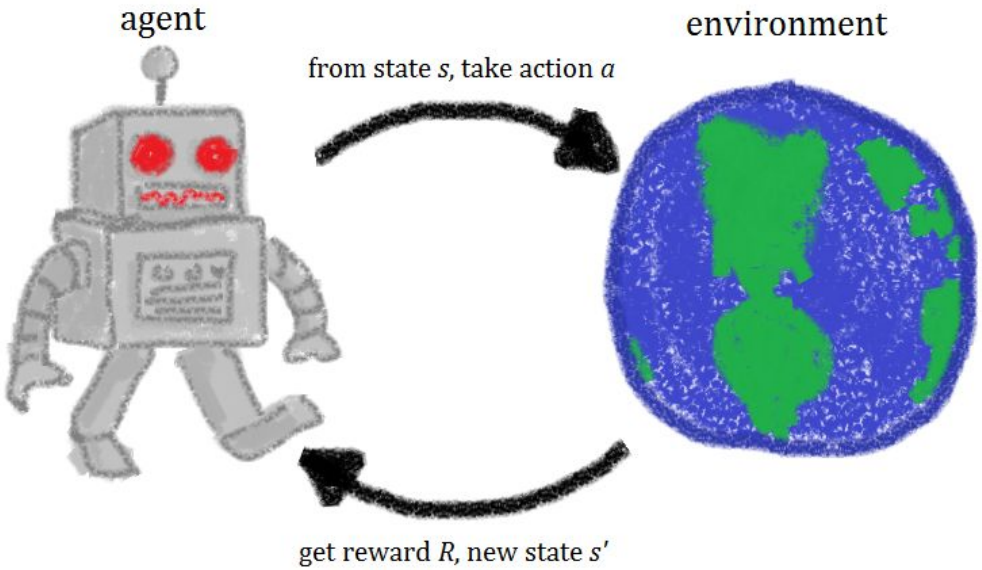
Reinforcement Learning

Dr. Dongchul Kim

Reinforcement Learning



Supervised vs Unsupervised

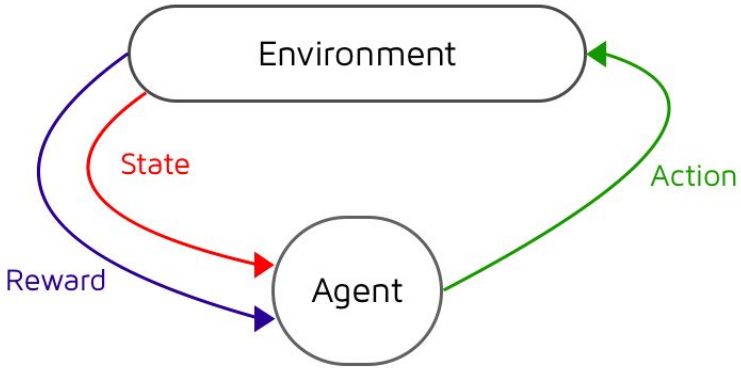
Reinforcement learning lies somewhere in between supervised and unsupervised learning.

Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all

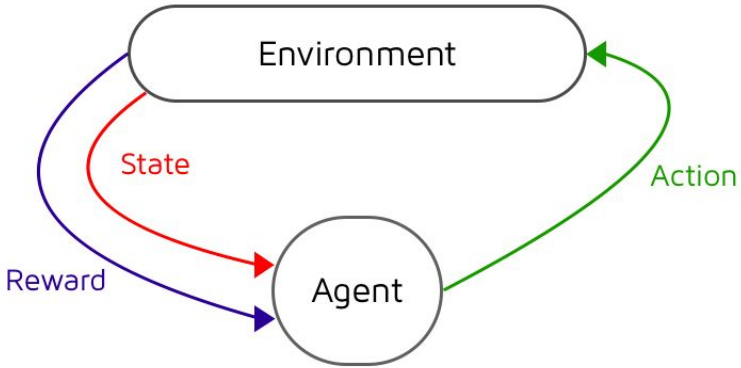
In reinforcement learning one has sparse labels – the rewards time-delayed

Based only on those rewards the agent has to learn to behave in the environment.

https://www.youtube.com/watch?v=V1eYniJ0Rnk



- Suppose you are an **agent**, situated in an **environment**.
- The environment is in a certain **state** (e.g. location of the paddle, location and direction of the ball, existence of every brick and so on).
- The agent can perform certain **actions** in the environment (e.g. move the paddle to the left or to the right).
- These actions sometimes result in a **reward** (e.g. break bricks and increase in score).



- Actions transform the environment and lead to a new state, where the agent can perform another action, and so on.
- The rules for how you choose those actions are called **policy**.
- The environment in general is **stochastic**, which means the next state may be somewhat random.

State, Action, and Reward

- One episode of this process (e.g. one game the agent played) forms a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

- To perform well in long-term, we need to take into account not only the immediate rewards, but also the **future rewards** we are going to get.

total future reward

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

Policy and Out Goal

A policy π: s $\rightarrow$a is any function that returns an action for a problem.

