Lecture 9: Value Function Approximation

Ziyu Shao

School of Information Science and Technology
ShanghaiTech University

April 25, 2025

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*

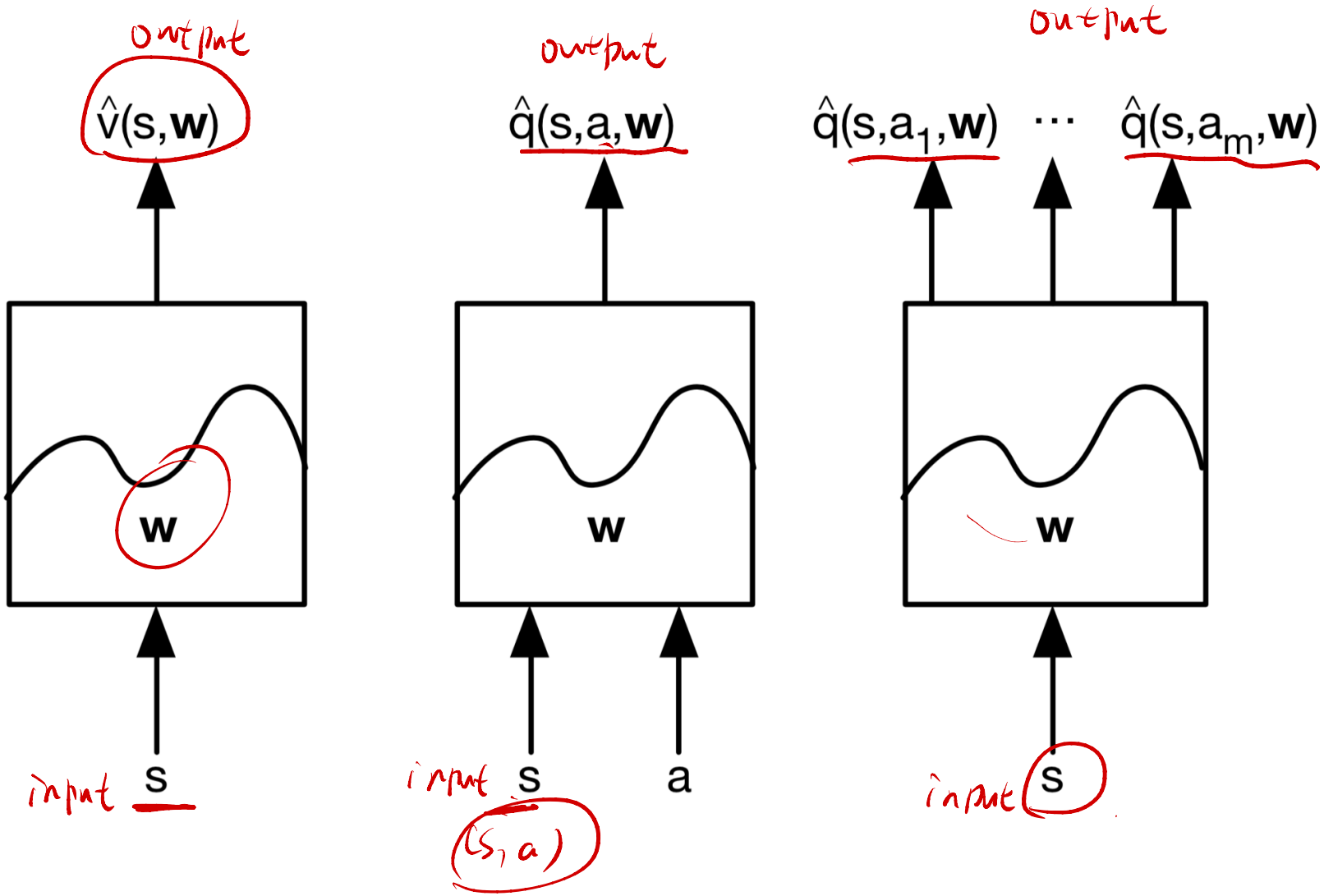
$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$
$$\hat{\pi}(s, a, \mathbf{w}) \approx \pi(a|s)$$

$V^*(s)$

$q^*(s,a)$

  - *Generalization* 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
- ...

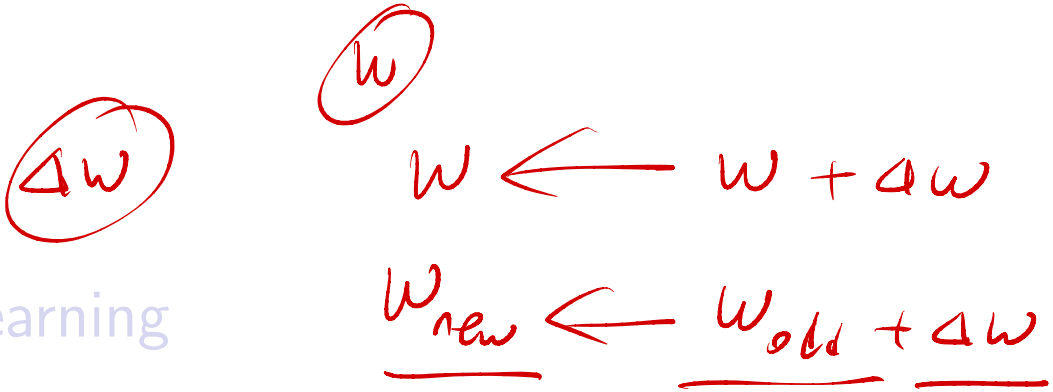
Which Function Approximator?

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

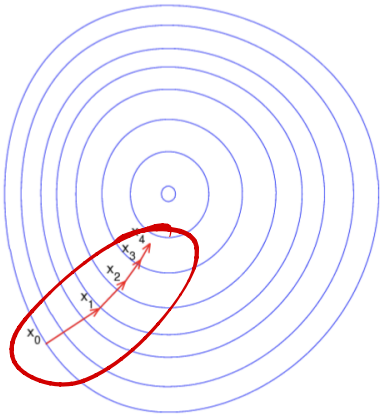
Outline

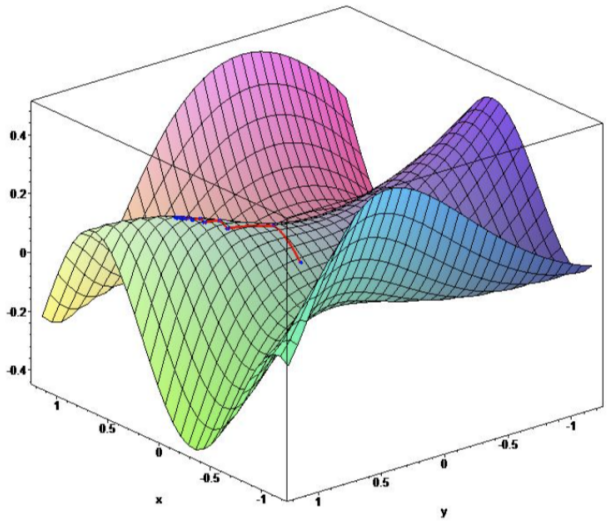


Gradient Descent

- 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 (learning rate)

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* × *prediction error* × *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

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

To error δ.

- 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":

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

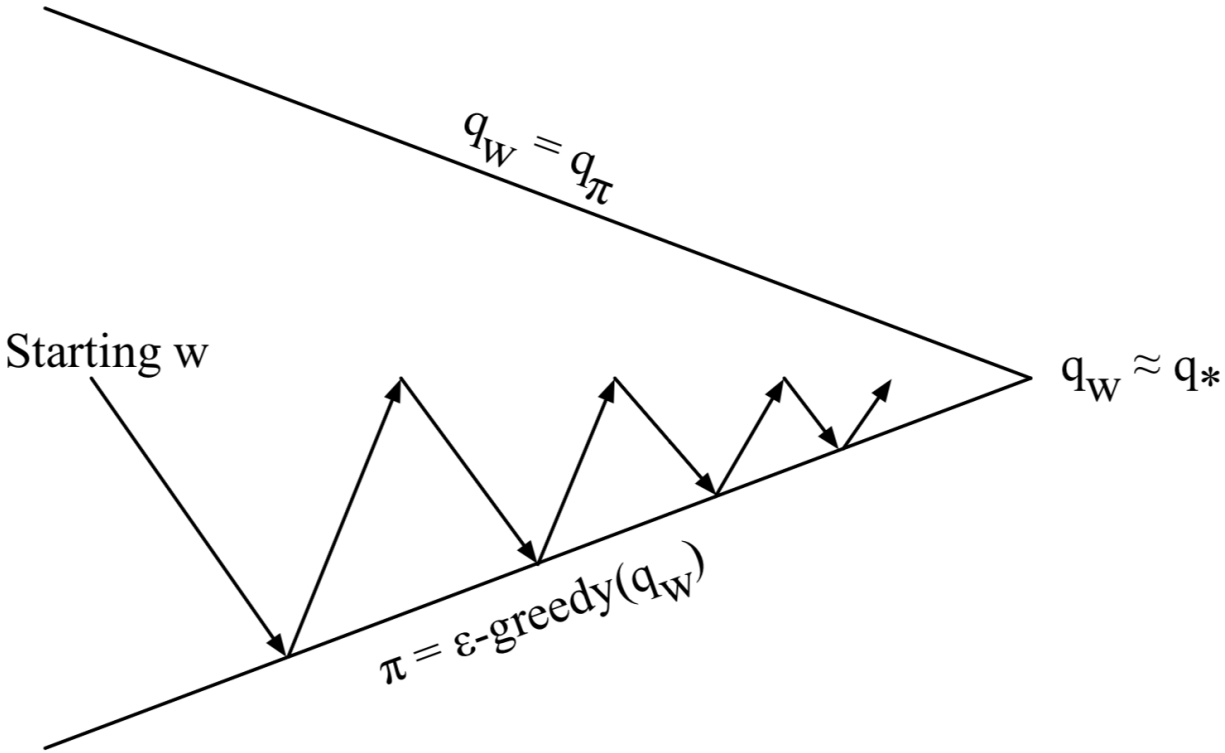
- Forward view linear TD(λ)

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

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 $R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})$

