7. Value Function Approximation

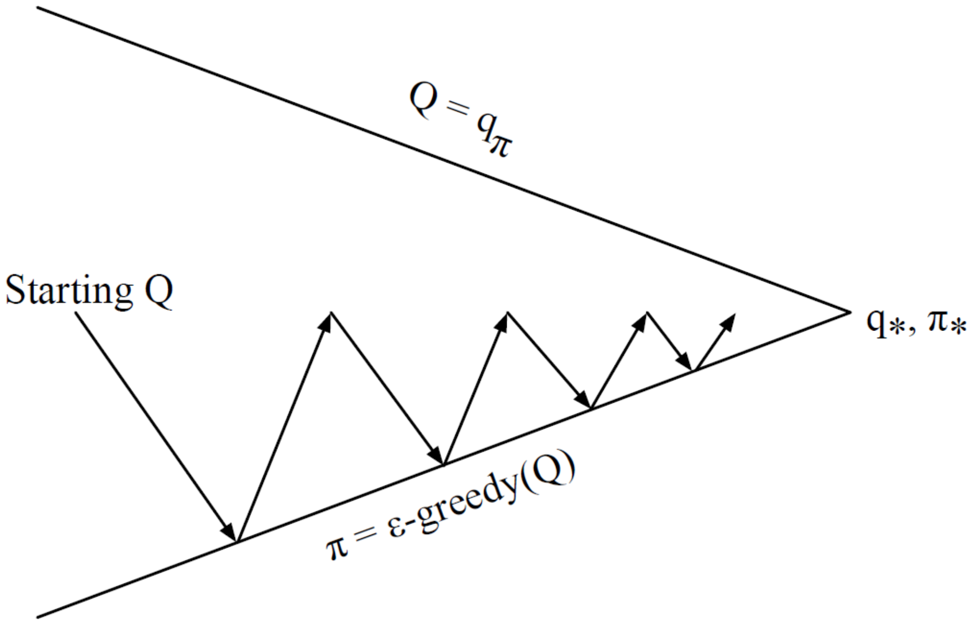
2019 Fall

Yusung Kim
yskim525@skku.edu

Review of Last Class (Model-Free RL)

- When we don't know a true model :
  - Generalized policy improvement
  - Importance of exploration
- Model-free control with Monte Carlo (MC)
- Model-free control with Temporal Difference (TD)
  - SARSA
  - Q-learning

Model-Free Control: On-Policy vs. Off-Policy



SARSA:

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

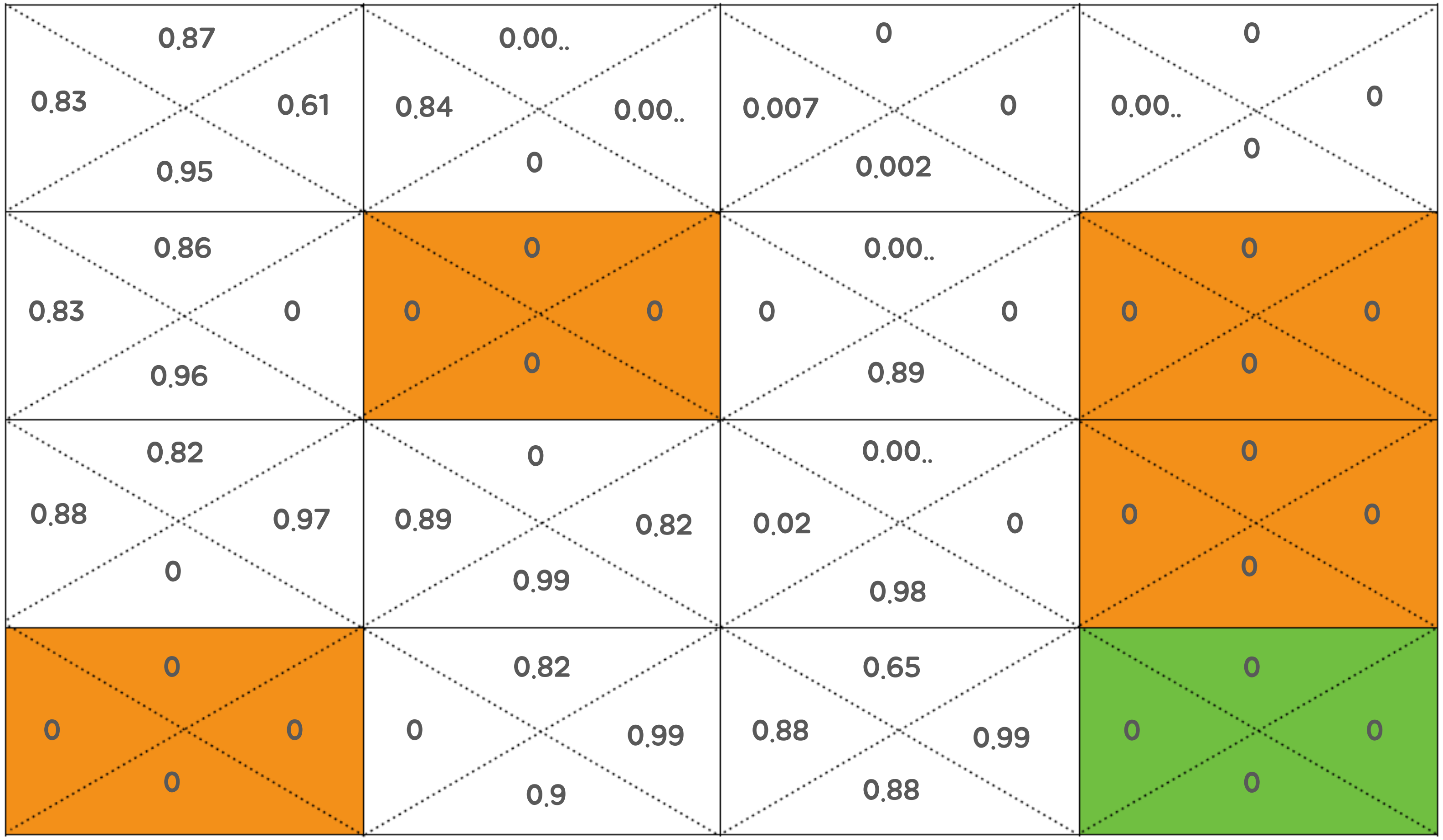
Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t) \right]$$

$$= R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_{a'} Q(S_{t+1}, a'))$$

$$= R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a')$$

Example : Q-Table for FrozenLake

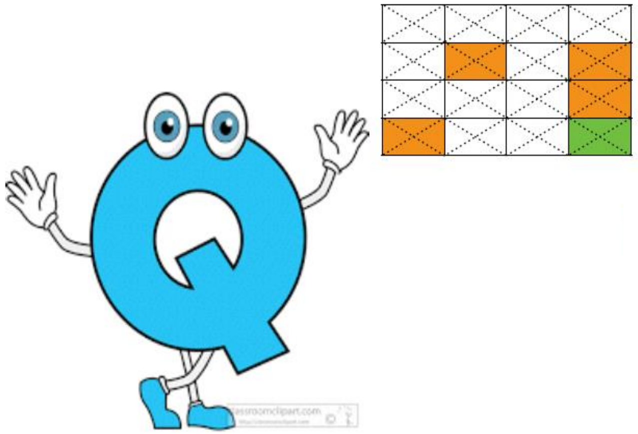


Limitation of Q-Learning using a Q-Table

- Case of FrozenLake with a small size (4 x 4) map

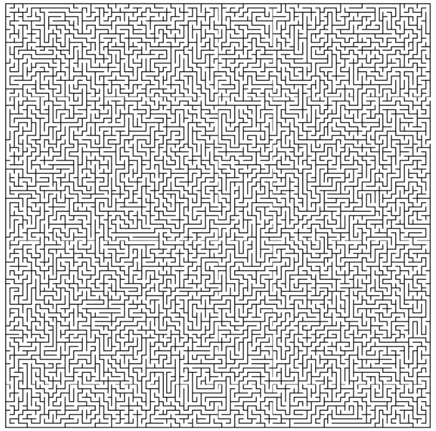
  Q table requires 64 sates (4 x 4 positions x 4 actions)

- What if larger problems ?



Q (s, a)

Large-scale Problem Examples

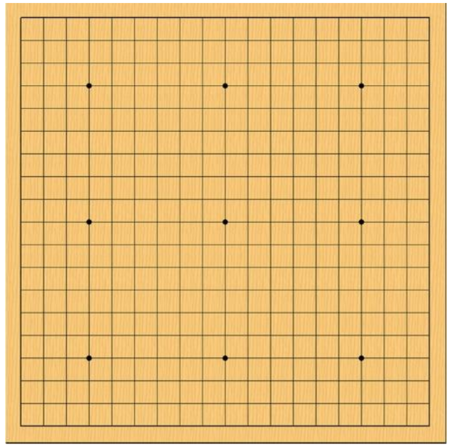


100 x 100 maze

#states : 100 x 100
#actions : 4

Atari video game

#states: $256^{84 \times 84 \times 4}$
#actions: 18



Go

#states : $3^{19*19*7}$
#actions : 19 * 19

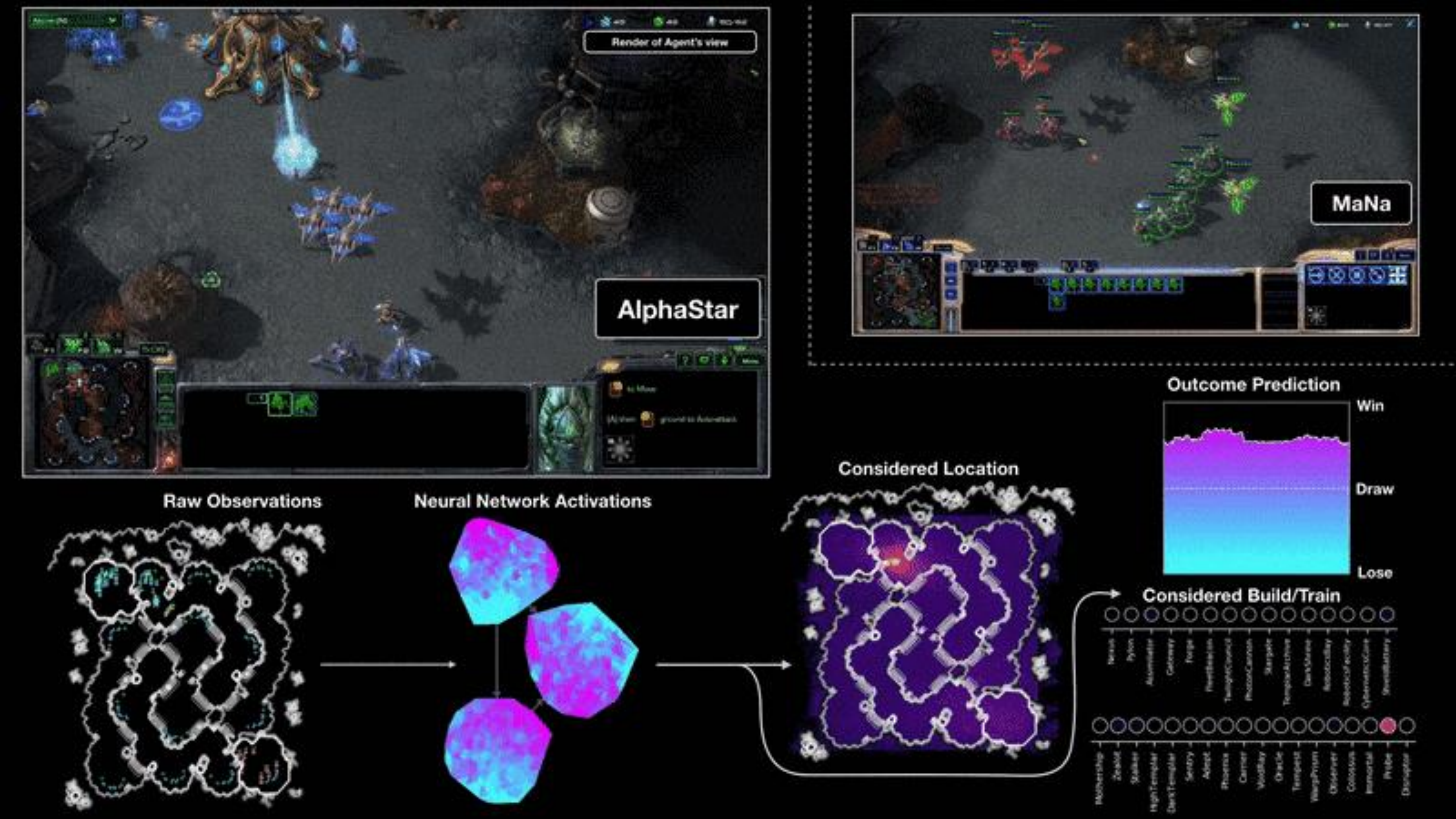
StarCraft

#states : partial
#actions : 10^8

Model-free Reinforcement Learning

- Model-free prediction (evaluation)
  - Estimate the value function of an unknown MDP
  - How good is this given policy ?

- Model-free control (improvement)
  - Optimize the value function of an unknown MDP
  - How can we learn a better policy ?



