

# RL Agent Components

- Often include one or more of:
  - **Model:** Agent's representation of how the world changes in response to agent's action
  - **Policy:** function mapping agent's states to action
  - **Value function:** future rewards from being in a state and/or action when following a particular policy

# Model

- Agent's representation of how the world changes in response to agent's action
- Transition / dynamics model predicts next agent state
  - $P(s_{t+1} = s' | s_t = s, a_t = a)$
- Reward model predicts immediate reward
  - $R(s_t = s, a_t = a) = \mathbb{E} [r_t | s_t = s, a_t = a]$

# Policy

- Policy  $\pi$  determines how the action chooses actions
- $\pi: S \rightarrow A$ , mapping from state to action
- Deterministic policy  $\pi(s) = a$
- Stochastic policy  $\pi(a | s) = P(a_t = a | s_t = s)$

# Value

- Value function  $V^\pi$ : expected discounted sum of future rewards under a particular policy  $\pi$ 
  - $V^\pi(s_t=s) = \mathbb{E}_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+1} + \gamma^3 r_{t+1} + \dots \mid s_t = s]$
- Discount factor  $\gamma$  weighs immediate vs future rewards
- Can be used to quantify goodness/badness of states and actions
- And decide how to act by comparing policies



# Types of RL Agents:

## What the Agent (Algorithm) Learns

- Value based
  - Explicit: Value function
  - Implicit: Policy (can derive a policy from value function)
- Policy based
  - Explicit: policy
  - No value function
- Actor Critic
  - Explicit: Policy
  - Explicit: Value function

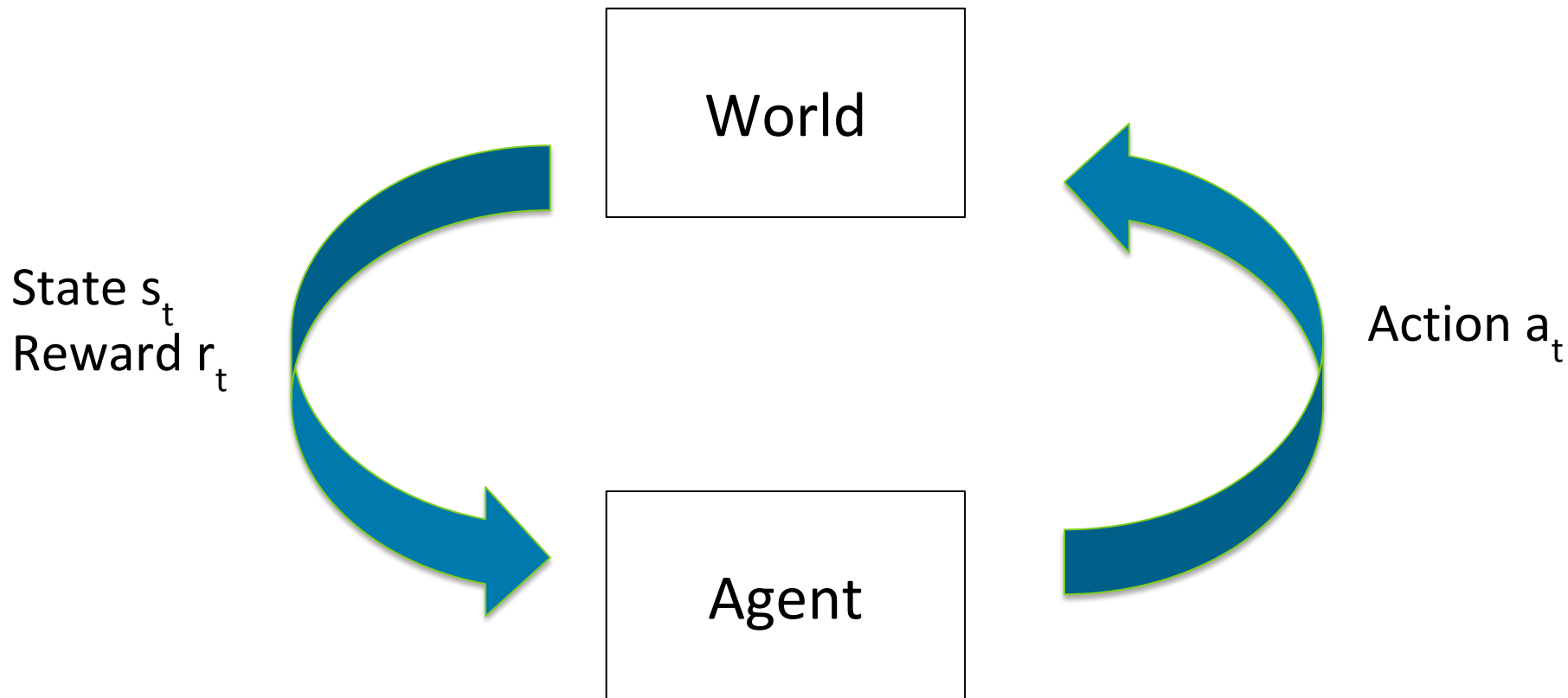
# Types of RL Agents

- Model Based
  - Explicit: model
  - May or may not have policy and/or value function
- Model Free
  - Explicit: Value function and/or Policy Function
  - No model

# Today: Given a Model of the World

1. Markov Processes
2. Markov Reward Processes (MRPs)
3. Markov Decision Processes (MDPs)
4. Evaluation and Control in MDPs

# Full Observability: Markov Decision Process (MDP)



- MDPs can model a huge number of interesting problems and settings
  - Bandits: single state MDP
  - Optimal control mostly about continuous-state MDPs
  - Partially observable MDPs = MDP where state is history

# Markov Reward Process (MRP)

- A Markov Reward Process is a Markov Chain + rewards
- **Definition of MRP:**
  - $S$  is a (finite) set of states
  - $P$  is dynamics / transition model, that specifies  $P(s_{t+1} = s' | s_t = s)$
  - $R$  is a reward function  $R(s_t = s) = \mathbb{E} [r_t | s_t = s]$
  - Discount factor  $\gamma \in [0,1]$
- Note: no actions
- If finite number ( $N$ ) of states, can express  $R$  as a vector

# Return & Value Function

- **Definition of Horizon:**
  - Number of time steps in each episode in a process
  - Can be infinite
  - Otherwise called **finite** Markov reward process
- **Definition of Return  $G_t$**  (for a Markov reward process):
  - Discounted sum of rewards from time step  $t$  to horizon
  - $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$
- **Definition of State value function  $V(s)$**  (for a MRP):
  - Expected return from starting in state  $s$
  - $V(s) = \mathbb{E} [G_t | s_t = s] = \mathbb{E} [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots | s_t = s]$

# Discount Factor

- Mathematically convenient (avoid infinite returns and values)
- Humans often act as if there's a discount factor  $< 1$
- $\gamma=0$ : Only care about immediate reward
- $\gamma=1$ : Future reward is as beneficial as immediate reward
- If episode lengths are always finite, can use  $\gamma=1$

# Computing the Value of a Markov Reward Process

- Could estimate by simulation
- Markov property yields additional structure
- MRP value function satisfies:

$$V(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s) V(s')$$

  
Immediate reward      Discounted sum of future rewards



# Matrix Form of Bellman Eqn for Markov Reward Processes

- For finite state MRP can express using matrices

$$\begin{bmatrix} V(s1) \\ \dots \\ V(sN) \end{bmatrix} = \begin{bmatrix} R(s1) \\ \dots \\ R(sN) \end{bmatrix} + \gamma \begin{bmatrix} P(s1|s1) & \dots & P(sN|s1) \\ P(s1|sN) & \dots & P(sN|sN) \end{bmatrix} \begin{bmatrix} V(s1) \\ \dots \\ V(sN) \end{bmatrix}$$

$$V = R + \gamma PV$$

# Analytic Solution for Value of MRP

- For finite state MRP can express using matrices

$$\begin{bmatrix} V(s1) \\ \vdots \\ V(sN) \end{bmatrix} = \begin{bmatrix} R(s1) \\ \vdots \\ R(sN) \end{bmatrix} + \gamma \begin{bmatrix} P(s1|s1) & \dots & P(sN|s1) \\ \vdots & \ddots & \vdots \\ P(s1|sN) & \dots & P(sN|sN) \end{bmatrix} \begin{bmatrix} V(s1) \\ \vdots \\ V(sN) \end{bmatrix}$$

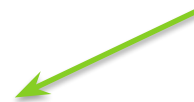
$$V = R + \gamma PV$$

$$V - \gamma PV = R$$

$$(I - \gamma P)V = R$$

$$V = (I - \gamma P)^{-1}R$$

Matrix inverse,  
 $\sim O(N^3)$



# Iterative Algorithm for Computing Value of a MRP

- Dynamic programming
- Initialize  $V_0(s) = 0$  for all  $s$
- For  $k=1$  until convergence
  - For all  $s$  in  $S$ :

$$V_k(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s) V_{k-1}(s')$$

- Computational complexity:  $O(S^2)$  for each  $t$

# Markov Decision Process (MDP)

- A Markov Decision Process is Markov Reward Process + actions
- **Definition of MDP:**
  - $S$  is a (finite) set of Markov states
  - $A$  is a (finite) set of actions
  - $P$  is dynamics / transition model for **each action**, that specifies  $P(s_{t+1} = s' | s_t = s, a_t = a)$
  - $R$  is a reward function  $R(s_t = s, a_t = a) = \mathbb{E} [r_t | s_t = s, a_t = a]^*$
  - Discount factor  $\gamma \in [0, 1]$
- MDP is a tuple:  $(S, A, P, R, \gamma)$

\*Reward sometimes defined as a function of the current state, or as a function of the state-action-next state. Most frequently in this class we will assume reward is a function of state and action

# MDP Policies

- Policy specifies what action to take in each state
  - Can be deterministic or stochastic
- For generality consider as a conditional distribution: given a state specifies a distribution over actions
- Policy  $\pi(a | s) = P(a_t=a | s_t=s)$

# Policy Evaluation for MDP

- MDP +  $\pi(a|s)$  = a Markov reward process
- Precisely it is a a MRP  $(S, R^\pi, P^\pi, \gamma)$  where

$$R^\pi(s) = \sum_{a \in A} \pi(a|s) R(s, a)$$

$$P^\pi(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s, a)$$

- Implies we can use same techniques to evaluate the value of a policy for a MDP as we could to compute the value of a MRP

# Slight Modification to Iterative Algorithm for Computing Value of a MRP

- Initialize  $V_0(s) = 0$  for all  $s$
- For  $k=1$  until convergence
  - For all  $s$  in  $S$ :

$$V_k^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V_{k-1}^\pi(s')$$

- Just replaced dynamics and reward model

# Slight Modification to Iterative Algorithm for Computing Value of a MRP

- Initialize  $V_0(s) = 0$  for all  $s$
- For  $k=1$  until convergence
  - For all  $s$  in  $S$ :

**Bellman backup for  
a particular policy**



$$V_k^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V_{k-1}^\pi(s')$$

- Just replaced dynamics and reward model



# MDP Control

- Compute the optimal policy

$$\pi^*(s) = \arg \max_{\pi} V^{\pi}(s)$$

- There exists a unique optimal value function
- Optimal policy for a MDP in an infinite horizon problem (agent acts forever) is:
  - Deterministic
  - Stationary (does not depend on time step)
  - Unique? Not necessarily, may be ties

# Policy Search

- One option is searching to compute best policy
- Number of deterministic policies is  $|A|^{|S|}$
- Policy iteration is generally more efficient than enumeration

# New Definition: State-Action Value Q

- State-action value of a policy

$$Q^{\pi}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s')$$


- Take action a, then follow policy

# Policy Iteration (PI)

1.  $i=0$ ; Initialize  $\pi_0(s)$  randomly for all states  $s$

2. While  $i \neq 0$  or  $|\pi_i - \pi_{i-1}| > 0$

- Policy **evaluation** of  $\pi_i$
- $i=i+1$
- Policy **improvement**



Use a L1 norm:  
measures if the  
policy changed  
for any state

# Policy Improvement

Compute state-action value of a policy  $\pi_i$

$$Q^{\pi_i}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi_i}(s')$$

Note

$$\begin{aligned} \max_a Q^{\pi_i}(s, a) &= \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi_i}(s') \\ &\geq R(s, \pi_i(s)) + \gamma \sum_{s' \in S} P(s'|s, \pi_i(s)) V^{\pi_i}(s') \\ &= V^{\pi_i}(s) \end{aligned}$$

Define new policy

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a) \quad \forall s \in S$$

# Policy Iteration (PI)

1.  $i=0$ ; Initialize  $\pi_0(s)$  randomly for all states  $s$

2. While  $i \neq 0$  or  $|\pi_i - \pi_{i-1}| > 0$  ←

- Policy **evaluation**: Compute value of  $\pi_i$
- $i=i+1$
- Policy **improvement**:

**Use a L1 norm:**  
measures if the policy changed for any state

$$Q^{\pi_i}(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) V^{\pi_i}(s')$$

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a)$$

# Delving Deeper Into Improvement

$$Q^{\pi_i}(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) V^{\pi_i}(s')$$

$$\max_a Q^{\pi_i}(s, a) \geq V^{\pi_i}(s)$$

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a)$$

- So if take  $\pi_{i+1}(s)$  then followed  $\pi_i$  forever,
  - expected sum of rewards would be at least as good as if we had always followed  $\pi_i$
- But new proposed policy is to always follow  $\pi_{i+1} \dots$

# Monotonic Improvement in Policy

- Definition

$$V^{\pi_1} \geq V^{\pi_2} \rightarrow V^{\pi_1}(s) \geq V^{\pi_2}(s) \quad \forall s \in S$$

- Proposition:  $V^{\pi'} \geq V^{\pi}$  with strict inequality if  $\pi$  is suboptimal (where  $\pi'$  is the new policy we get from doing policy improvement)



# Policy Iteration (PI)

1.  $i=0$ ; Initialize  $\pi_0(s)$  randomly for all states  $s$

2. While  $i \neq 0$  or  $|\pi_i - \pi_{i-1}| > 0$  ←

- Policy **evaluation**: Compute value of  $\pi_i$
- $i=i+1$
- Policy **improvement**:

**Use a L1 norm:**  
measures if the policy changed for any state

$$Q^{\pi_i}(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) V^{\pi_i}(s')$$

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a)$$

## Policy Iteration Can Take At Most $|A|^{|S|}$ Iterations\* (Size of # Policies)

1.  $i=0$ ; Initialize  $\pi_0(s)$  randomly for all states  $s$
2. Converged = 0;
3. While  $i \neq 0$  or  $|\pi_i - \pi_{i-1}| > 0$ 
  - $i=i+1$
  - Policy **evaluation**: Compute  $V^\pi$
  - Policy **improvement**:

$$Q^{\pi_i}(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V^{\pi_i}(s')$$

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a)$$

\* For finite state and action spaces

# MDP: Computing Optimal Policy and Optimal Value

- Policy iteration computes optimal value and policy
- Value iteration is another technique
  - Idea: Maintain optimal value of starting in a state  $s$  if have a finite number of steps  $k$  left in the episode
  - Iterate to consider longer and longer episodes

# Bellman Equation and Bellman Backup Operators

- Bellman equation
  - The value function for a policy must satisfy

$$V^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V^\pi(s')$$

- Bellman backup operator
  - Applied to a value function
  - Returns a new value function
  - Improves the value if possible

$$BV(s) = \max_a R(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) V(s')$$

- $BV$  yields a value function over all  $s$

# Value Iteration (VI)

1. Initialize  $V_0(s)=0$  for all states  $s$
2. Set  $k=1$
3. Loop until [finite horizon, convergence]
  - For each state  $s$

$$V_{k+1}(s) = \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s')$$

- View as Bellman backup on value function

$$\begin{aligned} V_{k+1} &= BV_k \\ \pi_{k+1}(s) &= \arg \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s') \end{aligned}$$

# Looking at Policy Iteration As Bellman Operations:

## Policy Evaluation: Compute Fixed Point of $B^\pi$

- Bellman backup operator for a particular policy

$$B^\pi V(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s)V(s')$$

- To do policy evaluation, repeatedly apply operator until  $V$  stops changing

$$V^\pi = B^\pi B^\pi \dots B^\pi V$$

# Looking at Policy Iteration As Bellman Operations: Policy Improvement, Slight Variant of Bellman

- Bellman backup operator for a particular policy

$$B^{\pi}V(s) = R^{\pi}(s) + \gamma \sum_{s' \in S} P^{\pi}(s'|s)V(s)$$

- To do policy improvement

$$\pi_{k+1}(s) = \arg \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)V^{\pi_k}(s')$$

# Going Back to Value Iteration (VI)

1. Initialize  $V_0(s)=0$  for all states  $s$
2. Set  $k=1$
3. Loop until [finite horizon, convergence]
  - For each state  $s$

$$V_{k+1}(s) = \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s')$$

- Doing a Bellman backup on value function

$$V_{k+1} = BV_k$$
$$\pi_{k+1}(s) = \arg \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s')$$



# Contraction Operator

- Let  $O$  be an operator
- If  $|OV - OV'| \leq |V - V'|$
- Then  $O$  is a contraction operator

# Will Value Iteration Converge?

- Yes, if discount factor  $\gamma < 1$  or end up in a terminal state with probability 1
- Bellman backup is a contraction if discount factor,  $\gamma < 1$
- If apply it to two different value functions, distance between value functions shrinks after apply Bellman equation to each

# Bellman Backup is a Contraction on $V$ ( $\gamma < 1$ )

$\|V - V'\|$  = Infinity norm (find max difference over all states, e.g.  $\max(s) |V(s) - V'(s)|$ )

$$\begin{aligned}\|BV_k - BV_j\| &= \left\| \left( \max_a R(s, a) + \gamma \sum_{s'} P(s'|s, a) V_k(s') \right) - \left( \max_{a'} R(s, a') + \gamma \sum_{s'} P(s'|s, a') V_j(s') \right) \right\| \\ &\leq \left\| \max_a R(s, a) + \gamma \sum_{s'} P(s'|s, a) V_k(s') - R(s, a) - \gamma \sum_{s'} P(s'|s, a) V_j(s') \right\| \\ &= \left\| \max_a \gamma \sum_{s'} P(s'|s, a) (V_k(s') - V_j(s')) \right\| \\ &\leq \left\| \max_a \gamma \sum_{s'} P(s'|s, a) \|V_k - V_j\| \right\| \\ &\leq \left\| \gamma \|V_k - V_j\| \max_a \sum_{s'} P(s'|s, a) \right\| \\ &= \gamma \|V_k - V_j\|\end{aligned}$$

Note: even if all inequalities are equalities, this still is a contraction as long as the discount factor is  $< 1$

# Value vs Policy Iteration

- Value iteration:
  - Compute optimal value if horizon= $k$ 
    - Note this can be used to compute optimal policy if horizon =  $k$
  - Increment  $k$
- Policy iteration:
  - Compute infinite horizon value of a policy
  - Use to select another (better) policy
  - Closely related to a very popular method in RL: policy gradient

# What You Should Know

- Define MP, MRP, MDP, Bellman operator, contraction, model, Q-value, policy
- Be able to implement
  - Value iteration & policy iteration
- Contrast benefits and weaknesses of policy evaluation approaches
- Be able to prove contraction properties
- Limitations of presented approaches and Markov assumptions
  - Which policy evaluation methods require Markov assumption?

# This Lecture: Policy Evaluation

- **Estimating the expected return of a particular policy if don't have access to true MDP models**
- Dynamic programming
- Monte Carlo policy evaluation
  - Policy evaluation when don't have a model of how the world work
    - Given on-policy samples
    - Given off-policy samples
- Temporal Difference (TD)
- Metrics to evaluate and compare algorithms

# Recall

- **Definition of return  $G_t$  for a MDP under policy  $\pi$ :**
  - Discounted sum of rewards from time step  $t$  to horizon when following policy  $\pi(a|s)$
  - $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$
- **Definition of state value function  $V^\pi(s)$  for policy  $\pi$ :**
  - Expected return from starting in state  $s$  under policy  $\pi$
  - $V^\pi(s) = \mathbb{E}_\pi [G_t | s_t = s] = \mathbb{E}_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots | s_t = s]$
- **Definition of state-action value function  $Q^\pi(s,a)$  for policy  $\pi$ :**
  - Expected return from starting in state  $s$ , taking action  $a$ , and then following policy  $\pi$
  - $Q^\pi(s,a) = \mathbb{E}_\pi [G_t | s_t = s, a_t = a] = \mathbb{E}_\pi [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a]$

# Dynamic Programming for Policy Evaluation

- Initialize  $V_0(s) = 0$  for all  $s$
- For  $k=1$  until convergence
  - For all  $s$  in  $S$ :

$$V_k^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V_{k-1}^\pi(s')$$

$$R^\pi(s) = \sum_{a \in A} \pi(a|s) R(s, a)$$

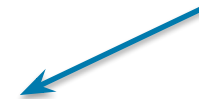
$$P^\pi(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s, a)$$



# Dynamic Programming for Policy Evaluation

- Initialize  $V_0(s) = 0$  for all  $s$
- For  $k=1$  until convergence
  - For all  $s$  in  $S$ :

**Bellman backup for  
a particular policy**



$$V_k^\pi(s) = B^\pi V_{k-1}^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V_{k-1}^\pi(s')$$

$$R^\pi(s) = \sum_{a \in A} \pi(a|s) R(s, a)$$

$$P^\pi(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s, a)$$

# Dynamic Programming for Policy $\pi$

## Value Evaluation

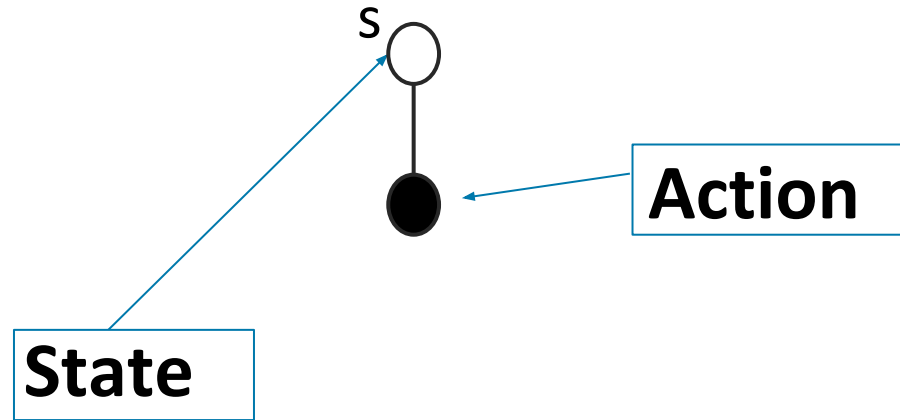
- Initialize  $V_0(s) = 0$  for all  $s$
- For  $i=1$  until convergence\*
  - For all  $s$  in  $S$ :

$$V_k^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V_{k-1}^\pi(s')$$

- In finite horizon case,  $V_k^\pi(s)$  is exact value of  $k$ -horizon value of state  $s$  under policy  $\pi$
- In infinite horizon case,  $V_k^\pi(s)$  is an estimate of infinite horizon value of state  $s$ 
  - $V^\pi(s) = \mathbb{E}_\pi [G_t | s_t = s] \cong \mathbb{E}_\pi [r_t + \gamma V_{i-1} | s_t = s]$

