Lecture 12: Value Function Approximation

Ziyu Shao

School of Information Science and Technology
ShanghaiTech University

May 25 & 27, 2020

Outline

Outline

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve *large* problems, e.g.

- Backgammon: 10^{20} states
- Chess: 10^{47} states
- Computer Go: 10^{170} states
- Helicopter: continuous state space

How can we scale up the model-free methods for *prediction* and *control*?

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

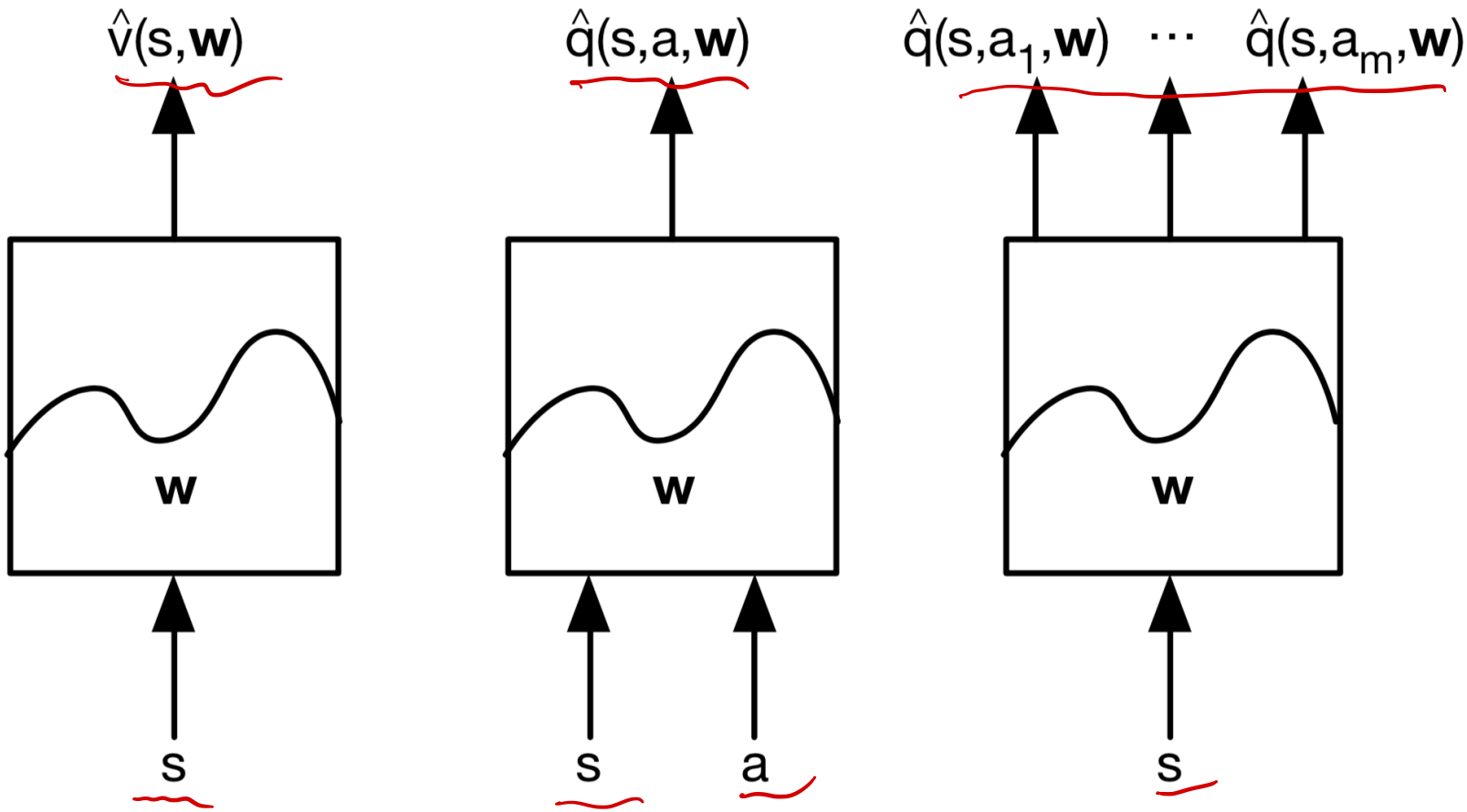
Scaling up RL with Function Approximation

- How to avoid explicitly learning or storing for every single state:
  - Dynamics or reward model
  - Value function, state-action function
  - Policy
- Solution for large MDPs:
  - Estimate with *function approximation*

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

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

Types of Value Function Approximation



Which Function Approximator?

There are many function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbor
- Fourier / wavelet bases
- ...

Which Function Approximator?

We consider differentiable function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbor
- Fourier / wavelet bases
- ...

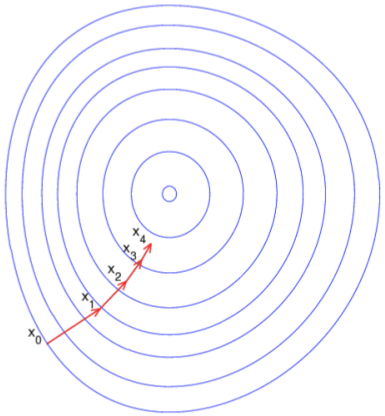
Furthermore, we require a training method that is suitable for non-stationary, non-iid data

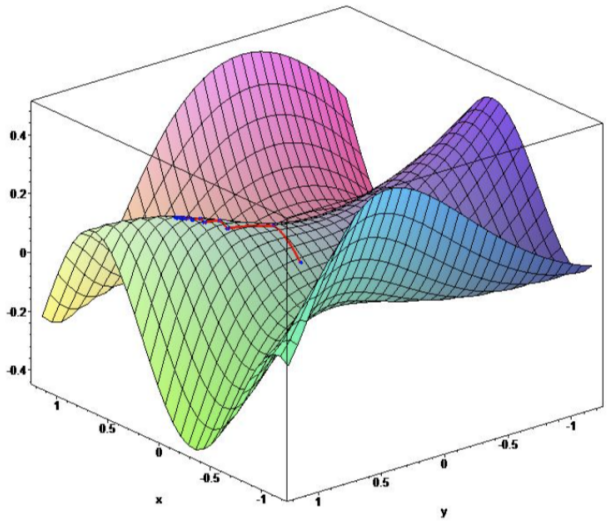
Outline

Gradient Descent

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

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





- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of negative gradient

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

where α is a step-size parameter

Value Function Approximation by SGD

- Goal: find parameter vector $\mathbf{w}$ minimizing 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

