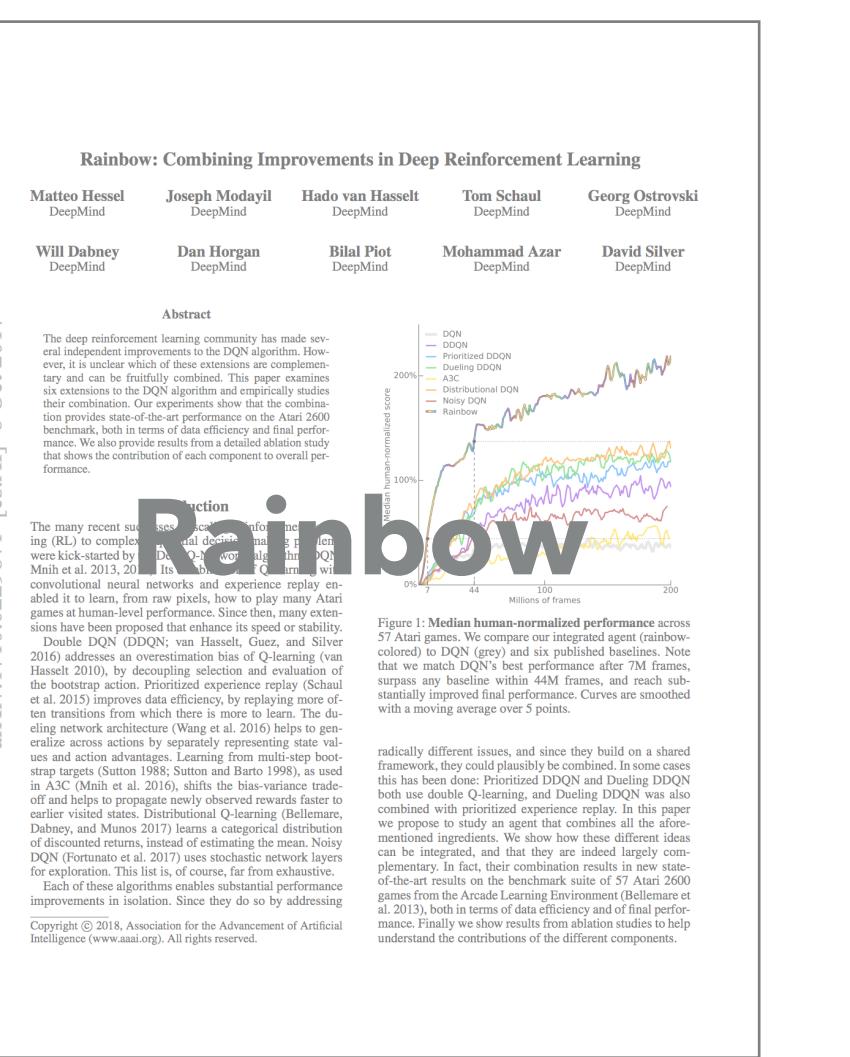
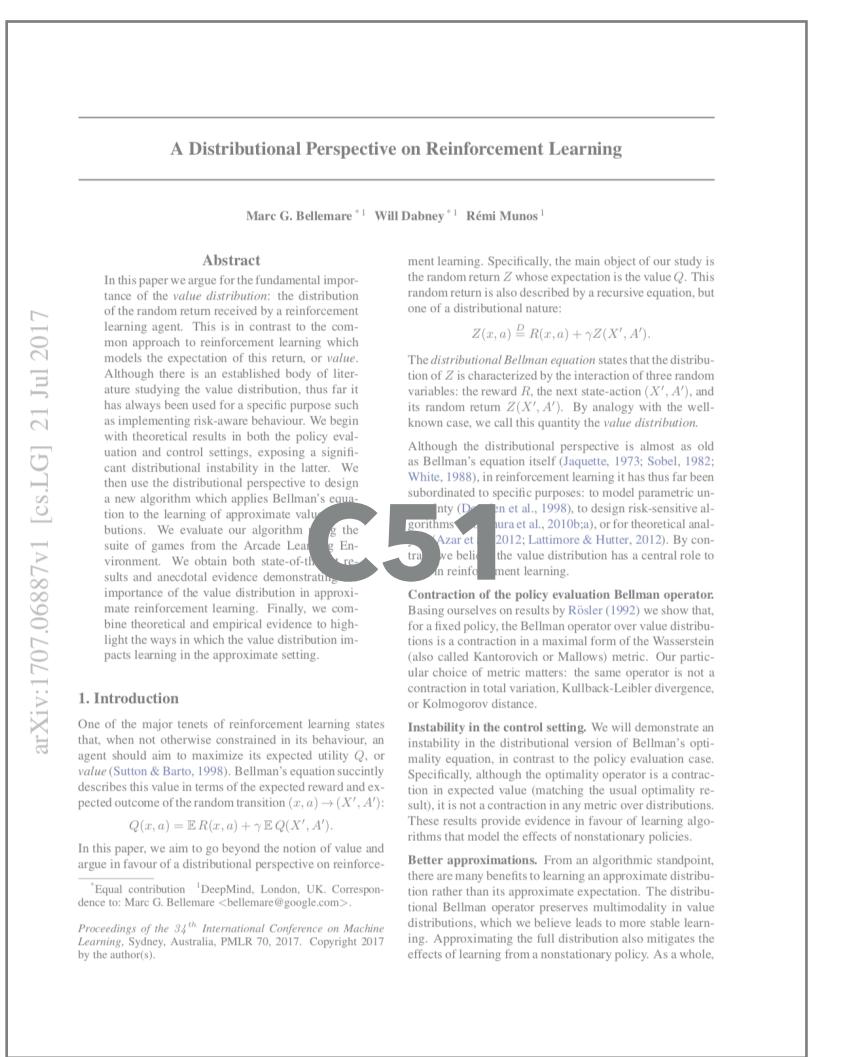
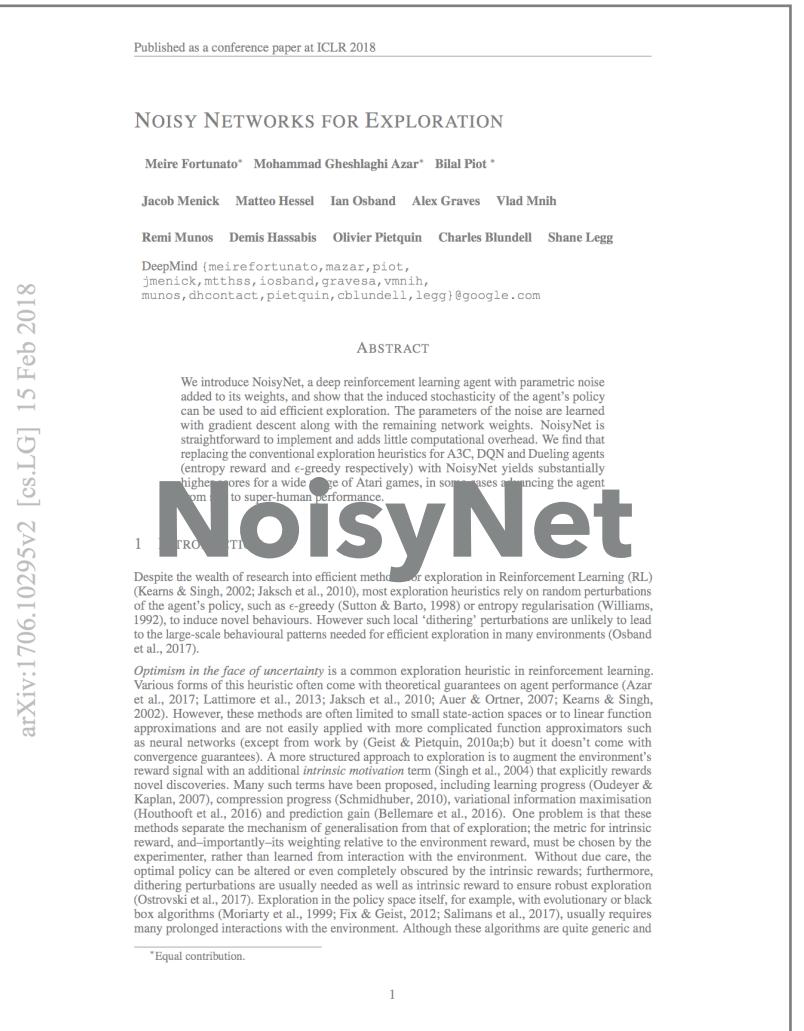
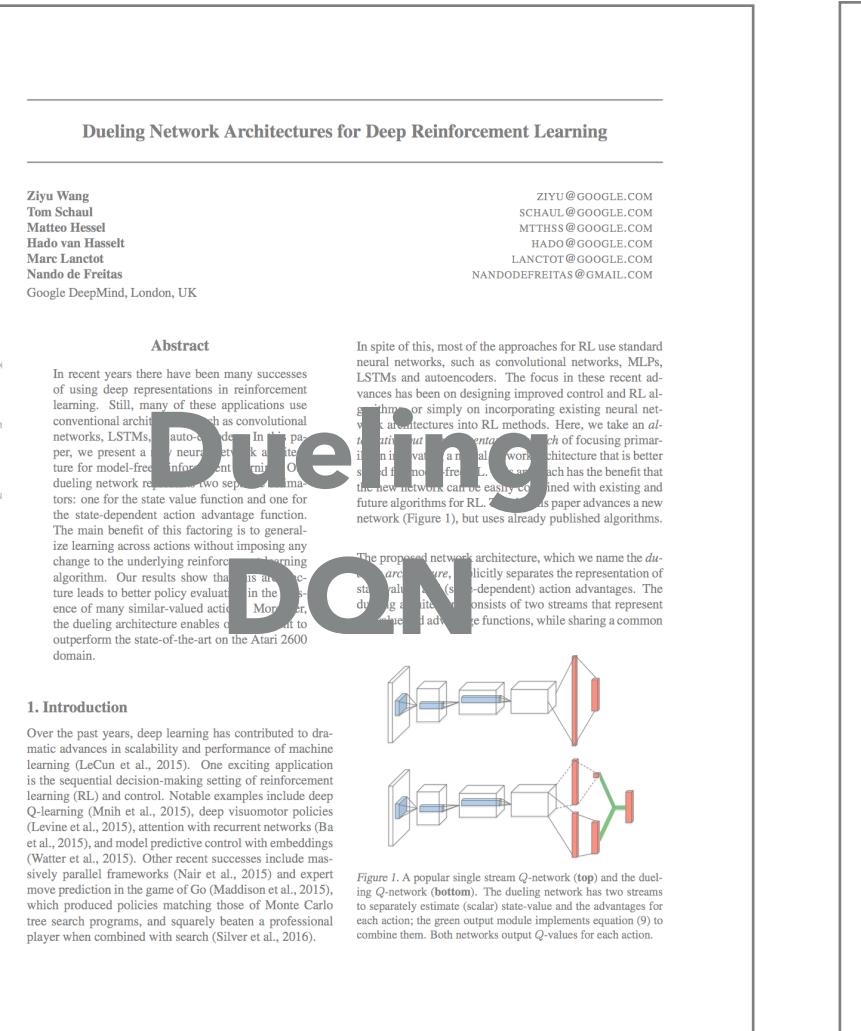
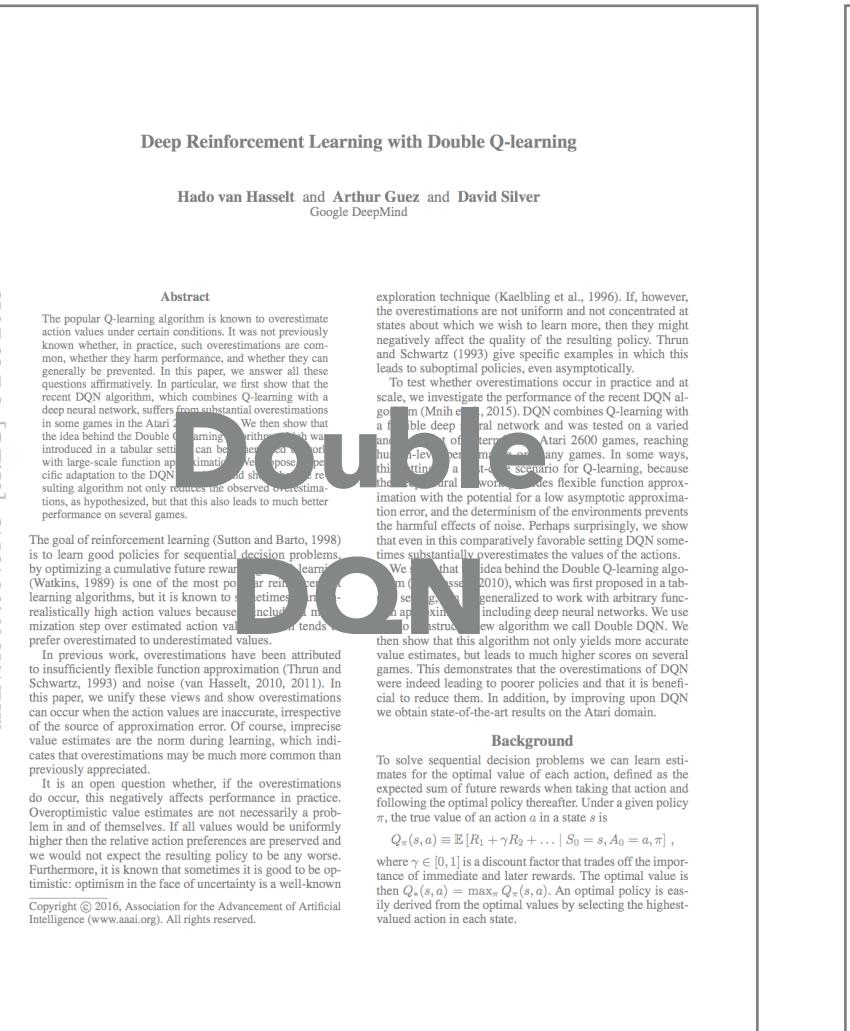
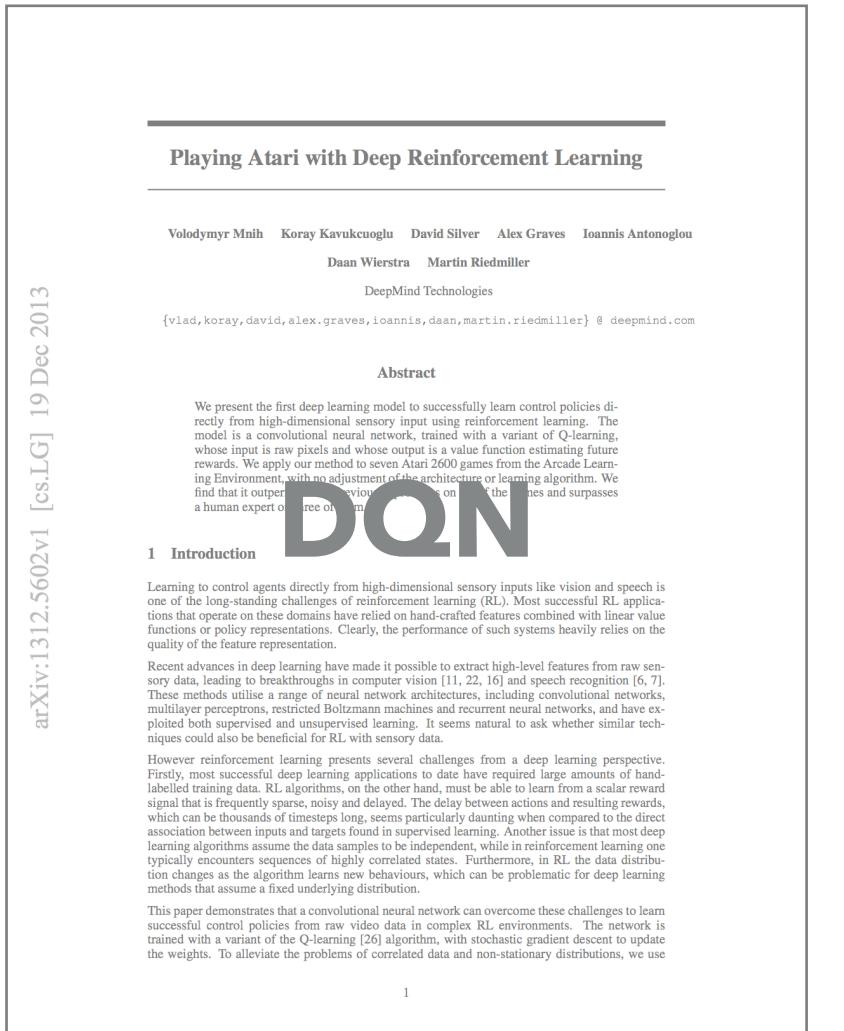


