CSC415: Introduction to Reinforcement Learning

Lecture 4: Function Approximation and Deep Q-Learning

Dr. Amey Pore

Winter 2026

January 28, 2026

Material taken from Sutton and Barto: Chp 5.2, 5.4, 6.4-6.5, 6.7. Structure adapted from David Silver's and Emma Brunskill's course on Introduction to RL.

Class Structure

- Last lecture:
  - Model-free prediction
  - Model-free Control
- This lecture:
  - How to scale RL

Today's Outline

- **Recall**
- Model Free Value Function Approximation
  - Policy Evaluation
  - Monte Carlo Policy Evaluation
  - Temporal Difference (TD) Policy Evaluation
- Course Logistics
- Control using Value Function Approximation
  - Control using General Value Function Approximation
  - SARSA with Function Approximation
  - Deep Q-Learning

RL Learning Paradigms

Type	Description
On-Policy	Learn to estimate and evaluate a policy from experience obtained from following that policy
Off-Policy	Learn to estimate and evaluate a policy using experience gathered from following a different policy
Online	Agent updates its policy while interacting with the environment in real-time
Offline	Agent learns from a fixed dataset of prior experience without further interaction

SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy TD control algorithm.

SARSA Update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right)$$

Key Characteristics:

- **On-policy:** Learns action-value function for the current policy π
- Uses the **actual action** taken in next state a_{t+1}
- Considers the policy's exploration behavior

SARSA Algorithm

Set initial ϵ-greedy policy π, $t = 0$, initial state $s = s_0$
Take $a_t \sim \pi(s_t)$ // Sample action from policy
Observe (r_t, s_{t+1})
loop
    Take action $a_{t+1} \sim \pi(s_{t+1})$
    Observe (r_t, s_{t+2})
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$
    $\pi(s_t) = \arg\max_a Q(s_t, a)$ w.prob $1 - \epsilon$, else random
    $t = t + 1$
end loop

Q-Learning

Q-Learning is an off-policy TD control algorithm that learns the optimal action-value function Q^* directly.

Q-Learning Update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

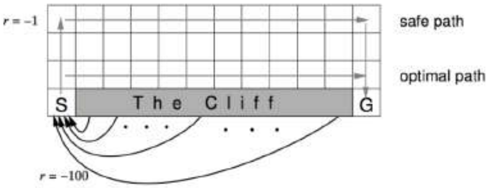
Key Characteristics:

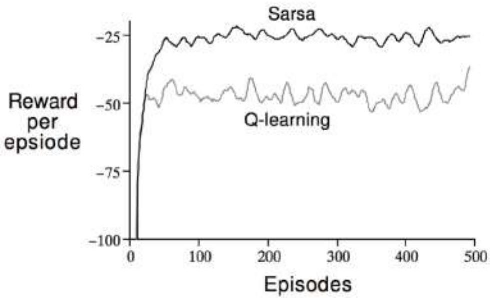
- **Off-policy:** Learns Q^* independent of the policy being followed
- Uses the **best action** in next state: $\max_{a'} Q(s_{t+1}, a')$
- Can learn optimal policy while following exploratory policy (e.g., ϵ-greedy)

Q-Learning Algorithm

Initialize $Q(s,a) \leftarrow 0, \forall s \in \mathcal{S}, a \in \mathcal{A}, t = 0$, initial state $s_t = s_0$
Set π_b to be ϵ-greedy w.r.t. Q
loop
    Take $a_t \sim \pi_b(s_t)$ // Sample action from policy
    Observe (r_t, s_{t+1})
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$
    $\pi(s_t) = \arg\max_a Q(s_t, a)$ w.prob $1 - \epsilon$, else random
    $t = t + 1$
end loop

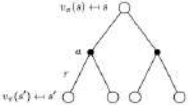
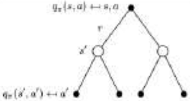
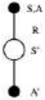
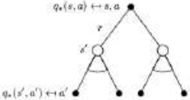
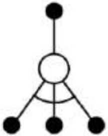
Recall: Cliff Walking Example





- **Q-Learning (Off-policy):** Learns the optimal path along the cliff edge. Falls more often during exploration.
- **SARSA (On-policy):** Learns a safer path away from the edge to account for ϵ-greedy exploration errors.
- Demonstrates difference between learning optimal policy Q^* vs policy being followed Q^π.