Render of Agent's view
AlphaStar
MaNa
Outcome Prediction
Win
Draw
Lose
Considered Location
Raw Observations
Neural Network Activations
Considered Build/Train

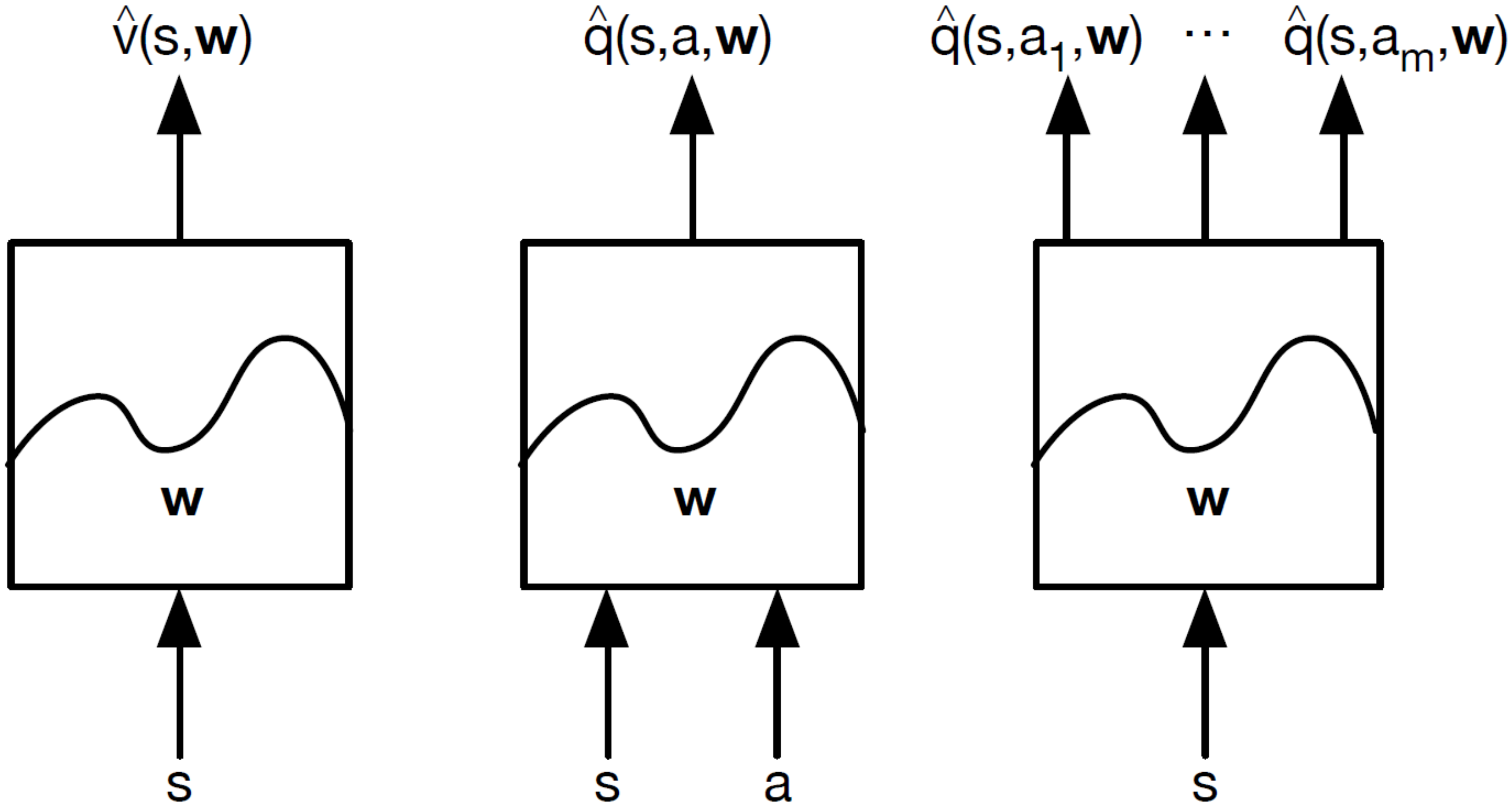
How can we scale up model-free RL?

Value Function Approximation

- So far we have represented value function by a lookup table
  - Every state s has an entry V(s)
  - Or every state-action pair s, a has an entry Q(s, a)
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with function approximation (parameterized function)

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

Type of Value Function Approximation



Motivation for VFA

- Don't explicitly store or learn for every single state
  - Dynamics or reward model
  - Value
  - State-action value
  - Policy
- Want more compact representation that generalizes across state or states and actions

Benefits of Generalization

- Reduce memory needed to store $(P, R)/V/Q/\pi$
- Reduce computation needed to compute $(P, R)/V/Q/\pi$
- Reduce experience needed to find a good $(P, R)/V/Q/\pi$

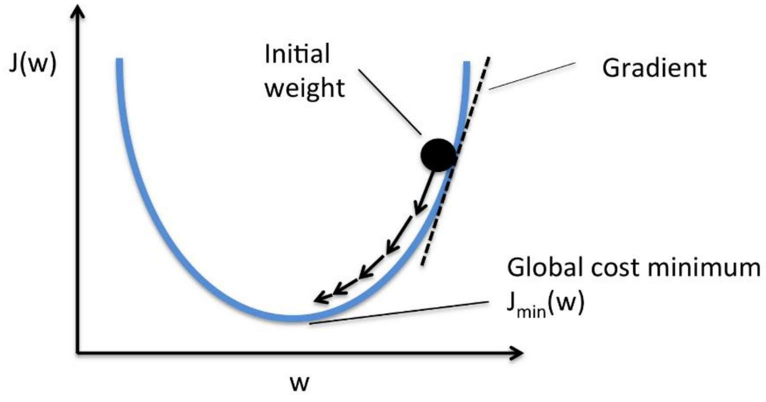
Function Approximators

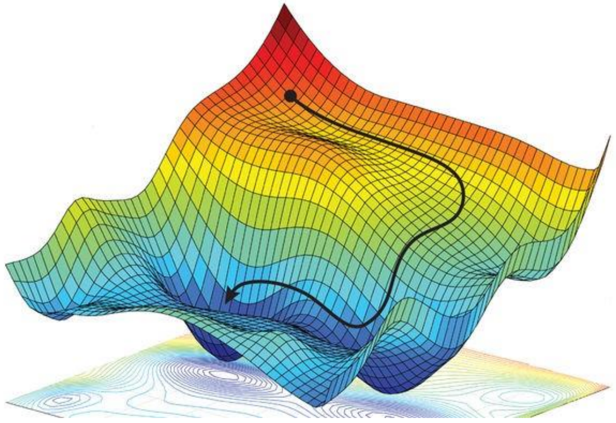
- Many possible function approximators including
  - Linear combinations of features
  - Neural networks
  - Decision trees
  - Nearest neighbors
  - Fourier/ wavelet bases
- In this class we will focus on function approximators that are differentiable
  - Nice smooth optimization
- Two very popular classes of differentiable function approximators
  - Linear feature representations
  - Neural networks

Review: Gradient Descent

- Consider a function $J(w)$ that is a differentiable function of a parameter vector w
- Goal is to find parameter w that minimizes J
- The gradient of $J(w)$ is ;

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$





Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector w minimizing mean-squared error between approximate $\hat{v}(s,w)$ and true value $v(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S,\mathbf{w}))^2]$$

- Gradient descent finds a local minimum

$$\begin{aligned}\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_\mathbf{w}J(\mathbf{w}) \\ &= \alpha\mathbb{E}_\pi\left[(v_\pi(S) - \hat{v}(S,\mathbf{w}))\nabla_\mathbf{w}\hat{v}(S,\mathbf{w})\right]\end{aligned}$$

- Stochastic gradient descent samples the gradient

Feature Vectors

- Represent state by a feature vector

$$\mathbf{x}(S) = \begin{pmatrix} \mathbf{x}_1(S) \\ \vdots \\ \mathbf{x}_n(S) \end{pmatrix}$$

- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece configurations in chess

Linear Value Function Approximation

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S)\mathbf{w}_j$$

- Objective function is quadratic in parameters w

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \mathbf{x}(S)^\top \mathbf{w})^2 \right]$$

- Update is

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$
$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)$$

update = step-size x prediction error x feature value

Incremental Prediction Algorithms

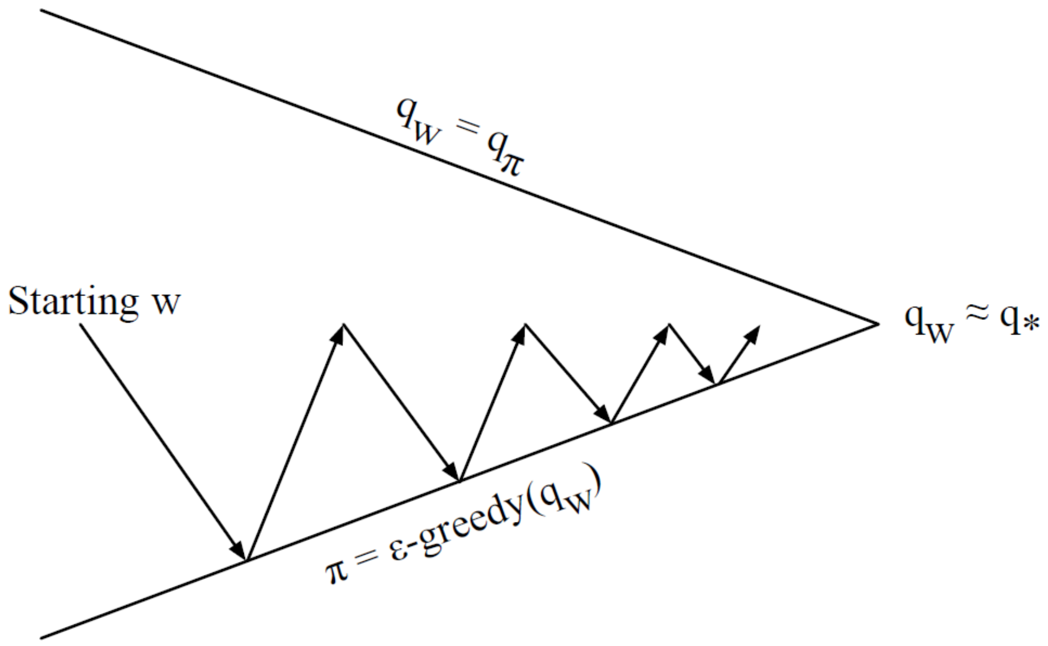
- Have assumed true value function $v_\pi(s)$ given by supervisor
- But in RL there is no supervisor, only rewards
- In practice, we substitute a target for $v_\pi(s)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w})$$

  - For TD, the target is the TD target $R_{t+1} + \gamma\hat{v}(S_{t+1}, w)$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w})$$

Control with Value Function Approximation



- Policy evaluation Approximate policy evaluation, $\hat{q}(s, a, w) \approx q_\pi$
- Policy improvement ϵ-greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, w) \approx q_\pi$$

- Minimzse mean-squared error between approximate action-value $\hat{q}(S, A, w)$ and true action-value $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2 \right]$$

- Use stochastic gradient descent

$$-\frac{1}{2}\nabla_\mathbf{w} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_\mathbf{w}\hat{q}(S, A, \mathbf{w})$$

$$\Delta\mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_\mathbf{w}\hat{q}(S, A, \mathbf{w})$$

Linear Action-Value Function Approximation

- Represent state and action by a feature vector

$$\mathbf{x}(S, A) = \begin{pmatrix} \mathbf{x}_1(S, A) \\ \vdots \\ \mathbf{x}_n(S, A) \end{pmatrix}$$

- Represent action-value function by linear combination of features

$$\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S, A)\mathbf{w}_j$$

Incremental Control Algorithms

- Like prediction, we must substitute a target for $Q_\pi(S,A)$
- For MC, the target is the return G_t

$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{Q}(s_t, a_t; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s_t, a_t; \boldsymbol{w})$$

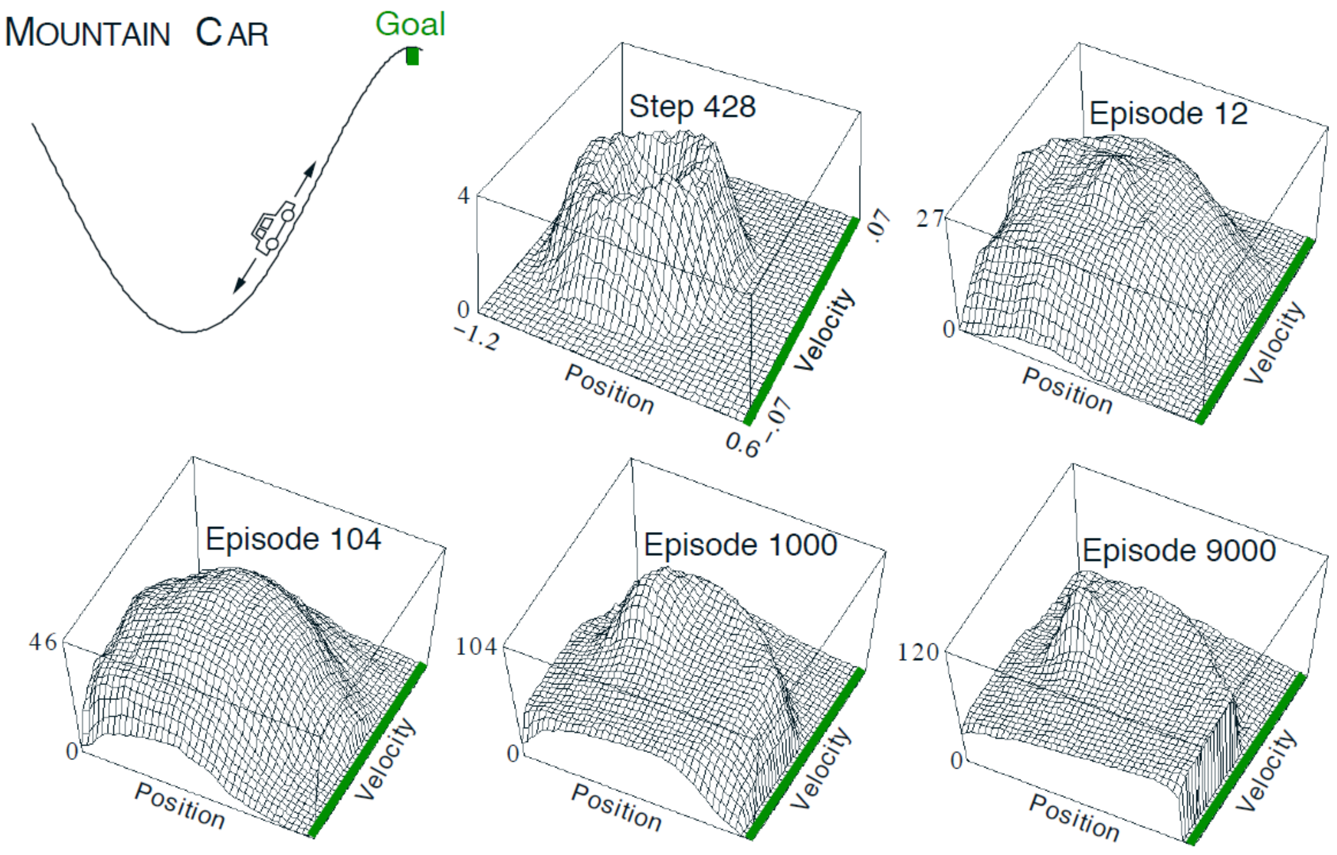
- For SARSA, use a TD target $R + \gamma \hat{Q}(s', a', w)$ which leverages the current function approximation value

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$

- For Q-learning, use a TD target $R + \gamma \max_{a\prime} \hat{Q}(s', a', w)$which leverages the current function approximation value