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

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

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

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	✓	✓	✗

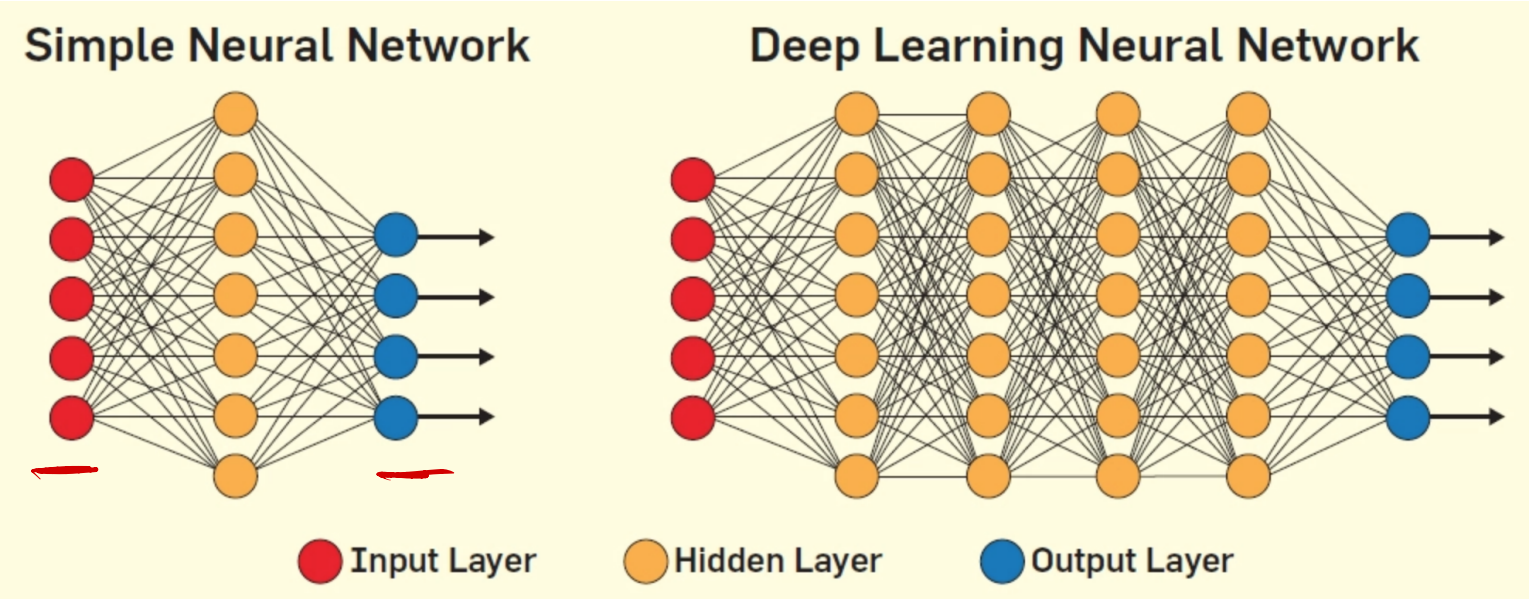
(✓)= chatters around near-optimal value function

Outline

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

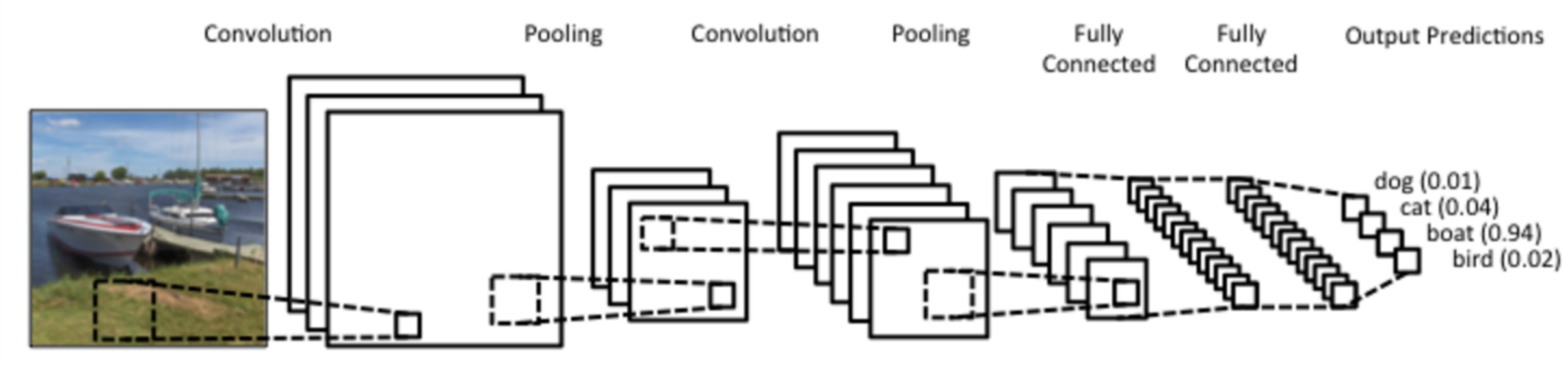


- Multiple layers of linear functions & non-linear operators between layers

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

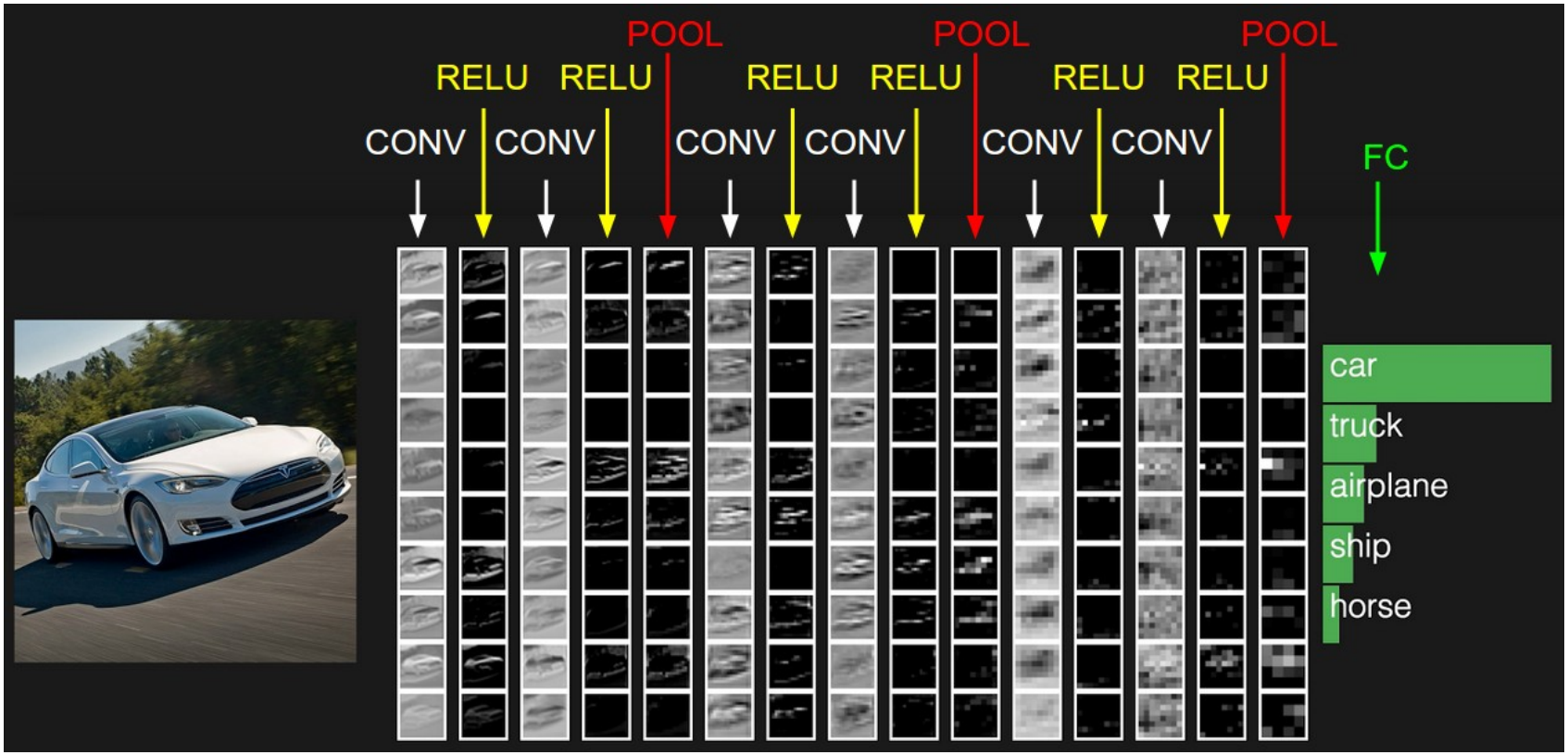
- The chain rule to backpropagate the gradient to update the weights using the loss function $L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - f(\mathbf{x};\theta) \right)^2$

Convolutional Neural Networks



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

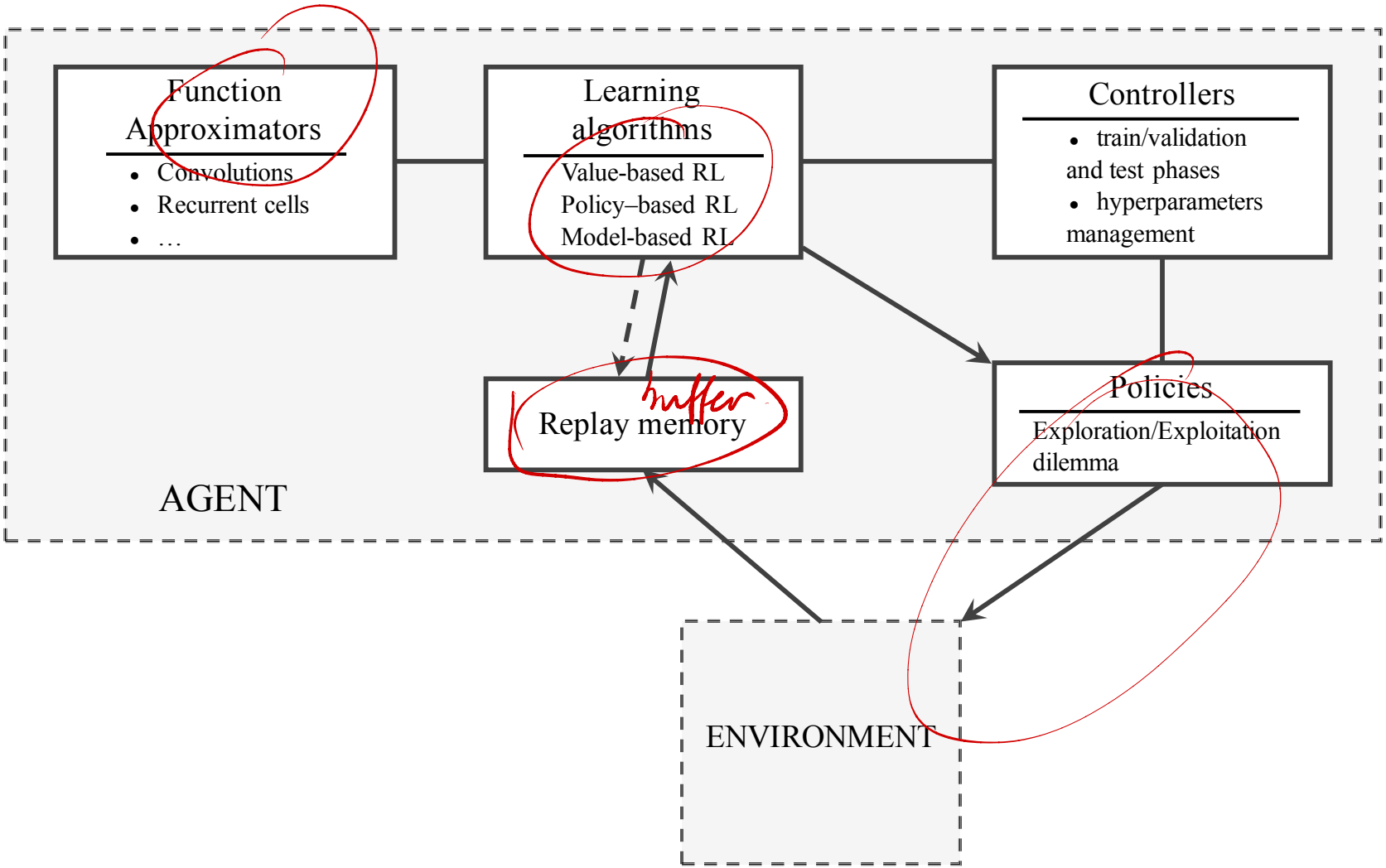
Example of Convolutional Neural Networks



