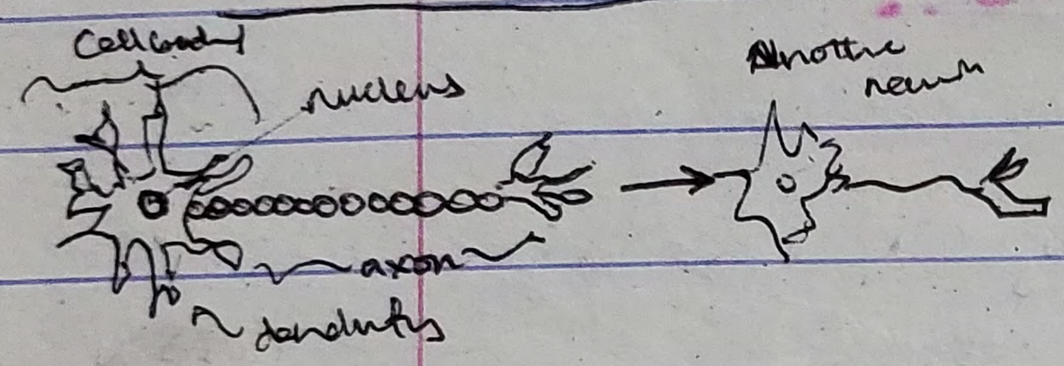
Week 2

ADVANCED LEARNING ALGORITHMS

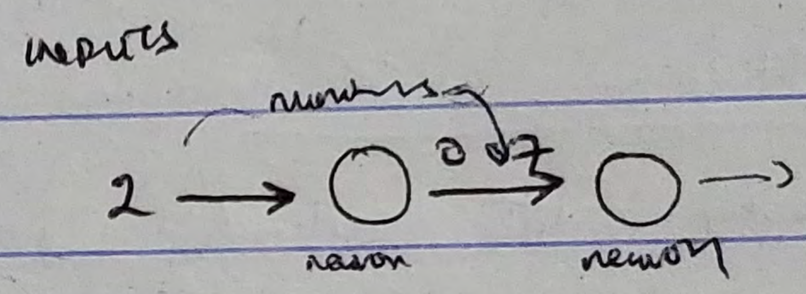
WK 1 NEURAL NETWORKS INTUITION

- The origins of neural network begin @ in algorithms that try to mimic the brain. Even though current Neural Networks have almost nothing to do w/how the brain ~~works~~ learns, there was the early motivation of trying to mimic the brain

BIOLOGICAL NEURON



SIMPLIFIED MATH'S MODEL of a NEURON



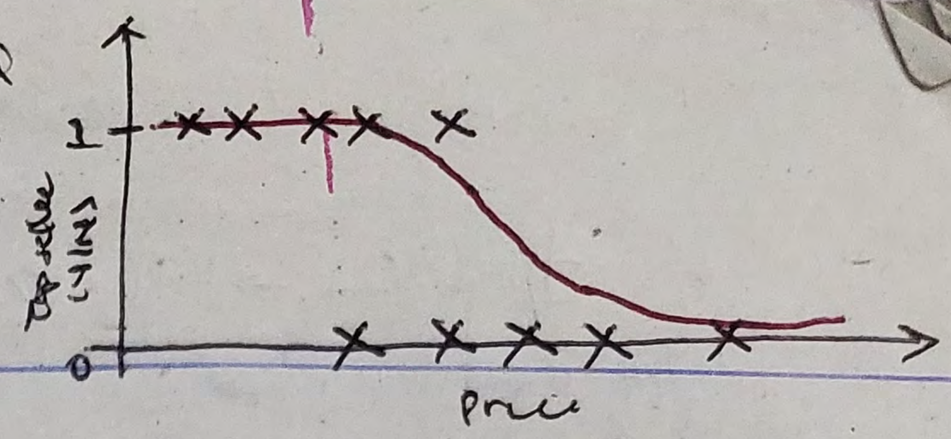
- A biological neuron consists of a cell body, w/inputs called dendrites and an output wire called the axon. It sends electrical impulses to other neurons. In artificial neural networks, we use a simplified mathematical model of a neuron. Input are numbers, and it performs computations to generate an output, which can be used as an input for another neuron <shown on the right>
- Rather than building one neuron @ a time, we're interested in simulating many neurons @ the same time.
- Even though we drew some similarities between biological neurons & artificial neurons, as of today, there is so much difference between the two due to the ever increasing discovery of the brain's complexity. ∴ We no longer know how the brain works

HOW DO NEURAL NETWORKS WORK? USE CASE <example>: DEMAND PREDICTION

- We look @ a product & determine if it'd be a top seller or not.

In this example, you are selling T-shirts & you'd like to know if which particular T-shirt will be a top seller (Yes/No)

- You have data on different T-shirts sold @ different prices as well as which ones became a top-seller



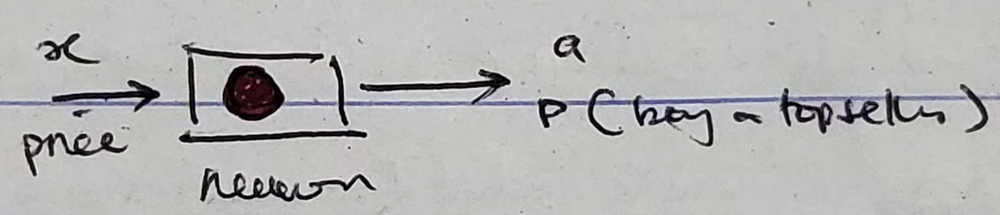
Here x = price (→ input) & when logistic regression is used to fit a sigmoid function to the data, then the output of data $= \frac{1}{1+e^{-(wx+b)}}$

a = activation

The output is denoted by $a = f(x) = \frac{1}{1+e^{-(wx+b)}}$

which is a neuroscience term which measures how much a neuron ~~sends~~ is sending a high output to other neurons

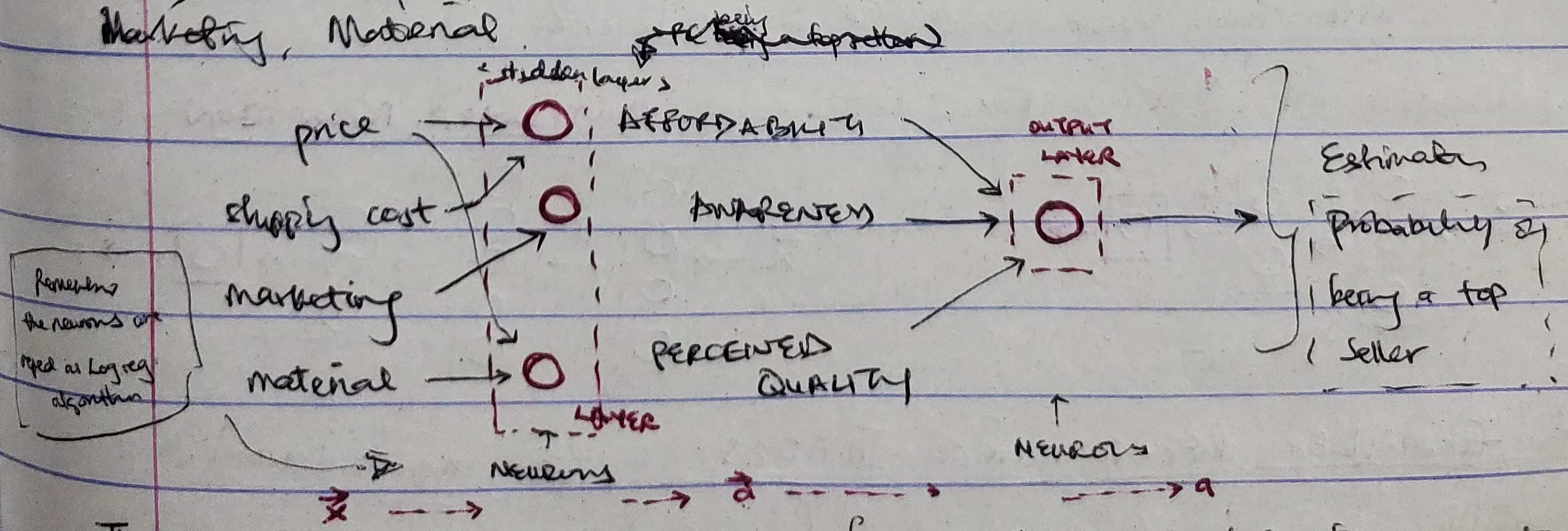
- Thus logistic regression algorithm can be thought of as a very simplified model of a single neuron in the brain. Where the neuron takes the price, x & computes the $\frac{1}{1+e^{(wx+b)}}$ formula on top and outputs the no. a which is the probability of that T-shirt being a top seller



∴ with this, building a neural network requires using a bunch of these neurons together.

X → NOW, LETS LOOK @ a MORE COMPLEX EXAMPLE of DEMAND PREDICTION

- We have 4 features for predicting Top Sellers (Y/N): Price (x_1), Shipping cost (x_2), Marketing (x_3), Material (x_4)



- The 3 neurons are grouped into a LAYER ⇒ {A group of neurons which take as input similar features and return output a few numbers together} Layers can have single or multiple neurons

- Affordability, Awareness & Perceived quality are called "ACTIVATIONS". ~~respect to~~ of the first layer.

- P(Being a top seller) activation of the output layer.

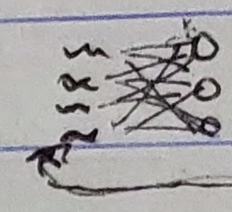
∴ The computation of our neural network is;

refers to the number calculated by a neuron. Activation = a vector output

<activation values>

4 numbers → Layer ⇒ 3 numbers → Output layer ⇒ 1 number

aka $\boxed{\text{input layer}}$



Note: In actuality, the 1st Layer of neurons has access to all the input features.

As the model continues learning, it figures out which features to ignore.

- The "Hidden layer" is called so because your dataset doesn't tell you the correct values for your activations [Affordability, awareness, perceived quality]. They are hidden

★ → One way to see Neural Network = It is just logistic regression but it is one that learns its own features and that makes it easier to make accurate predictions.

→ Instead you carry out manual feature engineering, it learns its own ~~features~~ features to make the learning problem easier for itself.

Summary: $\vec{x}$ → Hidden Layer → $\vec{a}$ → Output layer → a

- Note: You don't need to decide what features should be in the hidden layer. The algorithms learns from the data and chooses for itself.

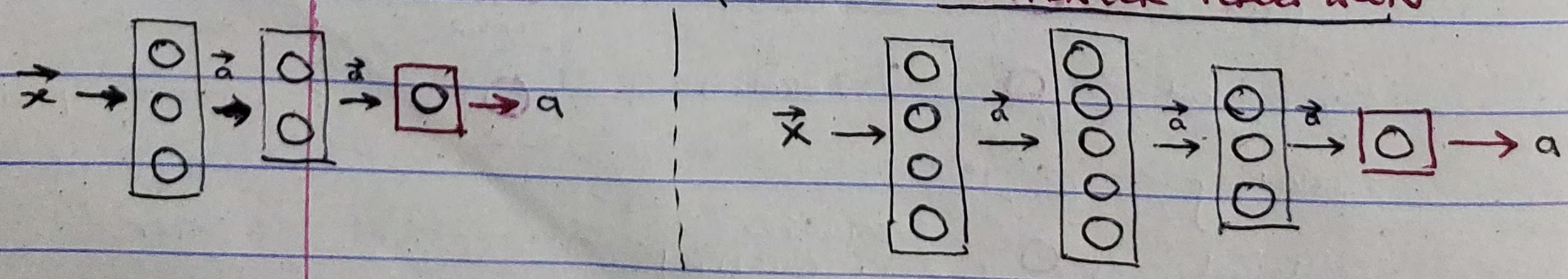
- When you're building your neural network, you ~~too~~ need to have 2 things in mind,

  (i) How many hidden layers do you want?

  (ii) How many neurons should be in each layer?

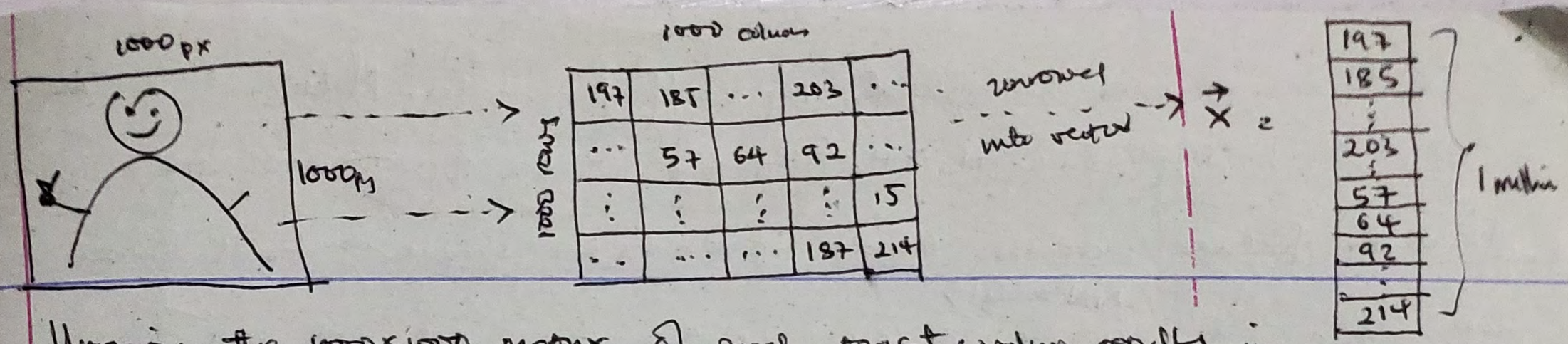
  } Neural Network Architecture

Examples of Multiple Hidden Layers aka MULTILAYER PERCEPTRON



EXAMPLE: RECOGNIZING IMAGES → Computer vision application

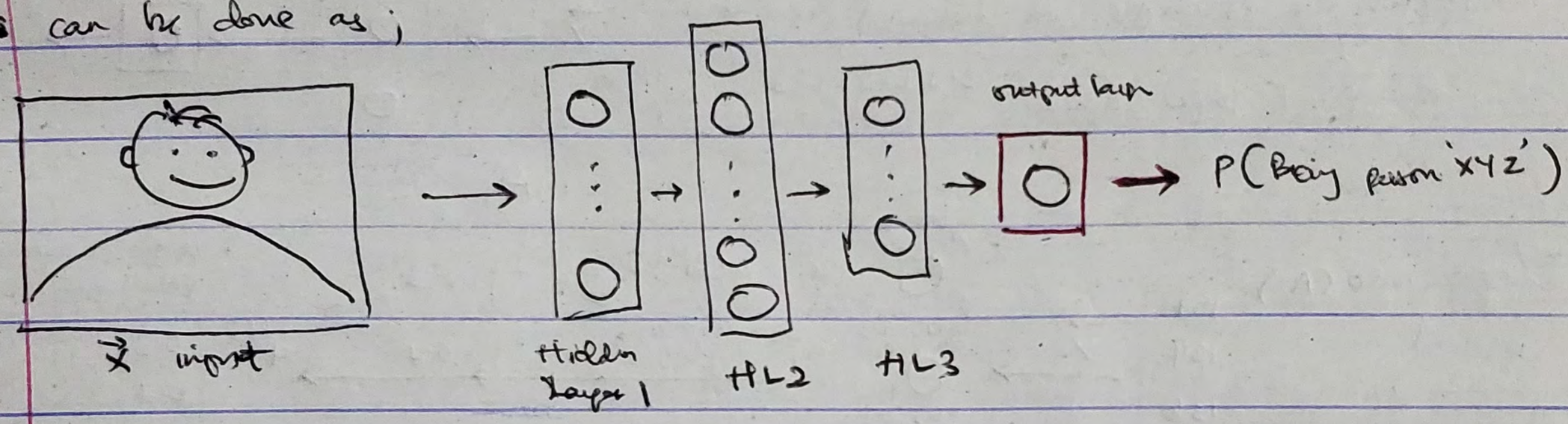
- When building a facial recognition application, you might want to train a neural network that inputs a picture $\vec{x}$ outputs the identity of the person in that image, which for example is a 1000 px by 1000 px, is represented as a 1000 × 1000 matrix of pixel intensity values e.g (0-255)



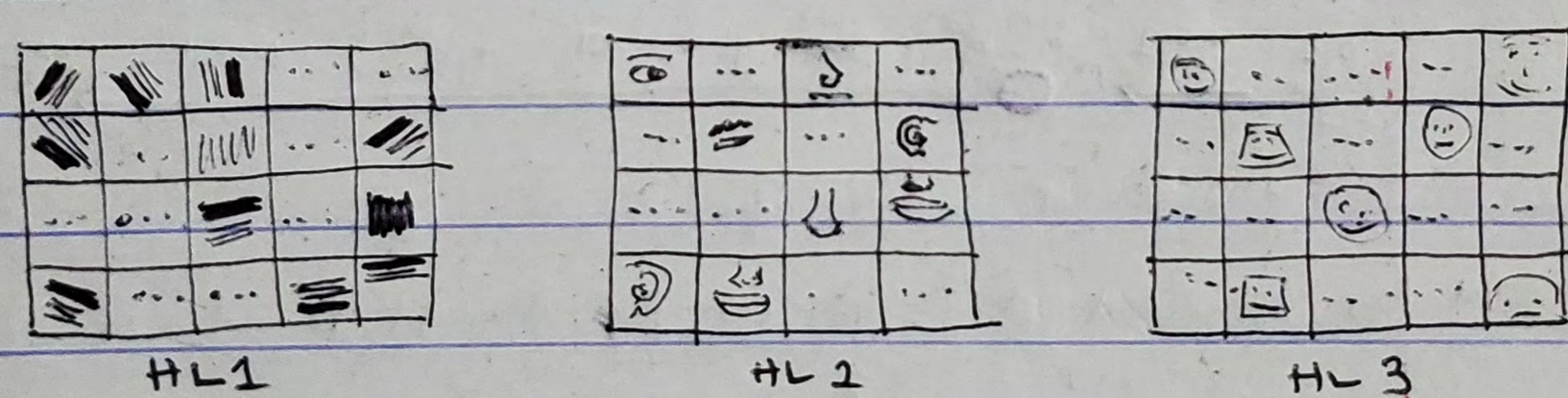
- Unrolling the 1000×1000 matrix of pixel intensity values results in a list or vector of 1 million = 1000×1000 pixel intensity values.

∴ The Facial Recognition Problem: Can you train a neural network which takes $\vec{X}$ with a million length vector and outputs the identity of the person.

This can be done as;



- If you were to train this neural network w/ a lot of images & you were to peer at the different neurons in the hidden layers you'd see something like;



In the earliest layers <HL1> of the neural network, you'd find out that the neurons are looking for varying degrees of lines and edges <length, width, thickness, etc>. In the next <HL2> layer, you'd find out that the neurons have learnt to group together lots of short lines & segments in order to form parts of the faces e.g. presence of eye or nose @ a certain position, corner of a nose, width of an ear. In the next layer, the NN (neural networks) aggregates different parts of faces to try to detect the presence/absence of larger or coarser face shapes. By determining how the face corresponds to different face shapes creates a rich set of features that helps the output layer detect the identity of the person in the picture.

