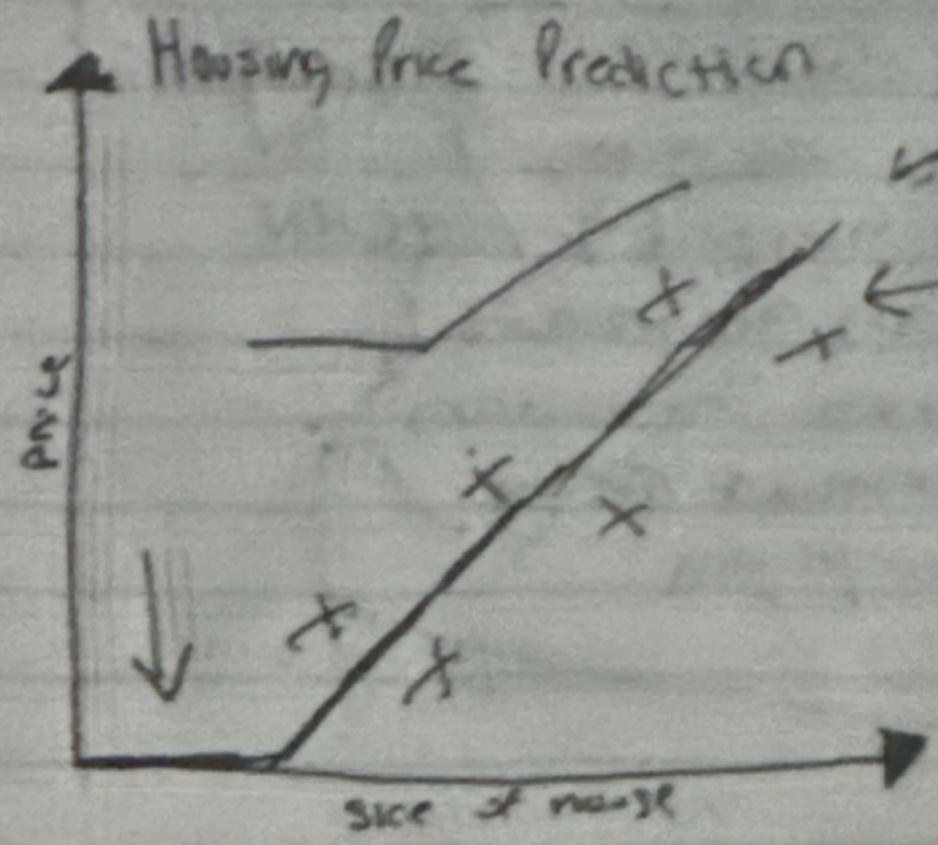
What is a neural network?



"neuron"

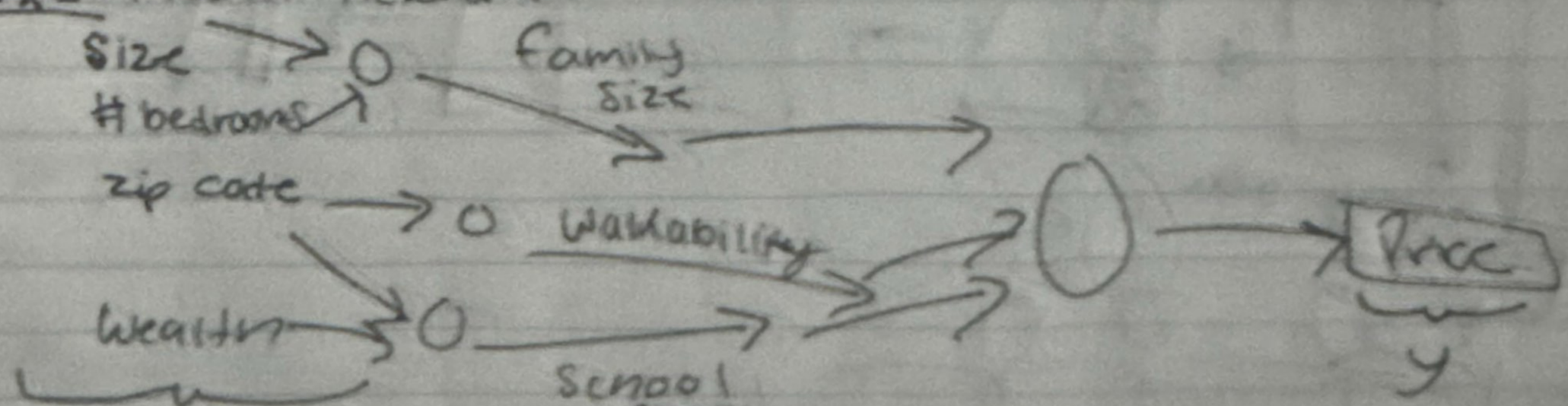
Size x → ○ → Price y

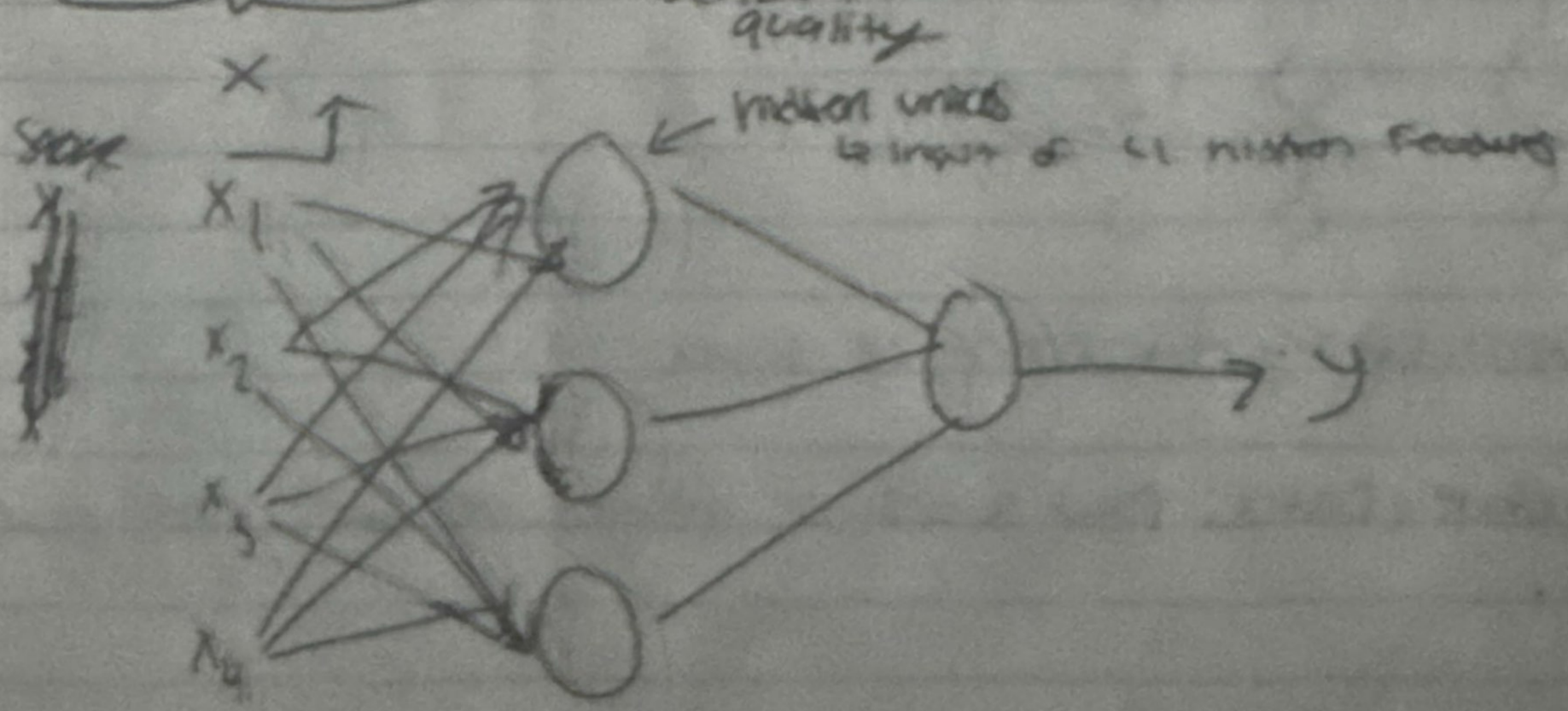
take a number

Relu function: Rectified Linear Unit

↳ That singular neuron is a logic rail and a neural network is stacking them.

