



# Feature generation for images

Machine Learning – CSE4546

Kevin Jamieson

University of Washington

November 27, 2018

©Kevin Jamieson

# Contains slides from...



- LeCun & Ranzato
- Russ Salakhutdinov
- Honglak Lee
- Google images...

# Convolution of images

(Note to EEs: deep learning uses the word “convolution” to mean what is usually known as “cross-correlation”, e.g., neither signal is flipped)

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image  $I$

1	0	1
0	1	0
1	0	1

Filter  $K$

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
Feature  
 $I * K$

# Convolution of images

(Note to EEs: deep learning uses the word “convolution” to mean what is usually known as “cross-correlation”, e.g., neither signal is flipped)

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

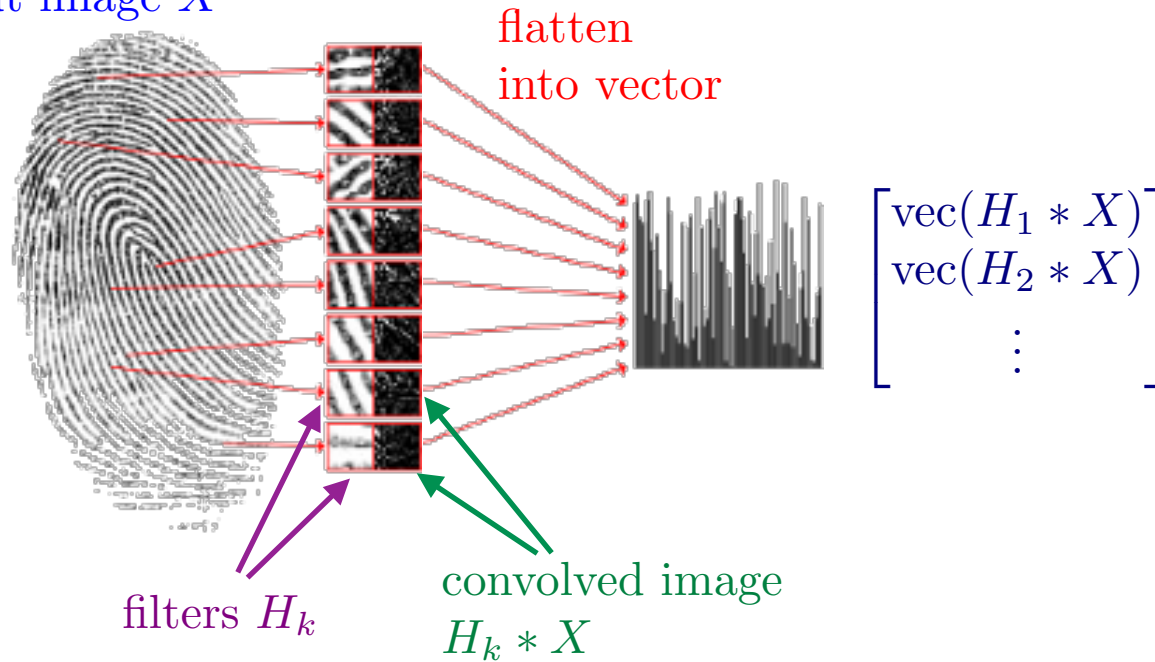
Image  $I$



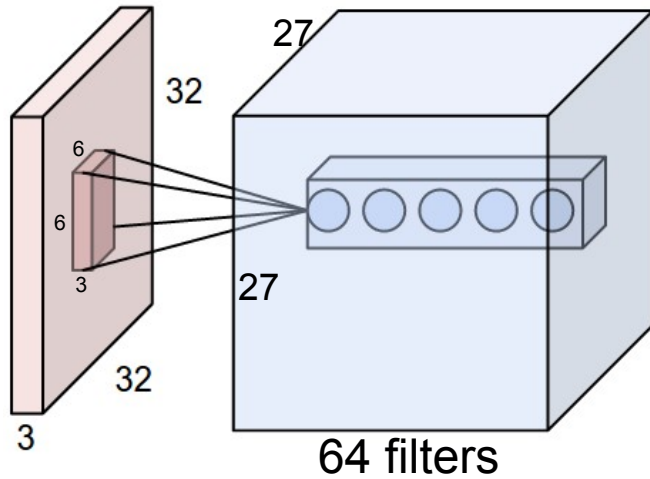
Operation	Filter $K$	Convolved Image $I * K$
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

# Convolution of images

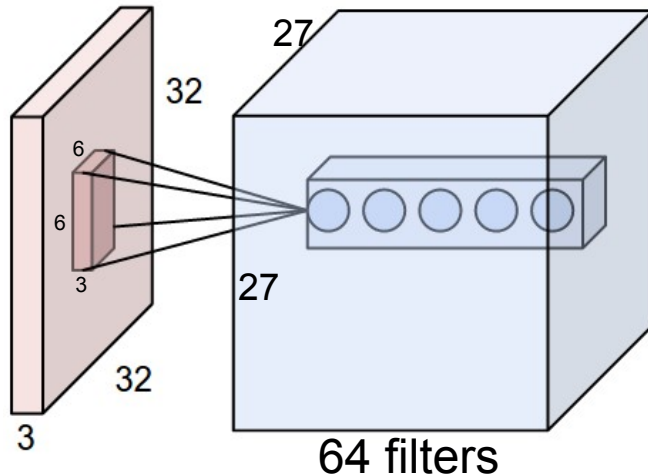
Input image  $X$



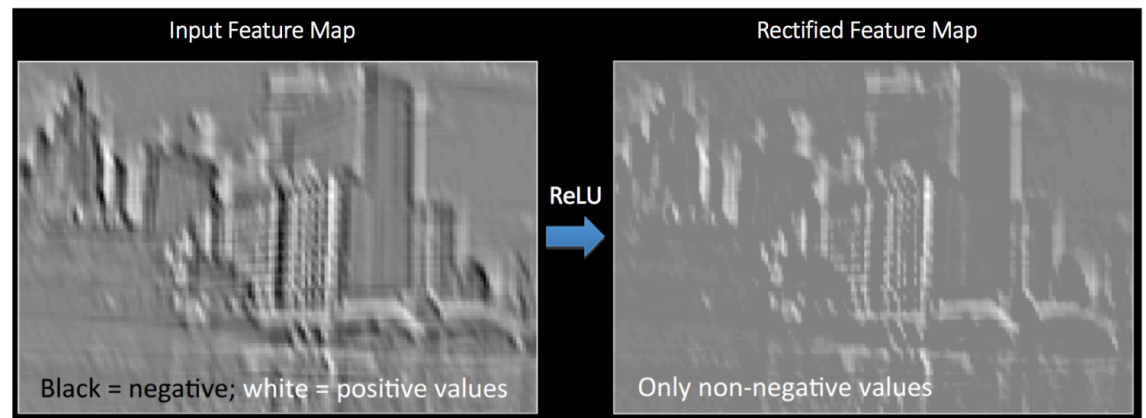
# Stacking convolved images



# Stacking convolved images



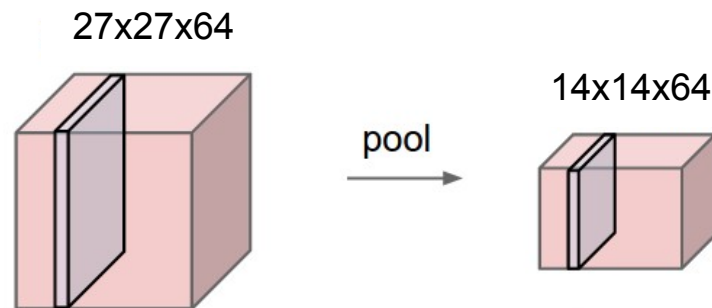
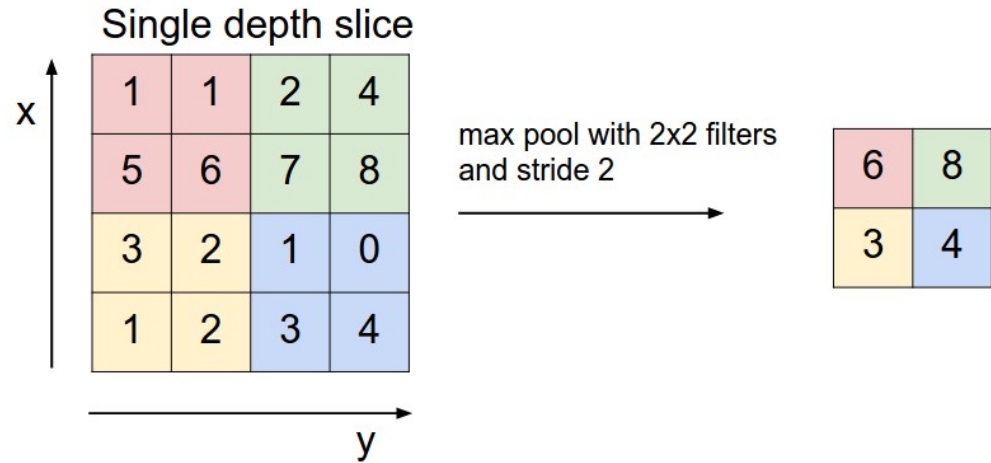
Apply Non-linearity to the output of each layer, Here: ReLu (rectified linear unit)



Other choices: sigmoid, arctan

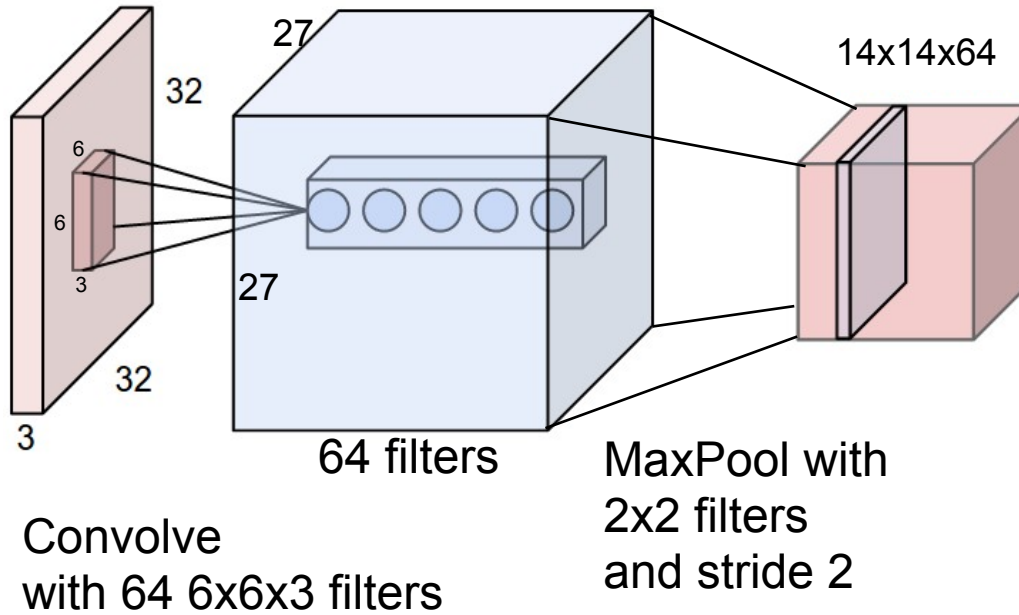
# Pooling

Pooling reduces the dimension and can be interpreted as “This filter had a high response in this general region”

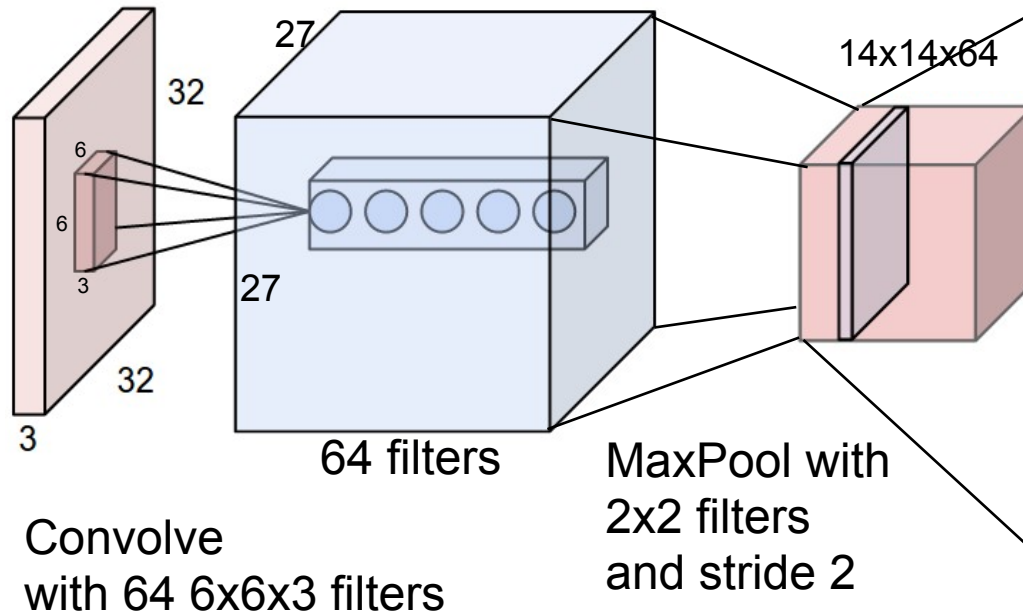




# Pooling Convolution layer



# Full feature pipeline



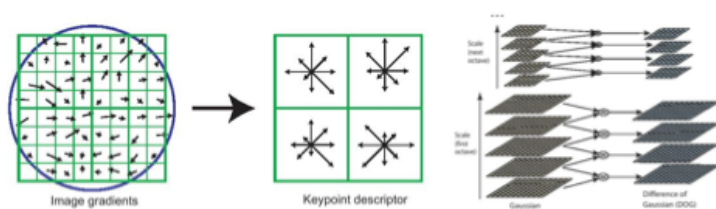
Flatten into a single vector of size  
 $14*14*64=12544$

How do we choose all the hyperparameters?

