Machine Learning

CS 539

Worcester Polytechnic Institute
Department of Computer Science
Instructor: Prof. Kyumin Lee

Computational Learning Theory: Why ML Work

A Way to Choose the Best Model

- It would be <u>really</u> helpful if we could get a guarantee of the following form:

$$\text{testingError} \leq \text{trainingError} + f(n, h, p)$$

n = size of training set

h = measure of the model complexity

p = the probability that this bound fails

We need p to allow for really unlucky test sets

- Then, we could choose the model complexity that minimizes the bound on the test error

A Measure of Model Complexity

- Suppose that we pick n data points and assign labels of + or – to them at random
- If our model class (e.g., a decision tree, polynomial regression of a particular degree, etc.) can learn any association of labels with data, it is too powerful!

> More power: can model more complex functions, but may overfit
>
> Less power: won't overfit, but limited in what it can represent

- **Idea**: characterize the power of a model class by asking how many data points it can learn perfectly for all possible assignments of labels
  - This number of data points is called the Vapnik-Chervonenkis (VC) dimension

VC Dimension

- A measure of the power of a particular class of models
  - It does not depend on the choice of training set
- The VC dimension of a model class is the maximum number of points that can be arranged so that the class of models can shatter

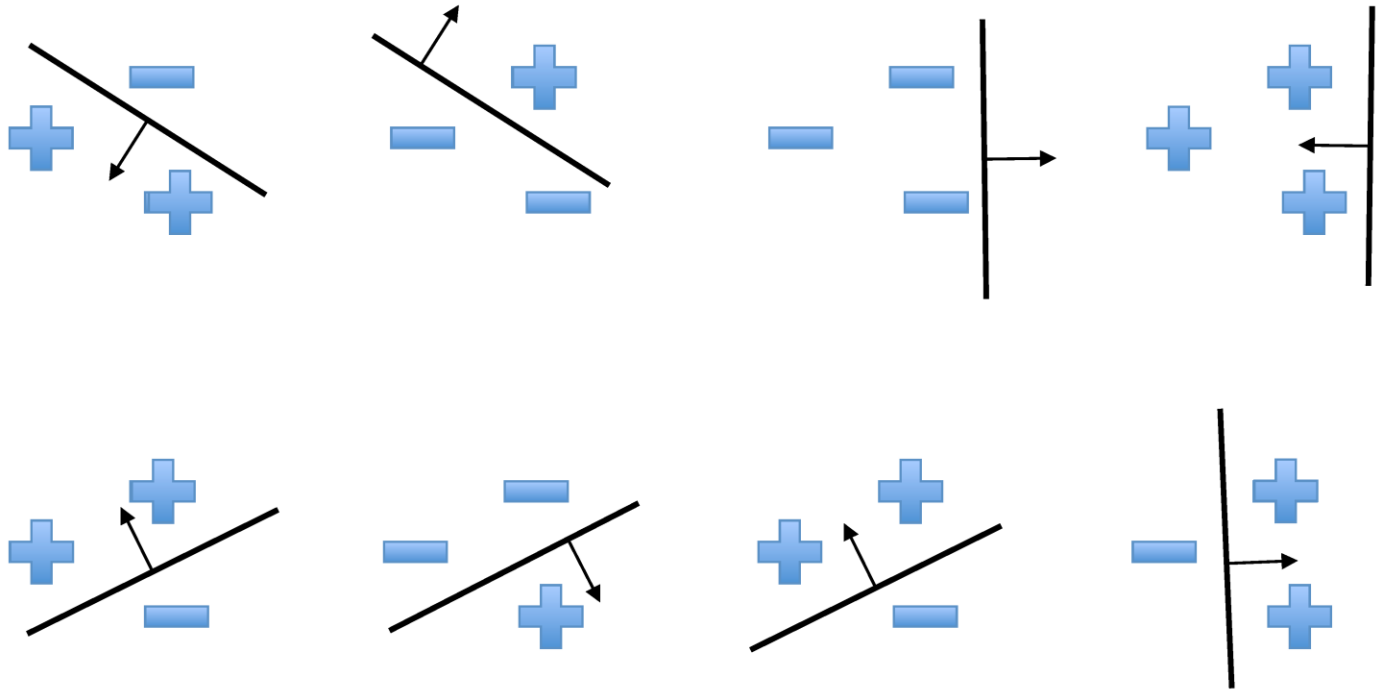
Definition: a model class can shatter a set of points

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(r)}$$

if for every possible labeling over those points, there exists a model in that class that obtains zero training error

An Example of VC Dimension

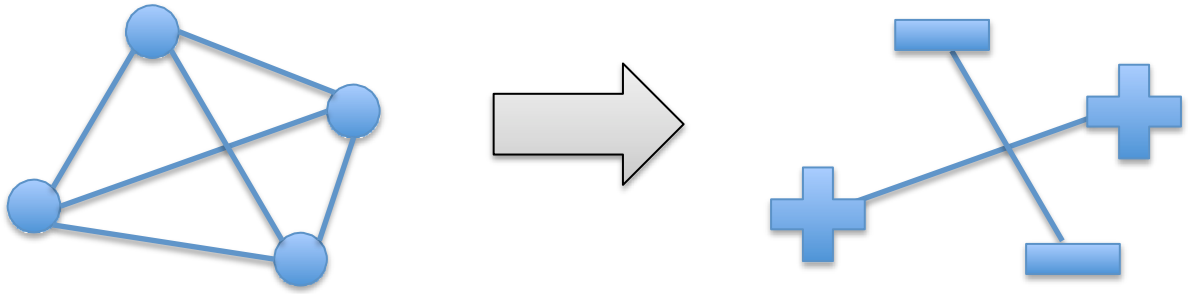
- Suppose our model class is a hyperplane (e.g., perceptron)
- Consider all labelings over three points in $\mathbb{R}^2$



- In $\mathbb{R}^2$, we can find a plane (i.e., a line) to capture any labeling of 3 points. A 2D hyperplane shatters 3 points

An Example of VC Dimension

- But, a 2D hyperplane cannot deal with some labelings of four points:



Connect all pairs of points; two lines will always cross

Can't separate points if the pairs that cross are the same class

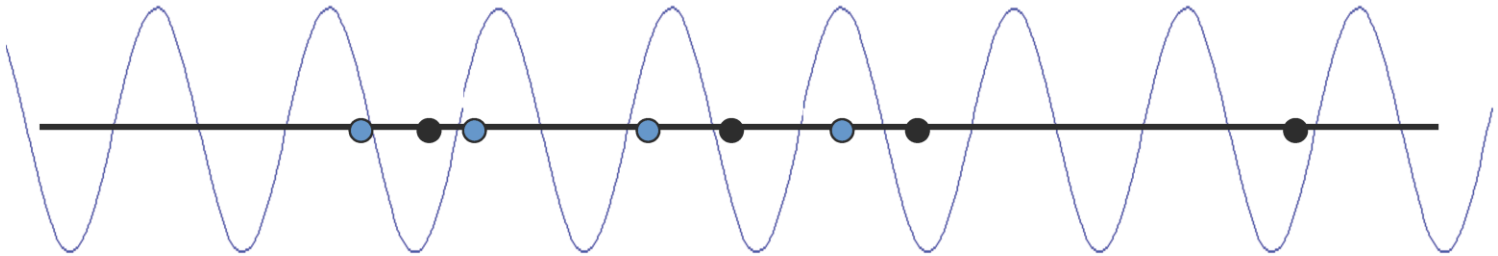
- Therefore, a 2D hyperplane cannot shatter 4 points

Some Examples of VC Dimension

- The VC dimension of a hyperplane in 2D is 3.
  - In d dimensions it is d+1
    - It's just a coincidence that the VC dimension of a hyperplane is almost identical to the # parameters needed to define a hyperplane

- A sine wave has infinite VC dimension and only 2 parameters!
  - By choosing the phase & period carefully we can shatter any random set of 1D data points (except for nasty special cases)

$$h(x) = a\sin(bx)$$



Assumptions

- Given some model class (which defines the hypothesis space H)
- Assume all training points were drawn i.i.d from distribution $\mathcal{D}$
- Assume all future test points will be drawn from $\mathcal{D}$

Definitions:

$$R(\boldsymbol{\theta}) = \text{testError}(\boldsymbol{\theta}) = E\left[\frac{1}{2}|y - h_{\boldsymbol{\theta}}(\boldsymbol{x})|\right]$$

"official" notation

notation we'll use

probability of misclassification

$$R^{\text{emp}}(\boldsymbol{\theta}) = \text{trainError}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left|y^{(i)} - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right|$$

A Probabilistic Guarantee of Generalization Performance

Vapnik showed that with probability $(1 - \eta)$:

$$\text{testError}(\boldsymbol{\theta}) \leq \text{trainError}(\boldsymbol{\theta}) + \sqrt{\frac{h(\log(2n/h) + 1) - \log(\eta/4)}{n}}$$

n = size of training set

h = VC dimension of model class

η = the probability that this bound fails

- So, we should pick the model with the complexity that minimizes this bound
  - The theory provides insight, but in practice we still need some magic

Take Away Lesson

Suppose we find a model with a low training error...

