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.

Important Information About Homework 2

- Homework 2 will now be due on Saturday February 10 (instead of February 7)
- We are making this change to try to give some background on deep learning, give people enough time to do homework 2, and still give people time to study for the midterm on February 14
- We will release the homework this week
- You will be able to start on some aspects of the homework this week, but we will be covering DQN which is the largest part, on Monday
- We will also be providing optional tutorial sessions on tensorflow

Tuesday 4-6pm – see piazza for details

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

Recall: Reinforcement Learning Involves

- Optimization
- Delayed consequences
- Exploration
- Generalization

Today: Focus on Generalization

- Optimization
- Delayed consequences
- Exploration
- **Generalization**

Table of Contents

1 Introduction

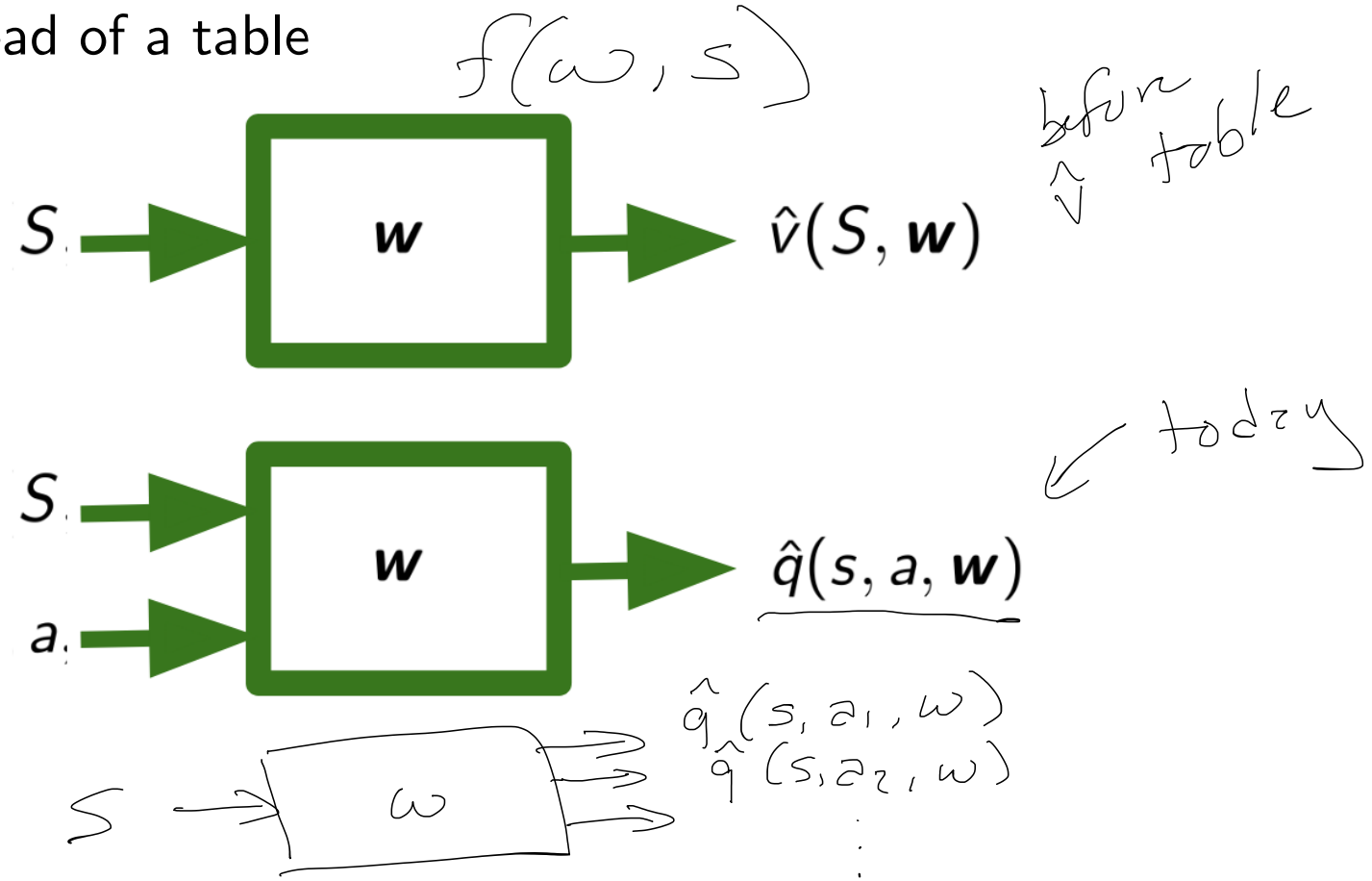
2 VFA for Prediction

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

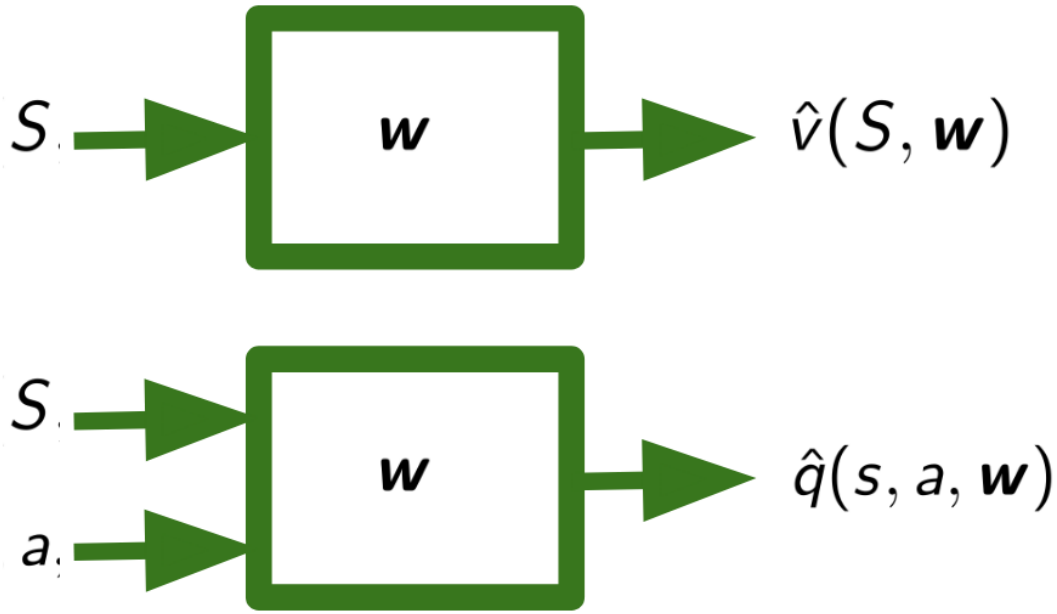
Benefits of Generalization

(EE)
control prob often S is ∞

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

$$\nabla_w J(w) = \left[\frac{\partial J(w)}{\partial w_1} \quad \frac{\partial J(w)}{\partial w_2} \quad \cdots \quad \frac{\partial J(w)}{\partial w_n} \right]$$

vector

local optima of J

$$w \leftarrow w - \alpha \nabla_w J(w)$$

α: learning rate

shorthand

$$\Delta w = \alpha \nabla_w J(w)$$

$$= -1/2 \, \alpha \, J(w) \quad \text{to min } J$$

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

$\hat{V}$ = func approx estimate

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

Stochastic Gradient Descent

- Goal: Find the parameter vector $\boldsymbol{w}$ 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 $\boldsymbol{w}$.
- Generally use mean squared error and define the loss as

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

- Can use gradient descent to find a local minimum

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

- Stochastic gradient descent (SGD) samples the gradient:

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

$$\Delta w = \alpha \left(v_\pi(s) - \hat{v}(s,w)\right) \nabla_w \hat{v}(s,w) \leftarrow \text{partic. } s$$

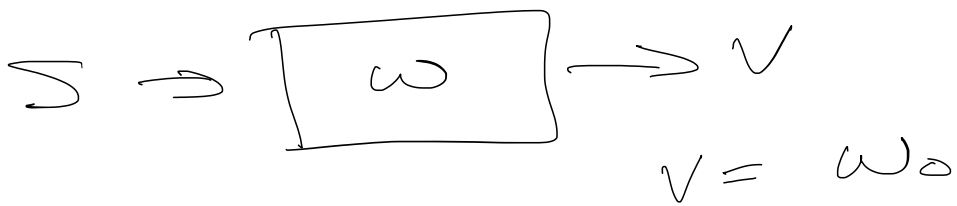
- Expected SGD is the same as the full gradient update

VFA Prediction Without An Oracle

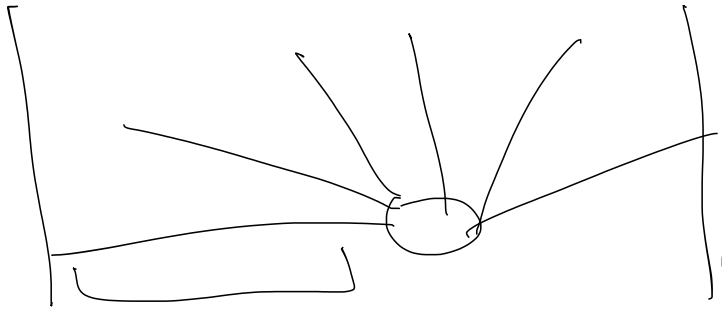
- Don't actually have access to an oracle to tell true $v_\pi(S)$ for any state s
- Now consider how to do value function approximation for prediction / evaluation / policy evaluation without a model
- Note: policy evaluation without a model is sometimes also called **passive reinforcement learning** 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 π (or had access to prior data)
  - Goal is to estimate V^π and/or Q^π
- Maintained a look up table to store estimates V^π and/or Q^π
- Updated these estimates after each episode (Monte Carlo methods) or after each step (TD methods)
- **Now: in value function approximation, change the estimate update step to include fitting the function approximator**



Feature Vectors



$x(s) =$

- Use a feature vector to represent a state

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

robotic navig

$x(s) =$ [long, lat, orient, height]

$x(s) =$ [intensity, dist lowground, ...]

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

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

- Objective function is

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

- Recall weight update is

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

- Update is:

$$\Delta w = \alpha \left(V_\pi(s) - \hat{V}(s, w) \right) \cdot \nabla \hat{V}(s, w) \cdot x(s)$$

- 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: $< S_1, G_1 >, < S_2, G_2 >, \ldots, < S_T, G_T >$
  - Susbstituting $G_t(S_t)$ for the true $v_\pi(S_t)$ when fitting the function approximator
- Concretely when using linear VFA for policy evaluation

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

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

- 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 $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \ldots, s_{k,L_k})$ given π
4: **for** $t = 1, \ldots, L_k$ **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

$$\Delta w = \alpha \left(Returns(s_t) - \hat{V}(s_t, w) \right) \cdot X(s_t)$$

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^π
- Updates $V^\pi(s)$ after each transition (s, a, r, s'):

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

- 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^π
- Updates estimate $V^\pi(s)$ after each transition (s, a, r, s'):

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

- Target is $r + \gamma V^\pi(s')$, a biased estimate of 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 ~~of~~ the true value $v^\pi(s)$
- 3 forms of approximation:
  1) sampling an expectation
  2) bootstrapping
  3) func 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 of the true value $v^{\pi}(s)$
- Supervised learning on a different set of data pairs: $< S_1, r_1 + \gamma \hat{v}^{\pi}(S_2, \boldsymbol{w}) >, < S_2, r_2 + \gamma \hat{v}(S_3, \boldsymbol{w}) >, \ldots$

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 of the true value $v^{\pi}(s)$
- Supervised learning on a different set of data pairs: $< S_1, r_1 + \gamma \hat{v}^{\pi}(S_2, \boldsymbol{w}) >, < S_2, r_2 + \gamma \hat{v}(S_3, \boldsymbol{w}) >, \ldots$
- In linear TD(0)

$$\begin{aligned} \Delta \boldsymbol{w} &= \alpha(r + \gamma \hat{v}^{\pi}(s', \boldsymbol{w}) - \hat{v}^{\pi}(s, \boldsymbol{w})) \bigtriangledown_{\boldsymbol{w}} \hat{v}^{\pi}(s, \boldsymbol{w}) && (9) \\ &= \alpha(r + \gamma \hat{v}^{\pi}(s', \boldsymbol{w}) - \hat{v}^{\pi}(s, \boldsymbol{w})) \boldsymbol{x}(s) && (10) \end{aligned}$$

Convergence Guarantees for Linear Value Function Approximation for Policy Evaluation[1]

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

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

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

[1] 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[2]

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

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

$(s, V^{\pi}(s))$ (s, G_t)

lik MC

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

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

[2] 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[3]

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

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

$(s, V^{\pi}(s))$ like $(s, r + \gamma \hat{V}(s'))$ TD

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

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

[3] ibed.