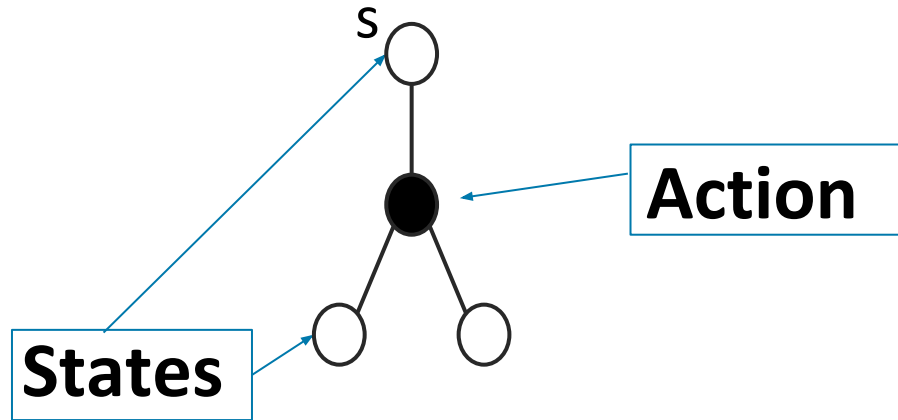
# Dynamic Programming Policy Evaluation

$$V^{\pi}(s) \leftarrow \mathbb{E}_{\pi}[r_t + \gamma V_{i-1} | s_t = s]$$



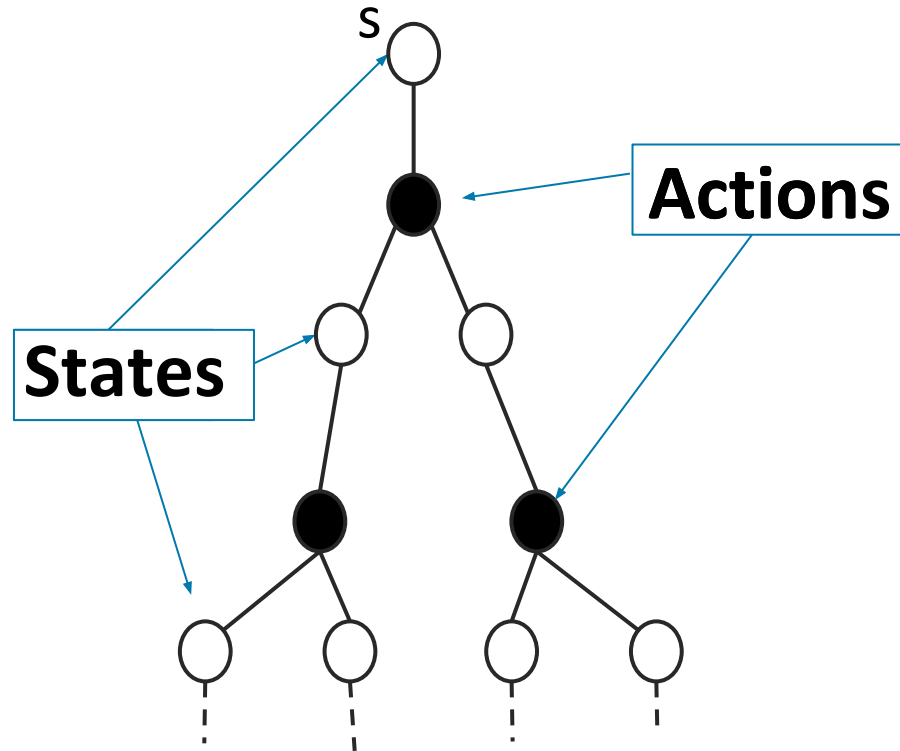
# Dynamic Programming Policy Evaluation

$$V^{\pi}(s) \leftarrow \mathbb{E}_{\pi}[r_t + \gamma V_{i-1} | s_t = s]$$



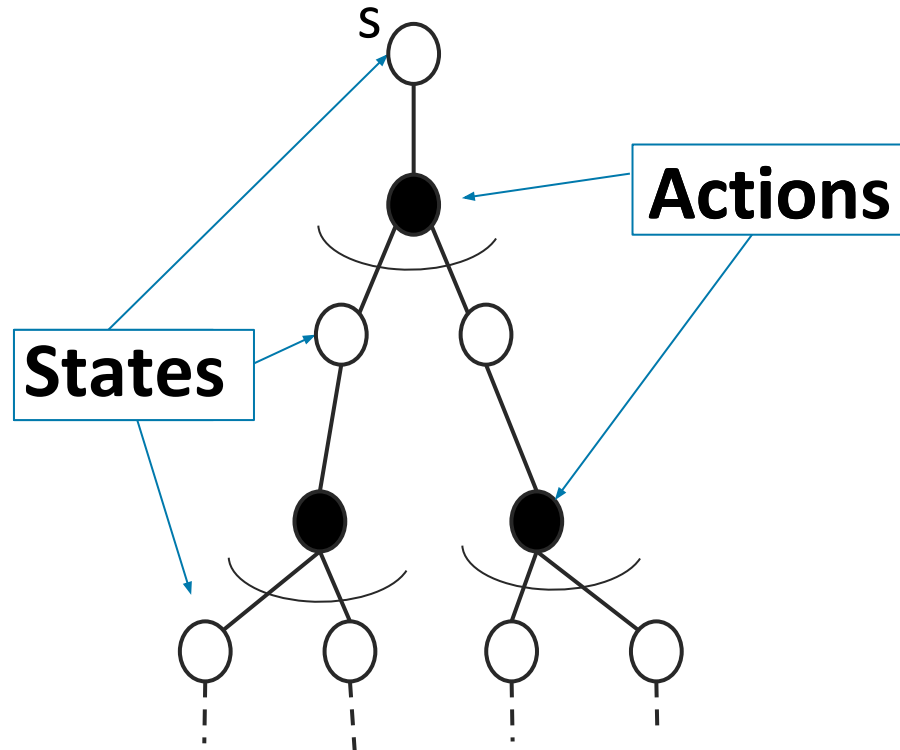
# Dynamic Programming Policy Evaluation

$$V^{\pi}(s) \leftarrow \mathbb{E}_{\pi}[r_t + \gamma V_{i-1} | s_t = s]$$



# Dynamic Programming Policy Evaluation

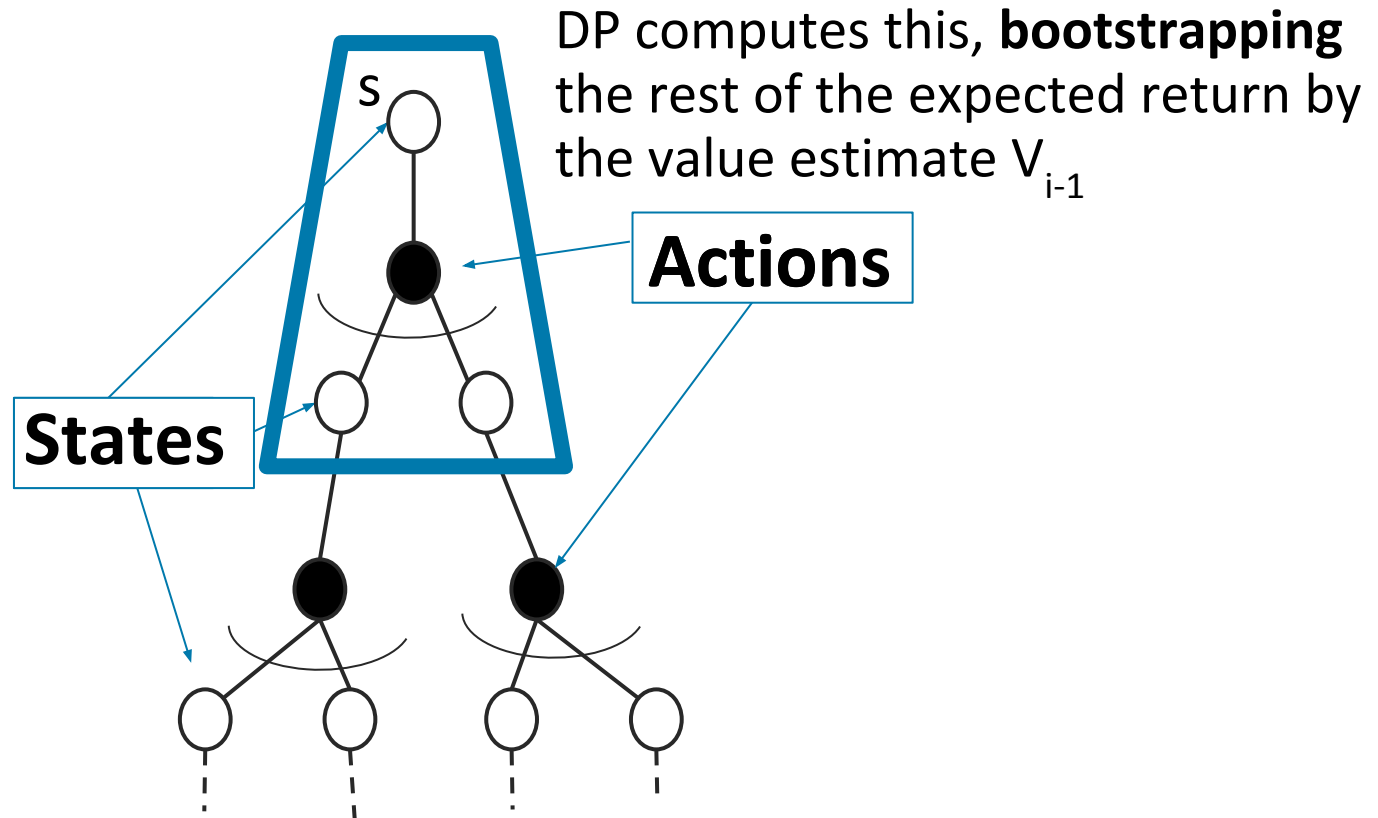
$$V^\pi(s) \leftarrow \mathbb{E}_\pi[r_t + \gamma V_{i-1} | s_t = s]$$



 = Expectation

# Dynamic Programming Policy Evaluation

$$V^\pi(s) \leftarrow \mathbb{E}_\pi[r_t + \gamma V_{i-1} | s_t = s]$$

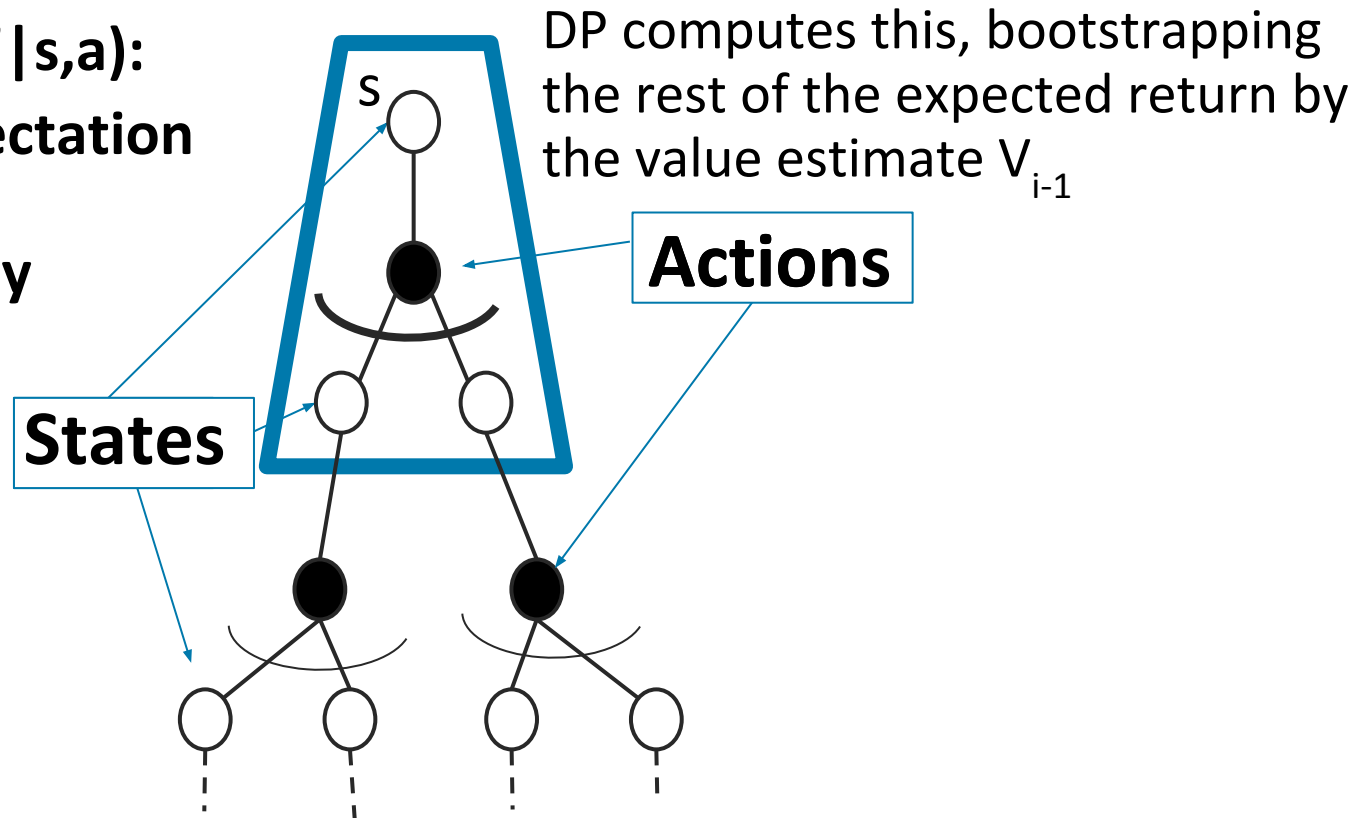


- **Bootstrapping:** Update for  $V$  uses an estimate

# Dynamic Programming Policy Evaluation

$$V^\pi(s) \leftarrow \mathbb{E}_\pi[r_t + \gamma V_{i-1} | s_t = s]$$

**Know model  $P(s' | s, a)$ :**  
**reward and expectation**  
**over next states**  
**computed exactly**



 = **Expectation**

- Bootstrapping: Update for  $V$  uses an estimate



# Policy Evaluation: $V^\pi(s) = \mathbb{E}_\pi [G_t | s_t = s]$

- $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi$
- Dynamic programming
  - $V^\pi(s) \cong \mathbb{E}_\pi [r_t + \gamma V_{i-1} | s_t = s]$
  - Requires model of MDP  $M$
  - Bootstraps future return using value estimate
- What if don't know how the world works?
  - Precisely, don't know dynamics model  $P$  or reward model  $R$
- **Today: Policy evaluation without a model**
  - Given data and/or ability to interact in the environment
  - Efficiently compute a good estimate of a policy  $\pi$

# This Lecture: Policy Evaluation

- Dynamic programming
- **Monte Carlo policy evaluation**
  - Policy evaluation when don't have a model of how the world work
    - Given on policy samples
    - Given off policy samples
- Temporal Difference (TD)
- Axes to evaluate and compare algorithms

# Monte Carlo (MC) Policy Evaluation

- $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi$
- $V^\pi(s) = \mathbb{E}_{T \sim \pi} [G_t | s_t = s]$ 
  - Expectation over trajectories  $T$  generated by following  $\pi$

# Monte Carlo (MC) Policy Evaluation

- $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi$
- $V^\pi(s) = \mathbb{E}_{T \sim \pi} [G_t | s_t = s]$ 
  - Expectation over trajectories  $T$  generated by following  $\pi$
- Simple idea: Value = mean return
- If trajectories are all finite, sample a bunch of trajectories and average returns
- By law of large numbers, average return converges to mean

# Monte Carlo (MC) Policy Evaluation

- If trajectories are all finite, sample a bunch of trajectories and average returns
- Does not require MDP dynamics / rewards
- No bootstrapping
- Does not assume state is Markov
- Can only be applied to episodic MDPs
  - Averaging over returns from a complete episode
  - Requires each episode to terminate

# Monte Carlo (MC) On Policy Evaluation

- Aim: estimate  $V^\pi(s)$  given episodes generated under policy  $\pi$ 
  - $s_1, a_1, r_1, s_2, a_2, r_2, \dots$  where the actions are sampled from  $\pi$
- $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi$
- $V^\pi(s) = \mathbb{E}_\pi [G_t | s_t = s]$
- MC computes empirical mean return
- Often do this in an incremental fashion
  - After each episode, update estimate of  $V^\pi$

# First-Visit Monte Carlo (MC) On Policy Evaluation

- After each episode  $i = s_{i1}, a_{i1}, r_{i1}, s_{i2}, a_{i2}, r_{i2}, \dots$ 
  - Define  $G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + \gamma^2 r_{i,t+2} + \dots$  as return from time step  $t$  onwards in  $i$ -th episode
  - For each state  $s$  visited in episode  $i$ 
    - For **first** time  $t$  state  $s$  is visited in episode  $i$ 
      - Increment counter of total first visits  $N(s) = N(s) + 1$
      - Increment total return  $S(s) = S(s) + G_{i,t}$
      - Update estimate  $V^\pi(s) = S(s) / N(s)$
- By law of large numbers, as  $N(s) \rightarrow \infty$ ,  $V^\pi(s) \rightarrow \mathbb{E}_\pi [G_t | s_t = s]$

# Every-Visit Monte Carlo (MC) On Policy Evaluation

- After each episode  $i = s_{i1}, a_{i1}, r_{i1}, s_{i2}, a_{i2}, r_{i2}, \dots$ 
  - Define  $G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + \gamma^2 r_{i,t+2} + \dots$  as return from time step  $t$  onwards in  $i$ -th episode
  - For each state  $s$  visited in episode  $i$ 
    - For **every** time  $t$  state  $s$  is visited in episode  $i$ 
      - Increment counter of total visits  $N(s) = N(s) + 1$
      - Increment total return  $S(s) = S(s) + G_{i,t}$
      - Update estimate  $V^\pi(s) = S(s) / N(s)$
- As  $N(s) \rightarrow \infty$ ,  $V^\pi(s) \rightarrow \mathbb{E}_\pi [G_t | s_t = s]$



# Incremental Monte Carlo (MC) On Policy Evaluation

- After each episode  $i = s_{i1}, a_{i1}, r_{i1}, s_{i2}, a_{i2}, r_{i2}, \dots$ 
  - Define  $G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + \gamma^2 r_{i,t+2} + \dots$  as return from time step  $t$  onwards in  $i$ -th episode
  - For state  $s$  visited at time step  $t$  in episode  $i$ 
    - Increment counter of total visits  $N(s) = N(s) + 1$
    - Update estimate

$$V^\pi(s) = V^\pi(s) \frac{N(s) - 1}{N(s)} + \frac{G_{it}}{N(s)} = V^\pi(s) + \frac{1}{N(s)} (G_{it} - V^\pi(s))$$

# Incremental Monte Carlo (MC)

## On Policy Evaluation Running Mean

- After each episode  $i = s_{i1}, a_{i1}, r_{i1}, s_{i2}, a_{i2}, r_{i2}, \dots$ 
  - Define  $G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + \gamma^2 r_{i,t+2} + \dots$  as return from time step  $t$  onwards in  $i$ -th episode
  - For state  $s$  visited at time step  $t$  in episode  $i$ 
    - Increment counter of total visits  $N(s) = N(s) + 1$
    - Update estimate

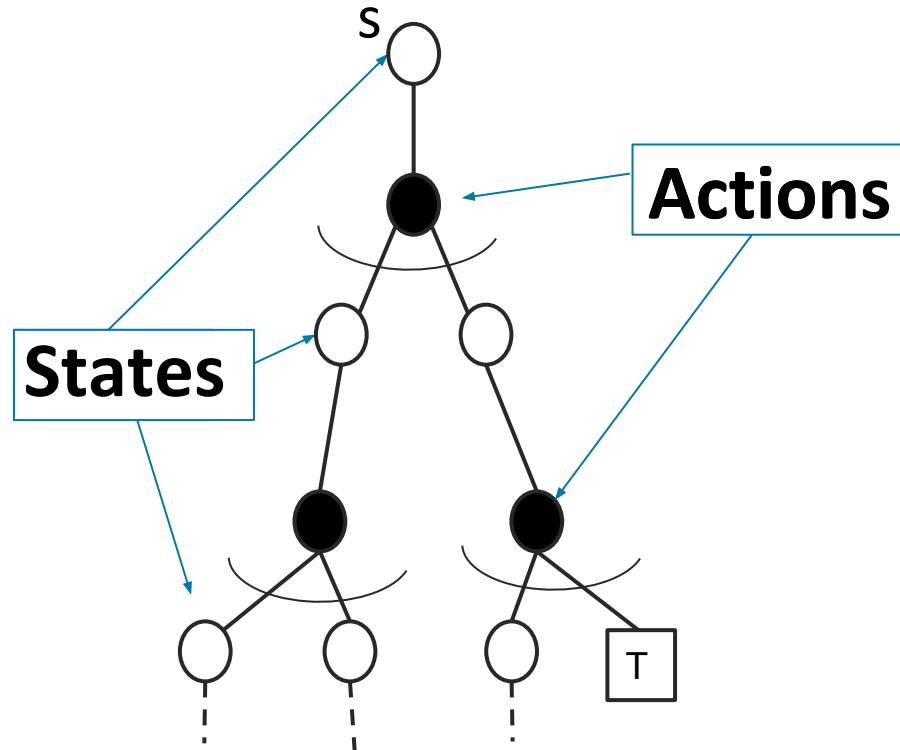
$$V^\pi(s) = V^\pi(s) + \alpha(G_{it} - V^\pi(s))$$

$\alpha = \frac{1}{N(s)}$  : identical to every visit MC

$\alpha > \frac{1}{N(s)}$  : forget older data, helpful for nonstationary domains

# MC Policy Evaluation

$$V^\pi(s) = V^\pi(s) + \alpha(G_{it} - V^\pi(s))$$



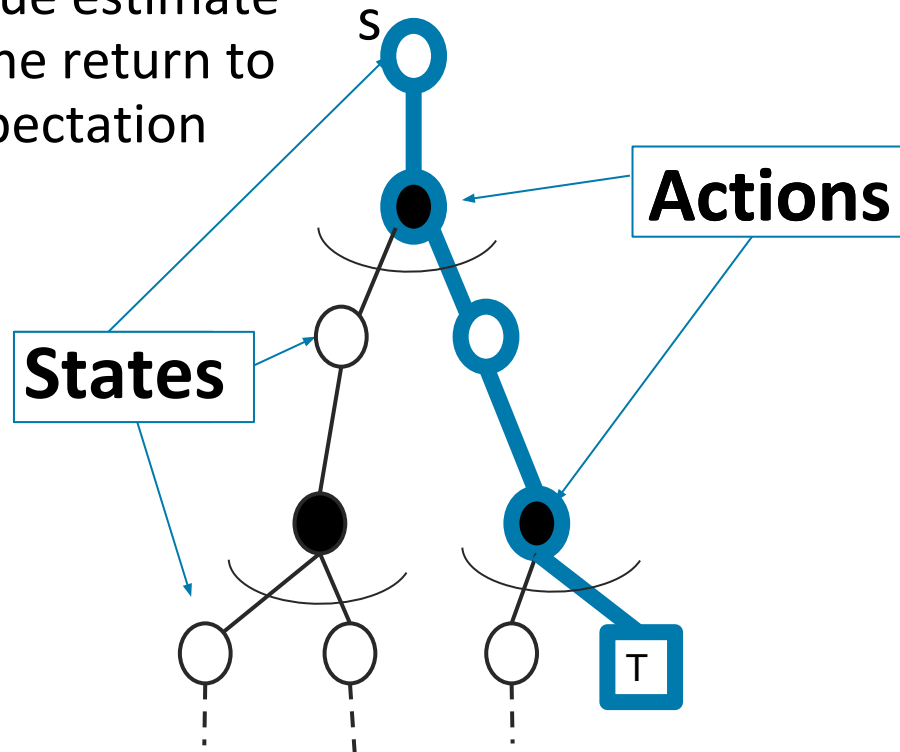
$\cup$  = Expectation

**T** = Terminal state

# MC Policy Evaluation

$$V^\pi(s) = V^\pi(s) + \alpha(G_{it} - V^\pi(s))$$

MC updates the value estimate using a **sample** of the return to approximate an expectation