- If the following holds:
  - H is sufficiently constrained in size
  - and/or the size of the training data set n is large,

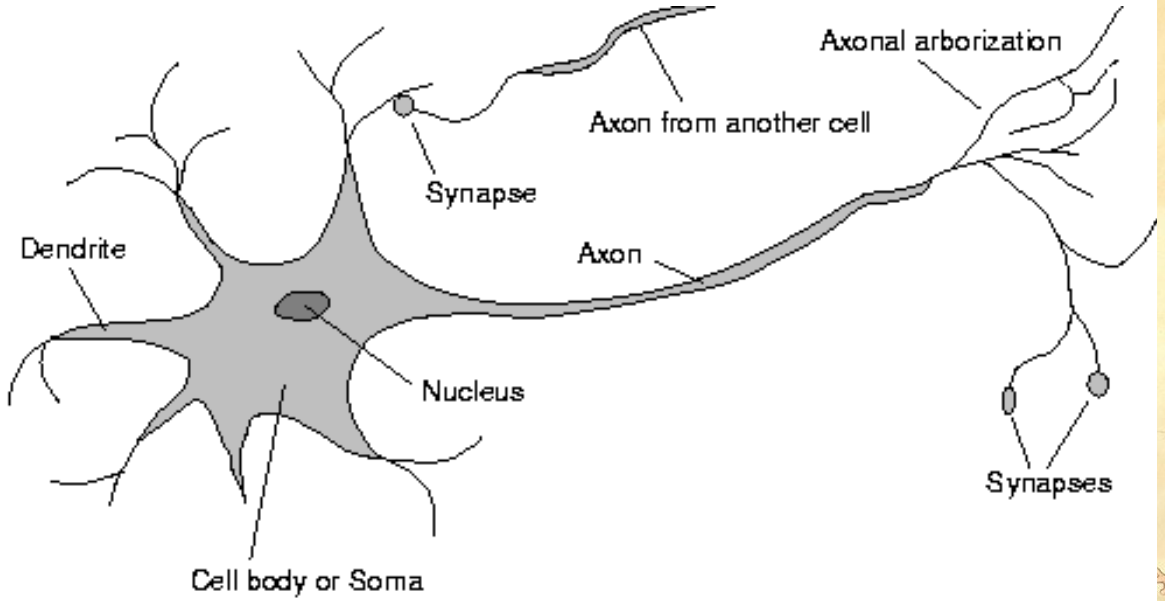
  then low training error is likely to be evidence of low generalization error

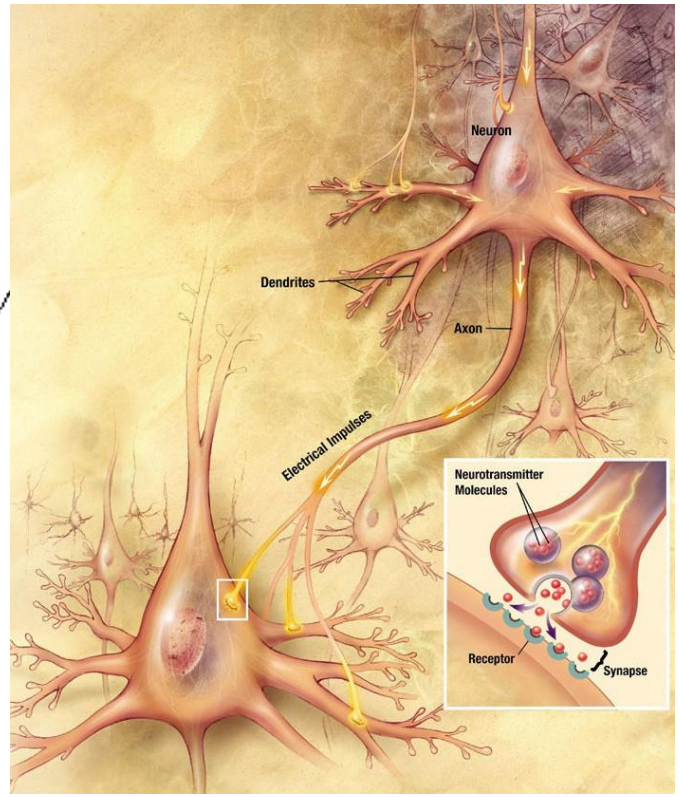
Neural Networks

Neural Function

- Brain function (thought) occurs as the result of the firing of **neurons**
- Neurons connect to each other through **synapses**, which propagate **action potential** (electrical impulses) by releasing **neurotransmitters**
  - Synapses can be **excitatory** (potential-increasing) or **inhibitory** (potential-decreasing), and have varying **activation thresholds**
  - Learning occurs as a result of the synapses' **plasticicity**: They exhibit long-term changes in connection strength
- There are about 10^{11} neurons and about 10^{14} synapses in the human brain!

Biology of a Neuron

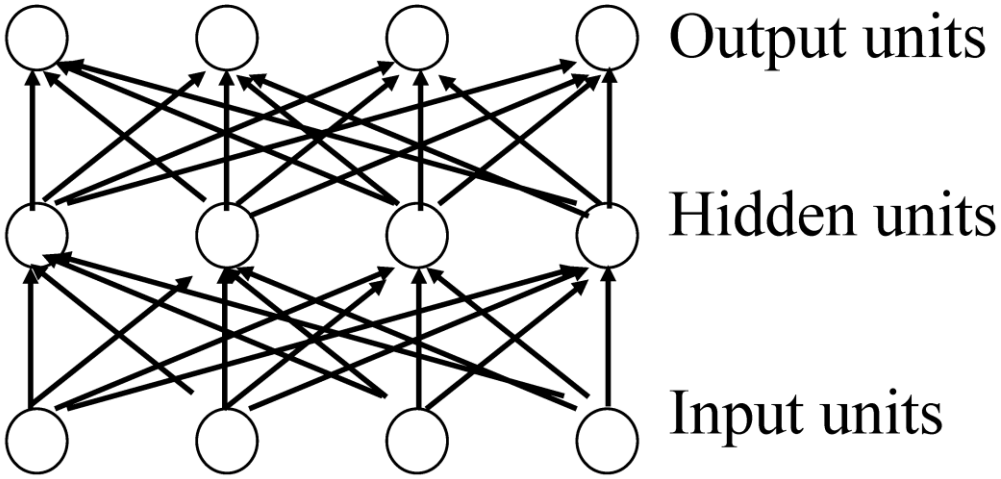




Neural Networks

- Origins: Algorithms that try to mimic the brain.
- 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

