8강. DQN

Human-level control through deep reinforcement learning

(Minh et al., 2015)

Contents

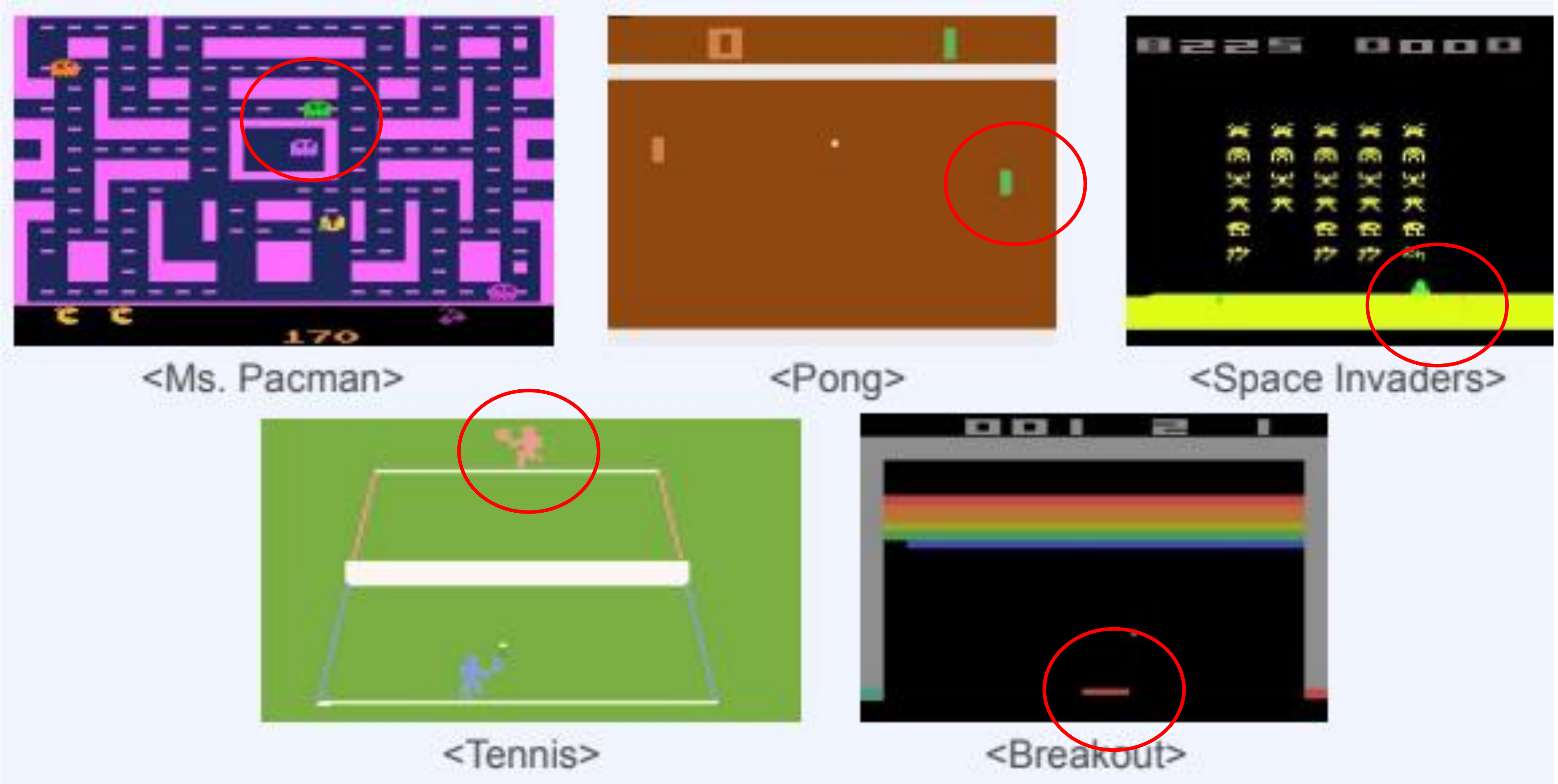
- DQN Overview
- Code Review
- Pong Game Exercise

DQN Overview

- Human-level control through deep RL(Minh et al., 2015)
- Solved complex env. and high dimensional input (image)
- 1^{st} success case using DNN (inspired by AlexNet)
- No hand-crafted feature
- In Atari 2600 game, the agent is better than an human expert

DQN: Environment

- Atari 2600 game

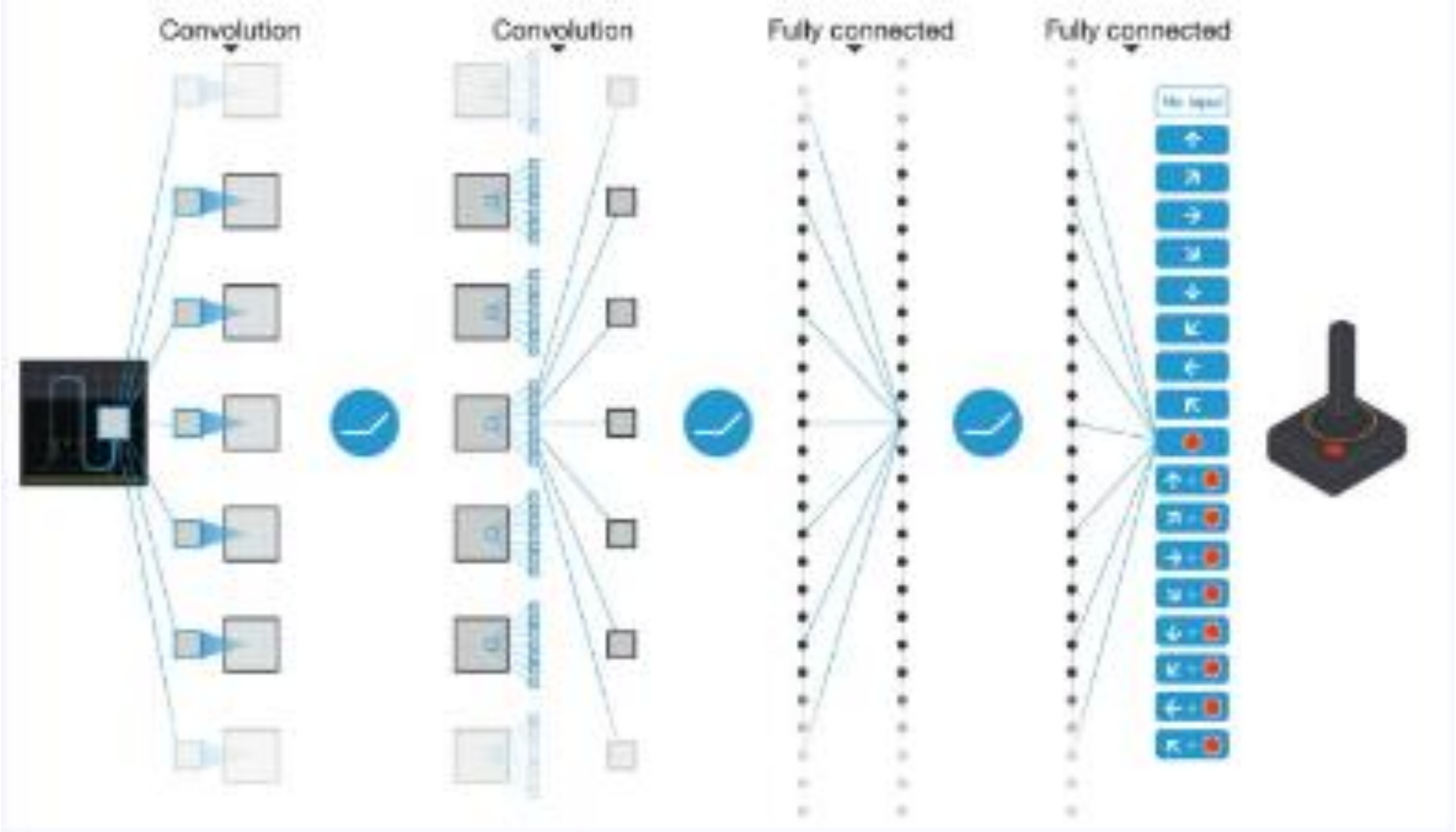


DQN: Overview

- DQN = Q-learning + Deep Neural Network
- Used CNN architecture for input images
- Prior to DQN, solutions using neural network has problems.
  - Learning is unstable or diverge
- DQN solved the problem with 2 methods