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$

Linear SARSA in Mountain Car



Deep Reinforcement Learning

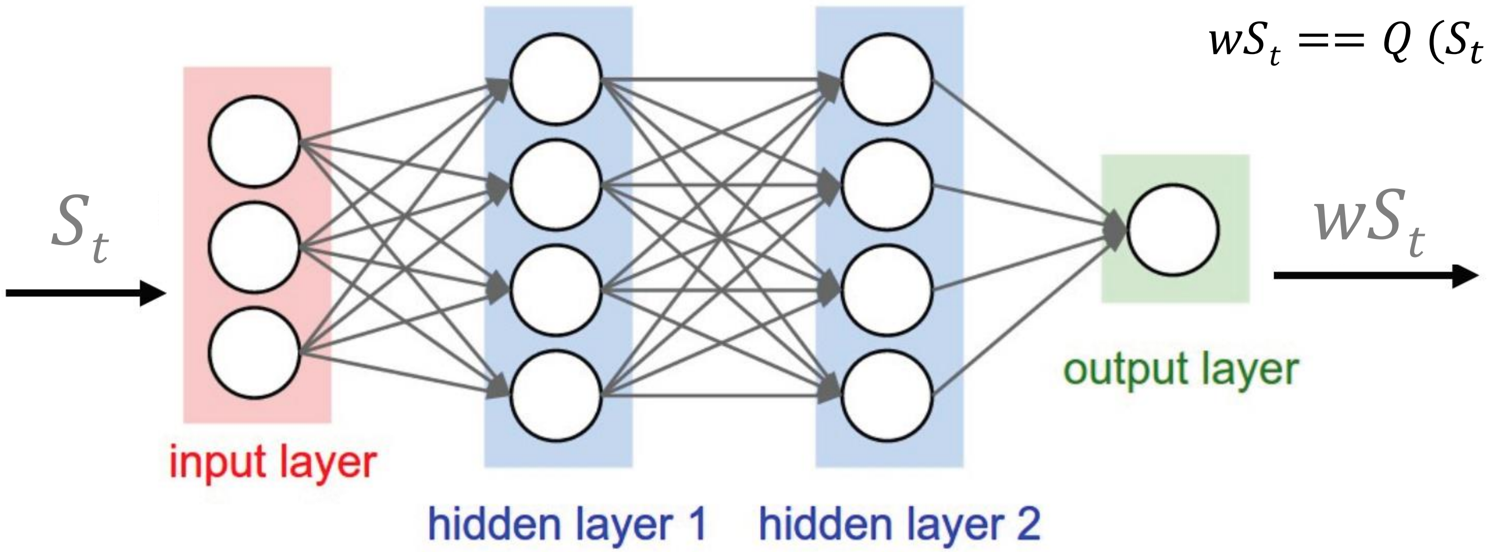
Deep Reinforcement Learning

- Use deep neural networks to represent
  - Value function
  - Policy
  - Model
- Optimize loss function by stochastic gradient descent (SGD)

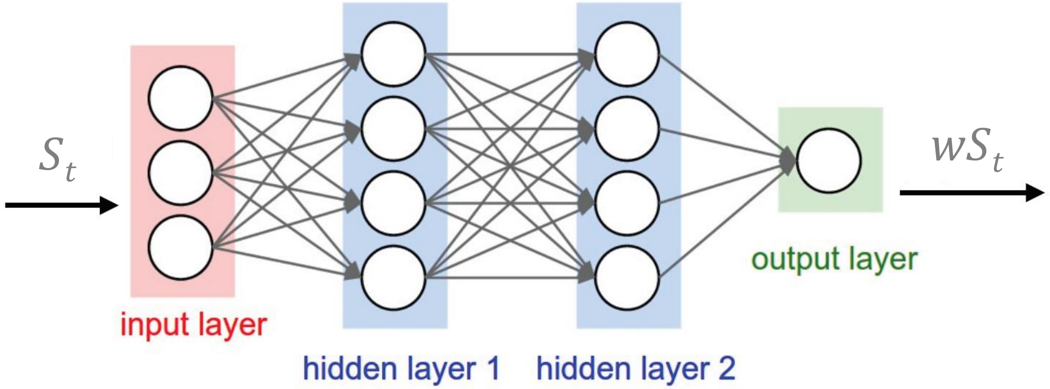
Deep Q-Networks (DQNs)

- Represent state-action value function by Q-network with weights w

$wS_t == Q\ (S_t\,, A)$ for all action A



Deep Q-Network Training



$$J(w) = E_\pi[(\underbrace{R_{t+1} + \max_a \gamma \underbrace{\hat{q}(S_{t+1}, a, w)}_{\text{get from } wS_{t+1}}}_{\text{Q-target}} - \underbrace{\hat{q}(S_t,\ A_t, w)}_{\text{Q-prediction get from } wS_t})2]$$

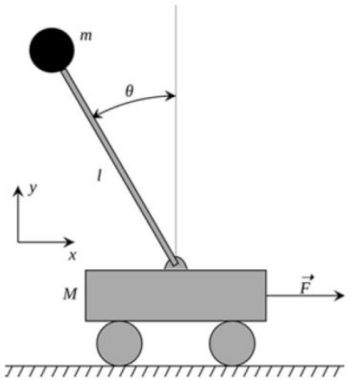
Q-target

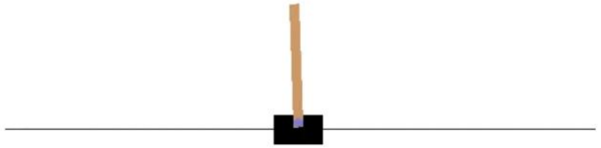
get from wS_{t+1}

Q-prediction
get from wS_t

Example: CartPole in OpenAI Gym

- Goal : Balance the pole on tope of a moving cart
- State : Pole angle, angular speed, cart position, horizontal velocity
- Actions : horizontal force to the cart (e.g. left and right)
- Reward : 1 at each time step if the pole is upright

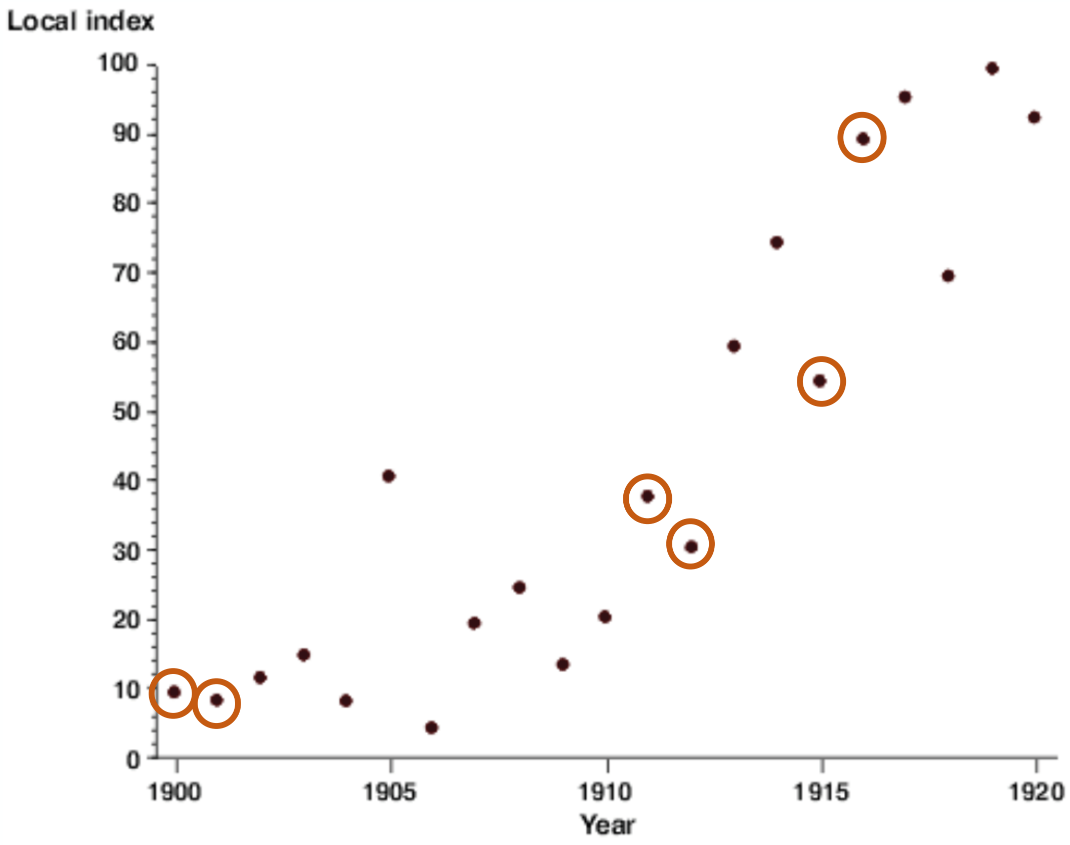




Convergence Issues

- $\hat{q}(S_t, A_t, w)$ denotes learner's current approximation to $q(S_t, A_t)$
- Minimize $J(w) = E_\pi[\,(R_{t+1} + \max_a \gamma \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w))^2\,]$
- Converges to q_* when using table lookup representation, but diverges using neural networks due to:
  - Correlations between samples
  - Non-stationary targets

Correlations between samples



Non-stationary targets

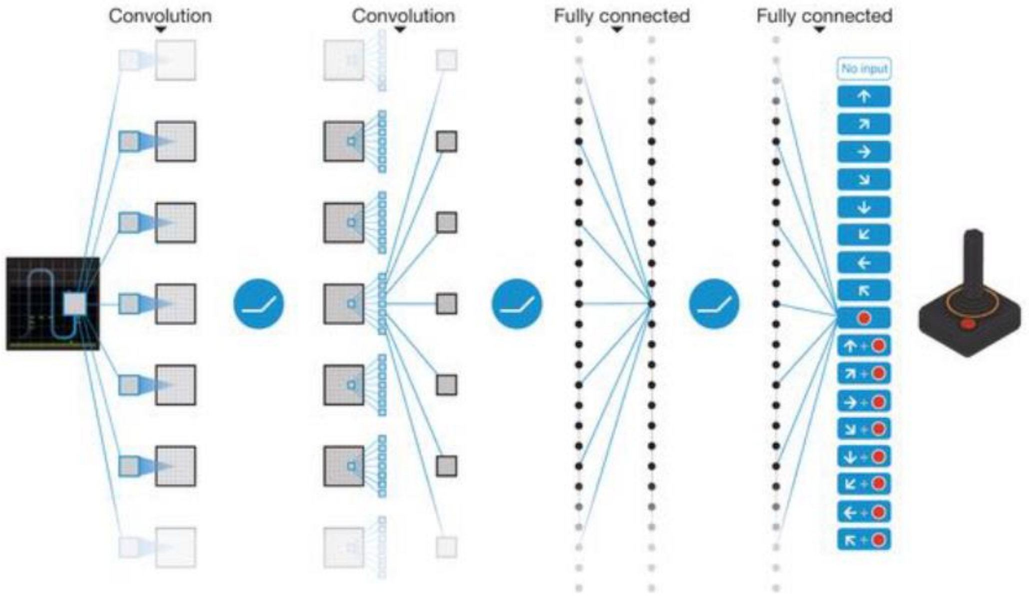
$$J(w) = E_\pi[(q_\pi(S,\ A) - \hat{q}(S,\ A, w)\ )^2]$$
$$= E_\pi\left[\left(R_{t+1} + \max_a \gamma\, \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w)\right)^2\right]$$

- We update a guess with a guess.
- Whenever updating prediction Q, target Q is affected too.
- We need to separate target Q from prediction Q.

Non-stationary targets

Deep Q-Networks (DQN) [NIPS 2013], [Nature 2015]

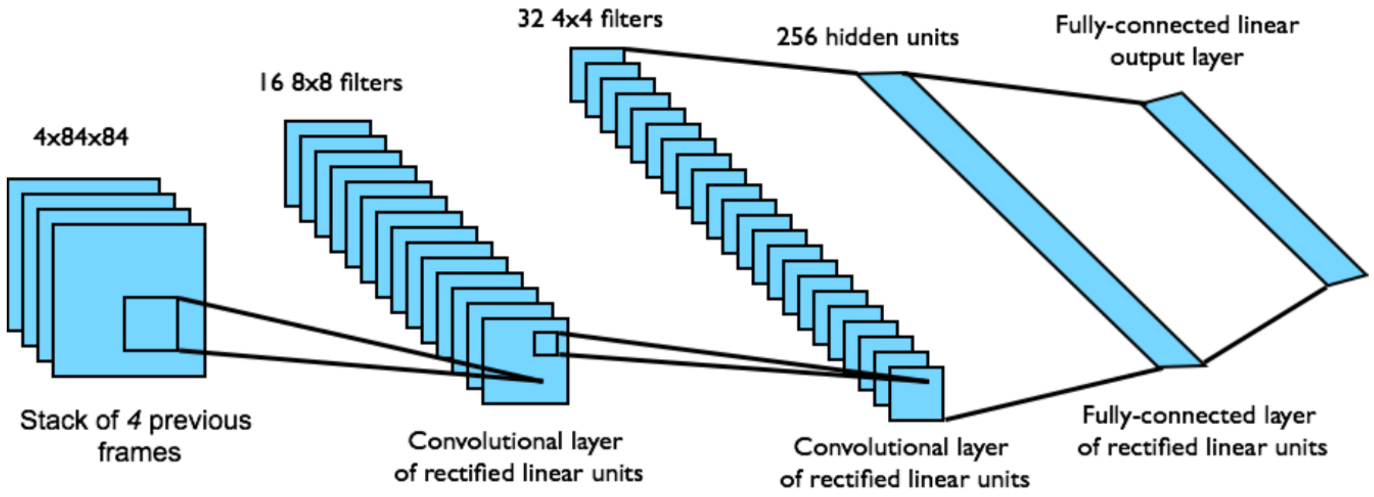
- Deep Neural Networks
- Experience Replay
  - solving correlations between samples
- Fixed Target
  - solving non-stationary targets



1) Deep Neural Networks

- End-to-end learning of values $Q(s,a)$ from pixels s
- Input state is stack of raw pixels from last 4 frames
- Output is $Q(s,a)$ for 18 joystick/button positions
- Reward is change in score for that step

Network architecture and hyper-parameters fixed across all games.