Summary: Convergence Guarantees for Linear Value Function Approximation for Policy Evaluation[4]

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

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

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

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

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

[4] 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

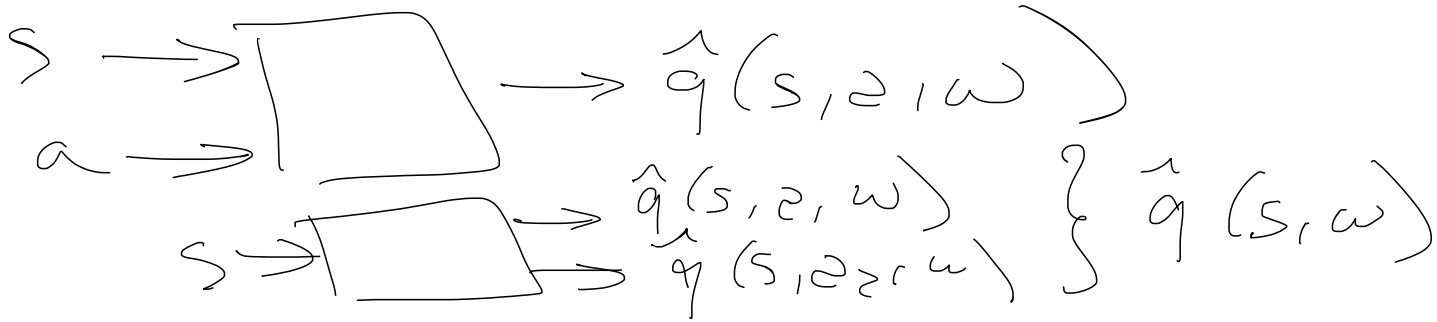
1 Introduction

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 $\hat{q}^{\pi}(s, a, \boldsymbol{w}) \approx q^{\pi}$
- Interleave
  - Approximate policy evaluation using value function approximation
  - Perform ϵ-greedy policy improvement

Action-Value Function Approximation with an Oracle

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

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

- Use stochastic gradient descent to find a local minimum

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

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

- Stochastic gradient descent (SGD) samples the gradient

(s,a) pair

Linear State Action Value Function Approximation with an Oracle

- Use features to represent both the state and action

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

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

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

- Stochastic gradient descent update: $\rightarrow [q^{\pi}(s,a) - \hat{q}^{\pi}(s,a,w)] \, x(s,a)$

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

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

$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{q}(s_t, a_t, \boldsymbol{w})) \bigtriangledown_{\boldsymbol{w}} \hat{q}(s_t, a_t, \boldsymbol{w}) \tag{24}$$

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

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \hat{q}(s', a', \boldsymbol{w}) - \hat{q}(s, a, \boldsymbol{w})) \bigtriangledown_{\boldsymbol{w}} \hat{q}(s, a, \boldsymbol{w}) \tag{25}$$

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

$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{q}(s_t, a_t, \boldsymbol{w})) \bigtriangledown_{\boldsymbol{w}} \hat{q}(s_t, a_t, \boldsymbol{w}) \tag{26}$$

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

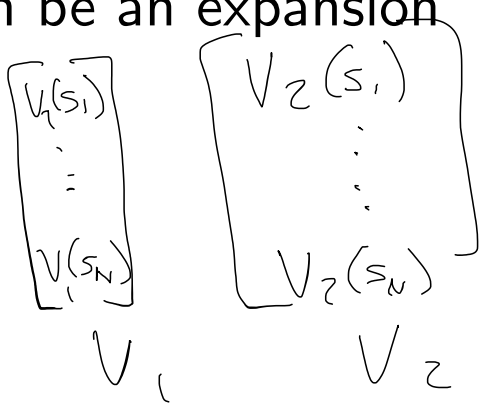
$$\Delta \boldsymbol{w} = \alpha(r + \gamma \hat{q}(s', a', \boldsymbol{w}) - \hat{q}(s, a, \boldsymbol{w})) \bigtriangledown_{\boldsymbol{w}} \hat{q}(s, a, \boldsymbol{w}) \tag{27}$$

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

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w}) - \hat{q}(s, a, \boldsymbol{w})) \bigtriangledown_{\boldsymbol{w}} \hat{q}(s, a, \boldsymbol{w}) \tag{28}$$

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



$\|BV_1 - BV_2\|_\infty \leq \|V_1 - V_2\|_\infty$

$\|\Pi V_1 - \Pi V_2\|$?

Π Func approx

Convergence of Control Methods with VFA

(✓) chatters

deep learning

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

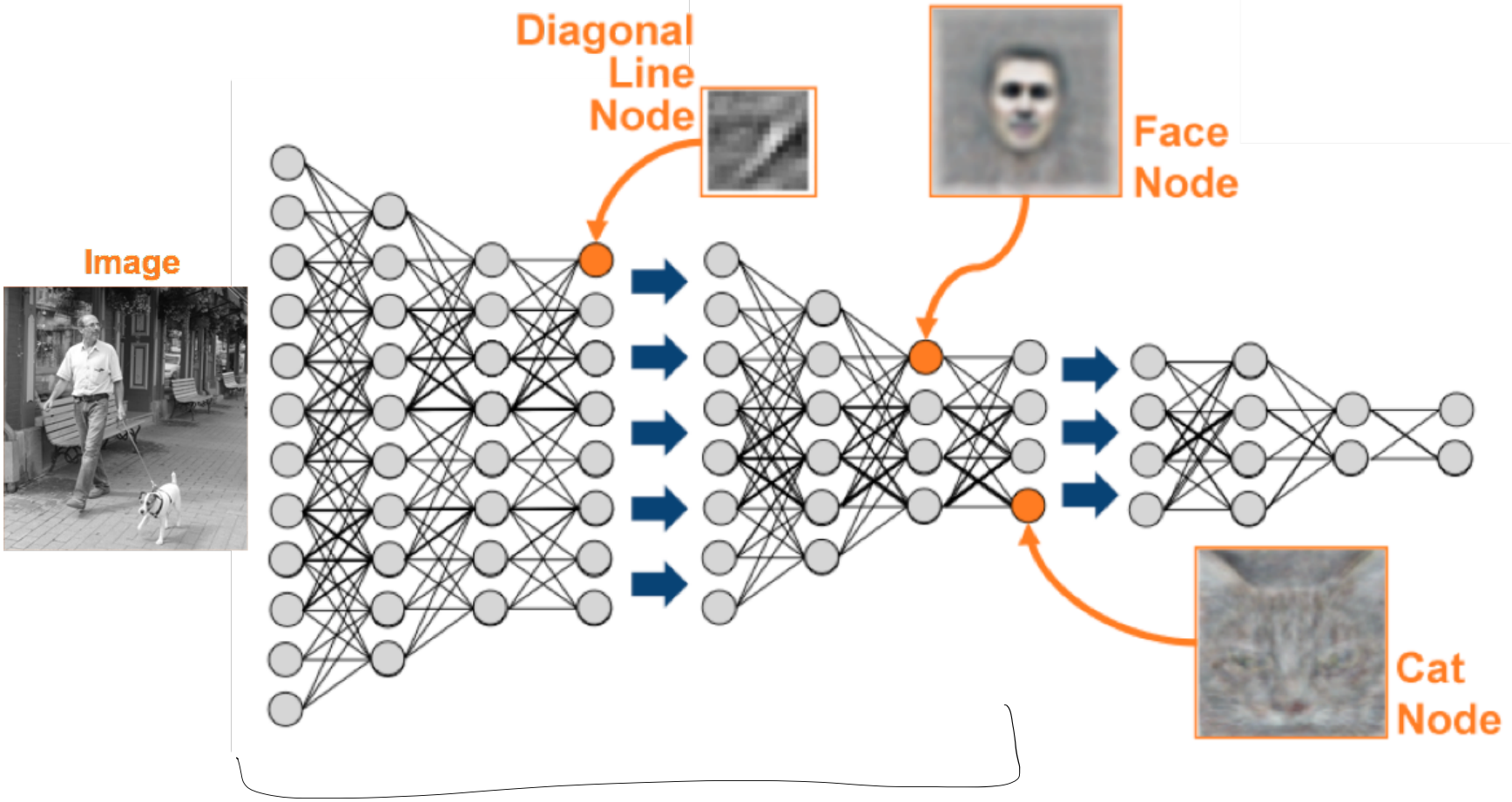
Deep Neural Networks

- 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

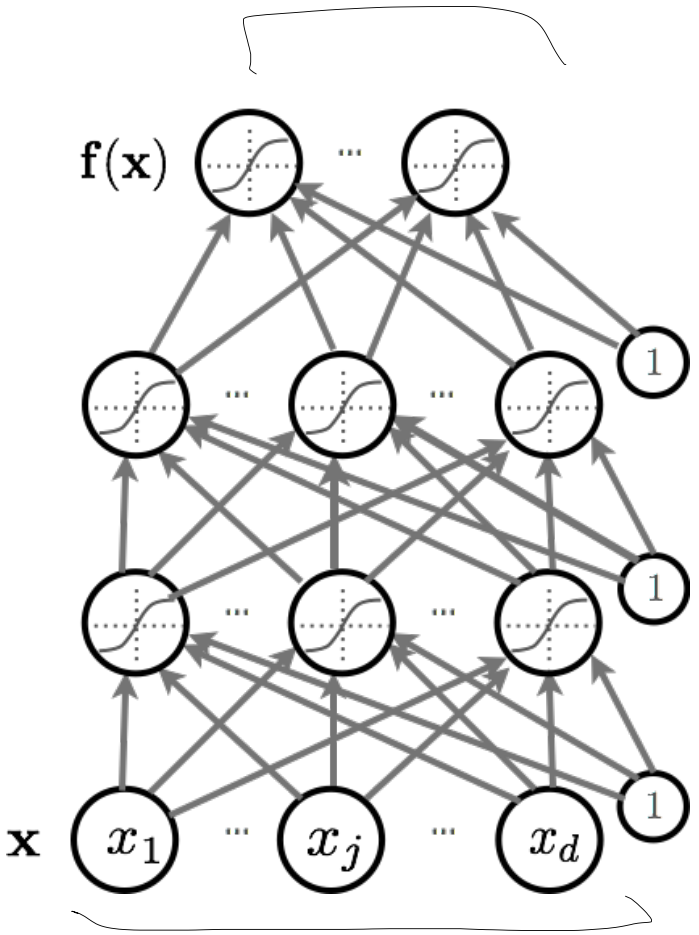


Deep Neural Networks

- 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

- 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:

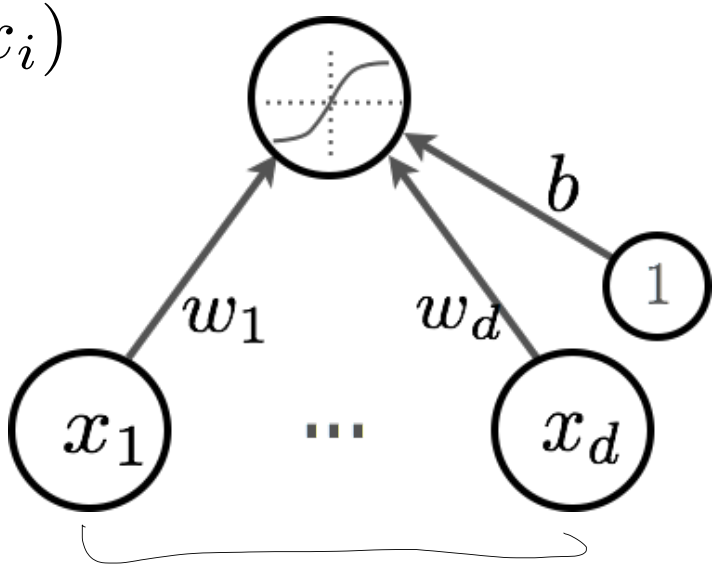
$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

