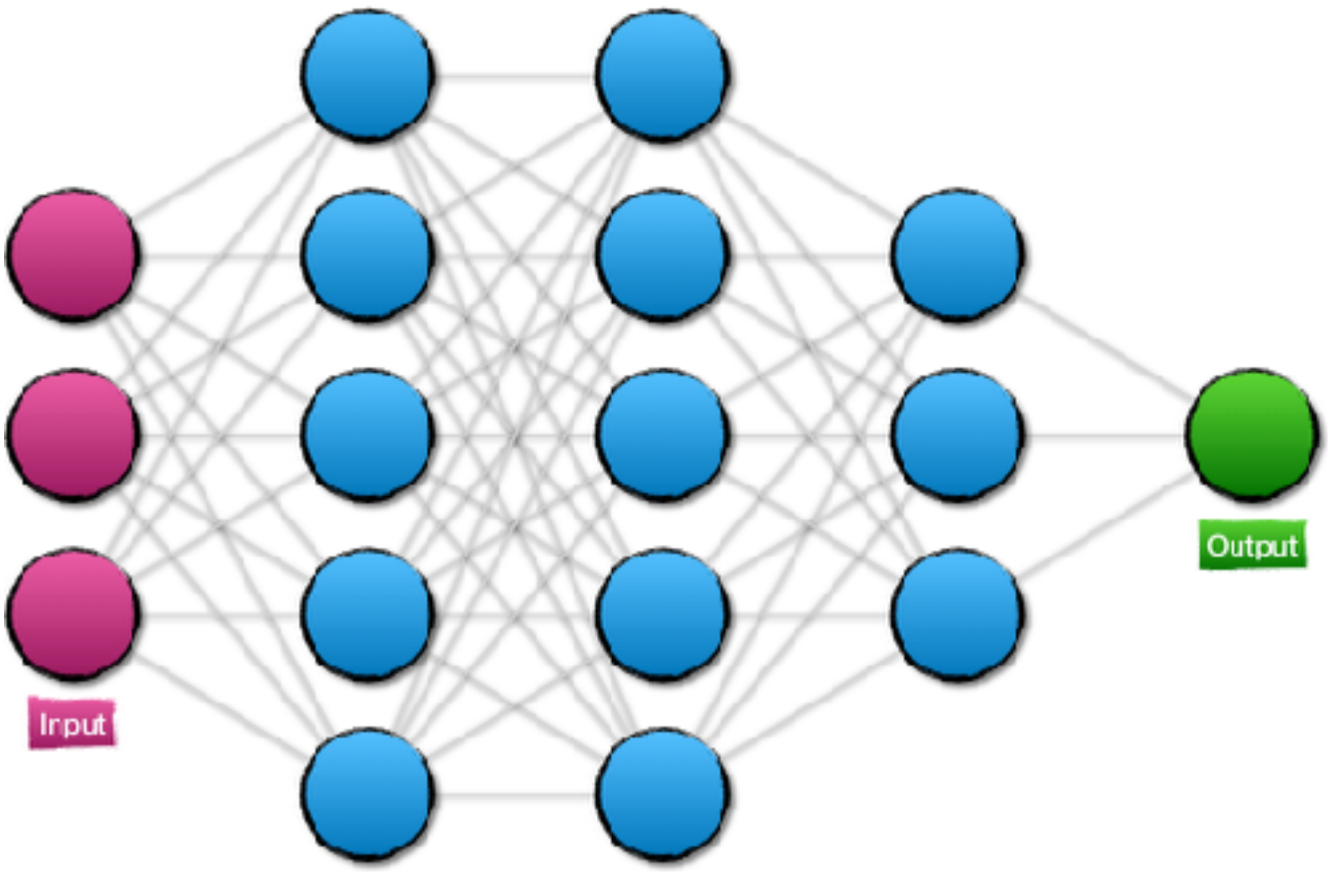
We fully understood Q-learning! But think about it. Can we apply this model to real life or more complex environment? Maybe not. In previous environment, we just used 16 states and 4 actions. Even if you simply represent an image of 16 * 16 pixels by RGB, we should prepare $256^{16 \cdot 16}$ tables. So, we need more universal model: **Neural Network**.



Compared with the Q-table, The Q-network receives the state and returns all possible action values. In previous Frozenlake game, Let's apply the Q-network by example state:

$$Q_{network}(s_{example}, a) = [0.5, 0.1, 0.0, 0.8]$$

[0.5,0.1,0.0,0.8] represents $(a_{left}, a_{right}, a_{up}, a_{down})$.



