Generalization

Reinforcement Learning

Computer Engineering Department
Sharif University of Technology

States are continuous or there are infinitely many of them.

Mohammad Hossein Rohban, Ph.D.

Q(s,a)

feature Representation

$\Phi(s)$

inp.

(s,a)

Spring 2024

Courtesy: Some slides are adopted from CS 285 Berkeley, and CS 234 Stanford, and Pieter Abbeel's compact series on RL.

$\pi = \epsilon\text{-greedy } \hat{Q}_w(s, a)$

Replay Buffer

$S_0, a_0, r_0, S_1, a_1, r_1 \ldots$

training sample

Function Approximation

MC

$(\phi(s_0, a_0), G_0)$

$(\phi(S_1, a_1), G_1)$

$\vdots$

TD for the network

$(\phi(s_0, a_0), r_0 + \gamma \max_a \hat{Q}_w(s_1, a))$

$(\phi(s, a_1), r_0 + \gamma \hat{Q}_w(s_1, a_1))$

$\vdots$

$\hat{Q}_{w^-}(s, a)$

Function approximation and deep RL

- The policy, value function, model, and agent state update are all functions
- We want to learn these from experience (data).
- If there are too many states, we need to approximate.
- This is often called deep reinforcement learning, when using neural networks to represent these functions.

Large-Scale Reinforcement Learning

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

  - Backgammon: 10^{20} states
  - Go: 10^{170} states
  - Helicopter: continuous state space
  - Robots: real world

- How can we apply our methods for prediction and control?

Value Function Approximation

Value Function Approximation

- So far we mostly considered lookup tables
  - 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
  - Individual environment states are often not fully observable

Value Function Approximation

- Solution for large MDPs:
  - Estimate value function with function approximation

$$
\begin{aligned}
v_{\mathbf{w}}(s) &\approx v_\pi(s) && \text{(or } v_*(s)\text{)} \\
q_{\mathbf{w}}(s,a) &\approx q_\pi(s,a) && \text{(or } q_*(s,a)\text{)}
\end{aligned}
$$

  - Update parameter w (e.g., using MC or TD learning)
  - Generalize to unseen states

Agent state

- When the environment state is not fully observable ($S_t^{env} \neq O_t$)
- Use the agent state: (with parameters ω)

$$\mathbf{s}_t = u_\omega(\mathbf{s}_{t-1}, A_{t-1}, O_t)$$

- Henceforth, S_t or s_t denotes the agent state
- Think of this as either a vector inside the agent, or, in the simplest case, just the current observation: $S_t = O_t$

Function Classes

Classes of Function Approximation

- Tabular: a table with an entry for each MDP state
- State aggregation: Partition environment states (or observations) into a discrete set
- Linear function approximation
  - Consider fixed agent state update (e.g. $S_t = O_t$)
  - Fixed feature map $x: \mathcal{S} \to \mathbb{R}^n$
  - Values are linear function of features: $v_w(s) = w^T x(s)$
  - Note: state aggregation and tabular are special cases of linear FA
- Differentiable function approximation
  - $v_w(s)$ is a differentiable function of w, could be non-linear
  - E.g., a convolutional neural network that takes pixels as input
  - Another interpretation: features are not fixed, but learnt

Classes of Function Approximation

- In principle, any function approximator can be used, but RL has specific properties:

  - Experience is not iid — successive time-steps are correlated
  - Agent's policy affects the data it receives
  - Regression targets can be non-stationary
    - ...because of changing policies (which can change the target and the data!)
    - ...because of bootstrapping
    - ...because of non-stationary dynamics (e.g., other learning agents)
    - ...because the world is large (never quite in the same state)

Classes of Function Approximation

- Which function approximation should you choose? This depends on your goals.

