Function Approximation

Blink Sakulkueakulsuk

Four main Bellman Equations

Bellman Expectation Equation

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})\,|S_t = s, A_t \sim \pi(s)]$$
$$q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})\,|S_t = s, A_t = a]$$

Bellman Optimality Equation

$$v^*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1})\,|S_t = s, A_t = a]$$
$$q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q^*\,(S_{t+1}, a')\,\middle|\,S_t = s, A_t = a)\right]$$

Prediction and Control

Prediction

- A problem when we perform **policy evaluation**
- Estimating v_π or q_π

Control

- A problem when we perform **policy optimization**
- Estimating v^* or q^*

Policy Evaluation

Given a policy, we want to estimate this

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \,|s,\pi]$$

We initialize all value to zero, and we iterate the equation above as an update

$$v_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \,|s,\pi] \quad , \quad \forall s$$

Note that

$$v_\pi(s) = \sum_a \pi(a|s) \sum_r \sum_{s\prime} p(r,s'|s,a)(r + \gamma v_\pi(s'))$$

If $v_{k+1}(s) = v_k(s), \forall s$, we solve v_π.

Policy Improvement

The example shows that we can improve a policy when we learn the value.

- In the example, we do not improve any policy

$$\begin{aligned}\forall s: \pi_{new}(s) &= argmax_a\, q_\pi(s,a) \\ &= argmax_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a]\end{aligned}$$

We use the new policy to evaluate, and repeat.

Monte-Carlo Policy Evaluation

Given the sequential decision problem, MC learns v_π from episodes under policy π

$$S_1, A_1, R_2, \dots, S_k \sim \pi$$

The return is calculated given an ending time T

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T$$

The value function is then calculated from the expected return

$$v_\pi(s) = \mathbb{E}[G_t | S_t = s, \pi]$$

We can use **sample average return** instead of **expected return**

- This is called **Monte Carlo policy evaluation**

Temporal Difference Learning

$$v_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t \sim \pi(S_t)]$$

Instead of calculating the expected return, we can sample the value.

$$v_{t+1}(S_t) = R_{t+1} + \gamma v_t(S_{t+1})$$

But instead of updating everything on the noisy value, we can update the value a little bit instead.

$$v_{t+1}(S_t) = v_t(S_t) + \alpha_t(R_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t))$$

MC Prediction vs TD Prediction

Prediction problem: learn v_π online from experience under policy π

Monte Carlo:

- Update value $v_n(S_t)$ with respect to sample return G_t

$$v_n(S_t) = v_n(S_t) + \alpha(G_t - v_n(S_t))$$

Temporal-difference learning:

- Update value $v_t(S_t)$ with respect to estimated return $R_{t+1} + \gamma v(S_{t+1})$

$$v_{t+1}(S_t) = v_t(S_t) + \alpha_t(R_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t))$$

Bootstrapping and Sampling

Bootstrapping

- Use the estimate of the next state to update
- DP and TD use
- MC does not

Sampling

- Update samples an expectation
- MC and TD sample
- DP does not

Temporal Difference Learning

We can also learn action values by updating value $q_t(S_t, A_t)$ with respect to estimated return $R_{t+1} + \gamma v(S_{t+1}, A_{t+1})$

$$q_{t+1}(S_t, A_t) = q_t(S_t, A_t) + \alpha_t(R_{t+1} + \gamma q_t(S_{t+1}, A_{t+1}) - q_t(S_t, A_t))$$

This algorithm is called SARSA, because it uses $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$

Multi-Step Returns

In general, we can formalize n-step return as

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ \dots \ + \gamma^{n-1} v(S_{t+1})$$