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:

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

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

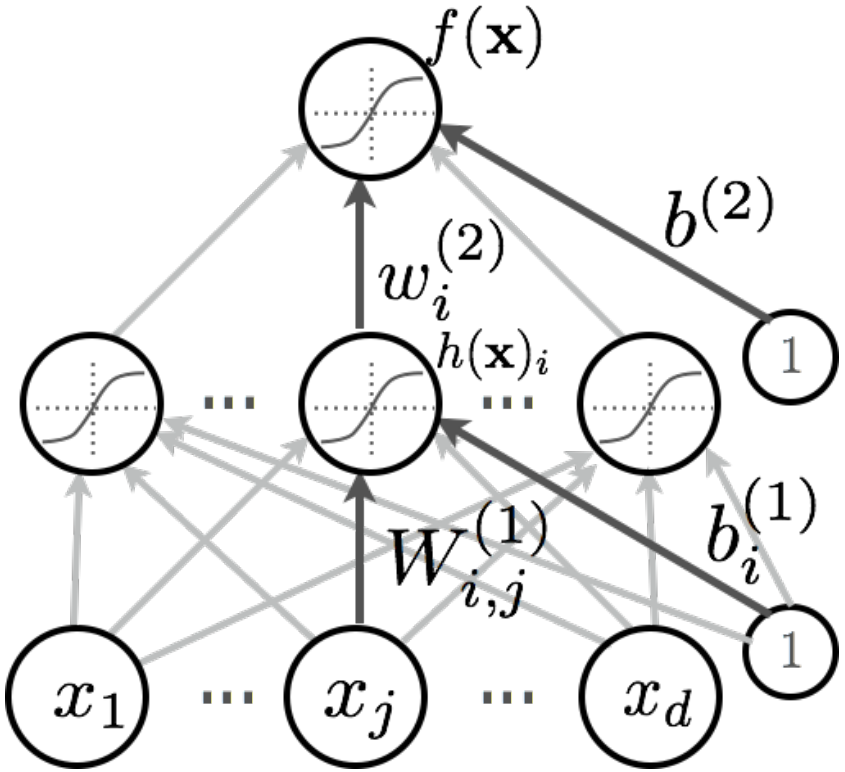
- Hidden layer activation:

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

- Output layer activation:

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

Output activation function

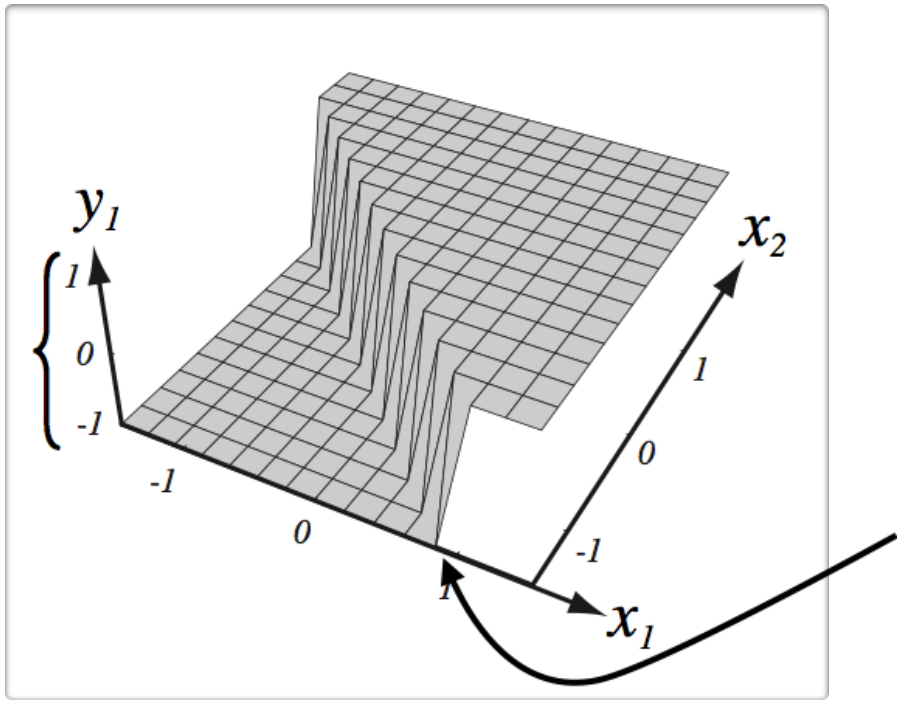


Artificial Neuron

- Output activation of the neuron:

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

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:

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

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

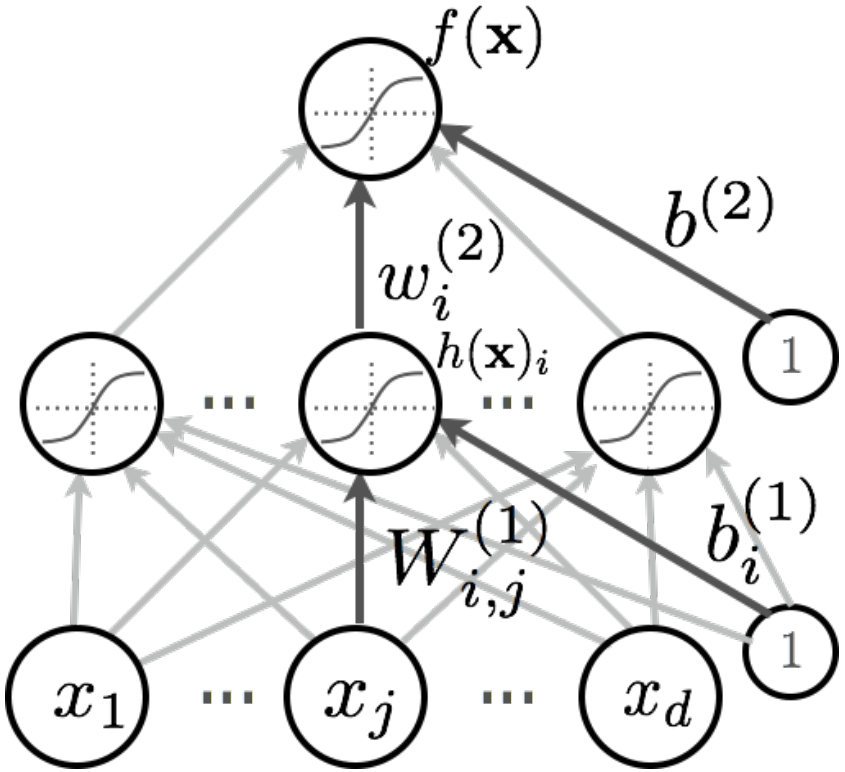
- Hidden layer activation:

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

- Output layer activation:

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

Output activation function

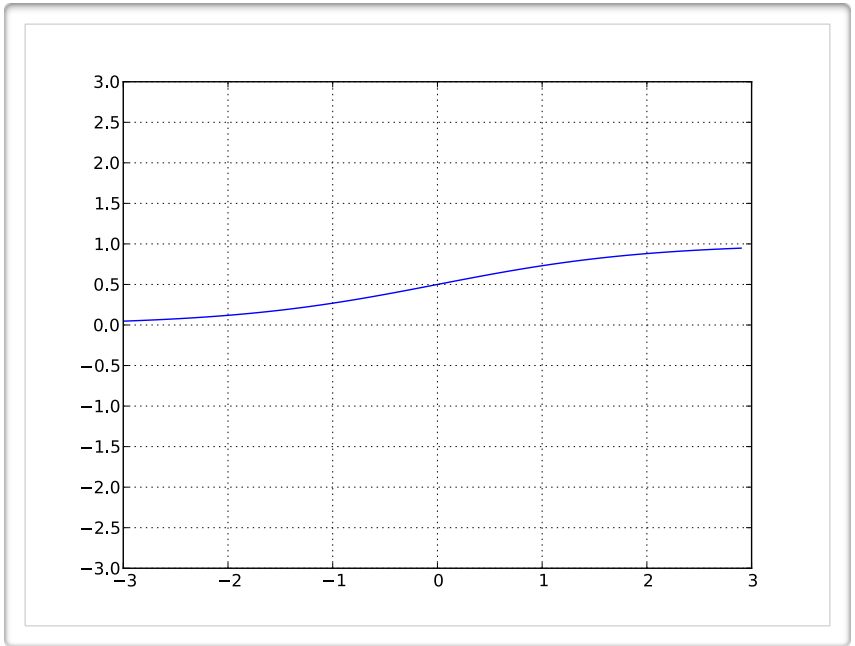


Activation Function

- Sigmoid activation function:

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

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



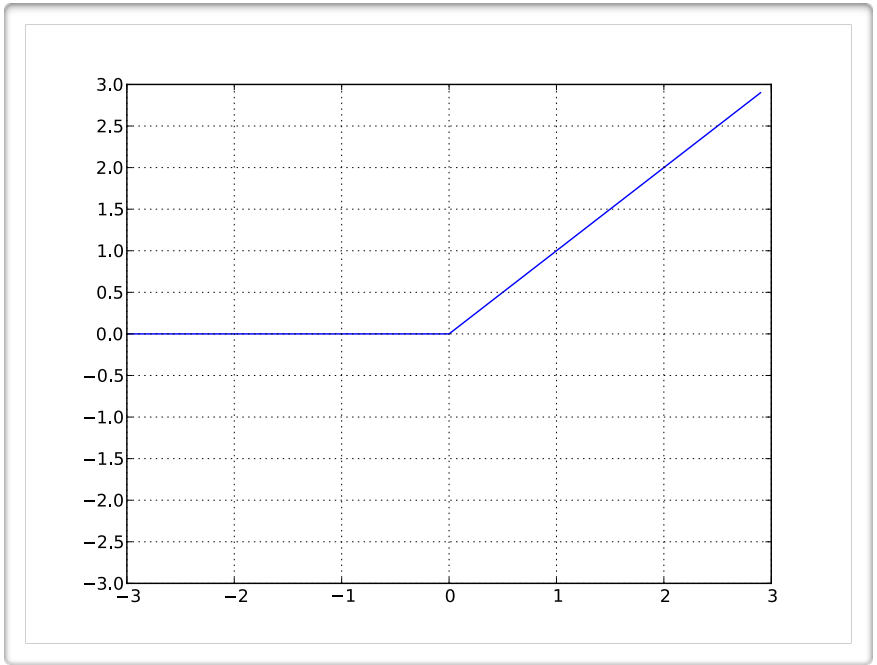
Activation Function

very popular

- 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

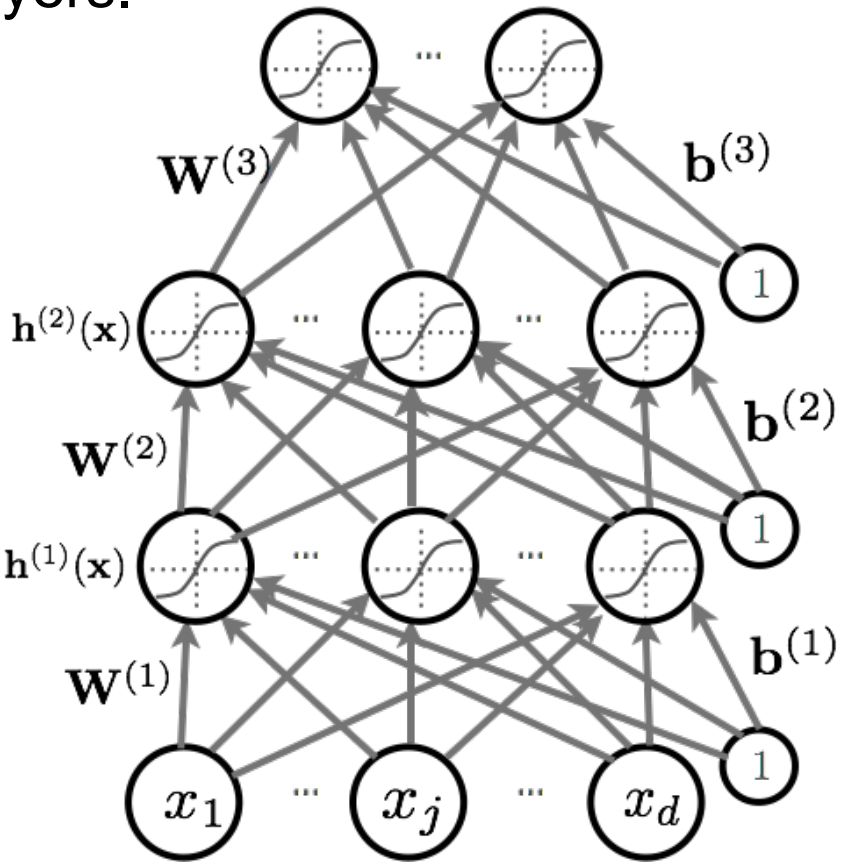
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - 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):