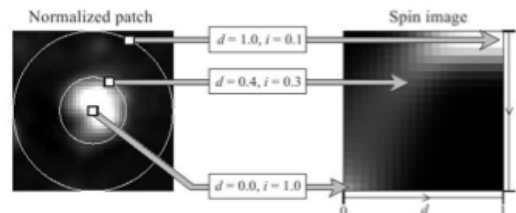
How do we choose the filters?

- Hand design them (digital signal processing, c.f. wavelets)
- Learn them (deep learning)

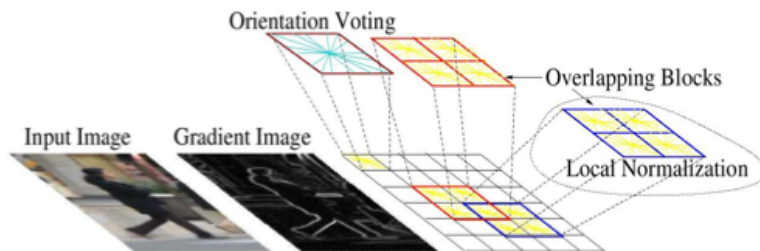
# Some hand-created image features



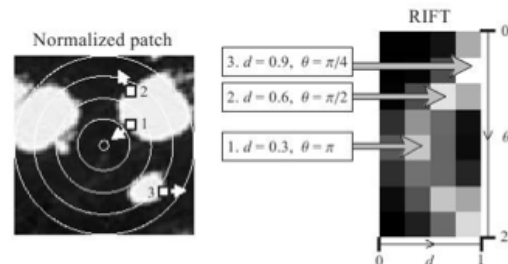
SIFT



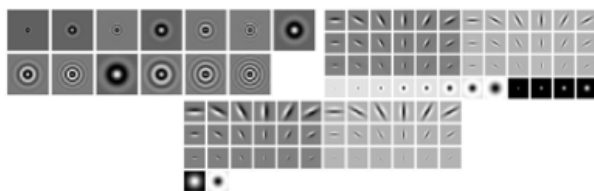
Spin Image



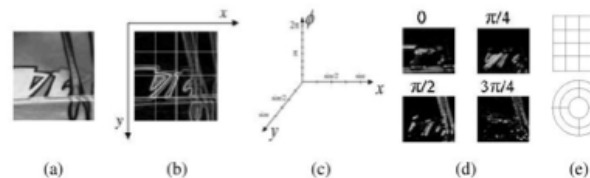
HoG



RIFT



Texton



GLOH

Slide from Honglak Lee



# ML Street Fight

Machine Learning – CSE546

Kevin Jamieson

University of Washington

November 27, 2018

# Mini case study



Inspired by Coates and Ng (2012)

Input is CIFAR-10 dataset: 50000 examples of 32x32x3 images

1. Construct set of patches by random selection from images
2. Standardize patch set (de-mean, norm 1, whiten, etc.)
3. Run k-means on random patches
4. Convolve each image with all patches (plus an offset)
5. Push through ReLu
6. Solve least squares for multiclass classification
7. Classify with argmax

# Mini case study



Methods of standardization:

# Mini case study

Dealing with class imbalance:

$$\hat{w} = \arg \min_w \sum_{i=1}^n \max\{0, 1 - y_i x_i^T w\}$$

Calibrate with ROC curve:

# Mini case study

Dealing with class imbalance:

$$\hat{w} = \arg \min_w \sum_{i=1}^n \max\{0, 1 - y_i x_i^T w\}$$

Change to weighted objective:

$$\hat{w}_\alpha = \arg \min_w \sum_{i=1}^n \frac{1}{\alpha} \mathbf{1}\{y_i = 1\} \max\{0, 1 - x_i^T w\} + \frac{1}{1-\alpha} \mathbf{1}\{y_i = -1\} \max\{0, 1 + x_i^T w\}$$

Natural choice:  $\alpha = \frac{\sum_{i=1}^n \mathbf{1}\{y_i=1\}}{n}$



# Mini case study

Dealing with class imbalance:

$$\hat{w} = \arg \min_w \sum_{i=1}^n \max\{0, 1 - y_i x_i^T w\}$$

Change to optimization procedure:

$$w_{t+1} = w_t - \gamma \nabla \ell((x_{I_t}, y_{I_t}), w_t)$$

$$\mathbb{P}(I_t = i | y_i = 1) \propto \frac{1}{\alpha}$$

$$\mathbb{P}(I_t = i | y_i = -1) \propto \frac{1}{1 - \alpha}$$

Natural choice:  $\alpha = \frac{\sum_{i=1}^n \mathbf{1}\{y_i=1\}}{n}$

# Mini case study

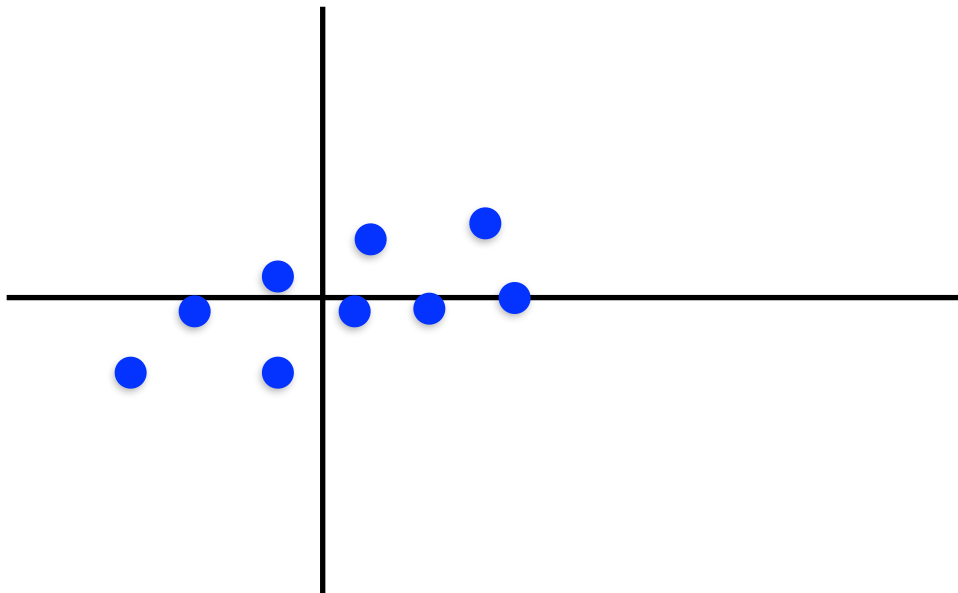
Dealing with outliers:

Observe  $\{(x_i, y_i)\}_{i=1}^n$ .  $\hat{w} = \arg \min_w \|\mathbf{X}w - \mathbf{y}\|^2 = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$

$x_i$  outliers 

Have undue influence on  $w$

$$\begin{aligned} \hat{y} &= \mathbf{X}\hat{w} = \mathbf{X}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \\ &= \mathbf{U}\mathbf{U}^T \mathbf{y} \end{aligned}$$



# Mini case study

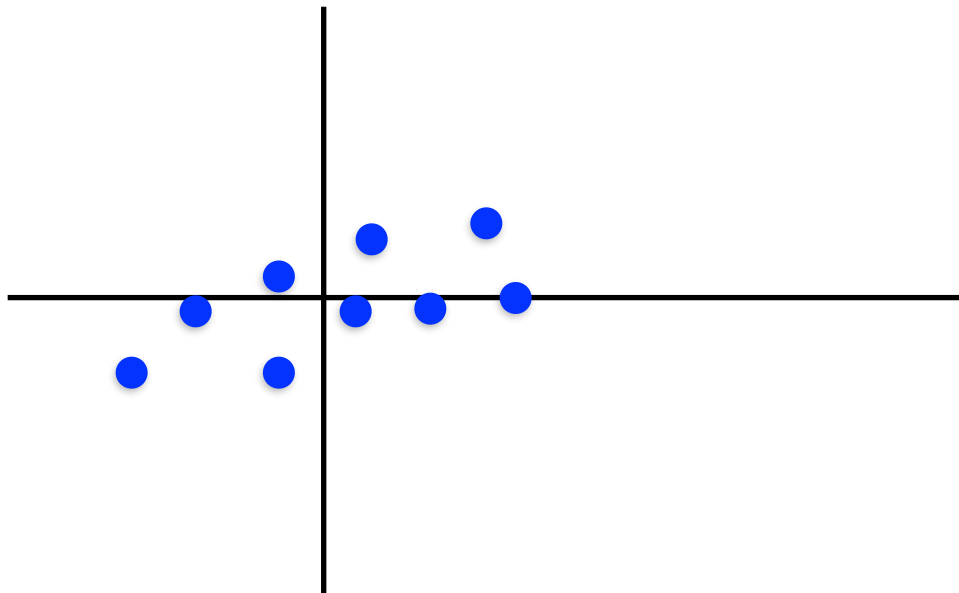
Dealing with outliers:

Observe  $\{(x_i, y_i)\}_{i=1}^n$ .  $\hat{w} = \arg \min_w \|\mathbf{X}w - \mathbf{y}\|^2 = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$

$x_i$  outliers 

Have undue influence on  $w$

$$\begin{aligned} \hat{\mathbf{y}} &= \mathbf{X}\hat{w} = \mathbf{X}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \\ &= \mathbf{U}\mathbf{U}^T \mathbf{y} \end{aligned}$$



Leverage score for  $x_i$ :  $(\mathbf{U}\mathbf{U}^T)_{i,i}$

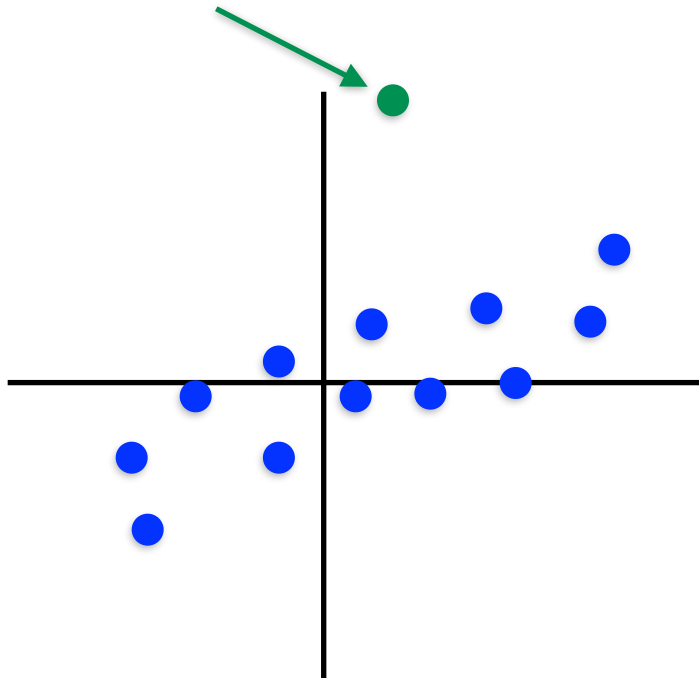
Remove points with large leverage scores from dataset

# Mini case study

Dealing with outliers:

Observe  $\{(x_i, y_i)\}_{i=1}^n$ .  $\hat{w} = \arg \min_w \|\mathbf{X}w - \mathbf{y}\|^2 = \sum_{i=1}^n (x_i^T w - y_i)^2$

$y_i$  outliers



Loss  $\ell_{LS}(z) = z^2$  is very sensitive to  $y_i$  outliers

$$\ell_{huber}(z) = \begin{cases} \frac{1}{2}z^2 & \text{if } |z| \leq 1 \\ |z| - \frac{1}{2} & \text{otherwise} \end{cases}$$