Relationship Between DP and TD

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s, a)$	 Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s, a)$	 Q-Value Iteration	 Q-Learning

Relationship Between DP and TD (2)

Full Backup (DP)	*Sample Backup (TD)*
Iterative Policy Evaluation	TD Learning
$V(s_t) \leftarrow \mathbb{E}[r_t + \gamma V(s_{t+1}) \mid s_t]$	$V(s_t) \xleftarrow{\alpha} r_t + \gamma V(s_{t+1})$
Q-Policy Iteration	Sarsa
$Q(s_t, a_t) \leftarrow \mathbb{E}[r_t + \gamma Q(s_{t+1}, a_{t+1}) \mid s_t, a_t]$	$Q(s_t, a_t) \xleftarrow{\alpha} r_t + \gamma Q(s_{t+1}, a_{t+1})$
Q-Value Iteration	Q-Learning
$Q(s_t, a_t) \leftarrow \mathbb{E}\left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') \mid s_t, a_t\right]$	$Q(s_t, a_t) \xleftarrow{\alpha} r_t + \gamma \max_{a'} Q(s_{t+1}, a')$

where $x \xleftarrow{\alpha} y \equiv x \leftarrow x + \alpha(y - x)$

Think Pair wise

Q1: Convergence to Q^*

Which of the following conditions are sufficient to ensure that Q-learning eventually learns the optimal action-value function Q^*, even if the agent is using ϵ-greedy exploration? (Select all that apply)

A) The exploration rate ϵ must eventually decay to zero.

B) Every state-action pair (s, a) is visited an infinite number of times.

C) The learning rate α satisfies the Robbins-Monro conditions.

D) The agent must follow the optimal policy π^* at all times during training.

Q2: Convergence to Optimal Policy π^* in Cliff Walking

In a gridworld like Cliff Walking, what must happen for an ϵ-greedy Q-learning agent to eventually converge to the optimal policy π^* (the shortest path)? (Select all that apply)

A) The agent must meet the GLIE (Greedy in the Limit with Infinite Exploration) conditions.

B) The exploration rate ϵ must be held at a constant non-zero value (e.g., $\epsilon = 0.1$).

C) The exploration rate ϵ_t must approach zero as the number of episodes $t \to \infty$.

D) The agent must switch to an on-policy algorithm like Sarsa.

Today's Outline

- Recall
- **Model Free Value Function Approximation**
  - Policy Evaluation
  - Monte Carlo Policy Evaluation
  - Temporal Difference (TD) Policy Evaluation
- Course Logistics
- Control using Value Function Approximation
  - Control using General Value Function Approximation
  - SARSA with Function Approximation
  - Deep Q-Learning

Limitations of Tabular Q-Learning

Challenges with Large MDPs

- **Memory:** Too many states to store. $Q(s, a)$ for every state-action pair (e.g., Atari: $256^{84\times 84}$ states, Chess: $\approx 10^{120}$ states)
- **Generalization:** Can't generalize to unseen states
- **Sample efficiency:** Need to visit every state-action pair many times
- **Continuous states:** Impossible to enumerate all states

Desired Properties: Want more compact representation that generalizes across state or states and actions:

- Reduce memory needed to store $(P, R)/V/Q/\pi$
- Reduce computation needed to compute $(P, R)/V/Q/\pi$
- Reduce experience needed to find a good $(P, R)/V/Q/\pi$