Ex2 Neural Network





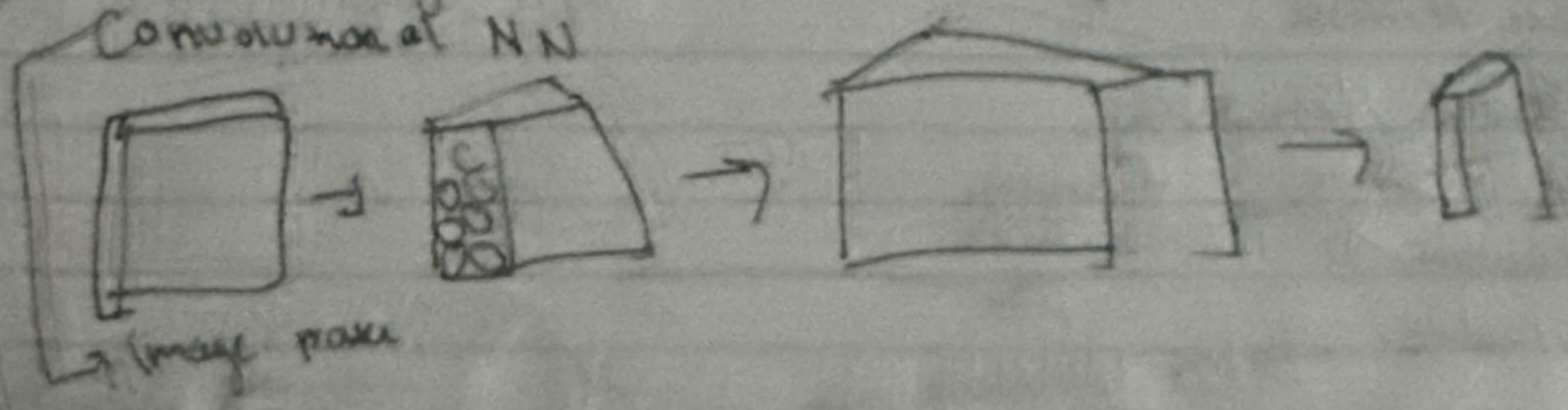
Neural networks are great at accurately mapping as functions from (x, y).

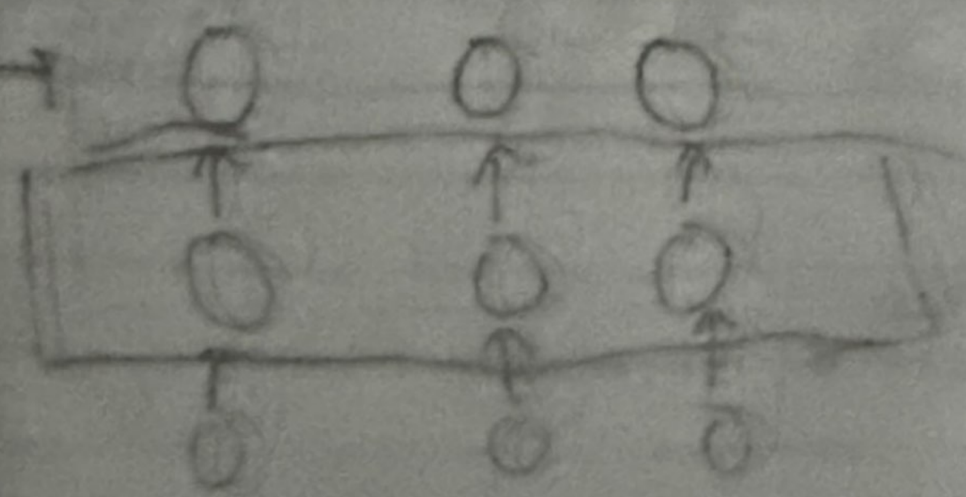
4 Supervised Learning with neural networks

Input (x)	Output (y)	Application	
Home feat	Price	Real estate	Standard NN
Ad, user info	Click on ad?	Online Advertising	
Image	Object (1 — 1000)	Photo tagging	CNN
Audio	Text transcript	Speech recognition	
English	Chinese	Machine Translation	
		Autonomous Driving	
		Computer vision	