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



$$(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$$

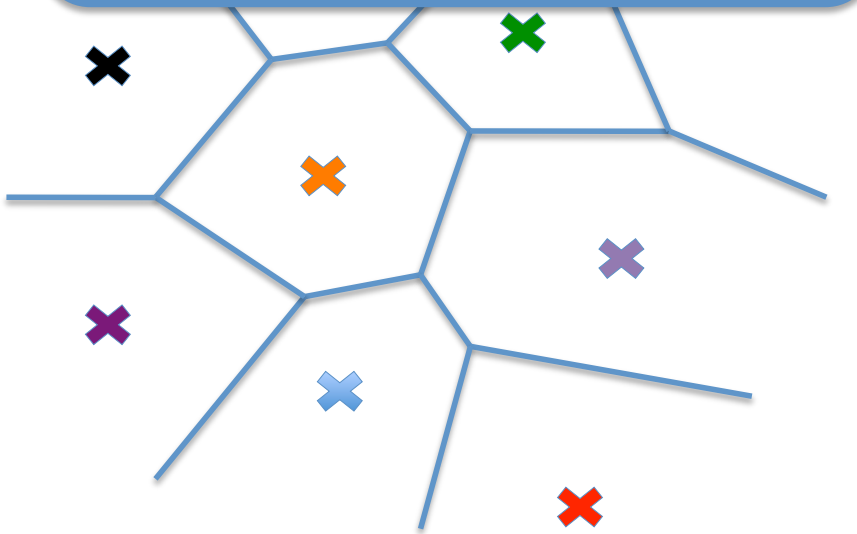
Deep Neural Networks

- 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

Local vs. Distributed Representations

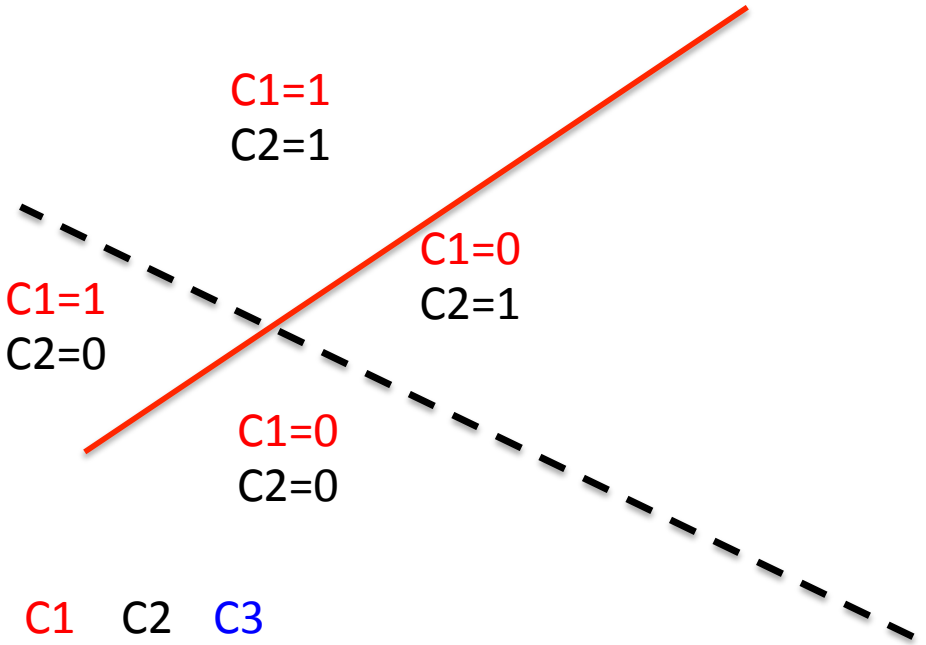
- Clustering, Nearest Neighbors, RBF SVM, local density estimators

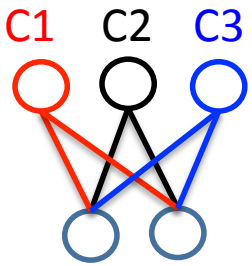
- Parameters for each region.
- # of regions is linear with # of parameters.



Learned prototypes

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



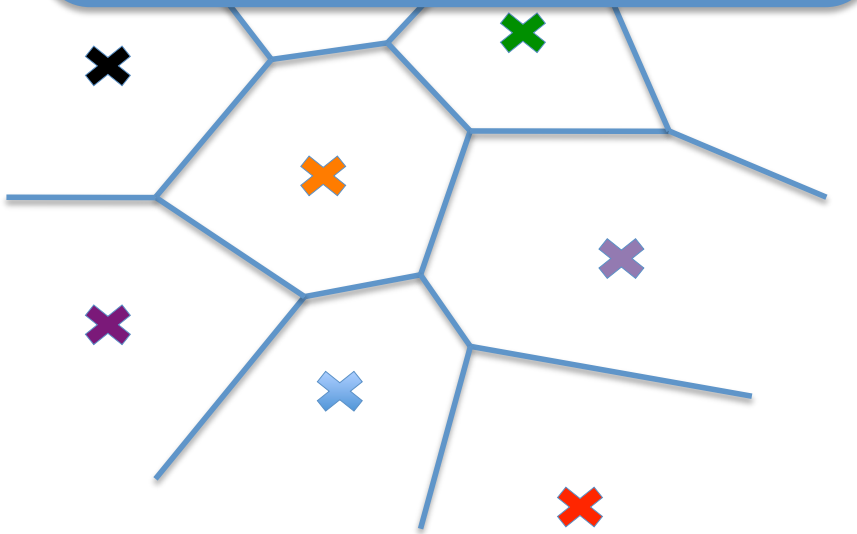


(Bengio, 2009, Foundations and Trends in Machine Learning)

Local vs. Distributed Representations

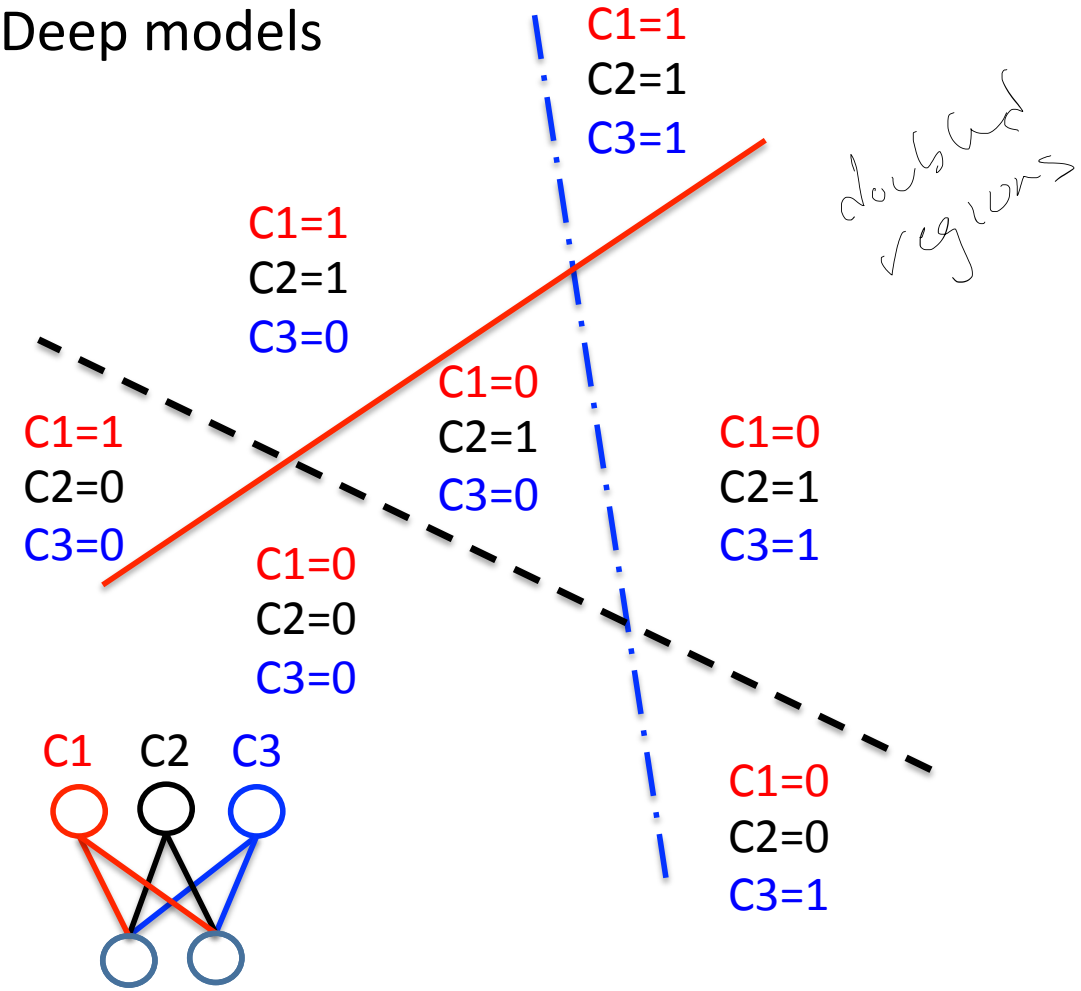
• Clustering, Nearest Neighbors, RBF SVM, local density estimators

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



Learned prototypes

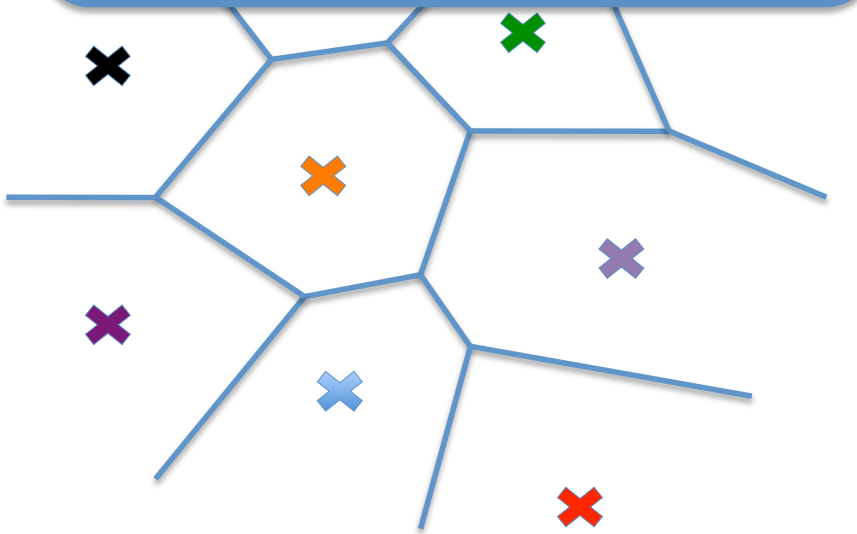
• RBMs, Factor models, PCA, Sparse Coding, Deep models