RL Adventure

DQN, Double DQN & Duel DQN

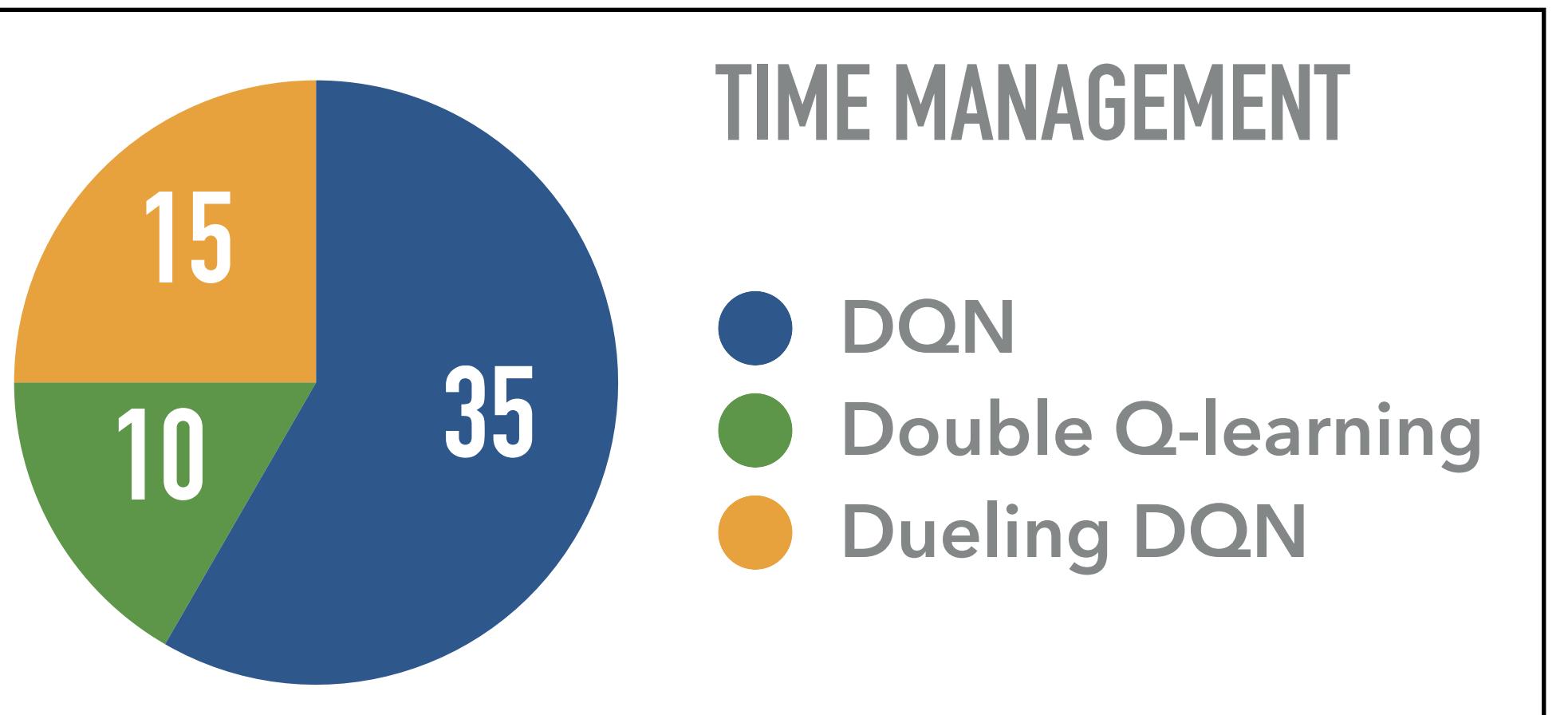
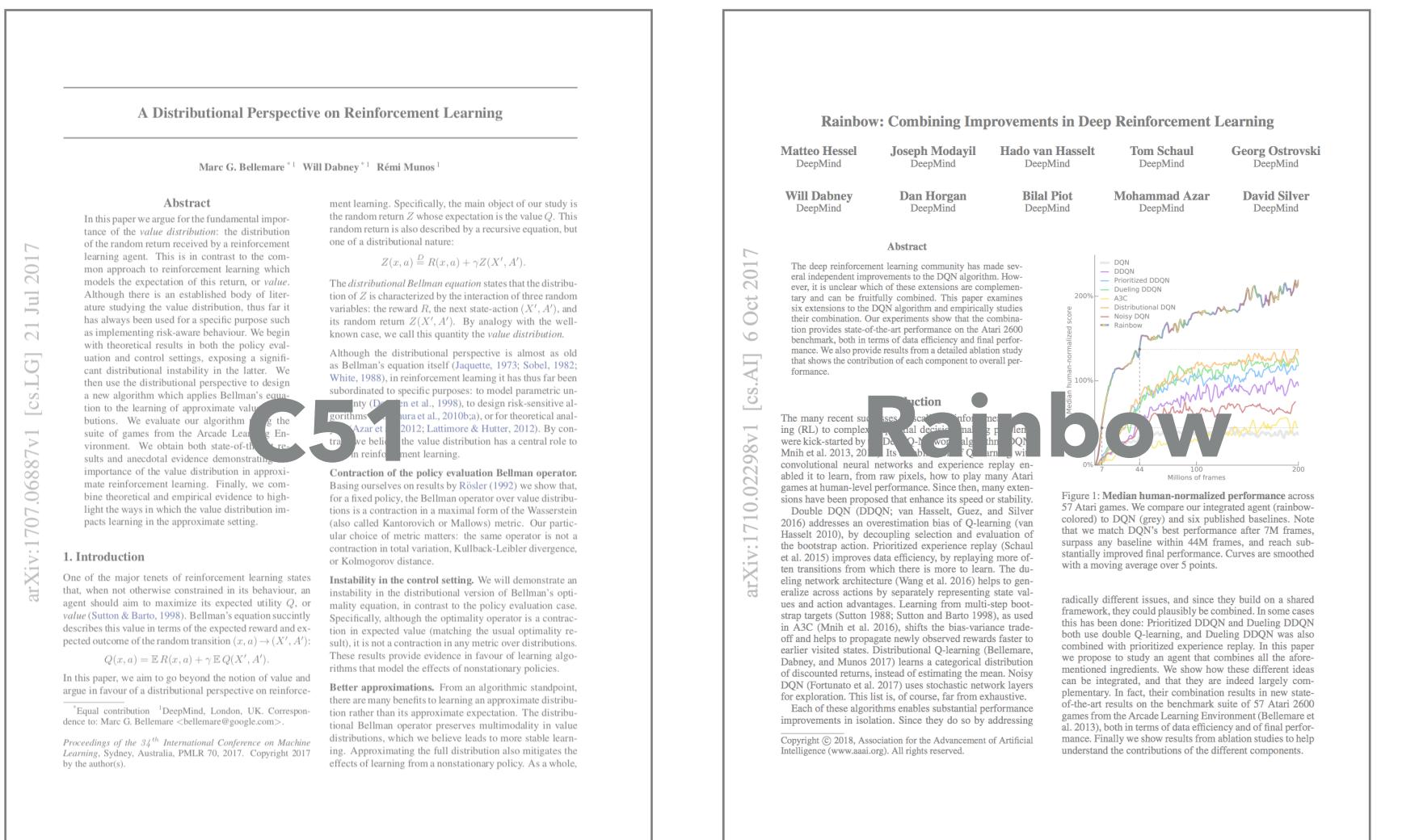
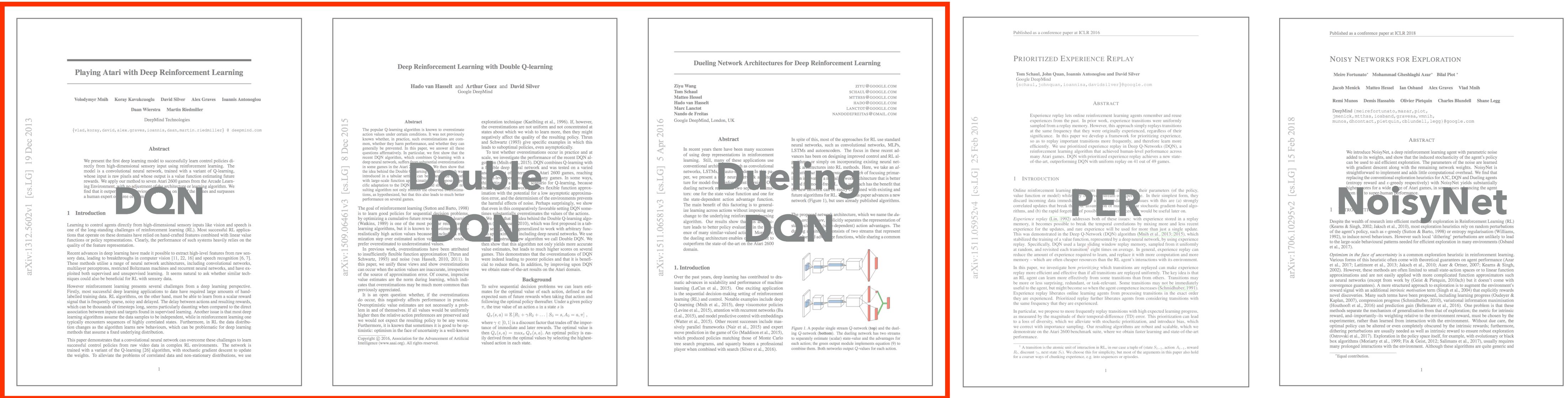
성태경

OUTLINE



+ Implementations

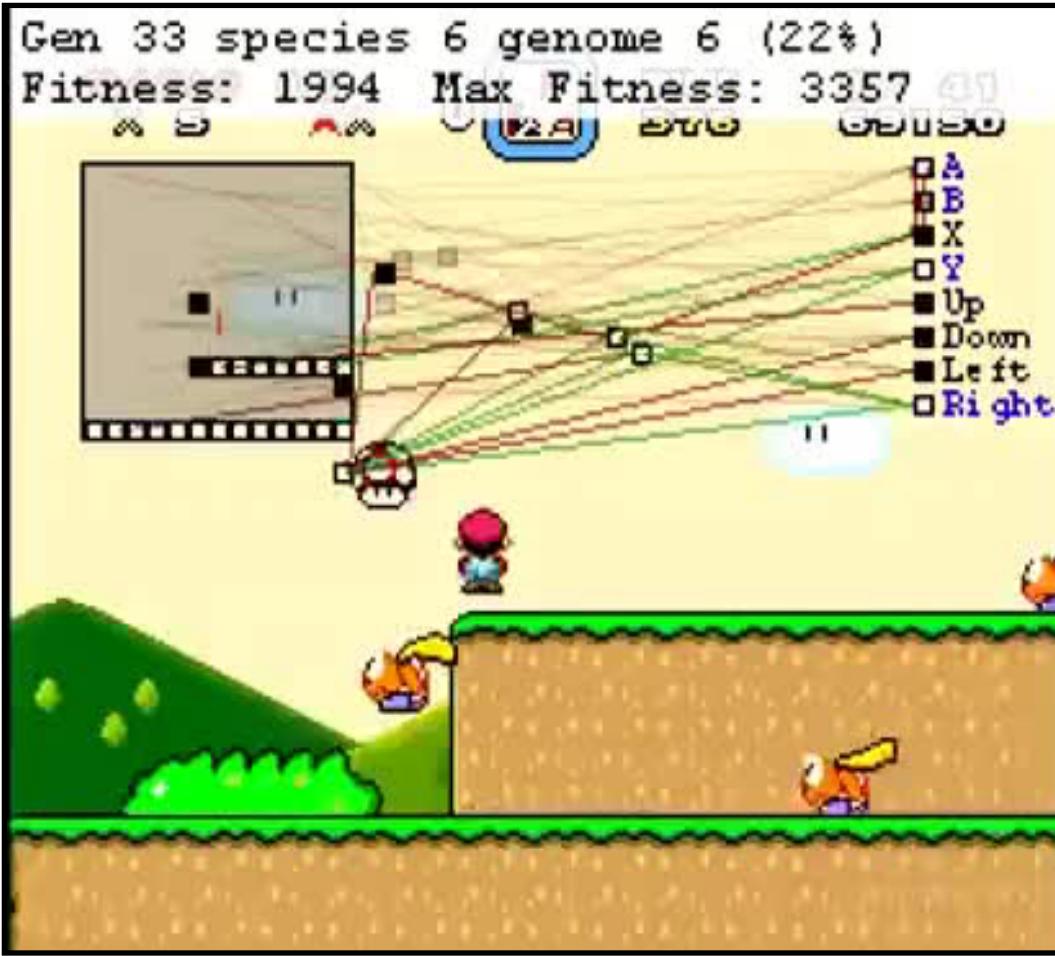
OUTLINE



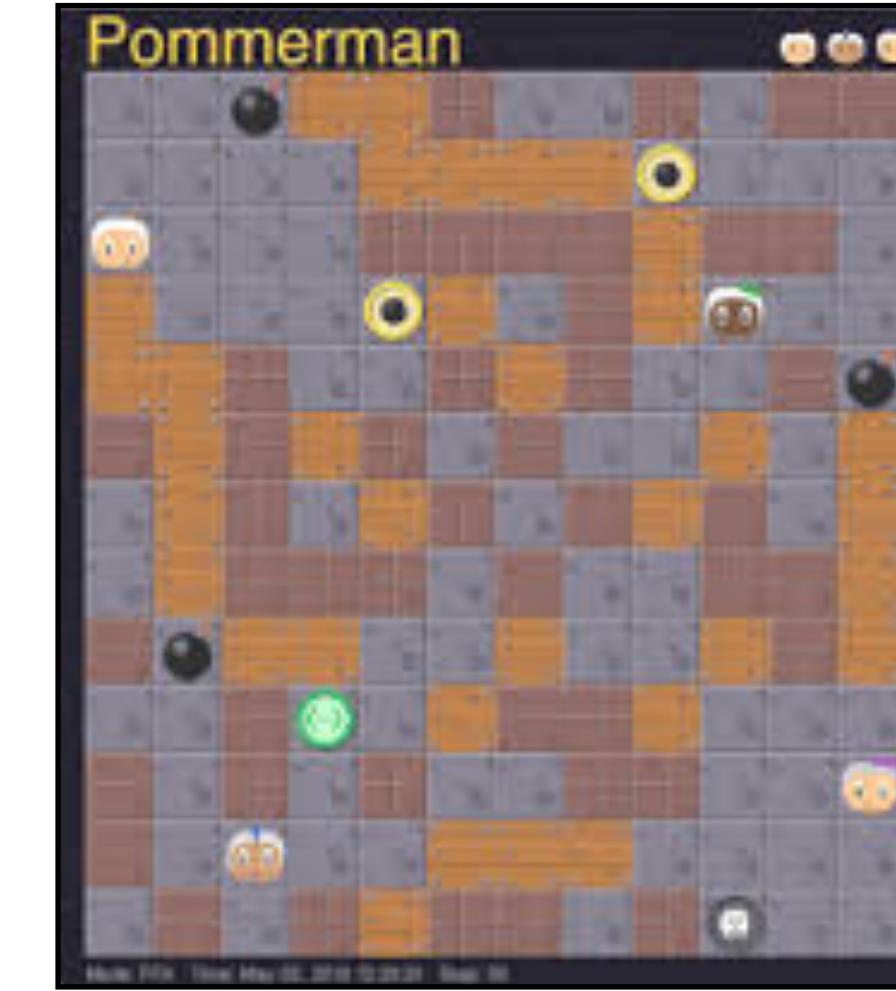
RL APPLICATIONS



[Atari]