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

- Stochastic gradient descent *samples* the gradient

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

- Expected update is equal to full gradient update

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

Linear Value Function Approximation

- Represent value function by a linear combination of features

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

- Objective function is quadratic in parameters $\mathbf{w}$

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

- 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

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

- Thus we have $\hat{v}(s_k, \mathbf{w}) = w_k$

Value Function Approximation for Model-free Prediction

- In practice, no access to oracle of the true value $v_\pi(s)$ for any state s
- Recall model-free prediction
  - Goal is to evaluate v_π following a fixed policy π
  - A lookup table is maintained to store estimates v_π or q_π
  - Estimates are updated after each episode (MC method) or after each step (TD method)
- What we do: include the function approximation step in the loop

Incremental Prediction Algorithms

- We assumed true value function $v_\pi(s)$ given by supervisor

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

- But in RL there is no supervisor, only rewards. In practice, we substitute a target for $v_\pi(s)$
  - For MC, the target is the return G_t

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

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

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

  - For TD(λ), the target is the λ-return G_t^λ

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

Monte-Carlo Prediction with VFA

- Return G_t is an unbiased, noisy sample of true value $v_\pi(S_t)$
- Why unbiased? $\mathbb{E}[G_t] = v_\pi(S_t)$
- Can therefore apply supervised learning to "training data":

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

- 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 prediction converges, in both linear and non-linear value function approximation

TD Prediction with VFA

- The TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ is a biased sample of true value $v_\pi(S_t)$
- Why biased? It is drawn from our previous estimate, rather than the true value: $\mathbb{E}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})] \neq v_\pi(S_t)$
- Can still apply supervised learning to "training data":

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

- When using *linear TD(0)*, the stochastic gradient descend update is

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

- This is also called as semi-gradient, as we ignore the effect of changing the weight vector $\mathbf{w}$ on the target
- Linear TD(0) converges (close) to global optimum

TD(λ) with Value Function Approximation

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

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

- Forward view linear TD(λ)

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

- Backward view linear TD(λ)

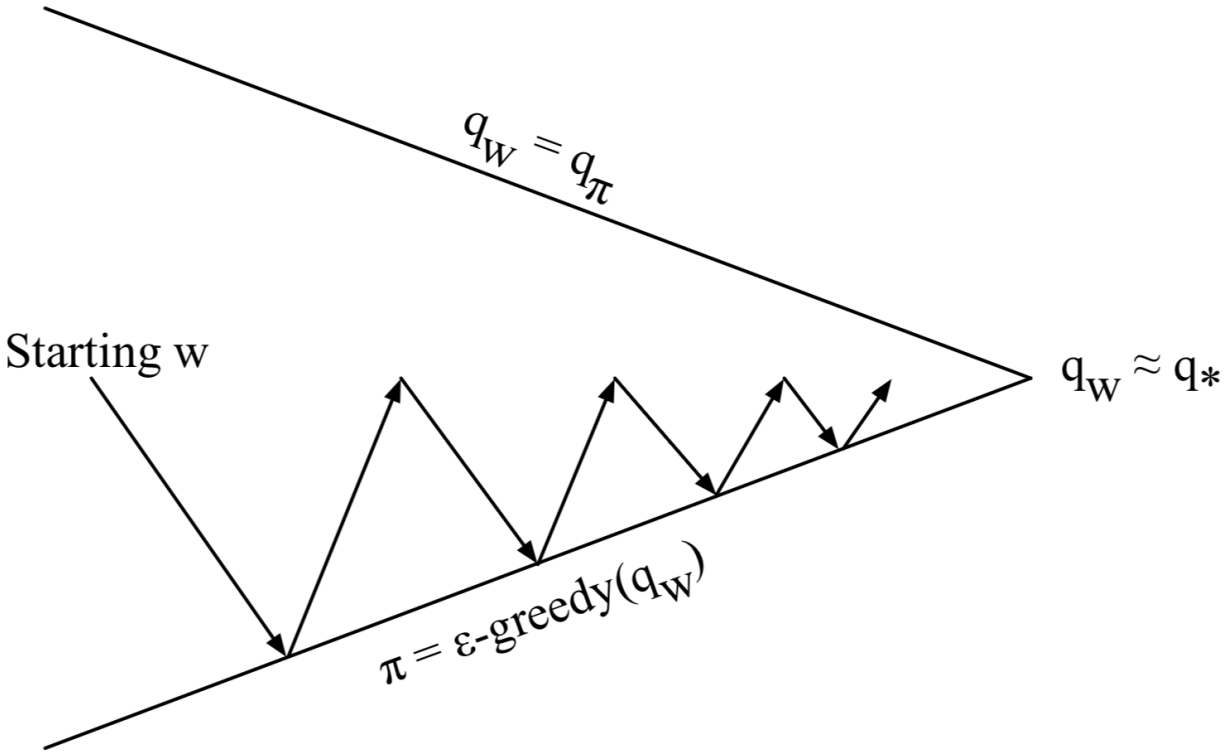
$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t\end{aligned}$$

- Forward view and backward view linear TD(λ) are equivalent

Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Control with Value Function Approximation



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

Policy improvement ϵ-greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function

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

- Minimize 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

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

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

Incremental Control Algorithms

Like prediction, we must substitute a *target* for $q_\pi(S, A)$

- For MC, the target is the return G_t

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

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

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

- For Q-learning, the target is the TD target $R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w})$

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

Incremental Control Algorithms

Like prediction, we must substitute a *target* for $q_\pi(S, A)$

- For forward-view TD(λ), target is the action-value λ-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(λ), equivalent update is

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

Example: Semi-gradient Sarsa for VFA Control

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

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Convergence of Control Methods with VFA

- TD with VFA doesn't follow the gradient of any objective function
- The updates involve doing an approximate Bellman backup followed by fitting the underlying value function
- That is why TD can diverge when off-policy or using non-linear function approximation
- Challenge for off-policy control: behavior policy and target policy are not identical, thus value function approximation can diverge

The Deadly Triad for the Danger of Instability and Divergence

Deadly Triad made up when we combine all of the following three elements:

- Function Approximation: a scalable way of generalizing from a state space much larger than the memory and computational resources
- Bootstrapping: update targets that include existing estimates (as in dynamic programming or TD methods) rather than relying exclusively on actual rewards and complete returns (as in MC methods)
- Off-policy training: training on a distribution of transitions other than that produced by the target policy

Convergence of Control Algorithms

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

(✓)= chatters around near-optimal value function

Outline

Batch Reinforcement Learning

- Incremental Methods or Online Learning
  - the agent gradually gathers experience in the environment
  - Gradient descent based algorithms are simple and appealing
  - But it is *not* sample efficient
- Batch methods or offline learning
  - Learn from the limited data without interacting further with the environment
  - seek to find the best fitting value function
  - Given the agent's experience ("training data")

