10강.DQN Variants

Contents

- Double DQN
- Prioritized Experience Replay(PER)
- Dueling DQN
- Code Ex.

Double DQN

Overview

- Q-learning overestimates action-value:
  - Overestimation happens in the earlier stage of training (when action-value is not accurate)
  - Overestimation may help exploration, but
  - Overestimation may be non-uniform throughout all action (Change optimal action accidentally)

- DQN is degraded by Overestimation problem (sub-optimal policy)

Problem: Overestimation

- Q-learning updates using the target $r + \gamma \max_{a'} Q(s'. a')$
  - Note both argmax action a' and Q value are computed by the same action value function

- Overestimated action-values are accumulated:
  - $\begin{cases} Q(s_t, a_t) \text{ <- } Q(s_{t+1}, a_{t+1}), \text{ <- } \ldots \text{ <- } Q(s_{T-1}, a_{T-1}) \\ a_k = \mathrm{argmax}_{a_k} Q(s_k, a_k) \end{cases}$

Solution: Double Q

- Double Q-learning solves overestimation by separating
  - $\begin{cases} \text{the action}-\text{value function in getting the target} \\ \text{The action}-\text{value function in getting argmax action} \end{cases}$
  - $y_t^Q = r_{t+1} + \gamma Q(s_{t+1}, \underset{a_{t+1}}{\operatorname{argmax}} Q(s_{t+1}, a_{t+1}; \theta_t); \theta_t)$

  ➔ $y_t^Q = r_{t+1} + \gamma Q(s_{t+1}, \underset{a_{t+1}}{\operatorname{argmax}} Q(s_{t+1}, a_{t+1}; \theta_t); \theta_t')$

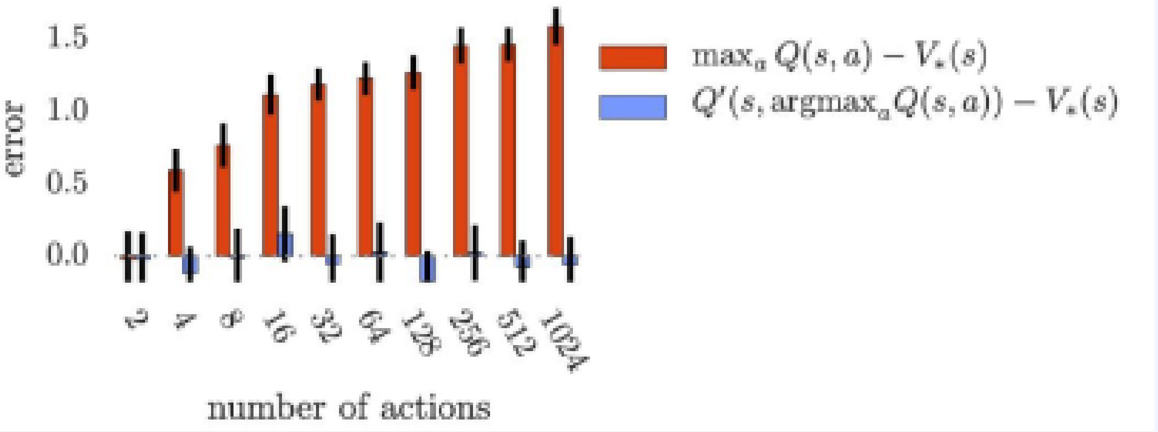
- Even if θ_t overestimates (overestimated a_{t+1}),
  there's no overestimation if θ_t' does not overestimate at a_{t+1}
- θ_t , θ_t' must be statistically independent

Solution: Double Q

- Overestimation get worse as the number of actions increase:

$$\gamma\epsilon\frac{m-1}{m+1}$$

where m is the number of actions, uniform random noise $[-\varepsilon, \varepsilon]$