It's like mapping from states to actions.

So, basically, a policy function says what action to perform in each state.

Our ultimate goal lies in finding the optimal policy which specifies the correct action to perform in each state, which maximizes the reward.

discounted future reward

But because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions.

The more into the future we go, the more it may diverge.

For that reason it is common to use **discounted future reward** instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

discounted future reward

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

Q-learning

- In Q-learning we define a function Q(s,a) representing the discounted future reward when we perform action **a** in state **s**, and continue optimally from that point on.

$$Q(s_t, a_t) = max_\pi R_{t+1}$$

- The way to think about `Q(s,a)` is that it is "the best possible score at the end of game after performing action **a** in state **s**". It is called Q-function, because it represents the "quality" of certain action in given state.

Q function

- how do we get that Q-function then?
- Let’s focus on just one transition <s,a,r,s′>.
- Just like with discounted future rewards in previous section,
- we can express Q-value of state s and action a in terms of Q-value of next state s′.

Bellman Equation

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

- This is called the **Bellman equation**. If you think about it, it is quite logical – maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state.
- The main idea in Q-learning is that **we can iteratively approximate the Q-function using the Bellman equation**. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns.

Q learning

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γmaxa' Q[s',a'] - Q[s,a])
    s = s'
until terminated
```

Example: 6 states and 4 actions environment

There are 6 states and 4 actions.

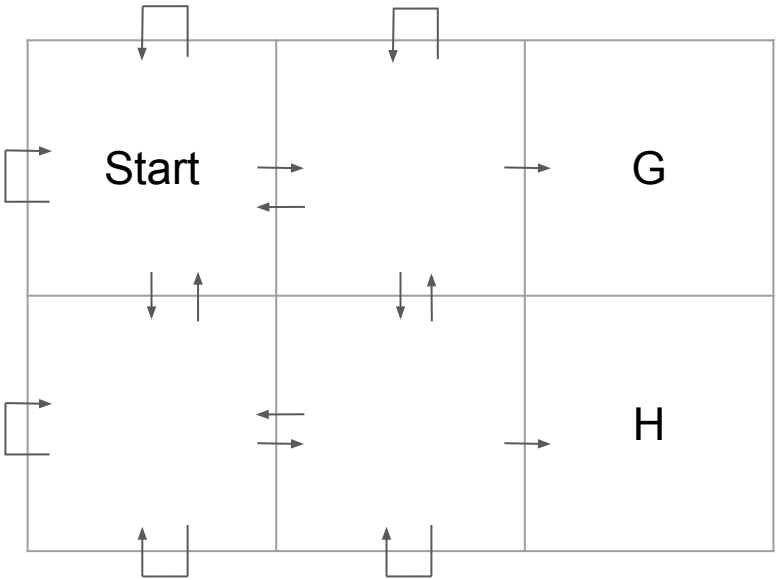
The agent starts from the start state.

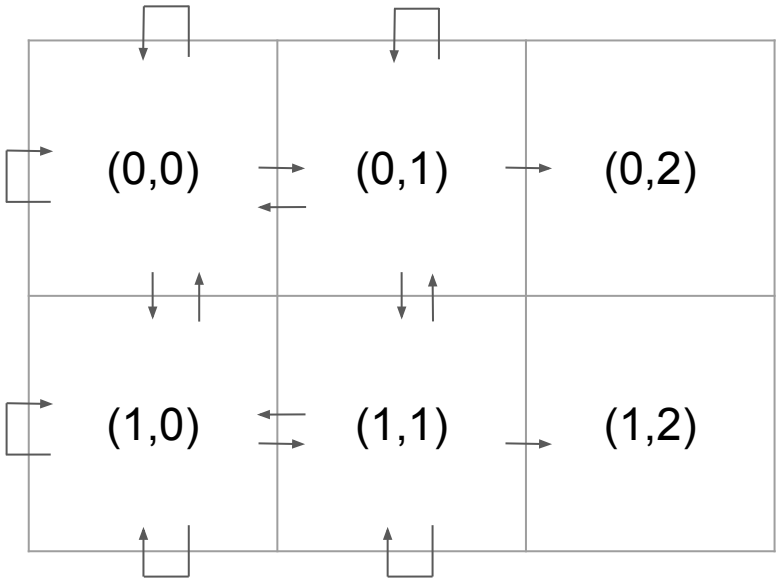
If the agent gets in the state G, there's a positive reward (+100).

If the agent moves in the state H, there will be a negative reward (-100).

The episode is terminated once the agent moves to G or H.

Learning rate (alpha) is 1.0.



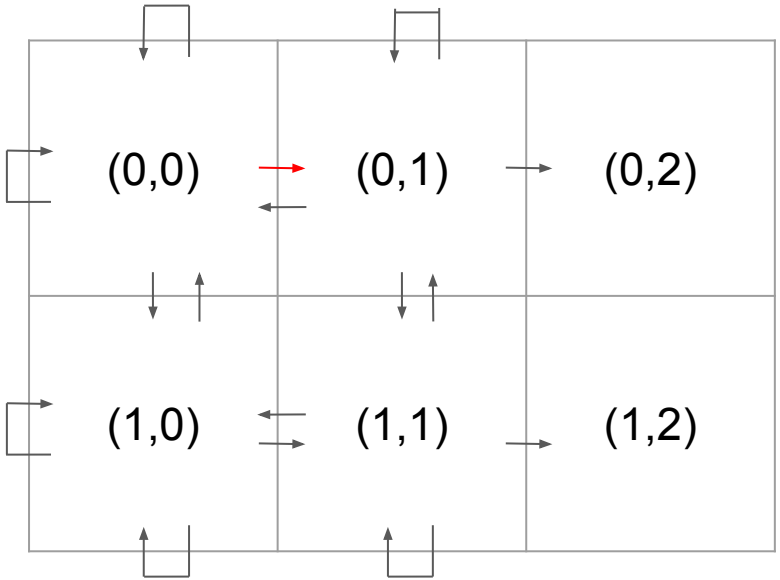


	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	0	0
(1,2)	0	0	0	0

Initially, an agent assumes a policy that 90% of the time takes a random action and 10% of the time uses a Q-table.

In the first episode, suppose the first action is chosen randomly and it's a move to the right.

Then, let's see how the Q-table is updated.



	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	0	0
(1,2)	0	0	0	0

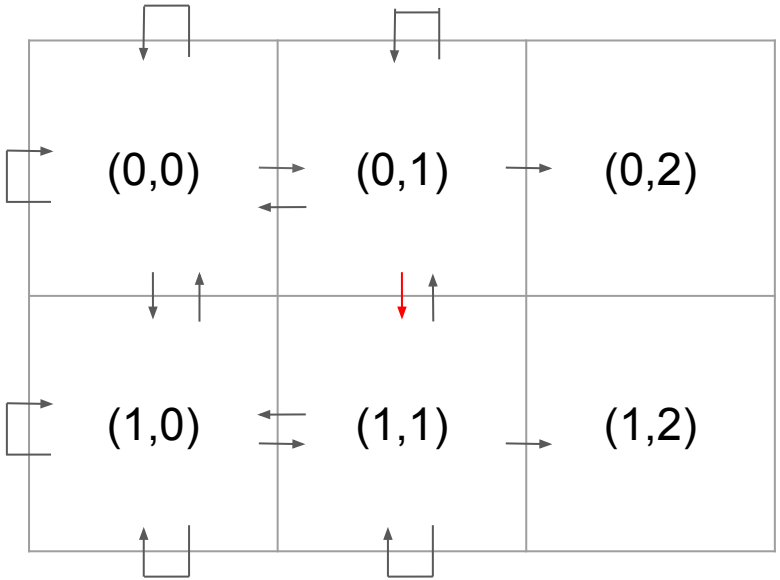
$$Q((0,0), \text{Right}) = \text{reward} + \text{gamma} \cdot \max_{a'} Q((0,1), a')$$
$$= 0 + 0.9 \cdot \max(0,0,0,0) = 0$$

$$\max_{a'} Q((0,1), a')$$

As the agent still follows a policy that 90% of the time takes a random action and 10% of the time uses a Q-table.

Assume the second action is chosen randomly and it's a move down.

Then, let's see how the Q-table is updated.



	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	0	0
(1,2)	0	0	0	0

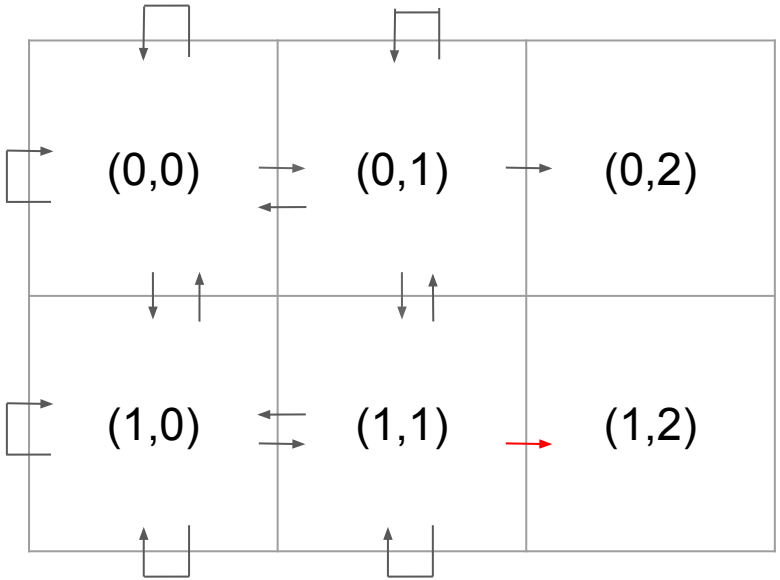
```
Q((0,1),Down) = reward + gamma · max_a' Q((1,1), a')
= 0 + 0.9·max(0,0,0,0) = 0
```

