Introduction to Reinforcement Learning based Control and Deep Reinforcement Learning based control

Associate Professor
(Maitre de Conférences)
Automatic Control, Reliability of Systems
mayank-shekhar.jha{at}univ-lorraine.fr

Research
Centre de Recherche en Automatique de Nancy
(CRAN) UMR 7039,
Faculté des Sciences et Technologies
Boulevard des Aiguillettes - BP 70239 - Bât.
1er cycle 54506 Vandoeuvre-lès-Nancy
Cedex France

Teaching
Polytech Nancy (ESSTIN),
2 Rue Jean Lamour 54509
Vandoeuvre-lès-Nancy,
Cedex France

Email: mayank-shekhar.jha [at] univ-lorraine.fr

Introduction

Markov Decision process

Dynamic Programming for planning and control

Model Free Prediction and Control (SARSA and Q-learning)

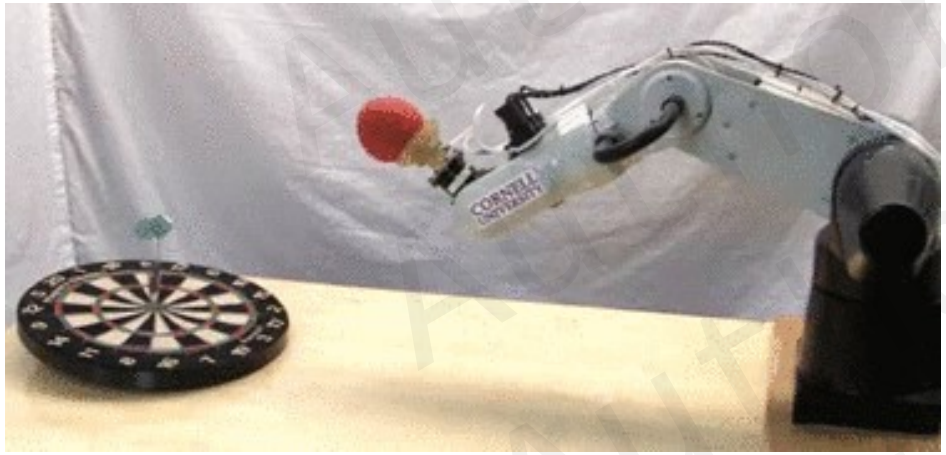
Function Approximation (Use Deep Neural Networks)

Discrete vs Continuous states

Temporal Difference (TD)

JHA Mayank , Email: mayank-shekhar.jha [at] univ-lorraine.fr

Discrete Space vs Continuous Space



Discrete Space

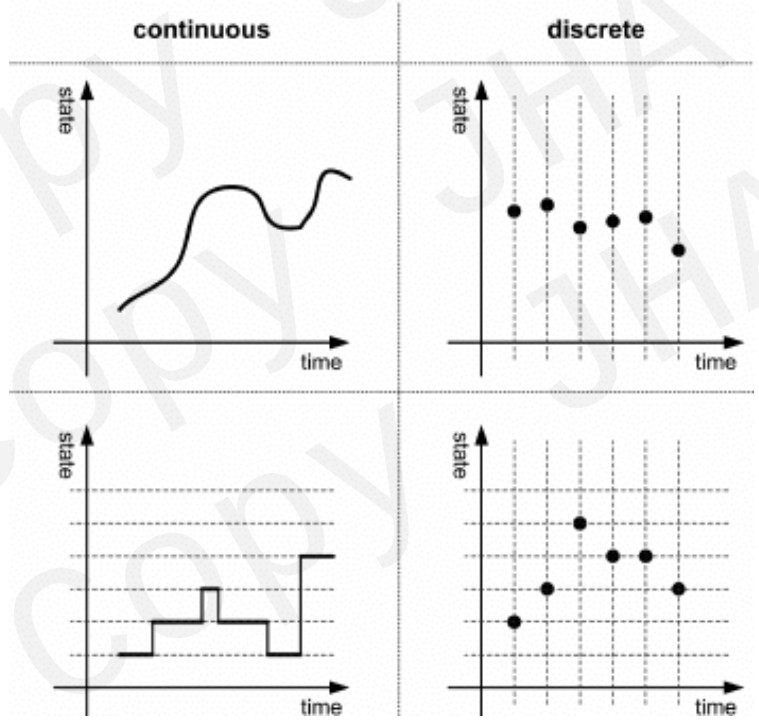
States: $s \in \{s_0, s_1, \dots s_n\}$

Actions: $a \in \{a_0, a_1, \dots a_m\}$

Continuous Spaces

Stats: $s \in \mathbb{R}^n$

Actions: $a \in \mathbb{R}^n$



Discrete Spaces (Value Representation).