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

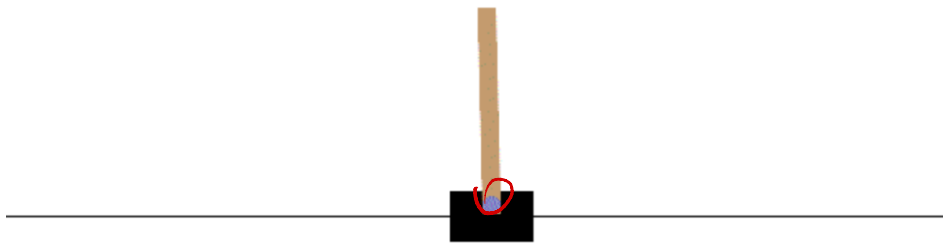
General Schema of Deep Reinforcement Learning



Outline

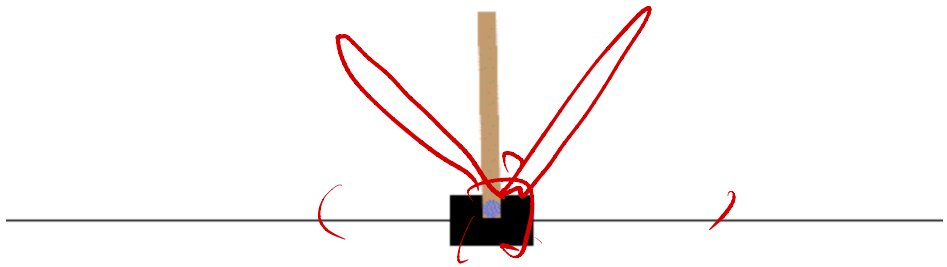
Example I: CartPole

- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track.
- The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.
- The goal is to keep the pole upright for as long as possible



Example I: CartPole

- Action: Left(Push cart to the left) & Right (Push cart to the right)
- State: (Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity)
- Reward: a reward of $+1$ is given for every step taken, including the termination step; the default reward threshold is 200.

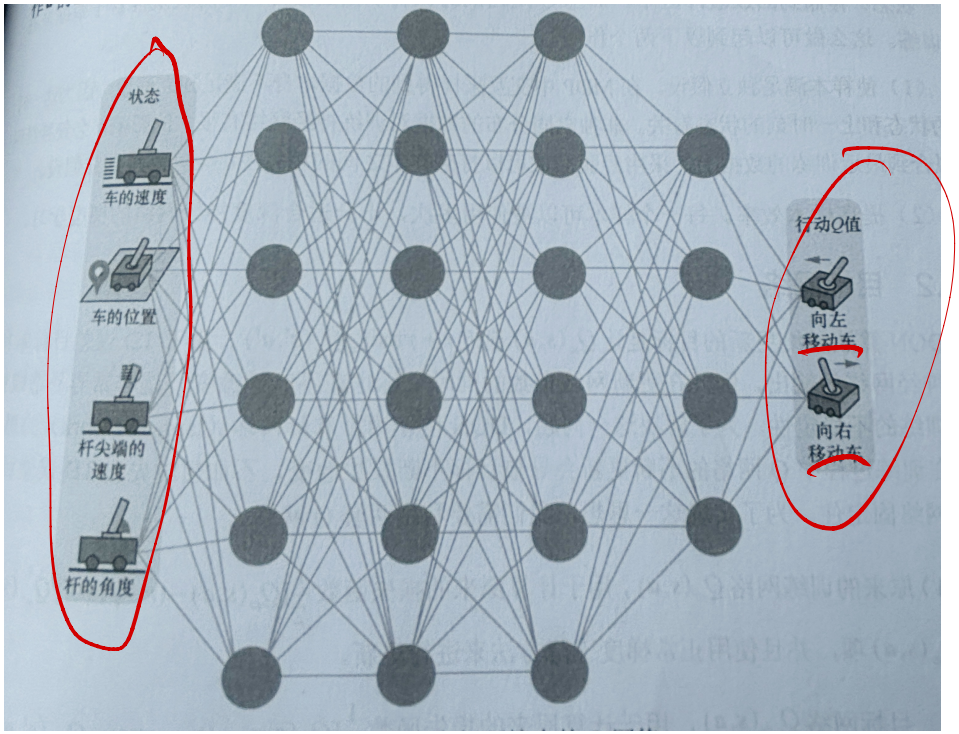


Example I: CartPole

- Cart Position($[-4.8, 4.8]$), Cart Velocity($[-\infty, \infty]$), Pole Angle($\pm 24^\circ$), Pole Angular Velocity($[-\infty, \infty]$)
- Episode end if any one of the following occurs:
  - Pole Angle is NOT belong to $[-12^\circ, 12^\circ]$
  - Cart Position is NOT belong to $[-4.8, 4.8]$
  - Episode length is greater than 200

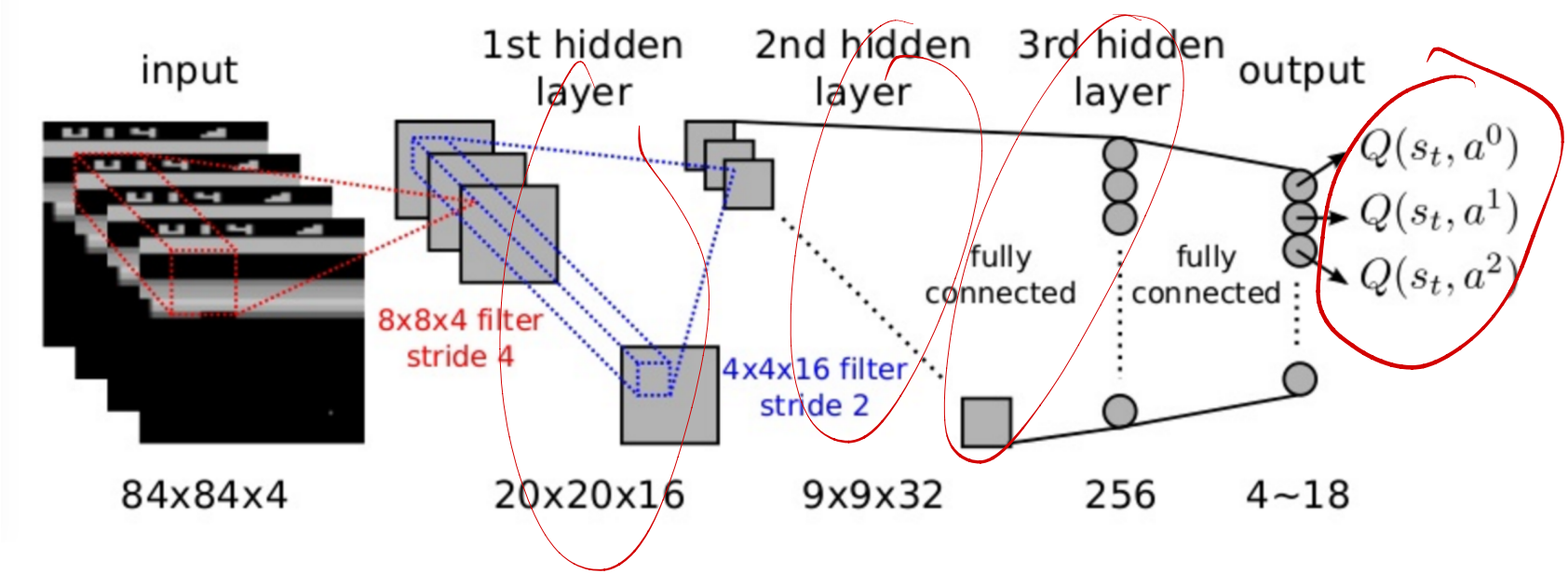
Example I: CartPole

- Corresponding deep Q-network(DQN) represents the optimal action value function with neural network approximator
- Approximating $Q^*(s, a)$ with $Q(s, a; w)$
- Input: state s
- Output: $\{Q(s, a; w), a \in \mathcal{A}\}$

