Week 5: Value Function Approximation and Deep Q-Learning

Bolei Zhou

The Chinese University of Hong Kong

October 6, 2020

Announcement

1. HW1 tutorial session at TA office hour tomorrow
2. TA hour office after every homework due will be homework tutorial session

This Week's Plan

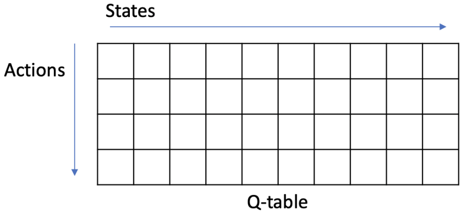
1. Introduction on function approximation
2. Value function approximation for prediction
3. Value function approximation for control
4. Batch RL and least square prediction and control
5. Deep Q-Learning: Entering the modern RL world

Introduction: Scaling up RL

1. Previous lectures on small RL problems:
    1. Cliff walk: 4×16 states
    2. Mountain car: 1600 states
    3. Tic-Tac-Toe: 10^3 states
2. Large-scale problems:
    1. Backgammon: 10^{20} states
    2. Chess: 10^{47} states
    3. Game of Go: 10^{170} states
    4. Robot Arm and Helicopter have continuous state space
    5. Number of atomic in universe: 10^{80}
3. Challenge: How can we scale up the model-free methods for prediction and control?

Introduction: Scaling up RL

1. In tabular methods we represent value function by a lookup table:
   1. Every state s has an entry $V(s)$
   2. Every state-action pair s, a has an entry $Q(s, a)$



2. Challenges with large MDPs:
   1. too many states or actions to store in memory
   2. too slow to learn the value of each state individually

Scaling up RL with Function Approximation

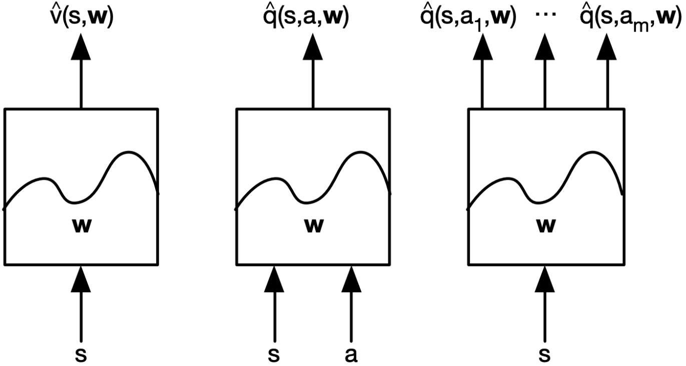
1. How to avoid explicitly learning or storing for every single state:
    1. Dynamics or reward model
    2. Value function, state-action function
    3. Policy
2. Solution: Estimate with function approximation

$$
\begin{aligned}
\hat{v}(s, \mathbf{w}) &\approx v^{\pi}(s) \\
\hat{q}(s, a, \mathbf{w}) &\approx q^{\pi}(s, a) \\
\hat{\pi}(a, s, \mathbf{w}) &\approx \pi(a|s)
\end{aligned}
$$

    1. Generalize from seen states to unseen states
    2. Update the parameter **w** using MC or TD learning

Types of value function approximation

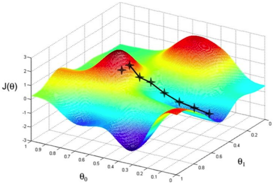
Several function designs:



Function Approximators

1. Many possible function approximators:
   1. Linear combinations of features
   2. Neural networks
   3. Decision trees
   4. Nearest neighbors
2. We will focus on function approximators that are differentiable
   1. Linear feature representations
   2. Neural networks

Review on Gradient Descend



1. Consider a function $J(\mathbf{w})$ that is a differentiable function of a parameter vector $\mathbf{w}$
2. Goal is to find parameter $\mathbf{w}^*$ that minimizes J
3. Define the gradient of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \left(\frac{\partial J(\mathbf{w})}{\partial w_1}, \frac{\partial J(\mathbf{w})}{\partial w_2}, ..., \frac{\partial J(\mathbf{w})}{\partial w_n}\right)^T$$

