DEEP LEARNING SPECIALIZATION

COURSERA-DEEPLEARNING.AI-ANDREW NG

These are notes taken from Deep Learning Specialization by deeplearning.ai in Coursera. All the screenshots shown here are taken from the course videos. I would advise you to first watch the videos of the course in order to completely understand the notes.

SKILLS YOU WILL GAIN

Tensorflow

Convolutional Neural Network

Artificial Neural Network

Deep Learning

Backpropagation

Python Programming

Hyperparameter

Hyperparameter Optimization

Machine Learning

Inductive Transfer

Multi-Task Learning

Facial Recognition System

NEURON

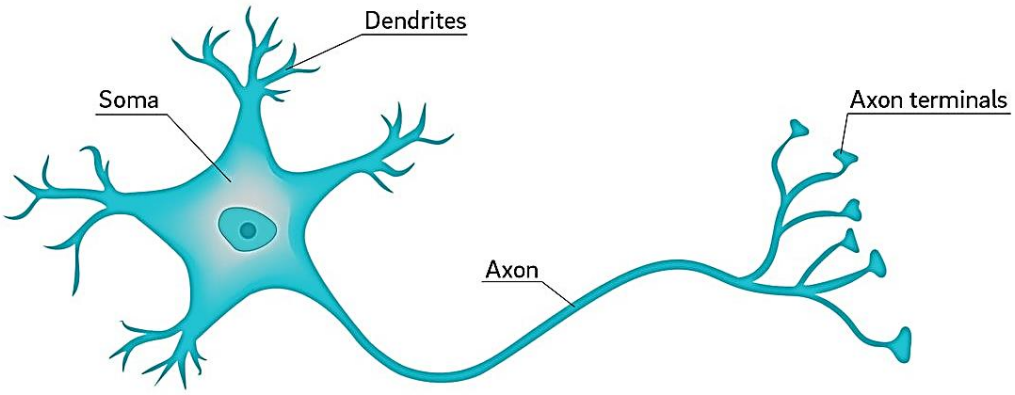
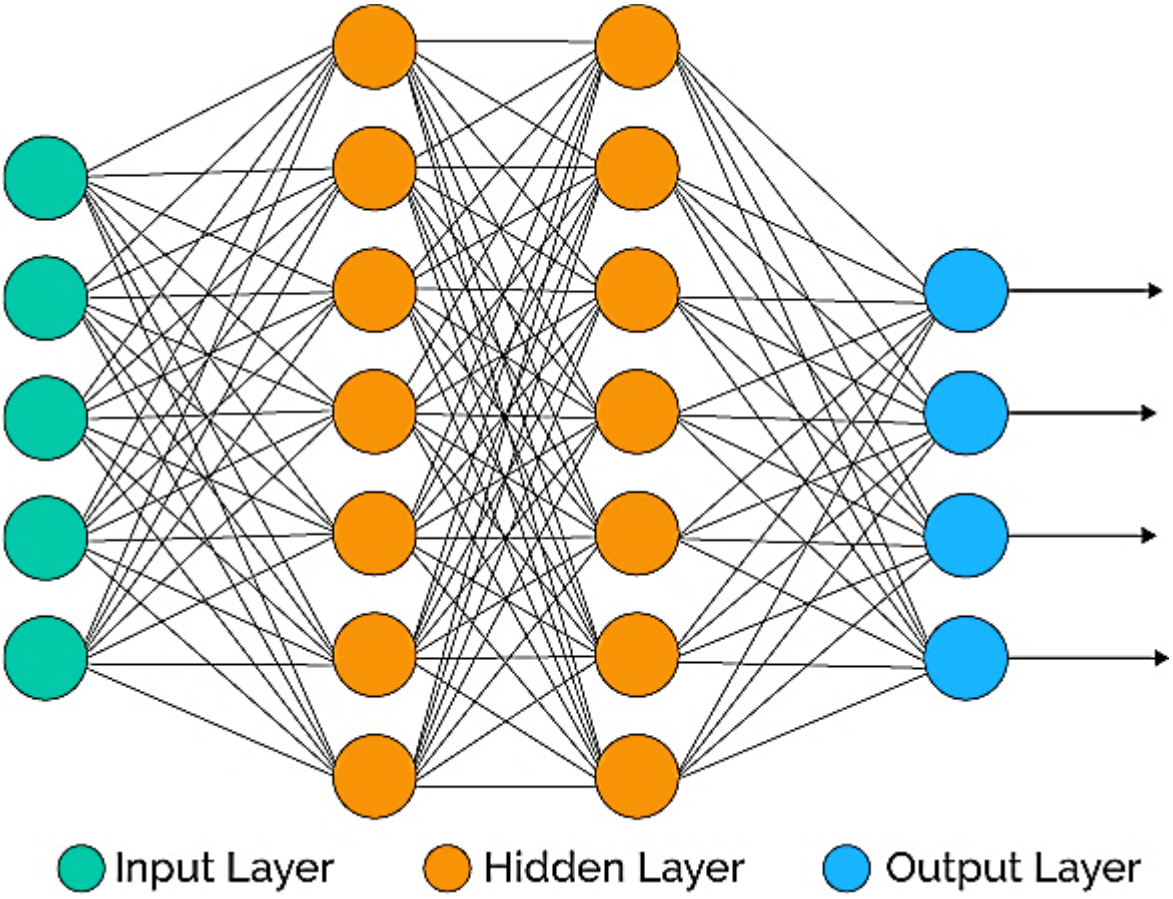


Illustration of a neuron. Credit: David Baillot/ UC San Diego

Neurons are cells within the nervous system that transmit information to other nerve cells, muscle, or gland cells. Most neurons have a cell body, an axon, and dendrites. The cell body contains the nucleus and cytoplasm. The axon extends from the cell body and often gives rise to many smaller branches before ending at nerve terminals. Dendrites extend from the neuron cell body and receive messages from other neurons. Synapses are the contact points where one neuron communicates with another. The dendrites are covered with synapses formed by the ends of axons from other neurons.



https://www.xenonstack.com/images/blog/Artificial-Neural-Network-Architecture.jpg

A **neural network** is a series of algorithms that recognizes the underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Neural networks can adapt to changing input; so, the

network generates the best possible result without needing to redesign the output criteria.

Courses in this Specialization

1. Neural Networks and Deep Learning
2. Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization
3. Structuring your Machine Learning project
4. Convolutional Neural Networks
5. Natural Language Processing: Building sequence models

- AI is the new Electricity
- Electricity had once transformed countless industries: transportation, manufacturing, healthcare, communications, and more
- AI will now bring about an equally big transformation.

COURSE 1

Week 1: Introduction

Week 2: Basics of Neural Network programming

Week 3: One hidden layer Neural Networks

Week 4: Deep Neural Networks

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

· superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

·m : number of examples in the dataset

·n_x : input size

·n_y : output size (or number of classes)

·$n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h{}^{[\text{number of layers } +1]}$.

·L : number of layers in the network.

Objects:

·$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

·$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

- $Y \in \mathbb{R}^{n_y \times m}$ is the label matrix
- $y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example
- $W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer
- $b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer
- $\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

- General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$
- $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

- $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$
- $J_1(\hat{y}, y) = \sum_{i=0}^{m} | y^{(i)} - \hat{y}^{(i)} |$

2 Deep Learning representations

For representations:

· nodes represent inputs, activations or outputs

· edges represent weights or biases

Here are several examples of Standard deep learning representations

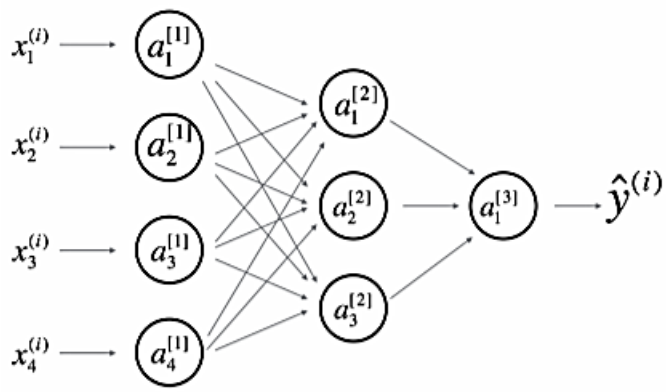


Figure 1: Comprehensive Network: representation commonly used for Neural Networks. For better aesthetic, we omitted the details on the parameters ($w_{ij}^{[l]}$ and $b_i^{[l]}$ etc...) that should appear on the edges

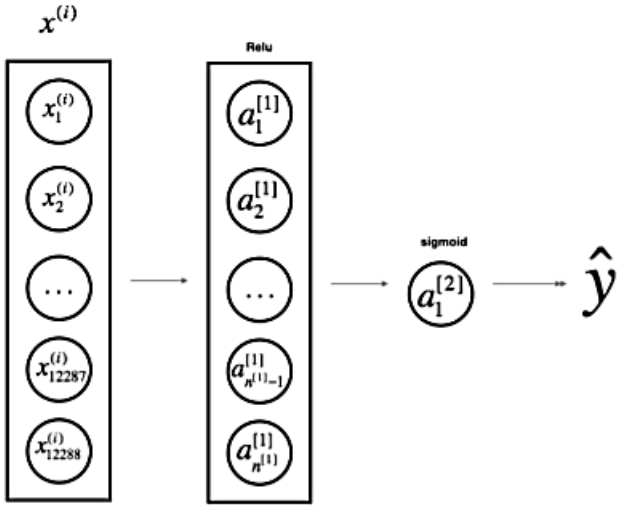


Figure 2: Simplified Network: a simpler representation of a two layer neural network, both are equivalent.

***Note**:

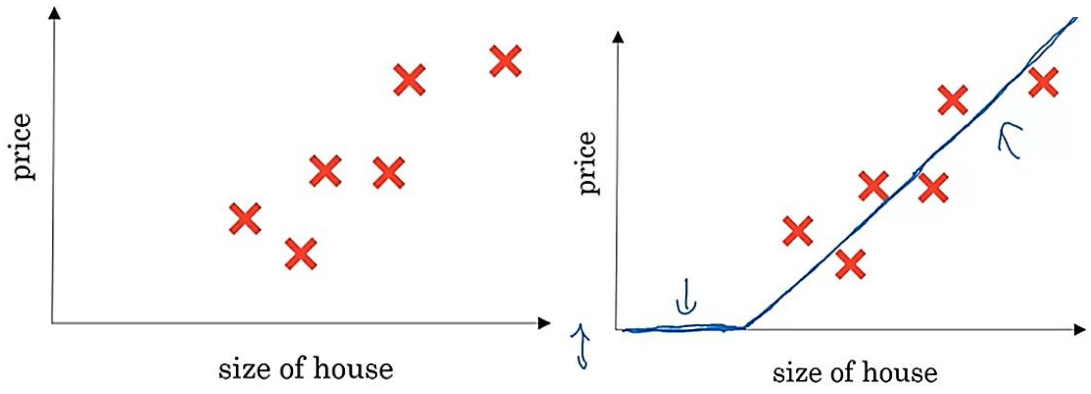
There are couple of optional videos in this course and I would suggest you guys to not skip it even if you don't have time to watch because those videos helped me lot in learning deep learning better way. Also, the videos where Ng talks with Heroes of Deep Learning would be a great inspiration which is much needed after doing the difficult assignments (Not at all difficult if you follow everything carefully).

Another advice is to derive all the formulas by yourself after learning from the course again even if it takes a full day. Even though Andrew Ng continuously says not to worry about the math I have found it very helpful to look in the internet about the math a little bit, just enough to understand how the activation functions work. That way you would get a better understanding of the notations used in the course and how they are derived and would be able to finish the assignments much faster. Also try to re – implement all the assignments at least once after submitting.