```
max_a' Q((1,1), a')
```

As the agent still follows a policy that 90% of the time takes a random action and 10% of the time uses a Q-table.

Assume the third action is chosen randomly and it's a move to the right.

Then, let's see how the Q-table is updated.



	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

```
Q((1,1), Right) = reward + gamma · max_a' Q((1,2), a')
= -100 + 0.9·max(0,0,0,0) = -100
```

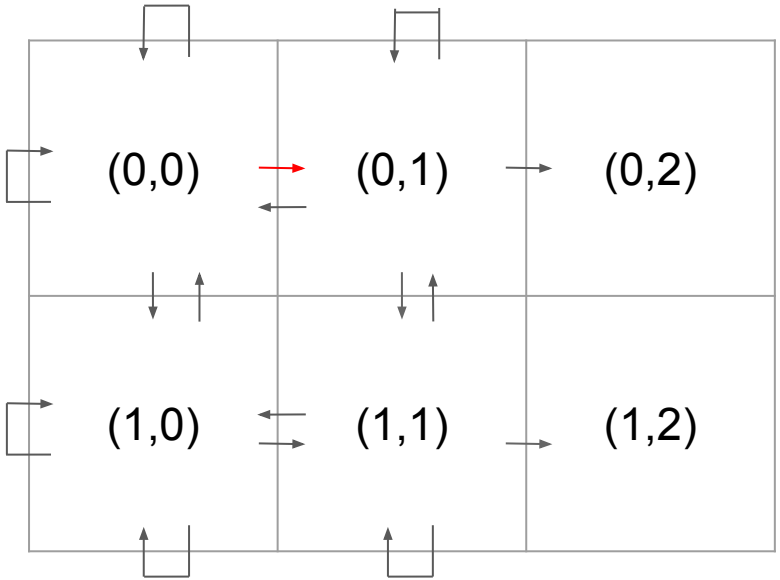
```
max_a' Q((1,2), a')
```

The first episode is terminated.

In the second episode, the agent still follows a policy that 90% of the time takes a random action and 10% of the time uses a Q-table.

Assume the first action is chosen randomly and it's a move to the right.

We know it does not change the Q-table.



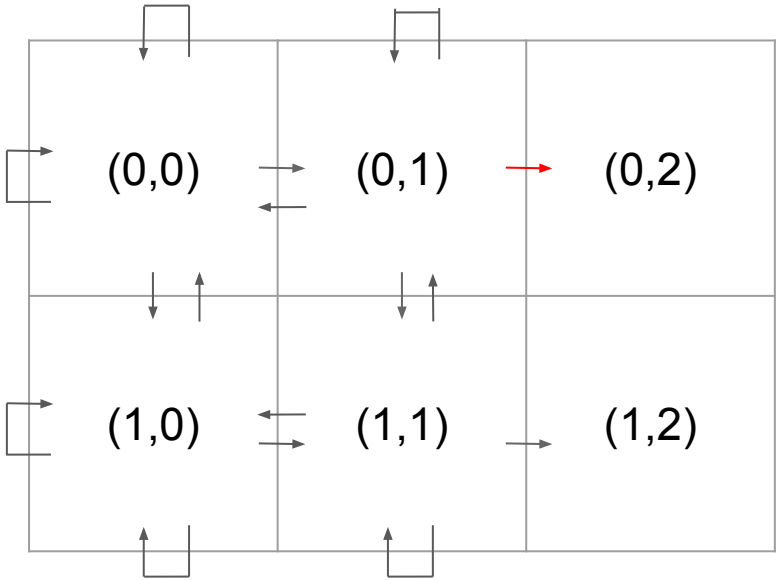
	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

$Q((0,0), \text{Right}) = \text{reward} + \text{gamma} \cdot \max_{a'} Q((0,1), a')$

$= 0 + 0.9 \cdot \max(0,0,0,0) = 0$

$\max_{a'} Q((0,1), a')$

Assume the second action is chosen randomly and it’s a move to the right.



	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	100	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

$Q((0,1), Right) = reward + gamma \cdot max_{a'} Q((0,2), a')$

$= 100 + 0.9 \cdot max(0,0,0,0) = 100$

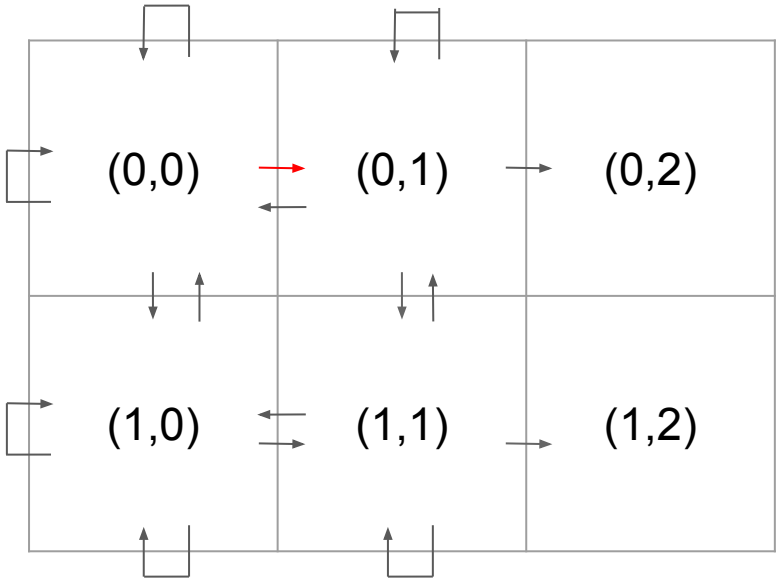
$max_{a'} Q((0,2), a')$

The second episode is terminated.

In the third episode, the agent still follows a policy that 90% of the time takes a random action and 10% of the time uses a Q-table.

