Deep Reinforcement Learning

Younghoon Kim

nongaussian@hanyang.ac.kr

Keywords

- How to define a custom Gym environments ★★★★★
- Deep Q-learning ★★★★★

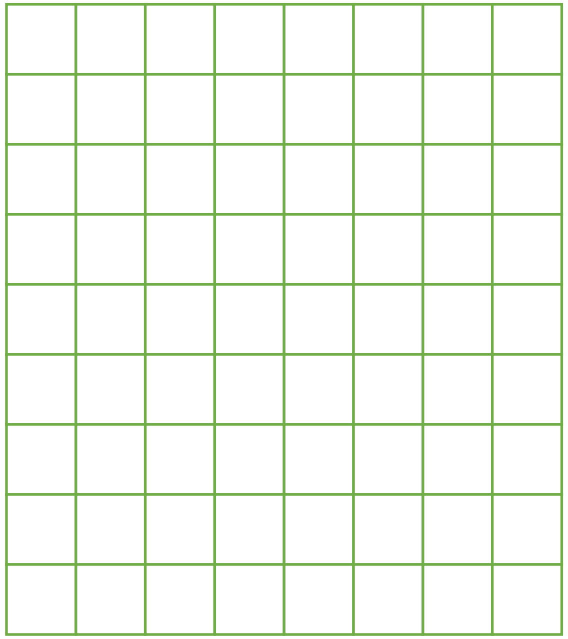
Custom Gym Environments

Shooting Airplane Gan

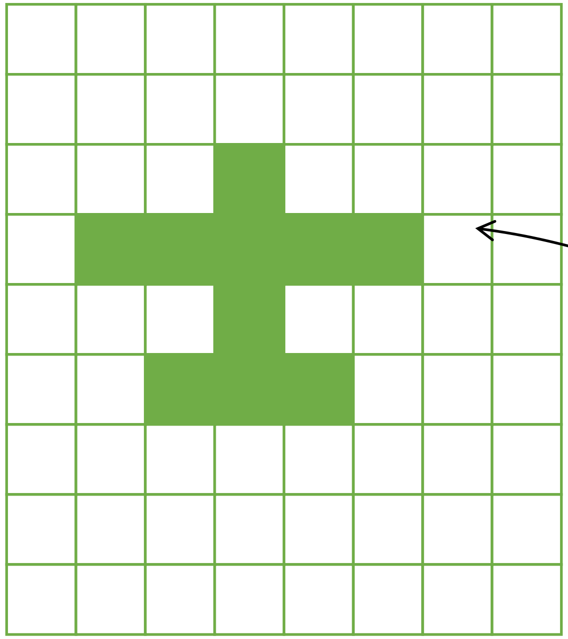
게임을 시작할 때 플레이어는 자신의 보드에 '비행기' 개체(아래 애니메이션에서 플레이어의 보드에서 볼 수 있듯이 '비행기' 모양을 형성하는 녹색 셀 10개)를 가지고 있습니다.

- 각 게임마다 비행기 셀은 무작위로 결정되지만 플레이어에게는 표시되지 않습니다.
- 플레이어는 숨겨진 비행기의 모든 셀을 맞추기 위해 셀을 선택합니다.
- 비행기를 공격하는 라운드 수가 적을수록 더 높은 점수를 얻습니다.

플레이어는 자신이 총에 맞은 위치만 볼 수 있으며, 빨간색과 노란색 포탄은 각각 명중과 빗맞음을 나타냅니다.



<Initial board>

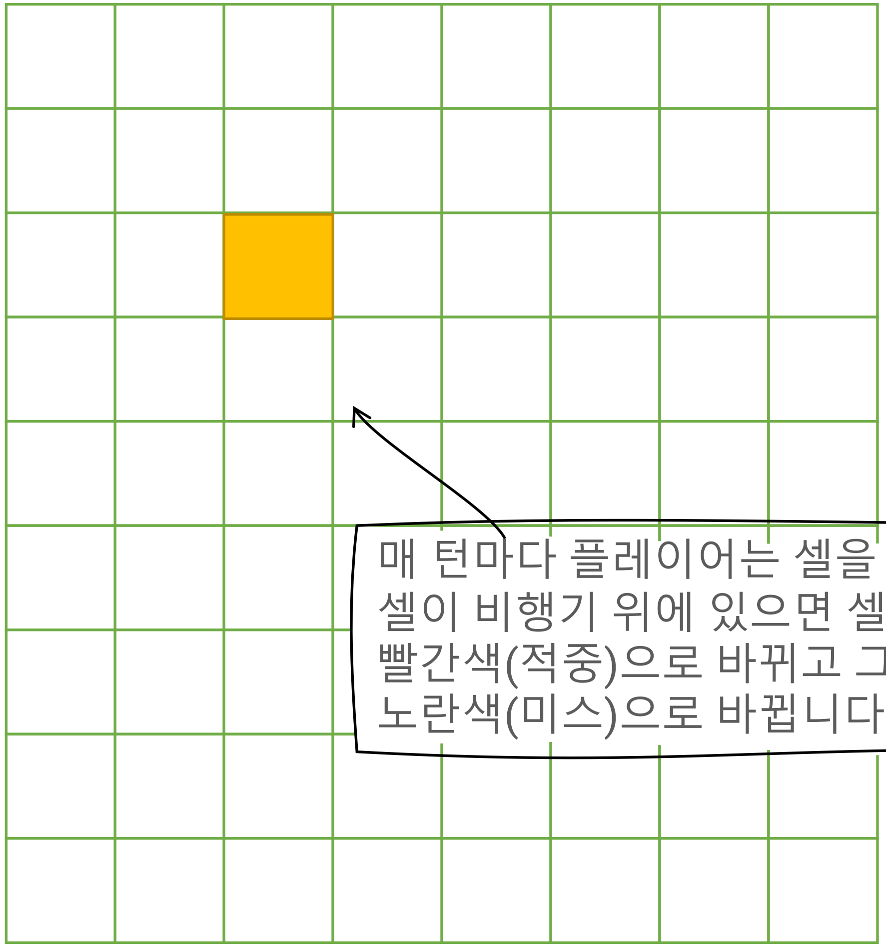


<Hidden board>

이 '비행기'는 무작위로 배치되어 플레이어에게 숨겨집니다. (비행기의 위치, 방향)