DQN: Neural Network

- Neural network that takes images as the input and discrete action value(Q function values) as the output



DQN: Solution to problems

- Time-correlated data in an episode
  - State changes a little, a state may trigger getting the future results
- Correlation between data makes learning harder (NOT i.i.d)
  - Gradients oscillate depending on an episode (optimization is hard)
  - Generalization for diverse states is very hard

- Solution:
  - Uniformly random sample from the transition data(s, a, r, s') buffer (Experience replay buffer)
  - Q-learning is off-policy. So, using old data from other distribution is okay

DQN: Solution to problems

- Small Q update can completely change a policy
  - This is especially true for argmax policy case
- The policy change cause a big change in data distribution
  - This means learning from a very dataset than the previous dataset
  - The learning become unstable

- Solution:
  - Although the policy change dramatically, data distribution sampled from the experience replay buffer do not change so dramatically

DQN: Solution to problems

- Note learning target, $r + \gamma \max_{a'} Q(s', a')$ has correlation with $Q(s', a')$ because they share parameters
- The target is changing while learning $Q(s, a)$ is not close to the target (Target oscillation problem)

- Solution:
  - Manage a separate neural network (a target network) to fix the target network parameter, and then update the target parameter every now and then

DQN: Loss function

$$L_i(\theta_i) = E_{\underline{(s,a,r,s')\sim U(D)}}[(r + \gamma \max_{a'} Q(s',a';\underline{\theta_i^-}) - Q(s,a;\theta_i))^2]$$

- θ_i^-: parameters to calculate target value at i-th iteration
  - θ_i: parameters to learn in i-th iteration
- U(D) : replay memory to store transitions
  - U(D) is a collection of data by policies, $\pi_0\, \pi_1 \ldots \pi_{\mathrm{I}}$
  - U(D) distribution follows the mean of data by policies, $\pi_0\, \pi_1 \ldots \pi_{\mathrm{I}}$
  - Note that the data distribution is different from the latest policy π_{i} , but q-learning is off-policy learning, so it's okay

DQN: State Preprocessing

- Original big image size problem:
  - 210x160 (0~127 RGB) image requires memory and computation

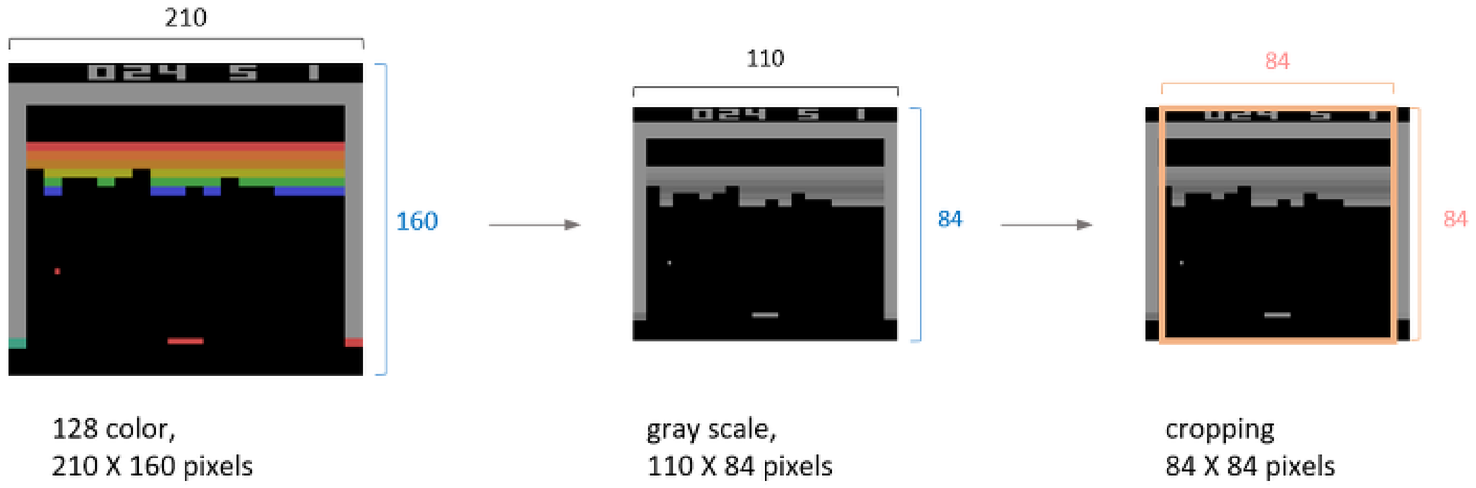
- 'Flickering' is a problem:
  - 2 states are the same, but there's pixel value fluctuation
  - To solve it, take max. pixel RGB value btw. t-1 pixel and t pixel:

    t-1:(57,34,72),

    t: (88,34,21) ➔ (88,34,72)

DQN: State Preprocessing

- Transform the RGB
  into Black and white,
  - Resizing into 84x84

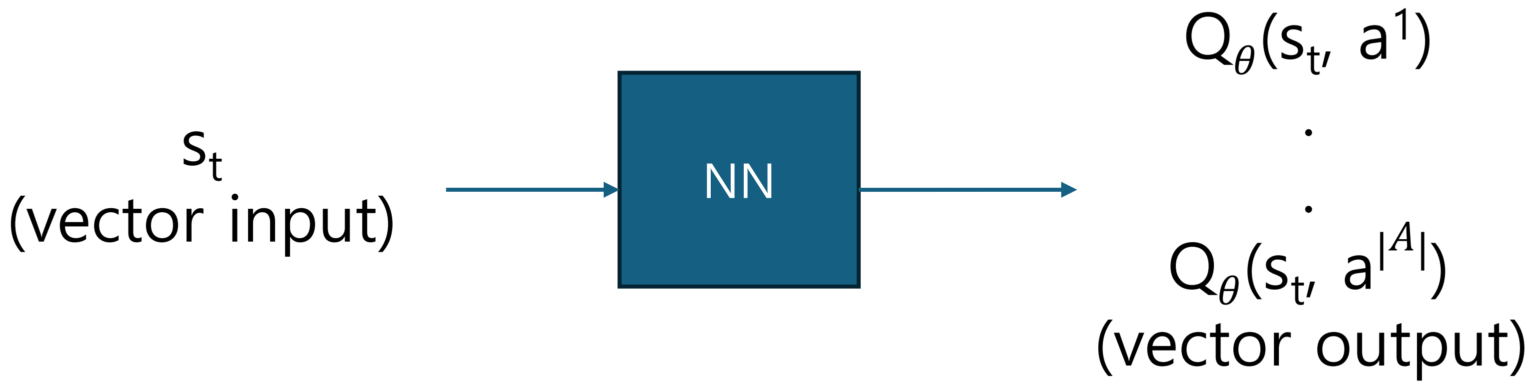


- Stack recent 4 frames
  - Note the agent could not predict the direction of moving object with 1 image.
  - The environment is not 1st order Markov property
  - Using 3 or 5 frame stack shows no different results

DQN: Q network

- If we allows action input, computation is high when argmax:

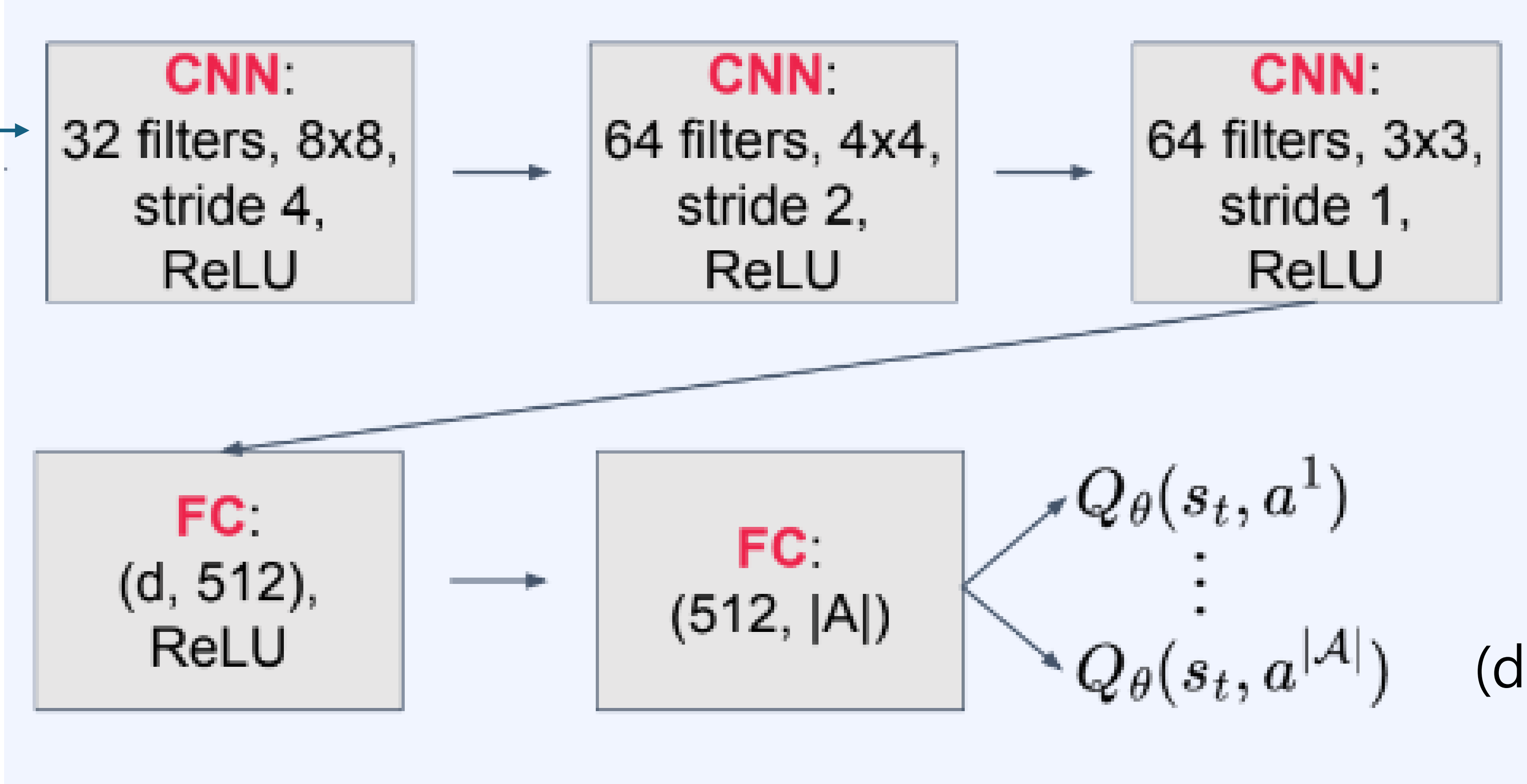
- Instead, output all the action-values of every actions once



DQN: Q network

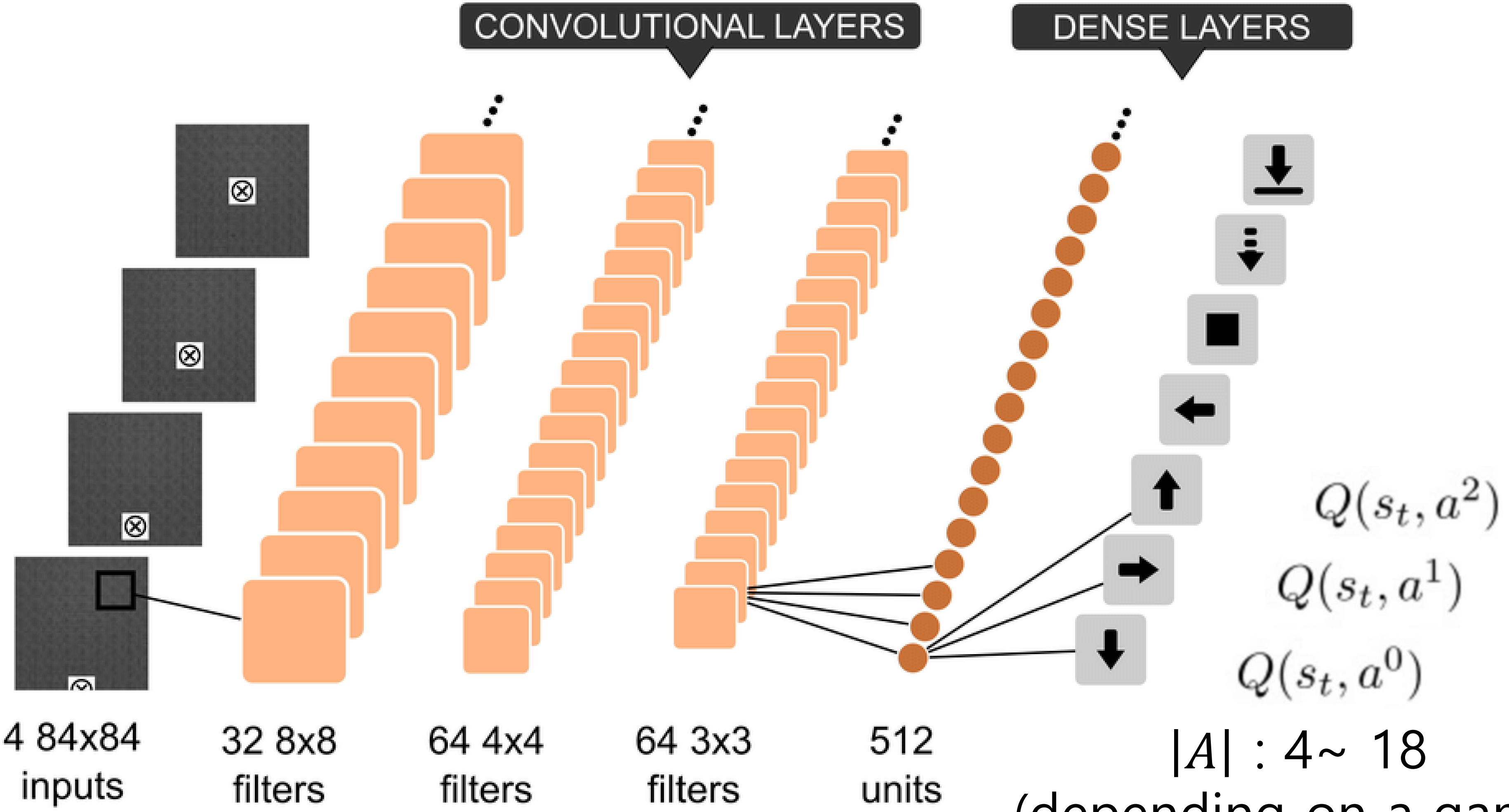
- Neural Network using CNN

s_t
(84,84,4)