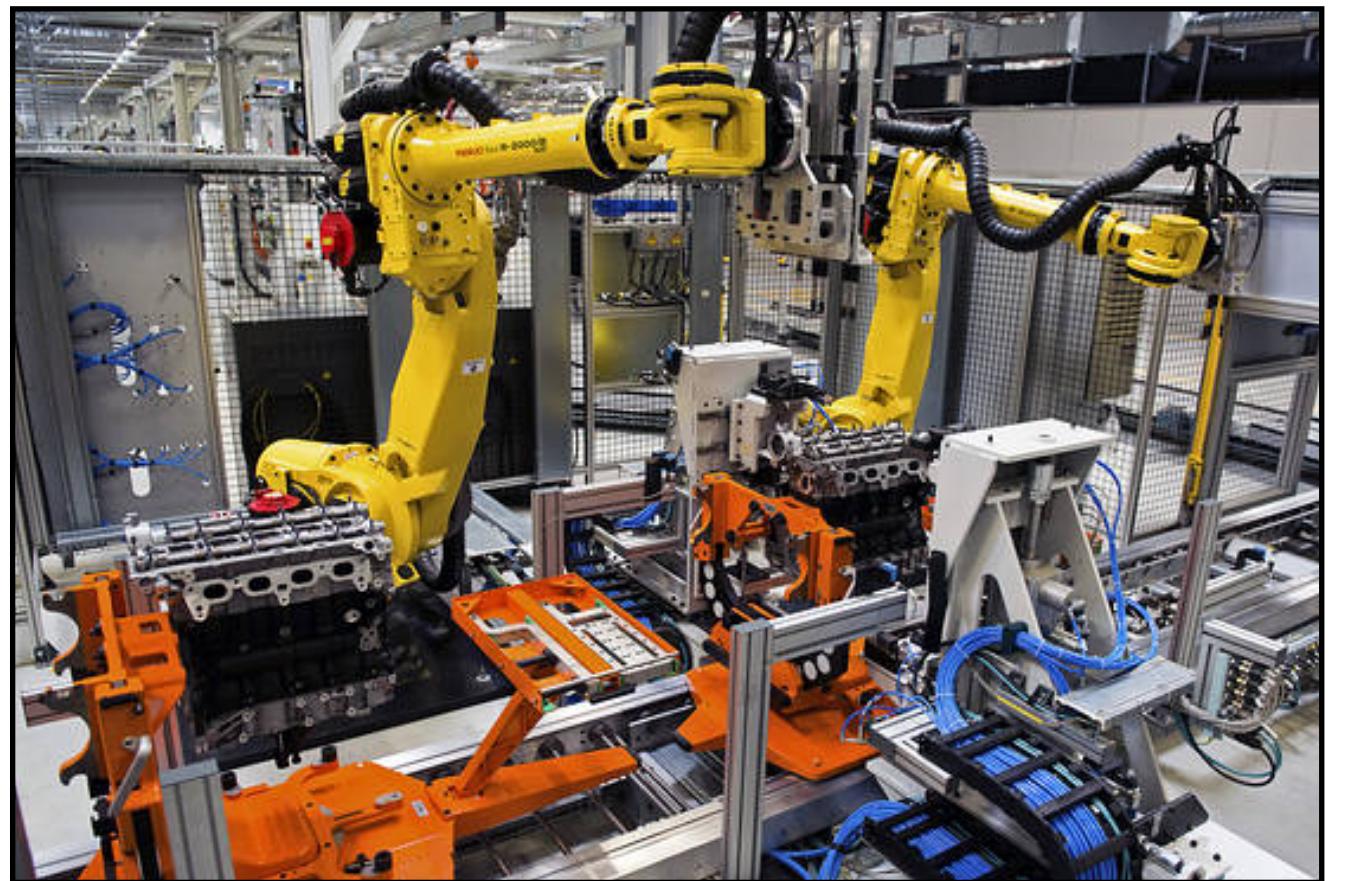
[Mario]



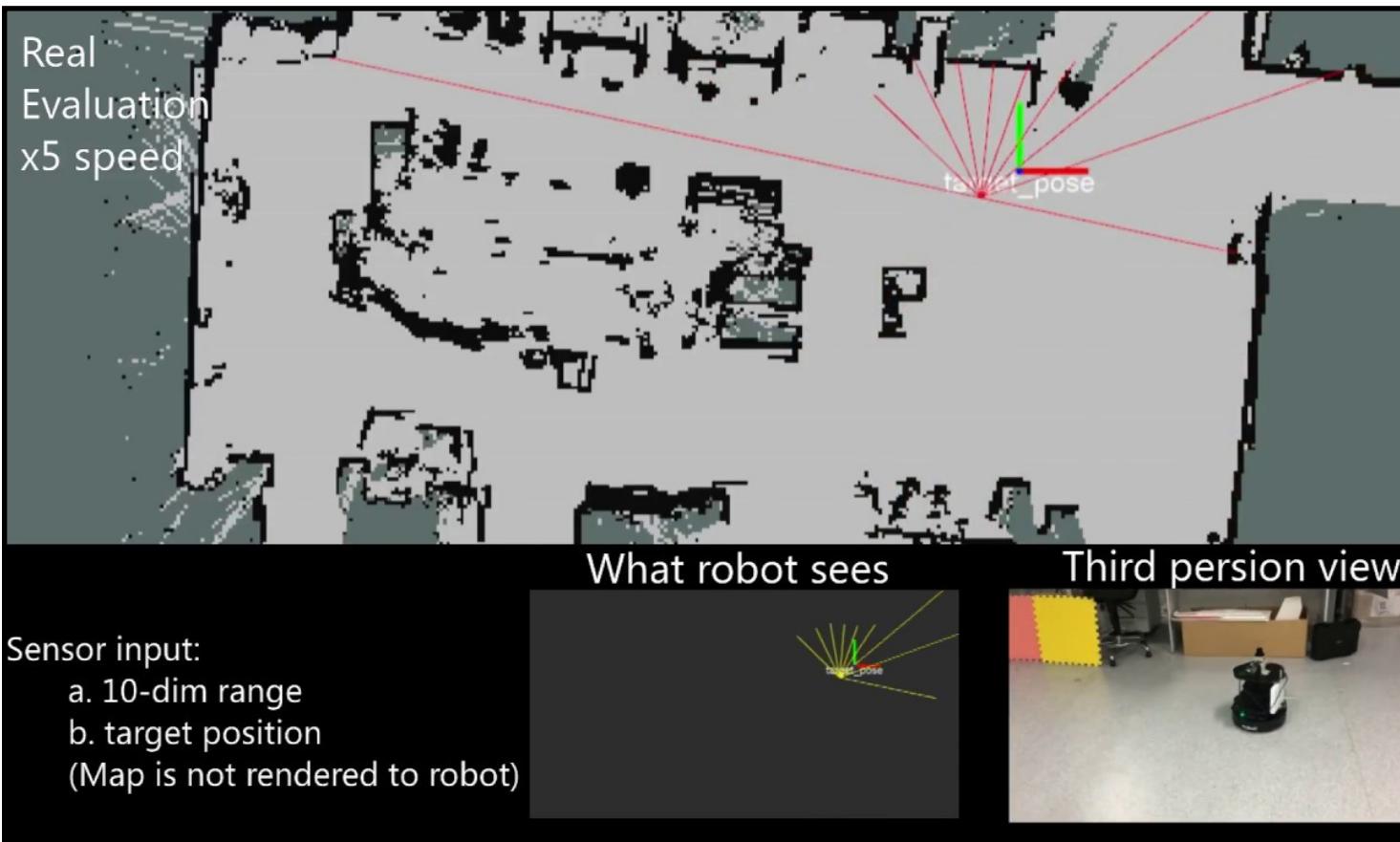
[Pomberman]



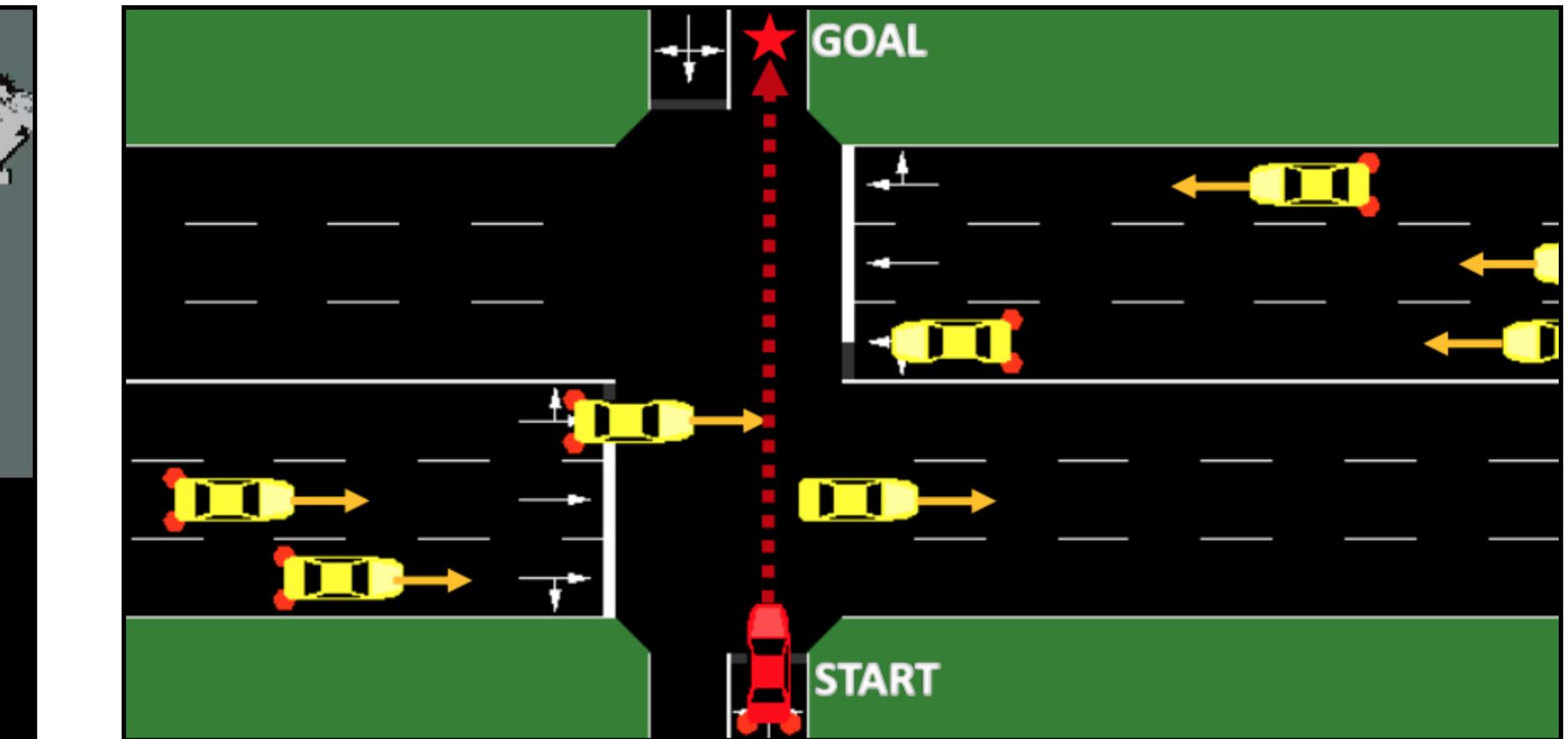
[Go]



[Robotics]

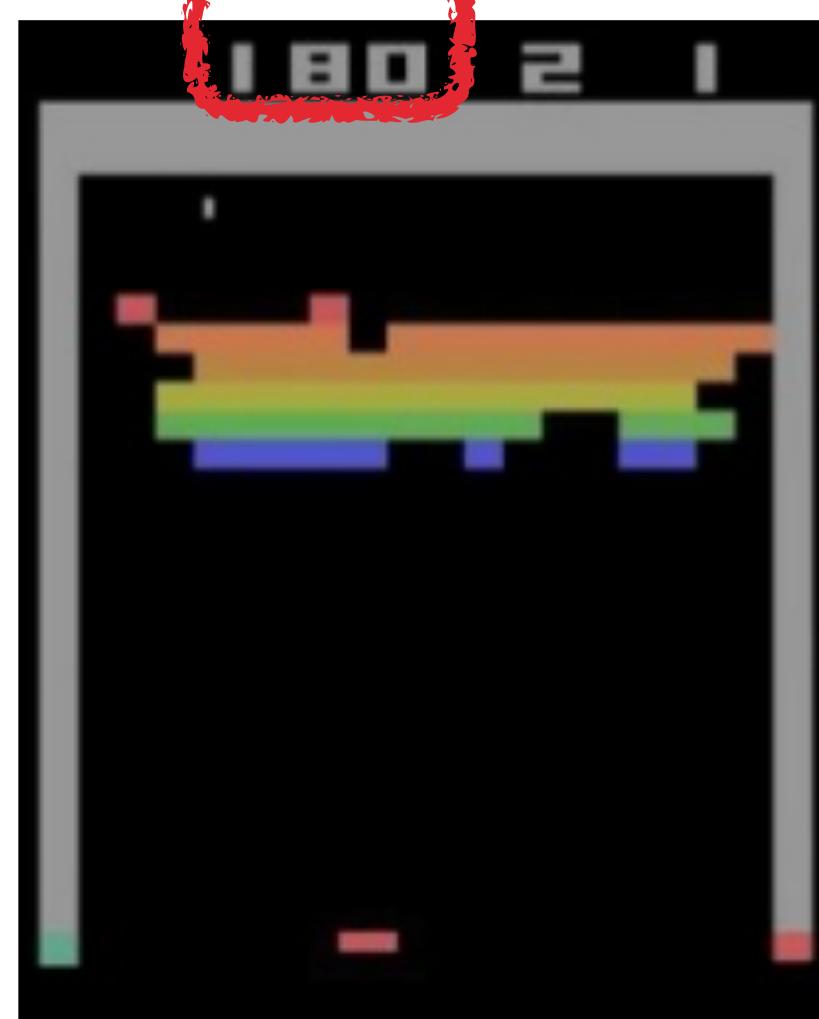


Sensor input:
a. 10-dim range
b. target position
(Map is not rendered to robot)

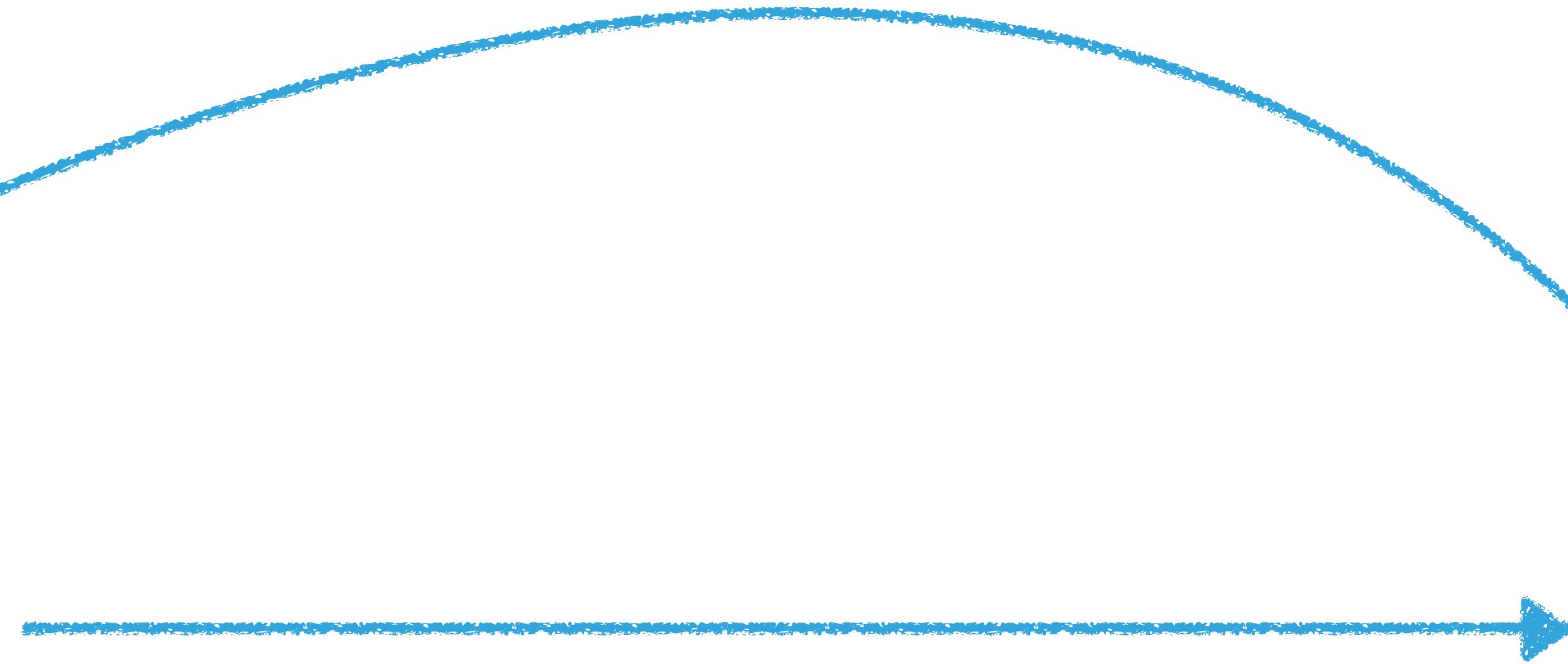


[Autonomous driving]