4. Adjust $\mathbf{w}$ in the direction of the negative gradient, where α is step-size

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

Value Function Approximation with an Oracle

1. We assume that we have the oracle for knowing the true value for $v^\pi(s)$ for any given state s
2. Then the objective is to find the best approximate representation of $v^\pi(s)$
3. Thus use the mean squared error and define the loss function as

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

4. Follow the gradient descend to find a local minimum

$$\begin{aligned}\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w})\\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \Delta\mathbf{w}\end{aligned}$$

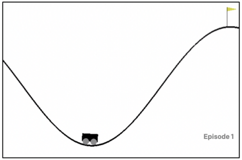
Representing State with Feature Vectors

1. Represent state using a feature vector

$$\mathbf{x}(s) = \left(x_1(s), ..., x_n(s)\right)^T$$

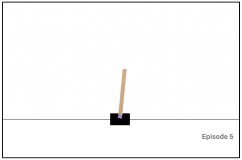
2. For example:

Mountain Car



Position of car, velocity of car

Cart Pole



Position of cart, velocity of cart, angle of pole, rotation rate of pole

Game of Go in AlphaGo

Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

48 places of 19x19 feature maps

Linear Value Function Approximation

1. Represent value function by a linear combination of features

$$\hat{v}(s,\mathbf{w}) = \mathbf{x}(s)^T\mathbf{w} = \sum_{j=1}^{n} x_j(s)w_j$$

2. The objective function is quadratic in parameter **w**

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

3. Thus the update rule is as simple as

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

$Update = StepSize \times PredictionError \times FeatureValue$

4. Stochastic gradient descent converges to global optimum. Because in the linear case, there is only one optimum, thus local optimum is automatically converge to or near the global optimum.

Linear Value Function Approximation with Table Lookup Feature

1. Table lookup is a special case of linear value function approximation
2. Table lookup feature is one-hot vector as follows

$$\mathbf{x}^{table}(S) = \left(\mathbf{1}(S = s_1), ..., \mathbf{1}(S = s_n)\right)^T$$

3. Then we can see that each element on the parameter vector $\mathbf{w}$ indicates the value of each individual state

$$\hat{v}(S, \mathbf{w}) = \left(\mathbf{1}(S = s_1), ..., \mathbf{1}(S = s_n)\right)\left(w_1, ..., w_n\right)^T$$

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

Value Function Approximation for Model-free Prediction

1. In practice, no access to oracle of the true value $v^{\pi}(s)$ for any state s
2. Recall model-free prediction
    1. Goal is to evaluate v^{π} following a fixed policy π
    2. A lookup table is maintained to store estimates v^{π} or q^{π}
    3. Estimates are updated after each episode (MC method) or after each step (TD method)
3. Thus what we can do is to **include the function approximation step in the loop**

Incremental VFA Prediction Algorithms

1. We assumed that true value function $v^{\pi}(s)$ given by supervisor/oracle

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

2. But in RL there is no supervisor, only rewards
3. In practice, we substitute the target for $v^{\pi}(s)$
   1. For MC, the target is the actual return G_t

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

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

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

Monte-Carlo Prediction with VFA

1. Return G_t is an unbiased, but noisy sample of true value $v^{\pi}(S_t)$
2. Why unbiased? $\mathbb{E}[G_t] = v^{\pi}(S_t)$
3. So we have the training data that can be used for supervised learning in VFA:

$$< S_1, G_1 >, < S_2, G_2 >, ..., < S_T, G_T >$$

4. Using linear Monte-Carlo policy evaluation

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

5. Monte-Carlo prediction converges, in both linear and non-linear value function approximation.

TD Prediction with VFA

1. TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ is a biased sample of true value $v^{\pi}(S_t)$
2. Why biased? It is drawn from our previous estimate, rather than the true value: $\mathbb{E}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})] \neq v^{\pi}(S_t)$
3. We have the training data used for supervised learning in VFA:

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

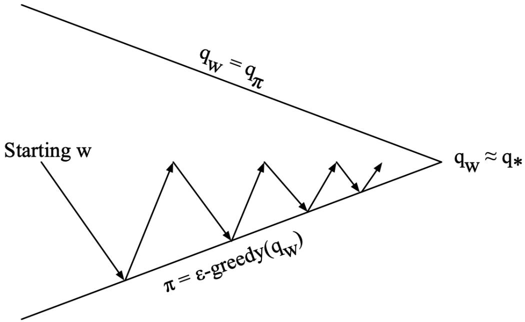
4. Using linear TD(0), the stochastic gradient descend update is

