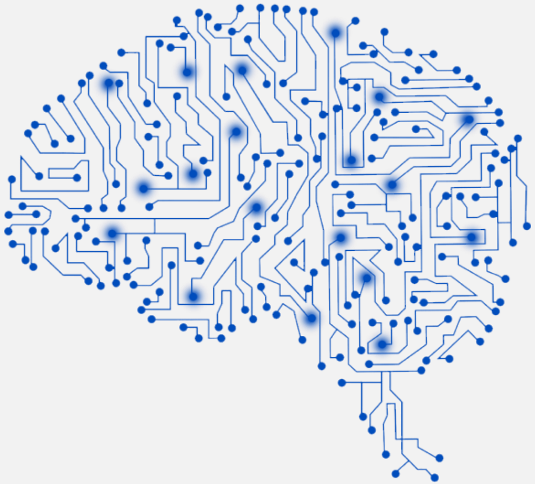


Deep Reinforcement Learning

Prof. Joongheon Kim
Korea University, Seoul, Korea
https://joongheon.github.io
joongheon@korea.ac.kr

Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

- **<u>Basics</u>**
- Q-Learning Implementation
- DQN Implementation

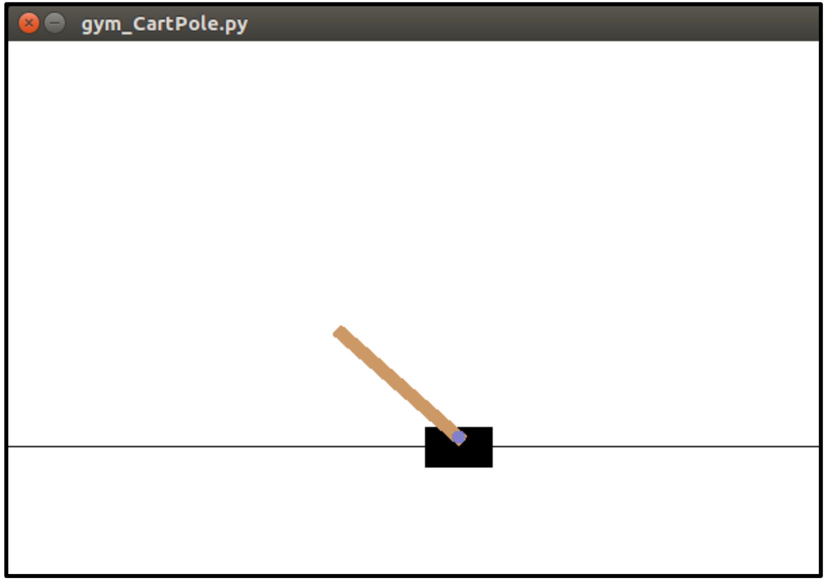
Imitation Learning and Autonomous Driving

Deep Reinforcement Learning
DRL Implementation

- **<u>Basics</u>**
- Q-Learning Implementation
- DQN Implementation

Basics, Hello World: CartPole

```python
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    #env.step(action)
```



Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

- Basics
- **Q-Learning Implementation**
- DQN Implementation

DDPG-based Vehicular Caching

Imitation Learning and Autonomous Driving

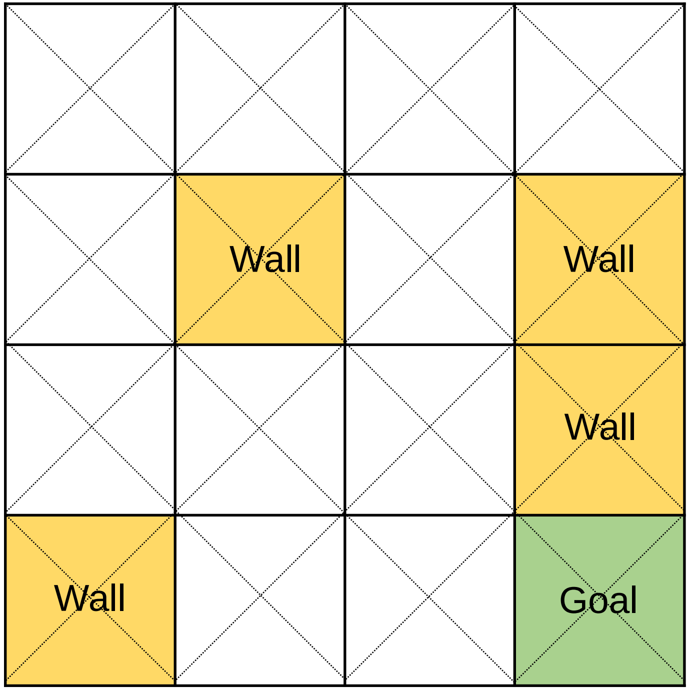
Deep Reinforcement Learning
DRL Implementation

