

Deep Reinforcement Learning for Value and Q Functions

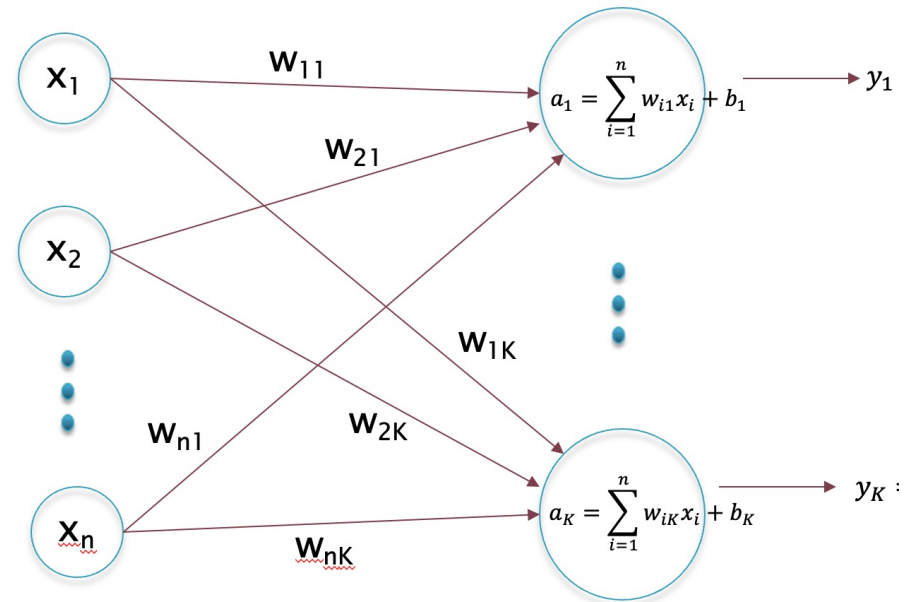
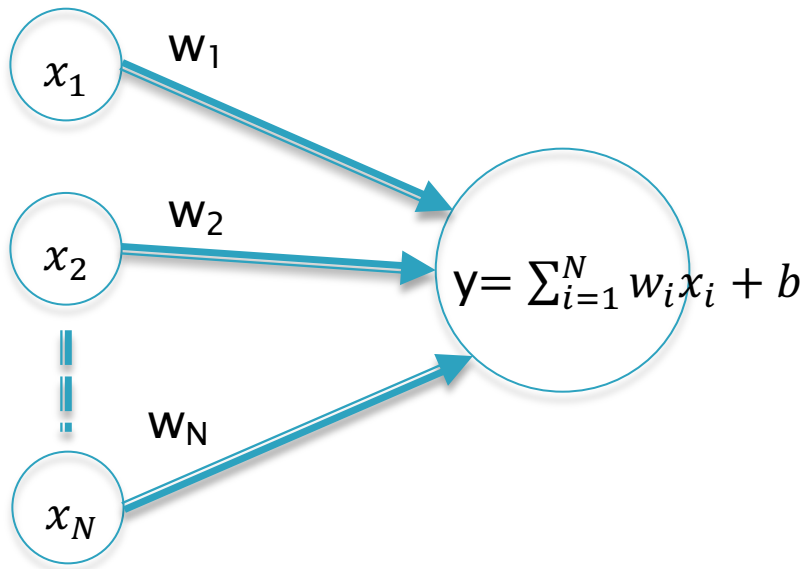
Lecture 7

Subir Varma

Summary – Regression

Choose weights to
Minimize Error

$$L(W) = \frac{1}{2MK} \sum_{j=1}^M \sum_{k=1}^K (t_k(j) - y_k(j))^2$$



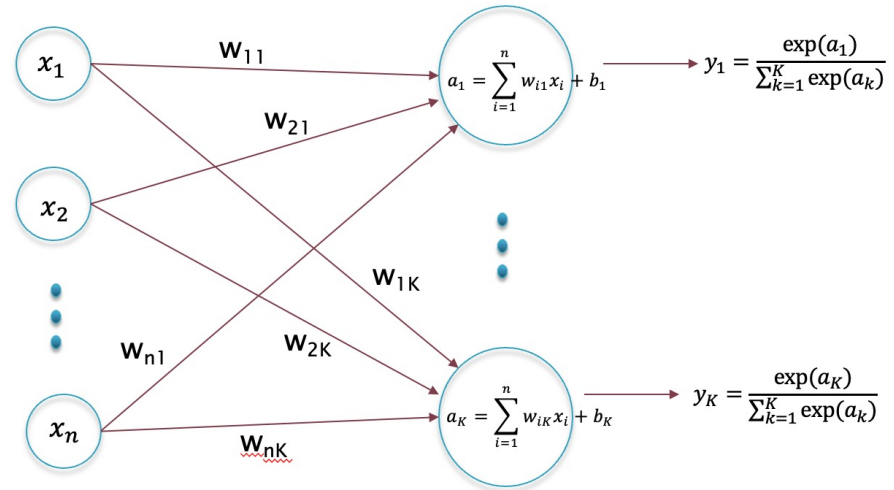
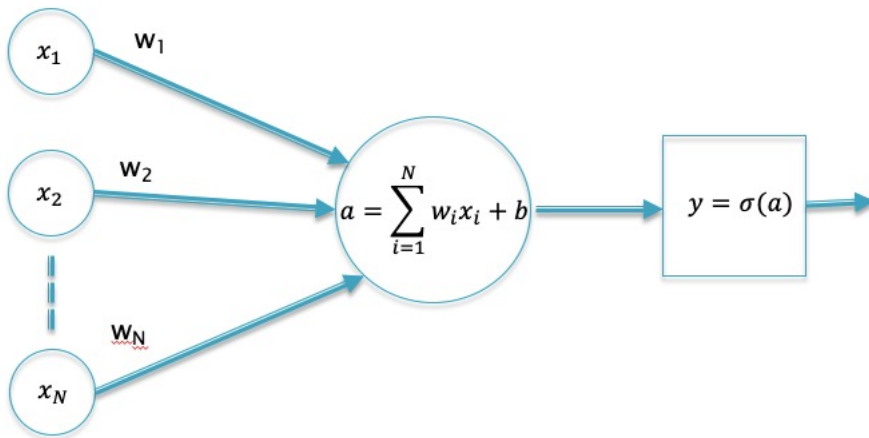
$$w_i \leftarrow w_i - \eta x_i (y - t)$$

$$w_{ik} \leftarrow w_{ik} - \eta x_i (y_k - t_k)$$

Summary – Logistic Regression

Choose weights to
Maximize Reward

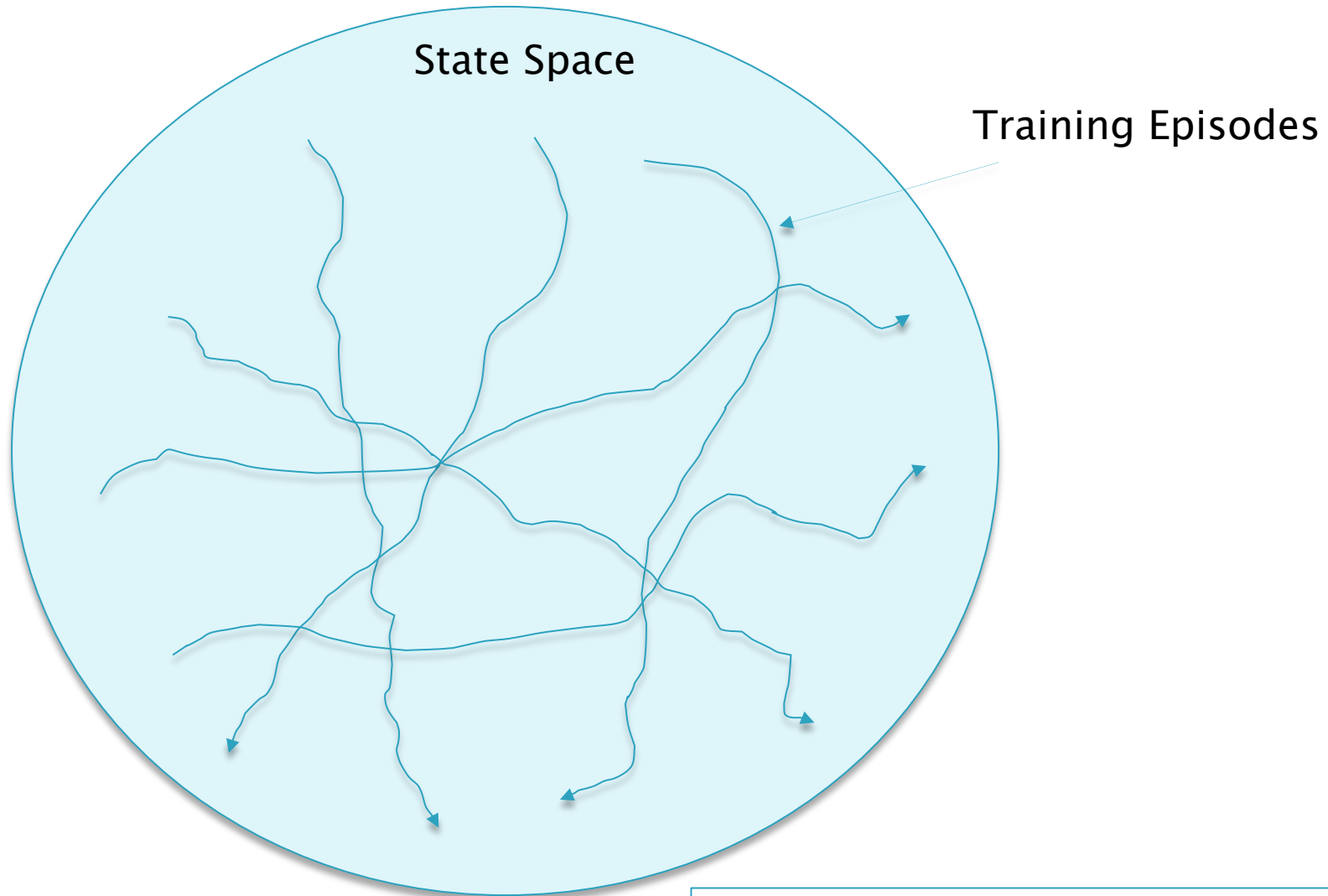
$$L(W) = \frac{1}{M} \sum_{j=1}^M \sum_{k=1}^K t_k(j) \log y_k(j)$$



$$w_i \leftarrow w_i - \eta x_i (y - t)$$

$$w_{ik} \leftarrow w_{ik} - \eta x_i (y_k - t_k)$$

Functional RL



Neural Network approximates the Value Function for parts of the State space outside the sample episodes

Deep RL with Q Functions: High Level Approach

Run Sample Episodes from the System

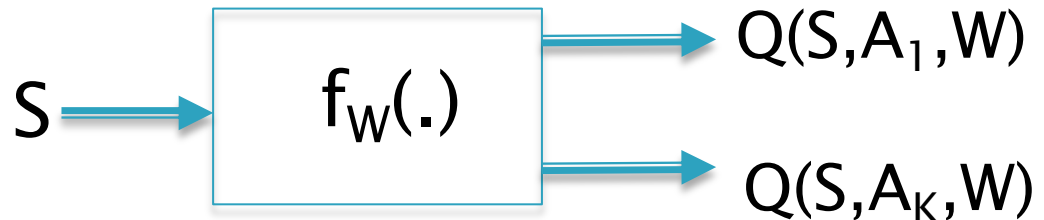


Use Monte Carlo, SARSA or Q-Learning to get samples of the mapping $(S, A) \rightarrow Q(S,A)$. This becomes the Training Data

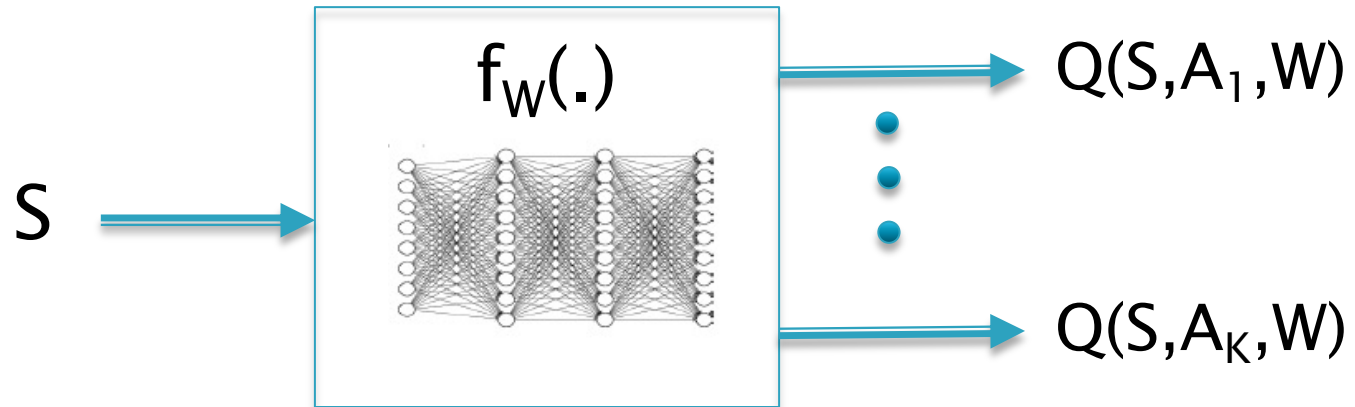


Update the Neural Network weights So that $Q(S,A)$ and $Q(S,A,W)$ move closer

We replace Table updates with NN Weight updates using Gradient Descent



Functional Reinforcement Learning

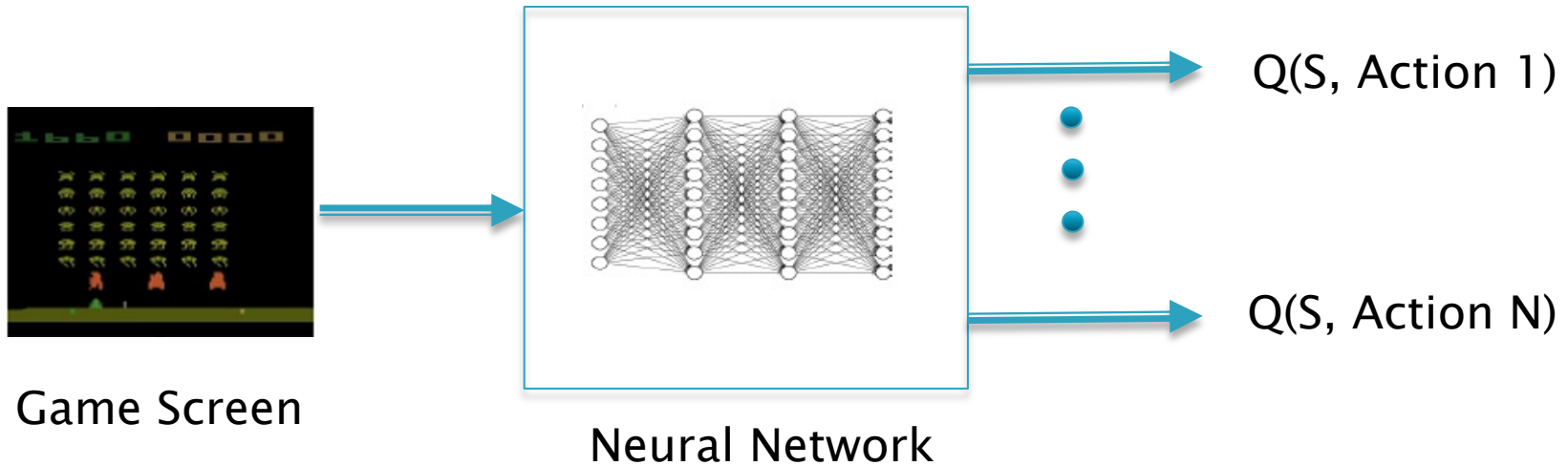


f_W is represented using a multilayer Neural Network

Benefits

- 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 $P, R/V/Q/\pi$

Functional Reinforcement Learning

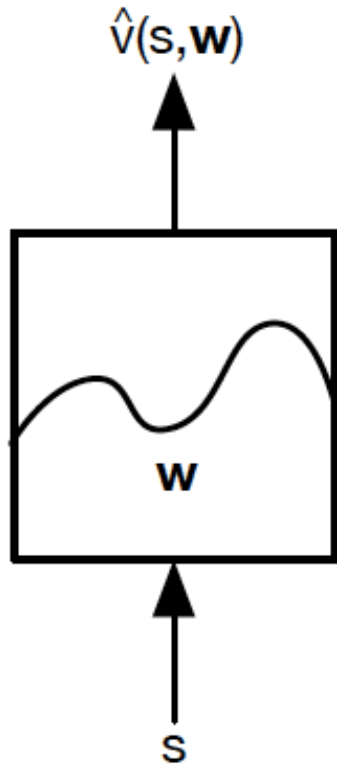


Learn Optimal Actions Directly from the Game Screen!