- It learns what to look for by itself from data. Note that as we go in the hidden layers i.e. $H_1 \to H_2 \to H_3$, the window through which the neurons looks for features increases: strokes → face parts & etc shapes → facial structure

— Using the <NN model> example on cars, i.e. training the model with cars data

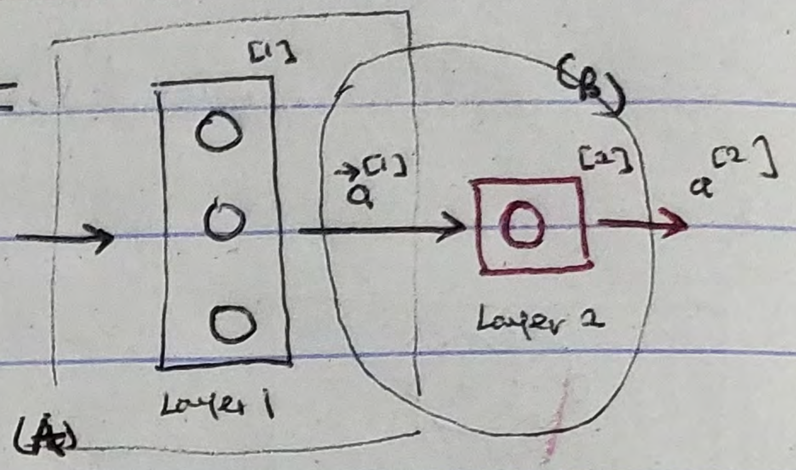
HL1 = lines & strokes, edges . HL2 = Parts of cars ; HL3 = more complete car shapes

WK 1 → NEURAL NETWORK MODEL

BUILDING a NEURAL NETWORK LAYER

- Remember our example on demand prediction : $\vec{x}$ → layer 0

Notation for layer numbering : $^{[1]}$ $^{[2]}$ $^{[n]}$ <superscript, square bracket>



- Zooming into (A)

Layer 1

$$\begin{bmatrix} 197, \\ 184, \\ 136, \\ 214 \end{bmatrix} \xrightarrow{\vec{x}} \begin{cases} a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]}) = 0.3 \\ a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]}) = 0.7 \\ a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]}) = 0.2 \end{cases} \longrightarrow \vec{a}^{[1]} \begin{bmatrix} 0.3, \\ 0.7, \\ 0.2 \end{bmatrix}$$

vector of activation values from layer 1

- Zooming into (B) ; Layer 2

Layer 2

$$\begin{bmatrix} 0.3 \\ 0.7 \\ 0.2 \end{bmatrix} \xrightarrow{\vec{a}^{[1]}} a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}) = 0.84 \longrightarrow a^{[2]}$$

→ A scalar value

↳ 0.84

- Once the NN has computed $a^{[2]}$; you can use threshold to get a binary prediction (yes or No)

is $a^{[2]} \geq 0.5$? yes ; $\hat{y} = 1$ | No ; $\hat{y} = 0$

- Activation value of layer l, unit (neuron) j,

$$\boxed{a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})}$$

output of layer $l-1$ <prev. layer>

parameters w & b of layer l, unit j

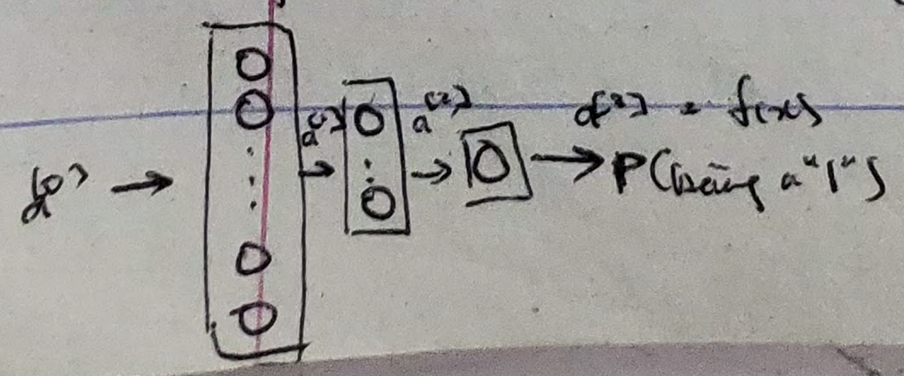
"activation function" → output activation values.

sigmoid function in this case

- Note: $\vec{x}$ will now be represented as $\vec{a}^{[0]}$

INFERENCE : MAKING PREDICTIONS → "FORWARD PROPAGATION"

- We'll be working on Handwritten Digit Recognition for 0 & 1

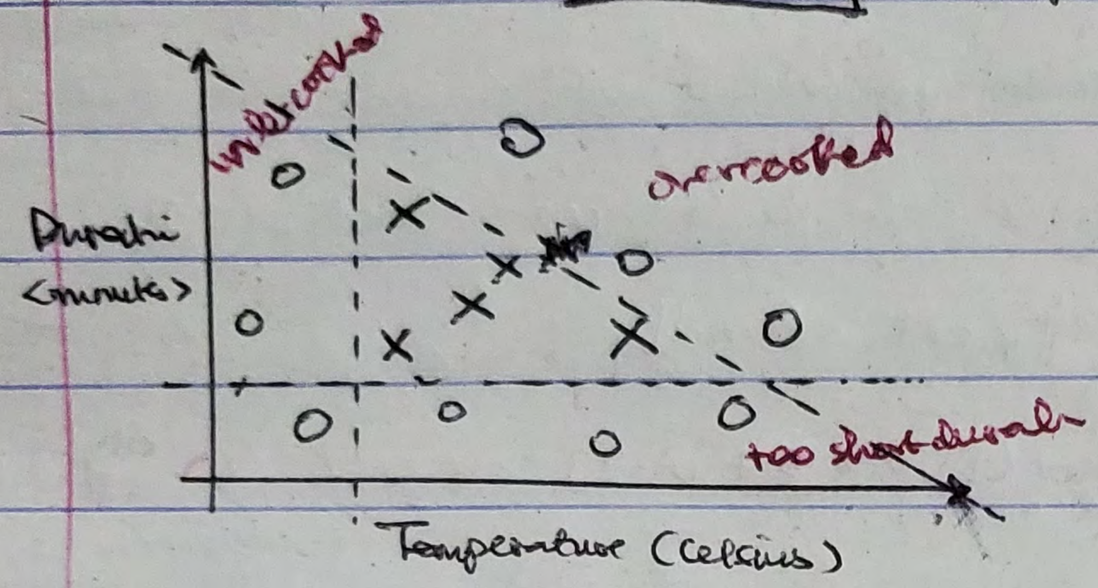


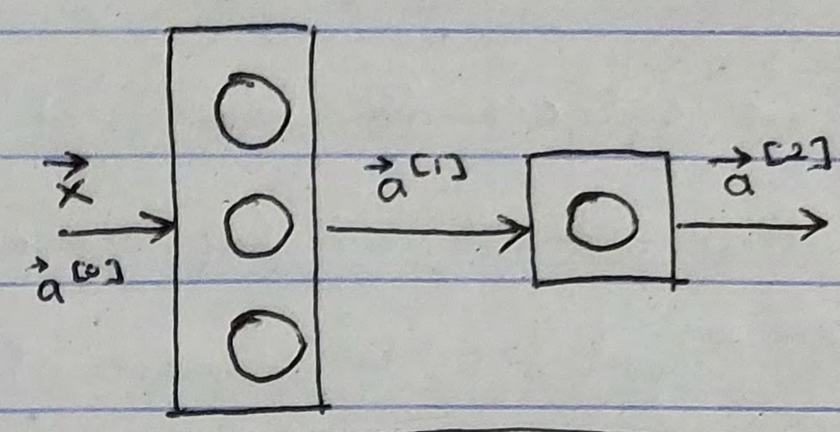
- Here $f(\vec{x})$ denotes the function computed by a NN as a function of $\vec{x}$

- Because this computation goes from left to right, the algorithm is also called ~~for~~ FORWARD PROPAGATION. This is because you're propagating the ~~activat~~ activations of the neurons from left to right

Wk 1 TENSORFLOW IMPLEMENTATIONS

Let's look @ a coffee roasting example;





is $a_1^{[2]} \geq 0.5$?

yes → $\hat{y} = 1$ × no → $\hat{y} = 0$ ○

∴ The task is, given a $\vec{x}$ with both temperature & duration e.g $\begin{bmatrix} 200°C \\ 17\ min \end{bmatrix}$, how can we do inference in a NN to get it to tells whether or not this temperature and duration setting will result in good coffee or NOT

— For the first layer $\vec{x} \rightarrow \boxed{\circ\circ\circ} \rightarrow \vec{a}^{[1]}$

1. `x = np.array([[200.0, 17.0]])` You set "x" to be an array of two numbers. Then you create "layer_1" i.e. the first hidden layer
2. `layer_1 = Dense(units=3, activation="sigmoid")`. ~~"units = 3" refers to~~
   - 3 neurons in the layer
   - using the sigmoid function as the activation funct
   - "Dense" is another name for the layers of a NN. There are other types
3. `a1 = layer_1(x)`
   - "layer_1" is a function which is now applied ~~to the~~ to get $\vec{a}^{[1]}$
   - A list of 3 numbers because we have 3 units e.g. $\begin{bmatrix} 0.2 \\ 0.7 \\ 0.3 \end{bmatrix}$

— the 2nd layer $\vec{a}^{[1]} \rightarrow \boxed{\circ} \rightarrow \vec{a}^{[2]}$

4. `layer_2 = Dense(units=1, activation="sigmoid")`
5. `a2 = layer_2(a1)`
   - "a2" can now be ~~the~~ used to predict quality by

```
if a2 >= 0.5:
    yhat = 1
else:
    yhat = 0
```

Note about numpy arrays

x.shape

(1,2)

x = np.array([[200, 17]]) → [[200 17]] ~ 1x2 matrix

x.shape

(2,1)

x = np.array([[200], [17]]) → [[200] [17]] ~ 2x1 matrix

used for Tensorflow

x.shape

(2,)

x = np.array([200, 17]) → [200 17] ~ 1D vector

used for linear & log regression

- we use matrices for Tensorflow because Tensorflow was designed to handle very large datasets and by representing the data in matrices instead of 1D arrays allows Tensorflow to be computationally efficient internally

- printing a1 results in tf.Tensor([[0.2 0.7 0.3]], shape=(1,3), dtype=float32)

A datatype that was created in order to store and carry out computations on matrices efficiently.

BUILDING A NEURAL NETWORK ARCHITECTURE

- Our previous method of building a NN using tensorflow involved passing $\vec{x}$ through layer_1 to get a1 then passing that through layer_2 to get a2.

There is a better way;

1. layer_1 = Dense(units=3, activation="sigmoid")
2. layer_2 = Dense(units=1, activation="sigmoid")
3. model = Sequential([layer_1, layer_2])

- What we do instead is to tell TensorFlow to string together layer_1 & layer_2 to form a Neural Network. This is what "Sequential()" does. It sequentially strings together layers

- The better simplified is: model = Sequential([Dense(units=3, activation="sigmoid"), Dense(units=1, activation="sigmoid")])

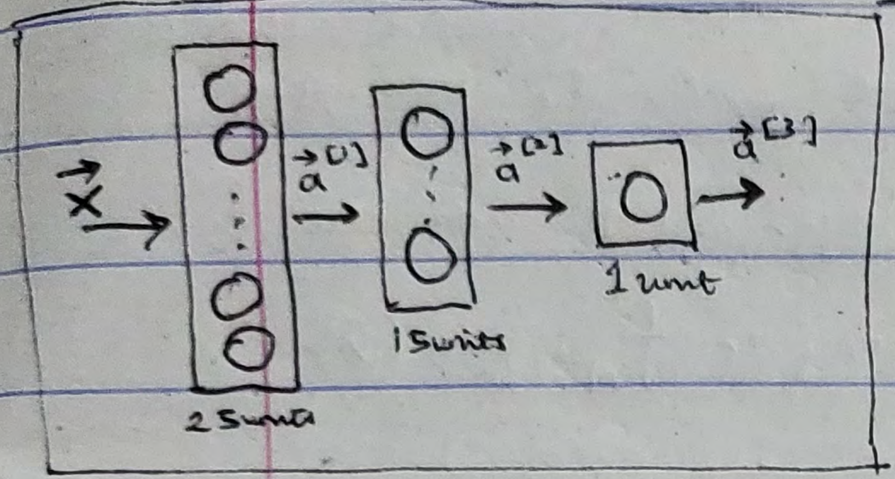
W1-1 ~ NEURAL NETWORK IMPLEMENTATION in PYTHON

↳ FORWARD PROPAGATION IN A SINGLE LAYER

KW2 NEURAL NETWORK TRAINING

* epochs = the number of steps in gradient descent

OVERVIEW ~~Summary~~ of How to TRAIN a NEURAL NETWORK in TENSORFLOW



- Q: Given set of (x, y) examples, how do you train & build this in code?

1. import tensorflow as tf
2. from tensorflow.keras import Sequential
3. from tensorflow.keras import Dense
4. model = Sequential([
   Dense(units=25, activation="sigmoid"),
   Dense(units=15, activation="sigmoid"),
   Dense(units=1, activation="sigmoid"),
   ]) ①

(i) Step 1 = Specify the model. This tells tensorflow how to compute forward inference

(ii) Step 2 = Compiles the model using a specific loss function

5. from tensorflow.keras.losses import BinaryCrossentropy
6. model.compile(loss=BinaryCrossentropy()) ②

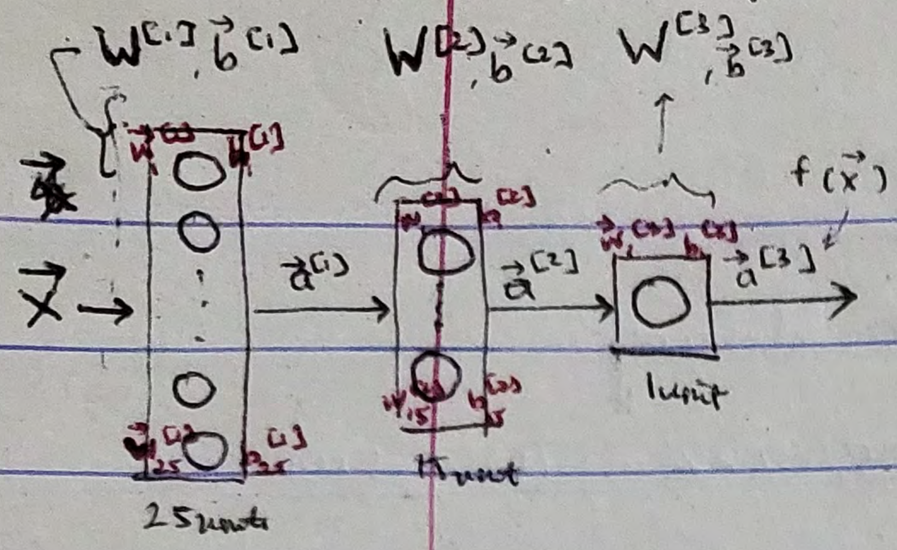
(iii) Step 3 = To train the model
→ minimize cost as a function of the parameters

7. model.fit(X, Y, epochs=100) ③

Let's go deeper: MODEL TRAINING STEPS

Steps Overview	logistic regression	Neural Network
① Specify how to compute output given input x and parameters w, b (i.e. define model) $f_{\vec{w},b} = ?$	$z = np.dot(w, x) + b$ $f_x = 1/(1+np.exp(-z))$	model = Sequential([Dense(...) Dense(...) Dense(...)])
② Specify loss and cost $L(f_{\vec{w},b}(\vec{x}), y)$ ... Loss for an example $J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$	Logistic loss $loss = -y * np.log(f_x) - (1-y) * np.log(1-f_x)$	binary cross entropy model.compile(loss = BinaryCrossentropy())
③ Train on data to minimize $J(\vec{w},b)$ ~ {using gradient descent}	$w = w - alpha * dj_dw$ $b = b - alpha * dj_db$	model.fit(X, Y, epochs=100)

1. Create the model




```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(units=25, activation="sigmoid"),
    Dense(units=15, activation="sigmoid"),
    Dense(units=1, activation="sigmoid"),
])
```

This code compact specifies the entire architecture of the NN.

It tells us the no. of layers, how many units in each layer and the activation functions that'll be used. It tells what we need to know to compute $f_{W^{[L]}, \vec{b}^{[L]}}(\vec{x}) = \vec{a}^{[3]}$ (L = layers no.)

2. → Loss & Cost function

- In the case of the handwritten digit classification problem, which is a binary classification problem. The loss function used is the same as that for the log reg:

$$L(f(\vec{x}), y) = -y\log(f(\vec{x})) - (1-y)\log(1-f(\vec{x}))$$

y = target label
$f(x)$ = output of NN

→ In tensorflow this logistic loss function is referred to as "Binary Cross Entropy" loss function

- It has that name because in stats, the eqn above is called 'cross entropy' & the 'binary' just points out that this is a binary classification problem [$\hat{y} = 0$ or 1].

SYNTAX: » model.compile(loss = BinaryCrossentropy())

→ Asks ~~tens~~ tensorflow to compile the NN using this loss function.

→ Having specified the loss w/respect to a single training example, tensorflow knows that the cost you want minimize is then the average of the loss on all the training examples. And optimizing this cost function will result in fitting the NN to your binary classification data.

