DEEP LEARNING MODELS

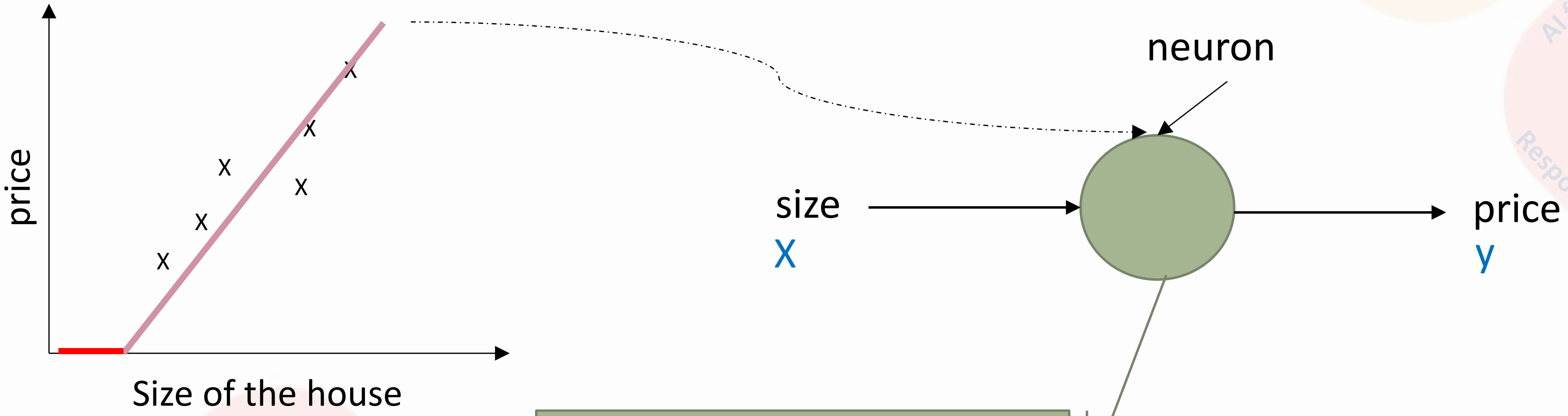
"Power of Neural Networks: Deep Learning Models Transforming Data into Intelligent Insights."

Prepared by : Bhupen

Responsible and Ethical

WHAT IS – DEEP LEARNING OR A DEEP NEURAL NETWORK(DNN)

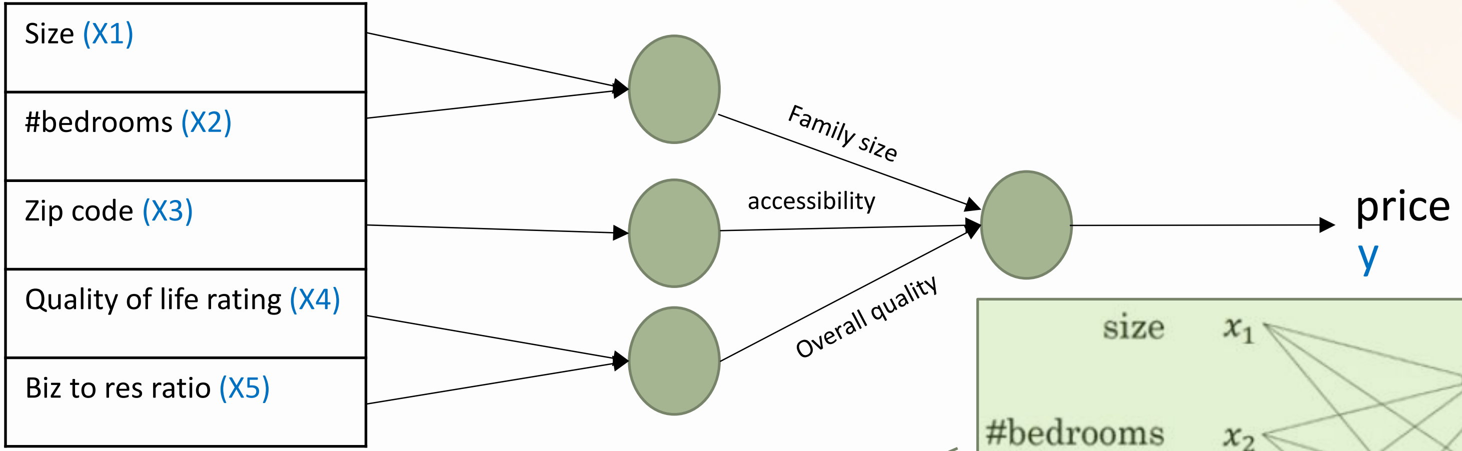
- The term, Deep Learning, refers to training Neural Networks, sometimes very large Neural Networks.
- Example – House price prediction



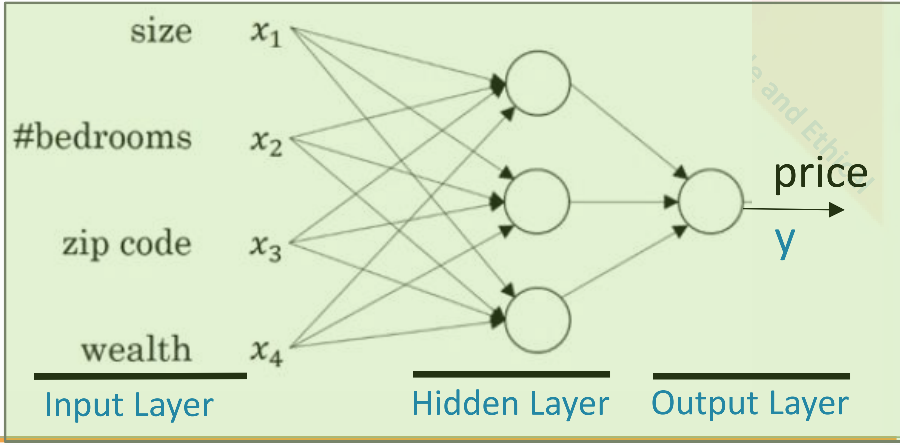
1. Single neuron NN
2. Neuron does the same thing as the regression line in the left plot
3. Single node NN can also be converted to multiple node NN (for multiple Xs)

GENERAL NN

- Let us say we have other parameters for quantifying the house price



Supervised learning problems

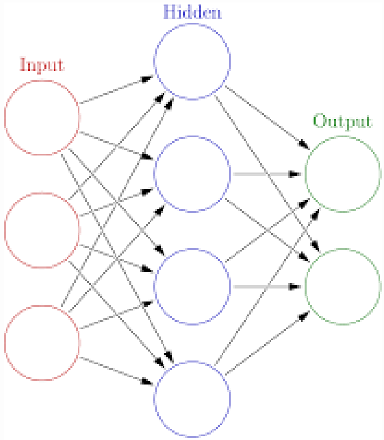


SUPERVISED LEARNING WITH NEURAL NETWORKS

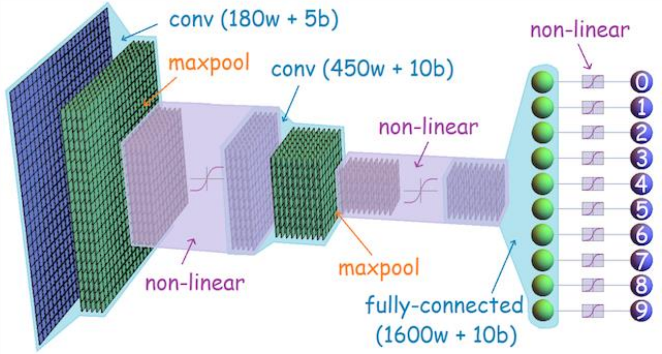
Neural networks have been a big revenue earner for supervised type of business problems …

Input(X)	Output(y)	Application	Type of neural network
Home features	Price	Real Estate	Standard NN
Adv, User profiles	Click on Adv links? (0/1)	Online/Digital	Standard NN
Image, Photo tagging	Object (?)	Computer Vision	Convolutional NN
Audio	Text output	Speech recognition	Temporal, sequence, Recurrent NN
English	Hindi	Machine Translation	Temporal, sequence, Recurrent NN
Radar, videos info	Position of other cars	Autonomous driving	Complex, custom, Hybrid NN

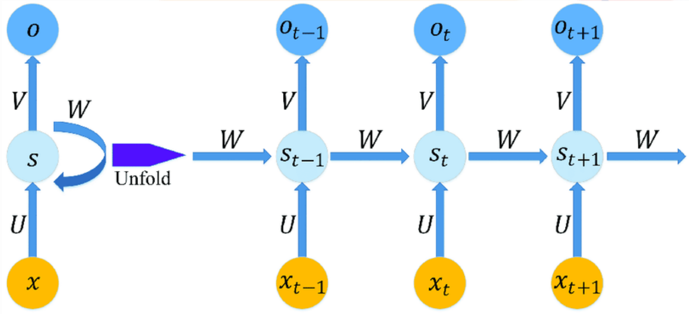
NEURAL NETWORK EXAMPLES



Standard NN



Convolutional NN



Recurrent NN

STRUCTURED VS UNSTRUCTURED DATA

Structured Data

Size	#bedrooms ..	Prices

User age	Ad ID …	Click (Y/N)

Un- structured Data

Audio

Images

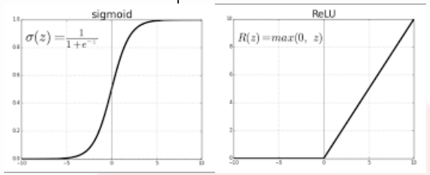
Text

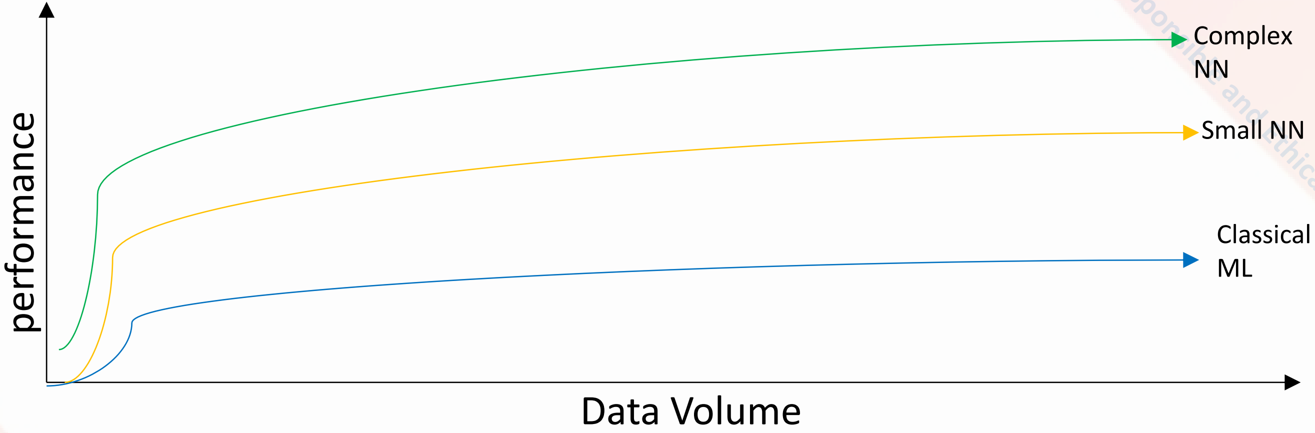
WHY DEEP LEARNING IS TAKING OFF NOW?

- Basic idea about NN and deep learning is many decades old, why is it that they are getting popular now?

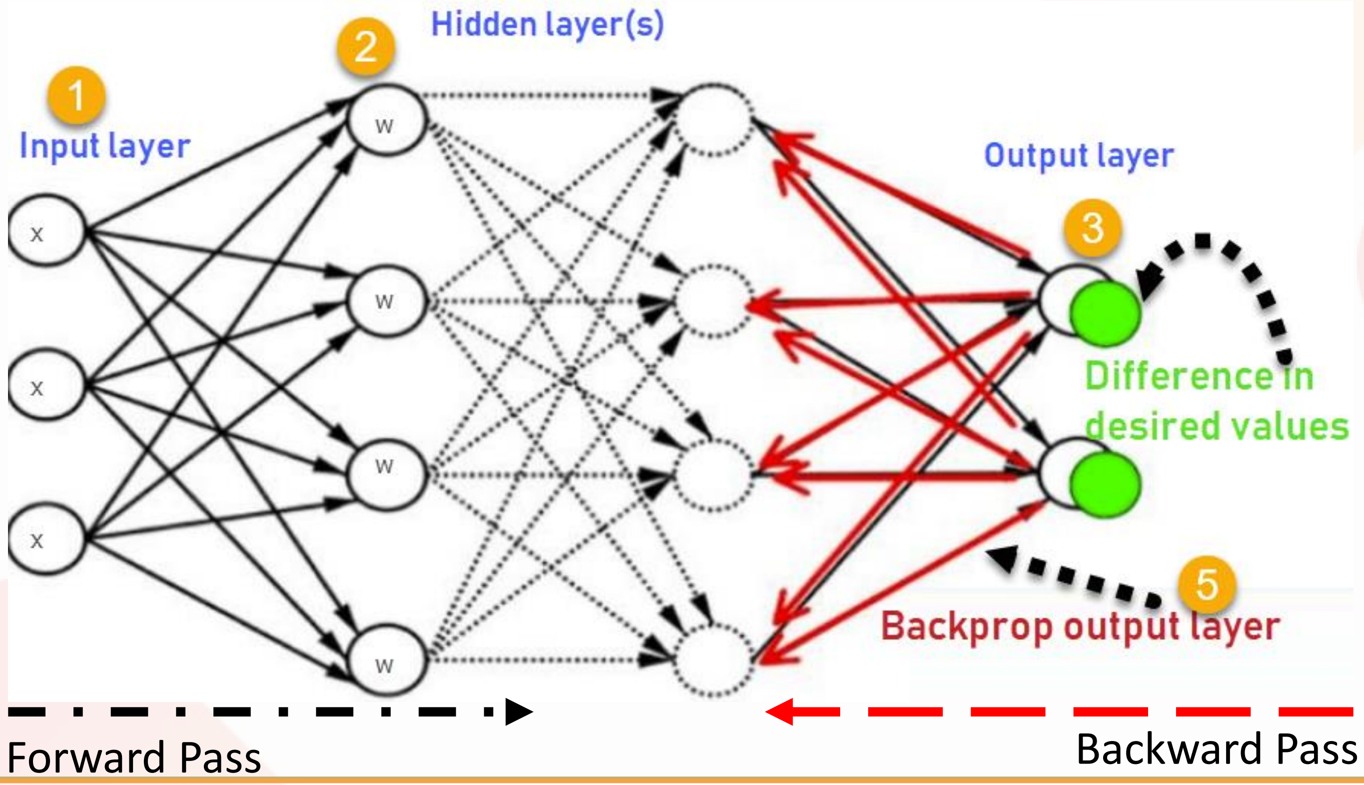
- **Reasons**
  - Computational power
  - Data availability
  - Good results for the business
  - Improvisations in algorithms