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

   This is also called as semi-gradient, as we ignore the effect of changing the weight vector **w** on the target
5. Linear TD(0) converges(close) to global optimum

Control with Value Function Approximation

Generalized policy iteration



1. Policy evaluation: approximate policy evaluation, $\hat{q}(.,.,\mathbf{w}) \approx q^\pi$
2. Policy improvement: ϵ-greedy policy improvement

Action-Value Function Approximation

1. Approximate the action-value function

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

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

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

3. Stochastic gradient descend to find a local minimum

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

Linear Action-Value Function Approximation

1. Represent state and action using a feature vector

$$\mathbf{x}(s,a) = \big(x_1(s,a), ..., x_n(s,a)\big)^T$$

2. Represent action-value function by a linear combination of features

$$\hat{q}(s,a,\mathbf{w}) = \mathbf{x}(s,a)^T\mathbf{w} = \sum_{j=1}^{n} x_j(s,a)w_j$$

3. Thus the stochastic gradient descend update

$$\Delta\mathbf{w} = \alpha(q^{\pi}(s,a) - \hat{q}(s,a,\mathbf{w}))\mathbf{x}(s,a)$$

Incremental Control Algorithm

Same to the prediction, there is no oracle for the true value $q^{\pi}(s,a)$, so we substitute a target

1. For MC, the target is the return G_t

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

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

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

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

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

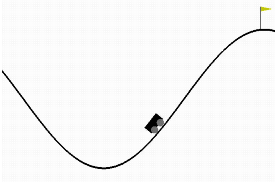
Semi-gradient Sarsa for VFA Control

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

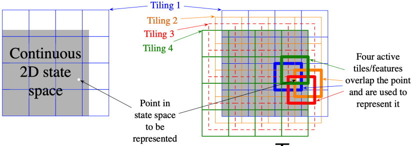
Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
    $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        If S' is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$

Mountain Car Example



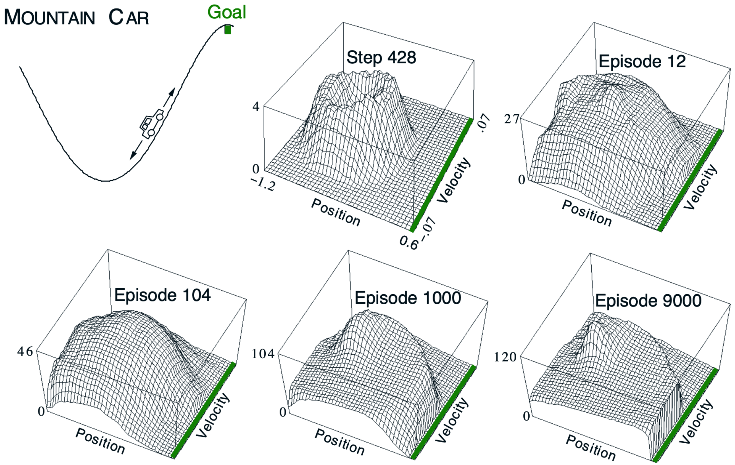
1. Control task: move the car up the hill with forward and backward acceleration
    1. Continuous state: [position of the car, velocity of the car]
    2. Action: [full throttle forward, full throttle backward, zero throttle]
2. Grid-tiling coding of the continuous 2D state space with 4 tilings



3. Q function approximator: $\hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s, a) = \sum_{i=1}^{4} w_i x_i(s, a)$

Mountain Car Example

1. Visualization of cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned over episodes



Mountain Car Example

1. Code example: `https://github.com/cuhkrlcourse/RLexample/blob/master/modelfree/q_learning_mountaincar.py`

Convergence of Control Methods with VFA

1. For Sarsa,

$$\Delta \mathbf{w} = \alpha\big(R_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}) - \hat{q}(s_t, a_t, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s_t, a_t, \mathbf{w})$$

2. For Q-learning,

$$\Delta \mathbf{w} = \alpha\big(R_{t+1} + \gamma \max_a \hat{q}(s_{t+1}, a, \mathbf{w}) - \hat{q}(s_t, a_t, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s_t, a_t, \mathbf{w})$$