In the Neural Network, when we put state into the input, we will receive reward as output.

$$H(x) = Wx = \hat{Q}(s, a)$$

where H is hypothesis, Wx is output of the Neural Network. So, we can calculate cost by Mean squared error.

$$cost(W) = \frac{1}{n}\sum_{i=1}^{n}(Wx_i - y_i)^2$$

where y is target label. In Q-learning, we don't have any target label because it is reinforcement learning. So, in this case, we will set target label as $Q(s, a)$. Because optimize $Q(s, a)$ is our goal.

$$cost(W) = \frac{1}{n}\sum_{i=1}^{n}(Wx_i - y_i)^2 \qquad cost(W) = (Wx - y)^2 \qquad y = r + \gamma \max Q(s')$$

Let's approximate Q^* function using weight:

$$\hat{Q}(s, a \,|\, \theta) \rightarrow Q^*(s, a)$$

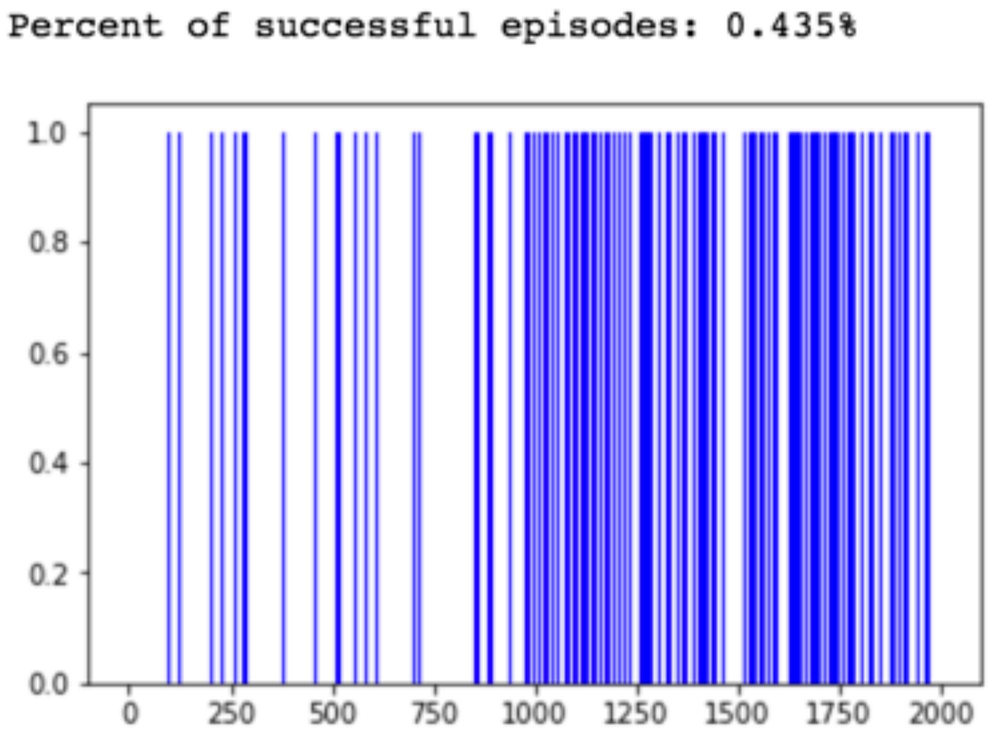
where θ is weight. So, we try to optimize Q-function by using s, a and θ. So, we can choose θ to minimize using this equation:

$$\min \sum_{t=0}^{T} [\hat{Q}(s_t, a_t \,|\, \theta) - (r_t + \gamma \max \hat{Q}(s_{t+1}, a' \,|\, \theta))]^2$$

It looks complicated. But it is just same as optimizing weight in Linear regression by Mean squared error. Putting all this together, we can list out the algorithm we'll use to train the network.

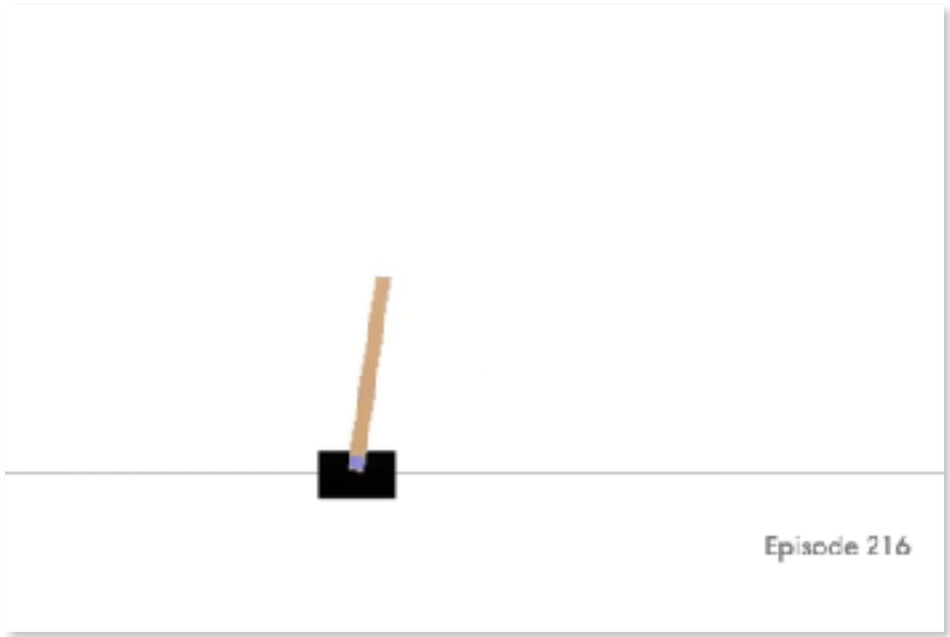
```
Initialize action-value network $Q$ with random weights
For episode = 1, $M$ do
 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
      For $t = 1, T$ do
           With probability $\epsilon$ select a random action $a_t$ ,
           otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
           Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
           Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
           Set $y_j = \begin{cases} r_j \; (for\ terminal\ \phi_{j+1}) \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) \; (for\ non-terminal\ \phi_{j+1}) \end{cases}$
           Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
      endfor
endfor
```