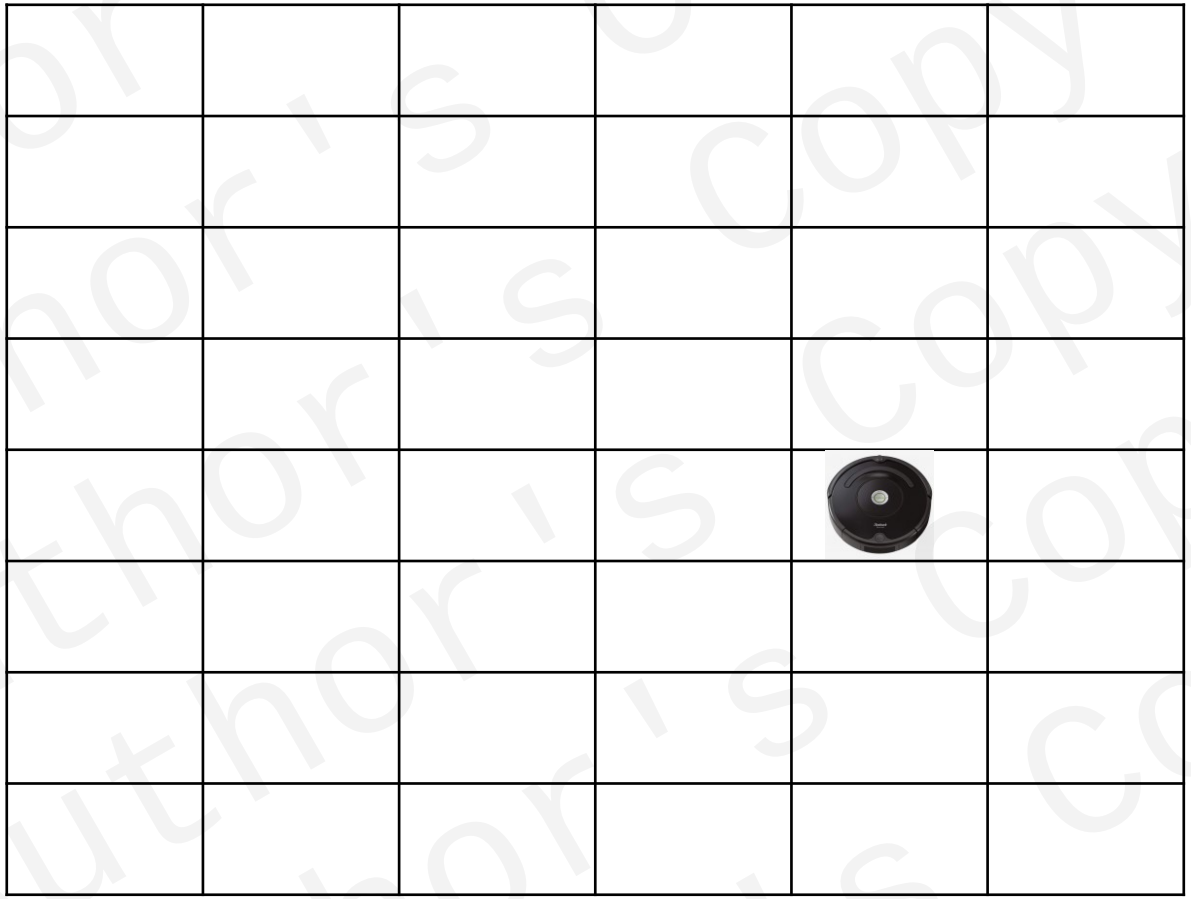
$V: S \rightarrow \mathbb{R}$

$V = \{$
$0: 0.9,$
$1: 0.1,$
$2: 0.5,$
$3: 0.7,$
$\}$

$Q: S \times A \rightarrow \mathbb{R}$

$Q = [[0.3, 0.1, 0.4, \ldots]$
$[0.2, 0.7, 0.5, \ldots]$
$[0.8, 0.2, 0.6, \ldots]$
$[\ \ldots\ ]$
$[$
$[$
$]]$

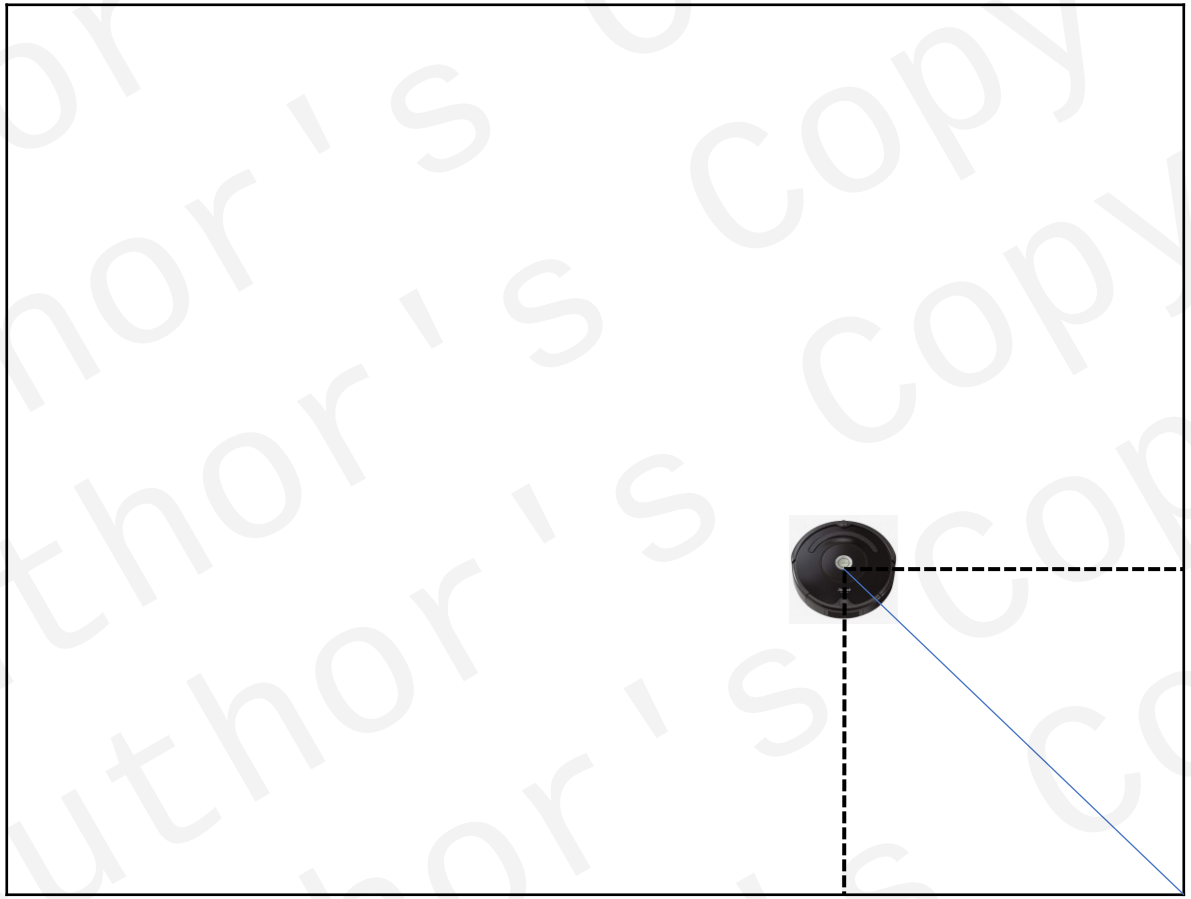
Why Continous ?