3. TD with VFA doesn't follow the gradient of any objective function
4. The updates involve doing an approximate Bellman backup followed by fitting the underlying value function
5. That is why TD can diverge when off-policy or using non-linear function approximation
6. 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

Potential problematic issues:

1. **Function approximation**: A scalable way of generalizing from a state space much larger than the memory and computational resources
2. **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)
3. **Off-policy training**: training on a distribution of transitions other than that produced by the target policy
4. See details at Textbook Chapter 11.3

Convergence of Control Methods

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	$\surd$	$(\surd)$	X
Sarsa	$\surd$	$(\surd)$	X
Q-Learning	$\surd$	X	X

$(\surd)$ moves around the near-optimal value function

Batch Reinforcement Learning

1. Incremental gradient descend update is simple
2. But it is not sample efficient
3. Batch-based methods seek to find the best fitting value function for a batch of the agent's experience

Least Square Prediction

1. Given the value function approximation $\hat{v}(s, \mathbf{w}) \approx v^{\pi}(s)$
2. The experience $\mathcal{D}$ consisting of $< state, value >$ pairs (may from one episode or many previous episodes)

$$\mathcal{D} = \{ < s_1, v_1^{\pi} >, ..., < s_T, v_T^{\pi} > \}$$

3. Objective: To optimize the parameter $\mathbf{w}$ that best fit all the experience $\mathcal{D}$
4. Least squares algorithms are used to minimize the sum-squared error between $\hat{v}(s_t, \mathbf{w})$ and the target values v_t^{π}

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

Stochastic Gradient Descent with Experience Replay

1. Given the experience consisting of $< state, value >$ pairs

$$\mathcal{D} = \{ < s_1, v_1^\pi >, ..., < s_T, v_T^\pi > \}$$

2. Iterative solution could be to repeat the following two steps
   1. Randomly sample one pair $< state, value >$ from the experience

   $$< s, v^\pi > \sim \mathcal{D}$$

   2. Apply the stochastic gradient descent update

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

3. The solution from the gradient descend method converges to the least squares solution

$$\mathbf{w}^{LS} = \arg\min_{\mathbf{w}} \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2$$

Solving Linear Least Squares Prediction

1. Experience replay finds least squares solution, but it may take many iterations
2. Using linear value function approximation $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w}$, we can solve the least squares solution directly

Solving Linear Least Squares Prediction 2

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

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

2. thus

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

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

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

3. For N features, the matrix inversion complexity is $O(N^3)$

Linear Least Squares Prediction Algorithms

1. Again, we do not know the true values v_t^π
2. In practice, the training data must use noisy or biased samples of v_t^π
   1. LSMC: Least squares Monte-Carlo uses return
      $v_t^\pi \approx G_t$
   2. LSTD: Least squares TD uses TD target
      $v_t^\pi \approx R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$

Linear Least Squares Prediction Algorithms

LSMC

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

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

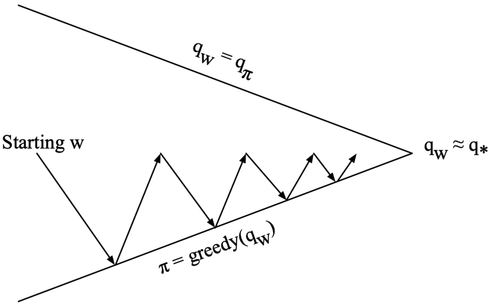
LSTD

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

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

Least Square Control with Value Function Approximation

Generalized policy iteration



1. Policy evaluation: Policy evaluation by least squares Q-learning
2. Policy improvement: greedy policy improvement

Least Squares Action-Value Function Approximation

1. Approximate action-value function $q^{\pi}(s, a)$
2. using linear combination of features $\mathbf{x}(s, a)$

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

3. minimize least squares error between $\hat{q}(s, a, \mathbf{w})$ and $q^{\pi}(s, a)$
4. The experience is generate from using policy π, consisting of $< state, action, action - value >$