And here is a result of FrozenLake by Q-network:



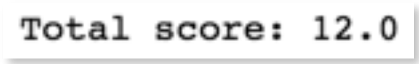
```
Percent of successful episodes: 0.435%
```

But actually this model isn't efficient. In other words, it has some problem. How can we fix it? The solution is **DQN**. But, before we update our model with DQN, let's look at other game using same model. The other game is **CartPole**. In this game, a freely swinging pole is attached to a cart. The cart can move to the left and right, and the goal is to keep the pole upright as long as possible.



In the Cart-Pole game, there are two possible actions, moving the cart left or right. So there are two actions we can take, encoded as 0 and 1. The game resets after the pole has fallen past a certain angle. For each frame while the simulation is running, it returns a reward of 1.0. The longer the game runs, the more reward we get. Then, our network's goal is to maximize the reward by keeping the pole vertical. It will do this by moving the cart to the left and the right.

It's very similar with previous 'FrozenLake' environment. But in this game, we will receive left or right as Q-reward. In 'FrozenLake' environment, we received up, down, left and right as Q-reward. If the pole doesn't have fall over 200 steps, we will consider that the agent is learned.



```
Total score: 12.0
```

Although we used Q-network, our network doesn't work well. Because our model has some problems. But, how can we fix our model? The answer is that: use **Deep Q-Network**. Finally, it is time to implement DQN.

