# Lecture 7: Function Approximation

**Chonggang Wang, PhD, IEEE Fellow**

InterDigital, Inc., USA

Office Hours: 10:30-11:30 AM on Fridays

Email: cw3403@columbia.edu

# RL Introduction: Schedule

- Lecture 1: Introduction to Reinforcement Learning
- Lecture 2: Bandit Problem and MDP
- Lecture 3:  Model-based RL
- Lecture 4:  Model-free RL (Part I)
- Lecture 5:  Model-free RL (Part II)
- Lecture 6:  Eligibility Traces
- Lecture 7: Function Approximation (11/4)
- Lecture 8: Policy Gradient (11/11)
- Lecture 9: Planning and Learning (11/18)
- Lecture 10: Deep Reinforcement Learning (12/02)
- Lecture 11: Advanced RL Topics (12/09)
- **Final: 12/16**

Prof. Chong Li

Prof. Chonggang Wang

# Recap of Previous Lectures 1-6

- Model-based RL
  - Dynamic Programming (DP) – Bootstrapping
    - Policy Iteration: Policy Evaluation + Policy Improvement
    - Value Iteration
- Model-free RL
  - Monte-Carlo (MC) – No Bootstrapping (Unbiased)
  - Temporal-Difference (TD) – Bootstrapping
    - Policy Evaluation (State Value v(s)): TD(0), n-Step TD, TD($\lambda$)
    - Policy Control (Action Value q(s, a)): Sarsa (On-Policy), Q-Learning (Off-Policy)
- All are Tabular Methods
  - Each update will only change the value of one *state* or one *state-action* pair, i.e., an entry in the *lookup table*
  - The lookup table may become unmanageable when the number of "*states*" or "*state-action*" pairs goes up
  - Good for episodic tasks, not for continuing tasks
  - ……

# Outline – Function Approximation

- Introduction & Preliminaries
- RL Prediction with Function Approximation
- RL Control with Function Approximation
- Batch Method for RL Applications

*materials are modified from David Silver's RL lecture notes

# Outline – Function Approximation

- Introduction & Preliminaries
- RL Prediction with Function Approximation
- RL Control with Function Approximation
- Batch Method for RL Applications

# Function Approximation

- **What** is Function Approximation?
  - To approximate value function **v(s)** and action value function **q(s, a)** in a parameterized functional format
    - v(s) → v(s, w) for all s; q(s, a) → q(s, a, w) for all (s, a) pairs
- **Why** do we need Function Approximation?
  - To evaluate/predict v(s) and q(s, a) for large or high-dimension state space
  - To evaluate/predict v(s) and q(s, a) for continuing tasks (non-episodic)
  - To evaluate/predict v(s) and q(s, a) for partially observable problems
- **How** to achieve Function Approximation?
  - To use supervised learning based on experience data (i.e., training examples)
    - To define an **Objective Function**: Mean-Squared Value Error (VE)
    - To leverage **Stochastic Gradient Descent** to search for optimal parameters w for estimated functions v(s, w) and q(s, a, w), in the fastest direction to minimize the Objective Function

# Motivation

- How to solve large-scale reinforcement learning problems:
  - Backgammon: 10^20 states
  - Computer GO: 10^170 states
  - Autonomous driving:  continuous state space
- Curse of dimensionality
  - How to leverage RL to achieve optimal control with the exponential growth of states and actions