  - Tabular: good theory but does not scale/generalize
  - Linear: reasonably good theory, but requires good features
  - Non-linear: less well-understood, but scales well. Flexible, and less reliant on picking good features first (e.g., by hand)
  - (Deep) neural nets often perform quite well, and remain a popular choice

Gradient-based Algorithms

Approximate Values By Stochastic Gradient Descent

- Goal: find w that minimize the difference between $v_w(s)$ and $v_\pi(s)$

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

- Gradient descent:

$$\Delta\mathbf{w} = -\frac{1}{2}\alpha\nabla_\mathbf{w}J(\mathbf{w}) = \alpha\mathbb{E}_d(v_\pi(S) - v_\mathbf{w}(S))\nabla_\mathbf{w}v_\mathbf{w}(S)$$

- Stochastic gradient descent (SGD), sample the gradient:

$$\Delta\mathbf{w} = \alpha(G_t - v_\mathbf{w}(S_t))\nabla_\mathbf{w}v_\mathbf{w}(S_t)$$

Note: Monte Carlo return G_t is a sample for $v_\pi(s_t)$

Linear function approximation

Feature Vectors

- Represent state by a feature vector

$$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

- $x: \mathcal{S} \to \mathbb{R}^n$ is a fixed mapping from state (e.g., observation) to features
- Short-hand: $x_t = x(S_t)$
- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

Linear value function approximation

- Approximate value function by a linear combination of features

$$v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{j=1}^{n} \mathbf{x}_j(s)\mathbf{w}_j$$

- Objective function (‘loss‘) is quadratic in w

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

- Stochastic gradient descent converges to the global optimum
- Update rule is simple

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \mathbf{x}(S_t) = \mathbf{x}_t \qquad \Longrightarrow \qquad \Delta\mathbf{w} = \alpha(v_\pi(S_t) - v_{\mathbf{w}}(S_t))\mathbf{x}_t$$

Prediction algorithms

- We can't update towards the true value function $v_\pi(s)$
- We substitute a target for $v_\pi(s)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w}_t = \alpha(G_t - v_{\mathbf{w}}(s))\nabla_{\mathbf{w}} v_{\mathbf{w}}(s)$$

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

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

  - TD(λ):

$$\Delta \mathbf{w}_t = \alpha(G_t^\lambda - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

$$G_t^\lambda = R_{t+1} + \gamma\Big((1-\lambda)v_{\mathbf{w}}(S_{t+1}) + \lambda G_{t+1}^\lambda\Big)$$

Monte-Carlo with Value Function Approximation

- The return G_t is an unbiased sample of $v_\pi(s)$
- Can therefore apply "supervised learning" to (online) "training data":

$$\{(S_0, G_0), \ldots, (S_t, G_t)\}$$

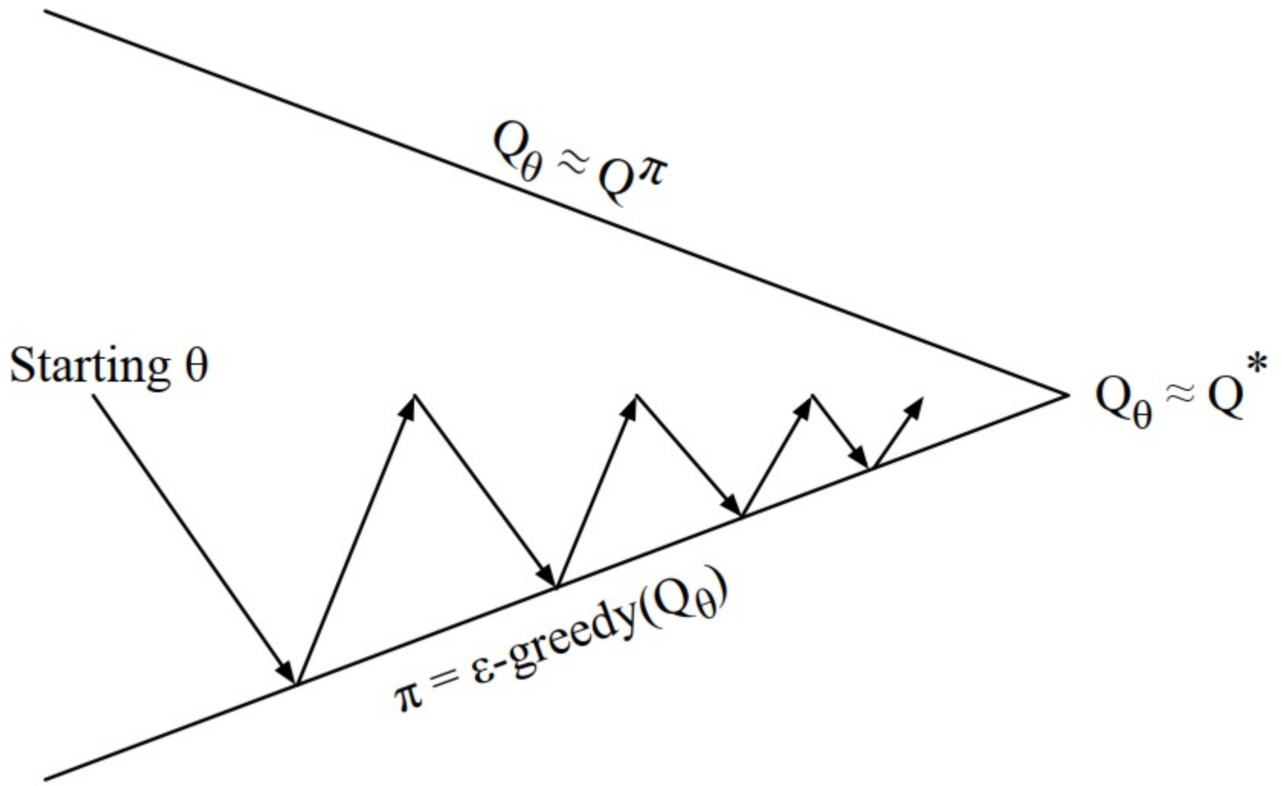
- For example, using linear Monte-Carlo policy evaluation

