CS60050 Machine Learning

Neural Networks

Introduction, Backpropagation, Activation Functions

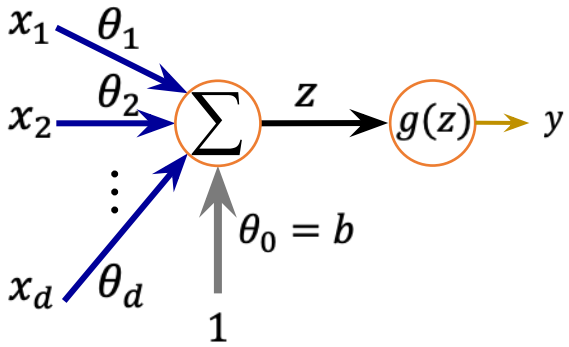
Somak Aditya, Sudeshna Sarkar
Department of CSE, IIT Kharagpur

Slide Courtesy: Prof. Sudeshna Sarkar (IIT KGP), Prof. Subhendu Trivedi/Rishi Kondor (UChicago), Andrew Ng (Stanford)

Neural Networks

- Origins: Algorithms (loosely) inspired by the brain.
- Very widely used in 80s and early 90s; popularity diminished in late 90s. (*the famed AI Winter*)
- Recent resurgence: State-of-the-art technique for many applications
- Disclaimer: Artificial neural networks are not nearly as complex or intricate as the actual brain structure

Logistic Regression



$$\boldsymbol{w} = [\theta_1\ \theta_2\ \dots \theta_d]^T$$

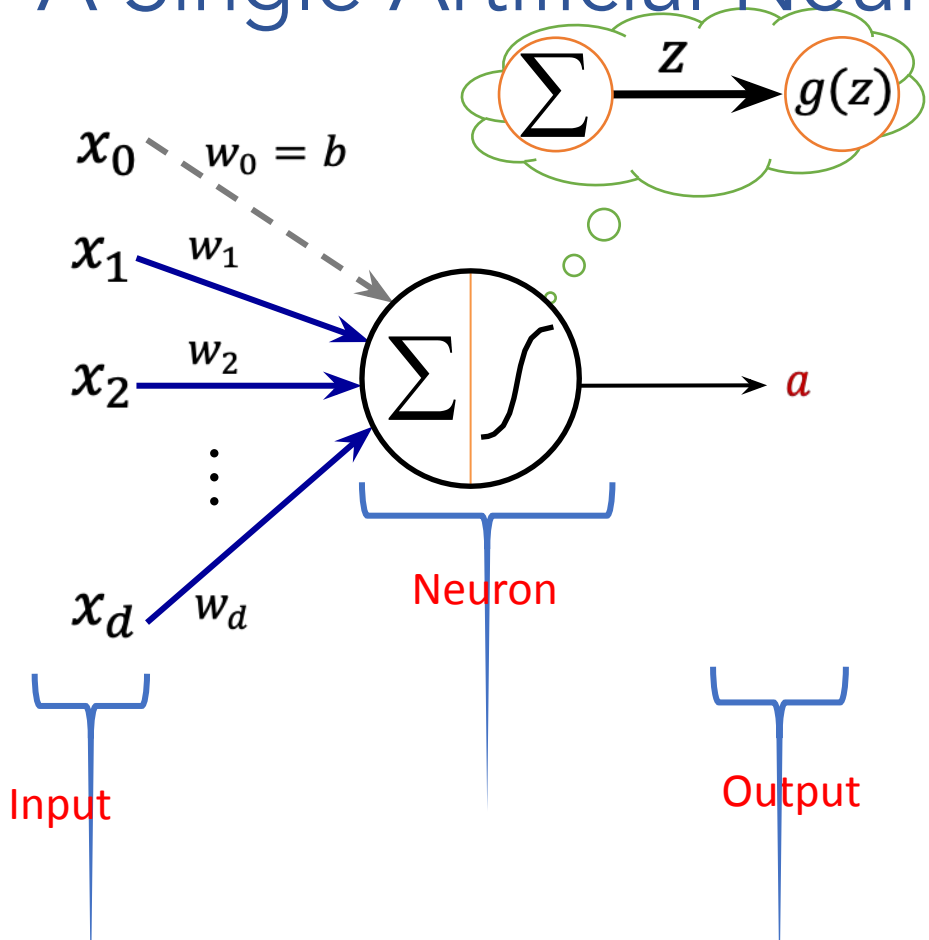
$$\boldsymbol{x} = [x_1\ x_2\ \dots x_d]^T$$

$$\boldsymbol{z} = b + \sum_{i=1}^{d} \theta_i x_i = [\boldsymbol{\theta}^T b] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$\boldsymbol{y} = g(z)$$

- Regression: $g(z) = z$
- Classification:
  - Binary: $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$
  - *Multi-class (more on this later)*

A Single Artificial Neuron



$\boldsymbol{w} = [w_1\ w_2 \dots w_d]^T$ and $\boldsymbol{x} = [x_1\ x_2 \dots x_d]^T$

$$\boldsymbol{z} = b + \sum_{i=1}^{d} w_i x_i = [\boldsymbol{w}^T b] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$\boldsymbol{a} = g(z)$$

Terminologies:-

$\boldsymbol{x}$: input, $\boldsymbol{w}$: weights, $\boldsymbol{b}$: bias

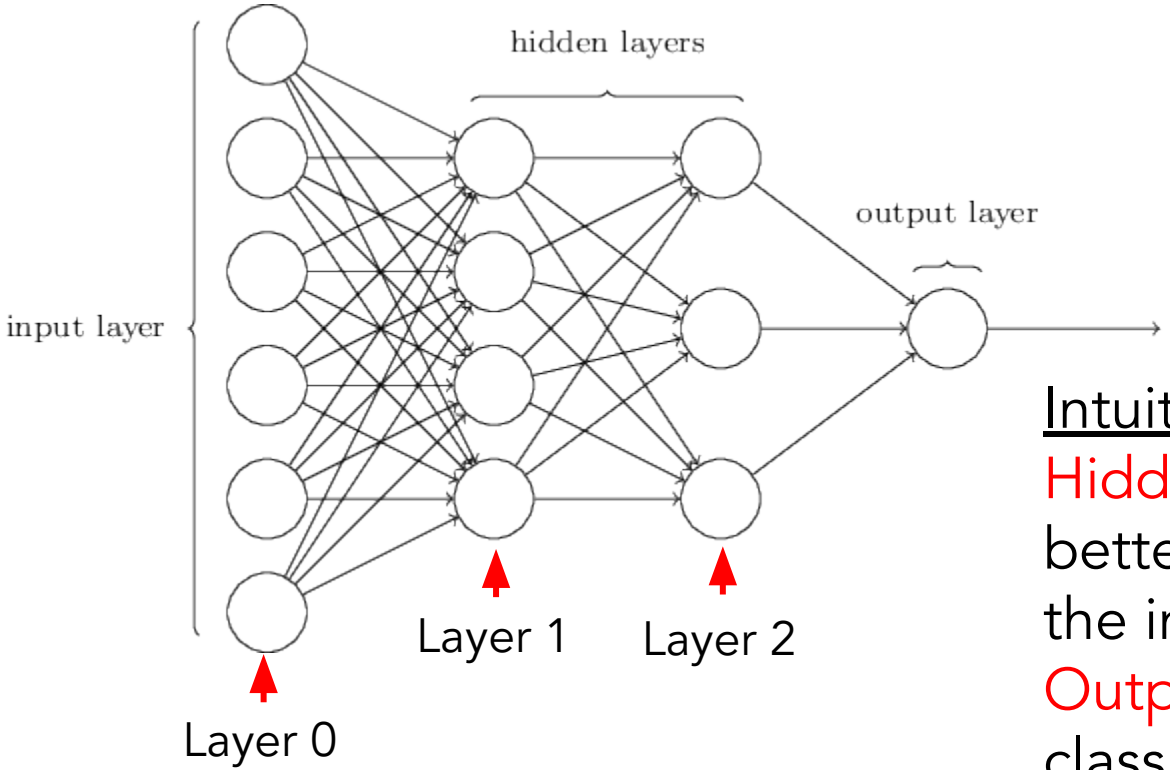
z: pre-activation (input activation)

g: activation function

y: activation (output activation)

a: activation at hidden units

Multilayer Perceptrons (MLP)



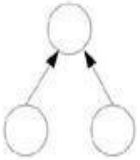
Intuition:-

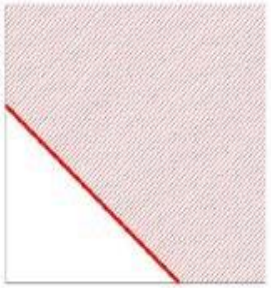
Hidden Layer: Extracts better representation of the input data

Output layer: Does the classification

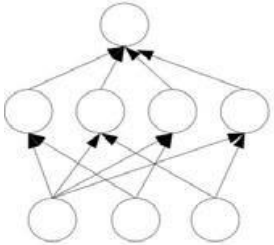
Beyond Single Layer

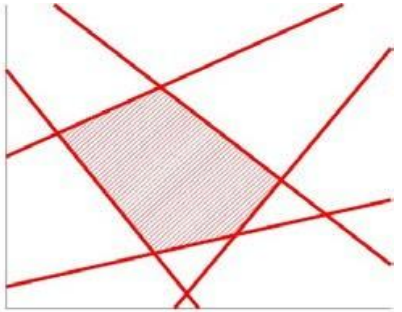
1 layer of trainable weights



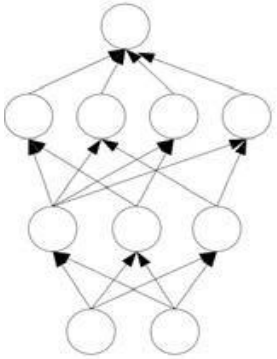


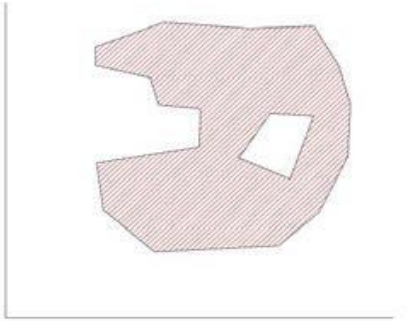
separating hyperplane





convex polygon region

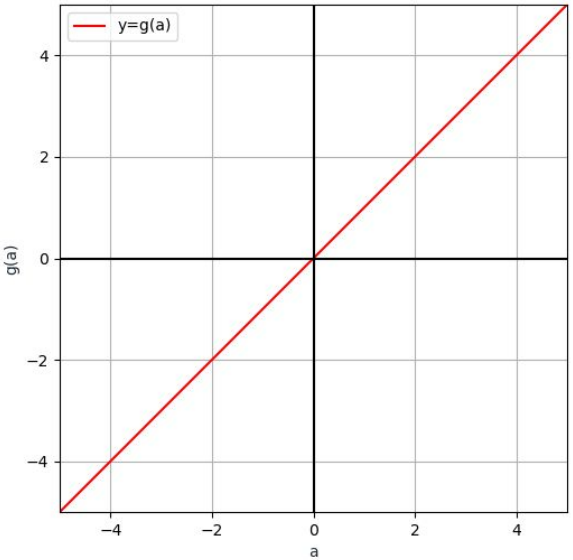




composition of polygons: convex regions

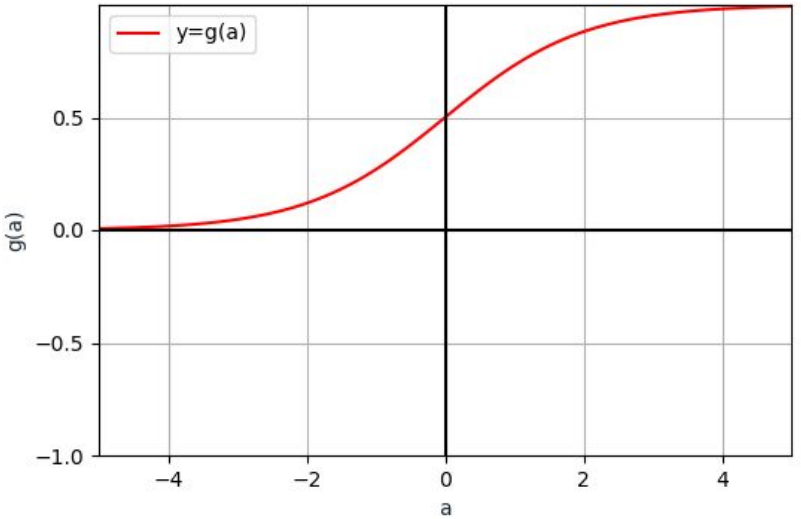
What happens to bias?
What happens to variance?