WEEK 1-INTRODUCTION

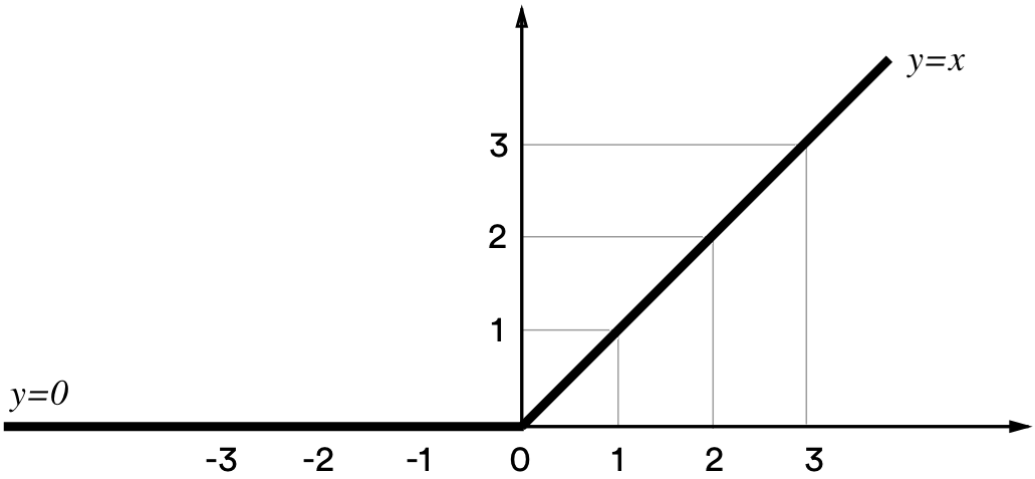
What is Neural Network?

Let's answer this question by taking an example of house price prediction. We have data set containing the size of houses and its price. We need to fit a line through this linear data that will allow us to predict future prices. What we normally do is to fit a straight line through the data. But as we know that the straight line goes to the negative of y-axis but the house prices cannot be negative so we bend the line near 0 on y-axis.

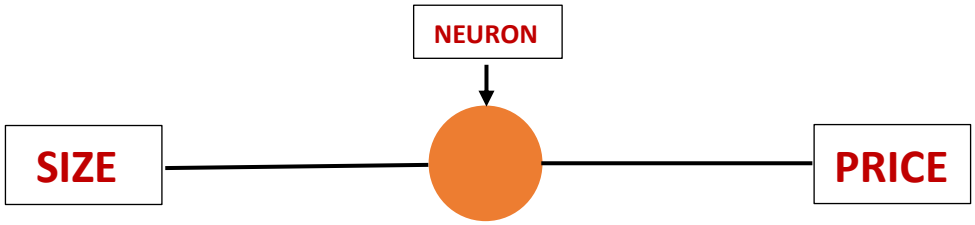


This is the simplest neural network we can implement. Line that we fit here is called a **ReLU (Re**ctified **L**inear **U**nit**)** function which accepts input values and gives its max (0,x). Which means if input value is positive it returns as it is. But if input is negative it returns 0.

ReLU (Rectified **L**inear **U**nit**)**



The neuron accepts input which is size of the house→takes max of the size→outputs the price.

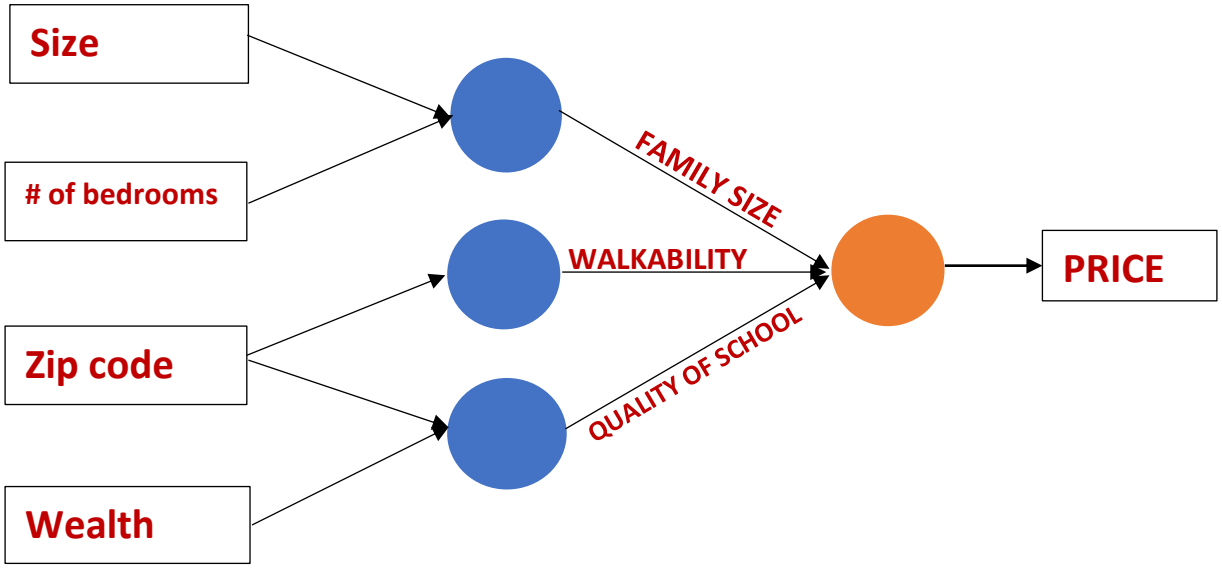


As we have implemented the smallest neural network in order to get a bigger neural network, we just put together several of these smaller neural networks. We would again add more features to our housing price prediction. Let's consider these simple neural networks which decides family size, walkability and school quality.

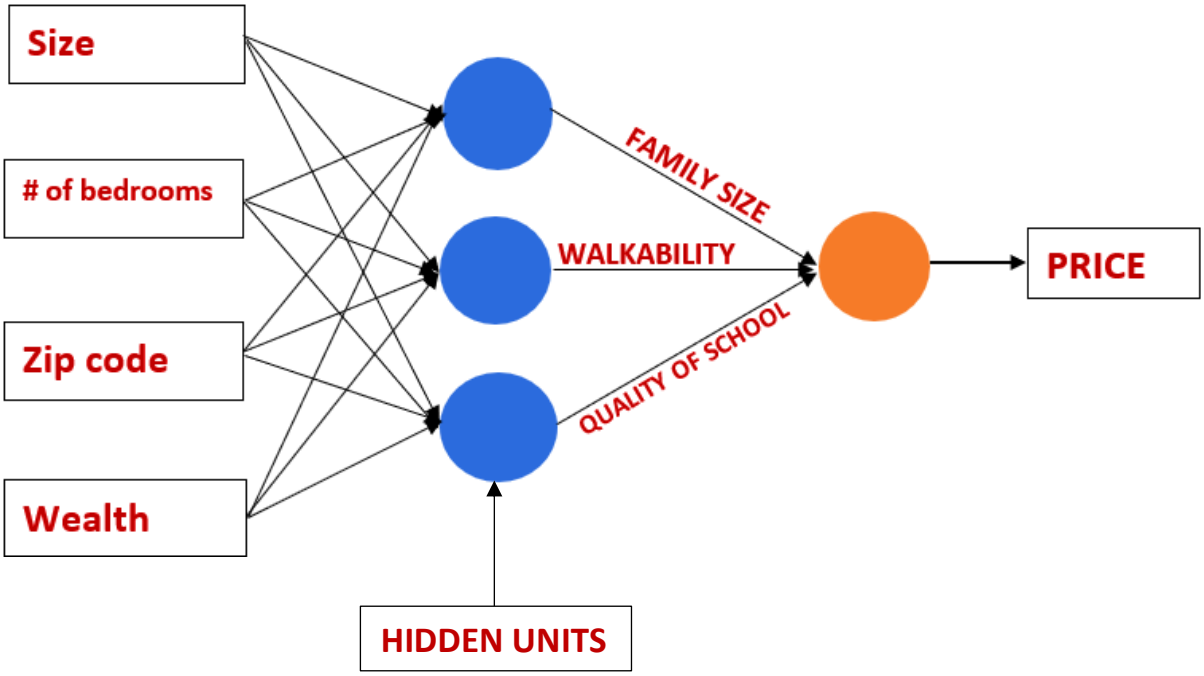
1. Size, # of bedrooms → Family size
2. Zip code (postal code)→Walkability
3. Zip code, Wealth→Quality of school

Now we would stack together all these to get a bigger neural network that takes features Family size, Walkability

and Quality of school as input and gives price of house as output.



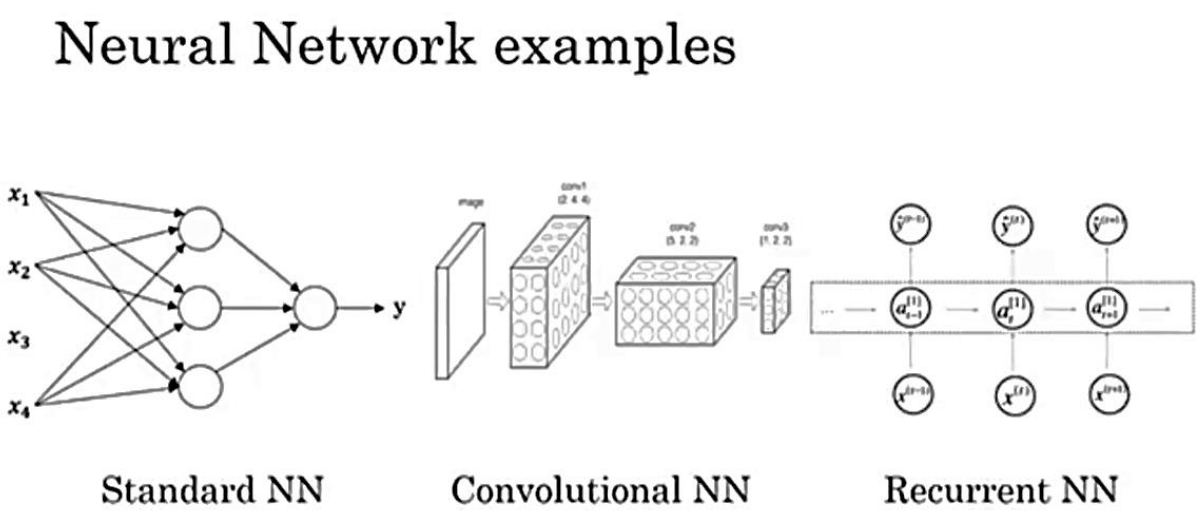
What we actually do here is we give the neural networks all the input features and let it decide which one leads to which output feature.



Supervised Learning

INPUT (X)	OUTPUT (Y)	APPLICATION	NN USED
Home features	Price	Real estate	Standard NN
Ad, user info	Click on ad? (0/1)	Online advertising	Standard NN
Image	Object (1...1000)	Photo tagging	CNN
Audio	Text transcript	Speech recognition	RNN
English	Chinese	Machine translation	RNN
Image, Radar info	Position of the cars	Autonomous driving	Hybrid

In the example we saw of housing price prediction we use standard neural networks. In image recognition we use **C**onvolutional **N**eural **N**etwork. In Text transcript and Machine translation we use **R**ecurrent **N**eural **N**etwork as the data is sequential data. And for bigger application like autonomous driving we use a hybrid of several of these neural networks.



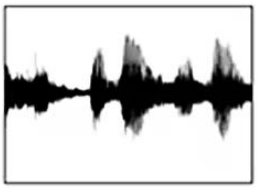
Now we will see difference between structured and unstructured data.

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Unstructured Data



Audio

Image

Four scores and seven years ago...

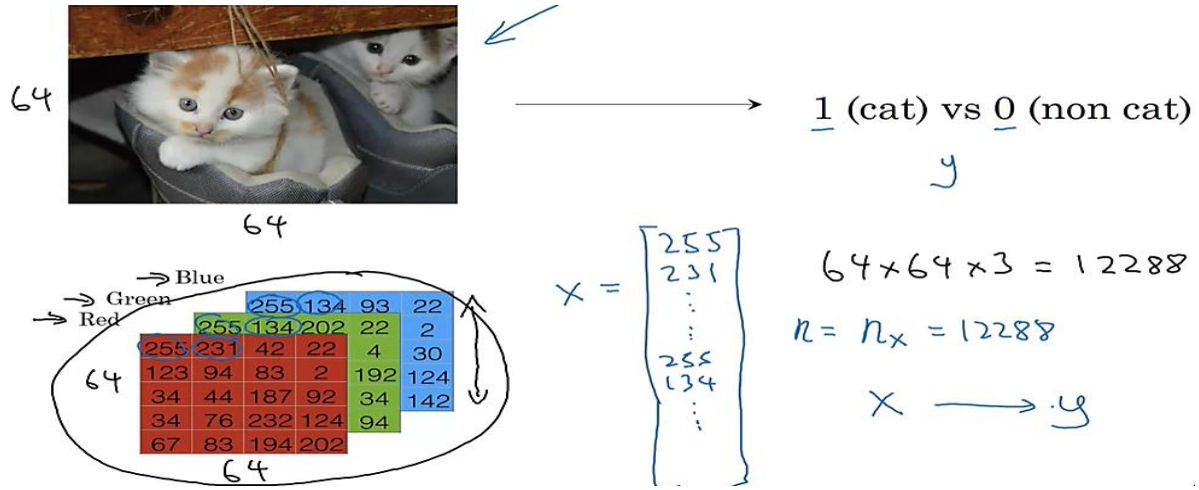
Text

Structured data refers to well defined data which for example we can get from a database. Here the features have a well-defined meaning. Unstructured data refers to not well-defined data like pictures, voice, text etc. Where the features do not have a very well-defined meaning.