# Mini case study

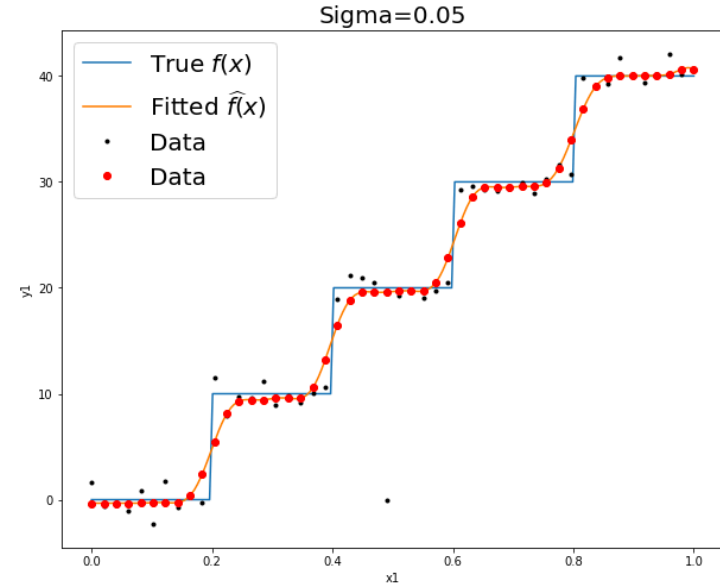
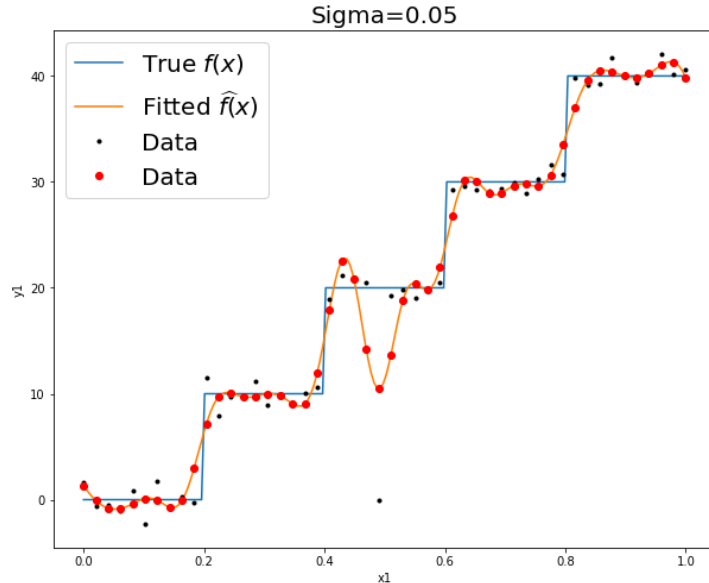
Dealing with outliers:

$y_i$  outliers

$$\arg \min_{\alpha} \sum_{i=1}^n \left( \sum_j k(x_i, x_j) \alpha_j - y_i \right)^2 + \lambda \sum_{i,j} \alpha_i \alpha_j k(x_i, x_j)$$

$$\ell_{huber}(z) = \begin{cases} \frac{1}{2} z^2 & \text{if } |z| \leq 1 \\ |z| - \frac{1}{2} & \text{otherwise} \end{cases}$$

$$\arg \min_{\alpha} \sum_{i=1}^n \ell_{huber} \left( \sum_j k(x_i, x_j) \alpha_j - y_i \right) + \lambda \sum_{i,j} \alpha_i \alpha_j k(x_i, x_j)$$



# Mini case study



Dealing with hyperparameters:



# Hyperparameter Optimization

Machine Learning – CSE546

Kevin Jamieson

University of Washington

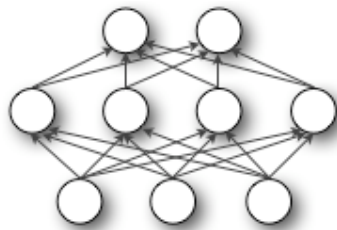
November 27, 2018





0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9 9 9 9 9 9 9

Training set



$N_{out} = 10$   
 $N_{hid}$   
 $N_{in} = 784$

0 0 0 0 0 0  
1 1 1 1 1 1  
2 2 2 2 2 2  
3 3 3 3 3 3  
4 4 4 4 4 4  
5 5 5 5 5 5  
6 6 6 6 6 6  
7 7 7 7 7 7  
8 8 8 8 8 8  
9 9 9 9 9 9

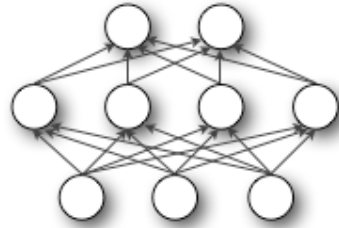
Eval set

### hyperparameters

learning rate  $\eta \in [10^{-3}, 10^{-1}]$

$\ell_2$ -penalty  $\lambda \in [10^{-6}, 10^{-1}]$

# hidden nodes  $N_{hid} \in [10^1, 10^3]$



$$N_{out} = 10$$

$$N_{hid}$$

$$N_{in} = 784$$



Hyperparameters

$(10^{-1.6}, 10^{-2.4}, 10^{1.7})$

$\hat{f}$

hyperparameters

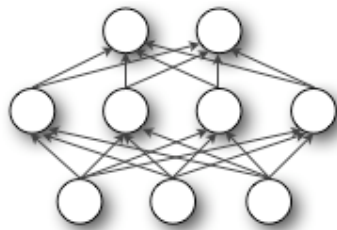
learning rate  $\eta \in [10^{-3}, 10^{-1}]$

$\ell_2$ -penalty  $\lambda \in [10^{-6}, 10^{-1}]$

# hidden nodes  $N_{hid} \in [10^1, 10^3]$

0000000000000000  
1111111111111111  
2222222222222222  
3333333333333333  
4444444444444444  
5555555555555555  
6666666666666666  
7777777777777777  
8888888888888888  
9999999999999999

Training set



$N_{out} = 10$   
 $N_{hid}$   
 $N_{in} = 784$

000000  
111111  
222222  
333333  
444444  
555555  
666666  
777777  
888888  
999999

Eval set

Hyperparameters  
( $10^{-1.6}, 10^{-2.4}, 10^{1.7}$ )

Eval-loss  
0.0577

$\hat{f}$

hyperparameters

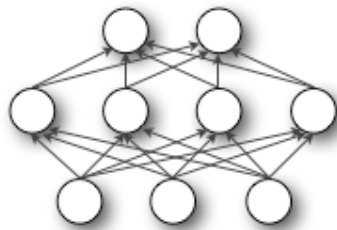
learning rate  $\eta \in [10^{-3}, 10^{-1}]$

$\ell_2$ -penalty  $\lambda \in [10^{-6}, 10^{-1}]$

# hidden nodes  $N_{hid} \in [10^1, 10^3]$

0000000000000000  
 1111111111111111  
 2222222222222222  
 3333333333333333  
 4444444444444444  
 5555555555555555  
 6666666666666666  
 7777777777777777  
 8888888888888888  
 9999999999999999

**Training set**



$N_{out} = 10$   
 $N_{hid}$   
 $N_{in} = 784$

000000  
 111111  
 222222  
 333333  
 444444  
 555555  
 666666  
 777777  
 888888  
 999999

**Eval set**

### Hyperparameters

### Eval-loss

$(10^{-1.6}, 10^{-2.4}, 10^{1.7})$	0.0577
$(10^{-1.0}, 10^{-1.2}, 10^{2.6})$	0.182
$(10^{-1.2}, 10^{-5.7}, 10^{1.4})$	0.0436
$(10^{-2.4}, 10^{-2.0}, 10^{2.9})$	0.0919
$(10^{-2.6}, 10^{-2.9}, 10^{1.9})$	0.0575
$(10^{-2.7}, 10^{-2.5}, 10^{2.4})$	0.0765
$(10^{-1.8}, 10^{-1.4}, 10^{2.6})$	0.1196
$(10^{-1.4}, 10^{-2.1}, 10^{1.5})$	0.0834
$(10^{-1.9}, 10^{-5.8}, 10^{2.1})$	0.0242
$(10^{-1.8}, 10^{-5.6}, 10^{1.7})$	0.029

### hyperparameters

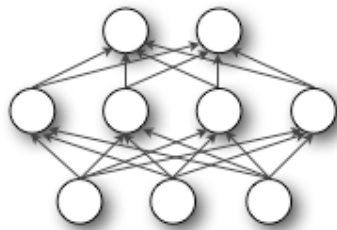
learning rate  $\eta \in [10^{-3}, 10^{-1}]$

$\ell_2$ -penalty  $\lambda \in [10^{-6}, 10^{-1}]$

# hidden nodes  $N_{hid} \in [10^1, 10^3]$

0000000000000000  
 1111111111111111  
 2222222222222222  
 3333333333333333  
 4444444444444444  
 5555555555555555  
 6666666666666666  
 7777777777777777  
 8888888888888888  
 9999999999999999

**Training set**



$N_{out} = 10$   
 $N_{hid}$   
 $N_{in} = 784$

000000  
 111111  
 222222  
 333333  
 444444  
 555555  
 666666  
 777777  
 888888  
 999999

**Eval set**

**Hyperparameters**

- $(10^{-1.6}, 10^{-2.4}, 10^{1.7})$
- $(10^{-1.0}, 10^{-1.2}, 10^{2.6})$
- $(10^{-1.2}, 10^{-5.7}, 10^{1.4})$
- $(10^{-2.4}, 10^{-2.0}, 10^{2.9})$
- $(10^{-2.6}, 10^{-2.9}, 10^{1.9})$
- $(10^{-2.7}, 10^{-2.5}, 10^{2.4})$
- $(10^{-1.8}, 10^{-1.4}, 10^{2.6})$
- $(10^{-1.4}, 10^{-2.1}, 10^{1.5})$
- $(10^{-1.9}, 10^{-5.8}, 10^{2.1})$
- $(10^{-1.8}, 10^{-5.6}, 10^{1.7})$

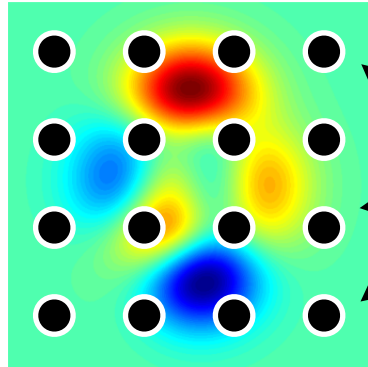
**Eval-loss**

- 0.0577
- 0.182
- 0.0436
- 0.0919
- 0.0575
- 0.0765
- 0.1196
- 0.0834
- 0.0242
- 0.029

How do we choose hyperparameters to train and evaluate?

# How do we choose hyperparameters to train and evaluate?

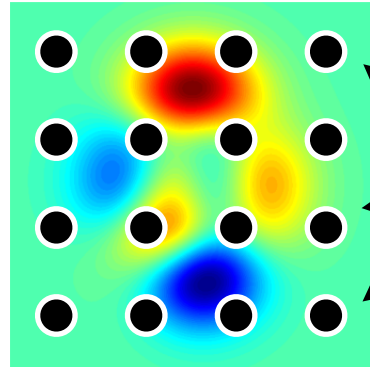
Grid search:



Hyperparameters  
on 2d uniform grid

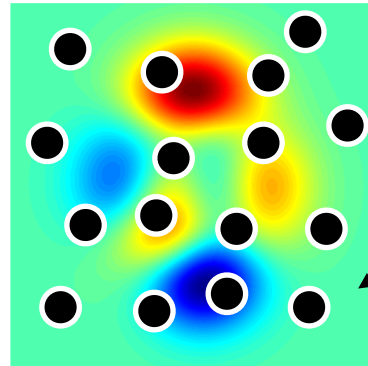
# How do we choose hyperparameters to train and evaluate?

Grid search:



Hyperparameters  
on 2d uniform grid

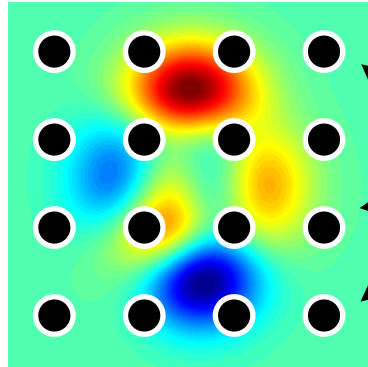
Random search:



Hyperparameters  
randomly chosen

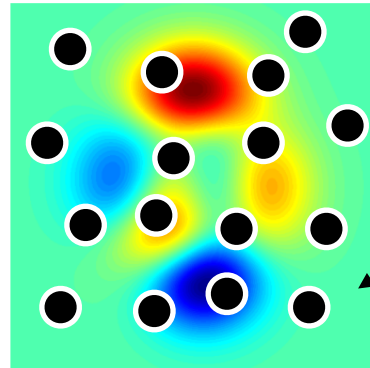
# How do we choose hyperparameters to train and evaluate?