- Basics
- **Q-Learning Implementation**
- DQN Implementation

Outline

- Q-Learning Implementation
  - **Q-Learning (Basics)**
  - Q-Learning (Exploit and Exploration)

Q-Learning (Basics)



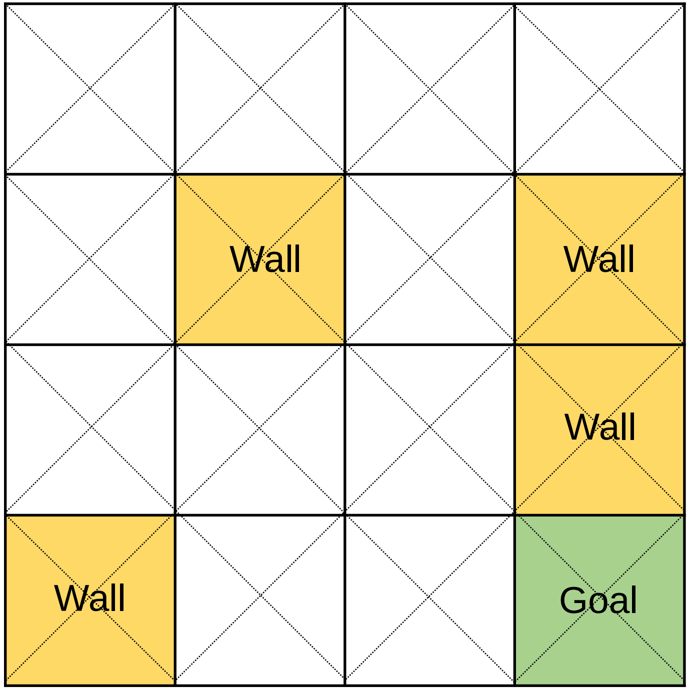

```python
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym.envs.registration import register
import random

'''
Q-Table
             action |  L  |  D  |  R  |  U |
----------------------------------------
state: 0            |     |     |     |    |
----------------------------------------
state: 1            |     |     |     |    |
----------------------------------------
state: 2            |     |     |     |    |
----------------------------------------
state: ...          |     |     |     |    |
----------------------------------------
'''

register(
    id='FrozenLake-v3',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={
        'map_name': '4x4',
        'is_slippery': False
    }
)

env = gym.make("FrozenLake-v3")
```

Q-Learning (Basics)




```python
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym.envs.registration import register
import random

'''
Q-Table
          action |  L  |  D  |  R  |  U |
----------------------------------------
state: 0         |     |     |     |    |
----------------------------------------
state: 1         |     |     |     |    |
----------------------------------------
state: 2         |     |     |     |    |
----------------------------------------
state: ...       |     |     |     |    |
----------------------------------------
'''

register(
    id='FrozenLake-v3',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={
        'map_name': '4x4',
        'is_slippery': False
    }
)

env = gym.make("FrozenLake-v3")
```

- Environment setting

Q-Learning (Basics)

```python
# Initialization with 0 in Q-table
Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
num_episodes = 1000 # Number of iterations

rList = []
successRate = []

def rargmax(vector):
    m = np.amax(vector) # Return the maximum of an array or maximum along an axis (0 or 1)
    indices = np.nonzero(vector == m)[0] # np.nonzero(True/False vector) => find the maximum
    return random.choice(indices) # Random selection

for i in range(num_episodes): # Updates with num_episodes iterations
    state = env.reset() # Reset
    total_reward = 0 # Reward graph (1: success, 0: failure)
    done = None

    while not done: # The agent is not in the goal yet
        action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
        new_state, reward, done, _ = env.step(action) # Result of the chosen action

        Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
        total_reward += reward
        state = new_state

    rList.append(total_reward) # Reward appending
    successRate.append(sum(rList)/(i+1)) # Success rate appending
```