WEEK 2 – Logistic Regression

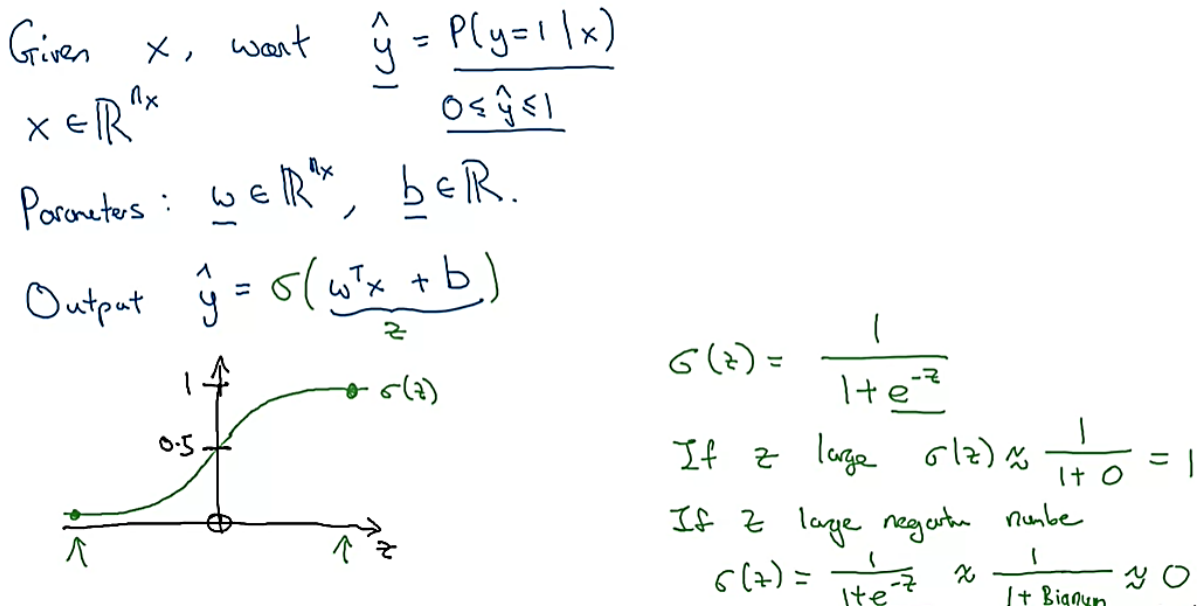
Binary classification

For understanding binary classification let's take an example of classifying an image as cat or no cat (1/0).



An image is represented as a 3D matrix with its height, width and 3 colour channels. The 3 colour channels represent the pixel intensity values of the respective colours in the image. For our input vector X we need to reshape this into a **(H * W * 3, 1**) vector which would be our input X and Y (cat/ not cat) will be our output.

The algorithm for this problem is written as $\hat{\boldsymbol{y}} = \boldsymbol{P(y = 1|x)}$ where $\hat{\boldsymbol{y}}$ is the probability that y = 1 given x. We have x as a $(\boldsymbol{n, m})$ matrix with parameters w and b where w is also an $(\boldsymbol{n, 1})$ dimensional matrix. Given these parameters and x we need to find $\hat{\boldsymbol{y}}$**.** One thing we can do is to fit a straight line i.e. $\hat{\boldsymbol{y}}$ **=** $\boldsymbol{w^T x + b}$ **.** But since this is a binary classification, we want the values to be between 0 and 1. So we can't use this as $\boldsymbol{w^T x + b}$ can be greater than 1 or even be -ve.



So here we use another activation function called **sigmoid** function $\hat{\boldsymbol{y}} = \boldsymbol{\sigma}(\boldsymbol{z})$ **where** $\boldsymbol{z} = \boldsymbol{w^T x} + \boldsymbol{b}$ **.** The sigmoid function $\boldsymbol{\sigma}(\boldsymbol{z}) = \frac{\mathbf{1}}{\mathbf{1}+\boldsymbol{e}^{-\boldsymbol{z}}}$**.** Therefor if the value of z is very high then $\hat{\boldsymbol{y}}$ ***is close to*** $\mathbf{1}$ and if the value of z is very low then $\hat{\boldsymbol{y}}$ ***is close to*** $\mathbf{0}$**.** Now in order to reduce z we calculate loss function to know the difference of prediction and true value.

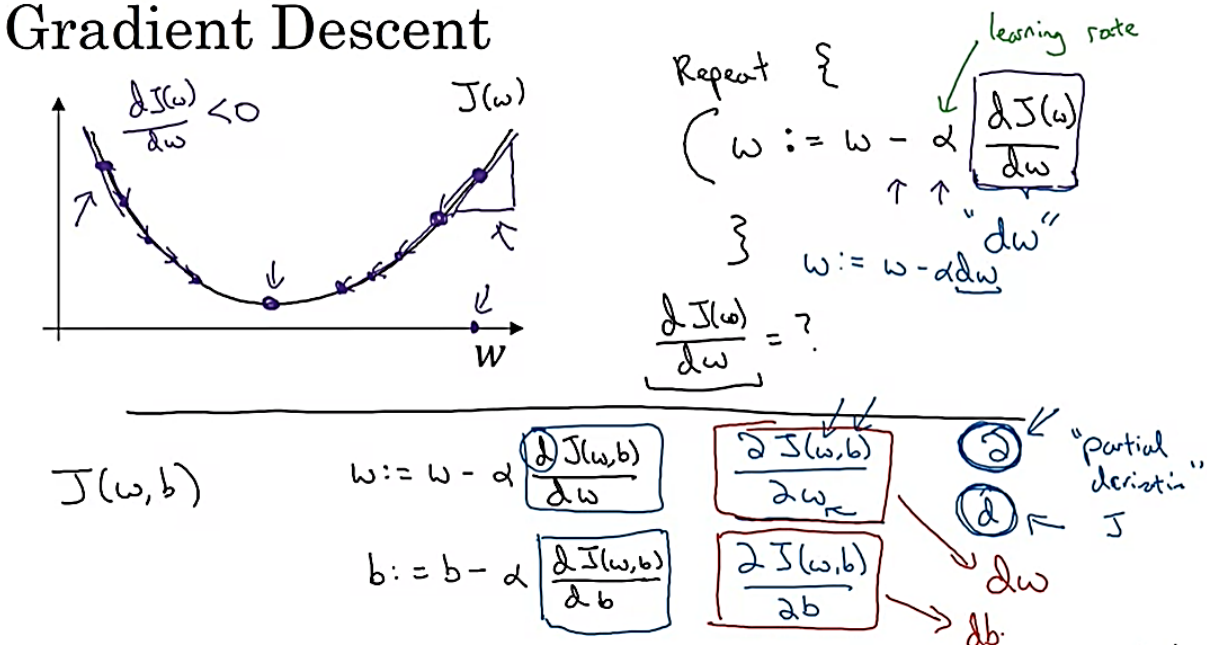
$$\boldsymbol{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\boldsymbol{y}\,\boldsymbol{log}\,\hat{\boldsymbol{y}} - (\mathbf{1}-\boldsymbol{y})\,\boldsymbol{log}(\mathbf{1}-\hat{\boldsymbol{y}})$$

- When y = 0 $\boldsymbol{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ **= -** $\boldsymbol{log}(\mathbf{1}-\hat{\boldsymbol{y}})$ i.e. loss would be less if $\hat{\boldsymbol{y}}$ is small i.e. close to 0.
- When y = 1 $\boldsymbol{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ **= -** $\boldsymbol{log}(\hat{\boldsymbol{y}})$ i.e. loss would be less if $\hat{\boldsymbol{y}}$ is large i.e. close to 1.

For m training example the cost function is written as:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(yhat^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} \left(y^{(i)} logyhat^{(i)} + (1-y^{(i)}) log(1-yhat^{(i)})\right)$$

In order to minimize the cost function, we need to do gradient descent. Let's say we have only one parameter for the sake of visualising the cost function.



Here we need to reach the global minima. In order to do that we need to climb down the hill. If we are on the left side of the hill slope is -ve, so gradient descent would be $\boldsymbol{w = w - \alpha(-dw)}$ **=** $\boldsymbol{w + \alpha(dw)}$**.** i.e. we are increasing value of w to reach bottom. If we are on the right side of the loss function curve then slope is +ve so gradient descent is $\boldsymbol{w = w - \alpha(dw)}$ therefor we have to decrease w in order to get to global minimum. When we have multiple features x1, x2 we have w1, w2 and b as parameters. We need to find dw1, dw2 and db to do gradient descent as:

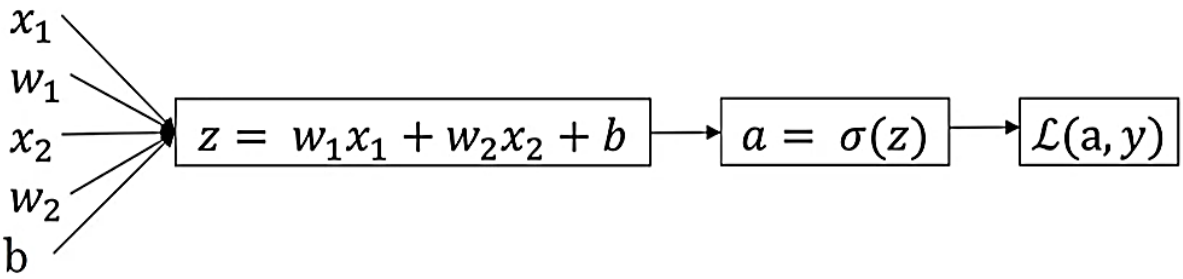
$$\boldsymbol{w_1 := w_1 - \alpha\, dw_1}$$

$$\boldsymbol{w_2 := w_2 - \alpha\, dw_2}$$

$$\boldsymbol{b := b - \alpha\, db}$$

The two main process in preparing neural network model are forward propagation to find loss and backward propagation to find the gradients.

Forward propagation



This the forward propagation to find loss. Now in order to find the gradient descent we need to find the gradients dw and db. In order to do that we use back propagation.

