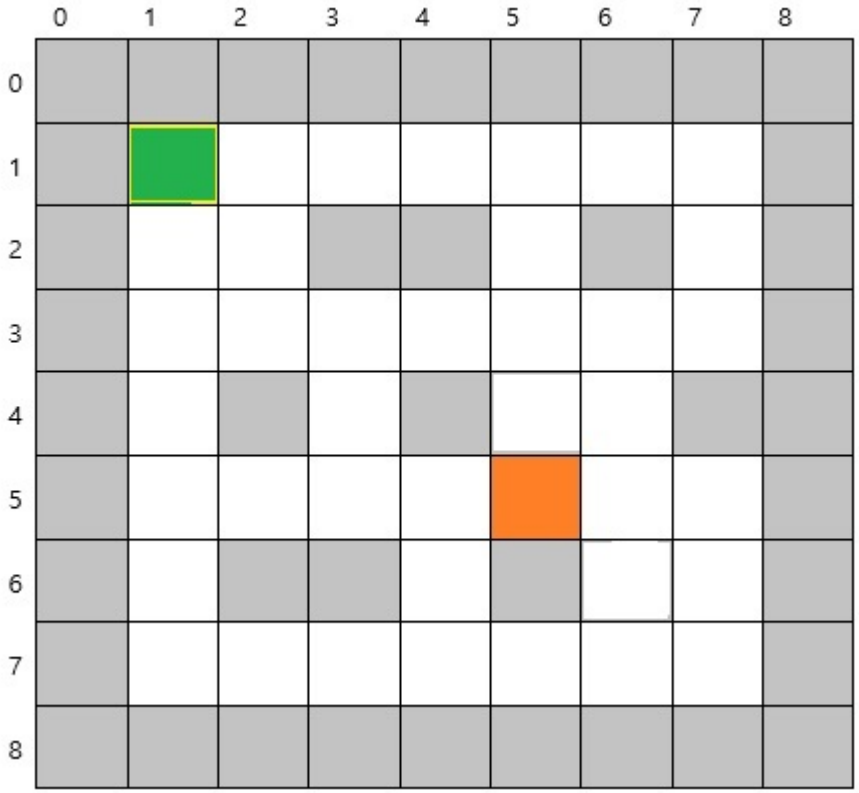
개발과제: DQN 을 이용한 frozen-lake 프로그램

- environment : 9X9 의 frozen lake



- cell 정의: 녹색: start; 주황색:goal; 흰색: frozen; 회색: hole.
- rewards:
  - F/S 로 갈 때 0; G 로 갈 때 9; H 로 갈 때 -9.

```
##  frozen-lake 문제에 대한 DQN 프로그램.
##

import numpy as np
import time
import random
import math
from datetime import datetime
from collections import namedtuple, deque
from itertools import count
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# if GPU is to be used
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

total_episodes = 60000       # Total number of episodes in training
max_steps = 99               # Max steps per episode in training.
gamma = 0.90                 # Discounting rate for expected return
Learning_rate = 0.00005      # 신경망 모델 learning rate (optimizer 에게 제공)
original_epsilon = 0.4       # Exploration rate
decay_rate = 0.000006        # Exponential decay rate for exploration.

TAU = 0.7                    # Q_net 파라메터를 Q_hat_net 로 copy 시에 반영 비율.
one_minus_TAU = 1 - TAU
```