$|A|$: 4~ 18
(depending on
a game)

DQN Q network



$|A|$: 4~ 18
(depending on a game)

DQN: Training Details

- Learning from 49 games of Atari 2000
- Reward shaping: +1 ~ -1 (Normalization)
- If there's life in game, life reduce is the end of episode
- Optimizer: RMSProp
- Batch size: 32
- ε-greedy: linear annealing from 1.0 - 0.1 (first 1M frames)
  , and then fixed to 0.1
- Total 50M frames: 38 days game play time
- Replay memory store: recent 1M transitions

DQN: Training Details

- Frame-skipping (action repeat):
  - Agent chooses an action every k frame (same actions within k)
  - Compute cost of game emulation is cheaper than network forward().
  - Reduces exploration complexity
- Hyper-parameter hand tunning:
  - Extracted for 5 games (Pong, Breakout, Seaquest, Sparce Invaders, Beam Rider), and used the same hyper-parameters for others
  - Grid search requires too much computation

DQN: Hyperparameters

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter *C* from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ε is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

DQN: Evaluation Details

- Agent performance is measured by the 30 repeat of 5 min. plays
- Epsilon is 0.05 to prevent overfitting in evaluation
- Baseline random agent takes an action every 6 frame (10 Hz)
  - Known fastest response time that a human player can take
  - Taking an action every frame didn't make a big difference
    (In Boxing, Breakout, Crazy Climber, Demon Attack, Krull
    , and Robotank, 5% higher performance)

DQN: Evaluation Details

- Human player played 60Hz frame rate w/o sound, pause, or reload.
- After 2 hours of practice, a human player performance measured by the 20 repeat of 5 min. plays

- To provide diverse starting points, evaluation was started after 30 times no-op
  - A robust policy does a good job from various starting points

DQN

x_t: 210x160 RGB image at t
s_t : a stacked 4 images

φ :
- Remove flickering
- Resizing 84x84 black & white
- Stack 4 frames

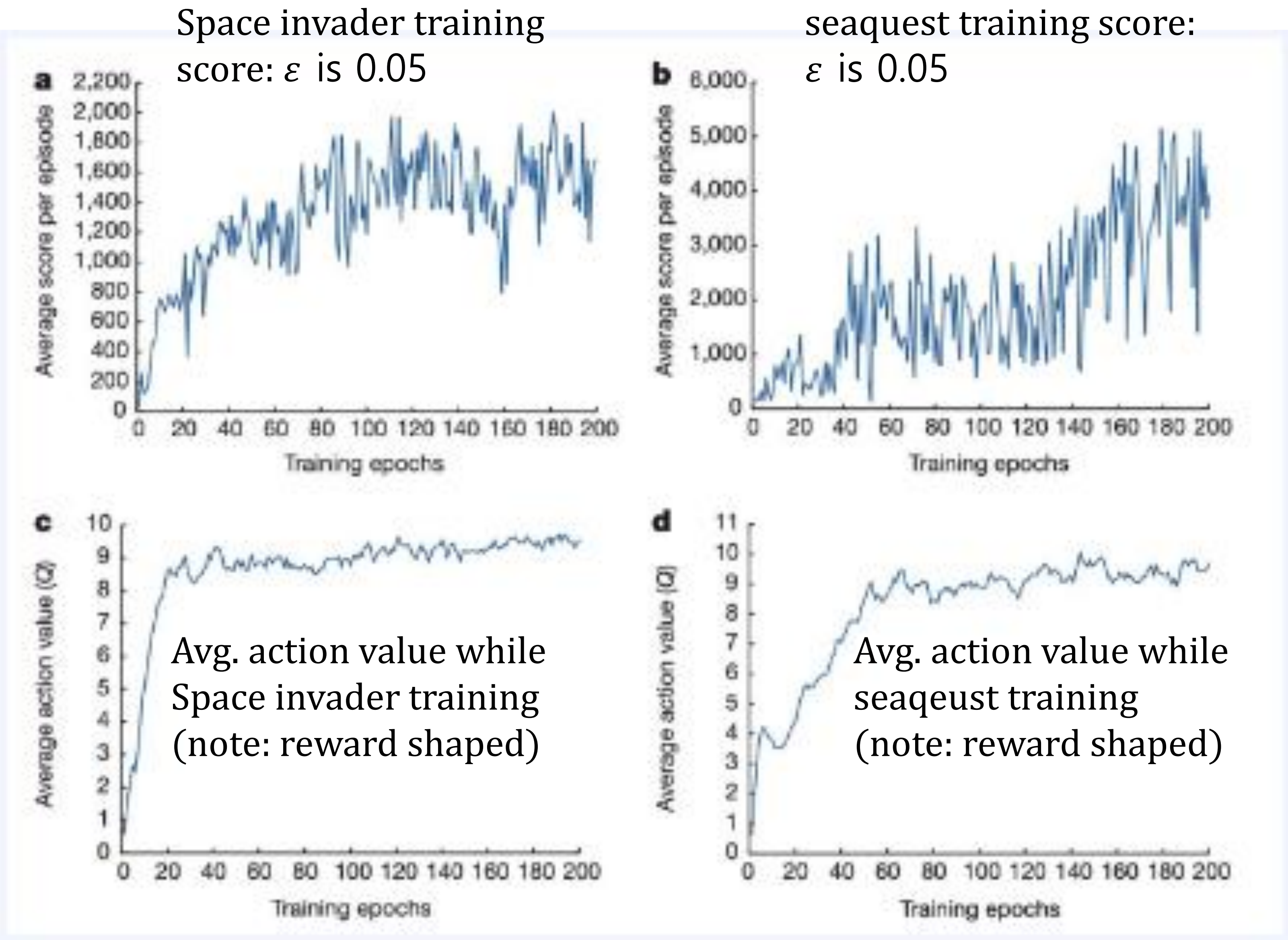
S_{t+1} : s_t contains 1~t states info.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

DQN: Results

- Gradient descent every 4 transition
- 32 batch size x 4 transition = 8 update / sample