Back propagation

First find da i.e. $\frac{dL}{da} = \frac{d}{da} -y\log(a) - (1-y)\log(1-a)$

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

Now to find dz i.e $\frac{dL}{dz} = \frac{dL}{da} \times \frac{da}{dz}$

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

$$\frac{da}{dz} = \frac{d}{dz}\sigma(z)$$

$$= \sigma(z) \times (1-\sigma(z))$$

$$= a(1-a)$$

$$\therefore \frac{dL}{dz} = \frac{a-y}{a(1-a)} \times a(1-a) = \underline{\underline{a^{(i)} - y^{(i)}}} \quad (dz^{(i)})$$

$$dw_1 = x_1^{(i)} dz^{(i)}$$

$$dw_2 = x_2^{(i)} dz^{(i)}$$

$$db = dz^{(i)}$$

Final For loop for Finding gradient descent is: (for 2 features)

$J=0, dw_1=0, dw_2=0, db=0$

For i in range(m):

$$z^{(i)} = w^T x^{(i)} + b$$
$$a^{(i)} = \sigma(z^{(i)})$$
$$J(w,b) = -y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)})$$
$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dw_1 = x_1^{(i)} dz^{(i)}, \quad dw_2 = x_2^{(i)} dz^{(i)}, \quad db = dz^{(i)}$$

$J = J/m, \; dw_1 = dw_1/m, \; dw_2 = dw_2/m, \; db = db/m$

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

Vectorization

$$X = \begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^m \\ x_2^1 & x_2^2 & \cdots & x_2^m \\ x_3^1 & x_3^2 & \cdots & x_3^m \\ \vdots & \vdots & \cdots & \vdots \\ x_{nx}^1 & x_{nx}^2 & \cdots & x_{nx}^m \end{bmatrix}_{(n_x \times m)}$$

$$W = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_{nx} \end{bmatrix}_{(n_x \times 1)} \qquad W^T \; [w_1 \; w_2 \; w_3 \; \cdots \; w_{nx}]$$

np.dot(W^T, X) =

$$[x_1^1 w_1 + x_2^1 w_2 + x_3^1 w_3 + \cdots + x_{nx}^1 w_{nx}, \; x_1^2 w_1 + x_2^2 w_2 + \cdots + x_{nx}^2 w_{nx}, \; x_1^m w_1 + x_2^m w_2 + \cdots x_{nx}^m w_{nx}]_{(1 \times m)}$$

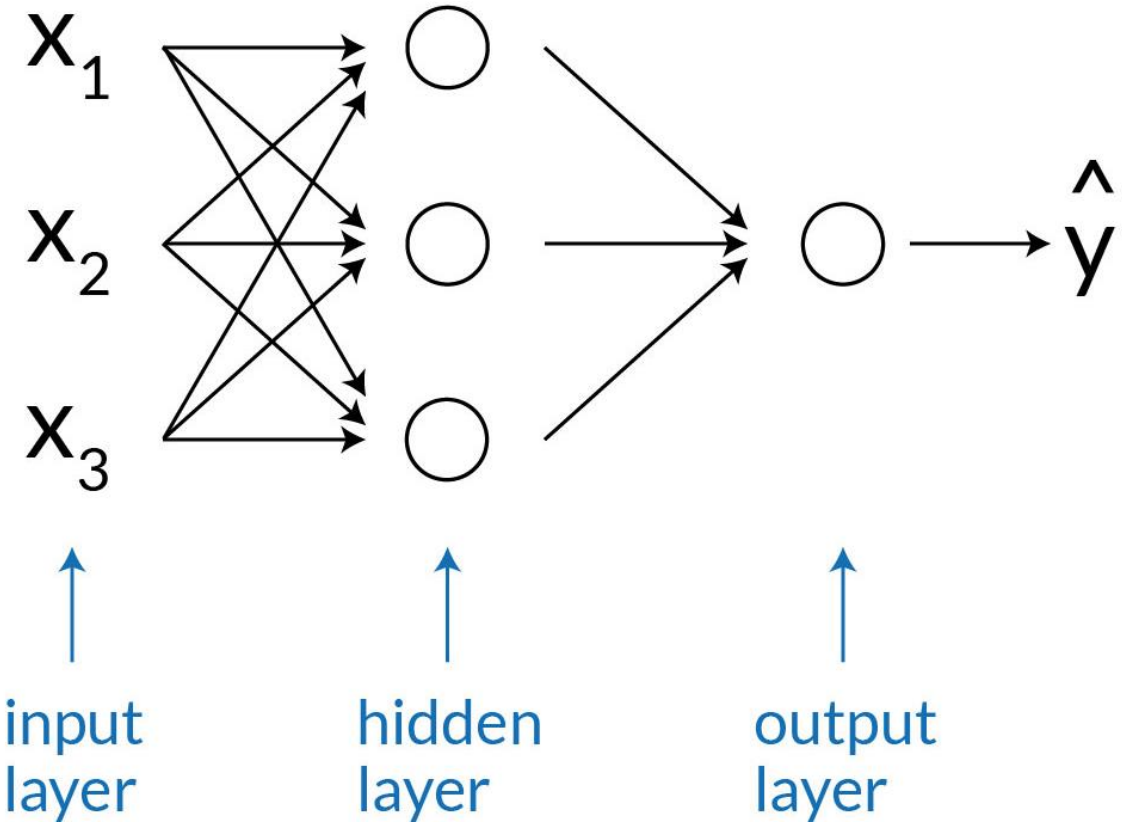
$W^T X + b =$ { b is scalar and is broadcasted to $(1,m)$ }

$$\left[\sum_{j=1}^{nx} x_j^1 w_j + b, \; x_j^2 w_j + b, \; x_j^3 w_j + b + \cdots + x_j^m w_j + b\right]_{(1 \times m)}$$

Then we update

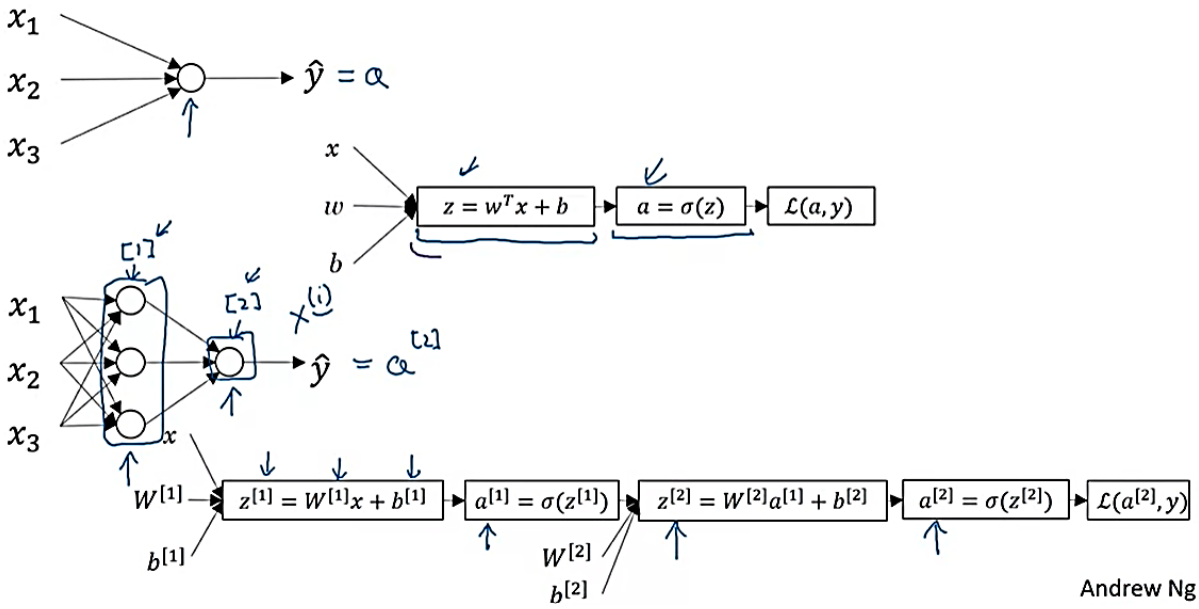
$$w_1 := w_1 - \alpha dw_1$$
$$w_2 := w_2 - \alpha dw_2$$
$$\vdots$$
$$w_{nx} := w_{nx} - \alpha dw_{nx}$$
$$b := b - \alpha db$$

WEEK 3 Shallow Neural Network



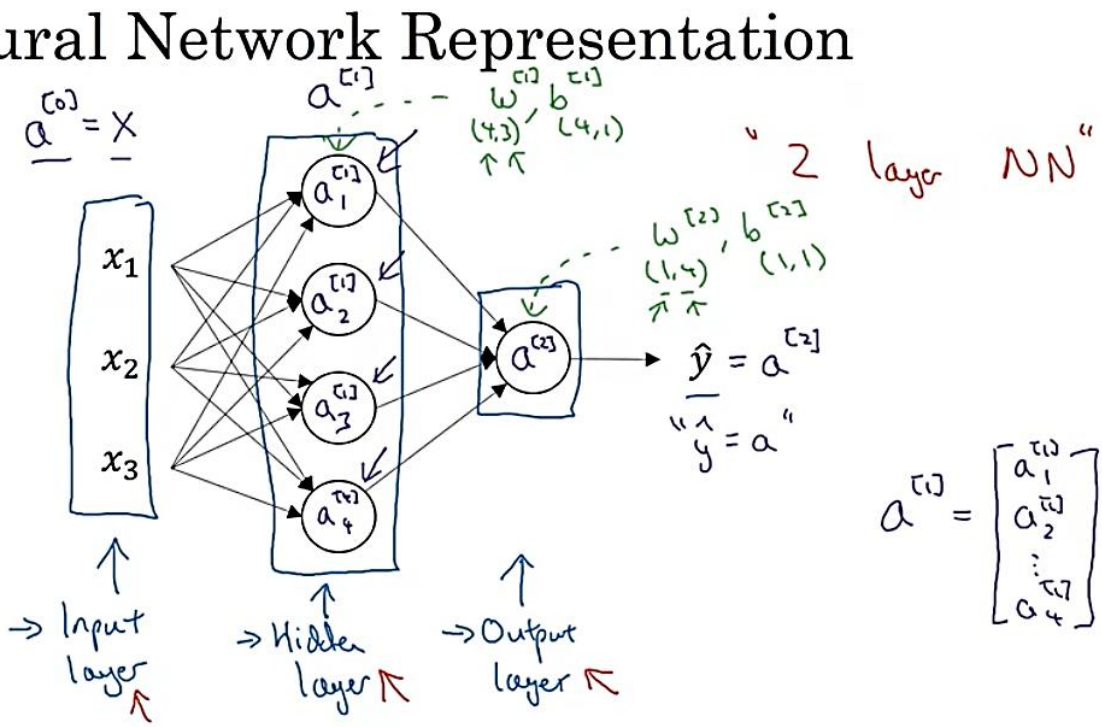
https://www.druva.com/blog/understanding-neural-networks-through-visualization/

Last, week we saw how to implement logistic regression in a neural network with input features X one neuron and output Y. This week we will see how to implement neural network with hidden layers.



As shown in the picture the hidden layers are represented using [ ] brackets. Instead of calculating one Z we now calculate more than one Z. First, we calculate Z1 using X, W1 and b and the activation function a we get is used as input to the next layer to calculate Z2.

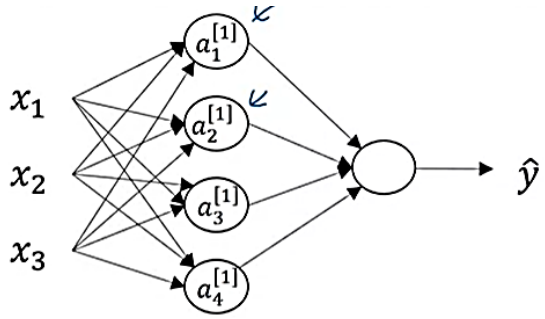
Just like we did forward propagation to compute loss function in week one here too we use forward propagation to calculate $\boldsymbol{L(a^{[2]}, y)}$ and back propagation to calculate derivatives $\boldsymbol{da^{[2]}, dz^{[2]}, dw^{[2]}, db^{[2]}, da^{[1]}, dz^{[1]}, dw^{[1]}, db^{[1]}}$



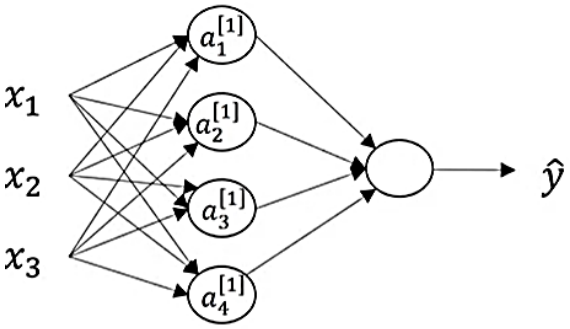
This neural network has three layers input layer ($\boldsymbol{a^{[0]}}$), hidden layer ($\boldsymbol{a^{[1]}}$) and output layer ($\boldsymbol{a^{[2]}}$). In this neural network the hidden layer has 4 neurons which takes 3 input values from X and parameters $\boldsymbol{w^{[1]}(4,3)}$ and $\boldsymbol{b^{[1]}(4,1)}$ where 4 is the number of neurons in the hidden layer, 3 is the number of input features. Each neuron gives output $\boldsymbol{a_1^{[1]}, a_2^{[1]}, a_3^{[1]}\ and\ a_4^{[1]}}$ which is then passed to the output layer with parameters $\boldsymbol{w^{[2]}(1,4)}$ and $\boldsymbol{b^{[2]}(1,1)}$ where 1 is the number of

output and 4 is the number of inputs. The output neuron gives output $\boldsymbol{a_1^{[2]}}$ which is a scalar. This neural network is also called two-layer NN without considering the input layer.