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

- Linear Monte-Carlo evaluation converges to the global optimum
- Even when using non-linear value function approximation it converges (but perhaps to a local optimum)

Control with value-function approximation

Control with Value Function Approximation



Policy evaluation **Approximate** policy evaluation, $q_{\mathbf{w}} \approx q_\pi$

Policy improvement E.g., ϵ-greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function $q_w(s, a) \approx q_\pi(s, a)$
- For instance, with linear function approximation with state-action features

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

- Stochastic gradient descent update

$$\begin{aligned} \Delta\mathbf{w} &= \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a))\nabla_{\mathbf{w}} q_{\mathbf{w}}(s, a) \\ &= \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a))\mathbf{x}(s, a) \end{aligned}$$

Action-Value Function Approximation (Alternative)

- Approximate the action-value function $q_w(s,a) \;\approx\; q_\pi(s,a)$
- For instance, with linear function approximation with state features

$$\mathbf{q}_{\mathbf{w}}(s) = \mathbf{W}\mathbf{x}(s) \qquad (\mathbf{W} \in \mathbb{R}^{m\times n},\ \mathbf{x}(s) \in \mathbb{R}^n \implies \mathbf{q} \in \mathbb{R}^m)$$

$$q_{\mathbf{w}}(s,a) = \mathbf{q}_{\mathbf{w}}(s)[a] = \mathbf{x}(s)^\top \mathbf{w}_a \qquad (\text{where } \mathbf{w}_a = \mathbf{W}_{a\cdot}^\top)$$

Action-Value Function Approximation

- Should we use action-in, or action-out?
  - Action in: $q_w(s, a) = w^T x(s, a)$
  - Action out: $q_w(s) = Wx(s)$ such that $q_w(s, a) = q_w(s)[a]$
- One reuses the same weights, the other the same features
- Unclear which is better in general
- If we want to use continuous actions, action-in is easier
- For (small) discrete action spaces, action-out is common (e.g., DQN)