Scale drives the deep learning success

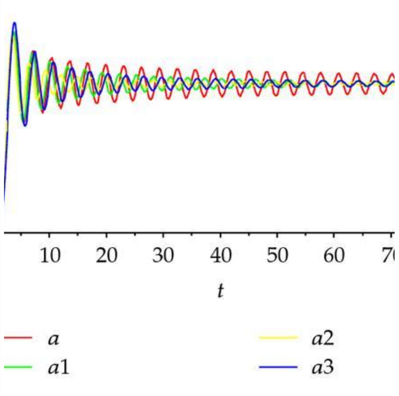


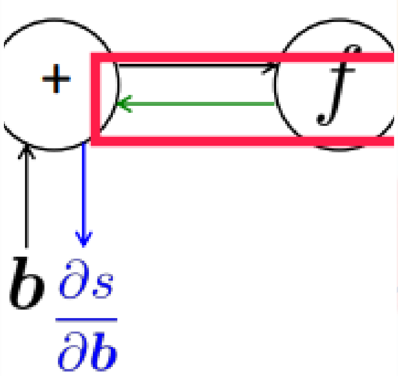


BINARY CLASSIFICATION EXAMPLE



LEARNING PROCESS IN DEEP LEARNING INVOLVES TWO MAIN PHASES





Forward Propagation: During forward propagation, the input data is passed through the layers of the neural network, and each neuron computes its output based on the learned weights and activation functions. The output from the final layer represents the prediction or classification result.

Backpropagation: Backpropagation is the process of updating the model's weights to minimize the difference between predicted outputs and actual labels. It uses an optimization algorithm (e.g., gradient descent) to adjust the weights in the direction that reduces the prediction errors.

KEY POINTS OF NN PROGRAMMING

- Supervised binary classification

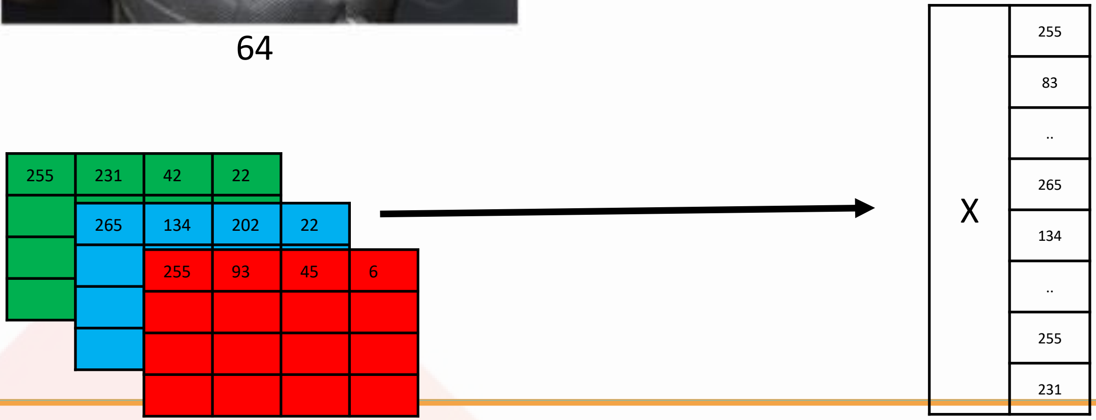
X0	X1	X2	..	y
..	..	..	..	..
..	..	..	..	..
..	..	..	..	..
..	..	..	..	..

- Classical machine learning – we solve by looping over all the training samples
- In NN,
  - the idea is to create vectors of all the samples and perform **vectorized operations** in one go
  - The learning is organized in 2 passes
    - Forward pass
    - Back propagation

EXAMPLE PROBLEM – DETECT IF THE IMAGE IS OF ‘CAT’

64

64



y
1 (cat)
0 (non-cat)

64x 64 x 3 = 12288

n_x = 12288

NOTATION

- (X, y) -> Set of input and response variables
- X is set of n_x and y = {0 , 1}
- m – training examples : { $(X^{(1)}, y^{(1)})$, $(X^{(2)}, y^{(2)})$, $(X^{(3)}, y^{(3)})$, … $(X^{(m)}, y^{(m)})$ }

$$X = \underbrace{\begin{bmatrix} | & | & \cdots & | \\ | & | & \cdots & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & \cdots & | \\ | & | & \cdots & | \end{bmatrix}}_{m \text{ rows}} \Bigg\} n_x \text{ columns}$$

X.shape will give an output of (n_x, m)

$$y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & y^{(4)} & \ldots & y^{(n-1)} & y^{(n)} \end{bmatrix}$$

y.shape will give an output of (1, m)

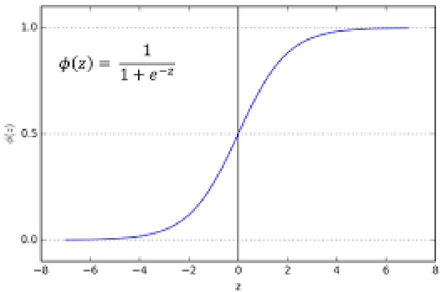
LOGISTIC REGRESSION

- Given X, we want y-pred = P (y = 1|x)

- X is a set of $\{n_x\}$
- Parameters for the algorithm: w – a set of $\{n_x\}$, and b is a real number (intercept)
- Output : y_pred = $w^T x + b$

- this is what we use for linear regression
- But this isn't the solution for logistic regression as we want output in {0, 1}
- $w^T x + b$ returns a real number (less than 0 or greater than 1)

- In case of Logistic regression we will have
  - Output : y_pred = sigmoid $(\underbrace{w^T x + b}_{z})$



If z is large, σ(z) = 1
If z is large negative, σ(z) = 0

LOGISTIC REGRESSION COST FUNCTION

- $\{ (X^{(1)}, y^{(1)}), (X^{(2)}, y^{(2)}), (X^{(3)}, y^{(3)}), \ldots (X^{(m)}, y^{(m)}) \}$ – we want $y\_pred^{(i)}$ same as $y^{(i)}$
- $y\_pred^{(i)} = \sigma\ (w^T x^{(i)} + b) = \sigma(z^{(i)})$
- Loss function (error)
  - $L\ (y\_pred, y) = -(y.\ \log y\_pred + (1 - y) \log (1 - y\_pred))$

Loss function – over a single sample
Cost function – over entire training set

$$J(w, b) = \frac{1}{M} \sum_1^m L(\hat{y}^{(i)} - y^{(i)})$$

$$J(w, b) = -\frac{1}{M} \sum_1^m [(y^{(i)}. \log\hat{y}^{(i)} + (1 - y^{(i)}). \log(1 - \hat{y}^{(i)})]$$

Objective is to have min loss

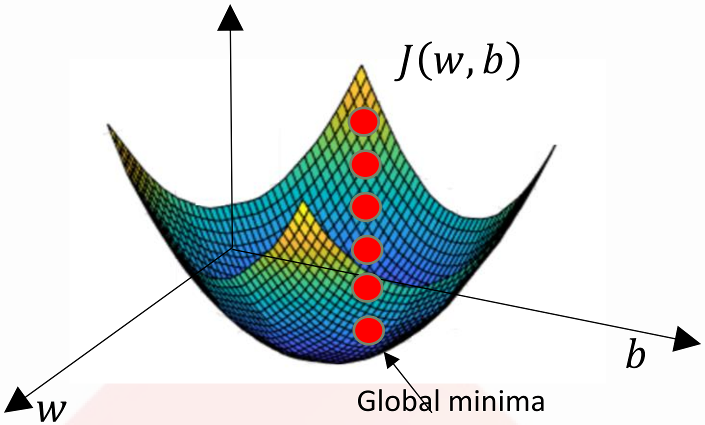
If y = 1	If y = 0
L (y_pred, y) = - log y_pred	L (y_pred, y) = – log (1 – y_pred)
We want log y_pred to be large, mean y_pred be large	We want log (1 - y_pred) to be small, y_pred to be small

GRADIENT DESCENT

We use GRADIENT DESCENT to find the optimum values for w, b that minimizes J(w, b)

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})$$



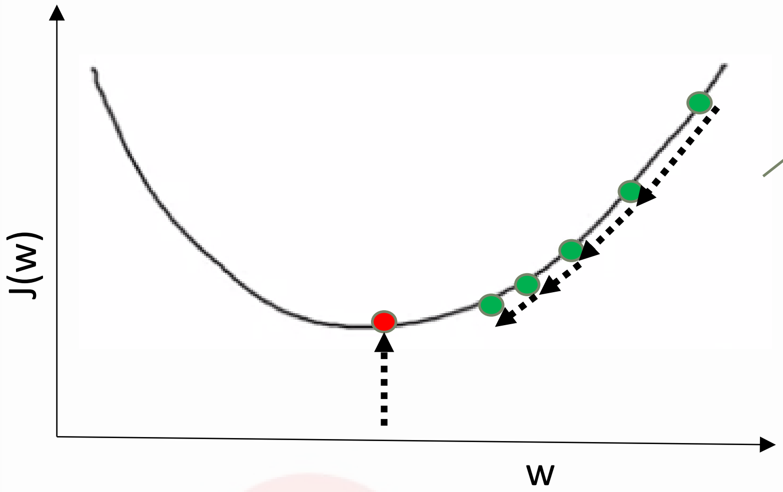
Goal : want to find w, b that minimize the cost function J(w, b)

Approach

- Start with any real values for w, b
  - Initialize with 0 or random numbers
- Take the derivative at that point (w, b)
- Update the weights

SIMPLIFYING..

- We will take only one parameter w, (not b)
- Learning rate of α



Repeat {

- Calculate derivative at w
- Update w

$$w := w - \alpha \frac{dJ(w)}{dw}$$

- OR

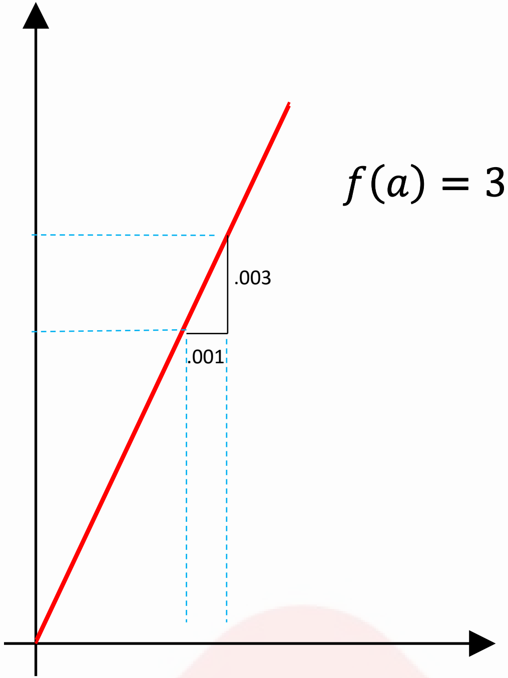
$$w := w - \alpha . dw$$

}

$$w := w - \alpha \frac{dJ(w,b)}{dw}$$

$$b := b - \alpha \frac{dJ(w,b)}{db}$$

DERIVATIVES – STRAIGHT LINE



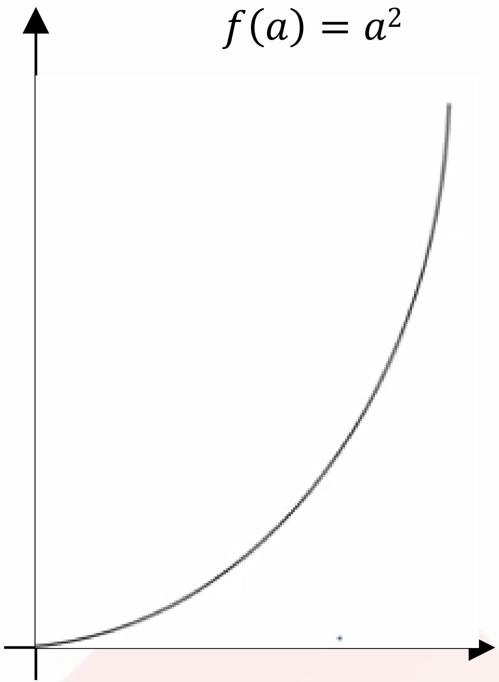
$$f(a) = 3a$$

At a = 2	At a = 5
a = 2 f(a) = 6 At slight increment of a = 2.001 f(a) = 6.003 Slope = change in f(a)/change in a = .003 /.001 = 3	a = 5 f(a) = 15 At slight increment of a = 5.001 f(a) = 15.003 Slope = change in f(a)/change in a = .003 /.001 = 3