Grid search:



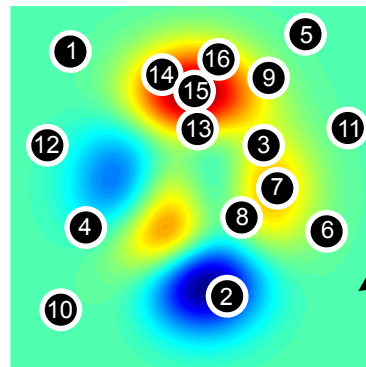
Hyperparameters  
on 2d uniform grid

Random search:



Hyperparameters  
randomly chosen

Bayesian Optimization:

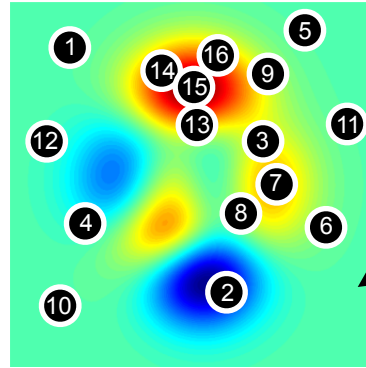


Hyperparameters  
**adaptively** chosen



# Bayesian Optimization:

How does it work?



Hyperparameters  
***adaptively*** chosen

# Recent work attempts to speed up hyperparameter evaluation by stopping poor performing settings before they are fully trained.

Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization. arXiv:1406.3896, 2014.

Alekh Agarwal, Peter Bartlett, and John Duchi. Oracle inequalities for computationally adaptive model selection. COLT, 2012.

Domhan, T., Springenberg, J. T., and Hutter, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, 2015.

András György and Levente Kocsis. Efficient multi-start strategies for local search algorithms. *JAIR*, 41, 2011.

Li, Jamieson, DeSalvo, Rostamizadeh, Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *ICLR* 2016.

## Hyperparameters

$(10^{-1.6}, 10^{-2.4}, 10^{1.7})$

$(10^{-1.0}, 10^{-1.2}, 10^{2.6})$

$(10^{-1.2}, 10^{-5.7}, 10^{1.4})$

$(10^{-2.4}, 10^{-2.0}, 10^{2.9})$

$(10^{-2.6}, 10^{-2.9}, 10^{1.9})$

$(10^{-2.7}, 10^{-2.5}, 10^{2.4})$

$(10^{-1.8}, 10^{-1.4}, 10^{2.6})$

$(10^{-1.4}, 10^{-2.1}, 10^{1.5})$

$(10^{-1.9}, 10^{-5.8}, 10^{2.1})$

$(10^{-1.8}, 10^{-5.6}, 10^{1.7})$

## Eval-loss

0.0577

0.182

0.0436

0.0919

0.0575

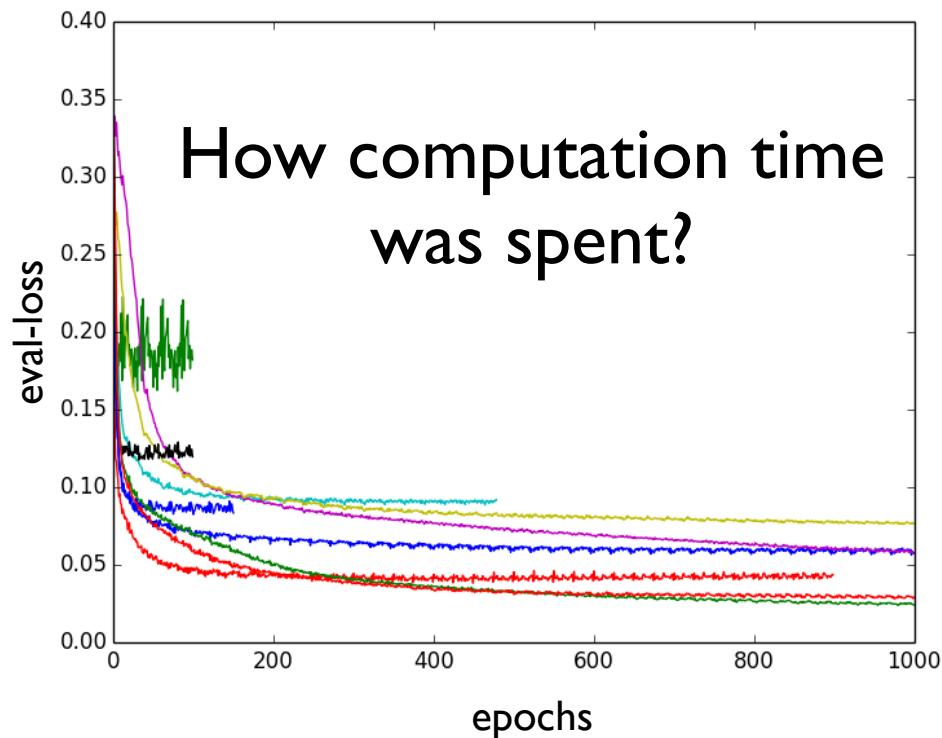
0.0765

0.1196

0.0834

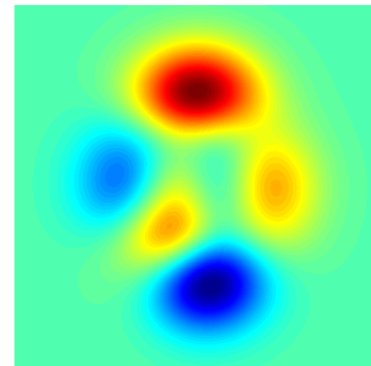
0.0242

0.029



# Hyperparameter Optimization

In general, hyperparameter optimization is non-convex optimization and little is known about the underlying function (only observe validation loss)



Your time is valuable, computers are cheap:

**Do not employ “grad student descent” for hyper parameter search.**

Write modular code that takes parameters as input and automate this embarrassingly parallel search. Use crowd resources (see [pywren](#))

Tools for different purposes:

- Very few evaluations: use random search (and pray) or be *clever*
- Few evaluations and long-running computations: see refs on last slide
- Moderate number of evaluations (but still  $\exp(\#\text{params})$ ) and high accuracy needed: use Bayesian Optimization
- Many evaluations possible: use random search. Why overthink it?



# Neural Networks

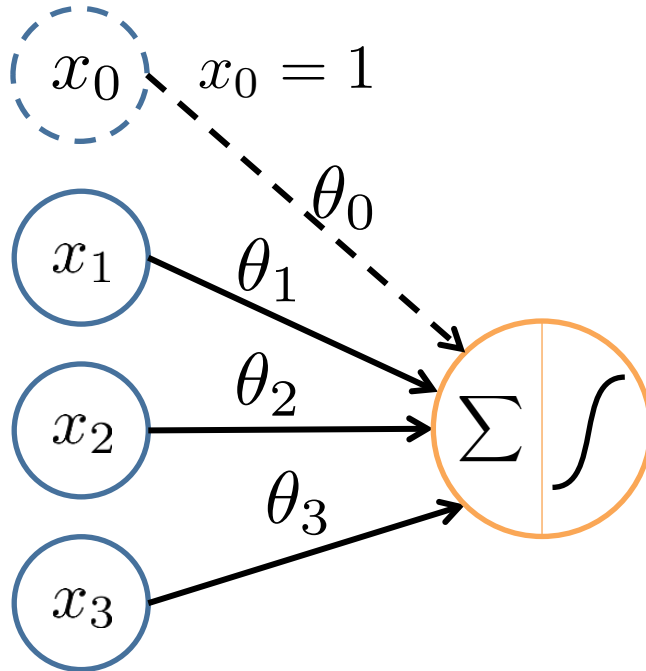
These slides were assembled by Eric Eaton, with grateful acknowledgement of the many others who made their course materials freely available online. Feel free to reuse or adapt these slides for your own academic purposes, provided that you include proper attribution. Please send comments and corrections to Eric.

# Neural Networks

- Origins: Algorithms that try to mimic the brain
- 40s and 50s: Hebbian learning and Perceptron
- Perceptrons book in 1969 and the XOR problem
- Very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications
- Artificial neural networks are not nearly as complex or intricate as the actual brain structure

# Single Node

“bias unit”

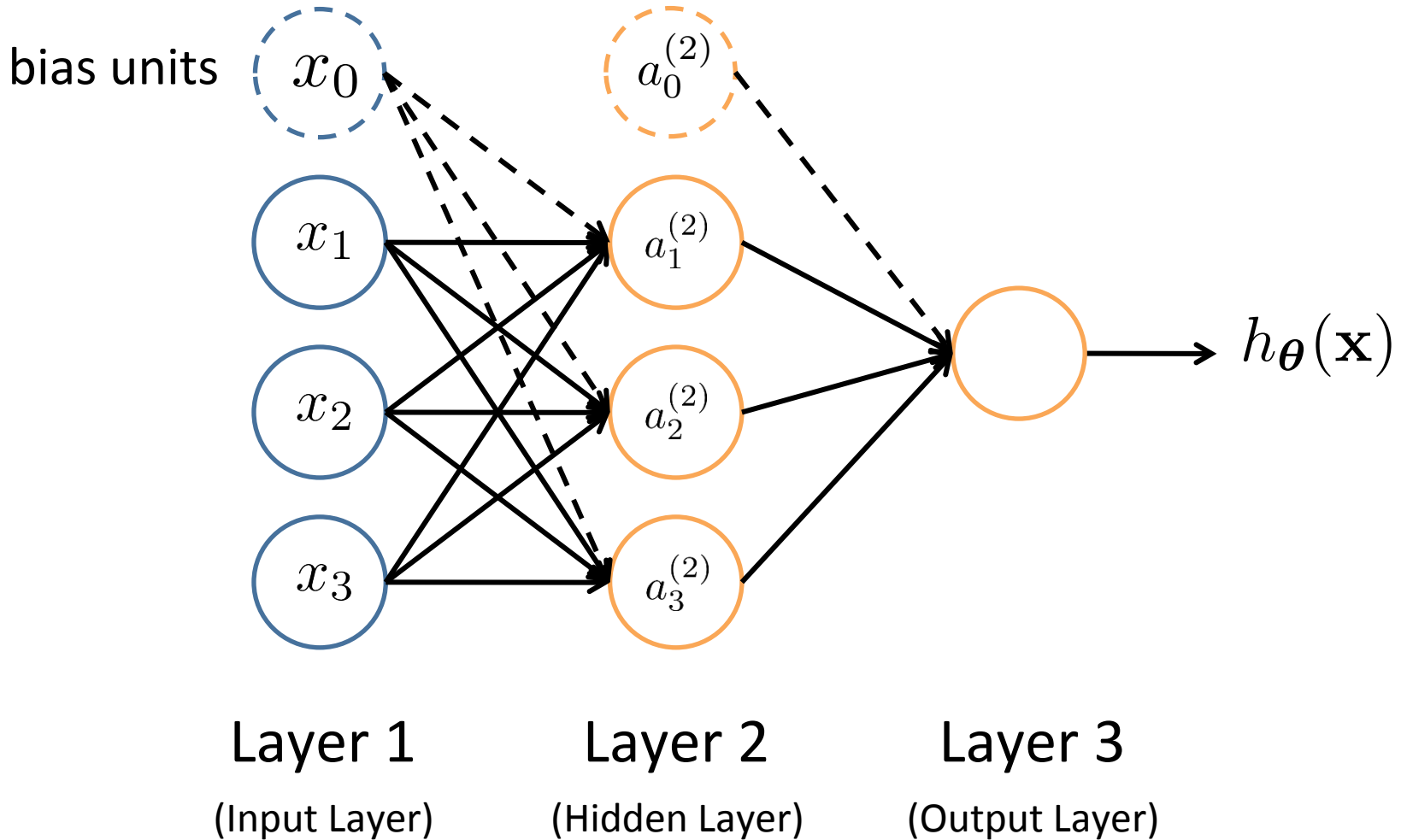


$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

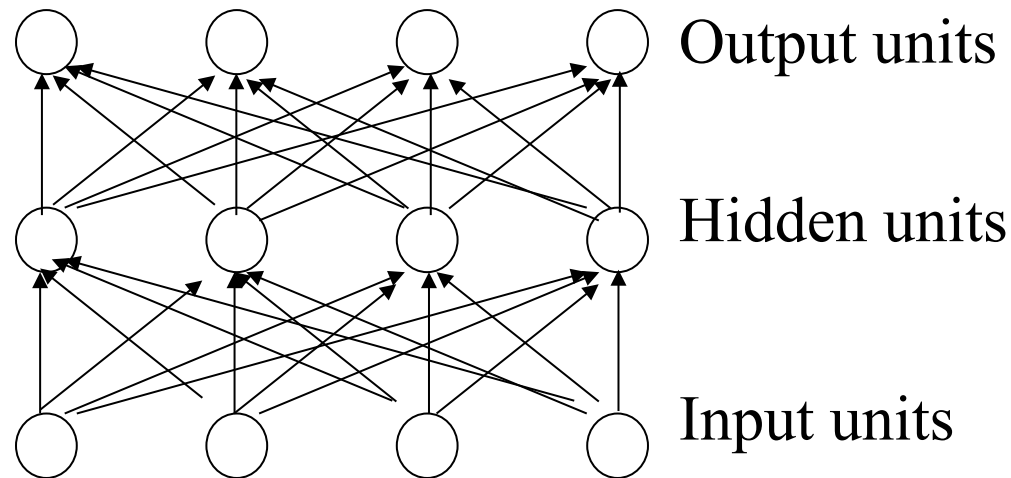
$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) \\ = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