 = Expectation

 = **Terminal state**

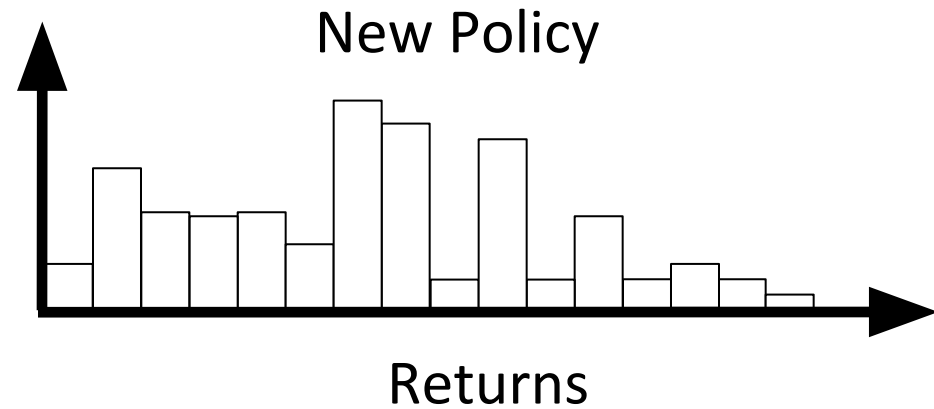
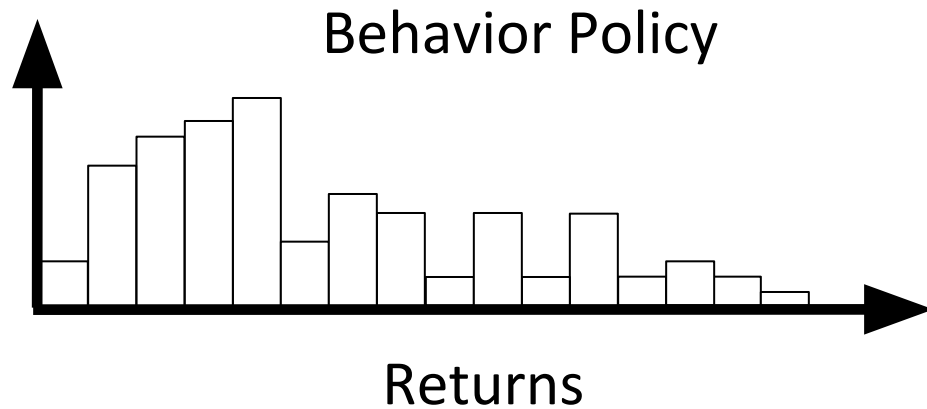
# Monte Carlo (MC) Off Policy Evaluation

- Aim: estimate  $V^\pi$  given episodes generated under policy  $\pi_1$ 
  - $s_1, a_1, r_1, s_2, a_2, r_2, \dots$  where the actions are sampled from  $\pi_1$
- $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi$
- $V^\pi(s) = \mathbb{E}_\pi [G_t | s_t = s]$
- Have data from another policy
- If  $\pi_1$  is stochastic can often use it to estimate the value of an alternate policy (formal conditions to follow)
- Again, no requirement for model nor that state is Markov

# Monte Carlo (MC) Off Policy

## Evaluation: Distribution Mismatch

- Distribution of episodes & resulting returns differs between policies



# Importance Sampling

- Goal: estimate the expected value of a function  $f(x)$  under some probability distribution  $p(x)$ ,  $\mathbb{E}_{x \sim p}[f(x)]$
- Have data  $x_1, x_2, \dots, x_n$  sampled from distribution  $q(s)$
- Under a few assumptions, can use samples to obtain an unbiased estimate of  $\mathbb{E}_{x \sim q}[f(x)]$

$$\mathbb{E}_{x \sim q}[f(x)] = \int_x q(x)f(x)$$

# Importance Sampling for Policy Evaluation

- Aim: estimate  $V^{\pi_1}$  given episodes generated under policy  $\pi_2$ 
  - $s_1, a_1, r_1, s_2, a_2, r_2, \dots$  where the actions are sampled from  $\pi_2$
- Have access to  $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi_2$
- Want  $V^{\pi_1}(s) = E_{\pi_1}[G_t | s_t = s]$
- Have data from another policy
- If  $\pi_2$  is stochastic can often use it to estimate the value of an alternate policy (formal conditions to follow)
- Again, no requirement for model nor that state is Markov



# Importance Sampling (IS) for Policy Evaluation

- Let  $h$  be a particular episode (history) of states, actions and rewards

$$h = (s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_{terminal})$$

# Probability of a Particular Episode

- Let  $h$  be a particular episode (history) of states, actions and rewards

$$h = (s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_{terminal})$$

$$\begin{aligned} p(h_j | \pi, s) &= p(a_{j1} | s_{j1}) p(r_{j1} | s_{j1}, a_{j1}) p(s_{j2} | s_{j1}, a_{j1}) p(a_{j2} | s_{j2}) \dots \\ &= \prod_{t=1}^{L_j-1} p(a_{j,t} | s_{j,t}) p(r_{j,t} | s_{j,t}, a_{j,t}) p(s_{j,t+1} | s_{j,t}, a_{j,t}) \\ &= \prod_{t=1}^{L_j-1} \pi(a_{j,t} | s_{j,t}) p(r_{j,t} | s_{j,t}, a_{j,t}) p(s_{j,t+1} | s_{j,t}, a_{j,t}) \end{aligned}$$

# Importance Sampling (IS) for Policy Evaluation

- Let  $h$  be a particular episode (history) of states, actions and rewards

$$h = (s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_{terminal})$$

$$V^{\pi_1}(s) \approx \frac{1}{N} \sum_{j=1}^N \frac{p(h_j | \pi_1, s)}{p(h_j | \pi_2, s)} G(h_j)$$

# Importance Sampling for Policy Evaluation

- Aim: estimate  $V^{\pi_1}$  given episodes generated under policy  $\pi_2$ 
  - $s_1, a_1, r_1, s_2, a_2, r_2, \dots$  where the actions are sampled from  $\pi_2$
- Have access to  $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi_2$
- Want  $V^{\pi_1}(s) = E_{\pi_1}[G_t | s_t = s]$
- IS = Monte Carlo estimate given off policy data
- Model-free method
- Does not require Markov assumption
- Under some assumptions, unbiased & consistent estimator of  $V^{\pi_1}$
- Can be used when agent is interacting with environment to estimate value of policies different than agent's control policy
- More later this quarter about batch learning

# Monte Carlo (MC) Policy Evaluation

## Summary

- Aim: estimate  $V^\pi(s)$  given episodes generated under policy  $\pi$ 
  - $s_1, a_1, r_1, s_2, a_2, r_2, \dots$  where the actions are sampled from  $\pi$
- $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi$
- $V^\pi(s) = \mathbb{E}_\pi [G_t | s_t = s]$
- Simple: Estimates expectation by empirical average (given episodes sampled from policy of interest) or reweighted empirical average (importance sampling)
- Updates value estimate by using a **sample** of return to approximate the expectation
- No bootstrapping
- Converges to true value under some (generally mild) assumptions

# Monte Carlo (MC) Policy Evaluation

## Key Limitations

- Generally high variance estimator
  - Reducing variance can require a lot of data
- Requires episodic settings
  - Episode must end before data from that episode can be used to update the value function

# This Lecture: Policy Evaluation

- Dynamic programming
- Monte Carlo policy evaluation
  - Policy evaluation when don't have a model of how the world work
    - Given on policy samples
    - Given off policy samples
- **Temporal Difference (TD)**
- Axes to evaluate and compare algorithms

# Temporal Difference Learning

- “If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning.” -- *Sutton and Barto 2017*
- Combination of Monte Carlo & dynamic programming methods
- Model-free
- **Bootstraps and samples**
- Can be used in episodic or infinite-horizon non-episodic settings
  - Immediately updates estimate of  $V$  after each  $(s,a,r,s')$  tuple



# Temporal Difference Learning for Estimating $V$

- Aim: estimate  $V^\pi(s)$  given episodes generated under policy  $\pi$
- $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots$  in MDP  $M$  under a policy  $\pi$
- $V^\pi(s) = \mathbb{E}_\pi [G_t | s_t = s]$
- Recall Bellman operator (if know MDP models)

$$B^\pi V(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V(s')$$

- In incremental every-visit MC, update estimate using 1 sample of return (for the current  $i^{\text{th}}$  episode)

$$V^\pi(s_{it}) = V^\pi(s_{it}) + \alpha(G_{it} - V^\pi(s_{it}))$$

- Insight: have an estimate of  $V^\pi$ , use to estimate expected return

$$V^\pi(s_t) = V^\pi(s_t) + \alpha([r_t + \gamma V^\pi(s_{t+1})] - V^\pi(s_t))$$

# Temporal Difference [TD(0)] Learning

- Aim: estimate  $V^\pi(s)$  given episodes generated under policy  $\pi$ 
  - $s_1, a_1, r_1, s_2, a_2, r_2, \dots$  where the actions are sampled from  $\pi$
- Simplest TD learning: update value towards estimated value

$$V^\pi(s_t) = V^\pi(s_t) + \alpha \underbrace{([r_t + \gamma V^\pi(s_{t+1})])}_{\text{TD target}} - V^\pi(s_t)$$

- TD error:

$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$

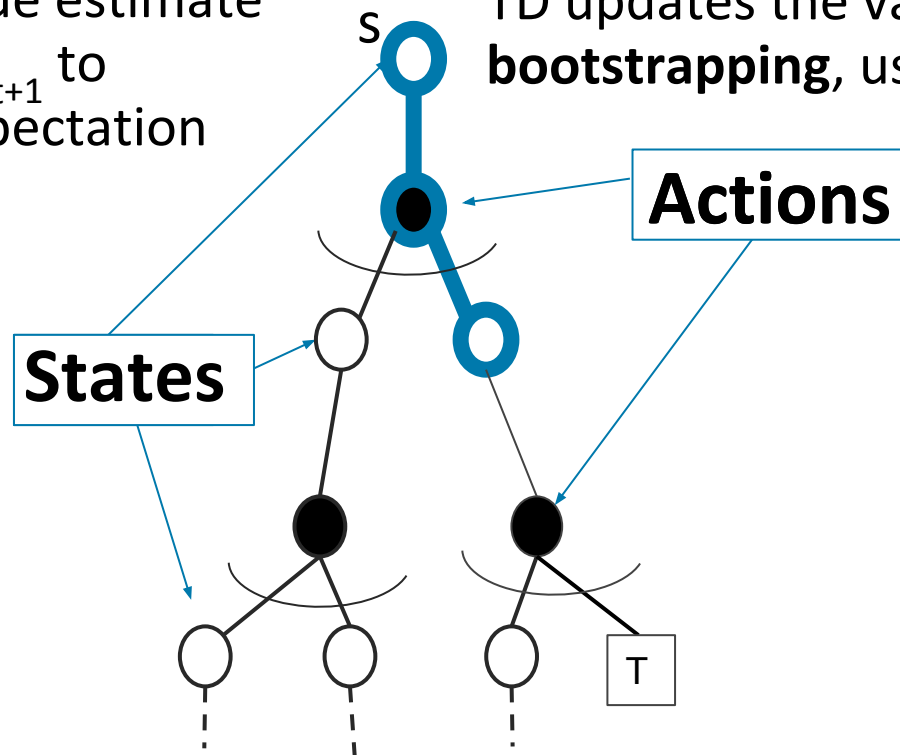
- **Can immediately update value estimate after (s,a,r,s') tuple**
- **Don't need episodic setting**

# Temporal Difference Policy Evaluation

$$V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha ([r_t + \gamma V^{\pi}(s_{t+1})] - V^{\pi}(s_t))$$

TD updates the value estimate using a **sample** of  $s_{t+1}$  to approximate an expectation

TD updates the value estimate by **bootstrapping**, uses estimate of  $V(s_{t+1})$



⌋ = Expectation

T = Terminal state

# This Lecture: Policy Evaluation

- Dynamic programming
- Monte Carlo policy evaluation
  - Policy evaluation when don't have a model of how the world work
    - Given on policy samples
    - Given off policy samples
- Temporal Difference (TD)
- **Axes to evaluate and compare algorithms**

# Some Important Properties to Evaluate Policy Evaluation Algorithms

- Usable when no models of current domain
  - DP: No      MC: Yes      TD: Yes
- Handles continuing (non-episodic) domains
  - DP: Yes      MC: No      TD: Yes
- Handles Non-Markovian domains
  - DP: No      MC: Yes      TD: No
- Converges to true value in limit\*
  - DP: Yes      MC: Yes      TD: Yes
- Unbiased estimate of value
  - DP: NA      MC: Yes      TD: No

\* For tabular representations of value function. More on this in later lectures

# Some Important Properties to Evaluate Model-free Policy Evaluation Algorithms

- Bias/variance characteristics
- Data efficiency
- Computational efficiency

# Bias/Variance of Model-free Policy Evaluation Algorithms

- Return  $G_t$  is an unbiased estimate of  $V^\pi(s_t)$
- TD target  $[r_t + \gamma V^\pi(s_{t+1})]$  is a biased estimate of  $V^\pi(s_t)$
- But often much lower variance than a single return  $G_t$
- Return function of multi-step seq. of random actions, states & rewards
- TD target only has one random action, reward and next state
- MC
  - Unbiased
  - High variance
  - Consistent (converges to true) even with function approximation
- TD
  - Some bias
  - Lower variance
  - TD(0) converges to true value with tabular representation
  - TD(0) does not always converge with function approximation

# Batch MC and TD

- Batch (Offline) solution for finite dataset
  - Given set of  $K$  episodes
  - Repeatedly sample an episode from  $K$
  - Apply MC or TD(0) to that episode
- What do MC and TD(0) converge to?



# Table of Contents

- 1 Generalized Policy Iteration
- 2 Importance of Exploration
- 3 Monte Carlo Control
- 4 Temporal Difference Methods for Control
- 5 Maximization Bias

# Class Structure

- Last time: Policy evaluation with no knowledge of how the world works (MDP model not given)
- This time: Control (making decisions) without a model of how the world works
- Next time: Value function approximation and Deep Q-learning

# Evaluation to Control

- Last time: how good is a specific policy?
  - Given no access to the decision process model parameters
  - Instead have to estimate from data / experience
- Today: how can we learn a good policy?

# Recall: Reinforcement Learning Involves

- Optimization
- Delayed consequences
- Exploration
- Generalization

# Today: Learning to Control Involves

- Optimization: Goal is to identify a policy with high expected rewards (similar to Lecture 2 on computing an optimal policy given decision process models)
- Delayed consequences: May take many time steps to evaluate whether an earlier decision was good or not
- Exploration: Necessary to try different actions to learn what actions can lead to high rewards

# Today: Model-free Control

- Generalized policy improvement
- Importance of exploration
- Monte Carlo control
- Model-free control with temporal difference (SARSA, Q-learning)
- Maximization bias

# Model-free Control Examples

- Many applications can be modeled as a MDP: Backgammon, Go, Robot locomotion, Helicopter flight, Robocup soccer, Autonomous driving, Customer ad selection, Invasive species management, Patient treatment
- For many of these and other problems either:
  - MDP model is unknown but can be sampled
  - MDP model is known but it is computationally infeasible to use directly, except through sampling

# On and Off-Policy Learning

- On-policy learning
  - Direct experience
  - Learn to estimate and evaluate a policy from experience obtained from following that policy
- Off-policy learning
  - Learn to estimate and evaluate a policy using experience gathered from following a different policy



# Table of Contents

- 1 Generalized Policy Iteration
- 2 Importance of Exploration
- 3 Monte Carlo Control
- 4 Temporal Difference Methods for Control
- 5 Maximization Bias

# Recall Policy Iteration

- Initialize policy  $\pi$
- Repeat:
  - Policy evaluation: compute  $V^\pi$
  - Policy improvement: update  $\pi$

$$\pi'(s) = \arg \max_a R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s') = \arg \max_a Q^\pi(s, a) \quad (1)$$

- Now want to do the above two steps without access to the true dynamics and reward models
- Last lecture introduced methods for model-free policy evaluation

# Model-free Generalized Policy Improvement

- Given an estimate  $Q^{\pi_i}(s, a) \forall s, a$
- Update new policy

$$\pi_{i+1}(s) = \arg \max_a Q^{\pi_i}(s, a) \quad (2)$$

# Model-free Policy Iteration

- Initialize policy  $\pi$
- Repeat:
  - Policy evaluation: compute  $Q^\pi$
  - Policy improvement: update  $\pi$  given  $Q^\pi$
- May need to modify policy evaluation:
  - If  $\pi$  is deterministic, can't compute  $Q(s, a)$  for any  $a \neq \pi(s)$
- How to interleave policy evaluation and improvement?
  - Policy improvement is now using an estimated  $Q$

# Table of Contents

- 1 Generalized Policy Iteration
- 2 Importance of Exploration**
- 3 Monte Carlo Control
- 4 Temporal Difference Methods for Control
- 5 Maximization Bias

# Policy Evaluation with Exploration

- Want to compute a model-free estimate of  $Q^\pi$
- In general seems subtle
  - Need to try all  $(s, a)$  pairs but then follow  $\pi$
  - Want to ensure resulting estimate  $Q^\pi$  is good enough so that policy improvement is a monotonic operator
- For certain classes of policies can ensure all  $(s, a)$  pairs are tried such that asymptotically  $Q^\pi$  converges to the true value

# $\epsilon$ -greedy Policies

- Simple idea to balance exploration and exploitation
- Let  $|A|$  be the number of actions
- Then an  $\epsilon$ -greedy policy w.r.t. a state-action value  $Q^\pi(s, a)$  is  $\pi(a|s) =$

# Monotonic<sup>19</sup> $\epsilon$ -greedy Policy Improvement

## Theorem

For any  $\epsilon$ -greedy policy  $\pi_i$ , the  $\epsilon$ -greedy policy w.r.t.  $Q^{\pi_i}$ ,  $\pi_{i+1}$  is a monotonic improvement  $V^{\pi_{i+1}} \geq V^\pi$

$$\begin{aligned} Q^\pi(s, \pi_{i+1}(s)) &= \sum_{a \in A} \pi_{i+1}(a|s) Q^{\pi_i}(s, a) \\ &= (\epsilon/|A|) \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \max_a Q^{\pi_i}(s, a) \end{aligned}$$

- Therefore  $V^{\pi_{i+1}} \geq V^{\pi_i}$  (from the policy improvement theorem)

<sup>19</sup>The theorem assumes that  $Q^{\pi_i}$  has been computed exactly.



# Monotonic<sup>21</sup> $\epsilon$ -greedy Policy Improvement

## Theorem

For any  $\epsilon$ -greedy policy  $\pi_i$ , the  $\epsilon$ -greedy policy w.r.t.  $Q^{\pi_i}$ ,  $\pi_{i+1}$  is a monotonic improvement  $V^{\pi_{i+1}} \geq V^{\pi_i}$

$$\begin{aligned} Q^{\pi_i}(s, \pi_{i+1}(s)) &= \sum_{a \in A} \pi_{i+1}(a|s) Q^{\pi_i}(s, a) \\ &= (\epsilon/|A|) \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \max_a Q^{\pi_i}(s, a) \\ &= (\epsilon/|A|) \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \max_a Q^{\pi_i}(s, a) \frac{1 - \epsilon}{1 - \epsilon} \\ &= (\epsilon/|A|) \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \max_a Q^{\pi_i}(s, a) \sum_a \frac{\pi_i(a|s) - \frac{\epsilon}{|A|}}{1 - \epsilon} \\ &\geq \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \sum_a \frac{\pi_i(a|s) - \frac{\epsilon}{|A|}}{1 - \epsilon} Q^{\pi_i}(s, a) \\ &= \sum_a \pi_i(a|s) Q^{\pi_i}(s, a) = V^{\pi_i}(s) \end{aligned}$$

- Therefore  $V^{\pi_{i+1}} \geq V^{\pi_i}$  (from the policy improvement theorem)

<sup>21</sup>The theorem assumes that  $Q^{\pi_i}$  has been computed exactly.

# Greedy in the Limit of Infinite Exploration (GLIE)

## Definition of GLIE

- All state-action pairs are visited an infinite number of times

$$\lim_{i \rightarrow \infty} N_i(s, a) \rightarrow \infty$$

- Behavior policy converges to greedy policy
- A simple GLIE strategy is  $\epsilon$ -greedy where  $\epsilon$  is reduced to 0 with the following rate:  $\epsilon_i = 1/i$

# Table of Contents

- 1 Generalized Policy Iteration
- 2 Importance of Exploration
- 3 Monte Carlo Control**
- 4 Temporal Difference Methods for Control
- 5 Maximization Bias

# Monte Carlo Online Control / On Policy Improvement

- 
- 1: Initialize  $Q(s, a) = 0$ ,  $Returns(s, a) = 0 \forall (s, a)$ , Set  $\epsilon = 1$ ,  $k = 1$
  - 2:  $\pi_k = \epsilon$ -greedy( $Q$ ) // Create initial  $\epsilon$ -greedy policy
  - 3: **loop**
  - 4:   Sample  $k$ -th episode  $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \dots, s_T)$  given  $\pi_k$
  - 5:   **for**  $t = 1, \dots, T$  **do**
  - 6:     **if** First visit to  $(s, a)$  in episode  $k$  **then**
  - 7:       Append  $\sum_{j=t}^T r_{kj}$  to  $Returns(s_t, a_t)$
  - 8:        $Q(s_t, a_t) = \text{average}(Returns(s_t, a_t))$
  - 9:     **end if**
  - 10:   **end for**
  - 11:    $k = k + 1$ ,  $\epsilon = 1/k$
  - 12:    $\pi_k = \epsilon$ -greedy( $Q^\pi$ ) // Policy improvement
  - 13: **end loop**
-

## Theorem

GLIE Monte-Carlo control converges to the optimal state-action value<sup>a</sup> function  $Q(s, a) \rightarrow q(s, a)$

---

<sup>a</sup> $v(s)$  and  $q(s, a)$  without any additional subscripts are used to indicate the optimal state and state-action value function, respectively.

# Model-free Policy Iteration

- Initialize policy  $\pi$
- Repeat:
  - Policy evaluation: compute  $Q^\pi$
  - Policy improvement: update  $\pi$  given  $Q^\pi$
- What about TD methods?