We can define multi-step temporal-difference learning as

$$v(S_t) \leftarrow v(S_t) + \alpha(G_t^{(n)} - v(S_t))$$

Mixing Multi-Step Returns

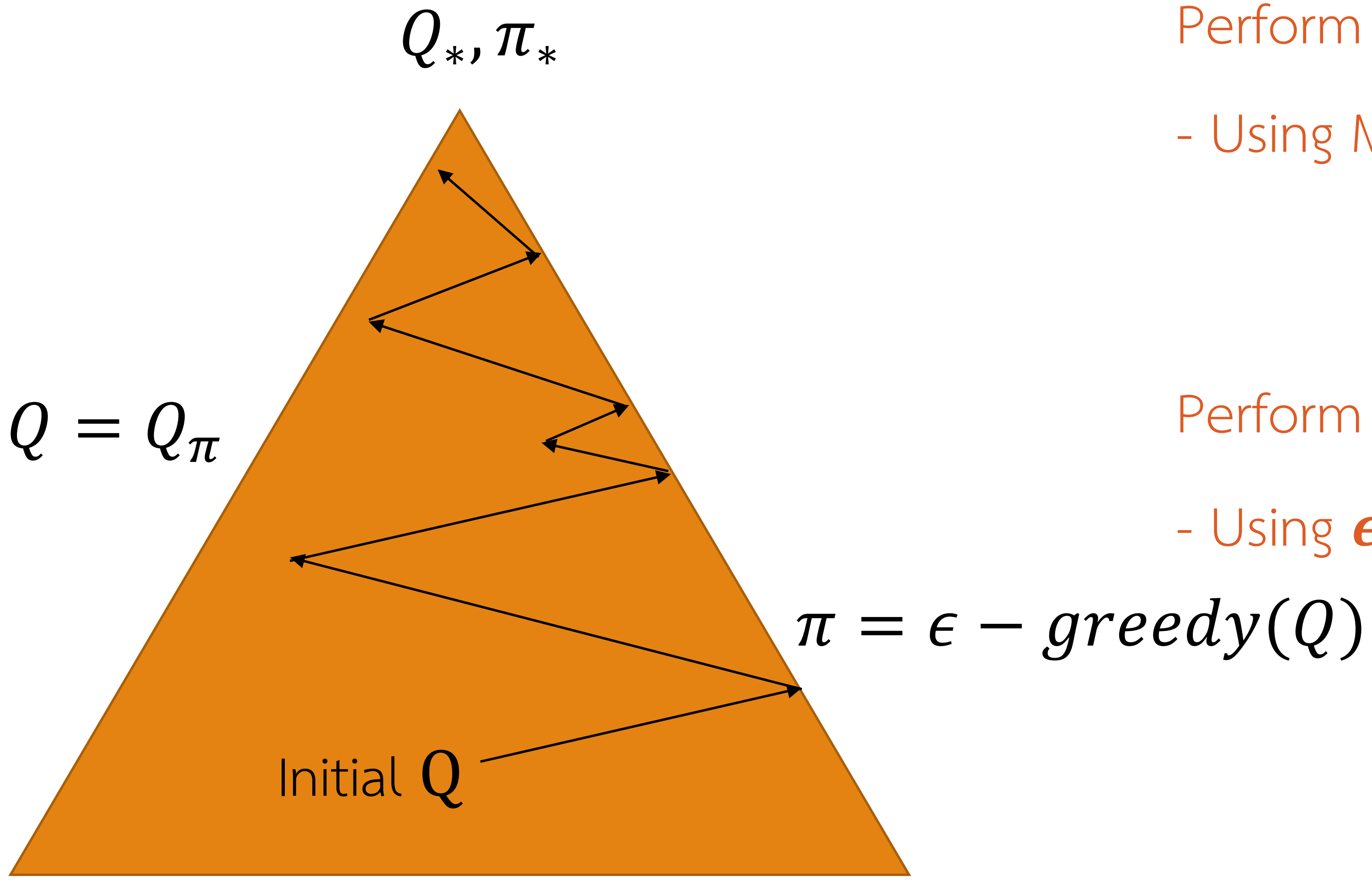
Consider a multi-step return equation,

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ \dots \ + \gamma^{n-1} v(S_{t+1})$$

Multi-step returns bootstrap on one state, $v(S_{t+n})$. However, we can modify the equation so you can bootstrap a little bit on many states

$$G_t^{\lambda} = R_{t+1} + \gamma((1-\lambda)v(S_{t+1}) \ + \lambda G_{t+1}^{\lambda})$$