HIGH-LEVEL PROCESS

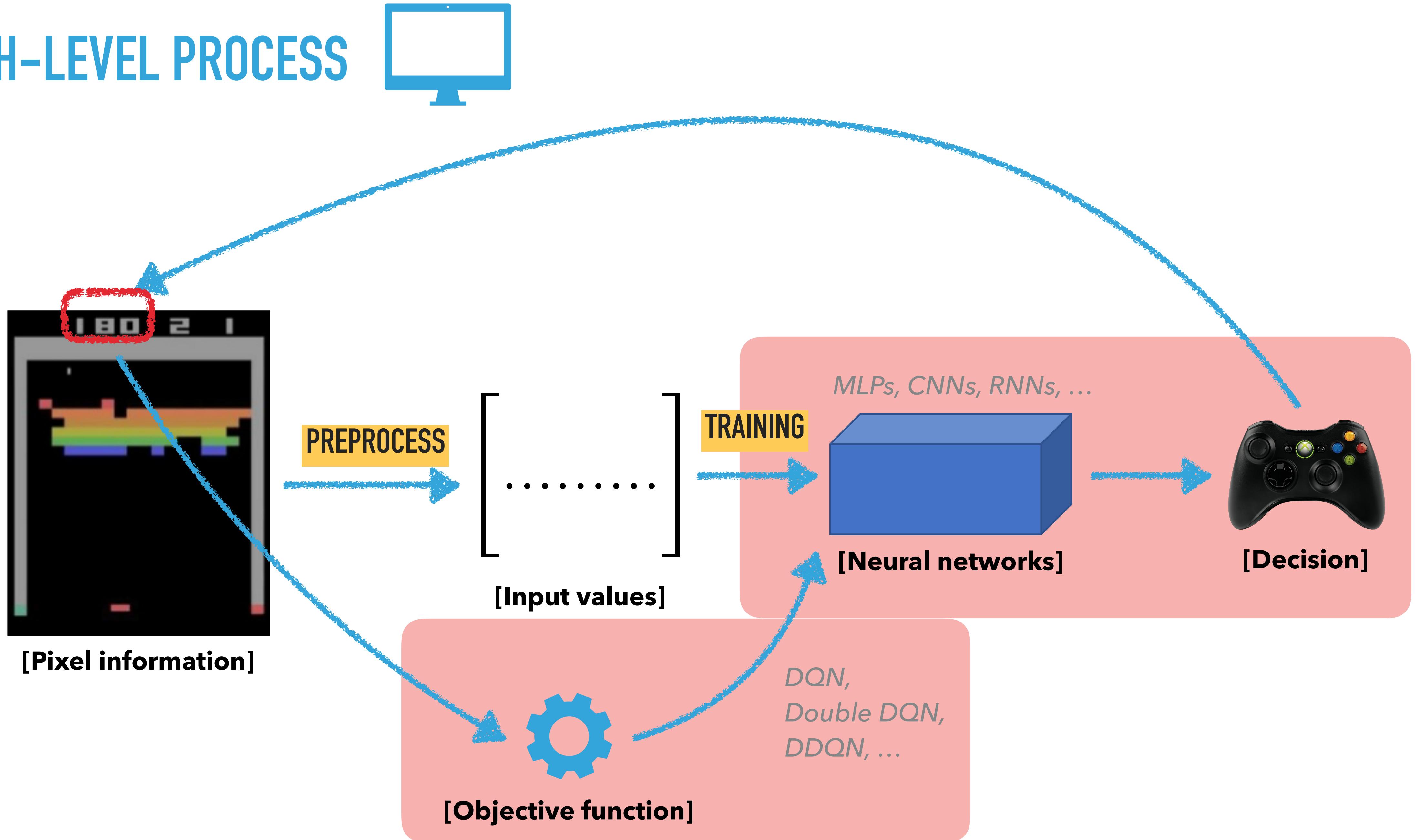


[Pixel information]



[Decision]

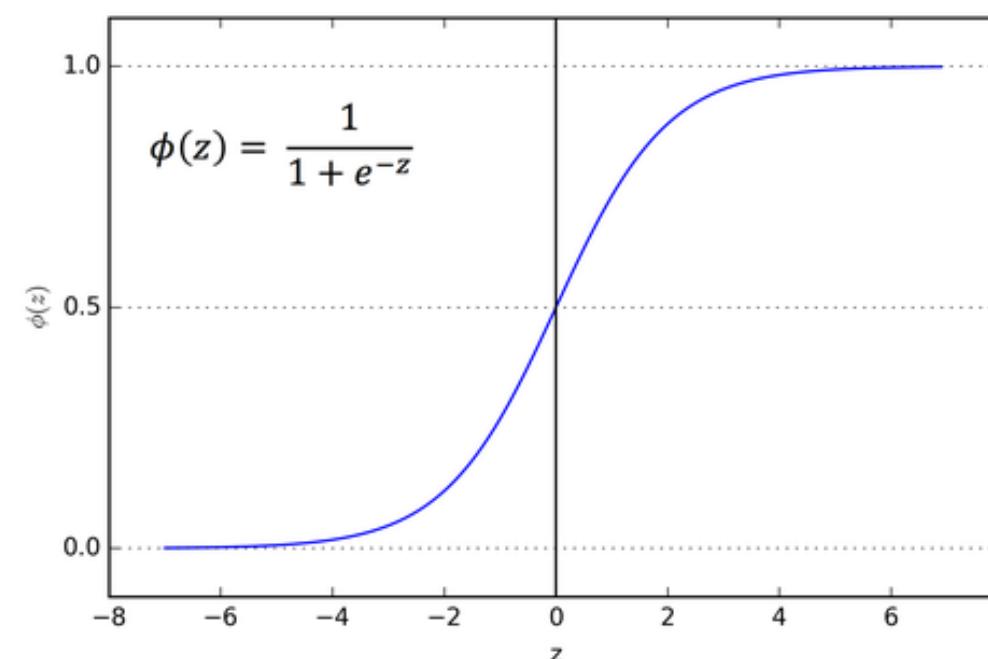
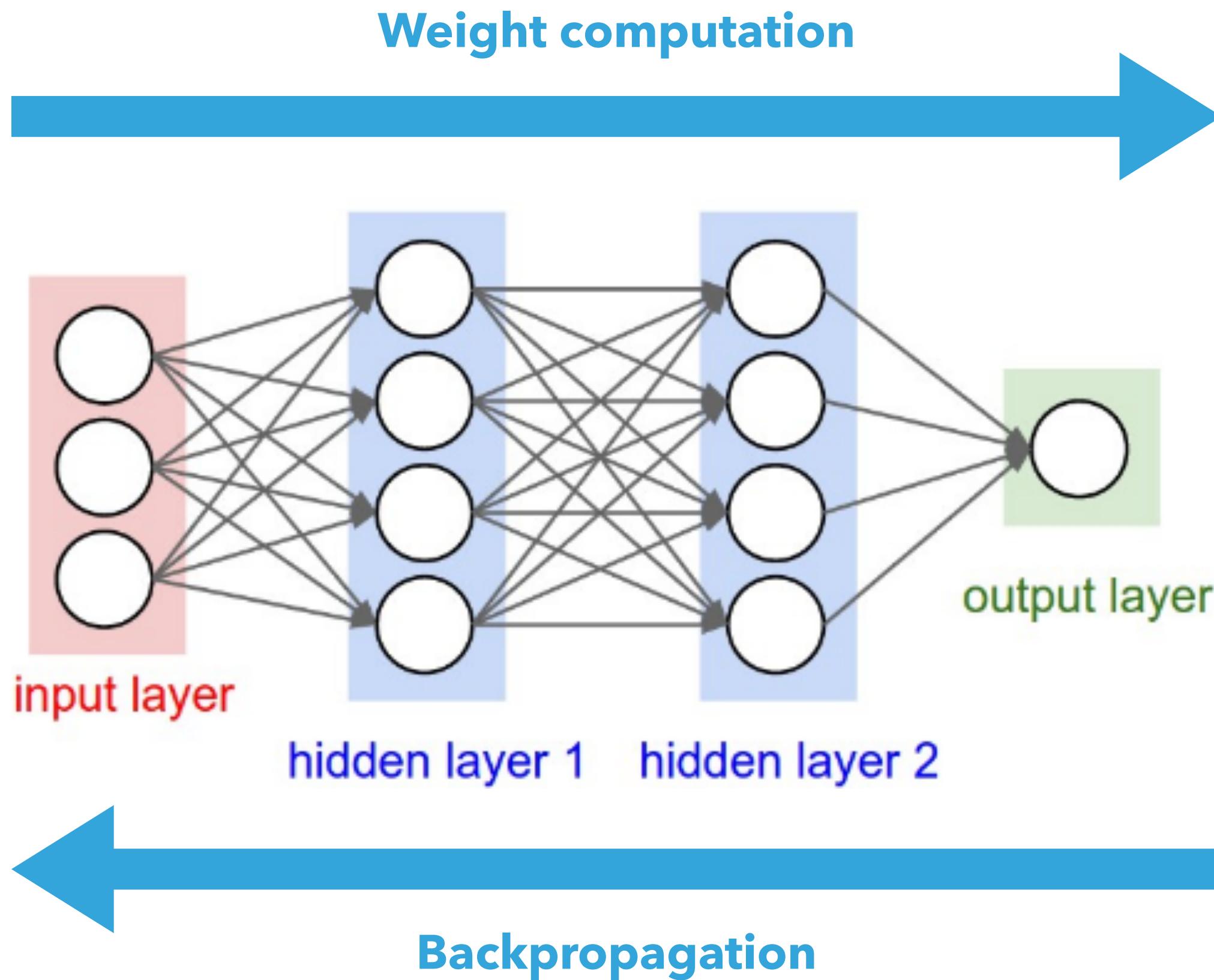
HIGH-LEVEL PROCESS

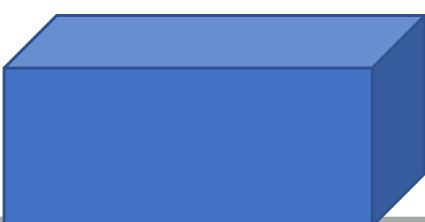


DEEP Q-NETWORK (DQN)



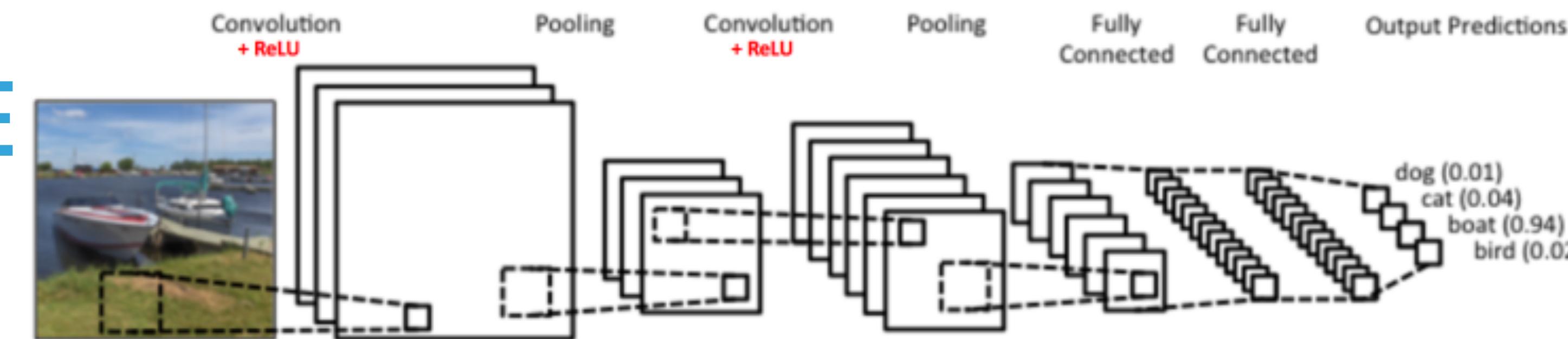
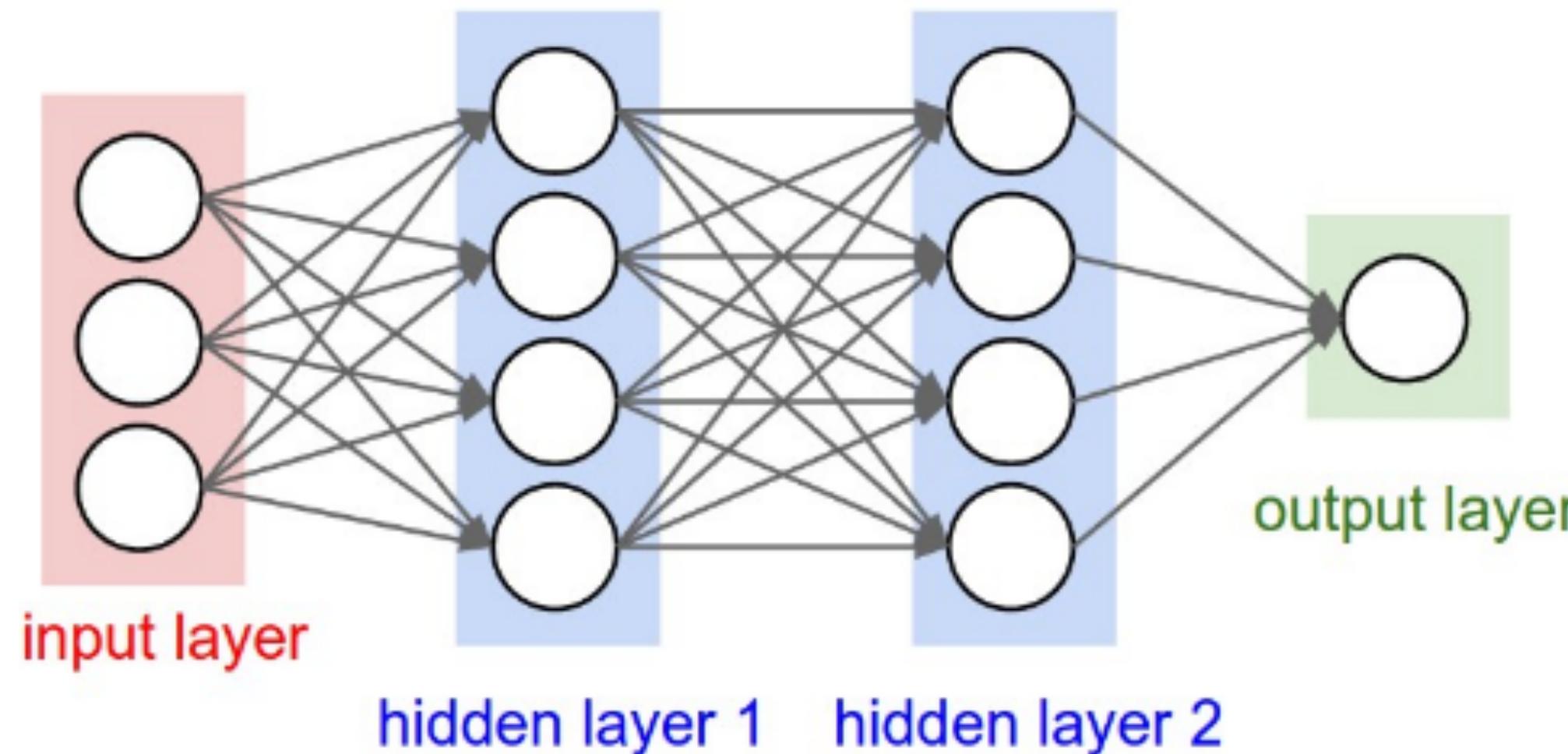
NEURAL NETWORKS IN ONE MINUTE



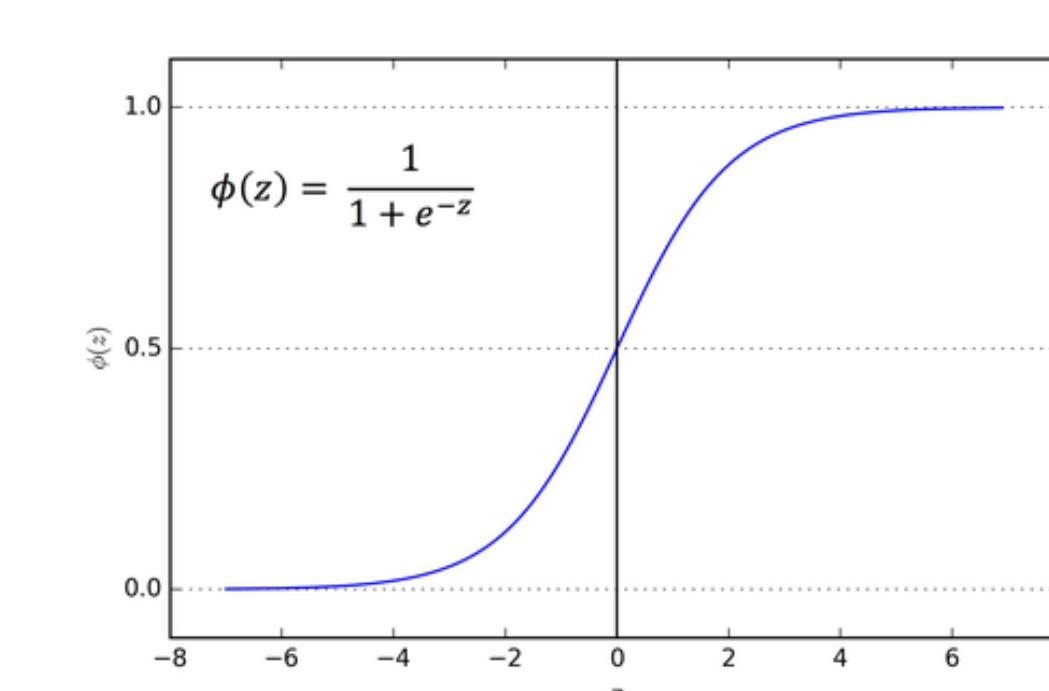


NEURAL NETWORKS IN ONE MINUTE

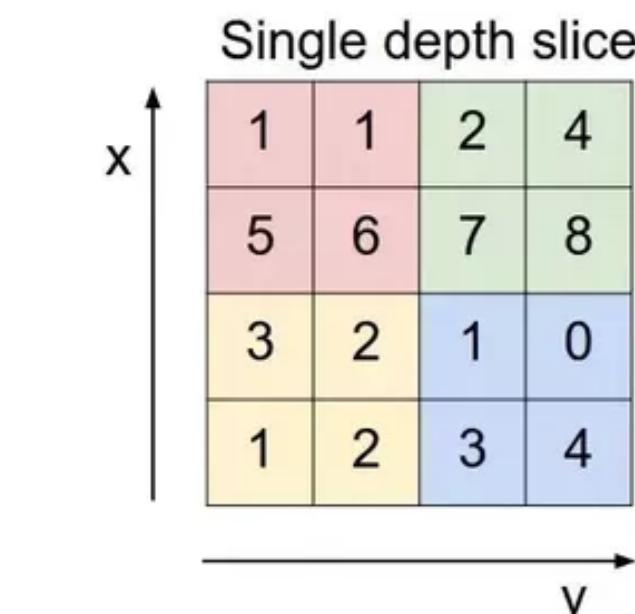
Weight computation



Convolutional neural network



Non-linear function



Max-pooling

Feedforward output, y_i
cat dog horse

5	4	2
4	2	8
4	4	1

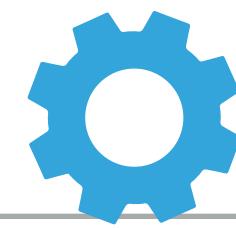
Softmax output, $S(y_i)$
cat dog horse

0.71	0.26	0.04
0.02	0.00	0.98
0.49	0.49	0.02

(3,)