On/Off Policy Learning with Experiences

- Off-policy learning
  - experience replay allows reusing samples from a different behavior policy
  - sample efficient since any experience can be used
- On-policy learning
  - usually introduce a bias when used with a ~~relay~~ replay buffer
  - as the trajectories are usually not obtained solely under the current policy
  - in a word, introduce a bias when using off-policy trajectories

Least Squares 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

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

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

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

Stochastic Gradient Descent with 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:
  1. Sample state, value from experience

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

  2. 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

$$\mathbf{w}_\pi = \arg\min_{\mathbf{w}} LS(\mathbf{w})$$

Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using *linear* value function approximation $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w}$
- We can solve the least squares solution directly

Linear Least Squares Prediction (2)

- At minimum of $LS(\mathbf{w})$, the expected update must be zero

$$\mathbb{E}_{\mathcal{D}}[\Delta \mathbf{w}] = 0$$

$$\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)(v_t^{\pi} - \mathbf{x}(s_t)^{\top}\mathbf{w}) = 0$$

$$\sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi} = \sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\mathbf{w}$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi}$$

- For N features, direct solution time is $O(N^3)$
- Incremental solution time is $O(N^2)$ using Shermann-Morrison

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

Linear Least Squares Prediction Algorithms

- We do not know true values v_t^π
- In practice, our "training data" must use noisy or biased samples of v_t^π

LSMC	Least Squares Monte-Carlo uses return
	$v_t^\pi \approx G_t$
LSTD	Least Squares Temporal-Difference uses TD target
	$v_t^\pi \approx R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$
LSTD(λ)	Least Squares TD(λ) uses λ-return
	$v_t^\pi \approx G_t^\lambda$

- In each case solve directly for fixed point of MC / TD / TD(λ)

Linear Least Squares Prediction Algorithms (2)

LSMC

$$0 = \sum_{t=1}^{T} \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)\mathbf{x}(S_t)^\top\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t)G_t$$

LSTD

$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)(\mathbf{x}(S_t) - \gamma\mathbf{x}(S_{t+1}))^\top\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t)R_{t+1}$$

Linear Least Squares Prediction Algorithms (2)

LSTD(λ)

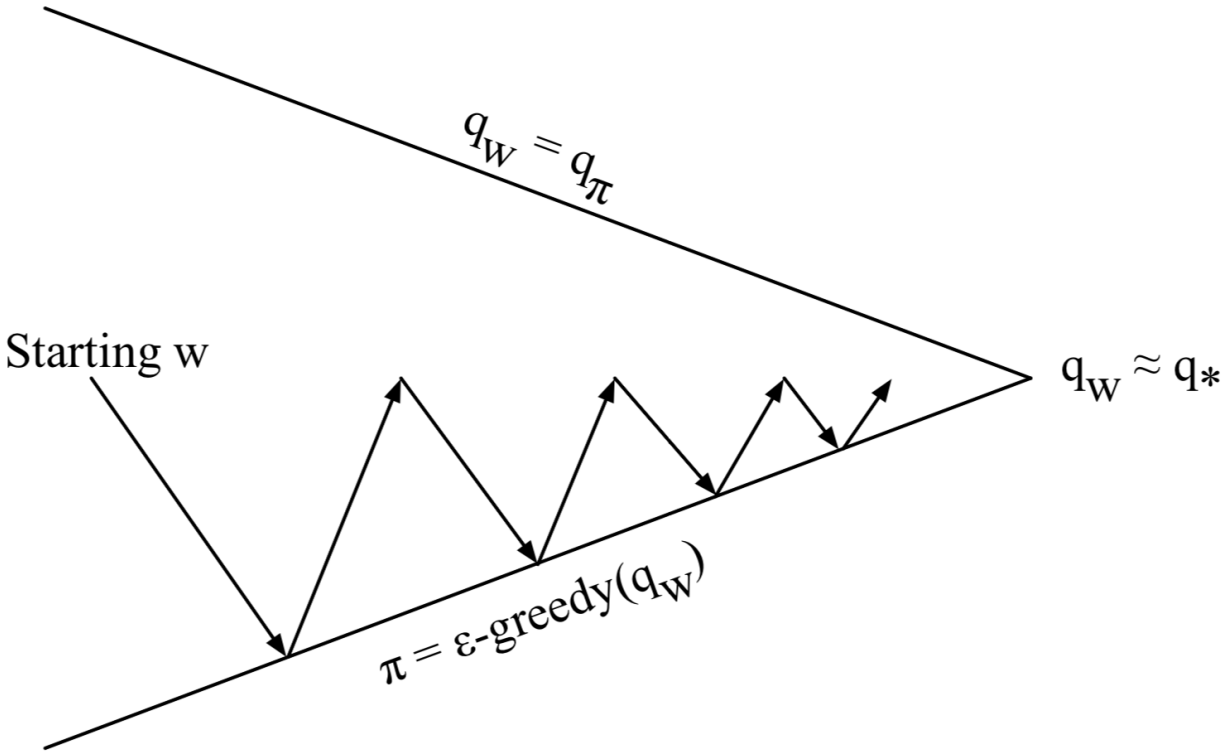
$$0 = \sum_{t=1}^{T} \alpha \delta_t E_t$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} E_t (\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top \right)^{-1} \sum_{t=1}^{T} E_t R_{t+1}$$

Convergence of Linear Least Squares Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

Least Squares Policy Iteration



Policy evaluation Policy evaluation by least squares Q-learning
Policy improvement Greedy policy improvement

Least Squares Action-Value Function Approximation

- Approximate action-value function $q_\pi(s, a)$
- using linear combination of features $\mathbf{x}(s, a)$

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

- Minimize least squares error between $\hat{q}(s, a, \mathbf{w})$ and $q_\pi(s, a)$
- from experience generated using policy π
- consisting of $\langle (state, action), value \rangle$ pairs

$$\mathcal{D} = \{\langle (s_1, a_1), v_1^\pi \rangle, \langle (s_2, a_2), v_2^\pi \rangle, \ldots, \langle (s_T, a_T), v_T^\pi \rangle\}$$

Least Squares Control

- For policy evaluation, we want to efficiently use all experience
- For control, we also want to improve the policy
- This experience is generated from many policies (previously old policies)
- So to evaluate $q_\pi(S, A)$ we must learn off-policy
- We use the same idea as Q-learning:
  - Use experience generated by old policy $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$
  - Consider alternative successor action $A' = \pi_{new}(S_{t+1})$
  - Update $\hat{q}(S_t, A_t, \mathbf{w})$ towards value of alternative action $R_{t+1} + \gamma \hat{q}(S_{t+1}, A', \mathbf{w})$