- For example, if you want to solve a regression problem, you can use the Mean Square Error loss function

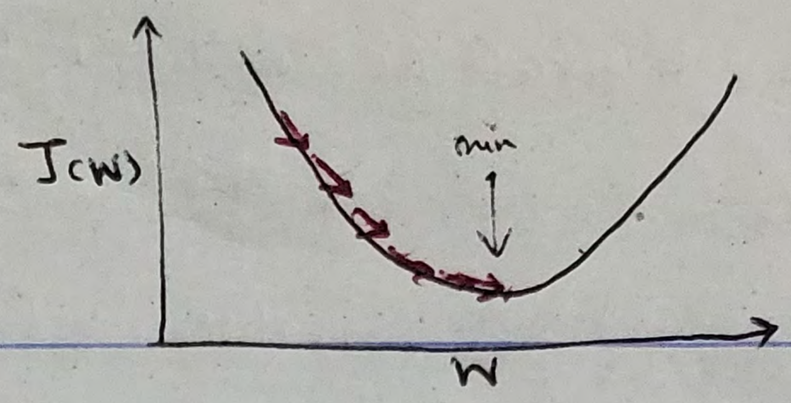
```
from tensorflow.keras.losses import MeanSquaredError

model.compile(loss = MeanSquaredError())
```

- $$J(W, B) = \frac{1}{m}\sum_{i=1}^{m} L(f(\vec{x}^{(i)}), y^{(i)})$$

W: $W^{[1]}, W^{[2]}, W^{[3]}$

B: $\vec{b}^{[1]}, \vec{b}^{[2]}, \vec{b}^{[3]}$

3. → Gradient Descent



repeat {

$$w_j^{[l]} = w_j^{[l]} - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

$$b_j^{[l]} = b_j^{[l]} - \alpha \frac{\partial}{\partial b_j} J(\vec{w}, b)$$

the key thing to compute is $\frac{\partial}{\partial w_j} J(w,b)$ & $\frac{\partial}{\partial b_j} J(w,b)$

∴ tensorflow uses <u>back propagation</u> in order to compute these partial derivative terms

└ all this is done in 'model.fit(X, y, epochs=100)'

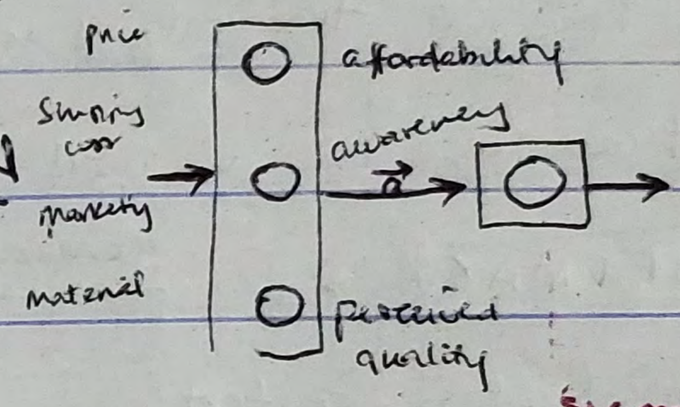
training set

└ 100 iterations

- Tensorflow even uses an algorithm that's faster than gradient descent.

WK2 ACTIVATION FUNCTIONS

ALTERATIVES to the SIGMOID FUNCTION

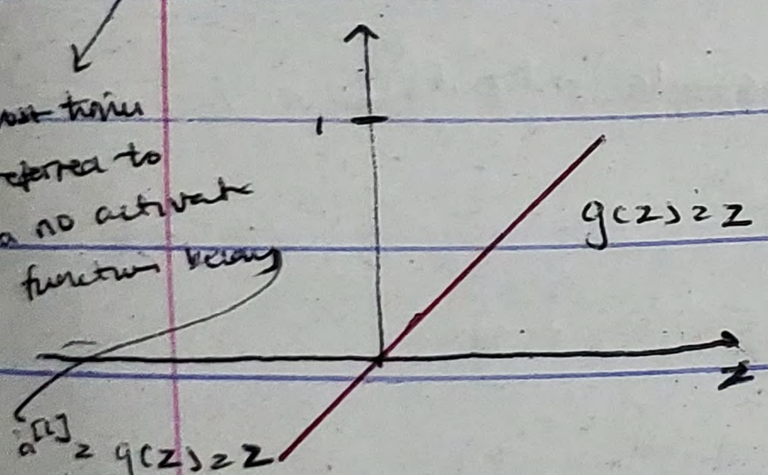


Something awareness might not be between 0 and 1

$$a_2^{[1]} = g(\overbrace{\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]}}^{z})$$

LINEAR ACTIVATION FUNCTION

most times referred to as no activation function being

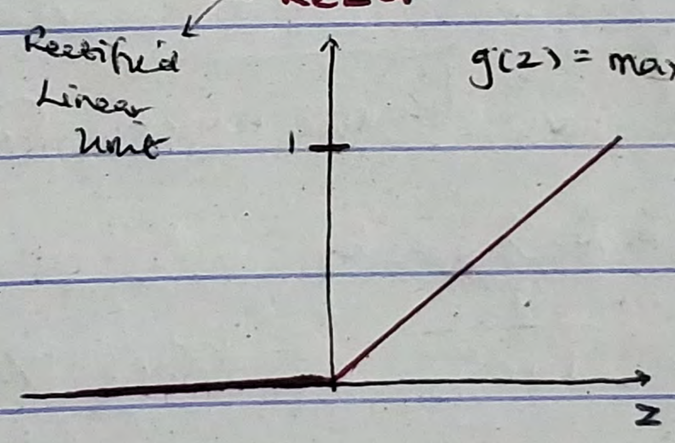


$a^{[l]} = g(z) = z$

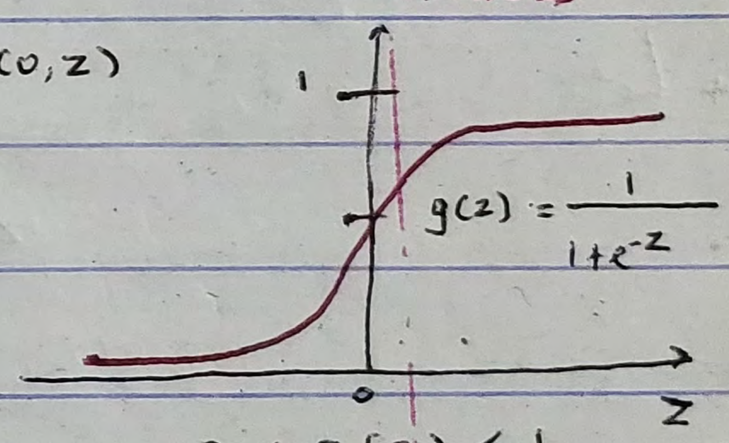
$a^{[l]} = z$

ReLU

Rectified Linear Unit



SIGMOID



$0 < g(z) < 1$

A: CHOOSING ACTIVATION FUNCTIONS

- Most time we choose activation functions based on label $\hat{y}$ ~~on expected output form~~

(i) Binary classification where $y = 0/1$ use <u>SIGMOID</u>

(ii) Regression where $y = +/-$ use <u>LINEAR ACTIVATION FUNCTION</u>

(iii) Regression where $y = 0$ or +ve number use <u>ReLU</u>

- What about the hidden layers?

★ <u>ReLU</u> is the most common choice. This is because ReLU is much faster to compute <because> and the function only goes flat in one part while the sigmoid goes flat in two places. This because gradient descent gets slow when functions have excess flat points

B: WHY DO WE NEED ACTIVATION FUNCTIONS

↳ basically, having no activation function results in a linear regression during.

~~You know~~ Your NN won't be able to fit ~~complex~~ the right models to complex data or better phrased, it won't be able to fit properly.

Even if you just use a linear activation function for the hidden layers and use a sigmoid function for the output layer, that NN will be equivalent to ~~LOGISTIC~~ REGRESSION

∴ Just use ReLU for your hidden layers

WK 2 → MULTICLASS CLASSIFICATION

- Multiclass classification refers to classification problems ~~where~~ where you can have more than just two output labels (i.e. more than 1 or 0)

→ Y can take on a small number of discrete categories >2

$\boxed{\text{SOFTMAX}}$ → a ~~generalization of logist~~ generalization of logistic regression to the multiclass classification context.

→ Let's look @ the formular for logistic regression (2 possible outputs);

$$z = \vec{w} \cdot \vec{x} + b$$

- $a_1 = g(z) = \dfrac{1}{1+e^{-z}} = P(y=1|\vec{x})$ e.g. $a_1 = 0.71$

$a_2 = 1 - a_1 = P(y=0|\vec{x})$ e.g. $a_2 = 0.29$

→ Now lets look @ SOFTMAX regression for a case of 4 possible outputs; $y = 1, 2, 3, 4$

- $z_1 = \vec{w}_1 \cdot \vec{x}_1 + b_1$; $a_1 = \dfrac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=1|\vec{x})$ e.g. 0.30
- $z_2 = \vec{w}_2 \cdot \vec{x}_2 + b_2$; $a_2 = \dfrac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=2|\vec{x})$ e.g. 0.20
- $z_3 = \vec{w}_3 \cdot \vec{x}_3 + b_3$; $a_3 = \dfrac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=3|\vec{x})$ e.g. 0.15
- $z_4 = \vec{w}_4 \cdot \vec{x}_4 + b_4$; $a_4 = \dfrac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=4|\vec{x})$ e.g. $= 1 - (0.3 + 0.2 + 0.15) = 0.35$

→ ∴ SOFTMAX regression for N possible outputs i.e. $y = 1, 2, 3, 4 \ldots N$

$z_j = \vec{w}_j \cdot \vec{x} + b_j$, $j = 1, \ldots, N$ $\qquad a_j = \dfrac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y=j|\vec{x})$

parameters $w_1, w_2, \ldots, w_N$

$b_1, b_2, \ldots, b_N$

LOSS FUNCTIONS & COST FUNCTION

recall for the Logistic regression where $a_1 = \frac{1}{1+ge^{-z}}$; and $a_2 = 1 - a_1$

$P(y=1|\vec{x})$ $P(y=0|\vec{x})$

$$\text{loss} = -y \log a_1 - (1-y) \log (1-a_1)$$

$$\therefore \text{loss} = -y \log a_1 - (1-y) \log a_2$$

$y=1$, loss $= -\log a_1$

$y=0$, loss $= -\log a_2$

$$J(\vec{w}, b) = \text{average loss}$$

for SOFTMAX regression; where

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \dots + e^{z_N}}$$

$$\vdots$$

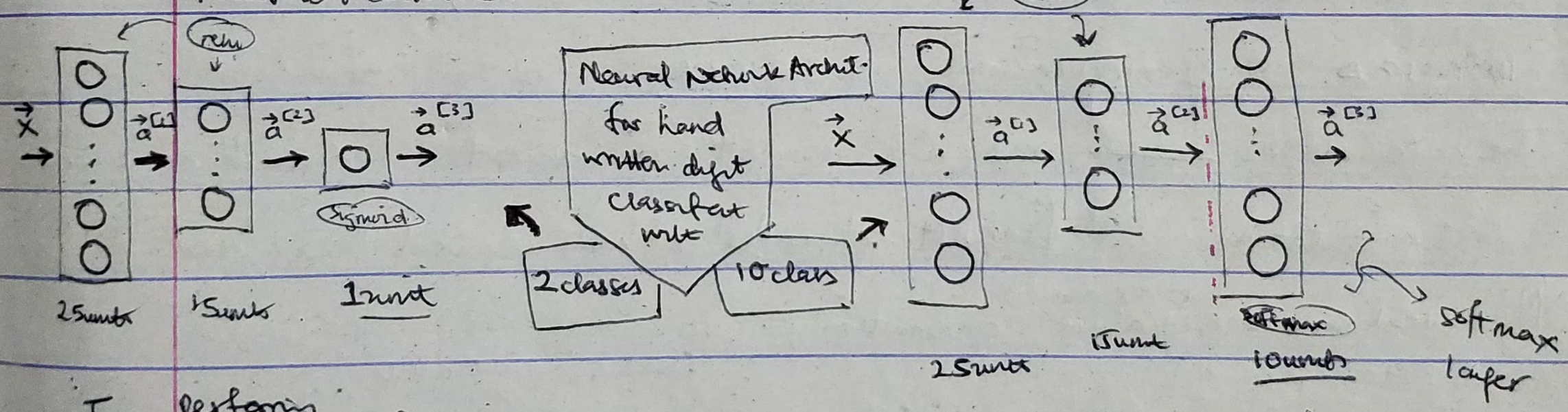
$$a_N = \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \dots + e^{z_N}}$$

$$\text{loss}(a_1, \dots, a_N, y) = \begin{cases} -\log a_1 & \text{if } y=1 \\ -\log a_2 & \text{if } y=2 \\ \vdots & \\ -\log a_N & \text{if } y=N \end{cases}$$

aka Crossentropy loss

NEURAL NETWORK WITH SOFTMAX

- Thus is essentially putting the SOFTMAX regression model in the output of the Neural Network.



- For performing ~~calculating~~ forward propagation in the NN w/ softmax; the $z_1^{[3]}$ to $z_{10}^{[3]}$ will be calculated in the softmax layer

$$\therefore \vec{a}^{[3]} = \begin{bmatrix} a_1^{[3]} \\ \vdots \\ a_5^{[3]} \\ \vdots \\ a_{10}^{[3]} \end{bmatrix}$$

- The softmax activation function has a unique property in that for example the sigmoid activation function's output $a_1^{[3]}$ is a function of only $z_1^{[3]}$

i.e. $a_1^{[3]} = g(z_1^{[3]})$ $\quad a_2^{[3]} = g(z_2^{[3]})$

while the output of the softmax activation function, e.g. $a_1^{[3]}$ is a function of $z_1^{[3]}$ to $z_{10}^{[3]}$

i.e. $\vec{a}^{[3]} = (a_1^{[3]}, \dots, a_{10}^{[3]}) = g(z_1^{[3]}, \dots, z_{10}^{[3]})$

Implementing softmax in Tensorflow will result in;

MNIST with Softmax

1. Specify the model
$f_{\vec{w},b}(\vec{x}) = ?$

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='softmax')
])
```

2. Specify loss and cost
$L(f_{\vec{w},b}(\vec{x}), y)$

```
from tensorflow.keras.losses import SparseCategoricalCrossentropy
model.compile(loss=SparseCategoricalCrossentropy())
```

3. Train on data to minimize $J(\vec{w},b)$

```
model.fit(X, Y, epochs=100)
```

→ Categorical because size > 1 category
→ sparse because Y can only take one value.

Note :- Even though this code is correct, its not the recommended version

IMPROVED IMPLEMENTATION OF SOFTMAX

* The code above ~~results in~~ has numerical roundoff errors and thus reduces the accuracy of the computation. But fear not citizen! Hope has arrived!

Numerical error = error due to premature approximation

→ **Normal Softmax Regression**

$(a_1, \dots, a_{10}) = g(z_1, \dots, z_{10})$

$$\text{Loss} = L(\vec{a}, y) = \begin{cases} -\log(a_1) & \text{if } y=1 \\ \vdots & \\ -\log(a_{10}) & \text{if } y=10 \end{cases}$$

```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='softmax')
])
model.compile(loss=SparseCategorical(Cross...()))
```

→ **More Accurate**

$$L(\vec{a}, y) \begin{cases} -\log\left(\dfrac{e^{z_1}}{e^{z_1}+\dots+e^{z_{10}}}\right) & \text{if } y=1 \\ \vdots & \\ -\log\left(\dfrac{e^{z_{10}}}{e^{z_1}+\dots+e^{z_{10}}}\right) & \text{if } y=10 \end{cases}$$

```
model = Sequential([
    Dense(units=25, activation="relu"),
    Dense(units=15, activation="relu"),
    Dense(units=10, activation="linear")
])
model.compile(loss=SparseCategoricalCrossentropy(
    from_logits=True))
```

...Entropy(from_logits=True)) ← soon

* Instead of performing the log transform of the output value then using that approx. value to calculate the loss → cost, the ~~loss → cost is calculated~~ neural network ~~Bp~~ outputs = ("linear") and the log transform is automatically applied to it when calculating the loss → cost

Also, note that we've changed the Neural Network to use a linear activation function rather than a softmax activation function. Therefore the output layer ~~note~~ no longer outputs $(a_1 \dots a_{10})$, rather it outputs $(z_1 \dots z_{10})$. This means

a transformation needs to be done once predicting

```
logits = model(X)
f_x = tf.nn.softmax(
        logits)
```

WK 2 MULTI-LABEL CLASSIFICATION

While multi class classification deals w/ situations where the output y could be any one of 2 or more possible categories there is still a different type of classification problem. The MULTI LABEL Classification problem is one where associated w/ each image, there could be multiple labels. In this problem, you could ask the image a series of questions. For example, a self driving car would need to learn how to classify a bus, a car and a pedestrian separately, if all of which could be or not be in the image @ the same time. ∴ the output y is a vector of 3 numbers : $y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ or $y = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ or $y = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

① In such a problem, you ~~can~~ might treat each question as a separate one and create NNs for each

② or you can alternatively train a single NN to simultaneously detect all 3 { cars, buses, pedestrian

WK 2 ADDITIONAL NEURAL NETWORK CONCEPTS

~~ADVANCED OPTIMIZATION~~ OPTIMIZATION ⇒ algorithms for minimizing cost functions which are better than gradient descent

∴ The Adam algorithm adjusts the learning rate (α) depending on how gradient descent proceeding. ADAM : Adaptive Moment estimation

→ It doesn't use one α but a different α (learning rate) for each parameter of the model

ADAM's intuition
- If w_j (or b) keeps moving a the same direction, increase α_j
- If w_j (or b) keeps oscillating, reduce α_j

For the Adam, the code implementation is very similar to what we had before except for the fact that we add one extra argument to the compile function