(3,)

Softmax



Q-LEARNING

- ▶ Objective: Learn a policy, which tells an agent what action to take under what circumstances

$$Q_{\pi}(s, a) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t R(x_t, a_t) \right], \gamma \in (0, 1)$$

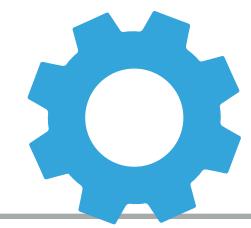
[Action selection]

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a))$$

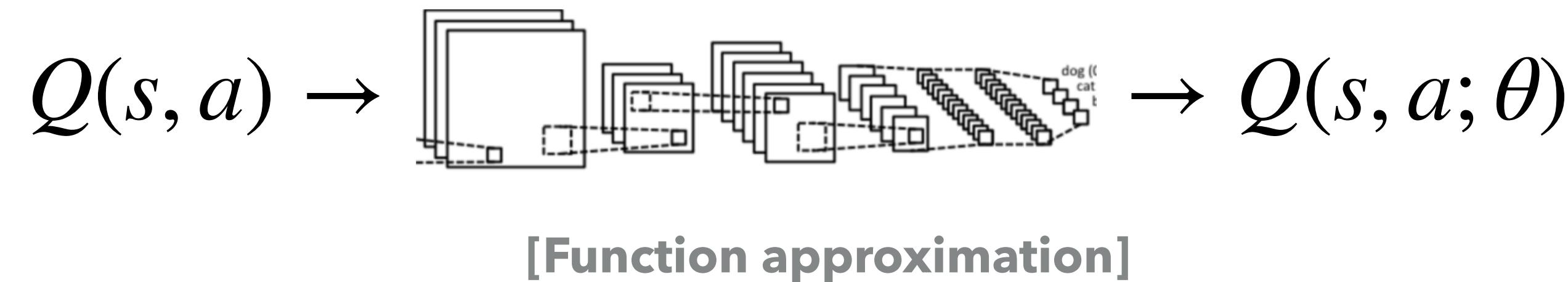
[Value iteration update]

C. J. C. H. Watkins, P. Dayan. Q-learning. 1992.

V. Minh, et al. Playing Atari with Deep Reinforcement Learning. NIPS, 2013

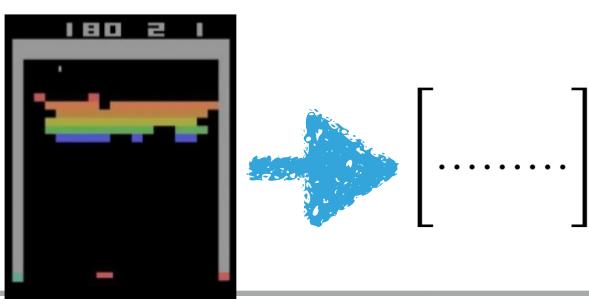


MOTIVATION

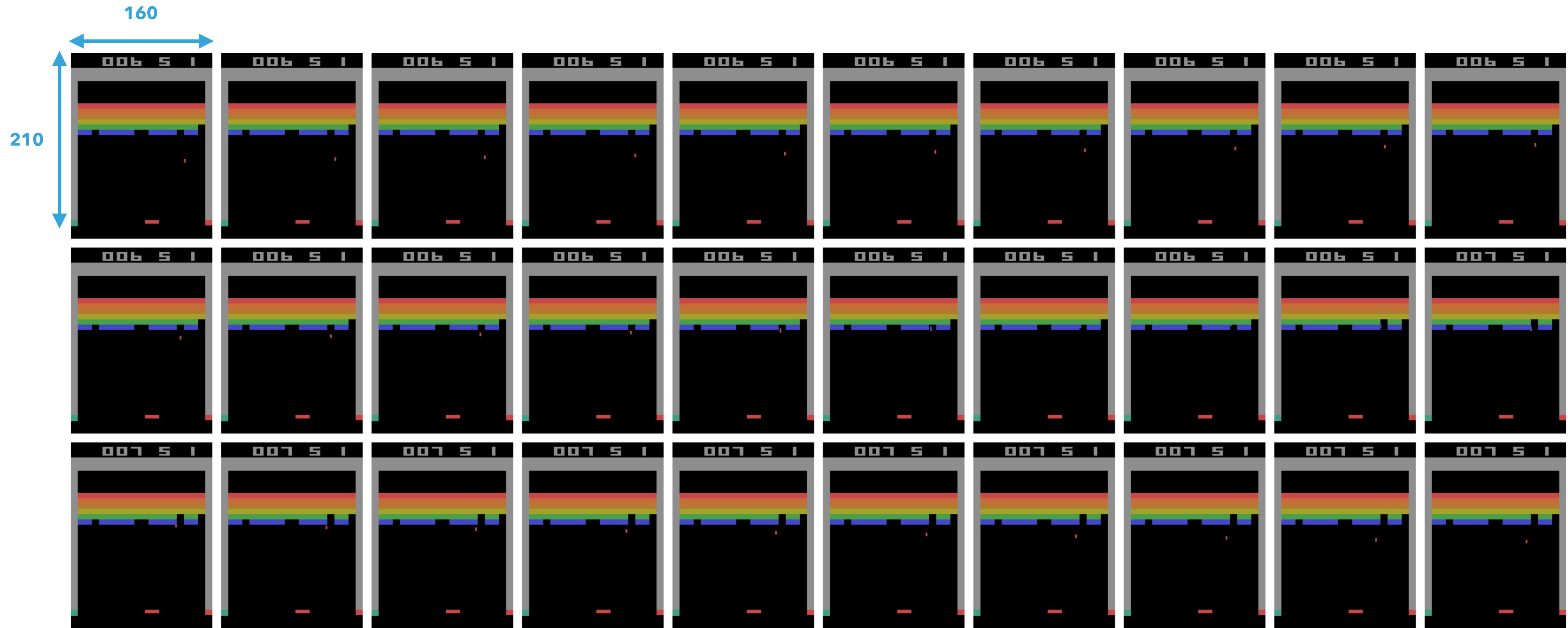


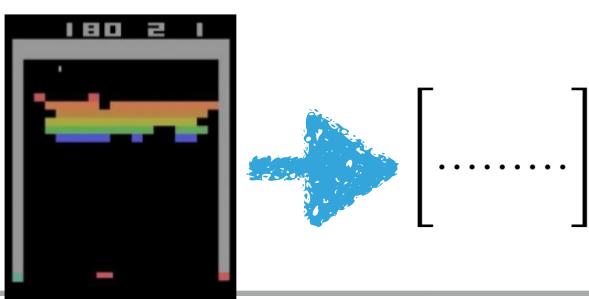
$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[\frac{\text{TD error}}{\text{Target value}} - \frac{\text{Estimated value}}{\text{Target value}} \right]^2$$

The equation shows the loss function $L_i(\theta_i)$ for a neural network parameter θ_i . The TD error is calculated as the difference between the target value (red) and the estimated value (blue). The target value is the sum of the immediate reward r and the discounted maximum Q-value from the next state s' for all actions a' . The estimated value is the current Q-value for state s and action a .

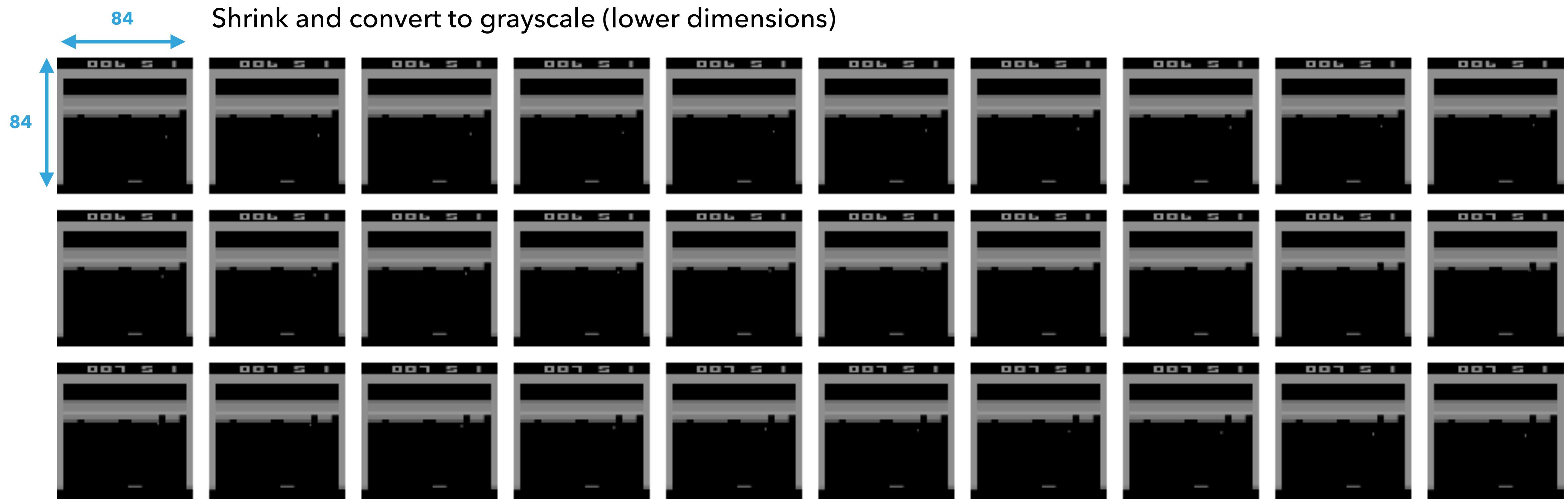


DQN - PREPROCESSING





DQN - PREPROCESSING





DQN - PREPROCESSING

Frame skipping

Parameter: 4

[Input]: $(84 \times 84 \times 4)$ $s_1 = (x_1, x_2, x_3, x_4)$
 $s_2 = (x_2, x_3, x_4, x_5)$
⋮

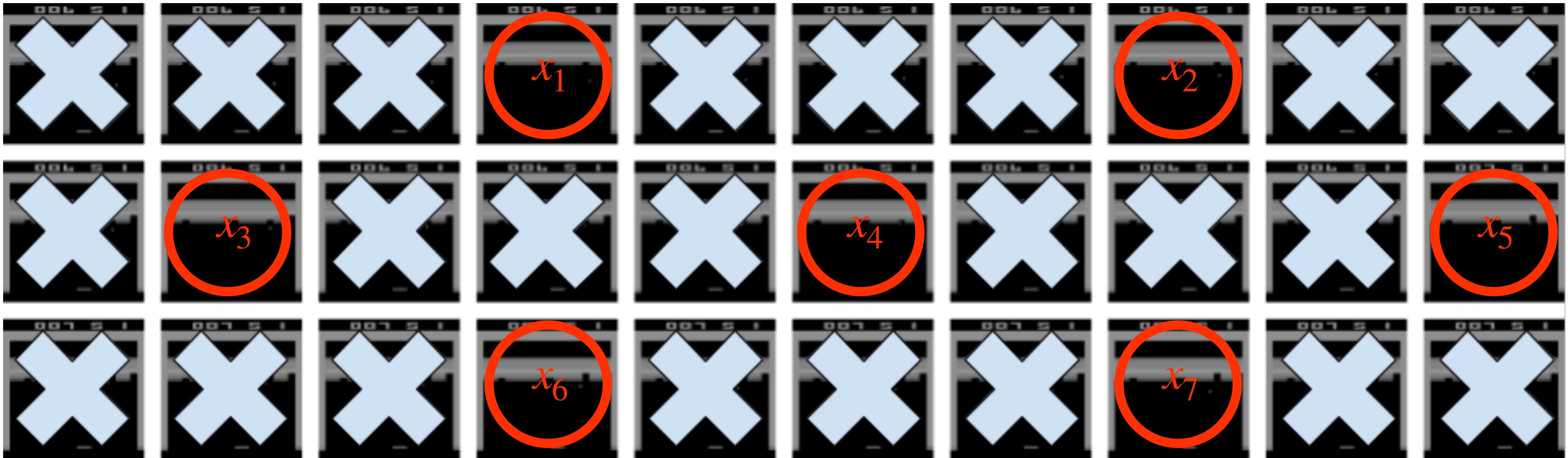


DQN - PREPROCESSING

Frame skipping

Parameter: 4

[Input]: $(84 \times 84 \times 4)$ $s_1 = (x_1, x_2, x_3, x_4)$
 $s_2 = (x_2, x_3, x_4, x_5)$
⋮

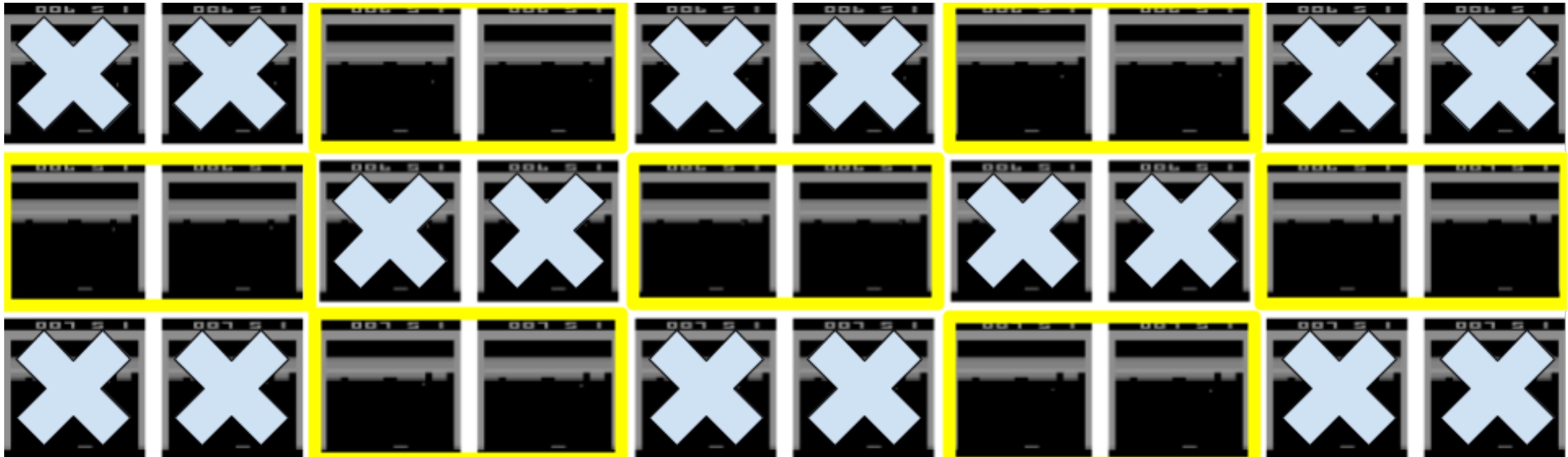


What if states are overlapped?