RNN - Recurrent Neural Network

Convolutional NN



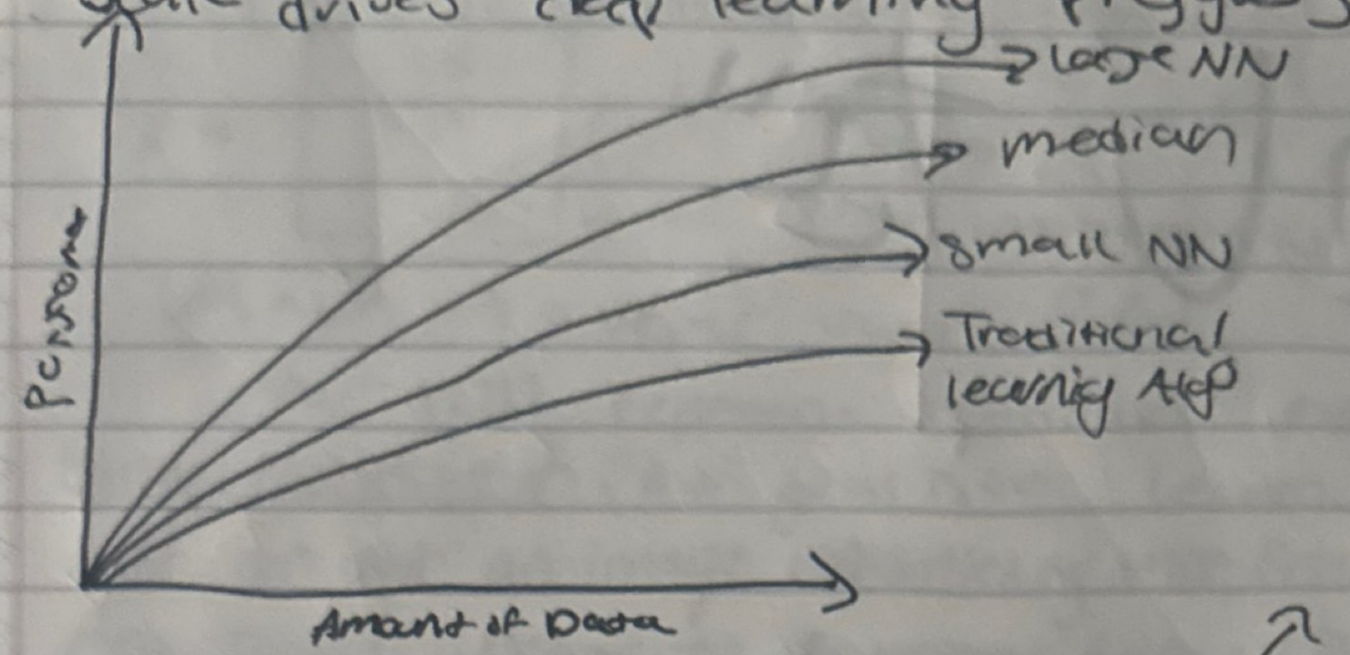


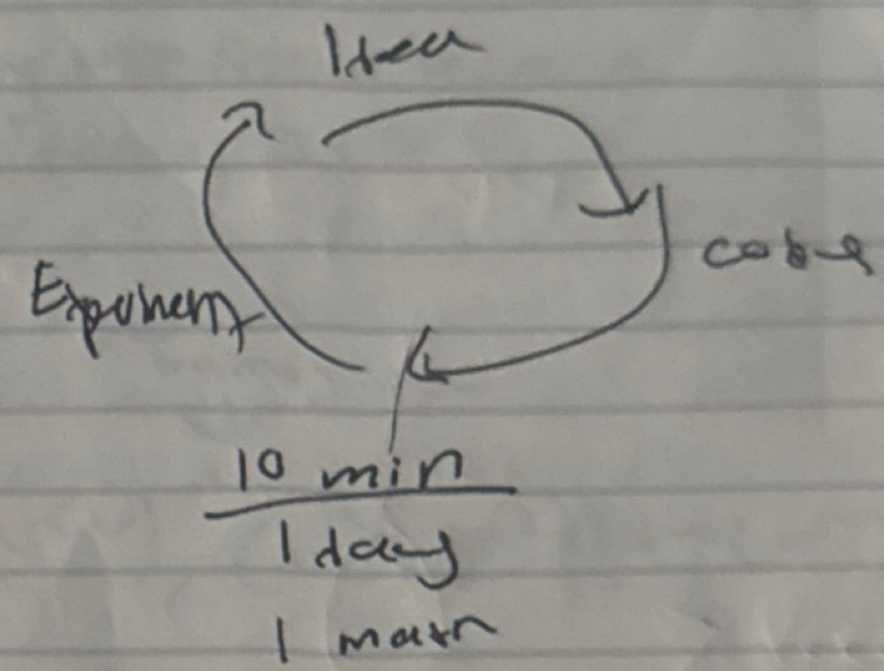
Structured data - databases of data

Unstructured Data: Raw audio or images or text like pixel values

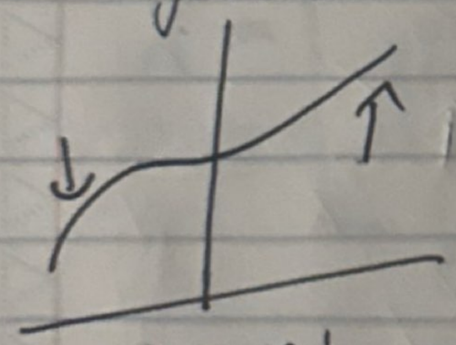
Why is Deep Learning taking off?

Scale drives deep learning progress



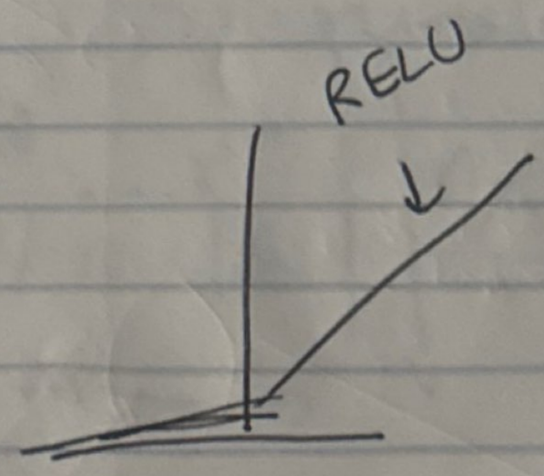


- Data
- Computation
- Algorithms



Sigmoid function

→ learning becomes really slow



Relu function

the gradient to less likely to gradually switch to 0.

What will we learn?
4 weeks on

Week 2 - Neural Networks and Deep Learning

Binary Classification

Example 1(cat) vs 0 (non cat)

y = denote the output

$$X = \begin{bmatrix} \vdots & \vdots & \\ \vdots & \vdots & \\ \vdots & \vdots & \end{bmatrix}$$ ← Feature Vector

pixel intensity value

64 × 64 × 3 = total value RGB

$n_x = 12288$ represents dim of the input feature

<u>Binary classification</u>: our goal is to learn a classifier that can input an image represented by this feature vector x. And predict whether the corresponding label y is 1 or 0.

Notation

(x, y) $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$ Training set

m - training examples : $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \ldots (x^{(m)}, y^{(m)})\}$

m = m train

m test = # of test examples

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \updownarrow n_x \qquad Y = \begin{bmatrix} y^{(1)}, y^{(2)} \ldots \end{bmatrix}$$

← m →

$Y \in \mathbb{R}^{1 \times m}$

$Y \in \mathbb{R}^{n_x \times m}$

X.shape → Python command

(nx, m)

Y.shape = (1, m)

8 Logistic Regression

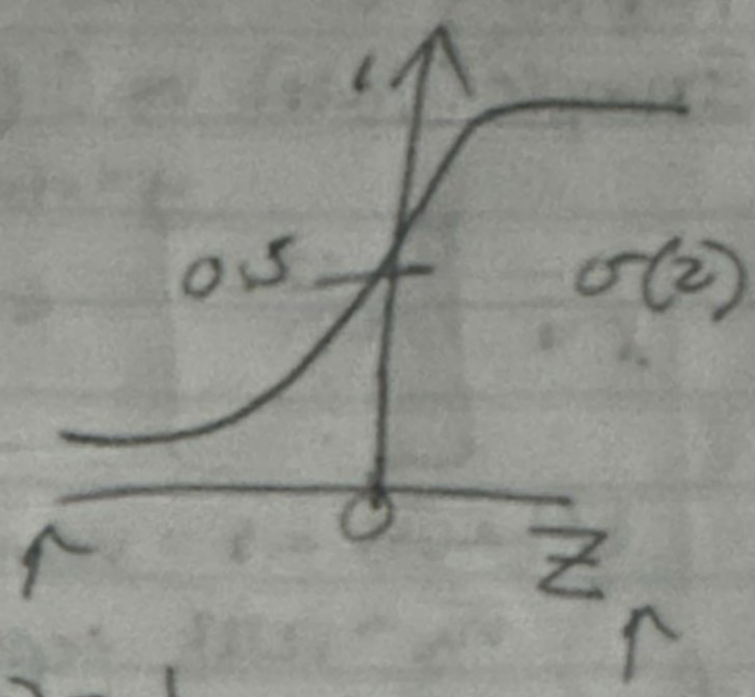
Given x, want $\hat{y} = P(y=1|x)$ ← you estimate of y

$0 \le \hat{y} \le 1$

$x \in \mathbb{R}^{n_x}$

Parameters: $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$

Output $\hat{y} = \sigma(\underbrace{w^T x + b}_{z})$



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

if z Large $\sigma(z) \approx \frac{1}{1+0} = 1$

$\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{Big num}}$

Implementing Linear Reg

Your job is to learn parameters w and b so that $\hat{y}$ is a good estimate of the chance of y being equal to one.