Sigmoid (logistic) activation function:  $g(z) = \frac{1}{1 + e^{-z}}$

# Neural Network



# Neural networks Terminology



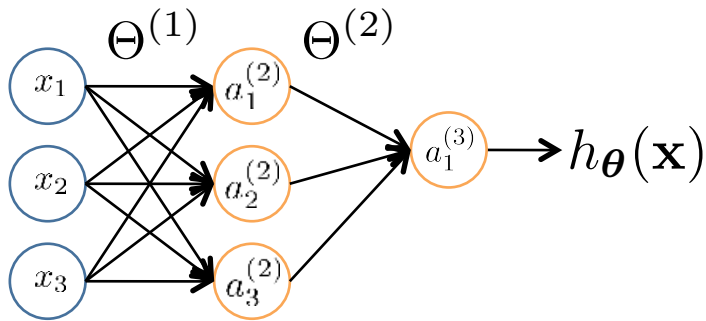
Layered feed-forward network

- Neural networks are made up of **nodes** or **units**, connected by **links**
- Each link has an associated **weight** and **activation level**
- Each node has an **input function** (typically summing over weighted inputs), an **activation function**, and an **output**



# Feed-Forward Process

- Input layer units are set by external data, which causes their output links to be **activated** at the specified level
- Working forward through the network, the **input function** of each unit is applied to compute the input value
  - Usually this is just the weighted sum of the activation on the links feeding into this node
- The **activation function** transforms this input function into a final value
  - Typically this is a **nonlinear** function, often a **sigmoid** function corresponding to the “threshold” of that node



$a_i^{(j)}$  = “activation” of unit  $i$  in layer  $j$   
 $\Theta^{(j)}$  = weight matrix stores parameters from layer  $j$  to layer  $j + 1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$

$$h_{\Theta}(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

If network has  $s_j$  units in layer  $j$  and  $s_{j+1}$  units in layer  $j+1$ ,  
 then  $\Theta^{(j)}$  has dimension  $s_{j+1} \times (s_j+1)$  .

$$\Theta^{(1)} \in \mathbb{R}^{3 \times 4} \quad \Theta^{(2)} \in \mathbb{R}^{1 \times 4}$$

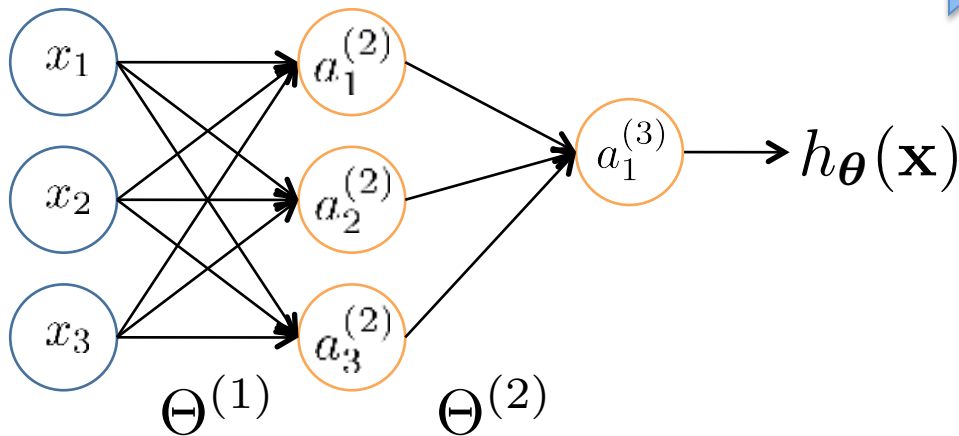
# Vectorization

$$a_1^{(2)} = g \left( \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \right) = g \left( z_1^{(2)} \right)$$

$$a_2^{(2)} = g \left( \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \right) = g \left( z_2^{(2)} \right)$$

$$a_3^{(2)} = g \left( \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \right) = g \left( z_3^{(2)} \right)$$

$$h_{\Theta}(\mathbf{x}) = g \left( \Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)} \right) = g \left( z_1^{(3)} \right)$$



Feed-Forward Steps:

$$\mathbf{z}^{(2)} = \Theta^{(1)} \mathbf{x}$$

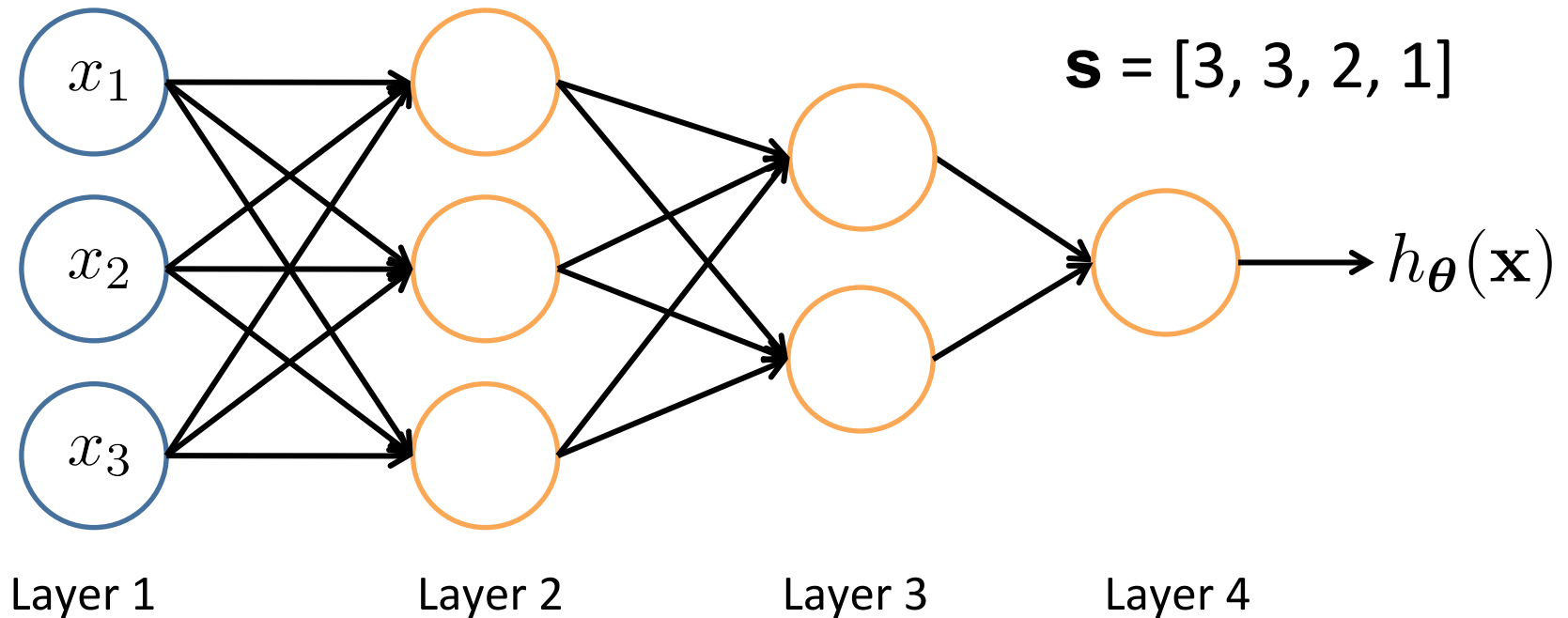
$$\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$$

Add  $a_0^{(2)} = 1$

$$\mathbf{z}^{(3)} = \Theta^{(2)} \mathbf{a}^{(2)}$$

$$h_{\Theta}(\mathbf{x}) = \mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$$

# Other Network Architectures



$L$  denotes the number of layers

$\mathbf{s} \in \mathbb{N}^{+L}$  contains the numbers of nodes at each layer

- Not counting bias units

- Typically,  $s_0 = d$  (# input features) and  $s_{L-1} = K$  (# classes)

# Multiple Output Units: One-vs-Rest



Pedestrian



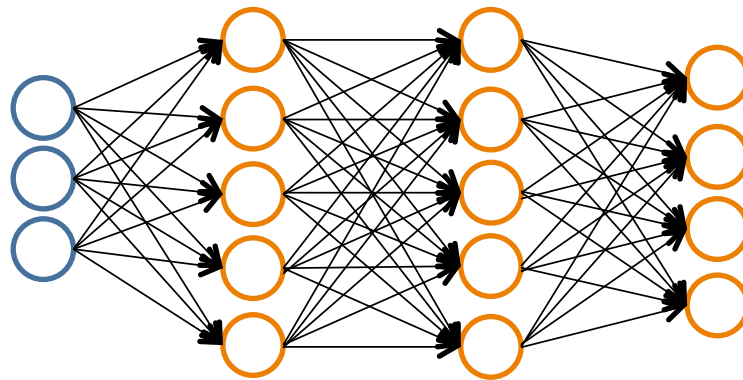
Car



Motorcycle



Truck



$$h_{\Theta}(\mathbf{x}) \in \mathbb{R}^K$$

We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

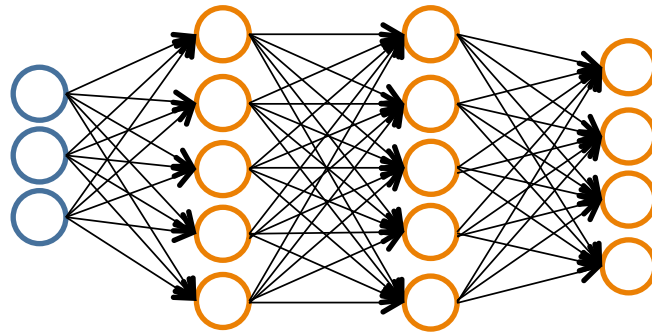
$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

# Multiple Output Units: One-vs-Rest



$$h_{\Theta}(\mathbf{x}) \in \mathbb{R}^K$$

We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

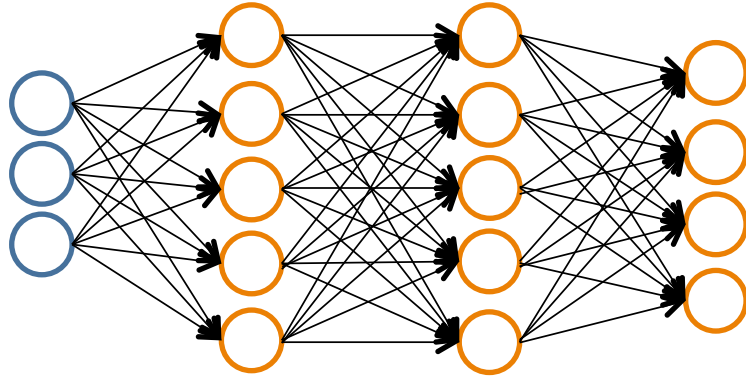
$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

- Given  $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$
- Must convert labels to 1-of- $K$  representation

– e.g.,  $y_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$  when motorcycle,  $y_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$  when car, etc.

# Neural Network Classification



**Given:**

$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$

$\mathbf{s} \in \mathbb{N}^{+L}$  contains # nodes at each layer  
–  $s_0 = d$  (# features)

## Binary classification

$y = 0$  or  $1$

1 output unit ( $s_{L-1} = 1$ )

## Multi-class classification ( $K$ classes)

$\mathbf{y} \in \mathbb{R}^K$  e.g.  $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ,  $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ ,  $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ ,  $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$   
pedestrian car motorcycle truck

$K$  output units ( $s_{L-1} = K$ )