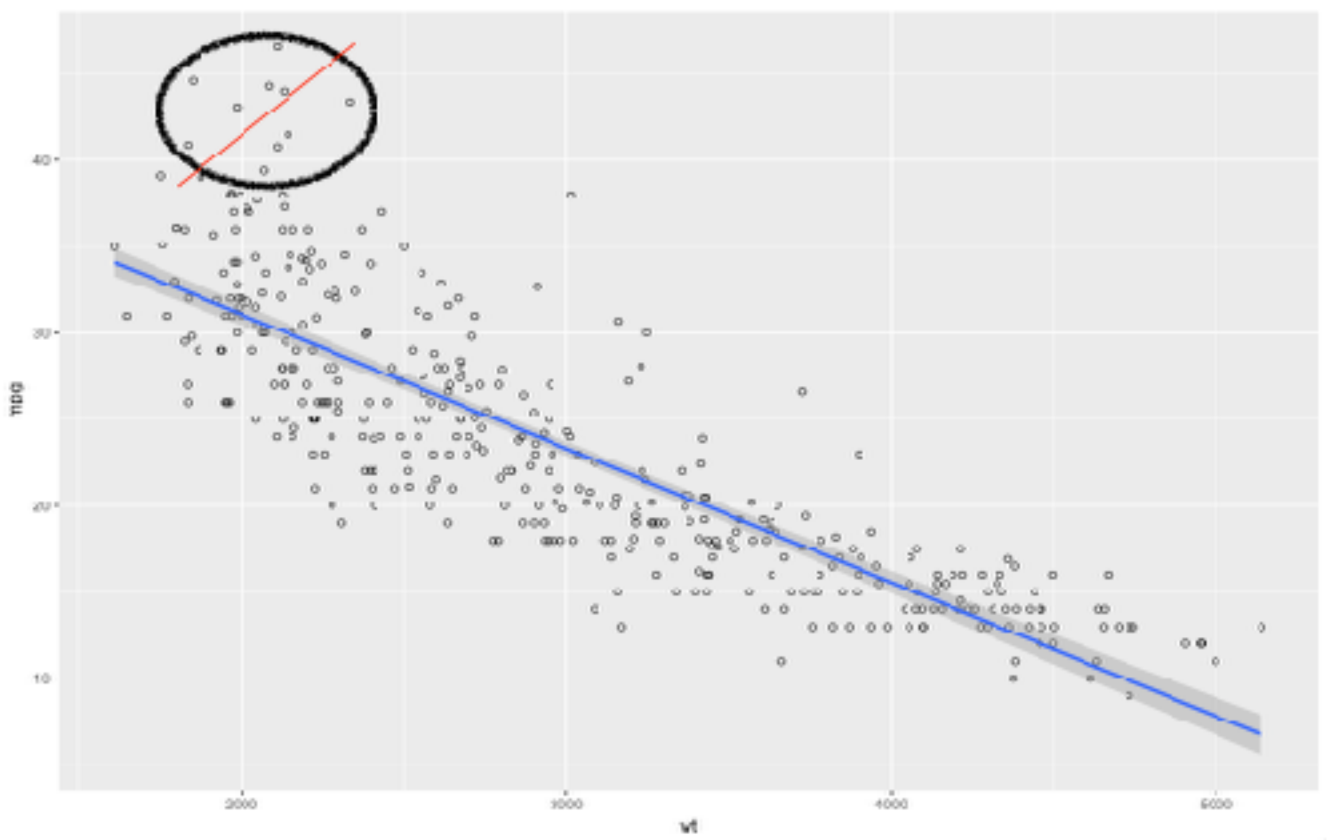
Deep Q-network

Our model has three main problems:

1. Too shallow
2. Correlations between samples
3. Non-stationary targets

First, we use only one layer. It isn't deep enough to make a good solution.

And second, it has problem that **Correlations between samples**. According to algorithm, each data that we received is very similar. Because when we received reward, the pole state isn't significantly different from previous state. So, there are correlations between samples. But, why is it a problem?



This is a graph of weight and mile per gallon about a car. And we can draw a blue line as a result of linear regression. But if we had only a few correlational data like in the black circle, we would draw different line as a red. It makes totally different result. Our previous model has same problem. The agent could be trained incorrectly because it was just trained by correlational samples.

Third problem is **Non-stationary targets**. Let's check our cost formula again.

$$\min \sum_{t=0}^{T} [\hat{Q}(s_t, a_t | \theta) - (r_t + \gamma \max \hat{Q}(s_{t+1}, a' | \theta))]^2$$

In this equation, $\hat{Q}(s_t, a_t | \theta)$ is our prediction. And $(r_t + \gamma \max \hat{Q}(s_{t+1}, a' | \theta))^2$ is our target.

$$\hat{Y} = \hat{Q}(s_t, a_t | \theta) \qquad Y = (r_t + \gamma \max \hat{Q}(s_{t+1}, a' | \theta)$$

So, our goal is making $\hat{Y}$ and Y similar. But, we have same θ in each equation. When we update our $\hat{Y}$, our target Y is updated too. It is like that immediately moving a target after shooting an arrow.

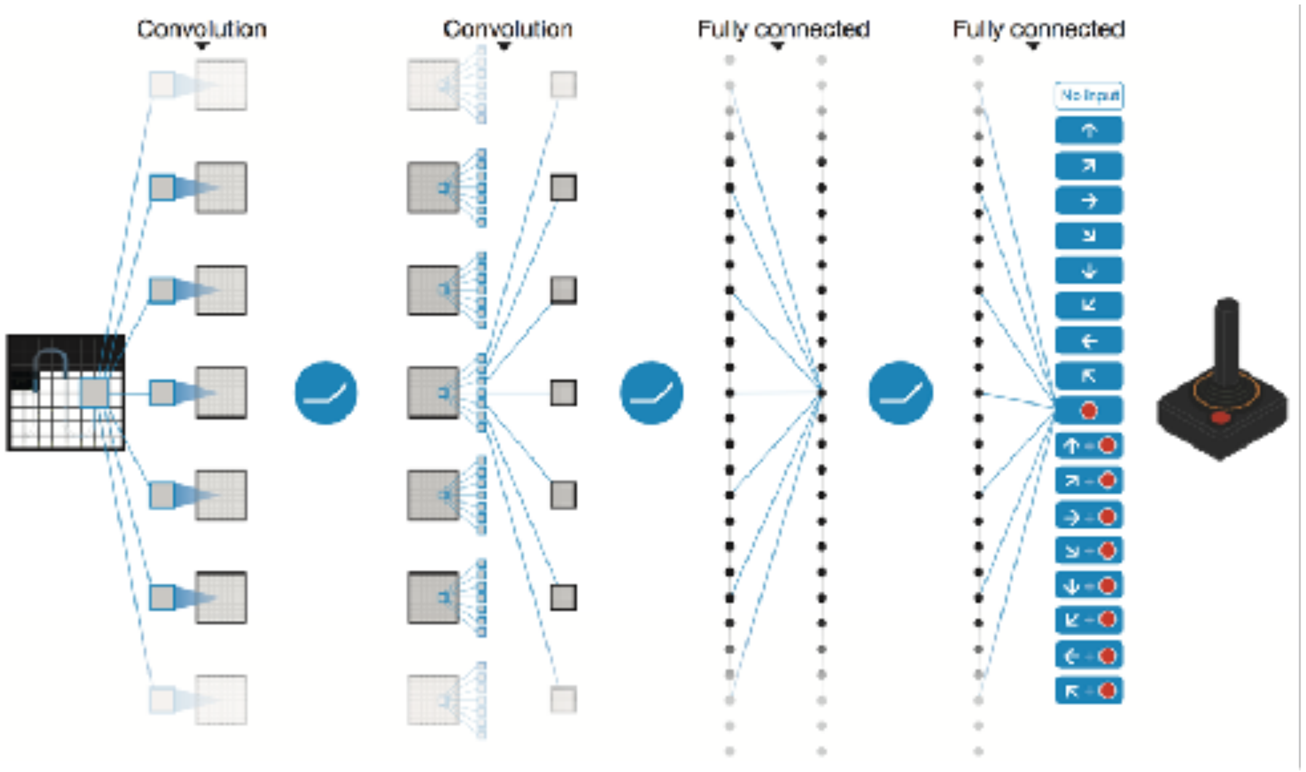
This is an ironical situation. So, we call this situation **Non-stationary targets**.