DQN - PREPROCESSING

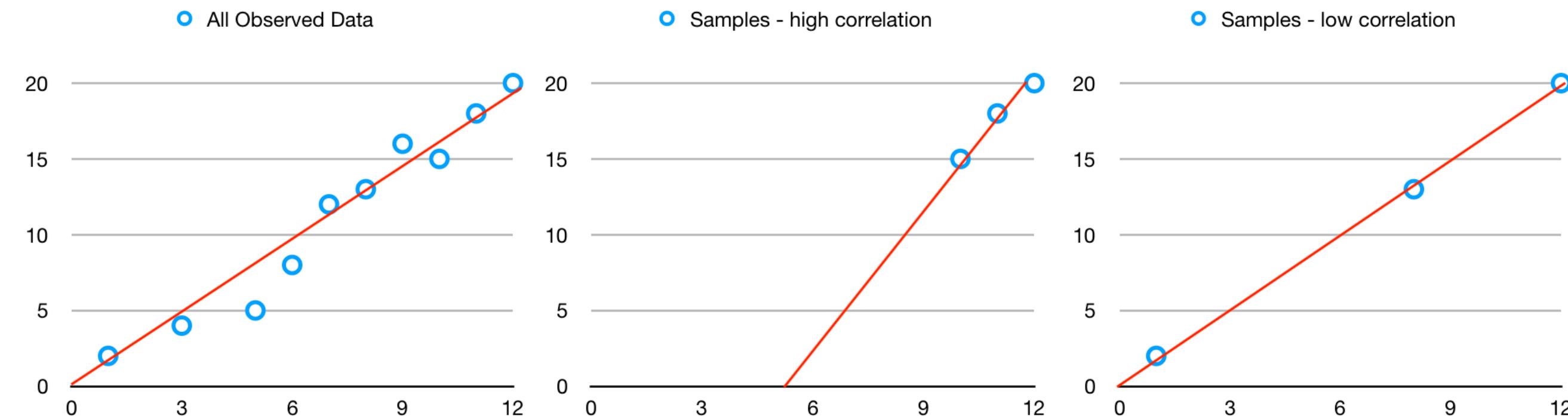
Frame skipping

DeepMind took the component-wise maximum over two consecutive frames (Atari setting)



PROBLEM

- ▶ Unstable update
 - ▶ High correlations of incoming data



- ▶ Non-stationary targets (same network parameters)

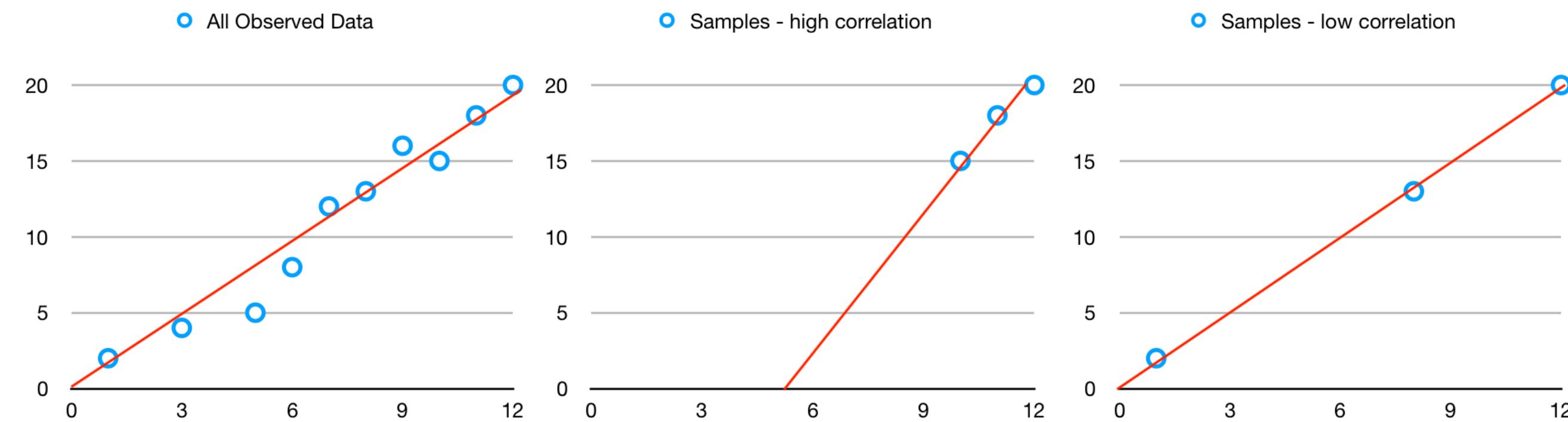
$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[(r + \gamma \max_a Q(s', a'; \theta_i) - Q(s, a; \theta_i))^2 \right]$$

[Objective function]

PROBLEM

- ▶ Unstable update

- ▶ High correlations of incoming data → Experience replay



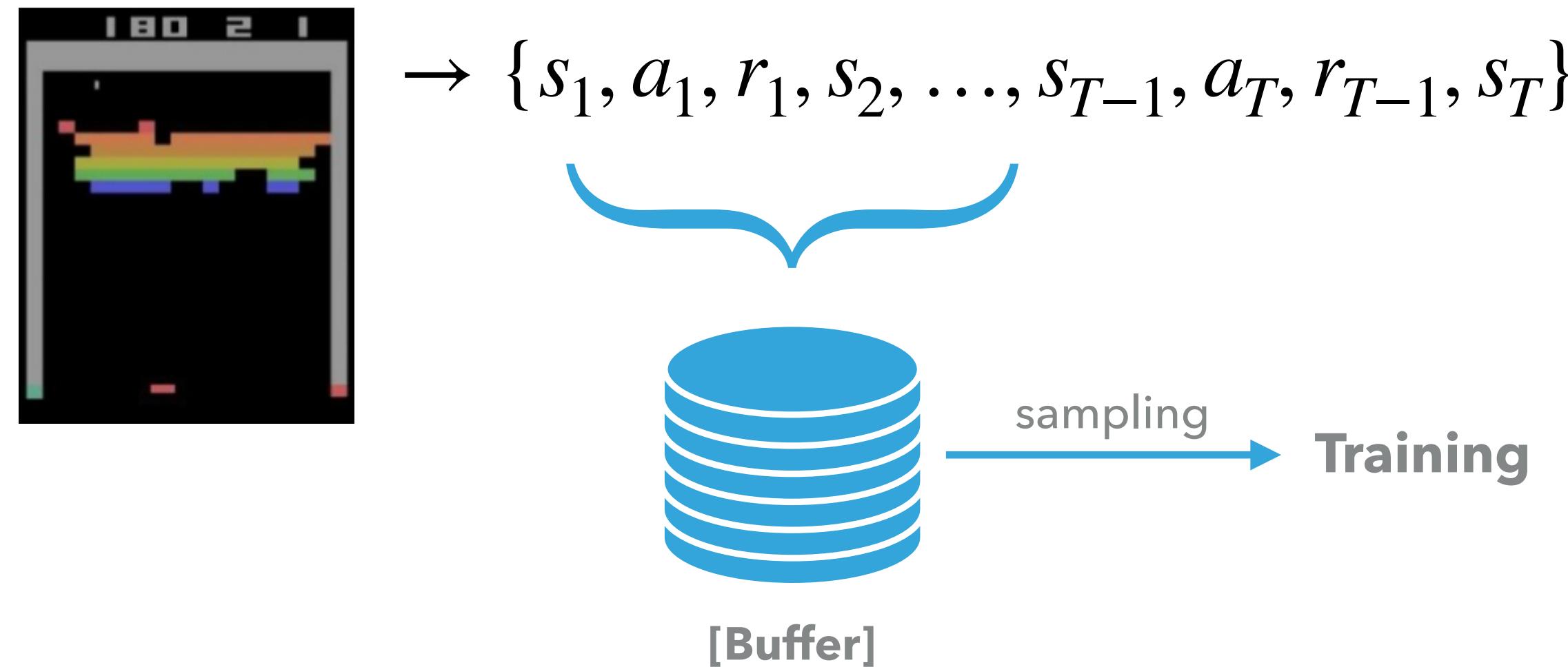
- ▶ Non-stationary targets (same network parameters) → Fixed targets

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[(r + \gamma \max_a Q(s', a'; \theta_i) - Q(s, a; \theta_i))^2 \right]$$

[Objective function]

TRICKS

► Experience replay

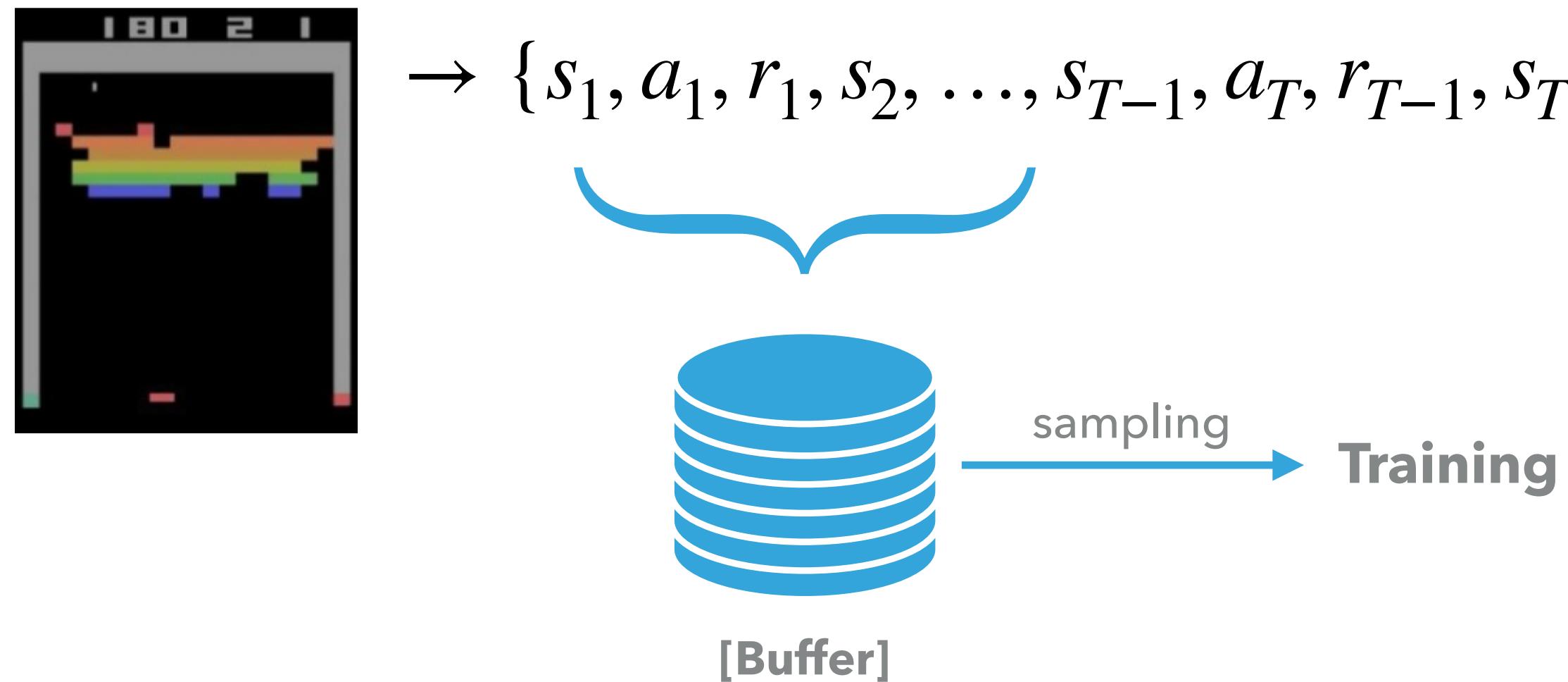


Matiisen, Tambet Demystifying Deep Reinforcement Learning. *Computational Neuroscience LAB*. 2015.

Mnih, V. et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

TRICKS

► Experience replay



► Fixed Q-targets

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[\left(r + \gamma \max_a Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

↓

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[\left(r + \gamma \max_a \hat{Q}(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

[Objective function]

IMPLEMENTATION

Use Cuda

```
USE_CUDA = torch.cuda.is_available()
Variable = lambda *args, **kwargs: autograd.Variable(*args, **kwargs).cuda() if USE_CUDA else autograd.Variable(*args, **kwargs)
```

Atari Environment

```
from common.wrappers import make_atari, wrap_deepmind, wrap_pytorch
```

```
env_id = "PongNoFrameskip-v4"
env = make_atari(env_id)
env = wrap_deepmind(env)
env = wrap_pytorch(env)
```

IMPLEMENTATION

Replay Buffer

```
from collections import deque

class ReplayBuffer(object):
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        state      = np.expand_dims(state, 0)
        next_state = np.expand_dims(next_state, 0)

        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        state, action, reward, next_state, done = zip(*random.sample(self.buffer, batch_size))
        return np.concatenate(state), action, reward, np.concatenate(next_state), done

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

IMPLEMENTATION

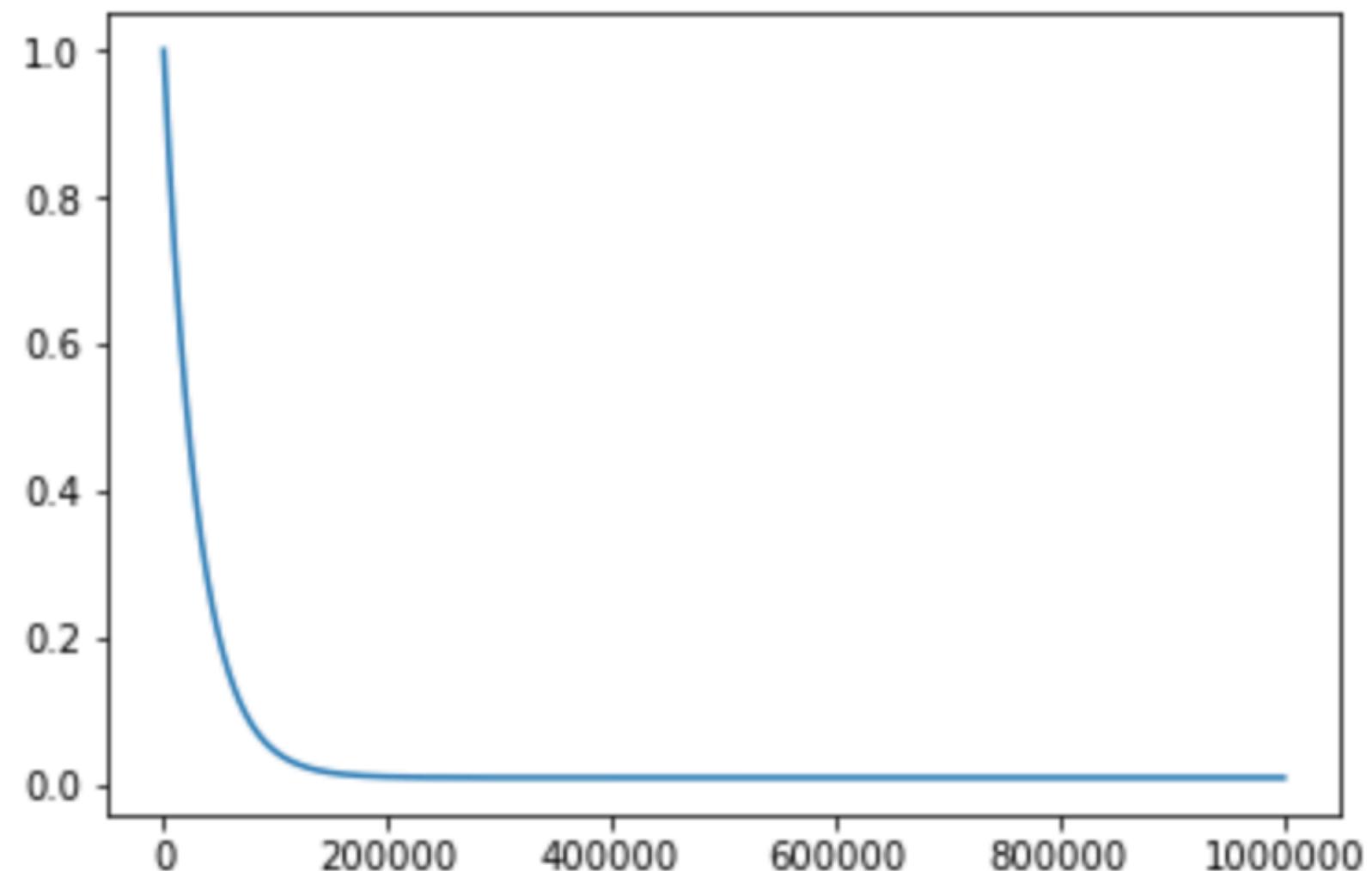
Epsilon greedy exploration

```
epsilon_start = 1.0
epsilon_final = 0.01
epsilon_decay = 30000

epsilon_by_frame = lambda frame_idx: epsilon_final + (epsilon_start - epsilon_final) * math.exp(-
1. * frame_idx / epsilon_decay)
```

```
plt.plot([epsilon_by_frame(i) for i in range(1000000)])
```

```
[<matplotlib.lines.Line2D at 0x7f73393b64d0>]
```



IMPLEMENTATION

```
class CnnDQN(nn.Module):
    def __init__(self, input_shape, num_actions):
        super(CnnDQN, self).__init__()

        self.input_shape = input_shape
        self.num_actions = num_actions

        self.features = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        self.fc = nn.Sequential(
            nn.Linear(self.feature_size(), 512),
            nn.ReLU(),
            nn.Linear(512, self.num_actions)
        )
```

IMPLEMENTATION

```
def forward(self, x):
    x = self.features(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

def feature_size(self):
    return self.features(autograd.Variable(torch.zeros(1, *self.input_shape))).view(1, -1).size(1)

def act(self, state, epsilon):
    if random.random() > epsilon:
        state = Variable(torch.FloatTensor(np.float32(state)).unsqueeze(0), volatile=True)
        q_value = self.forward(state)
        action = q_value.max(1)[1].data[0]
    else:
        action = random.randrange(env.action_space.n)
    return action
```

IMPLEMENTATION