Q-Learning (Basics)

```python
# Initialization with 0 in Q-table
Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
num_episodes = 1000 # Number of iterations

rList = []
successRate = []

def rargmax(vector):
    m = np.amax(vector) # Return the maximum of an array or maximum along an axis (0 or 1)
    indices = np.nonzero(vector == m)[0] # np.nonzero(True/False vector) => find the maximum
    return random.choice(indices) # Random selection

for i in range(num_episodes): # Updates with num_episodes iterations
    state = env.reset() # Reset
    total_reward = 0 # Reward graph (1: success, 0: failure)
    done = None

    while not done: # The agent is not in the goal yet
        action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
        new_state, reward, done, _ = env.step(action) # Result of the chosen action

        Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
        total_reward += reward
        state = new_state

    rList.append(total_reward) # Reward appending
    successRate.append(sum(rList)/(i+1)) # Success rate appending
```

- Randomly pick one when multiple argmax values exist

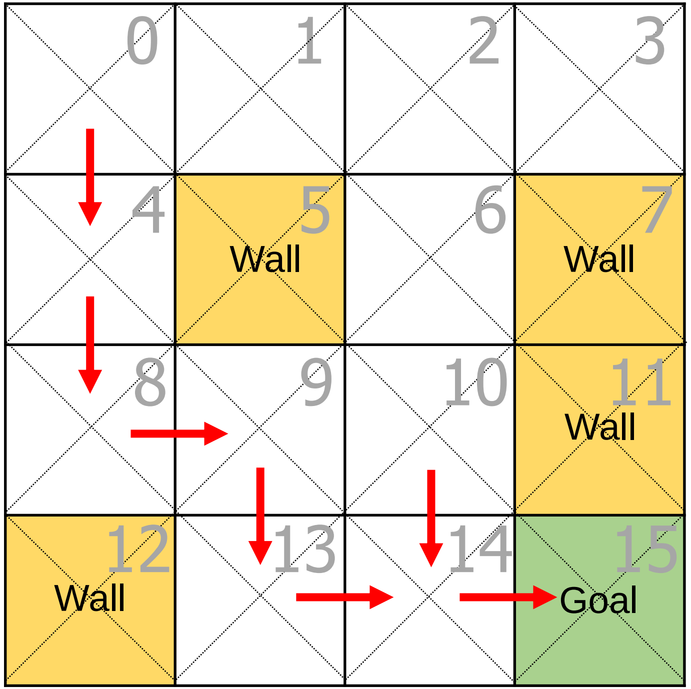
Q-Learning (Basics)

```
32  # Initialization with 0 in Q-table
33  Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
34  num_episodes = 1000 # Number of iterations
35
36  rList = []
37  successRate = []
38
39  def rargmax(vector):
40      m = np.amax(vector) # Return the max
41      indices = np.nonzero(vector == m)[0]
42      return random.choice(indices) # Rand
43
44  for i in range(num_episodes): # Updates
45      state = env.reset() # Reset
46      total_reward = 0 # Reward graph (1:
47      done = None
48
49      while not done: # The agent is not in the goal yet
50          action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
51          new_state, reward, done, _ = env.step(action) # Result of the chosen action
52
53          Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
54          total_reward += reward
55          state = new_state
56
57      rList.append(total_reward) # Reward appending
58      successRate.append(sum(rList)/(i+1)) # Success rate appending
```

- Iteration until the agent arrives at the goal or it cannot move anymore.
- (line 50) find the action which returns max Q value.
- (line 51) take the action which is the result of (line 50).
  - done: if the agent is at goal or cannot move anymore, done → True
- (line 53) Q-update
- (line 54) reward value accumulation
- (line 55) state value update for next iteration