Monte Carlo Control



Perform policy evaluation

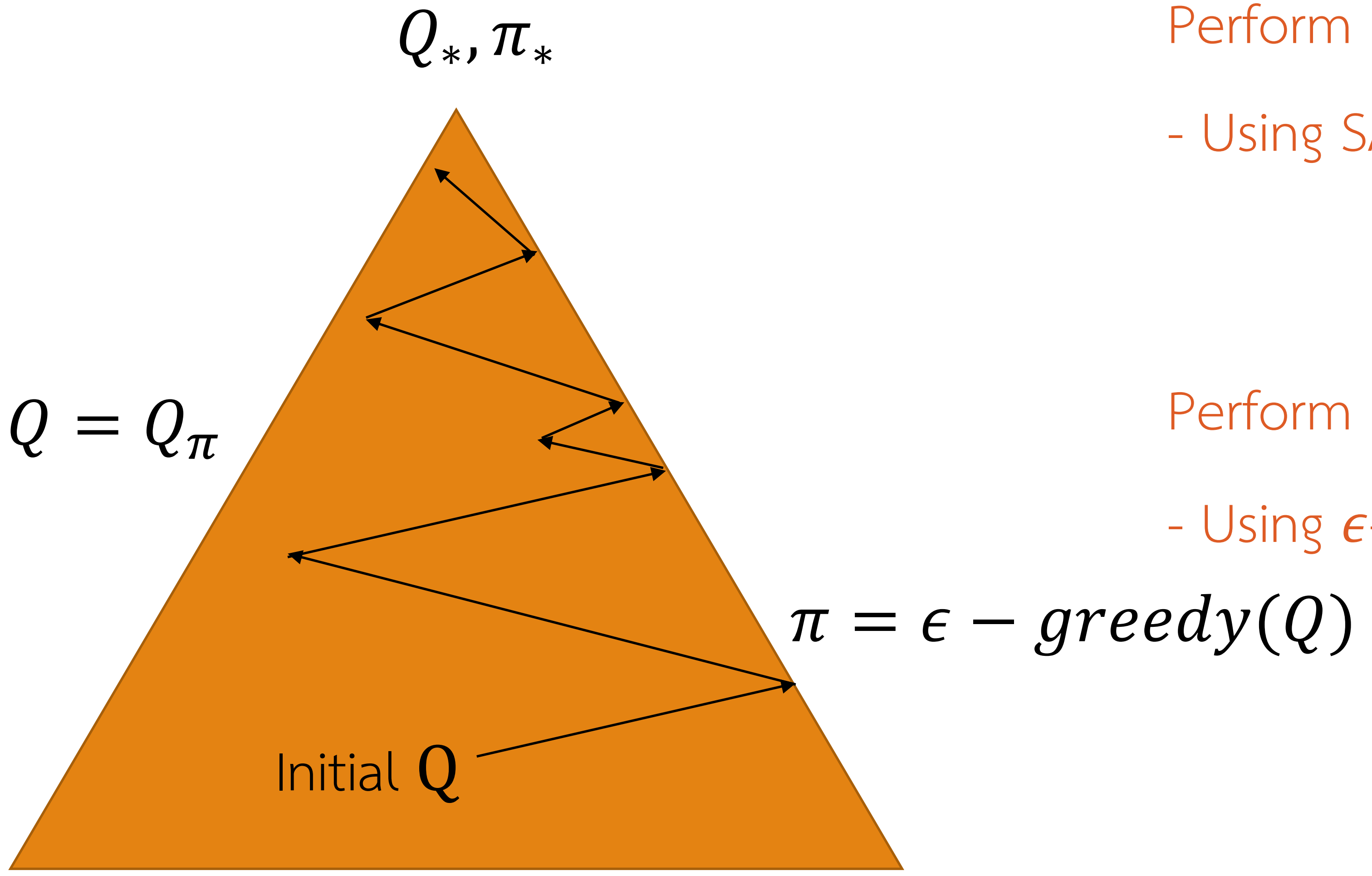
- Using Monte Carlo policy evaluation

$$q \approx q_\pi$$

Perform policy improvement

- Using **ϵ-greedy** policy improvement

SARSA



Perform policy evaluation

- Using SARSA

$$q \approx q_\pi$$

Perform policy improvement

- Using ϵ-greedy policy improvement

On-Policy and Off-policy Learning

On-policy learning

- Learn from actually performing
- Learn a policy π from samples from π

Off-policy learning

- Learn from other sources
- Learn a policy π from samples from b

Q-learning

Q-learning is an off-policy, model-free reinforcement learning

หา action ที่ได้ค่าที่ดีที่สุด

$$q_{t+1}(S_t, A_t) = q_t(S_t, A_t) + \alpha_t\left(R_{t+1} + \gamma \max_{a'} q_t(S_{t+1}, a') - q_t(S_t, A_t)\right)$$

Q-learning learns from the value of the greedy policy, thus an off-policy learning.