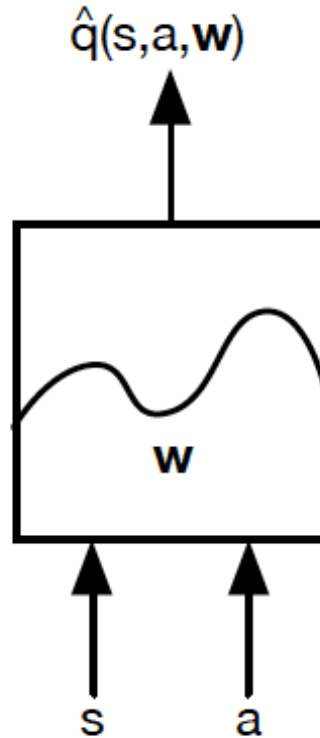
End-to-End Learning

State Space consists of millions of Pixels

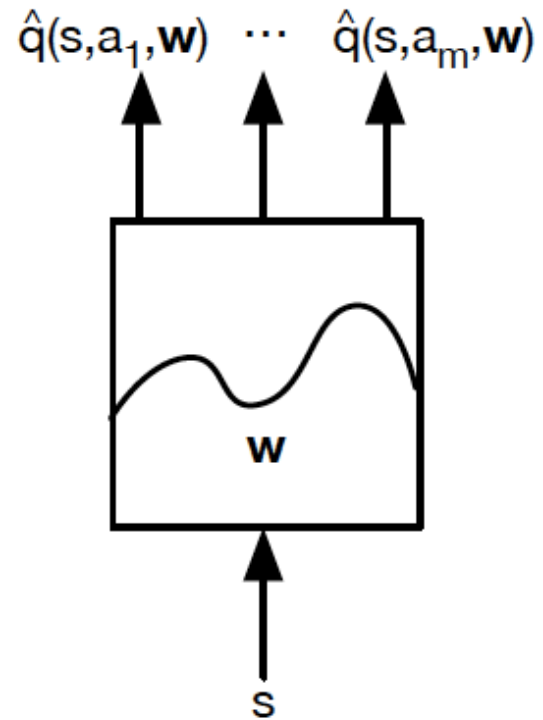
Types of Value Function Approximation



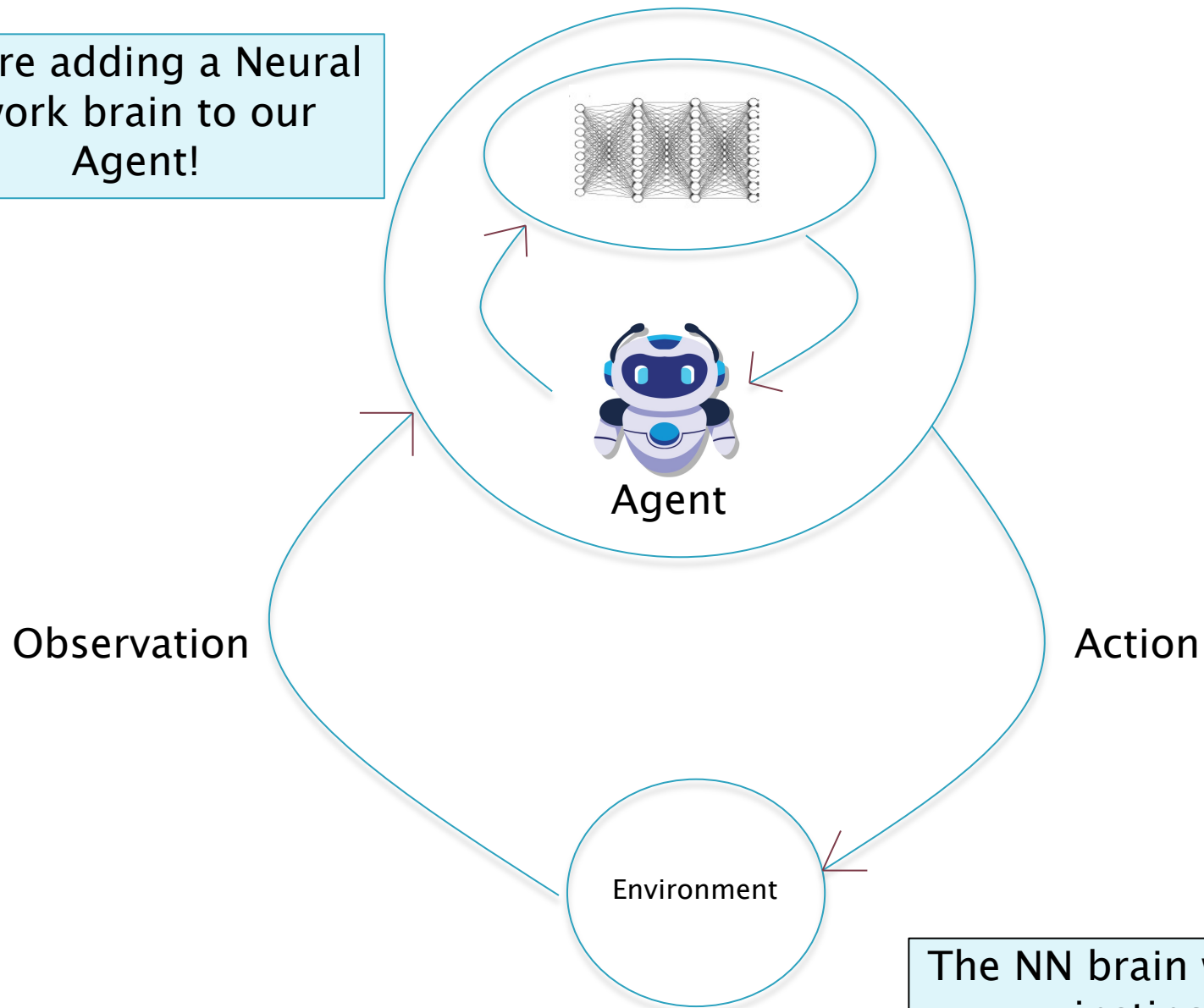
Approximate State Value Function



Two ways to approximate Action Value Function

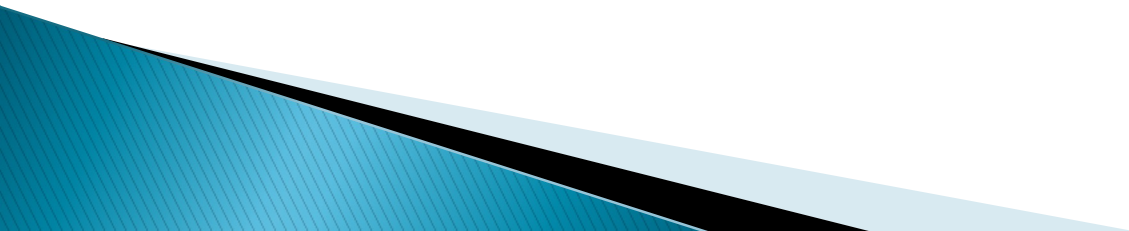


We are adding a Neural Network brain to our Agent!



The NN brain works by instinct:
 $A = \pi(S)$

Model Free Prediction with Value Function Approximations (Policy Known)



Approximating Value Functions: High Level Approach

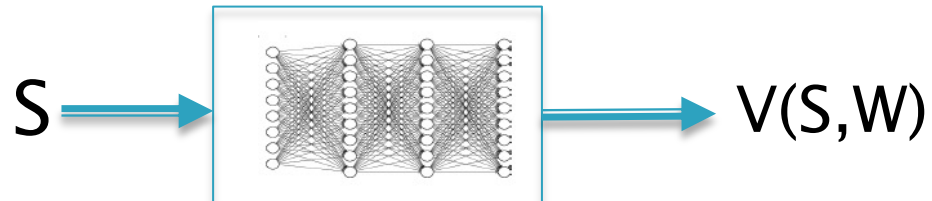
Run Sample Episodes from the MDP



Use Monte Carlo or TD Learning to get samples of $V(S)$



Use these samples to train the Neural Network
(i.e., find the weights W) using Supervised Learning



Training Using Gradient Descent

$$w_i \leftarrow w_i - \eta \frac{\partial L}{\partial w_i}$$

where

$$\frac{\partial L}{\partial w_i} = [y - t] x_i$$

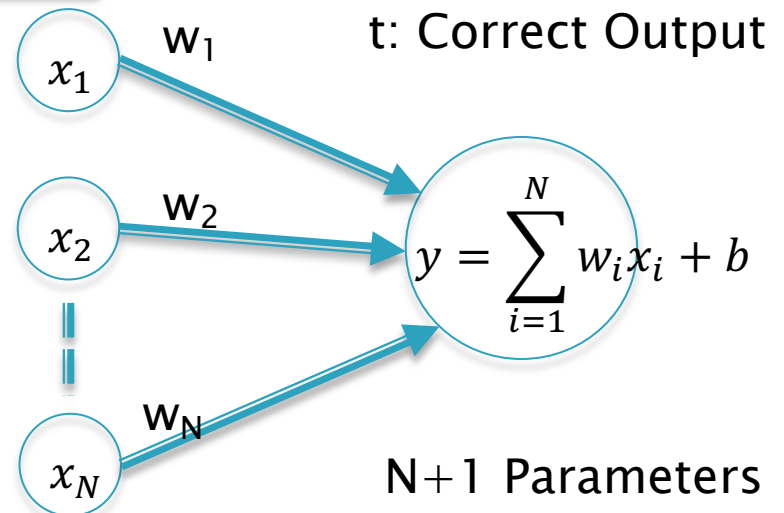
Error for j^{th} sample

$$L(W) = \frac{1}{2} [y - t]^2$$

$$y(j) = \sum_{i=1}^N w_i x_i + b$$

$$\frac{\partial L}{\partial b} = y - t$$

$$w_i \leftarrow w_i - \eta x_i [y - t]$$



Training Using Gradient Descent (with RL)

$$w_i \leftarrow w_i - \eta \frac{\partial L}{\partial w_i}$$

where

$$\frac{\partial L}{\partial w_i} = [V(S, W) - v(S)]x_i$$

$$\frac{\partial L}{\partial b} = V(S, W) - v(S)$$

$$w_i \leftarrow w_i - \eta x_i [V(S, W) - v(S)]$$

$v(S)$: 'Correct' or Target Value

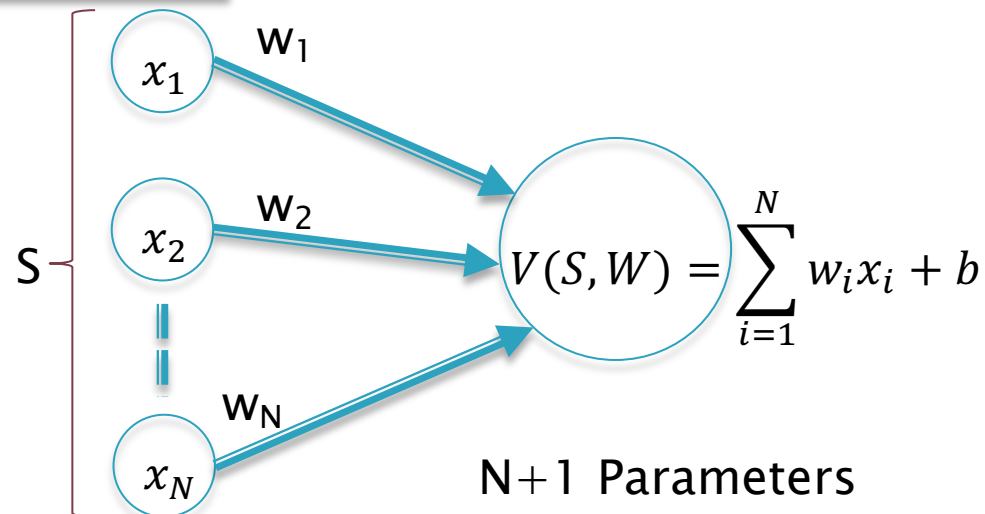
error

$$L(W) = \frac{1}{2} [V(S, W) - v(S)]^2$$

$$V(S, W) = \sum_{i=1}^N w_i x_i + b$$

$$S = (x_1, \dots, x_N)$$

Feature Vector



How to get the Target Values?

Monte Carlo based Value Function Approximation

$$v(S) \approx G(S)$$

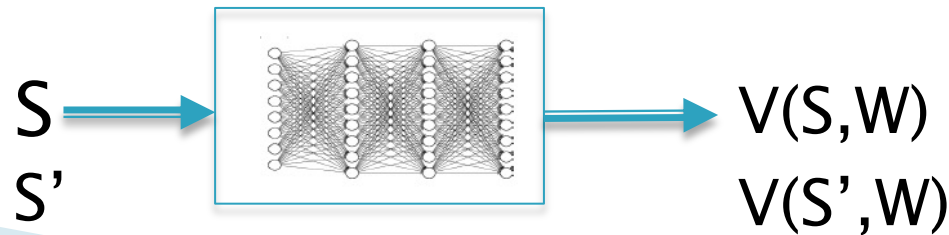
$$w_j \leftarrow w_j - \eta x_j [V(S, W) - G(S)]$$