# 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*

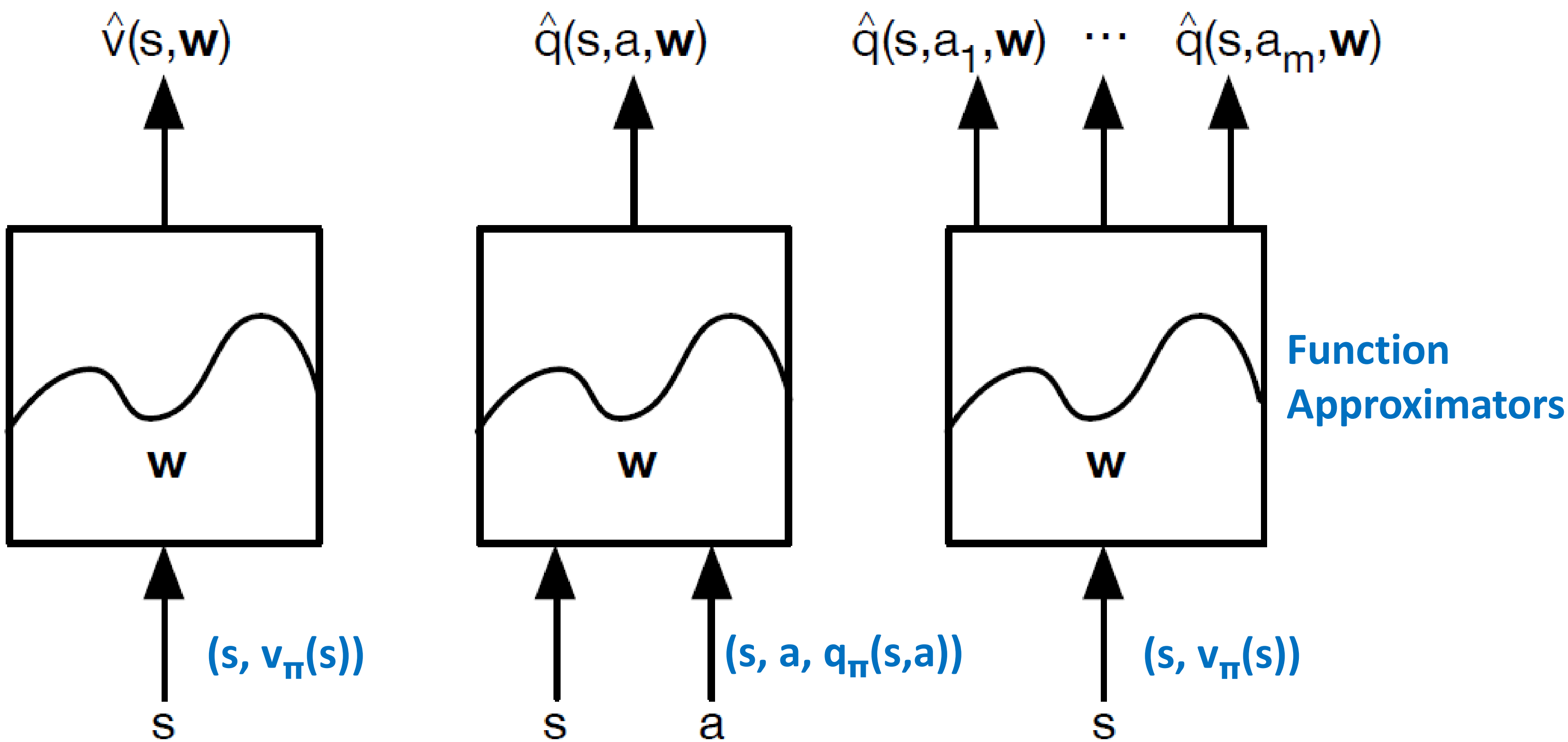
Approximate Value

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

True Value

*Generalise* from seen states to unseen states
*Update* parameter $\mathbf{w}$ using MC or TD learning

# Types of Approximation




- Typically, the number of weights (the dimensionality of *w*) is much less than the number of states $|S|$
- To update one weight will change the estimated value of many states

# Which Function Approximator?

- There are many function approximators:
  - Linear: linear combinations of features
  - Non-linear: neural networks
  - Decision tree
  - ….

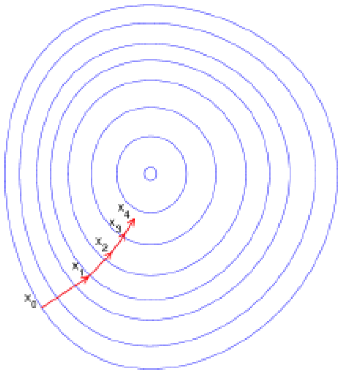
- We consider ***differentiable*** function approximators in this lecture:
  - Linear: linear combinations of features
  - Non-linear: neural networks
    - A static training set, uncorrelated data, stationary data, iid data