# Table of Contents

- 1 Generalized Policy Iteration
- 2 Importance of Exploration
- 3 Monte Carlo Control
- 4 Temporal Difference Methods for Control**
- 5 Maximization Bias

# Model-free Policy Iteration with TD Methods

- Use temporal difference methods for policy evaluation step
- Initialize policy  $\pi$
- Repeat:
  - Policy evaluation: compute  $Q^\pi$  using temporal difference updating with  $\epsilon$ -greedy policy
  - Policy improvement: Same as Monte carlo policy improvement, set  $\pi$  to  $\epsilon$ -greedy ( $Q^\pi$ )



# General Form of SARSA Algorithm

- 
- 1: Set initial  $\epsilon$ -greedy policy  $\pi$ ,  $t = 0$ , initial state  $s_t = s_0$
  - 2: Take  $a_t \sim \pi(s_t)$  // Sample action from policy
  - 3: Observe  $(r_t, s_{t+1})$
  - 4: **loop**
  - 5:   Take action  $a_{t+1} \sim \pi(s_{t+1})$
  - 6:   Observe  $(r_{t+1}, s_{t+2})$
  - 7:   Update Q given  $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ :
  - 8:   Perform policy improvement:
  - 9:    $t = t + 1$
  - 10: **end loop**
- 

- What are the benefits to improving the policy after each step?

• What are the benefits to updating the policy less frequently?

# Convergence Properties of SARSA

## Theorem

Sarsa for finite-state and finite-action MDPs converges to the optimal action-value,  $Q(s, a) \rightarrow q(s, a)$ , under the following conditions:

- 1 The policy sequence  $\pi_t(a|s)$  satisfies the condition of GLIE
- 2 The step-sizes  $\alpha_t$  satisfy the Robbins-Munro sequence such that

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$
$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

# Recall: Off Policy, Policy Evaluation

- Given data from following a behavior policy  $\pi_b$  can we estimate the value  $V^{\pi_e}$  of an alternate policy  $\pi_e$ ?
- Neat idea: can we learn about other ways to do things different than what we actually did?
- Discussed how to do this for Monte Carlo evaluation
- Used Importance Sampling
- First see how to do off policy evaluation with TD

# Importance Sampling for Off Policy TD (Policy Evaluation)

- Recall the Temporal Difference (TD) algorithm which is used to incremental model-free evaluation of a policy  $\pi_b$ . Precisely, given a state  $s_t$ , an action  $a_t$  sampled from  $\pi_b(s_t)$  and the observed reward  $r_t$  and next state  $s_{t+1}$ , TD performs the following update:

$$V^{\pi_b}(s_t) = V^{\pi_b}(s_t) + \alpha(r_t + \gamma V^{\pi_b}(s_{t+1}) - V^{\pi_b}(s_t)) \quad (3)$$

- Now want to use data generated from following  $\pi_b$  to estimate the value of different policy  $\pi_e$ ,  $V^{\pi_e}$
- Change TD target  $r_t + \gamma V(s_{t+1})$  to weight target by single importance sample ratio
- New update:

$$V^{\pi_e}(s_t) = V^{\pi_e}(s_t) + \alpha \left[ \frac{\pi_e(a_t|s_t)}{\pi_b(a_t|s_t)} (r_t + \gamma V^{\pi_e}(s_{t+1}) - V^{\pi_e}(s_t)) \right] \quad (4)$$

# Importance Sampling for Off Policy TD Cont.

- Off Policy TD Update:

$$V^{\pi_e}(s_t) = V^{\pi_e}(s_t) + \alpha \left[ \frac{\pi_e(a_t|s_t)}{\pi_b(a_t|s_t)} (r_t + \gamma V^{\pi_e}(s_{t+1}) - V^{\pi_e}(s_t)) \right] \quad (5)$$

- Significantly lower variance than MC IS. (Why?)
- Does  $\pi_b$  need to be the same at each time step?
- What conditions on  $\pi_b$  and  $\pi_e$  are needed for off policy TD to converge to  $V^{\pi_e}$ ?

# Q-Learning: Learning the Optimal State-Action Value

- Just saw how to use off policy TD to evaluate any particular policy  $\pi_e$
- Can we estimate the value of the optimal policy  $\pi^*$  without knowledge of what  $\pi^*$  is?
- Yes! Q-learning
- Does not require importance sampling
- Key idea: Maintain state-action  $Q$  estimates and use to bootstrap—use the value of the best future action
- Recall Sarsa

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha((r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)) \quad (6)$$

- Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha((r_t + \gamma \max_{a'} Q(s_{t+1}, a')) - Q(s_t, a_t)) \quad (7)$$

# Off-Policy Control Using Q-learning

- In the prior slide assumed there was some  $\pi_b$  used to act
- $\pi_b$  determines the actual rewards received
- Now consider how to improve the behavior policy (policy improvement)
- Let behavior policy  $\pi_b$  be  $\epsilon$ -greedy with respect to (w.r.t.) current estimate of the optimal  $q(s, a)$

# Q-Learning with $\epsilon$ -greedy Exploration

- 
- 1: Initialize  $Q(s, a), \forall s \in S, a \in A$   $t = 0$ , initial state  $s_t = s_0$
  - 2: Set  $\pi_b$  to be  $\epsilon$ -greedy w.r.t.  $Q$
  - 3: **loop**
  - 4:   Take  $a_t \sim \pi_b(s_t)$  // Sample action from policy
  - 5:   Observe  $(r_t, s_{t+1})$
  - 6:   Update  $Q$  given  $(s_t, a_t, r_t, s_{t+1})$ :
  
  - 7:   Perform policy improvement: set  $\pi_b$  to be  $\epsilon$ -greedy w.r.t.  $Q$
  - 8:    $t = t + 1$
  - 9: **end loop**
- 

- What conditions are sufficient to ensure that Q-learning with  $\epsilon$ -greedy exploration converges to optimal  $q$ ?
- What conditions are sufficient to ensure that Q-learning with  $\epsilon$ -greedy exploration converges to optimal  $\pi^*$ ?



# Table of Contents

- 1 Generalized Policy Iteration
- 2 Importance of Exploration
- 3 Monte Carlo Control
- 4 Temporal Difference Methods for Control
- 5 Maximization Bias

# Maximization Bias<sup>39</sup>

- Consider single-state MDP ( $|S| = 1$ ) with 2 actions, and both actions have 0-mean random rewards, ( $\mathbb{E}(r|a = a_1) = \mathbb{E}(r|a = a_2) = 0$ ).
- Then  $Q(s, a_1) = Q(s, a_2) = 0 = V(s)$
- Assume there are prior samples of taking action  $a_1$  and  $a_2$
- Let  $\hat{Q}(s, a_1), \hat{Q}(s, a_2)$  be the finite sample estimate of  $Q$
- Assume using an unbiased estimator for  $Q$ : e.g.  
$$\hat{Q}(s, a_1) = \frac{1}{n(s, a_1)} \sum_{i=1}^{n(s, a_1)} r_i(s, a_1)$$
- Let  $\hat{\pi} = \arg \max_a \hat{Q}(s, a)$  be the greedy policy w.r.t. the estimated  $\hat{Q}$
- *Even though each estimate of the state-action values is unbiased, the estimate of  $\hat{\pi}$ 's value  $\hat{V}^{\hat{\pi}}$  can be biased:*

---

<sup>39</sup>Example from Mannor, Simester, Sun and Tsitsiklis. Bias and Variance Approximation in Value Function Estimates. Management Science 2007

# Double Learning

- The greedy policy w.r.t. estimated  $Q$  values can yield a maximization bias during finite-sample learning
- Avoid using max of estimates as estimate of max of true values
- Instead split samples and use to create two independent unbiased estimates of  $Q_1(s_1, a_i)$  and  $Q_2(s_1, a_i) \forall a$ .
  - Use one estimate to select max action:  $a^* = \arg \max_a Q_1(s_1, a)$
  - Use other estimate to estimate value of  $a^*$ :  $Q_2(s, a^*)$
  - Yields unbiased estimate:  $\mathbb{E}(Q_2(s, a^*)) = Q(s, a^*)$
- Why does this yield an unbiased estimate of the max state-action value?
- If acting online, can alternate samples used to update  $Q_1$  and  $Q_2$ , using the other to select the action chosen
- Next slides extend to full MDP case (with more than 1 state)

# Double Q-Learning

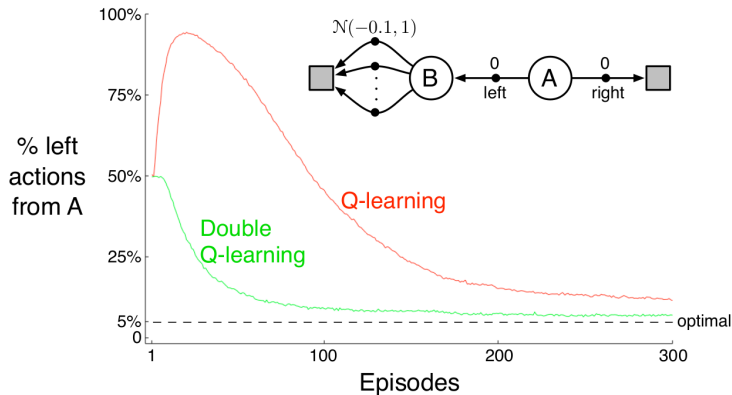
---

```
1: Initialize  $Q_1(s, a)$  and  $Q_2(s, a), \forall s \in S, a \in A$   $t = 0$ , initial state  $s_t = s_0$ 
2: loop
3:   Select  $a_t$  using  $\epsilon$ -greedy  $\pi(s) = \arg \max_a Q_1(s_t, a) + Q_2(s_t, a)$ 
4:   Observe  $(r_t, s_{t+1})$ 
5:   if (with 0.5 probability) then
6:      $Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha$ 
7:   else
8:      $Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha$ 
9:   end if
10:   $t = t + 1$ 
11: end loop
```

---

- Compared to Q-learning, how does this change the: memory requirements, computation requirements per step, amount of data required?

# Double Q-Learning (Figure 6.7 in Sutton and Barto 2018)



Due to the maximization bias, Q-learning spends much more time selecting suboptimal actions than double Q-learning.

# Table of Contents

- 1 Generalized Policy Iteration
- 2 Importance of Exploration
- 3 Monte Carlo Control
- 4 Temporal Difference Methods for Control
- 5 Maximization Bias

# Class Structure

- Last time: Policy evaluation with no knowledge of how the world works (MDP model not given)
- This time: Control (making decisions) without a model of how the world works
- **Next time: Value function approximation and Deep Q-learning**

# Lecture 5: Value Function Approximation

Emma Brunskill

CS234 Reinforcement Learning.

Winter 2018

The value function approximation structure for today closely follows much of David Silver's Lecture 6. For additional reading please see SB 2018 Sections 9.3, 9.6-9.7. The deep learning slides come almost exclusively from Ruslan Salakhutdinov's class, and Hugo Larochelle's class (and with thanks to Zico Kolter also for slide inspiration). The slides in my standard style format in the deep learning section are my own.



# Table of Contents

- 1 Introduction
- 2 VFA for Prediction
- 3 Control using Value Function Approximation
- 4 Deep Learning

# Class Structure

- Last time: Control (making decisions) without a model of how the world works
- **This time: Value function approximation and deep learning**
- Next time: Deep reinforcement learning

# Last time: Model-Free Control

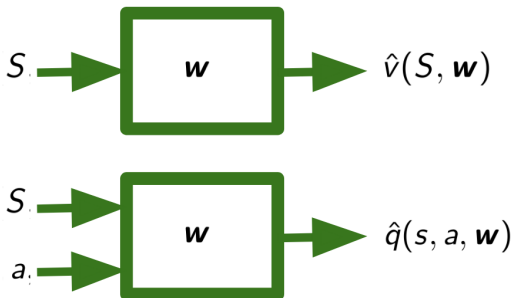
- Last time: how to learn a good policy from experience
- So far, have been assuming we can represent the value function or state-action value function as a vector
  - Tabular representation
- Many real world problems have enormous state and/or action spaces
- Tabular representation is insufficient

# Table of Contents

- 1 Introduction
- 2 VFA for Prediction
- 3 Control using Value Function Approximation
- 4 Deep Learning

# Value Function Approximation (VFA)

- Represent a (state-action/state) value function with a parameterized function instead of a table



# Motivation for VFA

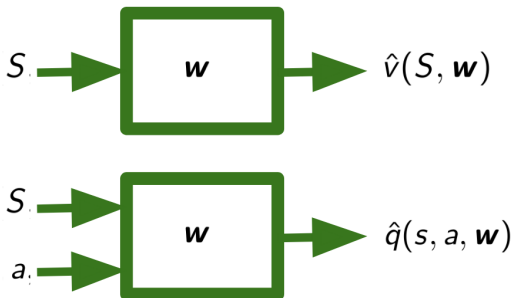
- Don't want to have to explicitly store or learn for every single state a
  - Dynamics or reward model
  - Value
  - State-action value
  - Policy
- Want more compact representation that generalizes across state or states and actions

# Benefits of Generalization

- 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 (VFA)

- Represent a (state-action/state) value function with a parameterized function instead of a table



- Which function approximator?



# Function Approximators

- Many possible function approximators including
  - Linear combinations of features
  - Neural networks
  - Decision trees
  - Nearest neighbors
  - Fourier / wavelet bases
- In this class we will focus on function approximators that are differentiable (Why?)
- Two very popular classes of differentiable function approximators
  - Linear feature representations (Today)
  - Neural networks (Today and next lecture)

# Review: Gradient Descent

- Consider a function  $J(\mathbf{w})$  that is a differentiable function of a parameter vector  $\mathbf{w}$
- Goal is to find parameter  $\mathbf{w}$  that minimizes  $J$
- The gradient of  $J(\mathbf{w})$  is

# Table of Contents

- 1 Introduction
- 2 VFA for Prediction
- 3 Control using Value Function Approximation
- 4 Deep Learning

# Value Function Approximation for Policy Evaluation with an Oracle

- First consider if could query any state  $s$  and an oracle would return the true value for  $v^\pi(s)$
- The objective was to find the best approximate representation of  $v^\pi$  given a particular parameterized function

# Stochastic Gradient Descent

- Goal: Find the parameter vector  $\mathbf{w}$  that minimizes the loss between a true value function  $v_\pi(s)$  and its approximation  $\hat{v}$  as represented with a particular function class parameterized by  $\mathbf{w}$ .
- Generally use mean squared error and define the loss as

$$J(\mathbf{w}) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2] \quad (1)$$

- Can use gradient descent to find a local minimum

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

- Stochastic gradient descent (SGD) samples the gradient:
- Expected SGD is the same as the full gradient update

# VFA Prediction Without An Oracle

- Don't actually have access to an oracle to tell true  $v_\pi(S)$  for any state  $s$
- Now consider how to do value function approximation for prediction / evaluation / policy evaluation without a model
- Note: policy evaluation without a model is sometimes also called **passive reinforcement learning** with value function approximation
  - "passive" because not trying to learn the optimal decision policy

# Model Free VFA Prediction / Policy Evaluation

- Recall model-free policy evaluation (Lecture 3)
  - Following a fixed policy  $\pi$  (or had access to prior data)
  - Goal is to estimate  $V^\pi$  and/or  $Q^\pi$
- Maintained a look up table to store estimates  $V^\pi$  and/or  $Q^\pi$
- 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**

# Feature Vectors

- Use a feature vector to represent a state

$$x(s) = \begin{pmatrix} x_1(s) \\ x_2(s) \\ \dots \\ x_n(s) \end{pmatrix} \quad (3)$$



# Linear Value Function Approximation for Prediction With An Oracle

- Represent a value function (or state-action value function) for a particular policy with a weighted linear combination of features

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

- Objective function is

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

- Recall weight update is

$$\Delta(\mathbf{w}) = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \quad (4)$$

- Update is:

- Update = step-size  $\times$  prediction error  $\times$  feature value

# Monte Carlo Value Function Approximation

- Return  $G_t$  is an unbiased but noisy sample of the true expected return  $v_\pi(S_t)$
- Therefore can reduce MC VFA to doing supervised learning on a set of (state,return) pairs:  $\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, \dots, \langle S_T, G_T \rangle$ 
  - Substituting  $G_t(S_t)$  for the true  $v_\pi(S_t)$  when fitting the function approximator
- Concretely when using linear VFA for policy evaluation

$$\Delta \mathbf{w} = \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \quad (5)$$

$$= \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \mathbf{x}(S_t) \quad (6)$$

- Note:  $G_t$  may be a very noisy estimate of true return

# MC Linear Value Function Approximation for Policy Evaluation

---

```
1: Initialize  $\mathbf{w} = \mathbf{0}, Returns(s) = 0 \forall (s, a), k = 1$ 
2: loop
3:   Sample  $k$ -th episode  $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \dots, s_{k,L_k})$  given  $\pi$ 
4:   for  $t = 1, \dots, L_k$  do
5:     if First visit to  $(s)$  in episode  $k$  then
6:       Append  $\sum_{j=t}^{L_k} r_{kj}$  to  $Returns(s_t)$ 
7:       Update weights
8:     end if
9:   end for
10:   $k = k + 1$ 
11: end loop
```

---

# Recall: Temporal Difference (TD(0)) Learning with a Look up Table

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

$$V^\pi(s) = V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s)) \quad (7)$$

- Target is  $r + \gamma V^\pi(s')$ , a biased estimate of the true value  $v^\pi(s)$
- Look up table represents value for each state with a separate table entry

# Temporal Difference (TD(0)) Learning with Value Function Approximation

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

$$V^\pi(s) = V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s)) \quad (8)$$

- Target is  $r + \gamma V^\pi(s')$ , a biased estimate of the true value  $v^\pi(s)$
- In value function approximation, target is  $r + \gamma \hat{v}^\pi(s')$ , a biased and approximated estimate of the true value  $v^\pi(s)$
- 3 forms of approximation:

# Temporal Difference (TD(0)) Learning with Value Function Approximation

- In value function approximation, target is  $r + \gamma \hat{v}^\pi(s')$ , a biased and approximated estimate of the true value  $v^\pi(s)$
- Supervised learning on a different set of data pairs:  
 $\langle S_1, r_1 + \gamma \hat{v}^\pi(S_2, \mathbf{w}) \rangle, \langle S_2, r_2 + \gamma \hat{v}^\pi(S_3, \mathbf{w}) \rangle, \dots$

# Temporal Difference (TD(0)) Learning with Value Function Approximation

- In value function approximation, target is  $r + \gamma \hat{v}^\pi(s')$ , a biased and approximated estimate of the true value  $v^\pi(s)$
- Supervised learning on a different set of data pairs:  
 $\langle S_1, r_1 + \gamma \hat{v}^\pi(S_2, \mathbf{w}) \rangle, \langle S_2, r_2 + \gamma \hat{v}^\pi(S_3, \mathbf{w}) \rangle, \dots$
- In linear TD(0)

$$\Delta \mathbf{w} = \alpha (r + \gamma \hat{v}^\pi(s', \mathbf{w}) - \hat{v}^\pi(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}^\pi(s, \mathbf{w}) \quad (9)$$

$$= \alpha (r + \gamma \hat{v}^\pi(s', \mathbf{w}) - \hat{v}^\pi(s, \mathbf{w})) \mathbf{x}(s) \quad (10)$$

# Convergence Guarantees for Linear Value Function Approximation for Policy Evaluation<sup>1</sup>

- Define the mean squared error of a linear value function approximation for a particular policy  $\pi$  relative to the true value as

$$MSVE(\mathbf{w}) = \sum_{\mathbf{s} \in \mathbf{S}} \mathbf{d}(\mathbf{s}) (\mathbf{v}^{\pi}(\mathbf{s}) - \hat{\mathbf{v}}^{\pi}(\mathbf{s}, \mathbf{w}))^2 \quad (11)$$

- where
  - $d(s)$ : stationary distribution of  $\pi$  in the true decision process
  - $v, \hat{w}^{\pi}(s) = \mathbf{x}(s)^T \mathbf{w}$ , a linear value function approximation

---

<sup>1</sup>Tsitsiklis and Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. 1997. <https://web.stanford.edu/~bvr/pubs/td.pdf>



# Convergence Guarantees for Linear Value Function Approximation for Policy Evaluation<sup>2</sup>

- Define the mean squared error of a linear value function approximation for a particular policy  $\pi$  relative to the true value as

$$MSVE(\mathbf{w}) = \sum_{s \in S} \mathbf{d}(s) (\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2 \quad (12)$$

- where
  - $d(s)$ : stationary distribution of  $\pi$  in the true decision process
  - $\hat{\mathbf{v}}^\pi(s) = \mathbf{x}(s)^T \mathbf{w}$ , a linear value function approximation
- Monte Carlo policy evaluation with VFA converges to the weights  $\mathbf{w}_{MC}$  which has the minimum mean squared error possible:

$$MSVE(\mathbf{w}_{MC}) = \min_{\mathbf{w}} \sum_{s \in S} d(s) (\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2 \quad (13)$$

---

<sup>2</sup>Tsitsiklis and Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. 1997. <https://web.stanford.edu/~bvr/pubs/td.pdf>

# Convergence Guarantees for Linear Value Function Approximation for Policy Evaluation<sup>3</sup>

- Define the mean squared error of a linear value function approximation for a particular policy  $\pi$  relative to the true value as

$$MSVE(\mathbf{w}) = \sum_{\mathbf{s} \in \mathcal{S}} d(\mathbf{s}) (\mathbf{v}^\pi * (\mathbf{s}) - \hat{\mathbf{v}}^\pi(\mathbf{s}, \mathbf{w}))^2 \quad (14)$$

- where
  - $d(\mathbf{s})$ : stationary distribution of  $\pi$  in the true decision process
  - $\hat{\mathbf{v}}^\pi(\mathbf{s}) = \mathbf{x}(\mathbf{s})^T \mathbf{w}$ , a linear value function approximation
- TD(0) policy evaluation with VFA converges to weights  $\mathbf{w}_{TD}$  which is within a constant factor of the minimum mean squared error possible:

$$MSVE(\mathbf{w}_{TD}) = \frac{1}{1 - \gamma} \min_{\mathbf{w}} \sum_{\mathbf{s} \in \mathcal{S}} d(\mathbf{s}) (\mathbf{v}^\pi * (\mathbf{s}) - \hat{\mathbf{v}}^\pi(\mathbf{s}, \mathbf{w}))^2 \quad (15)$$

---

<sup>3</sup>ibed.

# Summary: Convergence Guarantees for Linear Value Function Approximation for Policy Evaluation<sup>4</sup>

- Monte Carlo policy evaluation with VFA converges to the weights  $\mathbf{w}_{MC}$  which has the minimum mean squared error possible:

$$MSVE(\mathbf{w}_{MC}) = \min_{\mathbf{w}} \sum_{s \in S} d(s) (v^\pi * (s) - \hat{v}^\pi(s, \mathbf{w}))^2 \quad (16)$$

- TD(0) policy evaluation with VFA converges to weights  $\mathbf{w}_{TD}$  which is within a constant factor of the minimum mean squared error possible:

$$MSVE(\mathbf{w}_{TD}) = \frac{1}{1 - \gamma} \min_{\mathbf{w}} \sum_{s \in S} d(s) (v^\pi * (s) - \hat{v}^\pi(s, \mathbf{w}))^2 \quad (17)$$

- Check your understanding: if the VFA is a tabular representation (one feature for each state), what is the MSVE for MC and TD?

---

<sup>4</sup>ibed.