TD Learning based Value Function Approximation

$$v(S) \approx R + \gamma V(S', W)$$

$$w_j \leftarrow w_j - \eta x_j [V(S, W) - (R + \gamma V(S', W))]$$

The model needs to be run twice:
To compute $V(S, W)$ and $V(S', W)$

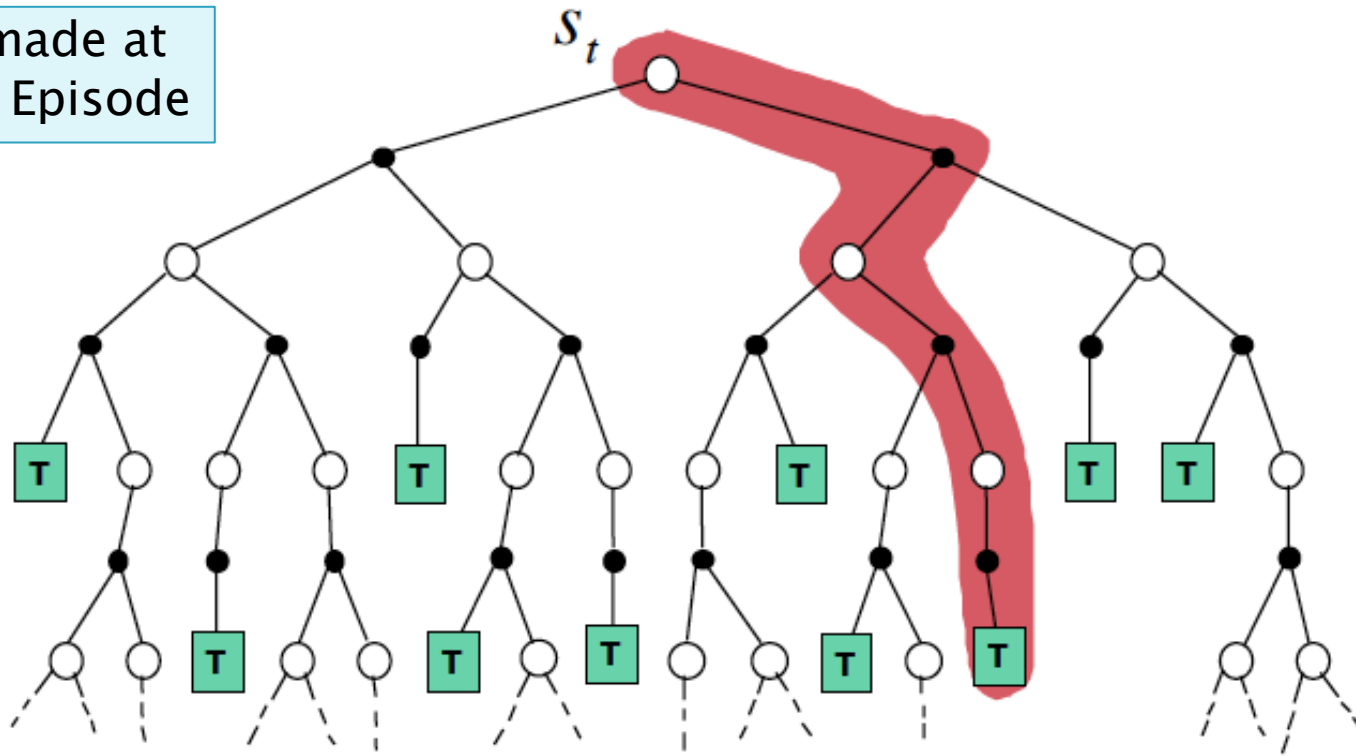


Monte Carlo Learning : Tabular Case

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t))$$



Updates made at end of an Episode



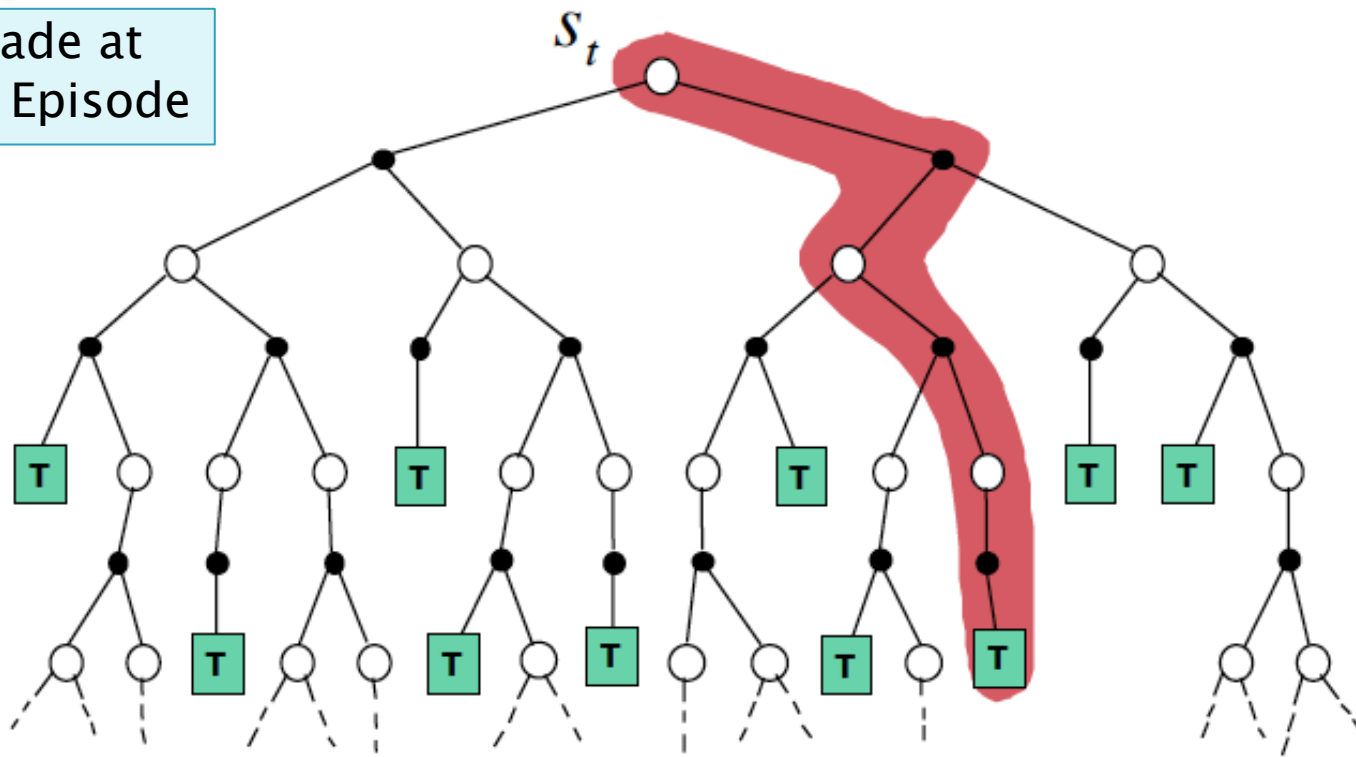
G_t : Latest estimate for $V(S_t)$

Monte Carlo Learning : Functional Case

$$w_j \leftarrow w_j - \eta x_j [V(S, W) - G(S)]$$



Update made at end of an Episode



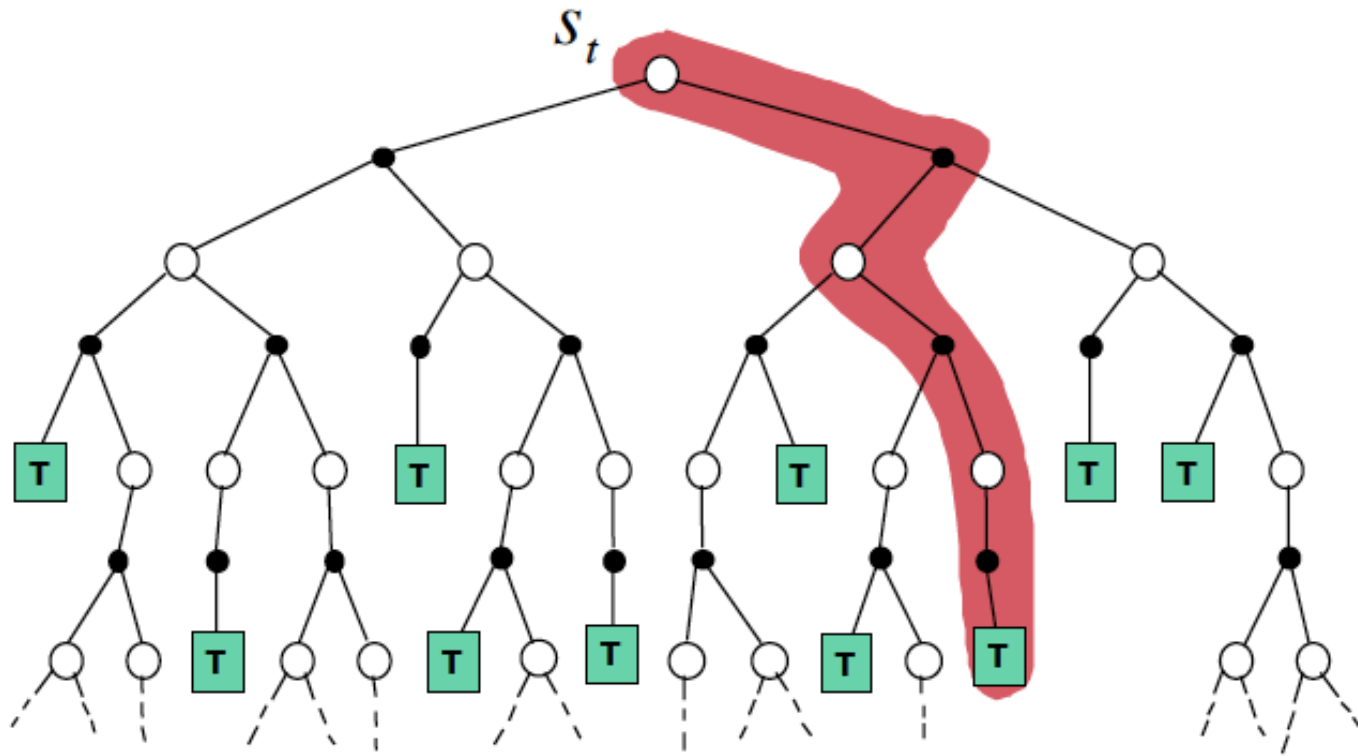
Issue: Every update causes ALL the Value Functions to change

Step 1: Run an Episode and Gather Training Data

(S_1, G_1)

(S_2, G_2)

·
·
·

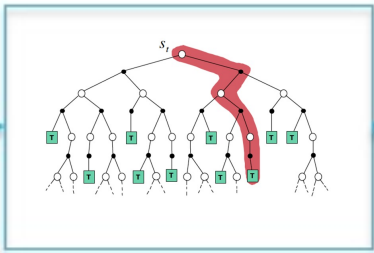


Step 2: Use the Training Data to Optimize the Model

Training Input Data



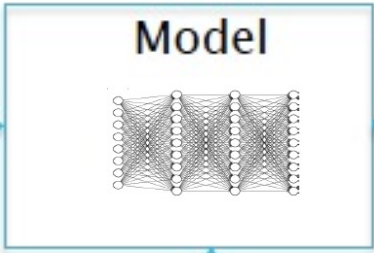
MC, TD



Generated by Interaction With the environment

(S_1, G_1)
 (S_2, G_2)
.
.
.

Generated by Neural Net Model



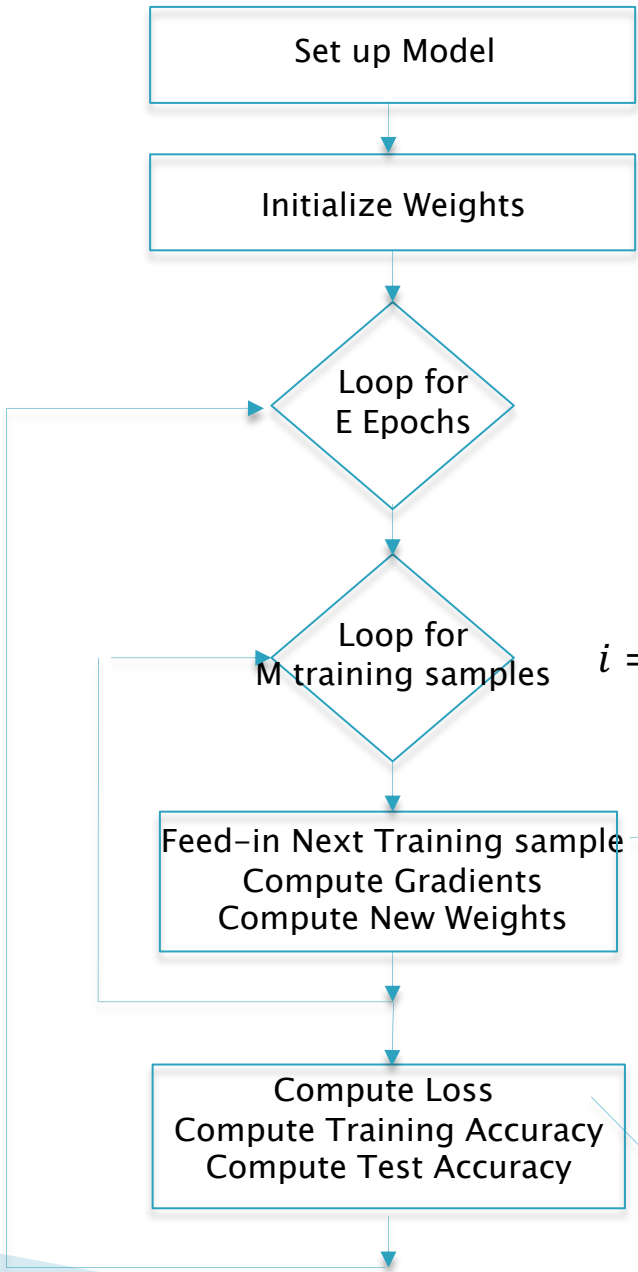
$$L(W) = \frac{1}{2} [V(S, W) - G(S)]^2$$

Adjust Weights To Reduce L(W)

- Gradient Descent
- Backprop

Training Algorithm Using Stochastic Gradient Descent MC Case

Linear Regression



$$V(S_i, W) = \sum_{j=1}^N w_j x_j(i) + b$$

$i = 1, 2, \dots, M$

$$\frac{\partial L}{\partial w_j} = x_j(i) [V(S_i, w) - G(S_i)]$$

$j = 1, 2, \dots, N$

$$w_j \leftarrow w_j - \eta x_j(i) [V(S_i, W) - G(S_i)]$$

$j = 1, 2, \dots, N$

$$L(W) = \frac{1}{2M} \sum_{i=1}^M [V(S_i, W) - G(S_i)]^2$$

Monte Carlo Learning with Value Function Approximation

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Initialize value-function weights \mathbf{w} as appropriate (e.g., $\mathbf{w} = \mathbf{0}$)

Repeat forever:

 Generate an episode $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using π

 For $t = 0, 1, \dots, T - 1$:

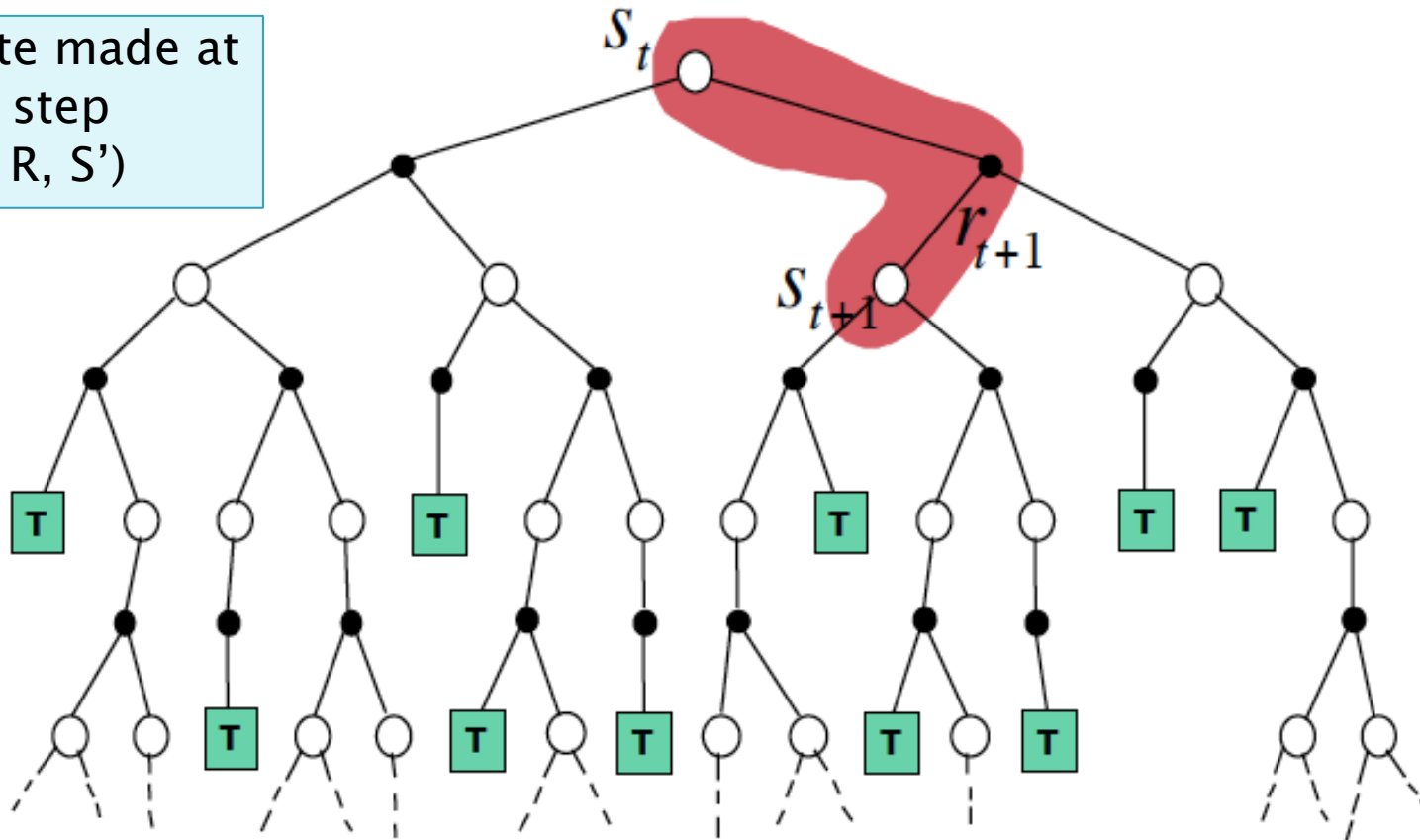
$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$

Since $E(G_t | S_t = s) = v_\pi(S_t)$ it follows that the weights \mathbf{w} of the Neural Network converge such that the output $V(S, \mathbf{w})$ converges to $v_\pi(S)$