[L, D, R, U]

```
'''
Final Q-Table
[[0. 1. 0. 0.]   0 (D)
 [0. 0. 0. 0.]   1
 [0. 0. 0. 0.]   2
 [0. 0. 0. 0.]   3
 [0. 1. 0. 0.]   4 (D)
 [0. 0. 0. 0.]   5
 [0. 0. 0. 0.]   6
 [0. 0. 0. 0.]   7
 [0. 0. 1. 0.]   8 (R)
 [0. 1. 0. 0.]   9 (D)
 [0. 1. 0. 0.]   10 (D)
 [0. 0. 0. 0.]   11
 [0. 0. 0. 0.]   12
 [0. 0. 1. 0.]   13 (R)
 [0. 0. 1. 0.]   14 (R)
 [0. 0. 0. 0.]]  15
Success Rate :  0.903
'''
```



Outline

- Q-Learning Implementation
  - Q-Learning (Basics)
  - **Q-Learning (Exploit and Exploration)**

Q-Learning (Exploit and Exploration)

```
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym.envs.registration import register
import random

register(
    id='FrozenLake-v3',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={
        'map_name': '4x4',
        'is_slippery': False
    }
)

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

Q = np.zeros([env.observation_space.n, env.action_space.n])
num_episodes = 1000

rList = []
successRate = []
e = 0.1 # exploit and exploration

mode = input(
    "Mode Selection [(1) e-greedy (2) decaying e-greedy (3) random noise (etc) original]: ")
r = 0.9 # discount factor
```

Q-Learning (Exploit and Exploration)

```python
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym.envs.registration import register
import random

register(
    id='FrozenLake-v3',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={
        'map_name': '4x4',
        'is_slippery': False
    }
)

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

Q = np.zeros([env.observation_space.n, env.action_space.n])
num_episodes = 1000

rList = []
successRate = []
e = 0.1 # exploit and exploration

mode = input(
    "Mode Selection [(1) e-greedy (2) decaying e-greedy (3) random noise (etc) original]: ")
r = 0.9 # discount factor
```

- Parameter setting

Q-Learning (Exploit and Exploration)

```python
def rargmax(vector):
    m = np.amax(vector)
    indices = np.nonzero(vector == m)[0]
    return random.choice(indices)

for i in range(num_episodes):
    state = env.reset()
    total_reward = 0
    done = None

    while not done:
        rand = random.random()
        # e-greedy / decaying e-greedy
        if (mode == '1' and rand < e) or (mode == '2' and (rand < e / (i + 1))):
            action = env.action_space.sample()
        # random noise
        elif mode == '3':
            action = rargmax(
                Q[state, :] + np.random.random(env.action_space.n) / (i + 1))
        # original
        else:
            action = rargmax(Q[state, :])

        new_state, reward, done, _ = env.step(action)
        Q[state, action] = reward + r * np.max(Q[new_state, :])
        total_reward += reward
        state = new_state

    rList.append(total_reward)
    successRate.append(sum(rList) / (i + 1))
```