# Convergence Rates for Linear Value Function Approximation for Policy Evaluation

- Does TD or MC converge faster to a fixed point?
- Not (to my knowledge) definitively understood
- Practically TD learning often converges faster to its fixed value function approximation point

# Table of Contents

- 1 Introduction
- 2 VFA for Prediction
- 3 Control using Value Function Approximation**
- 4 Deep Learning

# Control using Value Function Approximation

- Use value function approximation to represent state-action values  
 $\hat{q}^{\pi}(s, a, \mathbf{w}) \approx q^{\pi}$
- Interleave
  - Approximate policy evaluation using value function approximation
  - Perform  $\epsilon$ -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:

$$J(\mathbf{w}) = \mathbb{E}_\pi[(q^\pi(s, a) - \hat{q}^\pi(s, a, \mathbf{w}))^2] \quad (18)$$

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) = \mathbb{E}[(q^\pi(s, a) - \hat{q}^\pi(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}^\pi(s, a, \mathbf{w})] \quad (19)$$

$$\Delta(\mathbf{w}) = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \quad (20)$$

- Stochastic gradient descent (SGD) samples the gradient

# Linear State Action Value Function Approximation with an Oracle

- Use features to represent both the state and action

$$x(s, a) = \begin{pmatrix} x_1(s, a) \\ x_2(s, a) \\ \dots \\ x_n(s, a) \end{pmatrix} \quad (21)$$

- Represent state-action value function with a weighted linear combination of features

$$\hat{q}(s, a, \mathbf{w}) = x(s, a)^T \mathbf{w} = \sum_{j=1}^n x_j(s, a) w_j \quad (22)$$

- Stochastic gradient descent update:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \nabla_{\mathbf{w}} \mathbb{E}_{\pi} [(q^{\pi}(s, a) - \hat{q}^{\pi}(s, a, \mathbf{w}))^2] \quad (23)$$



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

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(s_t, a_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s_t, a_t, \mathbf{w}) \quad (24)$$

- For SARSA instead use a TD target  $r + \gamma \hat{q}(s', a', \mathbf{w})$  which leverages the current function approximation value

$$\Delta \mathbf{w} = \alpha(r + \gamma \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) \quad (25)$$

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

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(s_t, a_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s_t, a_t, \mathbf{w}) \quad (26)$$

- For SARSA instead use a TD target  $r + \gamma \hat{q}(s', a', \mathbf{w})$  which leverages the current function approximation value

$$\Delta \mathbf{w} = \alpha(r + \gamma \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) \quad (27)$$

- For Q-learning instead use a TD target  $r + \gamma \max_a \hat{q}(s', a', \mathbf{w})$  which leverages the max of the current function approximation value

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) \quad (28)$$

# Convergence of TD Methods with VFA

- TD with value function approximation is not following the gradient of an objective function
- 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

# Convergence of Control Methods with VFA

Algorithm	Tabular	Linear VFA	Nonlinear VFA
Monte-Carlo Control			
Sarsa			
Q-learning			

# Table of Contents

- 1 Introduction
- 2 VFA for Prediction
- 3 Control using Value Function Approximation
- 4 Deep Learning

# Other Function Approximators

- Linear value function approximators often work 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
- Local representations including Kernel based approaches have some appealing properties (including convergence results under certain cases) but can't typically scale well to enormous spaces and datasets
- Alternative is to leverage huge recent success in using deep neural networks

# Deep Neural Networks

- Today: a brief introduction to deep neural networks
- Definitions
- Power of deep neural networks
  - Neural networks / distributed representations vs kernel / local representations
  - Universal function approximator
  - Deep neural networks vs shallow neural networks
- How to train neural nets

# Deep Neural Networks

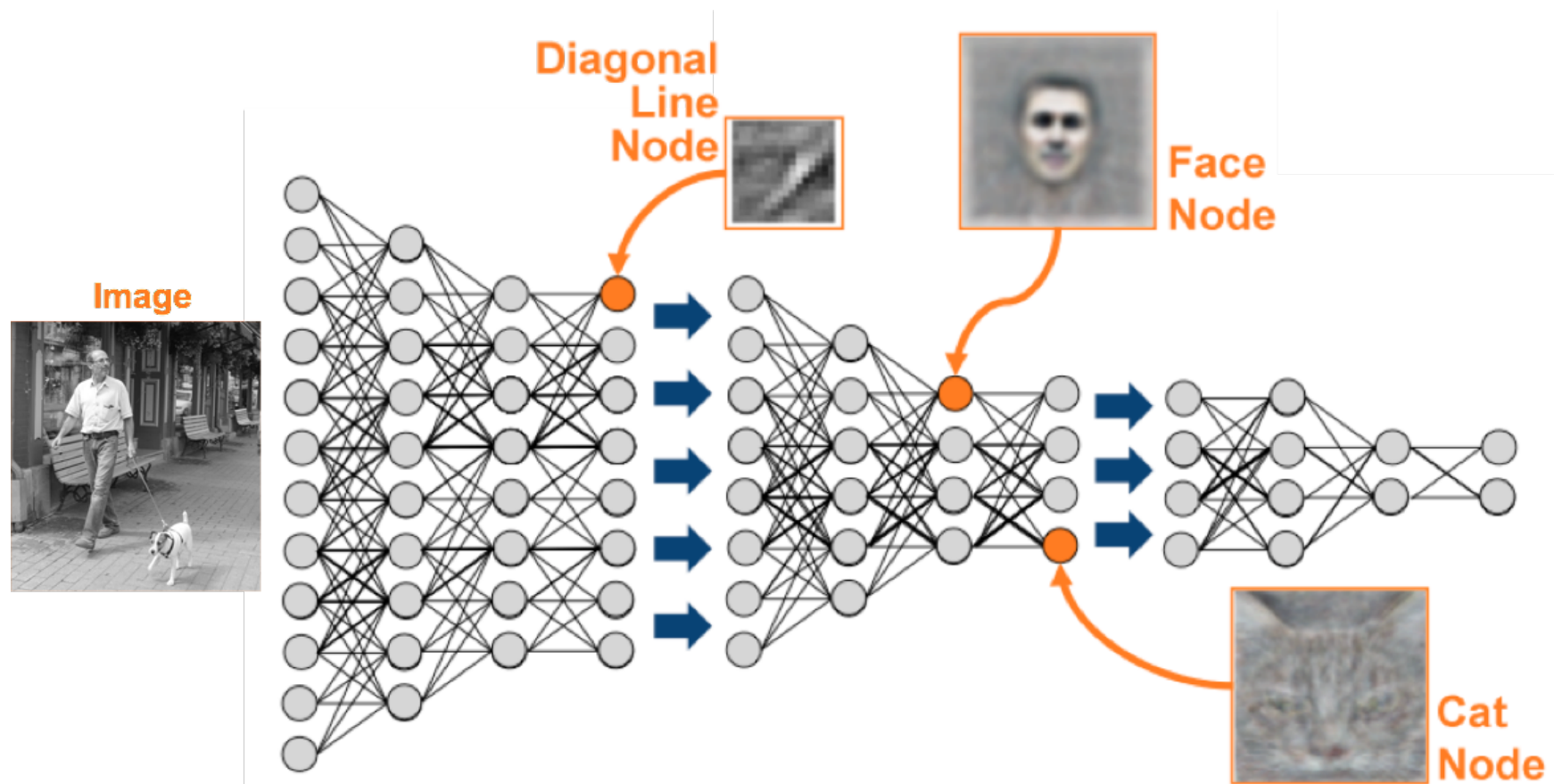
- Today: a brief introduction to deep neural networks
- Definitions
- Power of deep neural networks
  - Neural networks / distributed representations vs kernel / local representations
  - Universal function approximator
  - Deep neural networks vs shallow neural networks
- How to train neural nets



# Deep Neural Networks

- Today: a brief introduction to deep neural networks
- **Definitions**
- Power of deep neural networks
  - Neural networks / distributed representations vs kernel / local representations
  - Universal function approximator
  - Deep neural networks vs shallow neural networks
- How to train neural nets

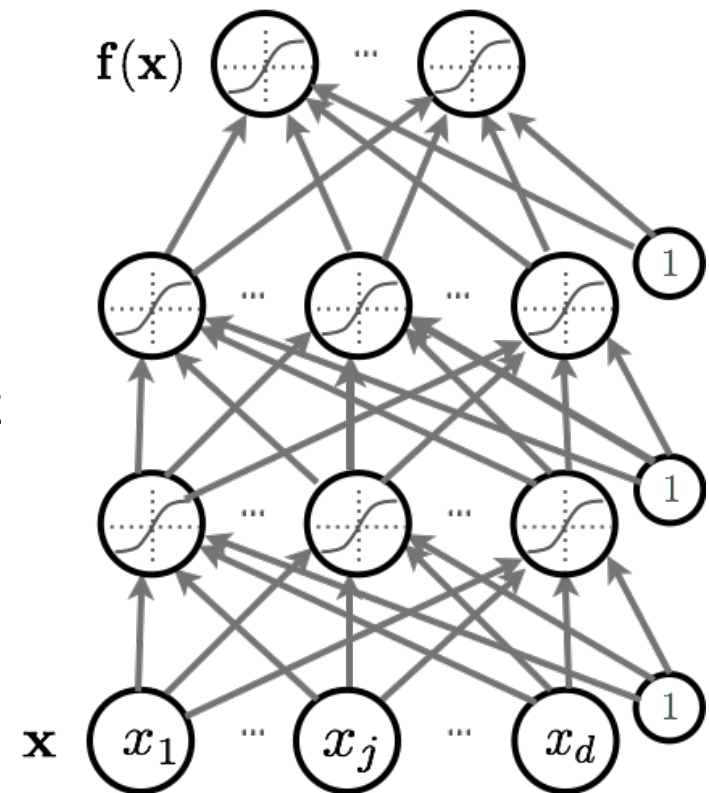
# Deep Learning



- Today: a brief introduction to deep neural networks
- Definitions
- **Power of deep neural networks**
  - Neural networks / distributed representations vs kernel / local representations
  - Universal function approximator
  - Deep neural networks vs shallow neural networks
- How to train neural nets

# Feedforward Neural Networks

- ▶ Definition of Neural Networks
  - Forward propagation
  - Types of units
  - Capacity of neural networks
- ▶ How to train neural nets:
  - Loss function
  - Backpropagation with gradient descent
- ▶ More recent techniques:
  - Dropout
  - Batch normalization
  - Unsupervised Pre-training



# Artificial Neuron

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

- Neuron output activation:

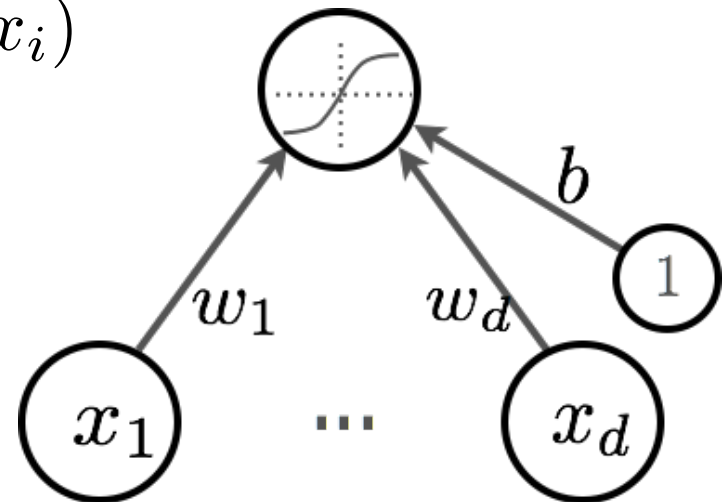
$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

where

$\mathbf{W}$  are the weights (parameters)

$b$  is the bias term

$g(\cdot)$  is called the activation function



# Single Hidden Layer Neural Net

- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$

$$\left(a(\mathbf{x})_i = b_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j\right)$$

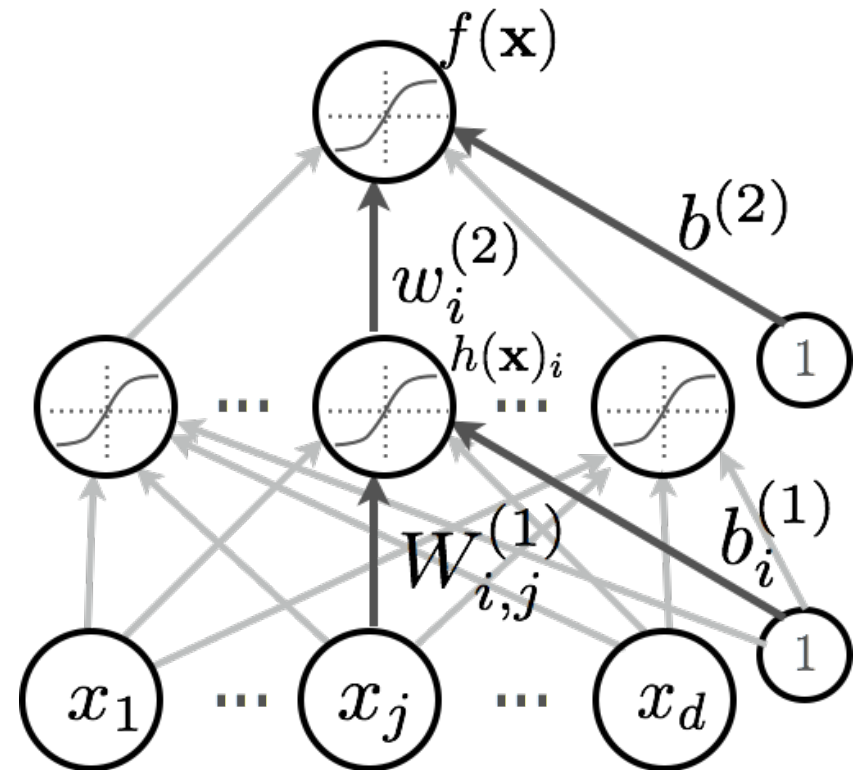
- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o \left( b^{(2)} + \mathbf{w}^{(2)\top} \mathbf{h}^{(1)} \mathbf{x} \right)$$

Output activation  
function

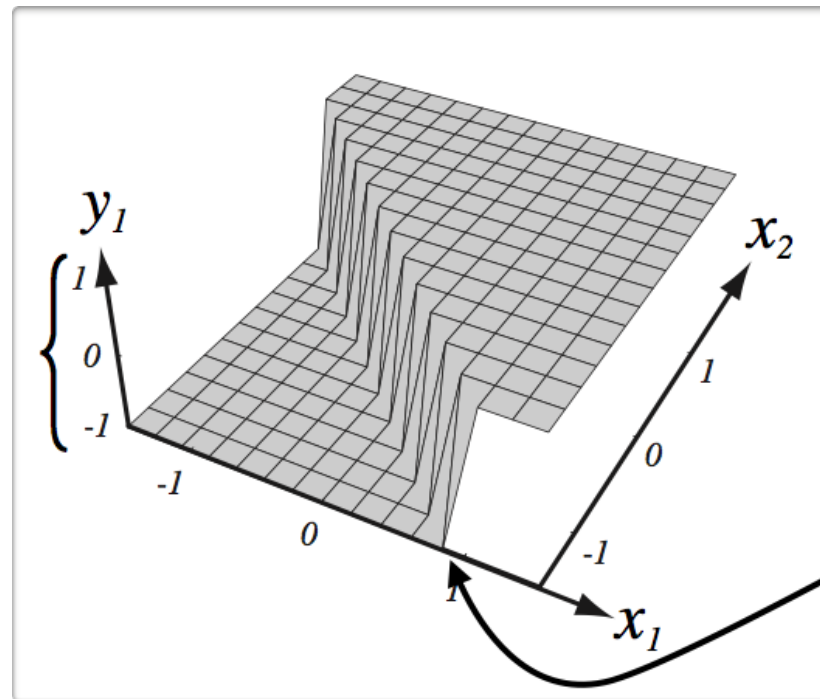


# Artificial Neuron

- Output activation of the neuron:

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

Range is  
determined  
by  $g(\cdot)$



(from Pascal Vincent's slides)

Bias only changes  
the position of the  
riff

# Single Hidden Layer Neural Net

- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$

$$\left(a(\mathbf{x})_i = b_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j\right)$$

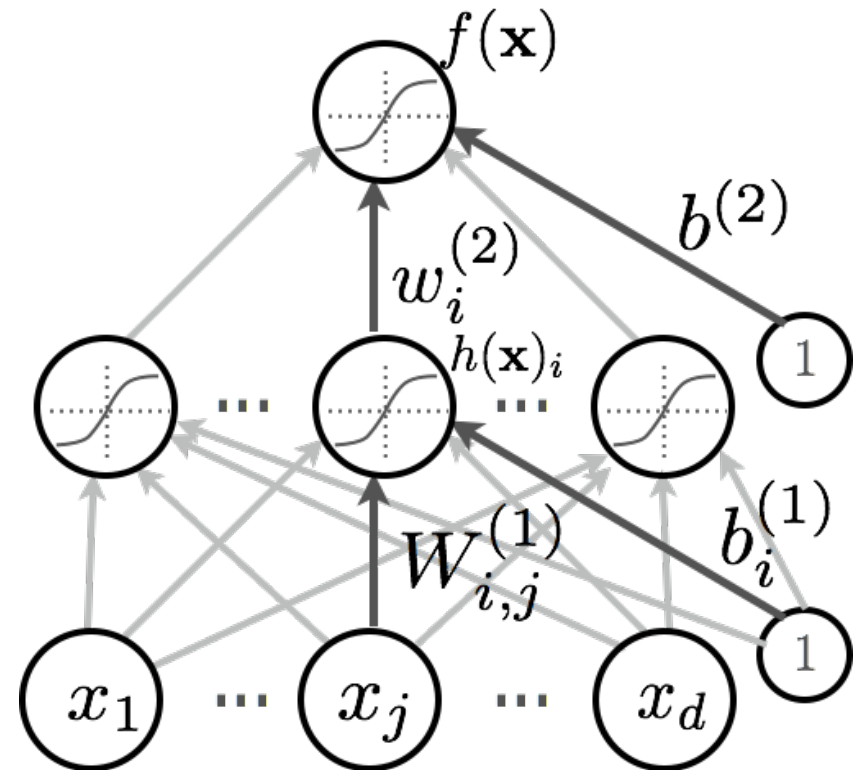
- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o \left( b^{(2)} + \mathbf{w}^{(2)\top} \mathbf{h}^{(1)} \mathbf{x} \right)$$

Output activation  
function



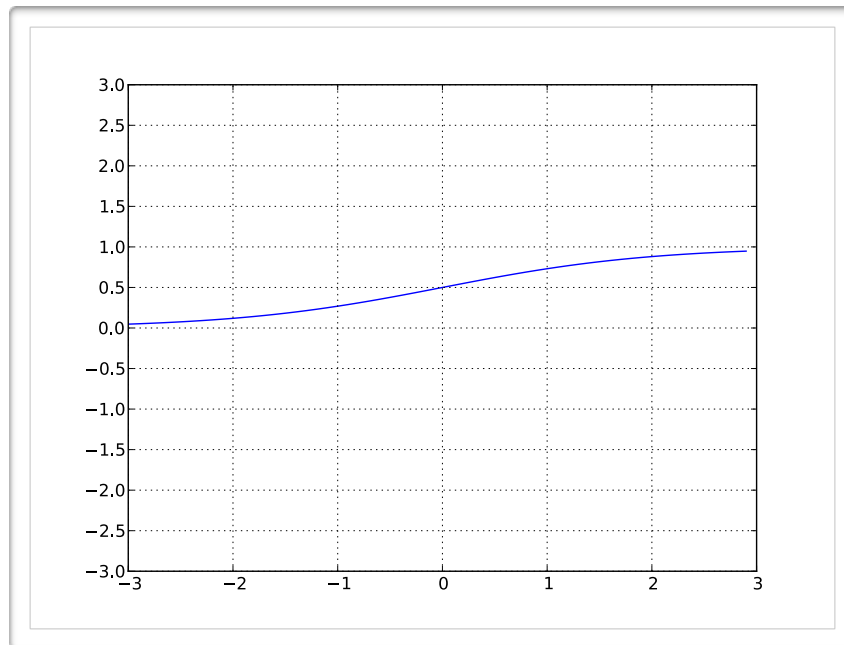


# Activation Function

- Sigmoid activation function:

- Squashes the neuron's output between 0 and 1
- Always positive
- Bounded
- Strictly Increasing

$$g(a) = \text{sigm}(a) = \frac{1}{1 + \exp(-a)}$$

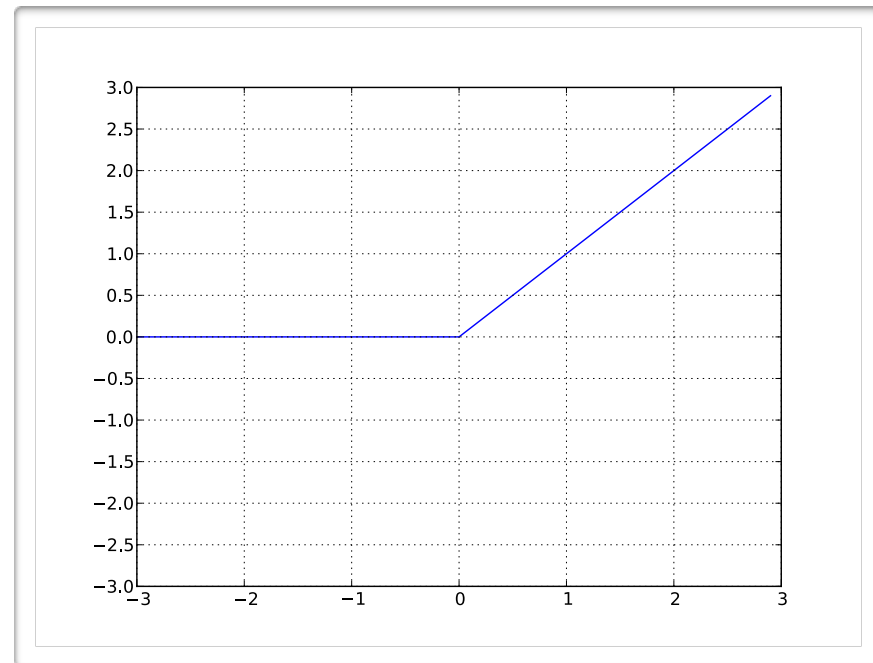


# Activation Function

- Rectified linear (ReLU) activation function:

- Bounded below by 0 (always non-negative)
- Tends to produce units with sparse activities
- Not upper bounded
- Strictly increasing

$$g(a) = \text{reclin}(a) = \max(0, a)$$



# Multilayer Neural Net

- Consider a network with  $L$  hidden layers.