* Keep w & b Seperate It's easier

$x_0 = 1, \; x \in \mathbb{R}^{n_x}$

$\hat{y} = \sigma(\theta^T x)$

$\theta \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \\ \} w \end{matrix}$

← We will not using notation

Linear Regression cost function

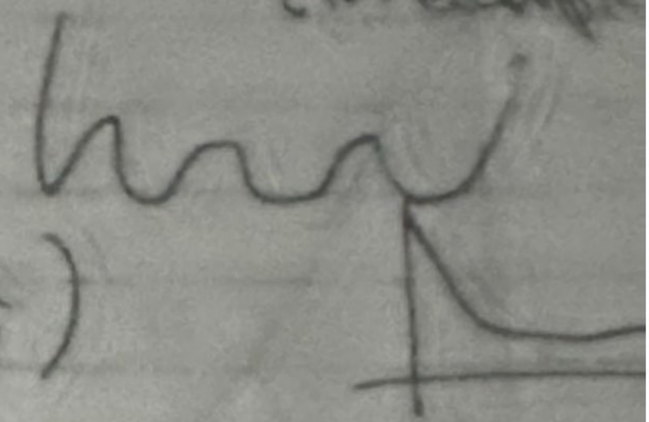
output $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$ & $z^{(i)} = w^T x^{(i)} + b$

Given $\{(x^{(1)}, y^{(1)}), \ldots (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

Single training example

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

Squared ERR



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

If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ← want $\log \hat{y}$ large

~~If y = 0:~~

want $\hat{y}$ large

If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← want $1-\hat{y}$ large

want $\hat{y}$ small

Cost function: $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

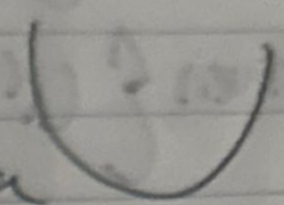
cost of your parameters.

$y^{(i)} \log \hat{y}^{(i)}$

$(1-y^{(i)})$

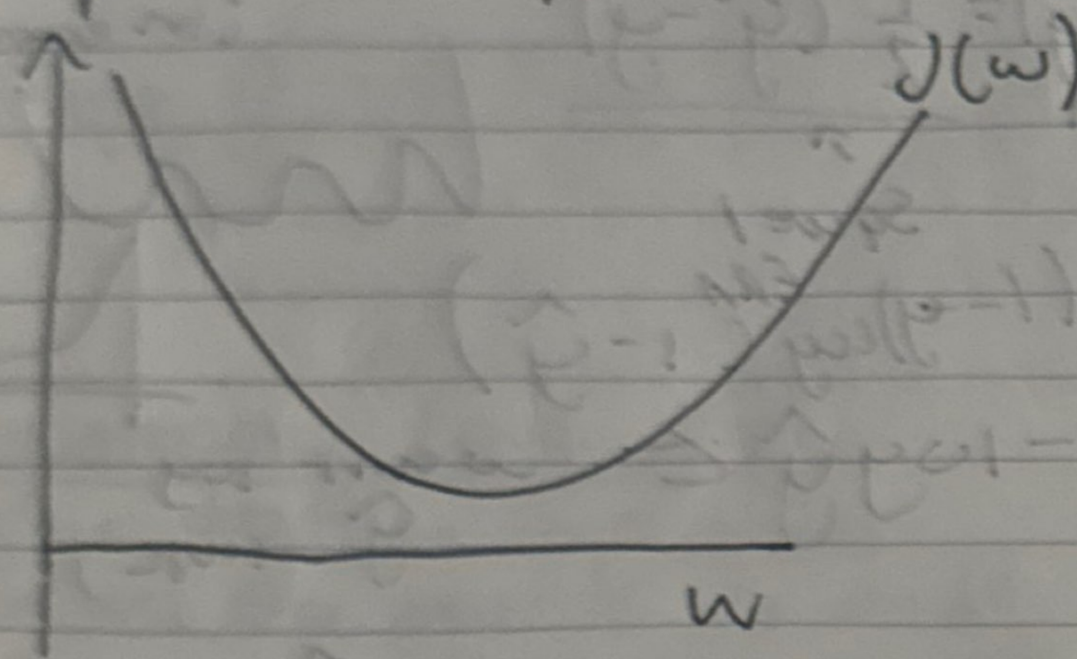
Gradient Descent

Goal: want to find w, b that minimize $J(w,b)$



What it does?

Starts at the initial point and then takes a step in the deepest downhill direction



Repeat {

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

}

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

variable name for derivative

Controls each step taken with each iteration

learning rate

$$\underbrace{\frac{dJ(w)}{dw}}_{\text{rep slope of the function}} =$$

(2) 'partial derivative symbol'

$J(w,b)$

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

(d) $w = J$

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

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

inner loop

actual updates in the loop

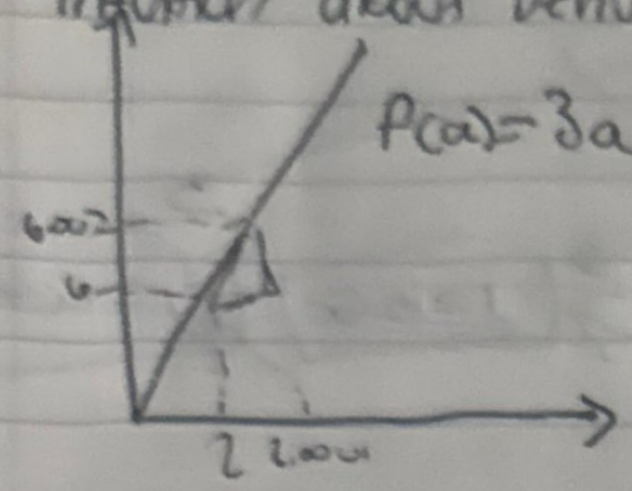
$$\boxed{\frac{\partial J(w,b)}{\partial w}}$$

Same thing for db

dw the amount you want to update w in a single iteration.

Derivatives

Intuition about Derivatives



$a=2 \quad f(a)=6$

$a=2.001 \quad f(a)=6.003$

$\frac{\text{height}}{\text{width}}$ → slope (derivative) of $f(a)$ at $a=2$ is 3..

$a=5 \quad f(a)=15$

$a=5.0001 \quad f(a)=15.0003$

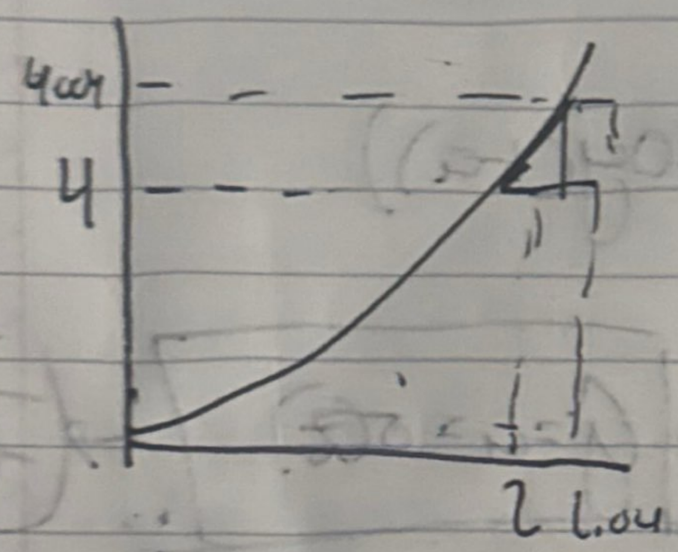
$\frac{d f(a)}{da}=3$

Slope at $a=5$ is also 3

$= \frac{d}{da} f(a)$ ← if I nudge a to the right a bit I expect the value to go up 3x as much.

---

more complex examples



$f(a)=a^2$

$a=2 \quad f(a)=4$

$a=2.0001 \quad f(a)\approx 4.0004$

Slope (derivative) of $f(a)$ at $a=2$ is 4.