Assume the first action is chosen randomly and it's a move to the right again.

We know it does not change the Q-table.



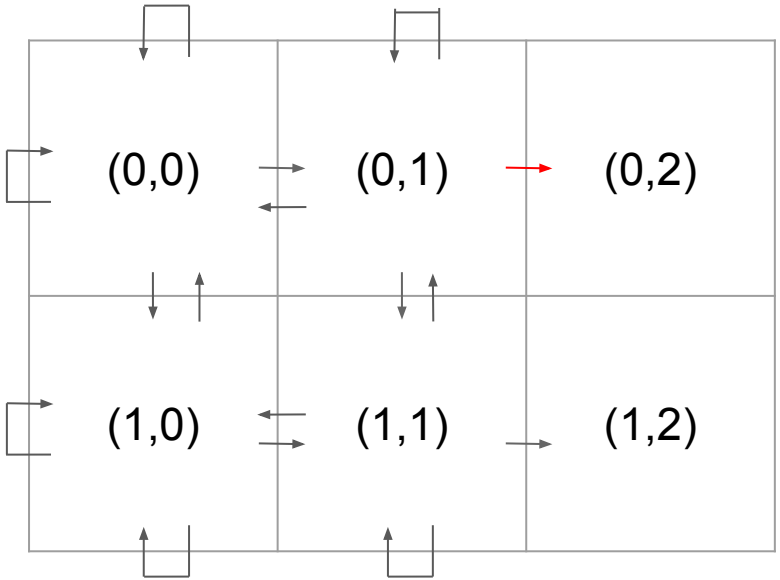
	←	↓	→	↑
(0,0)	0	0	90	0
(0,1)	0	0	100	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

$Q((0,0), \rightarrow) = \text{reward} + \text{gamma} \cdot \max_{a'} Q((0,1), a')$

$= 0 + 0.9 \cdot \max(0,0,100,0) = 90$

$\max_{a'} Q((0,1), a')$

Assume the second action is chosen randomly and it’s a move to the right.



	←	↓	→	↑
(0,0)	0	0	90	0
(0,1)	0	0	100	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

$Q((0,1), Right) = reward + gamma \cdot max_{a'} Q((0,2), a')$

$= 100 + 0.9 \cdot max(0,0,0,0) = 100$

$max_{a'} Q((0,2), a')$

After many episodes, the Q-table is converged and agent follows a policy that takes less probability of the random action (e.g., 90%->10%) and take an action using the Q-table.

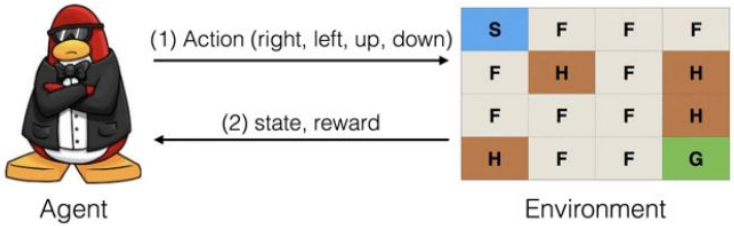
If the agent follows the policy using only Q-table we have now, we move from Start (0,0) to (0,1) and then move to (0,2) which is the goal.

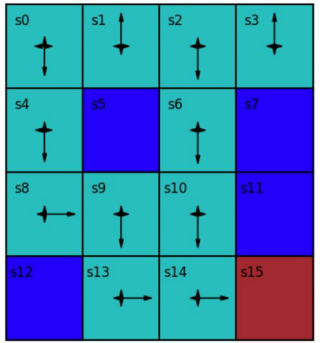
	←	↓	→	↑
(0,0)	0	0	90	0
(0,1)	0	0	100	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

Reference

https://github.com/PacktPublishing/Advanced-Deep-Learning-with-Keras/tree/master/chapter9-drl

Frozen Lake (16 states, 4 actions)





Q table

	S	E	N	W
s0	-0.08	-0.13	-0.11	-0.10
s1	-0.83	-0.17	-0.13	-0.20
s2	-0.01	-0.12	-0.06	-0.11
s3	-0.82	-0.21	-0.13	-0.18
s4	-0.04	-0.83	-0.19	-0.06
s5	0.00	0.00	0.00	0.00
s6	0.07	-0.80	-0.28	-0.78
s7	0.00	0.00	0.00	0.00
s8	-0.76	0.14	-0.02	-0.00
s9	0.39	0.21	-0.75	-0.02