packages importing

GPU 를 사용하기 위한 문장

전역변수들

```
memory_pos = 0    # replay_memory 내에 transition 을 넣을 다음 위치.
    # 0 에서 부터 커지다가  max_memory-1 까지 되면 다시 0 부터 시작함.

BATCH_SIZE = 16
model_update_cnt = 0  # Q_net 를 업데이트한 횟수.
copy_cnt = 4  # Q_net 업데이트를 copy_cnt 번 한 후마다 Q_hat_net 로 파라메터 복사.

max_memory = 2000   # capacity of the replay memory.

transition_cnt = 0    # 거쳐간  총 transition 수(episodes 간에 중단 없이)
# 매 (배치크기+작은 랜덤값) 마다 Q_net 의 parameter update 를 수행함.

random.seed(datetime.now().timestamp()) # give a new seed in random number generation.

# state space is defined as size_row X size_col array.
# The boundary cells are holes(H).
# S: start, G: goal, H:hole, F:frozen

max_row = 6
max_col = 6
n_actions = 4   # 0:up,    1:right,   2:down,    3:left.
n_observations = max_row * max_col  # total number of states
                  # 1-hot 벡터로 표현하므로 NN 입력의 신호수 = 총 state 수

env_state_space = \
 [ ['H', 'H', 'H', 'H', 'H', 'H'], \
   ['H', 'S', 'F', 'F', 'F', 'H'], \
   ['H', 'F', 'H', 'F', 'H', 'H'], \
   ['H', 'F', 'F', 'F', 'H', 'H'], \
   ['H', 'H', 'F', 'F', 'G', 'H'], \
   ['H', 'H', 'H', 'H', 'H', 'H'] ]
```

전역변수들


```
# offset of each move action:  up, right, down, left, respectively.
# a new state(location) = current state + offset of an action.

move_offset = [[-1,0], [0,1],   [1,0],  [0,-1]]
move_str =     ['up   ', 'right', 'down ', 'left ']

# replay memory: transition 들을 저장하는 버퍼.
# 저장되는 transition 의 4가지 정보: state_index, action, reward, next state index.
#   (주의: state 를 좌표 대신 상태번호(index) 로 나타냄.)
replay_memory = np.ndarray((max_memory, 4), dtype=int)
batch_transition = np.ndarray((BATCH_SIZE, 4), dtype=int)   # 배치 하나를 넣는데 사용.

is_replay_memory_full = 0   # 버퍼가 처음으로 완전히 채워지기 전에는 0. 그후로는 항상 1.
```