Least Squares Q-Learning

- Consider the following linear Q-learning update

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

- LSTDQ algorithm: solve for total update = zero

$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi(S_{t+1}), \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\mathbf{x}(S_t, A_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t, A_t)(\mathbf{x}(S_t, A_t) - \gamma \mathbf{x}(S_{t+1}, \pi(S_{t+1})))^\top \right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t, A_t) R_{t+1}$$

Least Squares Policy Iteration Algorithm

- The following pseudocode uses LSTDQ for policy evaluation
- It repeatedly re-evaluates experience $\mathcal{D}$ with different policies

function LSPI-TD$(\mathcal{D}, \pi_0)$
  $\pi' \leftarrow \pi_0$
  repeat
    $\pi \leftarrow \pi'$
    $Q \leftarrow$ **LSTDQ**$(\pi, \mathcal{D})$
    for all $s \in \mathcal{S}$ **do**
      $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$
    end for
  until $(\pi \approx \pi')$
  return π
end function

Convergence of Control Algorithms

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

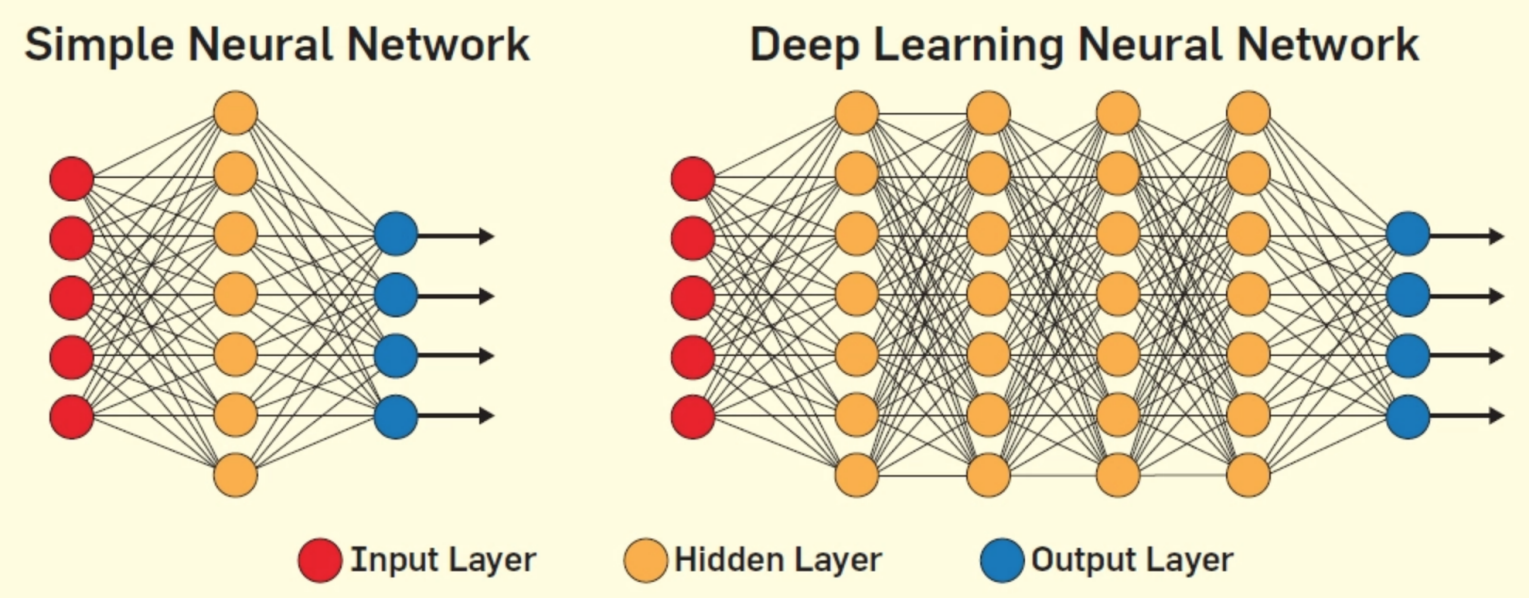
(✓) = chatters around near-optimal value function

Outline

Linear vs Nonlinear Value Function Approximation

- Linear VFA often works well given the right set of features
- But it requires carefully hand designing the feature set
- Alternative is to use a much richer function approximator that is able to directly learn from states without requiring manual designing of features
- Nonlinear function approximator: Deep neural networks

Deep Neural Networks



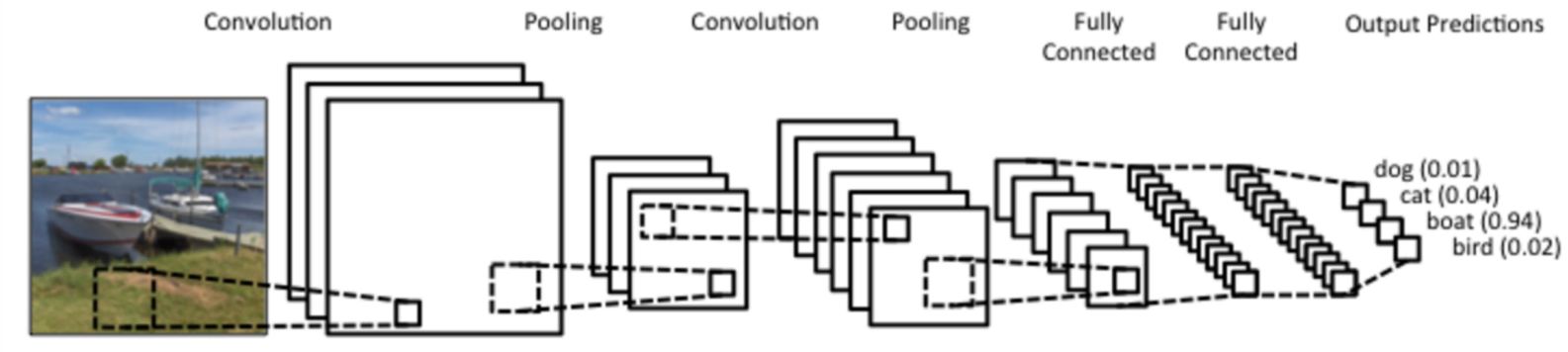
- Multiple layers of linear functions & non-linear operators between layers

$$f(\mathbf{x};\theta) = \mathbf{W}_{L+1}^T \sigma(\mathbf{W}_L^T \sigma(\ldots \sigma(\mathbf{W}_1^T \mathbf{x} + \mathbf{b}_1) + \ldots + \mathbf{b}_{L-1}) + \mathbf{b}_L) + \mathbf{b}_{L+1}$$