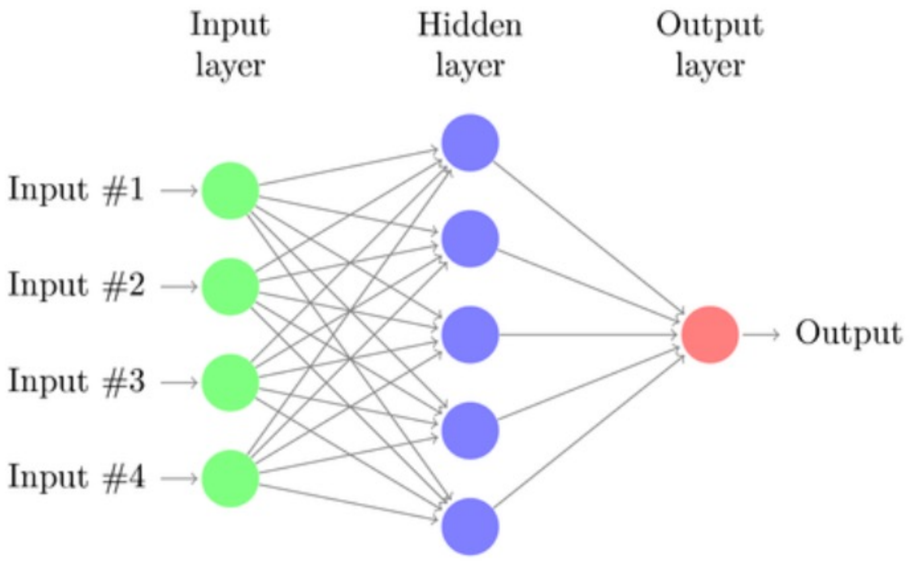
Deep reinforcement learning

RL with Function Approximation

- Linear value function approximators assume value function is a weighted combination of a set of features, where each feature a function of the state
- Linear VFA often works well given the right set of features
- But can require carefully hand designing that feature set
- An alternative is to use a much richer function approximation class that is able to directly go from states without requiring an explicit specification of features

Neural Networks as Function Approximators

- It is possible to use deep neural networks for function approximations.
- Deep networks are clearly more powerful and can model more complex environments.



Generalization

- Using function approximation to help scale up to making decisions in really large domains

Deep Reinforcement Learning

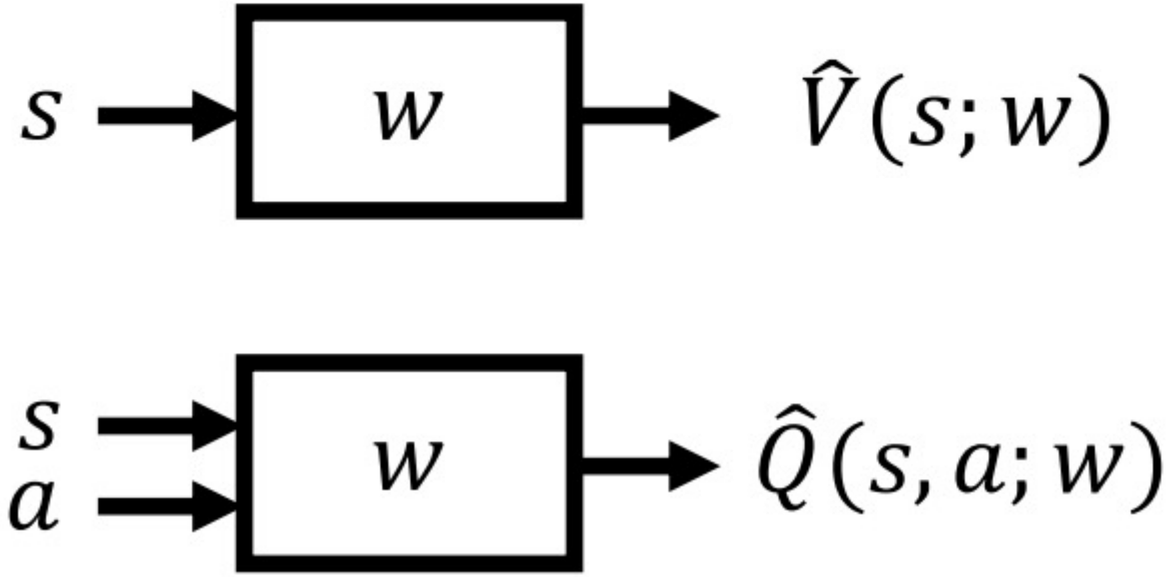
- Use deep neural networks to represent
  - Value, Q function
  - Policy
  - Model

- Optimize loss function by stochastic gradient descent (SGD)

Deep Q-Networks (DQN)

- Represent state-action value function by Q-network with weights w

$$\hat{Q}(s, a; \mathbf{w}) \approx Q(s, a)$$



Recall: Model free control

- Similar to policy evaluation, true state-action value function for a state is unknown and so substitute a target value
- In Monte Carlo methods, use a return G_t as a substitute target