- It converges to the optimal action-value function when we explore all state-action infinitely many times
- No greedy behavior needed

Double Q-learning

Since

$$\max_a q_t(S_{t+1}, a) = q_t(S_{t+1}, argmax_a q_t(S_{t+1}, a))$$

Q-learning will use the same values to select action, and evaluate the value

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

And this will result in the issue in the last slide. To solve this problem, we can introduce another policy to split action selection from evaluation.

Double Q-learning

Double Q-learning will

- store two q functions: q and q'

- update one q function for each iteration.

$$R_{t+1} + \gamma q'_t(S_{t+1}, argmax_a q_t(S_{t+1}, a))$$

$$R_{t+1} + \gamma q_t(S_{t+1}, argmax_a q'_t(S_{t+1}, a))$$

} เลือกอย่างใดอย่างนัง
ในการอัพเดทแต่ละรอบ

Off-policy Q-Learning Control

To control, we want to optimize both policies.

For the target policy π, we greedify the action on $q(s,a)$

$$\pi(S_{t+1}) = argmax_{a'} q(S_{t+1}, a')$$

For the behavior policy b, we can use ϵ-greedy on $q(s,a)$

all data of 6 weeks before is clear

but in big problem we cannot

Thus, Q-Learning target is simplified as:

$$R_{t+1} + \gamma \sum_{a} \pi(a|S_{t+1}) q(S_{t+1}, a) = R_{t+1} + \gamma \max_{\mathrm{a}} q(S_{t+1}, a)$$

Function Approximation

ประมาณค่า function อะไรบ้างอย่าง

Function Approximation

If we look closely, there are a lot of functions in many parts of reinforcement learning.

- policy	a function that outputs an action given a state
- value function	a function that outputs a value of a state
- agent state update	a function that outputs a next state given some inputs
- model	a function that outputs dynamic of the environment given some inputs

Function Approximation

All the functions are what we want to learn, but what if the problem space is too large?

- There are too many states, or too many actions

OR continuous 6 weeks before is useless. focus this week!!!

We can approximate these functions.

- This is often called **deep reinforcement learning**, when we use **deep neural network** to represent these functions.

Function Approximation and Large Problem Space

In recent years, deep reinforcement learning is used to solve large problems.

before dynamic problem can solve mediem problems.

- Go: 10^{170} states
- Drone: continuous state space
- Robots: real world space

Value Function Approximation

ก่อนหน้าเราจินตนาการในรูปแบบ array or sth.

So far, we use **lookup tables** to store all values.

- Storing a value $v(S)$ for each state S

- Storing an state-action value $q(S, A)$ for each pair of S, A

In a large problem, this method will run into problems.

- Too many states to store in memory

- Too slow to learn values of states individually

- States may not be fully observable

Value Function Approximation

To solve large MDPs, we can estimate the value function using function approximation instead

W - matrix , vector

รวมมองเป็น ML model

function Approximation

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

look up table

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

เอาฝั่งซ้าย มาประมาณ ค่าฝั่งขวา

รู้ขวา ระดับนึง และเอามาปรับให้เหมาะ

supervised learning

- We can generalize new states (unexplored) from old states (explored)

- We can use MC or TD to learn parameter $\mathbf{w}$.

Agent State Approximation

If we encounter a non-fully observable problem, we can approximate agent state instead.

agent state transition

vector

input vector can represent state

$$\mathbf{s}_t = u_\omega(\mathbf{s}_{t-1}, A_{t-1}, O_t)$$; update s_t

parameter

$\omega \in \mathbb{R}^n$ denotes learnable parameters

$\mathbf{s}_t$ denotes the agent state

- Think of this as a vector of features in a state

Function Approximation Classes

class 1

Tabular: A table containing states

State aggregation: Partitioning environment states into discrete states

ดู transition เป็น ช่อง สองห้อง and represent ในช่วง ว่าเป็น ตัวเดียวกัน

class 2

Linear function approximation

- Fixed agent state update → fixed dimension
- Fixed feature map $\mathbf{x} : S \Rightarrow \mathbb{R}^n$ → fixed feature
- Linear function as value function approximation $v_{\mathbf{w}}(s) = \mathbf{w}^{\mathrm{T}}\mathbf{x}(s)$

Class 3

Differentiable function approximation

- $v_{\mathbf{w}}(s)$ is a differentiable function. It can be non-linear.
- We can utilize Deep Learning to learn features, such as CNN.