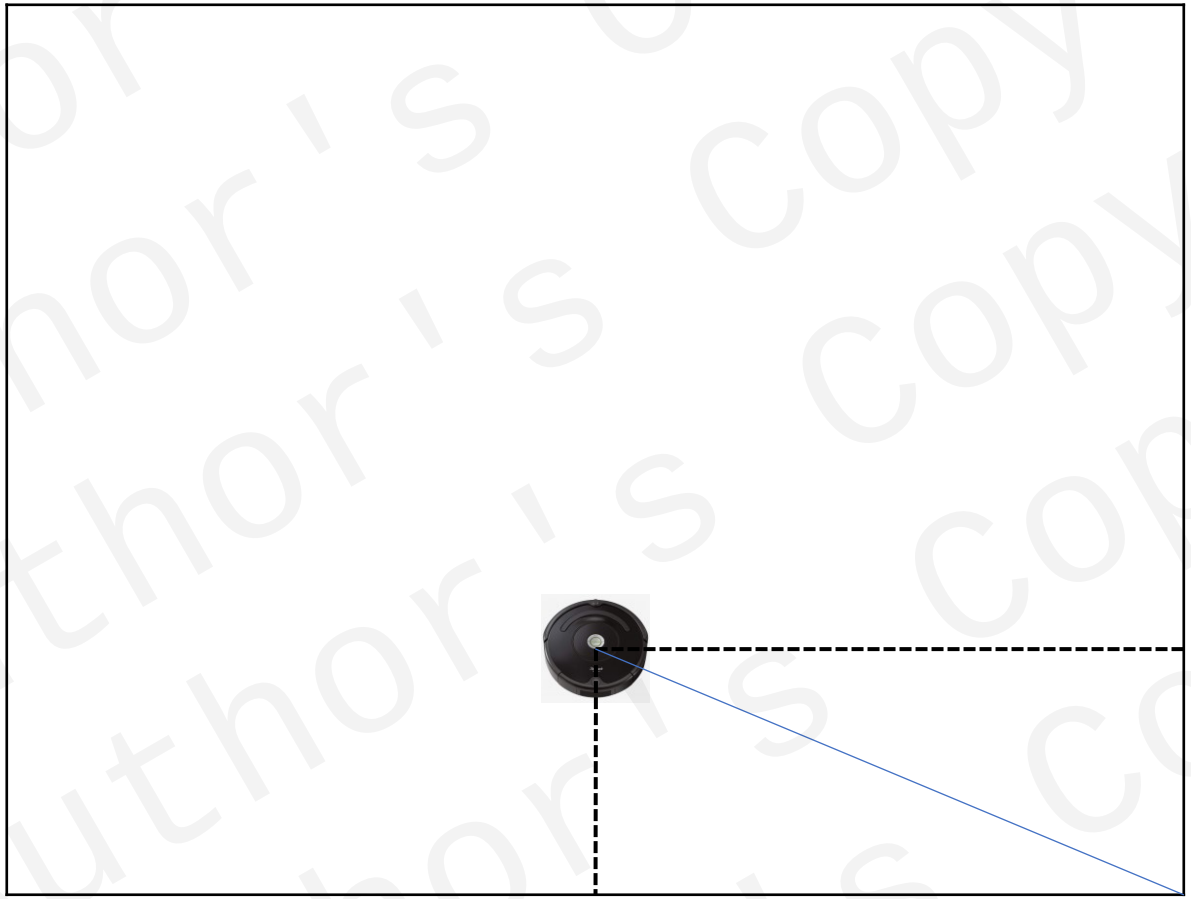


→ In real world,

position
speed
⋮
!

} Continuous variables.





Why Continuous Spaces

Function Approximation

What we want : Close approximation of true functions

$\hat{v}(s) \approx v_\pi(s)$

$\hat{q}(s,a) \approx q_\pi(s,a)$

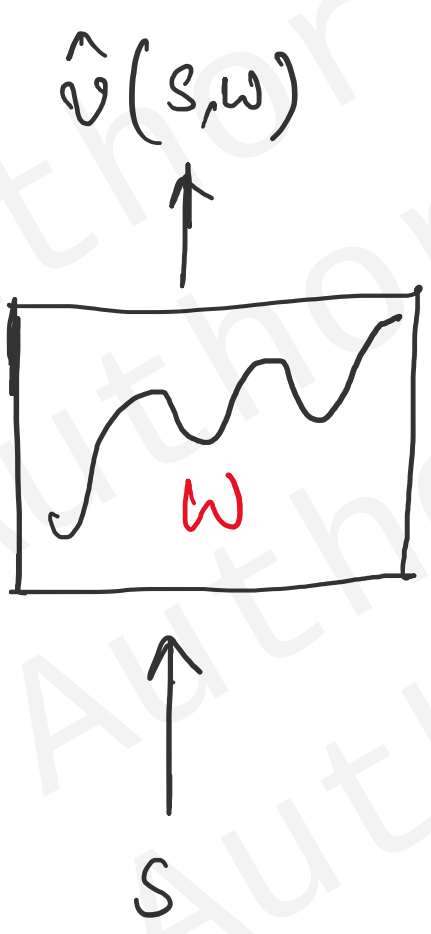
Approximate the true fn using 'approximators'