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

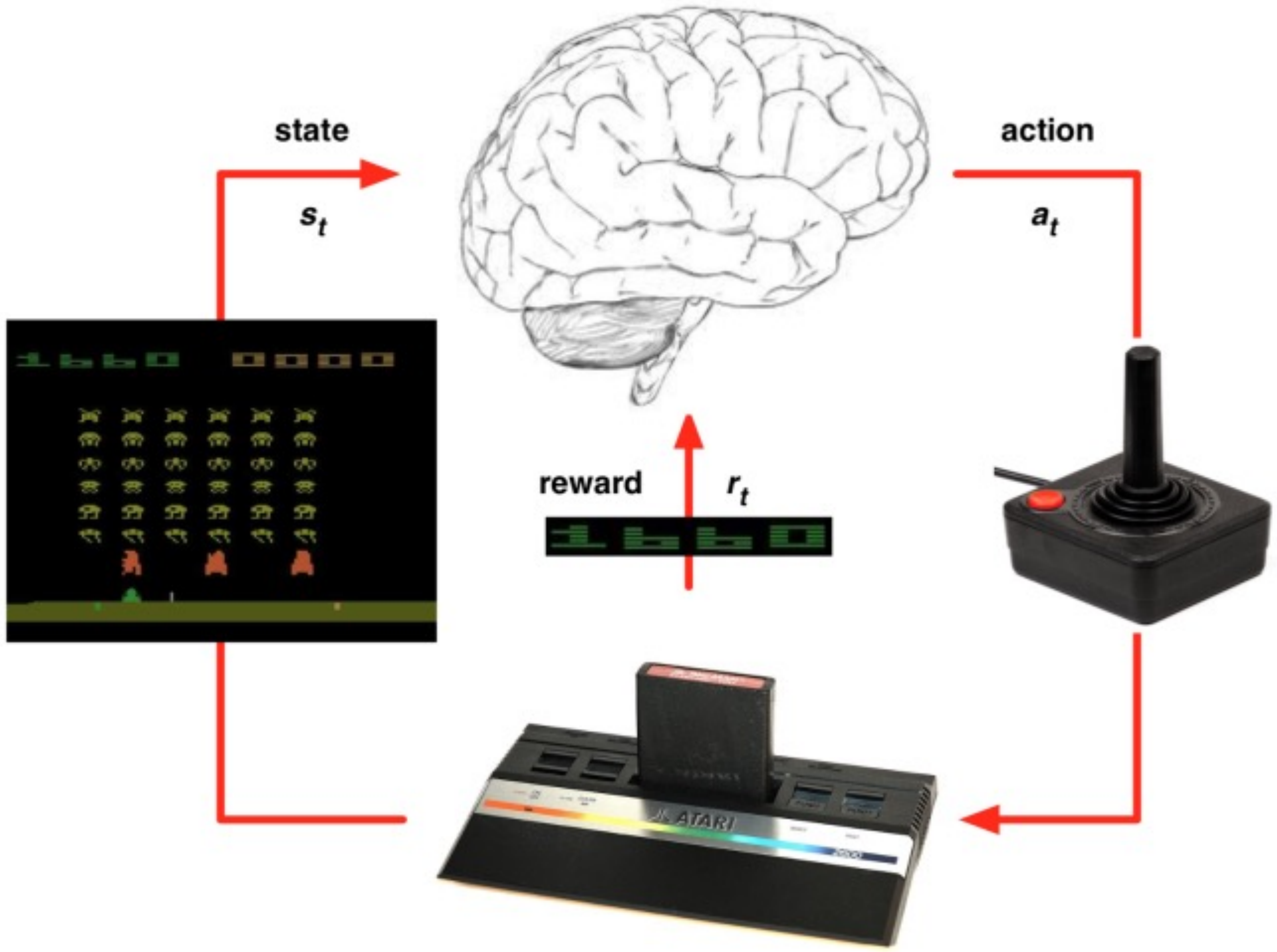
- For SARSA instead use a TD target

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \boldsymbol{w}) - \hat{Q}(s_t, a_t; \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{Q}(s_t, a_t; \boldsymbol{w})$$