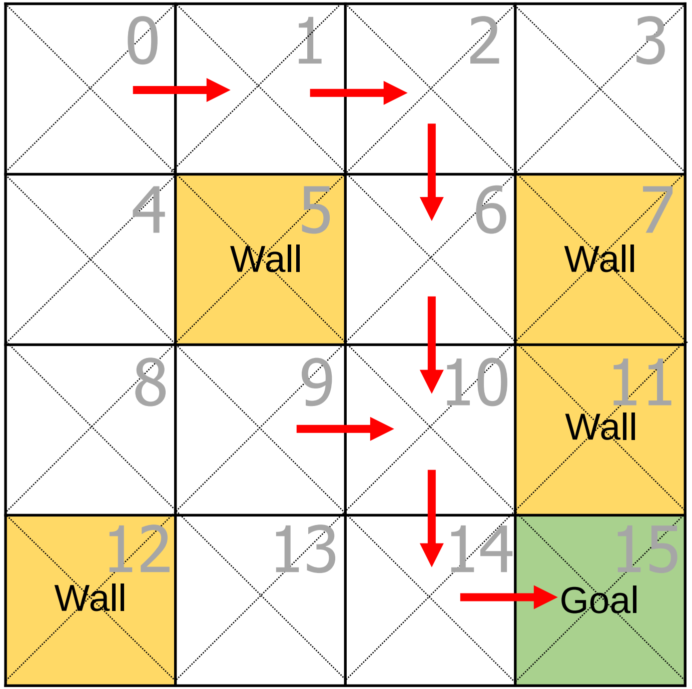
Q-Learning (Exploit and Exploration)

```
print("Final Q-Table")
print(Q)
print("Success Rate : ", successRate[-1])
plt.plot(range(len(rList)), rList)
plt.plot(range(len(successRate)), successRate)
plt.show()
'''
Mode Selection [(1) e-greedy (2) decaying e-greedy (3) random noise (etc) original]: 2
Final Q-Table
[[0.       0.       0.59049 0.      ]
 [0.       0.       0.6561  0.      ]
 [0.59049 0.729    0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.81     0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.       0.81     0.      ]
 [0.       0.9      0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.       0.       0.      ]
 [0.       0.       1.       0.      ]
 [0.       0.       0.       0.      ]]
Success Rate :  0.932
'''
```

Q-Learning (Exploit and Exploration)

[L, D, R, U]

```
Final Q-Table
[[0.        0.        0.59049 0.        ]   0 (R)
 [0.        0.        0.6561  0.        ]   1 (R)
 [0.59049 0.729     0.        0.        ]   2 (D)
 [0.        0.        0.        0.        ]   3
 [0.        0.        0.        0.        ]   4
 [0.        0.        0.        0.        ]   5
 [0.        0.81      0.        0.        ]   6 (D)
 [0.        0.        0.        0.        ]   7
 [0.        0.        0.        0.        ]   8
 [0.        0.        0.81      0.        ]   9 (R)
 [0.        0.9       0.        0.        ]   10 (D)
 [0.        0.        0.        0.        ]   11
 [0.        0.        0.        0.        ]   12
 [0.        0.        0.        0.        ]   13
 [0.        0.        1.        0.        ]   14 (R)
 [0.        0.        0.        0.        ]]  15
Success Rate :  0.932
```



Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

- Basics
- Q-Learning Implementation
- **DQN Implementation**

DDPG-based Vehicular Caching

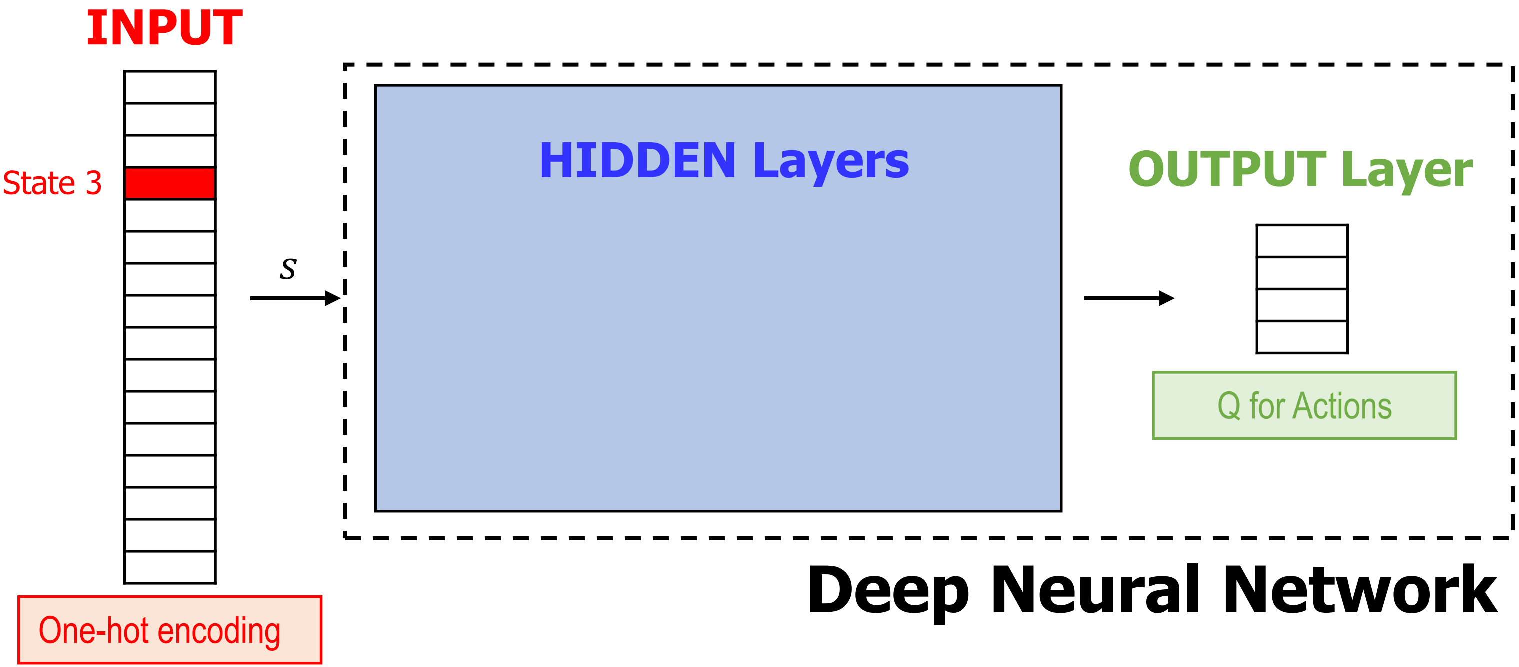
Imitation Learning and Autonomous Driving