TD Learning: Tabular Case

$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

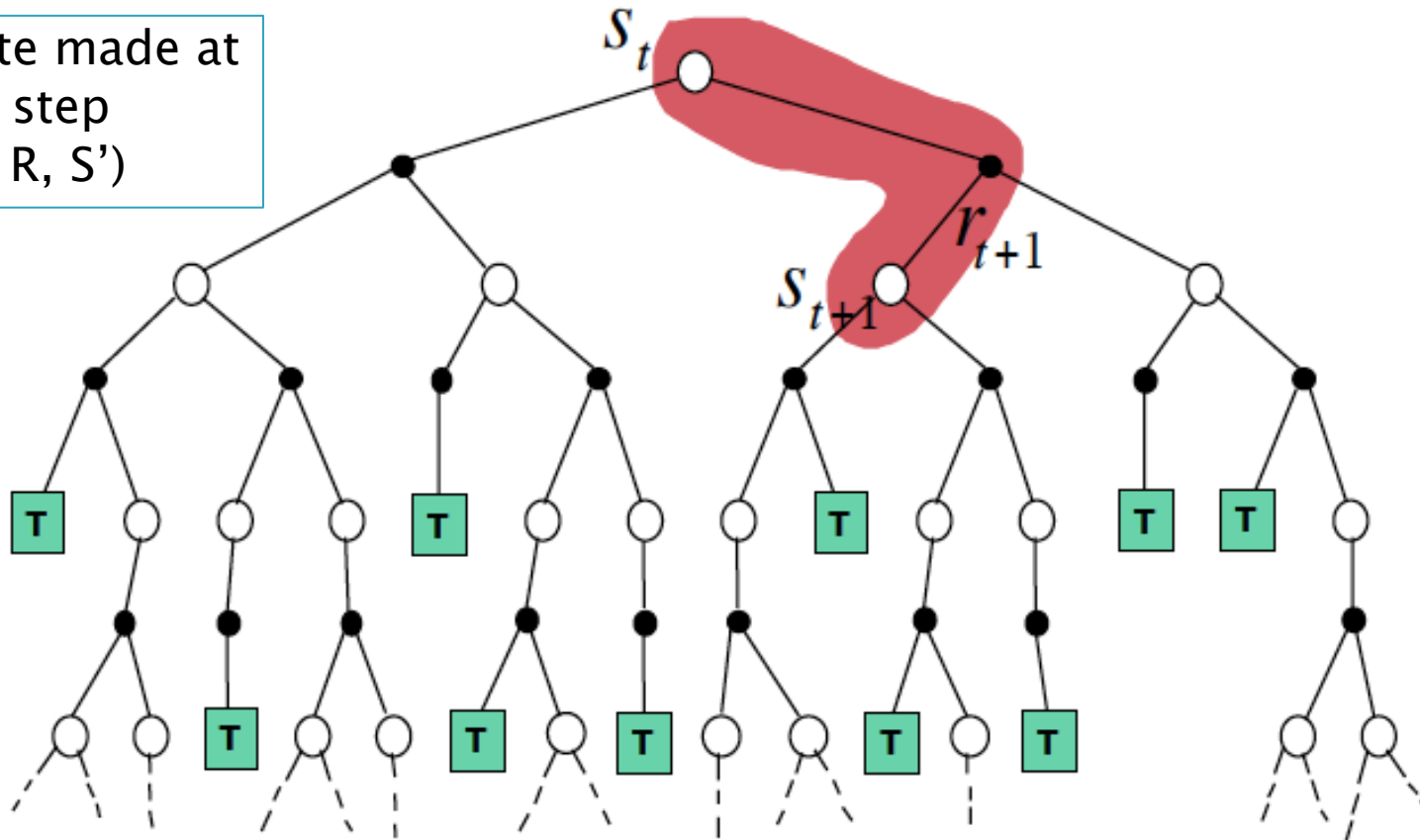
Update made at every step
(S, A, R, S')



TD Learning: Functional Case

$$w_j \leftarrow w_j - \eta x_j [V(S, W) - (R + \gamma V(S', W))]$$

Update made at every step
(S, A, R, S')

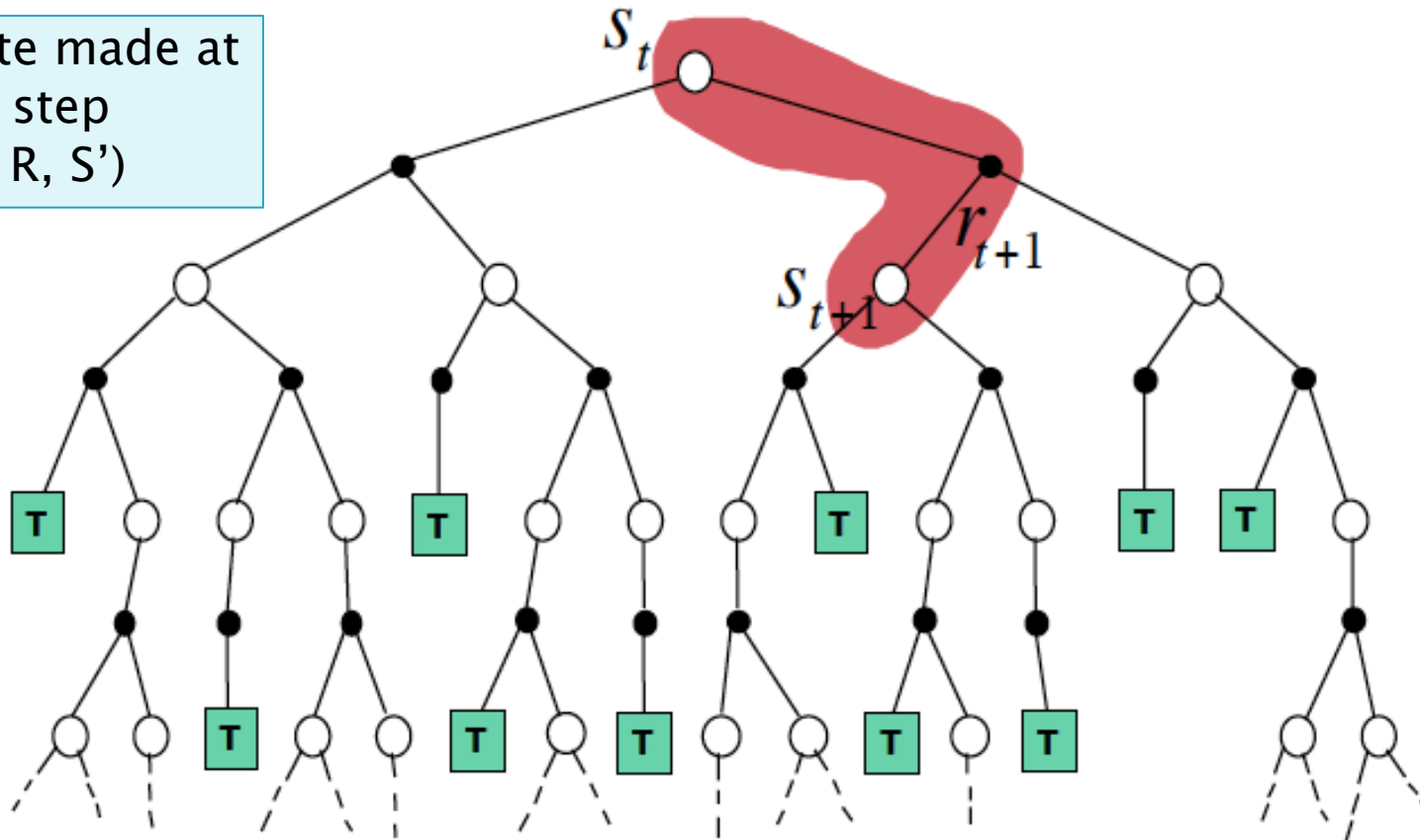


Issue1: Every update causes ALL the Value Functions to change

TD Learning: Functional Case

$$w_j \leftarrow w_j - \eta x_j [V(S, W) - (R + \gamma V(S', W))]$$

Update made at every step
(S, A, R, S')



Issue2: The regression target is no longer fixed. It is also a function of the weight parameters
Chasing a Moving Target!!

TD Learning with Value Function Approximation

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Initialize value-function weights \mathbf{w} arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Repeat (for each episode):

Initialize S

Repeat (for each step of episode):

Choose $A \sim \pi(\cdot|S)$

Take action A , observe R, S'

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$

$S \leftarrow S'$

until S' is terminal

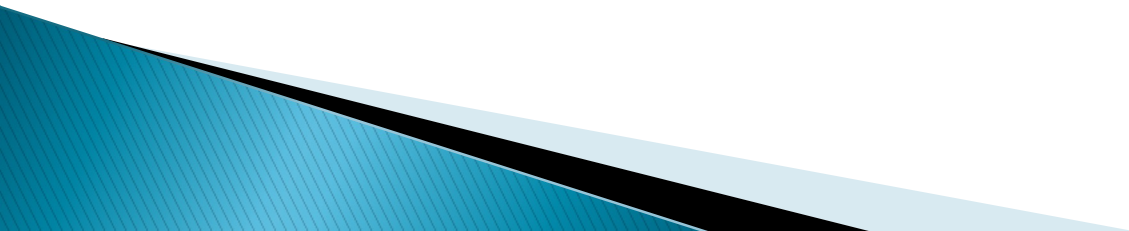
Policy is given



Do these Value Function Approximation Algorithms Converge?

Algorithm	Table Lookup	Linear	Non-Linear
MC	✓	✓	✓
TD(0)	✓	✓	✗

Model Free Control with Value Function Approximations

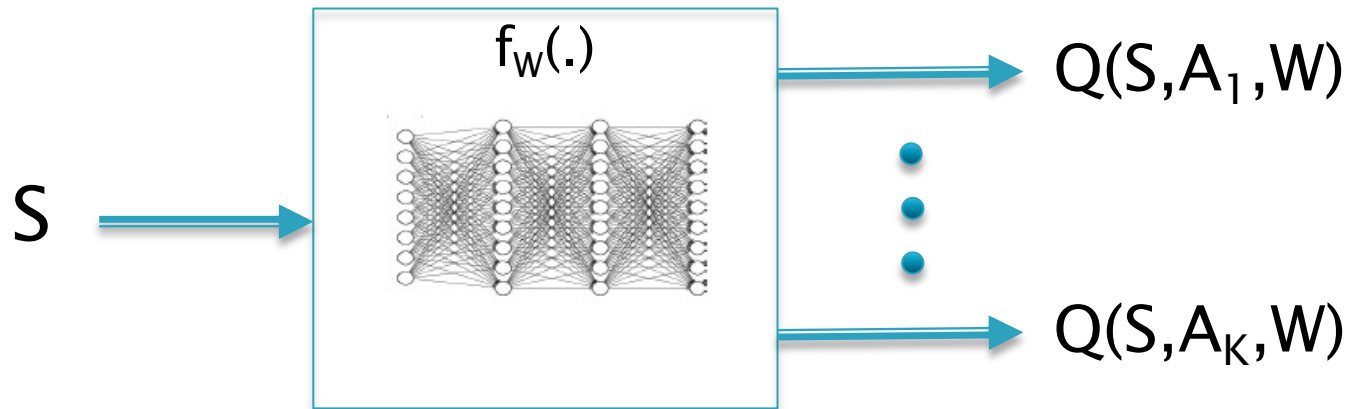


To Go From Policy Evaluation to Control..

Two changes:

1. Approximate $q(S,A)$ rather than $v(S)$
2. At each step, improve policy using epsilon greedy algorithm

Approximating the Optimal Action-Value Function



Approximating Q Functions: High Level Approach

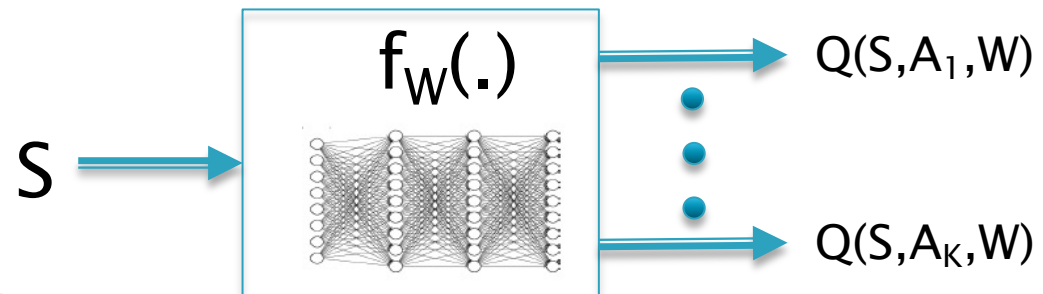
Run Sample Episodes from the MDP



Use Monte Carlo, SARSA or Q Learning to get samples of $q(S,A)$



Use these samples to train the Neural Network (i.e., find the weights W) using Supervised Learning



How to get the Target Values?

Monte Carlo based Q Function Approximation

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q(S, A_k, W) - G]$$

ϵ Greedy policy

SARSA Learning based Q Function Approximation

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q(S, A_k, W) - (R + \gamma Q(S', A', W))]$$

ϵ Greedy policy

ϵ Greedy policy

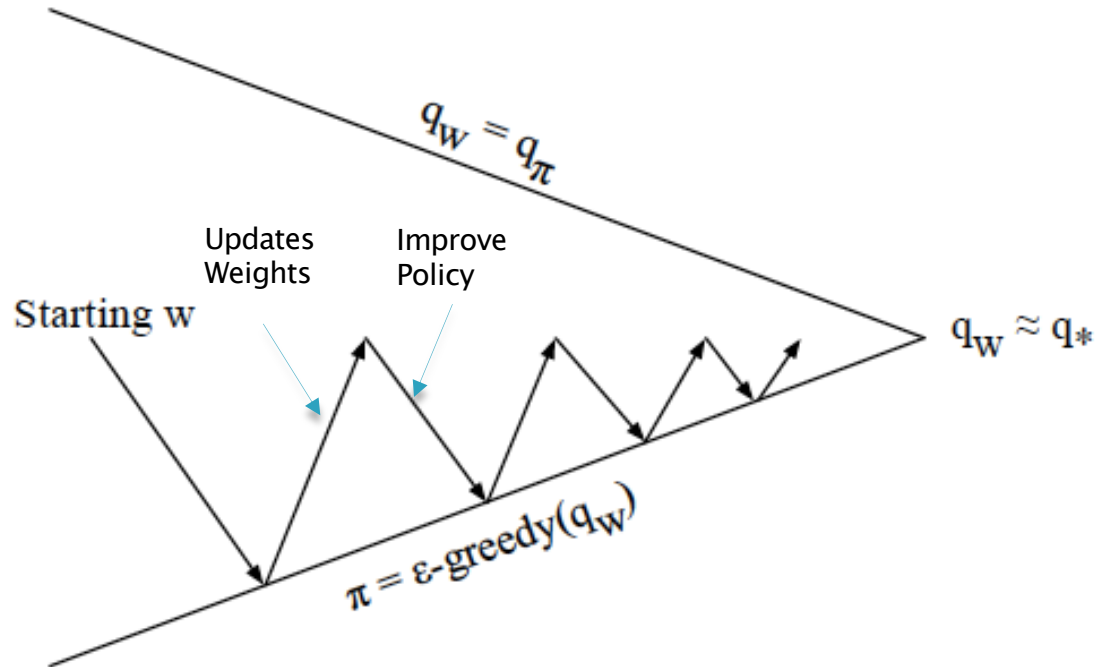
Q-Learning based Q Function Approximation

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q(S, A_k, W) - (R + \gamma \max_{A'} Q(S', A', W))]$$

ϵ Greedy policy

Optimal policy

Control with Action Value Function Approximation

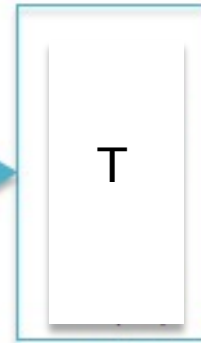
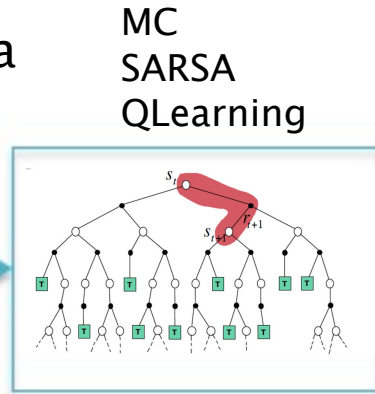
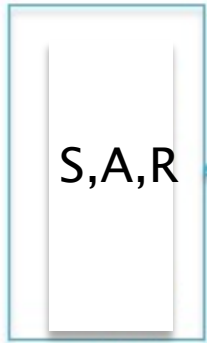


Policy evaluation **Approximate** policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

Use the Training Data to Optimize the Model

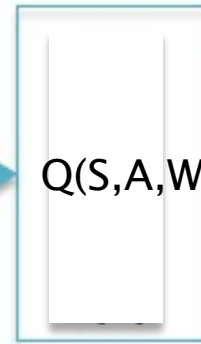
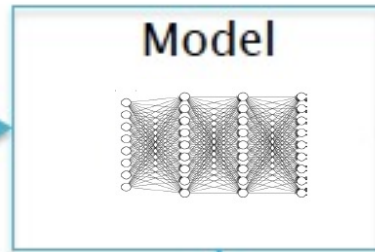
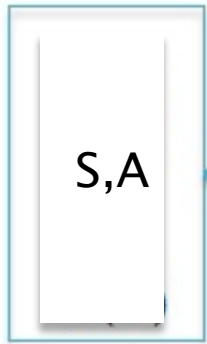
Training Input Data