# Understanding Representations

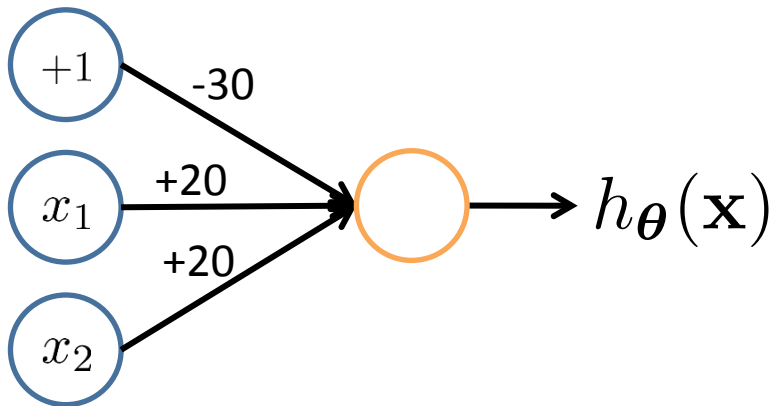


# Representing Boolean Functions

## Simple example: AND

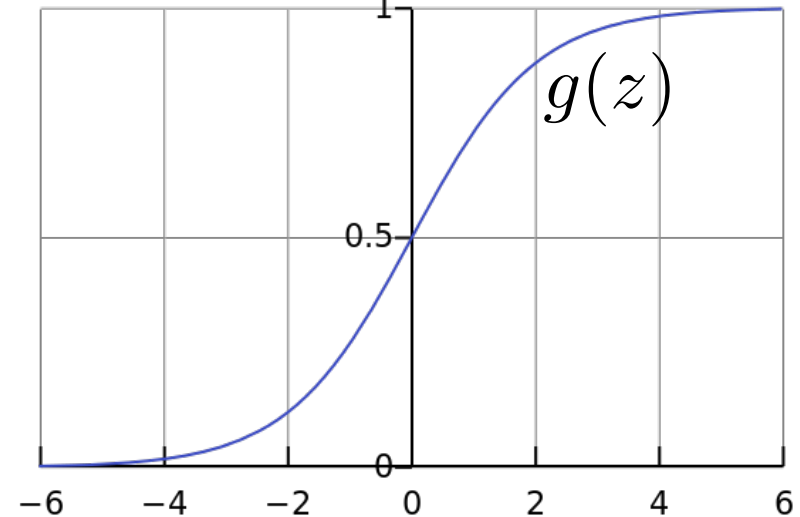
$$x_1, x_2 \in \{0, 1\}$$

$$y = x_1 \text{ AND } x_2$$



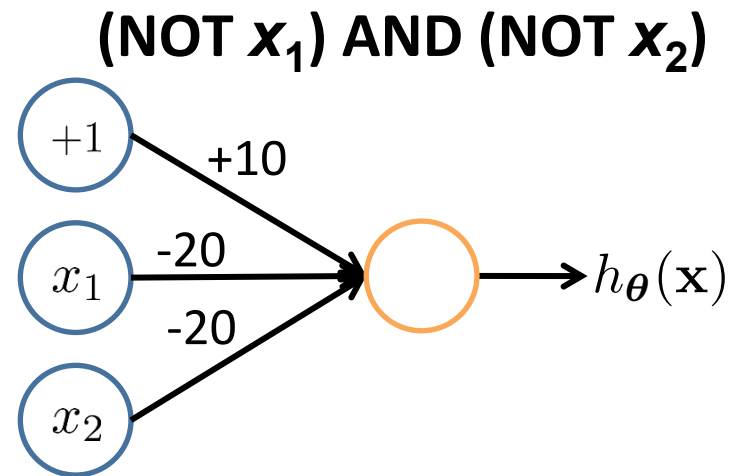
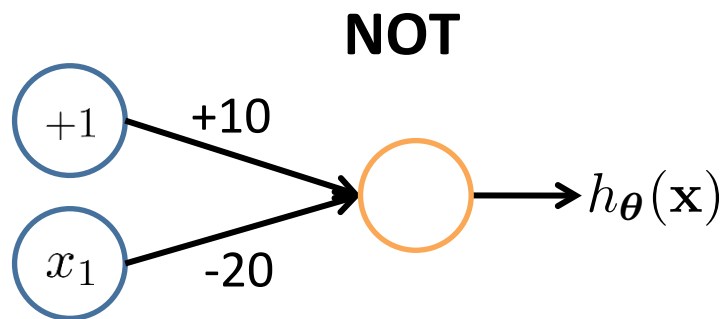
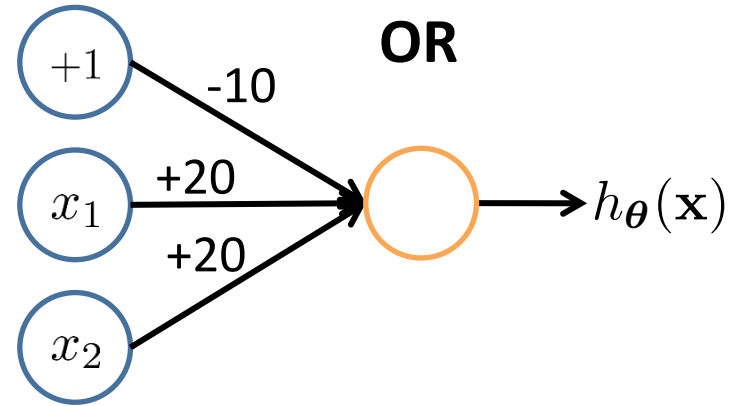
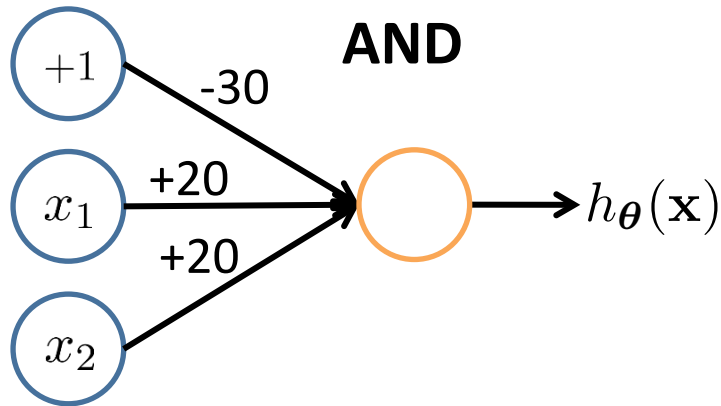
$$h_{\theta}(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

Logistic / Sigmoid Function

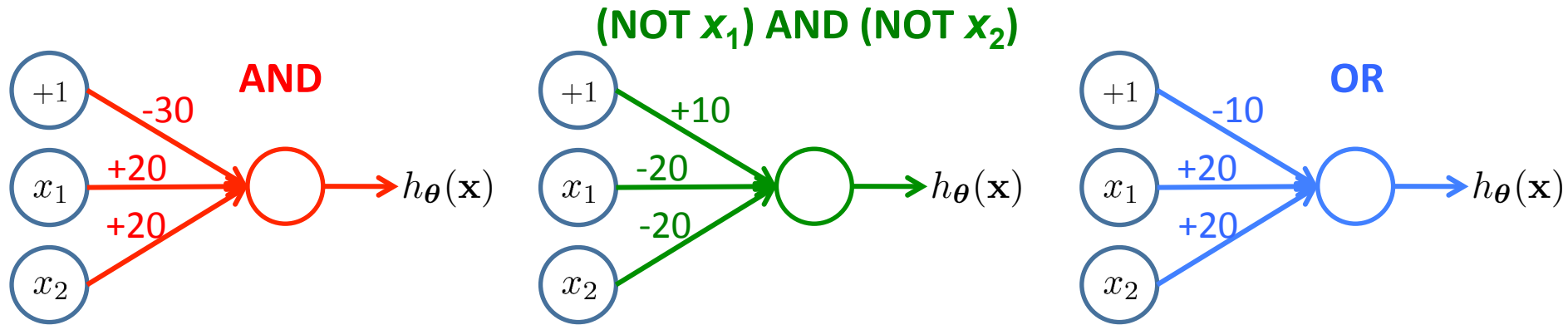


$x_1$	$x_2$	$h_{\theta}(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

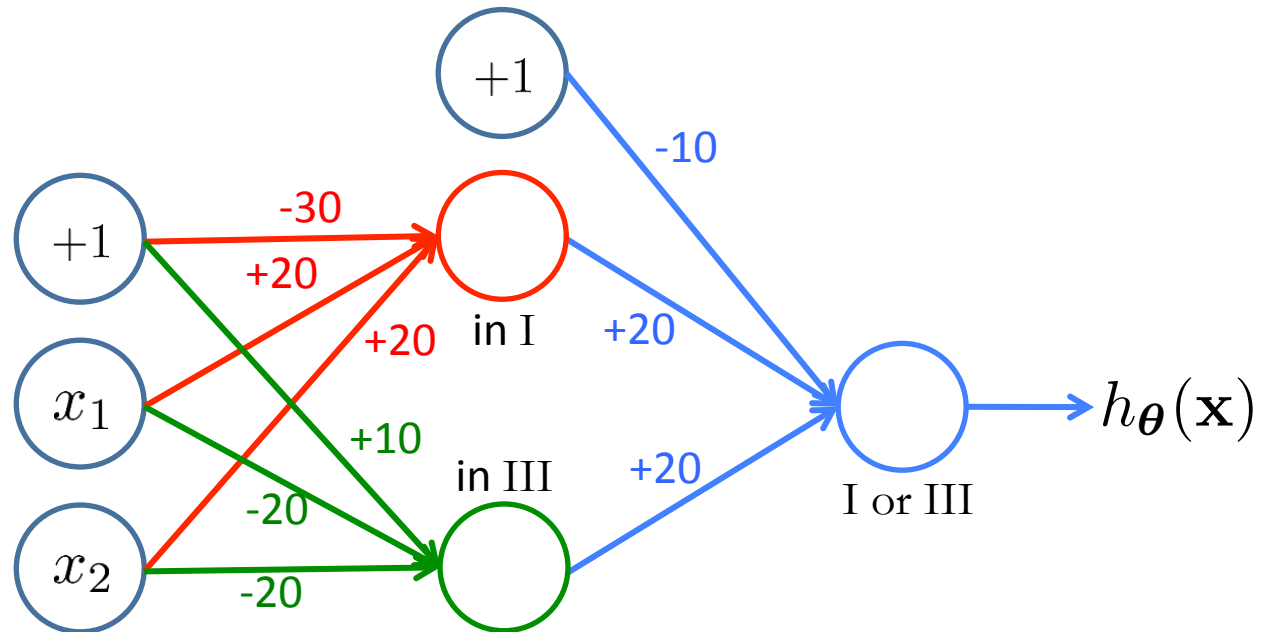
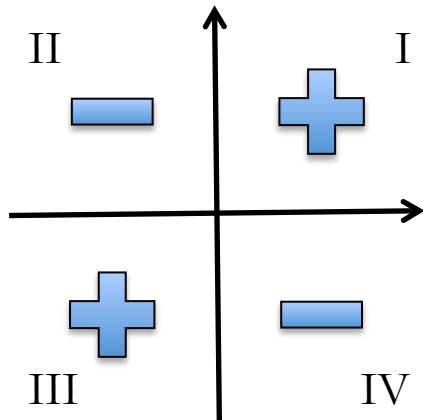
# Representing Boolean Functions



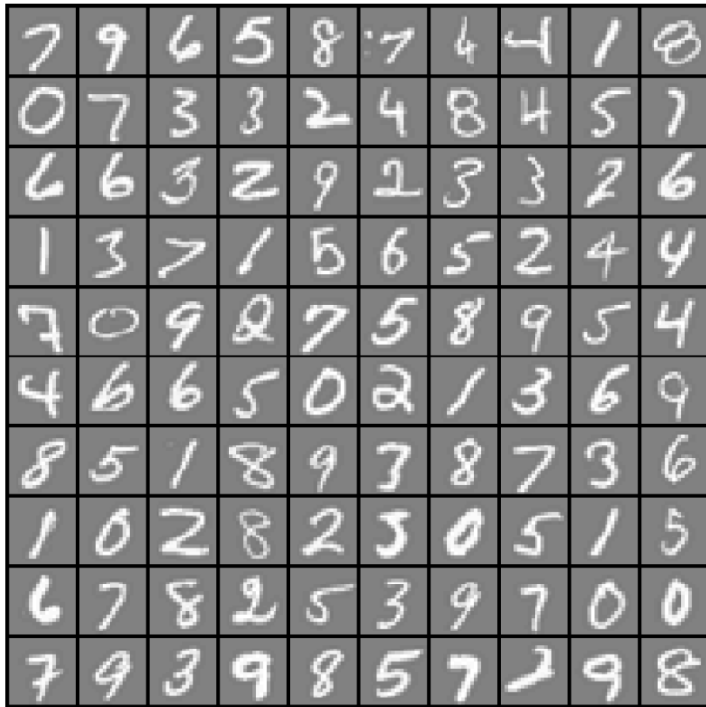
# Combining Representations to Create Non-Linear Functions