$\hat{v}(s, \underline{w}) \approx v_\pi(s)$

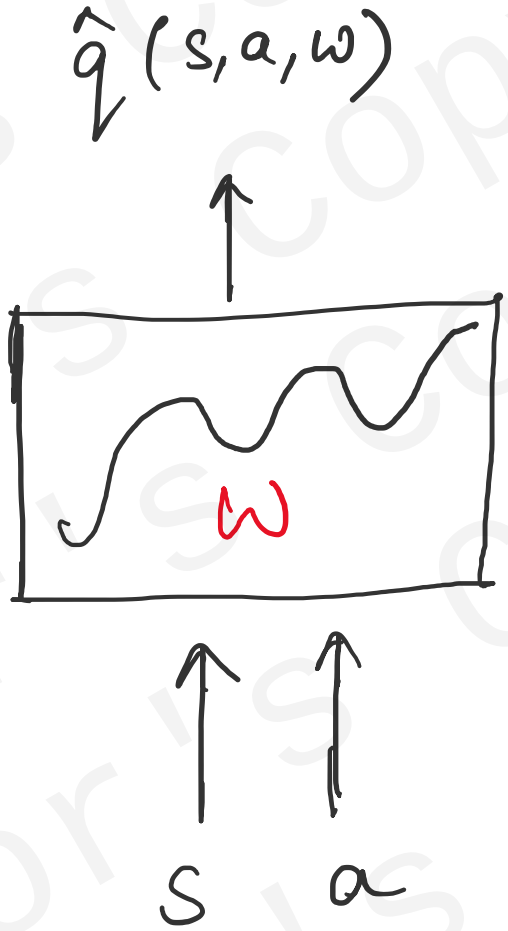
$\hat{q}(s, a, \underline{w}) \approx q_\pi(s,a)$

→ Find good 'w' iteratively !!

Function Approximation



State Value Approx



State - Action Value Approx

State to different Q-values.

State Value Approx

$\hat{v}(s, w) = ?$

state

parameters

How to map state, weights to value function?

state vector

→ states usually represented by feature vector

$$x(s) = \begin{pmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

State Value Approx

$$\hat{v}(s,w) = \underbrace{x(s)^T \cdot w}_{\text{simplest approach.}}$$

$$= x_1(s)\cdot w_1 + x_2(s)\cdot w_2 + \cdots x_n(s)\cdot w_n$$

$$= \sum_{j=1}^{n} x_j(s)\cdot w_j$$

tate vector

→ states usually

(Linear Function Approximation)

State Value Approx (Linear Fn Approx)

Value Fn ⇒ $\hat{v}(s,w) = x(s)^T \cdot w$

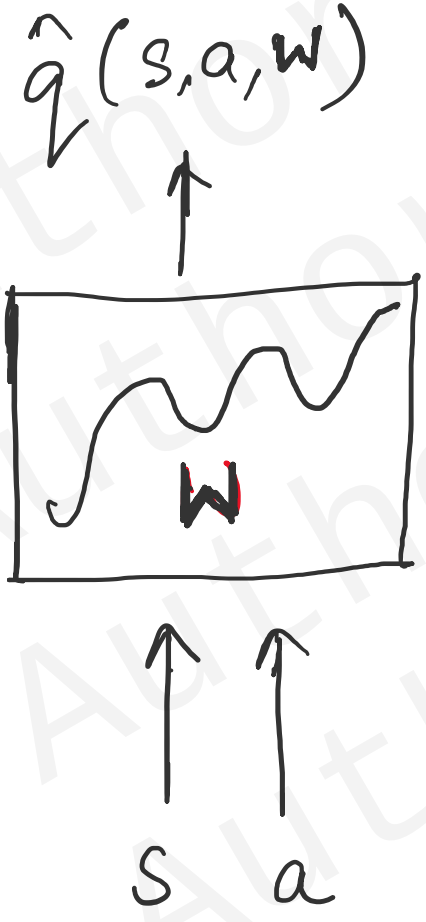
Minimize Error ⇒ $J(w) = \mathbb{E}_\pi\left[\left(v_\pi(s) - x(s)^T w\right)^2\right]$

Error Gradient ⇒ $\nabla_w J(w) = -2\left(v_\pi(s) - x(s)^T \cdot w\right) x(s)$

Update Rule ⇒ $\Delta w = -\alpha \frac{1}{2} \nabla_w J(w)$

$= \alpha\left(v_\pi(s) - x(s)^T \cdot w\right) x(s)$

Q-Value Approximation (Action-Value Approx)



$\hat{q}(s,a,w) = ?$

Action-Value Function

$x(s,a) = \begin{pmatrix} x_1(s,a) \\ x_2(s,a) \\ \vdots \\ x_n(s,a) \end{pmatrix}$

State/Action feature vector

→ Use Gradient Descent for weight adaptation !!