Local vs. Distributed Representations

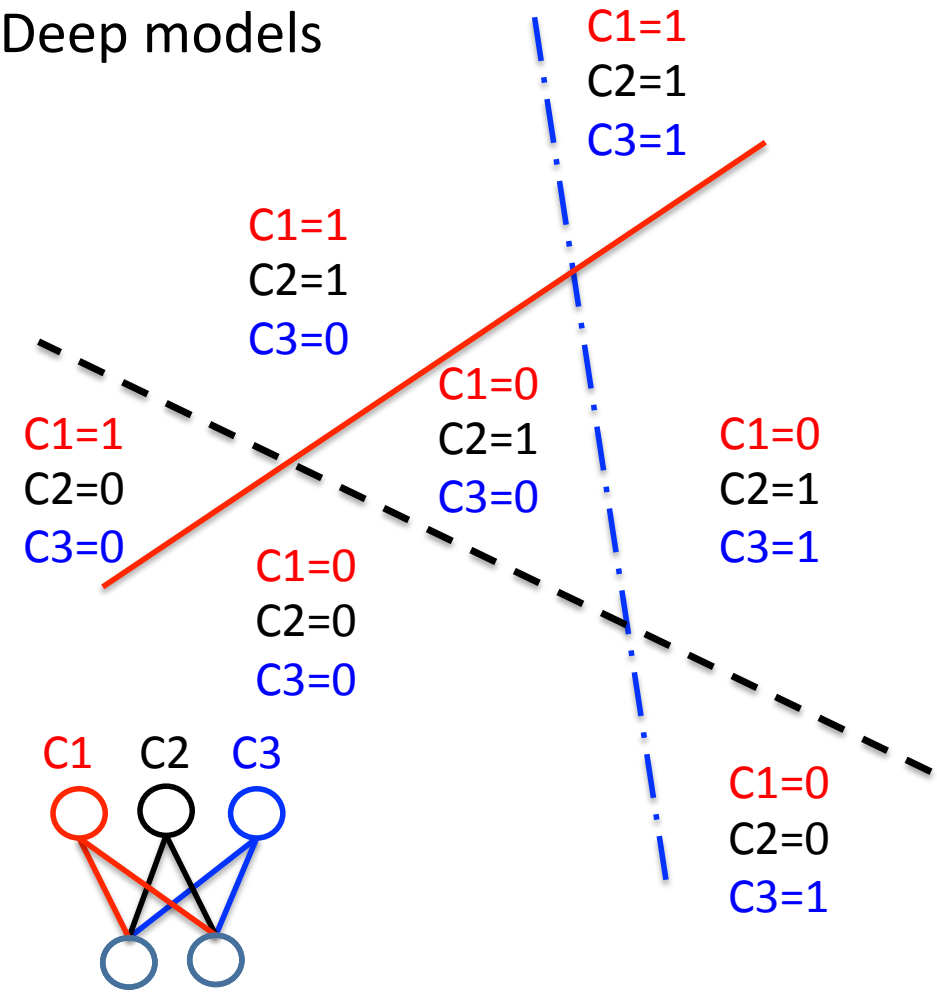
• Clustering, Nearest Neighbors, RBF SVM, local density estimators

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



Learned prototypes

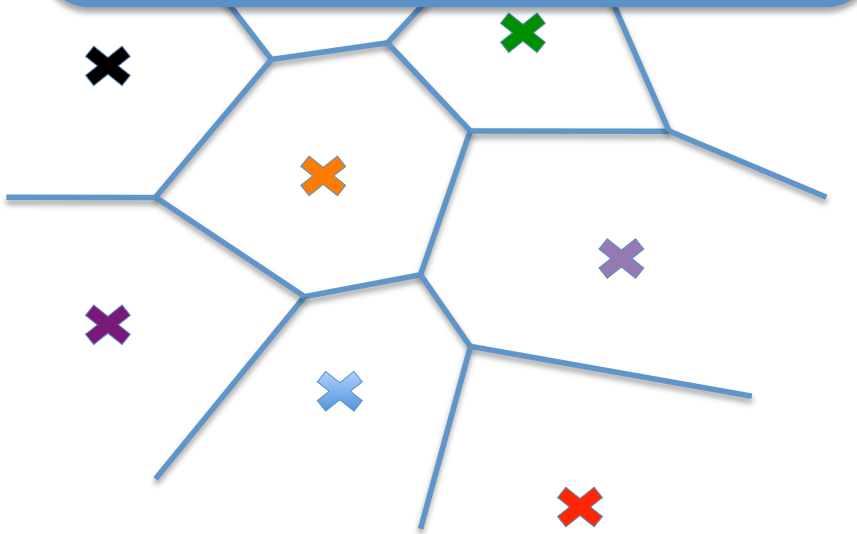
• RBMs, Factor models, PCA, Sparse Coding, Deep models



Local vs. Distributed Representations

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

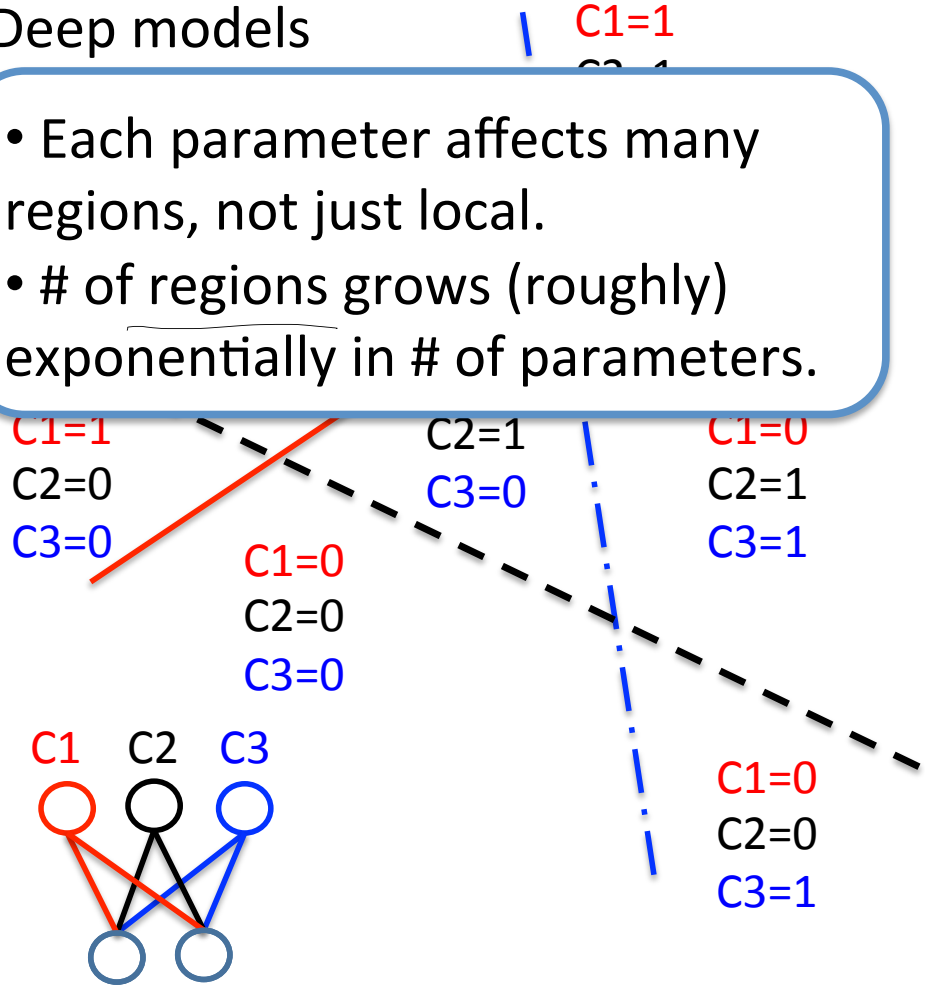
• Parameters for each region.
• # of regions is linear with # of parameters.



Learned prototypes

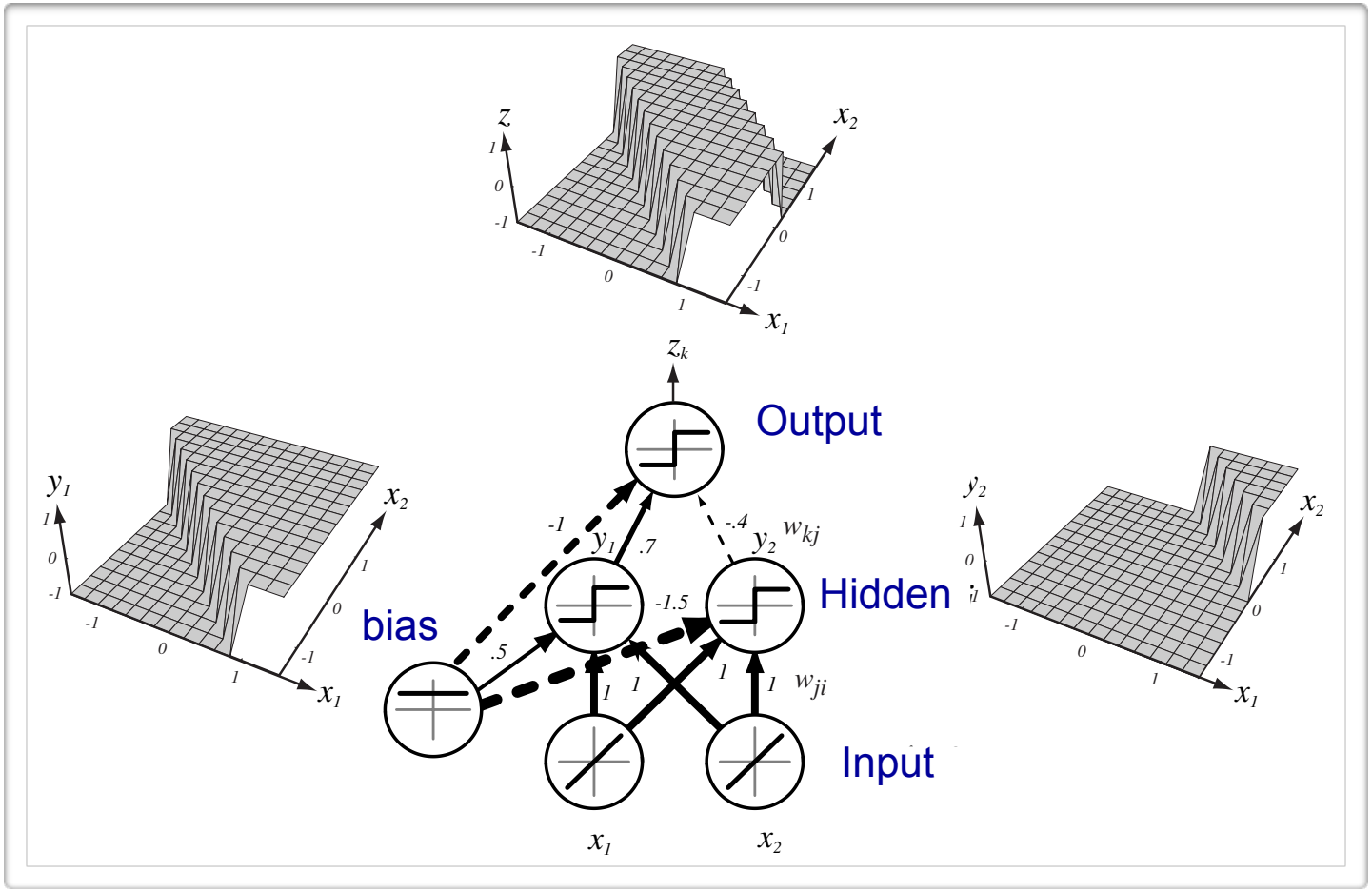
• RBMs, Factor models, PCA, Sparse Coding, Deep models