2) Experience Replay

- To remove correlations, build data-set from agent's own experiences
  - Take action at according to ϵ-greedy policy
  - Store experience (s_t , at , r_{t+1}, s_{t+1}) in replay memory D
  - Sample random mini-batch of transitions (s, a, r, s') from D
  - Optimize MSE between Q-learning predictions and Q-learning targets,

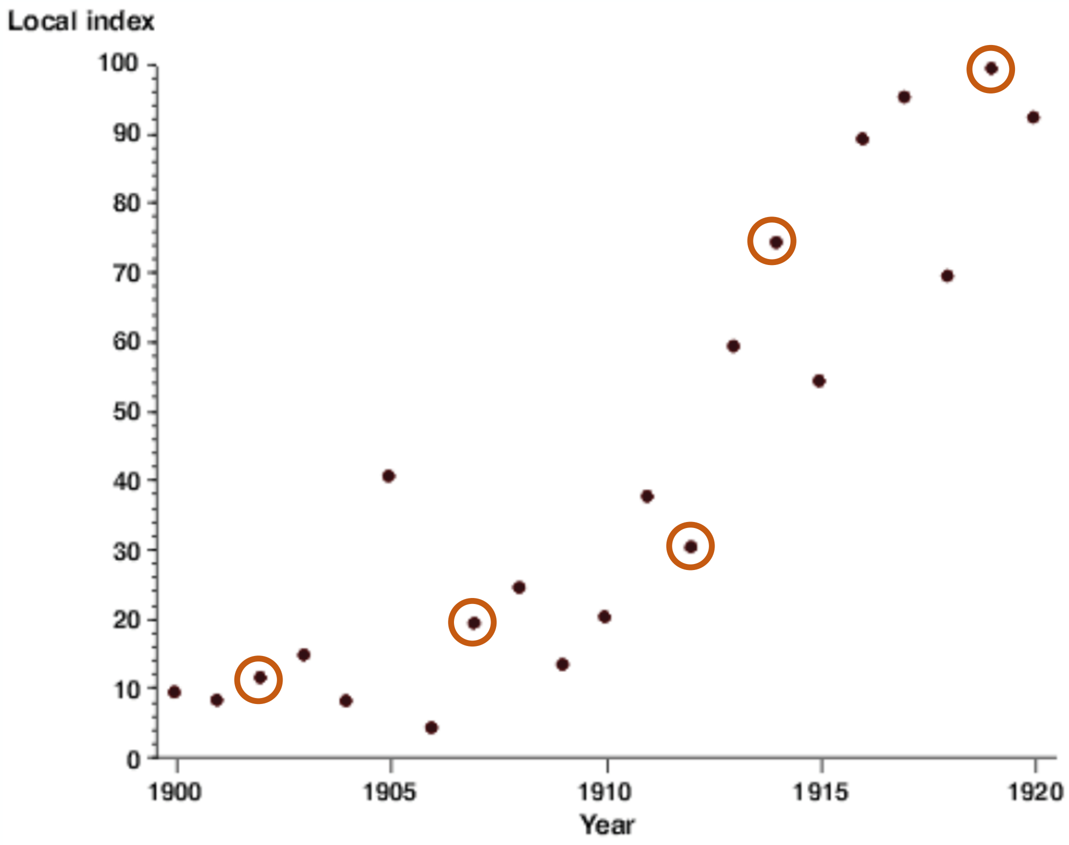
s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
...
$s_t, a_t, r_{t+1}, s_{t+1}$

e.g. $$\mathcal{L}(w) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right)^2\right]$$

Review: Correlations between samples

Store & Random Sample

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\dots$
$s_t, a_t, r_{t+1}, s_{t+1}$



"Playing Atari with Deep Reinforcement Learning" [NIPS 2013]

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

3) Fixed Target

- To avoid oscillations, fix parameters used in Q-learning target
  - Compute target value with regard to old, fixed parameters w^-

$$r + \gamma \max_{a'} Q(s', a', w^-)$$

  - Optimize MSE between target and Q-network

$$\mathcal{L}(w) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w) \right)^2 \right]$$

  - Separate target network, and periodically update fixed parameters $w^- \leftarrow w$

"Human-level control through deep reinforcement learning" [Nature 2015]

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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$

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 $\hat{Q} = Q$

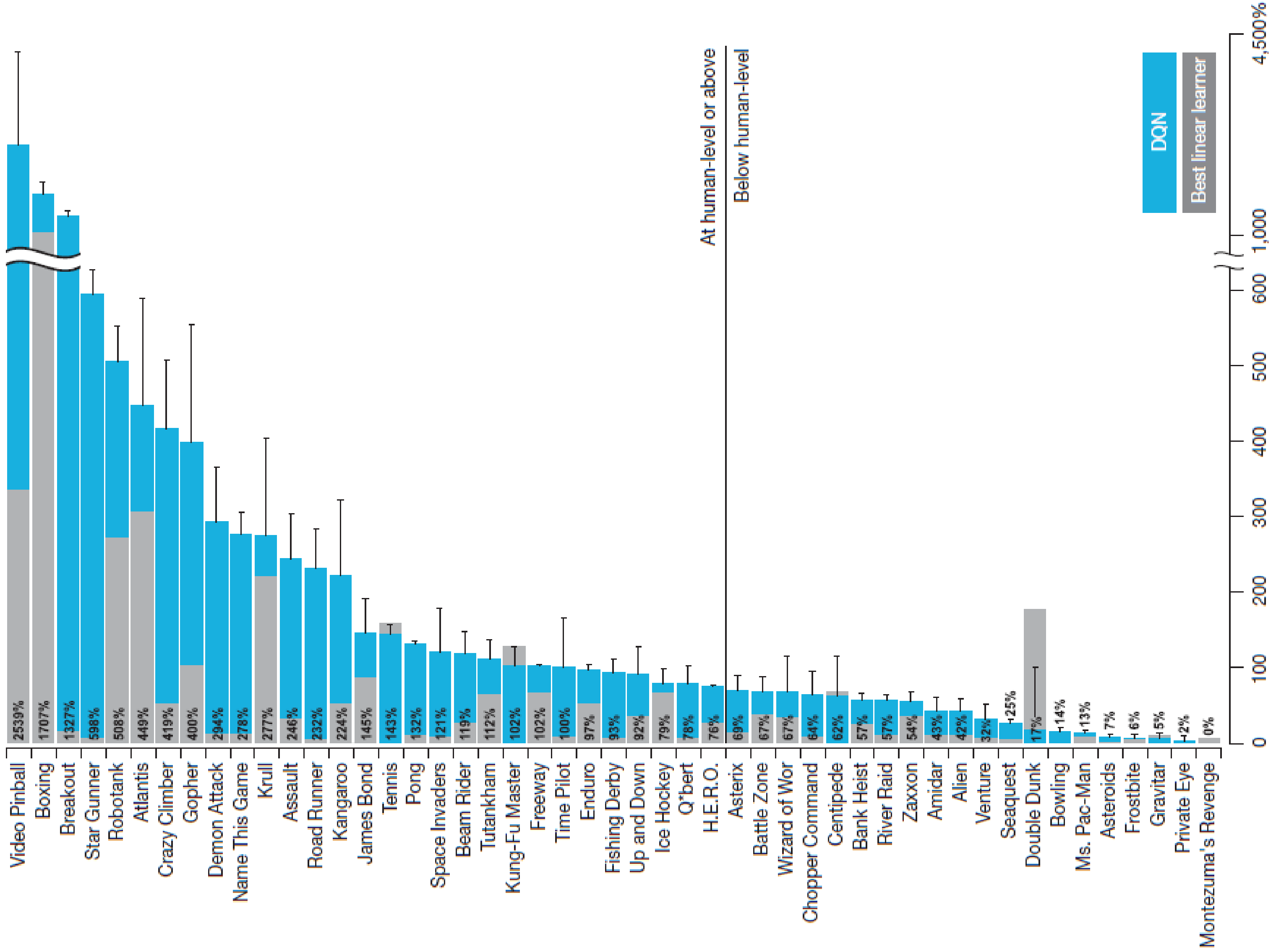
End For

End For

DQNs Summary

- DQN uses experience replay and fixed Q-targets
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
- Sample random mini-batch of transitions (s, a, r, s') from D
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimizes MSE between Q-network and Q-learning targets
- Uses stochastic gradient descent

DQN Results on 49 Atari Games



The effects of experience replay and fixed target network

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

Deep RL

- Success in Atari has led to huge excitement in using deep neural networks to do value function approximation in RL
- Some immediate improvements (many others!)
  - Double DQN (Deep Reinforcement Learning with Double Q-Learning, Van Hasselt et al, AAAI 2016)
  - Prioritized Replay (Prioritized Experience Replay, Schaul et al, ICLR 2016)
  - Dueling DQN (best paper ICML 2016) (Dueling Network Architectures for Deep Reinforcement Learning, Wang et al, ICML 2016)

Review DQN

- DQN consists of Target Network and Experience Replay.
- Q-network target is ;

$$R_{t+1} + \gamma \max_a Q(S_{t+1}, a; w^-)$$

- The max operator in DQN (and standard Q-learning) uses the same values both to select and to evaluate an action.
  - more likely to select overestimated values
  - resulting in overoptimistic value estimates

Recall: Double DQN

- Recall maximization bias challenge
  - Max of the estimated state-action values can be a biased estimate of the max
- Double Q-learning is to reduce overestimations
- DQN already has additional Target Network
  - Get most of the benefit of Double Q-learning, while keeping the rest of the DQN algorithm, with minimal computational overhead.

Recall: Double Q-Learning

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
        Take action A, observe R, S'
        With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

        else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

        $S \leftarrow S'$
    until S is terminal

Double DQN

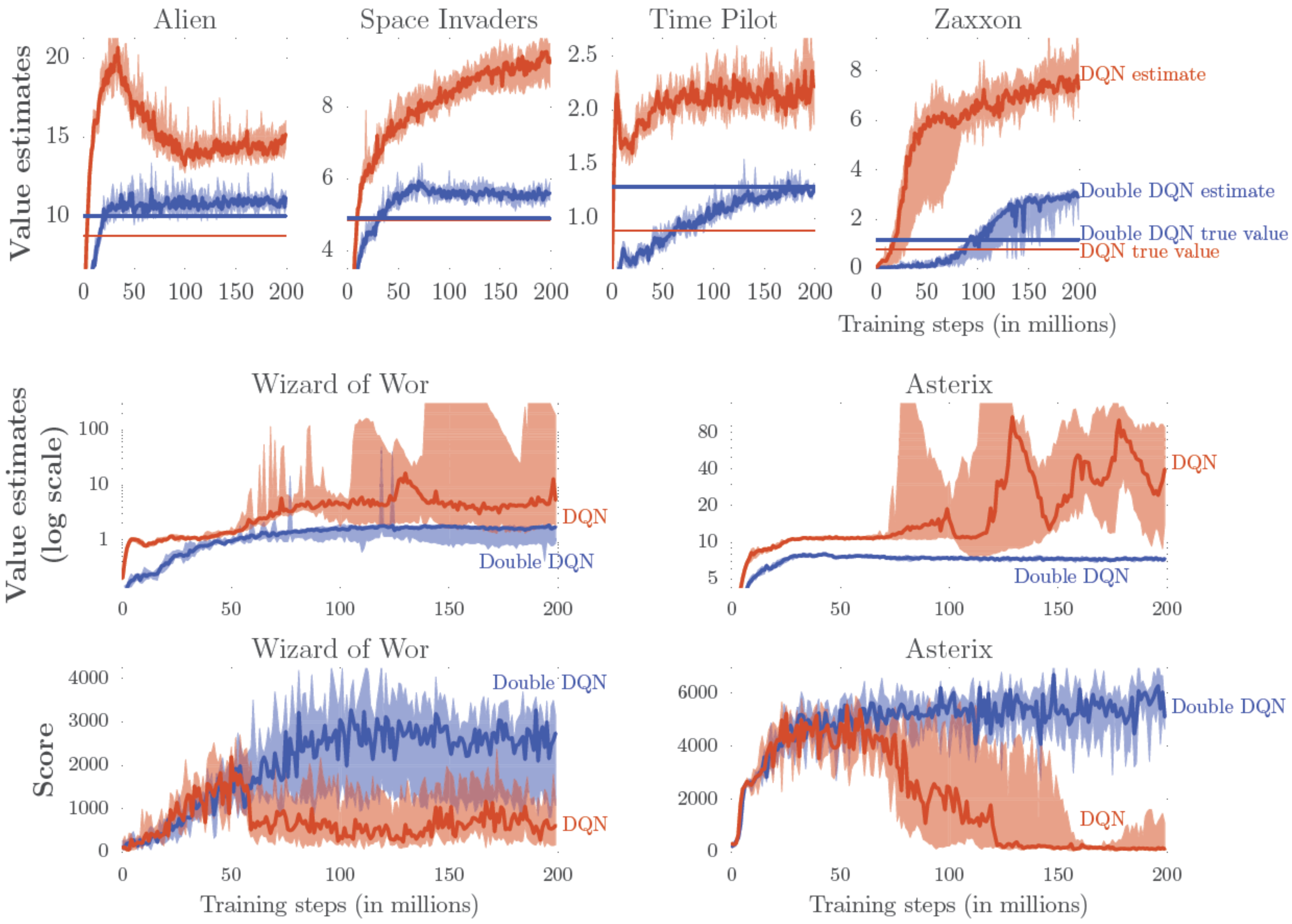
- Extend this idea to DQN
- Current Q-network w is used to select actions
- Older Q-network w^- is used to evaluate actions

$$R_{t+1} + \gamma \max_a Q(S_{t+1}, a; w^-)$$ => DQN Target

$$R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a; w^-); w^-)$$ => DQN Target

$$R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a; w); w^-)$$ => Double DQN Target

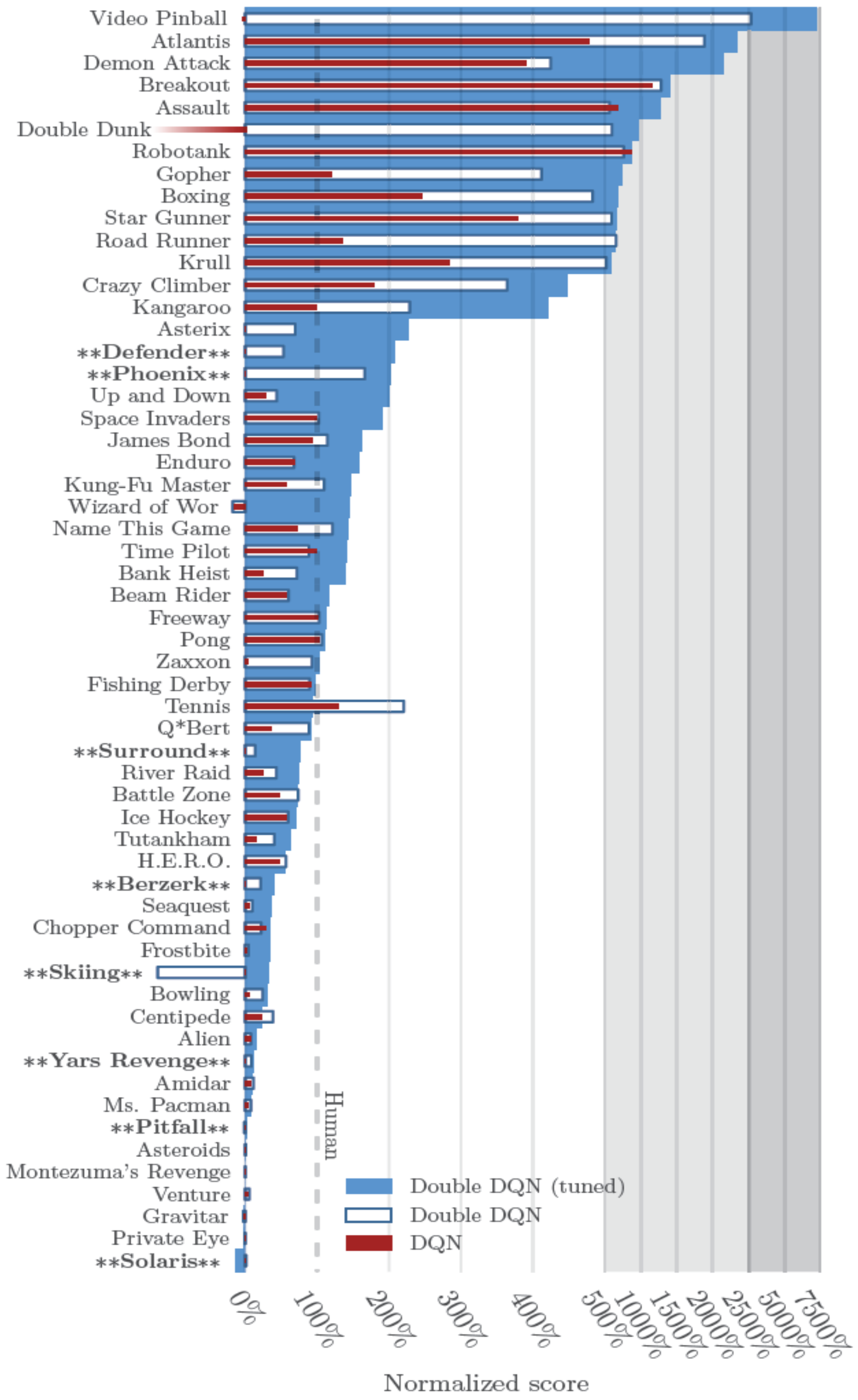
DQN vs. Double DQN



DQN vs. Double DQN

	no ops		human starts		
	DQN	DDQN	DQN	DDQN	DDQN (tuned)
Median	93%	**115%**	47%	88%	**117%**
Mean	241%	**330%**	122%	273%	**475%**