Vectorization



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$



Given input x:

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

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

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

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

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

$$z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}_{4\times1} \quad , \quad w^{[1]} = \begin{bmatrix} — w_1^{[1]} — \\ — w_2^{[1]} — \\ — w_3^{[1]} — \\ — w_4^{[1]} — \end{bmatrix}_{4\times3}$$

Superscript is the layer no.

here subscript is for the number of neuron (node no)

$$w_i^{[1]} = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}_{1\times3}$$

here subscript is the number of features

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{3\times1} \qquad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_1^{[2]} \\ b_1^{[3]} \\ b_1^{[4]} \end{bmatrix}_{4\times1}$$

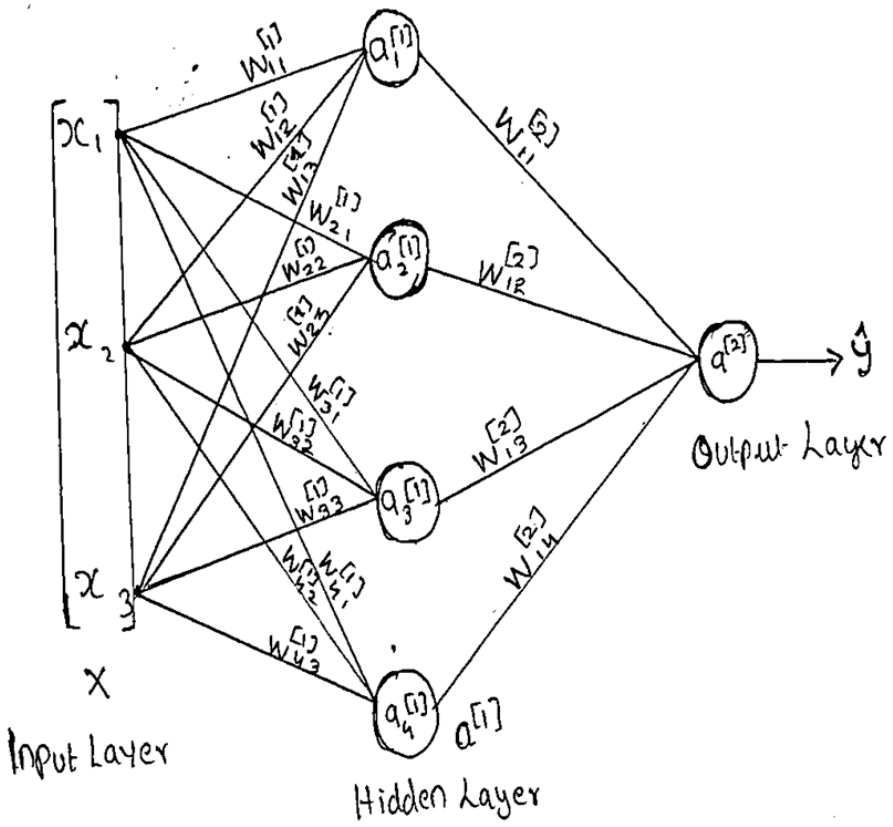
2. $a^{[1]} = \sigma(z^{[1]})$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} = \sigma(z_1^{[1]}) \\ a_2^{[1]} = \sigma(z_2^{[1]}) \\ a_3^{[1]} = \sigma(z_3^{[1]}) \\ a_4^{[1]} = \sigma(z_4^{[1]}) \end{bmatrix}_{4\times1}$$

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

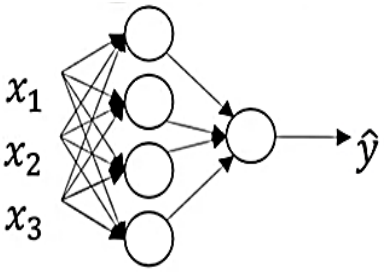
$w^{[2]} = \begin{bmatrix} w_{11}^{[2]} & w_{12}^{[2]} & w_{13}^{[2]} & w_{14}^{[2]} \end{bmatrix}_{1 \times 4}$ # 1i - means the weight of ith feature of 2nd layer and it has only one output node

$b^{[2]} = \begin{bmatrix} b_1^{[2]} \end{bmatrix}$, $z^{[2]} = \begin{bmatrix} z_1^{[2]} \end{bmatrix}$, $a^{[2]} = \begin{bmatrix} \sigma(z_1^{[2]}) \end{bmatrix}$



All these examples are for single training sample

Now we will see the implementation of neural network for multiple training samples



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

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

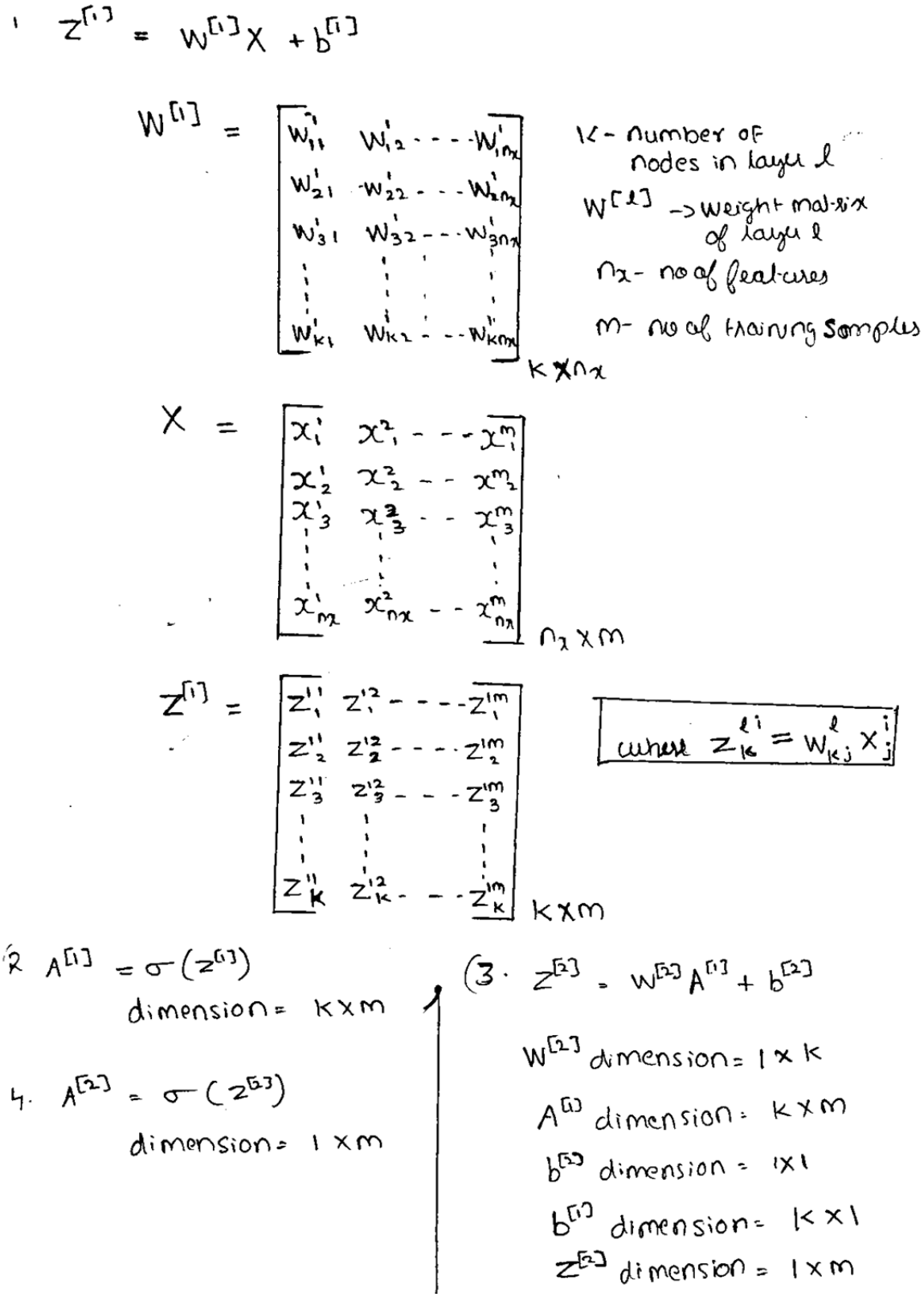
```
for i = 1 to m
```

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

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

When we include all the training samples (i) would represent the ith training sample.

Vectorization

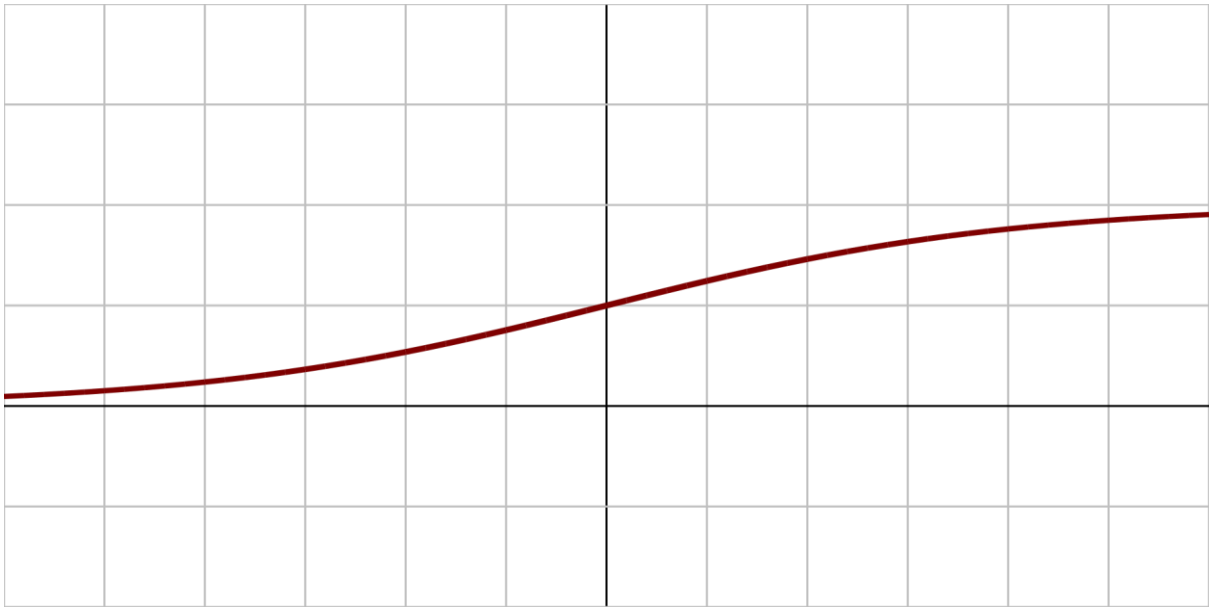


Please note that the notation k used here is just used for explaining this particular vectorization and should not be used as the correct notation. We would see the right notation for the number of nodes later in the course

Now let's talk about activation functions and decide which one is best to use.

Activation functions are generally represented as g ($z^{[l]}$) where l is the number of layers.

Sigmoid $(\frac{1}{1+e^{-z}})$



By Laughsinthestocks - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=44920533

Sigmoid functions are functions which are shaped like s across y-axis. This particular sigmoid function is the logistic function which when given input values gives an **output** between **0 and 1**. This is useful when we want to predict probabilities and can be used in the last neuron in binary classification as an activation function. If the input value is very large and close to ∞ we get output as 1, if it is very small and close to −∞ we get output as 0. In deep learning we generally do not use sigmoid function for the hidden layer neurons as activation function because the slope of sigmoid function is close zero as it starts to flatten out on both the ends and might affect the learning rate.