Derivatives is SAME as slope of a function

$$\frac{df(a)}{da} = 3$$

DERIVATIVE - MORE EXAMPLES



At a = 2	At a = 5
a = 2 f(a) = 4 At slight increment of a = 2.001 f(a) = 4.004001 Slope = change in f(a)/change in a = .004 /.001 = 4	a = 5 f(a) = 25 At slight increment of a = 5.001 f(a) = 25.010 Slope = change in f(a)/change in a = .010 /.001 = 10

Derivatives is different at different points

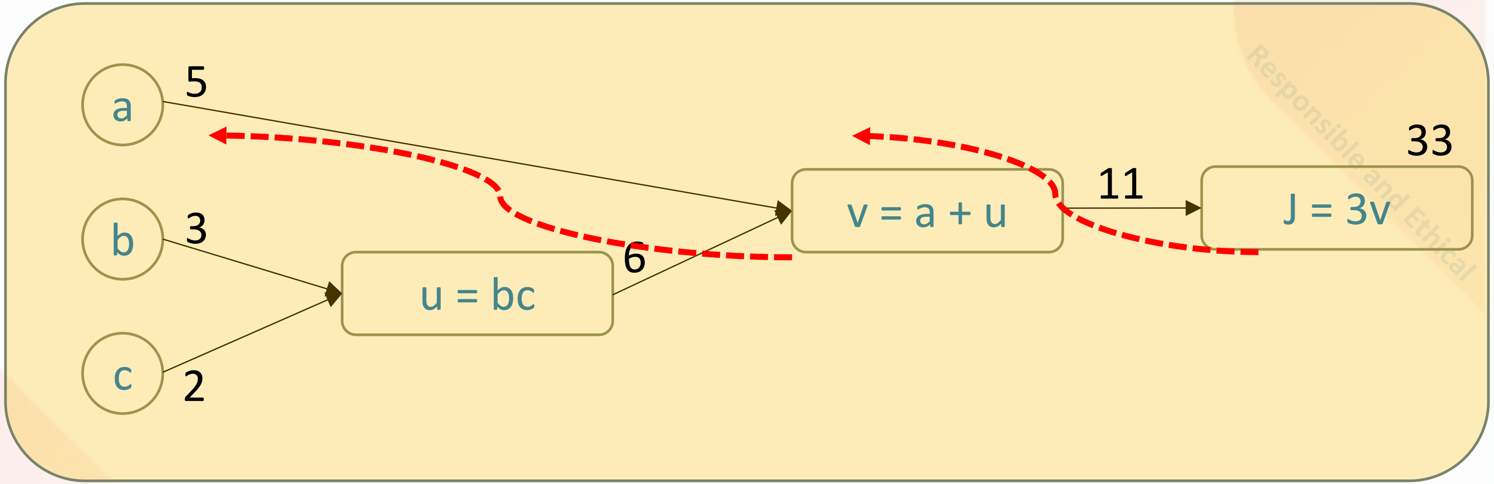
COMMON DERIVATIVES

Common Functions	Function	Derivative
Constant	c	0
Line	x	1
	ax	a
Square	x^2	$2x$
Square Root	$\sqrt{x}$	$(½)x^{-½}$
Exponential	e^x	e^x
	a^x	$\ln(a)\ a^x$
Logarithms	$\ln(x)$	$1/x$
	$\log_a(x)$	$1 / (x \ln(a))$
Trigonometry (x is in radians)	$\sin(x)$	$\cos(x)$
	$\cos(x)$	$-\sin(x)$
	$\tan(x)$	$\sec^2(x)$
Inverse Trigonometry	$\sin^{-1}(x)$	$1/\sqrt{(1-x^2)}$
	$\cos^{-1}(x)$	$-1/\sqrt{(1-x^2)}$
	$\tan^{-1}(x)$	$1/(1+x^2)$

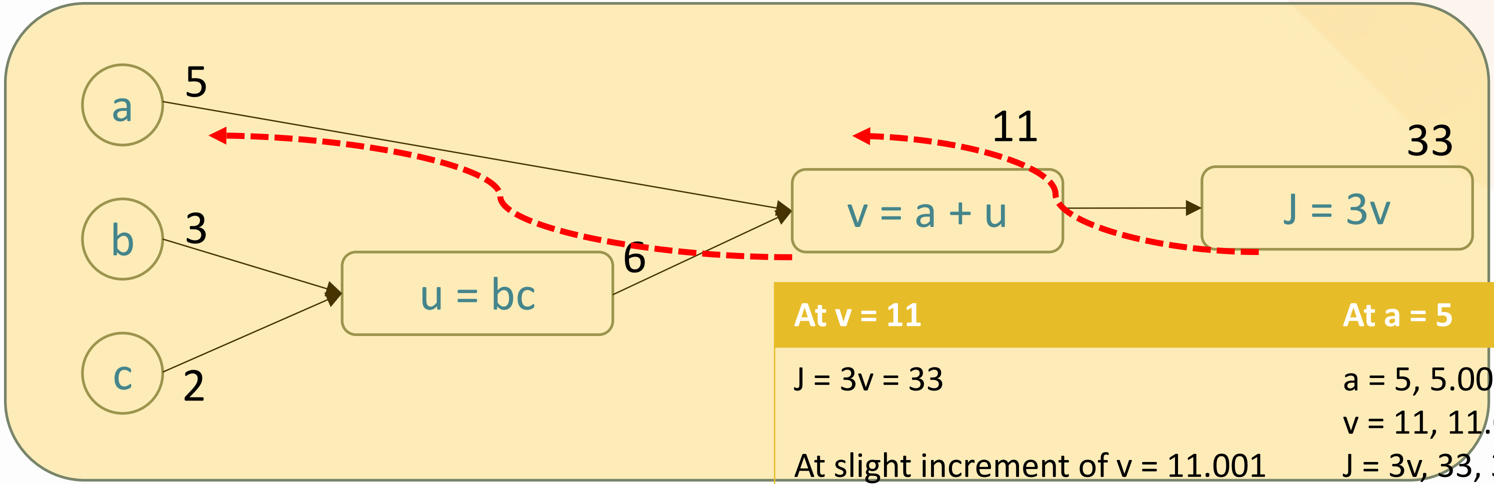
COMPUTATION GRAPHS

- J (a, b, c) = 3 (a + bc)
  - where we can say u = bc and v = a + bc and that multiplied by 3 = J
  - u = bc
  - u = a + u
  - J = 3v

- NN has forward pass and back propagation
- Computation graph makes it easier to understand



DERIVATIVES WITH COMPUTATION GRAPHS



At v = 11	At a = 5
J = 3v = 33	a = 5, 5.001 v = 11, 11.001
At slight increment of v = 11.001	J = 3v, 33, 33.003
J = 33.003	Slope = delta(J)/delta(a) = 3
Slope = change in f(a)/change in a = .003 /.001 = 3	$\frac{dJ}{da} = 3$
$\frac{dJ}{dv} = 3$	

Since a -> v -> J

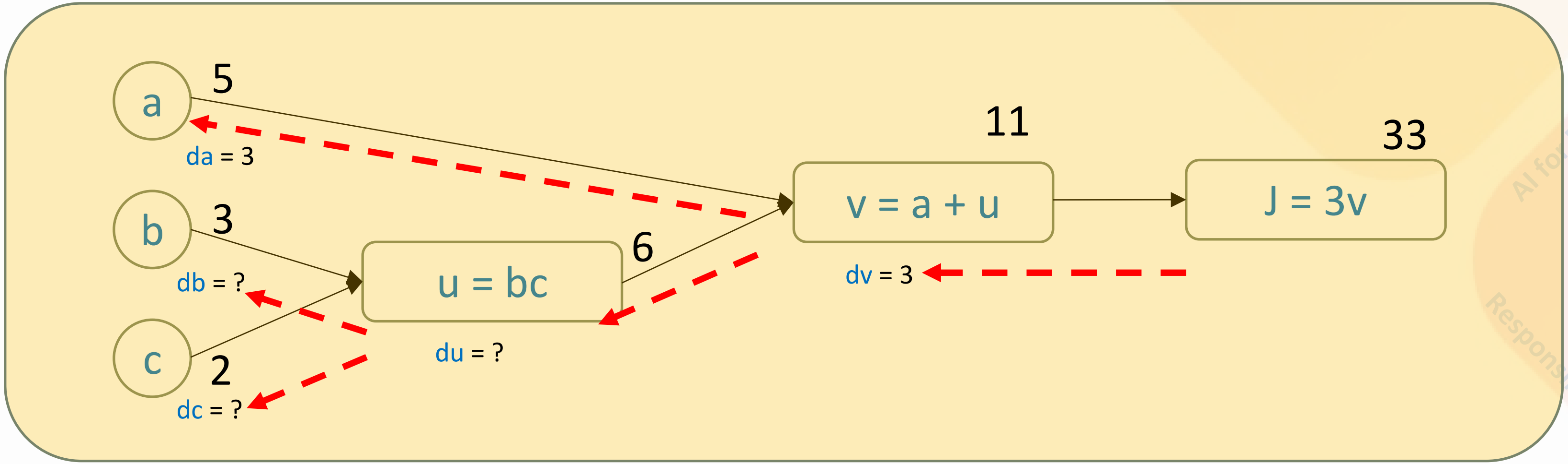
$$\frac{dJ}{da} = \frac{dj}{dv} \cdot \frac{dv}{da}$$

Chain rule

$\frac{dJ}{da}$ can be written as da

Derivative of final output variable wrt ‘a’

DERIVATIVES WITH COMPUTATION GRAPHS



At b = 3, 3.001

u = 6, 6.002
v = 11, 11.002
J = 33, 33.006
Slope = change in J/change in u
= .006 /.001 = 6

- b -> u -> v -> J [$\frac{dJ}{db} = \frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{db}$]

At u = 6, 6.001

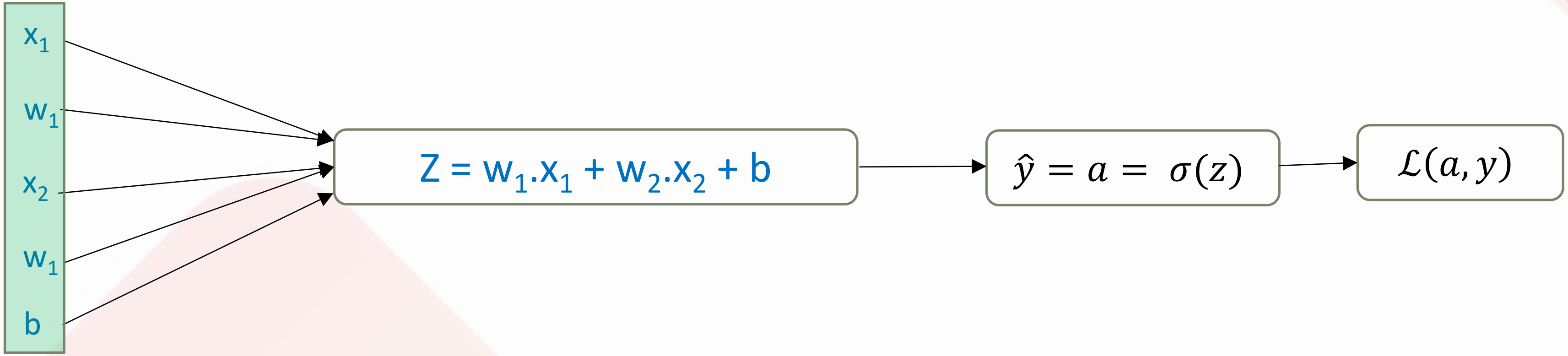
v = 11, 11.001
J = 33, 33.003
Slope = change in J/change in u
= .003 /.001 = 3

- u -> v -> J [$\frac{dJ}{du} = \frac{dJ}{dv} \cdot \frac{dv}{da}$]

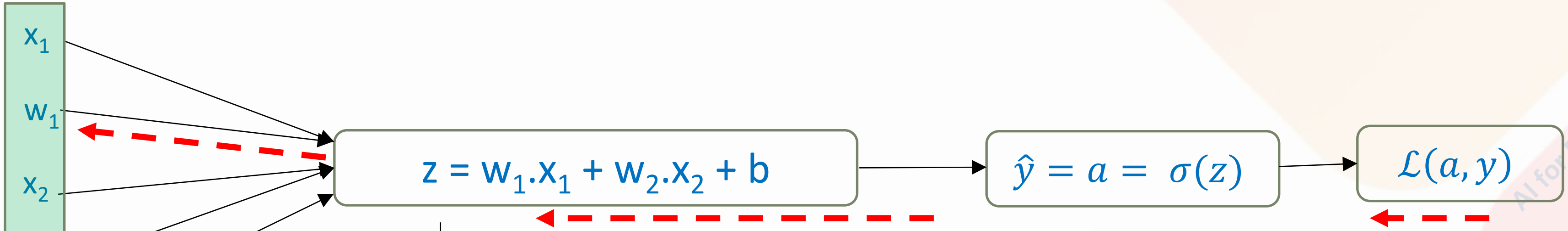
GRADIENT DESCENT – LOGISTIC REGRESSION

FOR A SINGLE TRAINING EXAMPLE	
$z = w^T x + b$	Node computation
$\hat{y} = a = \sigma(z)$	Predicted value
$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$	Loss function