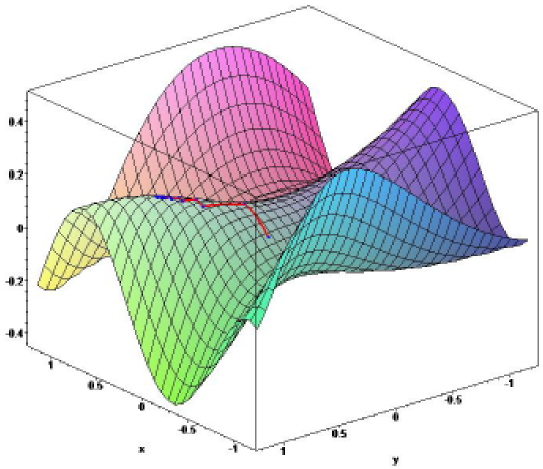
# Gradient Descent

- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$, a column vector
- Define the *gradient* of $J(\mathbf{w})$ to be

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

Partial Derivatives with respect to **w**





- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of -ve gradient to reduce *J*(w) (i.e., the Value Error (VE))

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

# Stochastic Gradient Descent

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

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

- Gradient descent finds a local minimum

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

Power Rule & Chain Rule for Derivatives

$$= \alpha\mathbb{E}_\pi\left[(\underline{v_\pi(S)} - \hat{v}(S, \mathbf{w}))\nabla_\mathbf{w}\hat{v}(S, \mathbf{w})\right]$$

Assume it is irrespective of *w*

- Stochastic gradient descent *samples* the gradient

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

- Expected update is equal to full gradient update

# Stochastic Gradient Descent - Example

- Suppose we want to fit a straight line

$$y = w_1 + w_2 x$$ **Approximate Function**

- Use data set

$$(x_1, y_1), \ldots, (x_n, y_n)$$ **True Function/Data**

- Objective function: Mean-Squared Value Error

$$J(w) = \sum_{i=1}^{n} (\underbrace{w_1 + w_2 x_i}_{\text{Estimated}} - \underbrace{y_i}_{\text{True Value}})^2$$

- SGD: sweep through the training set

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \eta \begin{bmatrix} 2(w_1 + w_2 x_i - y_i) \\ 2x_i(w_1 + w_2 x_i - y_i) \end{bmatrix}$$

Partial derivative of *Q*(w) with respect to *w*1

Partial derivative of *Q*(w) with respect to *w*2

# Outline

- Introduction & Preliminaries
- RL Prediction with Function Approximation
  - To find $v(s, \mathbf{w})$
- RL Control with Function Approximation
  - To find $q(s, a, \mathbf{w})$

**Incremental** Methods

- **Batch** Method for RL Applications
  - **Least-Square Method:** to find the best value function ($v(s, \mathbf{w})$) based on an experience data set by minimizing the sum of the errors/offsets.
  - Achieves a better utilization of samples

# RL Prediction with Value Approximation

- 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 $G_t$

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

  - For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$

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

  - For TD($\lambda$), the target is the $\lambda$-return $G_t^\lambda$

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

# How to compute the gradient?

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

- We can compute it for
  - Linear: linear combinations of features
  - Non-linear: neural networks

# 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 and pawn configurations in chess

- Ways to combine features
  - Polynomials
  - Fourier basis
  - Coding techniques
  - ….

# Linear 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**

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

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$

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

Update $=$ *step-size* $\times$ *prediction error* $\times$ *feature value*

# Table Lookup Features

- Table lookup is a special case of linear value function approximation
- Using *table lookup features*