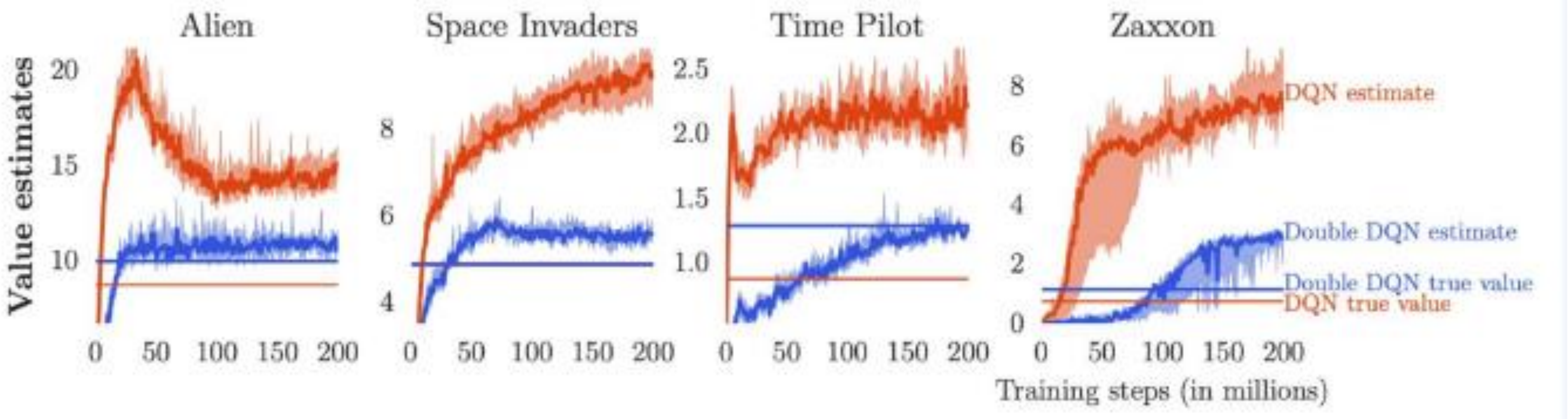


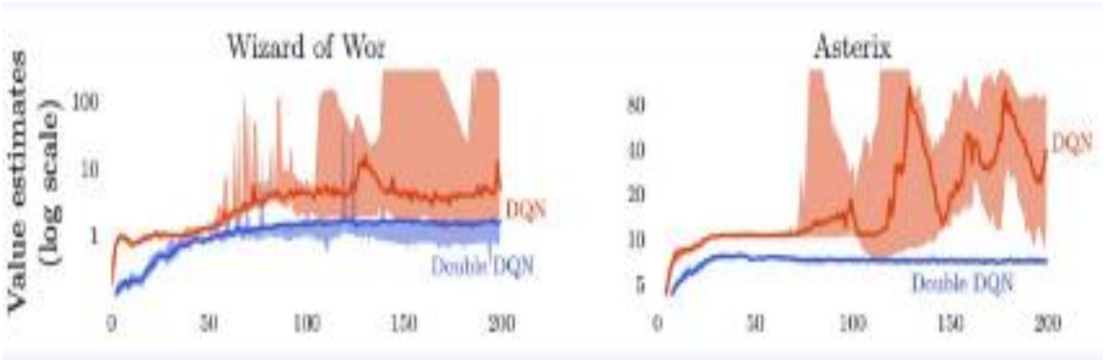
- Reusing the target network for θ_t' will reduce the resource:

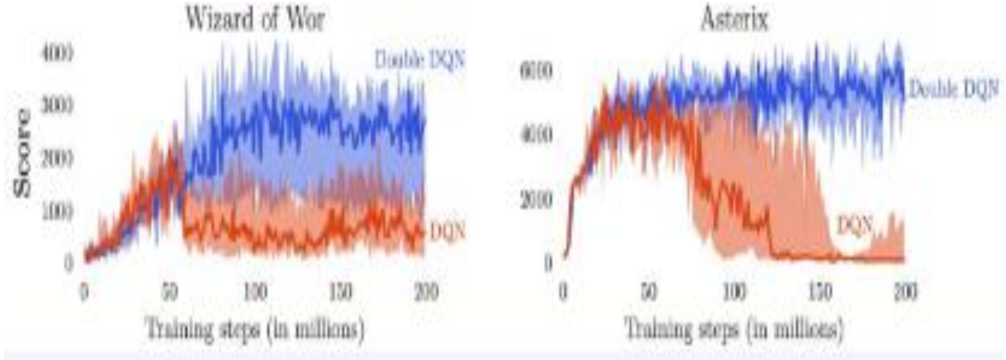
$$y_t^{\text{Double DQN}} = r_{t+1} + \gamma Q(s_{t+1}, \operatorname*{argmax}_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_t); \theta_t^-)$$

θ_t^-: parameter of the target network (past θ_t)

Results


Alien
Space Invaders
Time Pilot
Zaxxon
Value estimates
DQN estimate
Double DQN estimate
Double DQN true value
DQN true value
Training steps (in millions)


Wizard of Wor
Asterix
Value estimates
(log scale)
DQN
Double DQN


Wizard of Wor
Asterix
Score
Double DQN
DQN
Training steps (in millions)