Least Squares Control

1. For policy evaluation we want to efficiently use all the experience
2. For control we want to improve the policy
3. The experience is generated from many policies (previous old policies)
4. So to evaluate $q^{\pi}(s, a)$ we must learn off-policy
5. We follow the same idea of Q-Learning:
    1. Use the experience generated by old policy:

$$S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$$

    2. Consider the alternative successor action $A' = \pi_{new}(S_{t+1})$ using greedy
    3. 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

1. Consider the following linear Q-learning update

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

2. LSDQ algorithm: solve the total sum of the gradient = zero:

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

Least Squares Policy Iteration Algorithm

1. Use LSDQ for policy evaluation
2. Repeatedly re-evaluate 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**
            $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a)$
        end for
    until $(\pi \approx \pi')$
    return π
end function

Convergence of Control Methods

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	$\surd$	$(\surd)$	X
Sarsa	$\surd$	$(\surd)$	X
Q-Learning	$\surd$	X	X
LSPI	$\surd$	$(\surd)$	-

$(\surd)$ moves around the near-optimal value function

Deep Q-Learning

1. DeepMind's **Nature** paper: Mnih, Volodymyr; et al. (2015). **Human-level control through deep reinforcement learning**
    1. Entering the period of modern RL: deep learning + reinforcement learning

Review on Stochastic Gradient Descend for Function Approximation

1. Goal: Find the parameter vector $\mathbf{w}$ that minimizes the loss between a true value function $v_\pi(s)$ and its approximation $\hat{v}_\pi(s, \mathbf{w})$ as represented with a particular function approximator parameterized by $\mathbf{w}$
2. The mean square error loss function is as

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

3. Follow the gradient descend to find a local minimum

$$\begin{aligned}\Delta \mathbf{w} =& -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ \mathbf{w}_{t+1} =& \mathbf{w}_t + \Delta \mathbf{w}\end{aligned}$$

Linear Function Approximation

1. Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^T \mathbf{w} = \sum_{j=1}^{n} x_j(S) w_j$$

2. Objective function is $J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \mathbf{x}(S)^T \mathbf{w})^2 \right]$
3. Update is as simple as $\Delta \mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)$
4. But there is no oracle for the true value $v_\pi(S)$, we substitute with the target from MC or TD
   1. for MC policy evaluation,

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

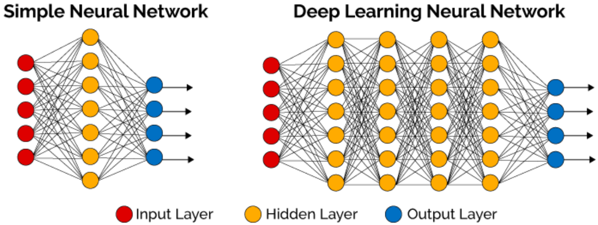
   2. for TD policy evaluation,

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

Linear vs Nonlinear Value Function Approximation

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

Deep Neural Networks

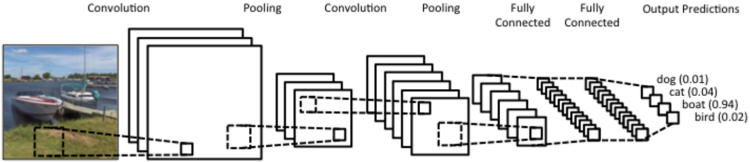


1. Multiple layers of linear functions, with non-linear operators between layers

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

2. 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



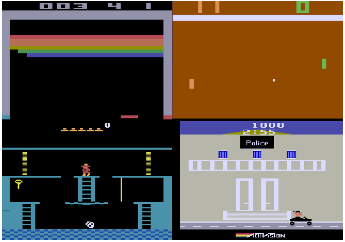
1. Convolution encodes the local information in 2D feature map
2. Layers of convolution, reLU, batch normalization, etc.
3. CNNs are widely used in computer vision (more than 70% top conference papers using CNNs)
4. A detailed introduction on CNNs: `http://cs231n.github.io/convolutional-networks/`

Deep Reinforcement Learning

1. Frontier in machine learning and artificial intelligence
2. Deep neural networks are used to represent
    1. Value function
    2. Policy function (policy gradient methods to be introduced)
    3. World model
3. Loss function is optimized by stochastic gradient descent (SGD)
4. Challenges
    1. Efficiency: too many model parameters to optimize
    2. The Deadly Triad for the danger of instability and divergence in training
        1. Nonlinear function approximation
        2. Bootstrapping
        3. Off-policy training