Computation graph : Assume we have 2 input variables, X1, and X2



GRADIENT DESCENT – LOGISTIC REGRESSION



$w_1 -> z -> a -> \mathscr{L}$

$$dw_1 = \frac{d\mathscr{L}}{w_1} = \frac{d\mathscr{L}}{da} . \frac{da}{dz} . \frac{dz}{dw_1}$$

$$= (a - y). \frac{dz}{dw_1} = dz_1 . x_1$$

$$dz = \frac{d\mathscr{L}}{dz}$$

$$= \frac{d}{dz}[-(ylog[(a) + (1 - y)log(1 - a))]$$

$z -> a -> \mathscr{L}$

$$\frac{d\mathscr{L}}{dz} = \frac{d\mathscr{L}}{da} . \frac{da}{dz}$$

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

$$da = \frac{d\mathscr{L}}{da}$$

$$= \frac{d}{da}[-(ylog[(a) + (1 - y)log(1 - a))]$$

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

Derivatives – with respect to single example

$dw_1 = x_1. dz$	$dw_2 = x_2.dz$	$db = dz$
$w_1 := w_1 - \alpha. dw_1$	$w_2 := w_2 - \alpha. dw_2$	$b : = b - \alpha. db$

GRADIENT DESCENT – LOGISTIC REGRESSION – MULTIPLE EXAMPLES

Recap :

$\hat{y} = \sigma(w^T x + b), \; \sigma(z) = \frac{1}{1+e^{-z}}$

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

… we need the following for each sample …. $dw_1^{(i)}$, $dw_2^{(i)}$, $db^{(i)}$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

LOGISTICS REGRESSION ON M EXAMPLES

$j = 0; dw_1 = 0; dw_2 = 0; db = 0$

for I = 1 to m examples in the training set:

$z^{(i)} = w^T.x^{(i)} + b$

$a^{(i)} = \sigma\,(z^{(i)})$

$j \mathrel{+}= -\,[\, y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})\,]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1 dz^{(i)}$

$dw_2 \mathrel{+}= x_2 dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J \mathrel{/}= m; dw_1 \mathrel{/}= m; dw_2 \mathrel{/}= m; db \mathrel{/}= m$

$w_1 := w_1 - \alpha.\, dw_1$
$w_2 := w_2 - \alpha.\, dw_2$
$b := b - \alpha.\, db$

With DNN and large datasets, loop method will cause performance issue

Solution is to use **vectorization**

VECTORIZATION

- Neural network programming
  - Whenever possible, avoid explicit for-loops

$j = 0$; $dw_1 =$; $dw_2 = 0$; $db = 0$

for I = 1 to m examples in the training set:

$z^{(i)} = w^T.x^{(i)} + b$

$a^{(i)} = \sigma\ (z^{(i)})$

$j \mathrel{+}= -\ [\ y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})\ ]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1 dz^{(i)}$

$dw_2 \mathrel{+}= x_2 dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J \mathrel{/}= m$; $dw_1 \mathrel{/}= m$; $dw_2 \mathrel{/}= m$; $db \mathrel{/}= m$

$dw = np.zeros(n_x, 1)$

$dw \mathrel{+}= x^{(i)} * dz^{(i)}$

$dw \mathrel{+}= x * dz$

VECTORIZING LOGISTIC REGRESSION – FORWARD PROPAGATION

1st Training sample

$z^{(1)} = \underline{w^T x^{(1)} + b}$

$a^{(1)} = \sigma(z^{(1)})$

2nd Training sample

$z^{(2)} = \underline{w^T x^{(2)} + b}$

$a^{(2)} = \sigma(z^{(2)})$

3rd Training sample

$z^{(3)} = \underline{w^T x^{(3)} + b}$

$a^{(3)} = \sigma(z^{(3)})$

… repeat for m samples

Using vectorized operations

1

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

rows stacked in columns (n_x, m)

2

$Z = [\underbrace{z^{(1)}\ z^{(2)} \dots z^{(m)}}_{\text{1xm row matrix}}] = w^T . X + [\underbrace{b\ b \dots b}_{\text{1xm row matrix}}]$

3

$[w^T.\underline{x^{(1)} + b}\ \ \underline{w^T x^{(2)} + b} \dots \underline{w^T.x^{(m)} + b}]$

4

Z = np.dot(w.T, X) + b

5

$A = [a^{(1)}\ a^{(2)} \dots a^{(m)}] = \sigma(z)$

VECTORIZING LOGISTIC REGRESSION GRADIENT OUTPUT – BACK PASS

For each of the sample $dz^{(1)} = a^{(1)} - y^{(1)}$ $dz^{(2)} = a^{(2)} - y^{(2)}$ … for all m training samples

Applying vectorization …

$$dZ = [dz^{(1)}\ dz^{(2)}\ \dots\ dz^{(m)}] \qquad A = [a^{(1)}\ a^{(2)}\ \dots\ a^{(m)}]$$

$$= A - Y \qquad Y = [y^{(1)}\ y^{(2)}\ \dots\ y^{(m)}]$$

Updating weights … loop method

$dw = 0$ $db = 0$

$dw \mathrel{+}= X^{(1)}\, dz^{(1)}$ $db \mathrel{+}= dz^{(1)}$

$dw \mathrel{+}= X^{(2)}\, dz^{(2)}$ $db \mathrel{+}= dz^{(2)}$

.. ..

.. ..

$dw\ /= m$ $db\ /= m$

Applying vectorization …

$dw = (1/m)\ X.dZ^T$ $db = (1/m)\ np.sum(dZ)$

$$X = \underbrace{\begin{bmatrix} | & | & & | \\ X^{(1)} & X^{(2)} & & X^{(m)} \\ \dots & \dots & \dots & \dots \\ | & | & & | \end{bmatrix}}_{(n_x,\ m)} \underbrace{\begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \\ dz^{(m)} \end{bmatrix}}_{(m,\ 1)}$$

PUTTING IT ALL TOGETHER …

Updating weights … loop method

$j = 0$; $dw_1 = $; $dw_2 = 0$; $db = 0$

for I = 1 to m examples in the training set:

$z^{(i)} = w^T.x^{(i)} + b$

$a^{(i)} = \sigma\,(z^{(i)})$

$j \mathrel{+}= -\,[\,y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})\,]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1 dz^{(i)}$

$dw_2 \mathrel{+}= x_2 dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J \mathrel{/}= m$; $dw_1 \mathrel{/}= m$; $dw_2 \mathrel{/}= m$; $db \mathrel{/}= m$

Updating weights … vectorization

$Z = [z^{(1)}\, z^{(2)} \ldots z^{(m)}] = w^T.\,X + [b\ b \ldots b]$ = np.dot(w.T, X) + b

$A = [a^{(1)}\, a^{(2)} \ldots a^{(m)}] = \sigma(z)$

$dZ = A - Y$

$dw = (1/m)\, X.dZ^T$

db = (1/m) np.sum(dZ)

w := w – (learning rate) * dw

b := b – (learning rate) * db

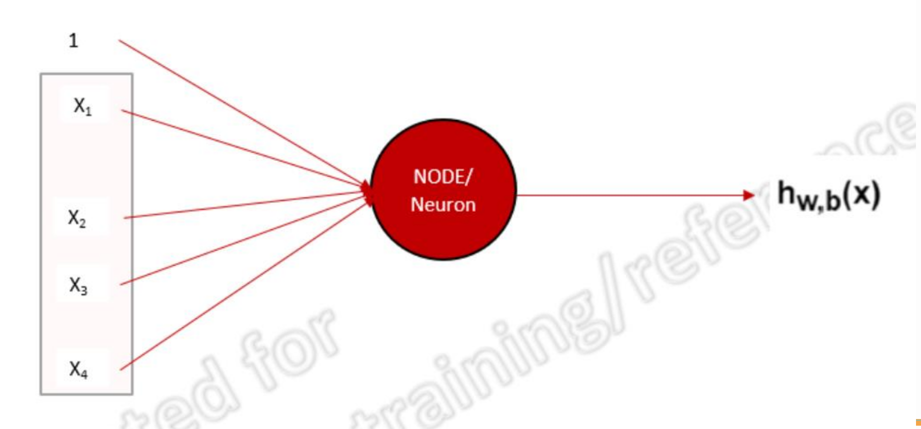
NEURAL NETWORK

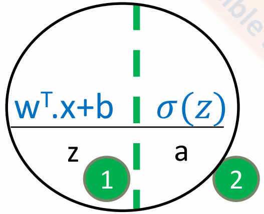
ELEMENTS OF A NEURAL NETWORK :-

Input Layer	Hidden Layer	Output Layer
• accepts input features. • provides information from the outside world to the network, • no computation is performed at this layer, • nodes here just pass on the information(features) to the hidden layer.	• Nodes are not exposed to the outer world, • part of the abstraction • Hidden layer performs computation on the features entered through the input layer and transfer the result to the output layer.	• brings up the information learned by the network to the outer world. A common activation function that is used is the sigmoid function: $$f(z) = \frac{1}{1 + e^{-z}}$$

NODES

- Biological neurons are connected hierarchical networks, with the outputs of some neurons being the inputs to others.
- We can represent these networks as connected layers of nodes.
- Each node takes multiple weighted inputs, applies the activation function to the summation of these inputs, and in doing so generates an output.



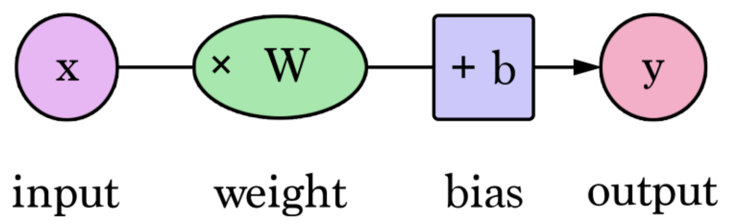


$$z = w^T x + b$$

$$a = \sigma(z)$$

WEIGHTS

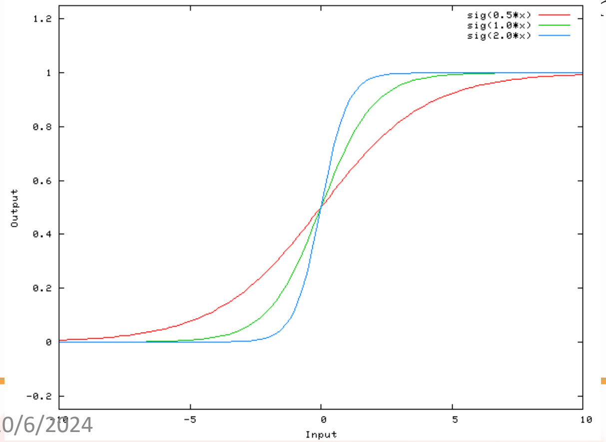
- Weight is the parameter within a neural network that transforms input data within the network's hidden layers.
- Within each node is a set of inputs, weight, and a bias value.
- As an input enters the node, it gets multiplied by a weight value and the resulting output is either observed, or passed to the next layer in the neural network.
- learnable parameter



BIAS

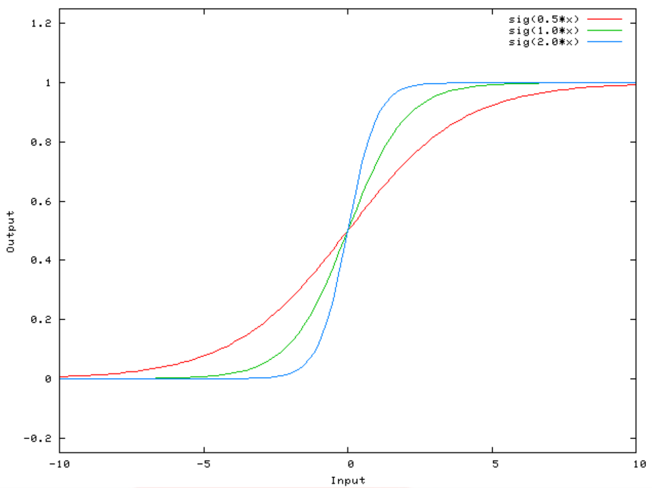
- a bias value allows you to shift the activation function to the left or right, which may be critical for successful learning.
- Consider this 1-input, 1-output network that has no bias:

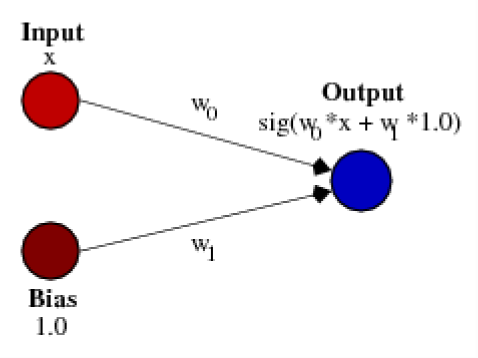
- The output of the network is computed by multiplying the input (x) by the weight (w0) and passing the result through some kind of activation function (e.g. a sigmoid function.)