Common Activation Functions ($g(.)$)



Linear activation function

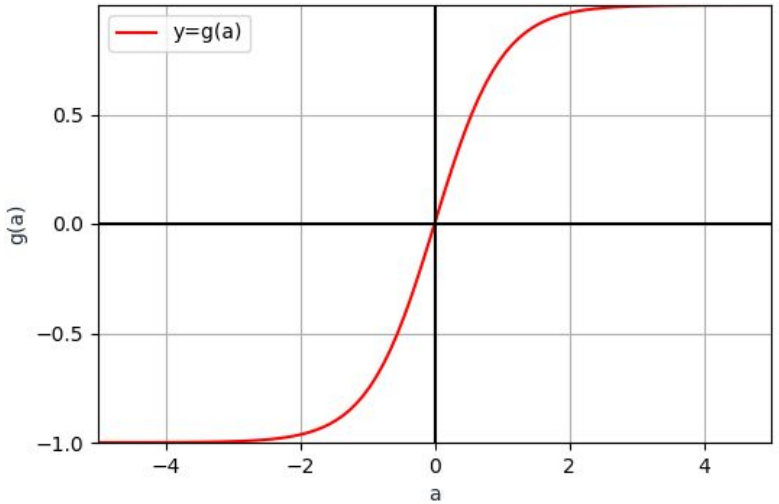
- $g(a) = a$
- Unbounded
- $g'(a) = 1$



Sigmoid activation function

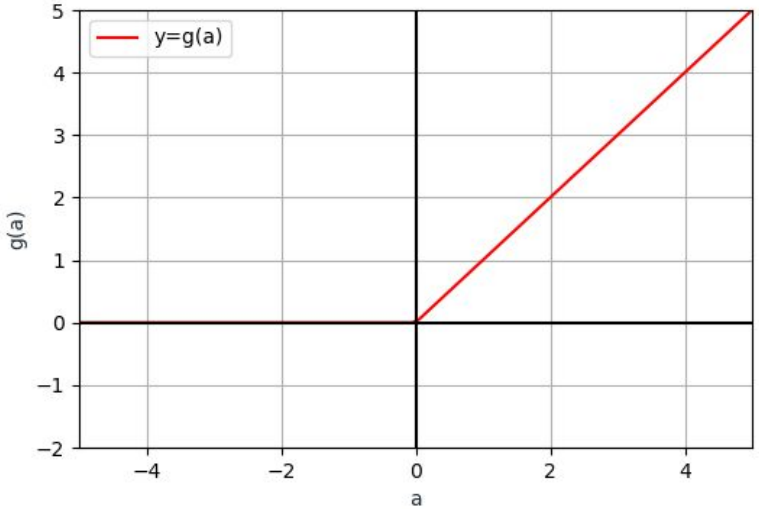
- $g(a) = \sigma(a) = \frac{1}{1+\exp(-a)}$
- Bounded (0, 1), Always positive
- $g'(a) = g(a)(1 - g(a))$

Common Activation Functions ($g(.)$)



tanh activation function

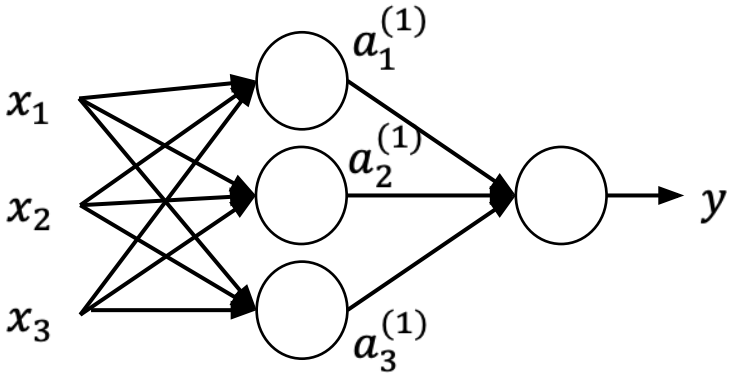
- $g(a) = tanh(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)}$
- Bounded (-1, 1)
- Can be positive or negative
- $g'(a) = 1 - g^2(a)$



ReLU activation function

- $g(a) = \max(0, a)$
- Bounded below by 0, But not upper-bounded
- $g'(a) = \begin{cases} 1, & a \geq 0 \\ 0, & a < 0 \end{cases}$

Why Non-Linear Activations**



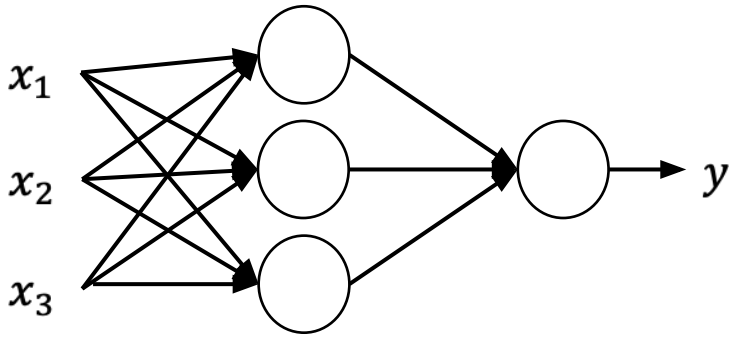
$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$a^{(1)} = g^{(1)}(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{1} + b^{(2)}$$
$$y = g^{(2)}(z^{(2)})$$

Assume if all are linear activation.
Task is to predict the yearly salary
$y \in \$0, \ldots., \$100{,}000$

$$a^{(1)} = z^{(1)} = W^{(1)}x + b^{(1)}$$
$$y = a^{(2)} = z^{(2)} = W^{(2)}a^{1} + b^{(2)}$$
$$= W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)}$$
$$= [W^{(2)}W^{(1)}]x + \{W^{(2)}b^{(1)} + b^{(2)}\}$$
$$= W'\,x + b'$$

$\Rightarrow$ y it is linear in x (the input)
$\Rightarrow$ y is in $\mathbb{R}$ (not what we need)

Vectorized Representation (*More on this later*)



$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$a^{(1)} = g^{(1)}(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{1} + b^{(2)}$$
$$y = g^{(2)}(z^{(2)})$$

$$z_1^{(1)} = W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}$$
$$z_2^{(1)} = W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}$$
$$z_3^{(1)} = W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}$$

3×3, 1×3

$$a_1^{(1)} = g^{(1)}(z_1^{(1)})$$
$$a_2^{(1)} = g^{(1)}(z_2^{(1)})$$
$$a_3^{(1)} = g^{(1)}(z_3^{(1)})$$

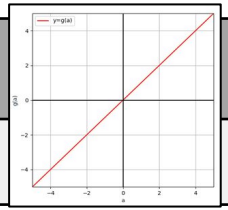
1×3

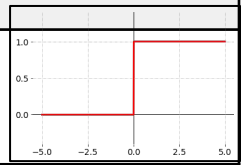
3×1,1×1

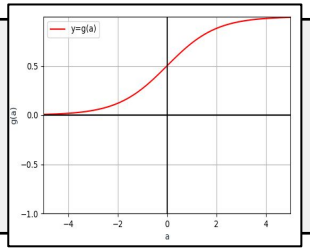
$$z_1^{(2)} = W_{11}^{(2)}a_1^{(1)} + W_{12}^{(2)}a_2^{(1)} + W_{13}^{(2)}a_3^{(1)} + b^{(2)}$$
$$y = g^{(2)}(z^{(2)})$$

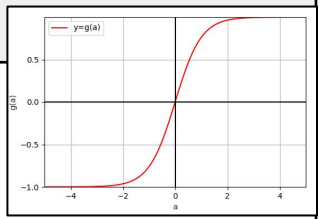
Common Activation Functions

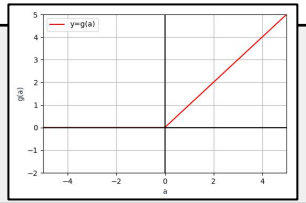
Name	Function	Gradient	Graph
Linear			
Binary step			
Sigmoid			
Tanh			
ReLU			











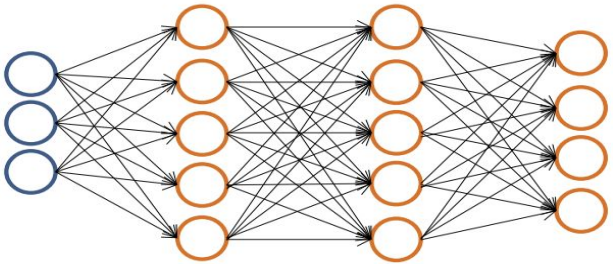
Other Network Architectures

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

Output Unit Activations

Regression: Linear

Expected Output $\in \mathbb{R}$

$$\hat{y} = w^T a + b$$

Maximizing log-likelihood $\Longrightarrow$
Minimizing squared error

$$J(\theta) = ||y - \hat{y}_\theta||^2$$

Binary Classification: Sigmoid

Expected Output $\in [0,1]$

$$\hat{y} = \sigma(w^T a + b)$$

$$J(\theta) = -\log p(y|x)$$
$$= -\log \sigma\big((2y-1)(\boldsymbol{w}^T \boldsymbol{a} + b)\big)$$

Multi-Class Classification: Softmax

Expected Output $\in [0,1]^C$

Say, 3 classes.

- If output is Class 1, [1 0 0]
- If output is Class 2, [0 1 0]
- If output is Class 3, [0 0 1]

Need to produce a vector $\hat{y}$ with $\hat{y}_i = p(y = i|x)$

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

Loss Functions

Regression

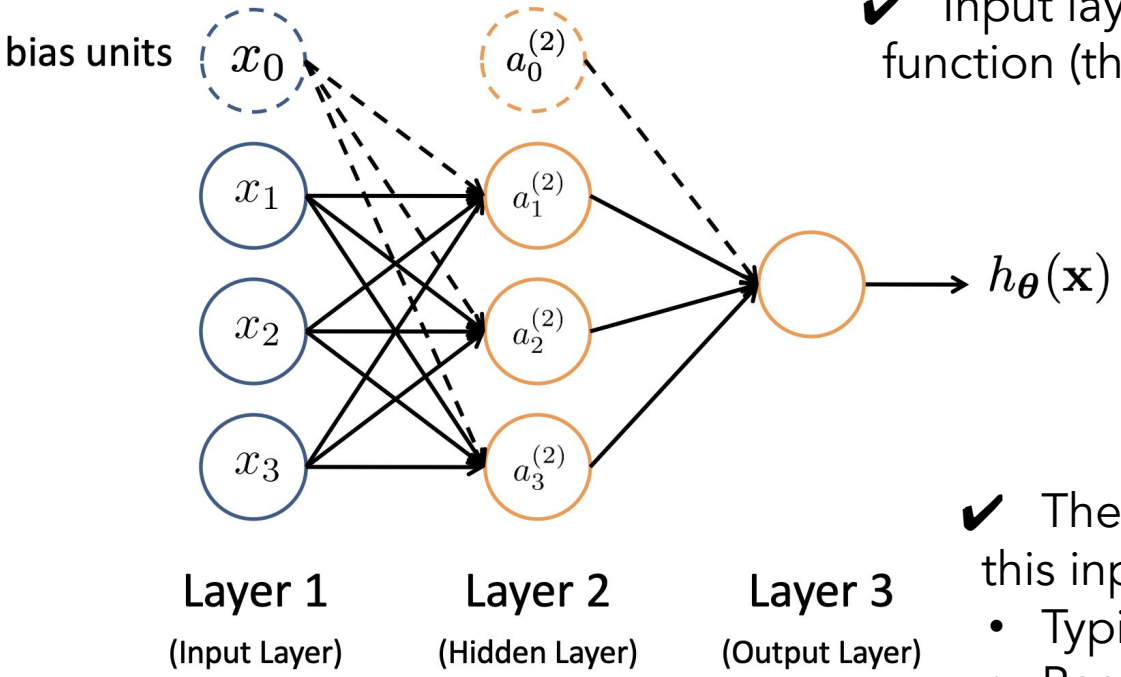
- Squared error: $\mathrm{L}(y, \hat{y}) = (y - \hat{y})^2$

Classification

- Cross entropy: : $L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_k \, y_k \log \hat{y}_k$
- Easy to scale to k classes.