```
>> model.compile(optimizer = tf.Keras.optimizers.Adam(learning_rate = 1e-3),
                 loss = tf.Keras.losses.SparseCategoricalCrossentropy(from_logits = True))
```

* The Adam optimizer needs an initial learning rate which we set; $\alpha = 10^{-3} = 0.001$

→ Adam optimization algorithm has become a defacto standard.

ADDITIONAL LAYER TYPES

- What we've used before is the Dense layer type. It's one where each layer gets, as its input, all the activations of the previous layers. Others include

(I) CONVOLUTIONAL LAYER:

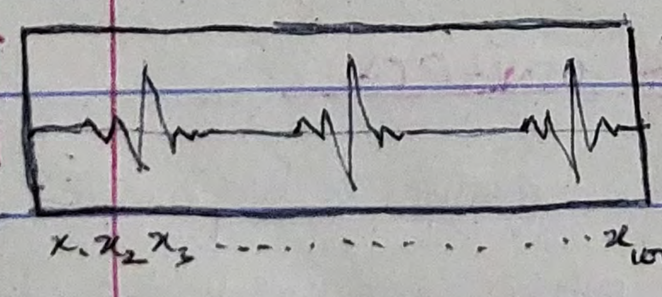
Here, each neuron only looks at PARTS of the previous layer's inputs. Why?

It's useful because (i) it leads to faster computation & (ii) it needs less training data, hence it's less prone to overfitting. Multiple convolutional layers in a NN results in a Convolutional Neural Network.

Illustration of a CNN <we'll use a 1D input instead of a 2D input>

Q. Can we classify EKG data based on its measurements (heights) @ different intervals to find out if the patient has heart problems or not?

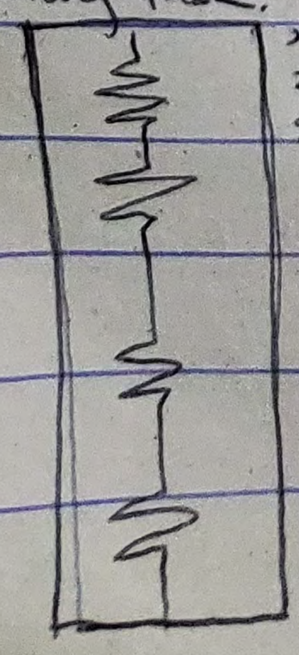
EKG / ECG
Electro Cardio-gram

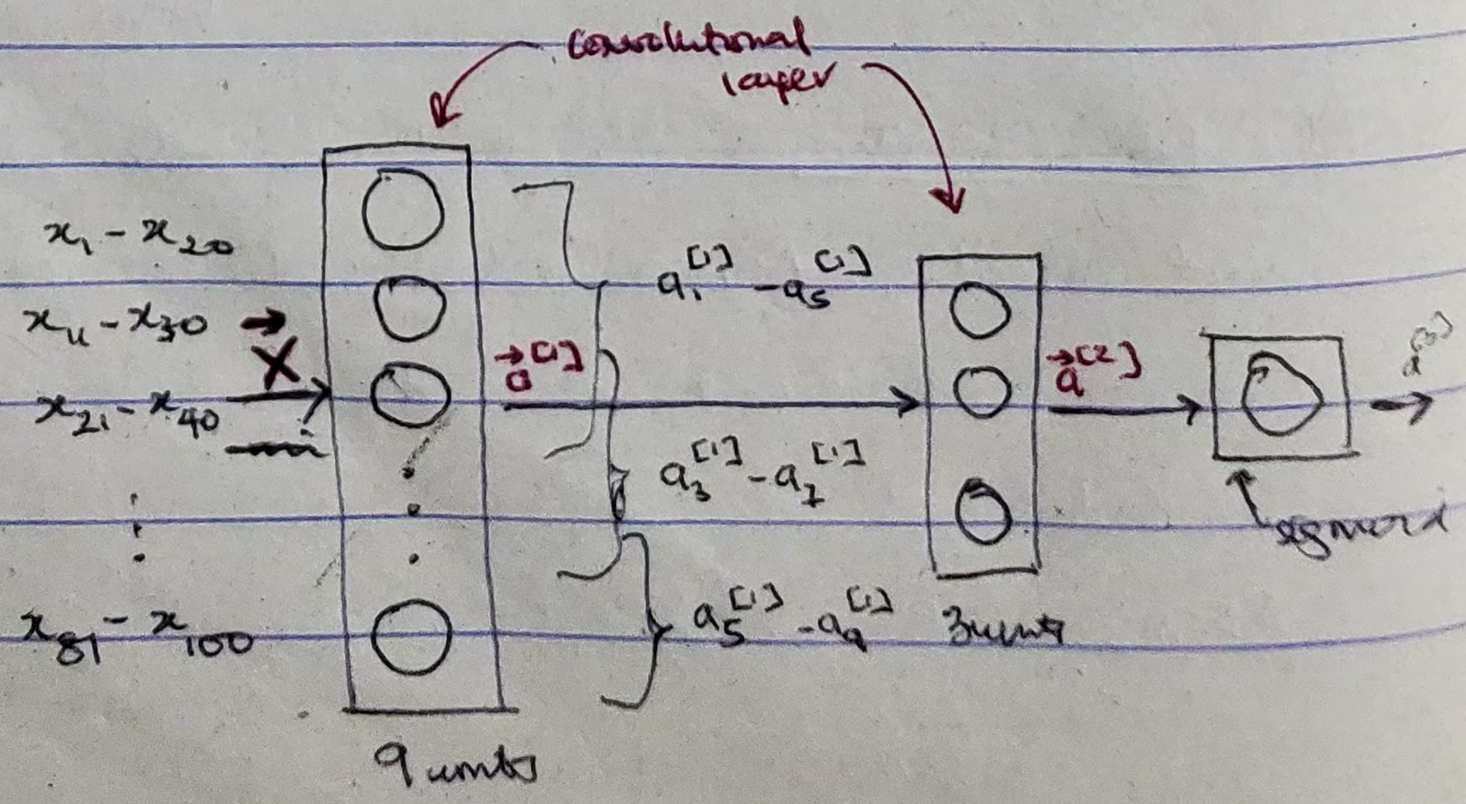


→ This is a list of numbers corresponding to the height (h) of the measurement surface @ different points in time

rotating the signal 90°

~~Learning task:~~





- The ~~ne~~ units in the first hidden layer looks through small windows @ the EKG time series data.
- The next hidden convolutional layer also looks @ 5 of the 9 activations from the previous hidden layer. Then the 3 values of $\vec{a}^{[2]}$ are passed into the sigmoid activation function which makes a binary classification regarding presence/absence of heart disease.

∴ with convolutional layers, you can choose how many neurons each layer will have and also how big the window of each neuron. By choosing these architectural parameters effectively, you can build new versions of Neural Networks that can be even more effective than the dense layer for some applications.

Wk2 BACK PROPAGATION

WHAT is A DERIVATIVE? If $w \uparrow \varepsilon$ (over up by) causes $J(w) \uparrow k \times \varepsilon$ then $\frac{\partial}{\partial w} J(w) = k$

[Example $w=3$, $J(w) = w^2 = 9$, $w \uparrow 0.001$

$J(3.001)$

$J(w) = 9.006001$

$\approx J(w) \uparrow 6 \times 0.001$ → $\frac{\partial}{\partial w} J(w) = 6$

If $w=2$, $J(w) = w^2 = 4$ $w \uparrow 0.001$; $J(w) = J(2.001) = 4.004001$

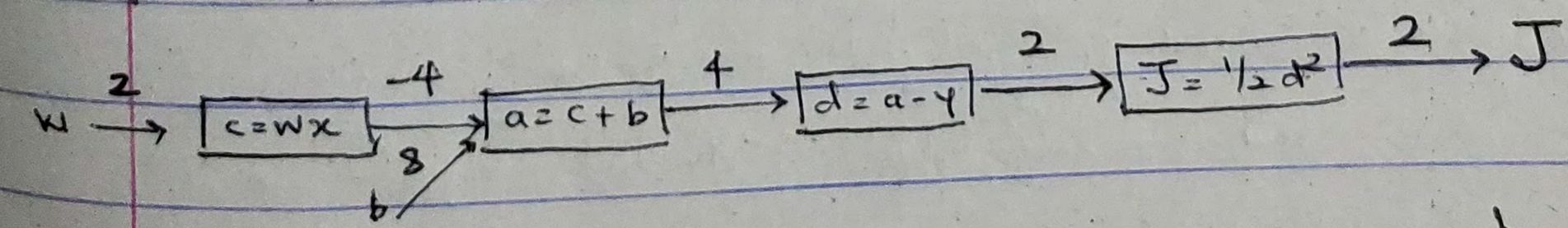
$\frac{\partial}{\partial w} J(w) = 4$ ← $J(w) \uparrow 4 \times 0.001$

this is a reasonable $J(w) = w^2$; $\frac{\partial}{\partial w} J(w) = 2w$

- Use python package **sympy** to calculate derivatives.

COMPUTATION GRAPHS working w/ a small neural network ; $\vec{x} \rightarrow \boxed{\bigcirc} \rightarrow a$ (linear activation)

- Let's find $J(w,b) = \frac{1}{2}(a-y)^2$ where $a = wx+b$, $w=2$, $b=8$, $x=-2$, $y=2$



Forward propagation step of how we compute a & $J(w,b)$

A set of nodes connected by arrows

Next: How do we find the derivate of J with respect to w & b? $\frac{\partial J}{\partial w}$ & $\frac{\partial J}{\partial b}$

Forward Prop →

← Backward Prop

Computing the DERIVATIVES ... $\frac{\partial J}{\partial w}$ & $\frac{\partial J}{\partial b}$

$w=2$; $b=8$; $x=-2$; $y=2$; $a=wx+b$; $J=\frac{1}{2}(a-y)^2$

$$w \xrightarrow[-4]{2} \boxed{c=wx} \xrightarrow[2]{-4} \boxed{a=c+b} \xrightarrow[2]{4} \boxed{d=a-y} \xrightarrow[2]{2} \boxed{J=\tfrac{1}{2}d^2} \xrightarrow{2} J$$

$b \xrightarrow{2}$

$\frac{\partial J}{\partial w}=-4$

$w\uparrow 0.001$; $c\uparrow -0.002$

$\therefore J\uparrow \underbrace{-0.004}_{4\varepsilon}$

$\frac{\partial J}{\partial w}=\frac{\partial c}{\partial w}\times\frac{\partial J}{\partial c}$

-2×2

$\partial J/\partial c=2$; $\partial J/\partial b=2$

$c\uparrow 0.001$; $a\uparrow 0.001$

$\Rightarrow J\uparrow 0.002$

$\frac{\partial J}{\partial c}=\frac{\partial a}{\partial c}\times\frac{\partial J}{\partial a}$

1×2

$b\uparrow 0.001$, $a\uparrow 0.001$

$\Rightarrow J\uparrow 0.002$

$\frac{\partial J}{\partial b}=\frac{\partial a}{\partial b}\times\frac{\partial J}{\partial a}$

1×2

$\frac{\partial J}{\partial a}=2$

$a\uparrow 0.001$, $d\uparrow 0.001$

$\therefore J\uparrow 0.002$

$\frac{\partial J}{\partial a}=\frac{\partial d}{\partial a}\times\frac{\partial J}{\partial d}$

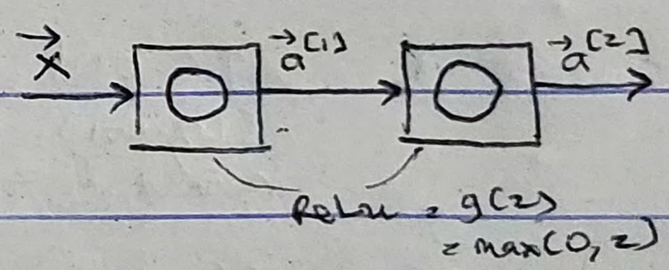
1×2

$\partial J/\partial d=2$

$d\uparrow \underbrace{0.001}_{\varepsilon}$; $J\uparrow \underbrace{0.002}_{2\varepsilon}$

$\frac{\partial J}{\partial w}$ ← back propagation ← $\partial J/\partial d$

NEURAL NETWORK EXAMPLE



$x=1$; $y=5$

$w^{[1]}=2$; $b^{[1]}=0$

$w^{[2]}=3$; $b^{[2]}=1$

$a^{[1]}=g(w^{[1]}x+b^{[1]})=w^{[1]}x+b^{[1]}=2$

$a^{[2]}=g(w^{[2]}a^{[1]}+b^{[2]})=w^{[2]}a^{[1]}+b^{[2]}=7$

$J(w,b)=\frac{1}{2}(a^{[2]}-y)^2=\frac{1}{2}(7-5)^2=2$

computational graph

$$\boxed{t^{[1]}=w^{[1]}\times x} \xrightarrow{2} \boxed{z^{[1]}=t^{[1]}+b^{[1]}} \xrightarrow{2} \boxed{a^{[1]}=g(z^{[1]})} \xrightarrow{2} \boxed{t^{[2]}=w^{[2]}\times a^{[1]}} \xrightarrow{6} \boxed{z^{[2]}=t^{[2]}+b^{[2]}} \xrightarrow{7} \boxed{a^{[2]}=g(z^{[2]})} \xrightarrow{7} \boxed{J=\tfrac{1}{2}(a^{[2]}-y)^2} \xrightarrow{2}$$

$2 \to w^{[1]}$; $0 \to b^{[1]}$; $3 \to w^{[2]}$; $1 \to b^{[2]}$

$\frac{\partial J}{\partial z^{[2]}}=2 \leftarrow \frac{\partial J}{\partial a^{[2]}}=2$

KK3 ADVICE FOR APPLYING MACHINE LEARNING

Suppose you've implemented regularised linear regression on housing prices;

$$J(\vec{w},b)=\frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)})-y^{(i)}\right)^2+\frac{\lambda}{2m}\sum_{j=1}^{n}w_j^2$$

But it makes unacceptably large errors in predictions. What do you try next?

- Get ~~two~~ more training examples
- Try smaller sets of features
- Try decreasing λ
- Try getting additional features
- Try adding polynomial features (x_1^2, x_2^2, x_1x_2, etc)
- Try ~~decreasing~~ increasing λ

On any given ML application, not all of the options presented would be fruitful if implemented. A key to being effective @ how you build a ML algorithm will be if you can find a way to make good choices about where to invest your time.

This can be known by performing "Machine Learning Diagnostic" → this is a test that you run to gain insight into what is/isn't working w/a learning algorithm, to gain guidance into improving its performance.

→ They takes time to implement but when implemented, it can be a very good use of time

EVALUATING A MODEL'S PERFORMANCE

- This is done by just splitting your data set into training sets and test sets. [70:30, 80:20]

Finally, you fit the parameters to the training data by minimizing cost function $J(\vec{w},b)$

$$J(\vec{w},b) = \left[\frac{1}{2m_{train}}\sum_{i=1}^{m_{train}}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m_{train}}\sum_{j=1}^{m} w_j^2\right]$$

only needed when training model

- Then to test the models performance, we compute the test error [average error on the test set]

$$J_{test}(\vec{w},b) = \frac{1}{2m_{test}}\left[\sum_{i=1}^{m_{test}}\left(f_{\vec{w},b}(\vec{x}^{(i)}_{test}) - y^{(i)}_{test}\right)^2\right]$$

* Notice the absence of the regularization term.

- You can also compute the training error <i.e. how well the model is doing on the training set>

$$J_{train}(\vec{w},b) = \frac{1}{2m_{train}}\left[\sum_{i=1}^{m_{train}}\left(f_{\vec{w},b}(\vec{x}^{(i)}_{train}) - y^{(i)}_{train}\right)^2\right]$$

∴ In an overfit model; the training error ($J_{train}(w,b)$) will be low while the test error ($J_{test}(w,b)$) will be high.

For a classification problem; [0/1]

- $J(\vec{w},b) = -\frac{1}{m_{train}}\sum_{i=1}^{m_{train}}\left[y^{(i)}\log(f_{\vec{w},b}(\vec{x}^{(i)})) + (1-y^{(i)})\log(1-f_{\vec{w},b}(\vec{x}^{(i)}))\right] + \frac{\lambda}{2m_{train}}\sum_{j=1}^{n} w_j^2$
- Test error, $J_{test}(\vec{w},b) = -\frac{1}{m_{test}}\sum_{i=1}^{m_{test}}\left[y^{(i)}_{test}\log(f_{\vec{w},b}(\vec{x}^{(i)}_{test})) + (1-y^{(i)}_{test})\log(1-f_{\vec{w},b}(\vec{x}^{(i)}_{test}))\right]$
- Train error; $J_{train}(w,b) = -\frac{1}{m_{train}}\sum_{i=1}^{m_{train}}\left[y^{(i)}_{train}\log(f_{\vec{w},b}(\vec{x}^{(i)}_{train})) + (1-y^{(i)}_{train})\log(1-f_{\vec{w},b}(\vec{x}^{(i)}_{train}))\right]$

Good, but there is a better and more common way of getting J_{train} & J_{test}

Instead of using the logistic loss to compute the test error and the training error, we measure the fraction of the test set and the fraction of the train set that the algorithm has misclassified

- This involves getting the algorithm to predict 1/0 on every example, $\hat{y} = \begin{cases} 1 & \text{if } f_{\vec{w},b}(\vec{x}^{(i)}) \geq 0.5 \\ 0 & \text{if } f_{\vec{w},b}(\vec{x}^{(i)}) < 0.5 \end{cases}$
  - we then count up the fraction of the examples where $\hat{y} \neq y$ → the true label

$J_{test}(\vec{w},b)$ = the fraction of the test set that has been misclassified

$J_{train}(\vec{w},b)$ = the fraction of the train set that has been misclassified

MODEL SELECTION & TRAINING | CROSS VALIDATION | TEST SETS

Supposing you have the following models for a problem

degree of x — 1st order

2nd order polynomial

$d=1$ 1. $f_{\vec{w},b}(\vec{x}) = w_1 x + b \rightarrow w^{<1>}, b^{<1>} \rightarrow J_{test}(w^{<1>}, b^{<1>})$