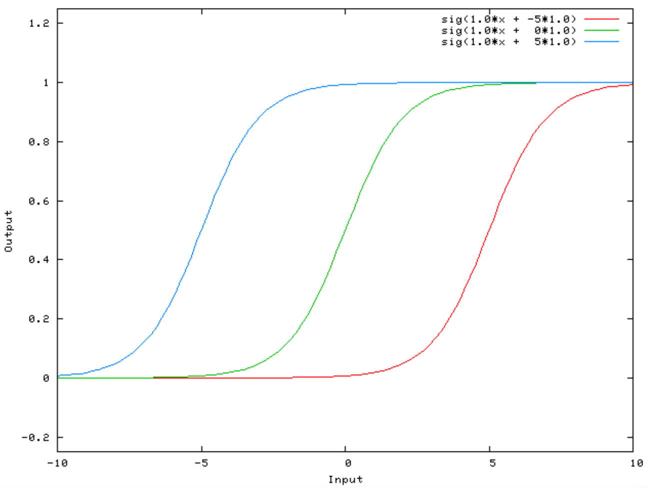


BIAS

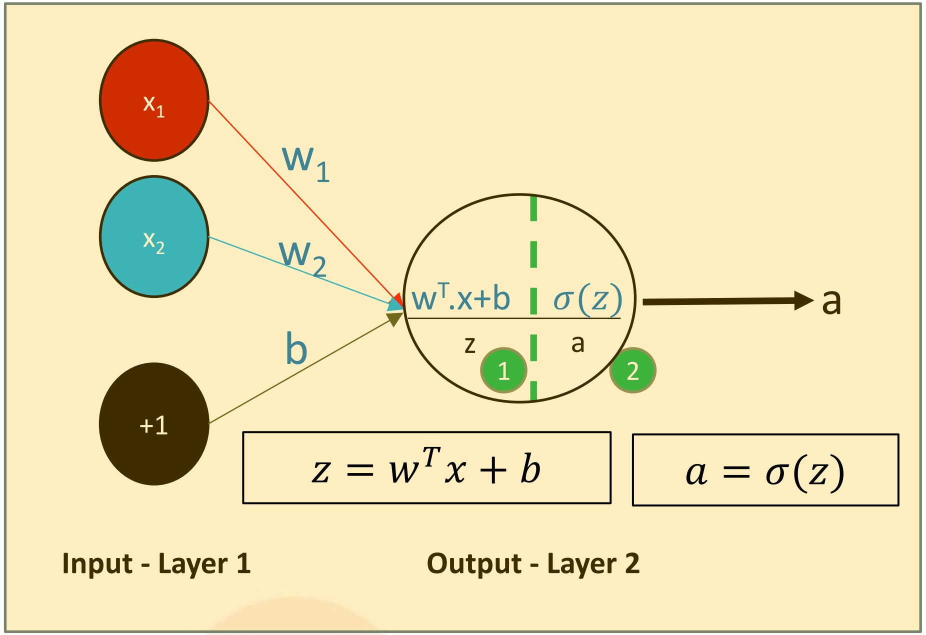
- Changing the weight w0 essentially changes the "steepness" of the sigmoid.
- what if you wanted the network to output 0 when x is 2? -- you want to be able to shift the entire curve to the right.







PERCEPTRON




```
X
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```
y = np.array([0, 0, 0, 1])
```

```
# display the weights vector
perceptron.W
```

```
array([0., 0.])
```

```
# display other initiated values
print(perceptron.lr)
print(perceptron.W)
print(perceptron.bias)
```

```
1
[0. 0.]
0
```

- most fundamental building block of deep neural networks: the neuron.
- combine several of them into a layer and create a neural network called the perceptron/ multi level perceptron or deep learning networks.

PERCEPTRON – FIT (TRAINING)

$$z = w^T x + b$$

```
print(perceptron.epochs)
```

```
100
```

```
X
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

i

```
# display other initiated values
print(perceptron.lr)
print(perceptron.W)
print(perceptron.bias)
```