- layer pre-activation for  $k > 0$

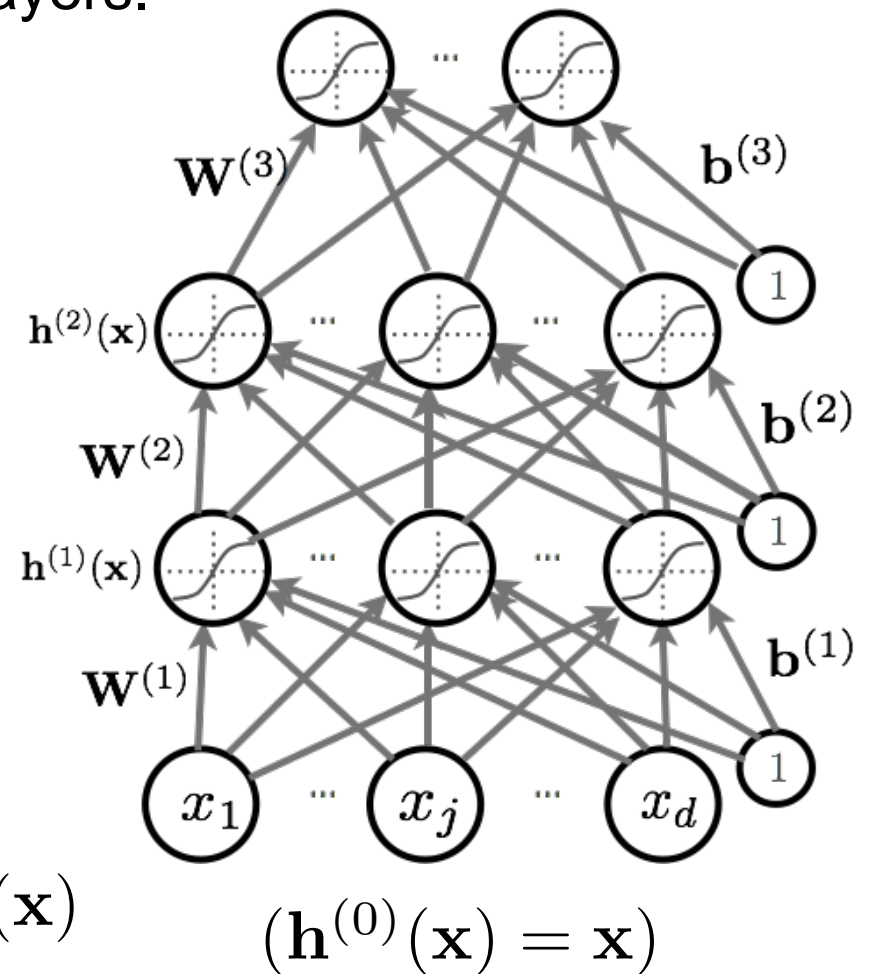
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation from 1 to  $L$ :

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ( $k=L+1$ ):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

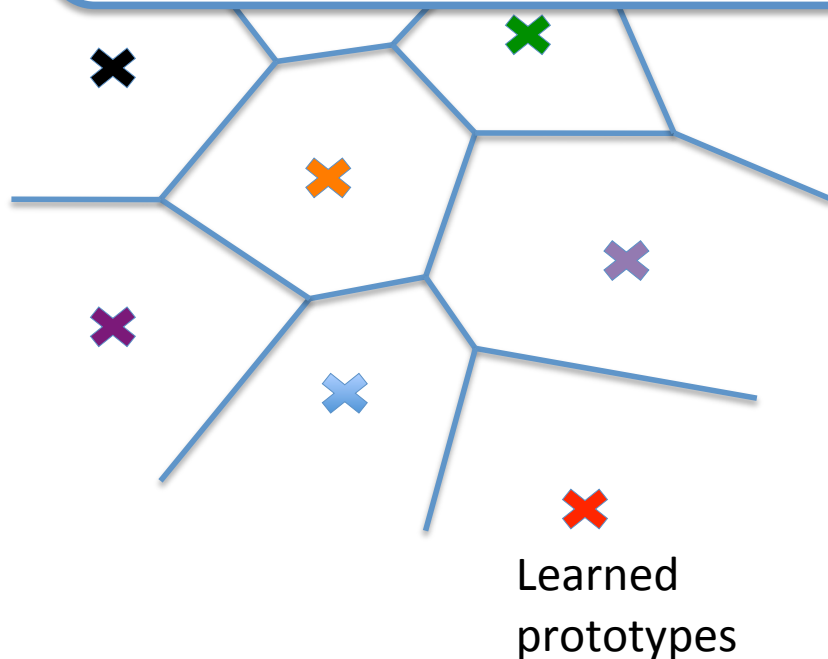


- Today: a brief introduction to deep neural networks
- Definitions
- **Power of deep neural networks**
  - Neural networks / distributed representations vs kernel / local representations
  - Universal function approximator
  - Deep neural networks vs shallow neural networks
- How to train neural nets

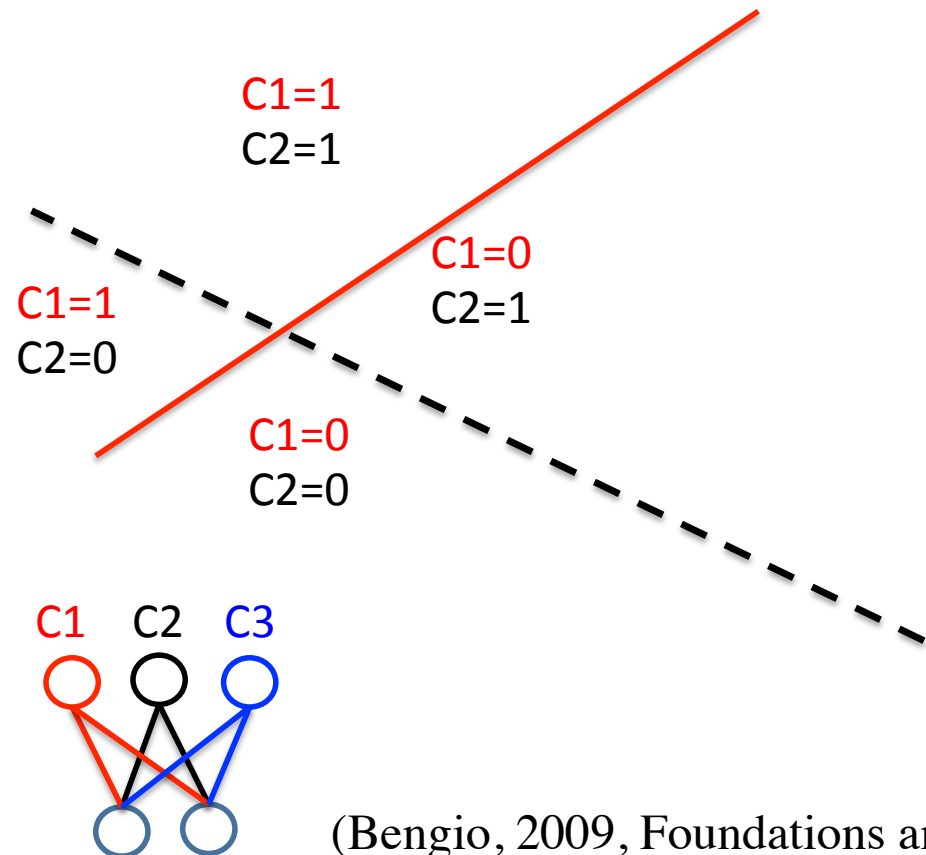
# Local vs. Distributed Representations

- Clustering, Nearest Neighbors, RBF SVM, local density estimators

- Parameters for each region.
- # of regions is linear with # of parameters.



- RBMs, Factor models, PCA, Sparse Coding, Deep models

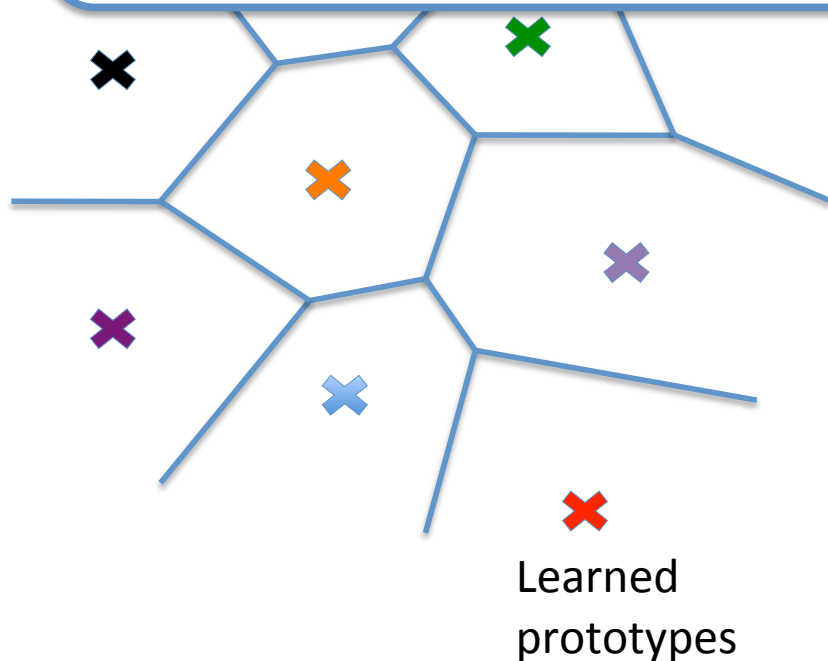


(Bengio, 2009, Foundations and Trends in Machine Learning)

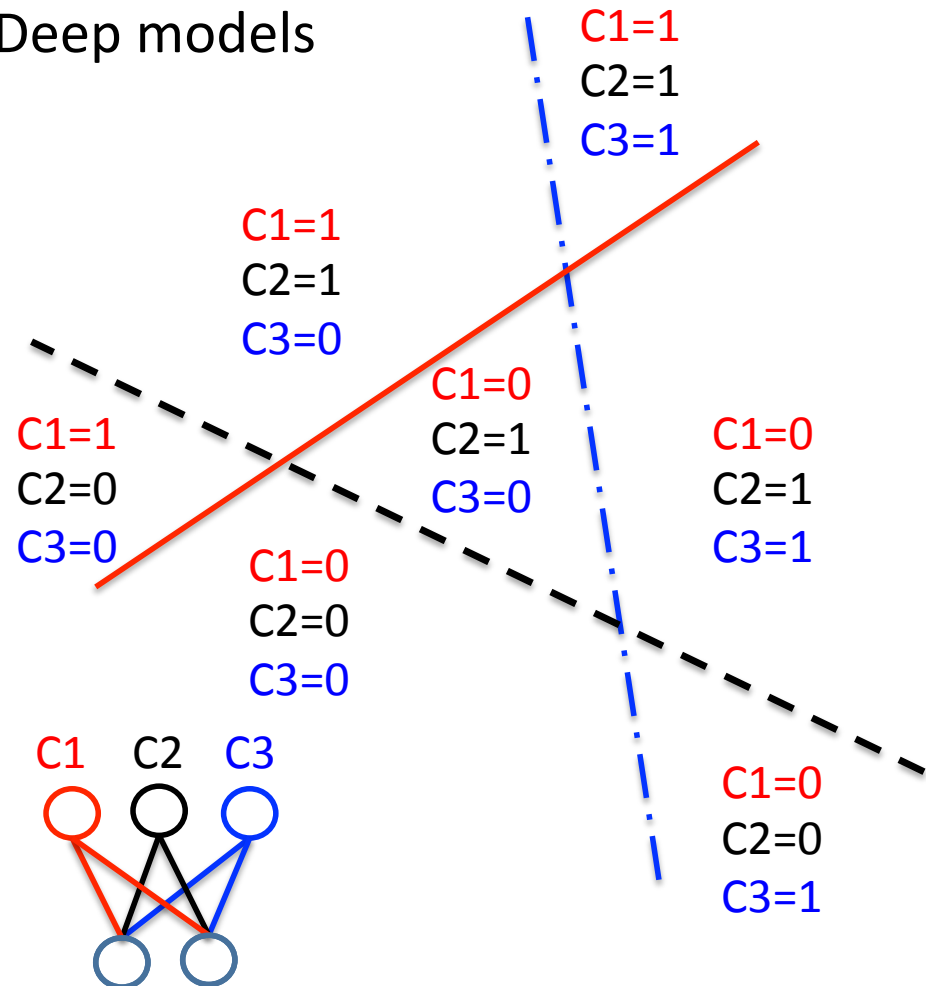
# Local vs. Distributed Representations

- Clustering, Nearest Neighbors, RBF SVM, local density estimators

- Parameters for each region.
- # of regions is linear with # of parameters.



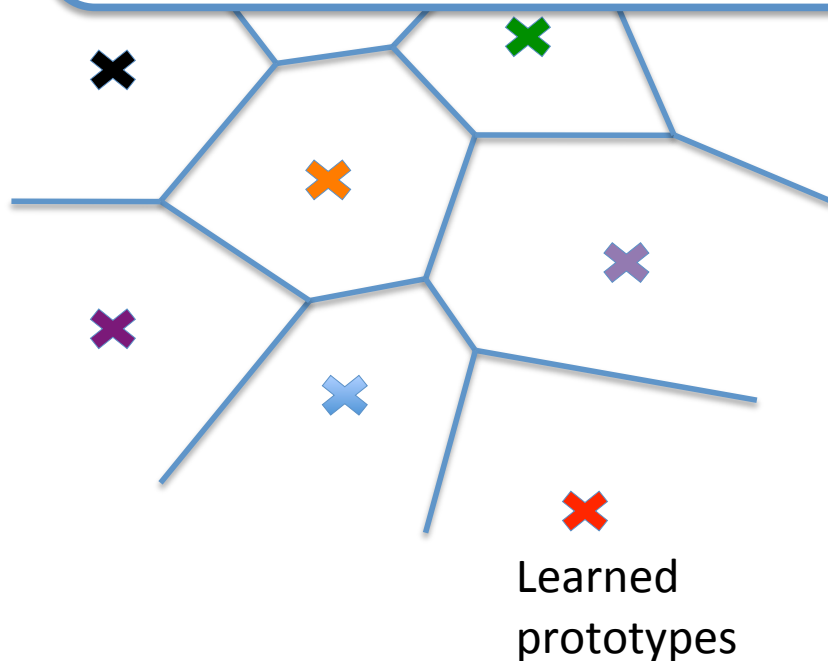
- RBMs, Factor models, PCA, Sparse Coding, Deep models



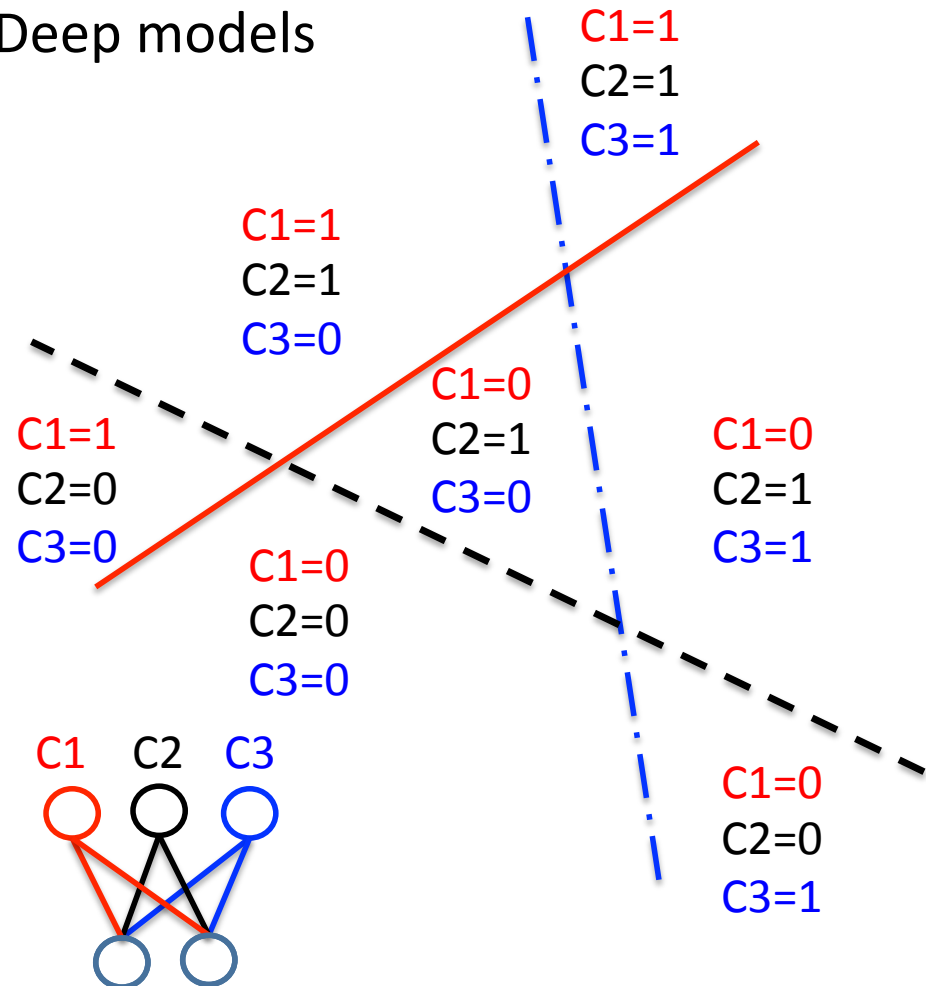
# Local vs. Distributed Representations

- Clustering, Nearest Neighbors, RBF SVM, local density estimators

- Parameters for each region.
- # of regions is linear with # of parameters.



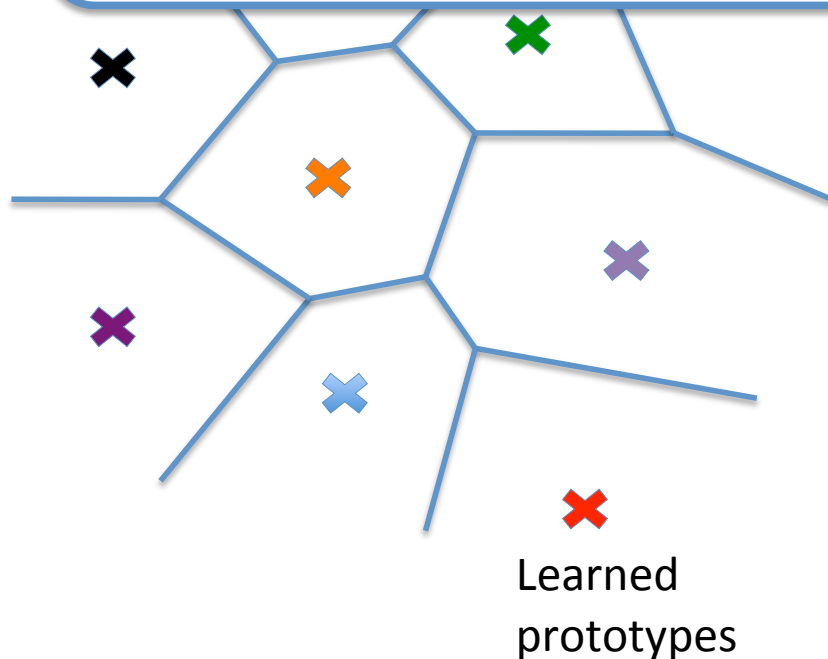
- RBMs, Factor models, PCA, Sparse Coding, Deep models



# Local vs. Distributed Representations

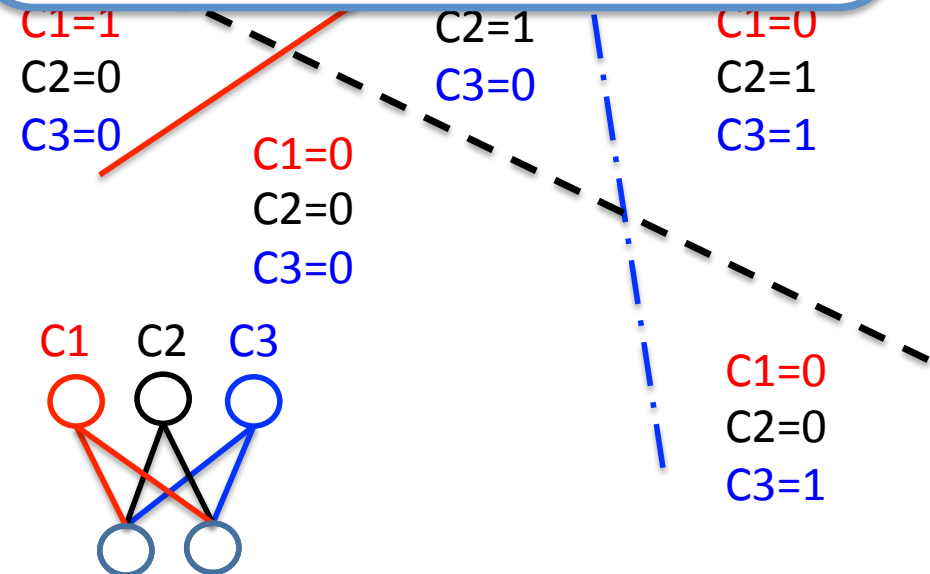
- Clustering, Nearest Neighbors, RBF SVM, local density estimators

- Parameters for each region.
- # of regions is linear with # of parameters.



- RBMs, Factor models, PCA, Sparse Coding, Deep models

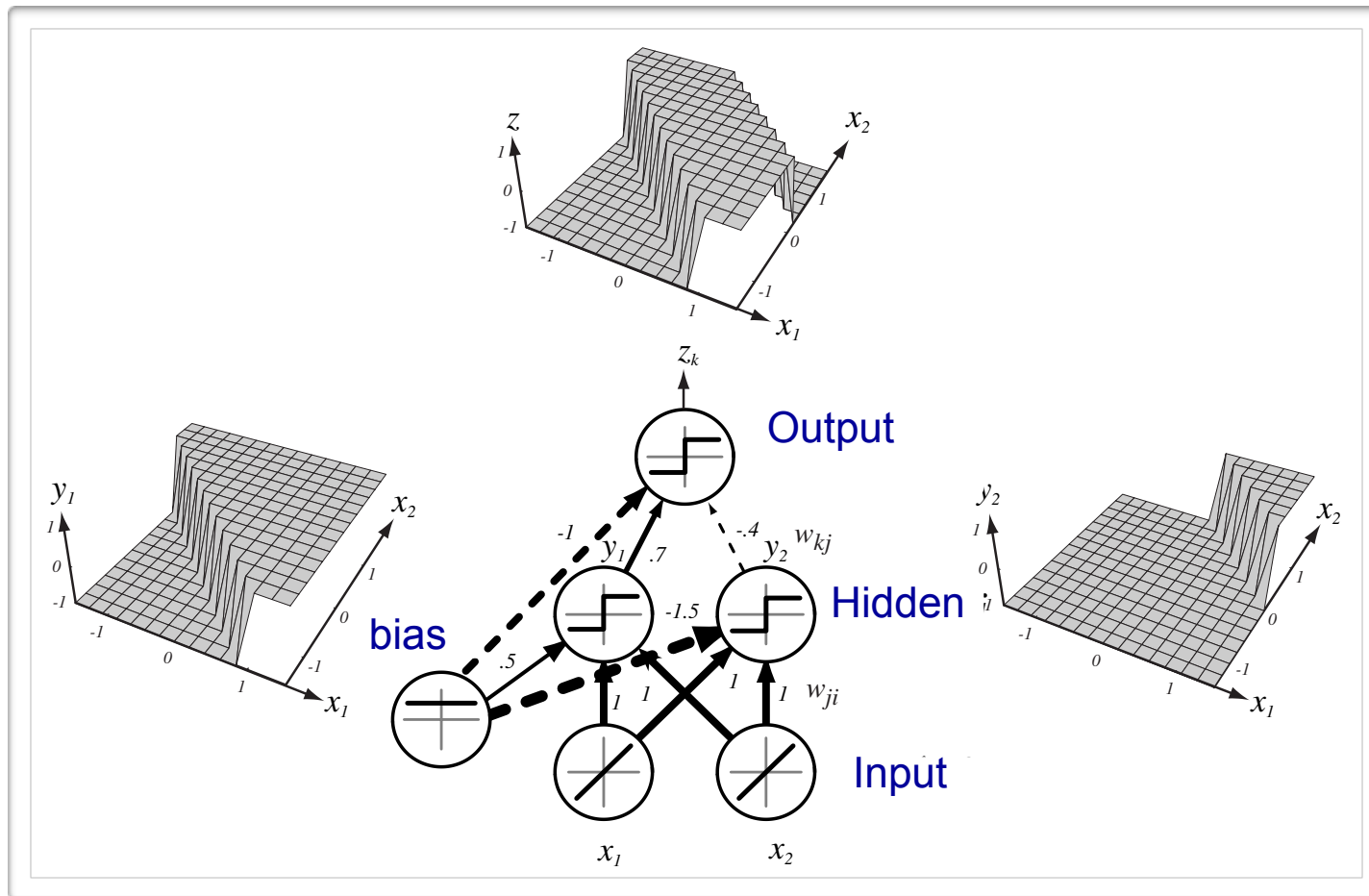
- Each parameter affects many regions, not just local.
- # of regions grows (roughly) exponentially in # of parameters.