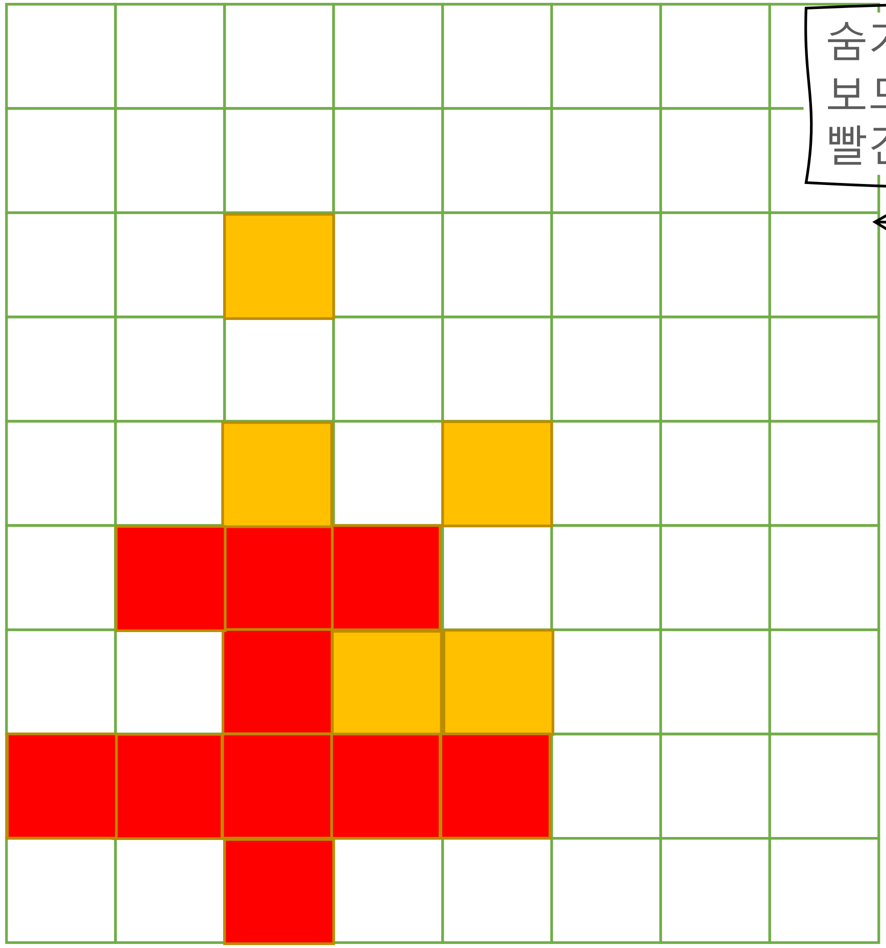
Shooting Airplane Game

플레이어는 보드의 각
셀을 공격할 수 있습니다.



매 턴마다 플레이어는 셀을 선택하고
셀이 비행기 위에 있으면 셀의 색이
빨간색(적중)으로 바뀌고 그렇지 않으면
노란색(미스)으로 바뀝니다.

Shooting Airplane Game



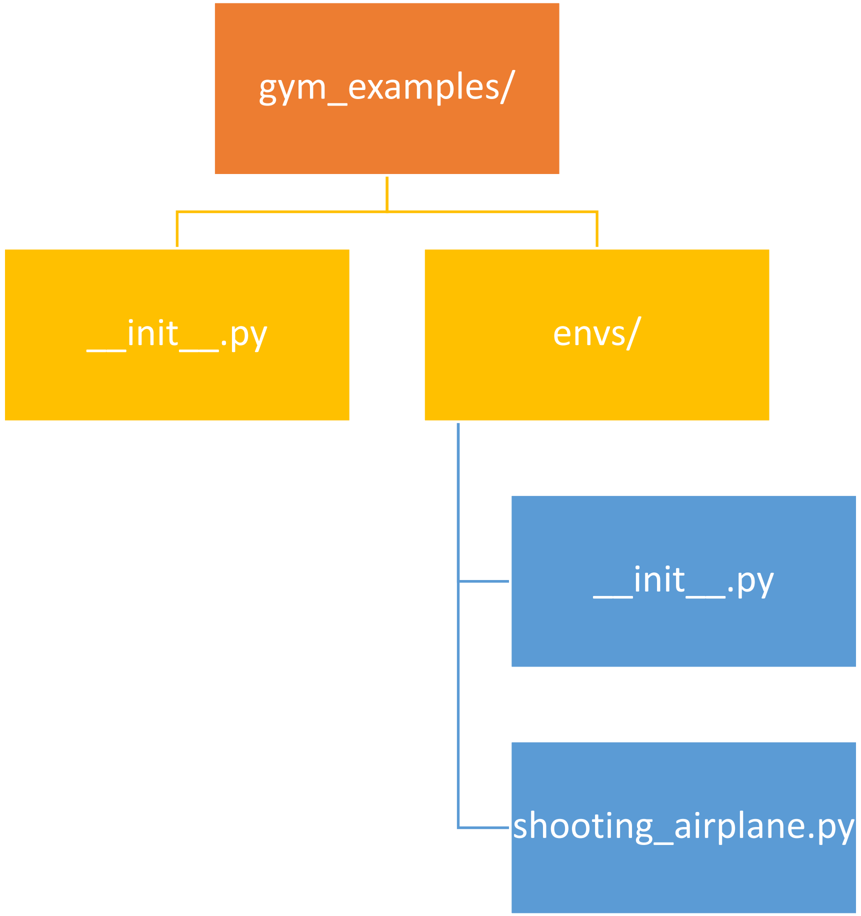
숨겨진 비행기를 공격하기 위해 반복하여 보드의 10 칸 (비행기의 모든 칸)이 빨간색으로 바뀌면 게임이 중지됩니다.

A Custom Gym Environment

- Subclassing gym.Env
  - Define observation and action spaces, and a step method
- Registering Envs
  - Register your custom env so that you can create an env as, for instance,

```
import gym
env = gym.make(
    'gym_examples:gym_examples/ShootingAirplane-v0',
    render_mode="text")
```