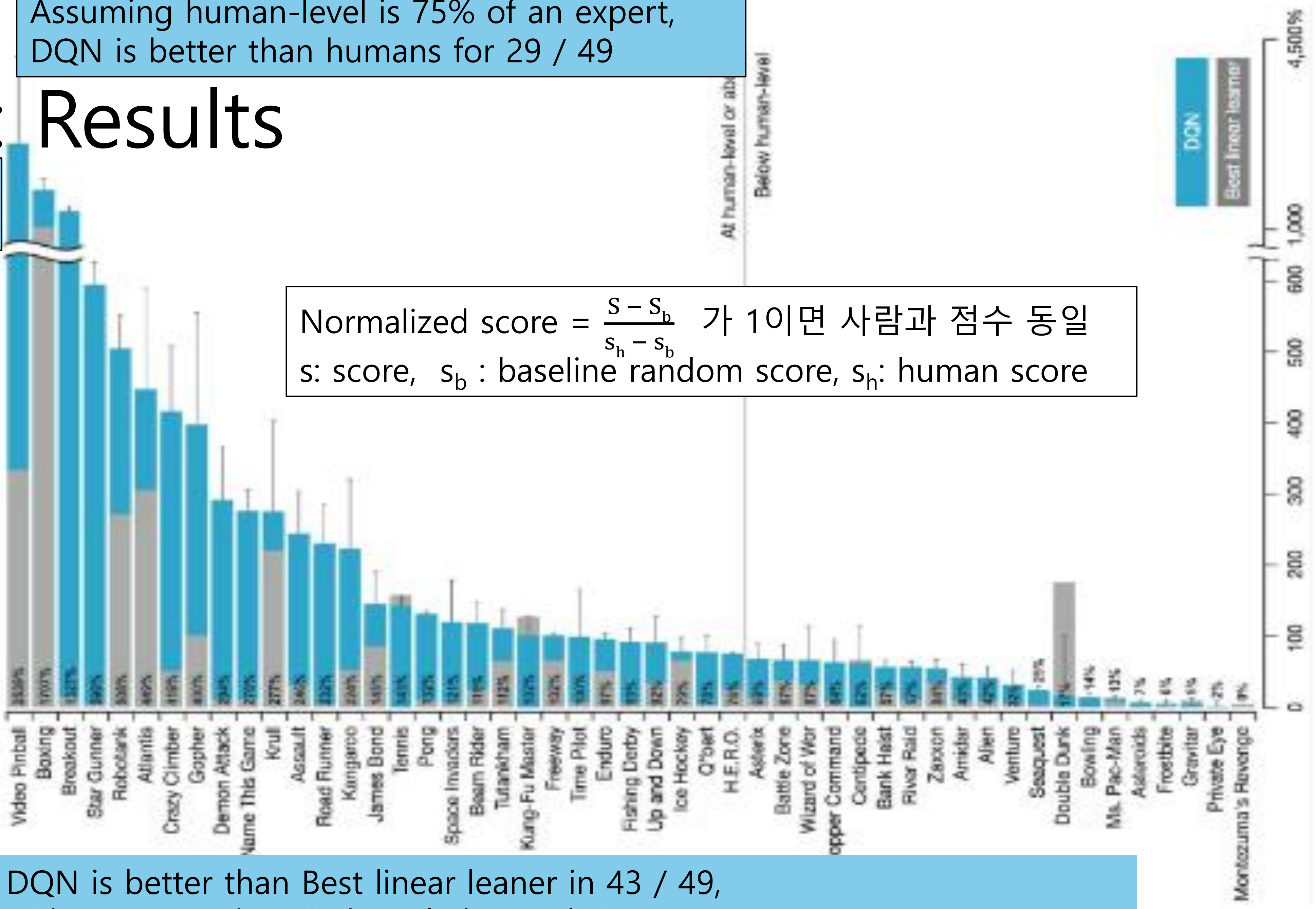
- Avg. action value is the shaped reward value (Not same scale as avg. return)



DQN: Results

Assuming human-level is 75% of an expert,
DQN is better than humans for 29 / 49

25.4 times better
than human



Normalized score = $\frac{s - s_b}{s_h - s_b}$ 가 1이면 사람과 점수 동일
s: score, s_b : baseline random score, s_h: human score

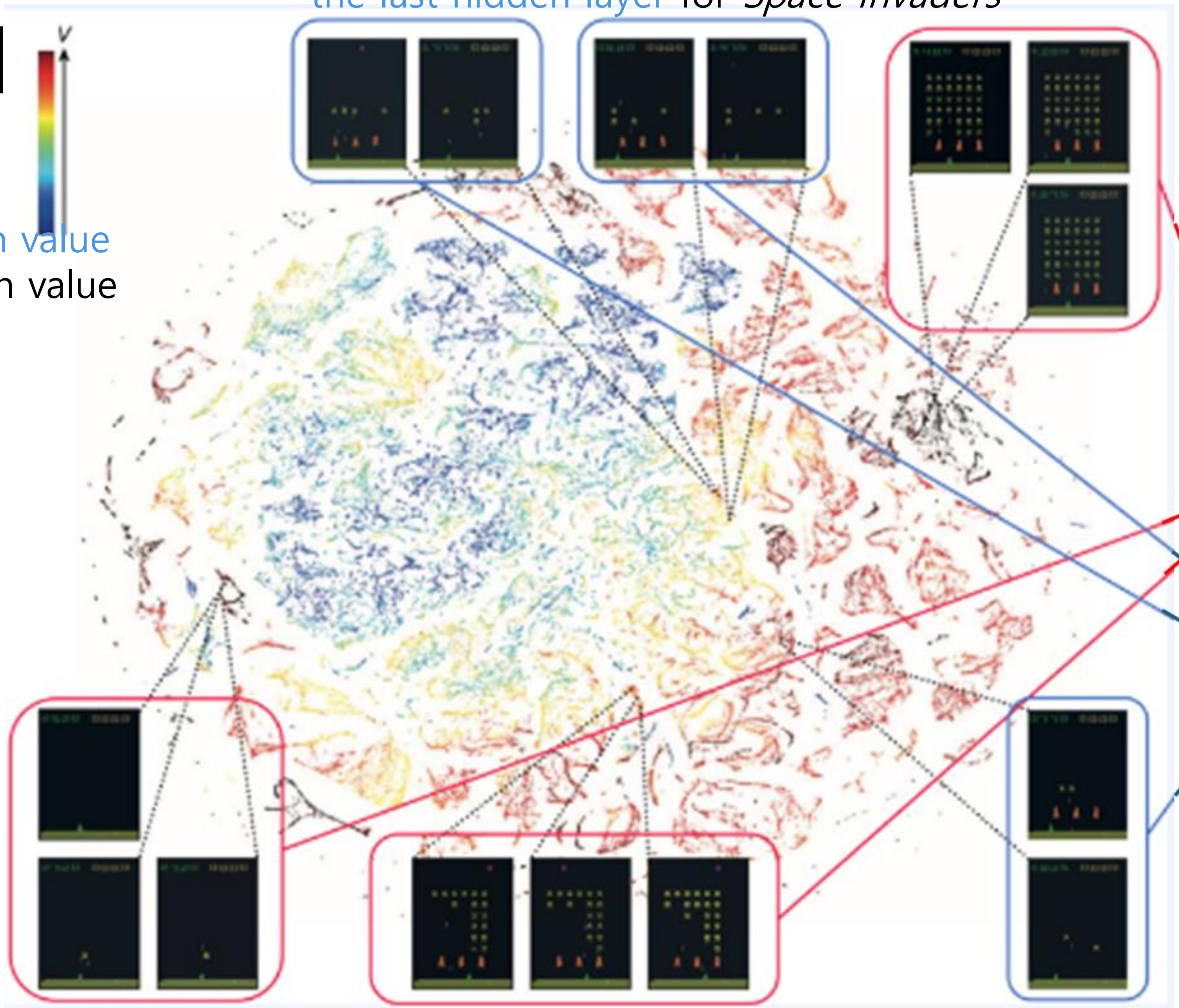
Previous SOTA
for Atari 2600 is
Best linear
learner

DQN is better than Best linear leaner in 43 / 49,
without game domain knowledge (only images)

T-SNE 2 dim. visualization of vector data of the last hidden layer for *Space Invaders*