- The chain rule to backpropagate the gradient to update the weights using the loss function $L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - f(\mathbf{x};\theta) \right)^2$

Convolutional Neural Networks

RNN/GNN

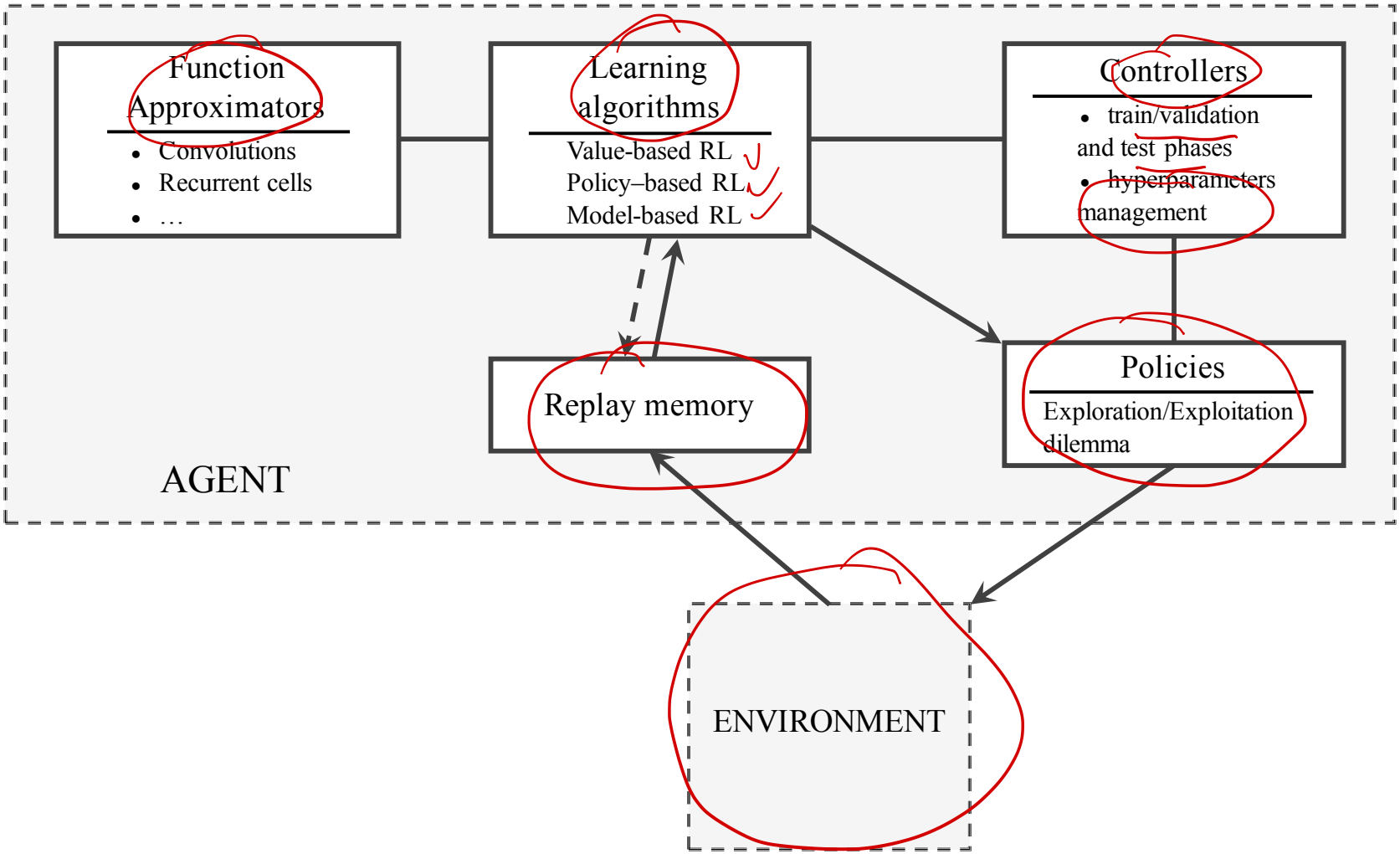


- Convolution encodes the local information in 2D feature map
- Layers of convolution, reLU, batch normalization, etc.
- A detailed introduction on CNNs: Stanford.
  `http://cs231n.github.io/convolutional-networks/`

Deep Reinforcement Learning

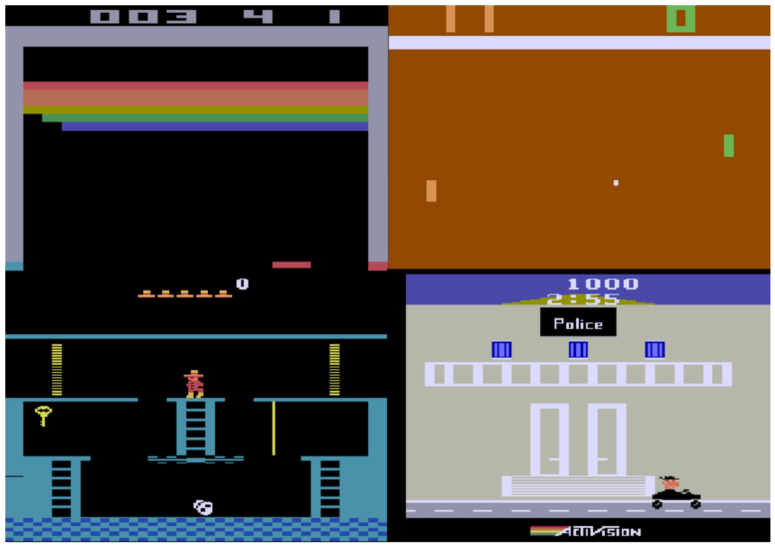
- Frontier in machine learning and artificial intelligence
- Deep neural networks is used to represent
  - Value function
  - Policy function (policy gradient methods)
  - Model
- Optimize loss function by stochastic gradient descent (SGD)
- Challenges
  - Efficiency: too many parameters to optimize
  - Convergence/stability of training: nonlinear function

General Schema of Deep Reinforcement Learning



Deep Q-Networks (DQN)

- DeepMind's **Nature** paper: Mnih, Volodymyr; et al. (2015). **Human-level control through deep reinforcement learning**
- DQN represents the action value function with neural network approximator
- DQN reaches a professional human gaming level across many Atari games using the same network and hyperparameters



4 Atari Games: Breakout, Pong, Montezuma's Revenge, Private Eye

Recall: Action-Value Function Approximation

- Approximate the action-value function

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

- Minimize the MSE (mean-square error) between approximate action-value and true action-value (assume oracle)