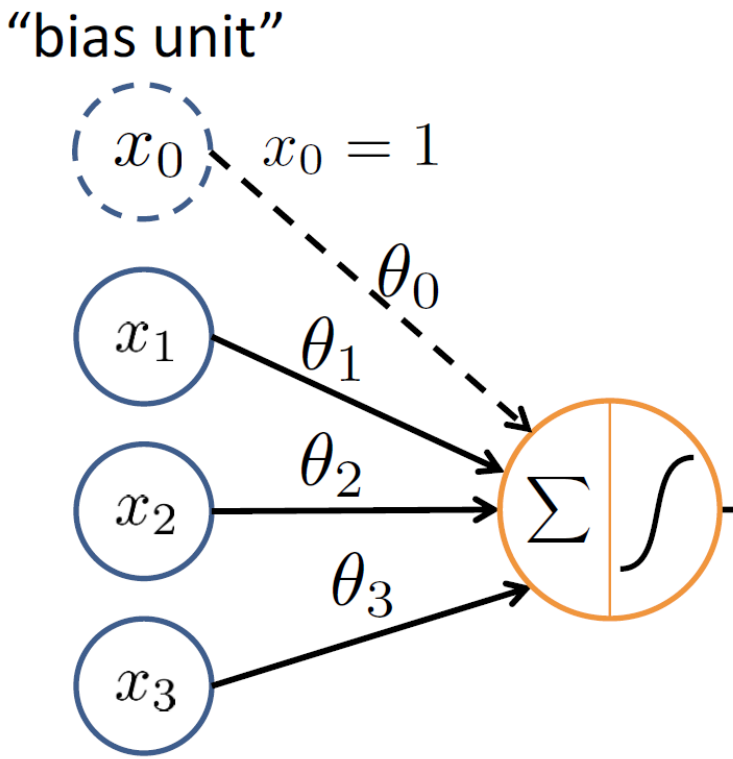
Neural Networks



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**

Neuron Model: Logistic Unit

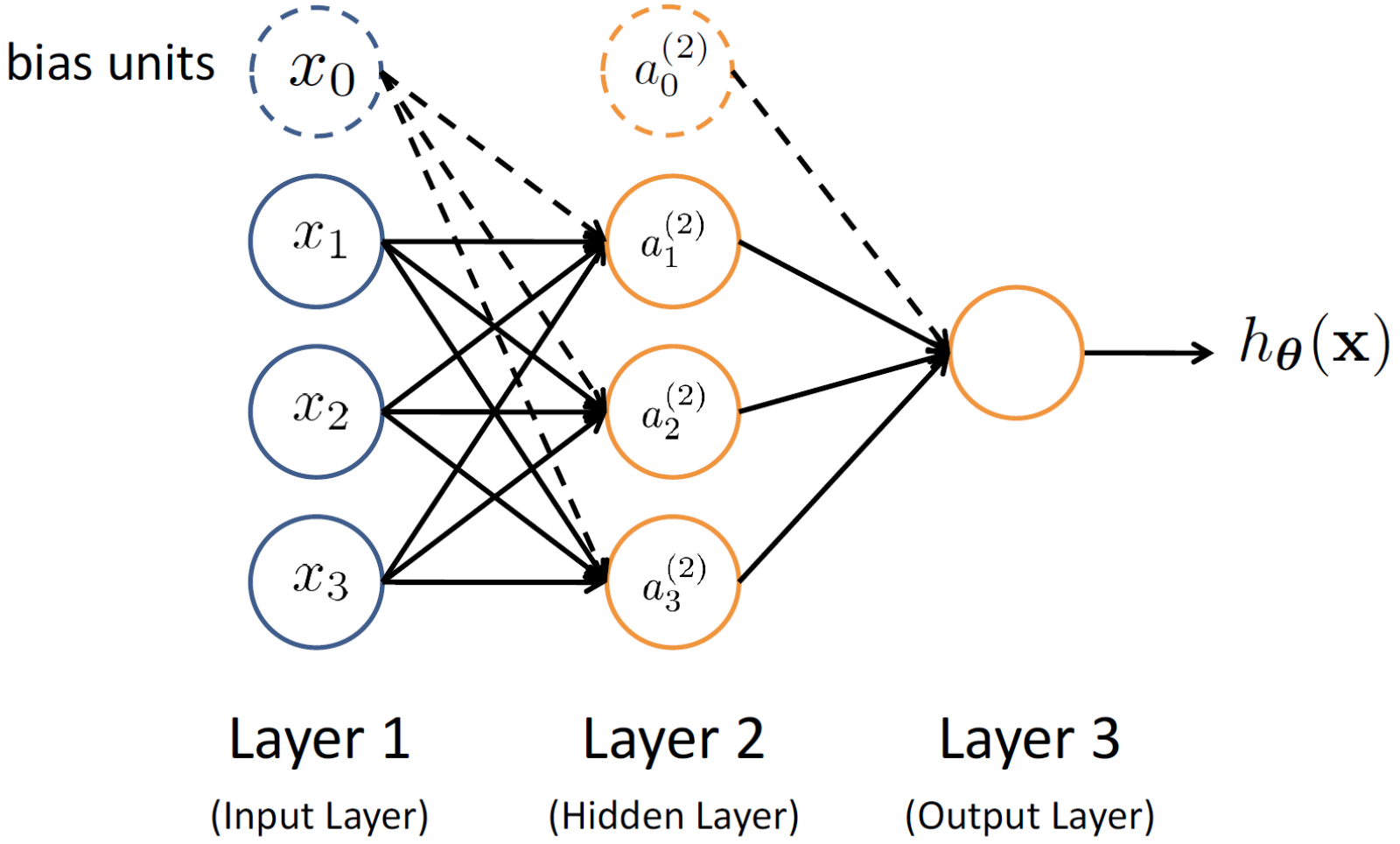


$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}\right)$$

$$= \frac{1}{1 + e^{-\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}}}$$

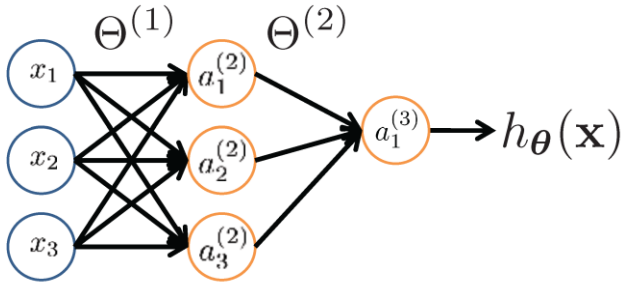
Neural Network



Feed-Forward Process

- Input layer units are set by some exterior function (think of these as **sensors**), 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

Neural Network



$a_i^{(j)} =$ "activation" of unit i in layer j

$\Theta^{(j)} =$ weight matrix controlling function mapping 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 $s_{j+1} \times (s_j+1)$

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

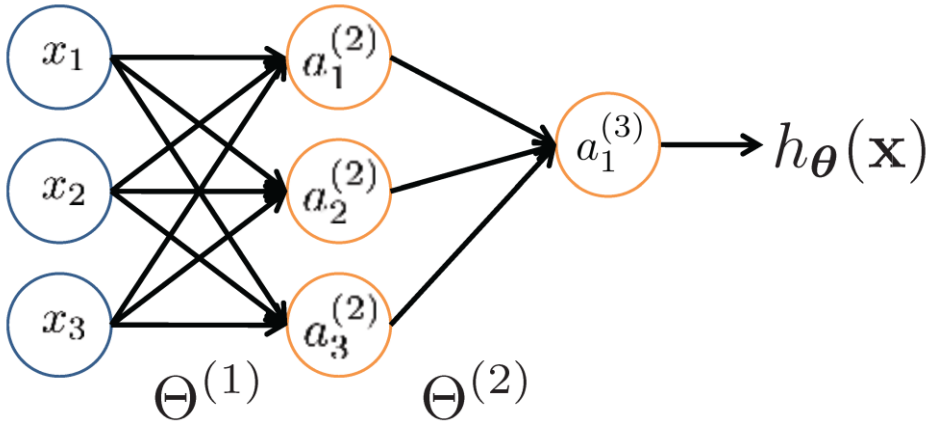
Vectorization

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

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

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

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



Feed-Forward Steps:

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

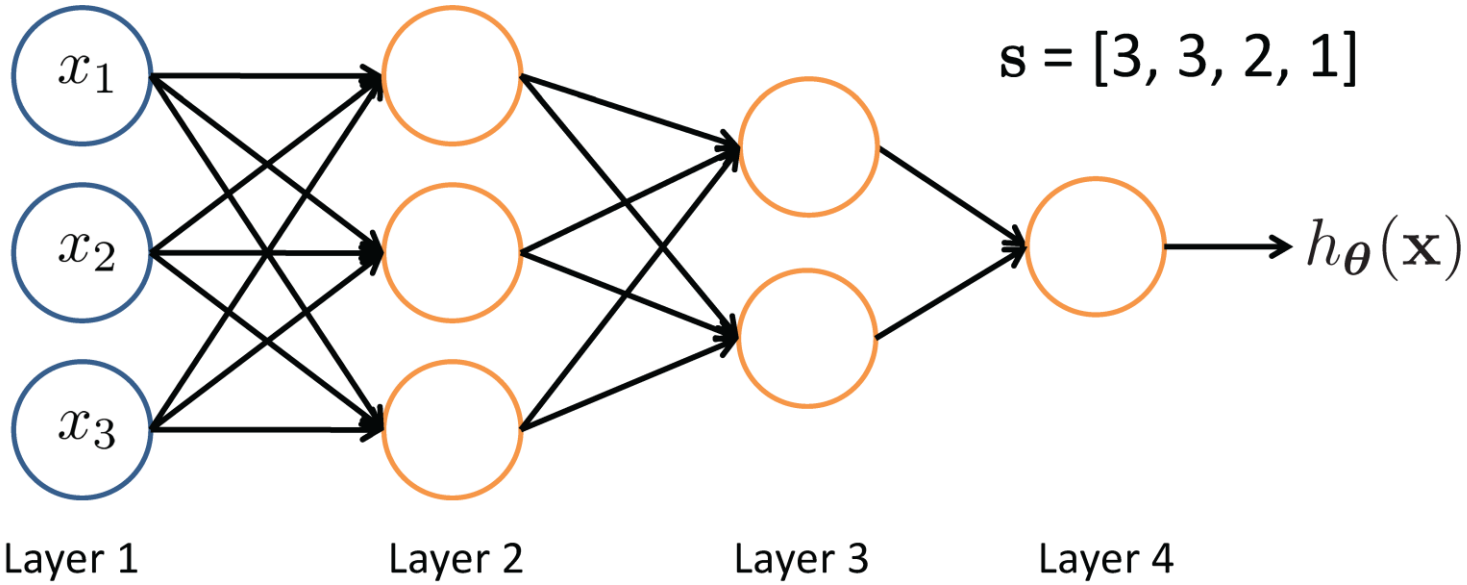
$$\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$$

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

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

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

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)

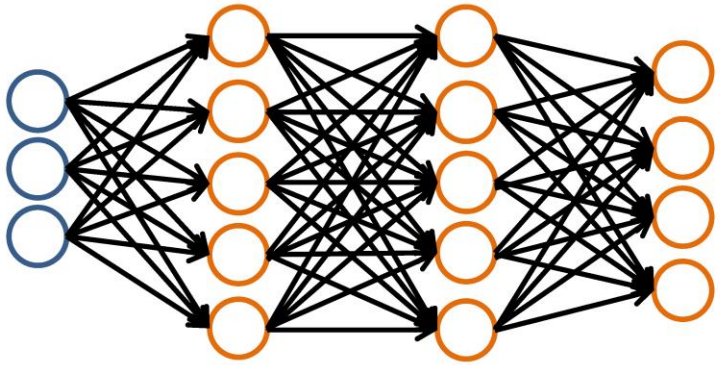
Multiple Output Units: One-vs-Rest

Pedestrian

Car

Motorcycle

Truck



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

We want:

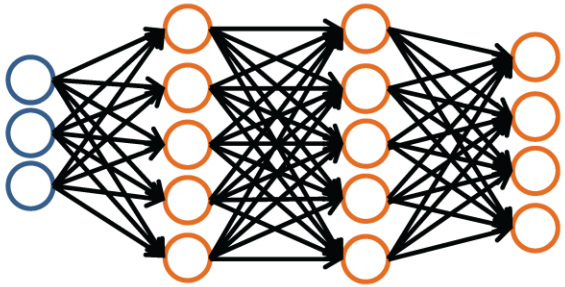
$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$ when pedestrian

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

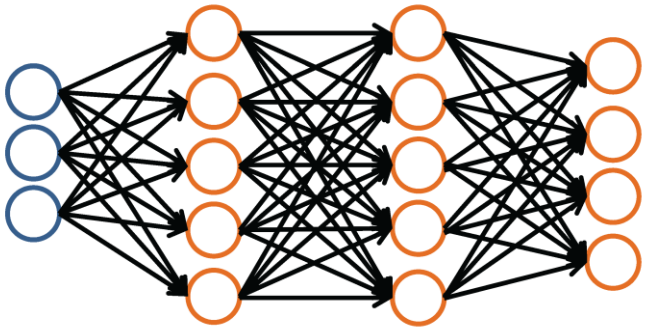
$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle

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

---

- Given $\{(\mathbf{x}_1, y_1),\ (\mathbf{x}_2, y_2),\ \ldots,\ (\mathbf{x}_n, y_n)\}$
- Must convert labels to 1-of-K representation
  - e.g., $\mathbf{y}_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle, $\mathbf{y}_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car, etc.

Neural Network Classification



Given:

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

$\mathbf{s} \in \mathbb{N}^{+L}$ 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 $(s_{L-1} = K)$

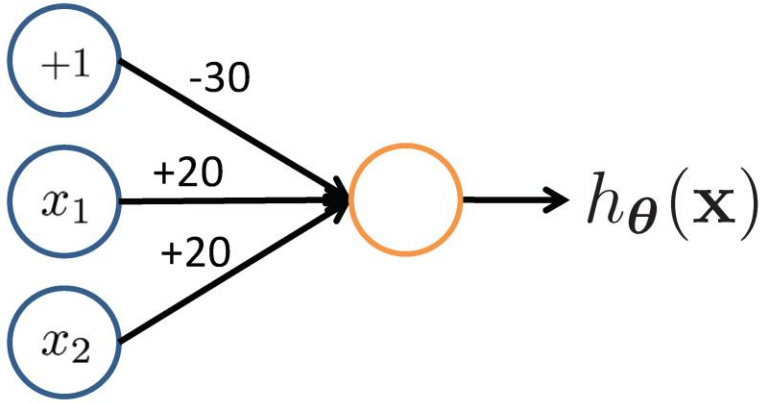
Understanding Representations

Representing Boolean Functions

Simple example: AND

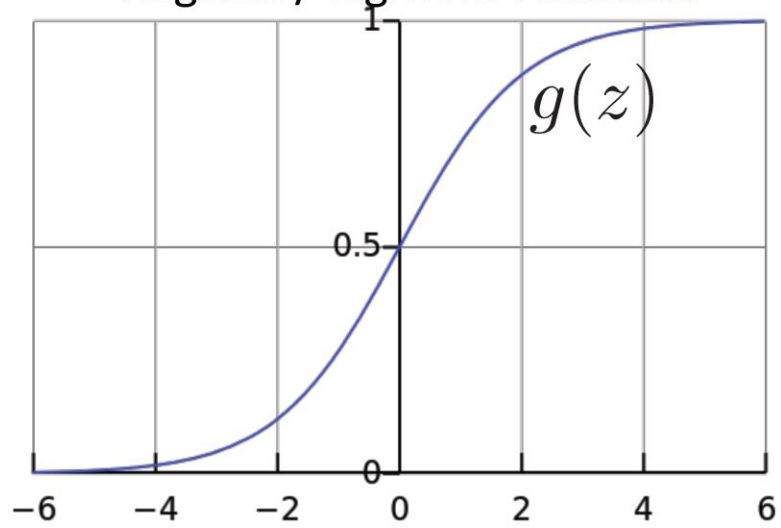
$x_1, x_2 \in \{0, 1\}$

$y = x_1 \text{ AND } x_2$



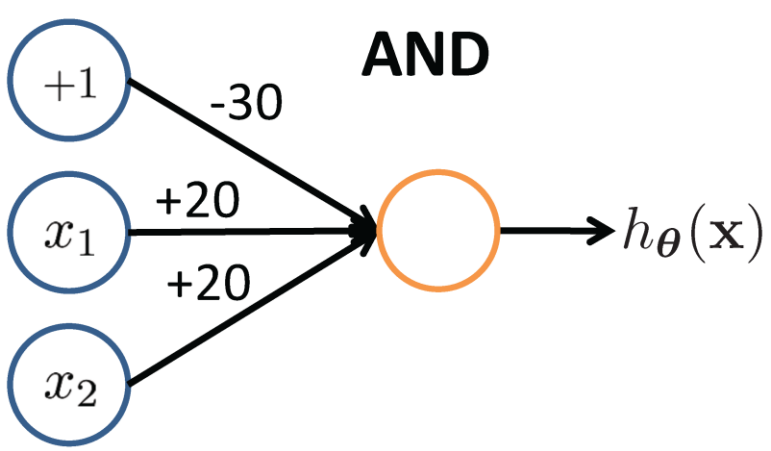
$$\mathrm{h}_\Theta(\mathbf{x}) = g(\text{-}30 + 20x_1 + 20x_2)$$