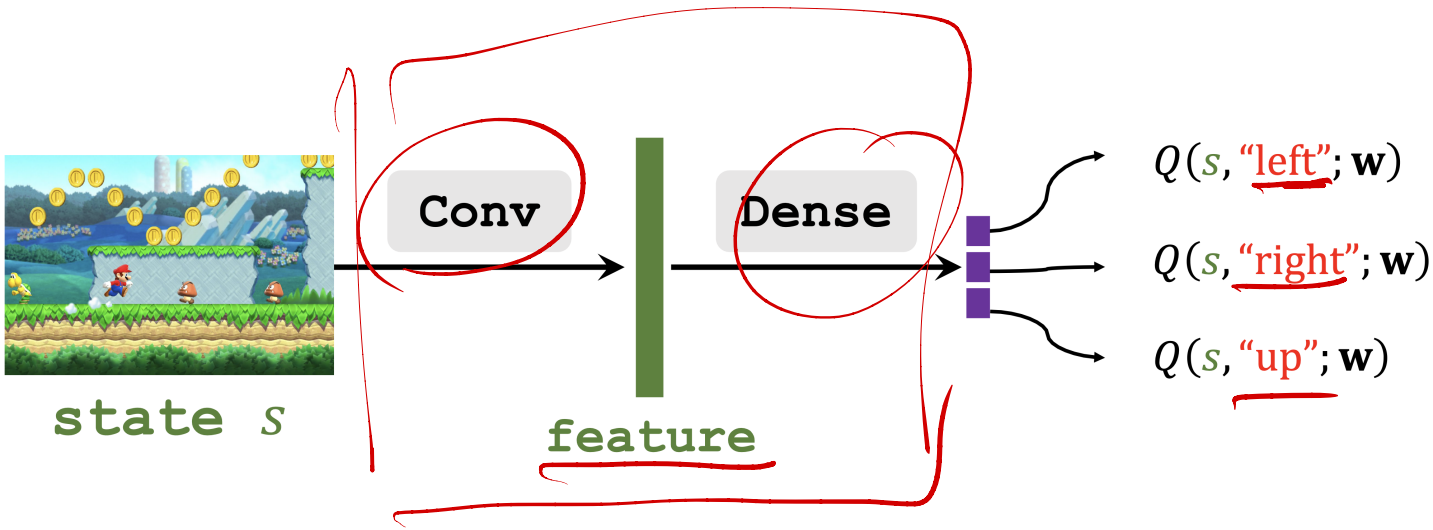


Example II: Atari Games

- Action: 18 joystick/button positions
- Input state s is a stack of raw pixels from last 4 frames(why?)
- Output of $Q(s, a, w)$ are end-to-end learning of values $Q(s, a)$ from input pixel frame



Example III: Super Mario



Recall: Classical Q-Learning

given current state s_t and Q functions $\{Q(s_t, a), a \in \mathcal{A}\}$

1. Use ϵ-greedy policy as the behavior policy

$$a_t = \begin{cases} \arg\max_a Q(s_t, a) & \text{with probability } 1 - \epsilon \\ \text{uniform sampling an action } a \in \mathcal{A} & \text{with probability } \epsilon \end{cases}$$

2. Performing action a_t and obtain reward r_{t+1} and new state s_{t+1}.
3. Collect the training data: quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$
4. Compute the TD target and TD error

$$y_t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')$$

$$\delta_t = y_t - Q(s_t, a_t)$$

5. Update the value of $Q(s_t, a_t)$ as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \delta_t$$

6. $t \leftarrow t + 1$ and loop until termination.

Optimal Action-Value Function Approximation

- Classical Q-learning: $Q(s, a) \to q^*(s, a)$
- Approximate the optimal action-value function

$$Q(S, A; \mathbf{w}) \approx q^*(S, A)$$

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

$$J(\mathbf{w}) = \mathbb{E}[(q^*(S, A) - Q(S, A; \mathbf{w}))^2]$$

- Stochastic gradient descend to find a local minimum

$$\Delta \mathbf{w} = \alpha(q^*(S, A) - Q(S, A; \mathbf{w}))\nabla_{\mathbf{w}} Q(S, A; \mathbf{w})$$

Incremental Control Algorithm

There is no oracle for the true value $q^*(S, A)$, so we substitute a target

- For deep Q-learning, the target is the TD target
  $y_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \mathbf{w})$
- TD error is $\delta_t = y_t - Q(s_t, a_t; \mathbf{w})$
- Thus

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \mathbf{w}) - Q(s_t, a_t; \mathbf{w}))\nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}) \\ &= \alpha(y_t - Q(s_t, a_t; \mathbf{w}))\nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}) \\ &= \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w})\end{aligned}$$

- Update $\mathbf{w}$ as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$$

Deep Q-Learning

given current state s_t, neural network parameters $\mathbf{w}$ and Q functions $\{Q(s_t, a; \mathbf{w}), a \in \mathcal{A}\}$

1. Use ϵ-greedy policy as the behavior policy

$$a_t = \begin{cases} \arg\max_a Q(s_t, a; \mathbf{w}) & \text{with probability } 1 - \epsilon \\ \text{uniform sampling an action } a \in \mathcal{A} & \text{with probability } \epsilon \end{cases}$$

2. Performing action a_t and obtain reward r_{t+1} and new state s_{t+1}.
3. Collect the training data: quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$
4. Compute the TD target and TD error

$$y_t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w}), \quad \delta_t = y_t - Q(s_t, a_t; \mathbf{w})$$

5. Performing Back-Propagation, obtain gradient value $\nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w})$ and update the parameters as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w})$$

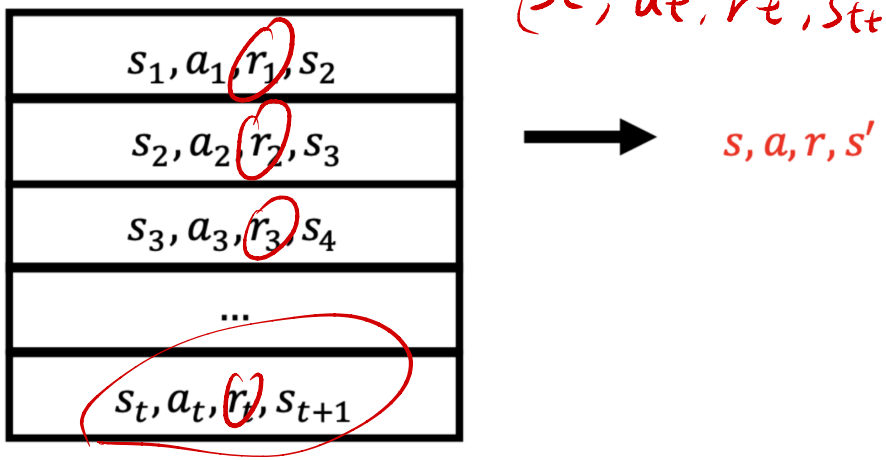
6. $t \leftarrow t + 1$ and loop until termination.

Issues with DQN

- Q-Learning with nonlinear value function approximation: divergent and instable
- Possible issues:
  1. Correlations between samples
  2. Non-stationary targets
  3. Bias Propagation by bootstrapping
  4. Overestimation by max operation
- Improving DQN with the following methods:
  1. Experience replay & prioritized experience replay
  2. Fixed Q-targets(target network)
  3. Double Q-learning
  4. Dueling network

Improving DQN: Experience Replay

- Consecutive states, s_t and s_{t+1}, are strongly correlated (which is bad.)
- Divide trails of agent into several quadruples $(s_t, a_t, r_{t+1}, s_{t+1})$, store them into a replay buffer $\mathcal{D}$ with size $b(10^5 \sim 10^6)$



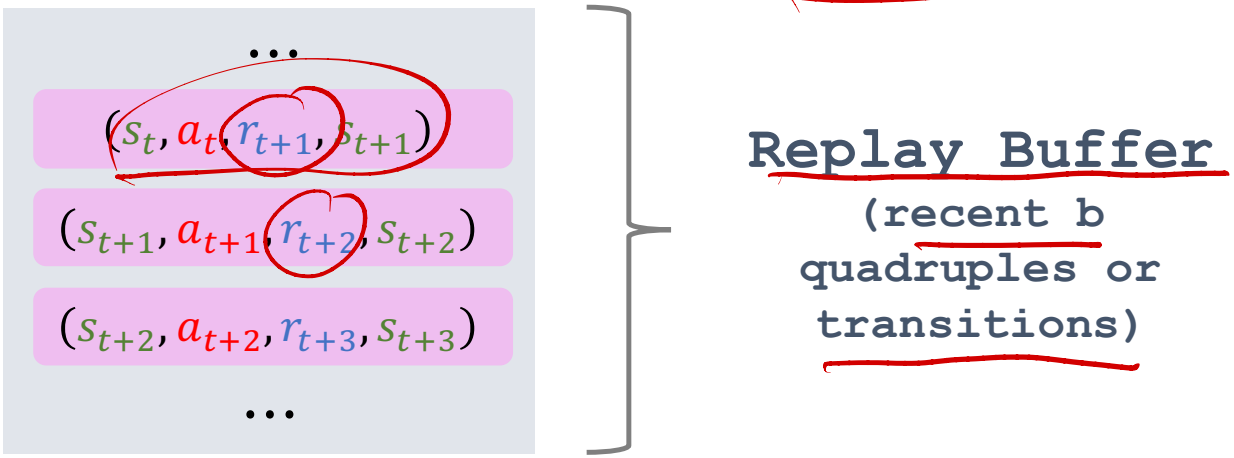
- When the number of quadruples is not big enough, we only make interactions with environment, but parameters $\mathbf{w}$ will NOT be updated.

Improving DQN: Experience Replay

- When the number of quadruples is big enough(e.g., 8×10^4), perform experience replay by random sampling an experience quadrupl from the replay buffer: $(s, a, r, s') \sim \mathcal{D}$, then update parameters $\mathbf{w}$.
- Reduce the correlations among samples
- Improve the sampling efficiency
- Suitable only for off-policy learning since the experiences are generated from behavior policy, usually different from the target policy.

Improving DQN: Prioritized Experience Replay

- Quadruples in Replay buffer $\mathcal{D}$ are NOT equally important



- Priority is put on the quadruples whose experiences are scarce (car accident scenarios in automatic driving)
- 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.
- If a quadruple has a high absolute value of TD error $|\delta_t|$, it will be given high priority.

Improving DQN: Prioritized Experience Replay

- Sampling probability $p_t \propto |\delta_t| + \epsilon$, where $\epsilon > 0$ can avoid the occurrence of zero p_t
- If uniform sampling is used, learning rate(step size) α is the same for all quadruples.
- If importance sampling is used, learning rate(step size) α shall be adjusted according to the priority.
- Scaling learning rate(step size) by $(bp_t)^{-\beta}$, where $\beta \in (0, 1)$.
- High-priority transitions (with high p_t) have low learning rates(step size).
- In the beginning, set β small; increase β to 1 over time.

Improving DQN: Prioritized Experience Replay

- Associate a quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$ with a TD error δ_t
- If such quadruple is newly collected before training, we do not know its TD error. Simply set its δ_t to be the maximum value with the highest priority.