Function Approximation Classes

Generally, any function approximator works. But reinforcement Learning has some specific properties too keep in mind.

มีการกระจายตัวเหมือนกัน และ independent

- Experience is not independent and identically distributed (i.i.d.). Current and future actions are correlated.

- Policy affects the observations and rewards

- Learning target can be non-stationary $\overset{\text{target}}{Y}$ เปลี่ยนไปเรื่อยๆตาม policy

Policy change, bootstrapping, dynamic of the problem

Function Approximation

How to choose what to use? * ใช้ไม่ได้ตรงๆ ขนาดนั้น

Tabular	Good theory, not scalable
Linear function approximation	Good theory, need good feature construction
Non-linear function approximation	Scalable, but not many theories to back up (yet) Less reliable on picking good features

MATH

Gradient Descent Revisited

Given $J(\mathbf{w})$, a differentiable function of parameter vector $\mathbf{w}$

$\downarrow$ หา gradient

The gradient of $J(\mathbf{w})$ is

Partial

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

Gradient Descent Revisited

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

We want to find the minimum value of $J(\mathbf{w})$. To do that, we can update $\mathbf{w}$ such that it moves in the direction of negative gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

α is learning rate or step-size parameter.

Approximation using Stochastic Gradient Descent

* point is direction not exact value *

Given an approximator with parameters $\mathbf{w}$, we want to minimize the value difference between this approximator $v_{\mathbf{w}}(S)$ and true value $v_\pi(S)$

คือ ?

error²

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

expectation ที่ถูก sth

constant

learn w ที่ทำให้ ก้อน $v_w(s)$ เปลี่ยน

d denotes a distribution of states

So the gradient descent is:

ตัด ⊖ cancel dif

learning rate

diff แล้ว

$$\Delta\mathbf{w} = -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) = \alpha\mathbb{E}_d\left[\left(v_\pi(S) - v_{\mathbf{w}}(S)\right)\nabla_{\mathbf{w}}v_{\mathbf{w}}(S)\right]$$

sample 1 ep. meaning expectation หายไป

discrete state (เล่นต่อๆ ไม่ได้ต้อง ดู next slide)

In sampling algorithm (MC,TD), **Stochastic** gradient descent can be used to sample the gradient.

$$\Delta\mathbf{w} = \alpha\left(v_\pi(S_t) - v_{\mathbf{w}}(S_t)\right)\nabla_{\mathbf{w}}v_{\mathbf{w}}(S_t)$$

Feature Vectors

If we want to approximate some functions, generally, it relies on states. To make it computable, we represent a state as a **feature vector**

$$\overset{\substack{\text{fixed mapping} \\ \text{state to feature} \\ \uparrow}}{\mathbf{x}}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

$\mathbf{x}: S \rightarrow \mathbb{R}^n$ is a fixed mapping from state to features

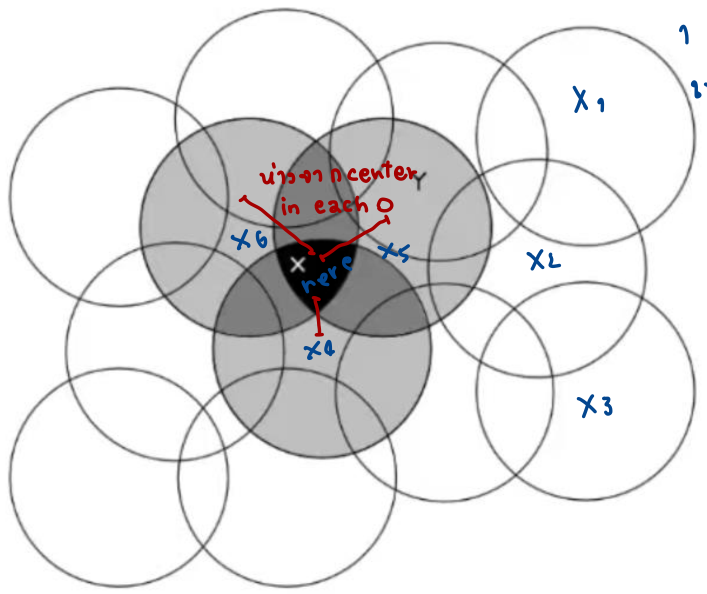
We can represent a feature vector of state S_t as $\mathbf{x}_t = \mathbf{x}(S_t)$