Step 1: Files & Directories



Step 2: Subclassing gym.Env

Github 홈페이지에서 zip 파일 다운로드 하여 디렉토리 채로 copy

```
import gym
from gym import spaces
import numpy as np

class ShootingAirplaneEnv(gym.Env):
    metadata = {"render_modes": ["text"], "render_fps": 4}

    def __init__(self, render_mode=None, size=8):
        self.size = size # The size of the square grid

        assert render_mode is None or render_mode in self.metadata["render_modes"]
        self.render_mode = render_mode

        # Observation: for each cell in size x size matrix, three states involving
        # { unseen (0), hit (1), miss (2) }
        self.observation_space = spaces.Box(low=0, high=255, shape=(size, size, 1), dtype=np.uint8)

        # We have size x size actions, (row, column) index to shoot
        self.action_space = spaces.MultiDiscrete([size, size])

        # Direction: 0
        #                 (0, -1)
        # (-2, 0) (-1, 0) (0, 0) (1, 0) (2, 0)
        #                 (0, 1)
        #         (-1, 2) (0, 2) (1, 2)
```

```python
        # Direction: 0
        #                 (0, -1)
        # (-2, 0) (-1, 0) (0, 0) (1, 0) (2, 0)
        #                 (0, 1)
        #         (-1, 2) (0, 2) (1, 2)
        # Direction: 1
        #       (-1, -2) (0, -2) (1, -2)
        #                (0, -1)
        # (-2, 0) (-1, 0) (0, 0) (1, 0) (2, 0)
        #                (0, 1)
        # Direction: 2
        #        (0, -2)
        #        (0, -1).     (2, -1)
        # (-1, 0) (0, 0) (1, 0) (2, 0)
        #        (0, 1)         (2, 1)
        #        (0, 2)
        # Direction: 3
        #                (0, -2)
        #(-2, -1)        (0, -1)
        #(-2, 0) (-1, 0) (0, 0) (1, 0)
        #(-2, 1)         (0, 1)
        #                (0, 2)

        self._relative_pos = np.array([
          [(0, -1), (-2, 0), (-1, 0), (0, 0), (1, 0), (2, 0), (0, 1), (-1, 2), (0, 2), (1, 2)],
          [(0, 1), (-2, 0), (-1, 0), (0, 0), (1, 0), (2, 0), (0, -1), (-1, -2), (0, -2), (1, -2)],
          [(0, -2), (0, -1), (2, -1), (-1, 0), (0, 0), (1, 0), (2, 0), (0, 1), (2, 1), (0, 2)],
          [(0, -2), (0, -1), (-2, -1), (1, 0), (0, 0), (-1, 0), (-2, 0), (0, 1), (-2, 1), (0, 2)],
        ], dtype='int32')
```

```python
        ], dtype='int32')


    def _get_obs(self):
        return self._board

    def _get_info(self):
        return { 'prob': 1.0,
                 'action_mask': np.array(
                   (self._board != 0) == False, dtype='int32').reshape((self.size,self.size))}

    def reset(self, seed=None, options=None):
        # We need the following line to seed self.np_random
        super().reset(seed=seed)

        # initialize board (observation)
        self._board = np.zeros([self.size, self.size, 1], dtype=np.uint8)

        while True:
            self._hidden_airplane = np.ones([self.size, self.size, 1], dtype=np.uint8) * 255

            # the direction of airplain from 0 to 3
            direc = self.np_random.integers(0, 4)

            # Choose the airplain's center uniformly at random
            center = self.np_random.integers(1, self.size-1, size=(2,))

            out_of_board = False
            for rel_x, rel_y in self._relative_pos[direc]:
                if center[0] + rel_x < 0 or center[0] + rel_x >= self.size:
                    out_of_board = True
                    break
```

```python
    def reset(self, seed=None, options=None):
        # We need the following line to seed self.np_random
        super().reset(seed=seed)

        # initialize board (observation)
        self._board = np.zeros([self.size, self.size, 1], dtype=np.uint8)

        while True:
            self._hidden_airplane = np.ones([self.size, self.size, 1], dtype=np.uint8) * 255

            # the direction of airplain from 0 to 3
            direc = self.np_random.integers(0, 4)

            # Choose the airplain's center uniformly at random
            center = self.np_random.integers(1, self.size-1, size=(2,))

            out_of_board = False
            for rel_x, rel_y in self._relative_pos[direc]:
                if center[0] + rel_x < 0 or center[0] + rel_x >= self.size:
                    out_of_board = True
                    break

                if center[1] + rel_y < 0 or center[1] + rel_y >= self.size:
                out_of_board = True
                    break

                self._hidden_airplane[center[0] + rel_x, center[1] + rel_y, 0] = 1

                if not out_of_board:
                    break

        return self._get_obs(), self._get_info()
```

```
                    break

        return self._get_obs(), self._get_info()

    def step(self, action):
        assert action[0] >= 0 and action[0] < self.size
        assert action[1] >= 0 and action[1] < self.size

        if self._board[action[0], action[1], 0] == 0:
            # if the cell on the airplane,
            if self._hidden_airplane[action[0], action[1], 0] == 1:
                self._board[action[0], action[1], 0] = 1
                reward = 1
            # missed
            else:
                self._board[action[0], action[1], 0] = 2
                reward = -1

        # should not fall on here, but..
        else:
            reward = -1

        # An episode is done iff all ten cells of airplain hit
        hits = np.sum(self._board == self._hidden_airplane)
        terminated = True if hits == 10 else False

        observation = self._get_obs()
        info = self._get_info()

        # observation, reward, if terminated, if truncated, info
        # truncated: true if episode truncates due to a time limit
        return observation, reward, terminated, False, info
```

```python
        # should not fall on here, but..
        else:
            reward = -1

        # An episode is done iff all ten cells of airplain hit
        hits = np.sum(self._board == self._hidden_airplane)
        terminated = True if hits == 10 else False

        observation = self._get_obs()
        info = self._get_info()

        # observation, reward, if terminated, if truncated, info
        # truncated: true if episode truncates due to a time limit
        return observation, reward, terminated, False, info

    def render(self):
        if self.render_mode == "text":
            return self._render_board()

    def _render_board(self):
        str = ''
        chars = [' ', 'H', 'M']
        for row in range(self.size):
            for col in range(self.size):
                str += chars[self._board[row, col, 0]]
                str += ' | '
            for col in range(self.size):
                str += 'H' if self._hidden_airplane[row, col, 0] == 1 else ' '
                str += "\n"

        print(str)
```

Step 3: Misc

- gym_examples/__init__.py

```python
from gym.envs.registration import register

register(
    id="gym_examples/ShootingAirplane-v0",
    entry_point="gym_examples.envs:ShootingAirplaneEnv",
)
```

Step 3: Misc

- gym_examples/envs/__init__.py

```python
from gym_examples.envs.shooting_airplane import ShootingAirplaneEnv
```

연습: Using Gym Environment

- env.reset(), env.step()을 사용하여 Jupyter Notebook에서 비행기 격추게임을 진행해보고 스크린샷을 제출해주세요~

Development of Reinforcement Learning

1. Markov decision process
2. Monte-Carlo RL
3. Temporal difference RL
4. Deep reinforcement learning (deep Q-learning)
5. (optional) DQN + Monte-Carlo tree search

Tabular-based vs. Machine Learning-based

Recall that we updated Q values as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