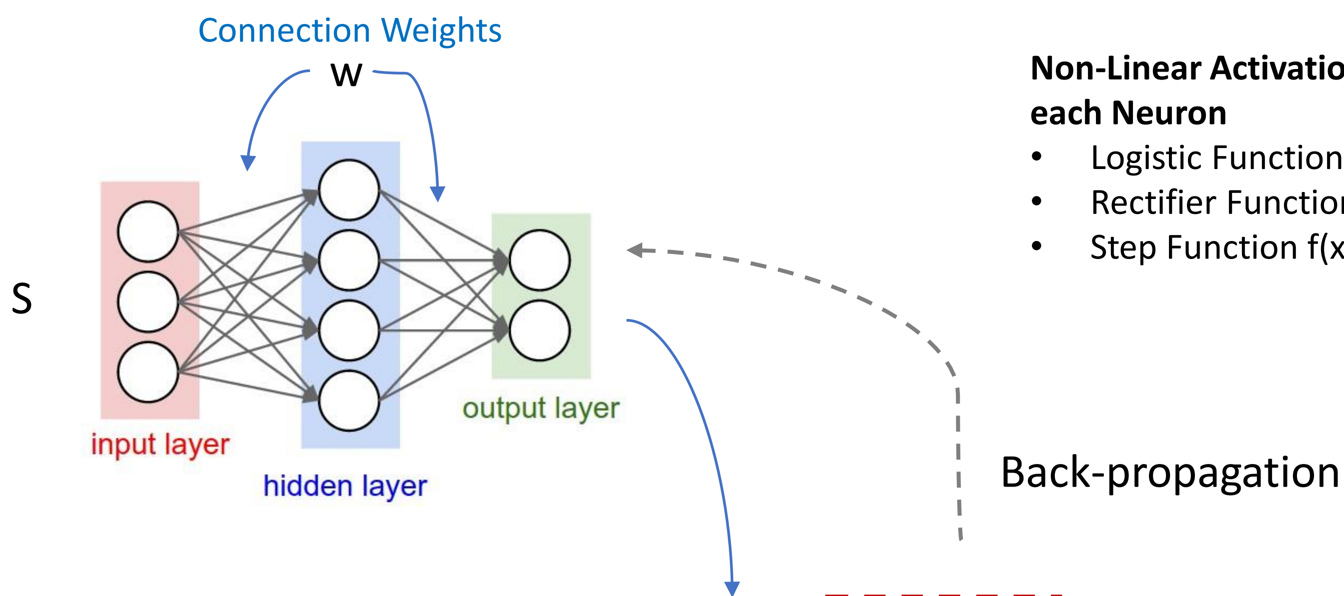
$$\mathbf{x}^{table}(S) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix}$$

- Parameter vector $\mathbf{w}$ gives value of each individual state

$$\hat{v}(S, \mathbf{w}) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix}$$

# Non-linear Approximation

- Neural network (works fine for static training set, uncorrelated data, stationary/iid data)




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

**Non-Linear Activation Functions at each Neuron**

- Logistic Function $f(x) = 1/(1+e^{-x})$
- Rectifier Function $f(x) = \max(0, x)$
- Step Function $f(x) = 1$ if $x>=0$

# Monte-Carlo with Value Function Approximation

- Return $G_t$ is an unbiased, noisy sample of true value $v_\pi(S_t)$
- Can therefore apply supervised learning to "training data":

$$\langle S_1, G_1\rangle, \langle S_2, G_2\rangle, ..., \langle S_T, G_T\rangle$$

- For example, using *linear Monte-Carlo policy evaluation*

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)\end{aligned}$$

- Monte-Carlo evaluation converges to a local optimum
- Even when using non-linear value function approximation

# Monte-Carlo with Value Function Approximation

- Input: The policy $\pi$ to be evaluated; a differentiable function $S \times R^d \rightarrow R$
- Algorithm Parameter: Step size $\alpha > 0$ (e.g., $\alpha = 1/t$)
- Initialize value-function weights $\boldsymbol{w} \in R^d$ arbitrarily (e.g., $\boldsymbol{w} = 0$)
- Loop forever (for each episode):

  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using the policy $\pi$

  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha * [G_t - \hat{v}(S_t, \boldsymbol{w})] * \nabla \hat{v}(S_t, \boldsymbol{w})$$

# TD(0) with Value Function Approximation

- The TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ is a *biased* sample of true value $v_\pi(S_t)$
- Can still apply supervised learning to "training data":

$$\langle S_1, R_2 + \gamma \hat{v}(S_2, \mathbf{w}) \rangle, \langle S_2, R_3 + \gamma \hat{v}(S_3, \mathbf{w}) \rangle, ..., \langle S_{T-1}, R_T \rangle$$

- For example, using *linear TD(0)*

$$\begin{aligned} \Delta \mathbf{w} &= \alpha (R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \\ &= \alpha \delta \mathbf{x}(S) \end{aligned}$$