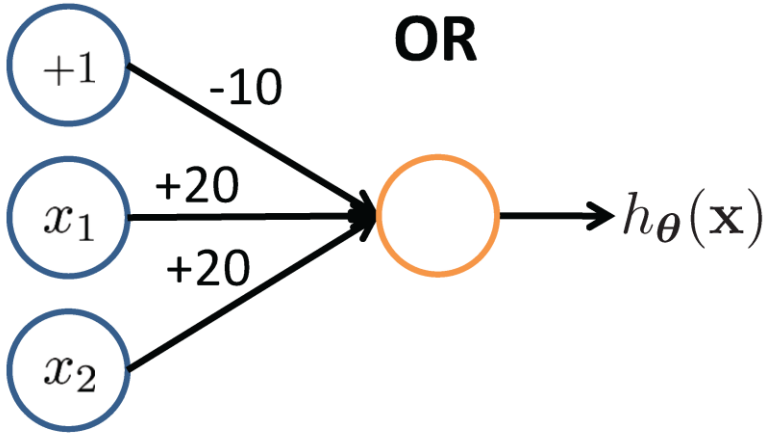
Logistic / Sigmoid Function

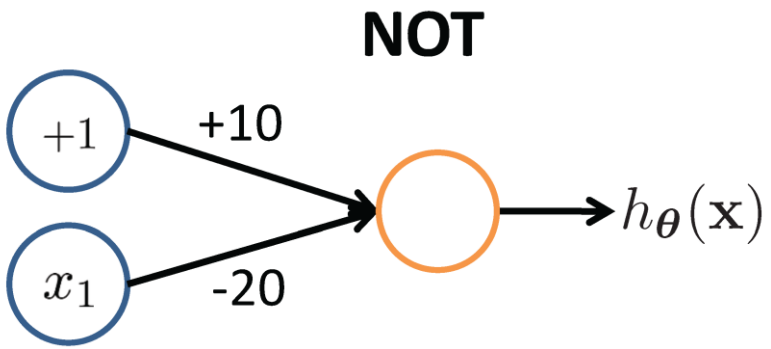


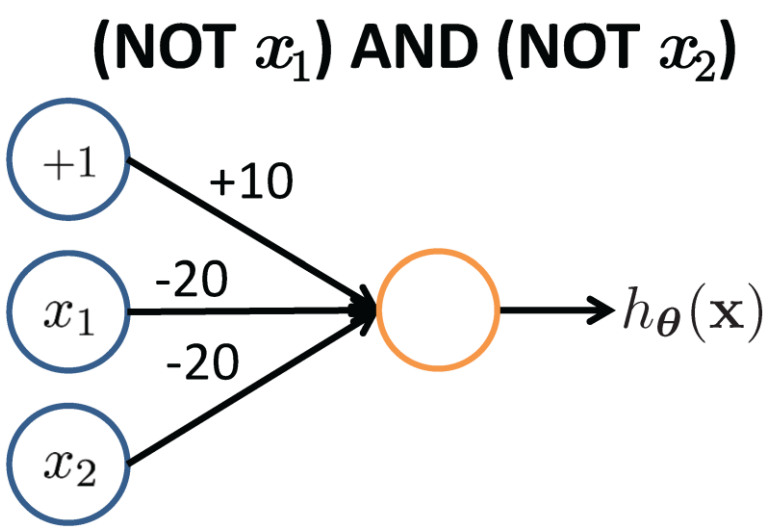
x_1	x_2	$\mathrm{h}_\Theta(\mathbf{x})$
0	0	$g(\text{-}30) \approx 0$
0	1	$g(\text{-}10) \approx 0$
1	0	$g(\text{-}10) \approx 0$
1	1	$g(10) \approx 1$