Results

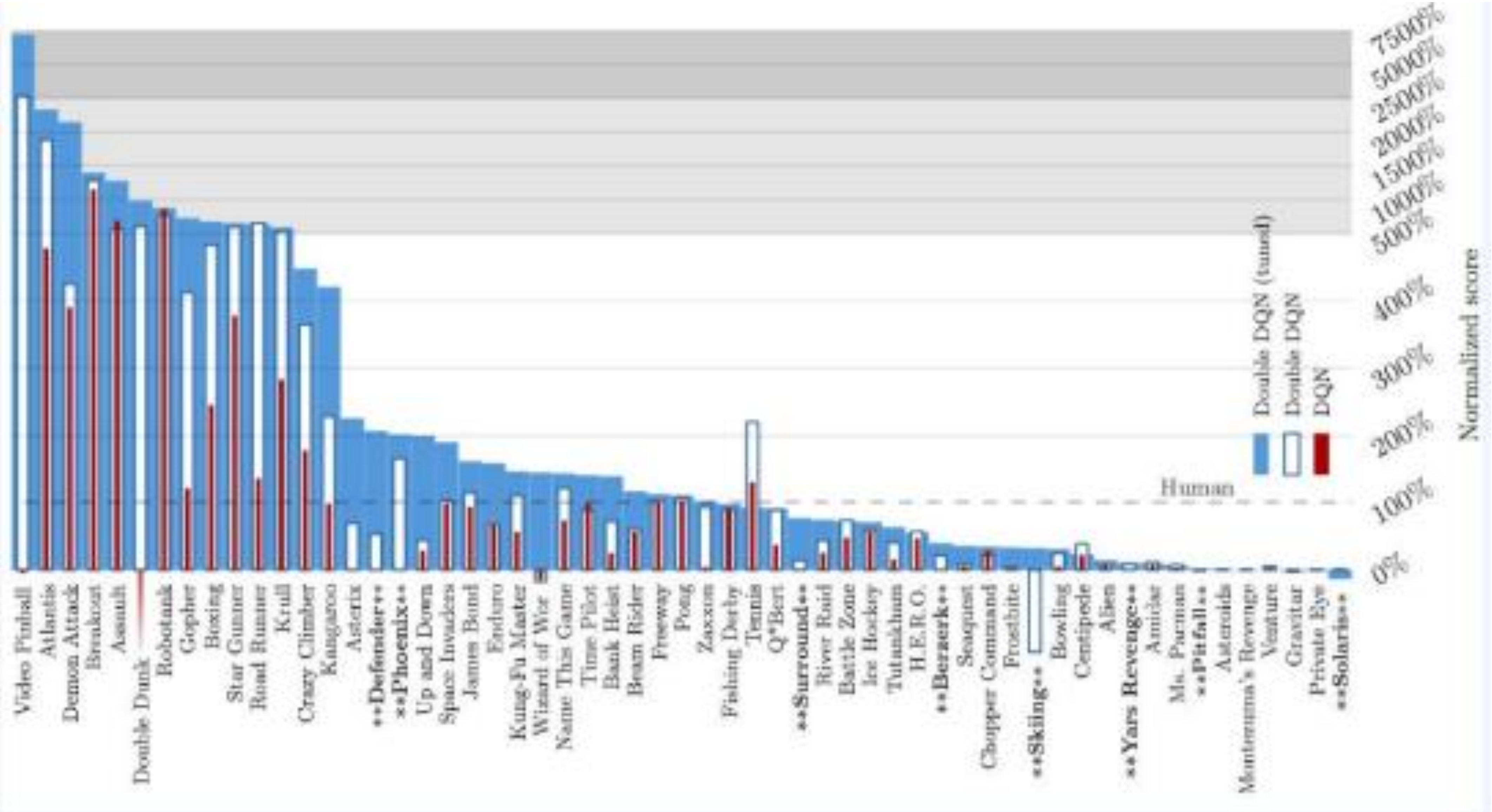
- Mean and media performance comparison over 49 games starting after several no-ops:

	DQN	Double DQN
Median	93.5%	114.7%
Mean	241.1%	330.3%

- Mean and media performance comparison over 49 games with agent starting after human playing:

	DQN	Double DQN	Double DQN (tuned)
Median	47.5%	88.4%	116.7%
Mean	122.0%	273.1%	475.2%

Results



7500%
5000%
2500%
2000%
1500%
1000%
500%
400%
300%
200%
100%
0%
Normalized score
Double DQN (tuned)
Double DQN
DQN
Human
Video Pinball
Atlantis
Demon Attack
Breakout
Assault
Double Dunk
Robotank
Gopher
Boxing
Star Gunner
Road Runner
Krull
Crazy Climber
Kangaroo
Asterix
Defender
Phoenix
Up and Down
Space Invaders
James Bond
Enduro
Kung-Fu Master
Wizard of Wor
Name This Game
Time Pilot
Bank Heist
Beam Rider
Freeway
Pong
Zaxxon
Fishing Derby
Tennis
Q*Bert
Surround
River Raid
Battle Zone
Ice Hockey
Tutankham
H.E.R.O.
Berzerk
Seaquest
Chopper Command
Frostbite
Skiing
Bowling
Centipede
Alien
Yars Revenge
Amidar
Ms. Pacman
Pitfall
Asteroids
Montezuma's Revenge
Venture
Gravitar
Private Eye
Solaris

Code

```
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)
```

Code

```
Qs = self.forward(states_tensor)  # (n_batch, n_action)
with torch.no_grad():
    next_Qs = self.forward(next_states_tensor)  # (n_batch, n_action)
next_target_Qs = self.forward_target_network(next_states_tensor)  # (n_batch, n_action)

# index dimension should be the same as the source tensor
chosen_Q = Qs.gather(dim=-1, index=actions_tensor.reshape(-1, 1)).reshape(-1)
next_argmax_actions = next_Qs.argmax(dim=-1).reshape(-1, 1)  # reshaping for gather
next_target_max_Q = next_target_Qs.gather(dim=-1, index=next_argmax_actions).reshape(-1)  # (n_batch)
target_Q = rewards_tensor + (1 - dones_tensor) * config.gamma * next_target_max_Q

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()
```

Prioritized Experience Replay(PER)

Overview

- Replay buffer:
  - Solves correlation of samples in an episode
  - Reduces data distribution shift caused by policy update
  - Reuses old data

- Problem:
  - Important samples (linked to the reward) deserves more replay
  - Well learnt (s, a) does not need more replaying
  - Uniform random mini-batch sampling is not always good

Overview

- Ordering samples by importance and taking top m samples may seriously harm data diversity

➔ Stochastic sampling take both importance and diversity into account

➔ the importance is measured by |TD error|

Method

- Sample data from replay buffer $P(i) = \frac{p_i{}^{\alpha}}{\sum_k p_k{}^{\alpha}}$
- p_i: $\mathrm{i_{th}}$ sample importance in replay memory ($\alpha \geq 0$)