DQN

Red : high action value
Black : low action value

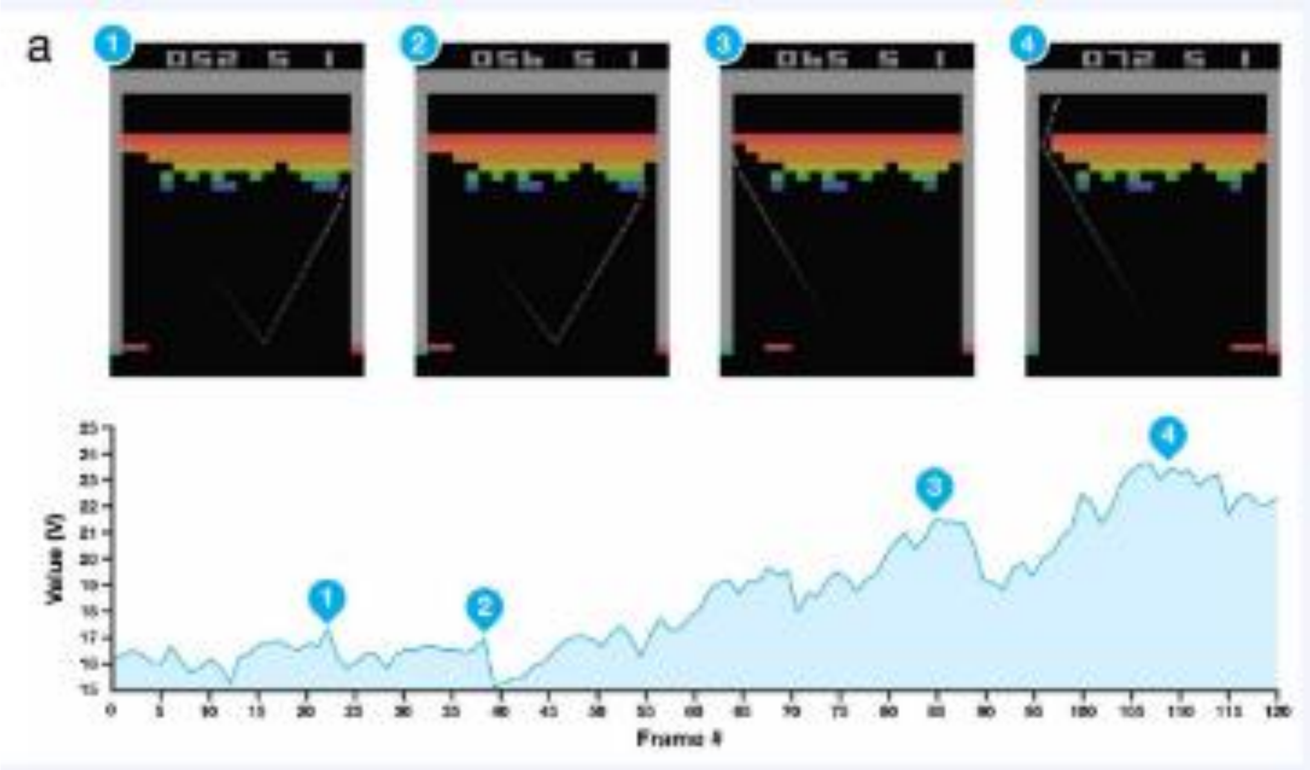


Similar states has points at similar locations in T-SNE

Although different visually, the states with similar expected reward (action-value) have similar locations

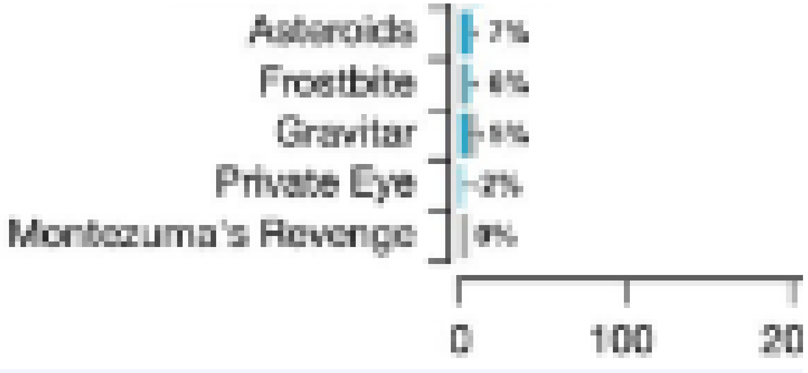
DQN: Results

- Agent's behavior shows that it learned a long-term strategy
  - E.g., In break out, the agent learned that the state number 4 has much higher value



DQN: Results

- But DQN suffers in a game like Montejuma’s revenge that requires a longer-term strategy
  - Sparse reward problem
  - Deceptive reward problem



Negative Reward occurs when the player hit the skull: deceptive reward

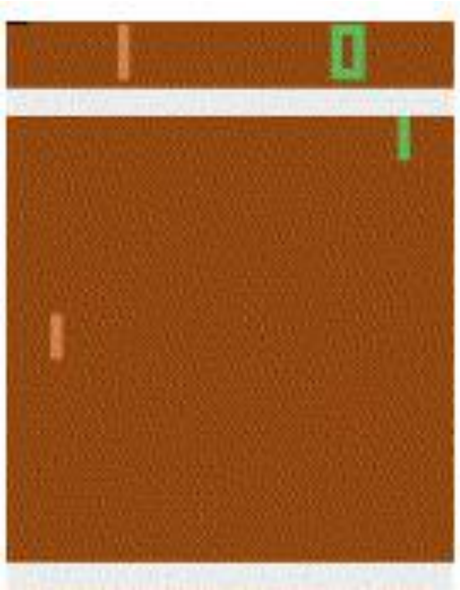
Code Ex.

- Python libraries:
  - gym==0.25.1
  - ale-py==0.7.5
  - torch==2.0.0 (cpu버전)
  - tensorboard==2.13.0

  *pip install gym[accept-rom-license]

- Code files:
  - Configuration.py
  - dqn_agent.py
  - eval.py
  - utils.py

Code Ex. : Pong



We'll use 'Pong-ram-v0' version

Observations

By default, the environment returns the RGB image that is displayed to human players as an observation. However, it is possible to observe

- The 128 Bytes of RAM of the console
- A grayscale image

instead. The respective observation spaces are

- `Box([0 ... 0], [255 ... 255], (128,), uint8)`

'Pong-ram-v0' version has 128 state (1 dim. vector)

Actions

Num	Action
0	NOOP
1	FIRE
2	RIGHT
3	LEFT
4	RIGHTFIRE
5	LEFTFIRE

Confgiration.py

```
config = AttrDict(
    gamma=0.99,
    lr=1e-4,
    batch_size=128,
    hidden_size=512,
    replay_capacity=50000,
    replay_init_ratio=0.5,
    train_env_steps=500000,
    target_update_period=2000,
    eps_init=1.0,
    eps_final=0.1,
    eps_decrease_step=50000,
    num_eval_episode=20,
    eval_start_step=0,
    eval_period=500,
    action_repeat=4
)
```

Dictionary that allows access to the key value by an attribute (e.g., config.gamma)

Roughly proportional to the complexity of game

50000 samples

Initial buffer-filling ratio before start learning

Total 500000 steps interaction

Target network update per 2000 steps

Epsilon decaying

Stop decaying after 50000 steps

Use avg. of 20 episodes to evaluate score

evaluate every 500 step

Utils.py

```
def create_env(config, render_mode='rgb_array'):
    env = gym.make("Pong-ram-v0", render_mode=render_mode)
    env = ObsNormalizationWrapper(env)
    env = RepeatedActionWrapper(env, config.action_repeat)
    return env
```

Env return RBG image

```
class RepeatedActionWrapper(gym.Wrapper):
    def __init__(self, env, n_repeat):
        self.env = env
        self.n_repeat = n_repeat
        self.recent_states = deque([], maxlen=4)
        self.observation_space = gym.spaces.Box(
            self.env.observation_space.low.repeat(self.n_repeat, axis=-1),
            self.env.observation_space.high.repeat(self.n_repeat, axis=-1),
            (self.env.observation_space.shape[0] * self.n_repeat,),
            np.uint8
        )
```

128 * 4 = (512,)

Customize pong-v0

Repeat env.observation_space.low
4 times and concatenation

Utils.py

```
def step(self, action):
    r_sum = 0
    done = False
    for _ in range(self.n_repeat):
        s_next, r, done, info = self.env.step(action)
        self.recent_states.append(s_next)
        r_sum += r

        if r != 0:
            done = True

        if done:
            break

    s_next = np.concatenate(self.recent_states, axis=0)
    return s_next, r_sum, done, info
```

4 times same actions

Add reward 4 times

Win: 1, Lose: -1.
To simply training, pong game ends after 1 match.