$\frac{d}{da} f(a)=4$ when $a=2$

$f(a)=a^2$

$\frac{d}{da} f(a)=2a$

$a=2$

$a=5 \quad f(a)=25$

$a=5.0001 \quad f(a)\approx 25.010$

$\frac{d}{da} f(a)=10$ when $a=5$

Gradient Descent on m examples

$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y)$ $\quad (x^{(i)}, y^{(i)})$

$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$ $\quad dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

$\frac{\partial}{\partial w} J(w,b) = \frac{1}{m} \sum \frac{\partial}{\partial w} \underbrace{\mathcal{L}(a^{(i)}, y^{(i)})}$

$J = 0;\ dw = 0;\ dw_2 = 0;\ db = 0$ $\quad dw_1^{(i)} - (x^{(i)}, y^{(i)})$

For $i = 1$ to m

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

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

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

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$ $\quad n=2$

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

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

$J /= m$

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

Vectorization

Python Vectorization

~~What is vectorization?~~

$z = w^T x + b$ $\quad w = [\ ]$ $\quad x = [\ ]$

Non-vectorized

```
z = 0
for i in range(n-x):
    z += w[i] * x[i]
z += b
```

Vectorized

$z = \underbrace{np.dot(w, x)}_{w^T x} + b$

~~[illegible]~~

Running your code when vectorized is about 300x faster

$u = Av$

$u_i = \sum_j A_{ij} v_j$

```
u = np.zeros((n,1))
for ...
for i
    u[i]
```

Vectorized

```
u = np.dot(A, v)
```

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])
```

$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$ $\quad u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \end{bmatrix}$