Forward Pass: Computing a Neural Network's Output

Neural Network – Forward Pass



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

✔ Working forward through the network, the **input function** of each unit is applied to compute the input value.

✔ The **activation function** transforms this input function into a final value

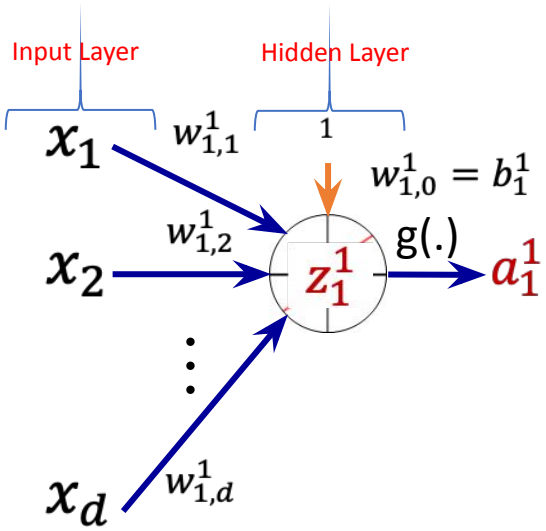
- Typically this is a nonlinear function
- Possibly "Thresholding"

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

Multilayer Neural Network



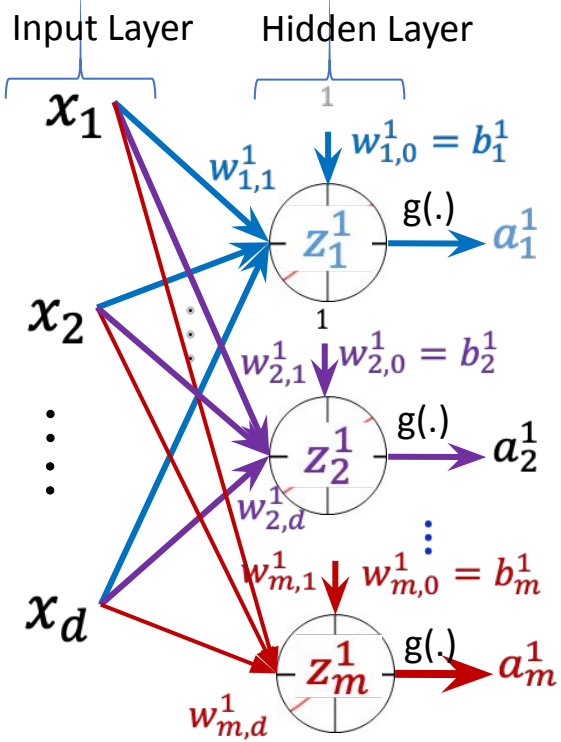
$$z_1^1 = b_1^1 + \sum_{i=1}^{d} w_{1,i}^1 x_i = [w_1^1 b_1^1] \begin{bmatrix} x \\ 1 \end{bmatrix}$$

$$[x_1\ x_2 \dots x_d]^T$$

$$a_1^1 = g(z_1^1)$$

$w_{i \leftarrow j}^l$ = Connection from $l-1^{\text{th}}$ layer, j^{th} neuron TO l^{th} layer, i^{th} neuron

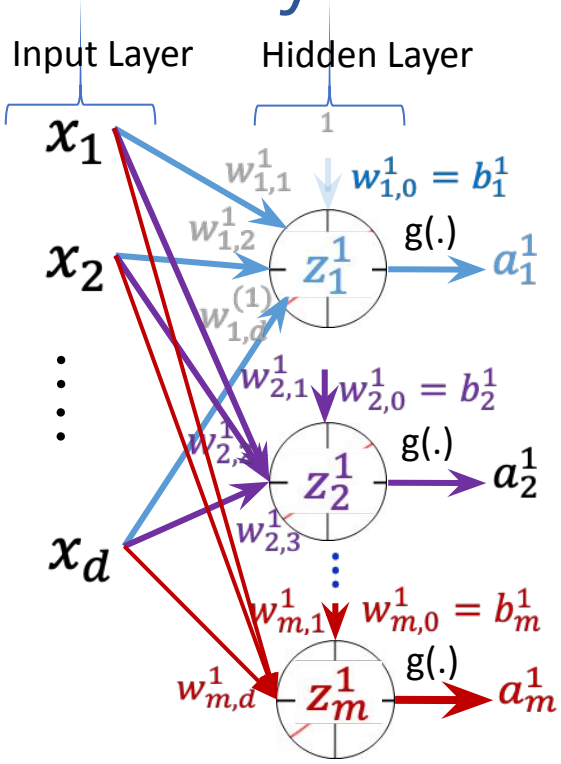
Multilayer Neural Network



$$a^{(0)} = x$$
$$z^{(1)} = \boldsymbol{w}^{(1)} a^{(0)}$$
$$a^{(1)} = g\left(\boldsymbol{z}^{(1)}\right)$$

$W^1 : m \times n$ matrix
$b^1 : m \times 1$ column vector
$X : d \times 1$ column vector
$Z^1 : m \times 1$ column vector
$A^1 : m \times 1$ column vector

Multilayer Neural Network



$$a^{(0)} = x$$
$$z^{(1)} = w^{(1)} a^{(0)}$$
$$a^{(1)} = g\left(z^{(1)}\right)$$

$$\begin{bmatrix} z^1_1 \\ z^1_2 \\ \vdots \\ z^1_m \end{bmatrix} = \begin{bmatrix} w^1_1 & b^1_1 \\ w^1_2 & b^1_2 \\ \vdots & \vdots \\ w^1_m & b^1_m \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix}$$

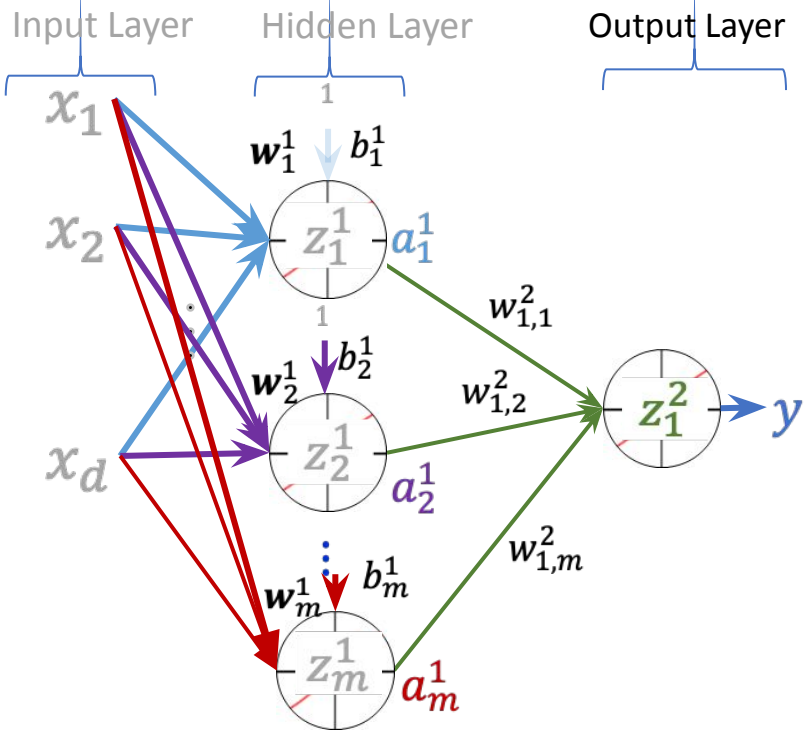
$$z^1 = [W^1 b^1] \begin{bmatrix} x \\ 1 \end{bmatrix}$$

$$a^1 = g\left(z^{(1)}\right)$$

$$z^1_1 = [w^1_1 b^1_1] \begin{bmatrix} x \\ 1 \end{bmatrix}$$
$$z^1_2 = [w^1_2 b^1_2] \begin{bmatrix} x \\ 1 \end{bmatrix}$$
$$\vdots$$
$$z^1_M = [w^1_m b^1_2] \begin{bmatrix} x \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a^1_1 \\ a^1_2 \\ \vdots \\ a^1_m \end{bmatrix} = \begin{bmatrix} g(z^1_1) \\ g(z^1_2) \\ \vdots \\ g(z^1_m) \end{bmatrix}$$

Multilayer Neural Network



Output Layer Pre-activation

$$z_1^{(2)} = \left[\boldsymbol{w}_1^{(2)}\ b_1^{(2)}\right]\begin{bmatrix}\boldsymbol{a}^{(1)}\\ 1\end{bmatrix}$$

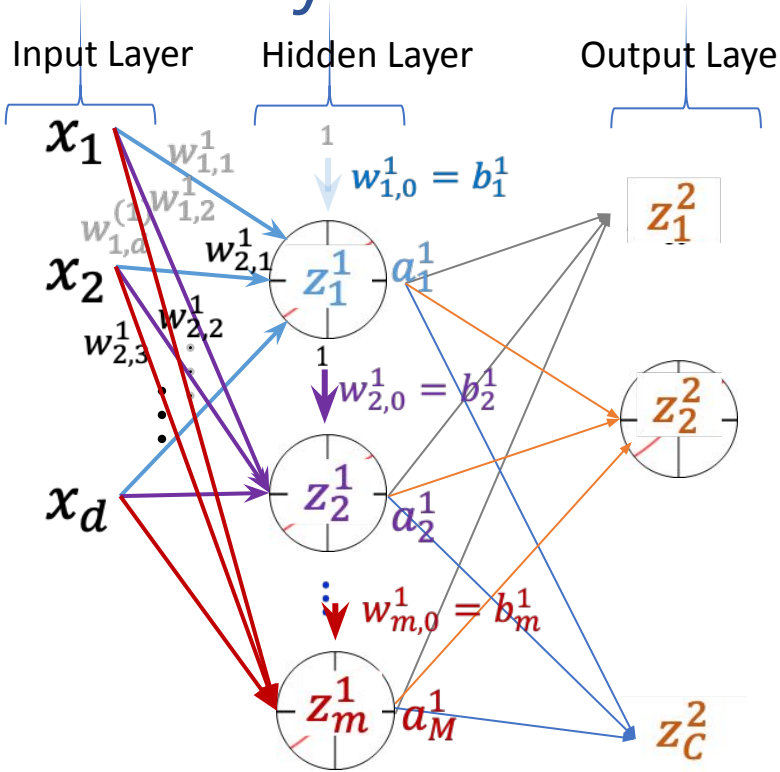
Output Layer Activation

$$y_1 = o\left(\boldsymbol{z}_1^{(2)}\right)$$

output

- Sigmoid for 2-class classification
- Softmax for multi-class classification
- Linear for regression