Representing Boolean Functions

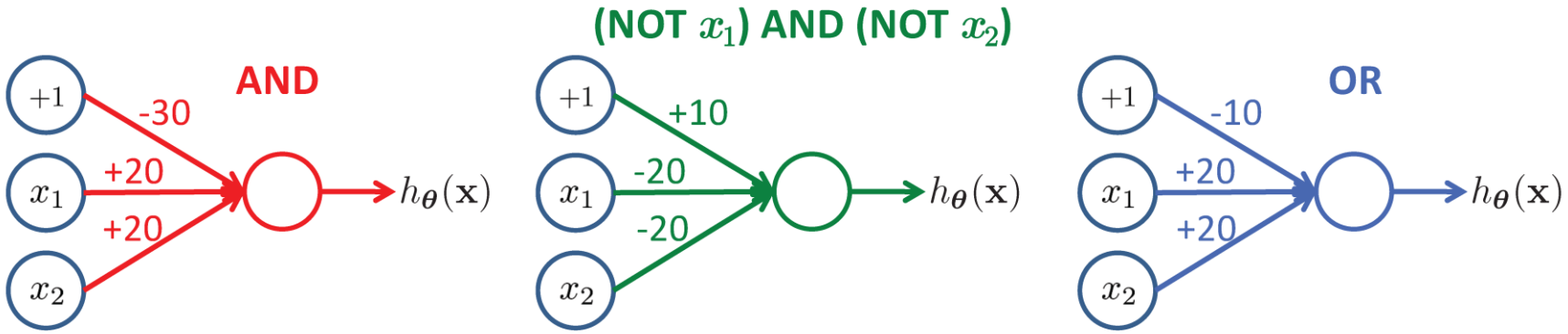


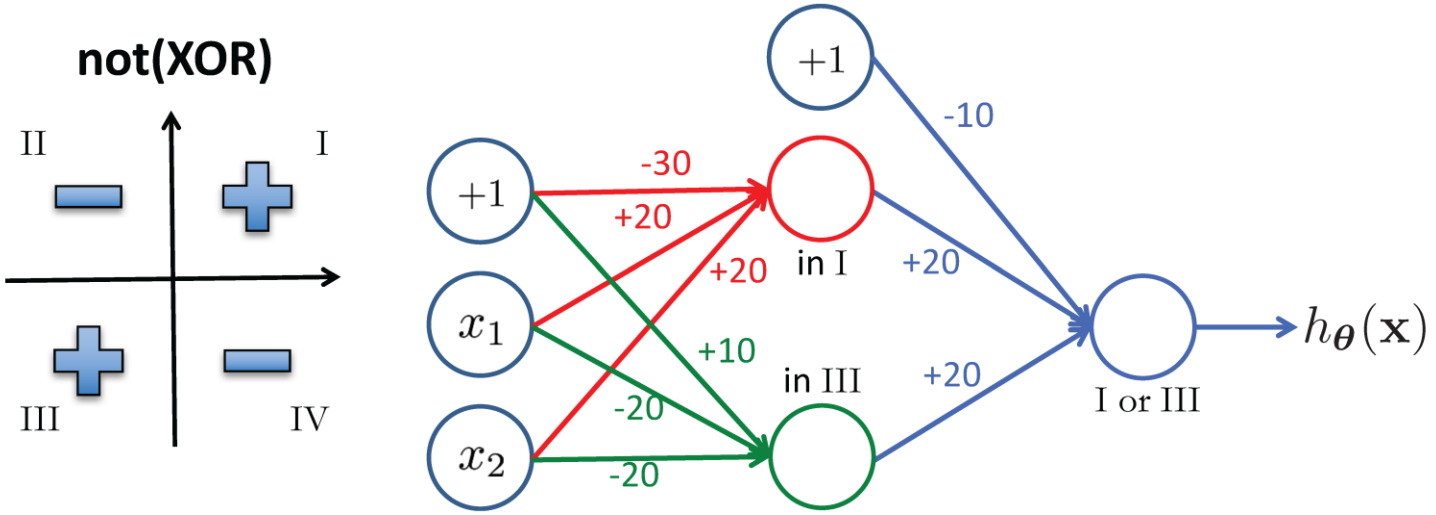




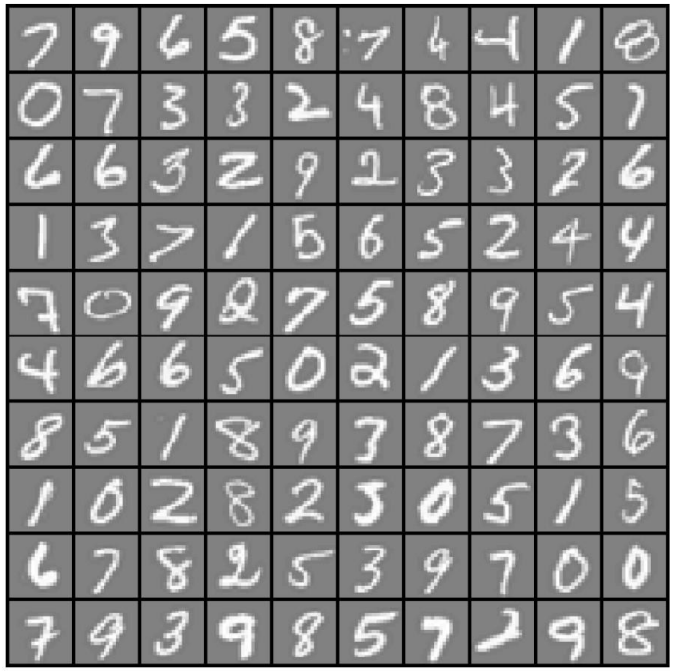


Combining Representations to Create Non-Linear Functions

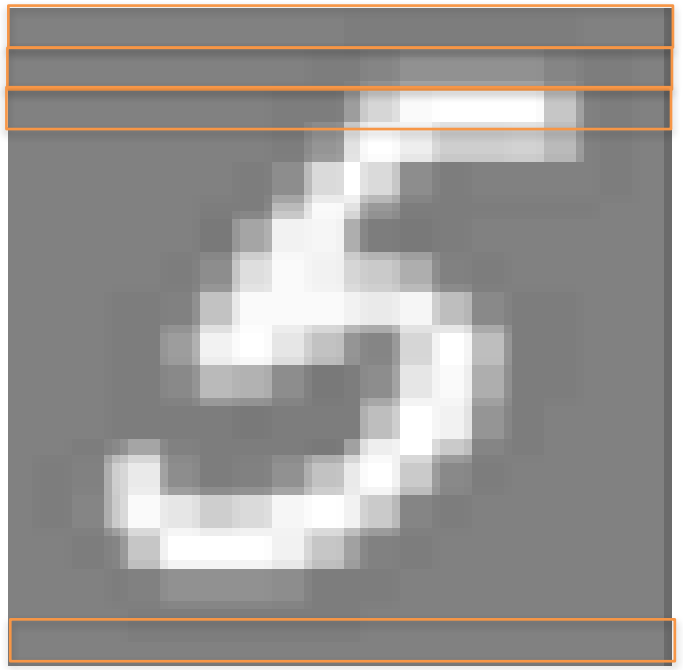




Layering Representations

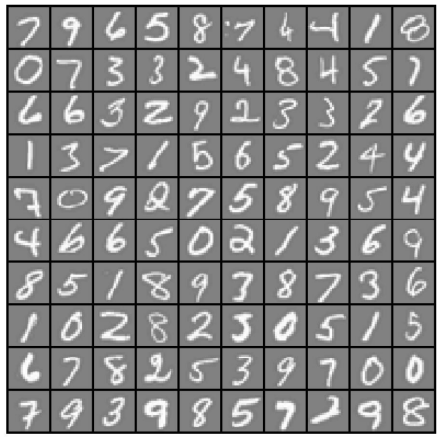


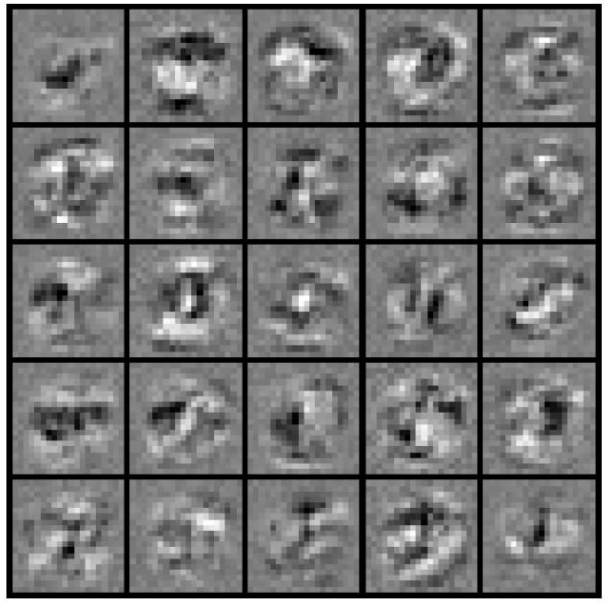
20 × 20 pixel images
$d = 400$ 10 classes

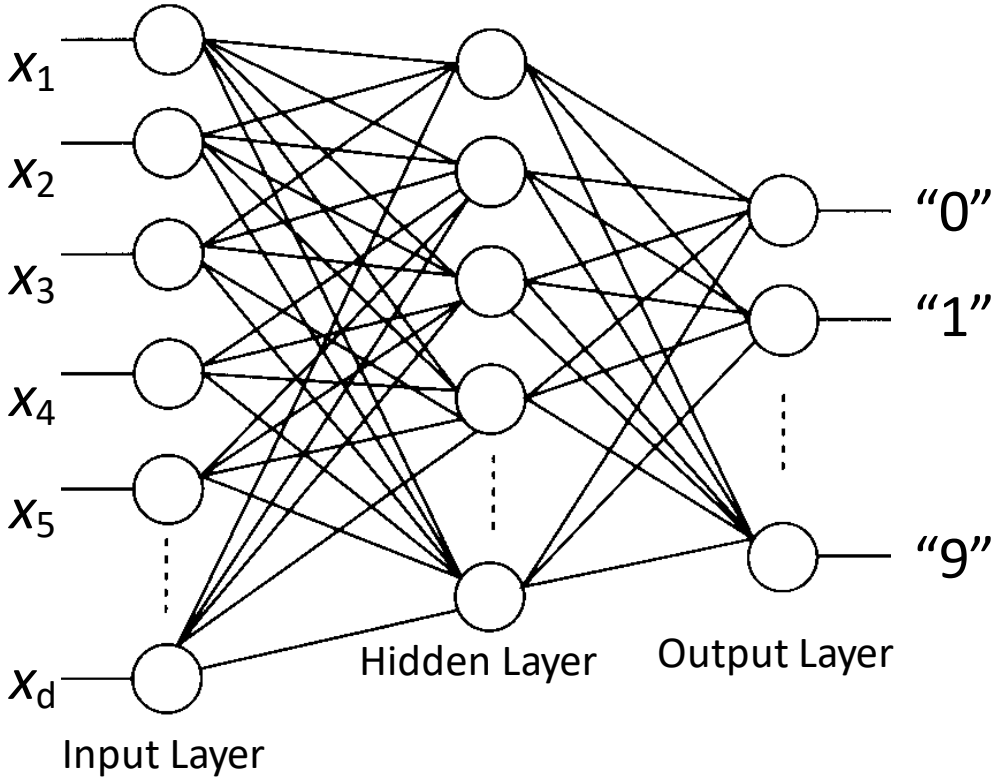


Each image is “unrolled” into a vector x of pixel intensities

Layering Representations



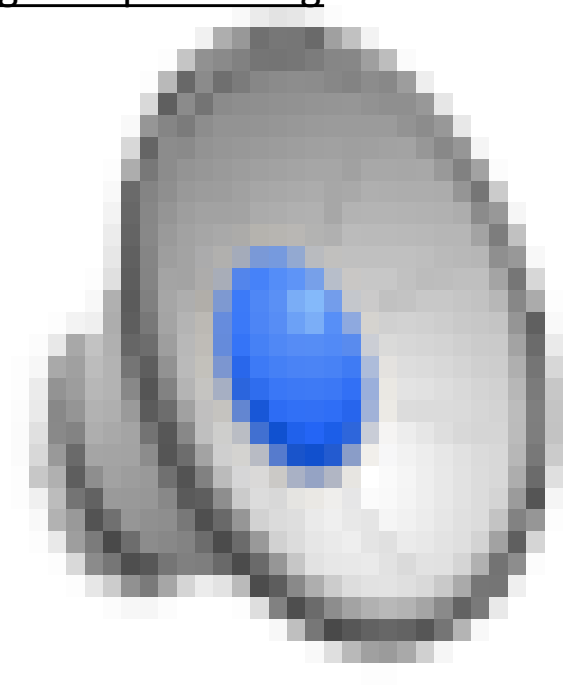




Visualization of
Hidden Layer

LeNet 5 Demonstration:
https://www.youtube.com/watch?v=FwFduRA_L6Q&ab_channel=YannLeCun

LeNet layers:
https://en.wikipedia.org/wiki/LeNet#:~:text=In%20general%2C%20LeNet%20refers%20to,in%20large%2Dscale%20image%20processing.