```
model = CnnDQN(env.observation_space.shape, env.action_space.n)

if USE_CUDA:
    model = model.cuda()

optimizer = optim.Adam(model.parameters(), lr=0.00001)

replay_initial = 10000
replay_buffer = ReplayBuffer(100000)
```

IMPLEMENTATION

Computing Temporal Difference Loss

```
def compute_td_loss(batch_size):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state      = Variable(torch.FloatTensor(np.float32(state)))
    next_state = Variable(torch.FloatTensor(np.float32(next_state)), volatile=True)
    action     = Variable(torch.LongTensor(action))
    reward     = Variable(torch.FloatTensor(reward))
    done       = Variable(torch.FloatTensor(done))

    q_values      = model(state)
    next_q_values = model(next_state)

    q_value        = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value   = next_q_values.max(1)[0]
    expected_q_value = reward + gamma * next_q_value * (1 - done)

    loss = (q_value - Variable(expected_q_value.data)).pow(2).mean()

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

    return loss
```

IMPLEMENTATION

Training

```
num_frames = 1400000
batch_size = 32
gamma      = 0.99

losses = []
all_rewards = []
episode_reward = 0

state = env.reset()
for frame_idx in range(1, num_frames + 1):
    epsilon = epsilon_by_frame(frame_idx)
    action = model.act(state, epsilon)

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

    state = next_state
    episode_reward += reward

    if done:
        state = env.reset()
        all_rewards.append(episode_reward)
        episode_reward = 0

    if len(replay_buffer) > replay_initial:
        loss = compute_td_loss(batch_size)
        losses.append(loss.data[0])

    if frame_idx % 10000 == 0:
        plot(frame_idx, all_rewards, losses)
```

IMPLEMENTATION

 Training

```
num_frames = 1400000
batch_size = 32
gamma      = 0.99

losses = []
all_rewards = []
episode_reward = 0

state = env.reset()
for frame_idx in range(1, num_frames + 1):
    epsilon = epsilon_by_frame(frame_idx)
    action = model.act(state, epsilon)

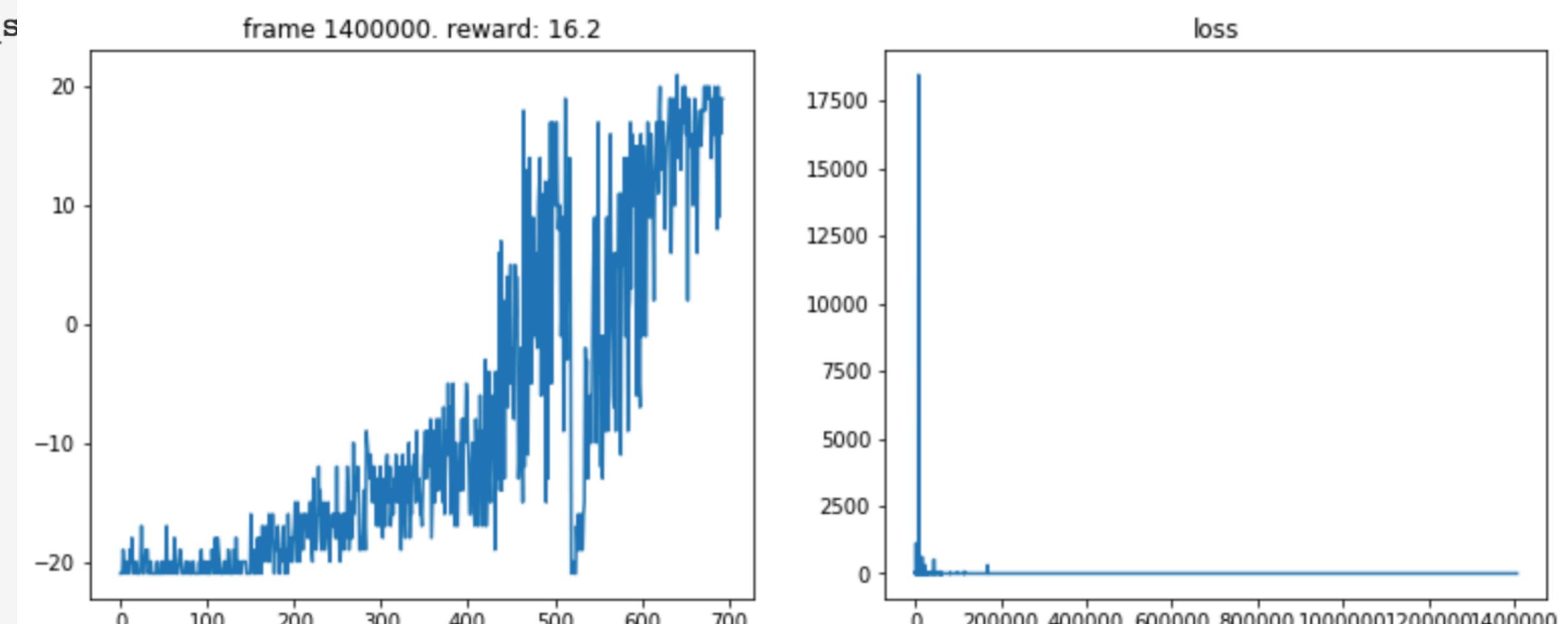
    next_state, reward, done, _ = env.step(action)
    replay_buffer.push(state, action, reward, next_s

    state = next_state
    episode_reward += reward

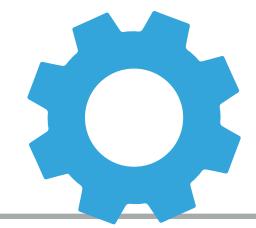
    if done:
        state = env.reset()
        all_rewards.append(episode_reward)
        episode_reward = 0

    if len(replay_buffer) > replay_initial:
        loss = compute_td_loss(batch_size)
        losses.append(loss.data[0])

    if frame_idx % 10000 == 0:
        plot(frame_idx, all_rewards, losses)
```



DOUBLE DQN

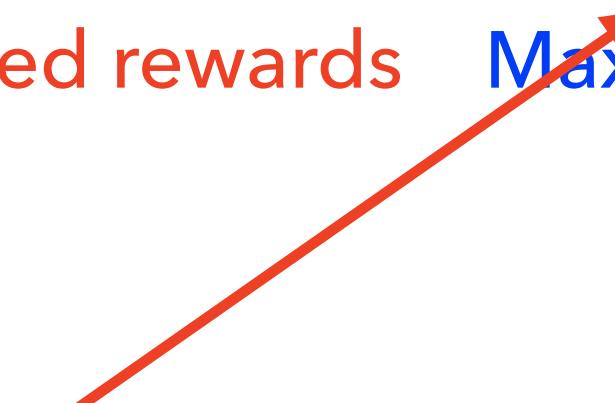


MOTIVATION

- ▶ Problems of DQN:

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a)$$

Q-target Accumulated rewards Maximum Q-value of next state



- ▶ Solution:

$$Q(s, a) = r(s, a) + \gamma Q(s', \operatorname{argmax}_a Q(s', a))$$

DQN Network choose action for next state

IMPLEMENTATION

```
current_model = CnnDQN(env.observation_space.shape, env.action_space.n)
target_model = CnnDQN(env.observation_space.shape, env.action_space.n)

if USE_CUDA:
    current_model = current_model.cuda()
    target_model = target_model.cuda()

optimizer = optim.Adam(current_model.parameters(), lr=0.0001)

replay_initial = 10000
replay_buffer = ReplayBuffer(100000)
```

Synchronize current policy net and target net

```
def update_target(current_model, target_model):
    target_model.load_state_dict(current_model.state_dict())
update_target(current_model, target_model)
```

IMPLEMENTATION Computing Temporal Difference Loss

```
def compute_td_loss(batch_size):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state      = Variable(torch.FloatTensor(np.float32(state)))
    next_state = Variable(torch.FloatTensor(np.float32(next_state)))
    action     = Variable(torch.LongTensor(action))
    reward     = Variable(torch.FloatTensor(reward))
    done       = Variable(torch.FloatTensor(done))

    q_values      = current_model(state)
    next_q_values = current_model(next_state)
    next_q_state_values = target_model(next_state)

    q_value      = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value = next_q_state_values.gather(1, torch.max(next_q_values, 1)[1].unsqueeze(1)).squeeze(1)
    expected_q_value = reward + gamma * next_q_value * (1 - done)

    loss = (q_value - Variable(expected_q_value.data)).pow(2).mean()

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

    return loss
```

IMPLEMENTATION Computing Temporal Difference Loss

```
def compute_td_loss(batch_size):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

    state      = Variable(torch.FloatTensor(np.float32(state)))
    next_state = Variable(torch.FloatTensor(np.float32(next_state)))
    action     = Variable(torch.LongTensor(action))
    reward     = Variable(torch.FloatTensor(reward))
    done       = Variable(torch.FloatTensor(done))

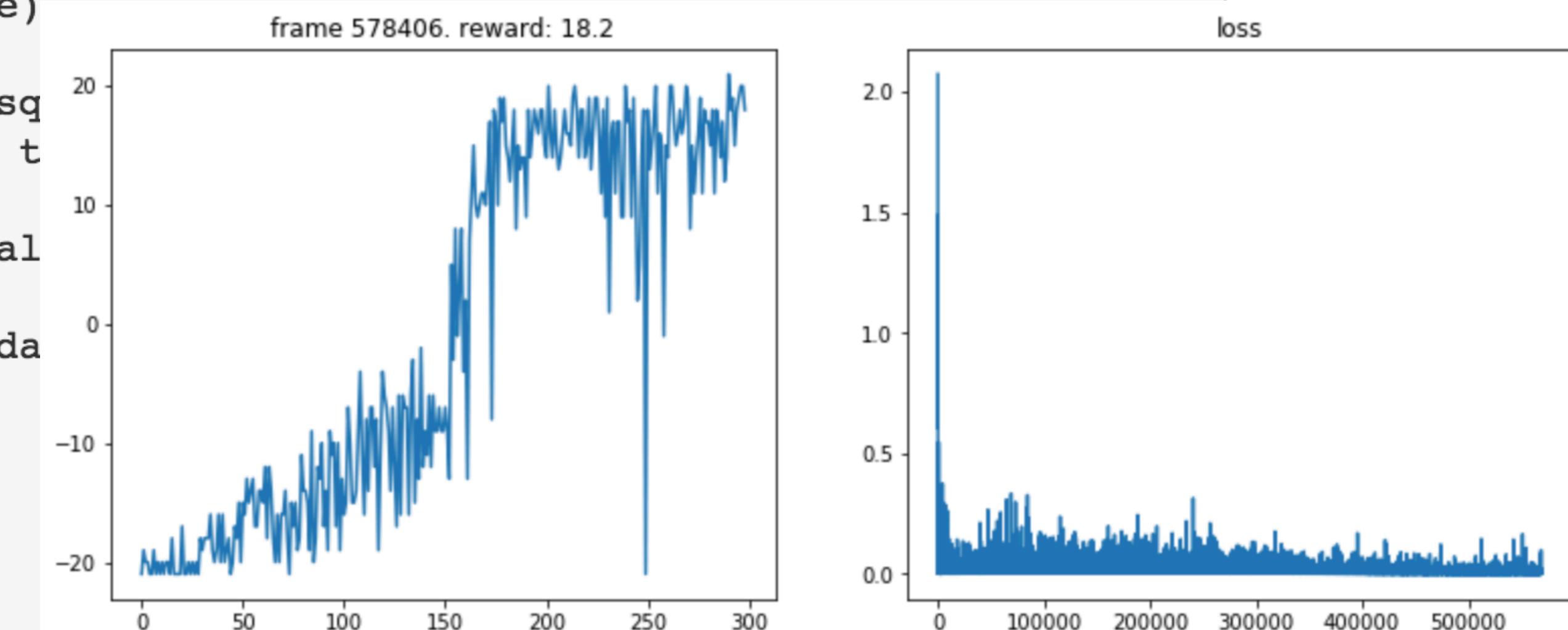
    q_values      = current_model(state)
    next_q_values = current_model(next_state)
    next_q_state_values = target_model(next_state)

    q_value      = q_values.gather(1, action.unsqueeze(1))
    next_q_value = next_q_state_values.gather(1, action.unsqueeze(1))
    expected_q_value = reward + gamma * next_q_value

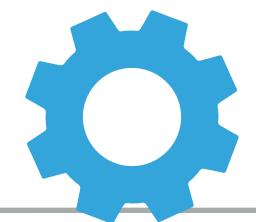
    loss = (q_value - Variable(expected_q_value.data))

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

    return loss
```



DUELING DQN



MOTIVATION

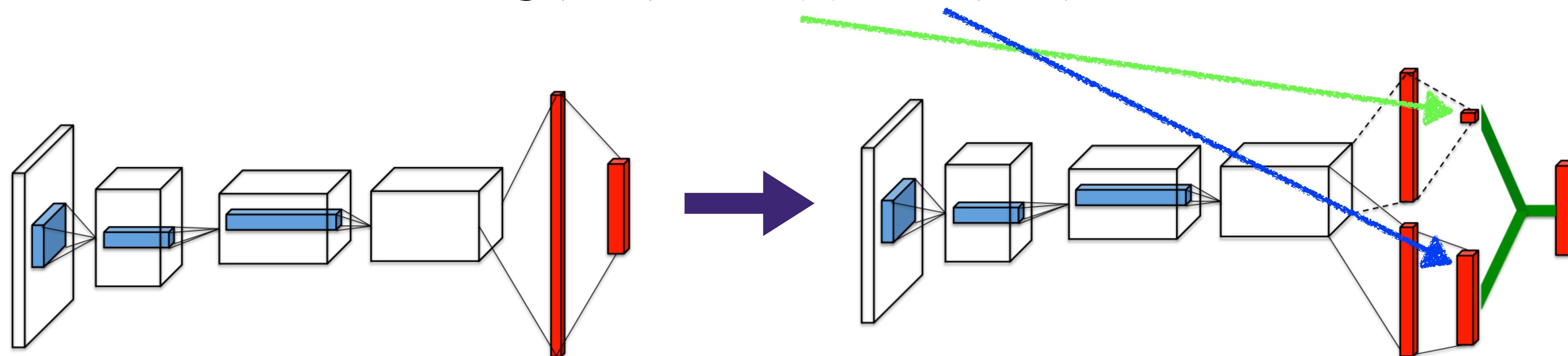
[Q-value decomposition]

The state value

$$Q(s, a) = V(s) + A(s, a)$$

Advantage value:

*How much better is to take this action versus
all other possible actions at that state*



- DDQN can learn which states are (or are not) valuable **without** having to learn the effect of each action at each state (since it's also calculating $V(s)$)

IMPLEMENTATION

```
class DuelingCnnDQN(nn.Module):
    def __init__(self, input_shape, num_outputs):
        super(DuelingCnnDQN, self).__init__()

        self.input_shape = input_shape
        self.num_actions = num_outputs

        self.features = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        self.advantage = nn.Sequential(
            nn.Linear(self.feature_size(), 512),
            nn.ReLU(),
            nn.Linear(512, num_outputs)
        )

        self.value = nn.Sequential(
            nn.Linear(self.feature_size(), 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )
```

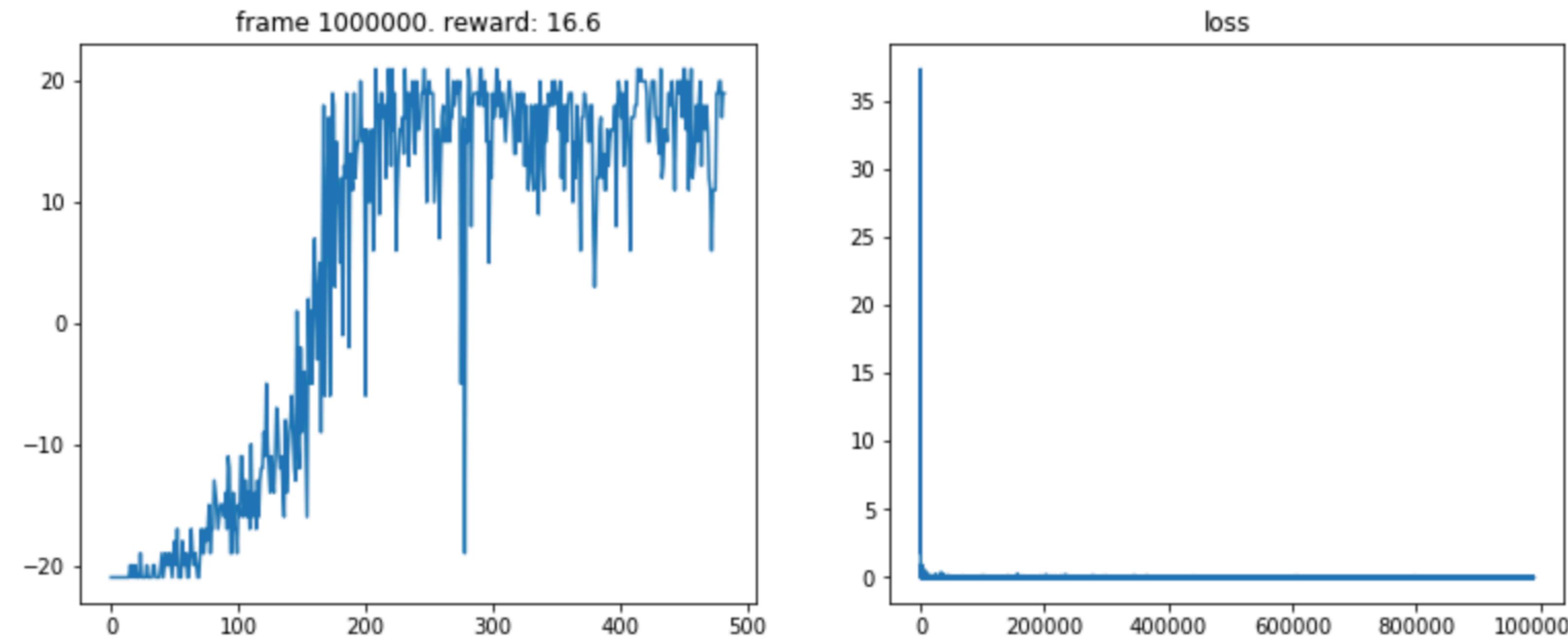
IMPLEMENTATION