Multilayer Neural Network

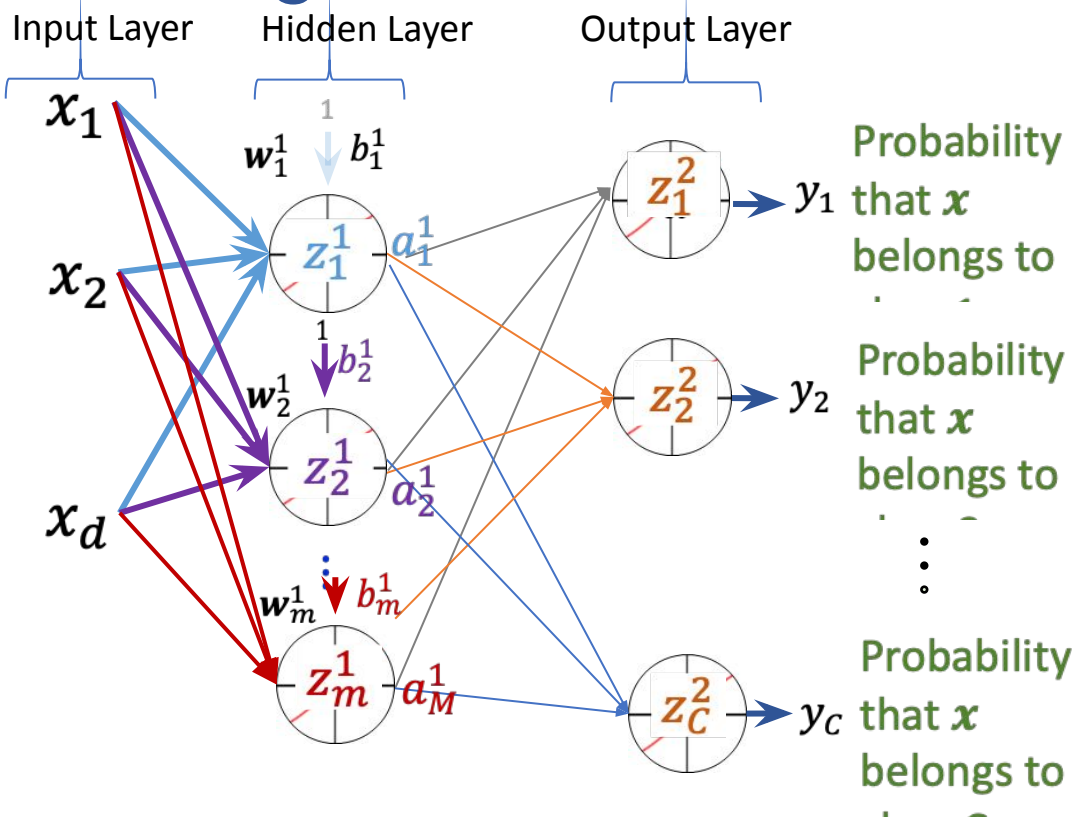


$$y_1 = o_1\left(z_1^{(2)}\right) = \frac{\exp(z_1^{(2)})}{\sum_c \exp(z_c^{(2)})}$$

$$y_2 = o_2\left(z_2^{(2)}\right) = \frac{\exp(z_2^{(2)})}{\sum_c \exp(z_c^{(2)})}$$

$$y_C = o_C\left(z_C^{(2)}\right) = \frac{\exp(z_C^{(2)})}{\sum_c \exp(z_c^{(2)})}$$

Training a Neural Network – Loss Function



Aim to maximize the probability corresponding to the correct class for any example $\boldsymbol{x}$

$$\max \boldsymbol{y}_c \equiv \max (log\, \boldsymbol{y}_c)$$
$$\equiv \min (-log\, \boldsymbol{y}_c)$$
$$\equiv -\sum_i \prod_{i=c} \log(y_i)$$

known as cross-entropy loss

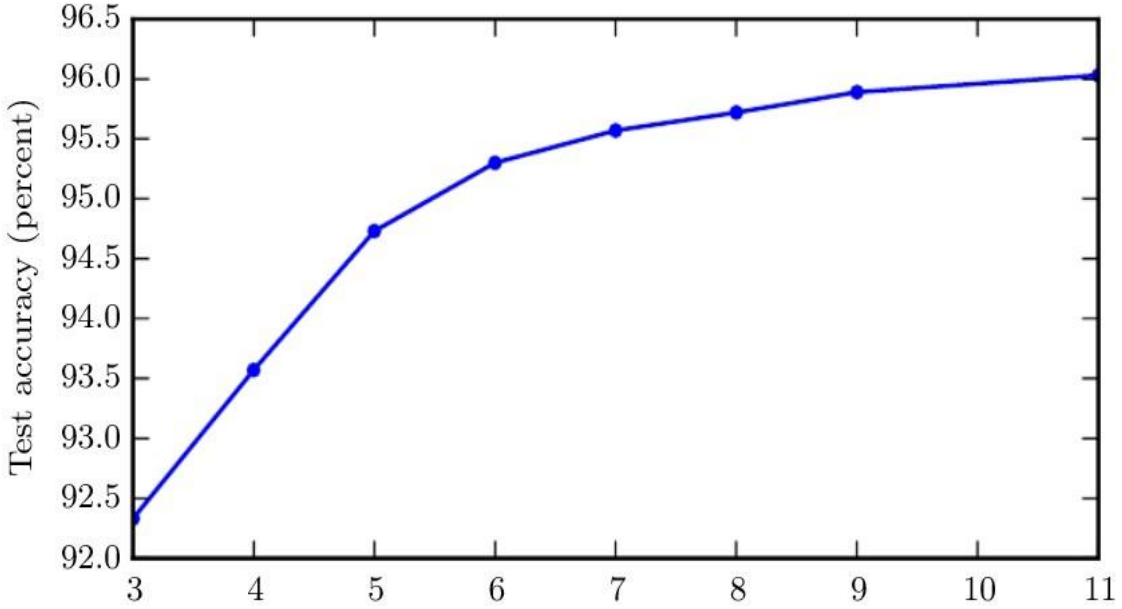
Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, we know a large MLP can represent this function
- But not guaranteed that our training algorithm will be able to learn that function
- Gives no guidance on how large the network will be (exponential size in worst case)

A visual proof that neural nets can compute any function

- See Chapter 4 of Neural Networks and Deep Learning by Michael Nielsen
- http://neuralnetworksanddeeplearning.com/chap4.html

Advantages of Depth

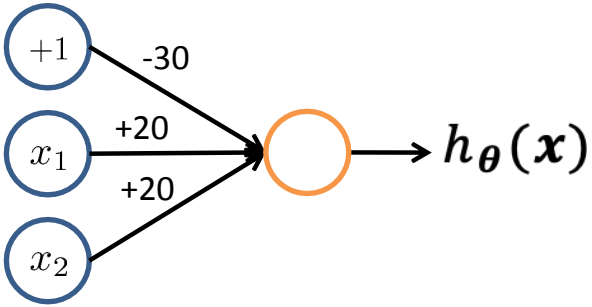


Representing Boolean Functions

Simple example: AND

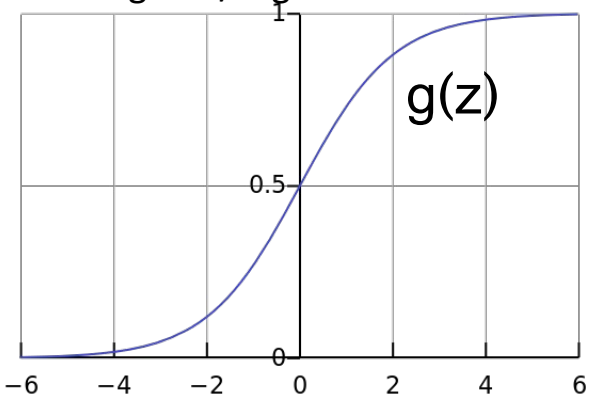
$x_1, x_2 \in \{0, 1\}$

$y = x_1 \text{ AND } x_2$



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

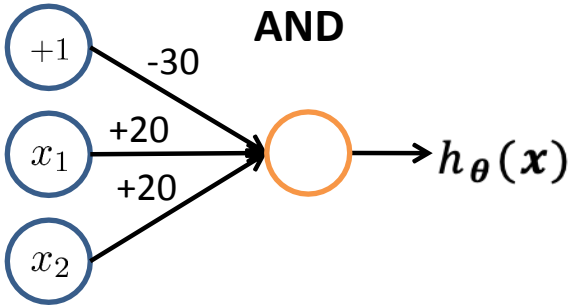
Logistic / Sigmoid Function

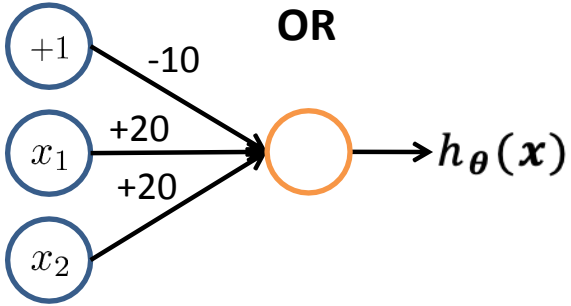


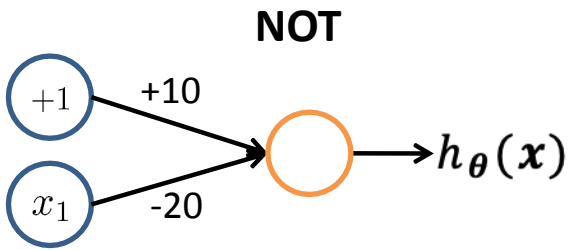
x_1	x_2	$h_\Theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

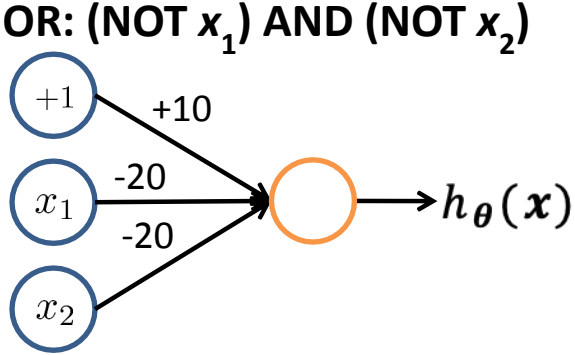
Based on slide and example by Andrew

Representing Boolean Functions



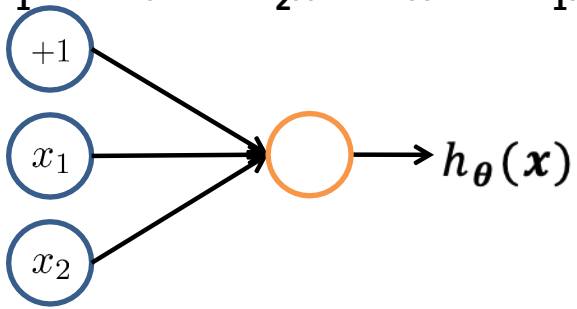


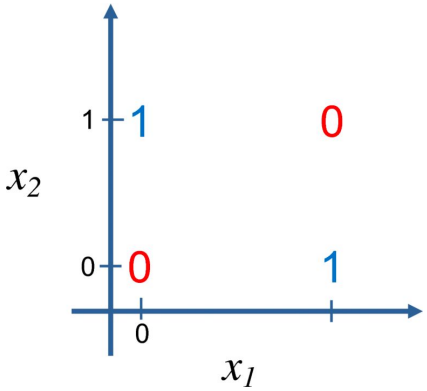




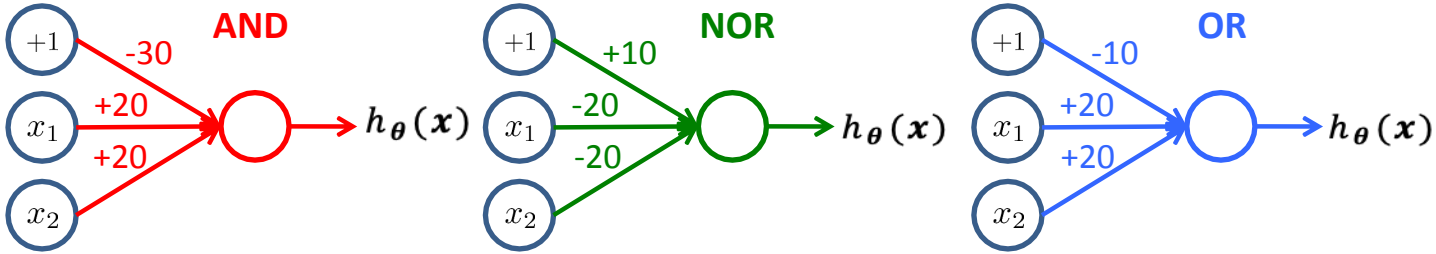
Representing Boolean Functions

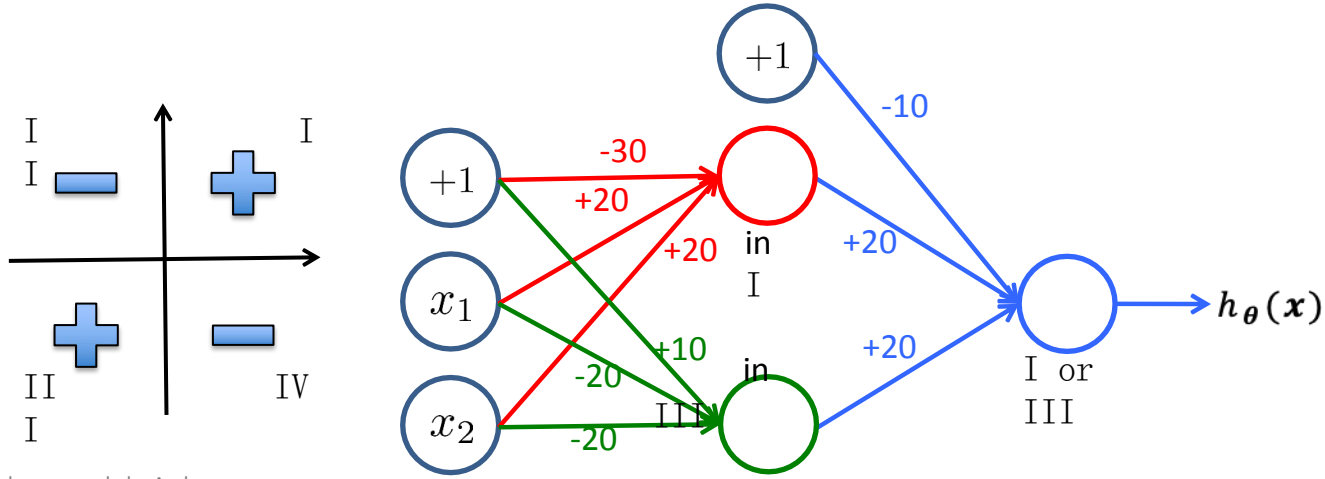
XOR: (x_1 AND (NOT x_2)) OR ((NOT x_1) AND x_2)