Neural Network Learning

Batch Perceptron

Given training data $\left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{n}$
Let $\boldsymbol{\theta} \leftarrow [0, 0, \ldots, 0]$
Repeat:
    Let $\boldsymbol{\Delta} \leftarrow [0, 0, \ldots, 0]$
    for $i = 1 \ldots n$, do
        if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} \leq 0$ // prediction for ith instance is incorrect
            $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} + y^{(i)}\boldsymbol{x}^{(i)}$
    $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta}/n$ // compute average update
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{\Delta}$
Until $\|\boldsymbol{\Delta}\|_2 < \epsilon$

- Simplest case: α = 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

- The trick is to assess the blame for the error and divide it among the contributing weights

Cost Function

Logistic Regression:

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

Neural Network: Binary classification (1 or 0)

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

General NN Cost Function

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(h_\Theta(\mathbf{x}_i), y_i\big) \quad + \frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(\Theta_{ji}^{(l)}\right)^2$$

Given $h_\Theta(\mathbf{x}_i) = \hat{y}$

Loss of Binary classification (1 or 0) ← log loss

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$$

Loss of Regression ← squared error loss

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

Loss of Multi-class classification with softmax regression ← cross-entropy loss

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{k} \mathbf{1}\{y = j\} \log \hat{y}_j = -\sum_{j=1}^{k} y_j \log \hat{y}_j$$

Optimizing the Neural Network

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(h_\Theta(\mathbf{x}_i), y_i\big) \quad + \frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(\Theta_{ji}^{(l)}\right)^2$$

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

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

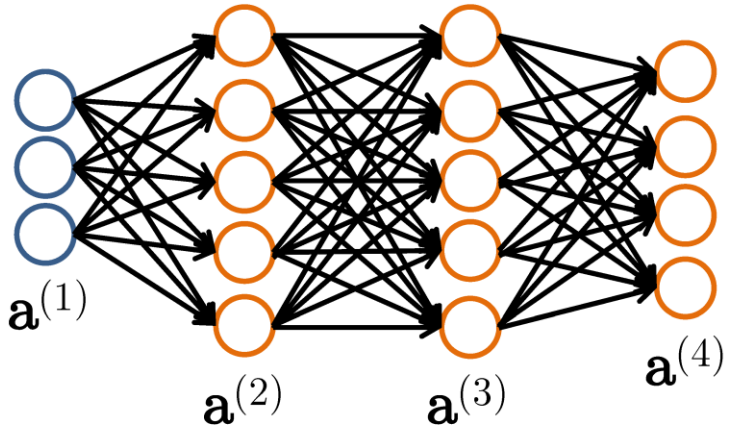
- But, tends to work well in practice

Need code to compute:

- $J(\Theta)$
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Forward Propagation

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



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

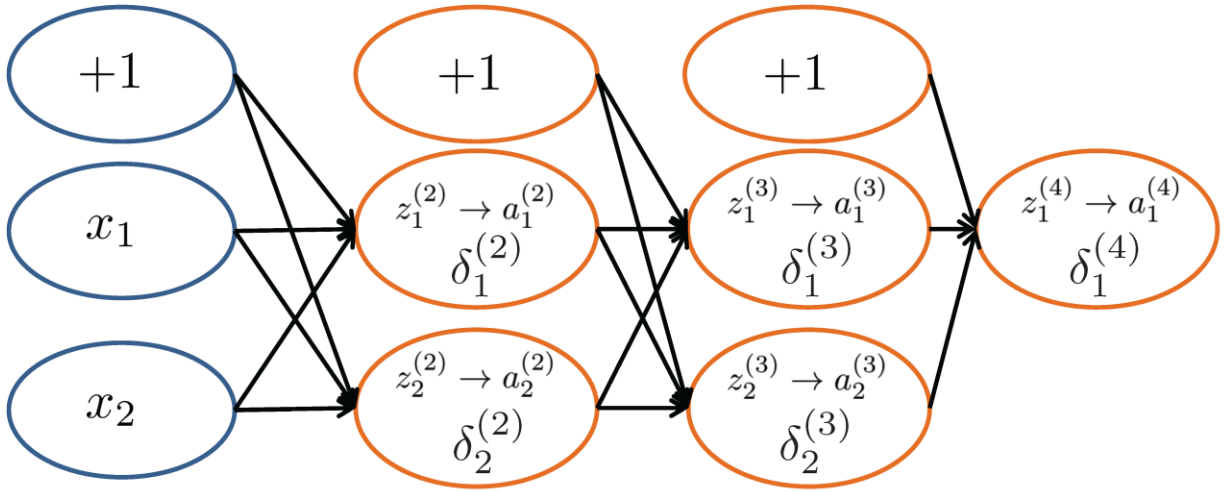
Backpropagation Intuition

- Each hidden node *j* is "responsible" for some fraction of the error $\delta_j^{(l)}$ in each of the output nodes to which it connects

- $\delta_j^{(l)}$ is divided according to the strength of the connection between hidden node and the output node

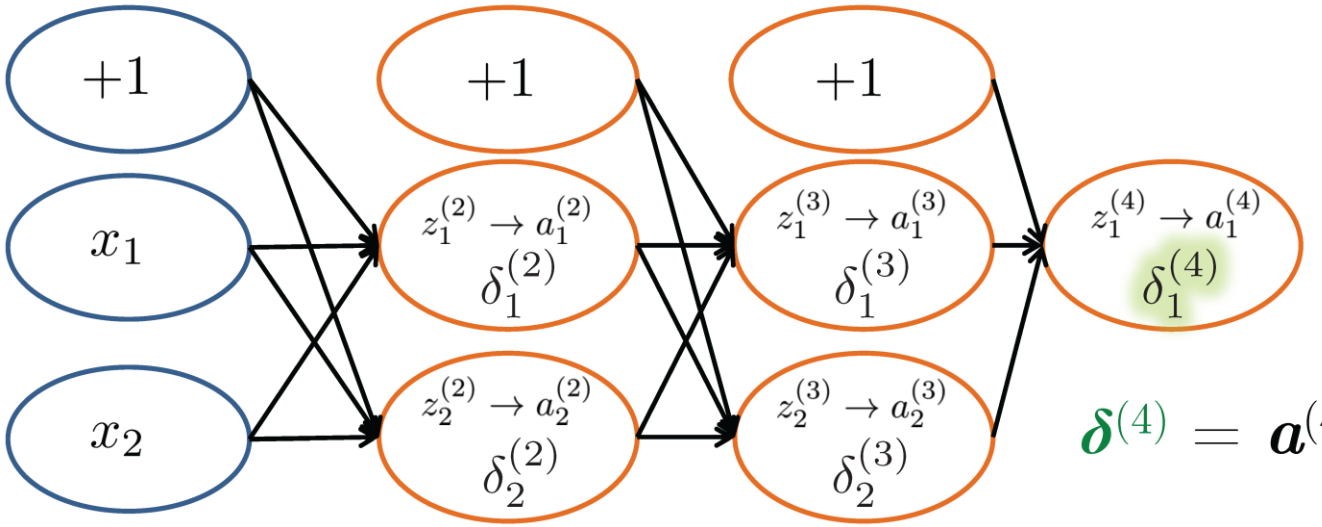
- Then, the "blame" is propagated back to provide the error values for the hidden layer

Backpropagation Intuition (binary classification)