Value Function Approximation

Solution: Use function approximation to estimate value function

Function Approximation

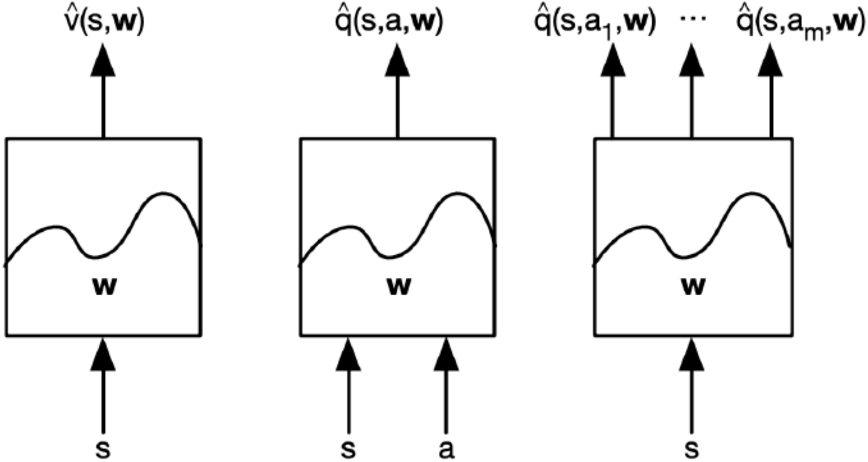
Instead of storing $V(s)$ or $Q(s,a)$ for each state/state-action pair, we approximate using a parameterized function:

$$\hat{V}(s; \mathbf{w}) \approx V^{\pi}(s), \quad \hat{Q}(s, a; \mathbf{w}) \approx Q^{\pi}(s, a)$$

where $\mathbf{w}$ are parameters (e.g., weights in neural network, linear function approximator)

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

Types of Value Function Approximation



Which Function Approximator?

We can approximate value functions using many different function approximators:

- Linear Combinations of Features
- Neural Network
- Decision Tree
- Nearest Neighbors
- ....

Which Function Approximator to choose?

We need to choose a function approximator based on:

- **State space:** Discrete vs continuous, low vs high dimensional
- **Differentiable:** Need gradients for gradient descent?
- **Interpretability:** Do we need to understand the function?
- **Convergence:** Does it converge to optimal solution?

State-Action Value Function Approximation for Policy Evaluation with an Oracle

- First assume we could query any state s and action a and an oracle would return the true value for $Q^\pi(s, a)$
- Similar to supervised learning: assume given $((s, a), Q^\pi(s, a))$ pairs
- The objective is to find the best approximate representation of Q^π given a particular parameterized function $\hat{Q}(s, a; \mathbf{w})$

Gradient Descent

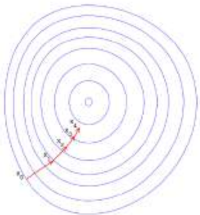
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$
- Define the gradient of $J(\mathbf{w})$ to be

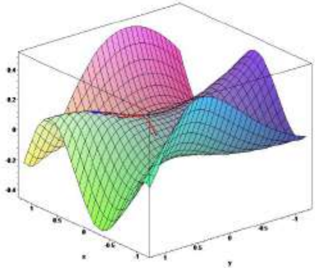
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{pmatrix}$$

- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of -ve 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 Stochastic Gradient Descent

- Goal: Find the parameter vector **w** that minimizes the loss between a true value function $Q^\pi(s,a)$ and its approximation $\hat{Q}(s,a;\mathbf{w})$.
- Generally use mean squared error and define the loss as

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

- Can use gradient descent to find a local minimum

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

- Stochastic gradient descent (SGD) uses a finite number of (often one) samples to compute an approximate gradient:

$$\begin{aligned}\nabla_\mathbf{w} J(\mathbf{w}) &= \nabla_\mathbf{w}\mathbb{E}_\pi[Q^\pi(s,a) - \hat{Q}(s,a;\mathbf{w})]^2 \\ &= -2\mathbb{E}_\pi[(Q^\pi(s,a) - \hat{Q}(s,a;\mathbf{w}))\nabla_\mathbf{w}\hat{Q}(s,a,\mathbf{w})]\end{aligned}$$

- Expected SGD is the same as the 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 $\mathbf{w}$