Combining Representations to Create Non Linear Functions

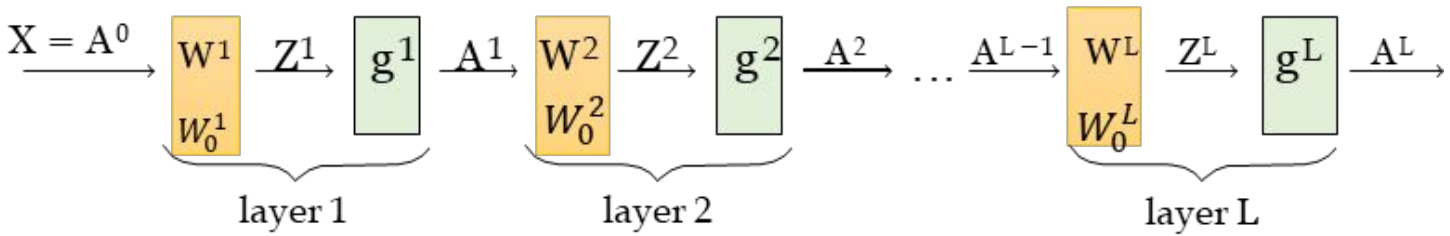




Based on example by Andrew

Feedforward Networks and Backpropagation

Forward Pass (Block Diagram)



Hidden layer pre-activation:

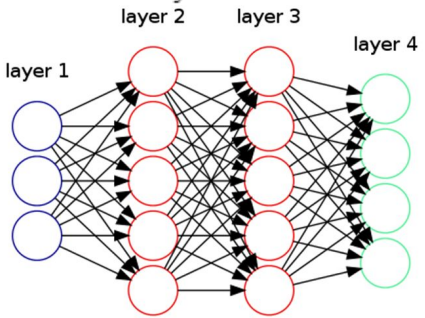
For $l = 1, \ldots, L$; $\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{W}_{\boldsymbol{0}}^{(l)}$

Hidden layer activation:

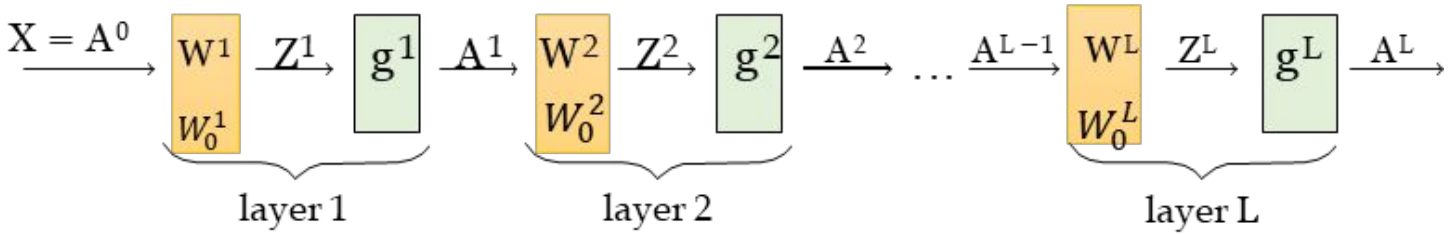
For $l = 1, \ldots, L-1$; $\boldsymbol{a}^{(l)} = g^{l}(\boldsymbol{z}^{(l)})$

Output layer activation:

For $l = L$; $\boldsymbol{y} = \boldsymbol{a}^{(L)} = o(\boldsymbol{z}^{(L)}) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta})$



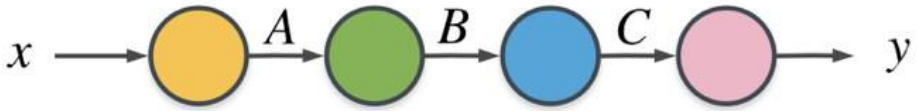
Error back-propagation



1. Compute Loss
2. Compute the derivative of the L w.r.t. the final output of the network A^L
3. Compute the derivative of L w.r.t. the pre-activation Z^L
4. Compute the derivative of L wrt W^L
5. Then compute the derivative of the L w.r.t. the final output of the network A^{L-1}
6. Then compute the derivative of L w.r.t. the pre-activation Z^{L-1}
7. Compute the derivative of L wrt W^{L-1}

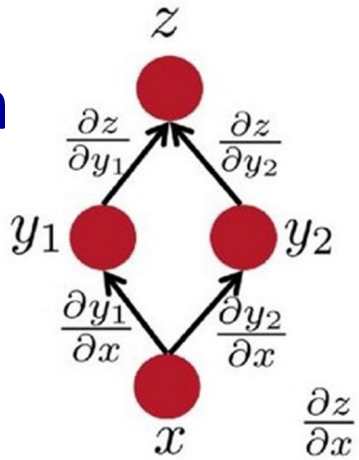
…. ….

Chain Rule



$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial C} \times \frac{\partial C}{\partial B} \times \frac{\partial B}{\partial A} \times \frac{\partial A}{\partial x}$$

Multiple Path



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Exercise:

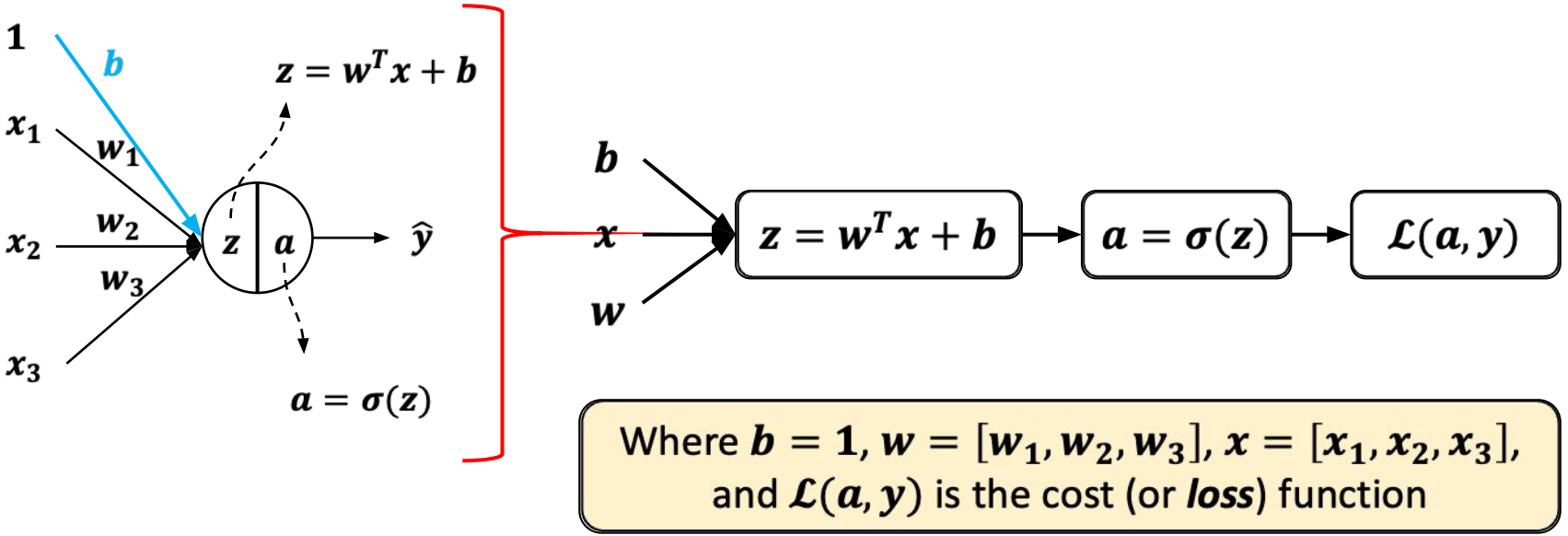
$$y_1 = w_1 x$$
$$y_2 = w_2 x$$
$$z = y_1 + y_2$$

What is $\frac{\partial z}{\partial x}$?

Backpropagation is just repeated application of the chain rule.

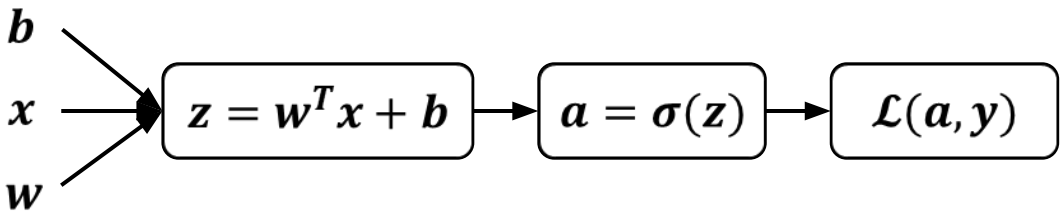
The Computation Graph of Logistic Regression

Let us translate logistic regression into a computation graph

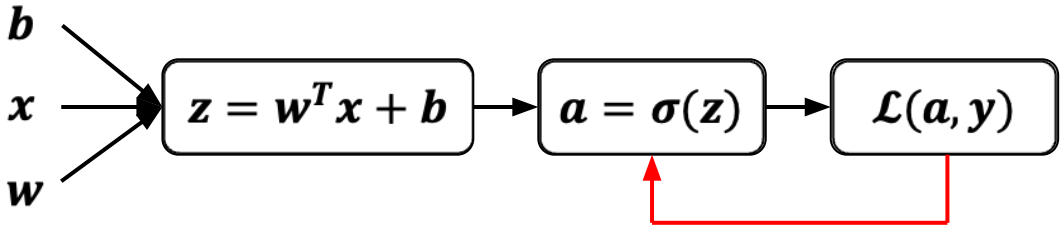


Forward Propagation

- The loss function can be computed by moving from left to right



Backward Propagation

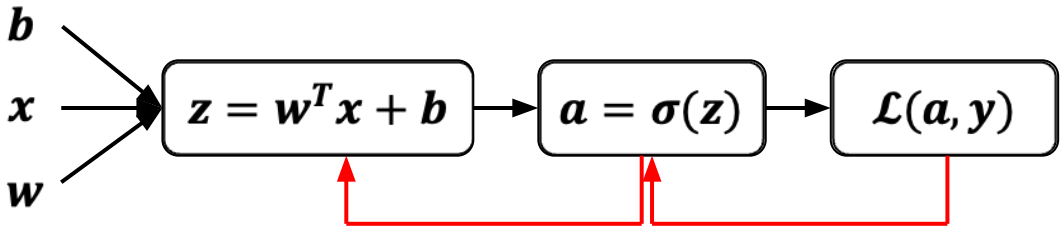


Partial derivative of $\mathcal{L}$ with respect to a

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial}{\partial a}\left(-y log(a) - (1-y) log(1-a)\right)$$