$d=2$ 2. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + b \rightarrow w^{<2>}, b^{<2>} \rightarrow J_{test}(w^{<2>}, b^{<2>})$

$d=3$ 3. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + w_3 x^3 + b \rightarrow w^{<3>}, b^{<3>} \rightarrow J_{test}(w^{<3>}, b^{<3>})$

⋮

$d=10$ 10. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + w_3 x^3 + \dots + w_{10} x^{10} + b \rightarrow w^{<4>}, b^{<4>} \rightarrow J_{test}(w^{<4>}, b^{<4>})$

~~To find how well the model~~ If you were to select the "best" model, you select the one w/ the least ~~test~~ $J_{test}(w^{<i>}, b^{<i>})$ error. This would be e.g. $J_{test}(w^{<5>}, b^{<5>})$. The problem with the method of model selection is that $J_{test}(w^{<5>}, b^{<5>})$ is likely to be an optimistic estimate of generalization error (i.e. $J_{test}(w^{<5>}, b^{<5>})$ < generalization error). Because an extra parameter d (degree of polynomial) was chosen using the test set.

EXPLANATION of the ABOVE in SIMPLER TERMS:

Using the test set to select the model introduces a bias towards the specific ~~model~~ characteristics of that test set. This can lead to an overly optimistic estimate of the generalization error, which is the error the model would make on unseen, real world data. The reason for this optimism is that the model selection process has effectively used the test set as part of the training process. By choosing the model based on the performance on the test set, the model implicitly adapts to the specific characteristics of that test set, which may not be representative of the broader data distribution.

- This can be solved by splitting our data into 3, instead of 2, subsets: Training, Cross validation, & Test set. 60:20:20

	Training examples	Cross validation examples	Test examples
Notation →	$(x^{(m_{train})}, y^{(m_{train})})$	$(x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$	$(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

* The cross validation ~~data~~ subset is used to cross check the validity of different models.

Also: validation set, development set; dev set.

Training error: $J_{train}(\vec{w}, b) = \frac{1}{2m_{train}}\left[\sum_{i=1}^{m_{train}} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2\right]$

Cross validation error: $J_{cv}(\vec{w}, b) = \frac{1}{2m_{cv}}\left[\sum_{i=1}^{m_{cv}} \left(f_{\vec{w},b}(\vec{x}_{cv}^{(i)}) - y_{cv}^{(i)}\right)^2\right]$

Test error: $J_{test}(\vec{w}, b) = \frac{1}{2m_{test}}\left[\sum_{i=1}^{m_{test}} \left(f_{\vec{w},b}(\vec{x}_{test}^{(i)}) - y_{test}^{(i)}\right)^2\right]$.

- Now we have our 10 models, we can evaluate each parameter $(w^{<i>}, b^{<i>})$ on the C.V. set instead of the test set, resulting in $J_{cv}(w^{<i>}, b^{<i>})$. In order to choose a model, we'll look @ the model w/the lowest cross validation error e.g. $J_{cv}(w^{<4>}, b^{<4>})$.

∴ our model selected = $w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

Finally if we want to report the estimate of the generalization error, we can do so using the test set to get the test error; $J_{test}(w^{<4>}, b^{<4>})$.

→ J_{test} is fair here because this the first time we're using the test set for anything.

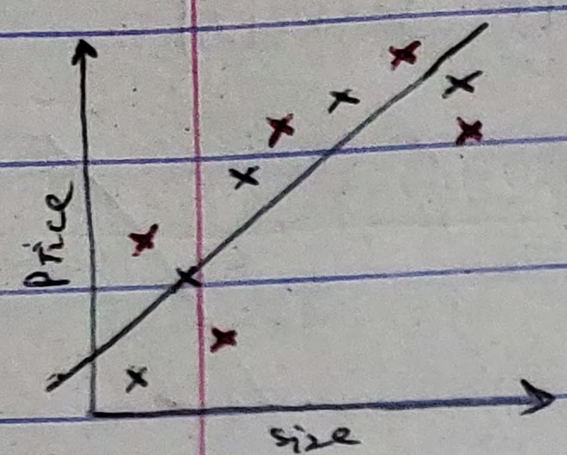
- This type of model selection also helps ~~use~~ us choose a neural network architecture: (i) how many layers and (ii) how many units each layer should have.

→ We use the ~~exhaustive~~ $\min J_{cv}$ to pick the model we would use and use J_{test} on that model to get an estimate of the generalization error.

NEXT BOOK

Wk 3

BIAS & VARIANCE



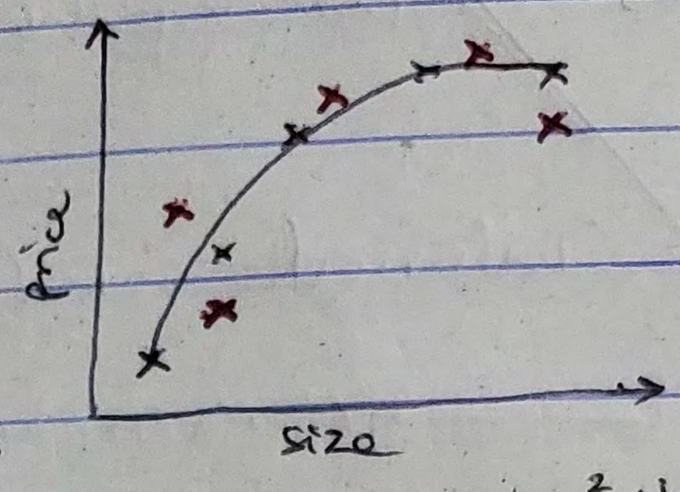
$f_{\vec{w},b}(x) = w_1 x + b$

(d=1)

HIGH BIAS <underfit>

J_{train} is HIGH

J_{cv} is HIGH



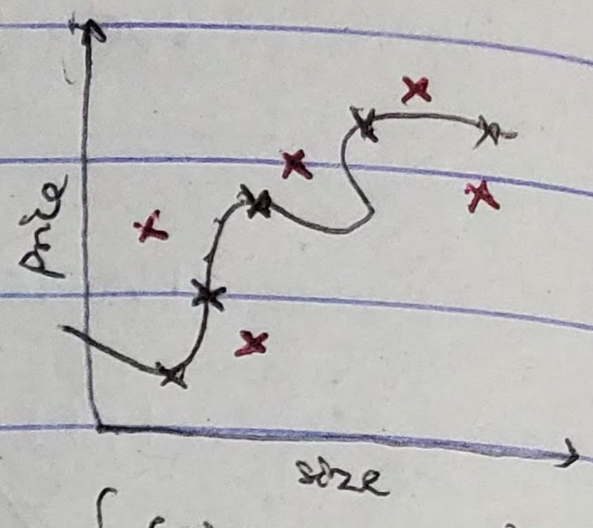
$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$

(d=2)

"Just right"

J_{train} is LOW

J_{cv} is LOW



$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

(d=4)

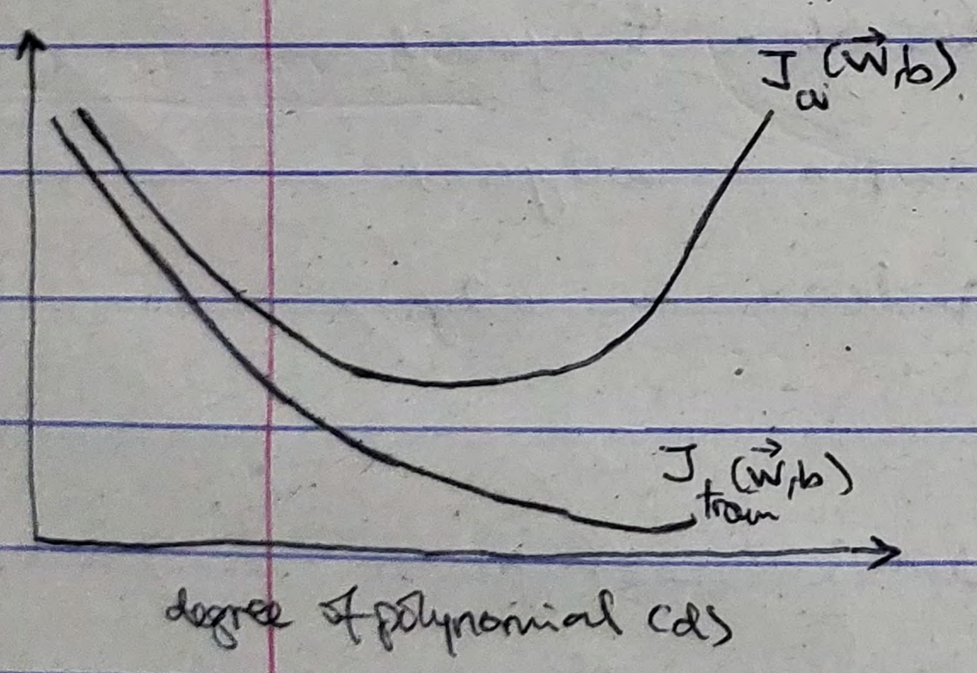
HIGH VARIANCE <overfit>

J_{train} is LOW

J_{cv} is HIGH

∴ the values of J_{train} & J_{cv} tell whether or not there's a high bias/variance.

~~HOW TO TELL~~



HOW TO TELL IF YOUR ALGORITHM HAS A BIAS OR VARIANCE PROBLEM

- High bias (underfit) J_{train} will be high ($J_{train} \approx J_{cv}$)
- High variance (overfit) $J_{cv} \gg J_{train}$ (J_{train} may be low)
- High bias & High variance → J_{train} will be high and $J_{cv} \gg J_{train}$

This can occur when training Neural Networks

doesn't really happen for linear model

REGULARIZATION & BIAS/VARIANCE

- Here looking @ how λ affects the bias/variance of the algorithm which in turn affects the overall performance.

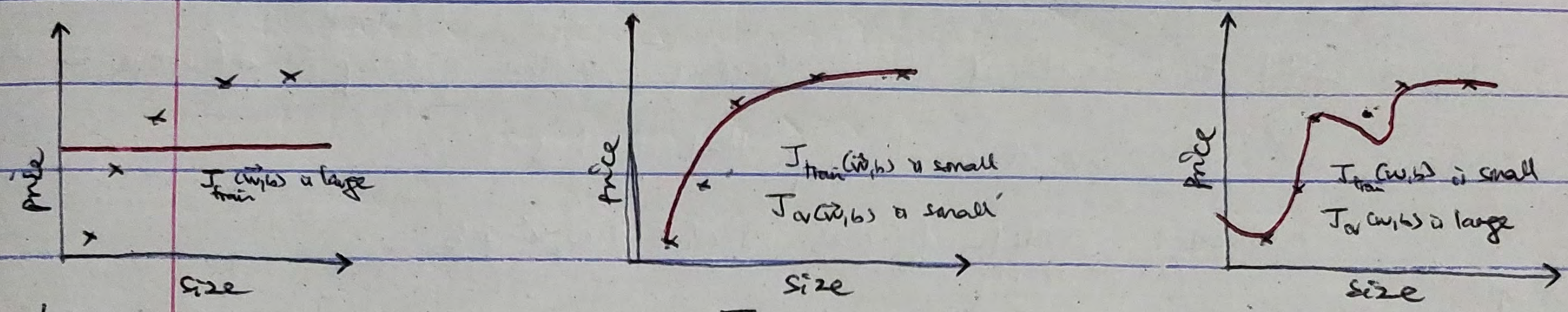
λ → regularization parameter.

Problem: If we're fitting a 4th order polynomial model to a data and we're using regularization, how can we choose the appropriate value for λ

model: $f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

we're using the Housing example

$$J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$



Large λ
- High bias (underfit)

$\lambda = 10000$, $w_1 \approx 0$, $w_2 \approx 0$

$f_{\vec{w},b}(\vec{x}) \approx b$

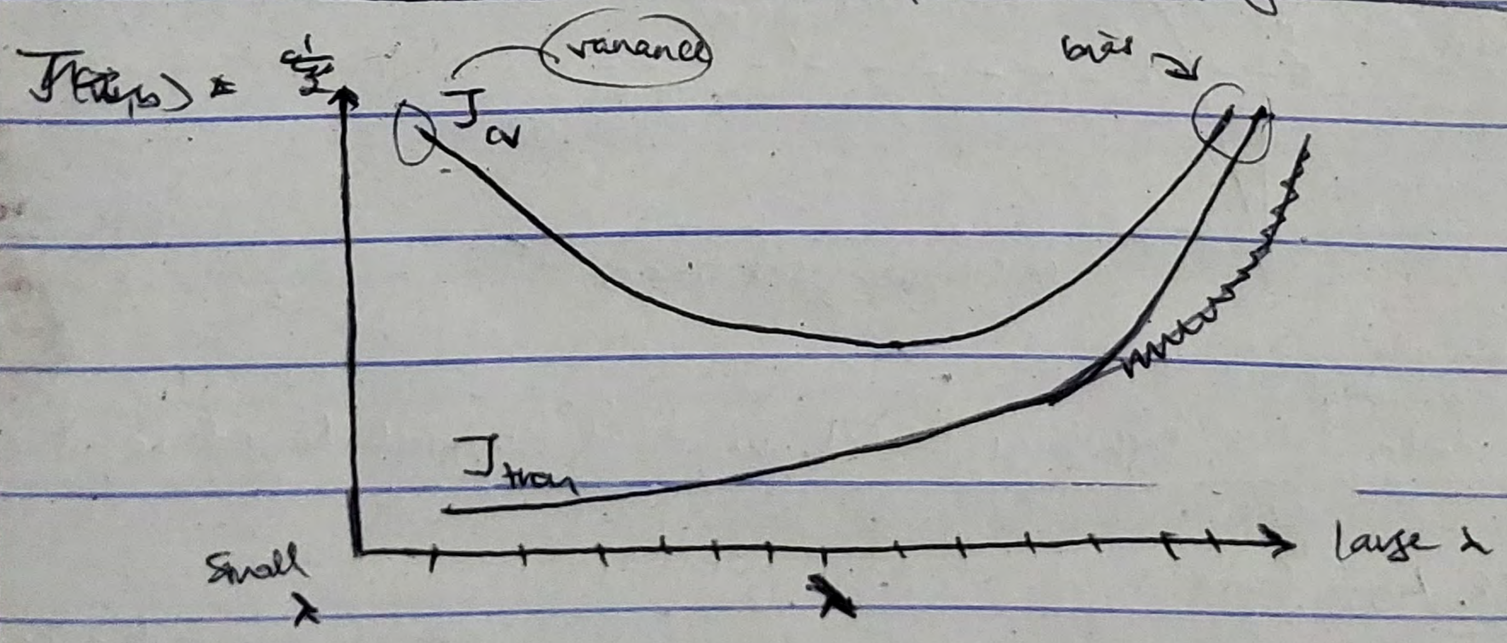
Intermediate λ
- "Just right"

Small λ
- High variance (overfit)

$\lambda \approx 0$

Intuition: Try out a lot of different values of λ and evaluate the J_{cv} @ each different point and pick the λ with the least J_{cv}

Bias & Variance as a function of regularization param λ



ESTABLISHING A BASELINE LEVEL OF PERFORMANCE

- This can be done by measuring
  - Human level performance
  - Competing algorithms performance
  - Guess based on experience

∴ The two key quantities to measure are;

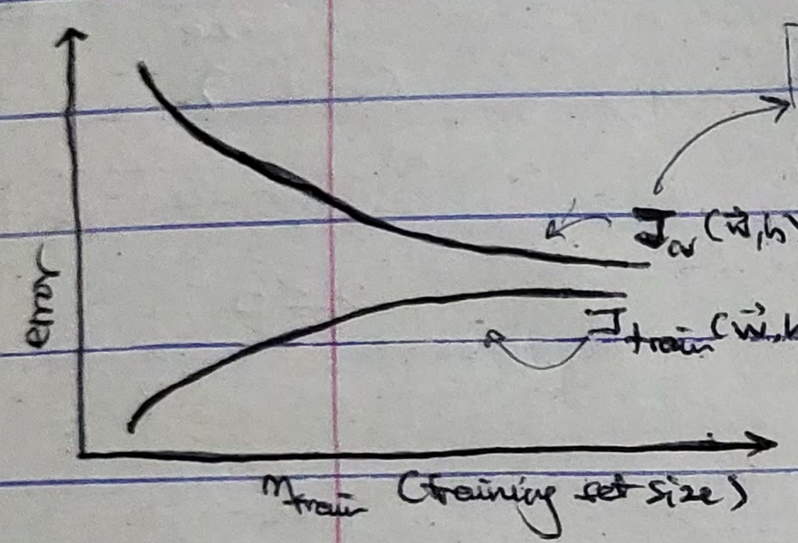
— (i) The difference between Training error & Baseline

— (ii) The difference between Cross validation error & Training error

LEARNING CURVES

- This is a way to understand how the learning algorithm is doing as a function of its experience → i.e. the amount of training that it has.

Example: model ; $f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$



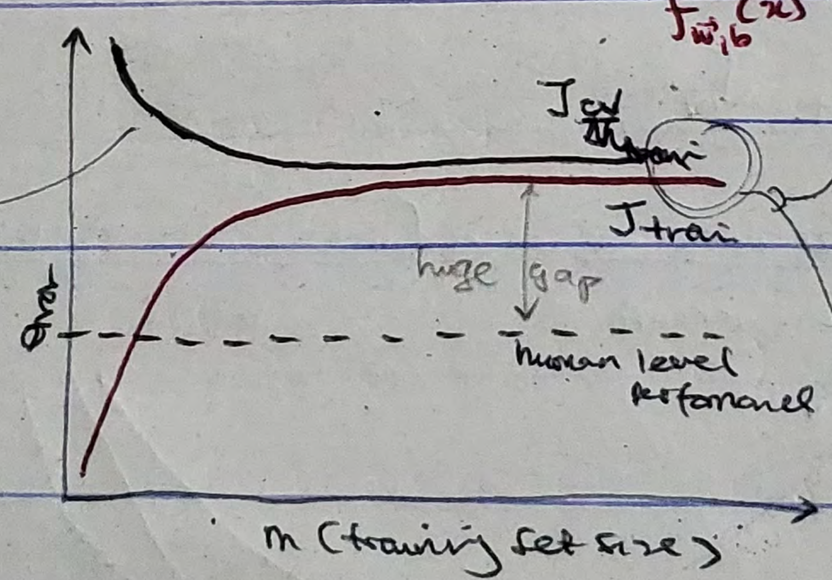
As the training set gets bigger, we learn a better model
∴ J_{cv} goes down

As training size increases, it becomes increasingly hard to fit the model ($f_{w,b}(x)$) to the data

∴ increases J_{train}

#1. HIGH BIAS LC

$f_{\vec{w},b}(x) = w_1 x + b$



J_{cv} also flattens out after a while as m increases

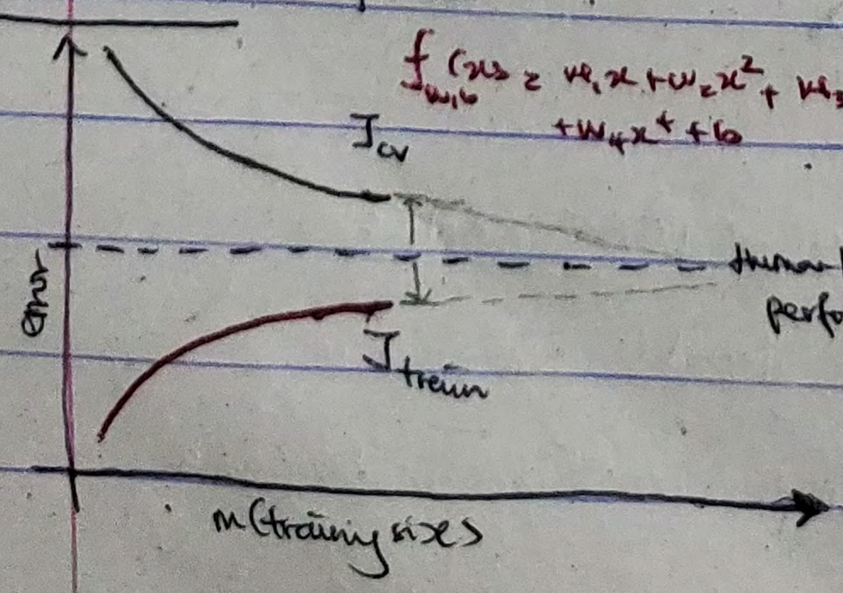
J_{train} curve flattens out after a while because as m increases @ a certain point, the simple model doesn't change that much

→ both stretch out to the right and do not meet

∴ throwing more data alone to a high bias model wouldn't do much to reduce the error.

#2. HIGH VARIANCE, Learning Curve

$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$ w/ small λ



- Looking @ the LC for high variance, we can see that if m increases to a certain large value, J_{train} approaches the human level performance as well as J_{cv} too.

- This shows that, just by increasing the training set size, it is possible to reduce J_{cv} and get the algorithm to perform better.

→ in reference to the question posed earlier this week

DEBUGGING A LEARNING ALGORITHM

You've implemented regularized linear regression on housing prices

$$J(\vec{w}, b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{\vec{w},b}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

But it makes unacceptably large errors in predictions. What do you try next

(i) Get more training examples ..... fixes high variance [✓]

(ii) Try smaller sets of features ..... fixes high variance [✓]

(iii) Try getting additional features ..... fixes high bias [B]

(iv) Try adding polynomial features ..... fixes high bias [B]

(v) Try decreasing λ ..... fixes high bias [B]

(vi) Try increasing λ ..... fixes high variance [✓]

→ excess features adds complexity to the model thereby increasing ~~variance~~ variance

- Note: Reducing training examples will an ~~answer~~ answer to reducing BIAS

BIAS/VARIANCE & NEURAL NETWORKS

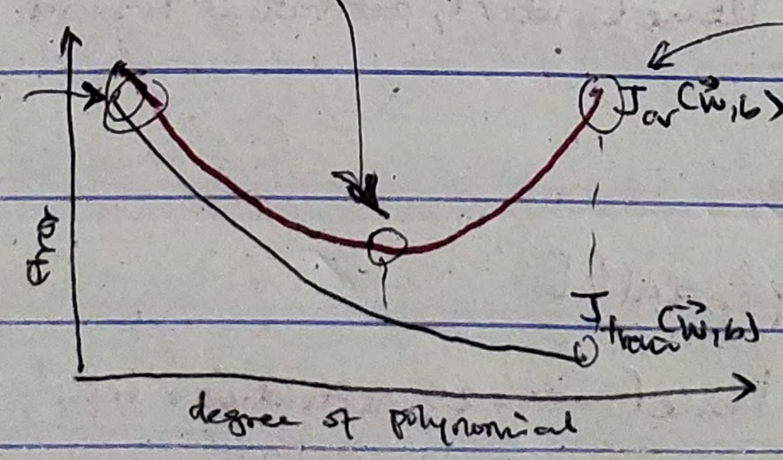
the bias-variance tradeoff with ML:

$f_{w,b}(x) = w_1 x + b$ — simple model

$f_{w,b}(x) = w_1 x + w_2 x^2 + b$

$f_{w,b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$ — complex model

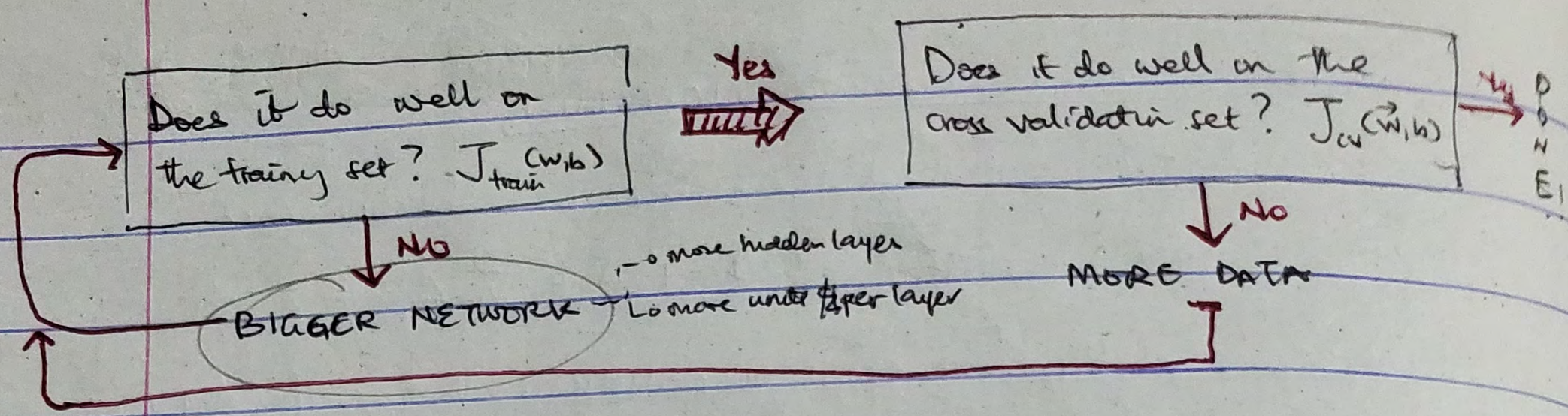
HIGH BIAS ← trade off → HIGH VARIANCE



- w/ Neural Networks, there is a way out of this tradeoff dilemma w/ some caveats.

→ Large Neural Networks are low bias machines

Check next page for a workflow for NN creation.



LIMITATIONS
- Training bigger networks require more powerful GPUs.
- More data is simply hard to come by.

- In Neural Networks, we don't fear the risk of overfitting when we increase the no. of hidden layers and/or units in each layer (i.e. opting for a bigger NN). A large neural network will usually do as well or better than a smaller one so long as regularization is chosen appropriately

→ Therefore, it doesn't hurt to go for a larger NN if you have access to enough computational power.

NEURAL NETWORK REGULARIZATION

- Unregularized MNIST model;

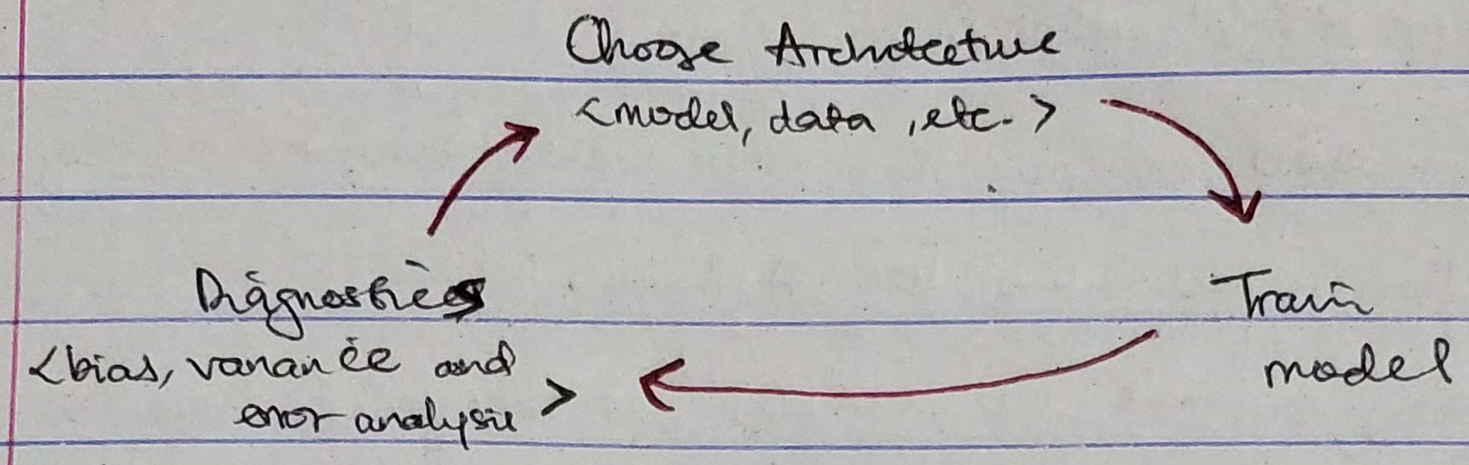
```
model = Sequential([
        Dense(units=25, activation="relu"),
        Dense(units=15, activation="relu"),
        Dense(units=1, activation="sigmoid")])
```

- Regularized MNIST model;