Quadruples & TD Error	Sampling Probabilities	Learning Rates
...	...	...
$(s_t, a_t, r_{t+1}, s_{t+1}), \delta_t$	$p_t \propto \lvert\delta_t\rvert + \epsilon$	$\alpha \cdot (b\, p_t)^{-\beta}$
$(s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}), \delta_{t+1}$	$p_{t+1} \propto \lvert\delta_{t+1}\rvert + \epsilon$	$\alpha \cdot (b\, p_{t+1})^{-\beta}$
$(s_{t+2}, a_{t+2}, r_{t+3}, s_{t+3}), \delta_{t+2}$	$p_{t+2} \propto \lvert\delta_{t+2}\rvert + \epsilon$	$\alpha \cdot (b\, p_{t+2})^{-\beta}$
...	...	...

Big $|\delta_t|$ ==> High probability ==> Small learning rate

Outline

Bootstrapping in DQN

- TD target y_t is partly an estimate made by the DQN Q:

$$y_t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w}).$$

- We use y_t, which is partly based on Q, to update Q itself.

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot (y_t - Q(s_t, a_t; \mathbf{w})) \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w})$$

Problem of Overestimation

- TD learning makes DQN overestimate action-values, which leads to wrong decisions and bad choices of actions in each state.
- Reason I: The maximization operation
  - TD target: $y_t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w})$
  - TD target is bigger than the real action-value
- Reason II: Bootstrapping propagates the overestimation

Reason I: Maximization

- Let $x_1, \ldots, x_n$ be observed real numbers
- Add zero-mean random noise to $x_1, \ldots, x_n$ and obtain $Q_1, \ldots, Q_n$
- The zero-mean noise does not affect the mean:

$$E[\text{mean}_{i \in \{1,\ldots,n\}} Q_i] = \text{mean}_{i \in \{1,\ldots,n\}} x_i$$

- The zero-mean noise increases the maximum:

$$E[\max_{i \in \{1,\ldots,n\}} Q_i] \geq \max_{i \in \{1,\ldots,n\}} x_i$$

- The zero-mean noise decreases the minimum:

$$E[\min_{i \in \{1,\ldots,n\}} Q_i] \leq \min_{i \in \{1,\ldots,n\}} x_i$$

Reason I: Maximization

- True optimal action-values $x_i = q^*(s, a_i), i = 1, \ldots, n$
- Noisy estimations made by DQN: $Q_i = Q(s, a_i; \mathbf{w}) = x_i + \epsilon_i$
- Suppose the estimation is unbiased: $E(\epsilon_i) = 0$
- Then we have

$$E[\max_{i \in \{1,\ldots,n\}} Q_i] \geq \max_{i \in \{1,\ldots,n\}} x_i$$

or equivalently

$$E[\max_{a \in \mathcal{A}} Q(s, a; \mathbf{w})] \geq \max_{a \in \mathcal{A}} q^*(s, a)$$

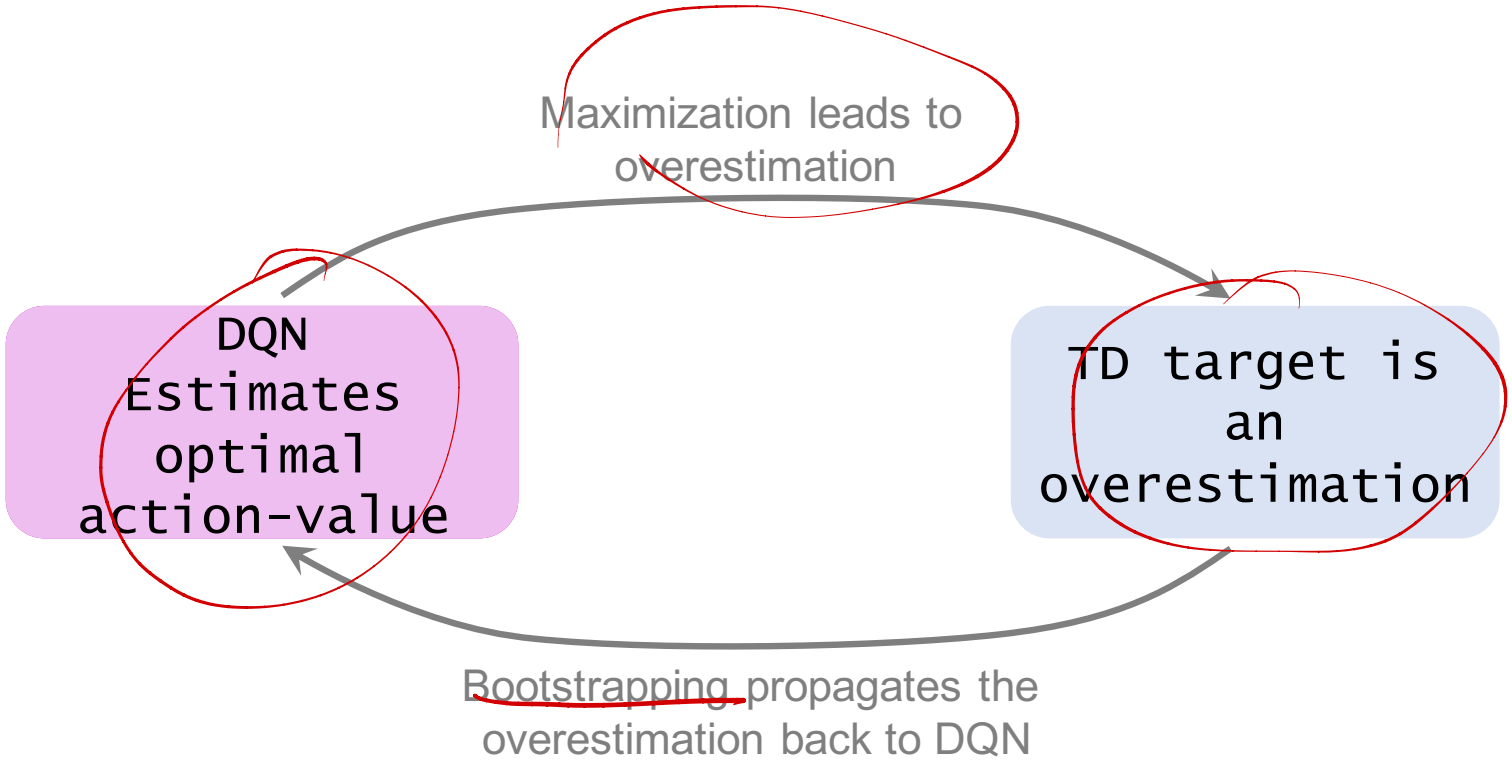
Reason I: Maximization

- We conclude that $q_{t+1} = \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \mathbf{w})$ is an overestimation of the optimal true action-value at time $t+1$
- The TD target, $y_t = r_{t+1} + \gamma q_{t+1}$ is thereby an overestimation
- TD learning pushes $Q(s_t, a_t; \mathbf{w})$ towards y_t which overestimates the true optimal action-value.

Reason II: Bootstrapping

- TD learning performs bootstrapping: in part use $q_{t+1} = \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \mathbf{w})$, then use the TD target to update $Q(s_t, a_t; \mathbf{w})$
- Suppose DQN overestimates the action-value.
- Then $Q(s_{t+1}, a; \mathbf{w})$ is an overestimation
- The maximization further pushes q_{t+1} up.
- When q_{t+1} is used for updating $Q(s_t, a_t; \mathbf{w})$, the overestimation is propagated back to DQN.

Why does overestimation happen?



Solutions

- Problem: DQN trained by TD overestimates action-values
- Solution 1: Use a target network to compute TD targets. (Address the problem caused by bootstrapping.)
- Solution 2: Use double DQN to alleviate the overestimation caused by maximization.

Solution 1: Target Network

- Target network $Q(s, a; \mathbf{w}^{-1})$: the same network structure as the DQN $Q(s, a; \mathbf{w})$, but with Different parameters $\mathbf{w}^{-1} \neq \mathbf{w}$
- Sample a quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$ from the replay buffer
- Use $Q(s, a; \mathbf{w}^{-1})$ to compute TD target:

$$y_t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w}^{-1})$$

- Compute TD error:

$$\delta_t = y_t - Q(s_t, a_t; \mathbf{w})$$

- Performing Back-Propagation, obtain gradient value $\nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w})$ and update the parameters of DQN as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w})$$

- Periodically update target network with super-parameter $\tau \in (0, 1)$ after multiple updates of DQN parameters $\mathbf{w}$:

$$\mathbf{w}^{-1} \leftarrow \tau \cdot \mathbf{w} + (1 - \tau) \cdot \mathbf{w}^{-1}$$