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

- Stochastic gradient descend to find a local minimum

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

Recall: Incremental Control Algorithm

Same to the prediction, there is no oracle for the true value $q_\pi(S, A)$, so we substitute a target

- 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 Sarsa, the target is the TD target $R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})$

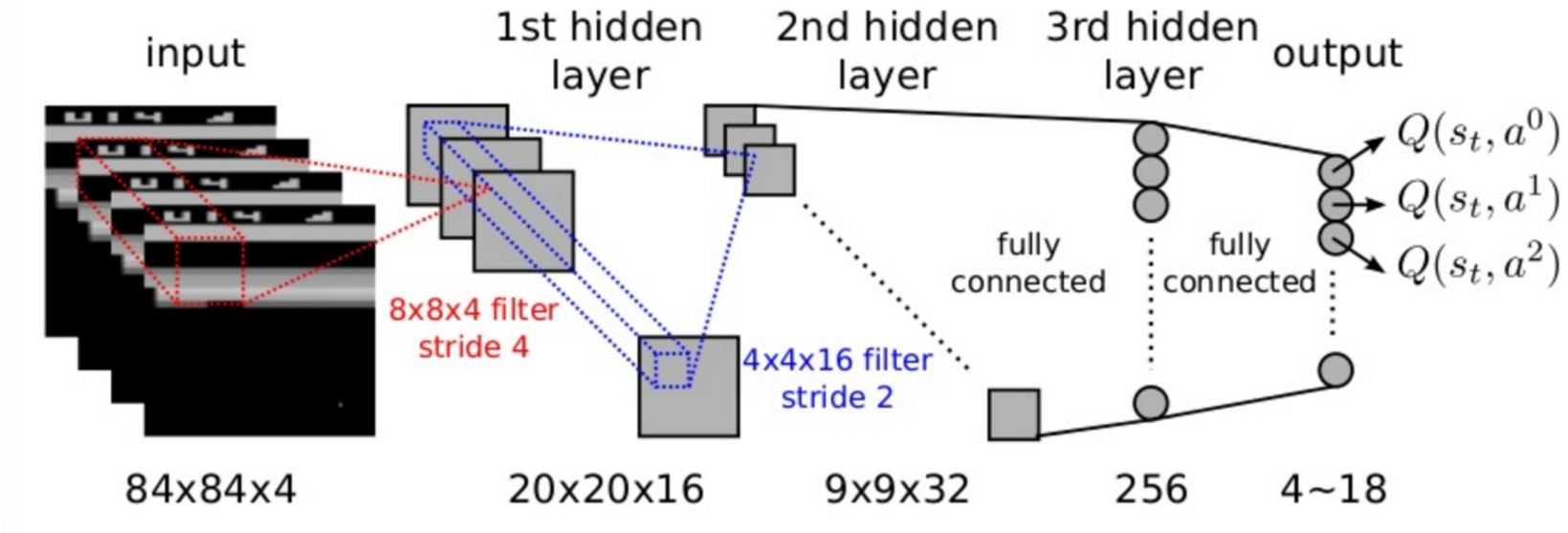
$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For Q-learning, the target is the TD target $R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w})$

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

DQN for Playing Atari Games

- End-to-end learning of values $Q(s, a)$ from input pixel frame
- Input state s is a stack of raw pixels from last 4 frames
- Output of $Q(s, a)$ is 18 joystick/button positions
- Reward is the change in score for that step
- Network architecture and hyperparameters fixed across all games

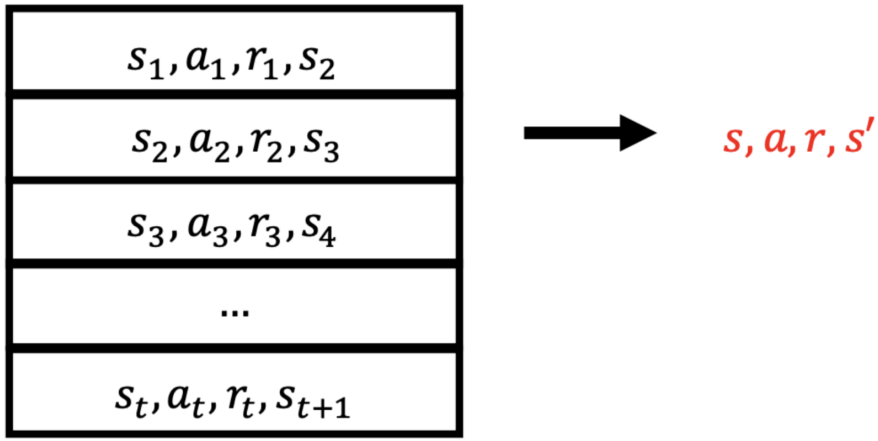


Q-Learning with Value Function Approximation

- Two of the issues causing problems:
  1. Correlations between samples
  2. Non-stationary targets
- Deep Q-learning (DQN) addresses both of these challenges by
  1. Experience replay
  2. Fixed Q-targets

DQNs: Experience Replay

- To reduce the correlations among samples, store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$



- To perform experience replay, repeat the following
  - sample an experience tuple from the dataset: $(s, a, r, s') \sim \mathcal{D}$
  - compute the target value for the sampled tuple: $r + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w})$
  - use stochastic gradient descent to update the network weights

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

DQNs: Fixed Targets

- To help improve stability, fix the target weights used in the target calculation for multiple updates
- Let a different set of parameter $\mathbf{w}^-$ be the set of weights used in the target, and $\mathbf{w}$ be the weights that are being updated
- To perform experience replay with fixed target, repeat the following
  - sample an experience tuple from the dataset: $(s, a, r, s') \sim \mathcal{D}$
  - compute the target value for the sampled tuple: $r + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}^-)$
  - use stochastic gradient descent to update the network weights

$$\Delta \mathbf{w} = \alpha \left(r + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w})$$