```
model = Sequential([
        Dense(units=25, activation="relu", kernel_regularizer=L2(0.01)),
        Dense(units=15, activation="relu", kernel_regularizer=L2(0.01)),
        Dense(units=1, activation="sigmoid", kernel_regularizer=L2(0.01))
        ])
```

WK 3 MACHINE LEARNING DEVELOPMENT PROCESS

ITERATIVE LOOP OF ML DEVELOPMENT



For example: Let's say that we want to build a spam classifier.

Supervised learning: $\vec{x}$ = features of email

Y = spam (1) or not spam (0)

Features: List the top 10,000 words to compute $x_1, x_2, \ldots, x_{10,000}$

$$\vec{x} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \begin{matrix} a \\ \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discount} \\ \vdots \end{matrix}$$

sample ~~example~~ message

From: cheapsales@buystuffhome.com
To: Andrew Ng
Subject: Buy now!

Deal of the week! Buy now!
Rolex watches - $100
Medicine (any kind) - £50
Also low cost Mortgages available

~ How to try to reduce your spam classifier's error?

* Collect more data. e.g. "Honeypot" project
* Develop sophisticated features based on email routing (from email header).
* Define sophisticated features from email body. e.g. should "discounting" & "discount" be treated as the same word?
* Design algorithms to detect misspellings, e.g., w4tches, medicine, m0rtgage

→ Remember our lesson on BIAS/variance trade-off. Not all these are worth chasing depending on the problem. Extra data won't help + high bias problem

- That's how the iterative loop is run. You choose your next step based on the problem you're facing. You can't just go with any step.

ERROR ANALYSIS

→ Basically involves manually going through the misclassified examples to gain insight into where the algorithm is going wrong.

Example: $m_{cv} = 500$ examples in cross validation set
(spam classifier)

Algorithm misclassifies 100 of them.

Error Analysis → Manually examine 100 examples and categorize them based on common traits

- Pharma : 21
- Deliberate misspellings : 3
- Unusual email routing : 7
- Steal passwords (phishing) : 18
- Spam message in embedded image : 5

- This analysis shows that where to focus our energy in building the model as "deliberate misspellings" & "embedded image" have low net impact of 3/100 & 5/100 while "pharma" & "phishing" spam mails have a much larger net impact and require more attention.

- Note: These categories aren't mutually exclusive. Some could overlap.
- If a large number (e.g. 1000) of examples are errors, you can randomly sample 100 examples and perform error analysis on them

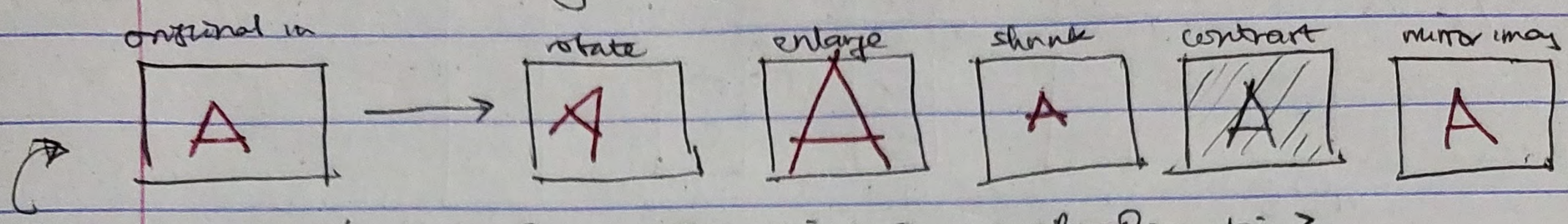
LIMITATION of ERROR ANALYSIS: It's difficult on problems that humans aren't good at e.g. "what ads will people click on?"

ADDING DATA

If getting additional data is quite difficult to come by, you can try adding more data of the types where error analysis has indicated it might help. (e.g. Pharma spam)

- Beyond getting brand new training examples (x, y), another technique = **DATA AUGMENTATION**

Data Augmentation = Modifying an existing training example to create a new training example



training example for image for OCR (optical Character ~~Reg~~ Recognition)

- You can also warp and distort the letter 'A' to create more training examples

- Data augmentation for speech also works.
  - Original audio → + Noisy background: Crowd, + Noisy background: car, + Bad cellphone connection audio
  - This is a critical technique for speech recognition.

Tip: The changes or distortions made to data should be representative of the type of noise/distortions in the test set.

→ Adding purely random/meaningless noise to your data doesn't help at all.

- **Data Synthesis** → Involves using artificial data inputs to create a new training example.

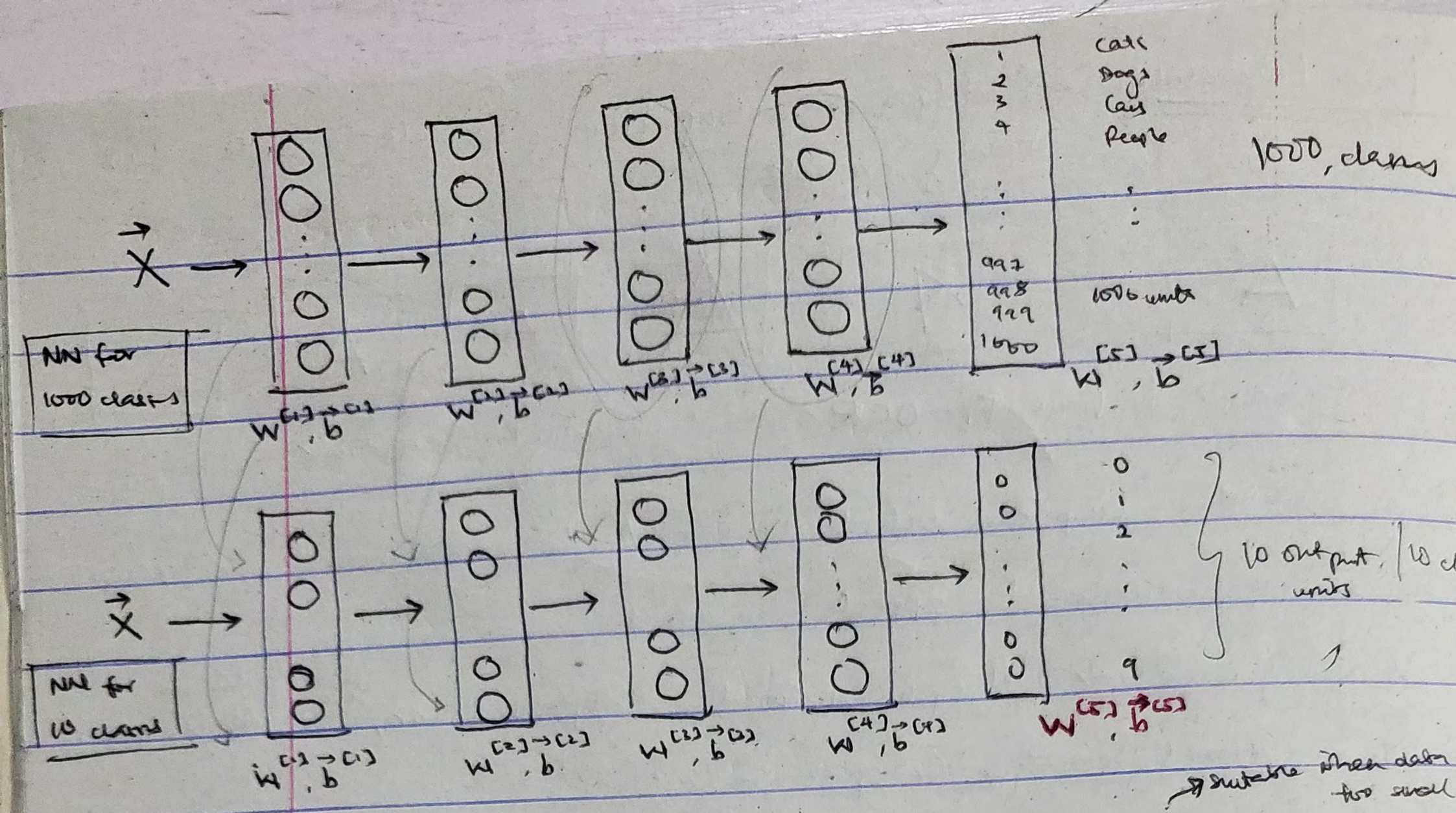
"If you're working on a photo OCR project, the real data could include text from billboards, magazines, plate numbers and all ~~and~~ To create synthetic data, you can use your text editor to type out words in different font styles ~~& of different colors~~ and screenshot them and apply different colors.

- This technique is mostly used for computer vision tasks

TRANSFER LEARNING: USING DATA FROM A DIFFERENT TASK

Let's say that you want to build a NN that'll recognise numbers 0 to 9 but you don't have enough data. ~~What you can~~ You then realise that you built a ~~NN~~ NN for multi-label classification trained on 1 million images to ~~c~~ classify each into 1000 classes.

You can copy over the NN architecture for that project i.e. the layers and the parameters $W^{[i]}, b^{[i]}$ for the hidden layers and replace the output layer with 10 output units (thereby needing new parameters which would be gotten via training).



suitable when data is too small

- Now we have two options: (i) only train output layers parameters using the gotten parameters as initialization → (ii) train all parameters → suitable when data is fairly okay

- The algorithm ~~[illegible]~~ is called TRANSFER LEARNING because the intuition is by learning to recognize cats, dogs, cars, people ... 1000, it would hopefully have learnt some plausible sets of parameters for the earlier layers for processing image inputs. Then by transferring these parameters ~~for the earlier layers~~ to the new neural network, the new NN starts off w/ the parameters in a much better place so that we have just little bit of further learning. Hopefully, it can end up at a pretty good model.

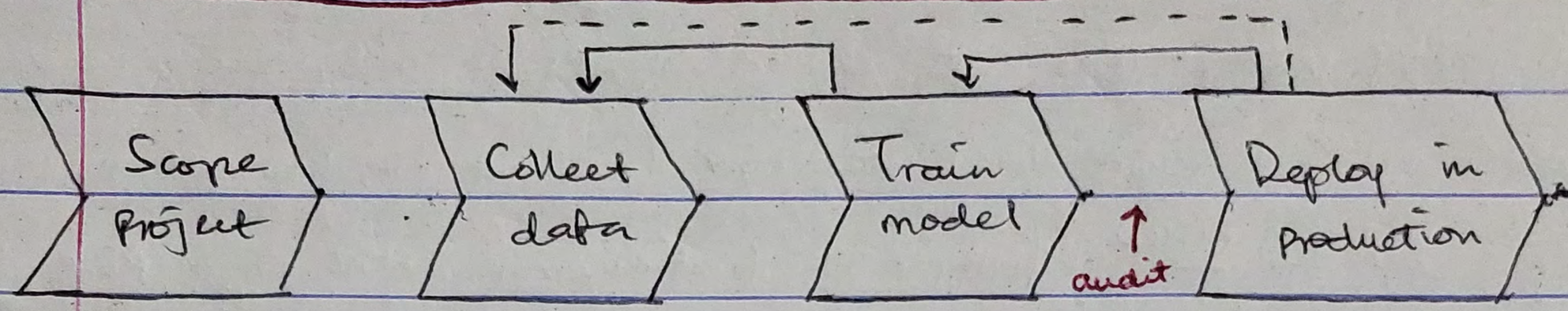
- There two steps are:

You might not even be the one that did it, you can download someone else pre-trained model

(i) Supervised pretraining: Training the NN on a very large datasets of example a 1,000,000 images of a somewhat unrelated task

(ii) Fine tuning: Here you take the parameters initialized or gotten from supervised pretraining and then run gradient descent further to fine tune the weights to suit the specific application that you have

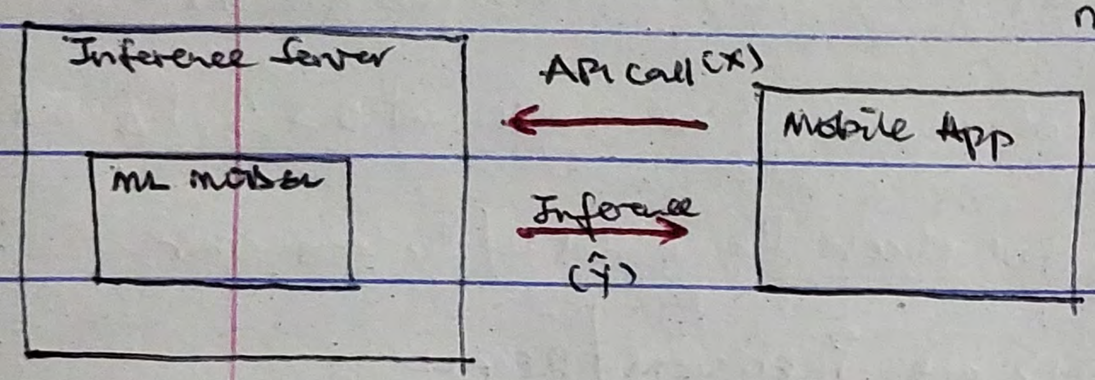
WK3 **Full cycle of a machine learning project**



<Define project> <Define & collect data> <Training, error analysis & iterative improvement> <Deploy, monitor and maintain system>

Monitor performance even when its deployed for use

DEPLOYMENT: Eg Implementing an Application that calls via the API & Inference Server with prediction model



→ A common way of implementing a high performing ML model (ex. speech recognition model), is to take the model and implement it in a server (aka Inference Server). Its job is to call your machine learning model in order to make predictions. Then if your team has implemented a mobile app, when a user talks to the mobile app, it can then make an API call to pass to your inference server the audio clip that was recorded and the inference server's job is to apply the machine learning model to it and then return the prediction to the mobile app, in this case, it would be the text transcript of what was said.

- Software engineering may be needed for: (i) Ensure reliable and efficient predictions, (ii) Scaling; (iii) Logging (iv) System monitoring (v) Model updates

* **MLOps** <machine learning operations>: Refers to the practice of systematically building, deploying and maintaining machine learning systems.

FAIRNESS, BIAS & ETHICS

- Get a diverse team to brainstorm things that might go wrong, w/ emphasis on possible harm to vulnerable groups
- Carry out literature search on standards/guideline for your industry
- Audit systems against possible harm prior to deployment.
- Develop mitigation plan (if applicable) and after deployment, monitor for possible harm.

→ Guidelines

Wk 3 SKEWED DATASETS

- When working ~~with~~ on problems with skewed datasets, we usually use a different error metric rather than just classification error to figure out how well the algorithm is doing. See the ~~error~~ problem w/ using a classification error metric on a Rare disease classifier example:

Train classifier, $f_{\vec{w},b}(\vec{x})$ $\begin{cases} y=1 & \text{if disease present,} \\ y=0 & \text{otherwise} \end{cases}$

$\boxed{\text{Note: only 0.5\% of patients have disease,}}$