experience replay 를 위한 버퍼

학습 후 DQN 을 이용하여 q values 를 출력해 보는 데 이용하는 함수


```
def compute_and_print_Q_values (s):
  r = s[0]
  c = s[1]
  if env_state_space[r][c] == 'G' or env_state_space[r][c] == 'H':
    action_values = [ 0.0 for i in range(n_actions)]
    action_values = np.array(action_values)
  else:
    state_idx = r * m   col + c   # state 의 번호를 만듬.
    state_idx_list =  tate_idx]  # 배치 차원을 넣는다. 배치는 하나의 예제 입력만 가짐.
    states_tsr = tor .tensor(state_idx_list).to(device)  # state 한개 가짐
    one_hot_states_ sr = F.one_hot(states_tsr, num_classes= n_observations)
    one_hot_states_t  = one_hot_states_tsr.float().to(device)
    with torch.no_grad()
      state_action_values =  et(one_hot_states_tsr) # 주의: 출력은 2차원: (1, n_actions).
    state_action_values = state_  ion_values[0]  # 배치 차원을 없앤다.
    action_values = state_action_va  pu().numpy()

  text = "s[" + str(r) + "," + str(c) + "]: "
  for i in range(n_actions):
    #text = text + str(action_values[i]) + ",  "
    text = text + "{:5.2f}".format(action_values[i])+ ", "
  print(text)
```

```
# state s 에서 greedy 하게 action 을 고른다.
def choose_action_with_greedy(s):
  state_idx = s[0] * max_col + s[1]   # state 의 표현을 상태번호로 바꾼다.
  state_idx_list = [state_idx]  # 1-차원 데이터임.배치 차원을 넣은 것임.
  states_tsr = torch.tensor(state_idx_list).to(device)  # state 하나를 가지는 배열.
  one_hot_states_tsr = F.one_hot(state_idx_list, num_classes= n_observations)
  one_hot_states_tsr = one_hot_states_tsr.float().to(device)

  # get q-a values of all actions for state s.
  with torch.no_grad():
    state_action_values = Q_net(one_hot_states_tsr)  # 출력은 2차원: (1, n_actions).

  max_a = torch.argmax(state_action_values, dim=1) # 값이 최대인 액션 번호를 얻는다.
  max_a = max_a[0]  # 입력 하나에 대한 결과를 가지는 리스트에서 액션 하나를 꺼냄.
  return max_a
## end def choose_action_with_greedy(s):

# state s 에서 epsilon-greedy 방식으로 다음 action 을 고른다.
def choose_action_with_epsilon_greedy(s, epsilon):
  state_idx = s[0] * max_col + s[1]    # state 의 표현을 상태번호로 바꾼다.
  state_idx_list = [state_idx]  # 배치 차원을 넣어 줌. 배치에 1 개만 가짐.
  states_tsr = torch.tensor(state_idx_list).to(device)  # state 한 개 가짐
  one_hot_states_tsr = F.one_hot(states_tsr, num_classes= n_observations)
  one_hot_states_tsr = one_hot_states_tsr.float().to(device)

  # get q-a values of all actions for a state in batch having one state.
  with torch.no_grad():
    state_action_values = Q_net(one_hot_states_tsr)  # 주의: 출력은 2차원: (bsz, n_actions).

  max_a = torch.argmax(state_action_values, dim=1) # 값이 최대인 액션.
  max_a = max_a[0]

  rn = random.random() # 0 ~ 1 사이 random number.
  if rn >= epsilon: # epsilon 보다 크면, 최대확률을 가진 action 을 선택.
    action = max_a
  else:
    rn1 = random.random()
    # 4 개의 action 중 하나를 무작위로 선택.
    if rn1 >= 0.75:
      action = 0
    elif rn1>= 0.5:
    ...........
```

action 선택 함수들:
greedy,
ε-greedy

신경망 모델 정의 함수


```
# 모델 정의:  Function approximation 에 사용할 신경망 모델 구조를 정의한다.
class DQN(nn.Module):
  def __init__(self, n_observations, n_actions):
      super(DQN, self).__init__()
      self.layer1 = nn.Linear(n_observations, 128)
      self.layer2 = nn.Linear(128, 128)
      self.layer3 = nn.Linear(128, n_actions)

  # x 는 2차원 데이터: dim0: batch, dim1: state를 나타내는 1-hot 입력벡터.
  def forward(self, x):
      x = F.relu(self.layer1(x))
      x = F.relu(self.layer2(x))
      return self.layer3(x)
```

배치 하나를 이용하여 학습을 수행하는 함수: learning_by_a_batch

```
def learning_by_a_batch():
  state_batch = batch_tran[:,
  action_batch = batch_tr sition[:
  reward_batch = batch_tr sition[:
  next_state_batch = batc transition[:,3]

  # Q_net 신경망의 입력층에 배치 데이터 준비. state 를 1-hot 벡터로 표시함.
  state_batch_tsr = torch.fr th)
  one_hot_state_batch = F.one te_batch_tsr classes= n_observations)
  one_hot_state_batch = one_hot ch.float().to(device)

  # Q_net 의 출력은 state 마다에 대한 여러 action 들의 q(s,a) 값이다.
  prediction_Q_net = Q_net(one_hot_state_batch)  # 출력의 shape: (bsz, n_actions)

  state_action_values_tsr = torch.zeros([BATCH_SIZE,], dtype=torch.float64).to(device)
  for i in range(BATCH_SIZE):
    state_action_values_tsr[i] = prediction_Q_net[i,action_batch[i]]
  # shape of state_action_values: (batch_size)

  # s' 이 next state 일때 max_a{q(s',a)} 를 구하자.(batch 내의 모든 transition 마다)
  with torch.no_grad():
      next_state_batc sr = torch.from_numpy(next_state_batch)
      one_hot_next_state tch = F.one_hot(next_state_batch_tsr, num_classes=n_observations)
      one_hot_next_state_batch .float().to(device)
      result_target_net = Q_hat_net(one_hot_ne batch)
        ##shape of result_target_net: (bsz, 4)

      max_q_of_next_states_in_batch = torch.max(result_target_net, dim=1).values
        # 주의:  .max 함수가 2가지를 출력하므로 .values 를 이용함. 결과적으로 출력의 shape:(bsz)
```