$\tau \approx 0.999$

Intuition: Why Fixed Target

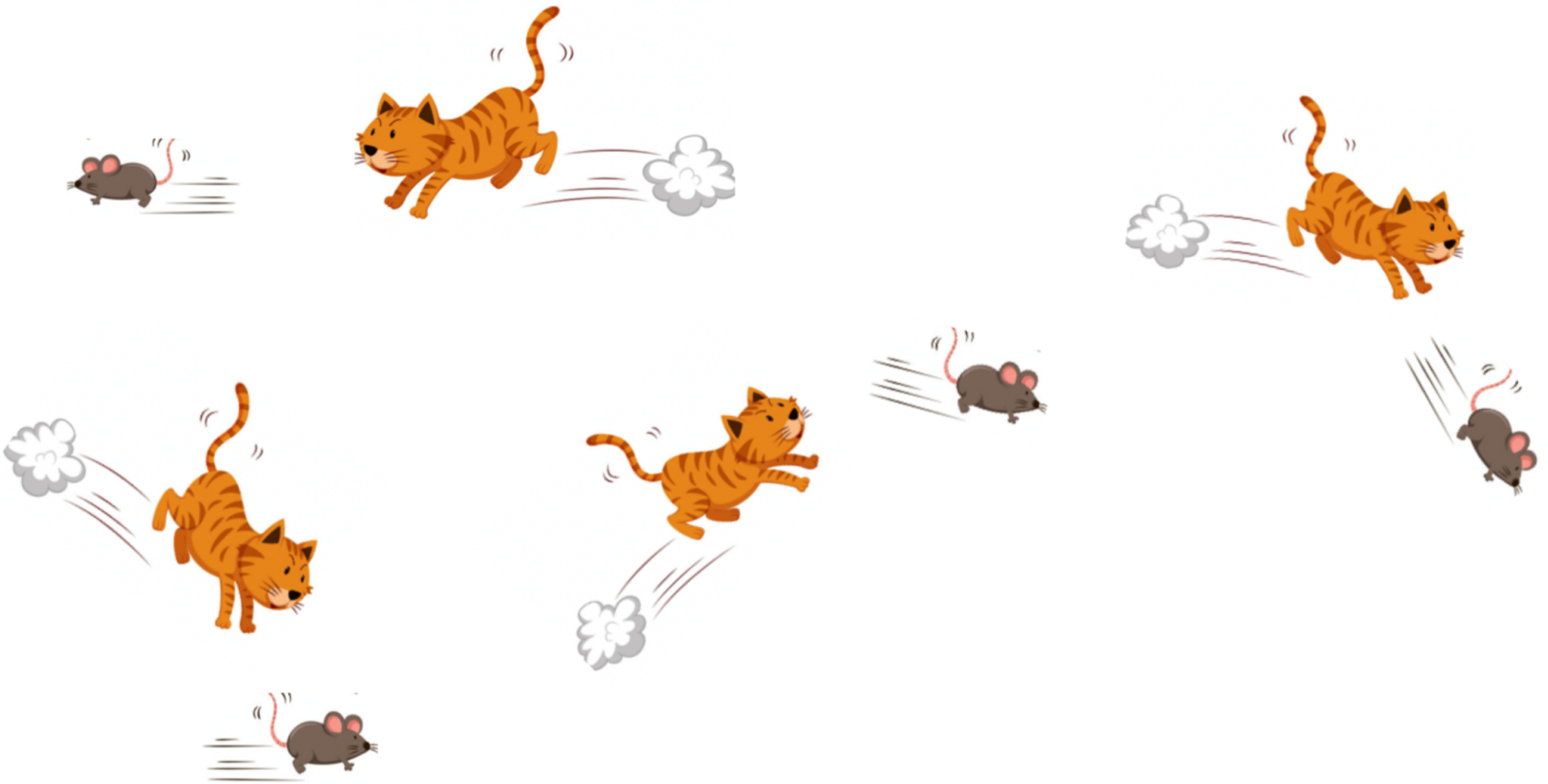
- To help improve stability, fix the parameters $\mathbf{w}^{-1}$ of target network used in the TD target calculation for multiple updates of DQN parameters $\mathbf{w}$
- In the original update, both Q estimation and TD target shifts at each time step
- Imagine a cat (Q estimation) is chasing after a mice (TD target or Q Target)
- The cat must reduce the distance to the mice

Q target

Q estimation

Intuition: Why fixed target

- Both the cat and mice are moving,



Intuition: 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

Solution 2: Double Q-learning

X_1, X_2, i.i.d.

$$\mathbb{E}[\max(X_1, X_2)] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])$$

- Target network can alleviate the bias caused by bootstrapping, but fail to alleviate the overestimation caused by maximization operation.
- Double Q-learning can alleviate the overestimation caused by maximization operation.
- Idea: use two networks to decouple the action selection from action evaluation

Classical DQN

- Action Selection using DQN

$$a^* = \arg\max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w})$$

- Action Evaluation using DQN

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, a^*; \mathbf{w})$$

Target Network

- Action Selection using target network

$$a^* = \arg\max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w}^{-1})$$

- Action Evaluation using target network

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, a^*; \mathbf{w}^{-1})$$

Double DQN

- Action Selection using DQN

$$a^* = \arg\max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w})$$

- Action Evaluation using target network

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, a^*; \mathbf{w}^{-1})$$

- Double DQN alleviates overestimation:

$$Q(s_{t+1}, a^*; \mathbf{w}^{-1}) \leq \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w}^{-1})$$

then

$$y_t(\text{Double DQN}) \leq y_t(\text{Target Network}).$$

- It is the best among the three; but overestimation still happens

Solution 2: Double DQN

- Target network $Q(s, a; \mathbf{w}^{-1})$: the same network structure as the DQN $Q(s, a; \mathbf{w})$, but with Different parameters $\mathbf{w}^{-1} \neq \mathbf{w}$
- Sample a quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$ from the replay buffer
- Action Selection using DQN

$$a^* = \arg\max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \mathbf{w})$$

- Compute TD target(action Evaluation using target network):

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, a^*; \mathbf{w}^{-1})$$

- Compute TD error:

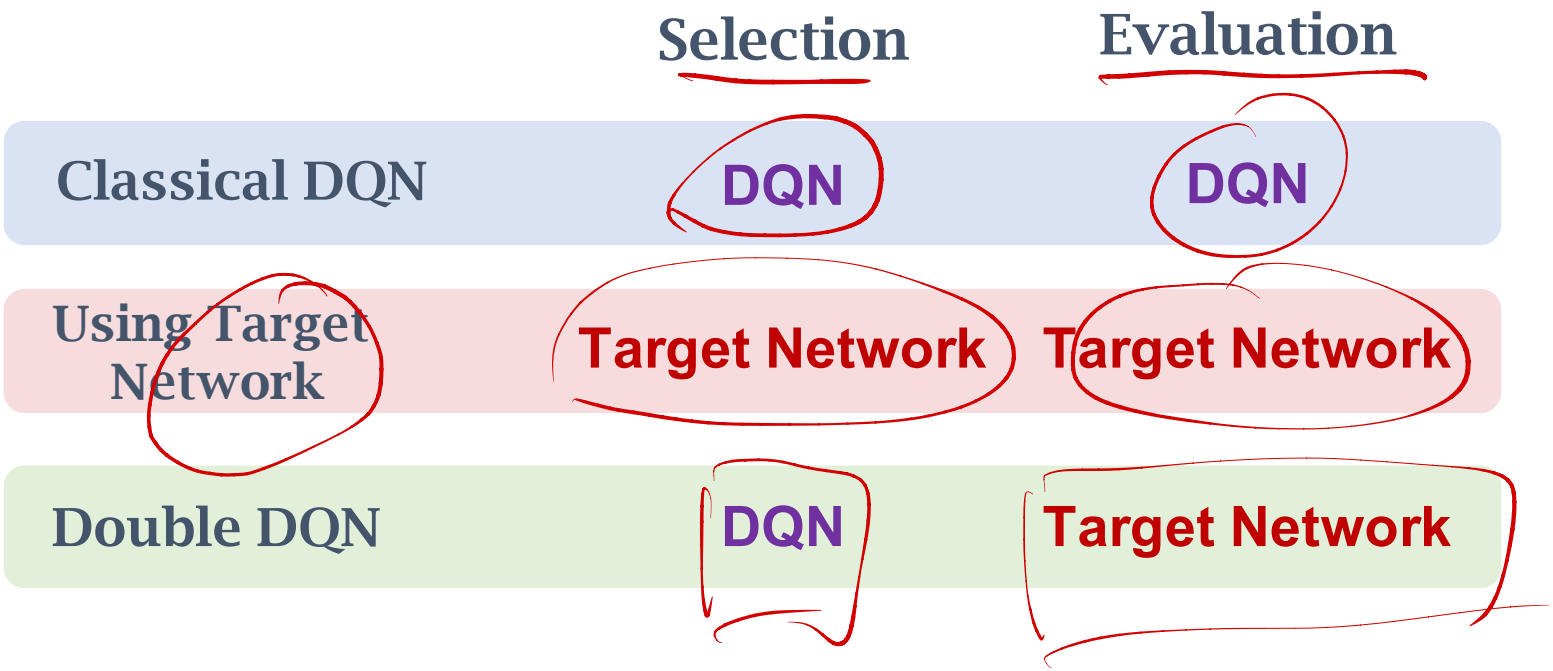
$$\delta_t = y_t - Q(s_t, a_t; \mathbf{w})$$

- Update the parameters of DQN with BP as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w})$$

- Periodically update target network with super-parameter $\tau \in (0, 1)$ after multiple updates of DQN parameters $\mathbf{w}$:

Computing TD Targets



Summary

- Because of the maximization, the TD target overestimates the true action-value.
- By creating a "positive feedback loop", bootstrapping further exacerbates the overestimation.
- Target network can partly avoid bootstrapping.
- Double DQN alleviates the overestimation caused by the maximization.

Outline

Dueling Network.

Optimal Advantage Function

$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$

Advantage Function.

- Optimal action-value function $Q^*(s,a)$ & optimal state-value function $V^*(s)$:

$$V^*(s) = \max_a Q^*(s,a)$$

- Optimal advantage function $A^*(s,a)$:

$$A^*(s,a) = Q^*(s,a) - V^*(s)$$

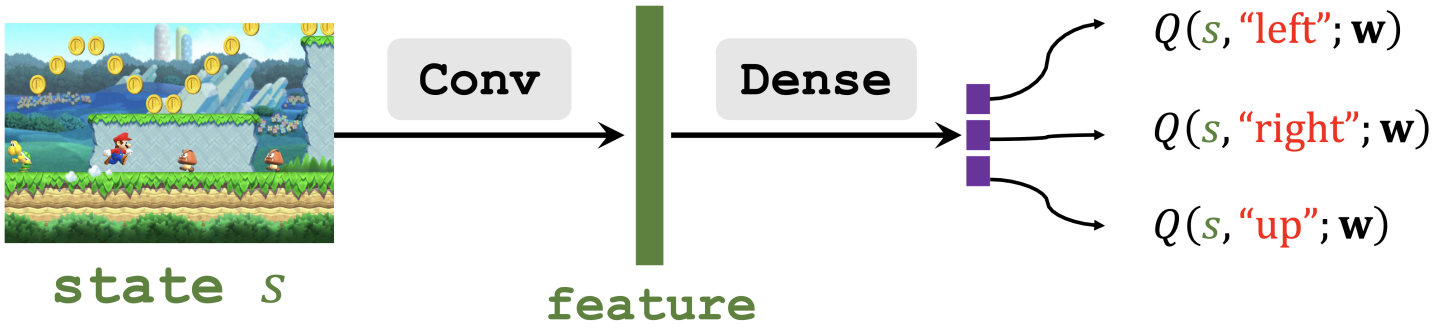
- Then we have

$$\max_a A^*(s,a) = \max_a Q^*(s,a) - V^*(s) = 0.$$

- So

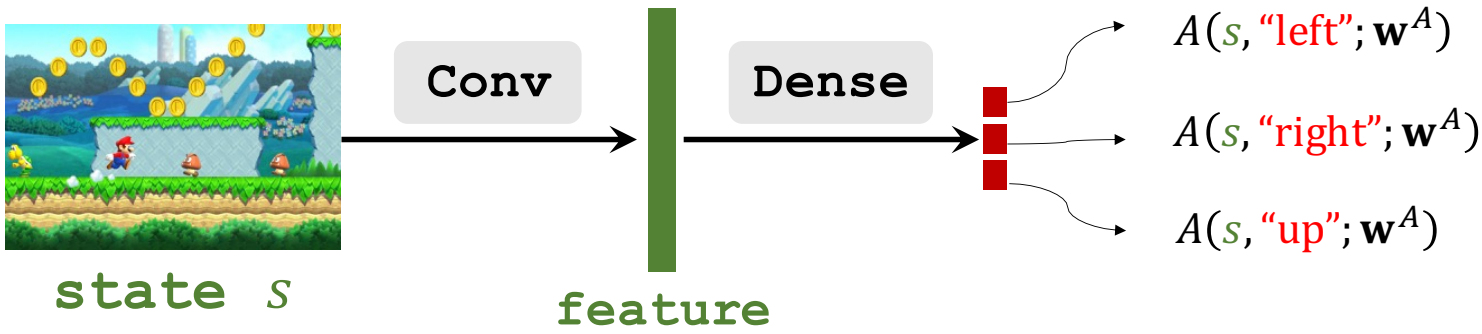
$$Q^*(s,a) = V^*(s) + A^*(s,a) - \max_a A^*(s,a).$$