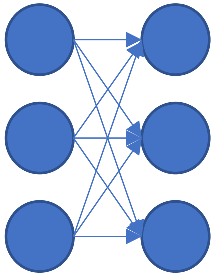
S_t, A_t

State, Action	Value
(0, 0)	
(0, 1)	
…	
(499, 5)	

$Q(S_t, A_t)$

<Table-based (e.g., the taxi problem)>

S_t, A_t



$Q(S_t, A_t; w)$

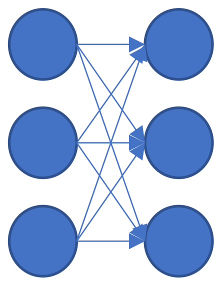
<Machine Learning-based>

Tabular-based vs. Machine Learning-based

25 taxi positions * 5 possible locations of the passenger * 4 destination locations

One-hot encoding for each factor
& concatenation

25 dim	5 dim	4 dim



$Q(S_t, A_t; w)$

<Multilayer perceptron>

Value Function Approximation

- Recall: TD 의 타겟과 업데이트 식

$$V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{Target}} - \overbrace{V(S_t)}^{\text{Estimation}}]$$

- 가치함수 (value function) $v(s; w)$: 모델 파라미터 w일 때 상태 s에 대한 가치 (value)를 계산
- Objective function for the value function approximation

$$J(w) = E_{\pi}\left[\left(v_{\pi}(s) - v(s; w)\right)^2\right]$$

TD(0) Value Function Learning with Approximation

- Training data for supervised learning with an episode

$$\langle s, v_\pi(s)\rangle = \langle S_1, R_2 + \gamma v(S_2; w)\rangle$$
$$\langle S_2, R_3 + \gamma v(S_3; w)\rangle$$
$$\vdots$$
$$\langle S_{T-1}, R_T\rangle$$

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E_\pi\left[\left(v_\pi(s) - v(s; w)\right)^2\right]}{\partial w}$$
$$= 2E_\pi\left[\left(v_\pi(s) - v(s; w)\right)\nabla_w v(s; w)\right]$$

- Stochastic gradient descent (SDG)

$$w \leftarrow w - \alpha[v_\pi(S_t) - v(S_t; w)]\nabla v(S_t; w)$$

Recall TD(0) For Policy Evaluating

For every $s_i \in S$, $\pi(s_i)$ is given (recall that the purpose of TD(0) is to evaluate a given policy)

- Repeat
  - Sample S_0 randomly and choose an action A_0 from S_0 w.r.t Q with π
  - For each t from 0 to T-1
    - Take the action A_t with π and observe R_{t+1} and S_{t+1}
    - $V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{Target}} - \underbrace{V(S_t)}_{\text{Estimation}}]$

V(s) 대신에 w를 업데이트

$$w_{t+1} \leftarrow w_t + \alpha[\underbrace{R_{t+1} + \gamma v(S_{t+1}; w)}_{\text{Target}} - \overbrace{v(S_t; w)}^{\text{Estimation}}]\nabla v(S_t; w)$$

TD(0) With SGD

For every $s_i \in S$, $\pi(s_i)$ is given (recall that the purpose of TD(0) is to evaluate a given policy)

- Repeat
  - Sample S_0 randomly and choose an action A_0 from S_0 with π
  - For each t from 0 to T-1
    - Take the action A_t with π and observe R_{t+1} and S_{t+1}
    - $w \leftarrow w + \alpha[R_{t+1} + \gamma v(S_{t+1}; w) - v(S_t; w)]\nabla v(S_t; w)$

Recall SARSA: On-Policy Control

- Repeat
  - Sample S_0 randomly and choose an action A_0 from S_0 with π
  - For each t from 0 to T-1
    - With the action A_t, observe R_{t+1} and S_{t+1} w.r.t Q with a ϵ-greedy policy
    - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$

SARSA with SGD

- Repeat
  - Sample S_0 randomly and choose an action A_0 from S_0 w.r.t Q with a ϵ-greedy policy
  - For each t from 0 to T-1
    - With the action A_t, observe R_{t+1} and S_{t+1} w.r.t Q with a ϵ-greedy policy
    - $w \leftarrow w + \alpha[R_{t+1} + \gamma q(S_{t+1}, A_{t+1}; w) - q(S_t, A_t; w)]\nabla q(S_t, A_t; w)$

Recall Q-Learning

For all $s_i \in S$

$$\pi(s_i) = \arg\max_{a \in A} q(s_i, a; w)$$

- Repeat
  - Sample S_0 randomly and choose an action A_0 from S_0 w.r.t Q with an ϵ-soft policy
  - For each t from 0 to T-1
    - Take the action A_t and observe R_{t+1} and S_{t+1} w.r.t Q with an ϵ-soft policy
    - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[\boxed{R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a)} - \boxed{Q(S_t, A_t)}\right]$

Target

Estimation

Q-Learning with Approximation

- Training data for supervised learning with an episode

$$\langle S_1, a_1, R_2 + \gamma q(S_2, a_2; w)\rangle$$
$$\langle S_2, a_2, R_3 + \gamma q(S_3, a_3; w)\rangle$$
$$\vdots$$
$$\langle S_{T-1}, R_T\rangle$$

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E_\pi\left[\left(q_\pi(s,a) - q(s,a;w)\right)^2\right]}{\partial w}$$
$$= 2E_\pi\left[\left(q_\pi(s,a) - q(s,a;w)\right)\nabla_w q(s,a;w)\right]$$

- Stochastic gradient descent (SDG)

$$w_{t+1} \leftarrow w_t + \alpha[q_w(S_t, a) - q(S_t, a; w)]\nabla q(S_t, a; w)$$

Q-Learning With Policy Gradient

- Given a discount γ and a policy π as inputs
  - Randomly initialize $q(s_i, a_k), \forall s_i \in S, \forall a_k \in A$
- Repeat
  - Sample S_0 randomly and choose an action A_0 from S_0 w.r.t Q with a ϵ-soft policy
  - For each t from 0 to T-1
    - Take the action A_t and observe R_{t+1} and S_{t+1} from the environment
    - If S_{t+1} is terminal (i.e., t=T-1), $G_t = R_{t+1}$
    - Otherwise, $G_t = R_{t+1} + \gamma \max_{a \in A} q(S_{t+1}, a; w)$
    - $w \leftarrow w + \alpha[G_t - q(S_t, A_t; w)]\nabla q(S_t, A_t; w)$