Q-value Approximation (Action-Vector Approx)

$\hat{q}(s, a_1, w), \ldots \hat{q}(s, a_m, w)$

$\hat{q}(s, a_1, w) = ?$

$\ldots$

$\hat{q}(s, a_m, w) = ?$

$$X(s,a) = \begin{pmatrix} x_1(s,a) \\ x_2(s,a) \\ \vdots \\ x_n(s,a) \end{pmatrix}$$

Q-Value Approximation (Action - Vector Approx)

$\hat{q}(s, a_1, w), \ldots \hat{q}(s, a_m, w)$

$$\hat{q}(s, a_1, w)$$

$$= \begin{pmatrix} x_1(s,a) & \ldots & x_n(s,a) \end{pmatrix} \cdot \begin{pmatrix} w_{11} & \ldots & w_{1m} \\ \vdots & & \vdots \\ w_{n1} & \ldots & w_{nm} \end{pmatrix}$$

$$= \begin{pmatrix} \hat{q}(s, a_1, w) & \ldots & \hat{q}(s, a_m, w) \end{pmatrix}$$

Non- linear Function Approximation

$$\hat{v}(s,w) = f(x(s)^T \cdot w)$$

Deep Neural Network

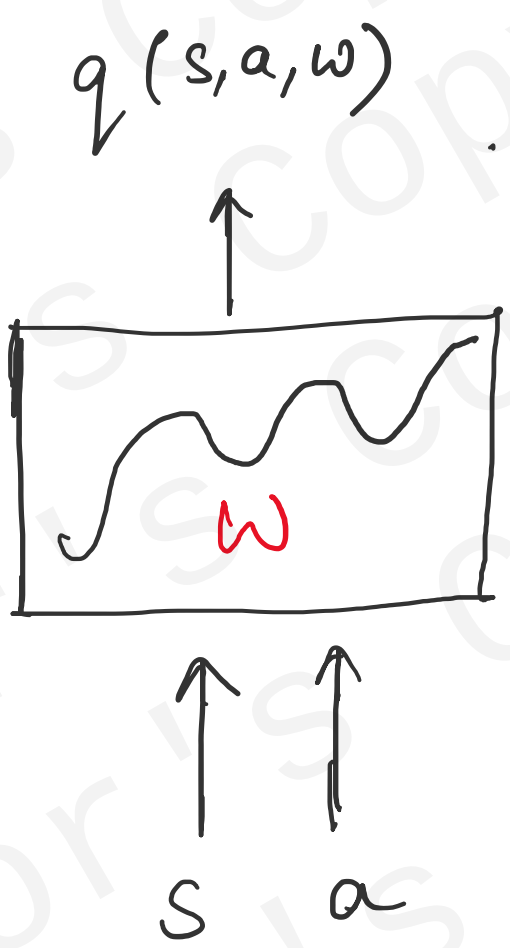
$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

Weight Adaptation : (iterative)

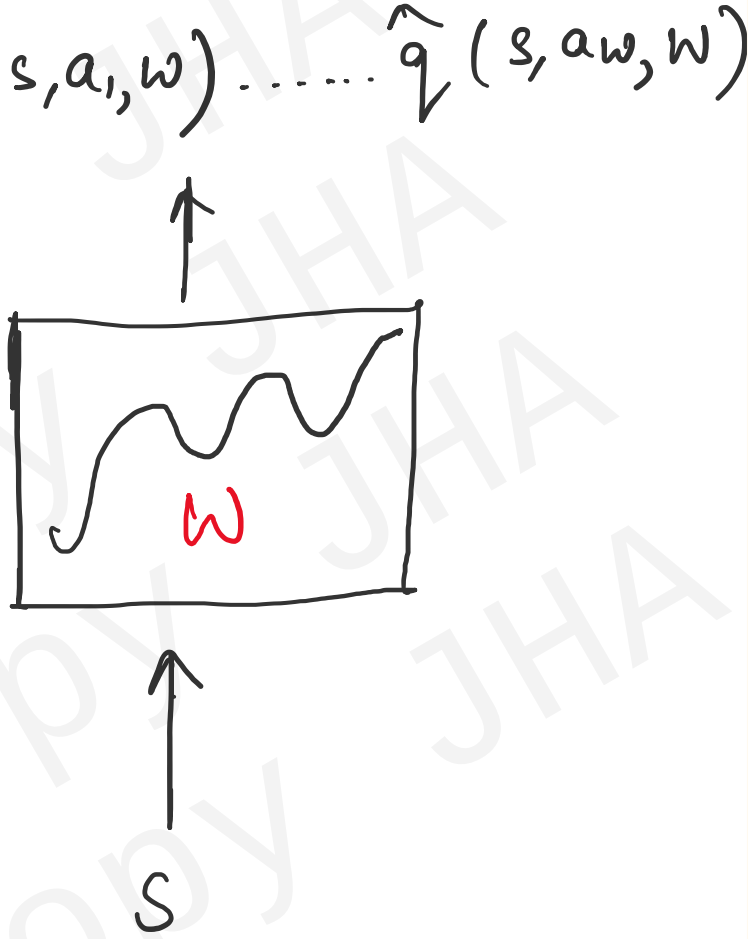
$$\Delta w = \alpha \left(v_\pi(s) - \hat{v}(s,w) \right) \nabla_w \hat{v}(s,w)$$

Q-Networks

Generalise from seen states to Unseen states :



State - Action Value Approx



State to different Q-values.