s10	0.45	-0.71	-0.06	0.28
s11	0.00	0.00	0.00	0.00
s12	0.00	0.00	0.00	0.00
s13	0.34	0.58	0.17	-0.70
s14	0.66	0.87	0.40	0.45
s15	0.00	0.00	0.00	0.00

Policy (left) according to Q-table (right)

How about breakout?

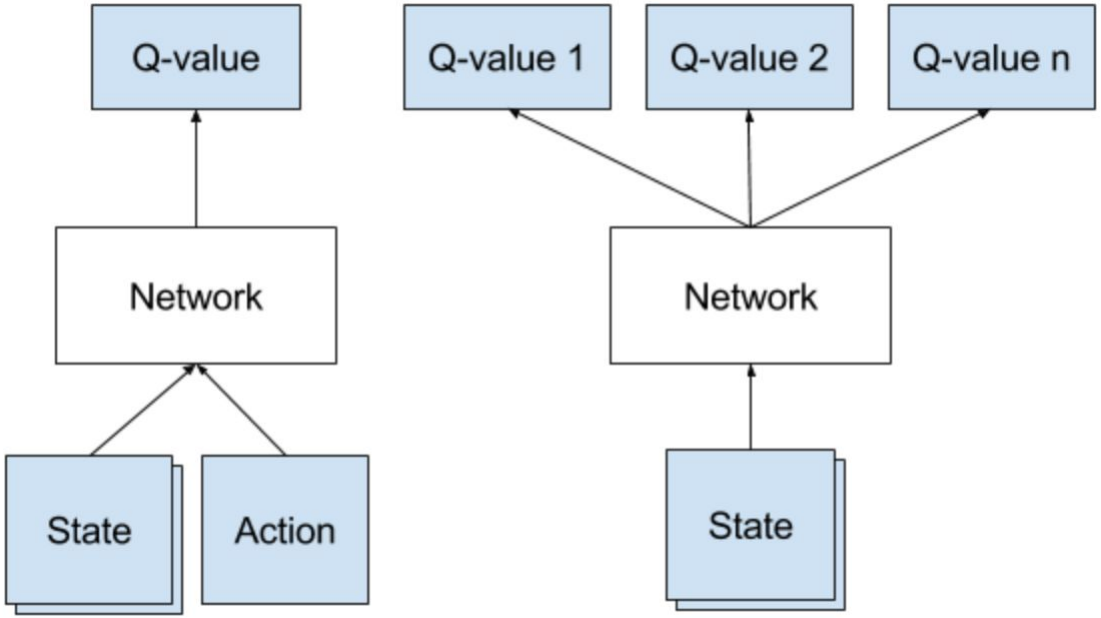
Q-learning for breakout?

- The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the existence of each individual brick.
- This intuitive representation is however game specific.
- Could we come up with something more universal, that would be suitable for all the games?
- Screen pixels???

Q-learning for breakout?

- There are too many distinct states.
- Imagine a screen with 100 x 100 resolution
- Solution by Deepmind
  - Deep Q network (DQN)

DQN


Q-value
Network
State
Action
Q-value 1
Q-value 2
Q-value n
Network
State

DQN

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8×8	4	32	ReLU	20x20x32
conv2	20x20x32	4×4	2	64	ReLU	9x9x64
conv3	9x9x64	3×3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

DQN

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
    select an action a
        with probability ε select a random action
        otherwise select a = argmaxa’Q(s,a’)
    carry out action a
    observe reward r and new state s’
    store experience <s, a, r, s’> in replay memory D

    sample random transitions <ss, aa, rr, ss’> from replay memory D
    calculate target for each minibatch transition
        if ss’ is terminal state then tt = rr
        otherwise tt = rr + γmaxa’Q(ss’, aa’)
    train the Q network using (tt - Q(ss, aa))^2 as loss

    s = s'
until terminated
```