Why fixed target

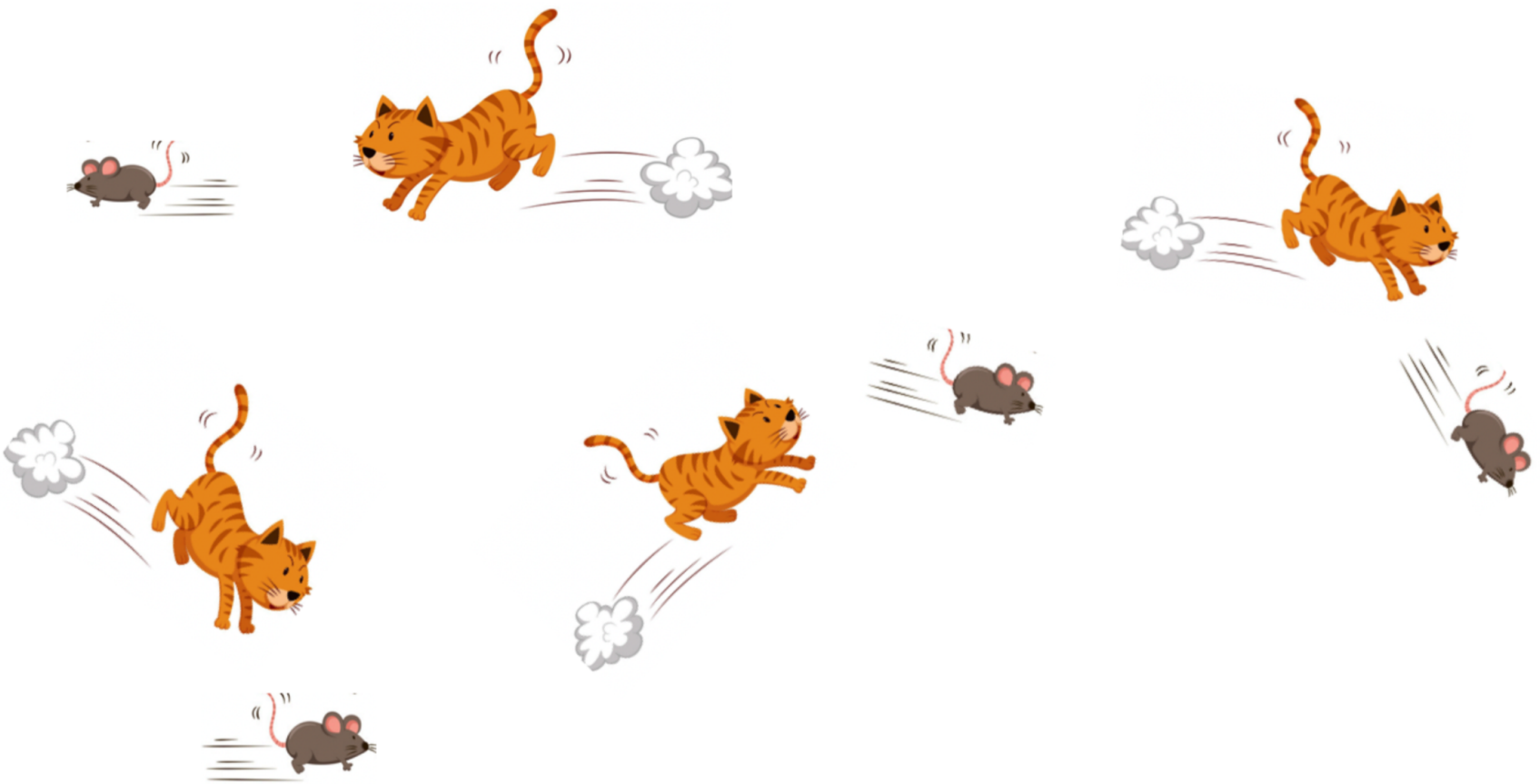
- In the original update, both Q estimation and Q target shifts at each time step
- Imagine a cat (Q estimation) is chasing after a mice (Q target)
- The cat must reduce the distance to the mice

Q target

Q estimation

Why fixed target

- Both the cat and mice are moving,

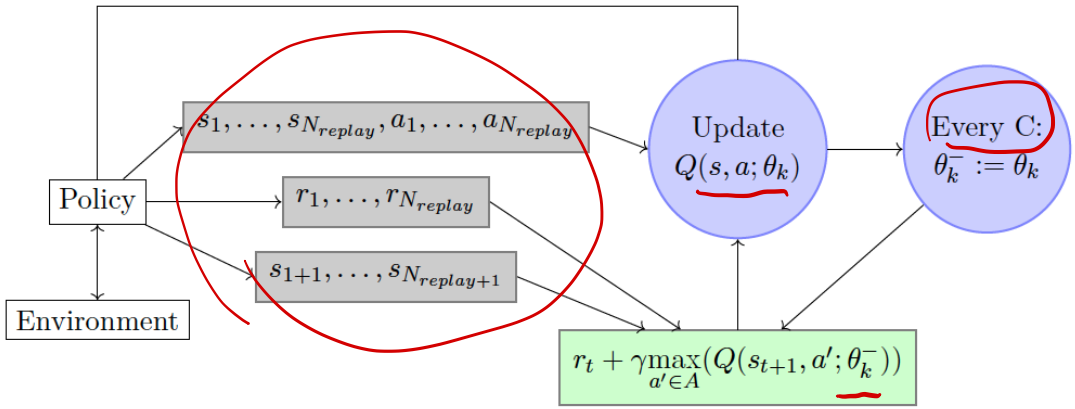


Why fixed target

- This could lead to a strange path of chasing (an oscillated training history)

- Solution: fix the target for a period of time during the training

Sketch of the DQN Algorithm with Parameter θ



$\theta = w$

- $Q(s, a; \theta_k)$ is initialized to random values (close to 0) everywhere in its domain and the replay memory is initially empty
- the target Q-network parameters θ_k^- are only updated every C iterations with the Q-network parameters θ_k and are held fixed between updates
- the update uses a mini-batch (e.g., 32 elements) of tuples $< s, a >$ taken randomly in the replay memory along with the corresponding mini-batch of target values for the tuples

Important Heuristics of DQN

- Target Q-network
- Replay memory
- Clipping the rewards to $[-1, 1]$: limit the scale of the error derivatives
- Multiple deep learning specific techniques including preprocessing
- CNN are used for the first layer of the neural network function approximator
- Stochastic gradient descent algorithm: RMSprop
- ϵ-greedy strategy to facilitate exploration

Abalation Study on DQNs

- Game score under difference conditions

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

Demo of DQNs

- Demo of deep q-learning for Breakout:
  `https://www.youtube.com/watch?v=V1eYniJ0Rnk`
- Demo of Flappy Bird by DQN:
  `https://www.youtube.com/watch?v=xM62SpKAZHU`

Summary of DQNs

- DQN uses experience replay and fixed Q-targets
- 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 $\mathbf{w}^-$
- Optimizes MSE between Q-network and Q-learning targets using stochastic gradient descent

Improving DQN

- Success in Atari has led to a huge excitement in using deep neural networks to do value function approximation in RL
- Many follow-up works on improving DQNs
  - **Double DQN**: Deep Reinforcement Learning with Double Q-Learning. Van Hasselt et al, AAAI 2016
  - **Dueling DQN**: Dueling Network Architectures for Deep Reinforcement Learning. Wang et al, best paper ICML 2016
  - **Prioritized Replay**: Prioritized Experience Replay. Schaul et al, ICLR 2016