```
import numpy as np
u = np.exp(v)
```

output vector

input vector

```
np.log(v)
np.abs(v)
np.max(v, 0)
```

```
v**2
1/v
```

Logistic Regression derivatives cont

$J=0$, ~~$dw_1=0$, $dw_2=0$~~ $db=0$ $\qquad dw = np.zeros(n,1)$

For $i=1$ to m:

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

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

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

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

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

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

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

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

$J = J/m$, ~~$dw_1 = dw_1/m$, $dw_2 = dw_2/m$~~ $db = db/m$

Get rid of one for loop

$dw /= m$

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

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

$\Downarrow$

$$\underbrace{[z^{(1)}, z^{(2)} \ldots z^{(m)}]}_{1\times m} = w^T \underbrace{X}_{n\times m} + \underbrace{[b \ldots b]}_{1\times m \text{ matrix}}$$

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

"Broadcasting"

$$\underbrace{Z = np.dot(w^T, X)}_{\text{vector}} + \underbrace{b}_{\mathbb{R}}$$

Using vectorization to compute gradients

Vectorizing Logistic Regression Gradient computation

$dZ = [dz^{(1)}, dz^{(2)} \dots dz^{(m)}]$ $\quad Y = [y^{(1)} \dots y^{(m)}]$.

$1 \times m$

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

$dZ = A - Y = [a^{(1)} - y^{(1)}, a^{(2)} - y^{(2)} \dots]$

one line of code computes this all at the same time

$dw = 0$

$dw \mathrel{+}= x^{(1)} dz^{(1)}$ → first training example

$dw \mathrel{+}= x^{(2)} dz^{(2)}$

$dw /= m$

$db = 0$

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

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

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

summing them up and dividing it by m.

Vectorized

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

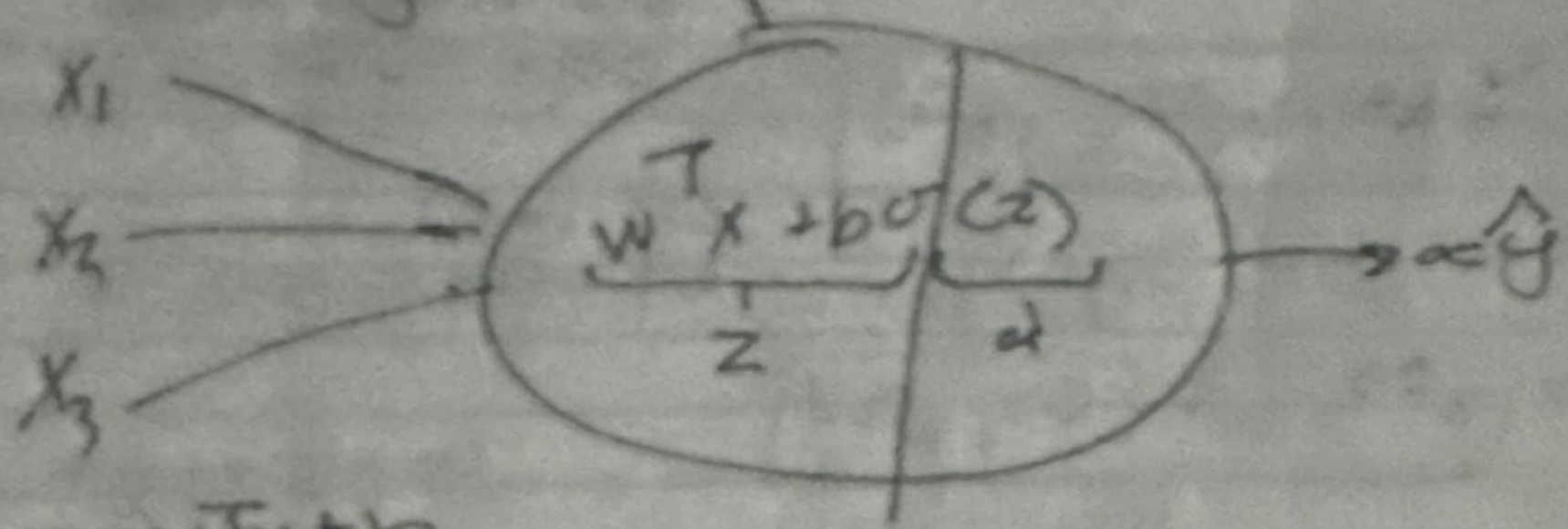
$$= \frac{1}{m} \text{np.sum}(dZ)$$ implement db

$$dw = \frac{1}{m} X dz$$ → implement dw

$$= \frac{1}{m} \begin{bmatrix} x^{(1)} & \dots & x^{(m)} \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{m} \end{bmatrix}$$

$$= \frac{1}{m} [x^{(1)} dz^{(1)} + x^{(m)} dz^{(m)}]$$

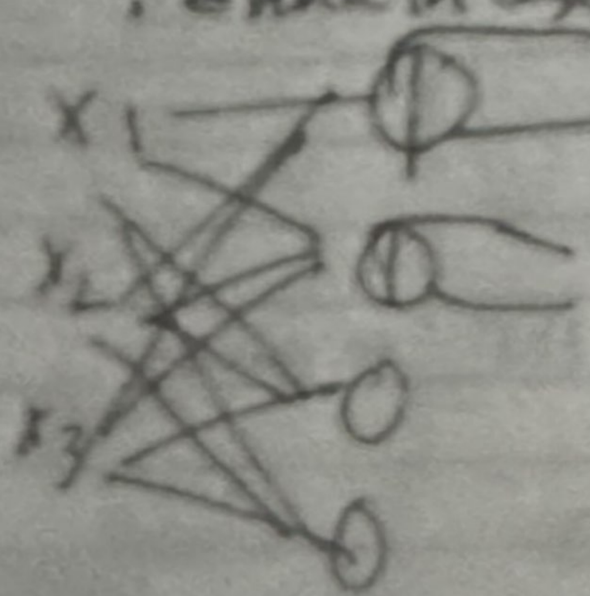
Computing NN output



$z = w^T x + b$
$a = \sigma(z)$

← calculate of the hidden layer
will compute for z then a

$a^{[l]}$ ← layer
$_i$ ← node in layer



$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$
$a_1^{[1]} = \sigma(z_1^{[1]})$
$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$
$a_2^{[1]} = \sigma(z_2^{[1]})$

→ vector

we have 4 logistic regression units
each one has a corresponding
vector

Vectorizing across multiple examples

notation

$a^{[l](i)}$ ← training example i

layer l

Recall:

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

(n_x, m)

Vertically is each node in hidden layer

horizontally is by training example.

Explanation

← note vectorized

for $1 \le i \le m$,

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

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

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

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

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

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

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

$$X = A^{[0]}$$

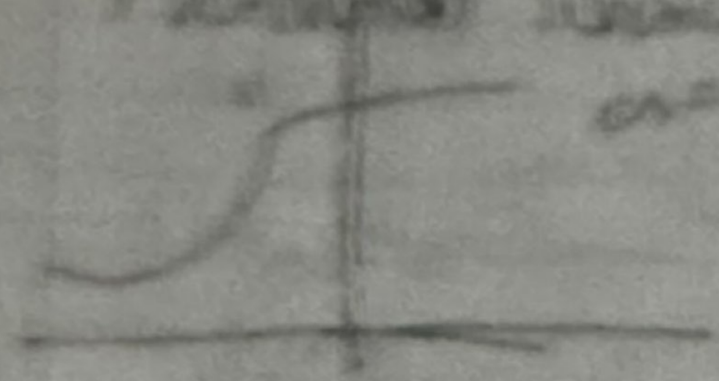
$z^{[1](1)} = W^{[1]} x^{(1)} + b^{[1]}$, $z^{[1](2)} = W^{[1]} x^{(2)} + b^{[1]}$, $z^{[1](3)}$