- For Q-learning

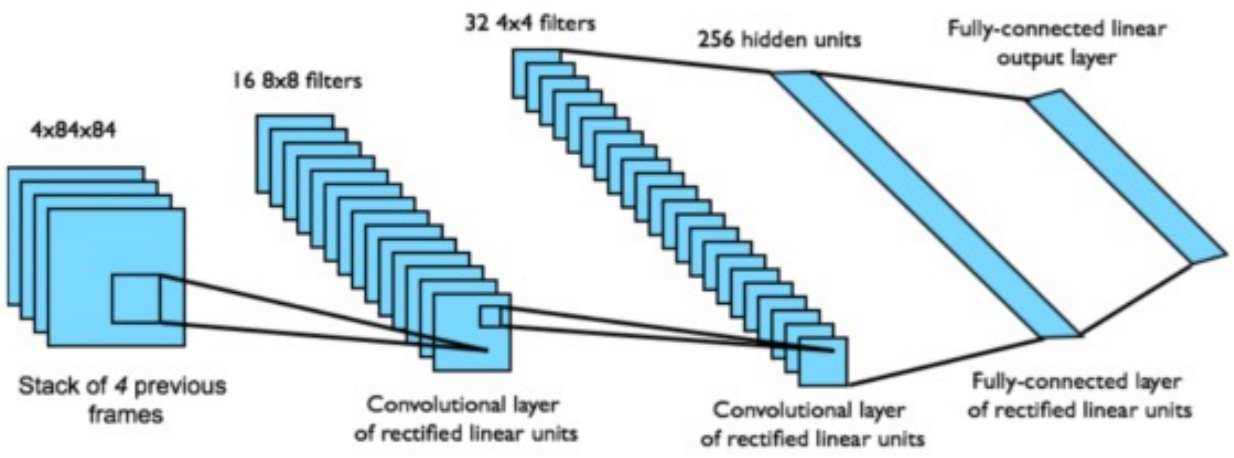
$$\Delta \boldsymbol{w} = \alpha(r + \gamma \max_a \hat{Q}(s_{t+1}, a; \boldsymbol{w}) - \hat{Q}(s_t, a_t; \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{Q}(s_t, a_t; \boldsymbol{w})$$

Doing deep RL in Atari



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
- Network architecture and hyperparameters fixed across all games



DQNs in Atari

- Q-learning converges to the optimal $Q^*(s, a)$ using table lookup representation.
- In value function approximation Q-learning, we can minimize MSE loss by stochastic gradient descent using a target Q estimate instead of true Q (as we saw with linear VFA)
- 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
  - Experience replay
  - Fixed Q-targets

DQNs: Experience Replay

- To help remove correlations, store dataset (called a replay buffer) D from prior experience

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\dots$
$s_t, a_t, r_{t+1}, s_{t+1}$

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

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

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

Problem

Can treat the target as a constant scalar, but the weights will get updated on the next round, changing the target value

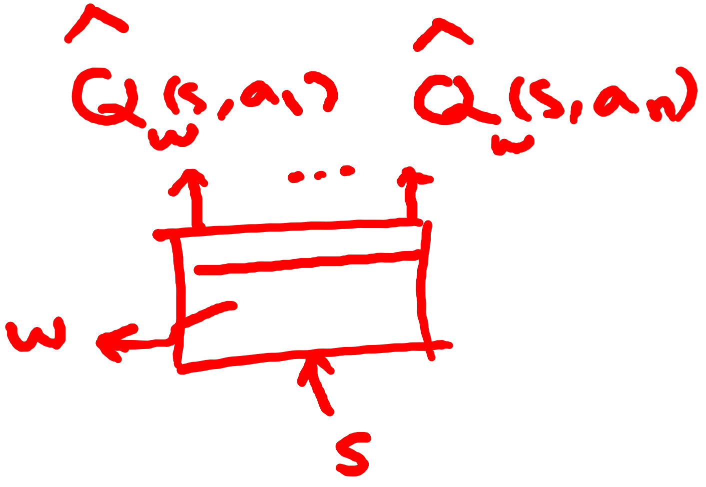
DQNs: Fixed Q-Targets

- 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 w^- be the set of weights used in the target, and w be the weights that are being updated
- Slight change to computation of target value:
  - $(s, a, r, s') \sim D$: sample an experience tuple from the dataset
  - Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; w^-)$
  - 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 Algorithm

linear func.