Revisit DQN


Conv
Dense
$Q(s, \text{“left”}; \mathbf{w})$
$Q(s, \text{“right”}; \mathbf{w})$
$Q(s, \text{“up”}; \mathbf{w})$
state s
feature

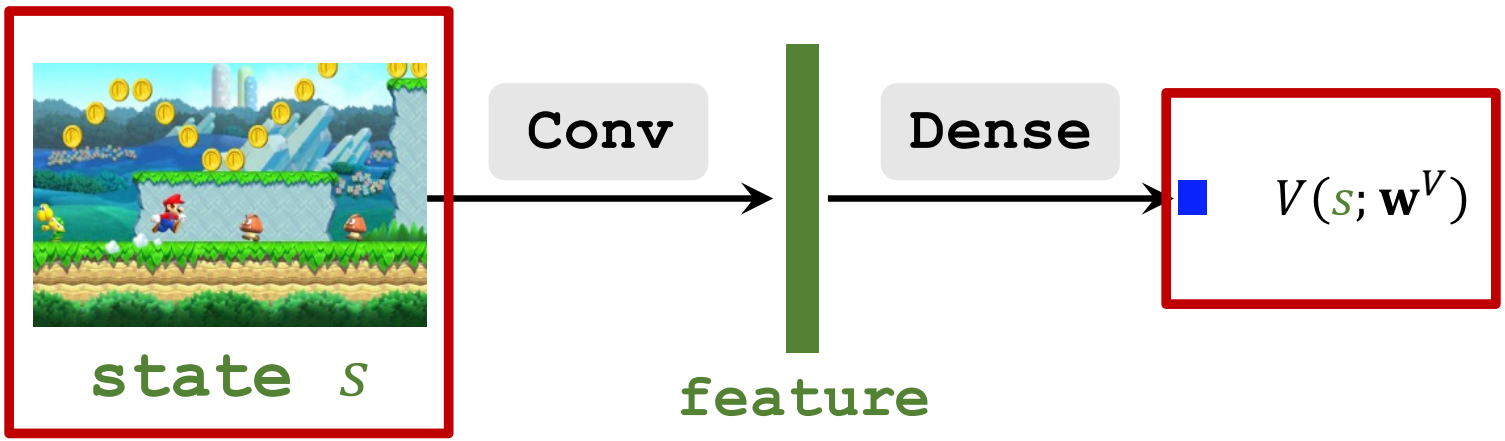
Approximating Optimal Advantage Function

- Approximate $A^*(s, a)$ by a neural network $A(s, a; \mathbf{W}^A)$



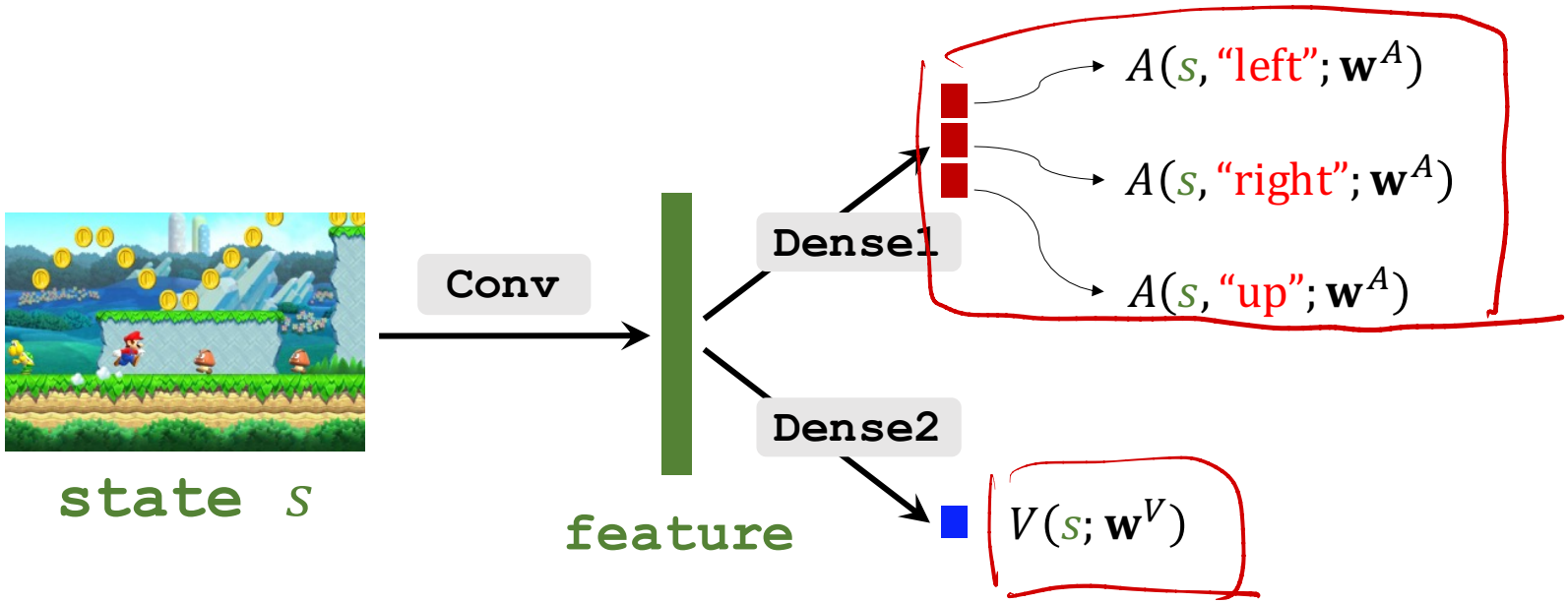
Approximating Optimal State-Value Function

- Approximate $V^*(s)$ by a neural network $V(s; \mathbf{W}^V)$



Approximating Optimal Action-Value Function

- Approximate $Q^*(s, a)$ by dueling network $Q(s, a; \mathbf{W}^A, \mathbf{W}^V)$



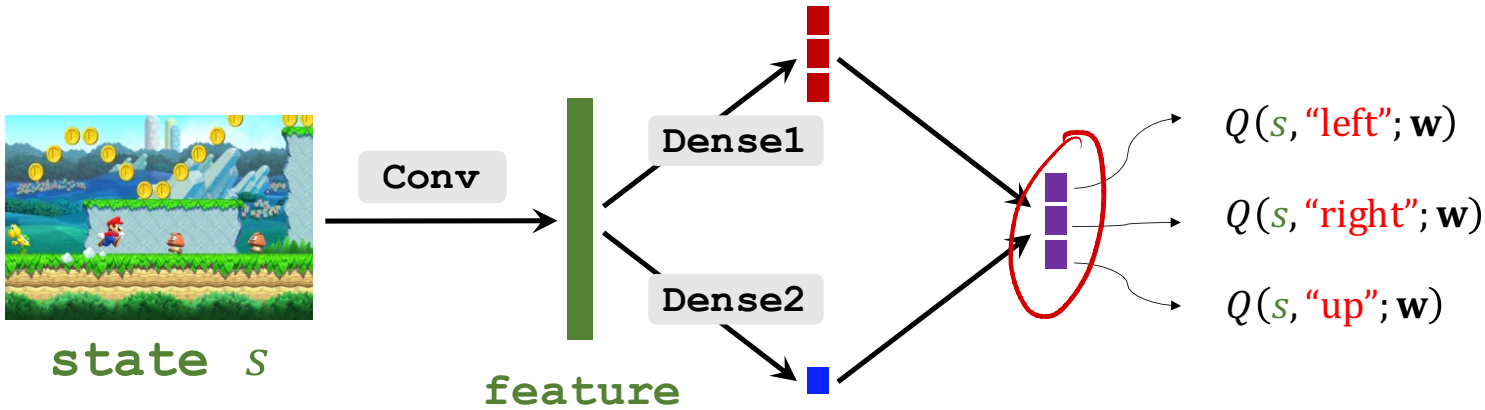
Dueling Network

- We have

$$Q(s,a;\mathbf{W}^A,\mathbf{W}^V) = V(s;\mathbf{W}^V) + A(s,a;\mathbf{W}^A) - \max_a A(s,a;\mathbf{W}^A)$$

- In practice with better results

$$Q(s,a;\mathbf{W}^A,\mathbf{W}^V) = V(s;\mathbf{W}^V) + A(s,a;\mathbf{W}^A) - \text{mean}_a A(s,a;\mathbf{W}^A)$$



Problem of Non-identifiability

- If We use the equation

$$Q(s, a; \mathbf{W}^A, \mathbf{W}^V) = V(s; \mathbf{W}^V) + A(s, a; \mathbf{W}^A)$$

- Non-identifiability problem: functions V and A can have fluctuations arbitrarily as long as the summary of both is constant, leading to the instability of neural network.
- The following equation solved the problem

$$Q(s, a; \mathbf{W}^A, \mathbf{W}^V) = V(s; \mathbf{W}^V) + A(s, a; \mathbf{W}^A) - \max_a A(s, a; \mathbf{W}^A)$$