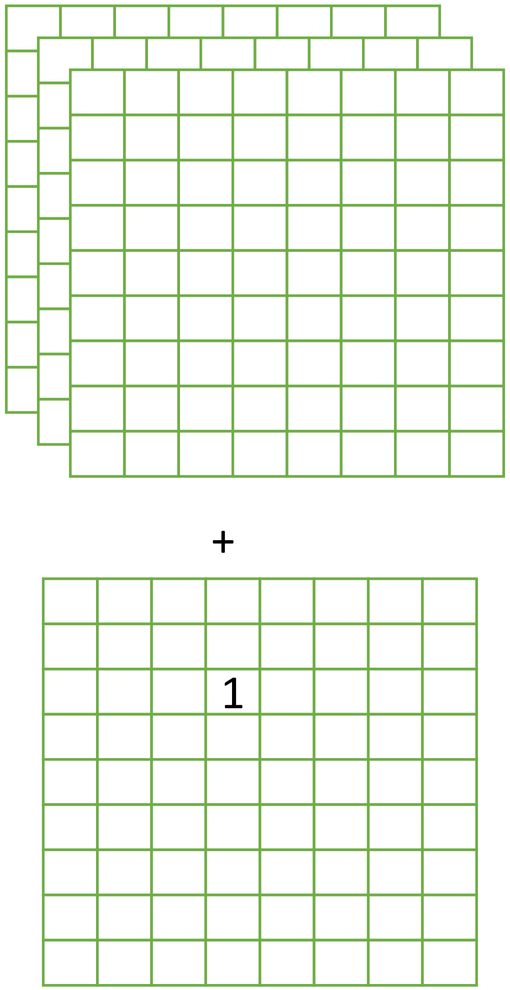
Q-Learning-based Policy Gradient For Shooting Airplane Game

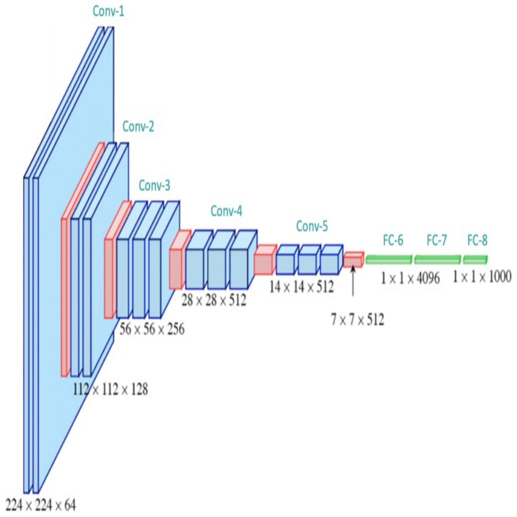
Training To Play The Shooting Game With Deep Q-Network

- Reference
  - https://keras.io/examples/rl/deep_q_network_breakout/

Architecture of Q-Network: Candidate 1.

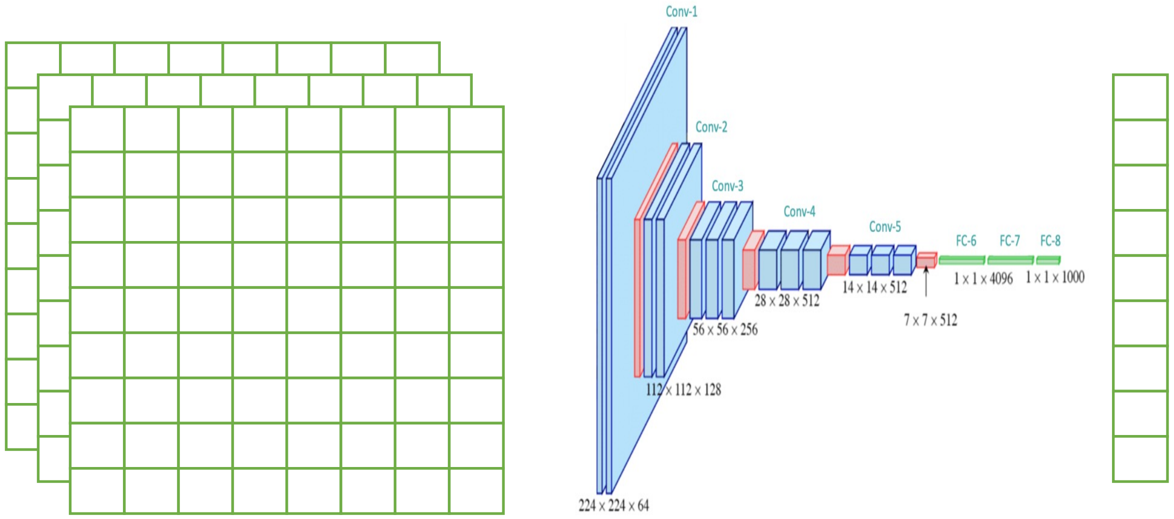
- Input
  - A tensor of (8, 8, 4) each channel of which represents unseen, hit and miss for each 64 cell, and the action
- Model
  - 3 convolutional layers + 2 fully connected layers
- Output
  - The action value





Architecture of Q-Network: Candidate 2.

- Input
  - A matrix of (8, 8, 3) each channel of which represents unseen, hit and miss for each 64 cell
- Model
  - 3 convolutional layers + 2 fully connected layers
- Output
  - An array of action value of 64 actions



Constants & Settings

```
import numpy as np
import gym
import torch
import torch.nn as nn
import torchvision.transforms as T

# Configuration paramaters for the whole setup
seed = 42
gamma = 0.99  # Discount factor for past rewards
epsilon = 1.0  # Epsilon greedy parameter
epsilon_min = 0.1  # Minimum epsilon greedy parameter
epsilon_max = 1.0  # Maximum epsilon greedy parameter
epsilon_interval = epsilon_max - epsilon_min # Rate at which to reduce chance
                                             # of random action being taken
batch_size = 16  # Size of batch taken from replay buffer
max_steps_per_episode = 60
max_episodes = 10000
```

Replay Buffers & Misc. Variables

```
# Experience replay buffers
action_history = []
action_mask_history = []
state_history = []
state_next_history = []
rewards_history = []
done_history = []
episode_reward_history = []
running_reward = 0
episode_count = 0
frame_count = 0

# Number of frames to take random action and observe output
epsilon_random_frames = 50000
# Number of frames for exploration
epsilon_greedy_frames = 200000.0
# Maximum replay length
# Note: The Deepmind paper suggests 1000000 however this causes memory issues
max_memory_length = 500000
# Train the model after 4 actions
update_after_actions = 4
# How often to update the target network
update_target_network = 10000
```

Open Gym Environment

```
env = gym.make('gym_examples:gym_examples/ShootingAirplane-v0', render_mode="text")
```