with a classification error metric you'll get 1% error on test set (99% correct diagnosis)

~~This~~ This is so incorrect that just have the model "print("y=0")" does a better job with 99.5% accuracy and 0.5% error. This shows that when we have models w/ 0.5%, 1% and 1-2% errors (around some small rate), you can't just choose the model w/ the least error.

- A common pair of error metrics (alternatives) are PRECISION / RECALL

→ Here, $y=1$ in presence of rare class we want to detect.

When ~~evaluating~~ ~~constructing~~ evaluating a learning algorithm's performance on one rare class

We make use of a CONFUSION MATRIX

		ACTUAL CLASS 1	ACTUAL CLASS 0
PREDICTED CLASS	1	TP 15	FP 5
	0	FN 10	TN 70
		⇓ 25	⇓ 75

- TP = True Positive ; TN = True -ve
- FP = False +ve ; FN = False -ve

When ~~evaluating~~ the alg's performance on the cv/test set, we would count up the values that correspond to the 2x2 matrix.

e.g. where A.C = 1 and P.C = 1 is 15 out of 100.

In the example, the skew cont much because on the actual class count (1) = 25 & count (0) = 75.

- PRECISION: Of all patients where we predicted $y=1$, what fraction actually have the rare disease?

$$\frac{\text{True Positive}}{\text{No. of Positives predicted}} = \frac{TP}{TP+FP} = \frac{15}{15+5} = \frac{\cancel{15}^{3}}{\cancel{20}_{4}} = 0.75$$

∴ this algorithm has a precision of 75%

- **RECALL**: Of all patients that actually have the rare disease, what fraction did we ~~actually~~ correctly detect as having it?

$$\frac{\text{True Positives}}{\text{No. of Actual Positives}} = \frac{\text{True P.}}{TP + FN} = \frac{15}{15+10} = \frac{\cancel{15}^{3}}{\cancel{25}_{5}} = 0.6$$

∴ this algorithm has a recall of 60%

Thus Precision & Recall ~~help~~ help guard against ~~model~~ poorly performing models for example our "print('y=0')" model. This would have a confusion matrix of;

		Actual class 1	Actual class 0
Predict class	1	TP 0	FP 0
	0	FN 25	TN 75

∴ Precision $= \frac{0}{0+0} = \infty$ ..... we say that precision is equal to 0

Recall $= \frac{0}{0+25} = 0$

In summary precision helps us know if when a learning algorithm tells us that a patient has the disease, there's a good chance that the patient actually has the disease <75%>

Also ~~it~~ Recall makes sure that of all the patients that have the disease, it's helpful to diagnose a reasonable fraction of them <60%>

→ • Precision : Is the model predicting the rare class right?

→ • Recall : Is the model actually detecting majority of the rare class?

TRADEOFF OFF PRECISION & RECALL

Lets say we have a Log. reg. model : $0 < f_{\vec{w},b}(\vec{x}) < 1$

- Predict 1 if $f_{\vec{w},b}(\vec{x}) \geq 0.5$
- Predict 0 if $f_{\vec{w},b}(\vec{x}) < 0.5$

(i) Suppose we want to predict y=1 (rare disease) only if we're very confident : Higher Precision, Lower Recall

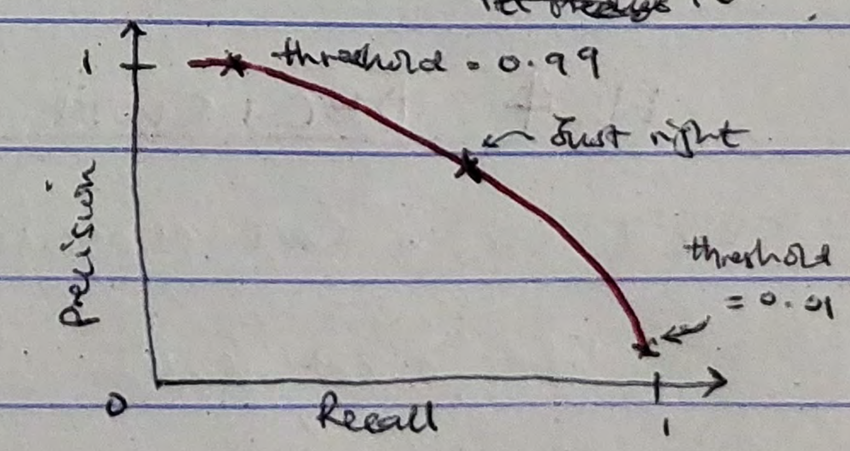
(ii) Suppose we want to ~~predict~~ avoid missing too many cases of rare disease i.e. when in doubt, predict y = 1

Lower Precision, Higher Recall

In general; predict 1 if: $f_{\vec{w},b}(\vec{x}) \geq$ threshold

set based on what you have in mind high for case (i) and low for case (ii)

$$\text{Precision} = \frac{TP}{\text{Tot Predicted } +ve}$$



Making this plot allows one to pick the threshold that one wants. Note that choosing the threshold ~~not done~~ not cross validation because its up to you to specify the value. ∴ many applications require you to manually choose the threshold

If you want to trade off precision & recall but you don't want to do so manually, you can make use of the metric → **F1 SCORE**

- F1 Score is most times used to automatically combine precision and recall to help you pick the best trade off between the two. The challenge with precision | recall → Challenge: Now you're that you're evaluating your algorithms using 2 different metrics.

Instead of taking the average of both <which is not advisable>, compute the F_1 score which is a way of computing an average of sorts but it gives more attention to which ever of the values is lower.

$$F_1 \text{ Score} = \frac{1}{\frac{1}{2}\left(\frac{1}{P} + \frac{1}{R}\right)} = 2 \cdot \frac{PR}{P+R}$$

Algorithms	Precision (P)	Recall (R)	~~Average (Mean)~~ Not advisable!	F_1 score
A	0.5	0.4	0.45	0.444
B	0.7	0.1	0.4	0.175
C	0.02	1.0	0.501	0.0392

→ this corresponds to a model that predicts "y = 1"

∴ Algorithm A is the better model.

Wk4 DECISION TREES

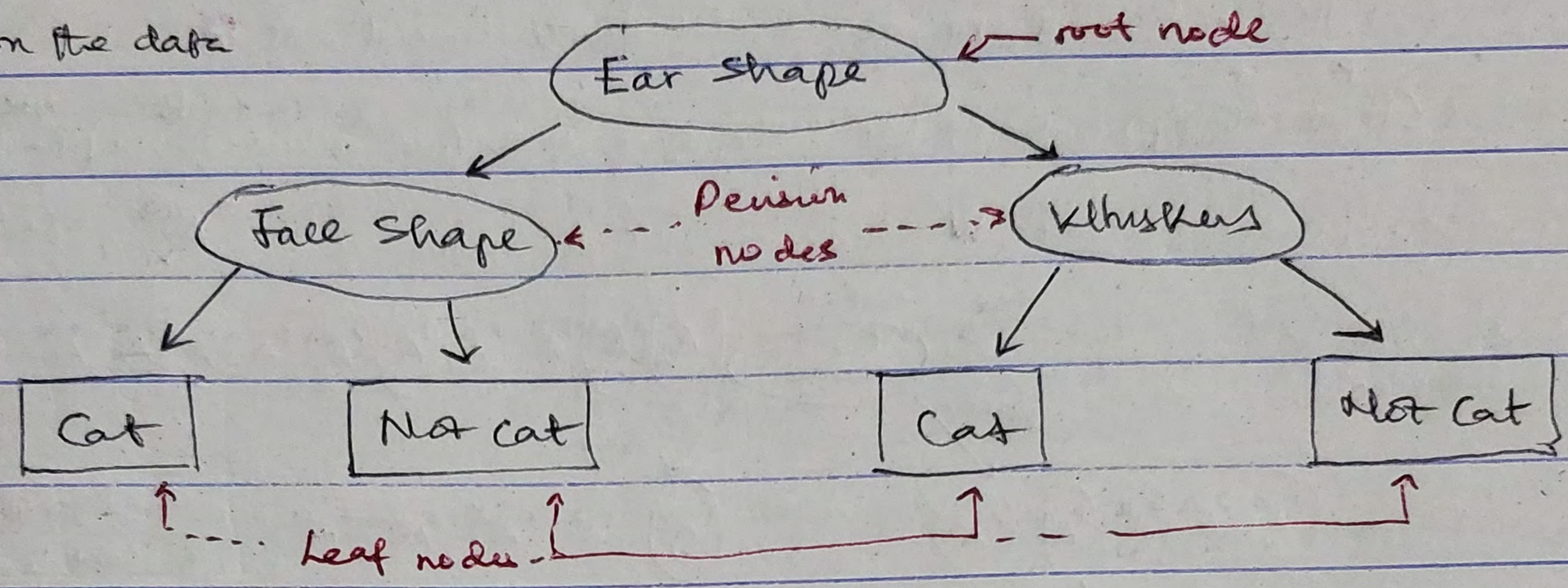
- We'll use a CAT CLASSIFICATION example to explain how Decision trees work.
  - → You have a cat adoption centre and given a few features, you want your classifier to determine whether an animal is a cat or not.

ms	Ear shape	Face shape	Whiskers	Cat
1	Pointy	Round	Present	1
2	Floppy	Not round	Present	1
3	Floppy	Round	Absent	0
4	Pointy	Not round	Present	0
5	Pointy	Round	Present	1
	...	...	...	...

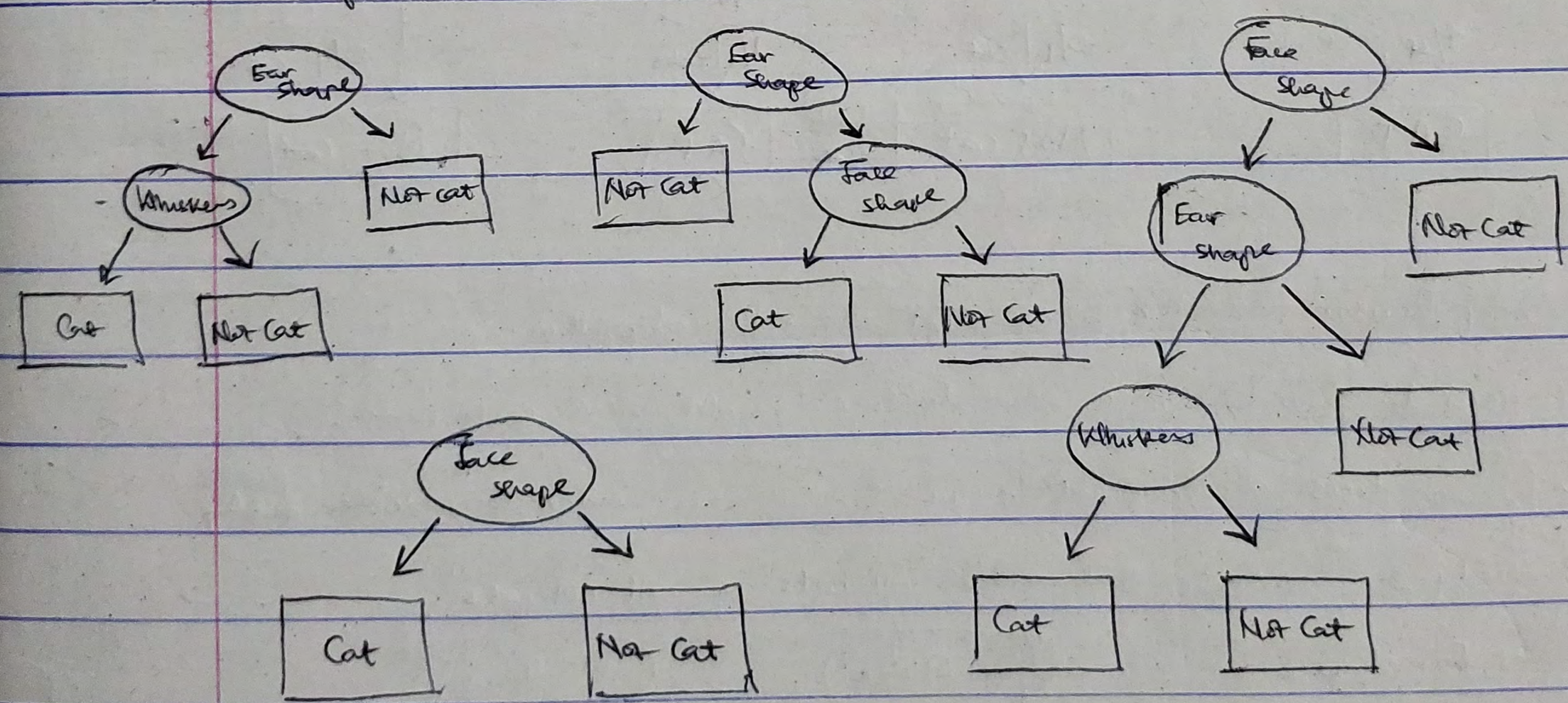
∴ you have $\vec{x}$ = ["Earshape", "Face shape", "Whiskers"] ; target y = "Cat"

Here, the features take on categorical discrete values & this is a binary classification task because the labels are either 1 or 0.

> Below is an example of the model you might get after training a Decision Tree algorithm on the data



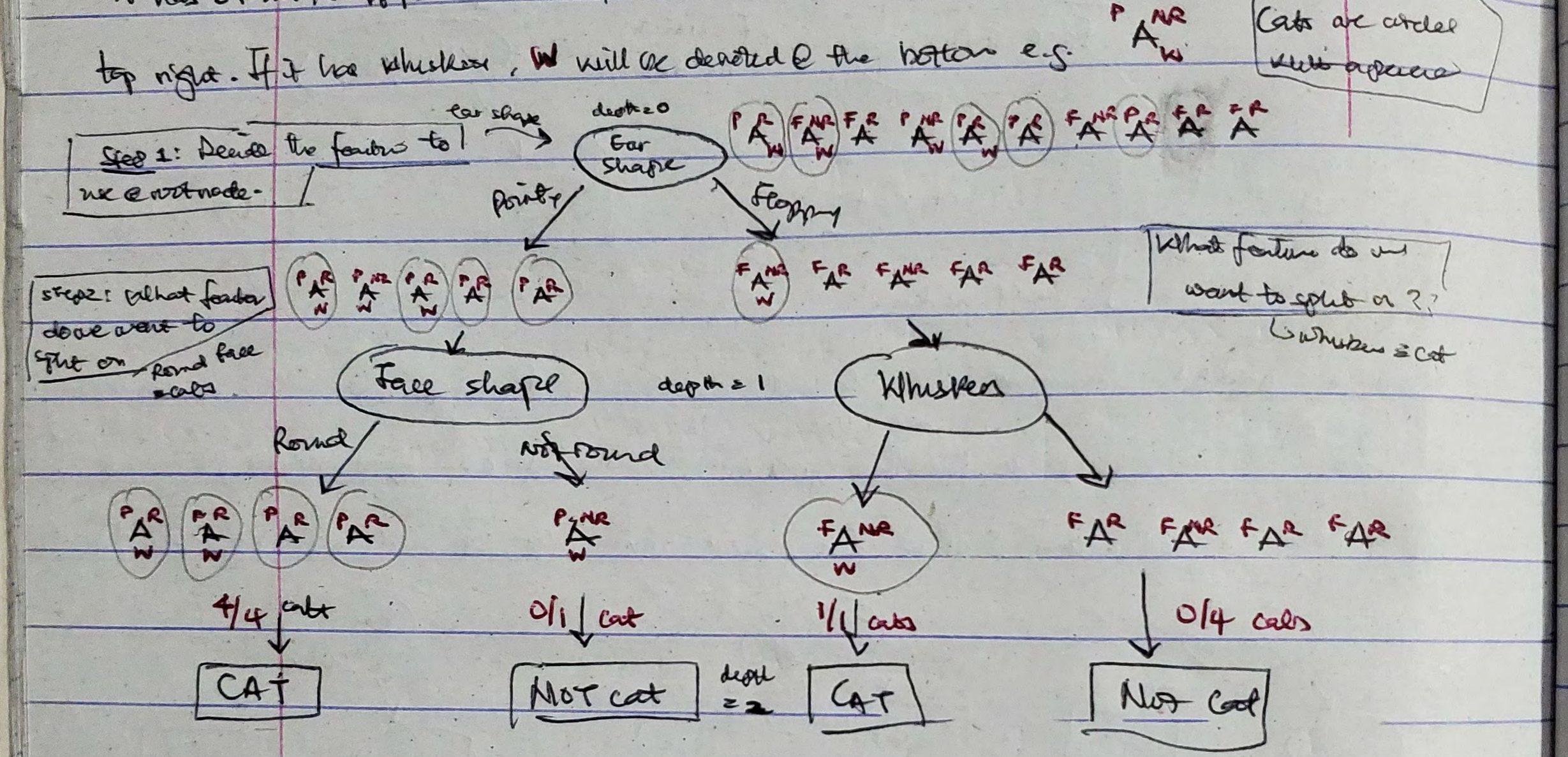
"Decision nodes" because they look @ a particular feature and based on ~~the~~ its value, it ~~does~~ decides which direction the algorithm goes down the tree. "Leaf nodes" because they make a prediction. Here are more examples of models for the D.T. algorithm



Among these decision trees, some will better or worse on the training set or on the CV & Test sets. ∴ the job of the decision tree learning algorithm is to pick one tree out of all ~~possible~~ possible ones, that ~~hopefully~~ does well on the training set and ~~also~~ also generalises well to new data

LEARNING PROCESS

- Here we'll look @ the overall steps of what needs to be done when building a decision tree. For the example; I will denote animal with A and the ~~kind~~ kind of ear it has (Pointy / Floppy) on the top left & the shape of the face it has (Round / Not Round) on the top right. If it has whiskers, W will be denoted @ the bottom e.g.



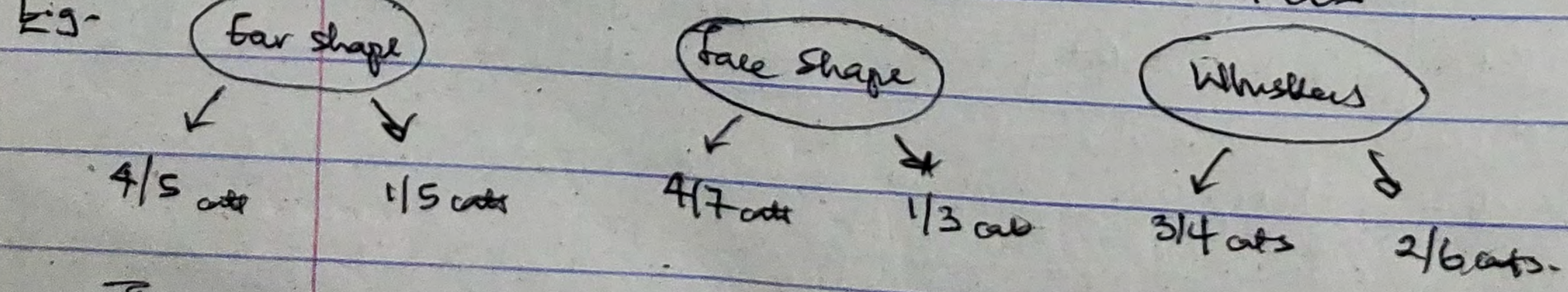
Key Decisions in the above steps during the algorithm.

(a) Decision 1: How to choose what feature to split on @ each node?

↳ Decision trees choose what feature to split on in order to maximise purity

↳ e.g. Purity = all subsets that contain all "cats" or all "not cats"

E.g.



The decision tree has to chose which of these features result in the greatest purity on the left or right subbranches

(b) <u>Decision 2</u>: When do you stop splitting?

(i) → When a node is 100% one class

(ii) → When splitting a node will result in the tree exceeding a maximum depth

depth = how many hops from root node to a particular node. 0 → 1 → 2 → 3 (root node)

This is done to make sure the tree doesn't get to big a (i) unwieldy and (ii) by keeping the tree depth small, it makes it less prone to overfitting.

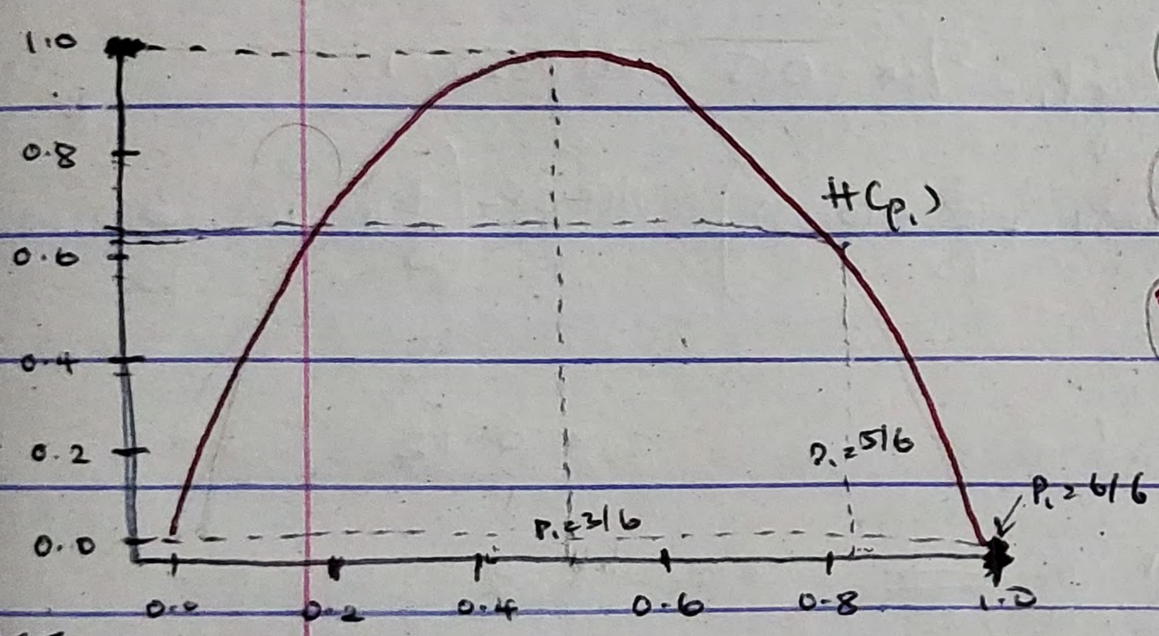
(iii) → When improvements in purity score are below a threshold

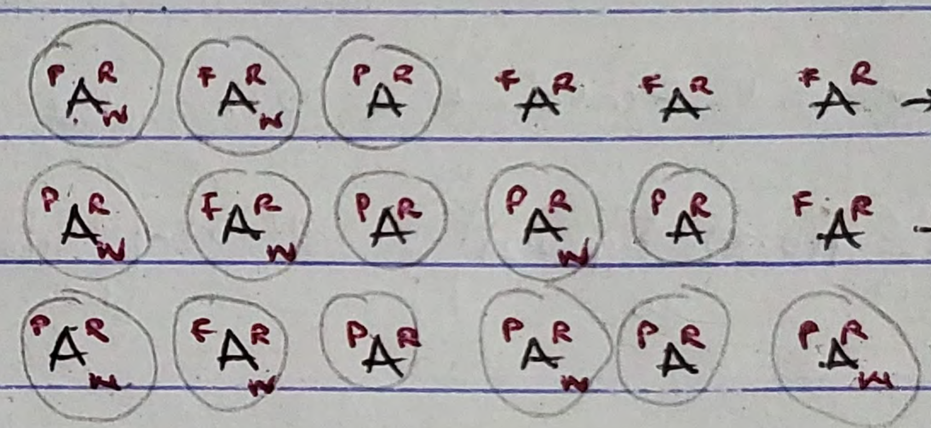
(iv) → When the number of examples in a node is below a threshold

WK 4 DECISION TREE LEARNING

- How do we quantify purity? → we use <u>ENTROPY</u> → a measure of the impurity of a set of data

Let p_1 = fraction of examples that are cats





$p_1 = 0$; $H(p_1) = 0$

$p_1 = 2/6$, $H(p_1) = 0.92$

- We'll measure the impurity of a set of examples using a function called entropy ($H(p_1)$) (H(·))

→ We see that purity increase as we go from to 50:50 mix of cats and dogs to all cats

- Let's define p_0 as the fraction of examples that are not cats; $p_0 = 1 - p_1$

$$\therefore H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0)$$

[By convention, when computing entropy, we take the log to base 2 rather than e]

$$= -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

* We take log base 2 just to make the peak of the curve be equal to one (1)

* Note "$0 \log(0)$" = 0 instead of undefined

<u>CHOOSING A SPLIT: INFORMATION GAIN</u>

- This is based on what choice of feature reduces ~~entropy~~ the most, or maximizes purity