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

- 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

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

Model Free VFA Policy Evaluation

- No oracle to tell true $Q^\pi(s, a)$ for any state s and action a
- Recall model-free policy evaluation (Lecture 3)
  - Following a fixed policy π (or had access to prior data)
  - Goal is to estimate V^π and/or Q^π
- Maintained a lookup table to store estimates V^π and/or Q^π
- Updated these estimates after each episode (Monte Carlo methods) or after each step (TD methods)
- **Now: in value function approximation, change the estimate update step to include fitting the function approximator**

Monte Carlo Value Function Approximation

- Return G_t is an unbiased but noisy sample of the true expected return $Q^{\pi}(s_t, a_t)$
- Therefore can reduce MC VFA to doing supervised learning on a set of (state, action, return) pairs:

$$\langle (s_1, a_1), G_1 \rangle, \langle (s_2, a_2), G_2 \rangle, \ldots, \langle (s_T, a_T), G_T \rangle$$

  - Substitute G_t for the true $Q^{\pi}(s_t, a_t)$ when fit function approximator

MC Value Function Approximation for Policy Evaluation

Initialize $\mathbf{w}$, $k = 1$
loop
  Sample k-th episode $(s_{k,1}, a_{k,1}, r_{k,1}, s_{k,2}, \ldots, s_{k,L_k})$ given π
  for $t = 1, \ldots, L_k$ **do**
    if First visit to (s, a) in episode k **then**
      $G_t(s,a) = \sum_{j=t}^{L_k} r_{k,j}$
      $\nabla_{\boldsymbol{w}} J(w) = -2[G_t(s,a) - \hat{Q}(s_t, a_t; w)] \nabla_{\boldsymbol{w}} \hat{Q}(s_t, a_t; w)$ (Compute Gradient)
      Update weights $\Delta \boldsymbol{w}$
    end if
  end for
  $k = k + 1$
end loop

Recall: Temporal Difference Learning w/ Lookup Table

- Uses bootstrapping and sampling to approximate V^π
- Updates $V^\pi(s)$ after each transition (s, a, r, s'):

$$V^\pi(s) = V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s))$$

- Target is $r + \gamma V^\pi(s')$, a biased estimate of the true value $V^\pi(s)$
- Represent value for each state with a separate table entry
- Note: Unlike MC we will focus on V instead of Q for policy evaluation here, because there are more ways to create TD targets from Q values than V values

Temporal Difference TD(0) Learning with Value Function Approximation

- Uses bootstrapping and sampling to approximate true V^π
- Updates estimate $V^\pi(s)$ after each transition (s, a, r, s'):

$$V^\pi(s) = V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s))$$

- Target is $r + \gamma V^\pi(s')$, a biased estimate of the true value $V^\pi(s)$
- In value function approximation, target is $r + \gamma V^\pi(s'; \mathbf{w})$, a biased and approximated estimate of the true value $V^\pi(s)$
- 3 forms of approximation:
  1. Sampling
  2. Bootstrapping
  3. Value function approximation

Temporal Difference TD(0) Learning with Value Function Approximation

- In value function approximation, target is $r + \gamma \hat{V}^{\pi}(s'; \mathbf{w})$, a biased and approximated estimate of the true value $V^{\pi}(s)$
- Can reduce doing TD(0) learning with value function approximation to supervised learning on a set of data pairs:
  - $(s_1, r_1 + \gamma \hat{V}^{\pi}(s_2; \mathbf{w}))$, $(s_2, r_2 + \gamma \hat{V}^{\pi}(s_3; \mathbf{w}))$, $\ldots$
- Find weights to minimize mean squared error

$$J(\mathbf{w}) = \mathbb{E}_{\pi}[(r_j + \gamma \hat{V}^{\pi}(s_{j+1}; \mathbf{w}) - \hat{V}(s_j; \mathbf{w}))^2]$$

- Use stochastic gradient descent, as in MC methods

TD(0) Value Function Approximation for Policy Evaluation

Initialize $\mathbf{w}, \mathbf{s}$

loop

  Given s sample $a \sim \pi(s)$, $r(s,a)$,$s' \sim p(s'|s,a)$

  $\nabla_{\boldsymbol{w}} J(w) = -2[r + \gamma \hat{V}(s'; w) - \hat{V}(s; w)] \nabla_{\boldsymbol{w}} \hat{V}(s; w)$