$$W^{[1]} = \begin{bmatrix} \text{———} \\ \text{———} \\ \text{———} \\ \text{———} \end{bmatrix} \qquad W^{[1]} x^{(1)} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} \qquad W^{[1]} x^{(2)} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

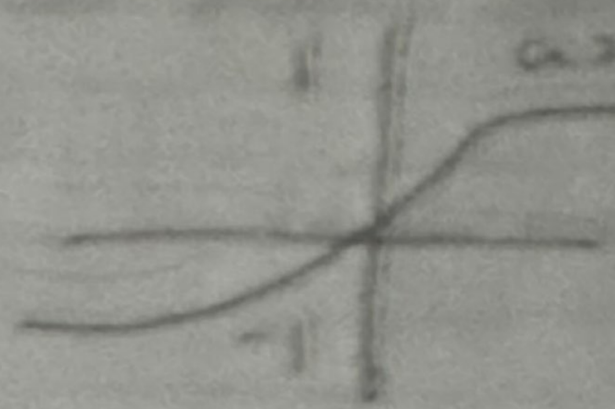
$$\begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} W^{[1]} = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix}$$

X

Activation functions



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



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

$g(z^{[1]}) = \tanh(z^{[1]})$

$y \in \{0, 1\}$

tanh function is superior

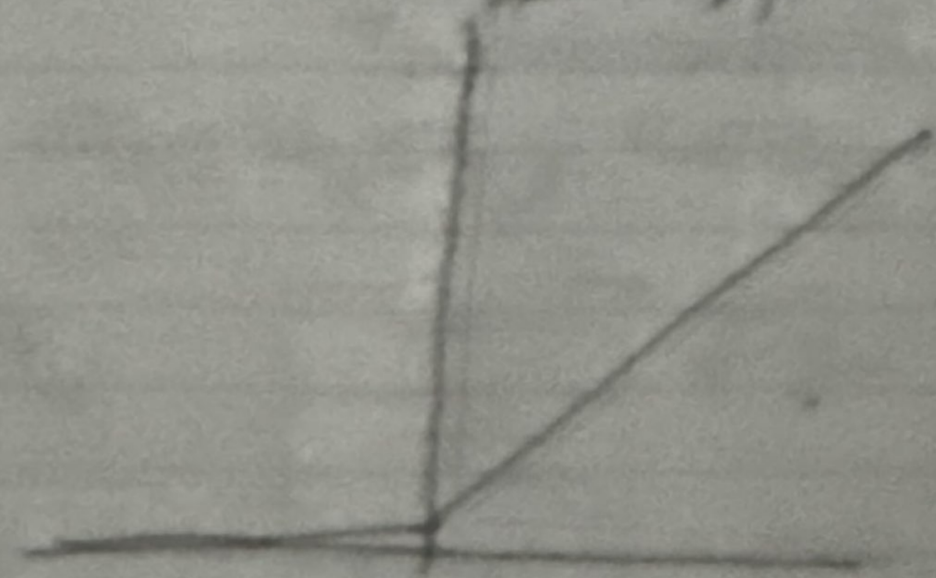
$a = \max(0, z)$

tanh
for hidden layer & for output

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

most commonly

$a = \max(0.001z, z)$



$R(a) = \max(0, z)$

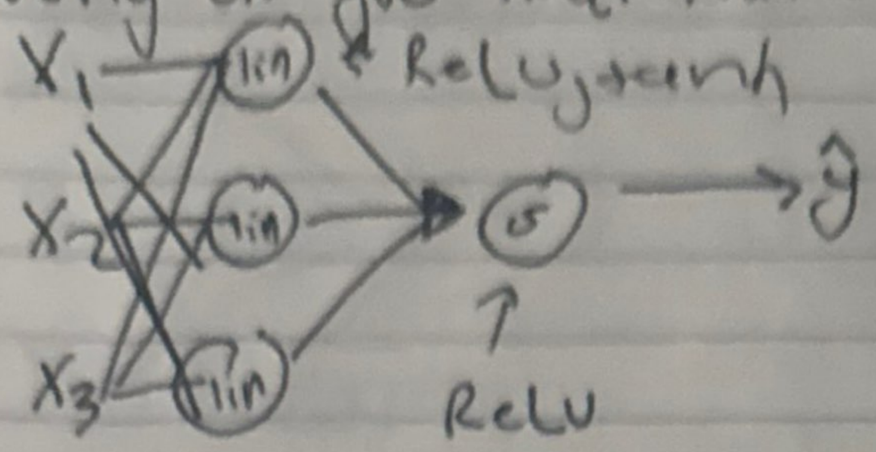
Leaky ReLU

If your output is zero, then
not the further output through
they nullify

Never use sigmoid
activation function

Always use the
tanh(z) $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Why do you need Non-Linear Activation functions



Given x_i

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

$a^{[1]} = \cancel{g^{[1]}(z^{[1]})} \; z^{[1]}$

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

$a^{[2]} = \cancel{g^{[2]}(z^{[2]})} \; z^{[2]}$

$g(z) = z$

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

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

$a^{[2]} = W^{[2]} \underbrace{(W^{[1]} x + b^{[1]})}_{a^{[1]}} + b^{[2]}$

$= \underbrace{(W^{[2]} W^{[1]})}_{W'} x + \underbrace{(W^{[2]} b^{[1]} + b^{[2]})}_{b'}$

$= W' x + b'$

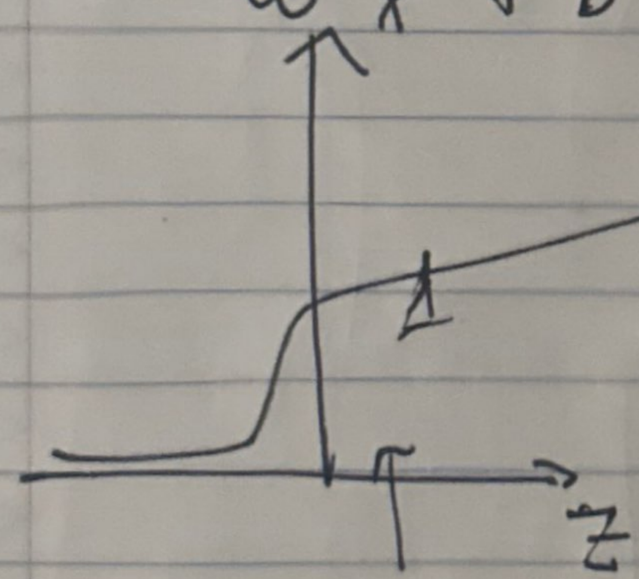
linear algebra.

$y \in \mathbb{R}$

$\$0 \ldots\ldots 1,000,000$

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

Sigmoid activation func



$\frac{d}{dz} g(z)$ = slope of $g(x)$ at z

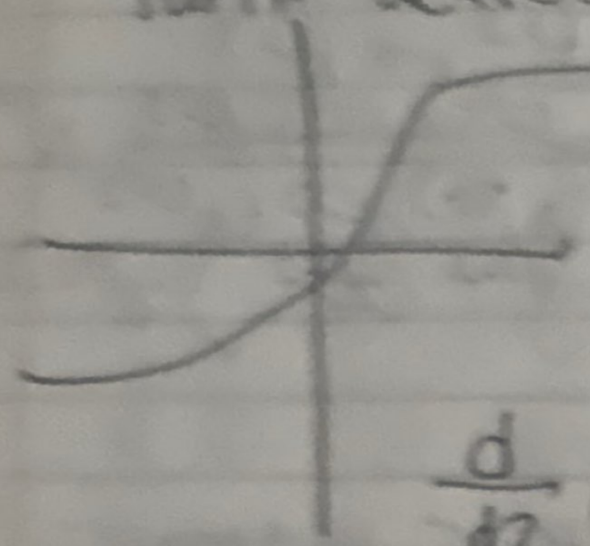
$= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$

$= g(z)(1 - g(z))$

$g'(z) = \frac{d}{dz} g(z)$

$= a(1-a)$

tanh activation function



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

$$\frac{d}{dz} g(z) = \text{slope of } g(z) \text{ at } z = 1 - (\tanh)$$

$z = 10$ $\tanh(z) \simeq 1$

$g'(z) \simeq 0$

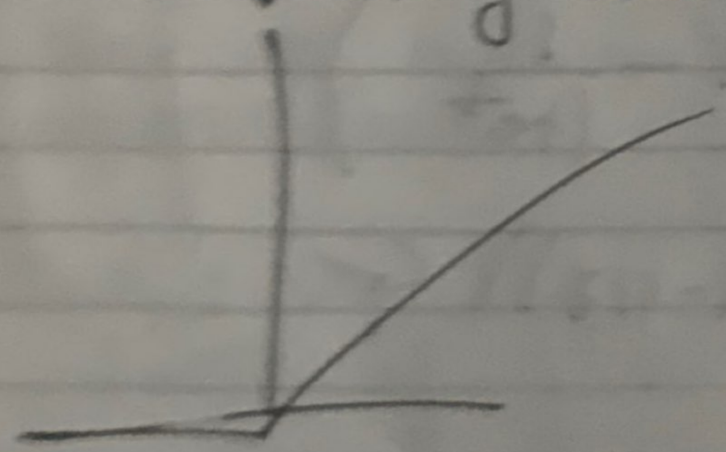
$z = -10$ $\tanh(z) \simeq -1$

$g'(z) \simeq 0$

$z = 0$ $\tanh(z) = 0$

$g'(z) = 1$

ReLU & Leaky ReLU



$g(z) = \max(0, z)$

Getting Your Matrix Dimensions Right

Implementing forward propagation

$n^{[0]} = 3 \quad n^{[1]} = 5, \; n^{[3]} = 4$

$n^{[2]} = 2, \; n^{[4]} = 1$

$Z^{[1]} = \underline{W^{[1]}} \cdot X + b^{[1]}$

n = number of units

Z → vector activations for hidden layer

$W^{[1]}(n^{[1]}, n^{[0]})$

$\begin{bmatrix} \cdot \\ \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$

$W^{[2]} : (5, 3) \quad (n^{[2]}, n^{[1]})$

formula for checking

$W^{[l]} : (n^{[l]}, n^{[l-1]}) \rightarrow dW$ should be the same

$b^{[l]} : (n^{[l]}, 1) \rightarrow db$

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

$\underset{(n^{[1]}, m)}{Z^{[1]}} = \underset{(n^{[1]}, n^{[0]})}{W^{[1]}} \cdot \underset{X}{X} + \underset{(n^{[0]}, m)}{b^{[1]}}$

$=$

$Z^{[0]}, dX : (n^{[0]}, 1)$

$Z^{[l]}, A^{[l]} : (n^{[l]}, m)$

$l = 0 \quad A^{[0]} = X = (n^{[0]}, m)$

$dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$

Why deep neural networks work: Key concepts

1. Importance of Depth in Neural Networks
   - It's not just size (number of parameters), but depth (many hidden layers) that matters
   - Deep networks can learn hierarchical representations

2. Example: Face Recognition
   - Input: Image of a face
   - Early Layers: Detect simple features