- Linear TD(0) converges (close) to global optimum

# TD(λ) with Value Function Approximation

- The $\lambda$-return $G_t^\lambda$ is also a biased sample of true value $v_\pi(s)$
- Can again apply supervised learning to "training data":

$$\left\langle S_1, G_1^\lambda \right\rangle, \left\langle S_2, G_2^\lambda \right\rangle, ..., \left\langle S_{T-1}, G_{T-1}^\lambda \right\rangle$$

- Forward view linear TD($\lambda$)

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t) \end{aligned}$$

- Backward view linear TD($\lambda$)

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t \end{aligned}$$

# TD(λ) with Value Function Approximation

Initialize $\mathbf{w}$ as appropriate for the problem, e.g., $\mathbf{w} = \mathbf{0}$

Repeat (for each episode):

  $\mathbf{z} = 0$

  $S \leftarrow$ initial state of episode

  Repeat (for each step of episode):

    $A \leftarrow$ action given by $\pi$ for $S$

    Take action $A$, observe reward, $R$, and next state, $S'$

    $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

    $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$

    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$

    $S \leftarrow S'$

  until $S'$ is terminal

# Convergence of Prediction Algorithms

| On/Off-Policy | Algorithm | Table Lookup | Linear | Non-Linear |
|---|---|---|---|---|
| On-Policy | MC | ✓ | ✓ | ✓ |
| | TD(0) | ✓ | ✓ | ✗ |
| | TD($\lambda$) | ✓ | ✓ | ✗ |
| Off-Policy | MC | ✓ | ✓ | ✓ |
| | TD(0) | ✓ | ✗ | ✗ |
| | TD($\lambda$) | ✓ | ✗ | ✗ |

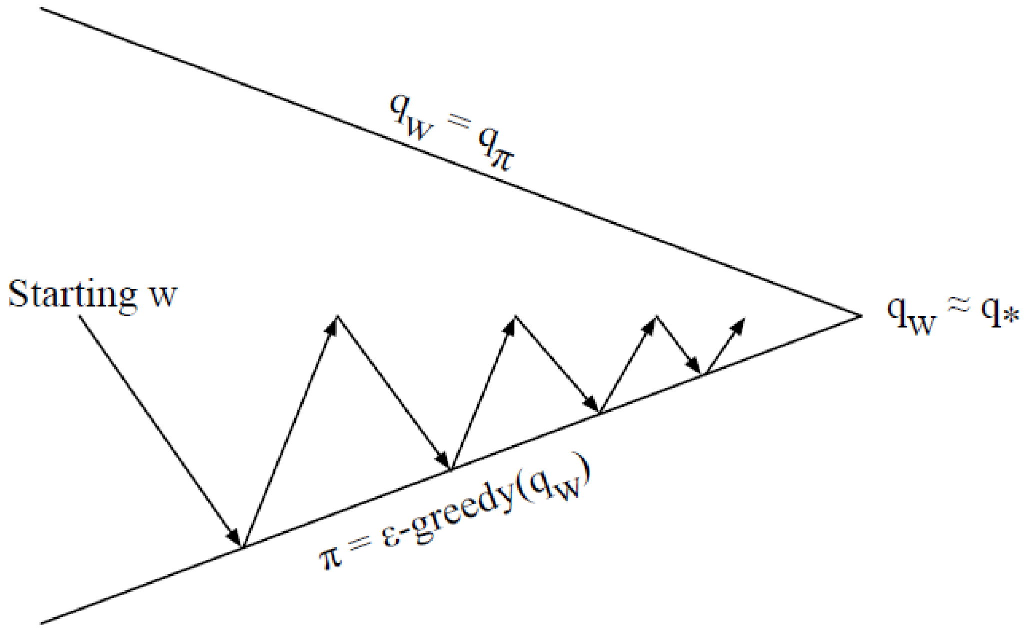
See Baird's counter-example (in the textbook) which shows the divergence of TD algorithm

Gradient TD algorithm resolved the divergence problem. See paper "Fast Gradient-Descent Methods for Temporal-Difference Learning with Linear Function Approximation" by Richard Sutton etc.