Summarized performance on 49 games



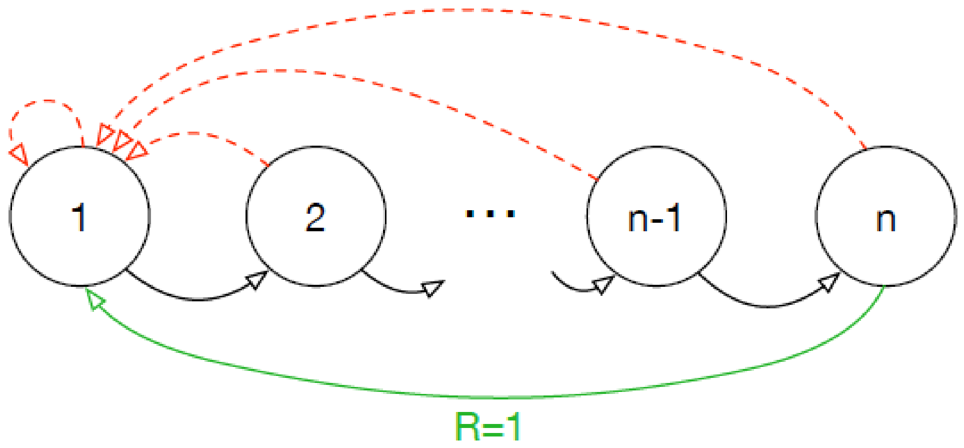
Double DQN Summary

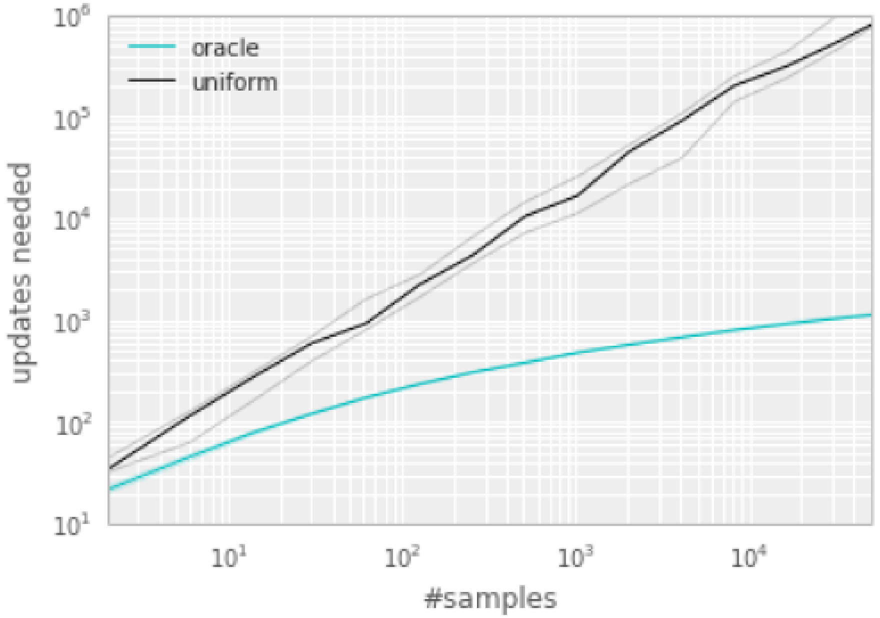
- Indicate overestimations are more common and severe in practice
- Double Q-learning can be used successfully in more stable and reliable learning
- Double DQN uses the existing DQN architecture without additional networks
- Double DQN outperformed single DQN

"Prioritized Experience Replay" [ICLR 2016]

- Experience Replay remembers and replays experiences from the past.
  - break the temporal correlations by random sampling.
  - experience can be reused.
- Uniformly sampled from a replay memory
- Simply replays experiences at the same frequency, regardless of their significance

A Motivating Example: Blind Cliffwalk





Prioritizing with TD-Error

- Experiences may be more or less surprising or important.
- Measure the magnitude of temporal-difference (TD) errors
    - TD error indicates how surprising the experience is.
    - Greedy TD-error prioritization
        - sampling O(1) and updating priorities O(log N)

Stochastic Prioritization

- Greedy TD-error prioritizations…
  - Focus on a small subset of the experience
  - Experiences with a low TD-error may not be replayed for a long time
  - Sensitive to noise spikes (when rewards are stochastic)
- Propose a stochastic monotonic sampling in a experience priority, while guaranteeing a non-zero probability

Stochastic Prioritization

- Probability of sampling experience i as $P(i) = \frac{p_i{}^{\alpha}}{\sum_k p_k^{\alpha}}$ where $p_i > 0$.
- The exponent $\alpha \in [0,1]$ determines how much prioritization is used.
  ($\alpha = 0$ corresponding to the uniform case)
- Option 1 (proportional prioritization) : $p_i = |\delta_i| + \epsilon$
- Option 2 (rank-based prioritization) : $p_i = \frac{1}{rank(i)}$
  where $rank(i)$ is the rank of experience i in sorted replay memory

Annealing the Bias

- Prioritized replay introduces bias because it changes experience distribution
- We can correct this bias by using importance-sampling (IS) weights that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta}$$

- Q-learning update by using $w_i \delta_i$ instead of δ_i
- linearly anneal β from its initial value β_0 to 1

$$\delta = (R + \gamma \max_a Q(S', a)) - Q(S, A)$$

$$Loss = MSE(\delta)$$

$$\downarrow$$

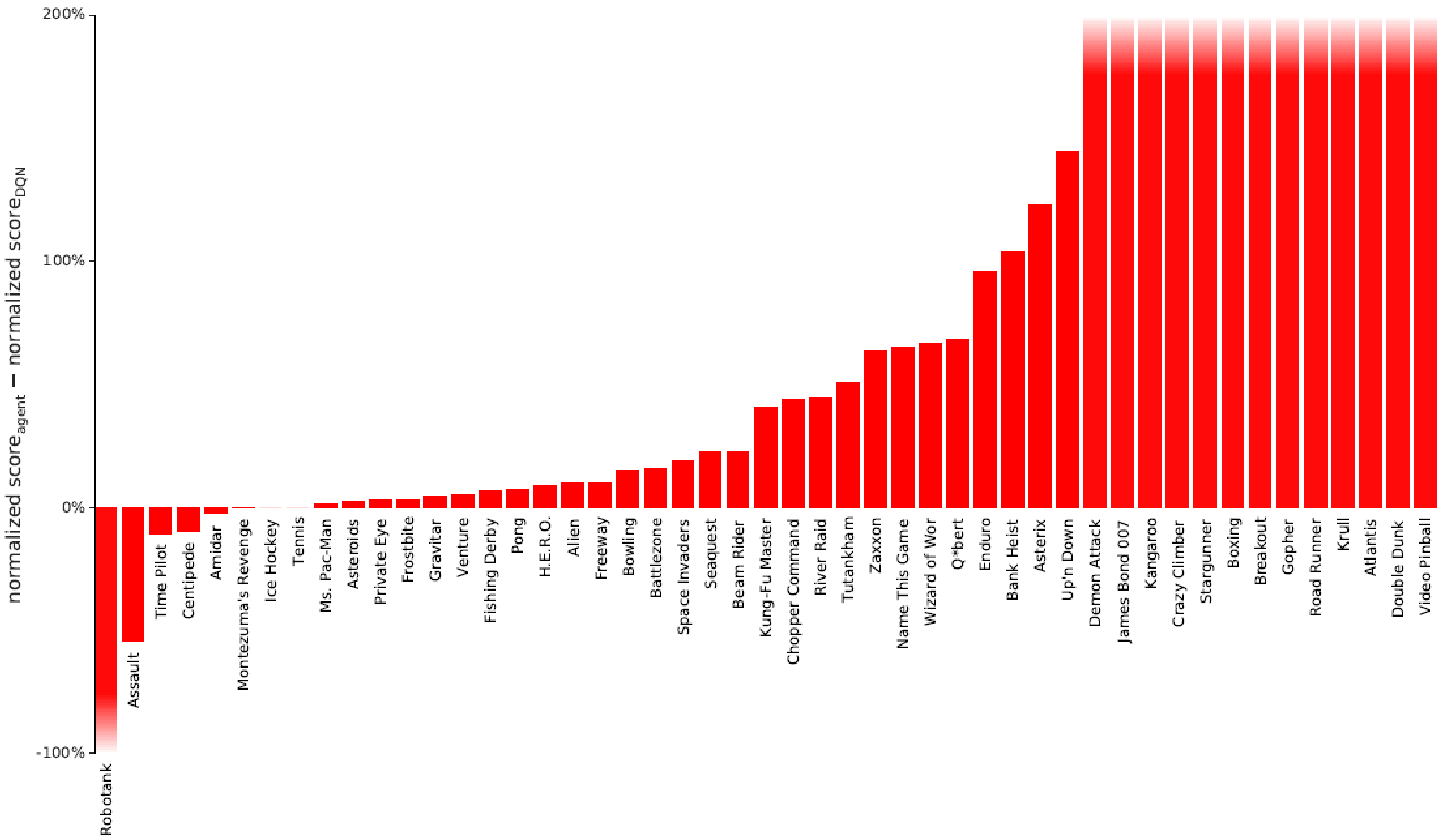
$$Loss = MSE(\frac{w_i}{\max w_i} * \delta)$$

PER Algorithm

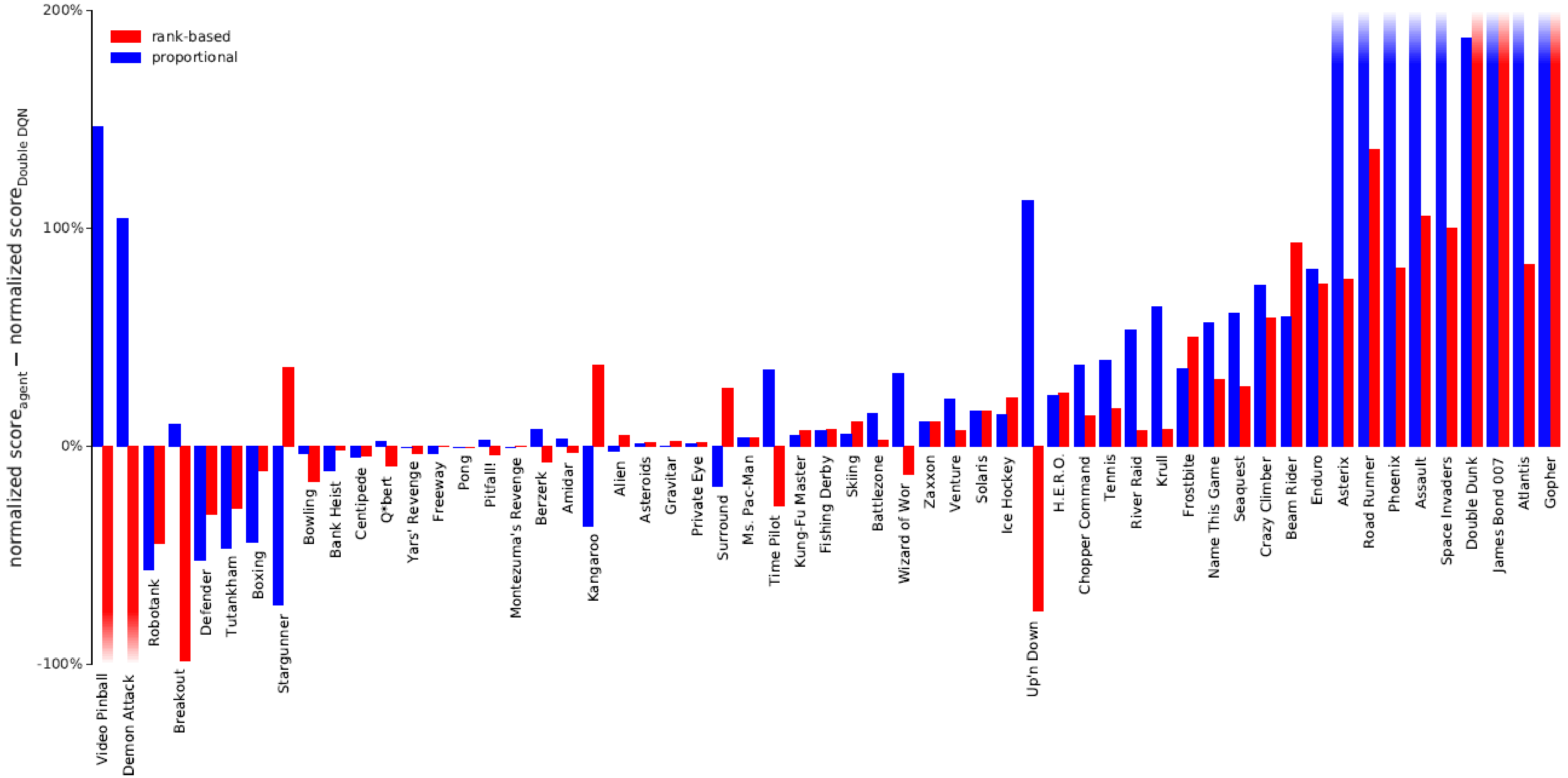
Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
    Observe S_t, R_t, γ_t
    Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
    if $t \equiv 0 \mod K$ **then**
        for $j = 1$ **to** k **do**
            Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
            Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
            Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
            Update transition priority $p_j \leftarrow |\delta_j|$
            Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
        end for
        Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
        From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
    end if
    Choose action $A_t \sim \pi_\theta(S_t)$
end for