```
1
[0. 0.]
0
```

array([0., 0.]) dot [0, 0] + 0

z

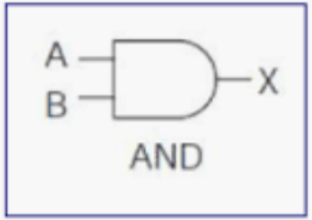
$$a = \sigma(z)$$

if z >= 0 return 1 else 0

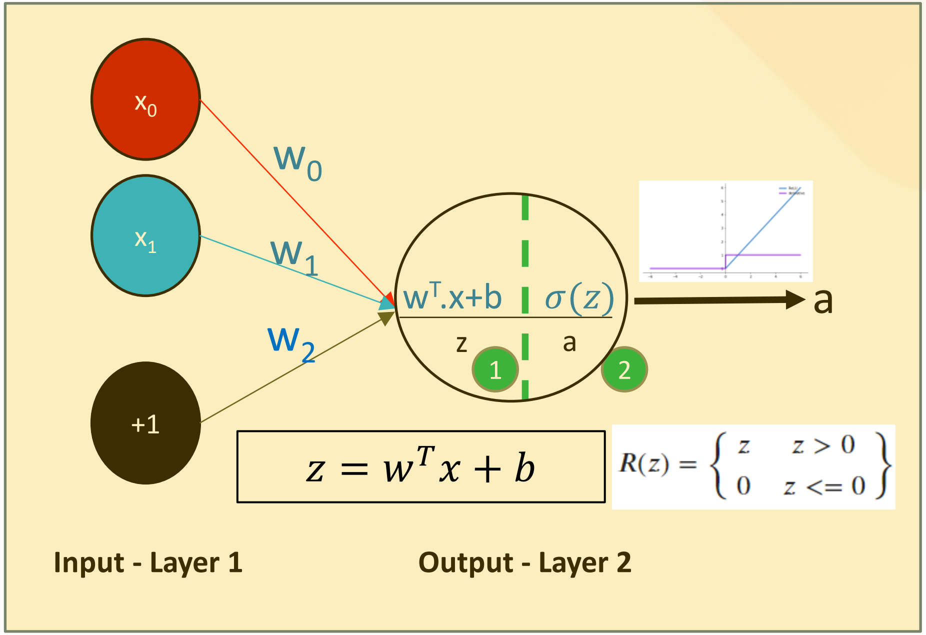
e = y[i] - a

w = w + lr * e * X[i]
bias = bias + lr * e

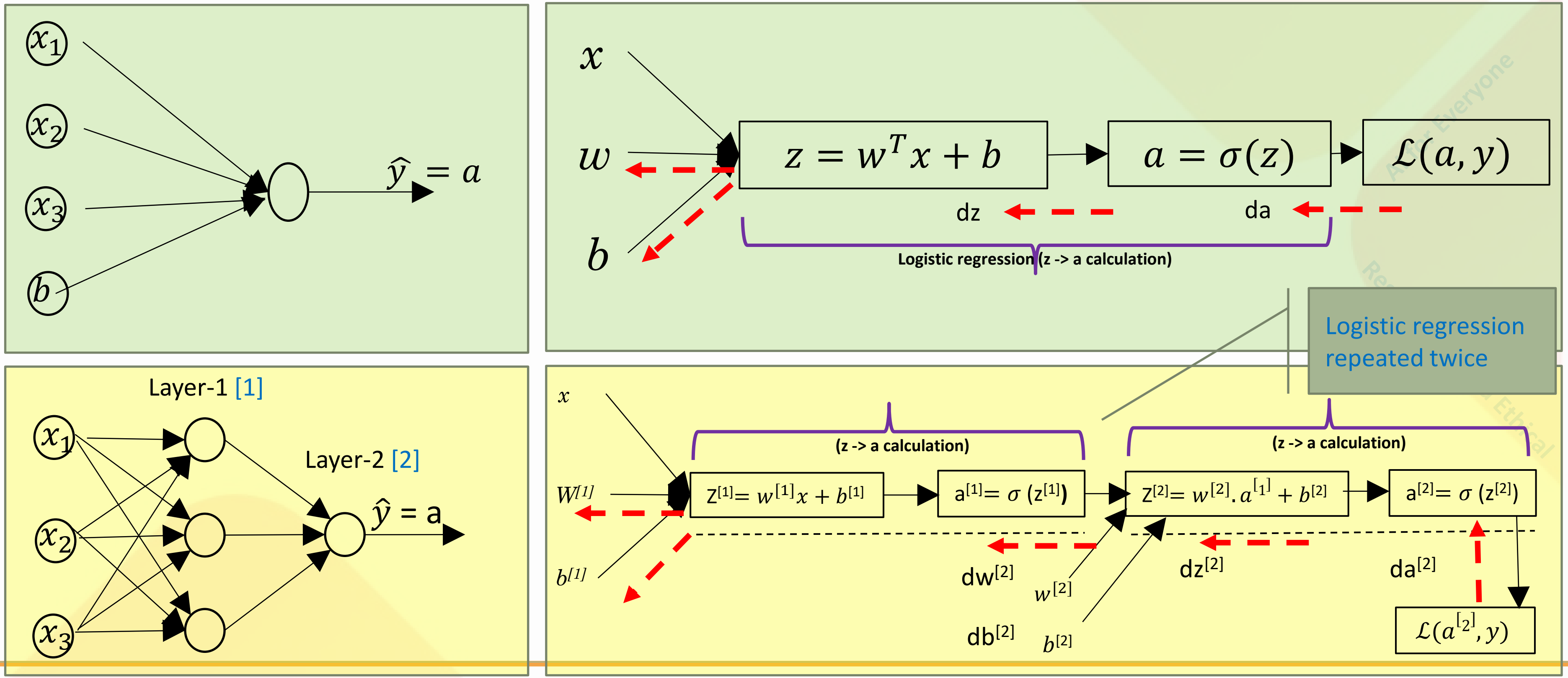
PERCEPTRON



x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



NEURAL NETWORKS



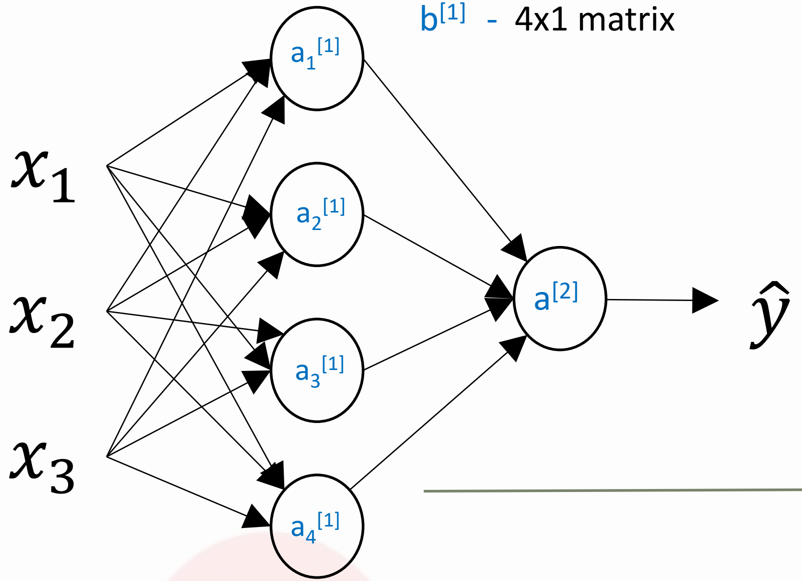
NEURAL NETWORK REPRESENTATION

$a^{[0]} = X$

$a^{[1]}$

$w^{[1]}$ - 4x3 matrix

$b^{[1]}$ - 4x1 matrix



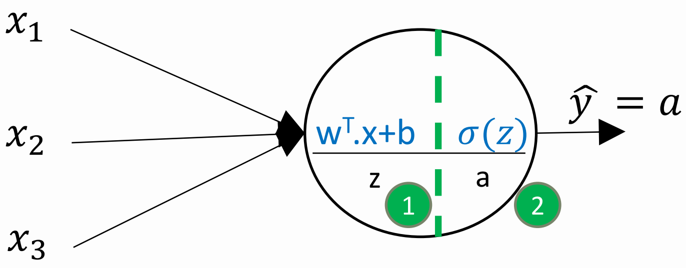
$\hat{y} = a^{[2]}$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

Input Layer | Hidden Layer | Output Layer

- Hidden layer –
  - is not exposed to the outside world
  - Associated with w, b
- Input layer – is not counted in the # of layers
  - The picture is of 2 layers NN

COMPUTING A NEURAL NETWORK'S OUTPUT

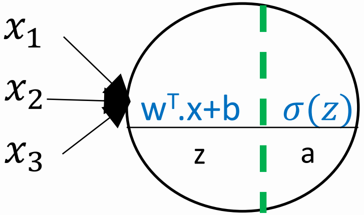


$z = w^T x + b$

$a = \sigma(z)$

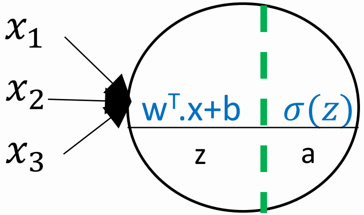
$Z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$

$a_1^{[1]} = \sigma(z_1^{[1]})$



$Z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$

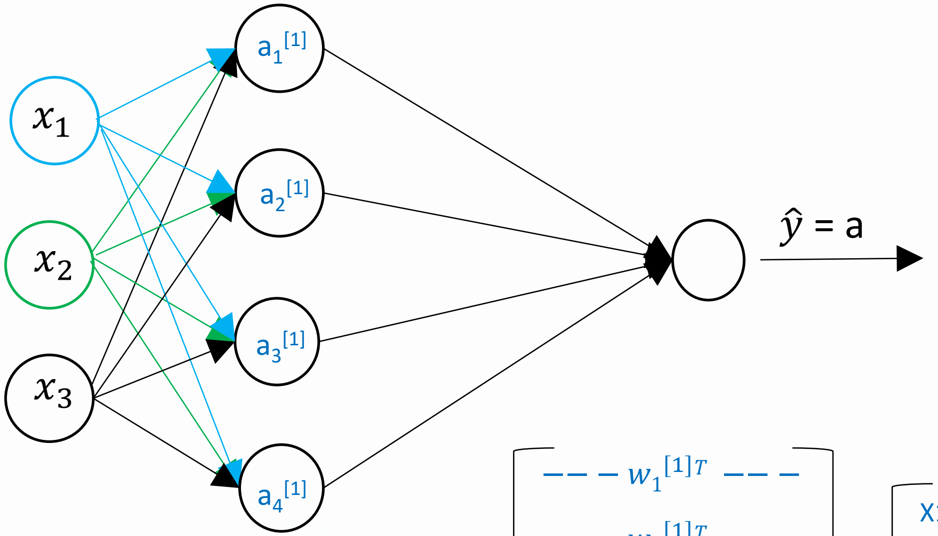
$a_2^{[1]} = \sigma(z_2^{[1]})$



$Z_i^{[l]}$ ← Layer, ← Node in the layer

Repeated logistic regression

COMPUTING A NEURAL NETWORK'S OUTPUT



$Z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]}$	$a_1^{[1]} = \sigma\,(z_1^{[1]})$
$Z_2^{[1]} = w_2^{[1]T}x + b_2^{[1]}$	$a_2^{[1]} = \sigma\,(z_2^{[1]})$
$Z_3^{[1]} = w_3^{[1]T}x + b_2^{[1]}$	$a_3^{[1]} = \sigma\,(z_3^{[1]})$
$Z_4^{[1]} = w_4^{[1]T}x + b_2^{[1]}$	$a_4^{[1]} = \sigma\,(z_4^{[1]})$

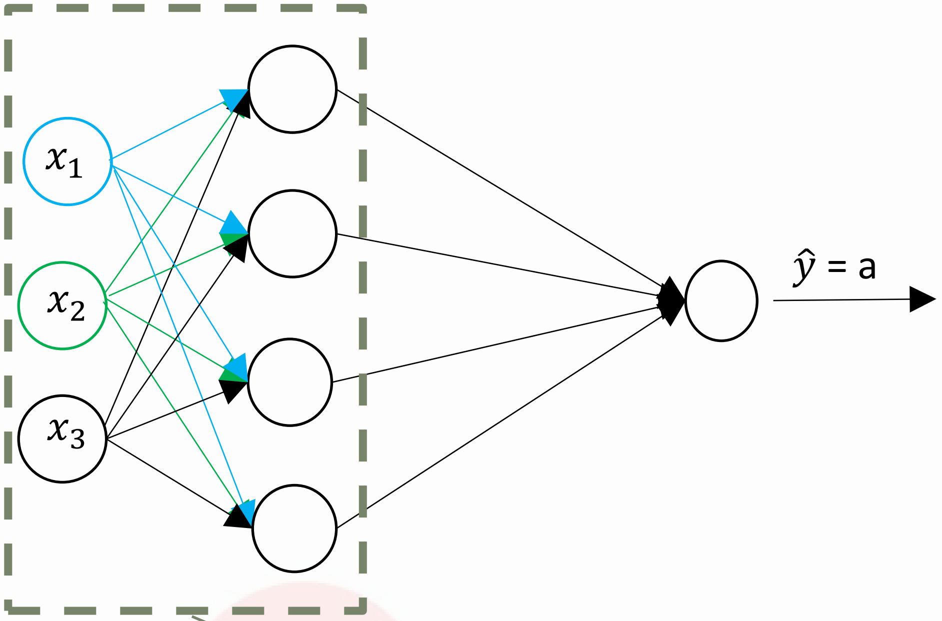
We need to vectorize the calculation as for loop is expensive

$$\underbrace{\begin{bmatrix} --- w_1^{[1]T} --- \\ --- w_2^{[1]T} --- \\ --- w_3^{[1]T} --- \\ --- w_4^{[1]T} --- \end{bmatrix}}_{W^{[1]}} \times \begin{bmatrix} X1 \\ X2 \\ X3 \end{bmatrix} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]}} = \underbrace{\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}}_{z^{[1]}}$$

4 x 3 matrix

$$\begin{vmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{vmatrix} = a^{[1]} = \sigma(z^{[1]})$$

DIMENSIONS OF THE MATRIX



Without the hidden layers, it is exactly like logistic regression

$$z^{[1]} = w^{[1]}.x + b^{[1]}$$

(4, 1) (4, 3) (3, 1) (4, 1)

$$a^{[1]} = \sigma\,(z^{[1]})$$

(4, 1) (4, 1)

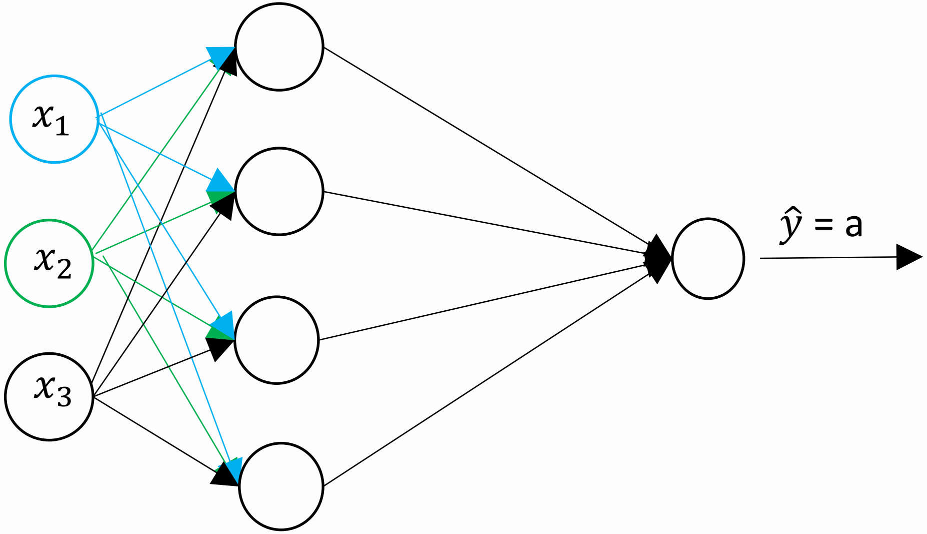
$$z^{[2]} = w^{[2]}.a^{[1]} + b^{[2]}$$

(1, 1) (1, 4) (4, 1) (4, 1)

$$a^{[2]} = \sigma\,(z^{[2]})$$

(1, 1) (1, 1)

VECTORIZING ACROSS MULTIPLE EXAMPLES



$z^{[1]}$	$= w^{[1]}.x$	$+$	$b^{[1]}$
$a^{[1]}$	$= \sigma(z^{[1]})$		
$z^{[2]}$	$= w^{[2]}.a^{[1]}$	$+$	$b^{[2]}$
$a^{[2]}$	$= \sigma(z^{[2]})$		

X -----------------------→ $a^{[2]} = \hat{y}$

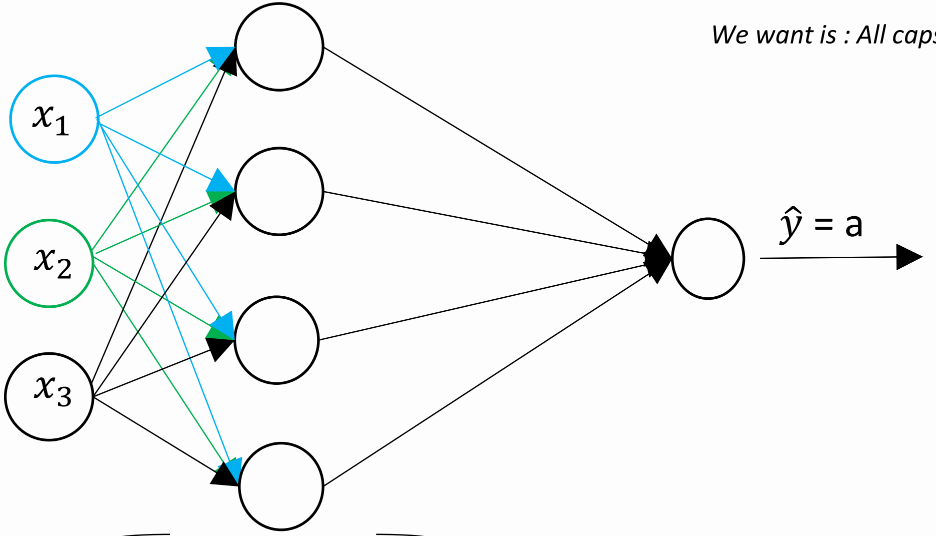
$X^{(1)}$ ....> $a^{[2](1)} = \hat{y}^{(1)}$
$X^{(2)}$ ....> $a^{[2](2)} = \hat{y}^{(1)}$
…
…
$X^{(m)}$ ....> $a^{[2](m)} = \hat{y}^{(1)}$

for I = 1 to m;

$z^{[1](i)} = w^{[1]}.x^{(i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = w^{[2]}.x^{(i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$

VECTORIZING ACROSS MULTIPLE EXAMPLES



$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & & x^{(m)} \\ \dots & \dots & \dots & \dots \\ | & | & & | \end{bmatrix}$$

Data rows stacked in columns (n_x, m)

We want is : All caps for the rows

$$\boxed{Z^{[1]} = W^{[1]}.X + b^{[1]}}$$

$$\boxed{A^{[1]} = \sigma\,(Z^{[1]})}$$

$$\boxed{Z^{[2]} = W^{[2]}.A^{[1]} + b^{[2]}}$$

$$\boxed{A^{[2]} = \sigma\,(Z^{[2]})}$$

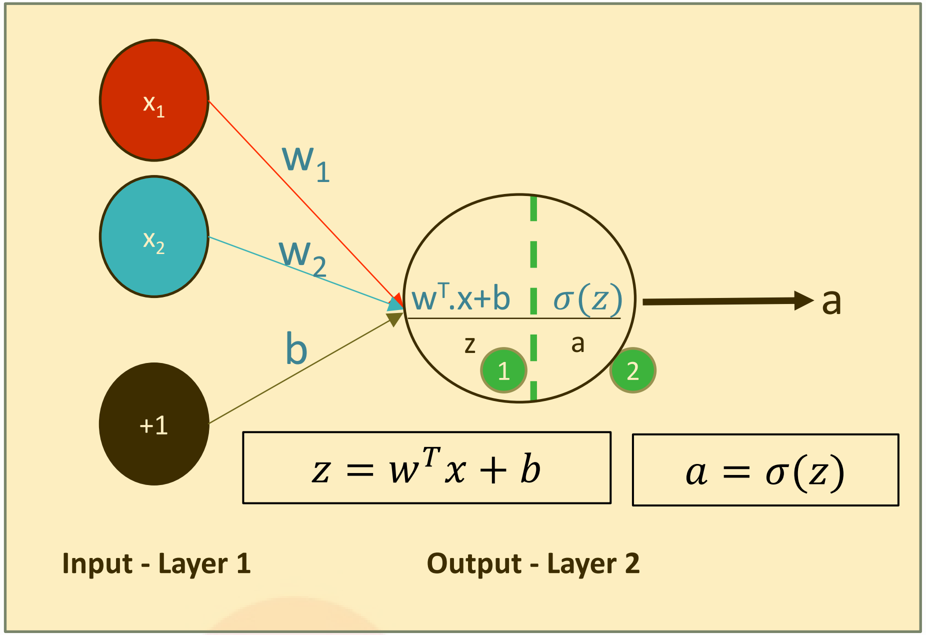
$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ X^{(1)} & X^{(2)} & & X^{(m)} \\ | & | & \dots & | \end{bmatrix}$$

Training samples

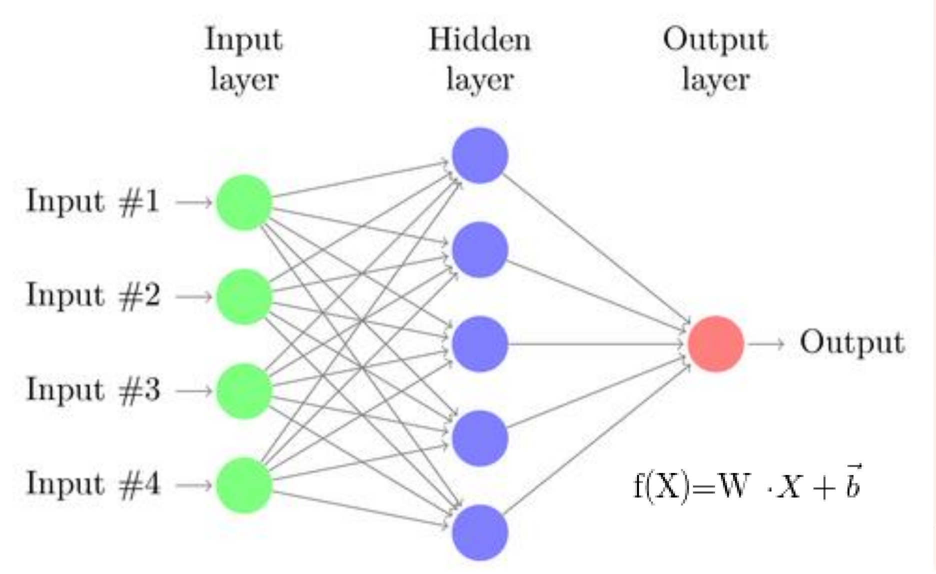
$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{(1)(1)} & a^{(2)(2)} & & a^{[1](m)} \\ | & | & \dots & | \end{bmatrix}$$

Nodes in the hidden layer

DENSE LAYER



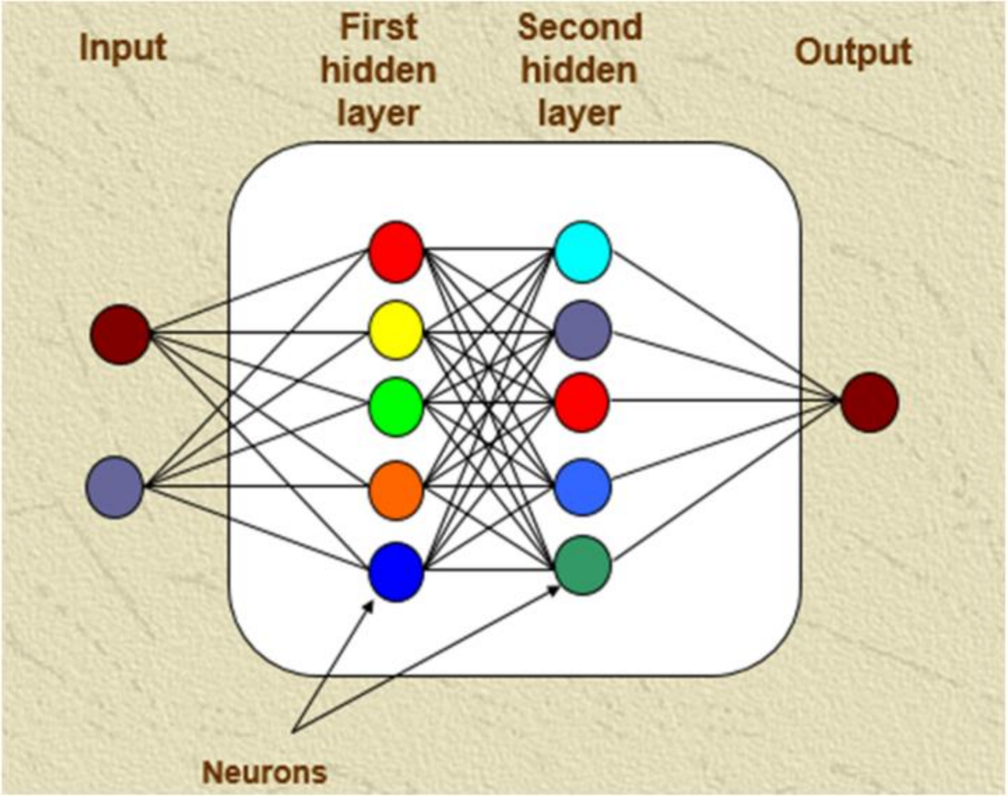
Perceptron



Note that bias term is now a vector and W is a weight matrix

stack a bunch of these perceptrons together, it becomes a hidden layer which is also known as a Dense layer

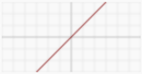
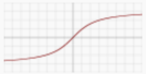
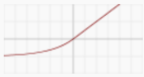