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



Capacity of Neural Nets

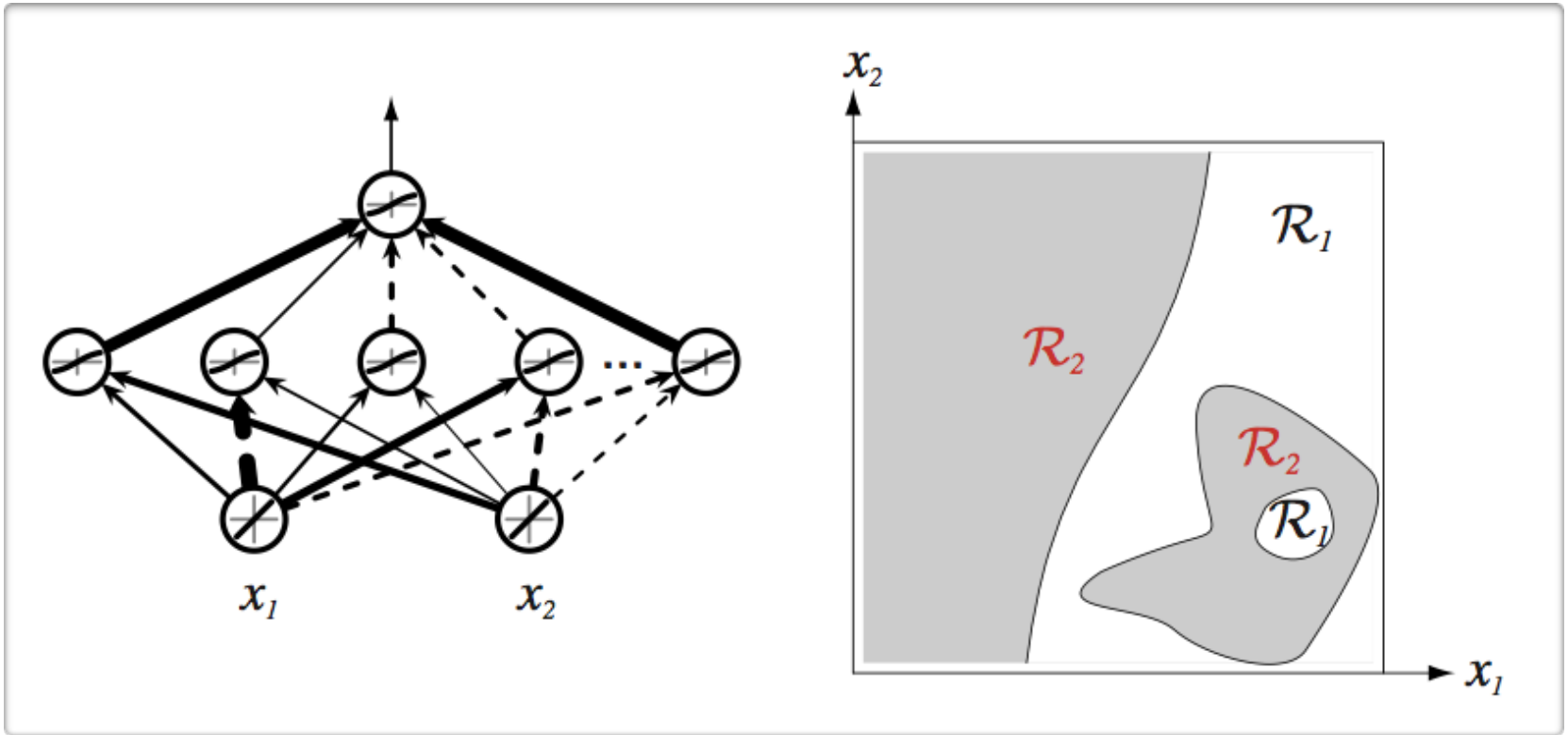
- Consider a single layer neural network



(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

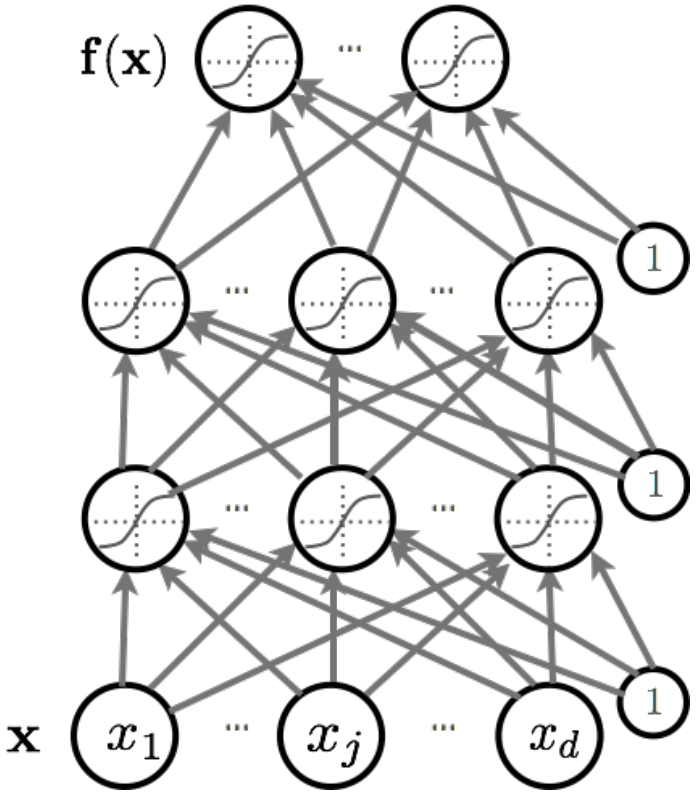
- 1 hidden layer neural networks are already a universal function approximator
- Implies the expressive power of deep networks are no larger than shallow networks
  - There always exists a shallow network that can represent any function representable by a deep (multi-layer) neural network
- But there 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(x) given an input 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: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
  - For t=1:T
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

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

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:
  - Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - A procedure to compute gradients: $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - Regularizer and its gradient: $\Omega(\boldsymbol{\theta}), \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

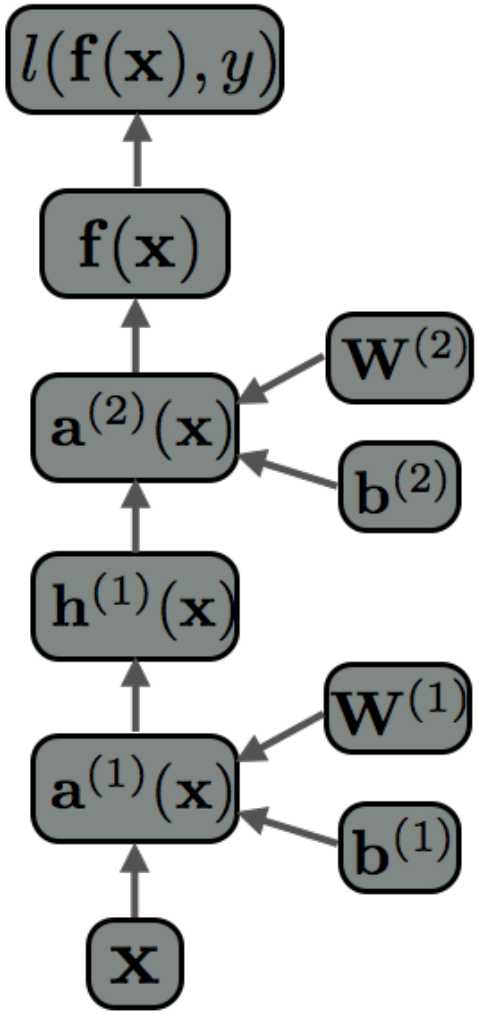
Computational Flow Graph

• 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

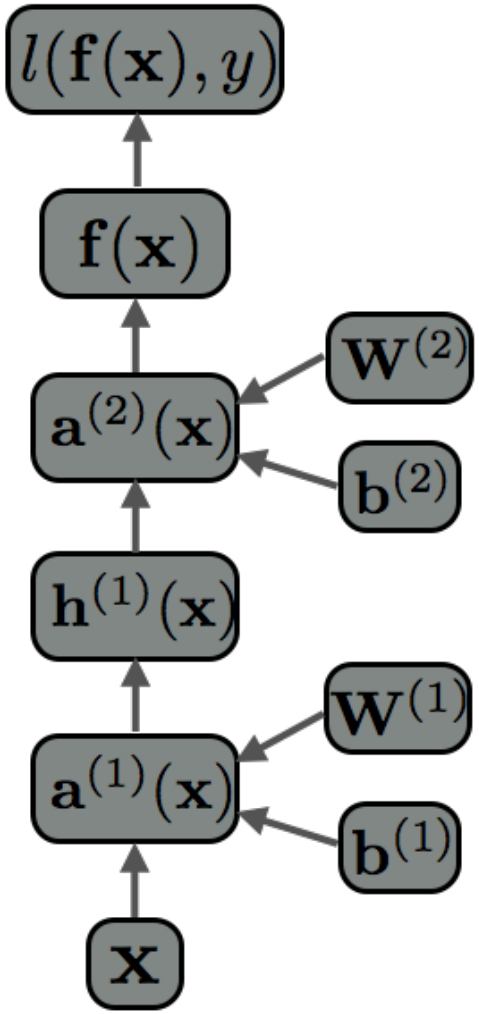


Computational Flow Graph

- 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



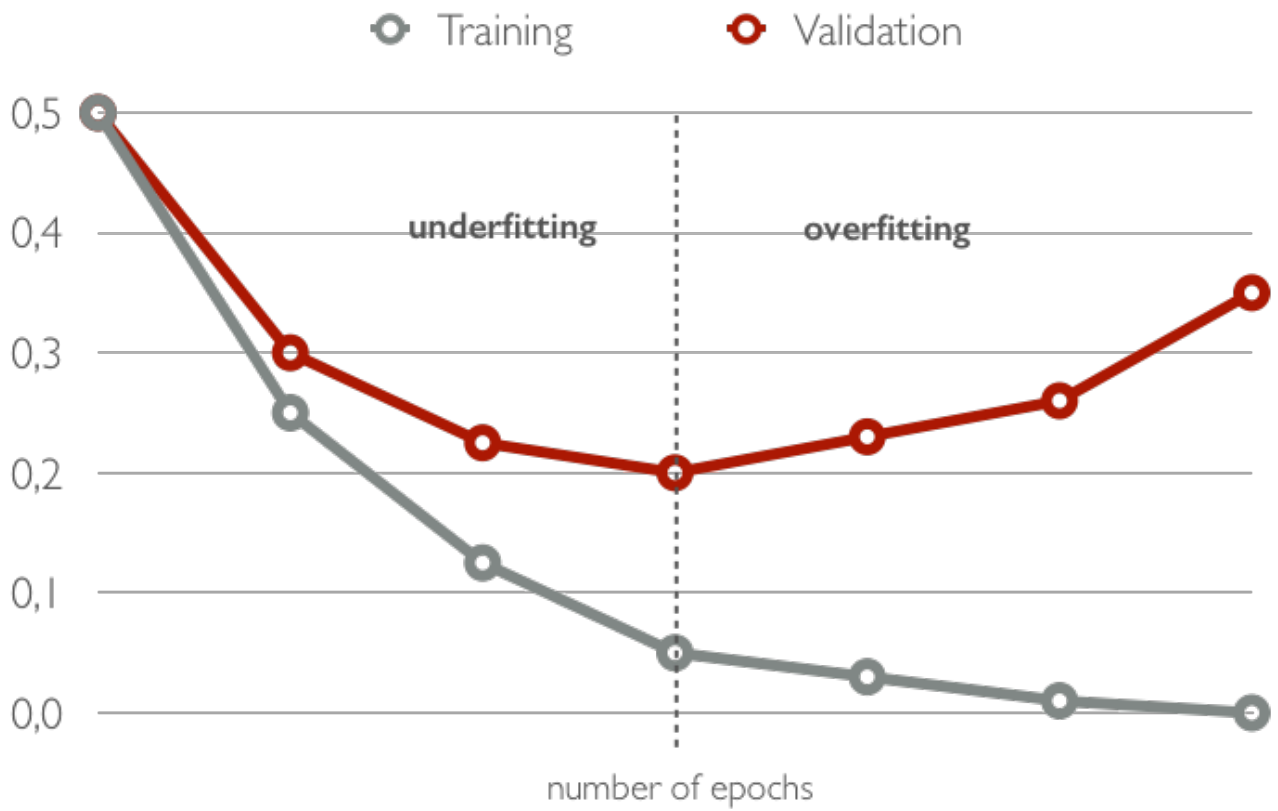
Model Selection

- 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:

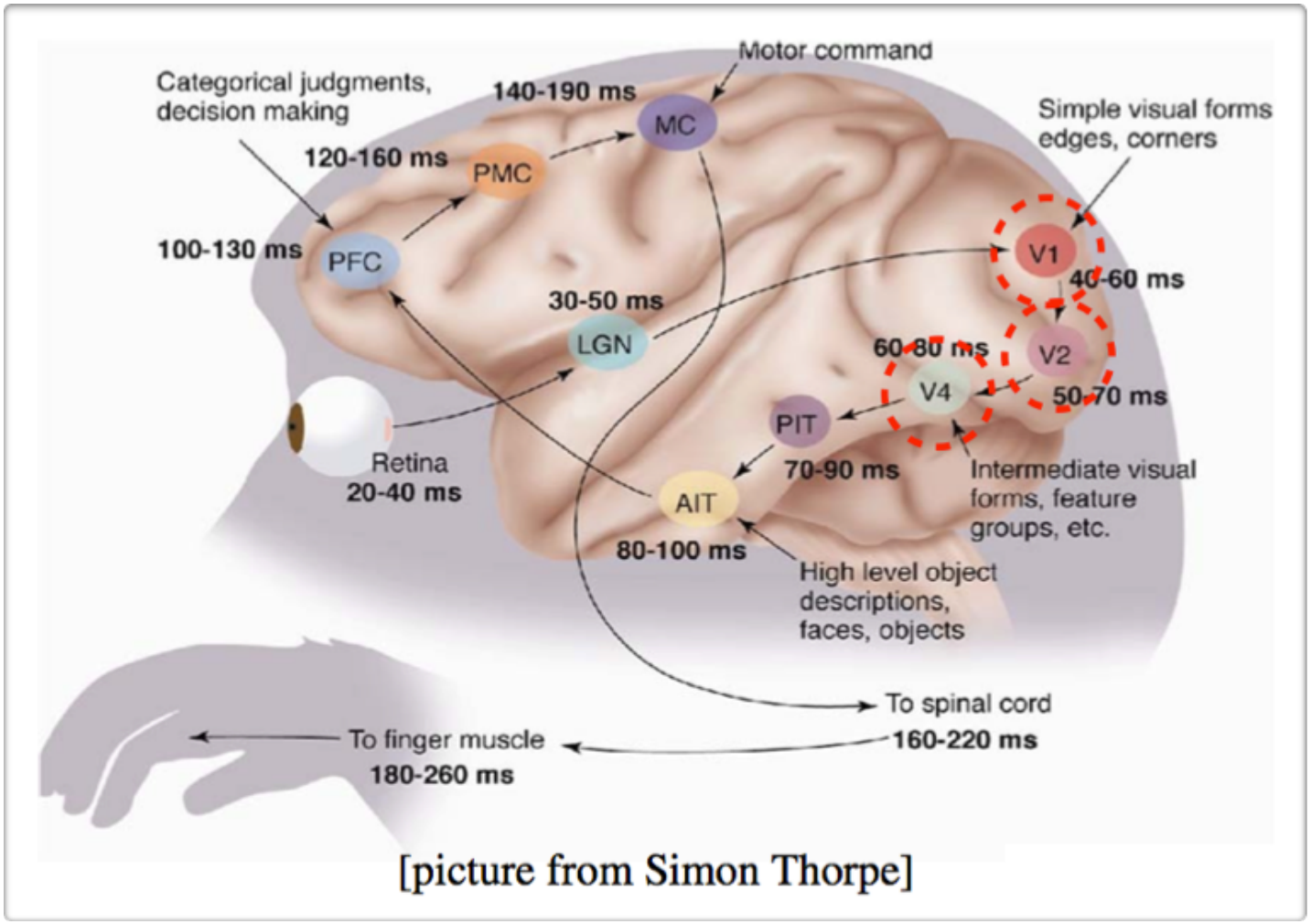
$$\overline{\nabla}_{\boldsymbol{\theta}}^{(t)} = \nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) + \beta \overline{\nabla}_{\boldsymbol{\theta}}^{(t-1)}$$

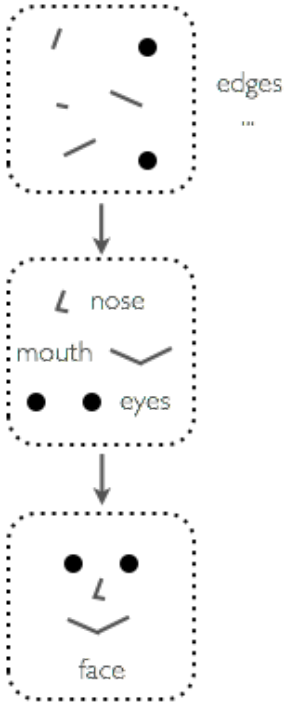
- can get ~~pass~~ past plateaus more quickly, by "gaining momentum"

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

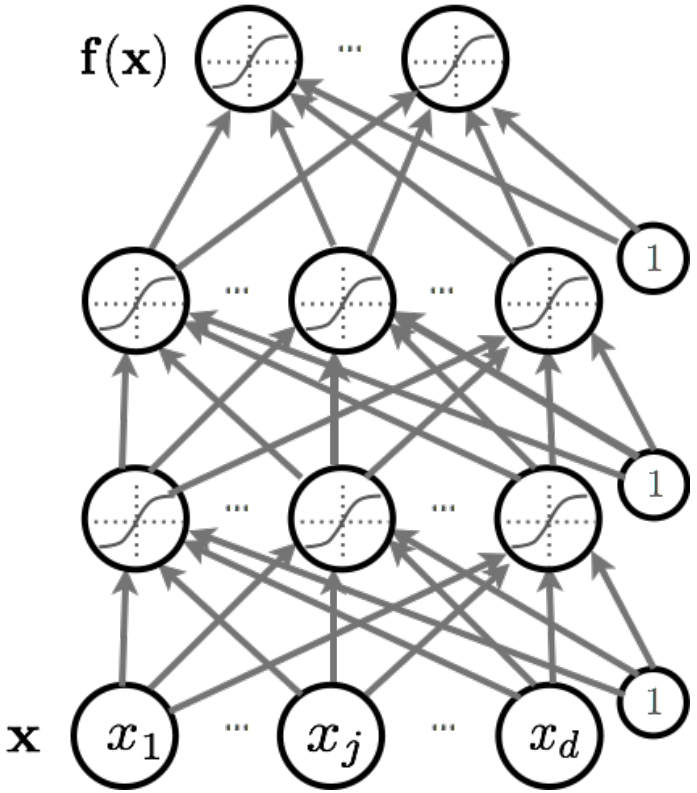
Inspiration from Visual Cortex





Feedforward Neural Networks

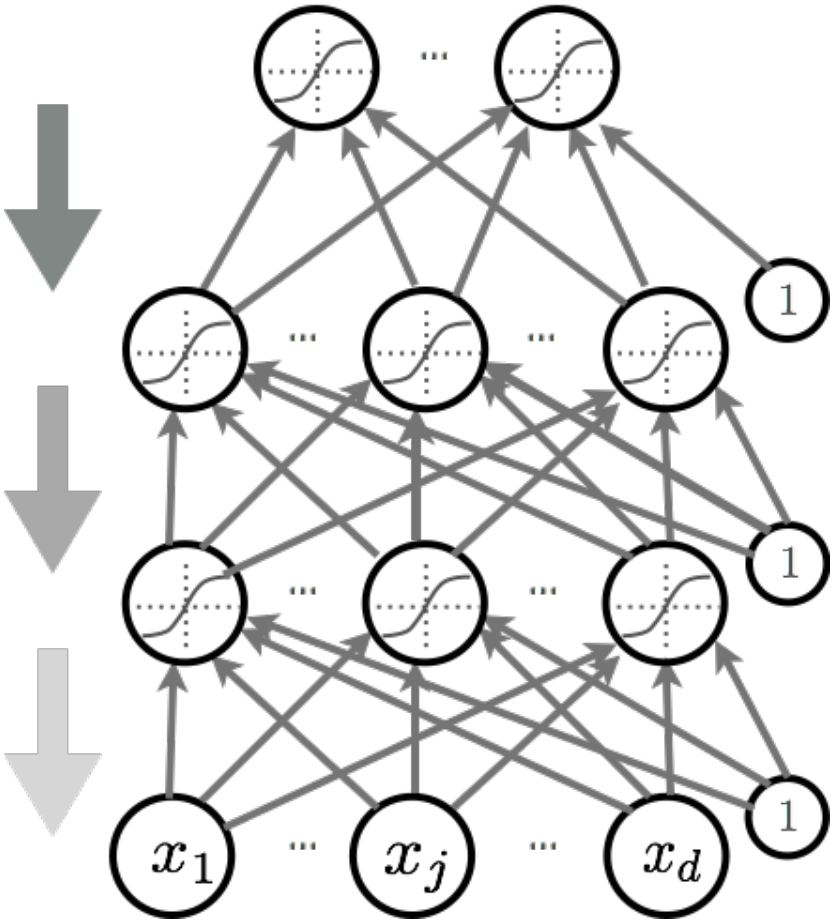
- How neural networks predict f(x) given an input x:
  - Forward propagation
  - Types of units
  - Capacity of neural networks
- How to train neural nets:
  - Loss function
  - Backpropagation with gradient descent
- More recent techniques:
  - Dropout
  - Batch normalization
  - Unsupervised Pre-training



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

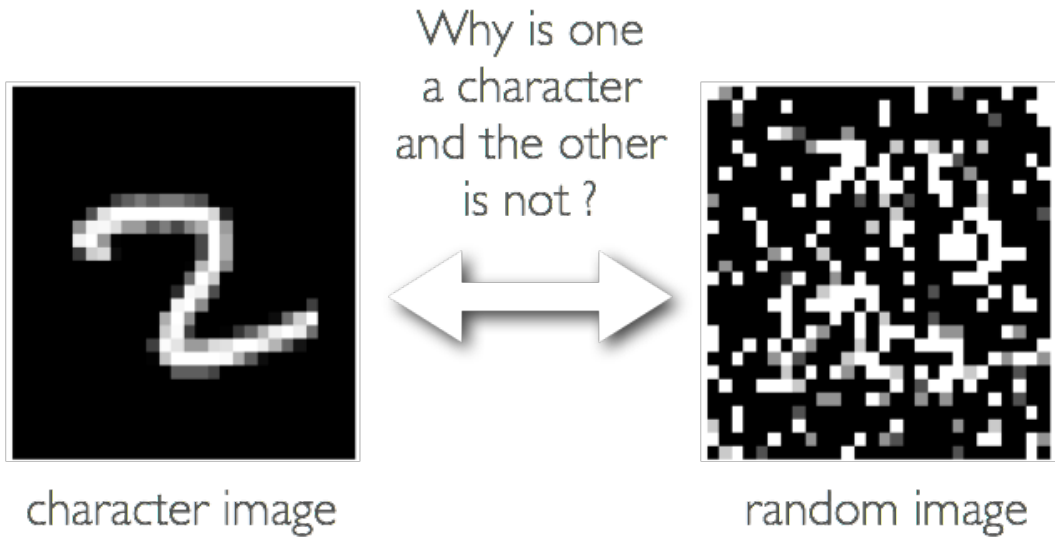


Why Training is Hard

- 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

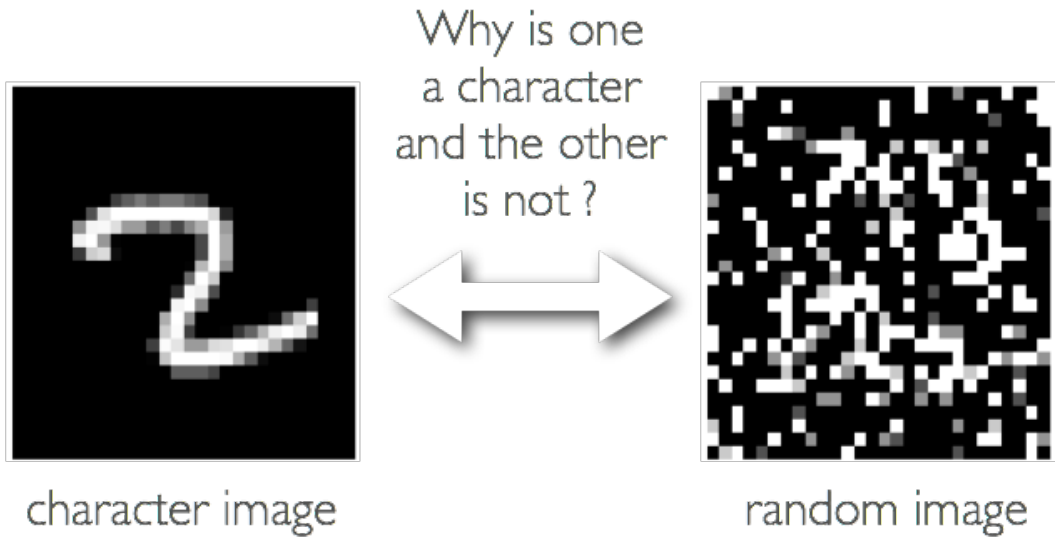
- Initialize hidden layers using unsupervised learning
  - Force network to represent latent structure of input distribution