  Update weights $\Delta \boldsymbol{w}$

  if s' is not a terminal state **then**

    Set $s = s'$

  else

    Restart episode, sample initial state s

  end if

end loop

Convergence of Prediction Algorithms

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

Today's Outline

- Recall
- Model Free Value Function Approximation
  - Policy Evaluation
  - Monte Carlo Policy Evaluation
  - Temporal Difference (TD) Policy Evaluation
- **Course Logistics**
- Control using Value Function Approximation
  - Control using General Value Function Approximation
  - SARSA with Function Approximation
  - Deep Q-Learning

Course Logistics

- Tomorrow's Mid-term will be held in DH2080: 90 mins.
- Assignment 1 is out. Due Feb 13th
- Project topics are updated.

Groups

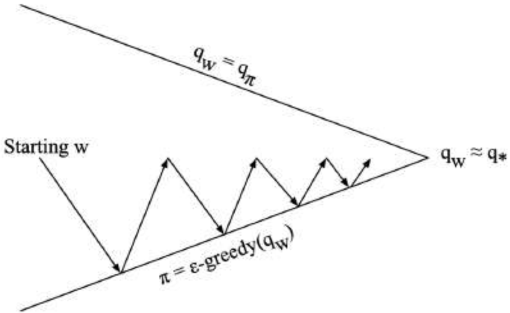
- Groups are created on Quercus. You can self-assign.
- If you have already formed groups, you can strategically choose the papers to review for A1.
- Project topics are updated.
- Groups are created on Quercus. You can self-assign.
- If you have already formed groups, you can strategically choose the papers to review for A1.

Break

Today's Outline

- Recall
- Model Free Value Function Approximation
    - Policy Evaluation
    - Monte Carlo Policy Evaluation
    - Temporal Difference (TD) Policy Evaluation
- Course Logistics
- **Control using Value Function Approximation**
    - Control using General Value Function Approximation
    - SARSA with Function Approximation
    - Deep Q-Learning

Control with Value Function Approximation



- **Policy evaluation** Approximate policy evaluation, $\hat{Q}^\pi(s, a; \mathbf{w}) \approx Q^\pi$
- **Policy improvement** ϵ-greedy policy improvement

Action-Value Function Approximation with an Oracle

- $\hat{Q}^\pi(s, a; \mathbf{w}) \approx Q^\pi$
- Minimize the mean-squared error between the true action-value function $Q^\pi(s, a)$ and the approximate action-value function:

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

- Use stochastic gradient descent to find a local minimum

$$\nabla_\mathbf{w} J(\mathbf{w}) = -2\mathbb{E}\left[(Q^\pi(s, a) - \hat{Q}^\pi(s, a; \mathbf{w}))\nabla_\mathbf{w}\hat{Q}^\pi(s, a; \mathbf{w})\right]$$

- Stochastic gradient descent (SGD) samples the gradient

Incremental Model-Free Control Approaches

- Similar to policy evaluation, true state-action value function for a state is unknown and so substitute a target value for true $Q(s_t, a_t)$

$$\Delta \mathbf{w} = \alpha(Q(s_t, a_t) - \hat{Q}(s_t, a_t; \mathbf{w}))\nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

- In Monte Carlo methods, use a return G_t as a substitute target

$$\Delta \mathbf{w} = \alpha(G_t - \hat{Q}(s_t, a_t; \mathbf{w}))\nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

- SARSA: Use TD target $r + \gamma \hat{Q}(s', a'; \mathbf{w})$ which leverages the current function approximation value

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

- Q-learning: Uses related TD target $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w})$

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

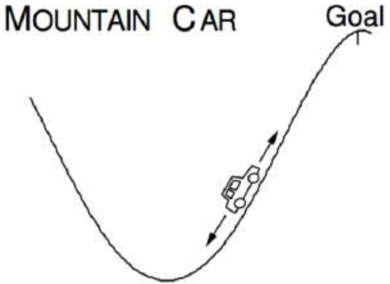
"Deadly Triad" which Can Cause Instability