- Importance:
  - TD error magnitude: $p_i = |\delta_i| + \varepsilon$
  - The order of TD error magnitude: $p_i = \frac{1}{rank(i)}$

Method

- Distribution shift correction:
  - PER presents the distribution shift 'again' every time when prioritized
  - This will cause 'bias' in terms of stochastic gradient descent

- Correct with Importance Sampling weight
- $\Delta\theta = \eta \sum_i \frac{1}{N} \nabla_\theta L_i$ -> $\Delta\theta = \eta \sum_i \frac{1}{N} \nabla_\theta L_i$ -> $\Delta\theta = \eta \sum_i \frac{1}{N} \omega i \nabla_\theta L_i$

  (uniform random) (Prioritized sampling) (prioritized sampling with bias correction)

  $\omega i = (\frac{1}{N} \frac{1}{P(i)})^\beta$

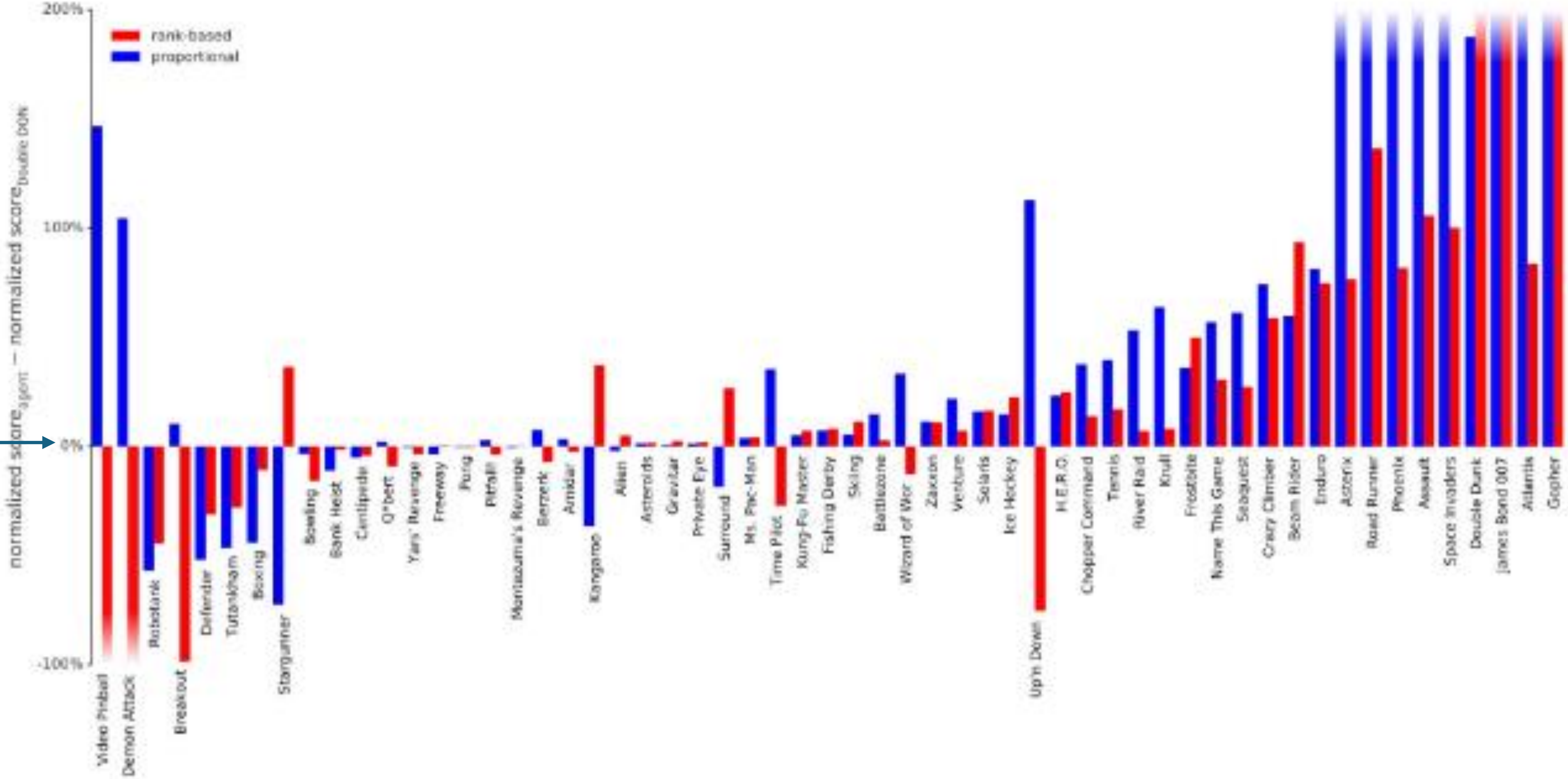
Results

- Mean and media performance comparison over 49 (or 57) games with agent starting after human playing:

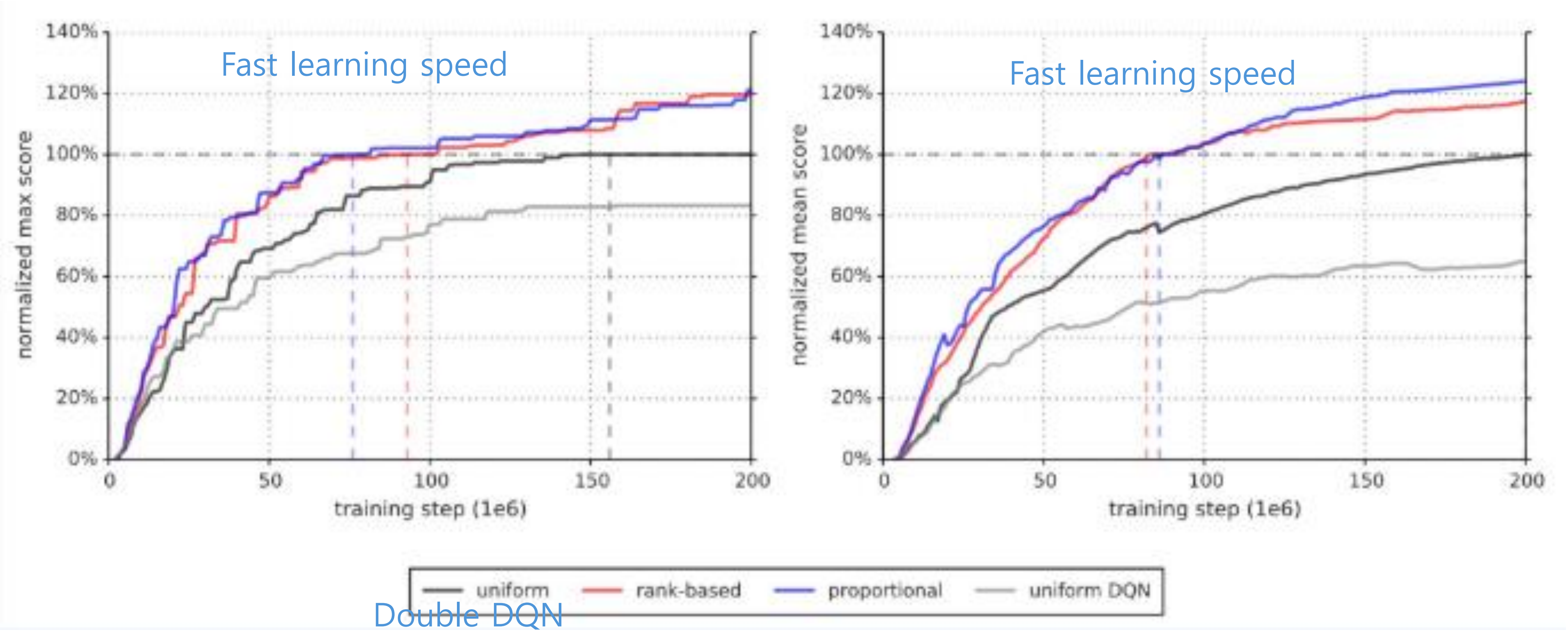
	DQN		Double DQN (tuned)		
	baseline	rank-based	baseline	rank-based	proportional
Median	48%	106%	111%	113%	128%
Mean	122%	355%	418%	454%	551%
> baseline	–	41	–	38	42
> human	15	25	30	33	33
# games	49	49	57	57	57

Table 1: Summary of normalized scores. See Table 6 in the appendix for full results.

Results



Double
DQN



Fast learning speed
Fast learning speed
normalized max score
normalized mean score
140%
120%
100%
80%
60%
40%
20%
0%
0
50
100
150
200
training step (1e6)
uniform
rank-based
proportional
uniform DQN
Double DQN

Code

```
class PrioritizedReplayMemory:
    def __init__(self, config):
        self.config = config
        self.buffer = deque([], maxlen=self.config.replay_capacity)
        self.abs_td_errors = np.ones(self.config.replay_capacity)
        self.alpha = config.alpha
        self.eps = config.eps_replay

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

    def append(self, transition):
        buffer_size = len(self.buffer)
        self.buffer.append(transition)
        if buffer_size == self.config.replay_capacity:
            abs_td_error_max = self.abs_td_errors[:-2].max()
            self.abs_td_errors[:-1] = self.abs_td_errors[1:]
            self.abs_td_errors[-1] = abs_td_error_max
```

One transition data must be sampled at least once

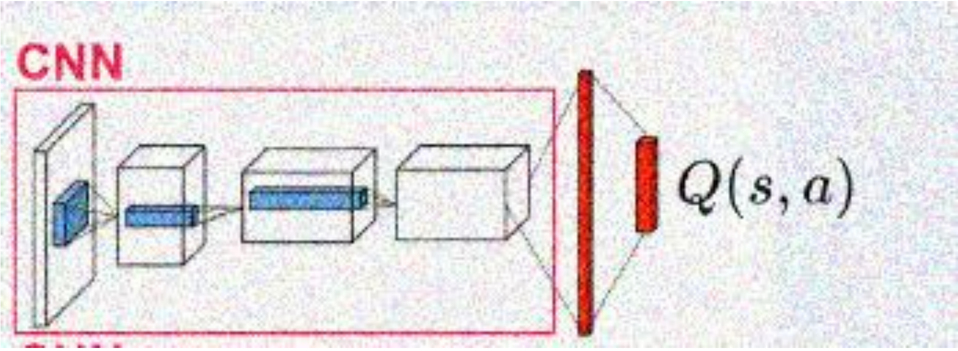
Once the buffer is full, a new sample input makes the td error array shifted, and assigned a max error value