     Think: horizontal / vertical edge detectors
   - Middle Layers: Detect parts of faces
   - Later Layers: Combine parts to detect whole faces

3. Example: Speech Recognition
   Input: Audio waveform
   Early Layers: Detect low-level features
   Middle Layers: Detect phonemes
   Later Layers: Recognize words, then phrases or sentences

4. Hierarchial Feature Learning
   Early layers: Simple features
   Later layers: Complex features
   This compositional structure mirrors how the brain may process information (simple → complex)

5. Circuit Theory Insight
   - Some functions are computable efficiently with deep networks
   XOR Tree:
     - Requires log(n) depth to compute XOR
     - Shallow networks require exponential number of neurons → inefficient

There are functions you can compute with a "small" L-layer deep neural network, that shallow networks require exponentially more units to compute.

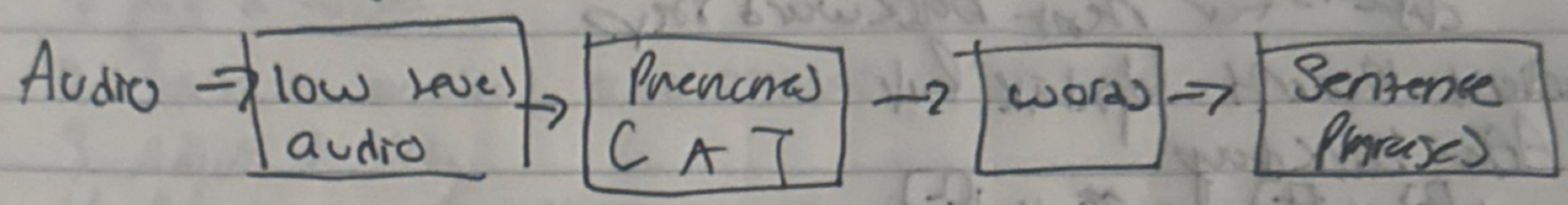
Pt2 | Building Blocks

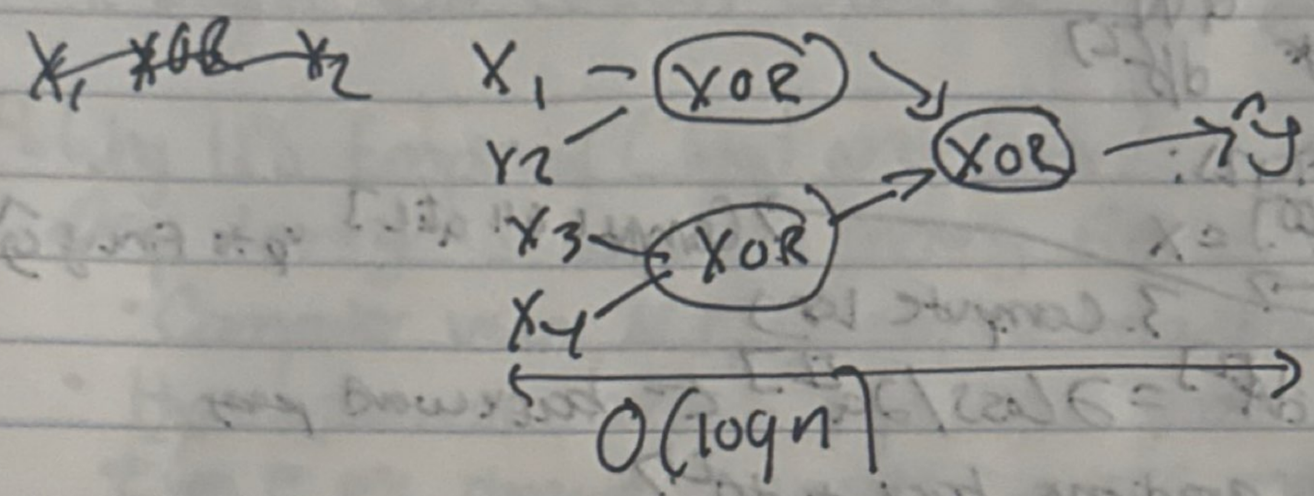
7. Practical Advice

- Start simple: try logistic regression or 1-2 hidden layers
- Depth = hyper parameter to tune
- Some applications benefit from very deep network (dozen of layers)

8. Branding of "Deep Learning"

- Popularity partly due to catchy branding, but effectiveness is real
- "Deep Learning" often = neural networks with many hidden layers





~~Beitdi~~

Building Blocks of Deep neural networks

- Key components
  - Weights: $W^{[L]}$
  - Bias: $b^{[L]}$

1. forward Propagation

Goal: Compute activation $a^{[L]}$ from input $a^{[L-1]}$

Steps:

1. compute $z^{[L]} = W^{[L]} * a^{[L-1]} + b^{[L]}$
2. $a^{[L]} = g(z^{[L]})$
3. Cache $z^{[L]}$ for use in back propagation

forward function:

Inputs: $a^{[L-1]}$, $W^{[L]}$, $b^{[L]}$

Outputs $a^{[L]}$, cache($z^{[L]}$)

2. Backward Propagation

Obtain compute gradients from loss wrt weights, bias and previous activations

Steps

Input $dA^{[l]}$ and cache($Z^{[l]}$)

Compute

$dZ^{[l]}$

$dW^{[l]}$, $db^{[l]}$

$dA^{[l-1]}$ for next backward step

Gradient Descent Update Rule

After backprop

$W^{[l]} = W^{[l]} - \alpha * dW^{[l]}$

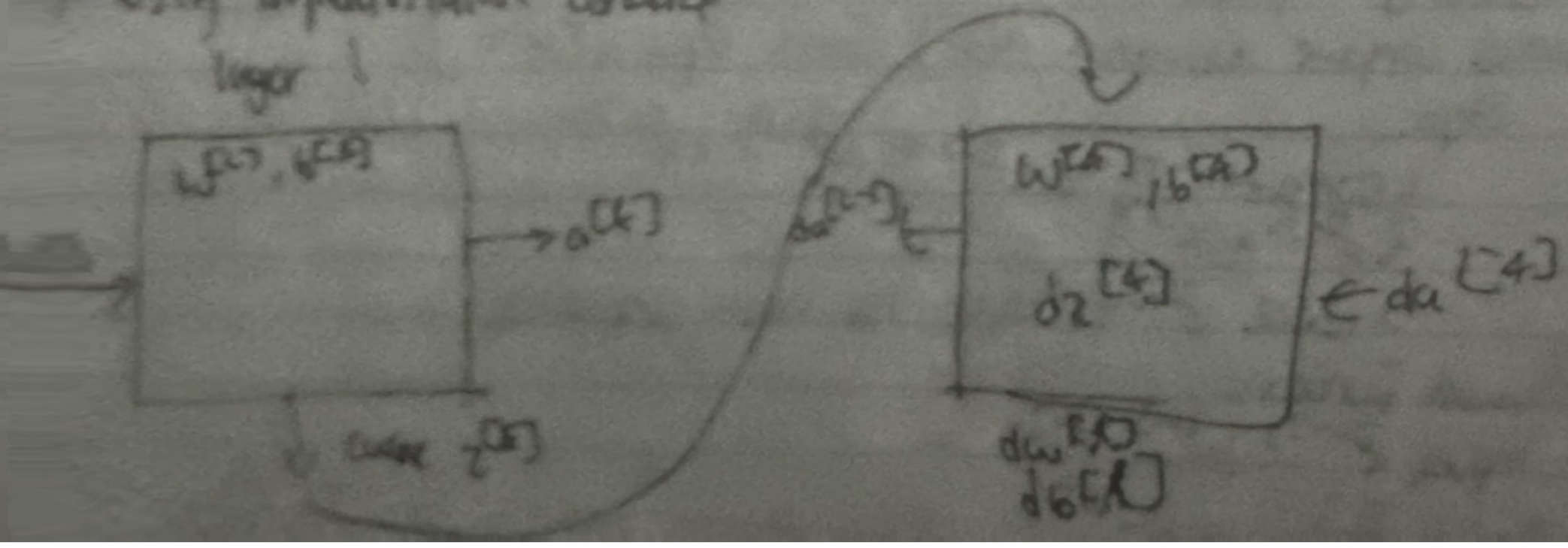
$b^{[l]} = b^{[l]} - \alpha * db^{[l]}$

Full Training Iterations:

1. Start with input $a^{[0]} = X$
2. forward pass → Compute all $a^{[L]}$ up to $\hat{y}$
3. Compute loss
4. Backward pass
   - Start with $dA^{[L]} = \partial(loss)/\partial a^{[L]}$ ← backward pass
   - compute all gradients back to $dA^{[1]}$

Use of Cache

- Stores $Z^{[l]}$, $W^{[l]}$, $b^{[l]}$ from forward pass
- Helps access needed values for backprop easily
- Mostly implementation concern



Understanding Hyper parameters

1. What are Hyper parameters

Model Parameters vs Hyper Parameters

- Parameters (learned during training)
  - W - weight
  - B - bias

Hyper parameters (set before training)
- learning rate α
- # of iterations
- number of hidden layers
- number of hidden units per layer (FC)
- Activation function

3. Tuning Hyperparameters is Empirical

Process overview
1. pick a value
2. Train a model and observe performance
3. Adjust & retry
4. Repeat with different combos of Hyper parameters

4. Why It's Empirical (Trial and Error)

Deep learning is used in many fields:
- Computer vision, NLP, speech, recommendation system
- Hyperparameter behavior may not generalize across domains
- Even in one domain, optimal parameters may change
  - New data
  - New Hardware
  - Algorithm improvements

5. Best practices
- always validate with a dev/validation set
- Re-check hyperparameters periodically
- Expect to develop better intuition over time
- Don't be discouraged by the trial & error as normal