## NOT XOR



# Layering Representations



20 × 20 pixel images  
 $d = 400$  10 classes



$x_1 \dots x_{20}$   
 $x_{21} \dots x_{40}$   
 $x_{41} \dots x_{60}$

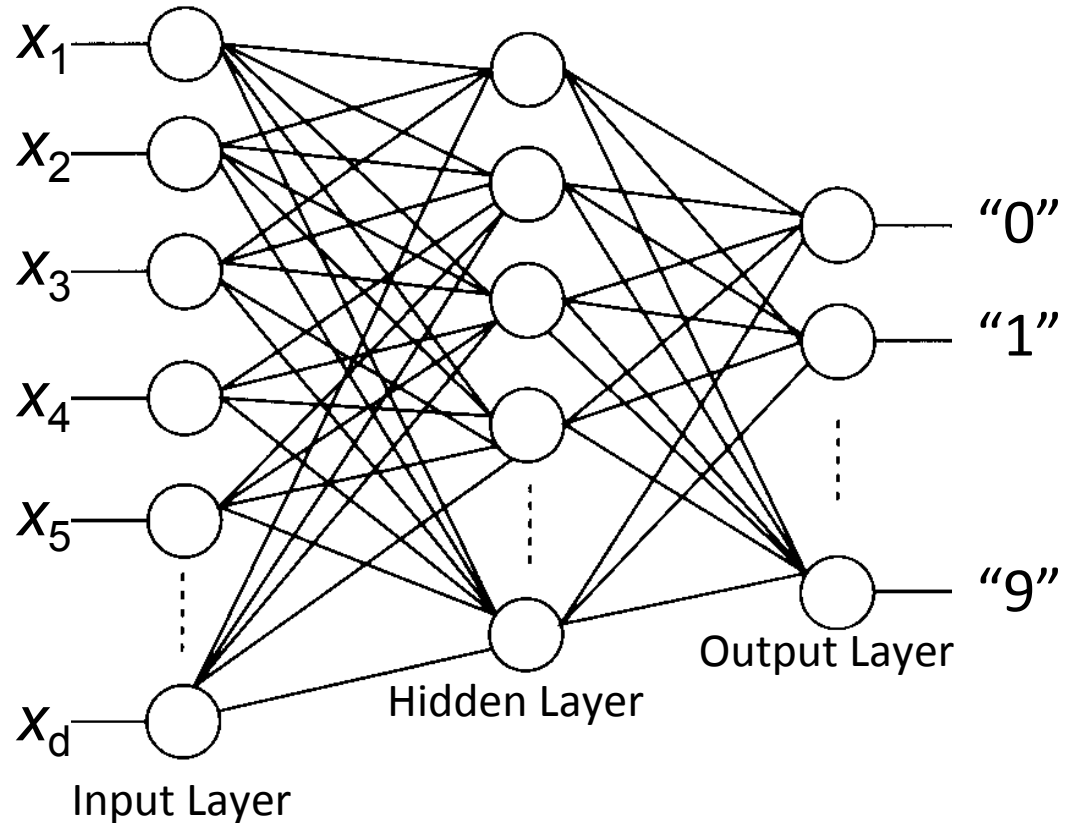


$x_{381} \dots x_{400}$

Each image is “unrolled” into a vector  $\mathbf{x}$  of pixel intensities

# Layering Representations

7	9	6	5	8	7	4	4	1	0
0	7	3	3	2	4	8	4	5	7
6	6	3	2	9	2	3	3	2	6
1	3	7	1	5	6	5	2	4	4
7	0	9	2	7	5	8	9	5	4
4	6	6	5	0	2	1	3	6	9
8	5	1	8	9	7	8	7	3	6
1	0	2	8	2	3	0	5	1	5
6	7	8	2	5	3	9	7	0	0
7	9	3	9	8	5	7	2	9	8



# Neural Network Learning

# Perceptron Learning Rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(y - h(\mathbf{x}))\mathbf{x}$$

Intuitive rule:

- If output is correct, don't change the weights
- If output is low ( $h(\mathbf{x}) = 0, y = 1$ ), increment weights for all the inputs which are 1
- If output is high ( $h(\mathbf{x}) = 1, y = 0$ ), decrement weights for all inputs which are 1

## Perceptron Convergence Theorem:

- If there is a set of weights that is consistent with the training data (i.e., the data is linearly separable), the perceptron learning algorithm will converge [Minsky & Papert, 1969]

# Batch Perceptron

Given training data  $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$

Let  $\boldsymbol{\theta} \leftarrow [0, 0, \dots, 0]$

Repeat:

Let  $\boldsymbol{\Delta} \leftarrow [0, 0, \dots, 0]$

for  $i = 1 \dots n$ , do

if  $y^{(i)} \mathbf{x}^{(i)} \boldsymbol{\theta} \leq 0$

// prediction for  $i^{th}$  instance is incorrect

$\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} + y^{(i)} \mathbf{x}^{(i)}$

$\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} / n$

// compute average update

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

Until  $\|\boldsymbol{\Delta}\|_2 < \epsilon$

- Simplest case:  $\alpha = 1$  and don't normalize, yields the fixed increment perceptron
- Each increment of outer loop is called an **epoch**



# Learning in NN: Backpropagation

- Similar to the perceptron learning algorithm, we cycle through our examples
  - If the output of the network is correct, no changes are made
  - If there is an error, weights are adjusted to reduce the error
- We are just performing (stochastic) gradient descent!

# Cost Function

Logistic Regression:

$$J(\theta) = -\frac{1}{n} \sum_{i=1}^n [y_i \log h_{\theta}(\mathbf{x}_i) + (1 - y_i) \log (1 - h_{\theta}(\mathbf{x}_i))] + \frac{\lambda}{2n} \sum_{j=1}^d \theta_j^2$$

Neural Network:

$$h_{\Theta} \in \mathbb{R}^K \quad (h_{\Theta}(\mathbf{x}))_i = i^{\text{th}} \text{output}$$

$$J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^n \sum_{k=1}^K y_{ik} \log (h_{\Theta}(\mathbf{x}_i))_k + (1 - y_{ik}) \log \left( 1 - (h_{\Theta}(\mathbf{x}_i))_k \right) \right] + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} \left( \Theta_{ji}^{(l)} \right)^2$$

$k^{\text{th}}$  class: true, predicted  
not  $k^{\text{th}}$  class: true, predicted

# Optimizing the Neural Network

$$J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^n \sum_{k=1}^K y_{ik} \log(h_{\Theta}(\mathbf{x}_i))_k + (1 - y_{ik}) \log(1 - (h_{\Theta}(\mathbf{x}_i))_k) \right] + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} \left( \Theta_{ji}^{(l)} \right)^2$$

Solve via:  $\min_{\Theta} J(\Theta)$

Unlike before,  $J(\Theta)$  is not convex, so GD on a neural net yields a local optimum

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \quad (\text{ignoring } \lambda; \text{ if } \lambda = 0)$$

Forward Propagation

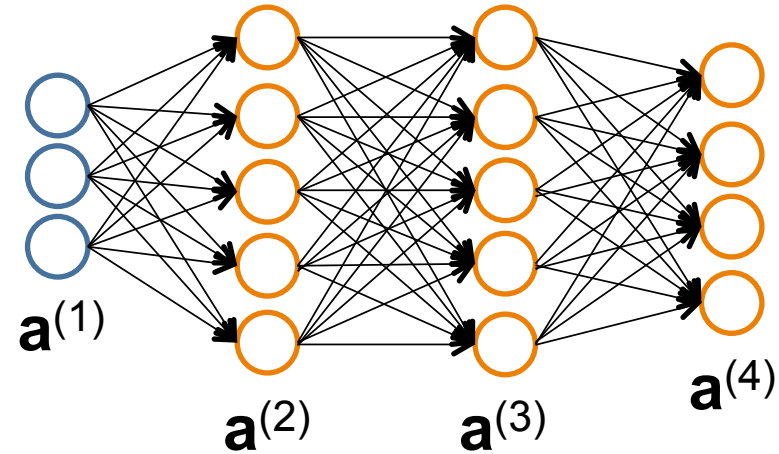
Backpropagation

# Forward Propagation

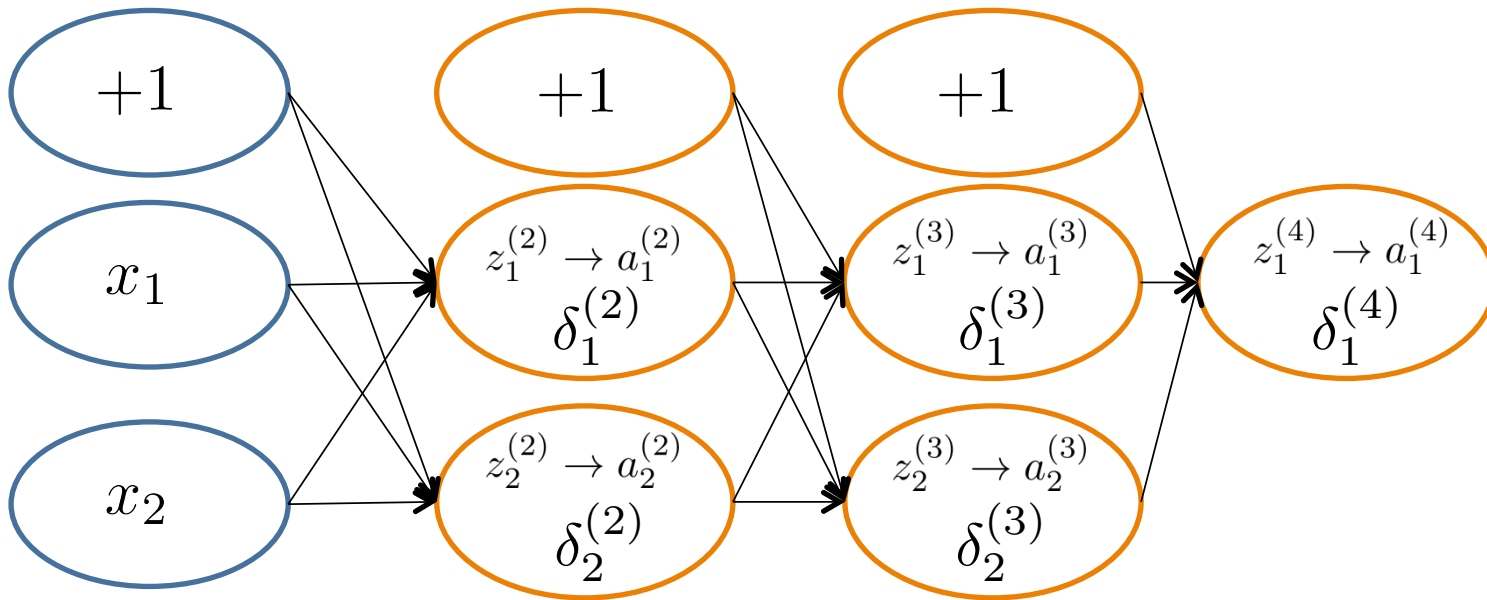
- Given one labeled training instance  $(\mathbf{x}, y)$ :

## Forward Propagation

- $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{a}^{(1)}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$  [add  $a_0^{(2)}$ ]
- $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
- $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$  [add  $a_0^{(3)}$ ]
- $\mathbf{z}^{(4)} = \Theta^{(3)}\mathbf{a}^{(3)}$
- $\mathbf{a}^{(4)} = h_{\Theta}(\mathbf{x}) = g(\mathbf{z}^{(4)})$



# Backpropagation Intuition

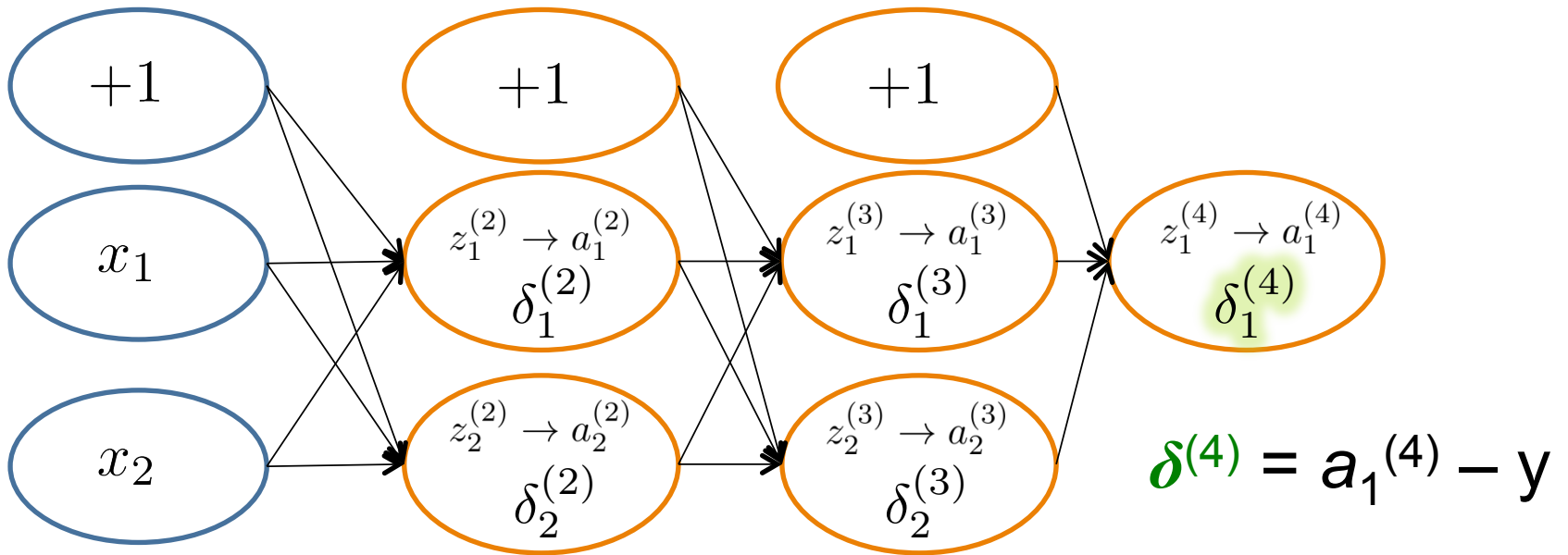


$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally, 
$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_{\Theta}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\Theta}(\mathbf{x}_i))$