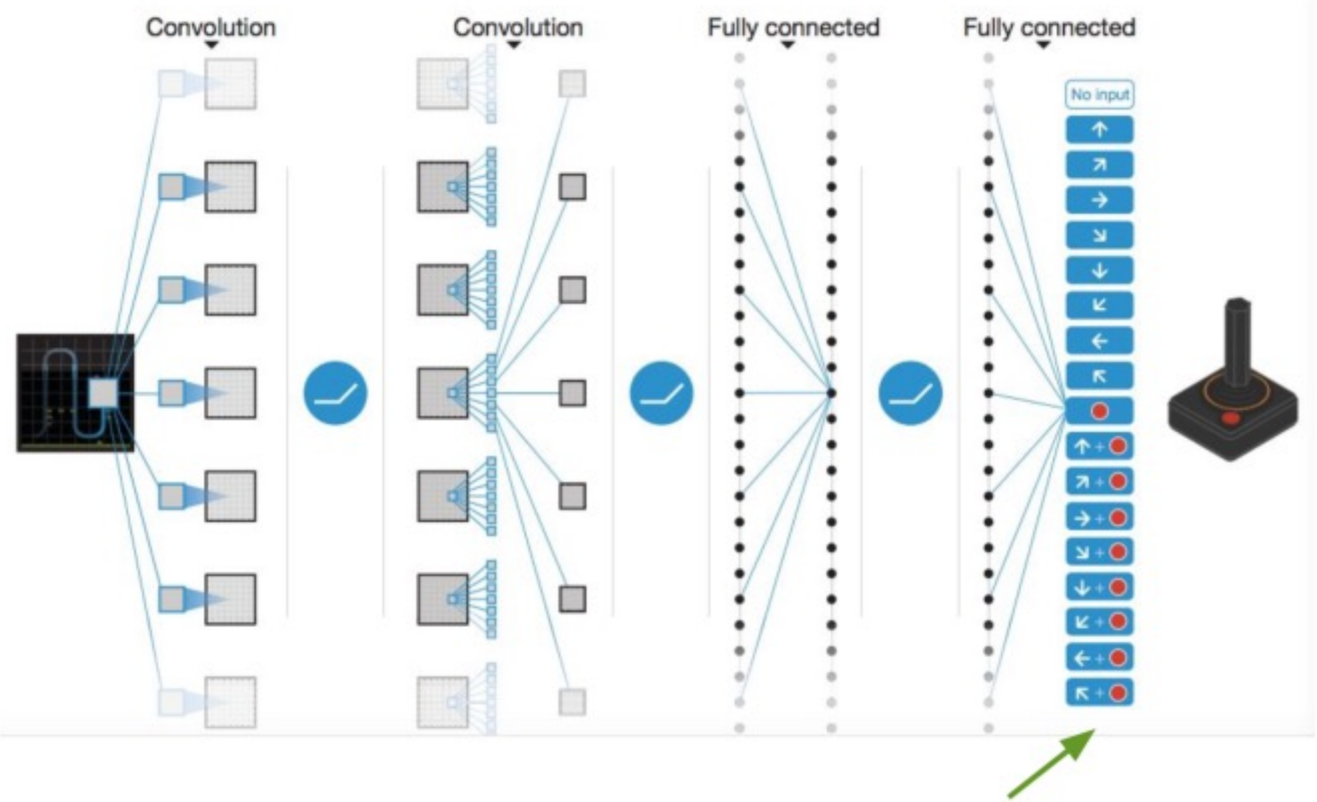


Input $\boldsymbol{C}$, α, $D = \{\}$, Initialize $\mathbf{w}$, $\mathbf{w}^- = \mathbf{w}$, $t = 0$
Get initial state s_0
loop
    Sample action a_t given ϵ-greedy policy for current $\hat{Q}(s_t, a; \mathbf{w})$
    Observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    Sample random minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from D
    for j in minibatch **do**
        if episode terminated at step $i+1$ **then**
            $y_i = r_i$
        else
            $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}^-)$
        end if
        Do gradient descent step on $(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))^2$ for parameters $\mathbf{w}$: $\Delta \mathbf{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))\nabla_{\mathbf{w}} \hat{Q}(s_i, a_i; \mathbf{w})$
    end for
    $t = t + 1$
    if mod(t,C) == 0 **then**
        $\mathbf{w}^- \leftarrow \mathbf{w}$
    end if
end loop

DQN Summary

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

DQN



1 network, outputs Q value for each action

Results

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