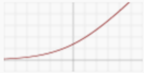
MULTI-LAYER PERCEPTRON NETWORK



stack a bunch of these dense layers together, it becomes a MULTI-LAYER PERCEPTRON network

ACTIVATION FUNCTIONS

ACTIVATION FUNCTIONS

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x \neq 0 \\ ? & \text{for} \quad x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x)+\alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

So what does an artificial neuron do?

Simply put, it calculates a “weighted sum” of its input, adds a bias and then decides whether it should be “fired” or not

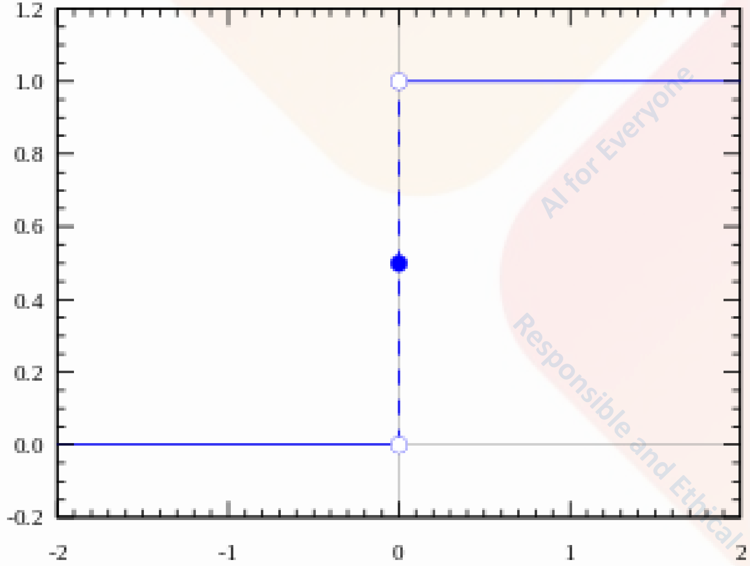
So consider a neuron. $Y = \sum (weight * input) + bias$

HEAVISIDE STEP FUNCTION

- threshold-based activation function

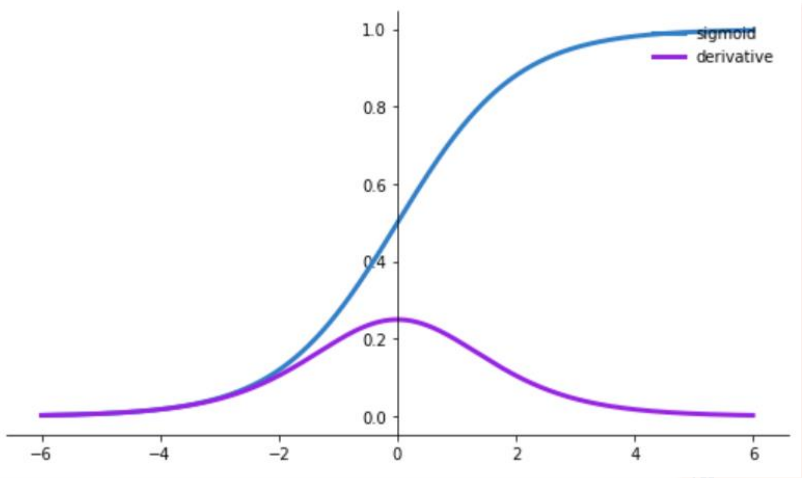
- If the value of Y is above a certain value, declare it activated.
- If it’s less than the threshold, then say it’s not.

- Activation function A = “activated”
  - if Y > threshold else not
  - Alternatively, A = 1 if y> threshold,
  - 0 otherwise



SIGMOID FUNCTION

- In between X values -2 to 2, y values are very steep.
- That means this function has a tendency to bring the y values to either end of the curve.
    - good for a classifier
- the output of the activation function is always going to be in range (0, 1)
- the y values tend to respond very less to changes in X. towards either end of the sigmoid function,
    - The gradient at that region is going to be small.
    - It gives rise to a problem of "vanishing gradients".
    - The network refuses to learn further or is drastically slow

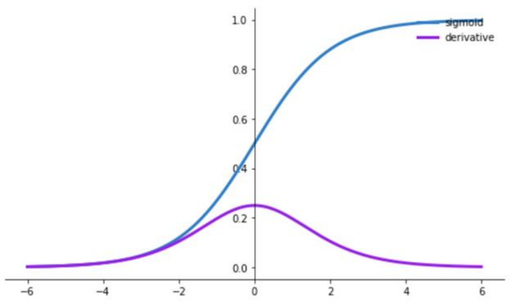


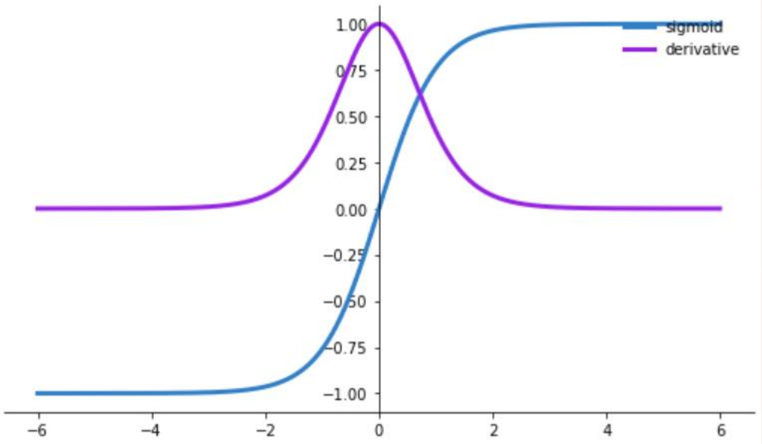
Pros	Cons
nonlinear	gives rise to a problem of "vanishing gradients".
smooth gradient	
good for a classifier.	

- As x approaches positive infinity, $e{-}x$ becomes very close to zero, and $\sigma(x)$ approaches 1.
- Similarly, as x approaches negative infinity, $e{-}x$ becomes very large, and $\sigma(x)$ approaches 0.
- However, for large positive or negative inputs, the change in the output of the sigmoid function becomes negligible.
- This leads to a situation where inputs that are significantly larger or smaller than the input range where the sigmoid function provides meaningful distinctions result in similar outputs, effectively "flattening out" the function.

TANH

- tanh function is called a shifted version of the sigmoid function.
- reason that tanh is preferred compared to sigmoid, is that the derivatives of the tanh are larger than the derivatives of the sigmoid.
- In other words, you minimize your cost function faster if you use tanh as an activation function.

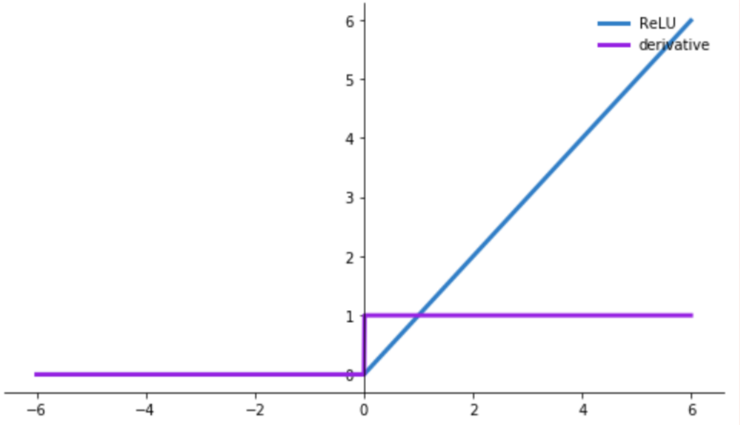




Pros	Cons
Nonlinear	"vanishing gradients"
gradient is stronger than sigmoid	
good for a classifier.	
bound to range (-1, 1)	

RELU

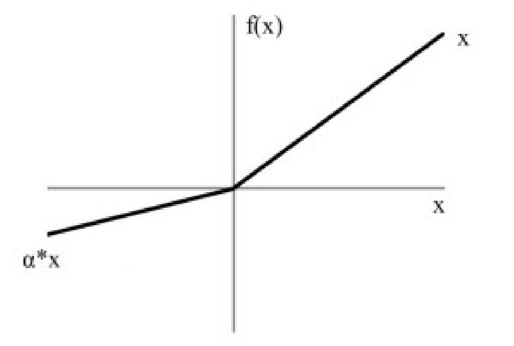
- A recent invention which stands for Rectified Linear Units.
- The formula is deceptively simple: max(0, z).
- Despite its name and appearance, it's not linear and provides the same benefits as Sigmoid but with better performance.
- sigmoid or tanh have upper limits to saturate whereas ReLU doesn't saturate for positive inputs.

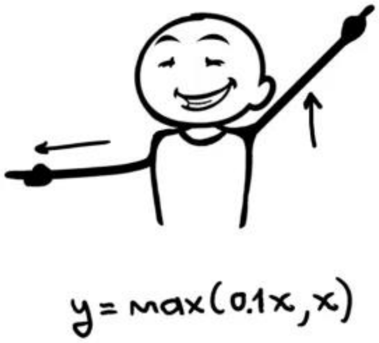


Pros	Cons
avoids and rectifies vanishing gradient problem.	only be used within Hidden layers
less computationally expensive than tanh and sigmoid	could result in Dead Neurons.
range of ReLu is [0, inf).	blow up the activation.

LEAKYRELU

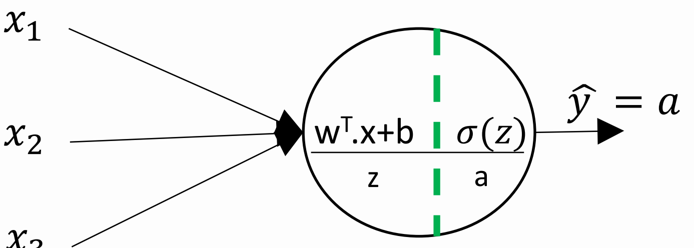
- LeakyRelu is a variant of ReLU.
- Leaky ReLUs allow a small, non-zero gradient when the unit is not active.
- Instead a small, non-zero, constant gradient α (Normally, α=0.01). of being 0 when z<0, a leaky ReLU allows
- *However, the consistency of the benefit across tasks is presently unclear.*



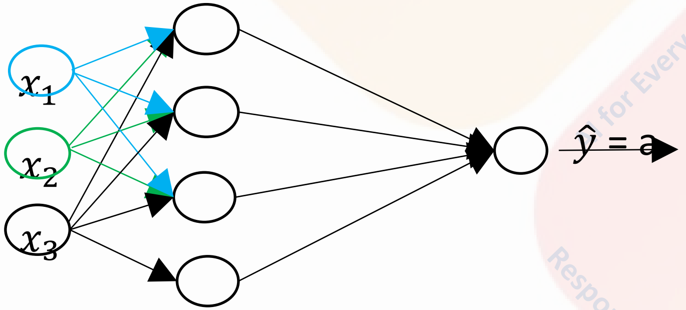


DEEP L-LAYER NN

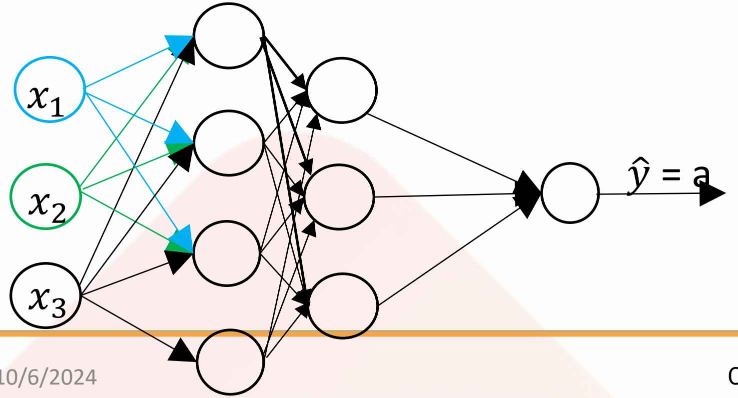
Logistic Regression



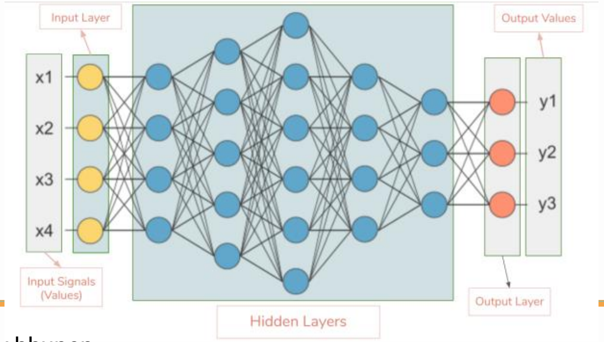
1-layer neural network



2-layer neural network

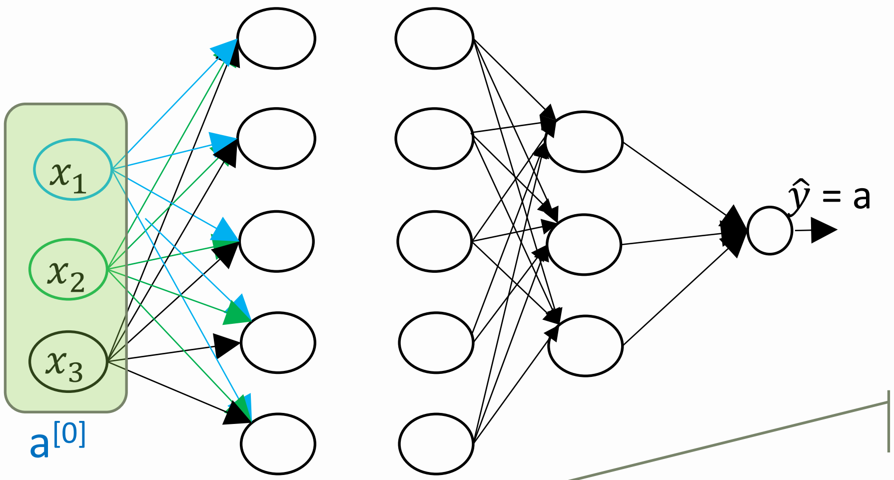


6-layer neural network



FORWARD PROPAGATION - DEEP NEURAL NETWORK

4-layer NN

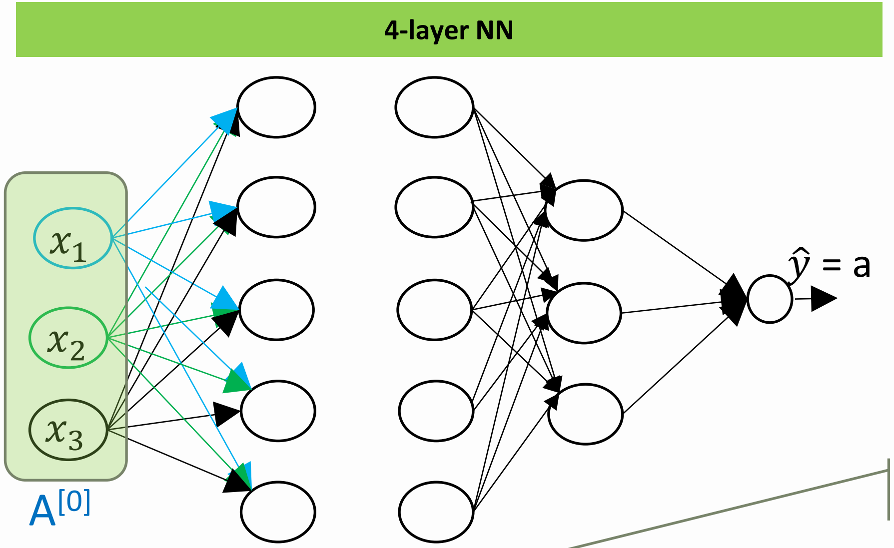


- L = 4 (number of layers)
- $n^{[l]}$ = number of units in layer l
  - $n^{[0]}$ - $n_x = 3$
  - $n^{[1]}$ - 5
  - $n^{[2]}$ - 5