DQN vs. DQN with PER



Double DQN vs. Double DQN with PER

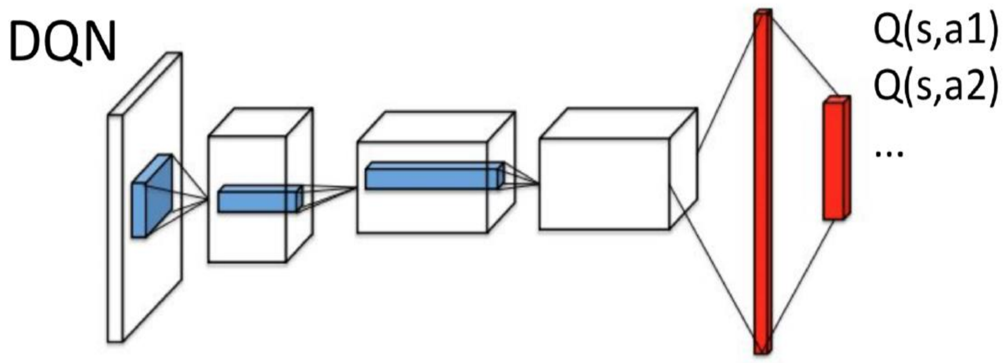


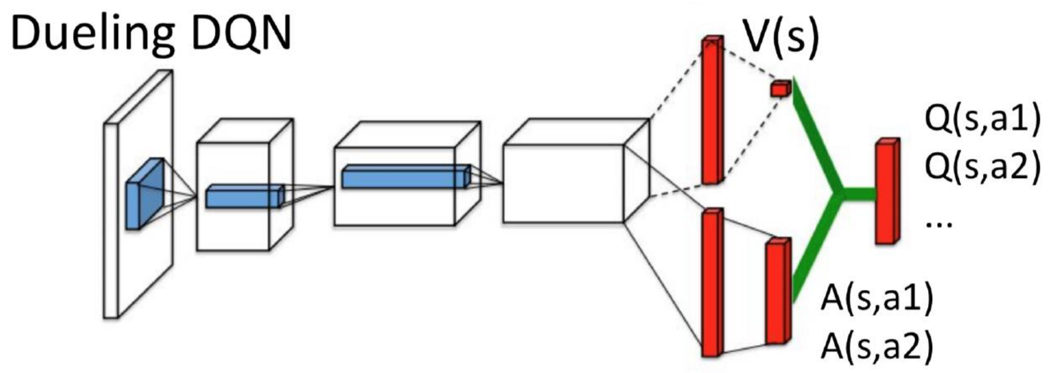
Prioritized Experience Replay Summary

- Prioritized replay method;
  - can make learning more efficient
  - speed up learning by a factor 2
  - improve performance on the Atari benchmark

"Dueling Network Architectures for Deep Reinforcement Learning" [ICML 2016, Best Paper]

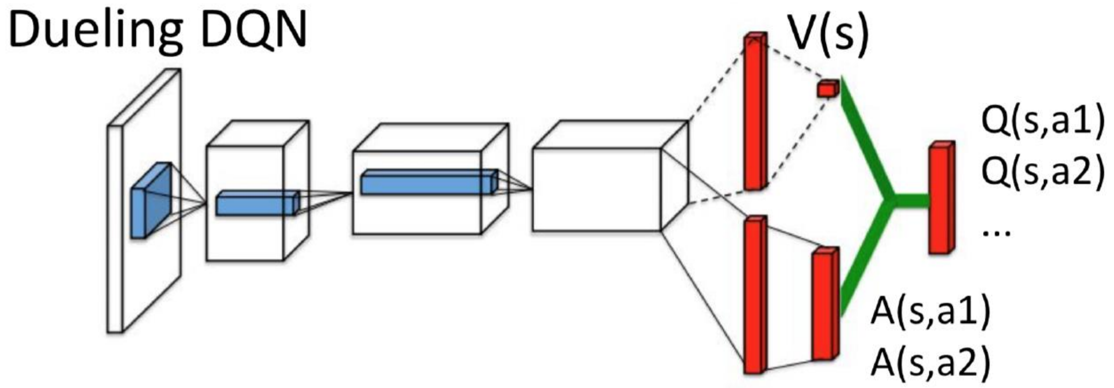
- A new neural network architecture for model-free reinforcement learning
- Easy combined with existing and future algorithms for RL





Dueling Network Architecture

- Separates the state values and action advantages, while sharing a common CNN module.
- The two streams are combined via a aggregating layer to produce Q-function.
- Intuitively, Dueling Network can learn which states are valuable without having to learn the effect of each action.



Dueling Network Design

- $Q(s, a) = V(s) + A(s, a)$
- It is unidentifiable in the sense that given Q.

  (we cannot recover V and A uniquely).
  - For example, When $Q = 4$, $V + A$ can be $1 + 3$, $2 + 2$, or $3 + 1$
- This lack of identifiability is mirrored by poor practical performance

  when this equation is used directly.

Dueling Network Design

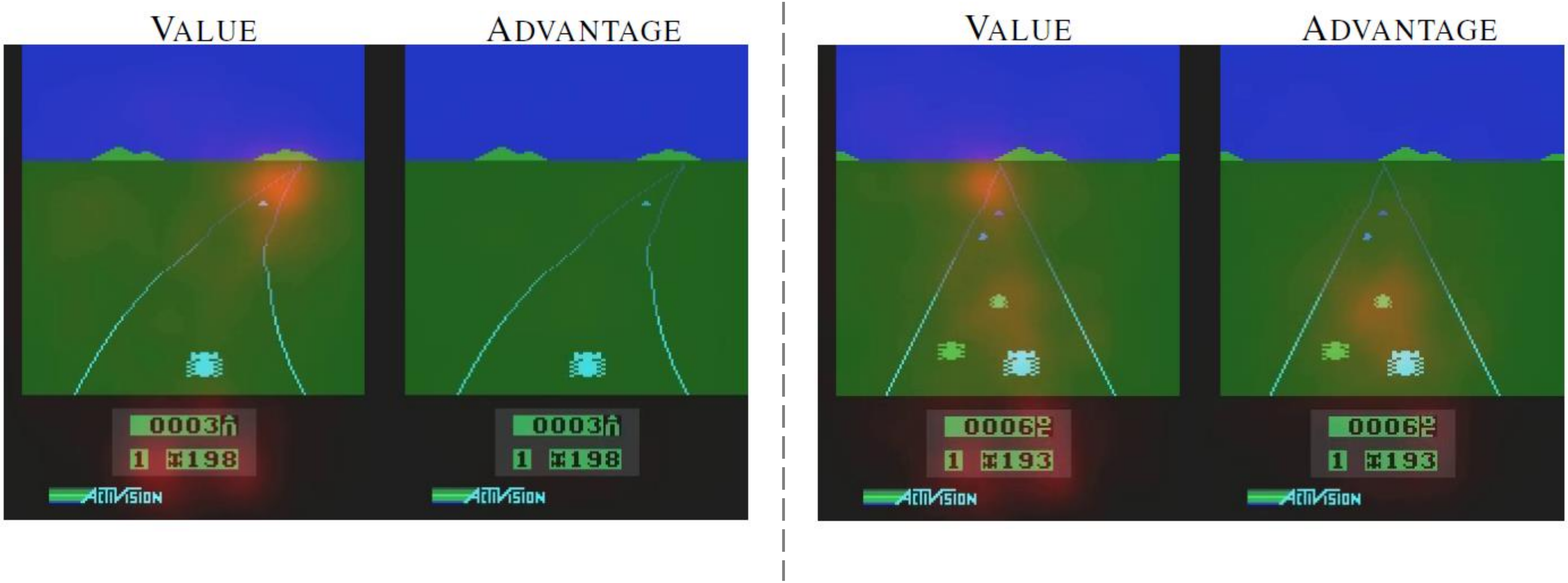
- To address this issue of identifiability
- Option1 : Force $A(s,a) = 0$ if a is action taken

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

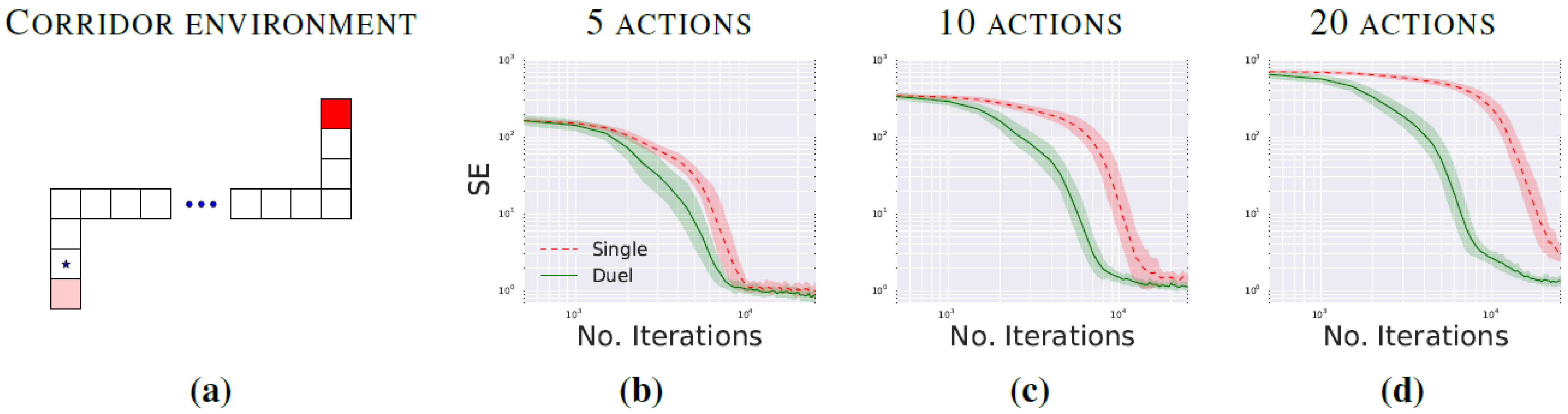
- Option2: Use mean as baseline (more stable)

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

Example: Value and Advantage



Simple Task Evaluation Result

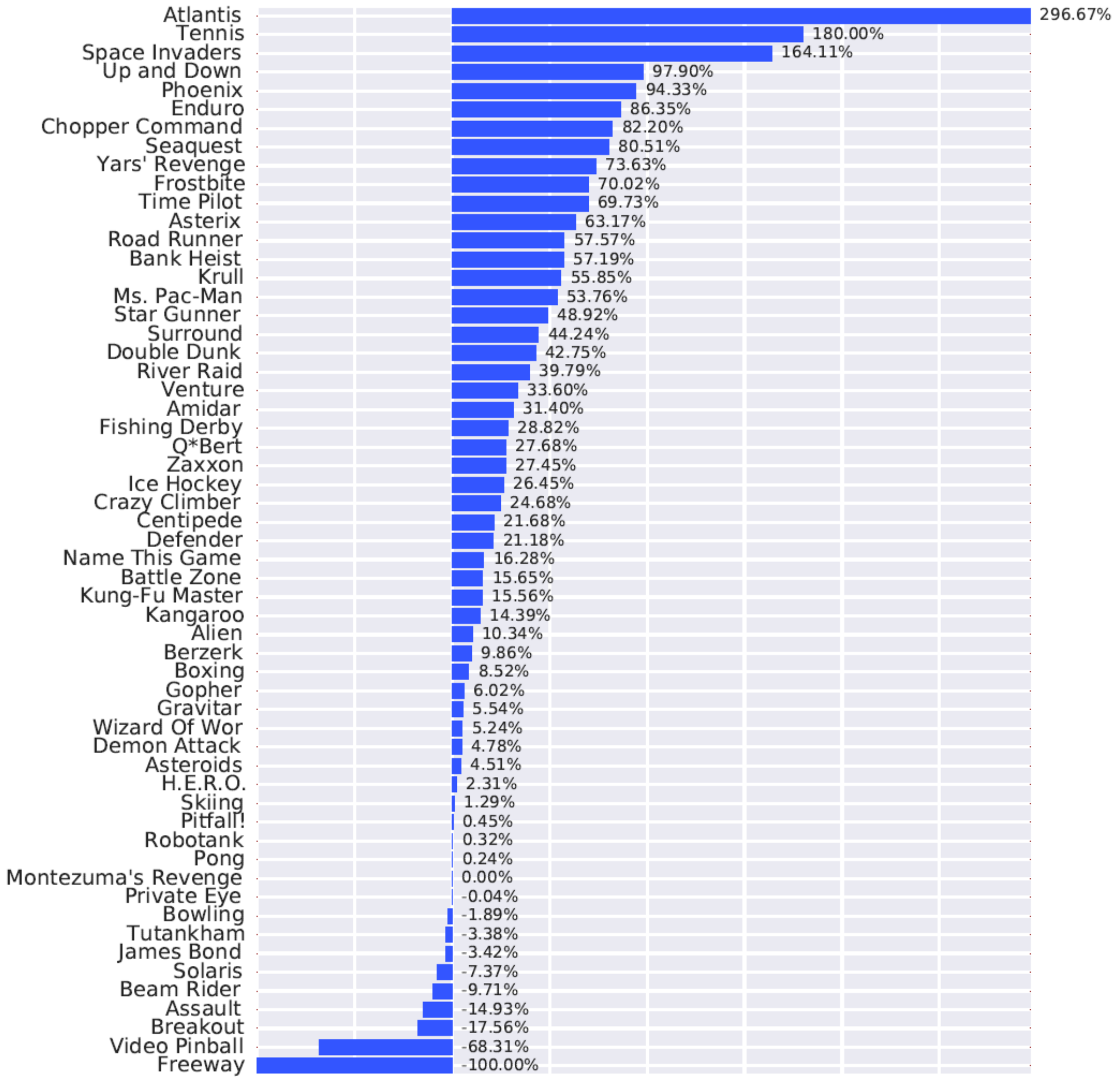


Performance on Atari Games

Baseline : Double DQN

vs.