- Distance of a robot from reference points

- Objects in an image

Coarse Coding วิธีการสร้าง feature vector

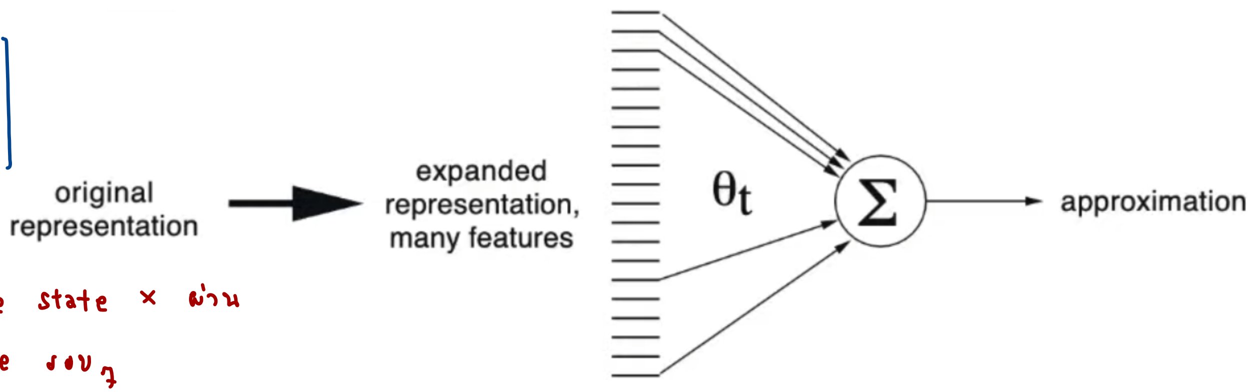
Coarse coding is a method to construct a feature vector through state representation.



1 วง = 1 feature ใน feature vector

ex. $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{bmatrix}$

* define state × ผ่าน feature รอบๆ



Linear Value Function Approximation

Let’s consider a simple case, a linear function approximator. We can express the value function as

$$v_{\mathbf{w}}(S) = \underset{\substack{\text{weight}\\ \text{feature}}}{\mathbf{w}}^{\mathrm{T}}\overset{\sqsupset}{\mathbf{x}}(S) = \sum_{j=1}^{n} \mathbf{x}_j(S)\mathbf{w}_j$$

The loss of this value function is then defined as

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

Linear Value Function Approximation

ex. $v_w(x) = w_1 v_1 + v_2 x_2 + \dots$

$$\nabla v = \begin{bmatrix} \frac{\partial(v_w(x))}{\partial w_1} \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \end{bmatrix} = \vec{x}$$

$$\frac{\partial v_w(x)}{\partial w_n} = \vec{x}$$

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

Then we can use stochastic gradient descent to update the parameter

- Stochastic gradient descent is a gradient descent algorithm that calculates loss of 1 input at a time
- We can update the parameter as

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \mathbf{x}(S_t) = \mathbf{x}_t \qquad \Rightarrow \qquad \Delta\mathbf{w} = \alpha\big(v_\pi(S_t) - v_{\mathbf{w}}(S_t)\big)\mathbf{x}_t$$

Update = step-size x prediction error x feature vector

Learning Targets

$$\Delta \mathbf{w}_t = \alpha\left(v_\pi(S_t) - v_{\mathbf{w}}(S_t)\right)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

We cannot update $v_{\mathbf{w}}(S_t)$ toward $v_\pi(S_t)$ since we do not know the true value. However, we can change the target according to algorithms we use.