Deep Q-Networks (DQN)

1. DeepMind's **Nature** paper: Mnih, Volodymyr; et al. (2015). **Human-level control through deep reinforcement learning**
2. DQN represents the action value function with neural network approximator
3. 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

1. Approximate the action-value function

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

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

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

3. 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

1. For MC, the target is the return G_t

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

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

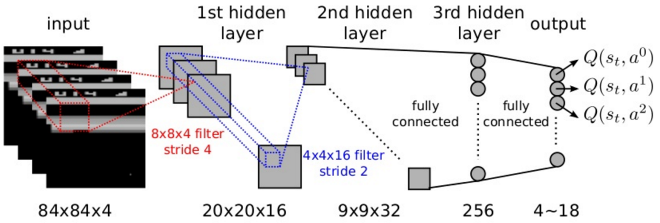
$$\Delta \mathbf{w} = \alpha\big(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w})$$

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

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

DQN for Playing Atari Games

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

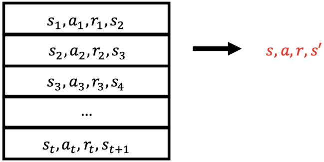


Q-Learning with Value Function Approximation

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

DQNs: Experience Replay

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



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

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

DQNs: Fixed Targets

1. To help improve stability, fix the target weights used in the target calculation for multiple updates
2. 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
3. To perform experience replay with fixed target, repeat the following
    1. sample an experience tuple from the dataset: $(s, a, r, s') \sim \mathcal{D}$
    2. compute the target value for the sampled tuple: $r + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}^-)$
    3. use stochastic gradient descent to update the network weights