- Informally, updates involve doing an (approximate) Bellman backup followed by best trying to fit underlying value function to a particular feature representation
- Bellman operators are contractions, but value function approximation fitting can be an expansion
  - To learn more, see Baird example in Sutton and Barto 2018
- "Deadly Triad" can lead to oscillations or lack of convergence
  - Bootstrapping
  - Function Approximation
  - Off policy learning (e.g. Q-learning)

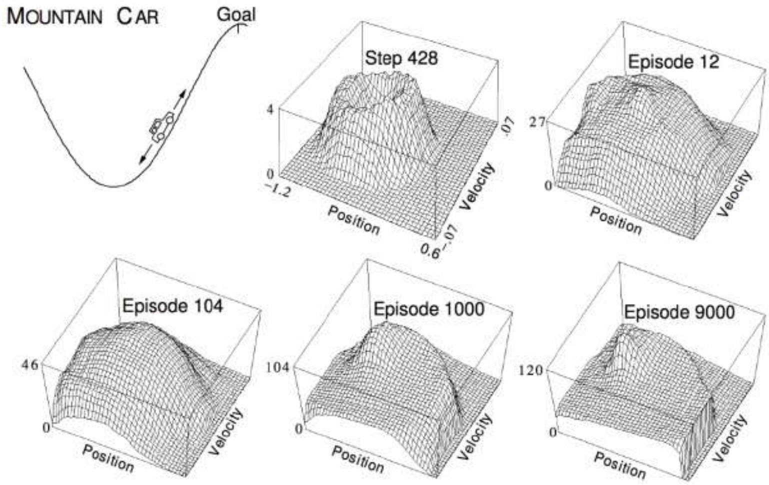
Example: Mountain Car

Mountain Car Problem

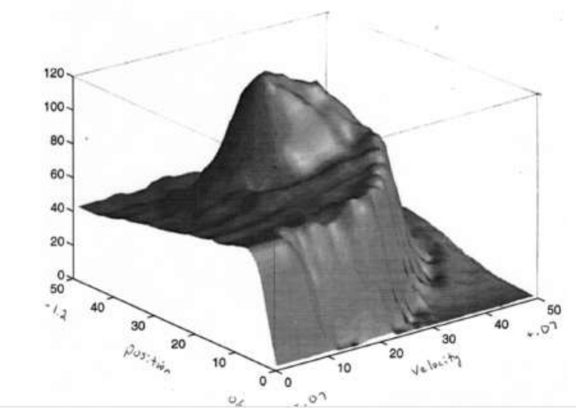
- Car stuck in valley between two hills
- Goal: Reach the top of the right hill
- State: Position and velocity
- Actions: Accelerate left, coast, accelerate right



Linear SARSA in Mountain Car



Linear Sarsa with Radial Basis Functions in Mountain Car



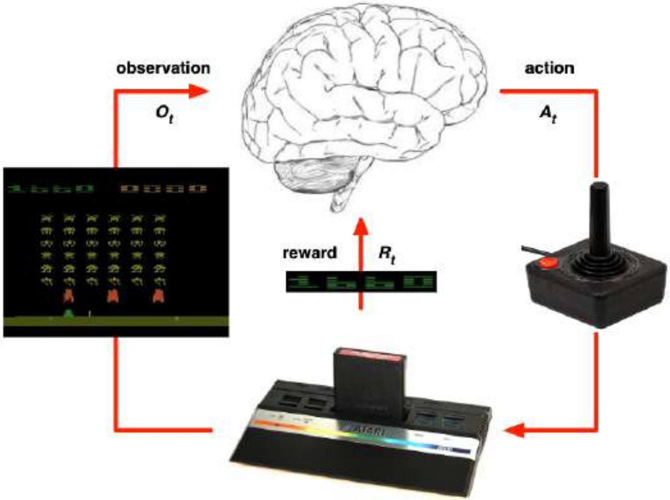
https://github.com/Ameyapores/MountainCar-SARSA