Derivative

$$g(z) = \frac{1}{1+e^{-z}} = a$$

$$g'(z) = \frac{d}{dz} g(z) = \frac{d}{dz} \frac{1}{1+e^{-z}} = \frac{1+e^{-z} \times \frac{d}{dz} 1 - 1 \times \frac{d}{dz} 1+e^{-z}}{(1+e^{-z})^2}$$

$$= \frac{1+e^{-z} \times 0 - \left[1 \times \left(\frac{d}{dz} 1 + \frac{d}{dz} e^{-z}\right)\right]}{(1+e^{-z})^2}$$

$$= \frac{0 - 1 \times (0 - e^{-z})}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2}$$

$$= \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1}{(1+e^{-z})} \times \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{(1+e^{-z})}\right)$$

$$= \frac{1}{1+e^{-z}} \times \left(1 - \frac{1}{1+e^{-z}}\right)$$

$$= g(z) \times (1 - g(z)) = \underline{\underline{a(1-a)}}$$

Case 1: $z = 10$

$g(z) \approx 1$, $g'(z) \approx 1 \times (1-1) \approx 0$

∴ as z increases slope nears to 0 $z \uparrow$ slope ≈ 0

Case 2: $z = -10$

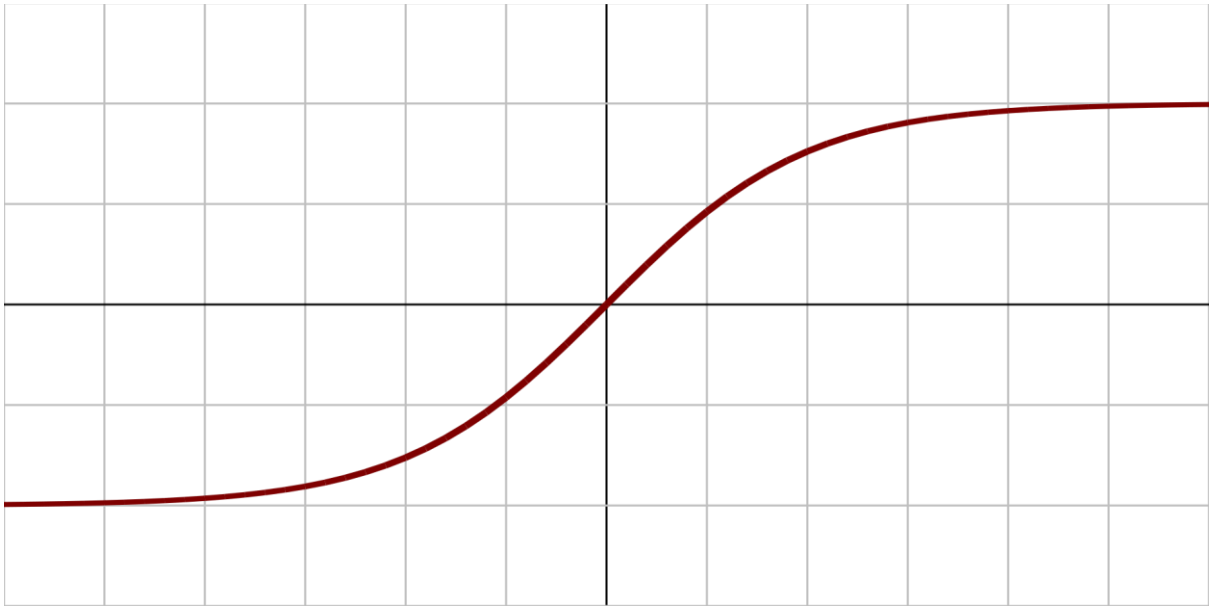
$g(z) \approx 0$, $g'(z) \approx 0 \times (1-0) \approx 0$

∴ as z is very low slop nears to 0 $z \downarrow$ Slop ≈ 0

Case 3: $z = 0$

$g(z) = \frac{1}{2}$, $g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$

Hyperbolic Tangent ($\frac{e^z - e^{-z}}{e^z + e^{-z}}$)



Hyperbolic tangent function is a sigmoid function with **range** between **-1 and 1**. It is superior function than sigmoid since it is centred at 0 rather than a t 0.5. Often, we use g ($z^{[l]}$) = tanh as it gives values between -1 and 1 whose mean is zero. The reason is that sometimes in a learning algorithm we centre the data. So, tanh will help in centring the data around 0 rather than around 0.5 which would help the next hidden layer when passed in as input. So, we can use different activation functions for different layers.

Derivative

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = \frac{d}{dz}\left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)$$

$$= \frac{\left(e^z + e^{-z}\right)\left(\frac{d}{dz}e^z - \frac{d}{dz}e^{-z}\right) - \left(e^z - e^{-z}\left(\frac{d}{dz}e^z + \frac{d}{dz}e^{-z}\right)\right)}{(e^z + e^{-z})^2}$$

$$= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z}) \times (e^z - e^{-z})}{(e^z + e^{-z})^2}$$

$$= \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$

$$= 1 - \left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)^2 = \underline{1 - (\tanh)^2}$$

$$\therefore g'(z) = \underline{1 - (g(z))^2 = 1 - a^2}$$

Case 1: $z = 10$

$g(z) \approx 1$, $g'(z) \approx 0$ $(1 - 1)$

$\therefore$ slope is 0 for large z

Case 2: $z = -10$

$g(z) \approx -1$, $g'(z) \approx 0$ $(1 - (-1)^2)$

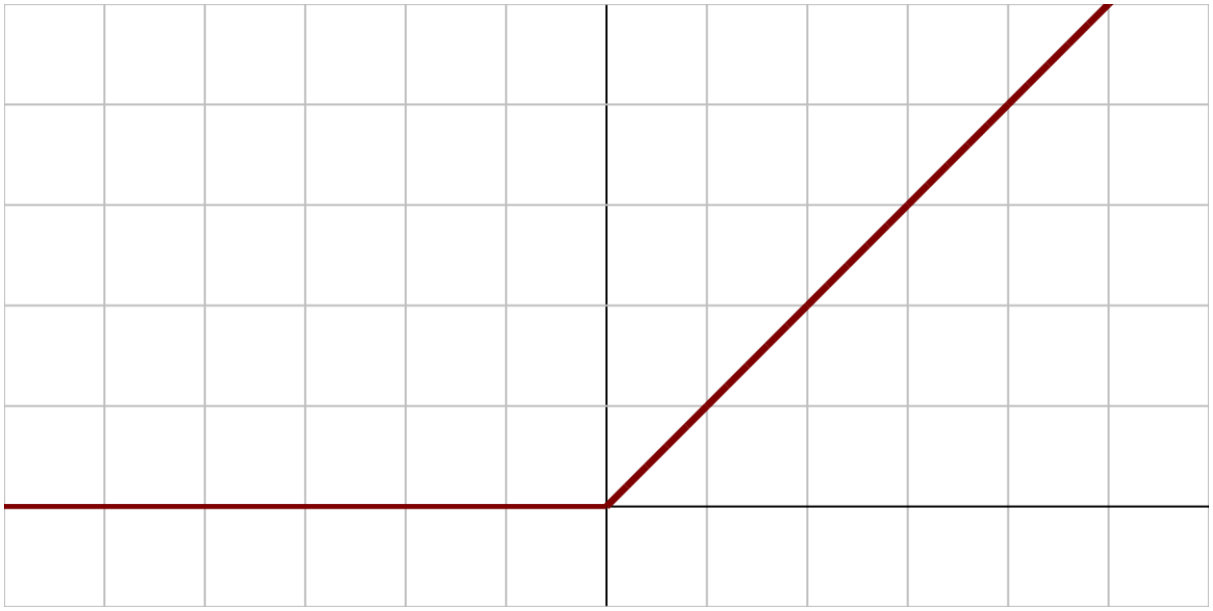
$\therefore$ slope is 0 for larg -ve z

Case 3: $z = 0$

$g(z) = 0$, $g'(z) = 1$

ReLU

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$



The **Re**ctified **L**inear **U**nit is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning. As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

Range: [0 to infinity)

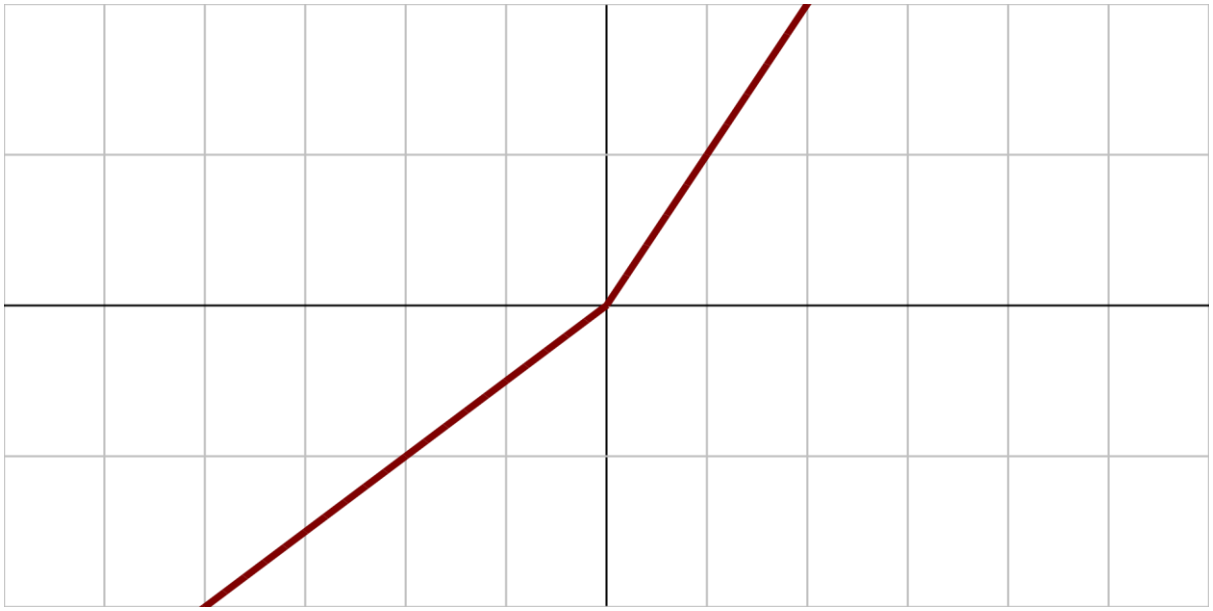
The function and its derivative **both are monotonic**.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately. The differential of ReLU lies between 0 and 1.

Leaky ReLU

$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

It is an attempt to solve the dying ReLU problem



By Laughsinthestocks at English Wikipedia - Transferred from en.wikipedia to Commons., CC0, https://commons.wikimedia.org/w/index.php?curid=46839644

f(x) = ax. When x>0 a =1, x<0 a = 0.01

The leak helps to increase the range of the ReLU function. Usually, the value of **a** is 0.01 or so.

When **a is not 0.01** then it is called **Randomized ReLU**.

Therefore, the **range** of the Leaky ReLU is (-∞ to ∞).

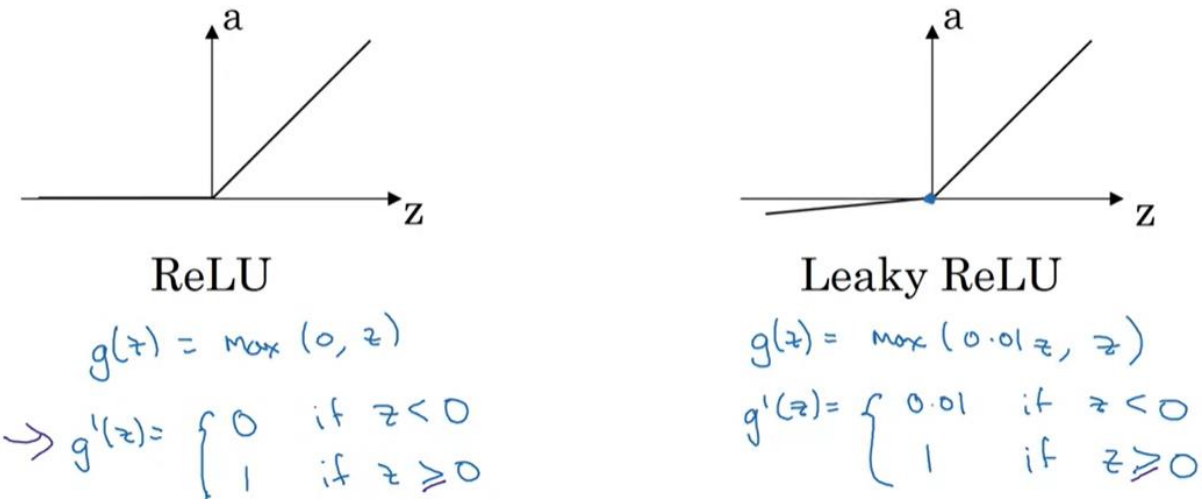
Both Leaky and Randomized ReLU functions are monotonic in nature. Also, their derivatives also monotonic in nature.

ReLU is the activation function that we mostly use in deep learning.