Deep Reinforcement Learning
DRL Implementation

- Basics
- Q-Learning Implementation
- **DQN Implementation**

DQN (NIPS 2013)

- Frozen Lake
  - Input: States 0~15 (totally 16) → one-hot encoding
  - Output: 4 actions (totally 4) → Q-values for Up, Down, Left, and Right



DQN (NIPS 2013)

File Name: class_dqn.py

```
import tensorflow as tf
import numpy as np

class DQN:
    def __init__(self, session, input_size, output_size, name="main"):
        self.session = session
        self.input_size = input_size
        self.output_size = output_size
        self.net_name = name
        self._build_network()

    # 네트워크 구성 (레이어, 활성화 함수)
    def _build_network(self, h_size=6, l_rate=1e-1):
        with tf.variable_scope(self.net_name):
            self._X = tf.placeholder(tf.float32, [None, self.input_size], name="input_x")

            W1 = tf.get_variable("W1", shape=[
                                self.input_size, h_size], initializer=tf.contrib.layers.xavier_initializer())
            layer1 = tf.nn.tanh(tf.matmul(self._X, W1))

            W2 = tf.get_variable("W2", shape=[
                                h_size, self.output_size], initializer=tf.contrib.layers.xavier_initializer())
            self._Qpredict = tf.matmul(layer1, W2)

        self._Y = tf.placeholder(shape=[None, self.output_size], dtype=tf.float32)
        self._loss = tf.reduce_mean(tf.square(self._Y - self._Qpredict))
        self._train = tf.train.AdamOptimizer(learning_rate=l_rate).minimize(self._loss)

    # x에 대한 y 결과 리턴 (2단계 네트워크 지나온 결과) -> 관계식 적용 결과
    def predict(self, state):
        x = np.reshape(state, [1, self.input_size])
        return self.session.run(self._Qpredict, feed_dict={self._X:x})

    # 들어온 데이터를 바탕으로 W 업데이트 (학습 시킨다)
    def update(self, x_stack, y_stack):
        return self.session.run([self._loss, self._train], feed_dict={self._X:x_stack, self._Y:y_stack})
```

DQN (NIPS 2013)