Concatenate 4 recent states

```
def reset(self):
    s = self.env.reset()
    for _ in range(20):
        s, _, _, _ = self.env.step(0)  # No action

    for _ in range(self.n_repeat):
        self.recent_states.append(s)

    s = np.concatenate(self.recent_states, axis=0)
    return s
```

20 steps No-op

Utils.py

```
class ObsNormalizationWrapper(gym.Wrapper):
    def __init__(self, env):
        self.env = env
        self.obs_subtration = 128
        self.obs_division = 128
        self.observation_space = gym.spaces.Box(
            (self.env.observation_space.low - self.obs_subtration) / self.obs_division,
            (self.env.observation_space.high - self.obs_subtration) / self.obs_division,
            self.env.observation_space.shape,
            np.float32
        )

    def reset(self):
        s = self.env.reset()
        s = (s - self.obs_subtration) / self.obs_division
        return s

    def step(self, action):
        s_next, r, done, info = self.env.step(action)
        s_next = (s_next - self.obs_subtration) / self.obs_division
        return s_next, r, done, info
```

Customize pong-v0

Normalization to -1 ~ 1

No change in shape

Agent learning (dqn_agent.py)

```
class ReplayMemory:
    def __init__(self, config):
        self.config = config
        self.buffer = deque([], maxlen=self.config.replay_capacity)

    def getsize(self):
        return len(self.buffer)

    def append(self, transition):
        buffer_size = len(self.buffer)
        self.buffer.append(transition)

    def sample(self, size):
        buffer_size = len(self.buffer)
        if buffer_size >= size:
            samples = random.sample(self.buffer, size)
        else:
            assert False, f"Buffer size ({buffer_size}) is smaller than the sample size
({size})"

        return samples
```

Attribute dictionary that allows access to the key value by attribute

Uniform random sampling

Assert False …. force the program to end

```python
class DQNAgent(nn.Module):
    def __init__(self, env, config):
        super().__init__()
        self.config = config
        self.replay_memory = ReplayMemory(self.config)

        d_state = env.observation_space.shape[0]
        n_action = env.action_space.n

        self.network = nn.Sequential(
            nn.Linear(d_state, config.hidden_size),
            nn.ELU(),
            nn.Linear(config.hidden_size, config.hidden_size),
            nn.ELU(),
            nn.Linear(config.hidden_size, n_action)
        )

        self.target_network = nn.Sequential(
            nn.Linear(d_state, config.hidden_size),
            nn.ELU(),
            nn.Linear(config.hidden_size, config.hidden_size),
            nn.ELU(),
            nn.Linear(config.hidden_size, n_action)
        )

        for param in self.target_network.parameters():
            param.requires_grad = False
```

nn.Module init. is required

…

Get State dims., action dims. to design the neural network

Parameters in the target network will have no gradients

```python
def update_target_network(self):
    self.target_network.load_state_dict(self.network.state_dict())

def set_optimizer(self):
    self.optimizer = torch.optim.AdamW(
        params=self.network.parameters(),
        lr=self.config.lr,
        weight_decay=1e-3
    )

def forward(self, x):
    Qs = self.network(x)
    return Qs

def forward_target_network(self, x):
    Qs = self.target_network(x)
    return Qs

def get_argmax_action(self, x):
    s = torch.from_numpy(x).reshape(1, -1).float()
    Qs = self.forward(s)
    argmax_action = Qs.argmax(dim=-1).item()
    return argmax_action
```

Set decay to prevent overfitting

Network input to nn.Module should be 2 dim.

Get integer value inside Tensor

```python
def train(self):
    transitions = self.replay_memory.sample(self.config.batch_size)
    states, actions, rewards, next_states, dones = zip(*transitions)

    states_array = np.stack(states, axis=0)  # (n_batch, d_state)
    actions_array = np.stack(actions, axis=0, dtype=np.int64)  # (n_batch)
    rewards_array = np.stack(rewards, axis=0)  # (n_batch)
    next_states_array = np.stack(next_states, axis=0)  # (n_batch, d_state)
    dones_array = np.stack(dones, axis=0)  # (n_batch)

    states_tensor = torch.from_numpy(states_array).float()  # (n_batch, d_state)
    actions_tensor = torch.from_numpy(actions_array)  # (n_batch)
    rewards_tensor = torch.from_numpy(rewards_array).float()  # (n_batch)
    next_states_tensor = torch.from_numpy(next_states_array).float()  # (n_batch, d_state)
    dones_tensor = torch.from_numpy(dones_array).float()  # (n_batch)

    Qs = self.forward(states_tensor)  # (n_batch, n_action)
    next_Qs = self.forward_target_network(next_states_tensor)  # (n_batch, n_action)
```

Uniform sampling of Replay buffer

Batch shape x State dim. (N x D)
[Tensor(N x D), Tensor(N x D), Tensor(N x D)]

Transform into tensors to put Into the network

Forward (states_tensor) through Q network. And get Qs output

```
def train(self):

    :

    # index dimension should be the same as the source tensor
    chosen_Q = Qs.gather(dim=-1, index=actions_tensor.reshape(-1, 1)).reshape(-1)
    target_Q = rewards_tensor + (1 - dones_tensor) * config.gamma * next_Qs.max(dim=-1).values

    criterion = nn.SmoothL1Loss()
    loss = criterion(chosen_Q, target_Q)

    # Update by gradient descent
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    return loss.item()
```

(Get Qs for actions taken)

(n_batch, 1)-> (n_batch)

If terminal state, target is just reward, r

* SmoothL1Loss:

Outlier makes the learning with MSE unstable

$$\mathrm{ln} = \begin{cases} \dfrac{0.5(x_n - yn)2}{\mathrm{beta}}, & \mathrm{if}|x_n - yn| < \mathrm{beta} \\ |x_n - yn| - 0.5 * \mathrm{beta}, \mathrm{otherwise} & \end{cases}$$

```python
def eval_agent(config, env, agent):
    score_sum = 0
    step_count_sum = 0
    for _ in range(config.num_eval_episode):
        s = env.reset()
        step_count = 0
        done = False
        score = 0
        while not done:
            with torch.no_grad():
                a = agent.get_argmax_action(s)

            s_next, r, done, info = env.step(a)
            step_count += 1

            score += r
            s = s_next

        score_sum += score
        step_count_sum += step_count

    score_avg = score_sum / config.num_eval_episode
    step_count_avg = step_count_sum / config.num_eval_episode
    return score_avg, step_count_avg
```

Average after 20 episode evaluation

Set no_grad in evaluation
Argmax action in evaluation

```python
if __name__ == "__main__":
    env = create_env(config)
    env_eval = create_env(config)
    agent = DQNAgent(env, config)
    agent.set_optimizer()

    dt_now = datetime.datetime.now()
    logdir = f"logdir/{dt_now.strftime('%y-%m-%d_%H-%M-%S')}"
    writer = SummaryWriter(logdir)

    # Reset Replay Buffer
    init_replay_buffer_size = int(config.replay_init_ratio * config.replay_capacity)
    s = env.reset()
    step_count = 0
    for _ in range(init_replay_buffer_size):
        a = np.random.choice(env.action_space.n)
        s_next, r, done, info = env.step(a)
        step_count += 1

        transition = (s, a, r, s_next, done)
        agent.replay_memory.append(transition)

        s = s_next
        if done:
            s = env.reset()
            step_count = 0
```