함수: learning_by_a_batch (계속)

```
next_state_values = []
for i, st in enumerate(next_state_batch):
  r = int(st / max_col)
  c = st % max_col
  if env_state_space[r][c] == 'G' or env_state_space[r][c] == 'H':
    next_state_values.append(0)   # terminal state 의 q(s,a) value 는 0.
  else:
    # non-terminal state 는 계산된 max_a(q(s',a))
    next_state_values.append(max_q_of_next_states_in_batch[i].item())

next_state_values_tsr = torch.tensor(next_state_values).to(device)

# update target 를 준비한다: R + gamma*max_a{q(s',a)}
reward_batch_tsr = torch.from_numpy(reward_batch).to(device)
target_state_action_values_tsr = (next_state_values_tsr * gamma) + reward_batch_tsr

# Huber loss 로 loss 를 계산한다.
loss = criterion(state_action_values_tsr, target_state_action_values_tsr)

optimizer.zero_grad() # parameters 의 gradient 를 0 으로 초기화.

# backward computation: 모든 parameter 의 gradient 를 구한다.
loss.backward()

# In-place gradient clipping
torch.nn.utils.clip_grad_value_(Q_net.parameters(), 100)

# 모델의 모든 parameters 를 loss 의 gradient 를 이용하여 update 한다.
optimizer.step()
```

두 개의 신경망모델, optimizer, criterion 함수 설정


```
# 인공신경망 모델 두 개를 만든다:
#   Q_net:  policy 를 나타내는 main 신경망모델 (policy net or prediction net 이라고 부름)
#   Q_hat_net: target 값을 생성하는 신경망모델 (target net 이라고 부름)

Q_net = DQN(n_observations, n_actions).to(device)
Q_hat_net = DQN(n_observations, n_actions).to(device)

# target_net 에 policy_net 의 파라메터를 복사해서 완전히 같은 모델로 초기화함.
Q_hat_net.load_state_dict(Q_net.state_dict())

optimizer = optim.AdamW(Q_net.parameters(), lr=Learning_rate, amsgrad=True)
criterion = nn.SmoothL1Loss()
```