File Name: dqn_2013.py

```
import numpy as np
import gym
from gym.envs.registration import register
import random
import tensorflow as tf
import class_dqn
from collections import deque
import matplotlib.pyplot as plt

env = gym.make("CartPole-v1")
env.max_episode_steps = 500

input_size = env.observation_space.shape[0]
output_size = env.action_space.n

learning_rate = 1e-1
discount_rate = 0.9
REPLAY_MEMORY = 50000
results = []

'''
DQN에는 두가지 문제가 있어 결과가 최상으로 나오지 않는다.
1) Correlation between samples
2) Non-stationary Targets
이 효과를 상쇄하기 위해서 2가지 방법을 사용하는데
하나는 레이어를 하나 더 만드는 것이고 또 하나는 버퍼에서 샘플링을 하여 학습에 사용하는 것이다.
'''
```

DQN (NIPS 2013)

```python
def simple_replay_train(DQN, train_batch):
    # Array of uninitialized (arbitrary) data of the given shape
    x_stack = np.empty(0).reshape(0, DQN.input_size)
    y_stack = np.empty(0).reshape(0, DQN.output_size)

    for state, action, reward, next_state, done in train_batch:
        # 현재 가지고 있는 W1, W2로 예측한 Q값
        Q = DQN.predict(state)

        if done:
            Q[0, action] = reward
        else:
            Q[0, action] = reward + discount_rate * \
                np.max(DQN.predict(next_state))

        # 트레이닝할 Data 만들기
        # X 값 : state, Y 값 : Q값 배열
        x_stack = np.vstack([x_stack, state])
        y_stack = np.vstack([y_stack, Q])

    # 금방 들어온 데이터로 학습시키기
    return DQN.update(x_stack, y_stack)

def bot_play(main_dqn):
    s = env.reset()
    reward_sum = 0
    while True:
        env.render()
        a = np.argmax(main_dqn.predict(s))
        s, reward, done, _ = env.step(a)
        reward_sum += reward
        if done:
            print("Total score : {}".format(reward_sum))
            break
```

DQN (NIPS 2013)

```python
def main():
    max_episodes = 2000
    replay_buffer = deque() # 표본을 저장할 버퍼, 이곳에서 샘플링을 하여 학습에 사용한다.

    with tf.Session() as sess:
        main_dqn = class_dqn.DQN(sess, input_size, output_size, name="main") # DQN 클래스는 Q-Network의 속성과 멤버함수가 작성되어 있다. ( dqn.py )
        tf.global_variables_initializer().run()

        for i in range(max_episodes):
            state = env.reset()
            e = 1. / ((i / 10) + 1)
            step_count = 0 # 한 번 테스트에 최대 몇번까지 움직였는가
            done = False

            while not done:
                # e-greedy 방식으로 action 선택
                if np.random.rand(1) < e:
                    action = env.action_space.sample()
                else:
                    # 현재 가지고 있는 W로 Q predict
                    action = np.argmax(main_dqn.predict(state)) # 현재 state에서 현재 가지고 있는 W, b 값으로 Q 계산 후 action 선택

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

                # 안 끝나는 게 좋은 거니까 끝나면 reward -10; 그리고 학습은 시키지 않아
                if done:
                    reward = -10

                # 현재 가지고 있는 W 값으로 얻은 결과를 쌓음
                replay_buffer.append((state, action, reward, next_state, done)) # 버퍼에 input과 output을 같이 저장
                if len(replay_buffer) > REPLAY_MEMORY: # 50000개까지만 저장하고 남으면 오래된 데이터 삭제
                    replay_buffer.popleft()

```

DQN (NIPS 2013)

```
                state = next_state
                step_count += 1
                # 너무 오래 하지 않도록 중간에 푸르닝
                if step_count > 10000:
                    break

            print("Episode : {}, steps : {}".format(i, step_count))
            results.append(step_count)

            if i % 10 == 1: # 10번 테스트 중 1번만 학습
                for _ in range(50):
                    minibatch = random.sample(replay_buffer, 10) # 버퍼에서 10개를 샘플링
                    loss, _ = simple_replay_train(main_dqn, minibatch) # 샘플링한 10개를 바탕으로 학습
                print("Loss : ", loss)

        bot_play(main_dqn) # 최종 결과로 마지막 게임 수행
        plt.title("(2013) Total step count on each episode")
        plt.plot(range(len(results)), results)
        plt.show()

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