Dueling Network

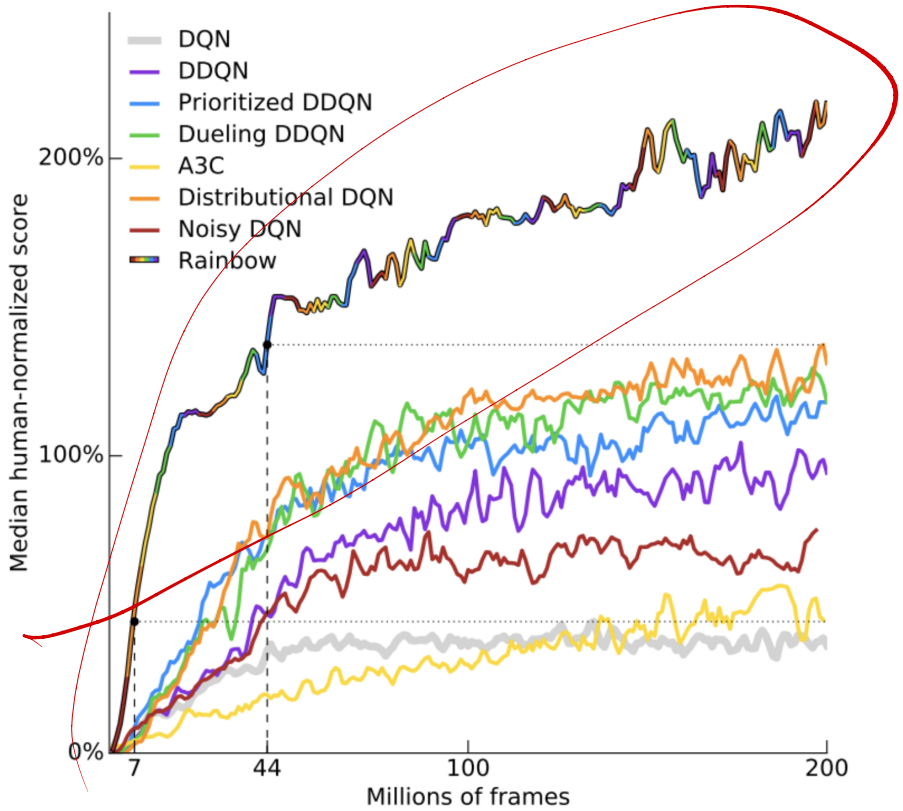
- Dueling network controls the agent in the same way as DQN
- Train dueling network by TD in the same way as DQN
- Do not train V and A separately
- Learn the parameters $\mathbf{W} = (\mathbf{W}^A, \mathbf{W}^V)$ in the same way as the other DQNs.
- Tricks can be used in the same way: prioritized experience replay, Double DQN, et al.

Rainbow DQN


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

Improving over the DQN

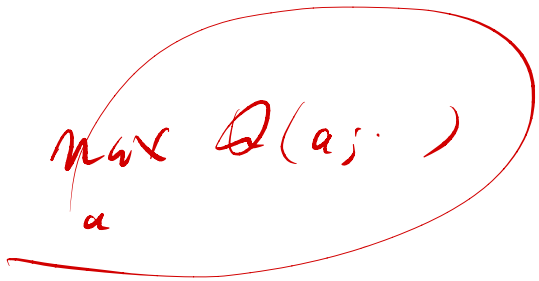
- Rainbow: Combining Improvements in Deep Reinforcement Learning. Matteo Hessel et al. AAAI 2018.
- It examines six extensions to the DQN algorithm and empirically studies their combination



Demo of DQNs

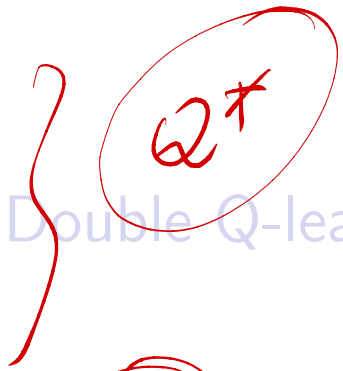
- Demo of deep q-learning for Breakout:
  `https://www.youtube.com/watch?v=V1eYniJ0Rnk`
- Demo of Flappy Bird by DQN:
  `https://www.youtube.com/watch?v=xM62SpKAZHU`

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



Recall: Classical On-Policy SARSA

given current state s_t, policy π and functions $\{q_\pi(s_t, a), a \in \mathcal{A}\}$

1. Sample action $a_t \sim \pi(\cdot|s_t)$, perform action a_t and obtain reward r_{t+1} and new state s_{t+1}.
2. Sample action $a_{t+1} \sim \pi(\cdot|s_{t+1})$ and collect the training data: quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ sarsa
3. Compute the TD target and TD error

$$y_t = r_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1})$$
$$\delta_t = y_t - q_\pi(s_t, a_t)$$

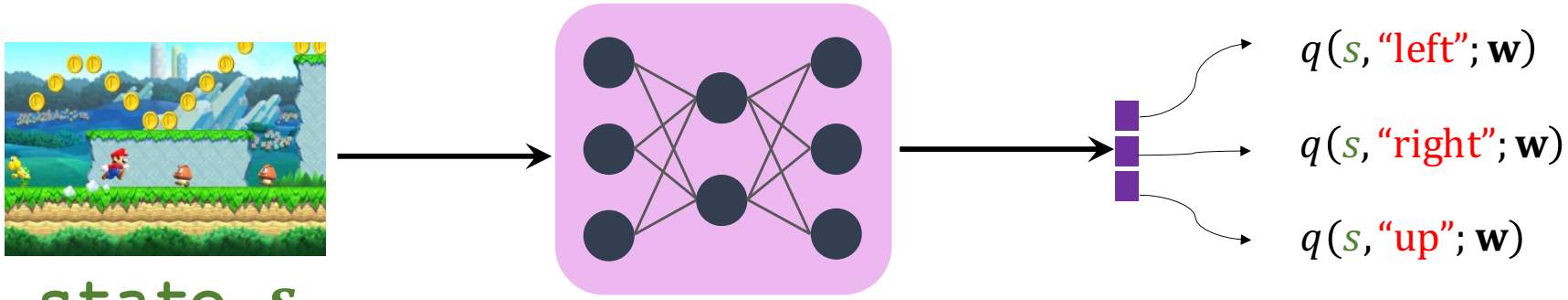
4. Update the value of $q_\pi(s_t, a_t)$ as follows:

$$q_\pi(s_t, a_t) \leftarrow q_\pi(s_t, a_t) + \alpha \cdot \delta_t$$

5. $t \leftarrow t + 1$ and loop until termination.

Value Network

- Given policy π, approximate action-value functions $q_\pi(s, a)$ by the value network $q(s, a; \mathbf{w})$.



Deep On-Policy SARSA

given current state s_t, policy π, neural network parameters $\mathbf{w}$ and functions $\{q(s_t, a; \mathbf{w}), a \in \mathcal{A}\}$

1. Sample action $a_t \sim \pi(\cdot|s_t)$, perform action a_t and obtain reward r_{t+1} and new state s_{t+1}.
2. Sample action $a_{t+1} \sim \pi(\cdot|s_{t+1})$ and collect the training data: quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$
3. Compute the TD target and TD error

$$y_t = r_{t+1} + \gamma q(s_{t+1}, a_{t+1}; \mathbf{w})$$
$$\delta_t = y_t - q(s_t, a_t; \mathbf{w})$$

4. Performing Back-Propagation, obtain gradient value $\nabla_{\mathbf{w}} q(s_t, a_t, \mathbf{w})$ and update the parameters as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w})$$

5. $t \leftarrow t + 1$ and loop until termination.

Part of Actor-Critic Algorithm

given current state s_t, policy π, neural network parameters $\mathbf{w}$ and functions $\{q(s_t, a; \mathbf{w}), a \in \mathcal{A}\}$

1. Sample action $a_t \sim \pi(\cdot|s_t)$, perform action a_t and obtain reward r_{t+1} and new state s_{t+1}.
2. Sample action $a'_{t+1} \sim \pi(\cdot|s_{t+1})$ and collect the training data: quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a'_{t+1})$
3. Compute the TD target and TD error

$$y_t = r_{t+1} + \gamma q(s_{t+1}, a'_{t+1}; \mathbf{w})$$
$$\delta_t = y_t - q(s_t, a_t; \mathbf{w})$$

4. Performing Back-Propagation, obtain gradient value $\nabla_{\mathbf{w}} q(s_t, a_t, \mathbf{w})$ and update the parameters as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w})$$

5. Update(improve) policy π according to some policy-based algorithm independent of SARSA

Outline

Classical Control Problems

- Commonly used as benchmarks for tabular RL and RL algorithms using linear function approximators
  - Cartpole: balancing a pole on a cart
  - Mountain Car: trying to get a car up a mountain using momentum
  - Acrobot: swinging a pole up using momentum and subsequently balancing it
- Still sometimes used to benchmark deep RL algorithms

Games

- Board games including mastering Go & Poker
- Video Games are also popular benchmarks because
  - many of these games have large observation space and/or large action space
  - they are often non-Markovian, which require specific care
  - they also usually require very long planning horizons (e.g., due to sparse rewards)
- Adopted for several research topics including
  - multi-task learning & transfer learning
  - lifelong-learning & curriculum learning
  - predictive planning & hierarchical planning
  - meta-RL & multi-agent RL
- Limitation: majority investigate discrete action decisions

Popular Video Game Platforms

- Arcade Learning Environment (ALE): a suite of iconic Atari games, including Pong, Asteroids, Montezuma's Revenge, Space Invaders, Seaquest and Breakout
- General Video Game AI (GVGAI) competition framework
- VizDoom: implements the Doom video game as a simulated environment for RL
- Minecraft
- Quake video game
- StarCraft

Continuous Control Systems & Robotics

- MuJoCo simulation framework: provide several locomotion benchmark tasks
- Roboschool: provides the same locomotion tasks along with more complex tasks involving humanoid robot simulations

Easy-to-use Wrappers

- OpenAI Gym: provides ready access to environments such as
  - Atari, board games, Box2d games
  - classical control problems
  - MuJoCo robotics simulations
- Gym Retro
- μniverse
- SerpentAI

Frameworks of Deep RL

Framework	Deep RL	Python interface	Automatic GPU support
DeeR	yes	yes	yes
Dopamine	yes	yes	yes
ELF	yes	no	yes
OpenAI baselines	yes	yes	yes
PyBrain	yes	yes	no
RL-Glue	no	yes	no
RLPy	no	yes	no
rllab	yes	yes	yes
TensorForce	yes	yes	yes

Useful Links

- OpenAI Gym: https://github.com/openai/gym
- OpenAI Baseline: https://github.com/openai/baselines/
- OpenAI Spinning UP: https://github.com/openai/spinningup
- Stable Baseline 3: https://github.com/DLR-RM/stable-baselines3
- RL Baselines3 Zoo: https://github.com/DLR-RM/rl-baselines3-zoo
- Tutorial of Tools for Robotic Reinforcement Learning: https://araffin.github.io/tools-for-robotic-rl-icra2022/

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.
- RL course slides from Shusen Wang