학습단계


```
#########################################################################################################
#   (1)   학습 단계
#########################################################################################################
if torch.cuda.is_available():
    num_episodes = total_episodes
else:
    num_episodes = 200

start_state = [1,1]
print("\n학습단계 시작.\n")

for i_episode in range(num_episodes):
  # get starting state
  S = start_state

  epsilon = original_epsilon * math.exp(-decay_rate*i_episode)  # epsilon 약간 감소시킴.

  if i_episode != 0 and i_episode % 4000 == 0:
    print('episode=', i_episode, '  epsilon=', epsilon)

  for t in range(max_steps):

    A = choose_action_with_epsilon_greedy(S, epsilon)

    # take action A to observe reward R, and new state S_.
    S_ , R = get_new_state_and_reward(S, A)

    # transition 하나를 replay memory 에 저장.
    s_idx = S[0] * max_col + S[1]
    next_s_idx = S_[0] * max_col + S_[1]

    replay_memory[memory_pos, 0] = s_idx
    replay_memory[memory_pos, 1] = A
    replay_memory[memory_pos, 2] = R
    replay_memory[memory_pos, 3] = next_s_idx
```

```
# replay memory 버퍼가 처음으로 완전히 차면, is_replay_memory_full 에 1 을 넣는다.
if is_replay_memory_full == 0 and memory_pos == max_memory-1:
  is_replay_memory_full =  1

# 다음 번에 넣을 위치를 정해 놓는다.
memory_pos = (memory_pos + 1) % max_memory

# Move to the next state
S = S_

# replay_memory 로 보낸 총 transition 총 개수.
transition_cnt += 1

random_number = random.randint(0, int(BATCH_SIZE/2) )

if transition_cnt >= (BATCH_SIZE+3) and transition_cnt % (BATCH_SIZE+random_number) == 0:

  ##### transition 들을 가져와서 배치 1 개를 만든다. 결과는 batch_transition 에 있다.###############

  #  replay_memory 에서 꺼내올 위치들을 random 으로 선정하여 random_number 리스트에 넣는다.
  if is_replay_memory_full == 1:
    #전체 영역에서 가져옴
    random_numbers = random.sample(range(0, max_memory), BATCH_SIZE)
  else:
    # 아직 버퍼가 완전히 차지 않은 상태임. 버퍼의 채워져 있는 부분에서 가져옴.
    random_numbers = random.sample(range(0, memory_pos-1), BATCH_SIZE)

  # replay_memory 에서 transition들을 꺼내 와서 배치 하나를 batch_transition 에 준비한다.
  for i in range(BATCH_SIZE):
    rnum = random_numbers[i]
    batch_transition[i,:] = replay_memory[rnum,:]
  ################## batch_transition 에 배치준비 완료 ##############################################
```

```
        # 배치 하나를 이용하여 모델을 훈련(parameter updating)시킨다.
        learning_by_a_batch()

        model_update_cnt += 1  # 모델이 update 된 총 횟수.

        # Q_net 의 parameter update 를 여러 번(copy_cnt번) 수행한 후에 Q_hat_net 의 parameter 를 복사해 온다.
        if model_update_cnt % copy_cnt == 0:

          # soft 복사 사용:  Q_net 의 parameter 값을 일부만 복사함(복사비율: TAU)
          Q_hat_net_state_dict = Q_hat_net.state_dict()
          Q_net_state_dict = Q_net.state_dict()
          for key in Q_net_state_dict:
              Q_hat_net_state_dict[key] = Q_net_state_dict[key]*TAU + Q_hat_net_state_dict[key]*one_minus_TAU
          Q_hat_net.load_state_dict(Q_hat_net_state_dict)

      # terminal state 에 도달하면 episode를 종료한다.
      if env_state_space[S[0]][S[1]] == 'G' or env_state_space[S[0]][S[1]] == 'H':
        break

print('학습단계 종료.\n')
```