Q-Network

```
import torch
import torch.nn as nn

num_actions = 64

class QModel(nn.Module):
    def __init__(self, num_actions):
        super(QModel, self).__init__()
        self.dropout = nn.Dropout(p=0.3)
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding='same')
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding='same')
        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(1152, 512)
        self.fc2 = nn.Linear(512, num_actions)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = self.dropout(x)
        x = nn.functional.relu(self.conv3(x))
```

```python
import torch
import torch.nn as nn

num_actions = 64

class QModel(nn.Module):
    def __init__(self, num_actions):
        super(QModel, self).__init__()
        self.dropout = nn.Dropout(p=0.3)
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding='same')
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding='same')
        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(1152, 512)
        self.fc2 = nn.Linear(512, num_actions)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = self.dropout(x)
        x = nn.functional.relu(self.conv3(x))
        x = self.flatten(x)
        x = nn.functional.relu(self.fc1(x))
        x = self.dropout(x)
        action = self.fc2(x)
        return action
```

Building Networks & Loss Function

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# The first model makes the predictions for Q-values which are used to
# make a action.
model = QModel(num_actions)
model.to(device)

# Build a target model for the prediction of future rewards.
# The weights of a target model get updated every 10000 steps thus when the
# loss between the Q-values is calculated the target Q-value is stable.
model_target = QModel(num_actions)
model_target.to(device)

loss_function = nn.SmoothL1Loss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.00025)
```

State Preprocessing

```python
# Function to preprocess the state
def preprocess_state(env_observ):
    st = torch.from_numpy(env_observ).squeeze()
    st = st.to(torch.int64)
    st = torch.nn.functional.one_hot(st,num_classes=3)
    st = st.permute(2, 0, 1)
    return st.to(torch.float32)
```

Epsilon-Soft Greedy Policy

For all $s_i \in S$

$$\pi(a, s_i) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s_i)|} & if\ a = \arg\max_{a \in A} q(s_i, a) \\ \frac{\epsilon}{|A(s_i)|} & \text{otherwise} \end{cases}$$

```
# Function to select an action
# model: the torch model to compuate action-state value (i.e., q-value)
# state: a torch tensor (3 x 8 x 8) of float32, which is output by preprocess_state
# mask: a 64-size array (np.array)
def get_greedy_epsilon(model, state, mask):
    global epsilon

    #if frame_count < epsilon_random_frames or np.random.rand(1)[0] < epsilon:
    if np.random.rand(1)[0] < epsilon:
        action = np.random.choice([ i for i in range(num_actions) if mask[i] == 1 ])
    else:
        with torch.no_grad():
            # add a batch axis
            state_tensor = state.unsqueeze(0)
            # compute the q-values
            q_values = model(state_tensor)
            # select the q-values of valid actions
            action = torch.argmax(
                q_values.squeeze() + torch.from_numpy(mask) * 100., # trick to select a valid action
                dim=0)

    # decay epsilon
```

```python
# Function to select an action
# model: the torch model to compuate action-state value (i.e., q-value)
# state: a torch tensor (3 x 8 x 8) of float32, which is output by preprocess_state
# mask: a 64-size array (np.array)
def get_greedy_epsilon(model, state, mask):
    global epsilon

    #if frame_count < epsilon_random_frames or np.random.rand(1)[0] < epsilon:
    if np.random.rand(1)[0] < epsilon:
        action = np.random.choice([ i for i in range(num_actions) if mask[i] == 1 ])
    else:
        with torch.no_grad():
            # add a batch axis
            state_tensor = state.unsqueeze(0)
            # compute the q-values
            q_values = model(state_tensor)
            # select the q-values of valid actions
            action = torch.argmax(
                q_values.squeeze() + torch.from_numpy(mask) * 100., # trick to select a valid action
                dim=0)

    # decay epsilon
    epsilon -= epsilon_interval / epsilon_greedy_frames
    epsilon = max(epsilon, epsilon_min)

    return action
```

Greedy Policy

Will be used for evaluation, not in the training phase

```
def get_greedy_action(model, state, mask):
    global epsilon

    with torch.no_grad():
        state_tensor = state.unsqueeze(0) # batch dimension
        q_values = model(state_tensor)

        action = torch.argmax(
                q_values.squeeze() + torch.from_numpy(mask) * 100., # trick to select a valid action
                dim=0)

    return action
```

Sampling A Batch From Replay Buffers

```
# sample a batch of _batch_size from replay buffers
# return numpy.ndarrays
def sample_batch(_batch_size):
    # Get indices of samples for replay buffers
    indices = np.random.choice(range(len(done_history)), size=_batch_size, replace=False)

    state_sample = np.array([state_history[i].squeeze(0).numpy() for i in indices])
    state_next_sample = np.array([state_next_history[i].squeeze(0).numpy() for i in indices])
    rewards_sample = np.array([rewards_history[i] for i in indices], dtype=np.float32)
    action_sample = np.array([action_history[i] for i in indices])

    # action mask is the mask for the valid actions at the '''next''' state
    action_mask_sample = np.array([action_mask_history[i] for i in indices])
    done_sample = np.array([float(done_history[i]) for i in indices])

    return state_sample, state_next_sample, rewards_sample, action_sample, \
        action_mask_sample, done_sample
```

Update Networks

```
# Function to update the Q-network
def update_network():
    # sample a batch of ...
    state_sample, state_next_sample, rewards_sample, \
        action_sample, action_mask_sample, done_sample = \
         sample_batch(batch_size)

    # Convert numpy arrays to PyTorch tensors
    state_sample = torch.tensor(state_sample, dtype=torch.float32).to(device)
    state_next_sample = torch.tensor(state_next_sample, dtype=torch.float32).to(device)
    action_sample = torch.tensor(action_sample, dtype=torch.int64).to(device)
    action_mask_sample = torch.tensor(action_mask_sample, dtype=torch.int64).to(device)
    rewards_sample = torch.tensor(rewards_sample, dtype=torch.float32).to(device)
    done_sample = torch.tensor(done_sample, dtype=torch.float32).to(device)

    # Compute the target Q-values for the states
    with torch.no_grad():
        future_rewards = model_target(state_next_sample)
        #future_rewards = future_rewards.cpu()
```

```python
# Compute the target Q-values for the states
with torch.no_grad():
    future_rewards = model_target(state_next_sample)
    #future_rewards = future_rewards.cpu()

    # compute the q-value for the next state and the action maximizing the q-value
    # note: the action should be valid (i.e., mask is set to 1)
    max_q_values = torch.max(
        future_rewards + action_mask_sample * 100., # trick to select a valid action
        dim=1).values.detach() - 100.