For MC, we use the return G_t

$$\Delta \mathbf{w}_t = \alpha\left(\overset{\overset{\text{return (ผลรวม reward) / 1 episode}}{\uparrow}}{G_t} - \underset{\underset{\text{learnable param}}{\downarrow}}{v_{\mathbf{w}}}(S_t)\right)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

For TD, we use TD target $R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})$

$$\Delta \mathbf{w}_t = \alpha\left(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)\right)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

For TD(λ),

$$\Delta \mathbf{w}_t = \alpha\left(G_t^\lambda - v_{\mathbf{w}}(S_t)\right)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$
$$G_t^\lambda = R_{t+1} + \gamma\left((1-\lambda)v_w(S_{t+1}) + \lambda G_{t+1}^\lambda\right)$$

Monte-Carlo with Value Function Approximation

The return of MC is an unbiased, noisy sample of $v_\pi(s)$

We can use (online) supervised learning to train the model

$$\{(\overset{\text{input}}{S_0}, \overset{\text{output}}{G_0}), \dots, (S_{\underset{i}{\cancel{t}}}, G_{\cancel{t}\, i})\}$$

many episode

(trend like dataset

$$\Big\downarrow \ \text{plugin}$$

So, the linear MC updates is

$$\Delta \mathbf{w} = \alpha\big(G_{\underset{i}{\cancel{t}}} - v_{\mathbf{w}}(S_{\underset{i}{\cancel{t}}})\big)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_{\underset{i}{\cancel{t}}}) = \alpha\big(G_{\underset{i}{\cancel{t}}} - v_{\mathbf{w}}(S_{\underset{i}{\cancel{t}}})\big)\mathbf{x}_t$$

MC with linear approximation converges to the global optimum

MC with non-linear value function approximation also converges (to local or global optimum)

TD Learning with Value Function Approximation

The target of TD is a biased sample of $v_\pi(S_t)$

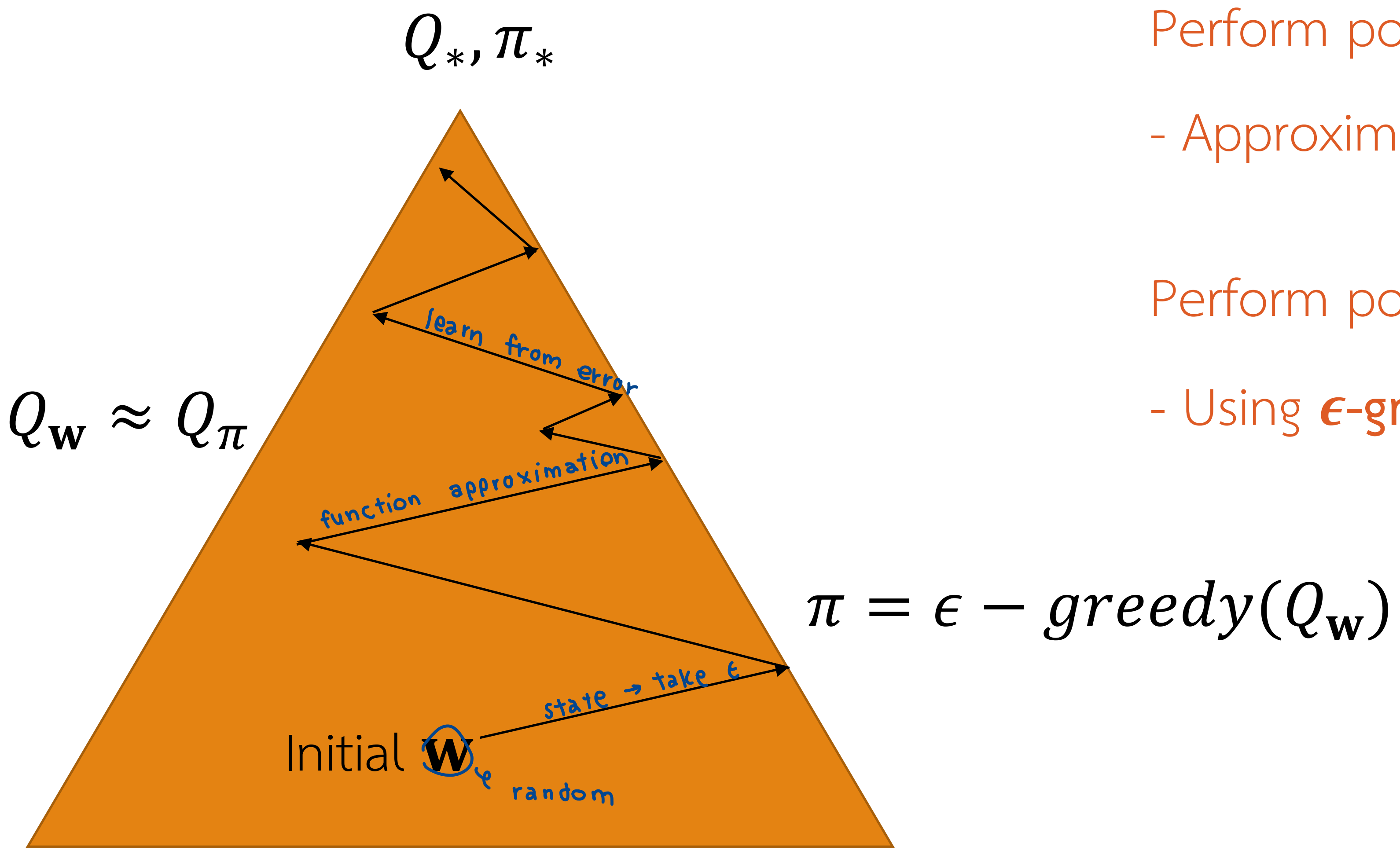
We can still use supervised learning to train the model