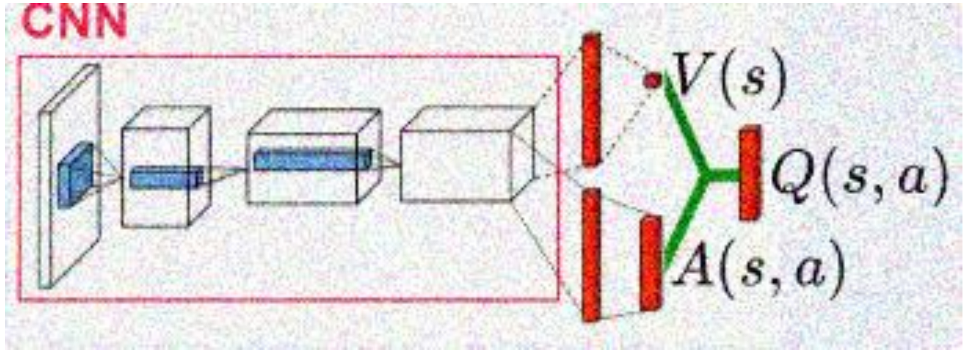
```
def sample(self, size):
    buffer_size = len(self.buffer)
    if buffer_size >= size:
        abs_td_errors = self.abs_td_errors[:buffer_size]  # get valid td errors

        if config.sampling_strategy == 'rank-based':
            ranks = abs_td_errors.argsort()[::-1]                index
            # 1 ~ buffer_size  a = [3,1,2] -> a.argsort() = [1, 2, 0]
            # [index_TD_0, index_TD_1, ..., index_TD_n-1]  ranks

            logits = 1 / np.arange(1, buffer_size + 1)  # [1/1, 1/2, ..., 1/buffer_size]  p_sample
            p_sample = np.power(logits, self.alpha)
            p_sample = p_sample / p_sample.sum()
            indices = np.random.choice(ranks, p=p_sample, size=config.batch_size)

        elif config.sampling_strategy == 'proportional':
            logits = abs_td_errors + self.eps
            p_sample = np.power(logits, self.alpha)
            p_sample = p_sample / p_sample.sum()
            indices = np.random.choice(np.arange(buffer_size), p=p_sample, size=config.batch_size)

         :

        samples = []
        for i in indices:
            samples.append(self.buffer[i])
        prob_samples = p_sample[indices]
```

Dueling DQN

Overview

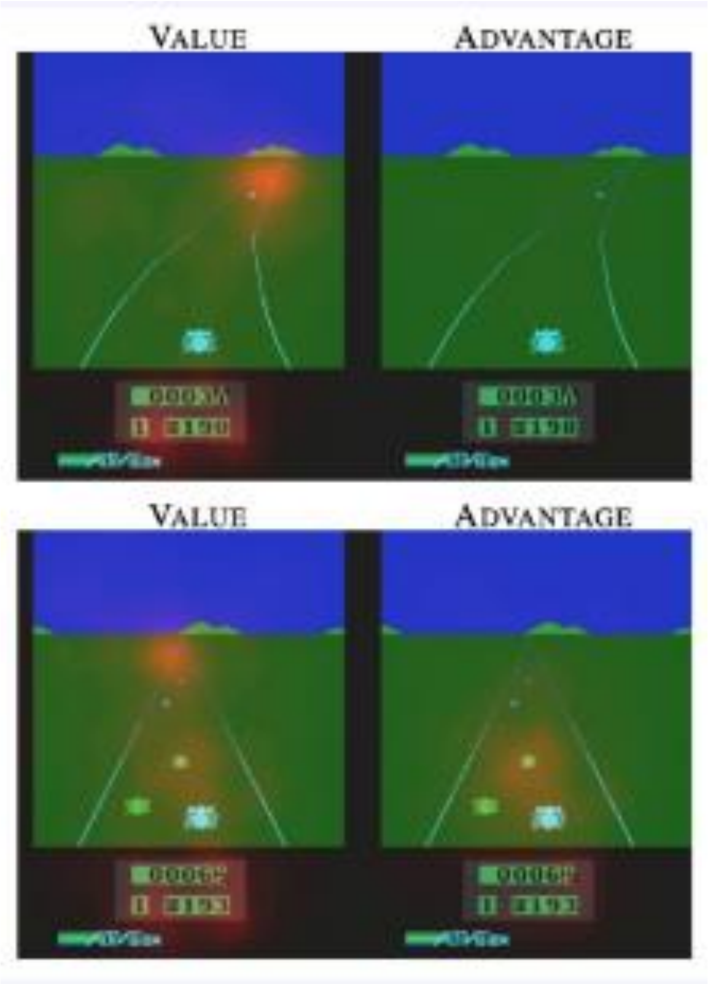
- Proposed a new NN architecture that fits RL
- NN that outputs both value and advantage function
- A(s, a) = Q(s, a) – V(s)