$$= \frac{-y}{a} + \frac{(1-y)}{(1-a)}$$

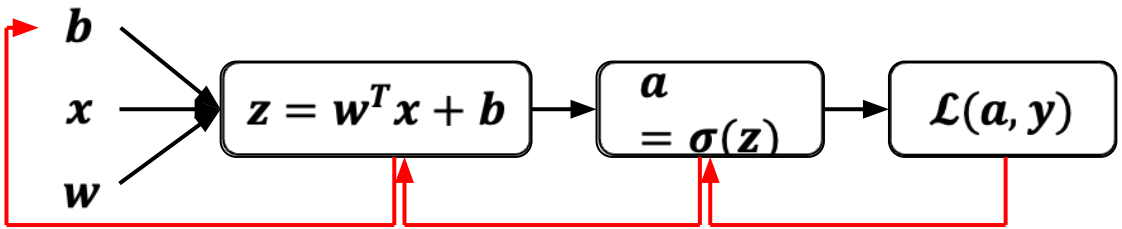
Backward Propagation



$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z} = \left(\frac{-y}{a} + \frac{(1-y)}{(1-a)} \right) \times \frac{\partial a}{\partial z} = \left(\frac{-y}{a} + \frac{(1-y)}{(1-a)} \right) \times a(1-a)$$

$$= a - y$$

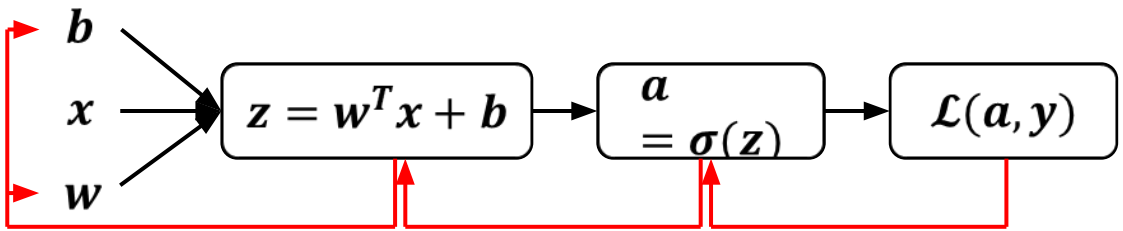
Backward Propagation



$\frac{\partial \mathcal{L}}{\partial b}$ = Partial derivative of $\mathcal{L}$ with respect to b

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial b} = (a - y) \times \frac{\partial z}{\partial b} = (a - y) \times 1 = (a - y)$$

Backward Propagation



$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w} = (a - y) \times \frac{\partial z}{\partial w} = (a - y)x$$

Backward Propagation: Summary

- Here is the summary of the gradients in logistic regression:

$$dz = \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z} = a - y$$

$$db = \frac{\partial \mathcal{L}}{\partial b} = a - y$$

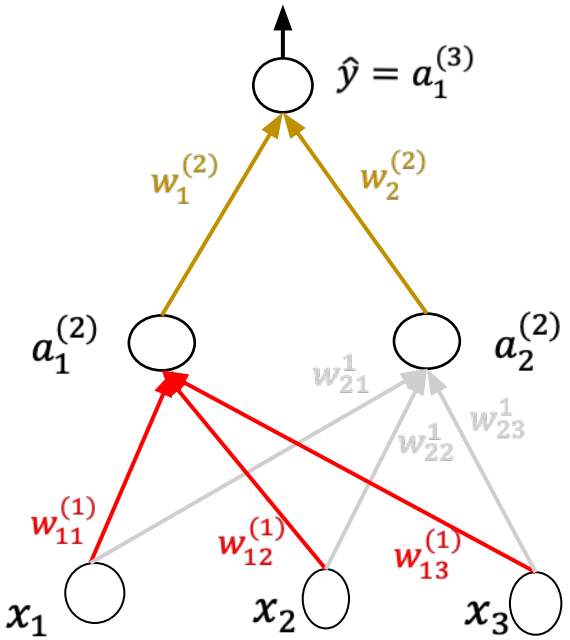
$$dw = \frac{\partial \mathcal{L}}{\partial w} = (a - y)x$$

Backward Pass MLP

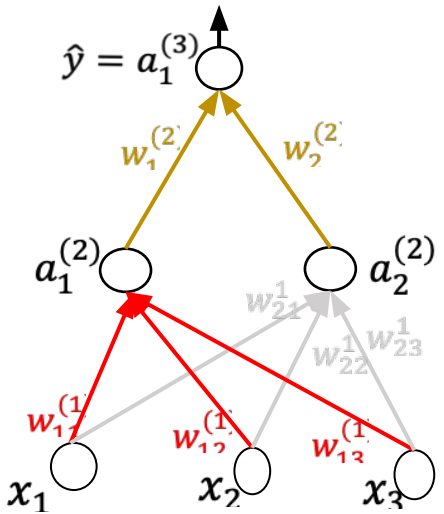
Optimizing the Neural Network

$$z_1^{(2)} = w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3$$
$$a_1^{(2)} = g(z_1^{(2)})$$
$$z_2^{(2)} = w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{23}^{(1)} x_3$$
$$a_2^{(2)} = g(z_2^{(2)})$$
$$z_1^{(3)} = w_1^{(2)} a_1^{(2)} + w_2^{(2)} a_2^{(2)}$$
$$\hat{y} = g\left(z_1^{(3)}\right) = a_1^{(3)}$$

- Loss $L = (\hat{y} - y)^2$
- *Compute* $\frac{\partial L}{\partial w_1^{(2)}}, \frac{\partial L}{\partial w_2^{(2)}}$
- *Compute* $\frac{\partial L}{\partial w_{11}^{(2)}}, \frac{\partial L}{\partial w_{12}^{(2)}}, \dots, \frac{\partial L}{\partial w_{23}^{(2)}}$



Top Layer Updates



Output $\hat{y} = a_1^3 = g\left(w_1^{(2)} a_1^{(2)} + w_2^{(2)} a_2^{(2)}\right)$, Loss $L = (\hat{y} - y)^2$

Compute $\frac{\partial L}{\partial w_1^{(2)}}, \frac{\partial L}{\partial w_2^{(2)}}$

$$\frac{\partial L}{\partial w_1^{(2)}} = \frac{\partial (\hat{y}-y)^2}{\partial w_1^{(2)}}$$

$$= 2(\hat{y} - y) \frac{\partial (g(w_1^{(2)} a_1^{(2)} + w_2^{(2)} a_2^{(2)}))}{\partial w_1^{(2)}}$$

$$= 2(\hat{y} - y) a_1^{3'} a_1^{(2)}$$

Weight update (Gradient Descent): $w_1^{(2)} = w_1^{(2)} - \eta \frac{\partial L}{\partial w_1^{(2)}}$

Assume $\delta = 2(\hat{y} - y) a_1^{3'}$, *the updates become*

- $w_1^{(2)} = w_1^{(2)} - \eta \delta a_1$; $\quad w_2^{(2)} = w_2^{(2)} - \eta \delta a_2$;

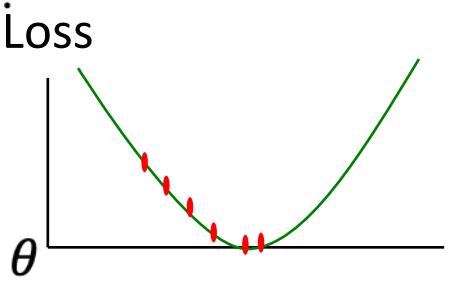
Background: Gradient descent

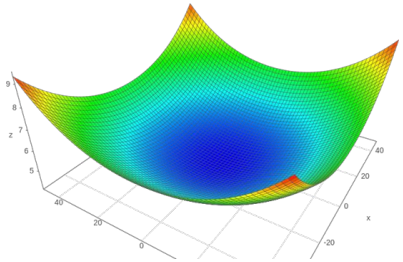
- Partial derivatives give us the slope (i.e. direction to move) in that dimension

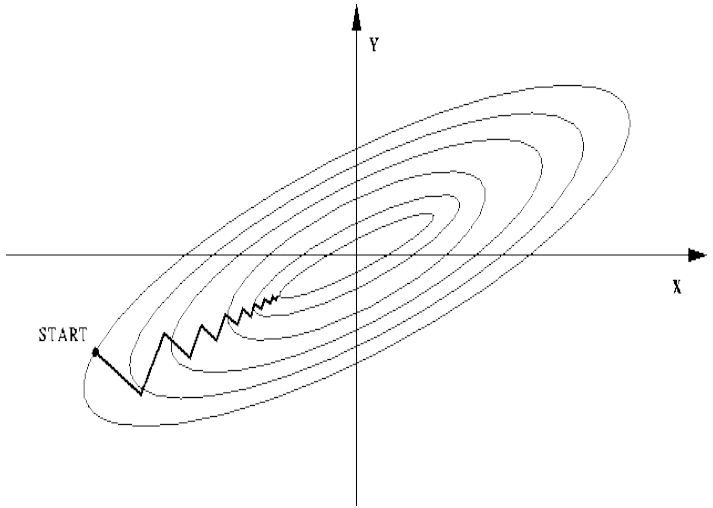
Approach:

- pick a starting point (θ)
- repeat:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

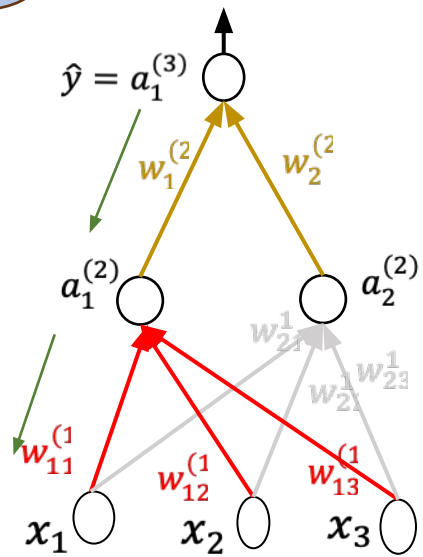
$$\theta_j = \theta_j - \eta \frac{d}{d\theta_j} \text{Loss}(\theta)$$







Next Layer Updates



Loss $L = (\hat{y} - y)^2$, *Compute* $\frac{\partial L}{\partial w_1^{(2)}}, \frac{\partial L}{\partial w_2^{(2)}}$

- $\frac{\partial L}{\partial w_{11}^{(1)}} = \frac{\partial (\hat{y}-y)^2}{\partial w_{11}^{(1)}}$

$$= 2(\hat{y} - y) \frac{\partial (g(w_1^{(2)} a_1^{(2)} + w_2^{(2)} a_2^{(2)}))}{\partial w_{11}^{(1)}}$$

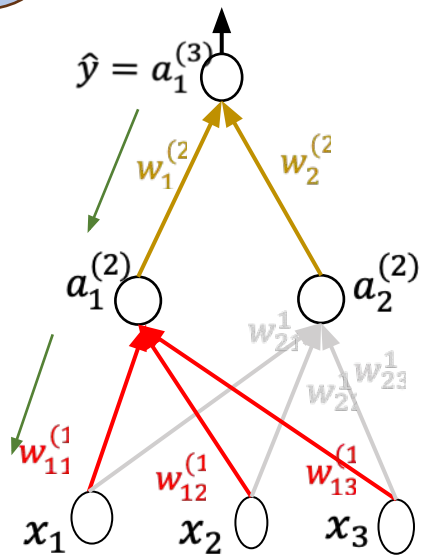
$$= 2(\hat{y} - y) a_1^{3'} \left(w_1^2 \frac{\partial \left(g(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3) \right)}{\partial w_{11}^{(1)}} + 0 \right)$$

$$= 2(\hat{y} - y) a_1^{3'} w_1^{(2)} g'(z_1) x_1$$

$$= 2(\hat{y} - y) a_1^{3'} w_1^{(2)} a_1' x_1$$

Next Layer Updates

$$\hat{y} = w_1^2 a_1 + w_2^2 a_2$$
$$a_1 = g(z_1)$$
$$z_1 = w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3$$