Generated by Interaction With the environment

(S1,T1)
(S2,T2)
⋮
⋮

Generated by Neural Net Model



$$L(W) = \frac{1}{2} [Q(S, A, W) - T]^2$$

Adjust Weights To Reduce L(W)

- Gradient Descent
- Backprop

K-ary Linear Networks

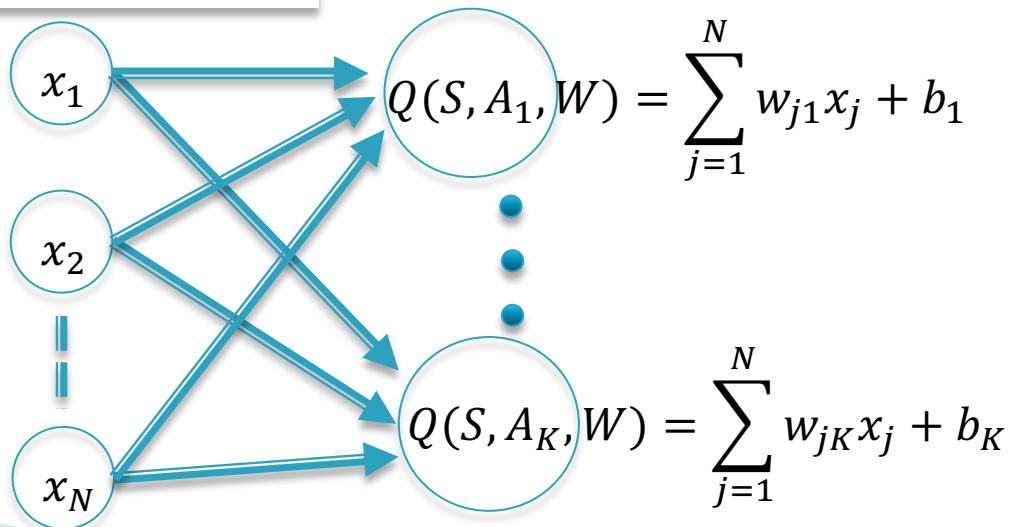
$$w_{jk} \leftarrow w_{jk} - \eta \frac{\partial L}{\partial w_{jk}}$$

where

$$\frac{\partial L}{\partial w_{jk}} = x_j [Q(S, A_k, W) - T]$$

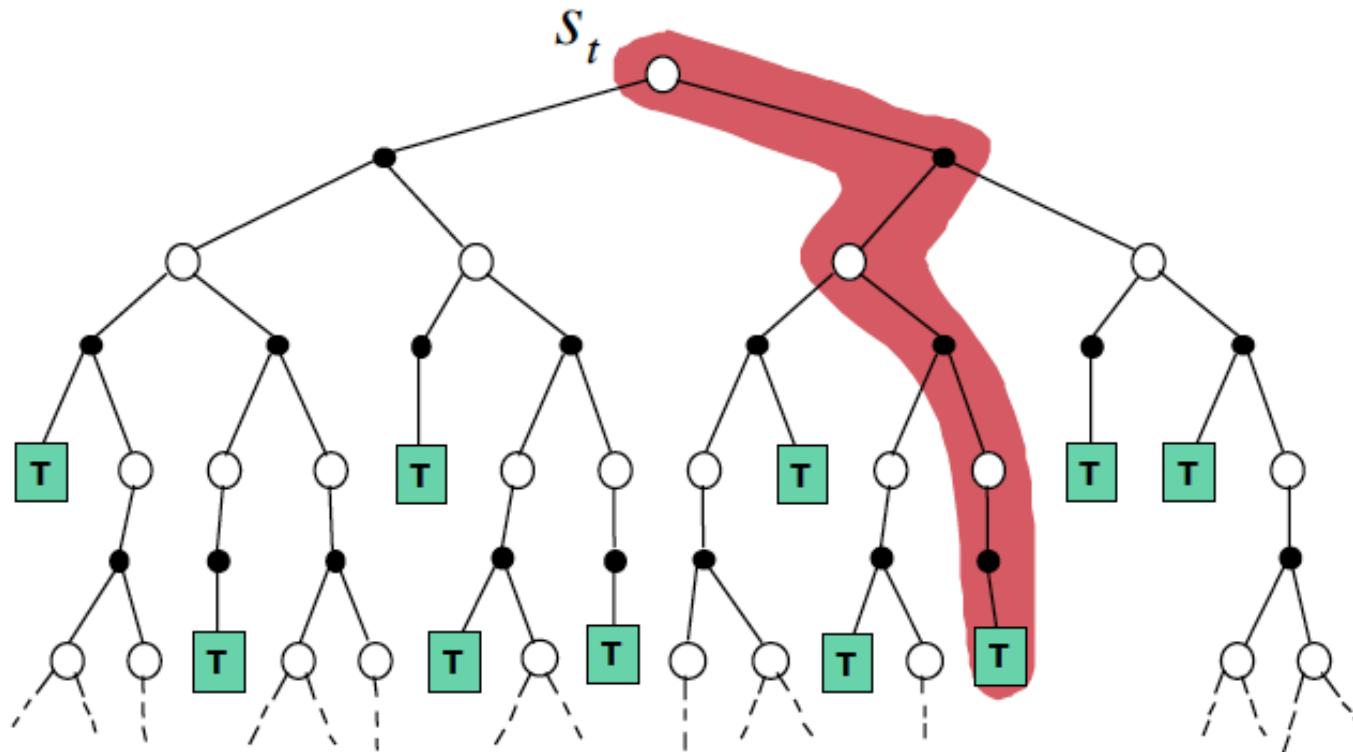
$$L(W) = \frac{1}{K} \sum_{k=1}^K (Q_k(S, A_k, W) - T)^2$$
$$Q(S, A_k, W) = \sum_{j=1}^N w_{jk} x_j + b_k$$

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q(S, A_k, W) - T]$$



Monte Carlo Control: Tabular Case

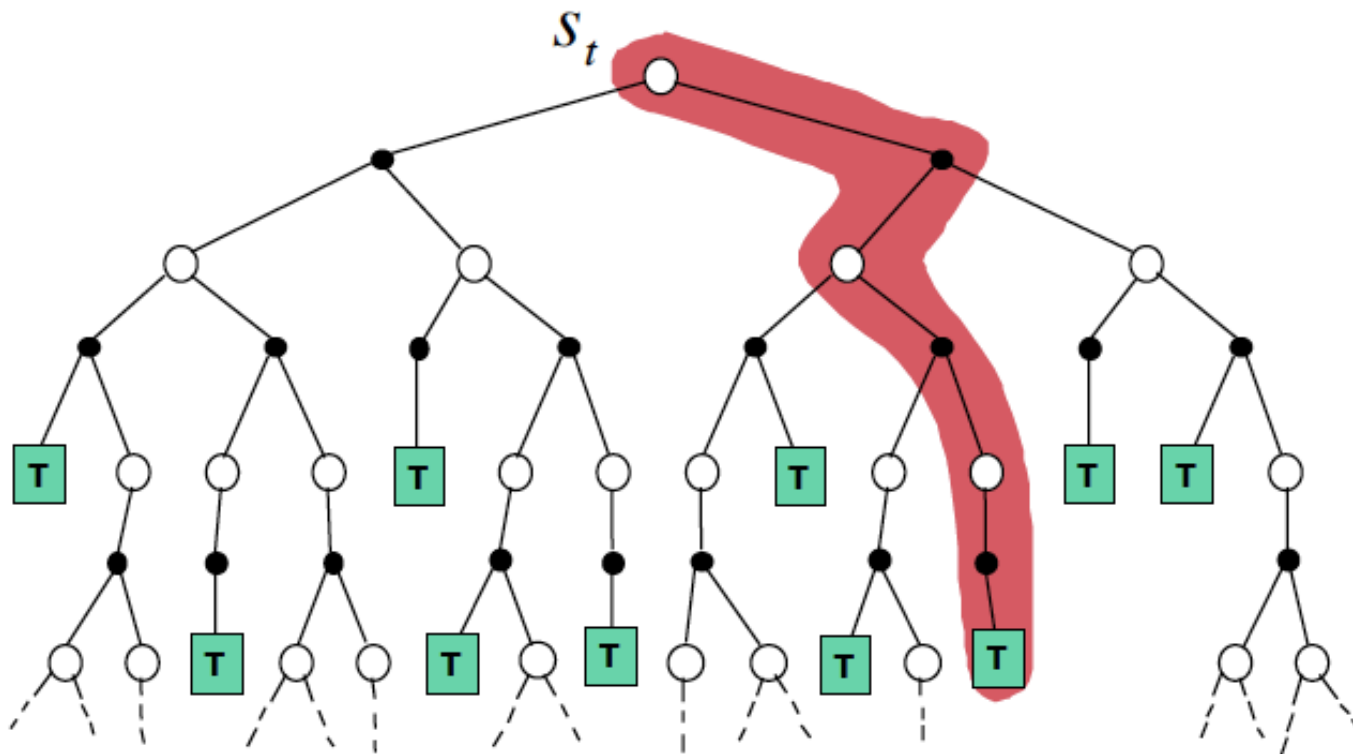
$$Q(S,A) \leftarrow Q(S,A) + \alpha(G - Q(S,A))$$



Policy Improvement Update made at end of each Episode

Monte Carlo Control: Functional Case

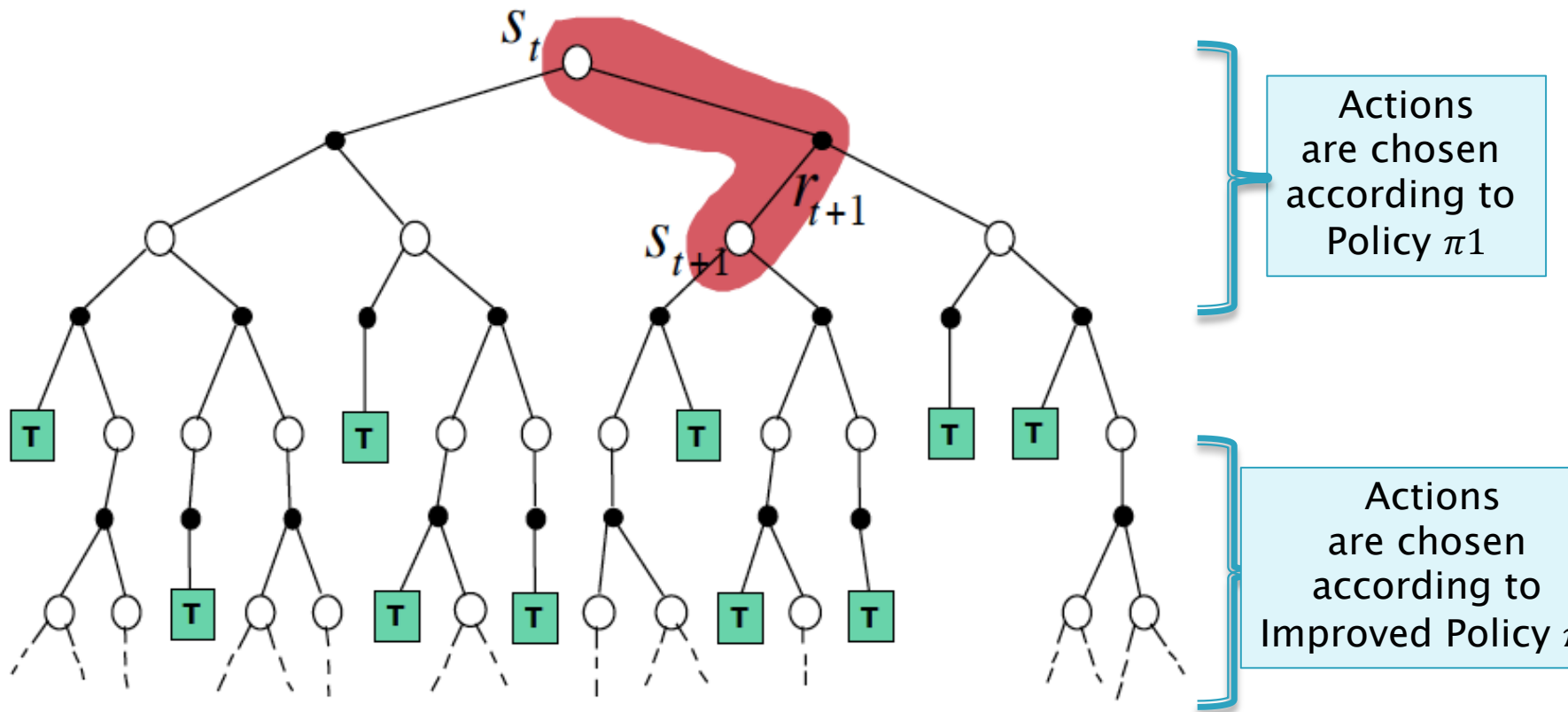
$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q(S, A_k, W) - G]$$



Weight Updates made at end of each Episode

SARSA Control: Tabular Case

$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma Q(S', A') - Q(S, A))$$

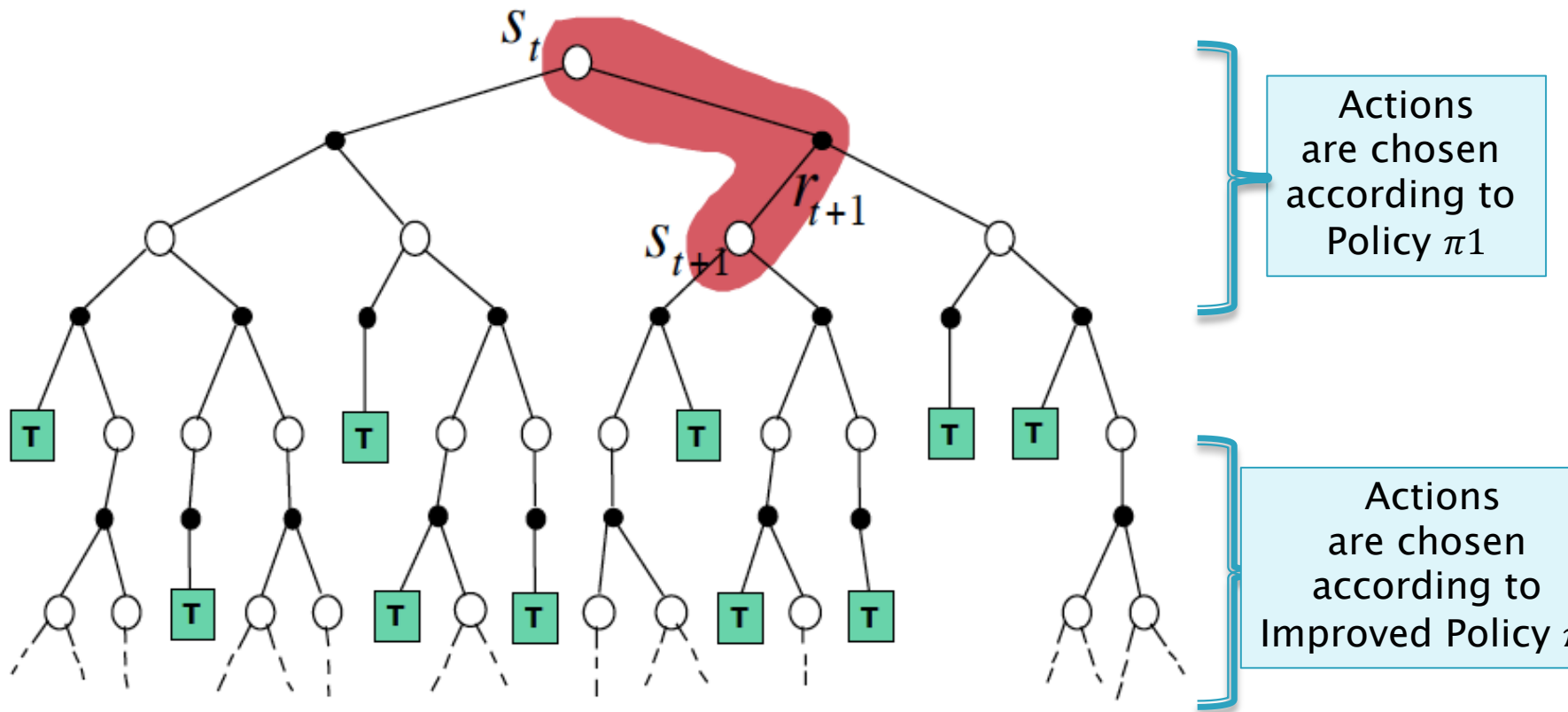


Policy Improvement Update made after each step

SARSA Control: Functional Case

ϵ Greedy Policy

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q(S, A_k, W) - (R + \gamma Q(S', A', W))]$$