Q- Networks

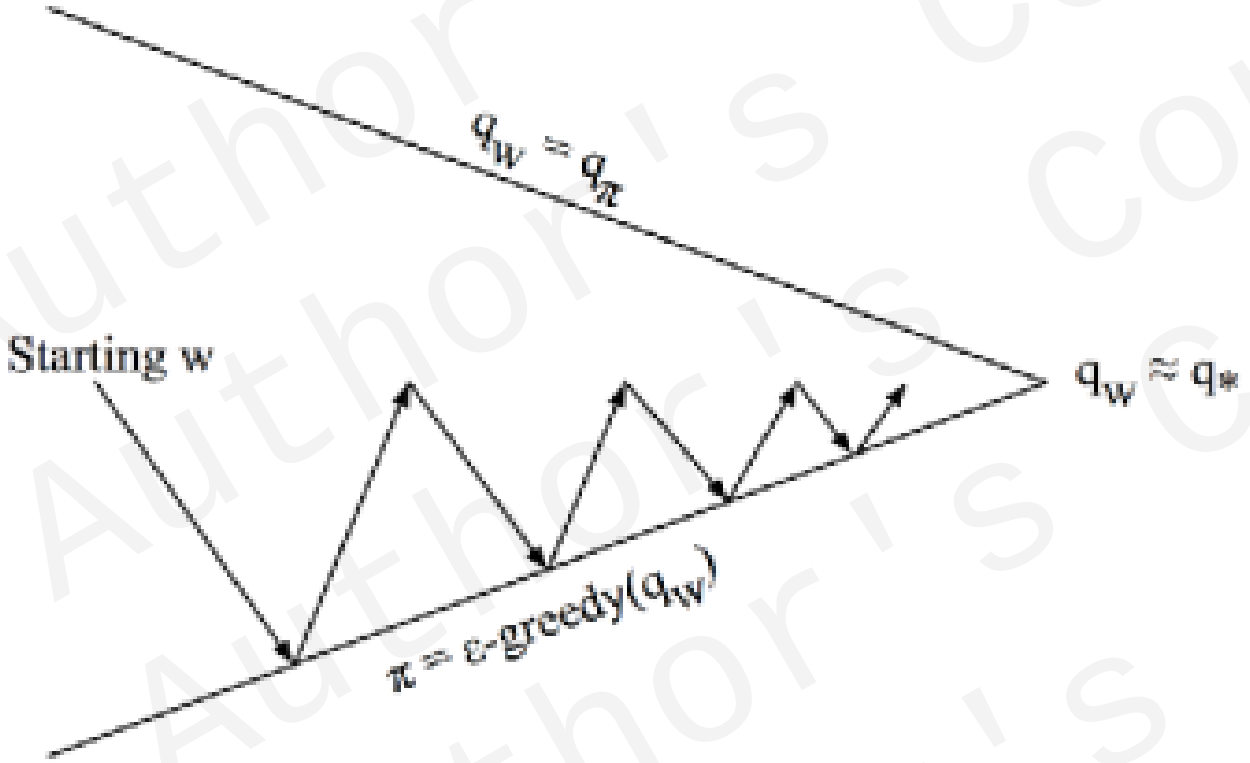
Goal: find parameter vector 'w' to minimize MSE between approximate value function ↕ true value function.

Use TD(0) :

TD target : $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

Weight update:

$$\Delta w = \alpha \left(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w) \right) \nabla_w \hat{q}(S_t, A_t, w)$$



Q-Network

- Optimale Q-values should obey Bellman Equation.

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q(s',a')^* \mid s,a\right]$$

- Target (RHS as target)

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

- Minimize MSE loss by SGD.

$$l = \left(\underbrace{r + \gamma \max_{a} Q(s',a',w)}_{\text{target}} - Q(s,a,w)\right)^2$$

- Minimize MSE loss by SGD.

$$l = \left(\underbrace{r + \gamma \max_a Q(s', a', w)}_{\text{target}} - Q(s, a, w) \right)^2$$

- Converges to Q^* using look up table.
- Diverges using Neural networks
  - → Correlation between samples
  - → Non-stationary targets.

Introduction

Markov Decision process

Dynamic Programming for planning and control

Model Free Prediction and Control (SARSA and Q-learning)

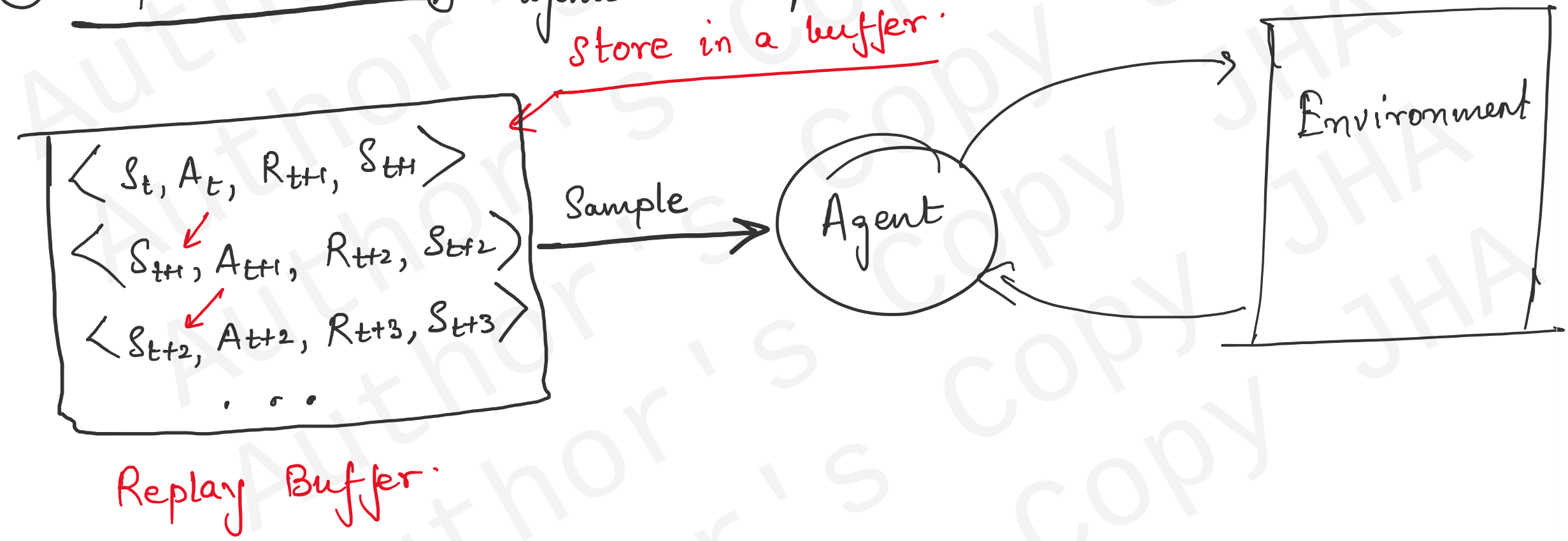
Function Approximation (Use Deep Neural Networks)

Deep Q networks (Deep learning for RL)

Deep Q- Networks (DQN)

① Experience Replay: Build data set from agent's own experience.