So, here are solutions:

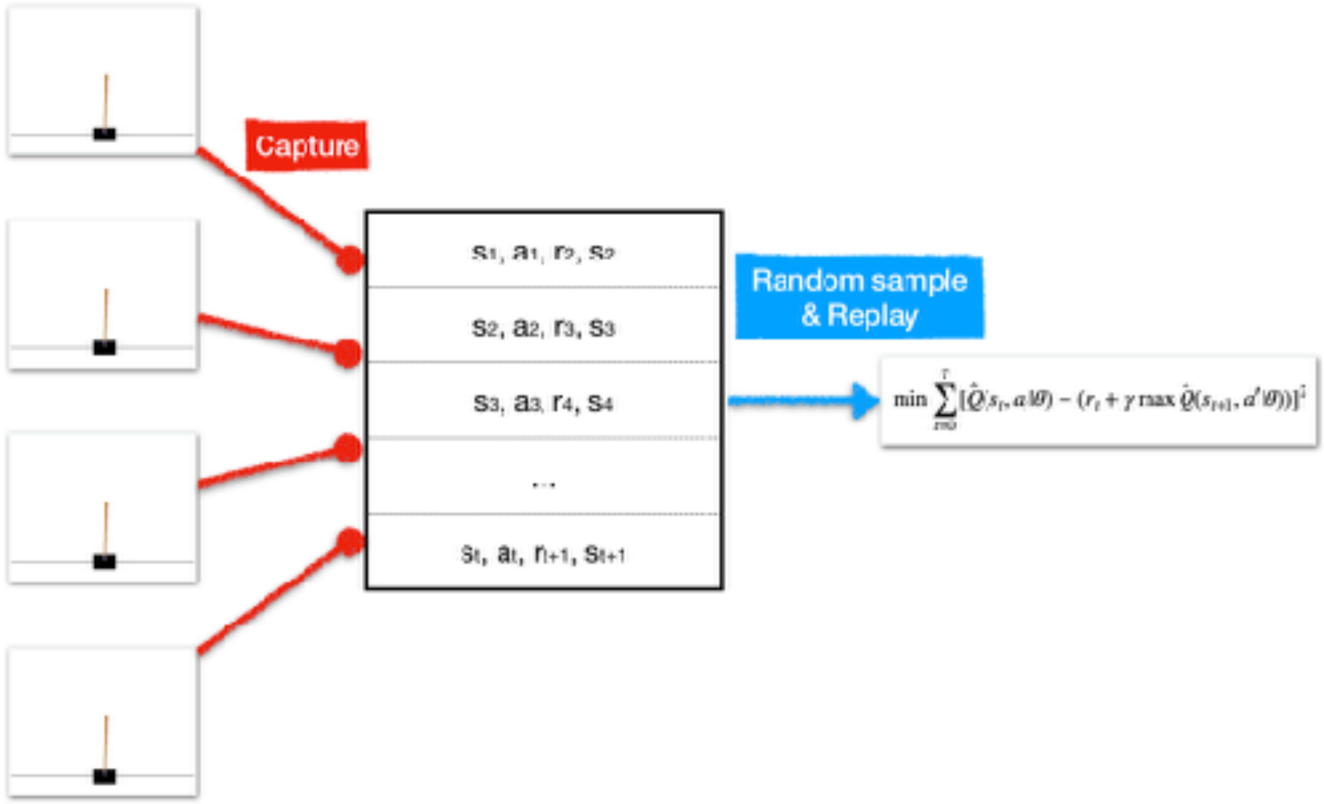
1. Too shallow
   → Go deep
2. Correlations between samples
   → Capture and replay
3. Non-stationary targets
   → Separate networks: Create a target network

And we call this solution network **Deep Q-network**. And let's dive into DQN's solutions.