Weight Updates made at each Step

SARSA with Action-Value Function Approximation

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Repeat (for each episode):

$S, A \leftarrow$ initial state and action of episode (e.g., ϵ -greedy)

 Repeat (for each step of episode):

 Take action A , observe R, S'

 If S' is terminal:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

 Go to next episode

 Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ϵ -greedy)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

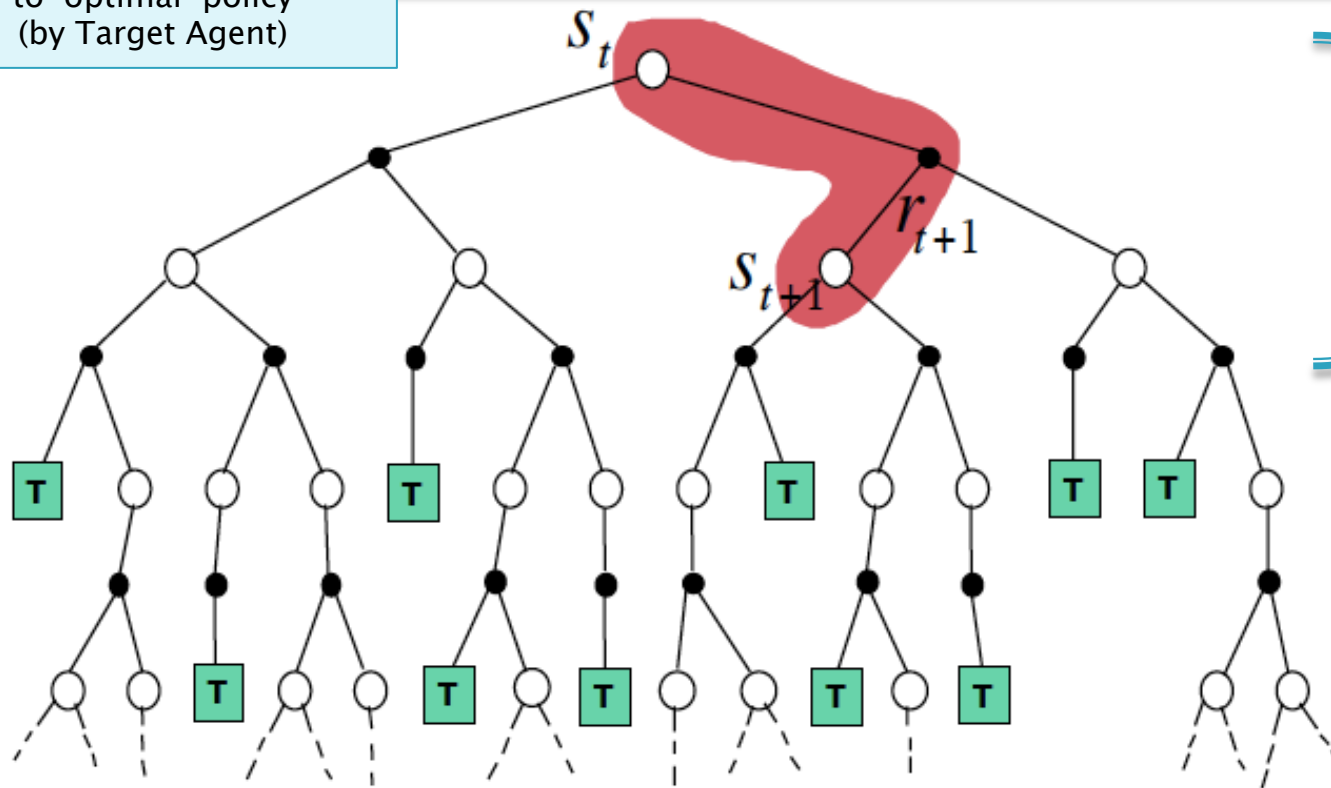
$S \leftarrow S'$

$A \leftarrow A'$

Q Learning: Tabular Case

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

The Q value updates made at each step according to 'optimal' policy (by Target Agent)

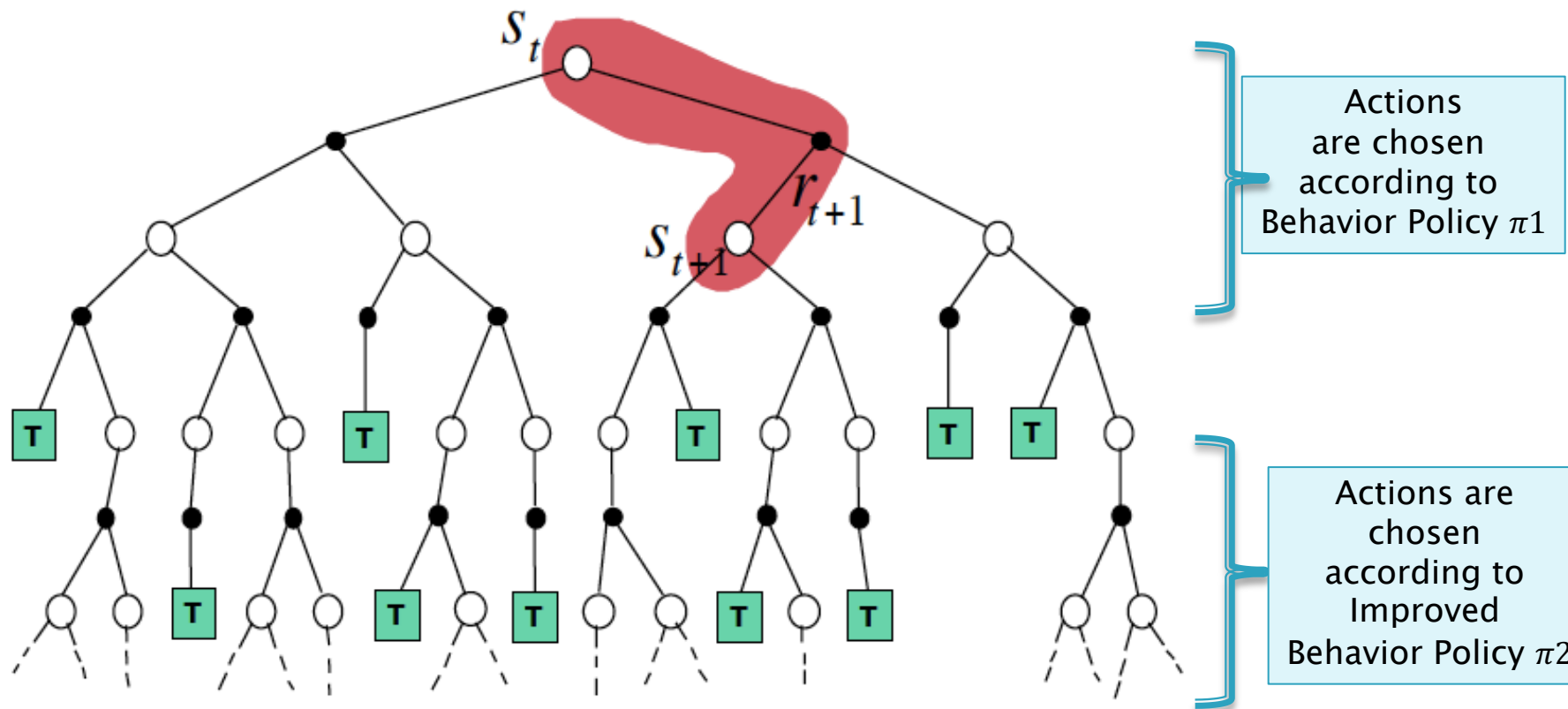


Actions chosen according to Behavior Policy π_1 (by Behavior Agent)

Actions chosen according to improved Behavior Policy π_2 (by Behavior Agent)

Q Learning: Functional Case

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q(S, A_k, W) - (R + \gamma \max_{A'} Q(S', A', W))]$$



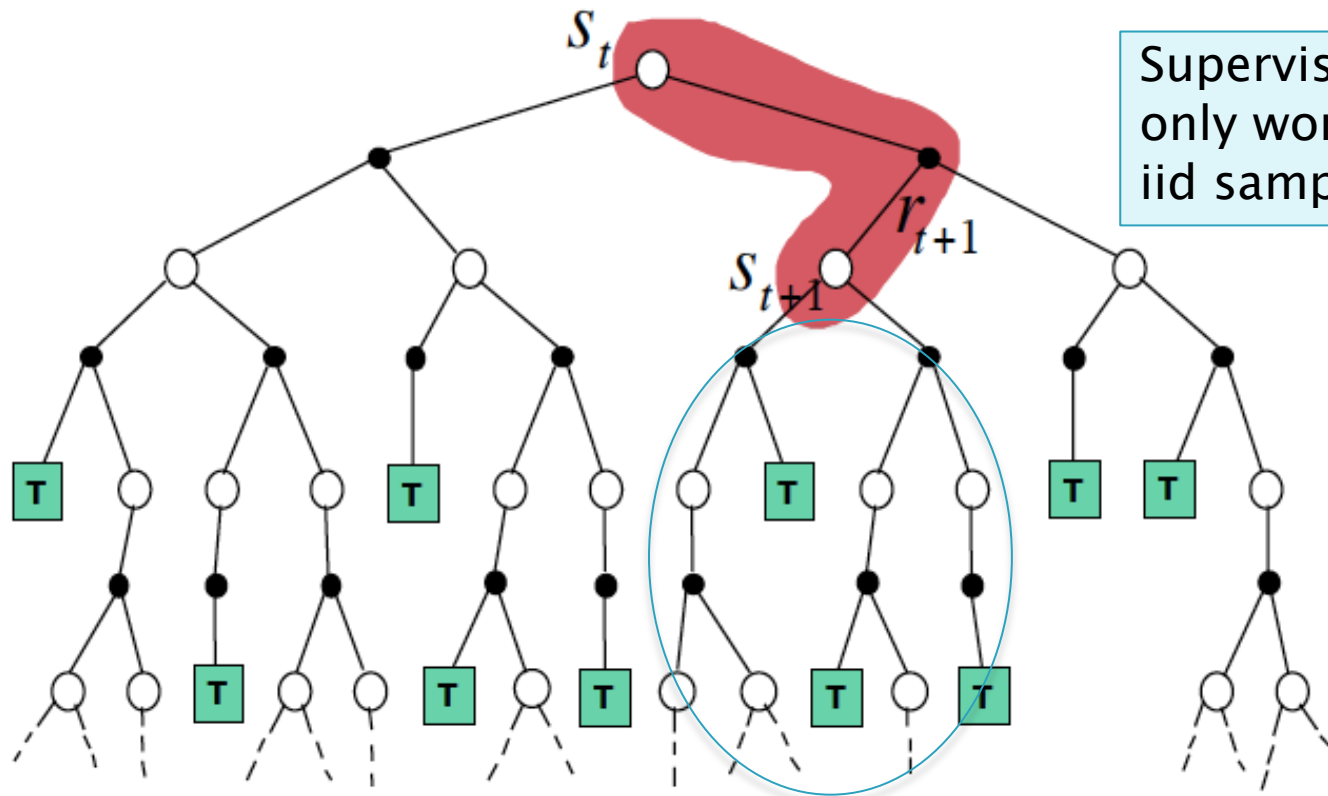
Weight Updates made at each Step

Issues with All These Functional Control Algorithms

Issue1: Every update causes ALL the Value Functions to change

Issue2: In Functional SARSA and Q-Learning, the regression target is no longer fixed. It is also a function of the weight parameters
Chasing a Moving Target!!

Issue 3: Correlated Samples



Supervised Learning
only works with
iid samples

The state sequence S_1, S_2, \dots is not iid
It is a function of the agent's actions

Do these Functional Control Algorithms Converge?

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗

(✓) = chatters around near-optimal value function

In Practice: Any problem with a large number of states diverges

How to Stabilize Functional Control Algorithms?

Stabilization of Functional Q Learning

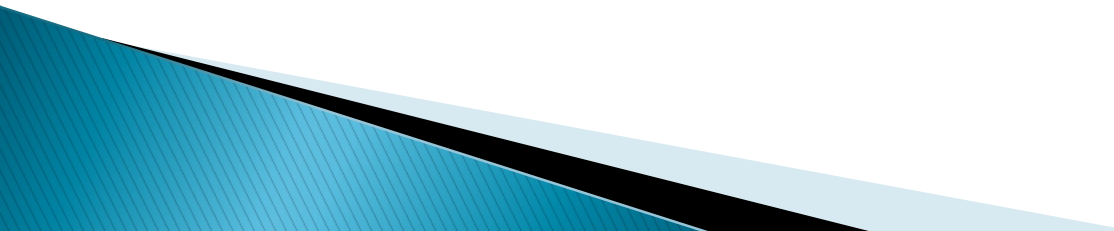
- ▶ DQN (Deep Q Networks): DeepMind, 2012

Stabilization of SARSA

- ▶ A3C (Asynchronous Advantage Actor Critic): DeepMind, 2016

Deep Q Networks (DQN)

Deep Q Networks

- ▶ High Level Idea: Make Deep Q Learning look like Supervised Learning
 - ▶ Two Main Ideas:
 1. Experience Re-Play: Decorrelates successive samples
 2. Use of Target Network: Stabilizes the Target value
- 

Q Learning with Experience Reuse

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

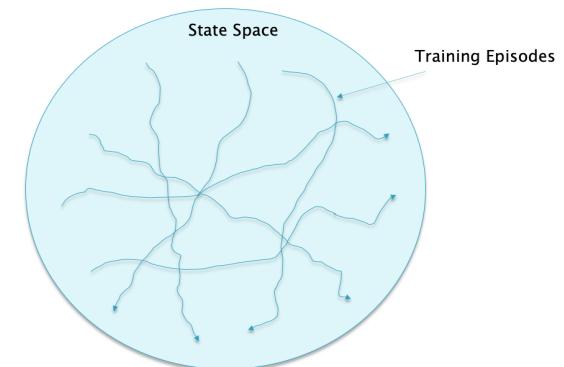
(S1,A1,R1,S1')

(S2,A2,R2,S2')

(S3,A3,R3,S3')

(S4,A4,R4,S4')

A Collection of
1-Step Transitions



Experience Replay

Replay Buffer

(S1,A1,R1,S1')

(S2,A2,R2,S2')

(S3,A3,R3,S3')

(S4,A4,R4,S4')

A Collection of
1-Step Transitions

Choose a Transition
at Random and Compute

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q_k(S, A_k, W) - (R + \gamma \max_{A'} Q(S', A', W))]$$

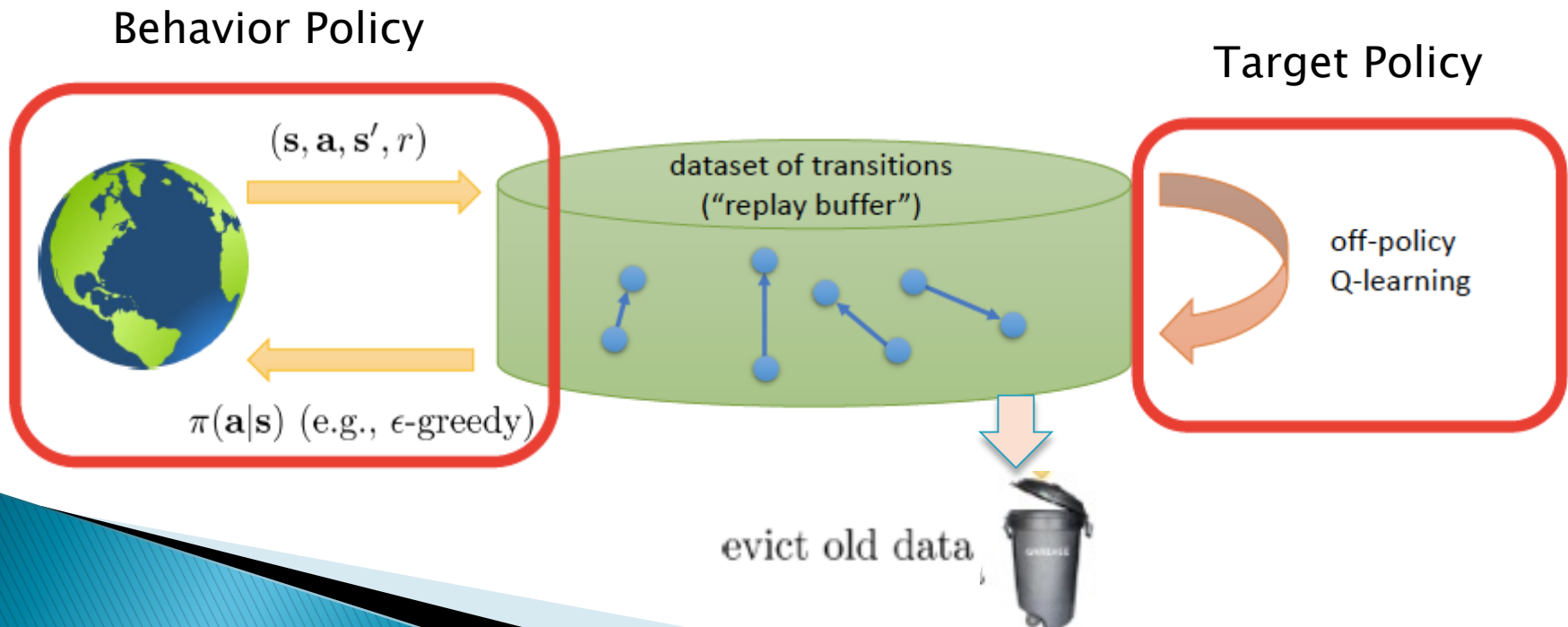
Samples are no longer correlated!