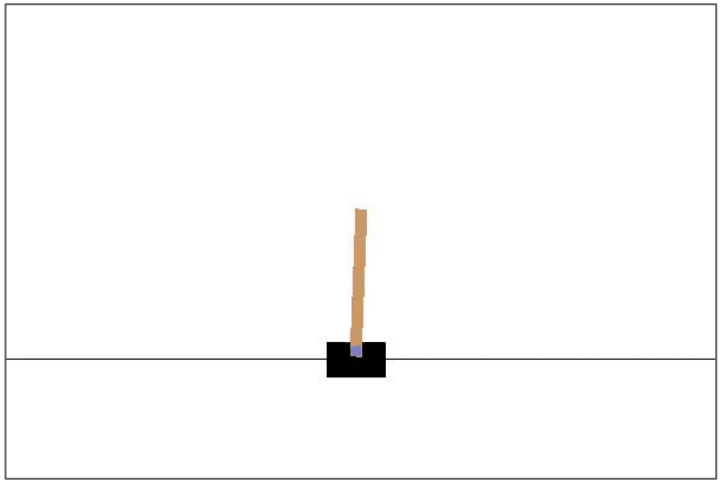
Loss function

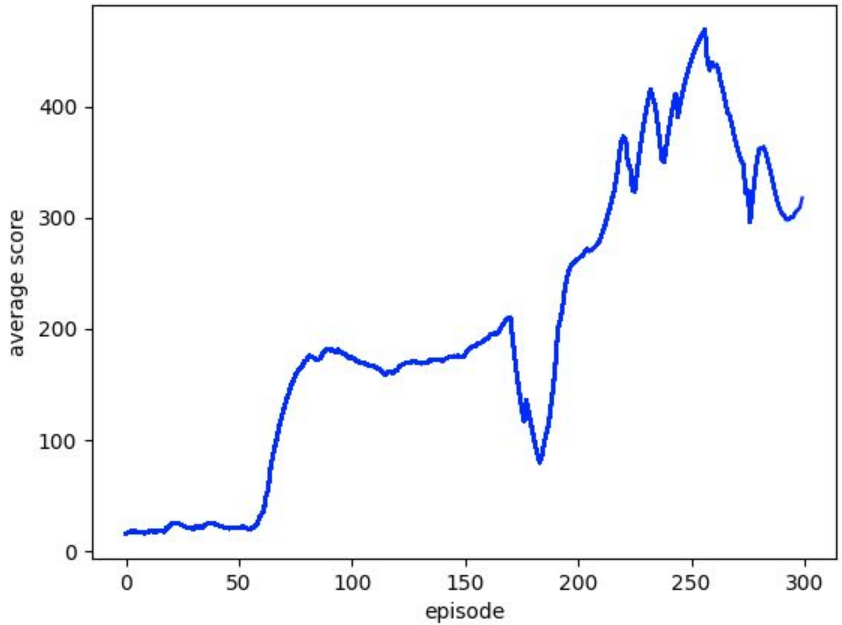
$$L = \frac{1}{2}[\underbrace{r + \gamma max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

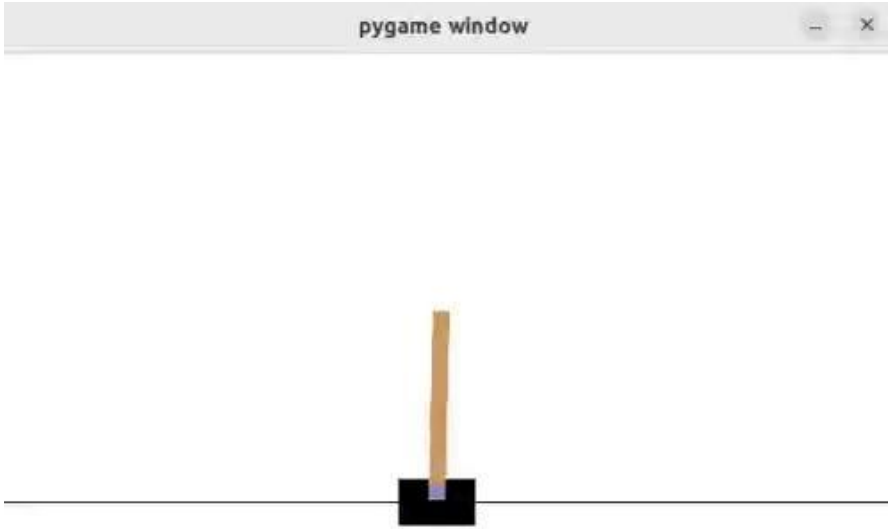
Cartpole

CartPole-v0

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. 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.




average score
episode
0
100
200
300
400
0
50
100
150
200
250
300


pygame window

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

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.