Convergence of Control Algorithms

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

(✓) = chatters around near-optimal value function

Using these ideas to do Deep RL in Atari



Q-Learning with Neural Networks

- Q-learning converges to optimal $Q^*(s, a)$ using tabular representation
- In value function approximation Q-learning minimizes MSE loss by stochastic gradient descent using a target Q estimate instead of true Q
- But Q-learning with VFA can diverge
- Two of the issues causing problems:
  - Correlations between samples
  - Non-stationary targets
- Deep Q-learning (DQN) addresses these challenges by using
  - Experience replay
  - Fixed Q-targets

DQNs: Experience Replay

- To help remove correlations, store dataset (called a **replay buffer**) $\mathcal{D}$ from prior experience

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
$\ldots$
$s_t, a_t, r_{t+1}, s_{t+1}$

$\rightarrow$ s, a, r, s'

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

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

- **Uses target as a scalar, but function weights will get updated on the next round, changing the target value**

DQNs: Fixed Q-Targets

- To help improve stability, fix the **target weights** used in the target calculation for multiple updates
- Target network uses a different set of weights than the weights being updated
- Let parameters $\mathbf{w}^-$ be the set of weights used in the target, and $\mathbf{w}$ be the weights that are being updated
- Slight change to computation of target value:
  - $(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
  - Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^-)$
  - Use stochastic gradient descent to update the network weights

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

DQN Pseudocode

Input: $E, \alpha, s, a, r, s' \sim \pi$; **Initialize** $\mathcal{D} = \emptyset$, $\mathbf{w} = 0$
Set other state $\mathbf{w}_0$
for episode $= 1, \dots, E$ **do do**
    Initialize s_1
    for $t = 1, \dots, T$ **do do**
        Observe reward r_t and next state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
        for $i = 1, \dots, K$ **do do**
            Sample random minibatch of transitions (s, a, r, s') from $\mathcal{D}$
            if s_{t+1} is terminal at step $t + 1$ **then then**
                Set $y_t = r_t$
            else
                Set $y_t = r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \mathbf{w}^-)$
            end if
            Perform gradient descent step on $(y_t - \hat{Q}(s_t, a_t; \mathbf{w}))^2$ w.r.t. $\mathbf{w}$
        end for
        Every C steps: $\mathbf{w}^- = \mathbf{w}$
    end for
end for

Note: There are several hyperparameters and algorithm choices. One needs to choose the neural network architecture, the learning rate, how often to update the target network. Often a minibatch buffer is used, not just for experience replay, but also to do batch updates of network weights. This is because a key benefit of neural network architectures is a parameter is updated the cost of passing a mini-batch through the network is about the same as for one sample.

Check Your Understanding L4N3: Fixed Targets

- In DQN we compute the target value for the sampled (s, a, r, s') using a separate set of target weights: $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^{-})$
- Select all that are true
  - This doubles the computation time compared to a method that does not have a separate set of weights
  - This doubles the memory requirements compared to a method that does not have a separate set of weights
  - Not sure

DQNs Summary

- 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
- Uses stochastic gradient descent