# Outline

- Introduction & Preliminaries
- RL Prediction with Function Approximation
- RL Control with Function Approximation
- Batch Method for RL Applications

# RL Control




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

Policy improvement $\epsilon$-greedy policy improvement

# Action-value Function Approximation

- Approximate the action-value function

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

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

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

- Use stochastic gradient descent to find a local minimum

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

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

# 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 fn by linear combination of features

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

- Stochastic gradient descent update

Update Target

$$\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)$$

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

# RL Control with Value Approximation

- For MC, the target is the return $G_t$

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

- For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

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

- For forward-view TD($\lambda$), target is the action-value $\lambda$-return

$$\Delta \mathbf{w} = \alpha(q_t^{\lambda} - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w})$$

- For backward-view TD($\lambda$), equivalent update is

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \\
E_t &= \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w}) \\
\Delta \mathbf{w} &= \alpha \delta_t E_t
\end{aligned}$$

# Convergence of Control Algorithms

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

(✓) = chatters around near-optimal value function

# Outline

- Introduction & Preliminaries
- RL Prediction with Function Approximation
- RL Control with Function Approximation
- Batch Method for RL Applications

# Motivation

- Gradient descent is simple and appealing
- But it is *not* sample efficient
- Batch methods seek to find the best fitting value function
- Given the agent's experience ("training data")

# Least Square Prediction

- Given value function approximation $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$
- And *experience* $\mathcal{D}$ consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$$

- Which parameters $\mathbf{w}$ give the *best fitting* value fn $\hat{v}(s, \mathbf{w})$?
- Least squares algorithms find parameter vector $\mathbf{w}$ minimising sum-squared error between $\hat{v}(s_t, \mathbf{w})$ and target values $v_t^\pi$,

$$\begin{aligned} LS(\mathbf{w}) &= \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2 \\ &= T\, \mathbb{E}_{\mathcal{D}} \left[ (v^\pi - \hat{v}(s, \mathbf{w}))^2 \right] \end{aligned}$$

# Experience Replay

- Given experience consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \ldots, \langle s_T, v_T^\pi \rangle\}$$

- Repeat:
  - Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

  - Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w})) \nabla_\mathbf{w} \hat{v}(s, \mathbf{w})$$

- Converges to least squares solution

$$\mathbf{w}^\pi = \underset{\mathbf{w}}{\operatorname{argmin}}\; LS(\mathbf{w})$$

# Experience Replay in Deep Q-Network (DQN)

DQN uses experience replay and fixed Q-targets

→Remove correlations between consecutive observations in experience data

→Better convergence

- Take action $a_t$ according to $\epsilon$-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions $(s, a, r, s')$ from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters $w^-$
- Optimise MSE between Q-network and Q-learning targets

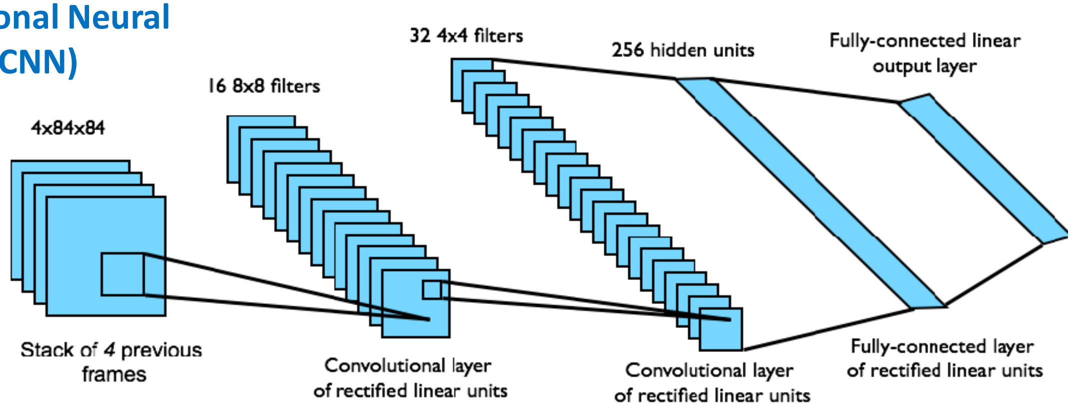
$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; w_i^-)}_{\text{Q-learning targets}} - \underbrace{Q(s, a; w_i)}_{\text{Q-network}}\right)^2\right]$$

- Using variant of stochastic gradient descent

# DQN in Atari Games

- 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)**




Network architecture and hyperparameters do not change across games

See paper “Human-level control through deep reinforcement learning” Nature, 2015