1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

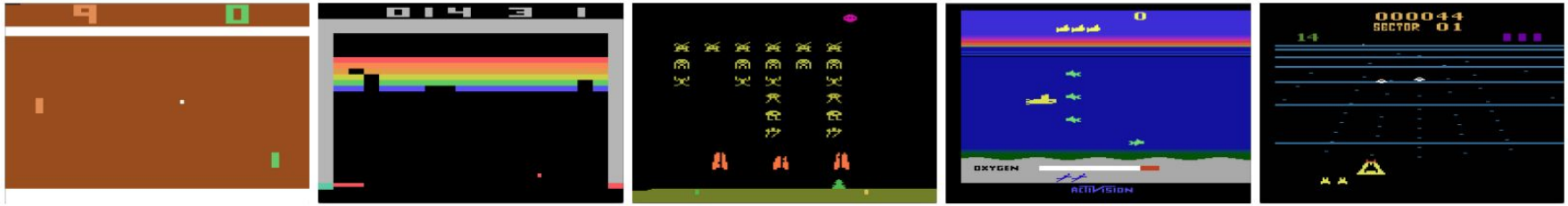


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

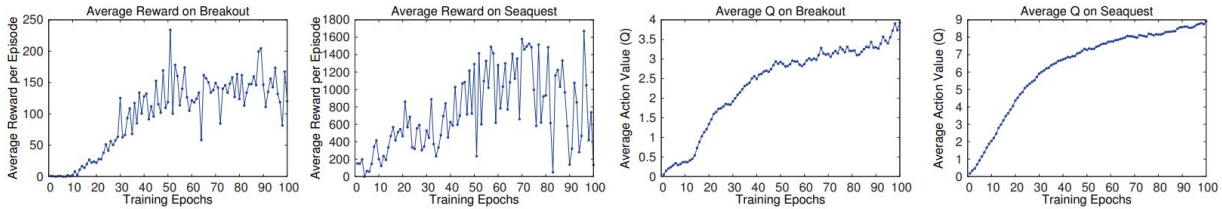


Figure 2: The two plots on the left show average reward per episode on Breakout and Seaquest respectively during training. The statistics were computed by running an ϵ-greedy policy with $\epsilon = 0.05$ for 10000 steps. The two plots on the right show the average maximum predicted action-value of a held out set of states on Breakout and Seaquest respectively. One epoch corresponds to 50000 minibatch weight updates or roughly 30 minutes of training time.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

explore-exploit dilemma

- Once you have figured out a strategy to collect a certain number of rewards, should you stick with it or experiment with something that could result in even bigger rewards?

```
import gym
import collections
import random

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

#Hyperparameters
learning_rate = 0.0005
gamma         = 0.98
buffer_limit  = 50000
batch_size    = 32
```

```
class ReplayBuffer():
    def __init__(self):
        self.buffer = collections.deque(maxlen=buffer_limit)

    def put(self, transition):
        self.buffer.append(transition)

    def sample(self, n):
        mini_batch = random.sample(self.buffer, n)
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []

        for transition in mini_batch:
            s, a, r, s_prime, done_mask = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r])
            s_prime_lst.append(s_prime)
            done_mask_lst.append([done_mask])

        return torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
               torch.tensor(r_lst), torch.tensor(s_prime_lst, dtype=torch.float), \
               torch.tensor(done_mask_lst)

    def size(self):
        return len(self.buffer)
```

```
class Qnet(nn.Module):
    def __init__(self):
        super(Qnet, self).__init__()
        self.fc1 = nn.Linear(4, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def sample_action(self, obs, epsilon):
        out = self.forward(obs)
        coin = random.random()
        if coin < epsilon:
            return random.randint(0,1)
        else :
            return out.argmax().item()
```

```
def train(q, q_target, memory, optimizer):
    for i in range(10):
        s,a,r,s_prime,done_mask = memory.sample(batch_size)

        q_out = q(s)
        q_a = q_out.gather(1,a)
        max_q_prime = q_target(s_prime).max(1)[0].unsqueeze(1)
        target = r + gamma * max_q_prime * done_mask
        loss = F.smooth_l1_loss(q_a, target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
def main():
    env = gym.make('CartPole-v1')
    q = Qnet()
    q_target = Qnet()
    q_target.load_state_dict(q.state_dict())
    memory = ReplayBuffer()
    print_interval = 20
    score = 0.0
    optimizer = optim.Adam(q.parameters(), lr=learning_rate)
    for n_epi in range(10000):
        epsilon = max(0.01, 0.08 - 0.01*(n_epi/200)) #Linear annealing from 8% to 1%
        s, _ = env.reset()
        done = False
        while not done:
            a = q.sample_action(torch.from_numpy(s).float(), epsilon)
            s_prime, r, done, truncated, info = env.step(a)
            done_mask = 0.0 if done else 1.0
            memory.put((s,a,r/100.0,s_prime, done_mask))
            s = s_prime
            score += r
            if done:
                break

        if memory.size()>2000:
            train(q, q_target, memory, optimizer)

        if n_epi%print_interval==0 and n_epi!=0:
            q_target.load_state_dict(q.state_dict())
            print("n_episode :{}, score : {:.1f}, n_buffer : {}, eps : {:.1f}%".format(
                                                    n_epi, score/print_interval, memory.size(), epsilon*100))
            score = 0.0
    env.close()

if __name__ == '__main__':
    main()
```

Lab 25

Train a DQN model with Cartpole environment.

Capture a video clip for your trained model and then upload it to YouTube.

Submit

1. Your code
2. Image of reward curve
3. The link for your YouTube clip