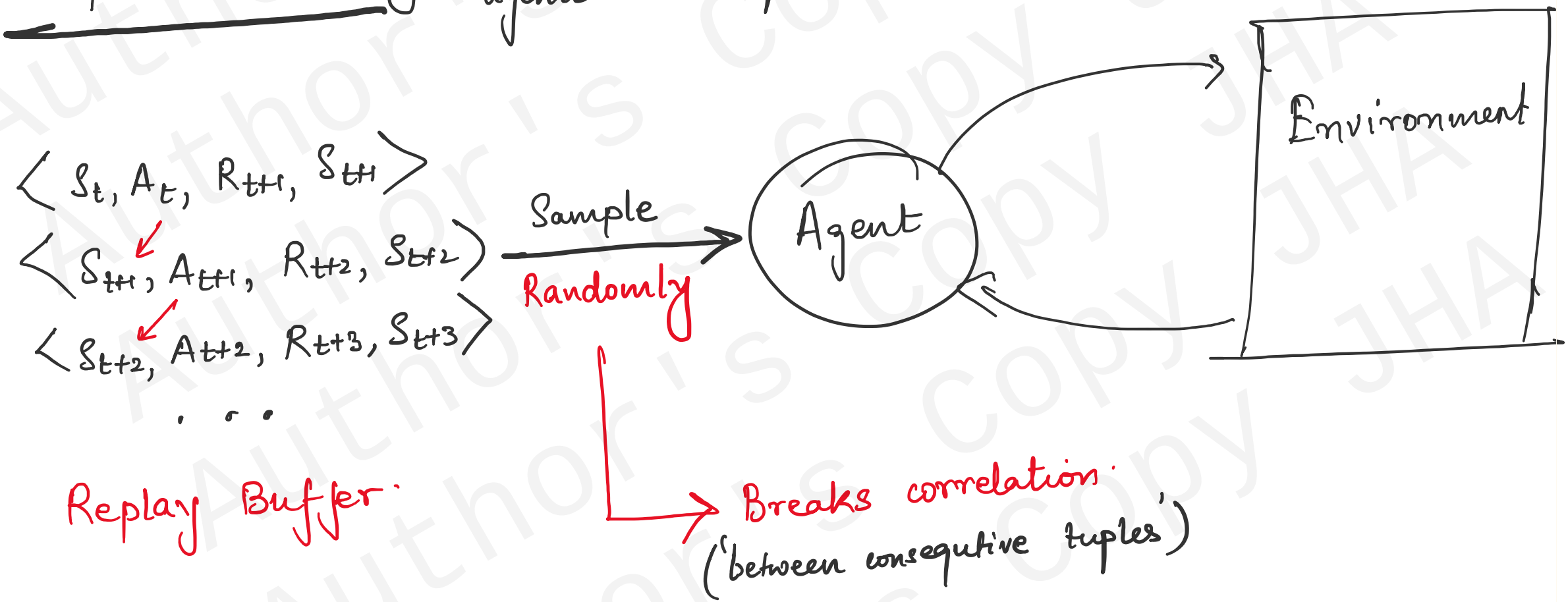
Store in a buffer.



Replay Buffer.

Deep Q- Networks (DQN)

1. Experience Replay: Build data set from agent's own experience.



Deep Q-Networks (DQN)

(2.) Fixed Q-targets:

$$\Delta w = \alpha \left(r + \gamma \max_{a} \hat{q}(s', a, w) - \hat{q}(s, a, w) \right) \nabla_w \hat{q}(s, a, w)$$

$q_\pi(s, a)$

$$= \alpha \left(q_\pi(s, A) - \hat{q}(s, A, w) \right) \nabla_w \hat{q}(s, A, w)$$

↳ updating same network by same network!!

↳ The target is moving!!

Deep Q-Networks (DQN)

(2.) Fixed Q-targets.

$$\Delta w = \alpha \left(r + \gamma \max_a \hat{q}(s', a, w) - \hat{q}(s, a, w) \right) \nabla_w \hat{q}(s, a, w)$$

$q_\pi(s, a)$

$$\Delta w = \alpha \left(q_\pi(S, A) - \hat{q}(S, A, w) \right) \nabla_w \hat{q}(S, A, w)$$

Target keeps moving with changing weights!

Deep Q-Networks (DQN)

(2.) Fixed Q-targets:

$$\Delta w = \alpha \left(r + \gamma \max_a \hat{q}(s', a, \bar{w}) - \hat{q}(s, a, w) \right) \nabla_w \hat{q}(s, a, w)$$

$\bar{w}$: fixed (hold weights fixed for some updates)

Use a Target Q network

Decouple update and target TD

Δw

$r + \gamma \max_a \hat{q}(s', a, \bar{w})$

DQN (Algorithm)

- Initialize replay memory D with capacity N.
- Initialize action-value function $\hat{q}$ with random weights 'w'
- Initialize targets action-value weights $w^- \leftarrow w$.
- for the episode $e \leftarrow 1$ to M:
  - intial input state x_1
  - prepare initial state: $s \leftarrow \phi(\langle x_1 \rangle)$
  - for time step $t \leftarrow 1$ to :

Sample
- → choose action A from state S using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S, A, w))$
- → Take action A, observe reward r, next state x_{t+1}
- → Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
- → Store experience tuple (S, A, R, S') in replay memory D
- $S \leftarrow S'$

DQN (Algorithm)