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

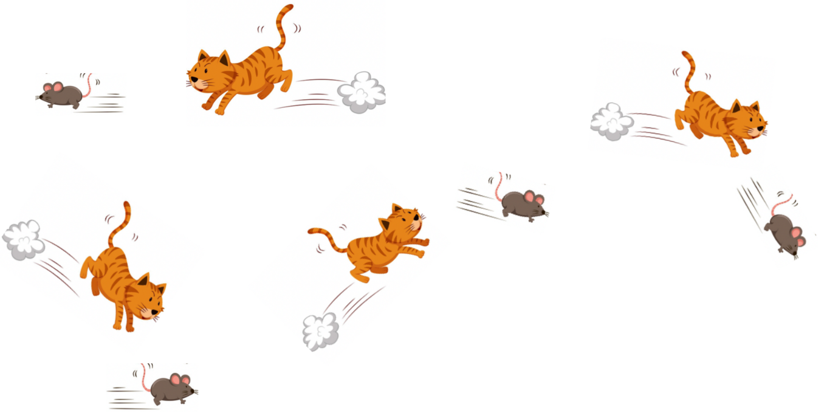
Why fixed target

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

Q target Q estimation

Why fixed target

1. Both the cat and mouse are moving,

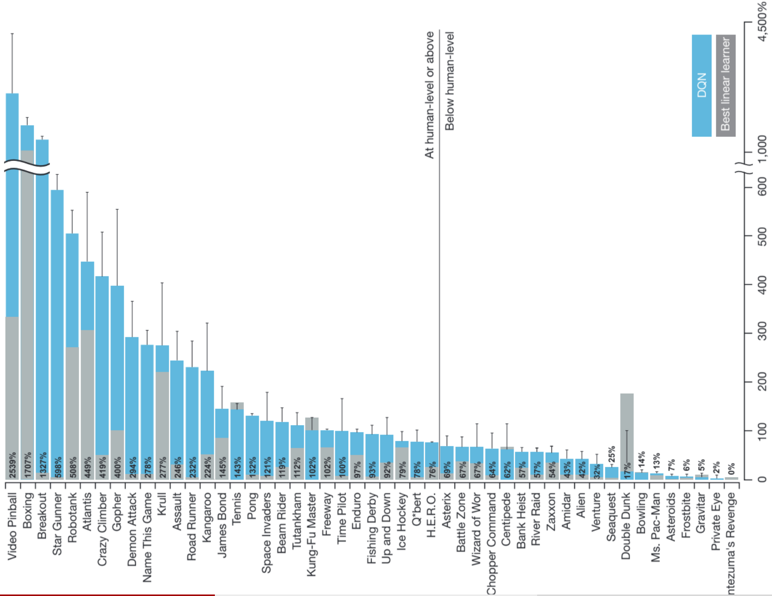


Why fixed target

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

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

Performance of DQNs on Atari



Abalation Study on DQNs

1. 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

1. Demo of deep Q-learning for Breakout: `https://www.youtube.com/watch?v=V1eYniJ0Rnk`
2. Demo of Flappy Bird by DQN: `https://www.youtube.com/watch?v=xM62SpKAZHU`
3. Code of DQN in PyTorch: `https://github.com/cuhkrlcourse/DeepRL-Tutorials/blob/master/01.DQN.ipynb`
4. Code of Flappy Bird: `https://github.com/xmfbit/DQN-FlappyBird`

Summary of DQNs

1. DQN uses experience replay and fixed Q-targets
2. Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
3. Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
4. Compute Q-learning targets w.r.t. old, fixed parameters $\mathbf{w}^-$
5. Optimizes MSE between Q-network and Q-learning targets using stochastic gradient descent

Improving DQN

1. Success in Atari has led to a huge excitement of using deep neural networks for value function approximation in RL
2. Many follow-up works on improving DQNs
   1. **Double DQN**: Deep Reinforcement Learning with Double Q-Learning. Van Hasselt et al, AAAI 2016
   2. **Dueling DQN**: Dueling Network Architectures for Deep Reinforcement Learning. Wang et al, best paper ICML 2016
   3. **Prioritized Replay**: Prioritized Experience Replay. Schaul et al, ICLR 2016
3. A nice tutorial on the relevant algorithms: `https://github.com/cuhkrlcourse/DeepRL-Tutorials`

Improving DQN: Double DQN

1. Handles the problem of the overestimation of Q-values
2. Idea: use the two networks to decouple the action selection from the target Q value generation
3. Vanilla DQN:

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

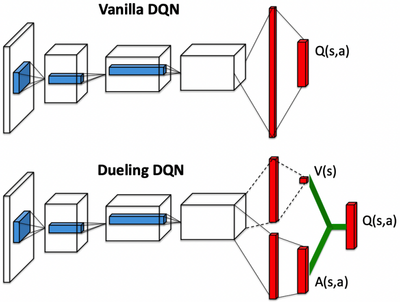
4. Double DQN:

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

5. Code: `https://github.com/cuhkrlcourse/DeepRL-Tutorials/blob/master/03.Double_DQN.ipynb`

Improving DQN: Dueling DQN

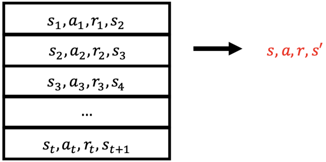
1. One branch estimates $V(s)$, other branch estimates the advantage for each action $A(s, a)$. Then $Q(s, a) = A(s, a) + V(s)$



2. 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
3. Code: `https://github.com/cuhkrlcourse/DeepRL-Tutorials/blob/master/04.Dueling_DQN.ipynb`

Improving DQN: Prioritized Experience Replay

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



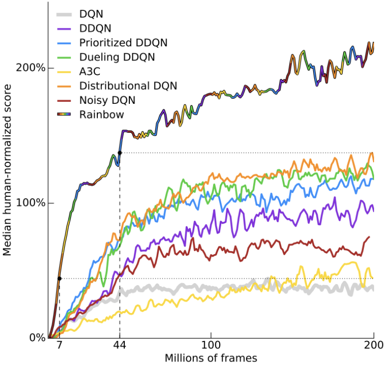
2. Priority is on the experience where there is a big difference between our prediction and the TD target, since it means that we have a lot to learn about it.
3. Define a a priority score for each tuple i

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

4. Code: `https://github.com/cuhkrlcourse/DeepRL-Tutorials/blob/master/06.DQN_PriorityReplay.ipynb`

Improving over the DQN

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



Optional Homework

1. Go through the Jupytor tutorial and training your own gaming agent: `https://github.com/cuhkrlcourse/DeepRL-Tutorials`
2. Good resource for your course project
   1. Make sure it works for simple environment such as Pong
3. Next week: Policy-based RL