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Derivative of ReLU and Leaky ReLU

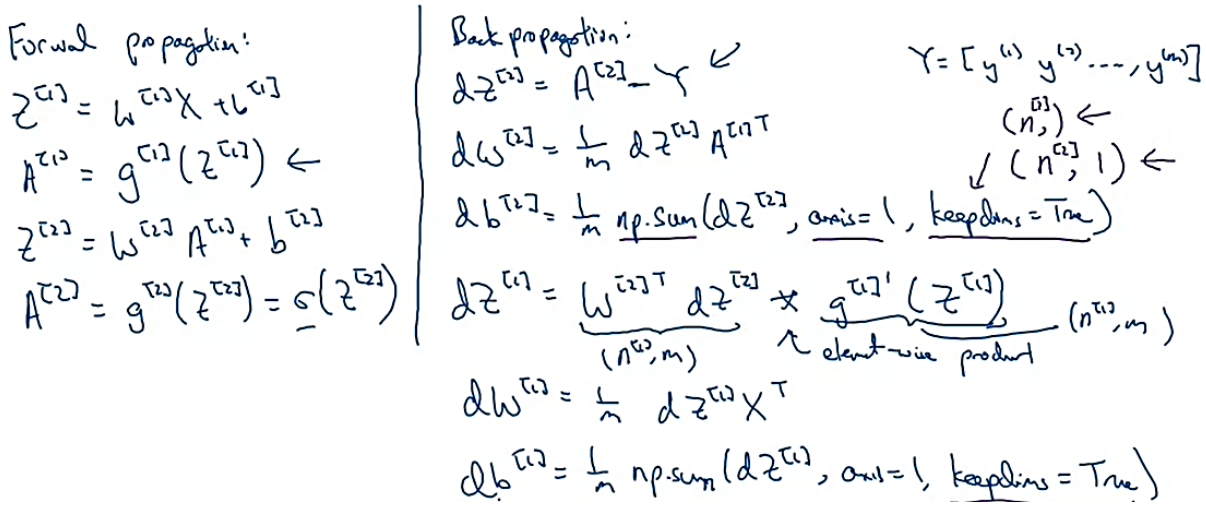
Recommend watching course video to understand better



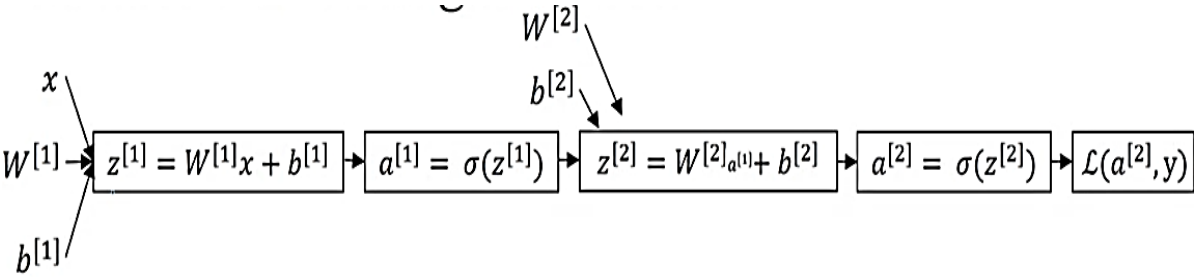
why do we need a non-linear activation function?

Without the non-linear activation function the neural network behaves just like a linear regression function. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks. without a non-linear function doesn't matter how many hidden layers we attach in the neutral net all will behave in the same way. Neuron cannot learn with just a linear function attached to it; it requires a non-linear activation function to learn as per the difference w.r.t error. So as the whole point of using a neural network is to get more features out of the data which intern help in better prediction won't be served if we use a linear function as activation function

Gradient Descent



Forward Propagation



Back propagation

For one training sample	For m training samples
$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]}a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]}x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Explanation

Please read the below notes carefully. These are important notes for doing program. Please consider watching videos before reading this.

Dimensions

$X = (n^0, m)$, $W^{[1]} = (n^1, n^0)$, $b^{[1]} = (n^1, 1)$

$Z^{[1]} = (n^1, m)$, $A^{[1]} = (n^1, m)$, $Y = (n^2, m)$

$W^{[2]} = (n^2, n^1)$, $b^{[2]} = (n^2, 1)$, $Z^{[2]} = (n^2, m)$

$A^{[2]} = (n^2, m)$

\# in this example $n^2 = 1$

$dz^{[2]} = (n^2, m)$; $db^{[2]} = (n^2, 1)$

$dw^{[2]} = (n^2, n^1)$

$dz^{[1]} = (n^1, m)$, $dw^{[1]} = (n^1, n^0)$

$db^{[1]} = (n^1, 1)$

For binary classification $n^2 = 1$ if we use 1 hidden layer

Now we will see how to calculate the backpropagation for gradient descent

$$dz^{[2]} = \frac{dL}{dz^{[2]}} = \frac{da^{[2]}}{dz^{[2]}} \times \frac{dL}{da^{[2]}}$$

$$\frac{dL}{da^{[2]}} = \frac{d}{da^{[2]}}\left(-y \log(a^{[2]}) - (1-y)\log(1-a^{[2]})\right)$$

$$\frac{dL}{da^{[2]}} = \frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

$$\frac{dL}{da^{[2]}} = \frac{-y + ya + a - ya}{a(1-a)} = \frac{a-y}{a(1-a)}$$

$$\frac{da^{[2]}}{dz^{[2]}} = \frac{d}{dz^{[2]}}\left(\frac{1}{1+e^{-z^{[2]}}}\right)$$

$$\frac{da^{[2]}}{dz^{[2]}} = \frac{(1+e^{-z^{[2]}}) \times \frac{d}{dz^{[2]}}(1) - 1 \times \frac{d}{dz^{[2]}}(1+e^{-z^{[2]}})}{(1+e^{-z^{[2]}})^2}$$

$$\frac{da^{[2]}}{dz^{[2]}} = \frac{0 - (0 - e^{-z^{[2]}})}{(1+e^{-z^{[2]}})^2} = \frac{e^{-z^{[2]}}}{(1+e^{-z^{[2]}})^2}$$

$$\frac{da^{[2]}}{dz^{[2]}} = \frac{1+e^{-z^{[2]}} - 1}{(1+e^{-z^{[2]}})^2} = \frac{1}{(1+e^{-z^{[2]}})}\left(1 - \frac{1}{(1+e^{-z^{[2]}})}\right)$$

$$\frac{da^{[2]}}{dz^{[2]}} = a^{[2]}(1-a^{[2]})$$

$$dz^{[2]} = \frac{dL}{dz^{[2]}} = a^{[2]}(1-a^{[2]}) \times \frac{a^{[2]} - y}{a^{[2]}(1-a^{[2]})}$$

$$= a^{[2]} - Y = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}, \quad db^{[2]} = \frac{1}{m}\sum_{i=1}^{m} dz^{[2]}$$

$$dz^{[1]} = \frac{dL}{dz^{[1]}} = \frac{dL}{da^{[1]}} \times \frac{da^{[1]}}{dz^{[1]}}$$

$$\frac{dL}{da^{[1]}} = \frac{dz^{[2]}}{da^{[1]}} \times \frac{dL}{dz^{[2]}}$$

$$\frac{dL}{da^{[1]}} = \frac{d}{da^{[1]}}\left(w^{[2]}a^{[1]} + b^{[2]}\right) \times dz^{[2]}$$

$$da^{[1]} = w^{[2]T} \times dz^{[2]}$$

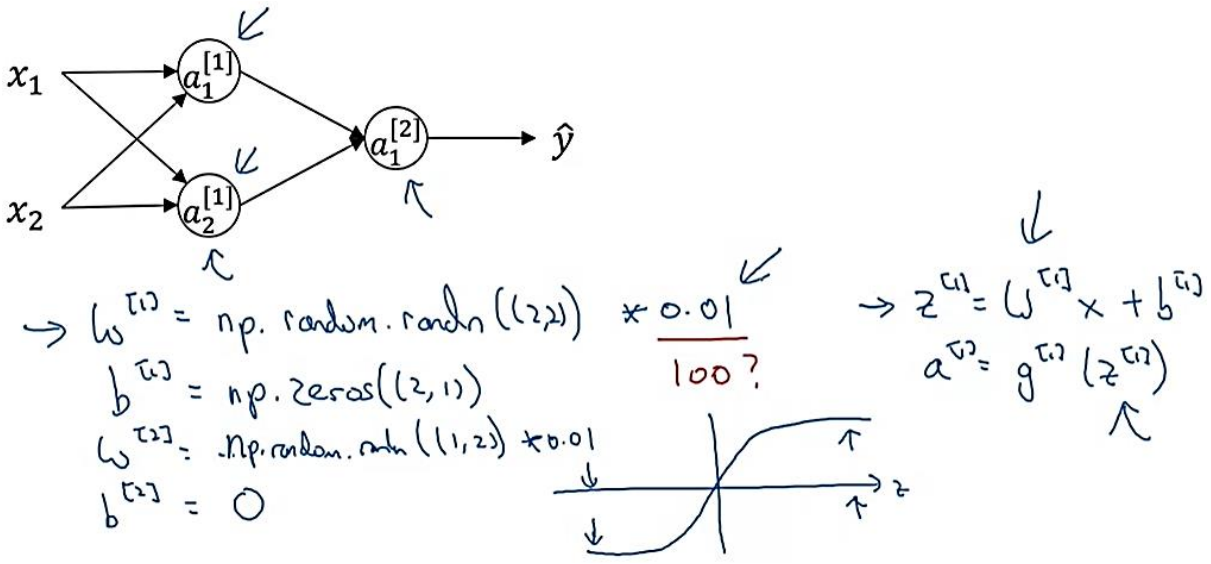
$$\frac{da^{[1]}}{dz^{[1]}} = \frac{d}{dz^{[1]}} g^{[1]}(z^{[1]}) = g^{[1]\prime}(z_1)$$

$$\therefore dz^{[1]} = \frac{dL}{dz^{[1]}} = w^{[2]T} dz^{[2]} \times g^{[1]\prime}(z_1)$$

→ when we write program give $w^{[2]}$ T to match the dims

$$dw^{[1]} = \frac{1}{m} dz^{[1]} x^T, \quad db^{[1]} = \frac{1}{m}\sum_{i=1}^{m} dz^{[1](i)}$$

Random Initialisation

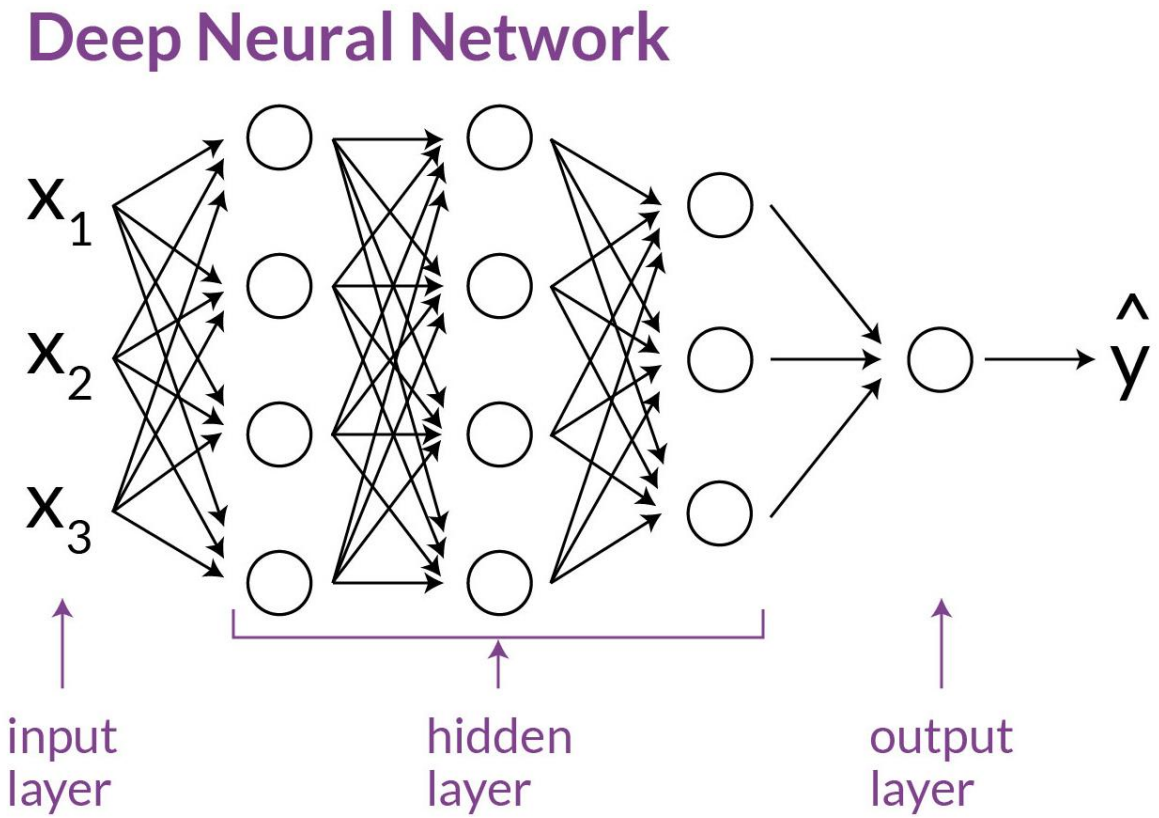


If we **initialise** the parameters as **zero** or matrix of zeros then we would end up calculating the **same activation function** for each node different layers. The whole point of introducing hidden units is

to get new features to the data, so if the **features** we get are all **same** in all nodes in a layer then that won't serve the purpose. So, we use random numbers to initialize the parameters so that each weight is different for different features so that the new features that we get from the nodes won't be equal to each other as shown in figure above.