```
def forward(self, x):
    x = self.features(x)
    x = x.view(x.size(0), -1)
    advantage = self.advantage(x)
    value      = self.value(x)
    return value + advantage - advantage.mean()

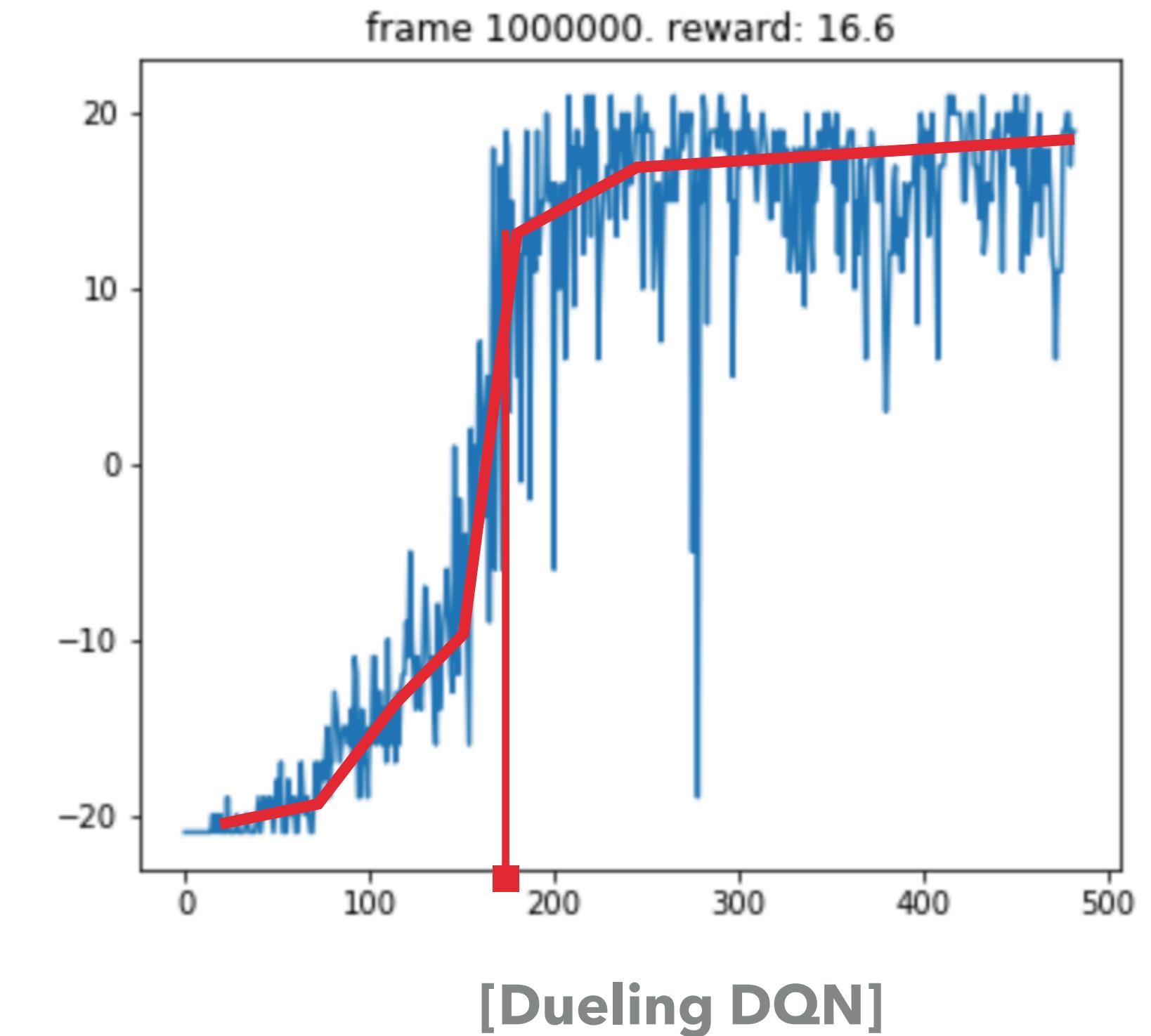
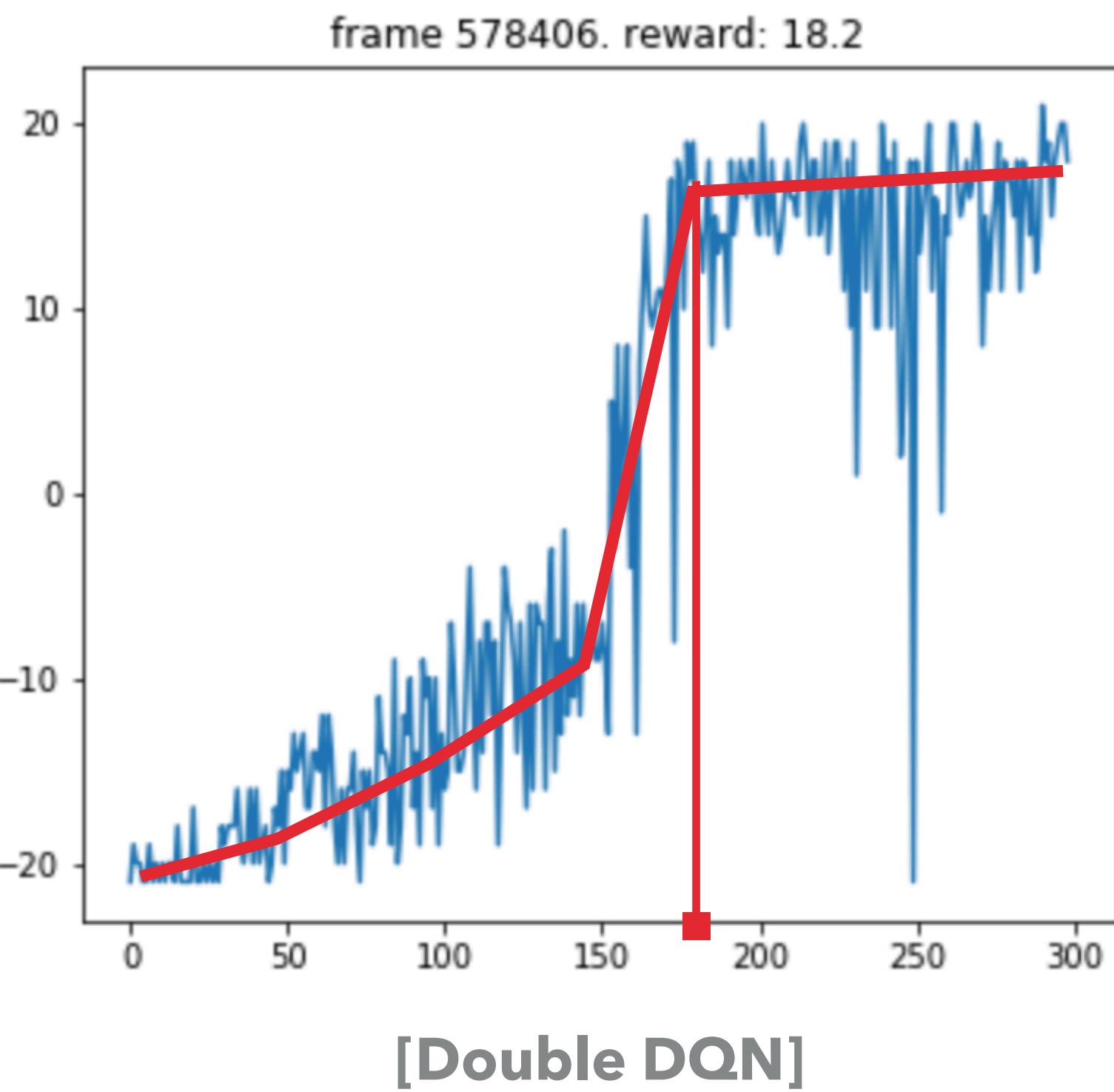
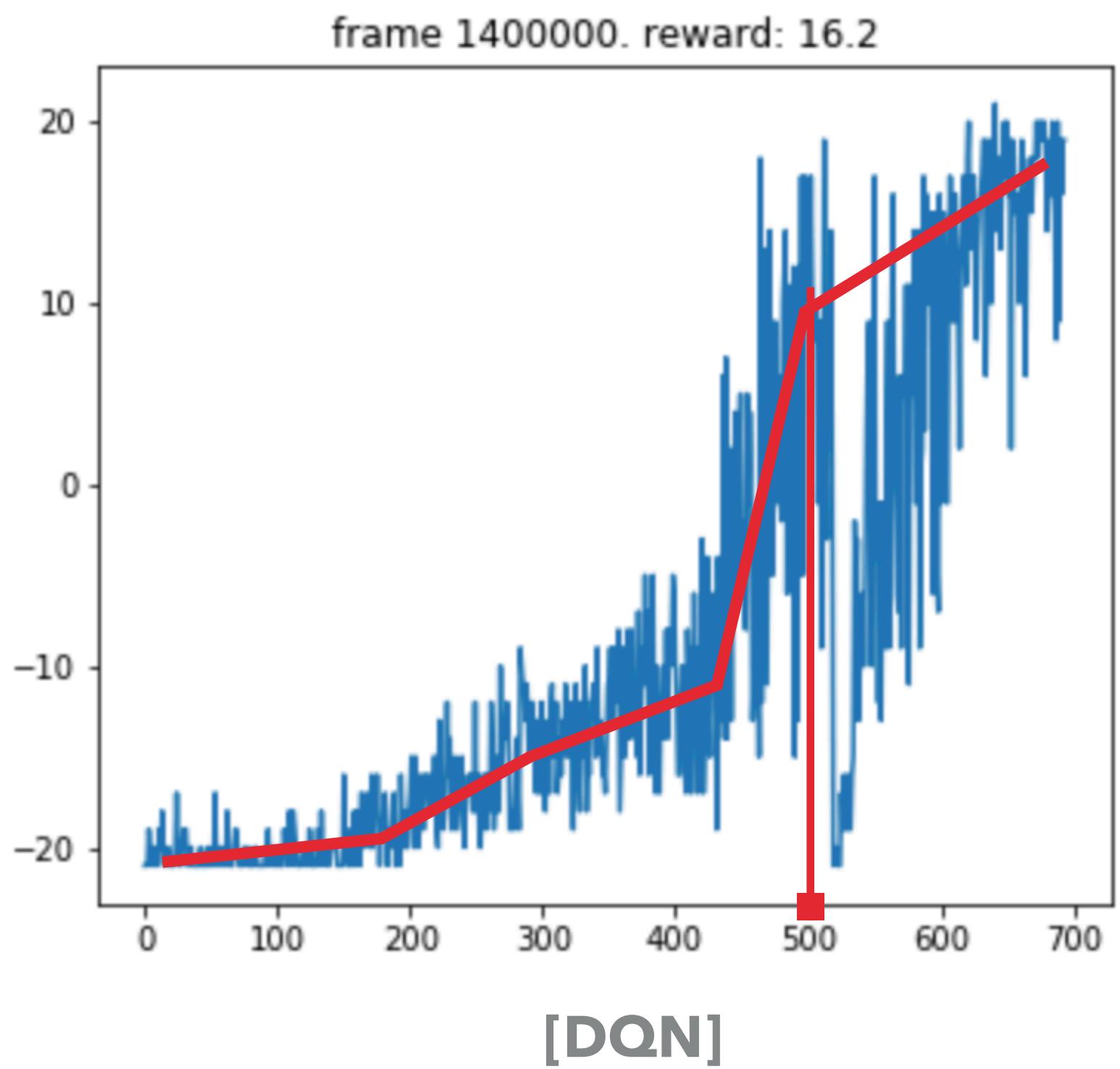
def feature_size(self):
    return self.features(autograd.Variable(torch.zeros(1, *self.input_shape))).view(1, -1).size(1)

def act(self, state, epsilon):
    if random.random() > epsilon:
        state      = Variable(torch.FloatTensor(np.float32(state)).unsqueeze(0), volatile=True)
        q_value   = self.forward(state)
        action    = q_value.max(1)[1].data[0]
    else:
        action = random.randrange(env.action_space.n)
    return action
```

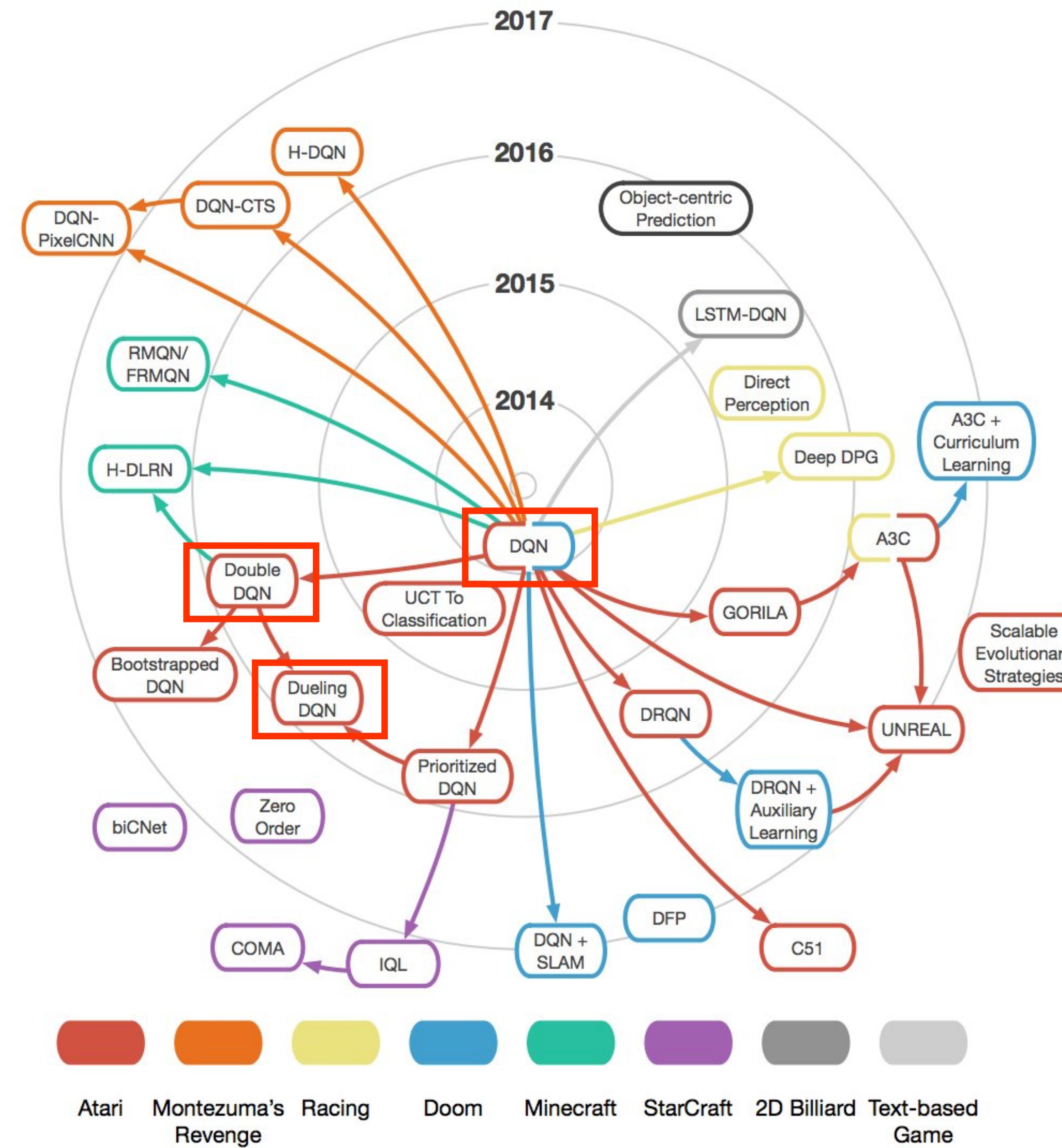
IMPLEMENTATION



EXPERIMENTAL COMPARISON



DQN LINEAGE



감사합니다