Representation that separates
the advantage from value

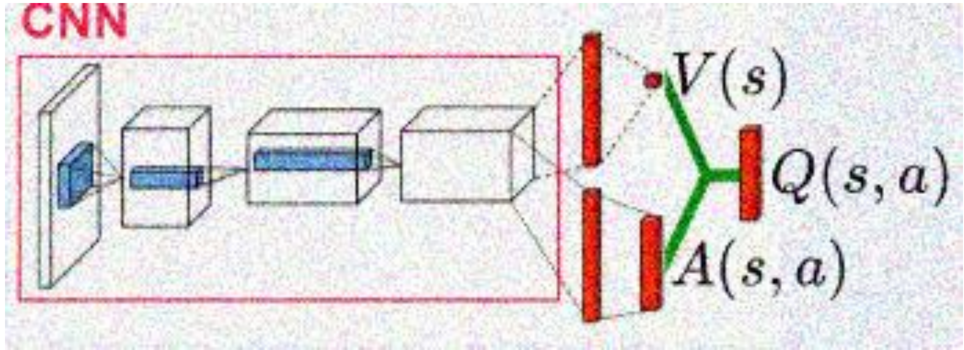
Overview



- Saliency map tells which part of the input has more affects on output
- Value network focuses on where a new car comes and the score
- Advantage network focuses one nearby in-front cars
  - It focuses nowhere if there’s no car around
  - No advantage!

Method

- Only actions in parts of states have bigger impact on return, therefore no need to calculate Q for every states.
- (e.g., state where a collision is probable)



- The architecture can be adapted Q-learning, SARSA, etc.,

Method

- Advantage conditions:

$$\begin{cases} \textit{mean advantage is } 0: \\ E_\pi[A(s,a)] = 0 \\ \\ \textit{the advantage of determinsitic policy}(e.g.\,\textit{argmax policy}) \textit{ is } 0 \\ A(s,a^*) = Q(s,a^*) - V(s) = 0 \end{cases}$$

Argmax policy: a* = $\mathrm{argmax}_{a\prime}\, Q(s,a')$

V(s) = $\sum_a \pi(a|s) Q(s,a) = Q(s,a*)$

Method

- Simple Q(s, a; θ, α, β) = V(s; θ, β) + A(s, a; θ, α) may output Q, but V and A may not be grounded
  - Non-identifiability problem:

$$Q(s, a; \theta, \alpha, \beta) = (V(s; \theta, \beta) - c) + (A(s, a; \theta, \alpha) + c)$$

- To fix, use argmax policy's advantage:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \underline{(A(s, a; \theta, \alpha) - \max_{a'} A(s, a'; \theta, \alpha))}$$

This is zero if a is a*

or $$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha))$$

Results

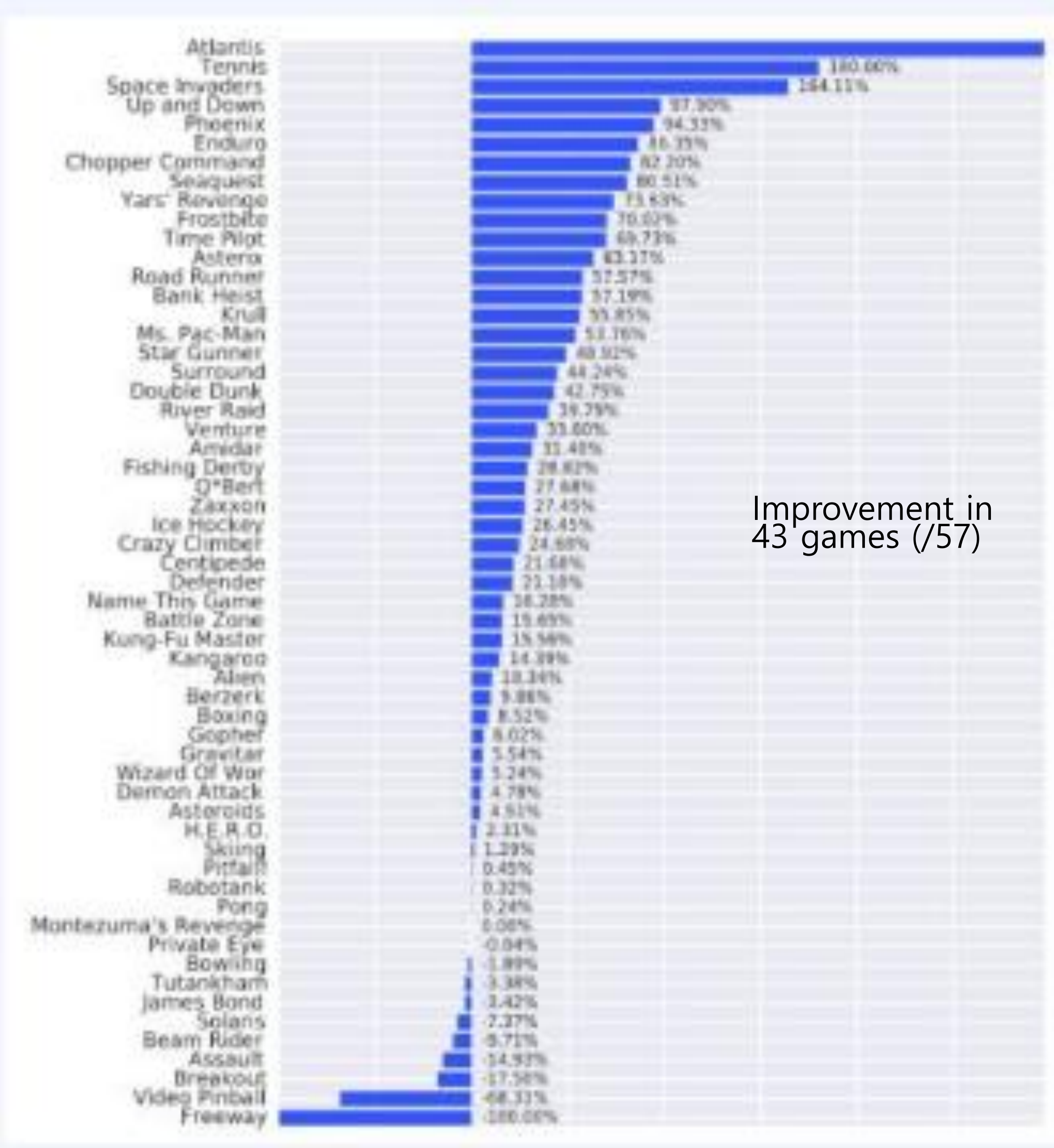
- Learning is fast because policy evaluation is fast
  - policy evaluation is fast because dueling agent learn V and adv.
  - Action-values that have little influence on env. are close to V