    # compute the target q-value
    # if the step was final, max_q_values should not be added
    target_q_values = rewards_sample + gamma * max_q_values * (1. - done_sample)

# It's forward propagation! Compute the Q-values for the taken actions
q_values = model(state_sample)
#q_values = q_values.cpu()
q_values_action = q_values.gather(dim=1, index=action_sample.unsqueeze(1)).squeeze(1)

# Compute the loss
loss = loss_function(q_values_action, target_q_values)

# Perform the optimization step
optimizer.zero_grad()
```

```
        # compute the q-value for the next state and the action maximizing the q-value
        # note: the action should be valid (i.e., mask is set to 1)
        max_q_values = torch.max(
            future_rewards + action_mask_sample * 100., # trick to select a valid action
            dim=1).values.detach() - 100.

        # compute the target q-value
        # if the step was final, max_q_values should not be added
        target_q_values = rewards_sample + gamma * max_q_values * (1. - done_sample)

    # It's forward propagation! Compute the Q-values for the taken actions
    q_values = model(state_sample)
    #q_values = q_values.cpu()
    q_values_action = q_values.gather(dim=1, index=action_sample.unsqueeze(1)).squeeze(1)

    # Compute the loss
    loss = loss_function(q_values_action, target_q_values)

    # Perform the optimization step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Training

```
for _ in range(max_episodes):
    state, info = env.reset()
    state = preprocess_state(state)
    action_mask = info['action_mask'].reshape((-1,))
    episode_reward = 0

    for timestep in range(1, max_steps_per_episode):
        frame_count += 1

        # Select an action
        #state_cuda = state.to(device)
        action = get_greedy_epsilon(model, state, action_mask)
        if action < 0:
            print(action_mask)

        # Take the selected action
        state_next, reward, done, _, info = env.step((action // 8, action % 8))
        state_next = preprocess_state(state_next)
        action_mask = info['action_mask'].reshape((-1,))

        episode_reward += reward
```

```python
episode_reward += reward

# Store the transition in the replay buffer
action_history.append(action)
action_mask_history.append(action_mask)
state_history.append(state)
state_next_history.append(state_next)
rewards_history.append(reward)
done_history.append(done)

state = state_next

# Update every fourth frame and once batch size is over 32
if frame_count % update_after_actions == 0 and len(done_history) > batch_size:
    update_network()

if frame_count % update_target_network == 0:
    model_target.load_state_dict(model.state_dict())

# Limit the state and reward history
if len(rewards_history) > max_memory_length:
    del rewards_history[:1]
    del state_history[:1]
    del state_next_history[:1]
    del action_history[:1]
    del action_mask_history[:1]
    del done_history[:1]
```

```python
            del action_mask_history[:1]
            del done_history[:1]

        if done:
            break

    episode_count += 1
    episode_reward_history.append(episode_reward)

    # Update running reward to check condition for solving
    if len(episode_reward_history) > 100:
        del episode_reward_history[:1]
    running_reward = np.mean(episode_reward_history)

    if episode_count % 10 == 0:
        print(f"Episode: {episode_count}, Frame count: {frame_count},",
         "Running reward: {running_reward}")

    if episode_count % 5000 == 0:
        torch.save(model, 'model.{}'.format(episode_count))
    #if running_reward > 20:
    #    print(f"Solved at episode {episode_count}!")
    #    break


torch.save(model, 'model.final')
```

Evaluation: Animating the Agent’s Play

```
import time, sys
from IPython.display import clear_output

board, info = env.reset()
state = preprocess_state(board)
action_mask = info['action_mask'].reshape((-1,))
done = False
env.render()

while not done:
    action = get_greedy_action(model, state, action_mask)
    print("action: ({}, {})".format(action // 8, action % 8))
    sys.stdout.flush()

    time.sleep(1.0)
    clear_output(wait=False)
    board, reward, done, _, info = env.step((action // 8, action % 8))
    state = preprocess_state(board)
    action_mask = info['action_mask'].reshape((-1,))
    env.render()
```

Let’s Watch AI’s Play

```
|
|
|      H
|    H H
|    HHHH
|    H H
|      H
|

action: (3, 4)
```

연습

- 학습한 DQN으로 Shooting airplane 게임을 플레이하여 마지막 상태 스크린 샷을 제출하세요~

Improving Deep Q-Learning

(Vanilla) Deep Q-Network Learning

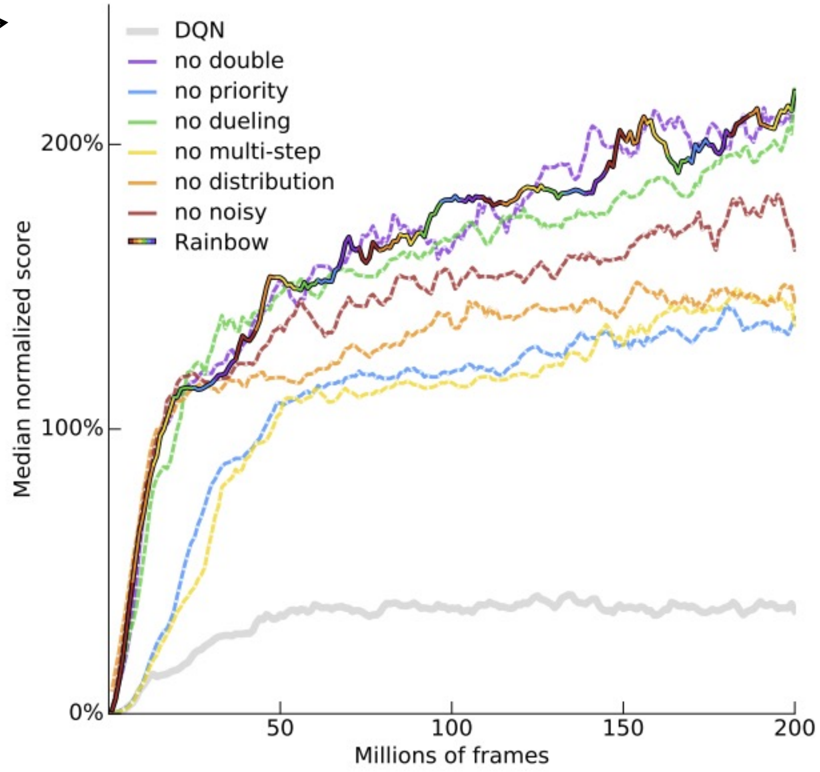
- Minimize MSE between estimation by Q-network and target

$$E_{S_t,A_t,R_{t+1},S_{t+1}}\left[\left(\underbrace{\boxed{R_{t+1}+\gamma\max_{a\in A}Q(S_{t+1},a;w')}}_{\text{Target}}-\underbrace{\boxed{Q(S_t,A_t;w)}}_{\text{Estimation with network}}\right)^2\right]$$

  - where
    - Target is computed w.r.t. old and fixed parameters w'
    - To deal with non-stationarity, target parameters w' are held fixed
- Using stochastic gradient descent

Extensions to DQN

The integrated agent (rainbow-colored) to DQN (gray) and to six different ablations (dashed lines)



[M. Hessel et al., Rainbow: Combining Improvements in Deep Reinforcement Learning, In AAAI, 2018.]