Q-Learning with Replay Buffer

full Q-learning with replay buffer:


1. collect dataset $\{(s_i, \mathbf{a}_i, s'_i, r_i)\}$ using some policy, add it to \mathcal{B}
2. sample a batch $(s_i, \mathbf{a}_i, s'_i, r_i)$ from \mathcal{B}
3. $w \leftarrow w - \eta x [Q(S, A, W) - (R + \gamma \max_{A'} Q(S', A', W))]$

K = 1 is common, though larger K more efficient



Solving Problem 2: Moving Target

full Q-learning with replay buffer:

- 
1. collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to \mathcal{B}
 2. sample a batch (s_i, a_i, s'_i, r_i) from \mathcal{B}
 3. $w \leftarrow w - \eta x [Q(S, A, W) - (R + \gamma \max_{a'} Q(S', A', W))]$

one gradient step, moving target

Q-learning is *not* gradient descent!

$$w \leftarrow w - \eta x [Q(S, A, W) - (R + \gamma \max_{A'} Q(S', A', W))]$$

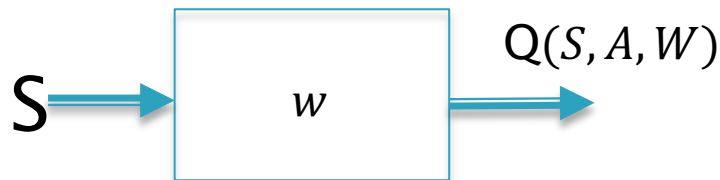
no gradient through target value

This is still a problem!

Solving Problem 2: Target Networks

$$w_{jk} \leftarrow w_{jk} - \eta x_j [Q_k(S, A_k, W) - (R + \gamma \max_{A'} Q(S', A', W'))]$$

Use Two Neural Networks!



Behavior DQN

Runs the main training loop
Weights updated at every Step



Target DQN

Computes targets for the Behavior DQN
Weights updated less frequently

Q-Learning with Target Networks

Q-learning with replay buffer and target network:

1. save target network parameters: $W' \leftarrow W$

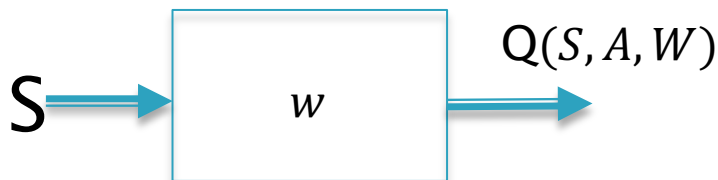
2. collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to \mathcal{B}

$N \times$
 $K \times$ 3. sample a batch (s_i, a_i, s'_i, r_i) from \mathcal{B}

4. $w \leftarrow w - \eta x [Q(s, a, w) - (R + \gamma \max_{A'} Q(s', a', w'))]$

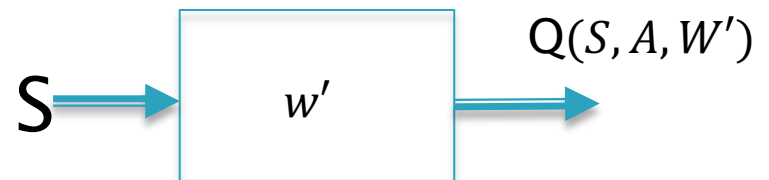
targets don't change in inner loop!

supervised regression



Behavior DQN

Runs the main training loop



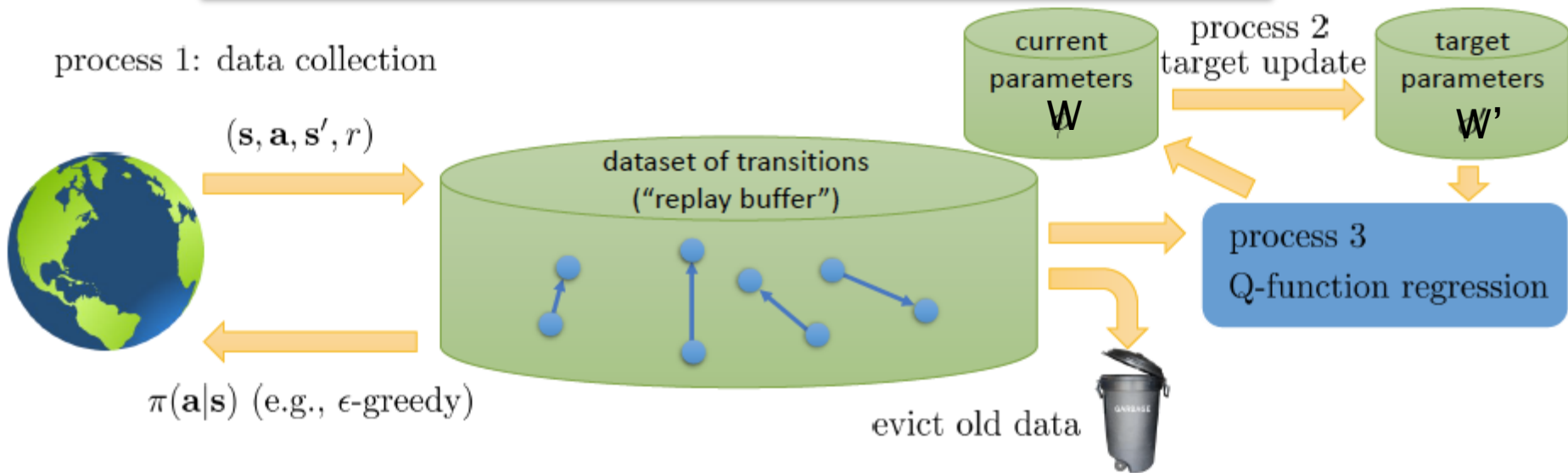
Target DQN

Computes targets for the Online DQN

Overall View for DQN

Q-learning with replay buffer and target network:

1. save target network parameters: $W' \leftarrow W$
2. collect M datapoints $\{(s_i, \mathbf{a}_i, s'_i, r_i)\}$ using some policy, add them to \mathcal{B}
3. sample a batch $(s_i, \mathbf{a}_i, s'_i, r_i)$ from \mathcal{B}
4. $w \leftarrow w - \eta x [Q(S, A, W) - (R + \gamma \max_{a'} Q(S', A', W'))]$



Playing Atari with Deep Q Networks (2013)

LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih^{1*}, Koray Kavukcuoglu^{1*}, David Silver^{1*}, Andrei A. Rusu¹, Joel Veness¹, Marc G. Bellemare¹, Alex Graves¹, Martin Riedmiller¹, Andreas K. Fiedelnd¹, Georg Ostrovski¹, Stig Petersen¹, Charles Beattie¹, Amir Sadik¹, Ioannis Antonoglou¹, Helen King¹, Dhharshan Kumaran¹, Daan Wierstra¹, Shane Legg² & Demis Hassabis²

The theory of reinforcement learning provides a normative account¹, deeply rooted in psychological² and neuroscientific³ perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems^{4,5}, the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms⁶. While reinforcement learning agents have achieved some successes in a variety of domains⁷⁻⁹, their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks¹⁰⁻¹² to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games¹³. We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

We set out to create a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks—a central goal of general artificial intelligence¹⁴ that has eluded previous efforts^{15,16}. To achieve this, we developed a novel agent: a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural networks¹⁷ known as deep neural networks. Notably, recent advances in deep neural networks^{18,19}, in which several layers of

agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E} [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a, \pi],$$

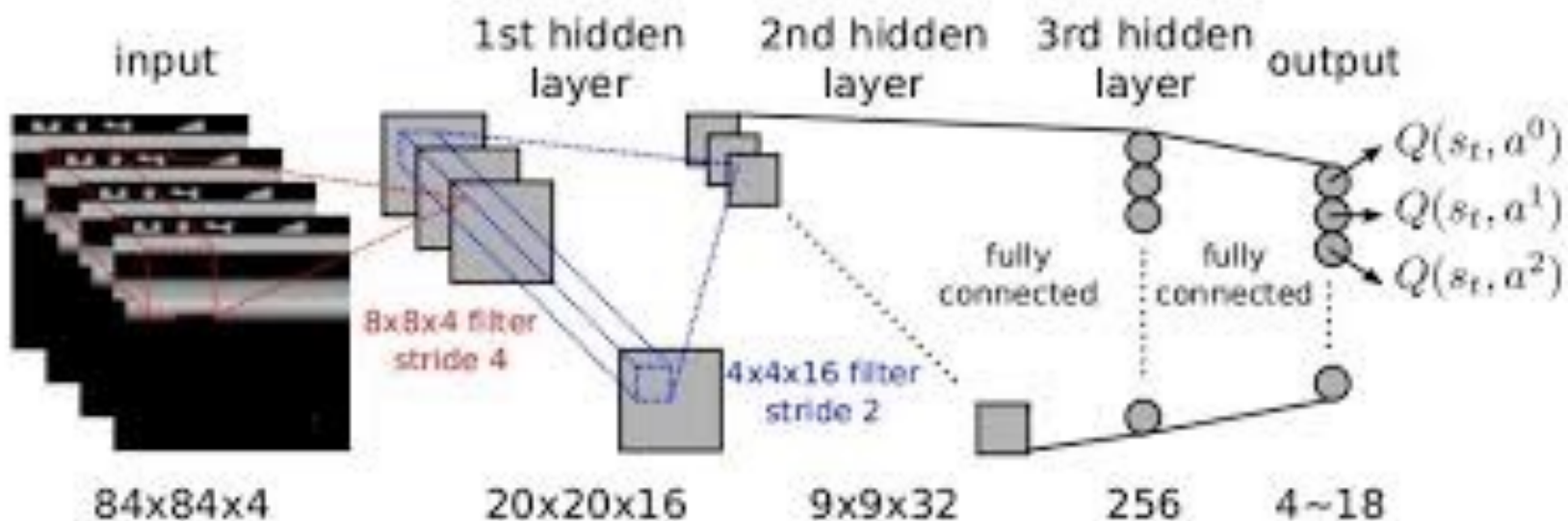
which is the maximum sum of rewards r_t discounted by γ at each time-step t , achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a) (see Methods)¹⁹.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function²⁰. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_{a'} Q(s', a')$. We address these instabilities with a novel variant of Q-learning which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay²¹⁻²³ that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution (see below for details). Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target.

While other stable methods exist for training neural networks in the reinforcement learning setting, such as neural fitted Q-iteration²⁴, these methods involve the repeated training of networks *denovo* on hundreds of iterations. Consequently, these methods, unlike our algorithm, are too inefficient to be used successfully with large neural networks. We parameterize an approximate value function $Q(s,a;\theta_i)$ using the deep convolutional neural network shown in Fig. 1, in which θ_i are the parameters (that is, weights) of the Q-network at iteration i . To perform experience replay we store the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t in a data set $D_t = \{e_1, \dots, e_t\}$. During learning, we apply Q-learning updates, on samples (or minibatches) of experience $(s,a,r,s') \sim U(D)$, drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function:

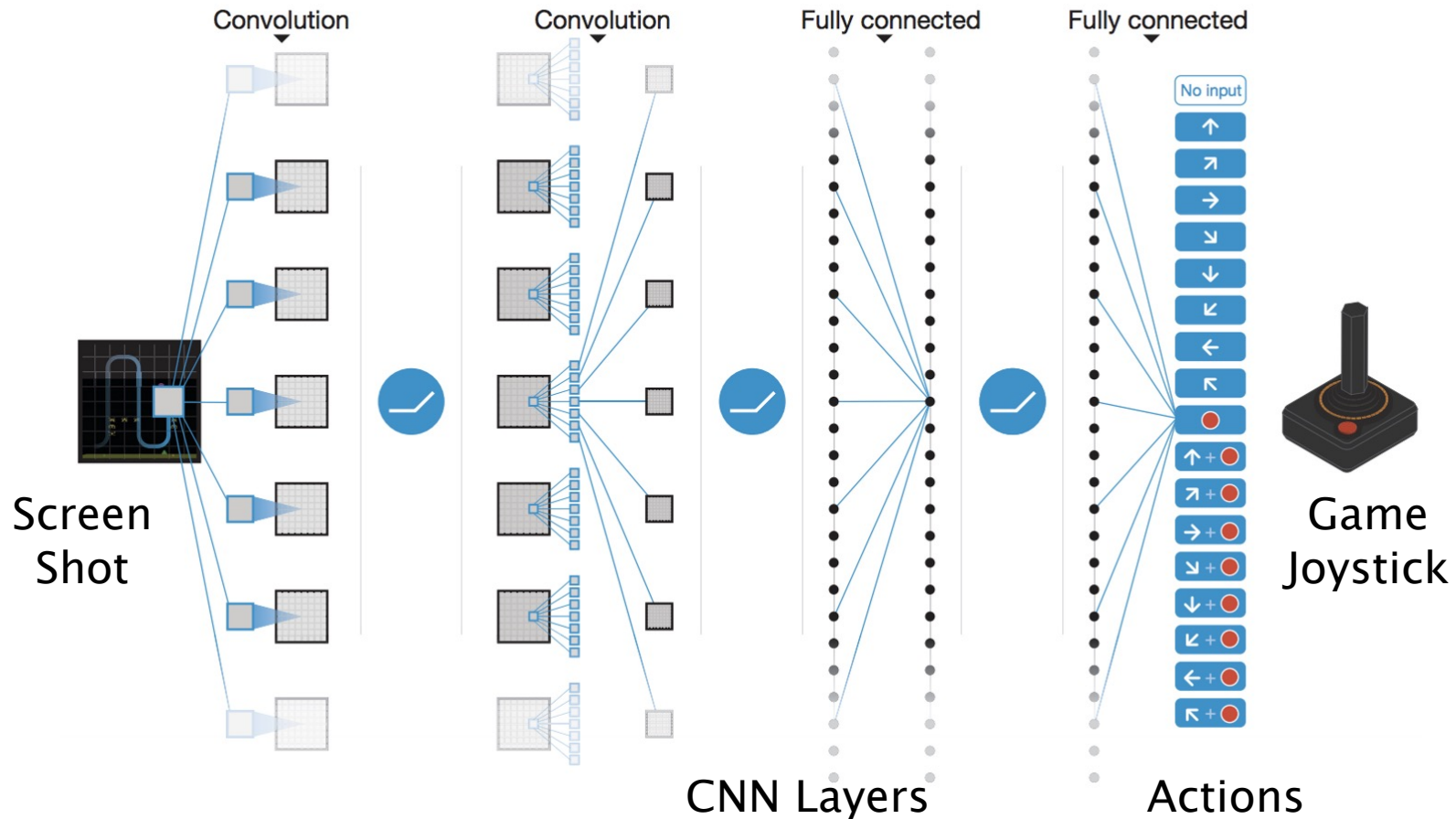
Atari Network Architecture: An Application of the DQN Algorithm

- ▶ End-to-end learning of values $Q(s, a)$ from pixels s
- ▶ Input state s 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



Convolutional Neural Network (CNN)

Deep Reinforcement Learning



Deep Models allows RL algorithms to solve Complex Decision Making Problems End-to-End

DQN Algorithm

Functional Q Learning with Experience Replay and Target Network

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

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 $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

DQN Algorithm

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

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 $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

← Initialize replay memory, Q-network

← Initialize Target network

← Play M episodes (full games)

← Initialize state
(starting game
screen pixels) at the
beginning of each
episode

DQN Algorithm

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

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 $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

(Behavior Agent)

← For each timestep t of the game

← With small probability, select a random action (explore), otherwise select greedy action from current policy

DQN Algorithm

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

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 $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

(Behavior Agent)



Take the action (a_t), and observe the reward r_t and next state s_{t+1}



Store transition in replay memory

DQN Algorithm

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

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 $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

(Target Agent)

Experience Replay:
Sample a random
minibatch of transitions
from replay memory
and perform a gradient
descent step