Results

Table 1. Mean and median scores across all 57 Atari games, measured in percentages of human performance.

	30 no-ops		Human Starts	
	Mean	Median	Mean	Median
Prior. Duel Clip	**591.9%**	**172.1%**	**567.0%**	**115.3%**
Prior. Single	434.6%	123.7%	386.7%	112.9%
Duel Clip	**373.1%**	**151.5%**	**343.8%**	**117.1%**
Single Clip	341.2%	132.6%	302.8%	114.1%
Single	307.3%	117.8%	332.9%	110.9%
Nature DQN	227.9%	79.1%	219.6%	68.5%

Double DQN



Improvement in 43 games (/57)

Code

```
class DuelingQNetwork(nn.Module):
    def __init__(self, env, config):
        super().__init__()
        d_state = env.observation_space.shape[0]
        n_action = env.action_space.n

        self.encoder = nn.Sequential(
            nn.Linear(d_state, config.hidden_size),
            nn.ELU(),
        )
        self.advantage_head = nn.Sequential(
            nn.Linear(config.hidden_size, config.hidden_size),
            nn.ELU(),
            nn.Linear(config.hidden_size, n_action)
        )
        self.value_head = nn.Sequential(
            nn.Linear(config.hidden_size, config.hidden_size),
            nn.ELU(),
            nn.Linear(config.hidden_size, 1)
        )
```

Code

```
class DuelingQNetwork(nn.Module):
                :

    def forward(self, x):
        h_encoder = self.encoder(x)
        advantages = self.advantage_head(h_encoder)
        value = self.value_head(h_encoder)
        Qs = value + (advantages - advantages.mean(dim=-1, keepdim=True))
        return Qs
```

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

        self.network = DuelingQNetwork(env, config)
        self.target_network = DuelingQNetwork(env, config)

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

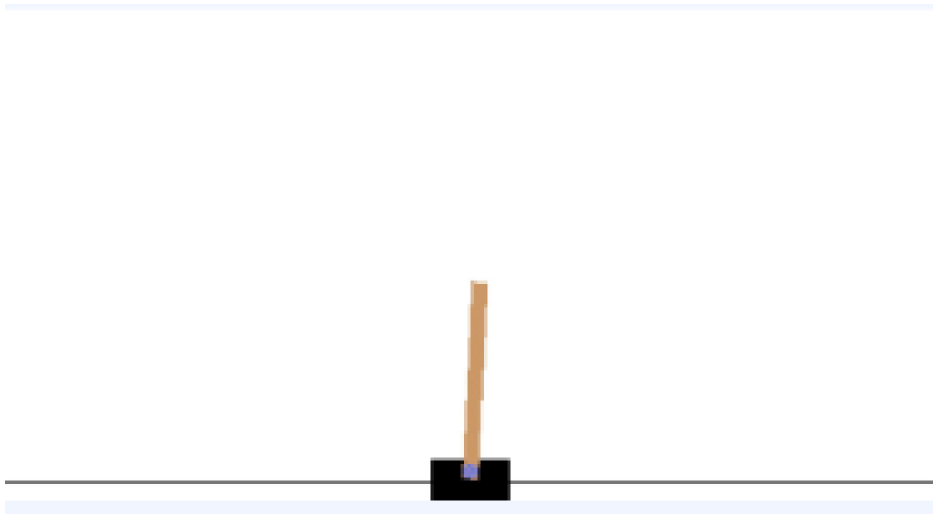
Code Ex.

Code Ex. Environment

- State:
  (cart position, cart velocity, pole angle, pole angular velocity)

- Action:
  (push car left, push car right)

- Reward:
  +1 for every step



Code Ex.

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

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

Code Ex.

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

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

Code Ex.

- Python libraries:
  - gym==0.25.1
  - ale-py==0.7.5
  - torch==2.0.0 (cpu버전)
  - tensorboard==2.13.0
- Code files:
  - Configuration.py
  - Dueling_dqn_agent.py
  - eval.py
  - utils.py