Double Q-Learning

- Use two separate networks
  - Current Q-network $w_{current}$ is used to select actions
  - Older Q-network w_{old} is used to evaluate actions

Evaluate the selected action with w_{old}

$$w_{current} \leftarrow w_{current} + \alpha \left(R_{t+1} + \gamma \boxed{Q \left(S_{t+1}, \boxed{\arg\max_{a \in A(S_t)} Q(S_t, a; w_{current})}; w_{old} \right)} - Q(S_t, a; w_{current}) \right)$$

Update $w_{current}$

Select action with $w_{current}$

[H. Hasselt et al., Deep Reinforcement Learning with Double Q-Learning, In AAAI, 2016.]

Prioritized Experience Replay

- Sample $\langle s_i, a_i, r_i, s_i' \rangle$ with priority
- The priority for the tuple i is proportional to DQN error, computed as

$$p_i^\alpha = \left| r_i + \gamma \max_a Q(s_i', a; w) - Q(s_i, a_i; w) \right|^\alpha$$

- That is, following the distribution

$$P(i) = \frac{p_i^\alpha}{\sum_{(s_k, a_k, r_k, s_k')} p_k^\alpha}$$

[T. Schaul et al., Prioritized Experience Replay, In ICLR, 2016.]

n-step Learning

- Recall n-step return from a given state S_t

$$G_{\mathrm{t}} \leftarrow R_{t+1} + \gamma R_{t+2} + \cdots \gamma^{n-1} R_{t+n}$$

- The n-step variant of DQN is minimizing the following loss

$$\left(G_t + \gamma^n \max_{a \in A(S_{t+n})} Q(S_{t+n}, a; w) - Q(S_t, A_t)\right)^2$$

Noisy Nets

- Instead of the standard linear function in a neural network

$$w \cdot x + b$$

- Use a noisy linear function

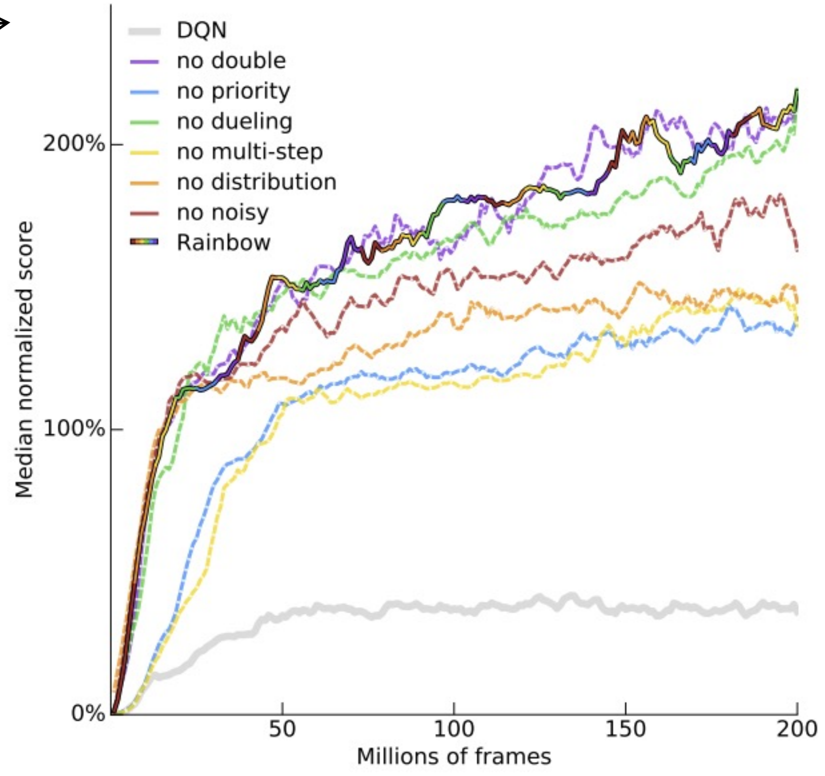
$$(\mu^w + \sigma^w \odot \epsilon^w) \cdot x + \mu^b + \sigma^b \odot \epsilon^b$$

  - where
    - $\mu^w, \mu^b, \sigma^w, \sigma^b$: learnable parameters
    - ϵ^w, ϵ^b: noise randome variables

[M. Fortunato et al., Noisy Networks for Exploration, In ICLR, 2018.]

Extensions to DQN

The integrated agent (rainbow-colored) to DQN (gray) and to six different ablations (dashed lines)



[M. Hessel et al., Rainbow: Combining Improvements in Deep Reinforcement Learning, In AAAI, 2018.]

연습

- 달착륙선의 착지 조정법을 학습
  - 환경 설치 및 사용법: box2d-lunarlander-gym.ipynb (카톡공유)
  - 힌트: Atari breakout 코드를 베이스로 수정
- 환경 매뉴얼
  - https://www.gymlibrary.dev/environments/box2d/lunar_lander/

Action Space	Discrete(4)
Observation Shape	(8,)
Observation High	[1.5 1.5 5. 5. 3.14 5. 1. 1.]
Observation Low	[-1.5 -1.5 -5. -5. -3.14 -5. -0. -0.]

- Action
  - There are four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine

팀과제

- Reversi (Othello) 게임의 Custom Gym Env를 완성하세요
    - 템플릿 gym_examples.zip에 포함되어 있음
- DQN을 구현하여 Reversi game agent 를 학습하세요
- 데모플레이를 보여주세요 (팀별 5분)

보조자료

- gym_examples/env/reversi_random_template.py
  - ReversiEnv의 템플릿 (힌트: ---fill here--- 부분을 완성)
  - reversi_random.py 로 파일명 변경 후 사용
- custom_gym_reversi.ipynb
  - ReversiEnv를 로딩하여 테스트하는 코드 포함
  - 주의: reversi_random.py를 업데이트한 후에는 Colab 세션을 다시 시작해야 gym.make를 통한 로딩이 적용
- reversi-pygame.py
  - PyGame을 이용한 GUI 기반 2인 대전 프로그램