Dueling Double DQN

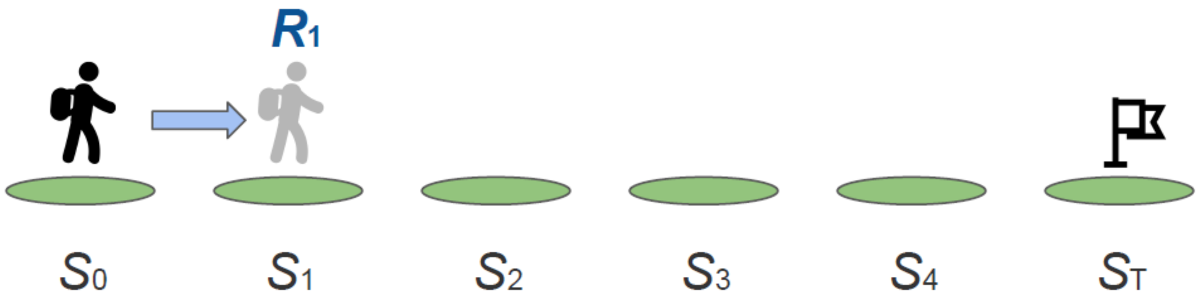


Performance Summary

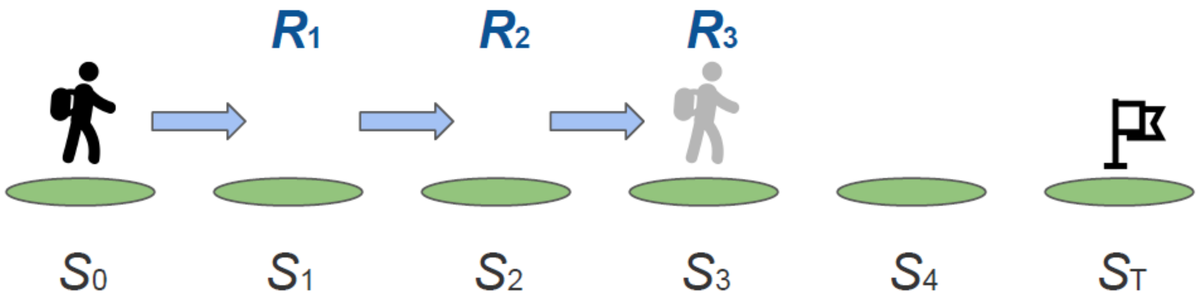
	30 no-ops		Human Starts	
	Mean	Median	Mean	Median
Prior. Duel Clip	**591.9%**	**172.1%**	**567.0%**	**115.3%**
Prior. Single	434.6%	123.7%	386.7%	112.9%
Duel Clip	**373.1%**	**151.5%**	**343.8%**	**117.1%**
Single Clip	341.2%	132.6%	302.8%	114.1%
Single	307.3%	117.8%	332.9%	110.9%
Nature DQN	227.9%	79.1%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

Prior: PER, Clip: clip the gradients, Single : Double DQN

Multi-Step Learning

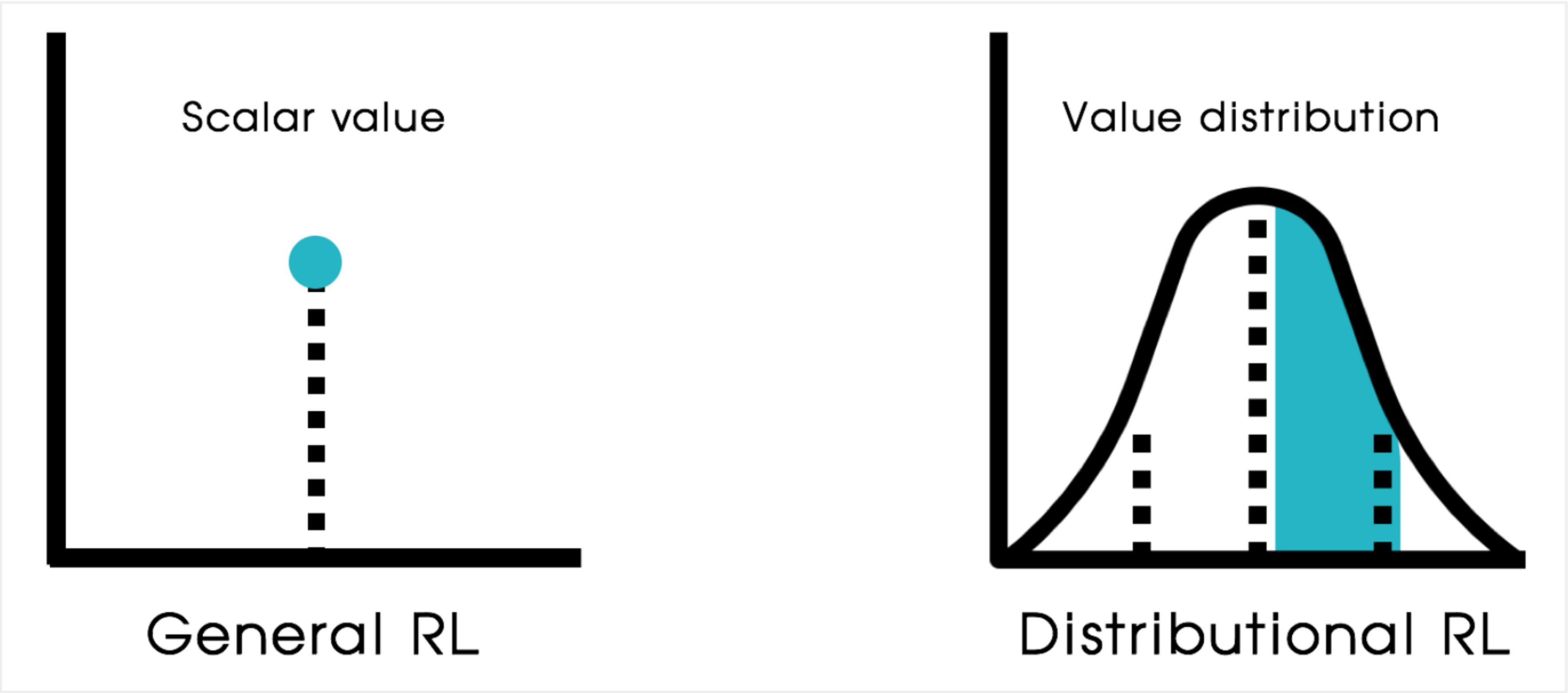


1 step : $R_1 + \gamma \underset{a}{max} Q(S_1, a)$



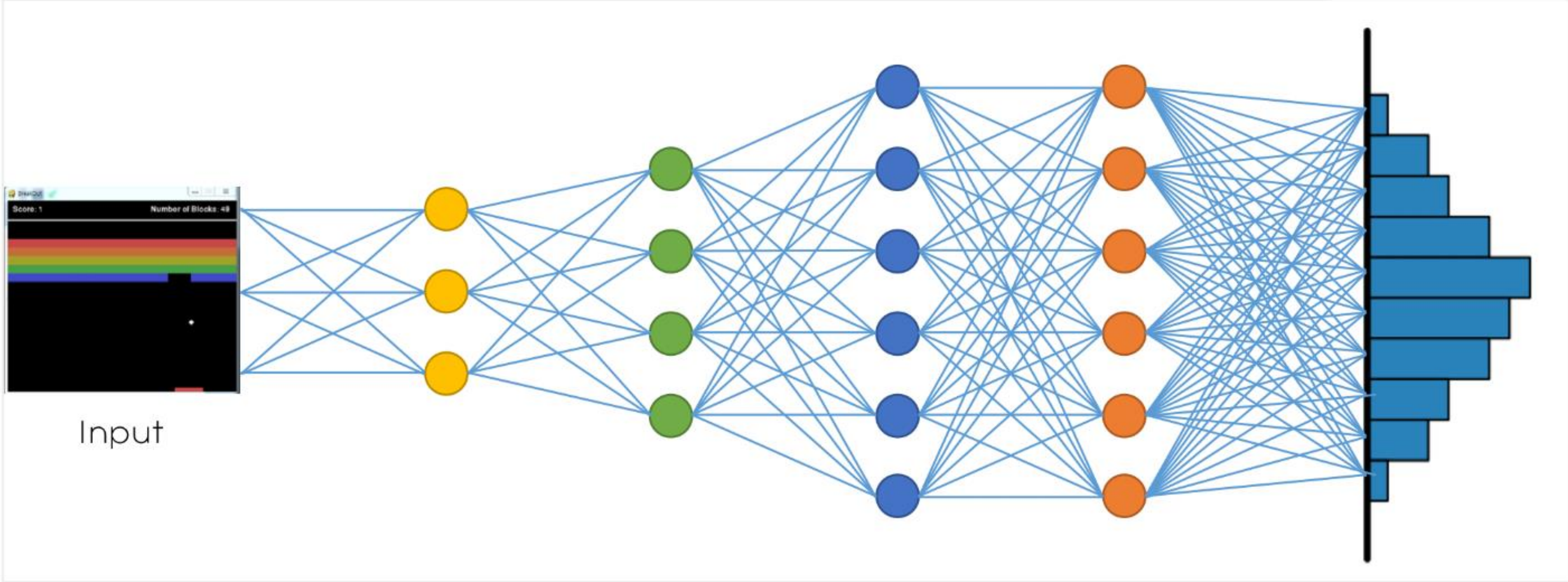
3 step : $R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 \underset{a}{max} Q(S_3, a)$

Distributional RL [ICML 2017]



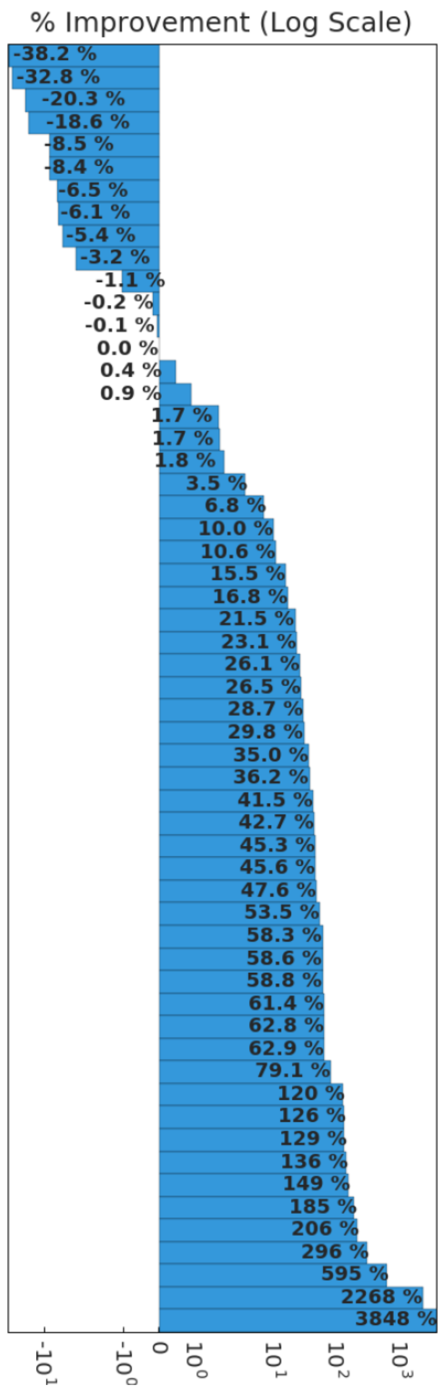
Bellman Equation: $Q(s,a) = R(s,a) + \gamma Q(s',a')$

Distributional Bellman Equation: $Z(s,a) = R(s,a) + \gamma Z(s',a')$

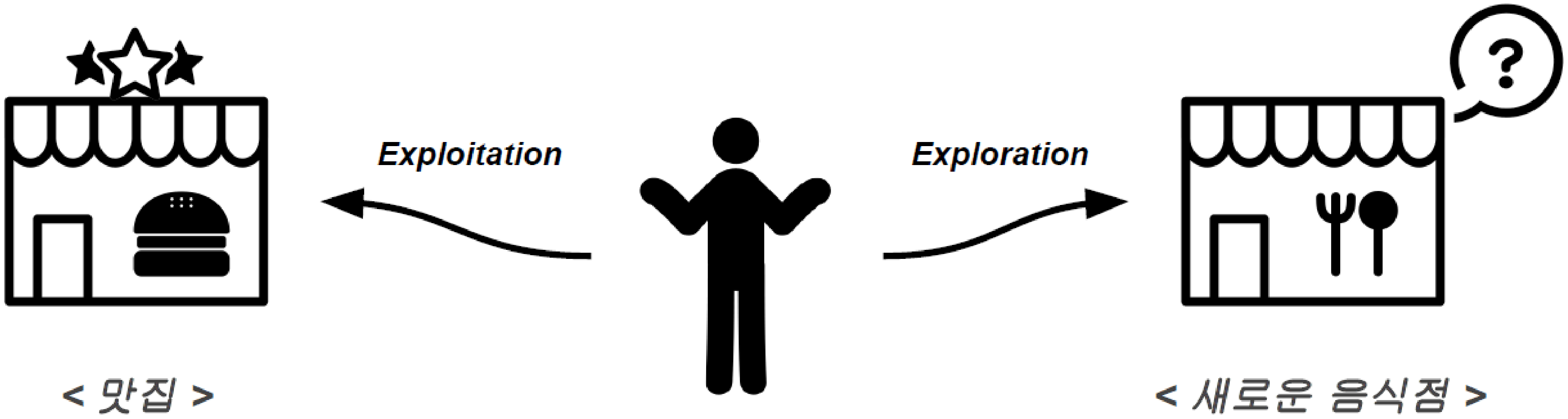


Performance Comparison

	Mean	Median	> H.B.	> DQN
DQN	228%	79%	24	0
DDQN	307%	118%	33	43
DUEL.	373%	151%	37	50
PRIOR.	434%	124%	39	48
PR. DUEL.	592%	172%	39	44
C51	**701%**	**178%**	**40**	**50**



Noisy Network [ICLR 2018]



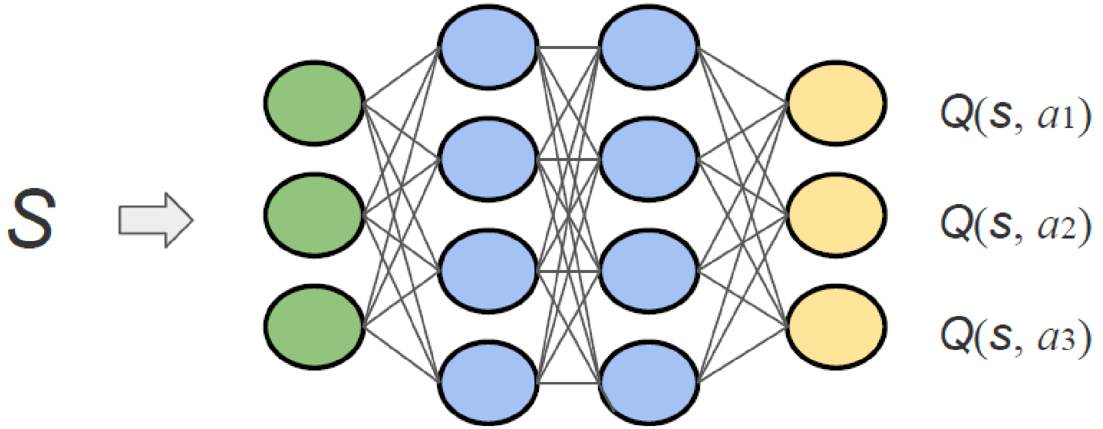
< Exploitation & Exploration >

Looking For Better Exploration

- Can you have more efficient exploration method than ϵ-greedy ?
  - ϵ-greedy is random perturbations of the policy
  - It is inefficient in large-scale behaviors