# Capacity of Neural Nets

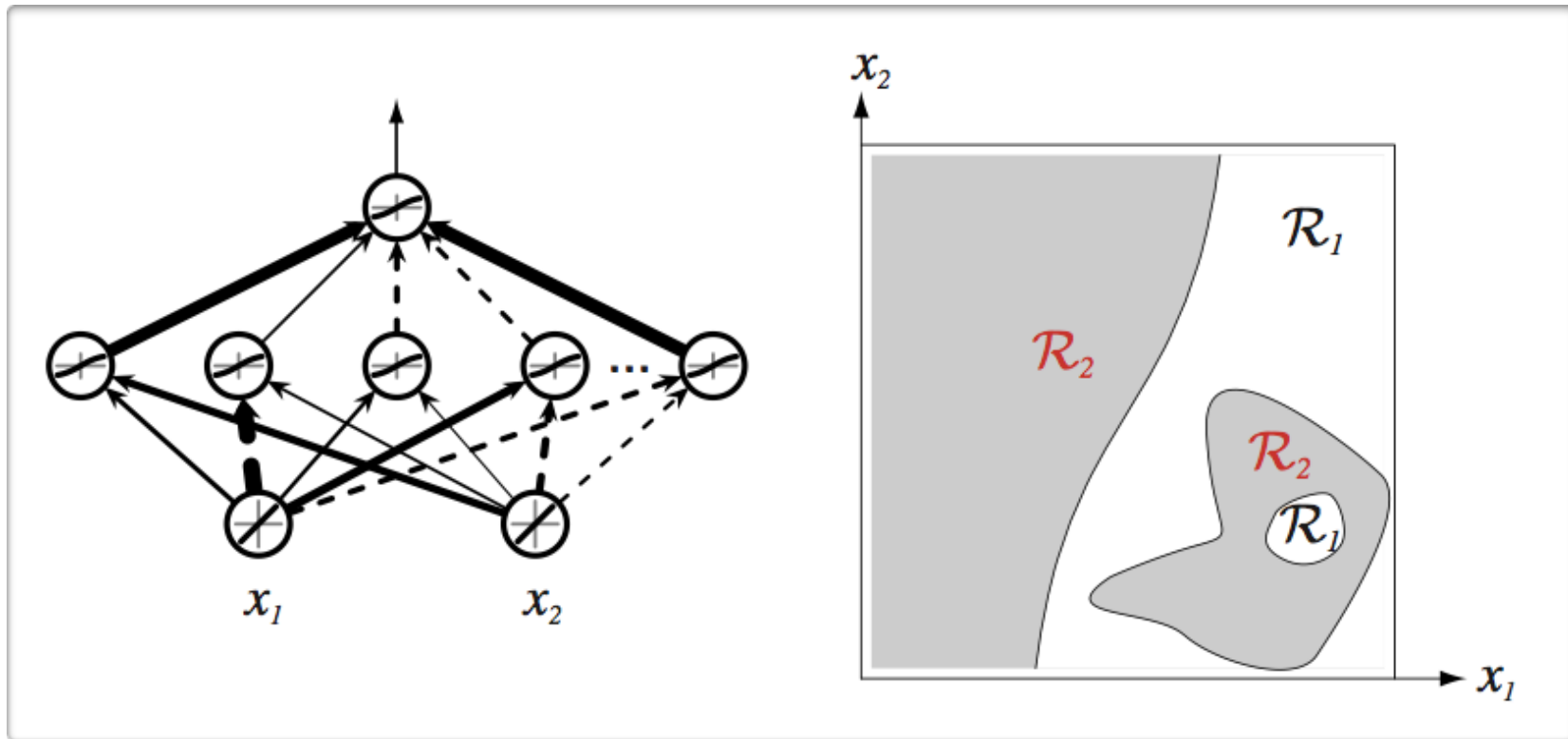
- Consider a single layer neural network



(from Pascal Vincent's slides)

# Capacity of Neural Nets

- Consider a single layer neural network



(from Pascal Vincent's slides)

# Universal Approximation

- Universal Approximation Theorem (Hornik, 1991):
  - “a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units”
- This applies for sigmoid, tanh and many other activation functions.
- However, this does not mean that there is learning algorithm that can find the necessary parameter values.

# Deep Networks vs Shallow

- 1 hidden layer neural networks are already a universal function approximator
- Implies the expressive power of deep networks are no larger than shallow networks
  - There always exists a shallow network that can represent any function representable by a deep (multi-layer) neural network
- But there can be cases where deep networks may be exponentially more compact than shallow networks in terms of number of nodes required to represent a function
- This has substantial implications for memory, computation and data efficiency
- Empirically often deep networks outperform shallower alternatives

# Deep Neural Networks

- Today: a brief introduction to deep neural networks
- Definitions
- Power of deep neural networks
  - Neural networks / distributed representations vs kernel / local representations
  - Universal function approximator
  - Deep neural networks vs shallow neural networks
- **How to train neural nets**

# Feedforward Neural Networks

- ▶ How neural networks predict  $f(\mathbf{x})$  given an input  $\mathbf{x}$ :

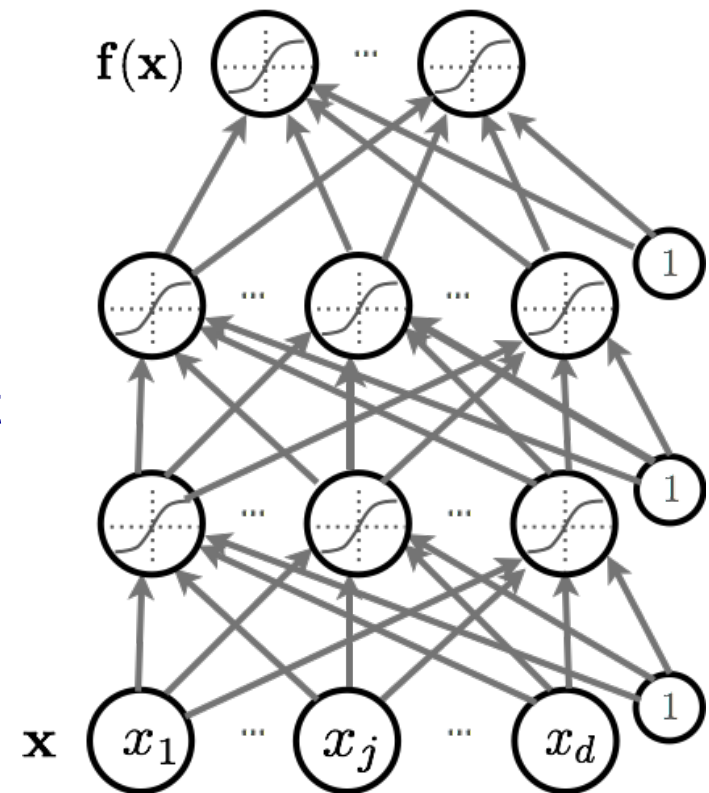
- Forward propagation
- Types of units
- Capacity of neural networks

- ▶ How to train neural nets:

- Loss function
- Backpropagation with gradient descent

- ▶ More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training



# Training

- Empirical Risk Minimization:

$$\arg \min_{\boldsymbol{\theta}} \frac{1}{T} \sum_t \underbrace{l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})}_{\text{Loss function}} + \underbrace{\lambda \Omega(\boldsymbol{\theta})}_{\text{Regularizer}}$$

- Learning is cast as optimization.
  - For classification problems, we would like to minimize classification error.

# Stochastic Gradient Descend

- Perform updates after seeing each example:

- Initialize:  $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

- For  $t=1:T$

- for each training example  $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$

$$\theta \leftarrow \theta + \alpha \Delta$$

} Training epoch  
=  
Iteration of all examples

- To train a neural net, we need:

- **Loss function:**  $l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

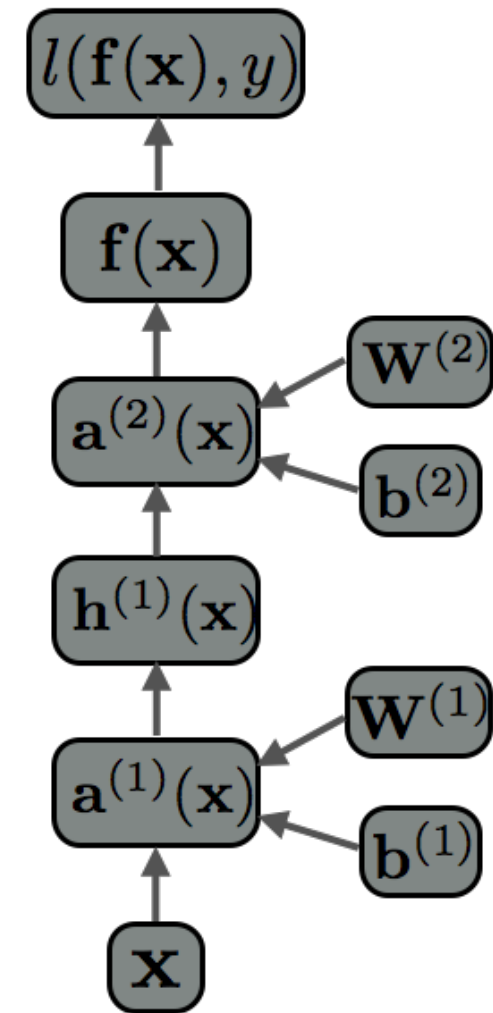
- A procedure to **compute gradients:**  $\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- **Regularizer** and its gradient:  $\Omega(\theta), \nabla_{\theta} \Omega(\theta)$



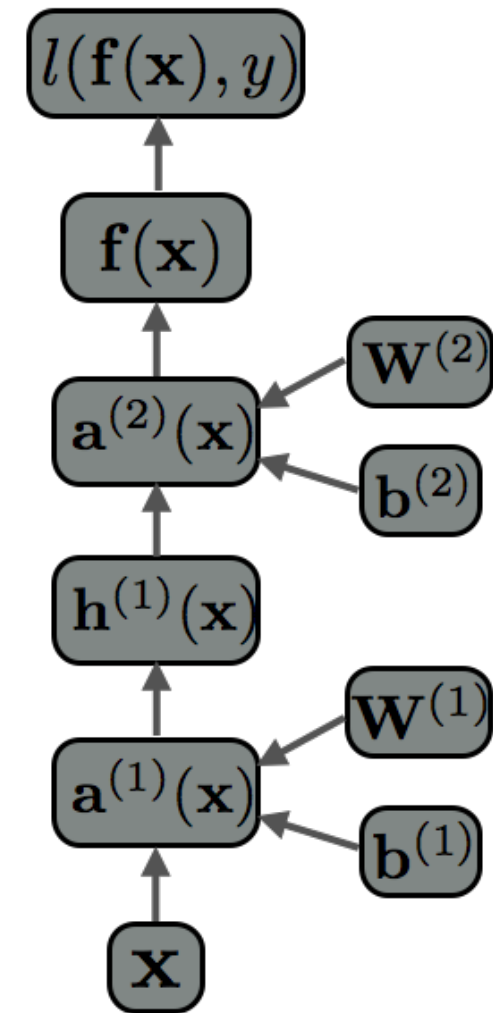
# Computational Flow Graph

- Forward propagation can be represented as an acyclic flow graph
- Forward propagation can be implemented in a modular way:
  - Each box can be an object with an **fprop method**, that computes the value of the box given its children
  - Calling the fprop method of each box in the right order yields forward propagation



# Computational Flow Graph

- Each object also has a **bprop method**
  - it computes the gradient of the loss with respect to each child box.
- By calling bprop in the **reverse order**, we obtain backpropagation

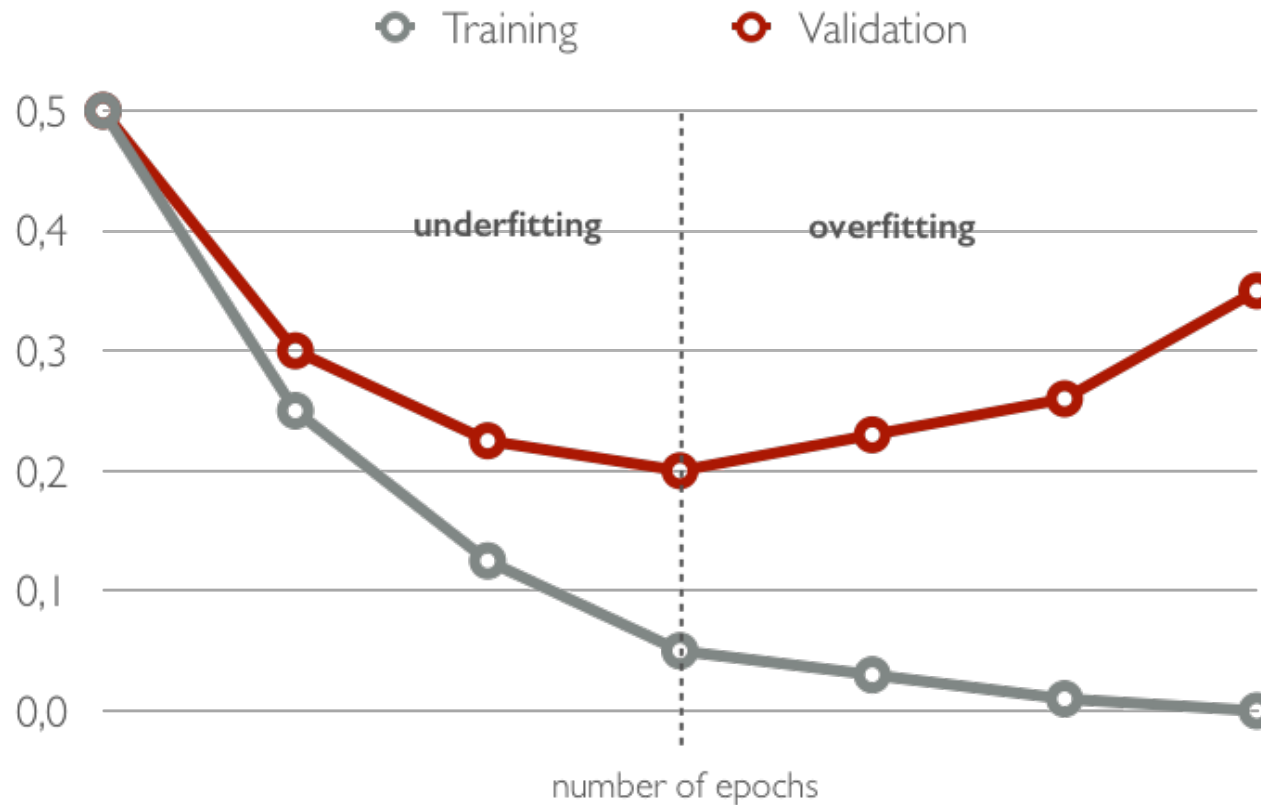


# Model Selection

- Training Protocol:
  - Train your model on the **Training Set**  $\mathcal{D}^{\text{train}}$
  - For model selection, use **Validation Set**  $\mathcal{D}^{\text{valid}}$ 
    - Hyper-parameter search: hidden layer size, learning rate, number of iterations/epochs, etc.
  - Estimate generalization performance using the **Test Set**  $\mathcal{D}^{\text{test}}$
- Generalization is the behavior of the model on **unseen examples**.

# Early Stopping

- To select the number of epochs, stop training when validation set error increases (with some look ahead).



# Mini-batch, Momentum

- Make updates based on a mini-batch of examples (instead of a single example):

- the gradient is the average regularized loss for that mini-batch
- can give a more accurate estimate of the gradient
- can leverage matrix/matrix operations, which are more efficient

- **Momentum**: Can use an exponential average of previous gradients:

$$\overline{\nabla}_{\theta}^{(t)} = \nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) + \beta \overline{\nabla}_{\theta}^{(t-1)}$$

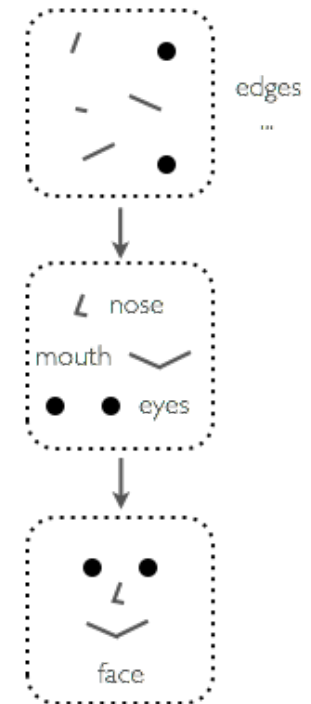
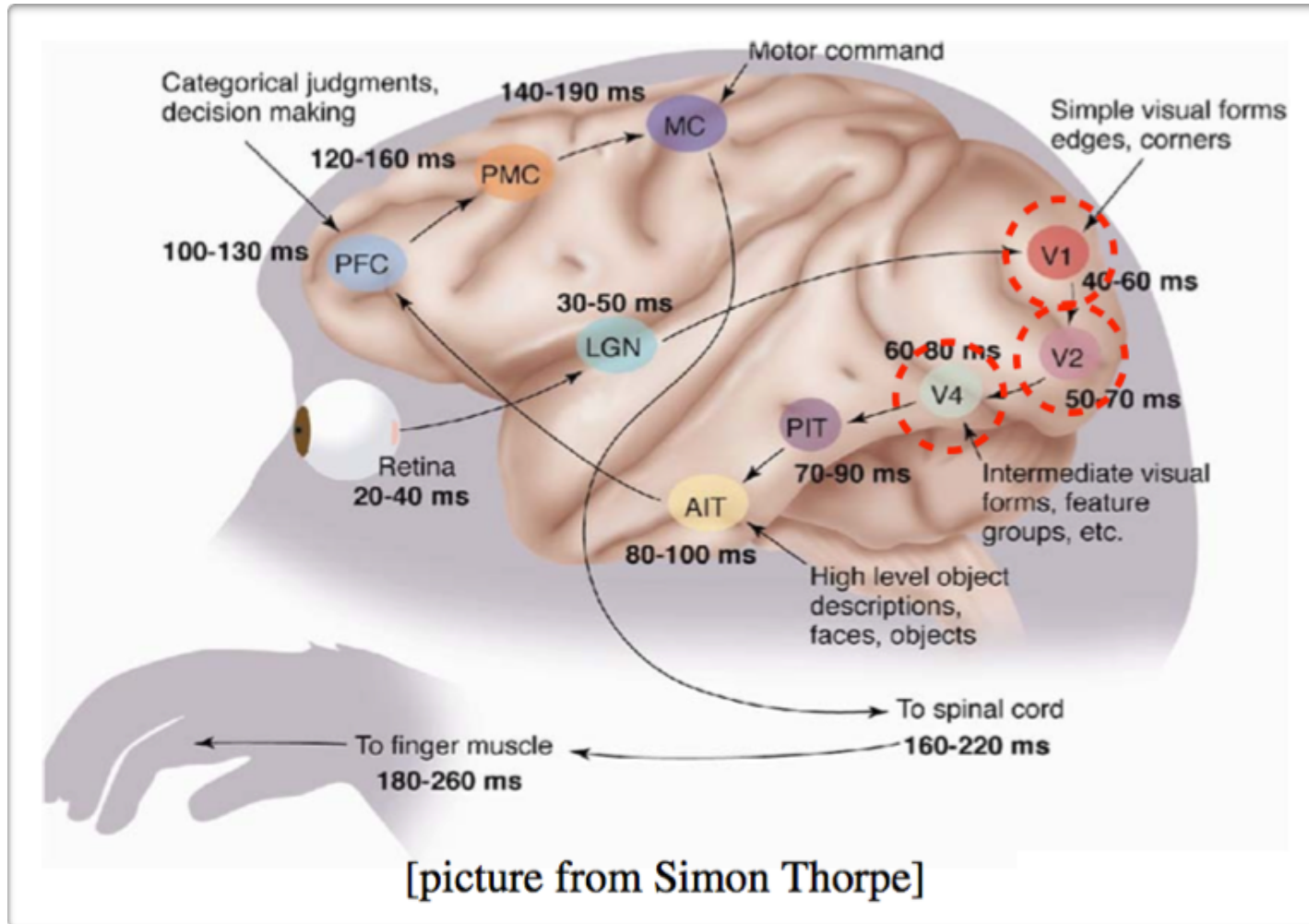
- can get ~~pass~~ plateaus more quickly, by “gaining momentum”

past

# Learning Distributed Representations

- Deep learning is research on learning models with **multilayer representations**
  - multilayer (feed-forward) neural networks
  - multilayer graphical model (deep belief network, deep Boltzmann machine)
- Each layer learns “**distributed representation**”
  - Units in a layer are not mutually exclusive
    - each unit is a separate feature of the input
    - two units can be “active” at the same time
  - Units do not correspond to a partitioning (clustering) of the inputs
    - in clustering, an input can only belong to a single cluster

# Inspiration from Visual Cortex



# Feedforward Neural Networks

- ▶ How neural networks predict  $f(\mathbf{x})$  given an input  $\mathbf{x}$ :

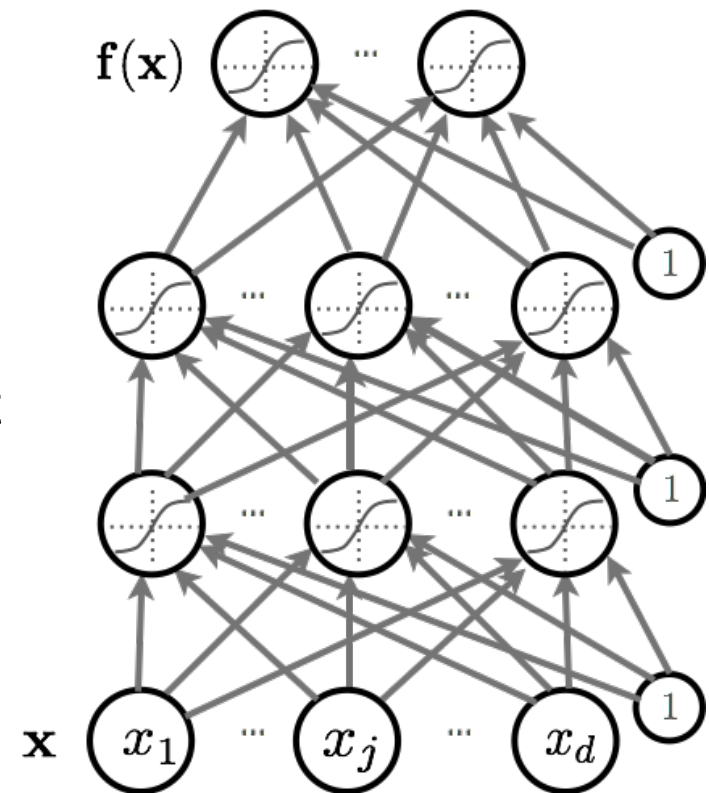
- Forward propagation
- Types of units
- Capacity of neural networks