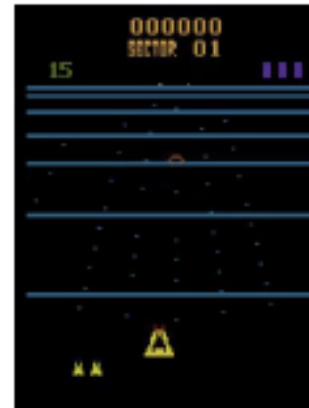
DQN on Atari



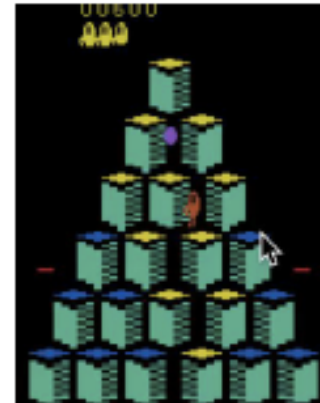
Pong



Enduro



Beamrider



Q*bert

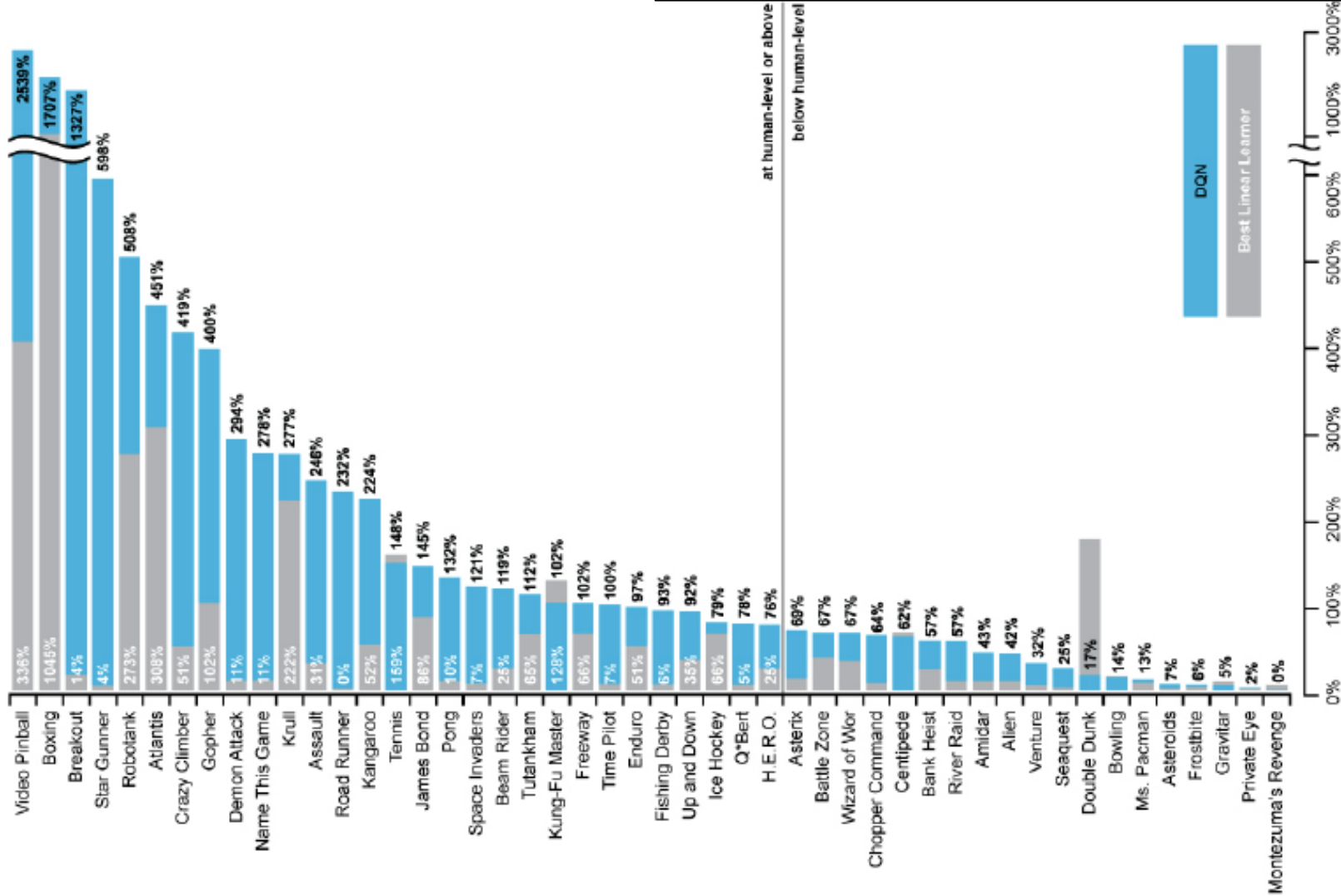
- 49 ATARI 2600 games.
- From pixels to actions.
- The change in score is the reward.
- Same algorithm.
- Same function approximator, w/ 3M free parameters.
- Same hyperparameters.
- Roughly human-level performance on 29 out of 49 games.

Stability Techniques

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

Atari Results

The performance of DQN is normalized with respect to a professional human games tester (that is, 100% level) and random play (that is, 0% level). Note that the normalized performance of DQN, expressed as a percentage, is calculated as: $100 \times (\text{DQN score} - \text{random play score}) / (\text{human score} - \text{random play score})$. It can be seen that DQN outperforms competing methods (also see [Extended Data Table 2](#)) in almost all the games, and performs at a level that is broadly comparable with or superior to a professional human games tester (that is, operationalized as a level of 75% or above) in the majority of games. Audio output was disabled for both human players and agents. Error bars indicate s.d. across the 30 evaluation episodes, starting with different initial conditions.



Asynchronous Methods for Deep RL (2016)

Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih¹
Adrià Puigdomènech Badia¹
Mehdi Mirza^{1,2}
Alex Graves¹
Tim Harley¹
Timothy P. Lillicrap¹
David Silver¹
Koray Kavukcuoglu¹

¹ Google DeepMind

² Montreal Institute for Learning Algorithms (MILA), University of Montreal

VMNIH@GOOGLE.COM
ADRIAP@GOOGLE.COM
MIRZAMOM@IRO.UMONTREAL.CA
GRAVESA@GOOGLE.COM
THARLEY@GOOGLE.COM
COUNTZERO@GOOGLE.COM
DAVIDSILVER@GOOGLE.COM
KORAYK@GOOGLE.COM

Abstract

We propose a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers. We present asynchronous variants of four standard reinforcement learning algorithms and show that parallel actor-learners have a stabilizing effect on training allowing all four methods to successfully train neural network controllers. The best performing method, an asynchronous variant of actor-critic, surpasses the current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU. Furthermore, we show that asynchronous actor-critic succeeds on a wide variety of continuous motor control problems as well as on a new task of navigating random 3D mazes using a visual input.

1. Introduction

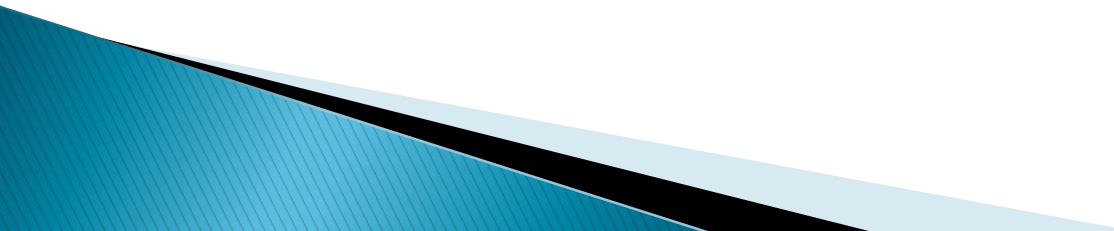
Deep neural networks provide rich representations that can enable reinforcement learning (RL) algorithms to perform effectively. However, it was previously thought that the combination of simple online RL algorithms with deep

line RL updates are strongly correlated. By storing the agent's data in an experience replay memory, the data can be batched (Riedmiller, 2005; Schulman et al., 2015a) or randomly sampled (Mnih et al., 2013; 2015; Van Hasselt et al., 2015) from different time-steps. Aggregating over memory in this way reduces non-stationarity and decorrelates updates, but at the same time limits the methods to off-policy reinforcement learning algorithms.

Deep RL algorithms based on experience replay have achieved unprecedented success in challenging domains such as Atari 2600. However, experience replay has several drawbacks: it uses more memory and computation per real interaction; and it requires off-policy learning algorithms that can update from data generated by an older policy.

In this paper we provide a very different paradigm for deep reinforcement learning. Instead of experience replay, we asynchronously execute multiple agents in parallel, on multiple instances of the environment. This parallelism also decorrelates the agents' data into a more stationary process, since at any given time-step the parallel agents will be experiencing a variety of different states. This simple idea enables a much larger spectrum of fundamental on-policy RL algorithms, such as Sarsa, n-step methods, and actor-critic methods, as well as off-policy RL algorithms such as Q-learning, to be applied robustly and effectively using deep neural networks.

What Problem Are We Solving?

- ▶ We found out how to stabilize Deep Q-Learning by using the DQN algorithm
 - But this was based on exploiting the off-line nature of the Q-Learning algorithm
 - ▶ How can we stabilize Deep On-Line algorithms, such as the deep version of SARSA?
- 

Asynchronous Reinforcement Learning

- ▶ Exploits multithreading of standard CPU
- ▶ Execute many instances of agent in parallel
- ▶ Network parameters shared between threads
- ▶ Parallelism decorrelates data
 - ▶ Viable alternative to experience replay

A3C High Level Architecture

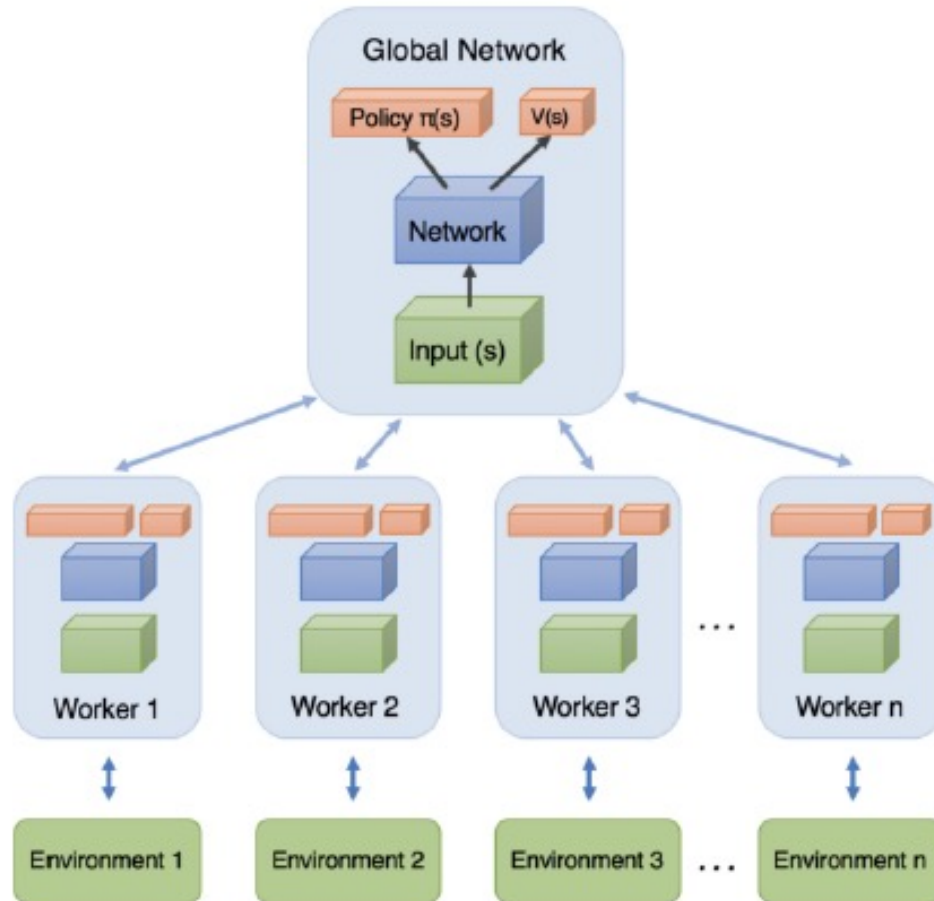
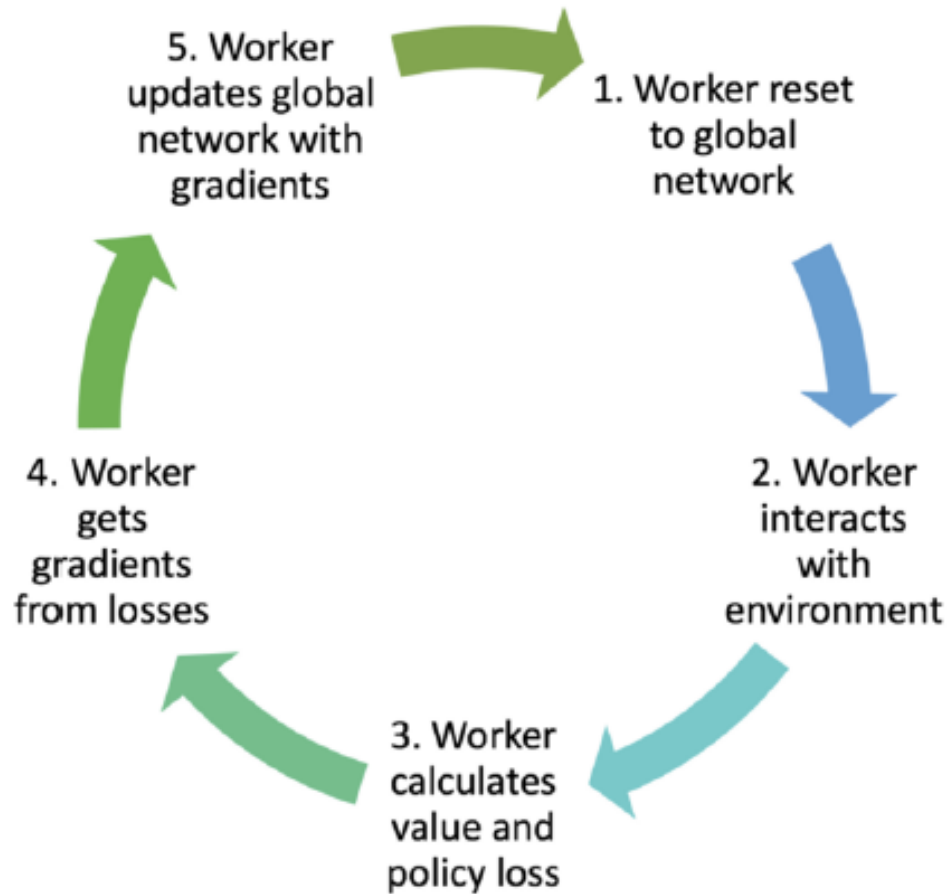


Diagram of A3C high-level architecture.

Training Workflow for Each Worker



A3C Algorithm

- ▶ There is a single global network
- ▶ There are multiple agents, each of which has its own set of neural network parameters
- ▶ Each of these agents interacts with its copy of the environment in parallel with the other agents that are doing the same
 - The experience of each agent is independent of the others since they use their own exploration policies
- ▶ The effect of multiple workers applying online updates in parallel is less correlated than a single agent applying online updates
- ▶ Each Agent may be exploring a different portion of the environment

A3C Algorithm

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ , θ^- , and counter $T = 0$.

Initialize thread step counter $t \leftarrow 0$

Initialize target network weights $\theta^- \leftarrow \theta$

Initialize network gradients $d\theta \leftarrow 0$

Get initial state s

repeat

Take action a with ϵ -greedy policy based on $Q(s, a; \theta)$

Receive new state s' and reward r

$$y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$$

Accumulate gradients wrt θ : $d\theta \leftarrow d\theta + \frac{\partial(y - Q(s, a; \theta))^2}{\partial \theta}$

$s = s'$

$T \leftarrow T + 1$ and $t \leftarrow t + 1$

if $T \bmod I_{target} == 0$ **then**

Update the target network $\theta^- \leftarrow \theta$

end if

if $t \bmod I_{AsyncUpdate} == 0$ or s is terminal **then**

Perform asynchronous update of θ using $d\theta$.

Clear gradients $d\theta \leftarrow 0$.

end if

until $T > T_{max}$

Worker Task accumulates gradients



Further Reading

- ▶ All the Journal papers referred to in the lecture, in particular:
 - “Human Level Control with Deep Reinforcement Learning”
 - “Playing Atari with Deep Reinforcement Learning”
- ▶ Sutton and Barto: Sections 9.1–9.3, 9.7