DQNs in Atari

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

DQN

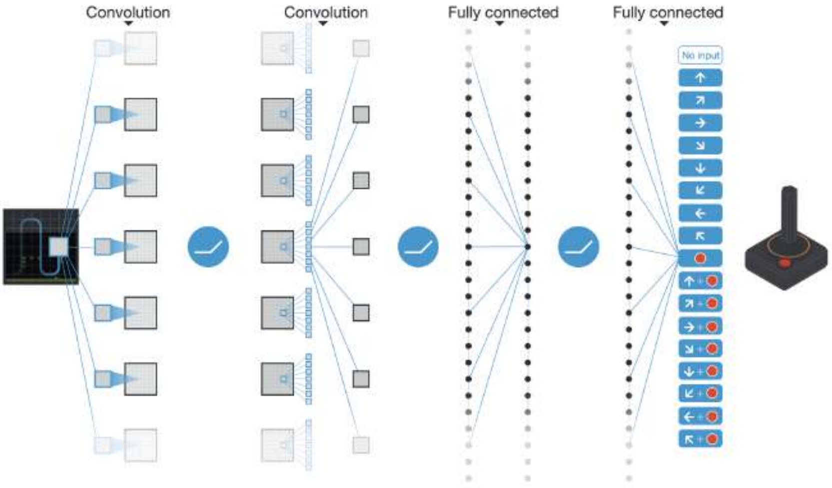


Figure: Human-level control through deep reinforcement learning. Mnih et al, 2015

DQN Results in Atari

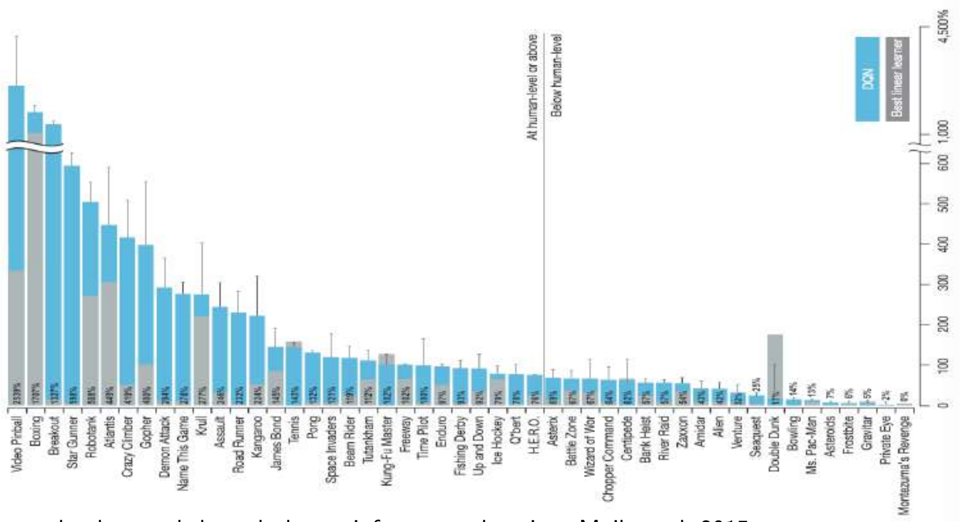


Figure: Human-level control through deep reinforcement learning. Mnih et al, 2015

Which Aspects of DQN were Important for Success?

Game	Linear	Deep Network
Breakout	3	3
Enduro	62	29
River Raid	2345	1453
Seaquest	656	275
Space Invaders	301	302

Note: just using a deep NN actually hurt performance sometimes!

Which Aspects of DQN were Important for Success?

Game	Linear	Deep Network	DQN w/ fixed Q
Breakout	3	3	10
Enduro	62	29	141
River Raid	2345	1453	2868
Seaquest	656	275	1003
Space Invaders	301	302	373

Which Aspects of DQN were Important for Success?

Game	Linear	Deep Network	DQN w/ fixed Q	DQN w/ replay	DQN w/replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Seaquest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

- Replay is **hugely** important
- Why? Beyond helping with correlation between samples, what does replaying do?

Deep RL

- Success in Atari has led to huge excitement in using deep neural networks to do value function approximation in RL
- Some immediate improvements (many others!)
  - **Double DQN** (Deep Reinforcement Learning with Double Q-Learning, Hasselt et al, AAAI 2016)
  - **Prioritized Replay** (Prioritized Experience Replay, Schaul et al, ICLR 2016)
  - **Dueling DQN** (best paper ICML 2016) (Dueling Network Architectures for Deep Reinforcement Learning, Wang et al)

What You Should Understand (for mid-term)

- Be able to implement Policy Iteration and Value Iteration.
- Be able to implement TD(0) and MC on policy evaluation
- Be able to implement Q-learning and SARSA and MC control algorithms
- Know about MDP structure
- Key features in DQN and function approximation that are critical.

Thank You!

Questions?