  - Encourage hidden layers to encode that structure

Unsupervised Pre-training

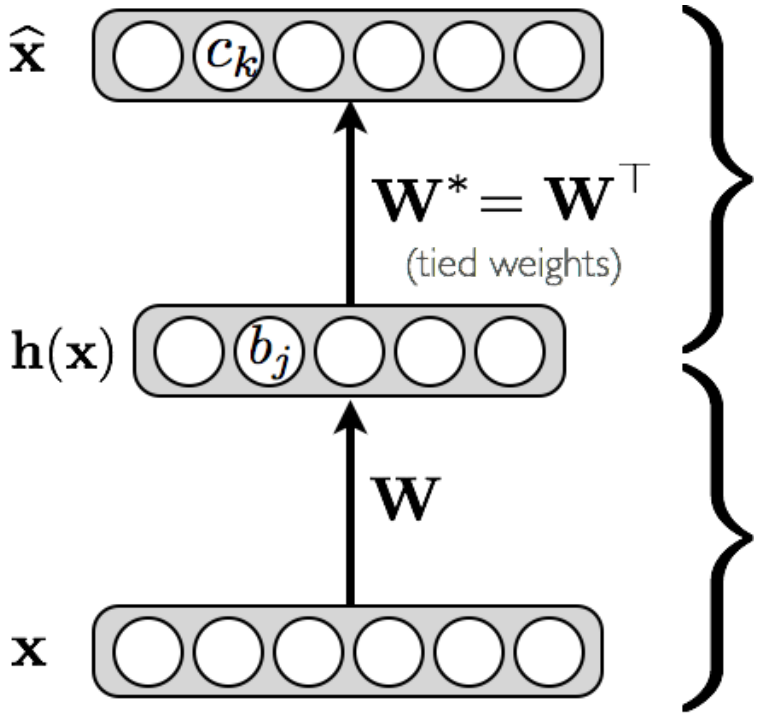
- Initialize hidden layers using unsupervised learning
  - This is a harder task than supervised learning (classification)



  - Hence we expect less overfitting

Autoencoders: Preview

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



Decoder

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

For binary units

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

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

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

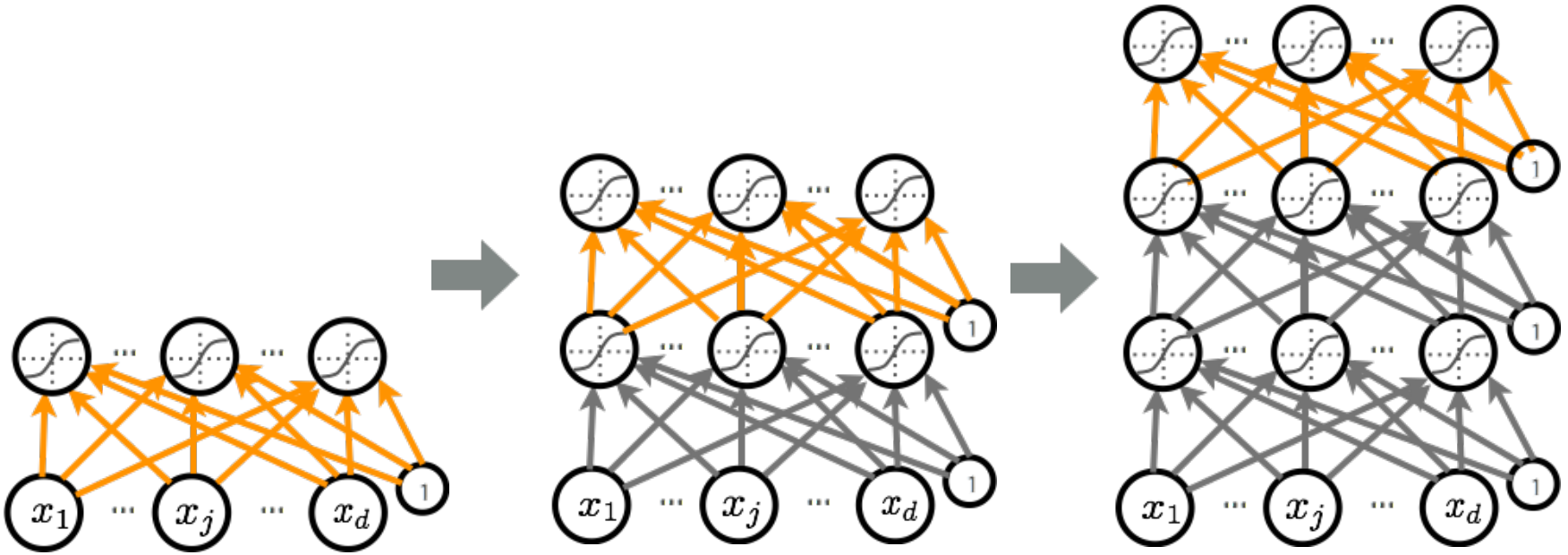
- Loss function for real-valued inputs

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

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

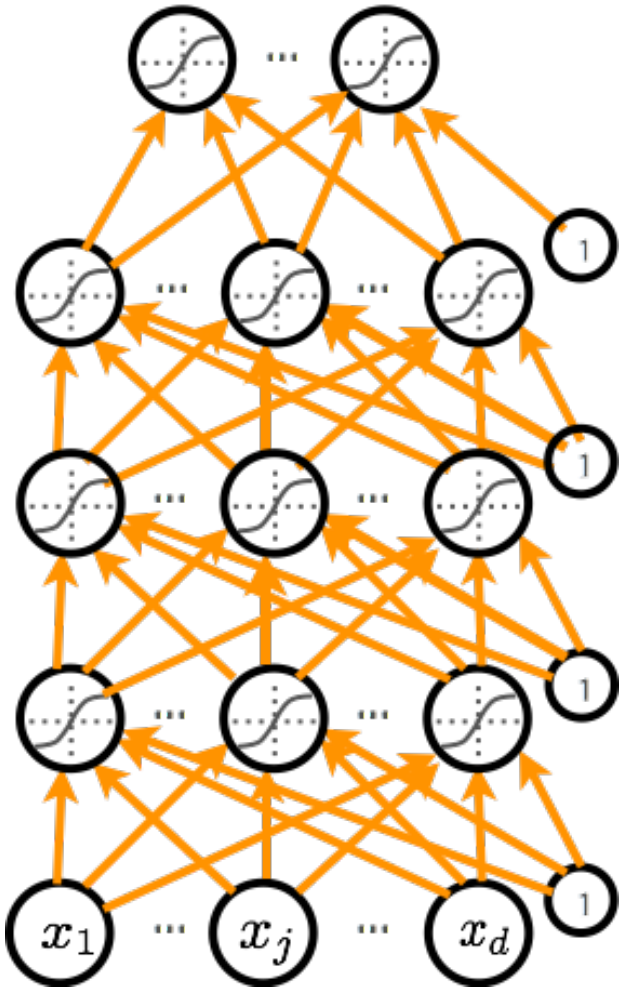
Pre-training

- 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



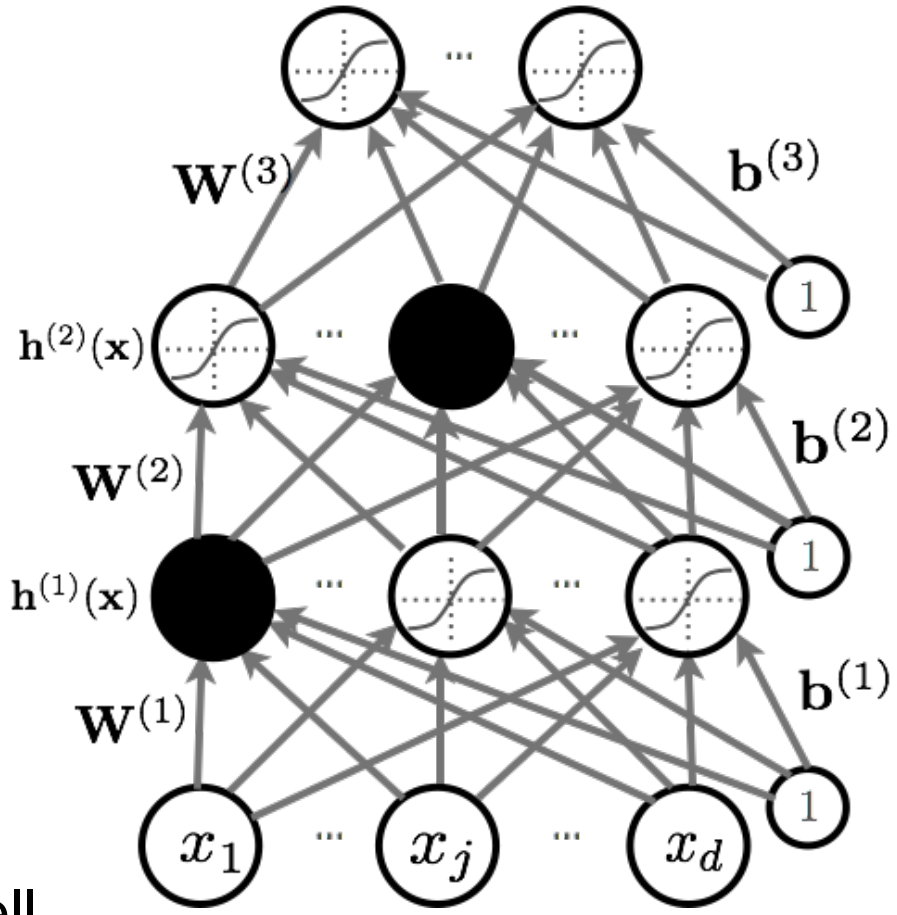
Why Training is Hard

- 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

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

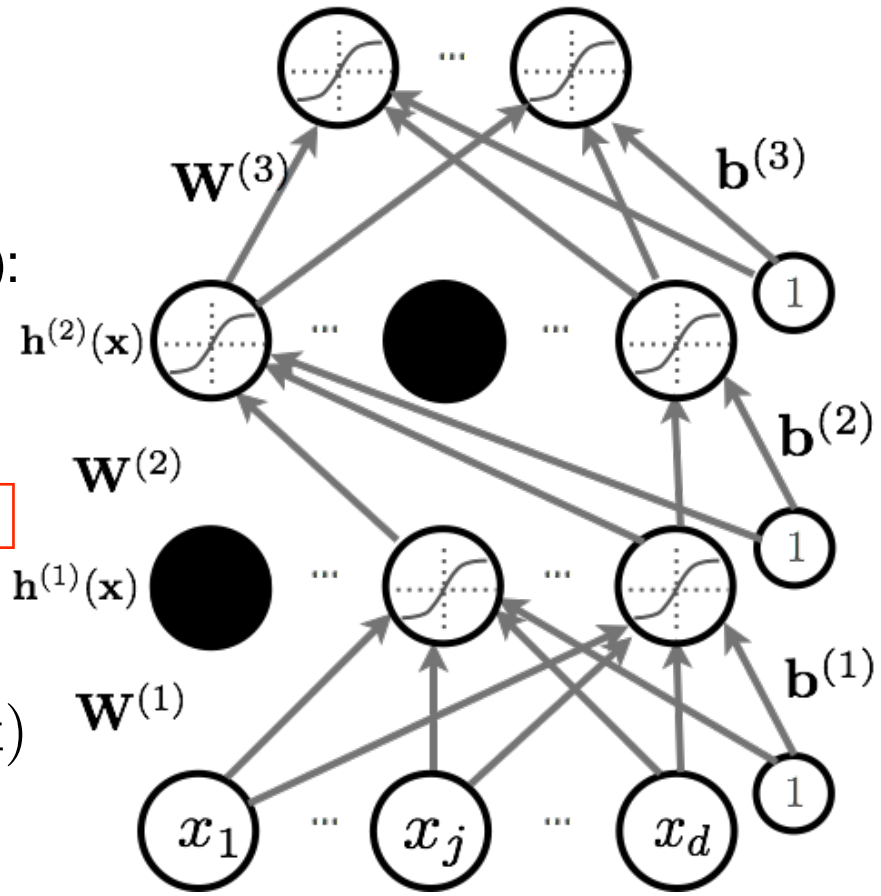
  - hidden layer activation (k=1 to L):

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

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: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

Learned linear transformation to adapt to non-linear activation function (γ and β are trained)

Batch Normalization

- Why normalize the pre-activation?
  - can help keep the pre-activation in a non-saturating regime (though the linear transform $y_i \leftarrow \gamma \widehat{x}_i + \beta$ 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