# Backpropagation Intuition

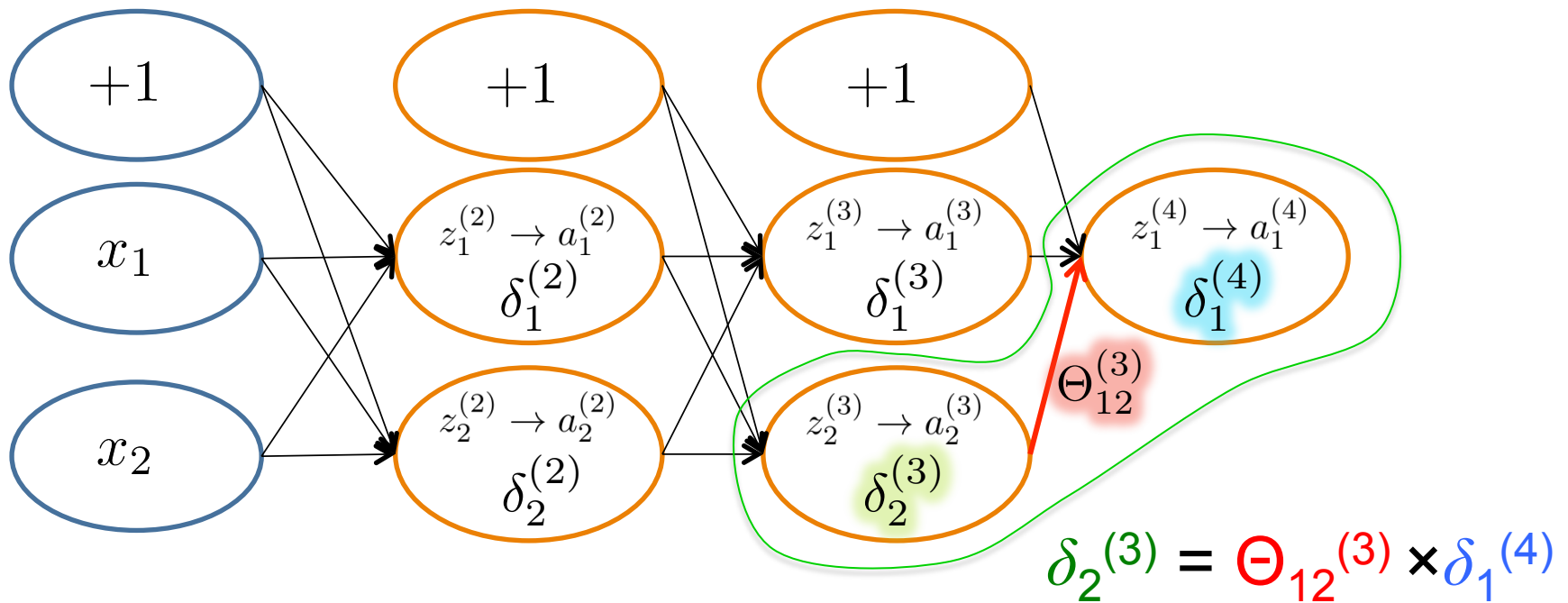


$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally, 
$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_{\Theta}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\Theta}(\mathbf{x}_i))$

# Backpropagation Intuition

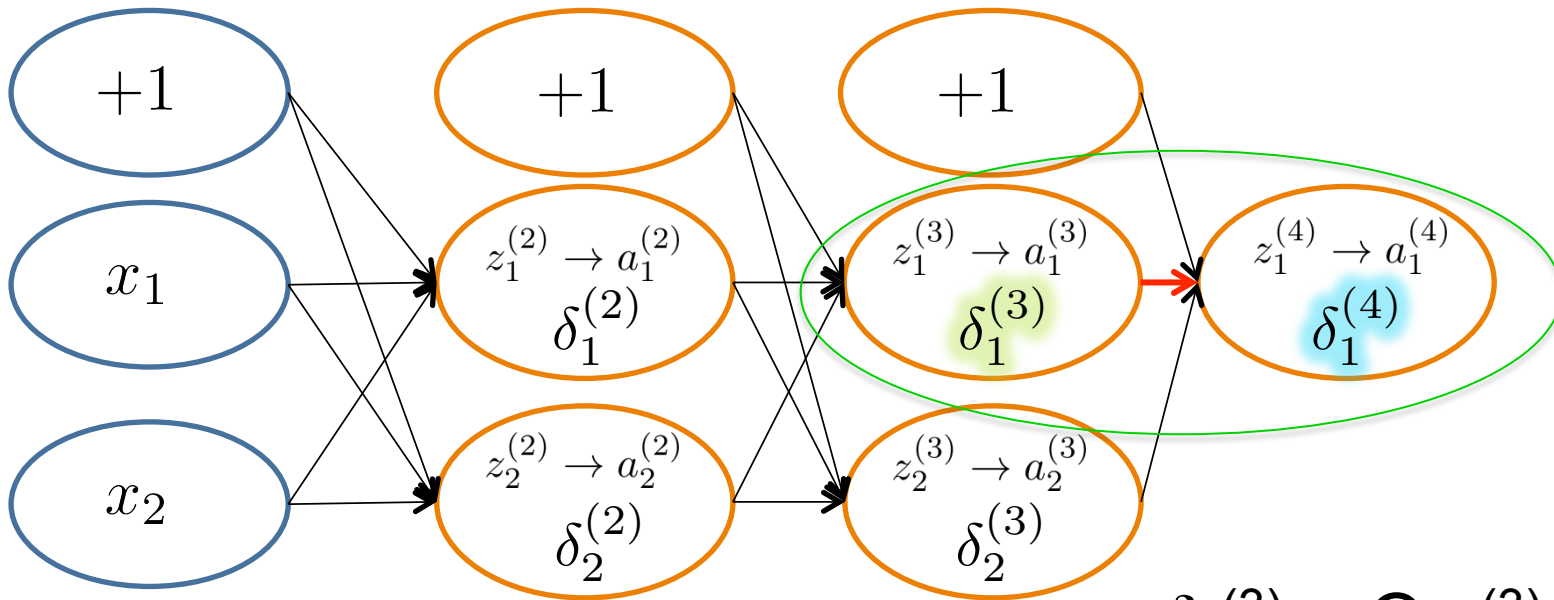


$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally, 
$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_{\Theta}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\Theta}(\mathbf{x}_i))$

# Backpropagation Intuition



$$\delta_2^{(3)} = \Theta_{12}^{(3)} \times \delta_1^{(4)}$$

$$\delta_1^{(3)} = \Theta_{11}^{(3)} \times \delta_1^{(4)}$$

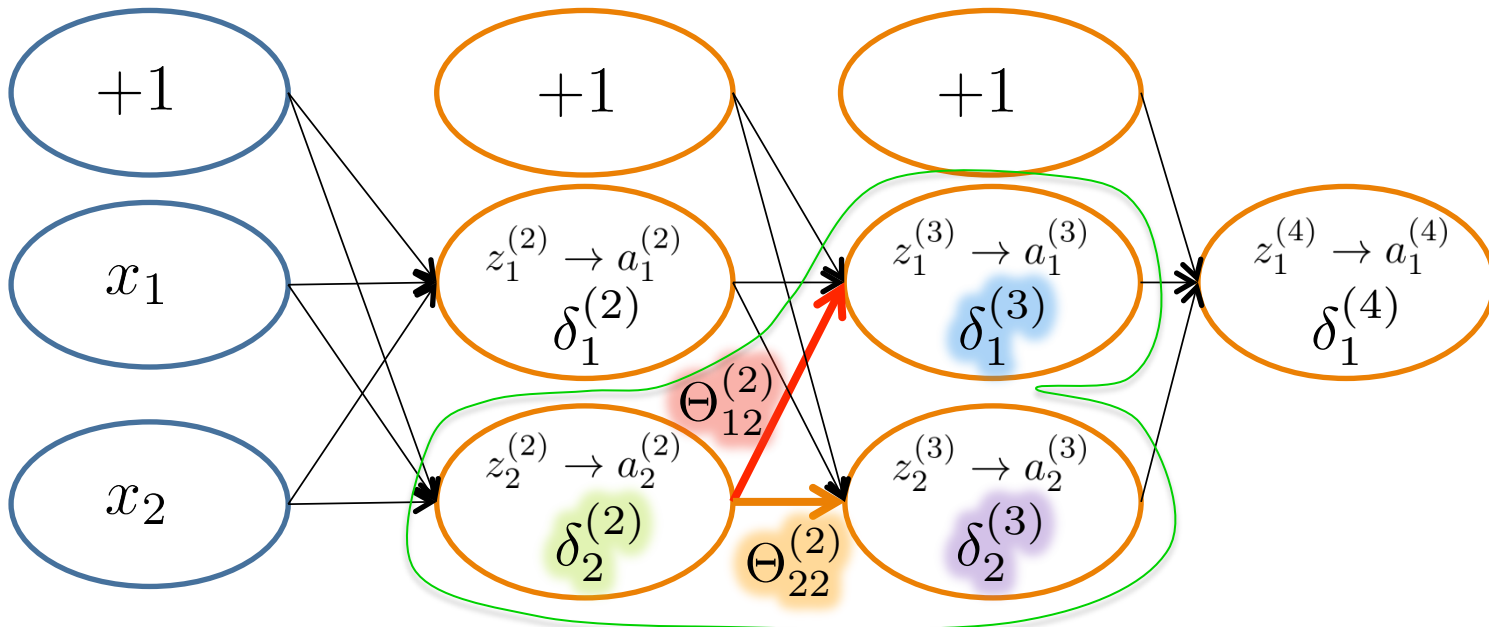
$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

Formally, 
$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_{\Theta}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\Theta}(\mathbf{x}_i))$



# Backpropagation Intuition



$$\delta_2^{(2)} = \Theta_{12}^{(2)} \times \delta_1^{(3)} + \Theta_{22}^{(2)} \times \delta_2^{(3)}$$

$\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

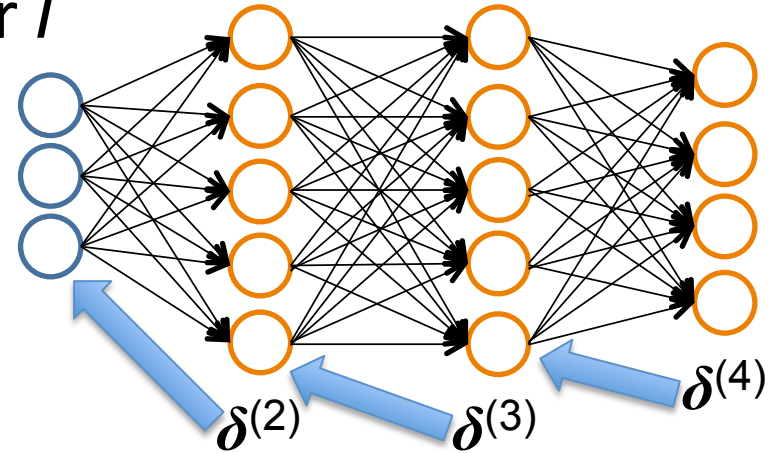
Formally, 
$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$$

where  $\text{cost}(\mathbf{x}_i) = y_i \log h_{\Theta}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\Theta}(\mathbf{x}_i))$

# Backpropagation: Gradient Computation

Let  $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$

(#layers  $L = 4$ )



## Backpropagation

- $\delta^{(4)} = \mathbf{a}^{(4)} - \mathbf{y}$
- $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(\mathbf{z}^{(3)})$
- $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(\mathbf{z}^{(2)})$
- (No  $\delta^{(1)}$ )

Element-wise product  $.*$

$$g'(\mathbf{z}^{(3)}) = \mathbf{a}^{(3)} .* (1 - \mathbf{a}^{(3)})$$

$$g'(\mathbf{z}^{(2)}) = \mathbf{a}^{(2)} .* (1 - \mathbf{a}^{(2)})$$

# Backpropagation

Set  $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$  (Used to accumulate gradient)

For each training instance  $(\mathbf{x}_i, y_i)$ :

Set  $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute  $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$  via forward propagation

Compute  $\delta^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors  $\{\delta^{(L-1)}, \dots, \delta^{(2)}\}$

Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient  $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

$\mathbf{D}^{(l)}$  is the matrix of partial derivatives of  $J(\Theta)$

# Training a Neural Network via Gradient Descent with Backprop

Given: training set  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$

Initialize all  $\Theta^{(l)}$  randomly (NOT to 0!)

Loop // each iteration is called an epoch

Set  $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$  (Used to accumulate gradient)

For each training instance  $(\mathbf{x}_i, y_i)$ :

Set  $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute  $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$  via forward propagation

Compute  $\delta^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors  $\{\delta^{(L-1)}, \dots, \delta^{(2)}\}$

Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient  $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

Update weights via gradient step  $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$

Until weights converge or max #epochs is reached

Backpropagation

# Several Practical Tricks

## Initialization

- Problem is highly non-convex, and heuristics exist to start training (at the least, randomize initial weights)

## Optimization tricks

- Momentum-based methods
- Decaying step size
- Dropout to avoid co-adaptation / overfitting

## Minibatch

- Use more than a single point to estimate gradient

...

# Neural Networks vs Deep Learning?

DNN are big neural networks

- Depth: often ~5 layers (but some have 20+)
  - Typically not fully connected!
- Width: hidden nodes per layer in the thousands
- Parameters: millions to billions

Algorithms / Computing

- New algorithms (pre-training, layer-wise training, dropout, etc.)
- Heavy computing requirements (GPUs are essential)