- Neural networks + Perturbations (Noise)
  - State-dependent exploration

Noisy Network



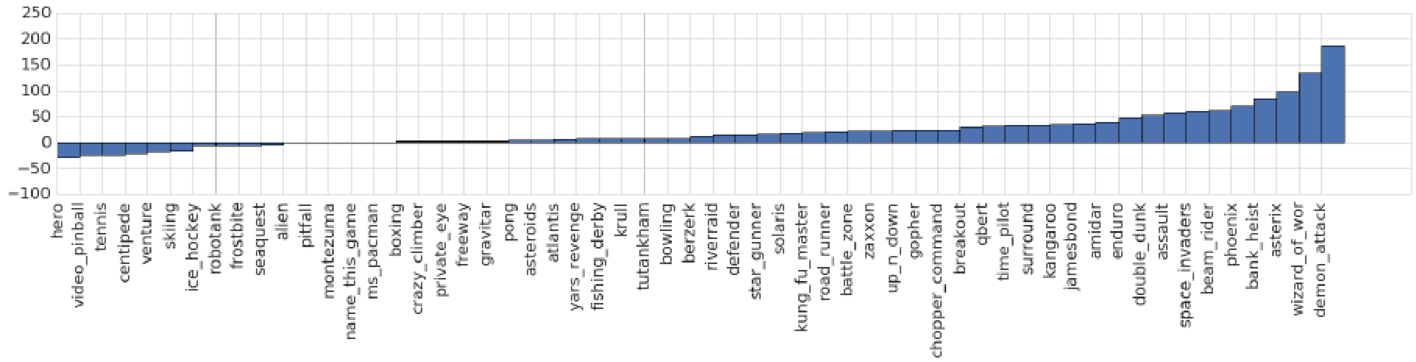
$$y = wx + b$$

$$\downarrow$$

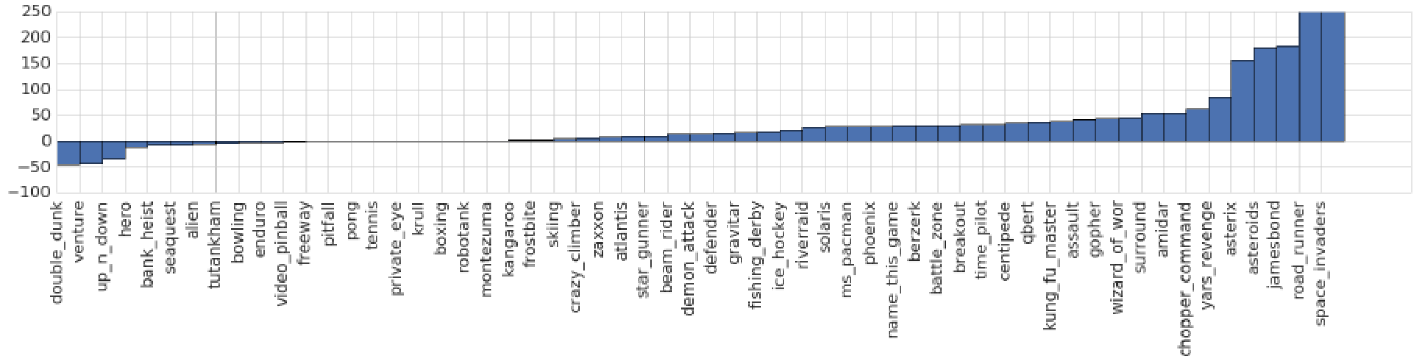
$$y = (\mu^w + \sigma^w \odot \epsilon^w)x + (\mu^b + \sigma^b \odot \epsilon^b)$$

$\odot$: *element-wise multiplication*

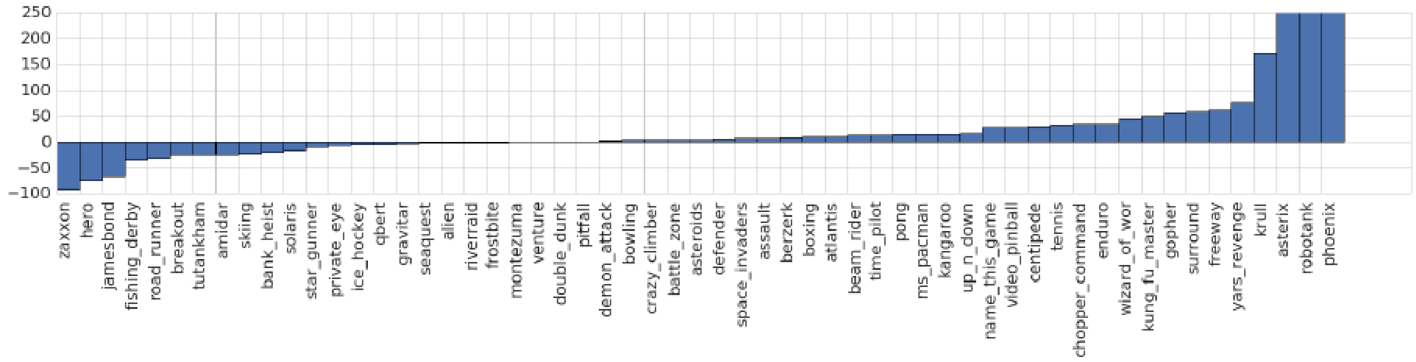
Performance Results



(a) Improvement in percentage of NoisyNet-DQN over DQN (Mnih et al., 2015)



(b) Improvement in percentage of NoisyNet-Dueling over Dueling (Wang et al., 2016)

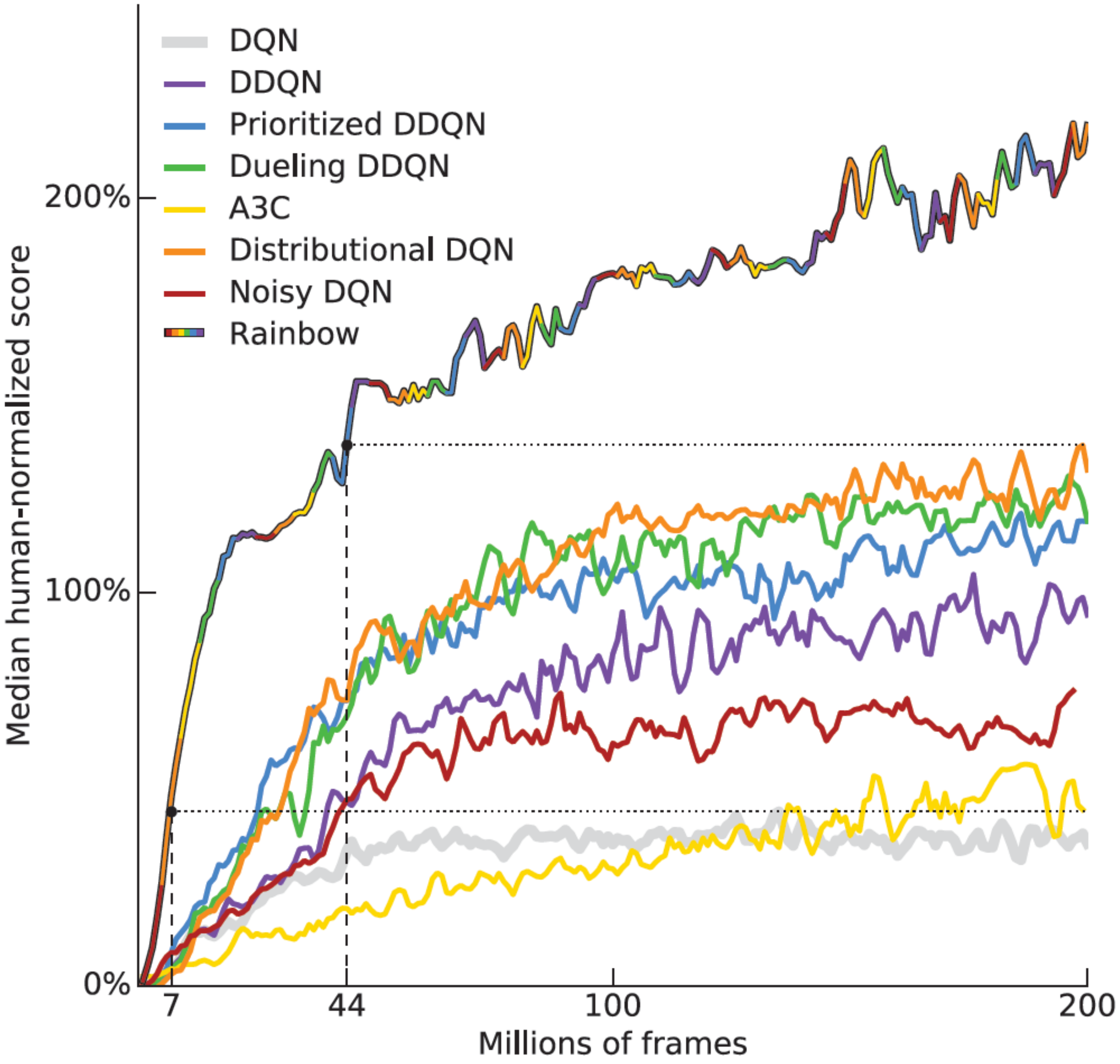


(c) Improvement in percentage of NoisyNet-A3C over A3C (Mnih et al., 2016)

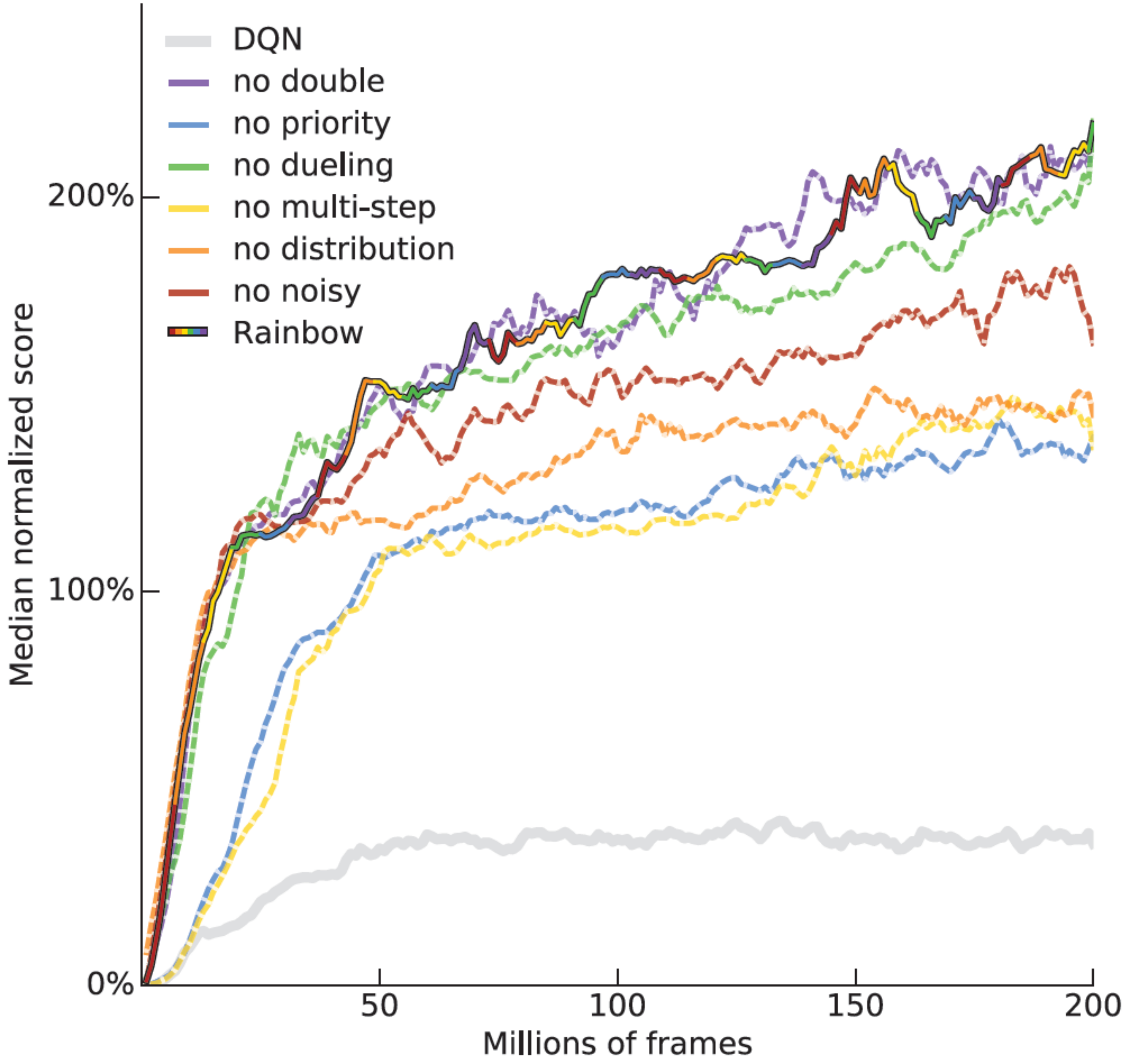
RAINBOW [AAAI 2018]

• DQN	[NIPS 2013], [Nature 2015]
• Double DQN	[AAAI 2016]
• Prioritized Experience Replay	[ICLR 2016]
• Dueling Network Architecture	[ICML 2016]
• Multi-Step Learning	[ICML 2016]
• Distributional Reinforcement Learning	[ICML 2017]
• Noisy Nets	[ICLR 2018]

Performance Comparison across 57 Atari games



Performance Analysis



Reference

[NIPS DQN] Mnih et al, "Playing Atari with Deep Reinforcement Learning," NIPS, 2013.

[Nature DQN] Mnih et al, "Human-level control through deep reinforcement learning," Nature, 2015.

[Double DQN] Hasselt et al, "Deep Reinforcement Learning with Double Q-learning," AAAI, 2016.

[PER] Schaul et al, "Prioritized Experience Replay," ICLR, 2016.

[Deulling DQN] Wang et al, "Dueling Network Architectures for Deep Reinforcement Learning," ICML, 2016.

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

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