- Initialize replay memory D with capacity N.
- Initialize action-value function $\hat{q}$ with random weights w
- Initialize targets action-value weights $w^- \leftarrow w$.
- for the episode $e \leftarrow 1$ to M:
  - intial input state x_1
  - prepare initial state: $s \leftarrow \phi(\langle x_1 \rangle)$
  - for time step $t \leftarrow 1$ to T:

Sample

- choose action A from state S using policy $\pi \leftarrow \epsilon$-Greedy $(\hat{q}(S, A, w))$
- Take action A, observe reward r, next state x_{t+1}
- Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
- Store experience tuple (S, A, R, S') in replay memory D
- $S \leftarrow S'$

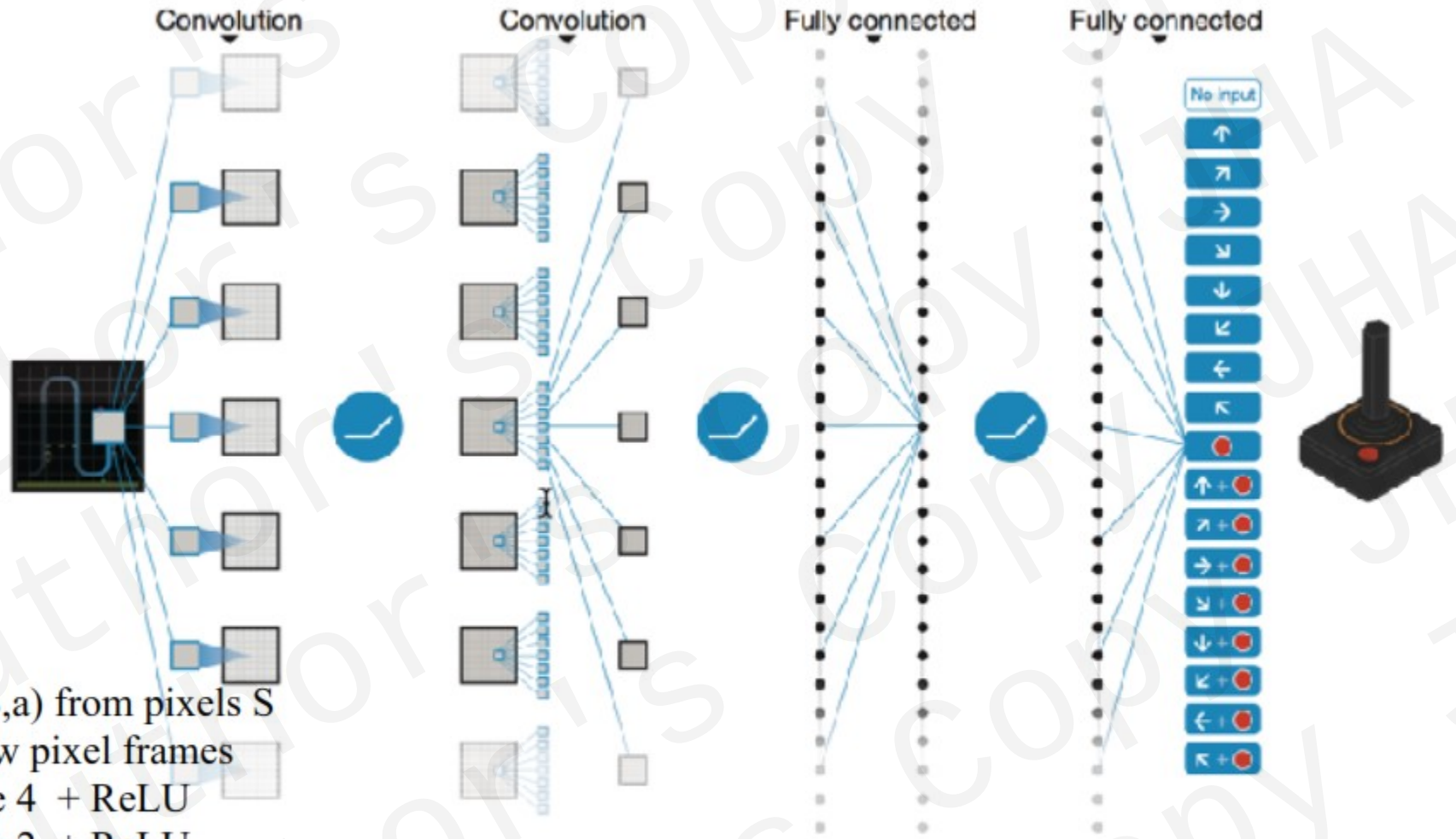
Learn

Obtain random minibatches of tuples (s_j, a_j, r_j, s_{j+1}) from D

Set target $y_j = r_j + \max_a \hat{q}(s_{j+1}, a, w^-)$

Update: $\Delta w = \alpha (y_j - \hat{q}(s_j, a_j, w)) \nabla_w \hat{q}(s_j, a_j, w)$

Every 'C' steps, reset: $w^- \leftarrow w$.



end-to-end learning Q(S,a) from pixels S
input state: 4 stacked raw pixel frames
32 8x8 filters with stride 4 + ReLU
64 4x4 filters with stride 2 + ReLU
64 3x3 filters with stride 1 + ReLU
fully connected 512 units + ReLU
fully connected output units, one per action

[Source: Mnih et al., Nature 2015 (DeepMind)]

Improvements to DQN

① Double DQN : Remove the bias caused by $\max_a Q(s,a,w)$

→ Current Q-Network → w used to select actions.

→ Older Q-network w^- used for evaluate actions.

$$l = \left(r + \gamma\, q\left(s', \underset{a'}{\operatorname{argmax}}\, q(s',a',w), w^-\right) - q(s,a,w) \right)^2$$

select actions

evaluate the actions.

② Prioritised Replay :

→ Weight experience according to surprise.

→ store experience in priority queue according to DQN error.