- ▶ How to train neural nets:

- Loss function
- Backpropagation with gradient descent

- ▶ More recent techniques:

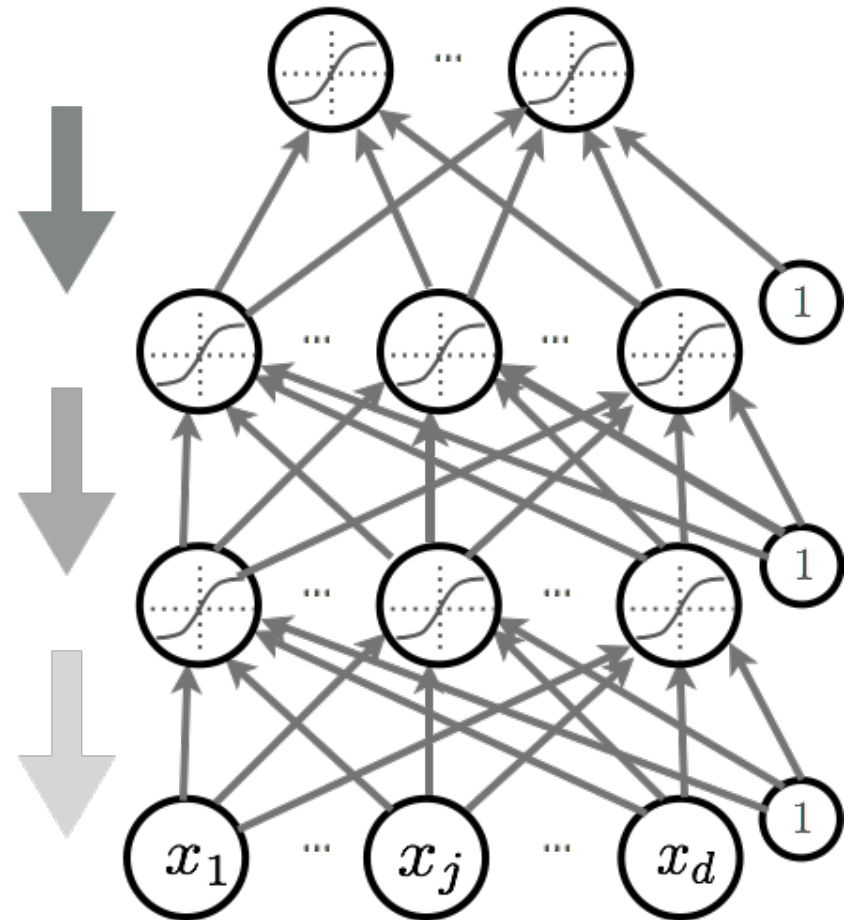
- Dropout
- Batch normalization
- Unsupervised Pre-training





# Why Training is Hard

- First hypothesis: **Hard optimization problem** (underfitting)
  - vanishing gradient problem
  - saturated units block gradient propagation
- This is a well known problem in recurrent neural networks

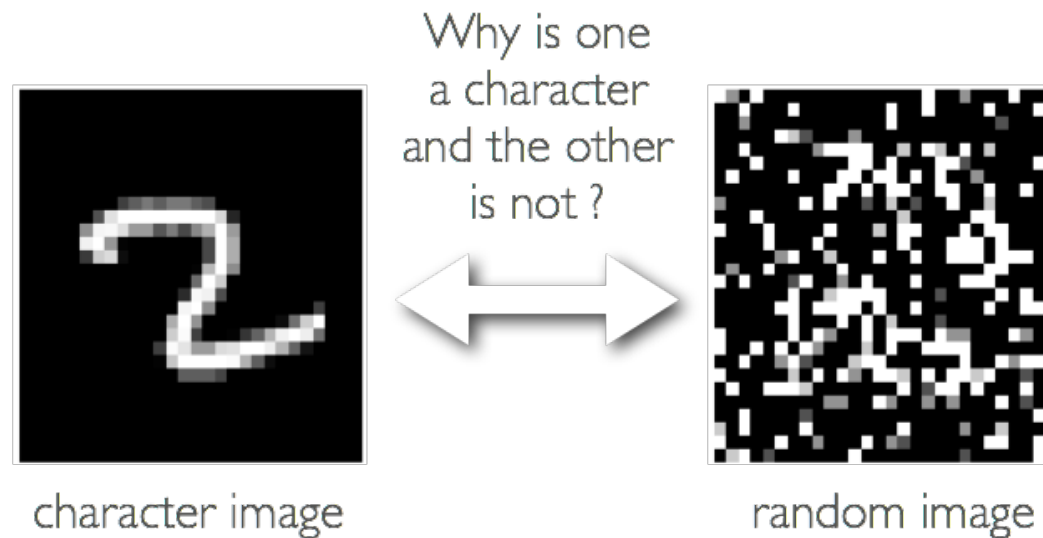


# Why Training is Hard

- First hypothesis (**underfitting**): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.
- Second hypothesis (**overfitting**): use better regularization
  - Unsupervised pre-training
  - Stochastic drop-out training
- For many large-scale practical problems, you will need to use both: better optimization and better regularization!

# Unsupervised Pre-training

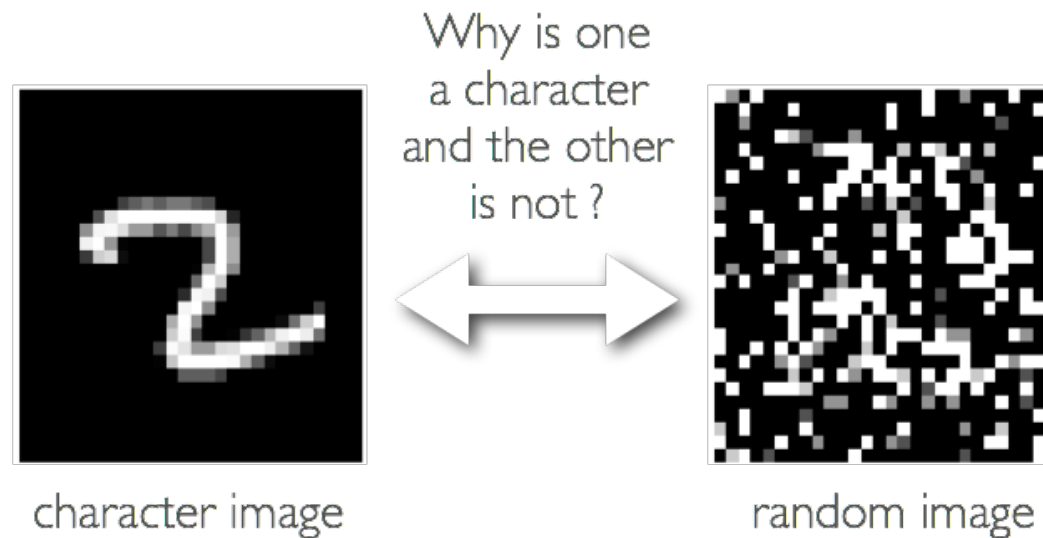
- Initialize hidden layers using **unsupervised learning**
  - Force network to represent latent structure of input distribution



- Encourage hidden layers to encode that structure

# Unsupervised Pre-training

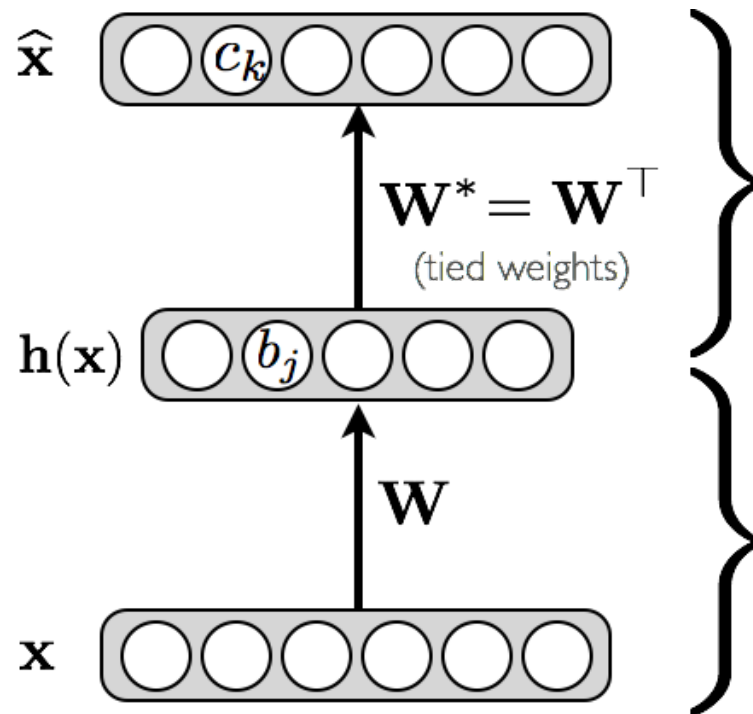
- Initialize hidden layers using **unsupervised learning**
  - This is a harder task than supervised learning (classification)



- Hence we expect less overfitting

# Autoencoders: Preview

- Feed-forward neural network trained to reproduce its input at the output layer



## Decoder

$$\begin{aligned}\hat{\mathbf{x}} &= o(\hat{\mathbf{a}}(\mathbf{x})) \\ &= \text{sigm}(\underbrace{\mathbf{c} + \mathbf{W}^* \mathbf{h}(\mathbf{x})}_{\text{For binary units}})\end{aligned}$$

## Encoder

$$\begin{aligned}\mathbf{h}(\mathbf{x}) &= g(\mathbf{a}(\mathbf{x})) \\ &= \text{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x})\end{aligned}$$

# Autoencoders: Preview

- Loss function for **binary inputs**

$$l(f(\mathbf{x})) = - \sum_k (x_k \log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k))$$

- Cross-entropy error function  $f(\mathbf{x}) \equiv \hat{\mathbf{x}}$

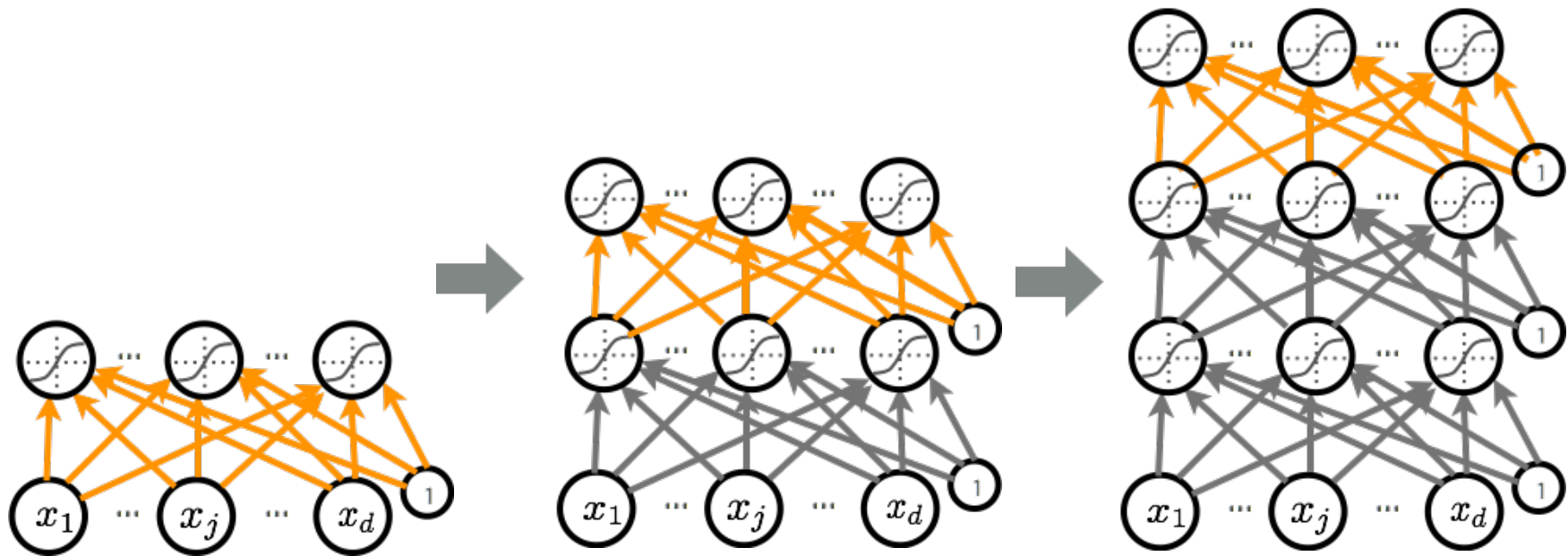
- Loss function for **real-valued inputs**

$$l(f(\mathbf{x})) = \frac{1}{2} \sum_k (\hat{x}_k - x_k)^2$$

- sum of squared differences
- we use a linear activation function at the output

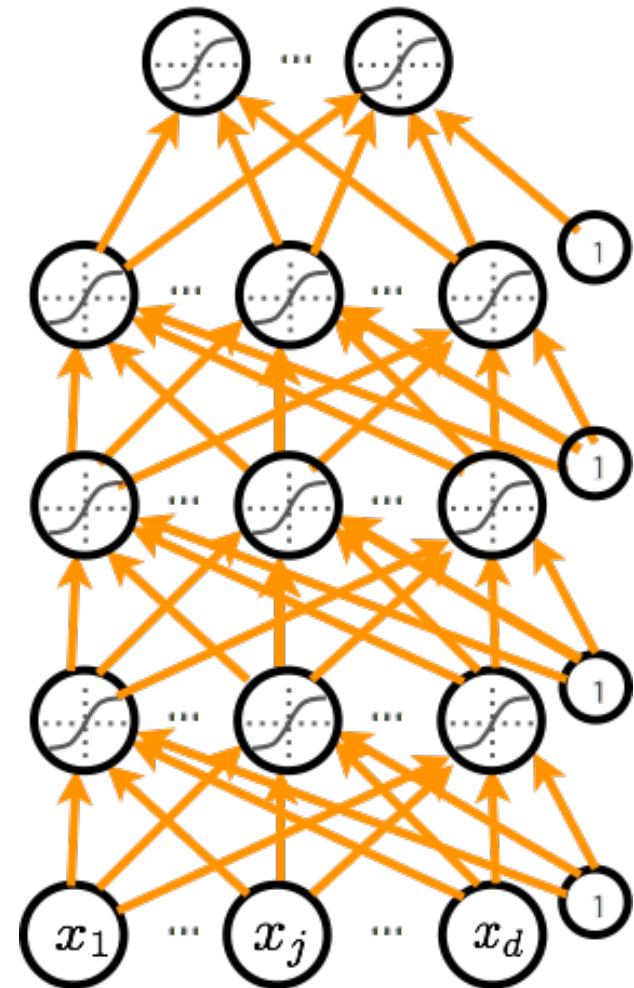
# Pre-training

- We will use a greedy, layer-wise procedure
  - Train one layer at a time with unsupervised criterion
  - Fix the parameters of previous hidden layers
  - Previous layers can be viewed as feature extraction



# Fine-tuning

- Once all layers are pre-trained
  - add output layer
  - train the whole network using supervised learning
- We call this last phase **fine-tuning**
  - all parameters are “tuned” for the supervised task at hand
  - representation is adjusted to be more discriminative





# Why Training is Hard

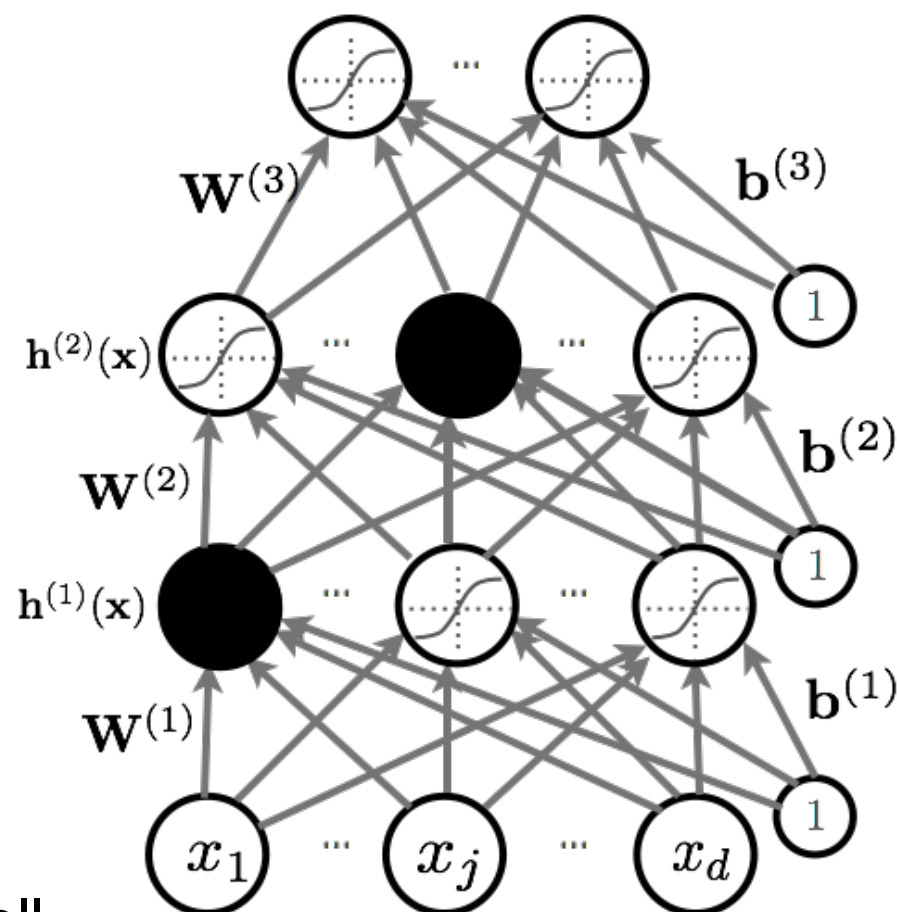
- First hypothesis (underfitting): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.
- Second hypothesis (overfitting): use better regularization
  - Unsupervised pre-training
  - Stochastic drop-out training
- For many large-scale practical problems, you will need to use both: better optimization and better regularization!

# Dropout

- **Key idea:** Cripple neural network by removing hidden units stochastically

- each hidden unit is set to 0 with probability 0.5
- hidden units cannot co-adapt to other units
- hidden units must be more generally useful

- Could use a different dropout probability, but 0.5 usually works well



# Dropout

- Use random binary masks  $m^{(k)}$

- layer pre-activation for  $k > 0$

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

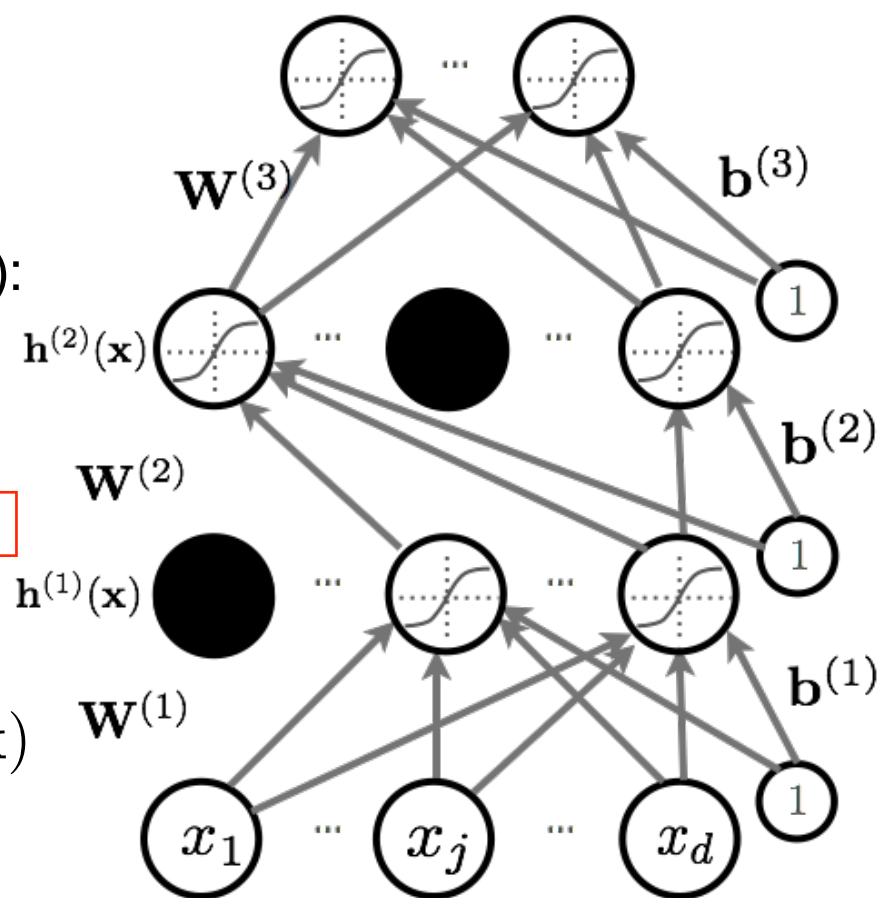
- hidden layer activation ( $k=1$  to  $L$ ):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \odot \mathbf{m}^{(k)}$$

this symbol may confuse some

- Output activation ( $k=L+1$ )

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



# Dropout at Test Time

- At test time, we replace the masks by their **expectation**
  - This is simply the constant vector 0.5 if dropout probability is 0.5
  - For single hidden layer: equivalent to taking the geometric average of all neural networks, with all possible binary masks
- Can be combined with unsupervised pre-training
- Beats regular backpropagation on many datasets
- **Ensemble**: Can be viewed as a geometric average of exponential number of networks.

# Why Training is Hard

- First hypothesis (**underfitting**): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.
- Second hypothesis (**overfitting**): use better regularization
  - Unsupervised pre-training
  - Stochastic drop-out training
- For many large-scale practical problems, you will need to use both: better optimization and better regularization!

# Batch Normalization

- Normalizing the inputs will speed up training (Lecun et al. 1998)
  - could normalization be useful at the level of the hidden layers?
- **Batch normalization** is an attempt to do that (Ioffe and Szegedy, 2014)
  - each unit's pre-activation is normalized (mean subtraction, stddev division)
  - during training, mean and stddev is computed for each minibatch
  - backpropagation takes into account the normalization
  - at test time, the global mean / stddev is used

# Batch Normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$


**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$



Learned linear transformation to adapt to non-linear activation function ( $\gamma$  and  $\beta$  are trained)

# Batch Normalization

- Why normalize the pre-activation?
  - can help keep the pre-activation in a non-saturating regime (though the linear transform  $y_i \leftarrow \gamma \hat{x}_i + \beta$  could cancel this effect)
- Use the **global mean and stddev** at test time.
  - removes the stochasticity of the mean and stddev
  - requires a final phase where, from the first to the last hidden layer
    - propagate all training data to that layer
    - compute and store the global mean and stddev of each unit
  - for early stopping, could use a running average