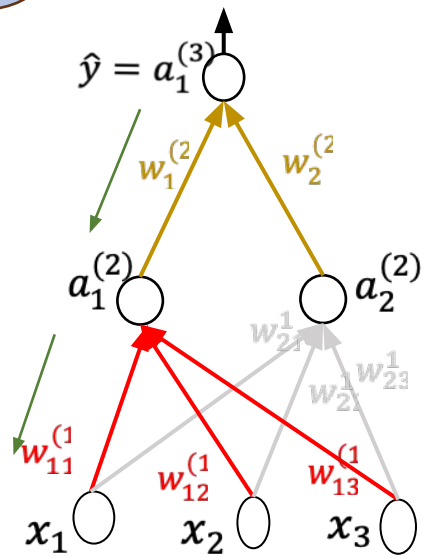
- Loss $L = (\hat{y} - y)^2$, *Compute* $\frac{\partial L}{\partial w_1^{(2)}}, \frac{\partial L}{\partial w_2^{(2)}}$

$$\frac{\partial L}{\partial w_{11}^{(1)}} = \frac{\partial (\hat{y}-y)^2}{\partial w_{11}^{(1)}} = 2(\hat{y} - y) a_1^{3'} w_1^{(2)} a_1^{(2)'} x_1$$

- *Weight Update* $w_{11}^{(1)} = w_{11}^{(1)} - \eta \frac{\partial L}{\partial w_{11}^{(1)}}$

$$w_{11}^{(1)} = w_{11}^{(1)} - \eta \delta a_1^{3'} w_1^{(2)} a_1^{(2)'} x_1$$

All Updates Together Now



Loss $L = (\hat{y} - y)^2$

- *For top layer* $\frac{\partial L}{\partial w_1^{(2)}} = \delta a_1^{(1)}, \frac{\partial L}{\partial w_2^{(2)}} = \delta a_2^{(1)}$

$$\frac{\partial L}{\partial w_i^{(2)}} = \underbrace{\delta}_{\text{Local Error}} \quad \underbrace{a_i^{(1)}}_{\text{Local Input}}$$

- *Next Layer* $\frac{\partial L}{\partial w_{ij}^{(1)}} = \delta w_i^{(2)} a_i' x_j$

Assume, $\delta_i = \delta w_i^{(2)} a_i'$. *($\delta_j$ already contains the error term)*

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = . \quad \underbrace{\delta_i}_{\text{Local Error}} \quad \underbrace{x_j}_{\text{Local Input}}$$

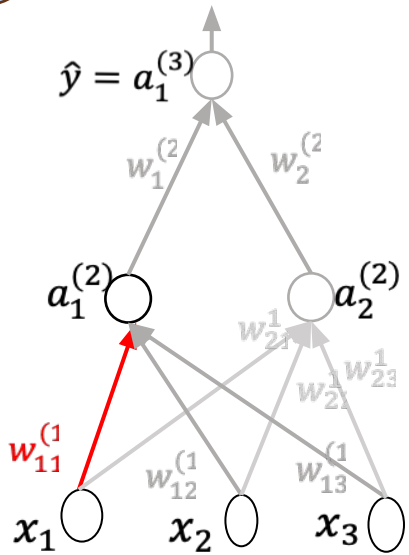
All Updates Together Now

Loss $L = (\hat{y} - y)^2$

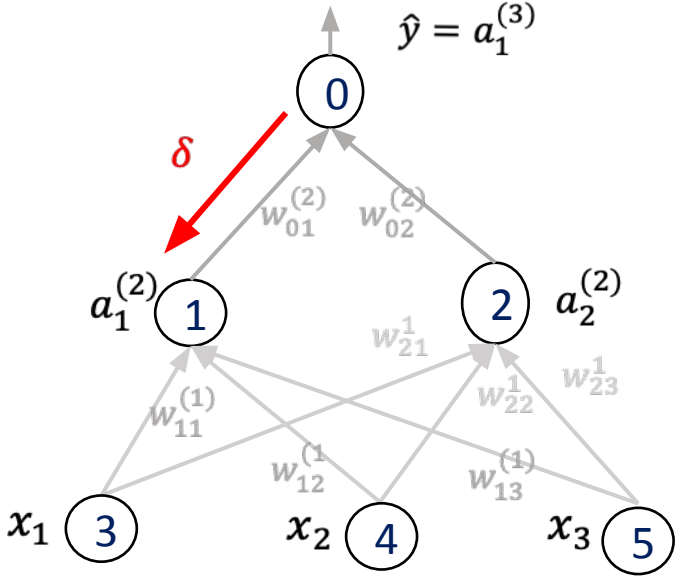
- *Let w_{ij} be the connection from node j to node i*

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l+1)} a_j^{(l)}$$

δ_i is the local error (going from i backwards) and a_j is the local input at node j.



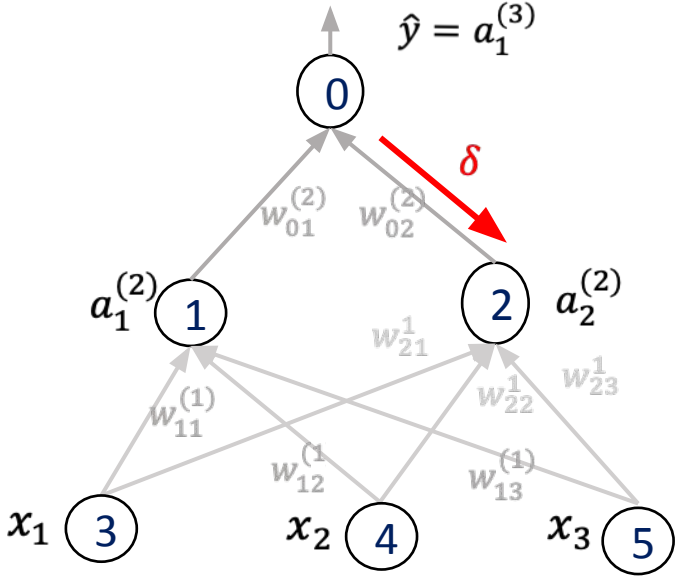
Cleaning Up: Top Layer



Local error from 0: $\delta_0 = 2(\hat{y} - y)a_1^{3'}$, *Local input from 1:* $a_1^{(1)}$

$$\frac{\partial L}{\partial w_{01}^{(l)}} = \delta_0 a_1^{(1)}$$

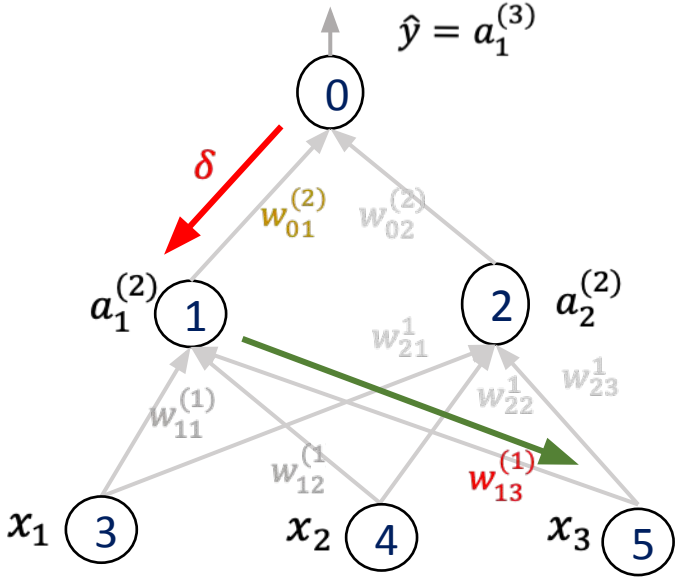
Cleaning Up: Top Layer



Local error from 0: $\delta_0 = 2(\hat{y} - y)a_1^{3'}$, *Local input from 1:* $a_2^{(1)}$

$$\frac{\partial L}{\partial w_{01}^{(2)}} = \delta_0 a_2^{(1)}$$

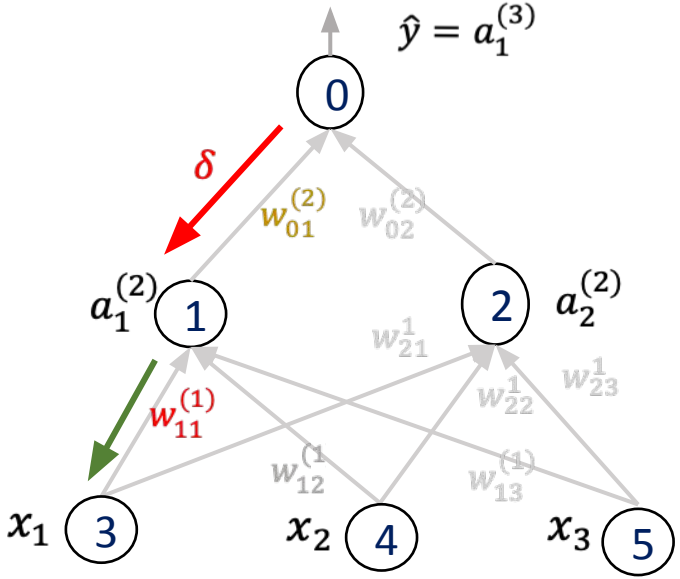
Cleaning Up: Next Layer



Local error from 1: $\delta_1 = (\delta)(w_{01}^{(2)})a_1^{(2)'}$, *Local input from 5:* x_3

$$\frac{\partial L}{\partial w_{13}^{(1)}} = \delta_1 x_3$$

Cleaning Up: Next Layer



Local error from 1: $\delta_1 = (\delta)(w_{01}^{(2)})a_1^{(2)'}$, *Local input from 3:* x_1

$$\frac{\partial L}{\partial w_{11}^{(1)}} = \delta_1 x_1$$

Lets Vectorize

- Let $W^{(2)} = \begin{bmatrix} w_{01}^{(2)} \\ w_{02}^{(2)} \end{bmatrix}$,
- Let $W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} \end{bmatrix}$
- Let $A^{(1)} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$, $A^{(2)} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$

Forward computation (Vectorized)

1. Compute $Z^{(1)} = A^{(1)^\top} W^{(1)}$
2. Applying element-wise non-linearity $A^{(2)} = g(Z^{(2)})$
3. Compute Output $Z^{(2)} = A^{(2)^\top} W^{(2)}, \ \hat{y} = g(Z^{(2)})$
4. Compute Loss on example $L = (\hat{y} - y)^2$

Backward Pass (Vectorized)

1. Top: Compute δ
2. Gradient with respect to $W^{(2)} = \delta(A^{(2)})$
3. Compute $\delta_1 = (W^{(2)^\top}\delta) \odot A^{(2)'}$ (Hadamard product)
4. Gradient w.r.t $W^{(1)} = \delta_1(A^{(1)})$
5. Update $W^{(2)} := W^{(2)} - \eta\delta \quad (A^{(2)})$
6. Update $W^{(1)} := W^{(1)} - \eta\delta_1(A^{(1)})$

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 NN 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

Training a Neural Network

- Pick a network architecture (connectivity pattern between nodes)
- # input units = # of features in dataset
- # output units = # classes
- Choose number of hidden layers and number of units in each layer.

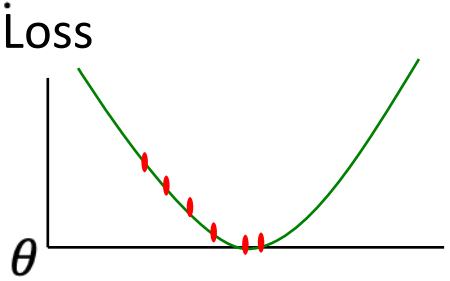
Gradient descent

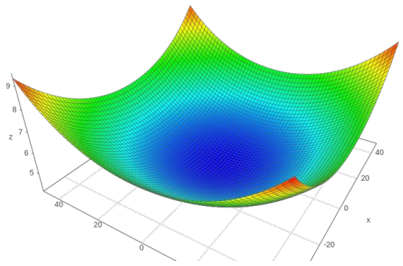
- Partial derivatives give us the slope (i.e. direction to move) in that dimension

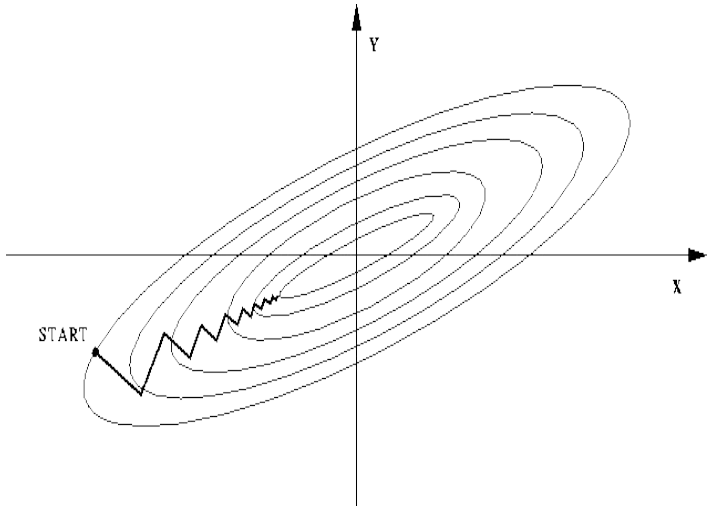
Approach:

- pick a starting point (θ)
- repeat:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$\theta_j = \theta_j - \eta \frac{d}{d\theta_j} \text{Loss}(\theta)$$







Batch, Stochastic and Minibatch

- Optimization algorithms that use the entire training set to compute the gradient are called batch or deterministic gradient methods. Ones that use a single training example for that task are called stochastic or online gradient methods
- Most of the algorithms we use for deep learning fall somewhere in between!
- These are called minibatch or minibatch stochastic methods

Batch, Stochastic and Mini-batch Stochastic Gradient Descent

Algorithm 1 Batch Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

while stopping criteria not met **do**

  Compute gradient estimate over N examples:

  $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$

  Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

end while

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

while stopping criteria not met **do**

  Sample example $(\mathbf{x}^{(i)},\mathbf{y}^{(i)})$ from training set

  Compute gradient estimate:

  $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$

  Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

end while

Mini-batch

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.