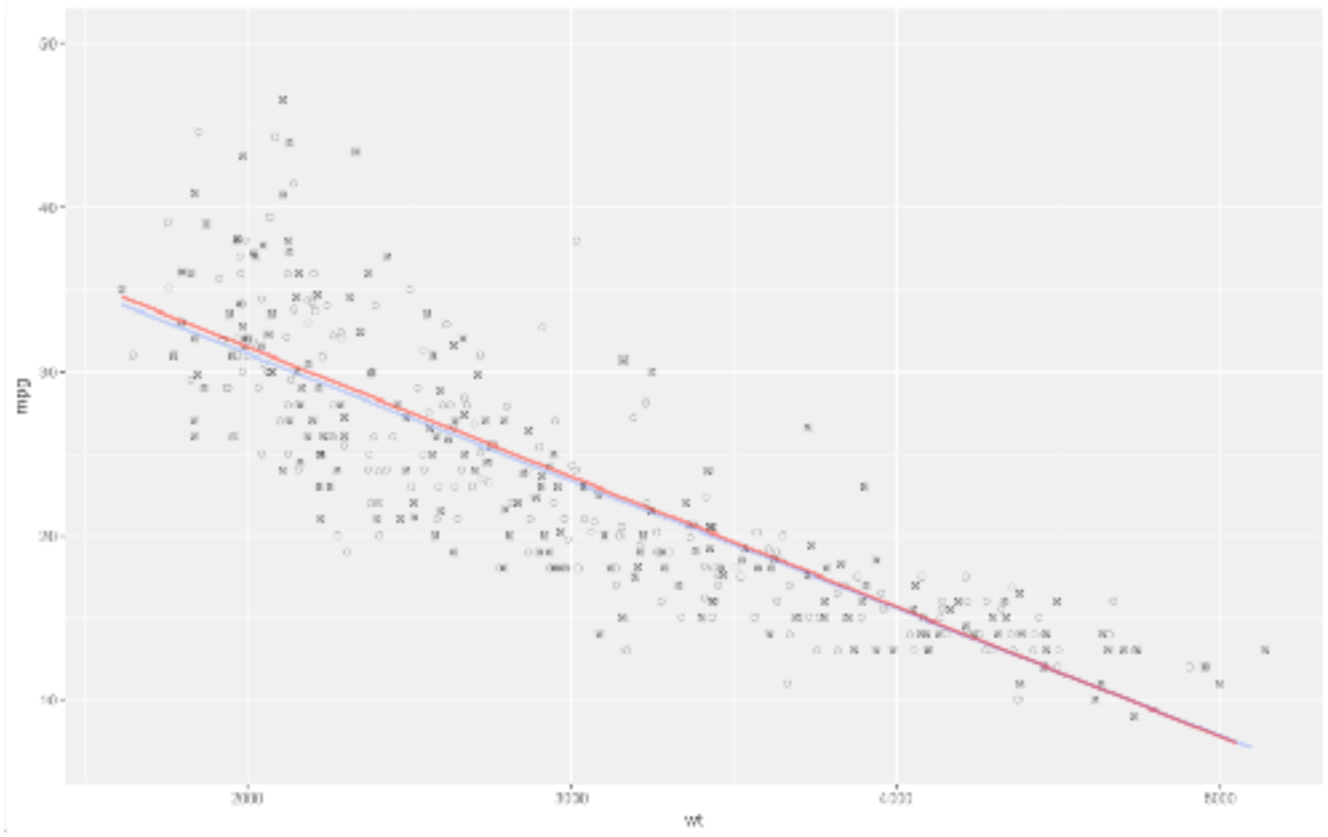
First, **Go deep**. Actually our previous network isn't deeper enough. So, we should make our network deeper as making more layers.



Second, **Capture and replay.** This is solution of Correlations between samples problem. This concept is very simple. To prevent Correlations between samples, we save data in buffer after the agent's action. Then choose random sample in the buffer and update by algorithm.


Capture
s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
...
$s_t, a_t, r_{t+1}, s_{t+1}$
Random sample & Replay
$\min \sum_{t=0}^{T} [\hat{Q}(s_t, a|\theta) - (r_t + \gamma \max \hat{Q}(s_{t+1}, a'|\theta))]^2$

Let's see the graph of weight and mile per gallon about a car again.