Improving DQN: Double DQN

- Handles the problem of the overestimation of Q-values
- Idea: use the two networks to decouple the action selection from action evaluation (the target Q value generation)
- Vanilla DQN:

$$\Delta \mathbf{w} = \alpha \left(r + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w})$$

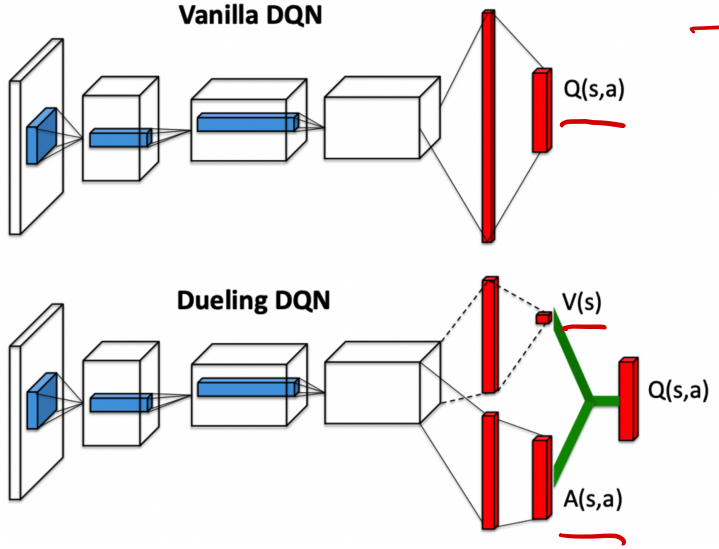
- Double DQN:

$$\Delta \mathbf{w} = \alpha \left(r + \gamma \hat{Q}(s', \max_{a'} \hat{Q}(s', a', \mathbf{w}), \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w})$$

- Use current weights of network $\mathbf{w}$ for action selection and target network weights $\mathbf{w}^-$ for action evaluation

Improving DQN: Dueling DQN

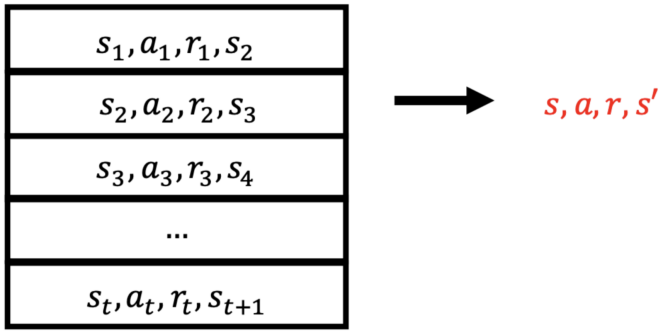
- One branch estimates $V(s)$, other branch estimates the advantage for each action $A(s, a)$. Then $Q(s, a) = A(s, a) + V(s)$



- By decoupling the estimation, intuitively the DuelingDQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state

Improving DQN: Prioritized Experience Replay

- Transition $(s_t, a_t, r_{t+1}, s_{t+1})$ is stored in and sampled from the replay memory $\mathcal{D}$



- Priority is on the experience where there is a big difference between our prediction and the TD target, since it means that we have a lot to learn about it.
- Define a priority score for each tuple i

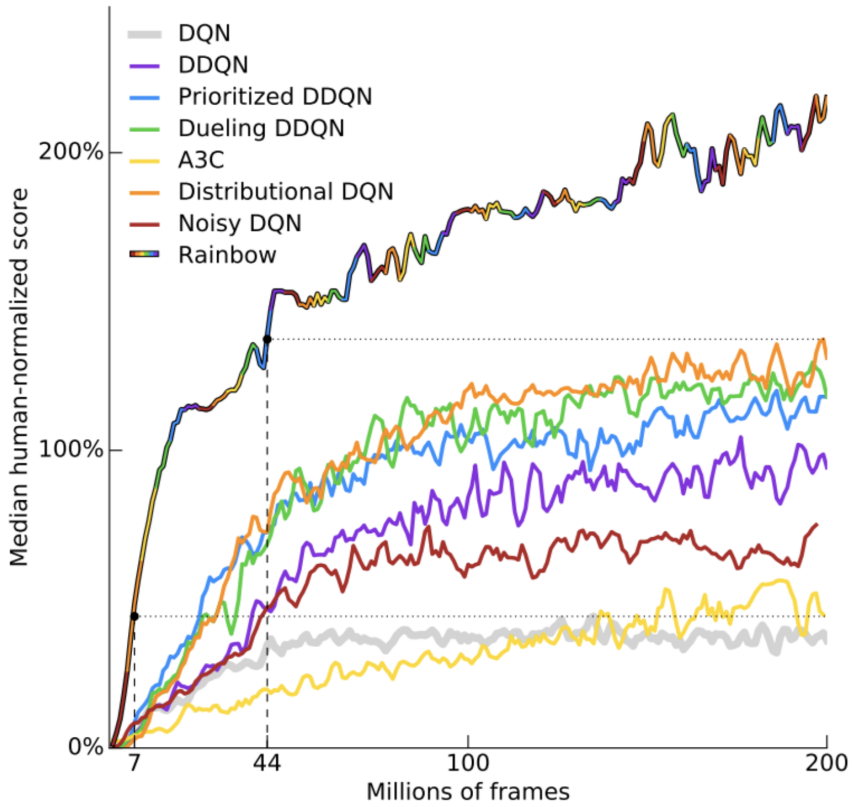
$$p_i = |r + \gamma \max_{a'} Q(s_{i+1}, a', \mathbf{w}^-) - Q(s_i, a_i, \mathbf{w})|$$

Rainbow DQN


Multistep DQN
DQN
Noisy Net
Dueling DQN
Double DQN
Prioritized Replay
Categorical DQN

Improving over the DQN

- Rainbow: Combining Improvements in Deep Reinforcement Learning. Matteo Hessel et al. AAAI 2018. `https://arxiv.org/pdf/1710.02298.pdf`
- It examines six extensions to the DQN algorithm and empirically studies their combination



Limitation of DQN

- Not well-suited to deal with large and/or continuous action spaces: just discrete action space
- Cannot explicitly learn stochastic policies: just deterministic policies
- To address such limitations we need policy based approaches

Outline

Main References

- Reinforcement Learning: An Introduction (second edition), R. Sutton & A. Barto, 2018.
- RL course slides from Richard Sutton, University of Alberta.
- RL course slides from David Silver, University College London.