Require: Initial parameter $\boldsymbol{\theta}$

while stopping criterion not met **do**

  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

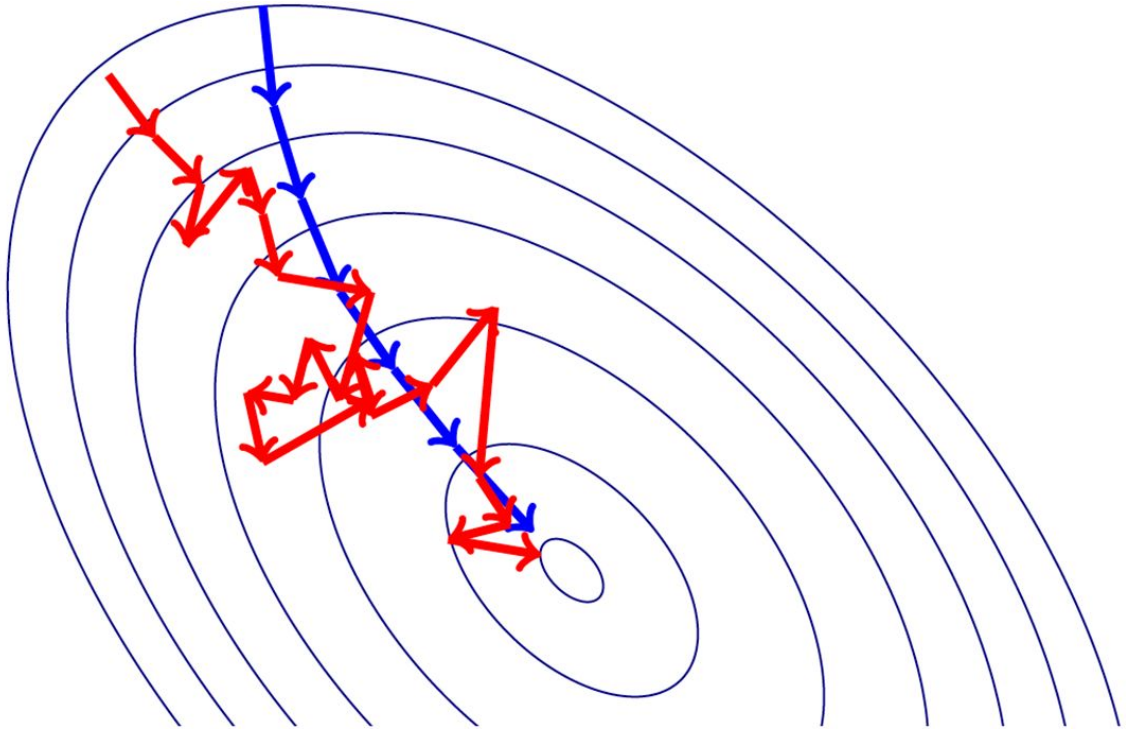
  Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)})$

  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$

end while

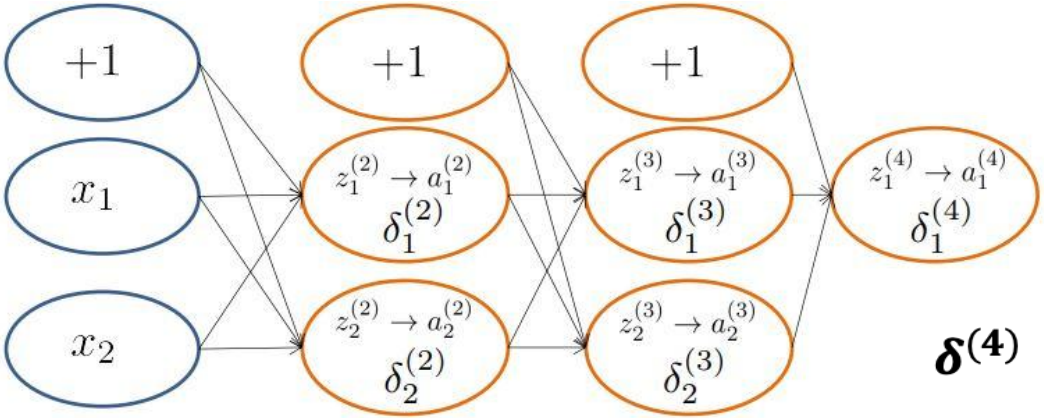
Images courtesy: Shubhendu Trivedi et. Al, Goodfellow et. al..

Batch and Stochastic Gradient Descent



Extra Slides for BackProp Visualization

Backpropagation Visualization



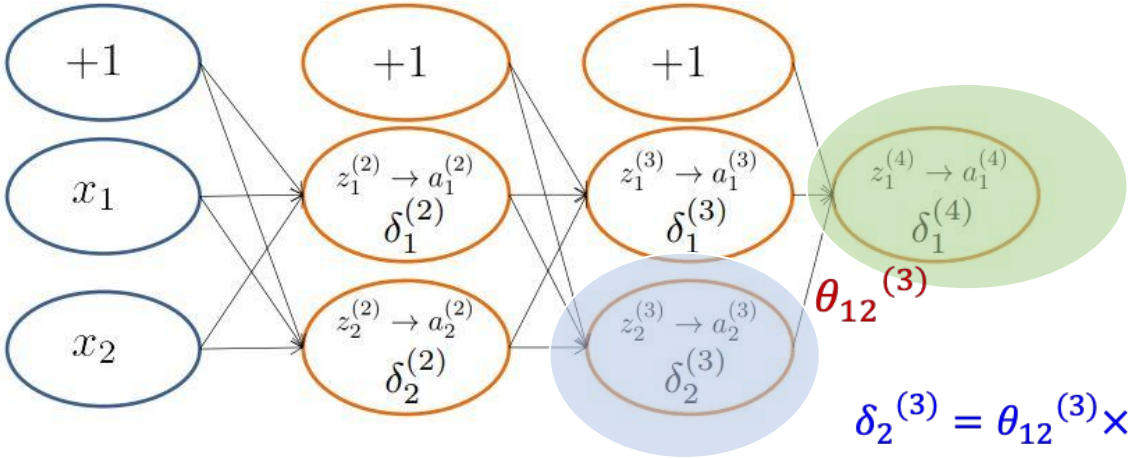
$$\boldsymbol{\delta^{(4)} = a^{(4)} - y}$$

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

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

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



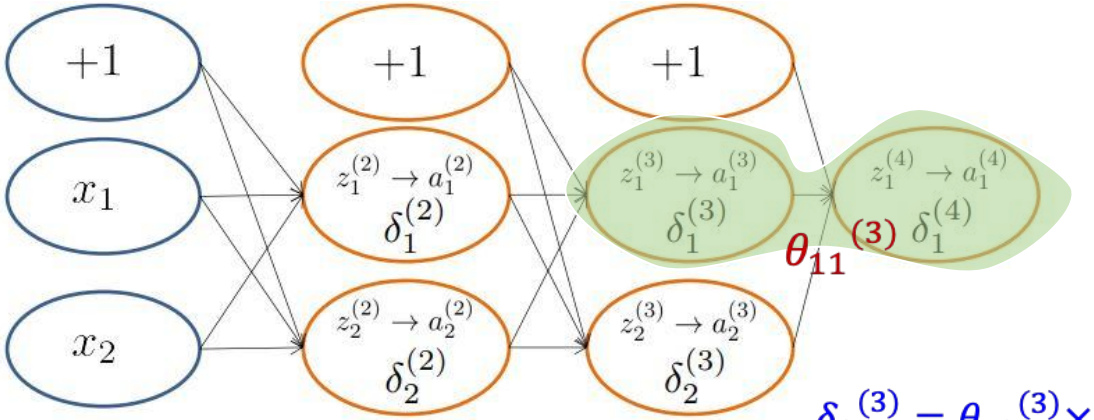
$\delta_2^{(3)} = \theta_{12}^{(3)} \times \delta_1^{(4)} \times a'^{(3)}_2$

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

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

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



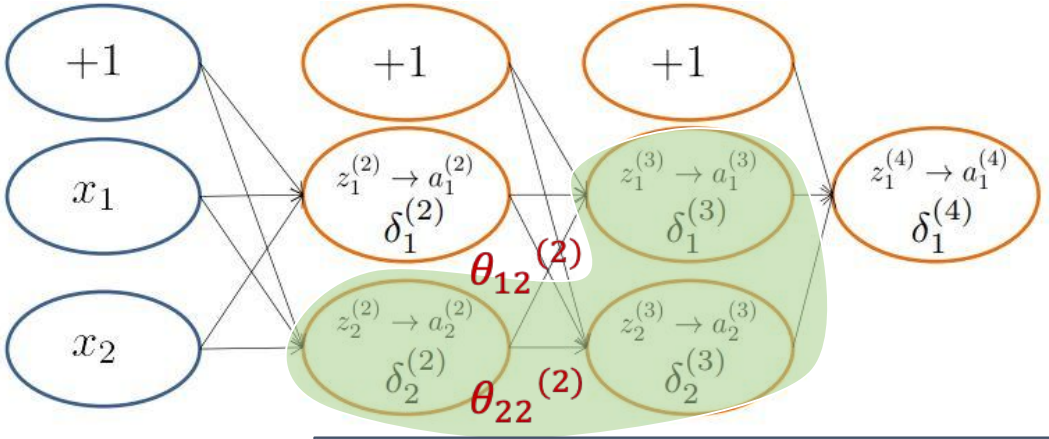
$$\delta_2^{(3)} = \theta_{12}^{(3)} \times \delta_1^{(4)} \times a'^{(3)}_2$$
$$\delta_1^{(3)} = \theta_{11}^{(3)} \times \delta_1^{(4)} \times a'^{(3)}_1$$

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

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

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



$$\delta_2^{(2)} = \theta_{12}^{(2)} \times \delta_1^{(3)} \times a'^{(2)}_2 + \theta_{22}^{(2)} \times \delta_2^{(3)} \times a'^{(2)}_2$$

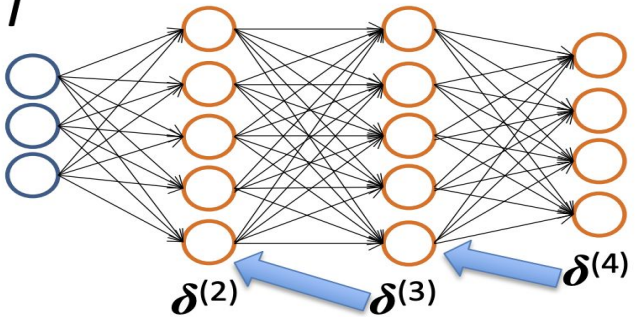
$\delta_j^{(l)}$ = "error" of node j in layer l

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

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

Gradient Computation

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



Element-wise product .*

Backpropagation

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

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

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

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

(ignoring λ; if λ = 0)