In this graph, circles are same as previous scatter graph. And blue line is linear regression line by circles. However I choose half of previous samples randomly, and mark as 'x'. And red line is linear regression line by 'x's. As you can see, each result are almost same. Like this, we can train our Q-network by choosing random samples in buffer.

So, we can summarize that:

```
Initialize action-value network $Q$ with random weights $\theta$
For episode = 1, $M$ do
 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
       For $t = 1, T$ do
             With probability $\epsilon$ select a random action $a_t$ ,
             otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
             Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
             Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
```

```
            Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
            Sample random minibatch of transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
            Set $y_j = \begin{cases} r_j \ (for\ terminal\ \phi_{j+1}) \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)\ (for\ non-terminal\ \phi_{j+1}) \end{cases}$
            Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
        endfor
endfor
```

Third, **Separate networks: Create a target network**. This is solution of Non-stationary targets problem. It is really simple. Just create two network. Let's update our equation.

$$\min \sum_{t=0}^{T} [\hat{Q}(s_t, a_t | \theta) - (r_t + \gamma \max \hat{Q}(s_{t+1}, a' | \bar{\theta}))]^2$$

Only one difference is $\bar{\theta}$ in target label. We will separate network as training and target. First of all, we will update only training network. And after several times, update target network too. In our algorithm, this solution corresponds to

Set $\hat{Q}_j = r_j$ if the episode ends at $j+1$, otherwise set $\hat{Q}_j = r_j + \gamma \max_{a'} Q(s_j', a')$

We will update $\hat{Q}_j = r_j + \gamma \max_{a'} Q(s_j', a')$ by another network.

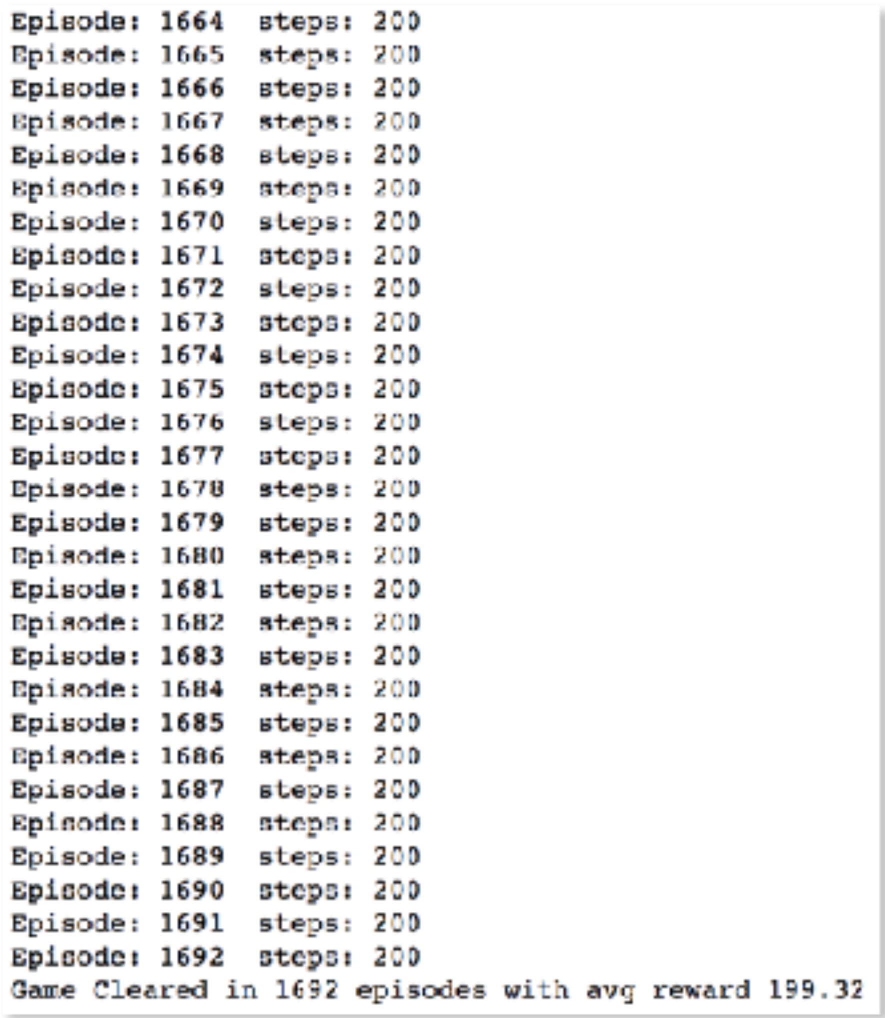
Finally, we can summarize our algorithm as DQN:

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value network $Q$ with random weights $\theta$
Initialize target action-value network $\hat{Q}$ with random weights $\bar{\theta} = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
        For $t = 1, T$ do
            With probability $\epsilon$ select a random action $a_t$ ,
            otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
            Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
            Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
```