  - $n^{[3]}$ - 3
  - $n^{[4]}$ - 1
- $a^{[l]}$ – activations in layer l = $g^{[l]}(z^{[l]})$
- $w^{[i]}$ – weights for computing $z^{[l]}$

$z^{[l]} = w^{[l]}. a^{[l-1]} + b^{[l]}$; $a^{[l]} = g^{[l]}. (z^{[l]})$

Layer 1	Layer 2	Layer 3	Layer 4
$z^{[1]} = w^{[1]}.x + b^{[1]}$	$z^{[2]} = w^{[2]}.x + b^{[2]}$	$z^{[3]} = w^{[3]}.x + b^{[3]}$	$z^{[4]} = w^{[4]}.x + b^{[4]}$
$a^{[1]} = g^{[1]}. (z^{[1]})$	$a^{[2]} = g^{[2]}. (z^{[2]})$	$a^{[3]} = g^{[3]}. (z^{[3]})$	$a^{[4]} = g^{[4]}. (z^{[4]}) = \hat{y}$

FORWARD PROPAGATION - DEEP NEURAL NETWORK



- L = 4 (number of layers)
- $n^{[l]}$ = number of units in layer l
  - $n^{[0]}$ - $n_x = 3$
  - $n^{[1]}$ - 5
  - $n^{[2]}$ - 5
  - $n^{[3]}$ - 3
  - $n^{[4]}$ - 1
- $a^{[l]}$ – activations in layer l = $g^{[l]}(z^{[l]})$
- $w^{[i]}$ – weights for computing $z^{[l]}$

$Z^{[l]} = W^{[l]}.A^{[l-1]} + b^{[l]}$; $A^{[l]} = g^{[l]}.(Z^{[l]})$

Layer 1	Layer 2	Layer 3	Layer 4
$Z^{[1]} = W^{[1]}.X + b^{[1]}$ $Z^{[1]} = W^{[1]}.A^{[0]} + b^{[1]}$	$Z^{[2]} = W^{[2]}.A^{[1]} + b^{[2]}$		
$A^{[1]} = g^{[1]}.(Z^{[1]})$	$A^{[2]} = g^{[2]}.(Z^{[2]})$		

GETTING THE MATRIX DIMENSIONS RIGHT

- Neural networks are complex

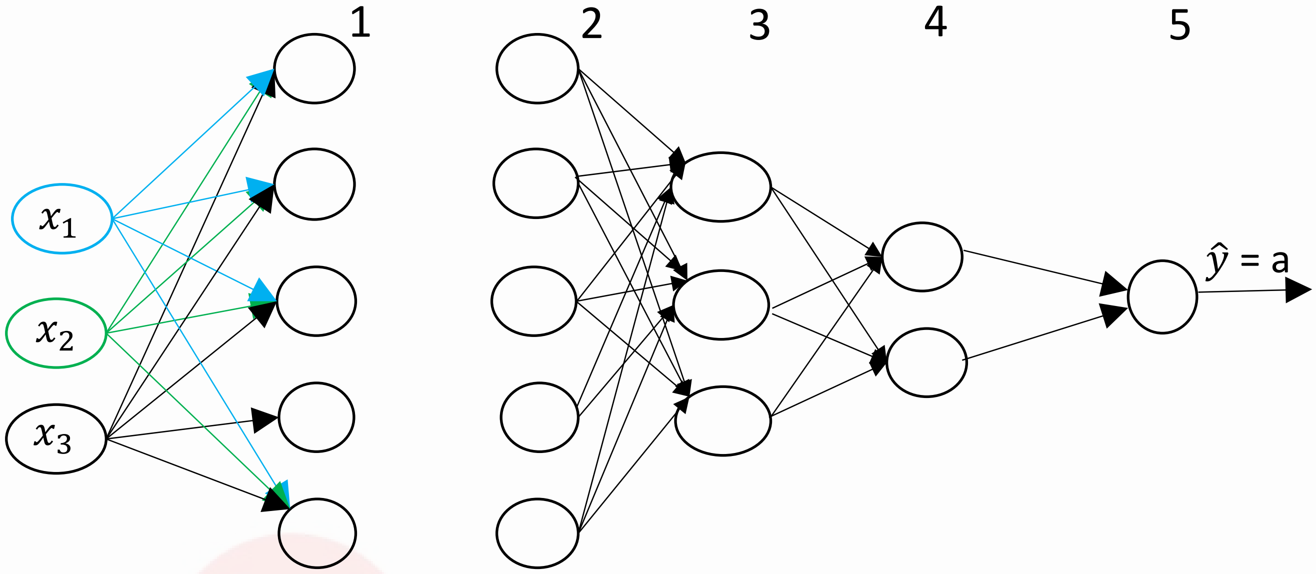
- Lots of notations and layers

- Need to plan and design the network carefully
  - This is where matrix comes into play
    - Matrix dimensions become important to deal with

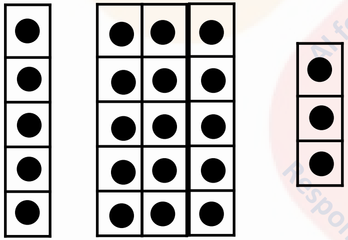
  - Use pen and paper initially to draw and validate (re validate) the designs

MATRIX DIMENSIONS

Parameters $W^{[l]}$ and $b^{[l]}$



$Z^{[1]}$	=	$w^{[1]}.$	X	+	$b^{[1]}$
(5,1)		(5,3)	(3,1)		



$Z^{[2]}$	=	$w^{[2]}.$	$a^{[1]}$	+	$b^{[1]}$
(5,1)		(5,5)	(5,1)		

$n^{[0]} = 3$ $n^{[1]} = 5$ $n^{[2]} = 5$ $n^{[3]} = 3$ $n^{[4]} = 2$ $n^{[5]} = 1$

- A summary of how to train DL models on AWS

Responsible and Ethical

HOW TO TRAIN DEEP LEARNING MODELS ON AWS

Data Preparation:

Organize and preprocess your training data.

Ensure that your data is stored securely and efficiently using Amazon S3 (Simple Storage Service) or other storage options on AWS.

Selecting the DL Framework:

Choose the Deep Learning framework that best suits your needs.

AWS supports popular frameworks like TensorFlow, PyTorch, and MXNet.

You can use AWS Deep Learning AMIs (Amazon Machine Images) that come pre-installed with these frameworks.

HOW TO TRAIN DEEP LEARNING MODELS ON AWS

Create

a **Training Script**:

- chosen DL framework that defines your model architecture, loss functions, data loading, and training procedures.

Choose

an **AWS Compute Instance**:

- Select an appropriate AWS EC2 instance with the right amount of CPU and GPU resources to handle your DL workload.

Use

AWS SageMaker (Optional):

- AWS SageMaker is a fully managed service that simplifies the end-to-end ML workflow. You can use it to create a SageMaker Notebook instance, where you develop and run your training script.

HOW TO TRAIN DEEP LEARNING MODELS ON AWS

Data Parallelism (Optional)

For large-scale training, you can leverage data parallelism using AWS Data Parallelism Library (DPL) or other distributed training frameworks.

Hyperparameter Tuning (Optional)

you can use SageMaker's automatic model tuning capabilities. It performs hyperparameter optimization using techniques like Bayesian optimization or random search.

HOW TO TRAIN DEEP LEARNING MODELS ON AWS

Monitor	**Training Progress** •Use AWS CloudWatch or SageMaker's built-in monitoring capabilities to track training progress and metrics.
Model	**Evaluation** •After training, evaluate your trained model's performance on a separate validation dataset. •This step helps ensure that the model generalizes well to new data and meets your desired accuracy and quality metrics.

HOW TO TRAIN DEEP LEARNING MODELS ON AWS

Model Deployment:

- Once you are satisfied with the model's performance, you can deploy it on AWS using SageMaker endpoints, AWS Lambda, or other services.
- This allows you to use the trained model for inference or integrate it into your applications.

Model Versioning and Management:

- AWS provides versioning and management capabilities for your trained models.
- You can keep track of different model versions and easily switch between them as needed.


AI for Everyone
Responsible and Ethical
STOP ME IF YOU HAVE
QUESTIONS !!!