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

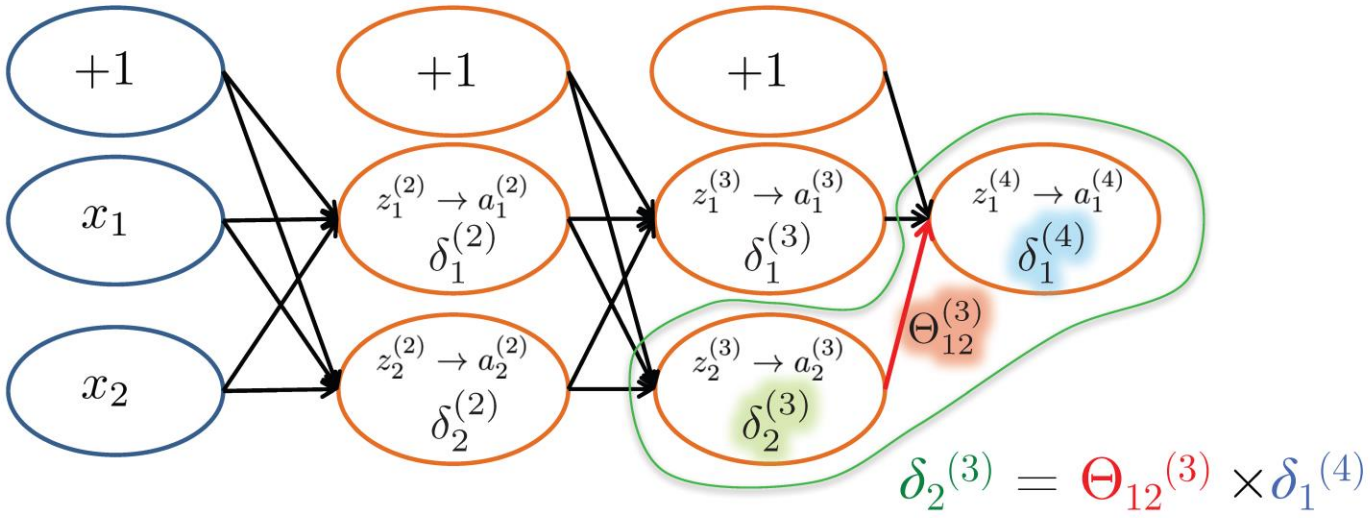
Backpropagation Intuition (binary classification)



$$\boldsymbol{\delta}^{(4)} = \boldsymbol{a}^{(4)} - \mathbf{y}$$

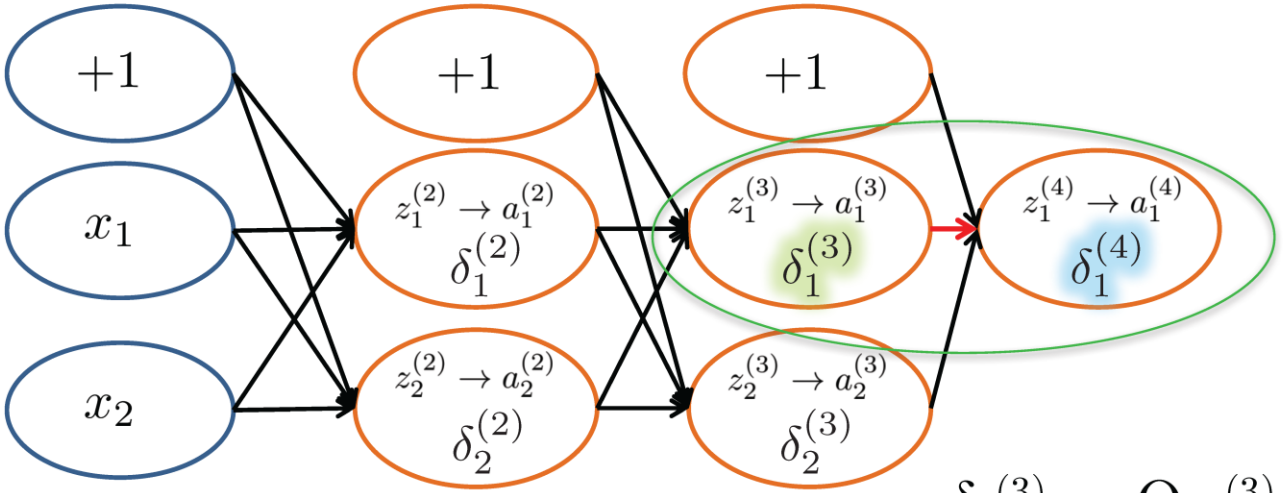
$\delta_j^{(l)} =$ “error” of node j in layer l

Backpropagation Intuition (binary classification)



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

Backpropagation Intuition (binary classification)

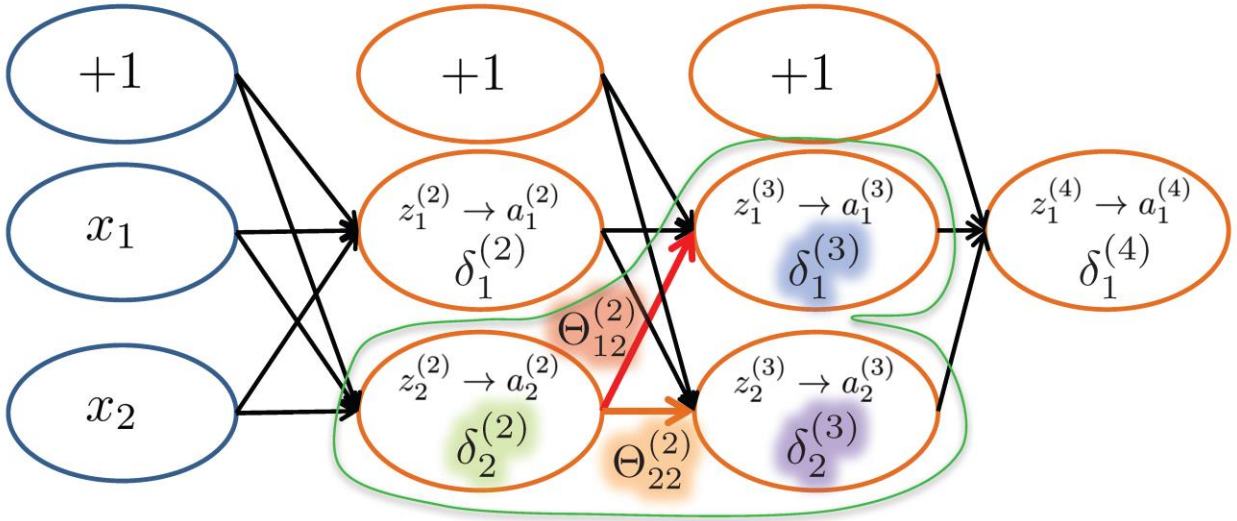


$$\delta_2^{(3)} = \Theta_{12}^{(3)} \times \delta_1^{(4)}$$

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

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

Backpropagation Intuition (binary classification)



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

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

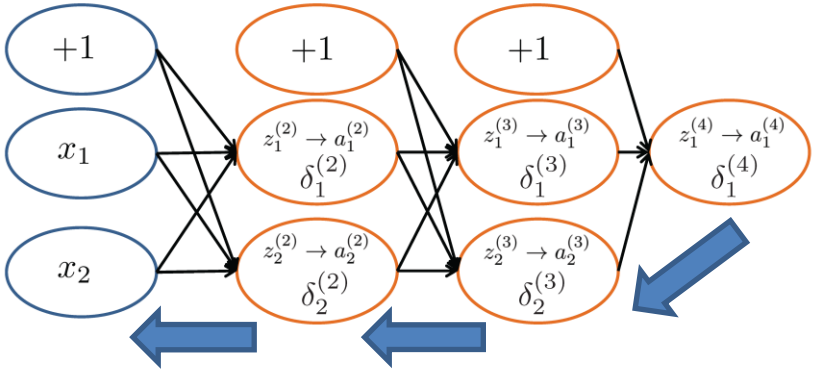
Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i) = \dfrac{\partial \text{cost}(x_i)}{\partial a_j^{(2)}} \times \dfrac{\partial a_j^{(2)}}{\partial z_j^{(2)}}$

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 (binary classification)

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

(#layers L = 4)



Backpropagation

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

Element-wise product .*

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

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

Assume g() is sigmoid

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} = \frac{\partial\, cost(x_i)}{\partial\, a_i^{(l+1)}} \cdot \frac{\partial\, a_i^{(l+1)}}{\partial\, z_i^{(l+1)}} \cdot \frac{\partial\, z_i^{(l+1)}}{\partial\, \Theta_{ij}^{(l)}}$$

(ignoring λ; if $\lambda = 0$)

Mathematical Proof: https://web.cs.wpi.edu/~kmlee/cs539/cs229-notes-deep_learning.pdf

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

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

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

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

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

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

Note: Can vectorize $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ as $\boldsymbol{\Delta}^{(l)} = \boldsymbol{\Delta}^{(l)} + \boldsymbol{\delta}^{(l+1)} \mathbf{a}^{(l)\mathsf{T}}$

Training a Neural Network via Gradient Descent with Backprop

Given: training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
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 $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
        Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_i$
        Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
        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