```
                Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
                Sample random minibatch of transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
                Set $y_j = \begin{cases} r_j \ (if\ episode\ teminates\ at\ step\ j+1) \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \bar{\theta}) \ (otherwise) \end{cases}$
                Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
                Every $C$ steps reset $\hat{Q} = Q$
        endfor
endfor
```

3. Results

And here is our final model's result:



```
Episode: 1664  steps: 200
Episode: 1665  steps: 200
Episode: 1666  steps: 200
Episode: 1667  steps: 200
Episode: 1668  steps: 200
Episode: 1669  steps: 200
Episode: 1670  steps: 200
Episode: 1671  steps: 200
Episode: 1672  steps: 200
Episode: 1673  steps: 200
Episode: 1674  steps: 200
Episode: 1675  steps: 200
Episode: 1676  steps: 200
Episode: 1677  steps: 200
Episode: 1678  steps: 200
Episode: 1679  steps: 200
Episode: 1680  steps: 200
Episode: 1681  steps: 200
Episode: 1682  steps: 200
Episode: 1683  steps: 200
Episode: 1684  steps: 200
Episode: 1685  steps: 200
Episode: 1686  steps: 200
Episode: 1687  steps: 200
Episode: 1688  steps: 200
Episode: 1689  steps: 200
Episode: 1690  steps: 200
Episode: 1691  steps: 200
Episode: 1692  steps: 200
Game Cleared in 1692 episodes with avg reward 199.32
```

As I mentioned, according to CartPole document, it defines "solving" as getting average reward of 195.0 over 100 consecutive trials. Our average reward is 199.32. And I also attached result video and before training video.

3. Conclusion

As Gym updated, they limited max steps by 200 in CarPole game. So, our model should accord to new policy. Although it isn't enough, we can find our model works well.

And we just used Fully network model. We can update our model as RNN or CNN, and also we would optimize parameters like learning rate, sample size, decay factor, bias, and activate functions. There are a lot of game agent model. So also we will be able to challenge more complex games.

4. Reference

HTTPS://WWW.YOUTUBE.COM/WATCH?V=G-DKXOLSF98

HTTPS://HUNKIM.GITHUB.IO/ML/

HTTPS://WWW.UDACITY.COM/COURSE/DEEP-LEARNING-NANODEGREE-FOUNDATION--ND101

HTTP://WWW.NATURE.COM/NATURE/JOURNAL/V518/N7540/FULL/NATURE14236.HTML?FOXTROTCALLBACK=TRUE

HTTPS://DEEPMIND.COM/BLOG/DEEP-REINFORCEMENT-LEARNING/

HTTP://AIKOREA.ORG/AWESOME-RL/

HTTPS://MEDIUM.COM/EMERGENT-FUTURE

HTTP://KVFRANS.COM/SIMPLE-ALGORITMS-FOR-SOLVING-CARTPOLE/

HTTP://KARPATHY.GITHUB.IO/2016/05/31/RL/

HTTPS://WWW.GITBOOK.COM/BOOK/DNDDNJS/RL/DETAILS

HTTPS://SIMONS.BERKELEY.EDU/TALKS/TUTORIAL-REINFORCEMENT-LEARNING

HTTP://RLL.BERKELEY.EDU/DEEPRLCOURSE/

HTTP://SELFDRIVINGCARS.MIT.EDU/

HTTPS://WWW.YOUTUBE.COM/WATCH?V=PTAIH9KSNJO&T=2457S

HTTPS://WWW.YOUTUBE.COM/WATCH?V=AURX-RP_SS4&LIST=PLJKEIQLKCTZYN3CYBLJ8R58SBNOROBQCP

HTTP://WWW0.CS.UCL.AC.UK/STAFF/D.SILVER/WEB/TEACHING.HTML

HTTPS://WWW.YOUTUBE.COM/WATCH?V=RTXI449ZJSC

HTTPS://GIST.GITHUB.COM/STOBER/1943451

HTTPS://MEDIUM.COM/EMERGENT-FUTURE/SIMPLE-REINFORCEMENT-LEARNING-WITH-TENSORFLOW-PART-0-Q-LEARNING-WITH-TABLES-AND-NEURAL-NETWORKS-D195264329D0

HTTP://HOME.DEIB.POLIMI.IT/RESTELLI/MYWEBSITE/PDF/RL5.PDF

HTTP://INTROTODEEPLEARNING.COM/6.S091DEEPREINFORCEMENTLEARNING.PDF

HTTPS://WWW.CS.TORONTO.EDU/~VMNIH/DOCS/DQN.PDF

HTTPS://STACKOVERFLOW.COM/QUESTIONS/10722064/TRAINING-A-NEURAL-NETWORK-WITH-REINFORCEMENT-LEARNING

MACHINE LEARNING, TOM MITCHELL