Logging directories
Give the log directory as input

Uniform random action sampling

```python
if __name__ == "__main__":
    :
    # Train agent
    s = env.reset()
    step_count = 0
    for step_train in range(config.train_env_steps):
        eps = get_eps(config, step_train)
        is_random_action = np.random.choice(2, p=[1 - eps, eps])
        if is_random_action:
            a = np.random.choice(env.action_space.n)  # uniform random action
        else:
            a = agent.get_argmax_action(s)

        s_next, r, done, info = env.step(a)
        step_count += 1

        transition = (s, a, r, s_next, done)
        agent.replay_memory.append(transition)

        s = s_next
        if done:
            s = env.reset()
            step_count = 0

        if step_train % config.target_update_period == 0:
            agent.update_target_network()

        if step_train % 4 == 0:
            loss = agent.train()
```

Pick random action by Epsilon greedy policy

Update target every 100 steps

Train every 4 transitions

eval.py

Output RGB
(no need call
env.render())

```
if __name__ == "__main__":
    args = parse_args()
    config.num_eval_episode = args.num_eval
    env = create_env(config, render_mode='human')
    agent = DQNAgent(env, config)

    if args.model_path:
        state_dict = torch.load(args.model_path)
        agent.load_state_dict(state_dict)

    eval_agent_with_rendering(config, env, agent)
```

```
def eval_agent_with_rendering(config, env, agent):
    score_sum = 0
    step_count_sum = 0
    for _ in range(config.num_eval_episode):
        s = env.reset()
        step_count = 0
        done = False
        score = 0
        while not done:
            with torch.no_grad():
                a = agent.get_argmax_action(s)
            s_next, r, done, info = env.step(a)
            step_count += 1

            score += r
            s = s_next

        score_sum += score
        step_count_sum += step_count

    score_avg = score_sum / config.num_eval_episode
    step_count_avg = step_count_sum / config.num_eval_episode
```

train agent

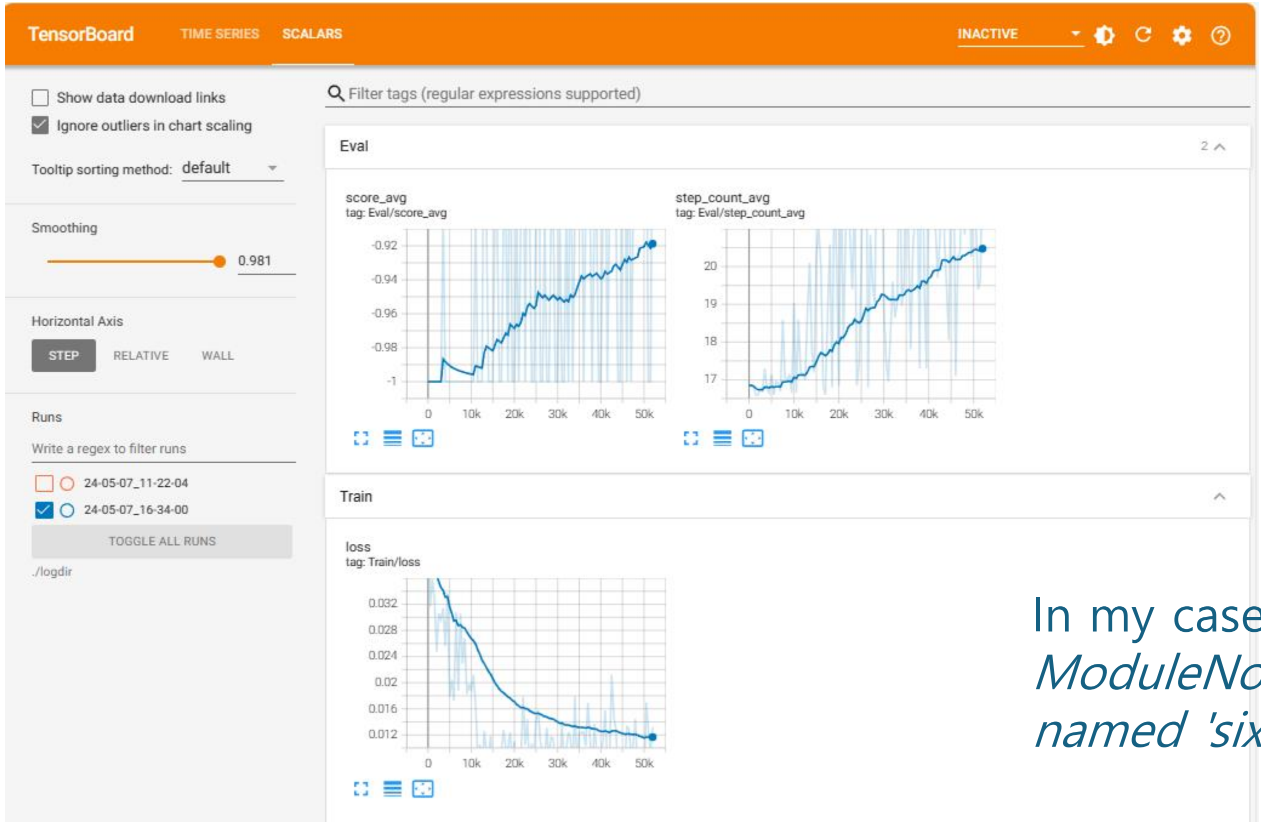
- $python dqn_agent.py

```
(RL)  $ls
__pycache__            dqn_agent.py           logdir
configuration.py       eval.py                utils.py
(RL)  $python dqn_agent.py
/Users/taehwan/Project/FastCampus/RL/lib/python3.8/site-packages/torch/utils/tensorb
stutils Version classes are deprecated. Use packaging.version instead.
  if not hasattr(tensorboard, "__version__") or LooseVersion(
/Users/taehwan/Project/FastCampus/RL/lib/python3.8/site-packages/gym/envs/registrati
ent Pong-ram-v0 is out of date. You should consider upgrading to version `v4`.
  logger.warn(
A.L.E: Arcade Learning Environment (version 0.7.5+db37282)
[Powered by Stella]
/Users/taehwan/Project/FastCampus/RL/lib/python3.8/site-packages/gym/core.py:329: De
er in old step API which returns one bool instead of two. It is recommended to set `
is will be the default behaviour in future.
  deprecation(
/Users/taehwan/Project/FastCampus/RL/lib/python3.8/site-packages/gym/wrappers/step_a
: WARN: Initializing environment in old step API which returns one bool instead of t
=True` to use new step API. This will be the default behaviour in future.
  deprecation(
/Users/taehwan/Project/FastCampus/RL/lib/python3.8/site-packages/gym/spaces/box.py:1
 lowered by casting to float32
  logger.warn(f"Box bound precision lowered by casting to {self.dtype}")
/Users/taehwan/Project/FastCampus/RL/lib/python3.8/site-packages/gym/utils/passive_e
N: Core environment is written in old step API which returns one bool instead of two
ment with new step API.
  logger.deprecation(
/Users/taehwan/Project/FastCampus/RL/lib/python3.8/site-packages/gym/utils/passive_e
.bool8` is a deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(done, (bool, np.bool8)):
[0] eps: 1.000 loss: 0.061 score_avg: -0.800 step_count_avg: 21.000
[500] eps: 0.991 loss: 0.029 score_avg: -1.000 step_count_avg: 16.900
[1000] eps: 0.982 loss: 0.024 score_avg: -1.000 step_count_avg: 17.000
[1500] eps: 0.973 loss: 0.011 score_avg: -1.000 step_count_avg: 16.750
[2000] eps: 0.964 loss: 0.047 score_avg: -1.000 step_count_avg: 16.850
[2500] eps: 0.955 loss: 0.017 score_avg: -1.000 step_count_avg: 16.650
```

500000 environment step training take about 1 hour in cpu.

tensorboard

- $tensorboard –logdir=logdir



In my case, there was an error, *ModuleNotFoundError: No module named 'six'*. So, I did 'pip install six'

results

- $python eval.py –num_eval=5 –model_path=logdir/24-05-07_11-22-04/state_dict.pth