테스트단계


```
print("테스트 단계 시작.\n")

for e in range(4):
  S = start_state
  total_reward = 0
  print("\nEpisode = ", e, "   start state: (",S[0], ",", S[1], ")")
  leng = 0
  for i in range(99):
    A = choose_action_with_greedy(S)
    S_, R = get_new_state_and_reward(S,A)
    print("the move is ", move_str[A], "  to (", S_[0], ",", S_[1], ")")
    leng += 1
    total_reward += R
    S = S_
    if env_state_space[S[0]][S[1]] == 'G' or env_state_space[S[0]][S[1]] == 'H':
      break
  print("episode ends. episode length = ", leng, ". total reward = ", total_reward)

print("테스트단계 종료.")

# 모든 state-action pair 들의 q 값을 Q_net 를 이용하여 출력하여 본다.
print("\n학습 후의 Q values:")
for i in range(max_row):
  for j in range(max_col):
    s = [i, j]  # a state.
    compute_and_print_Q_values(s)

print("프로그램 종료!")
```

```
학습단계 시작.

episode= 4000   epsilon= 0.39051428390316373
episode= 8000   epsilon= 0.3812535148310019
episode= 12000   epsilon= 0.3722123583244823
episode= 16000   epsilon= 0.3633856064274825
episode= 20000   epsilon= 0.354768174686863
episode= 24000   epsilon= 0.346355099223682
episode= 28000   epsilon= 0.33814153387386353
episode= 32000   epsilon= 0.33012274739667297
episode= 36000   epsilon= 0.3222941207493919
episode= 40000   epsilon= 0.31465114442662134
episode= 44000   epsilon= 0.30718941586268245
episode= 48000   epsilon= 0.2999046368956165
episode= 52000   epsilon= 0.29279261129132506
episode= 56000   epsilon= 0.2858492423264229
학습단계 종료.

테스트 단계 시작.


Episode =  0    start state: ( 1 , 1 )
the move is  right   to ( 1 , 2 )
the move is  right   to ( 1 , 3 )
the move is  down    to ( 2 , 3 )
the move is  down    to ( 3 , 3 )
the move is  down    to ( 4 , 3 )
the move is  right   to ( 4 , 4 )
episode ends. episode length =  6 . total reward =  9

Episode =  1    start state: ( 1 , 1 )
the move is  right   to ( 1 , 2 )
the move is  right   to ( 1 , 3 )
the move is  down    to ( 2 , 3 )
the move is  down    to ( 3 , 3 )
the move is  down    to ( 4 , 3 )
```

```
학습 후의 Q values:
s[0,0]:  0.00,  0.00,  0.00,  0.00,
s[0,1]:  0.00,  0.00,  0.00,  0.00,
s[0,2]:  0.00,  0.00,  0.00,  0.00,
s[0,3]:  0.00,  0.00,  0.00,  0.00,
s[0,4]:  0.00,  0.00,  0.00,  0.00,
s[0,5]:  0.00,  0.00,  0.00,  0.00,
s[1,0]:  0.00,  0.00,  0.00,  0.00,
s[1,1]: -9.01,  5.29,  5.23, -9.01,
s[1,2]: -8.83,  5.89, -8.97,  4.76,
s[1,3]: -9.02,  5.26,  6.54,  5.31,
s[1,4]: -3.38,  1.54, -5.37,  5.86,
s[1,5]:  0.00,  0.00,  0.00,  0.00,
s[2,0]:  0.00,  0.00,  0.00,  0.00,
s[2,1]:  4.73, -8.95,  5.83, -8.61,
s[2,2]:  0.00,  0.00,  0.00,  0.00,
s[2,3]:  5.81, -9.15,  7.27, -9.02,
s[2,4]:  0.00,  0.00,  0.00,  0.00,
s[2,5]:  0.00,  0.00,  0.00,  0.00,
s[3,0]:  0.00,  0.00,  0.00,  0.00,
s[3,1]: -0.53,  6.46, -8.64, -2.67,
s[3,2]: -8.55,  7.19,  6.05,  5.86,
s[3,3]:  6.58, -8.87,  8.08,  6.42,
s[3,4]:  0.00,  0.00,  0.00,  0.00,
s[3,5]:  0.00,  0.00,  0.00,  0.00,
s[4,0]:  0.00,  0.00,  0.00,  0.00,
s[4,1]:  0.00,  0.00,  0.00,  0.00,
s[4,2]: -2.58,  8.08, -5.47, -0.46,
s[4,3]:  0.13,  8.96, -9.08,  7.18,
s[4,4]:  0.00,  0.00,  0.00,  0.00,
s[4,5]:  0.00,  0.00,  0.00,  0.00,
s[5,0]:  0.00,  0.00,  0.00,  0.00,
s[5,1]:  0.00,  0.00,  0.00,  0.00,
s[5,2]:  0.00,  0.00,  0.00,  0.00,
s[5,3]:  0.00,  0.00,  0.00,  0.00,
s[5,4]:  0.00,  0.00,  0.00,  0.00,
s[5,5]:  0.00,  0.00,  0.00,  0.00,
프로그램 종료!
```