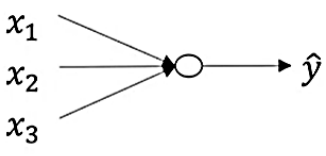
If we multiply the randomly generated number by 100 rather than 0.01 then we would end up calculating a large value for Z and if our activation function is tanh or any sigmoid function then for large z we would end up at the flat part of the function where the slope would be close to zero so learning process would be really slow.

WEEK 4 Deep Neural Network

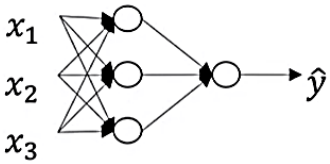


https://www.druva.com/blog/understanding-neural-networks-through-visualization/

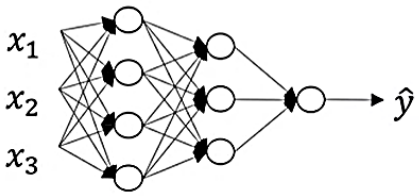
What is a deep neural network?



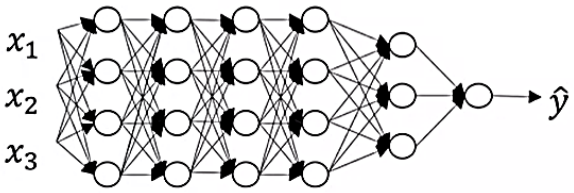
logistic regression



1 hidden layer



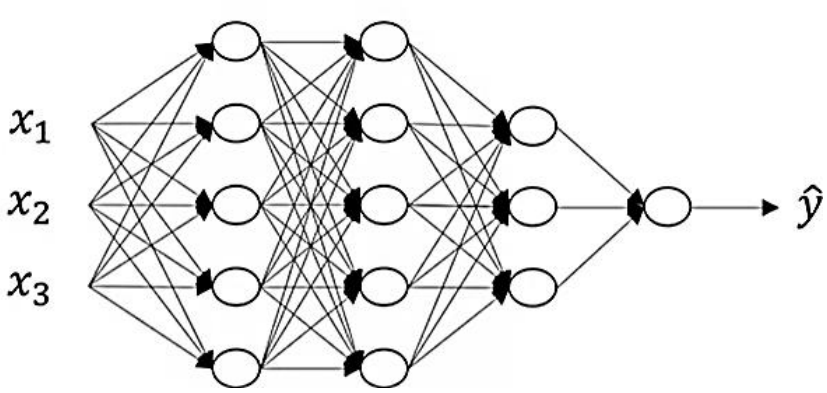
2 hidden layers



5 hidden layers

A deep NN is an NN with more than two or more hidden layer. Deep neural networks can learn some functions that shallower NN can't.

Deep neural network notation



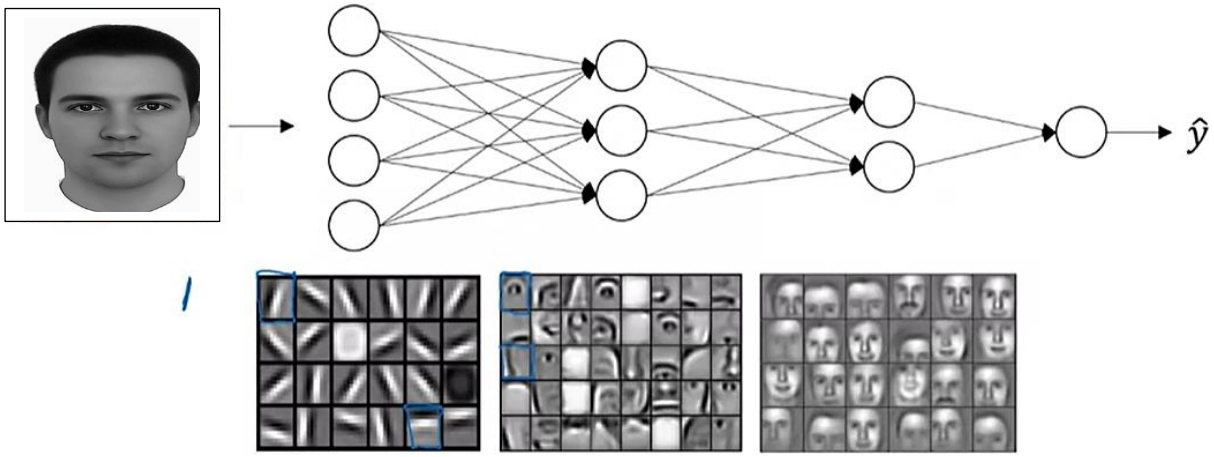
$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$

$A^{[l]} = g^{[l]}(Z^{[l]})$ → Activation matrix of layer l

$W^{[l]}, b^{[l]}$ → Parameters for layer l

$n^{[l]}$ → Number of hidden units in layer l

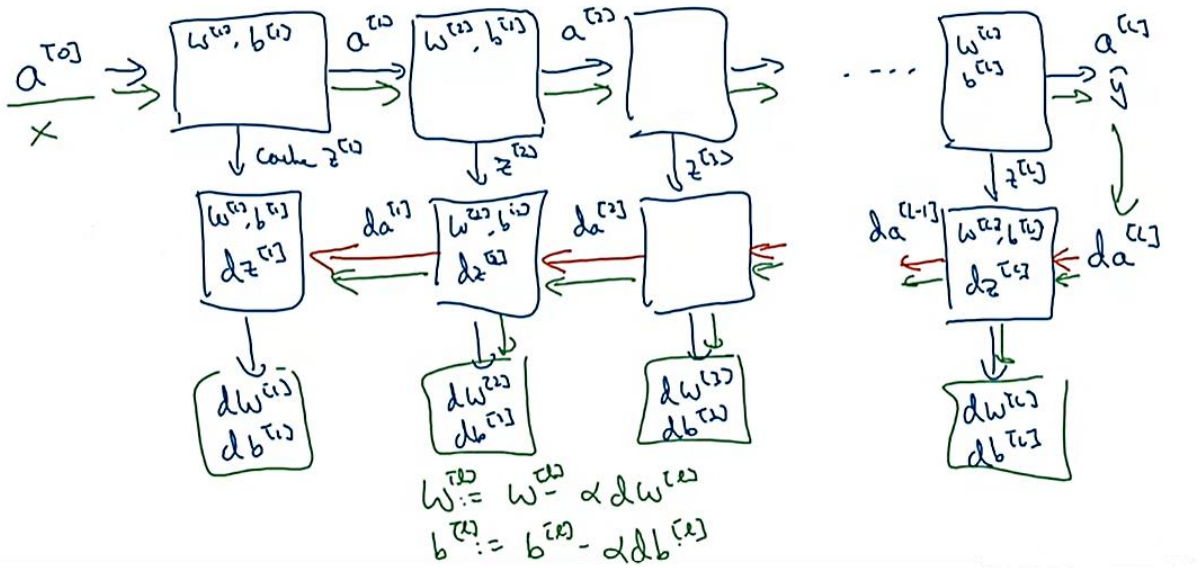
Why we use Deep Neural Networks?



In this example we have given images as input to the neural network. The first layer might detect features like different kinds of

edges, the second layer might detect features like nose, eyes, etc. the third layer might detect features like different faces. So, by having more than one layer we are able to get more useful features form the images.

Building blocks of NN



In forward propagation we input $A^{[l-1]}$ with parameters $W^{[l]}$ and $b^{[l]}$ and gets output $A^{[l]}$ also we store $Z^{[l]}$ in cache to be used for backpropagation. We continue this till we get $A^{[L]}$ which is equal to $\hat{y}$. In backward propagation we give $dA^{[l]}$ with $W^{[l]}$ and $b^{[l]}$ and gets $dA^{[l-1]}$ as output and we calculate $dW^{[l]}$ and $db^{[l]}$.

Forward Propagation

For l = 1 to L:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

$$\hat{y} = A^{L}$$

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]}) = \hat{Y}$$

Back Propagation

For l = L to 1:

$$dz^{[l]} = \frac{dL}{dz^{[l]}} = \frac{dL}{dA^{[l]}} \times \frac{dA^{[l]}}{dz^{[l]}}$$

$$= dA^{[l]} \times g^{[l]\prime}(z^{[l]})$$

$$\begin{cases} \frac{dA^{[l]}}{dz^{(l)}} = \frac{d}{dz^{(l)}} g(z^{(l)}) \end{cases}$$

$$\text{New. } dA^{[l]} = \frac{dz^{[l+1]}}{dA^{[l]}} \times \frac{dL}{dz^{[l+1]}}$$

$$= \frac{d}{dA^{[l]}} \left(W^{[l+1]} A^{[l]} + b^{[l+1]} \right) \times dz^{[l+1]}$$

$$dA^{[l]} = W^{[l+1]} \times dz^{[l+1]}$$

$n^{[l+1]} \times n^{l} \cdot n^{[l+1]} \times m$ which is not possible

$$\therefore dA^{[l]} = \left(W^{[l+1]} \cdot T \right) \cdot dz^{[l+1]}$$

$$\therefore dz^{[l]} = \frac{dL}{dz^{[l]}} = \left(\left(W^{[l+1]} \cdot T \right) \cdot dz^{[l+1]} \right) * g^{[l]\prime}(z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dz^{[l]} \cdot A^{[l-1]} \cdot T$$

$$db^{[l]} = \frac{1}{m} np.sum(dz^{[l]}, axis=1, keepdims = True)$$

then $$dA^{[l-1]} = \left(W^{[l]} \cdot T \right) \cdot dz^{[l]}$$

*Note

For l = L, $dA^{[l]} = \frac{dL}{dA^{[L]}} = -\frac{y}{a} - \frac{1-y}{1-a}$ if we are using sigmoid as output activation function and there is no $dZ^{[l+1]}$ to calculate.

To generalize:

$$dZ^{[L]} = A^{[L]} - Y$$

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]^T}$$

$$db^{[L]} = \frac{1}{m} np.sum(dZ^{[L]}, axis = 1, keepdims = True)$$

$$dZ^{[L-1]} = W^{[L]^T} dZ^{[L]} * g'^{[L-1]}(Z^{[L-1]})$$

Note that * denotes element-wise multiplication)

$$\vdots$$

$$dZ^{[1]} = W^{[2]} dZ^{[2]} * g'^{[1]}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]^T}$$

Note that $A^{[0]^T}$ is another way to denote the input features, which is also written as X^T

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

Parameters v/s Hyperparameters

Parameters

$$W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]} \ ...$$

Hyperparameters

Learning rate α → Decides the rate at which gradients the parameters change.

No of iterations → Decides no of times gradient descent Is done to adjust the parameters.

No of hidden layers l → Decides the number of variety of features.

No of hidden units $n^{[l]}$ → Decides the number of activation units.

Choice of activation function → Decides the nature of features produced in the NN units.

All the hyperparameters are tuned to decide which ones are best to get the best results. It's very rare to have knowledge of the best value for all these parameters in advance. So, the best way to know is to do multiple trial and error steps and find out which ones minimizes the cost v/s iteration curve faster and perfectly. We can also check for accuracy score.