$$\{(S_0, R_1 + \gamma v_{\mathbf{w}}(S_1)), \dots, (S_t, R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}))\}$$

So, the **linear TD** updates is

$$\Delta \mathbf{w}_t = \alpha\big(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)\big)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \alpha \delta_t \mathbf{x}_t$$

Control with Value Function Approximation



Perform policy evaluation

- Approximate policy evaluation

$$q_{\mathbf{w}} \approx q_\pi$$

Perform policy improvement

- Using ***ϵ*-greedy** policy improvement

Action-Value Function Approximation

We can apply function approximation to action-value function $q_{\mathbf{w}}(s,a) \approx q_{\pi}(s,a)$

For example, the linear action-value function is

$$q_{\mathbf{w}}(s,a) = \mathbf{x}(s,a)^{\mathrm{T}}\mathbf{w}$$

Stochastic gradient descent update is

$$\begin{aligned}\Delta\mathbf{w} &= \alpha\big(q_{\pi}(s,a) - q_{\mathbf{w}}(s,a)\big)\nabla_{\mathbf{w}} q_{\mathbf{w}}(s,a) \\ &= \alpha\big(q_{\pi}(s,a) - q_{\mathbf{w}}(s,a)\big)\mathbf{x}(s,a)\end{aligned}$$

Deep Reinforcement Learning

We can use deep neural networks as our function approximator

- Apply DNN to approximate value function or action-value function
- Using DNN with MC, TD, Q, or Double-Q

OMG

Online Neural Q-Learning

Neural network: $O_t \rightarrow \mathbf{q}_\mathbf{w}$

Exploration policy: $\pi_t = \epsilon\text{-greedy}(\mathbf{q}_t)$, then $A_t \sim \pi_t$

Weight update:

$$\Delta\mathbf{w} = k\left(R_{t+1} + \gamma \max_a q_\mathbf{w}(S_{t+1}, a) - q_\mathbf{w}(S_t, A_t)\right)\nabla_\mathbf{w} q_\mathbf{w}(S_t, A_t)$$

Optimizer: SGD, Adam,...

train by

Deep Q-Network

Neural network: $O_t \rightarrow \mathbf{q_w}$

Exploration policy: $\pi_t = \epsilon$-greedy$(\mathbf{q}_t)$, then $A_t \sim \pi_t$

A replay buffer to store and sample past transitions $(S_i, A_i R_{i+1}, S_{i+1})$

$\downarrow$ for feature

Target network parameters $\mathbf{w}^-$

Weight update:

$$\Delta \mathbf{w} = \left(R_{t+1} + \gamma \max_a q_{\mathbf{w}^-}(S_{t+1}, a) - q_{\mathbf{w}}(S_t, A_t)\right) \nabla_{\mathbf{w}} q_{\mathbf{w}}(S_t, A_t)$$

Update $w_t^- \leftarrow w_t$ periodically (i.e., every 1000 steps)

Optimizer: SGD, Adam,...