→ the reduction of entropy in decision tree learning = <u>INFORMATION GAIN</u>

- Now lets calculate the entropy for the features that we can split on

(A) Ear shape

Pointy: $P_1 = 4/5 = 0.8$, $H(0.8) = 0.72$

Floppy: $P_1 = 1/5 = 0.2$, $H(0.2) = 0.72$

(B) Face shape

Round: $P_1 = 4/7 = 0.57$, $H(0.57) = 0.99$

Not Round: $P_1 = 1/3 = 0.33$, $H(0.33) = 0.92$

(C) Whiskers

Present: $P_1 = 3/4 = 0.75$, $H(0.75) = 0.81$

Absent: $P_1 = 2/6 = 0.33$, $H(0.33) = 0.92$

- Due to the varying examples in each branch, it wouldn't be wise to judge based on the entropy values alone. One would need to take the weighted average of all.

- Sticking to the convention in decision tree building, we'll compute the reduction in entropy (from the root node) compared to if we hadn't split at all. Recall, at the root node, we had 5 cats out of 10 examples; $P_1 = 5/10 = 0.5$ | $H(0.5) = 1$

$H(0.5) - \left(\frac{5}{10} H(0.8) + \frac{5}{10} H(0.2)\right) = 0.28$

$H(0.5) - \left(\frac{7}{10} H(0.57) + \frac{3}{10} H(0.33)\right) = 0.03$

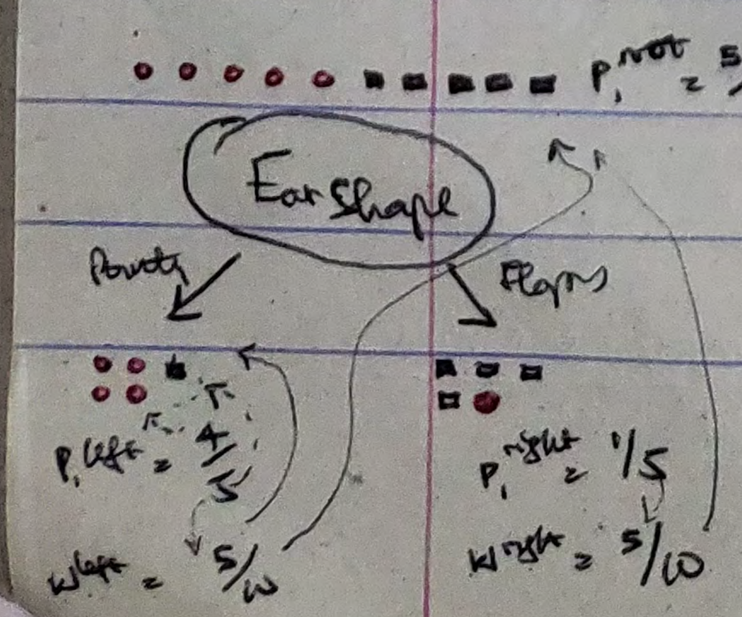
$H(0.5) - \left(\frac{4}{10} H(0.75) + \frac{6}{10} H(0.33)\right) = 0.12$

These numbers are **INFORMATION GAIN** and it measures the reduction in entropy that you get in your tree resulting in making a split. From entropy 1, you split and get a value of 0.72 and the difference of those two values is 0.28 <weighted average entropy @ left & right subbranches>

* Why do we compute "Reduction in Entropy" instead of just "Entropy"?

└ This is because one of the stopping criteria for splitting is when the reduction in entropy is too small below a threshold. In this case, you're just increasing the size of the tree unnecessarily and risking overfitting.

- Lets look @ Information Gain:



$$= H(P_1^{root}) - \left(w^{left} H(P_1^{left}) + w^{right} H(P_1^{right})\right)$$

Here is the overall process of building a decision tree

i) Start w/all examples at the root node

ii) Calculate information gain for all possible features and pick the one w/the highest information gain

iii) Split dataset according to select feature and create left and right branches of the tree.

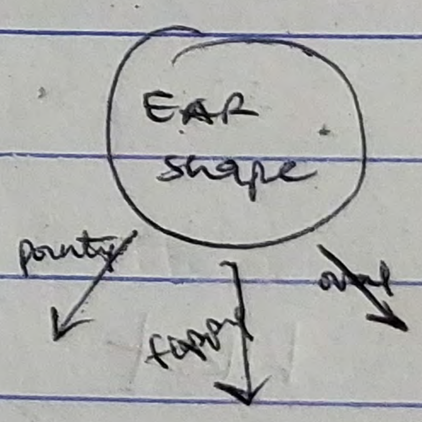
iv) Keep repeating splitting process until stopping criteria is met:

- When a node is 100% one class
- When splitting a node will result in the tree exceeding a max. depth.
- Information gain from additional splits is less than threshold
- When no. of examples in a node is below a threshold.

USE OF NEURAL ONE-HOT ENCODING OF CATEGORICAL VALUES

There's our new data;

Ear shape (x_1)	Face Shape (x_2)	Whiskers (x_3)	Cat (y)
pointy	Round	Present	1
oval	Not round	Present	1
oval	Round	Absent	0
pointy	Round	Present	0
floppy	Not round	Present	1
pointy	Not round	Absent	0
floppy	Round	Absent	1
...	...	...	...



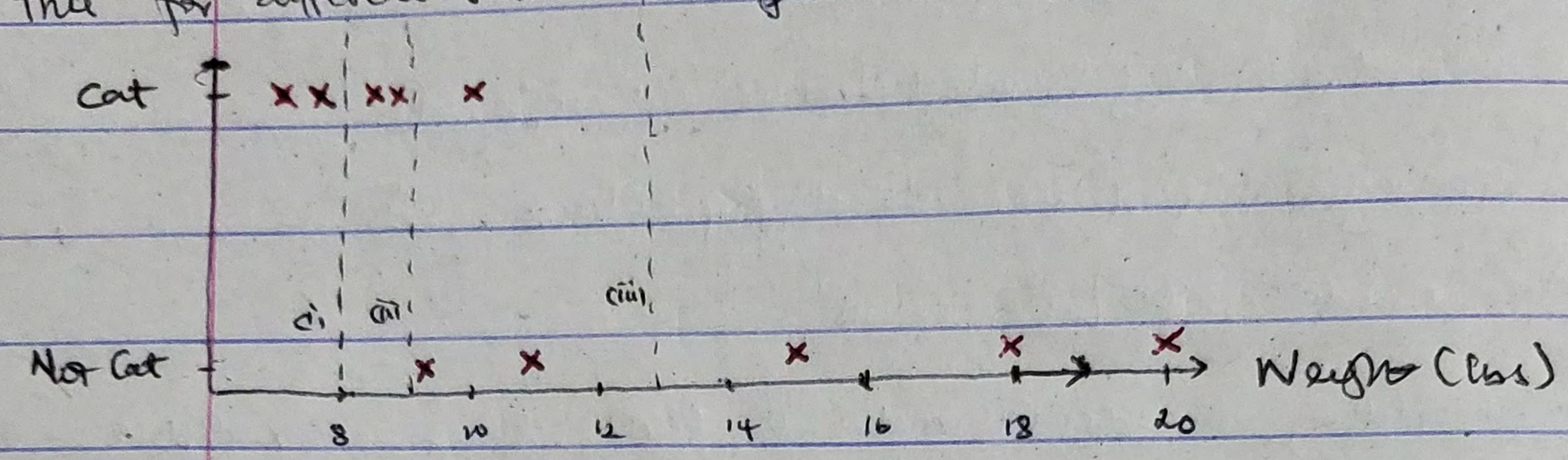
We can apply one-hot encoding to the x_1 feature. For one hot encoding, if a categorical feature can have on K values, create K binary features (0 or 1 valued). Therefore (x_1) will be split into 3 new columns

~~Ear shape~~	Pointy ears	Floppy ears	Oval ears	...
~~pointy~~	1	0	0	...
~~oval~~	0	0	1	...
~~oval~~	0	0	1	...
~~pointy~~	1	0	0	...
~~floppy~~	0	1	0	...
~~pointy~~	⋮	⋮	⋮	...

- One hot encoding also works when applying to neural network training & logistic / linear regression training

CONTINUOUS FEATURES

* Split data based on threshold & calculate the information gain after split
  - Do this for different values and go w/ the highest info gain



(i) $H(0.5) - \left(\frac{2}{10} H\left(\frac{2}{2}\right) + \frac{8}{10} H\left(\frac{3}{8}\right)\right) = 0.24$ <weight ≤ 8 lbs>

(ii) $H(0.5) - \left(\frac{4}{10} H\left(\frac{4}{4}\right) + \frac{6}{10} H\left(\frac{1}{6}\right)\right) = 0.61$ ✓ <weight ≤ 9 lbs>

(iii) $H(0.5) - \left(\frac{7}{10} H\left(\frac{5}{7}\right) + \frac{3}{10} H\left(\frac{0}{3}\right)\right) = 0.40$ <weight ≤ 13 lbs>

Recommendation: Lets say for a continuous valued feature (such as weight of the animals) on 10 animals, you'd like to split on that feature. Here's how you go about it,

- Choose the 9 midpoints between the 10 examples as possible splits and find the split that gives the highest information gain.

KWK 4 TREE ENSEMBLES

- One of the weaknesses of using a single decision tree is that that decision tree can be highly sensitive to small changes in the data
  - → ∴ building a lot of decision trees to curtail this problem leads to what is known as a TREE ENSEMBLE.
- We take the majority vote of all the predictions from the tree. This makes our algorithm more robust.

SAMPLING WITH REPLACEMENT

- → We use this technique to construct multiple random training sets that are all slightly different from our initial training set.

* This is a key building block for building an ensemble of trees.

RANDOM FOREST ALGORITHM

- How to generate a tree sample

Given training set of size m:

For b = 1 to B:

Use sampling w/ replacement to create a new training set of size m

Train a decision tree on the new dataset

* B is typically ~~recommended~~ recommended as 64 to 128 trees. Interestingly B doesn't hurt but @ some point, you end up w/ diminishing returns and it doesn't get much better when B > 1000. <B = 1000 just doesn't make sense>

* This specific implementation of tree ensemble is sometimes called a BAGGED DECISION TREE

- We can implement one modification to this algorithm more to change it to a Random Forest & make it work better

- Idea :-> At each node, when choosing a feature to use to split, if n features are available, pick a random subset of k < n features and allow the algorithm to only choose from that subset of features

* This is mostly used when n is relatively large. In that case; $k = \sqrt{n}$

XGBOOST <EXTREME GRADIENT BOOSTING>

* One that works even better than a Random Forest Algorithm [- It runs quickly and its usually used]

The idea: We look @ the decision trees we've trained so far and look @ what we're still not yet doing well on. Then when building the next decision tree, we focus more on the examples that we're not doing well on.

! Please Turn over to see the algorithm >

Boosted Trees Algorithm:

Given training set of size m:

For b = 1 to B:

- Use sampling w/replacement to create a new training set of size m <from the second one upwards> But instead of picking from all examples with equal (1/m) probability, make it more likely to pick misclassified examples from previously trained trees.
- Train a decision tree on the new dataset

Benefits:

(i) Open source implementation of boosted trees

(ii) Fast efficient implementation

(iii) Good choice of default splitting criteria & criteria for when to stop splitting

(iv) Built in regularization to prevent overfitting

(v) Highly competitive algorithm for ML competitions (e.g. Kaggle comp)

NOTE: Rather than doing sampling w/replacement, XGBoost assigns different weights to diff. training examples. And since it doesn't need to generate a lot of randomly chosen training sets, this makes it even more efficient than sampling w/replacement. But the intuition is still correct.

CLASSIFICATION

```
from xgboost import XGBClassifier
model = XGBClassifier()
model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)
```

REGRESSION

```
from xgboost import XGBRegressor
model = XGBRegressor()
model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)
```

WHEN TO USE DECISION TREES

- Decision Trees & Tree Ensembles
  - Works well on tabular (structured) data
  - Not recommended for unstructured data (images, audio, text)
  - Fast to train
  - Small decision trees may be human ~~interp~~ interpretable [you can look @ small trees to know how it works]

  <It's preferable to use XGBOOST>

- NEURAL NETWORKS
  - Works well on all types of data, including tabular (structured) and unstructured data
  - May be slower than a decision tree
  - Works with transfer learning
  - When building a system of multiple models working together, it might be easier to string together multiple neural networks.

- Check out GridSearch CV's documentation for hyperparameter tuning