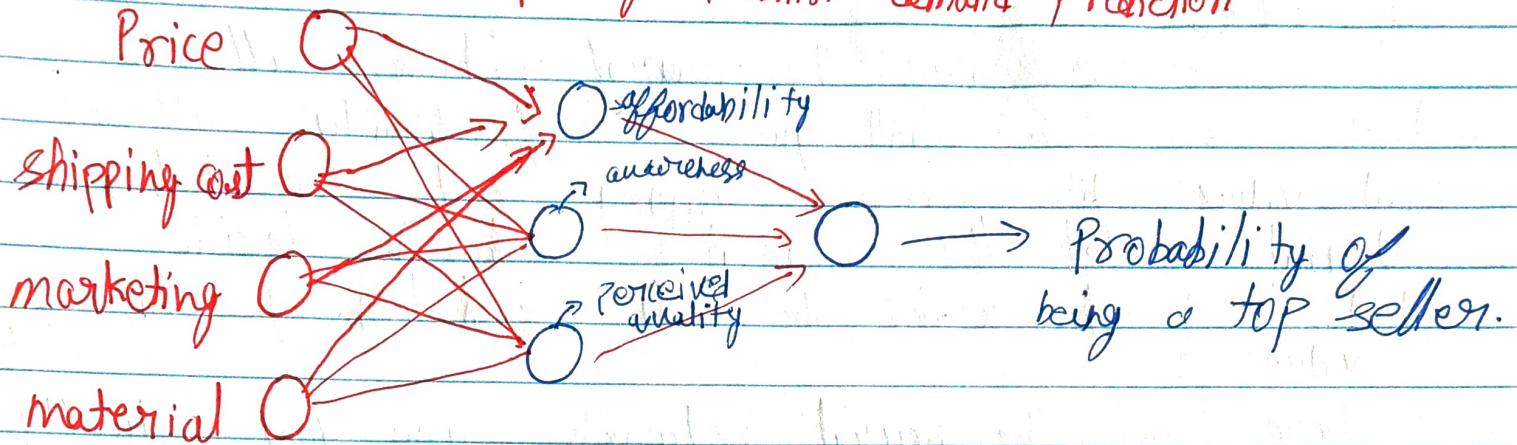


Neural Networks

"Let's take an example of T-shirt demand prediction"



So, in the neural net's example above we try to feed it with some input values & it gives some output values,

then the second layer computes ^{It's} ~~3~~ ⁴ new numbers & 3rd layer perceptron gives you the final probability of the t-shirt being top-seller.

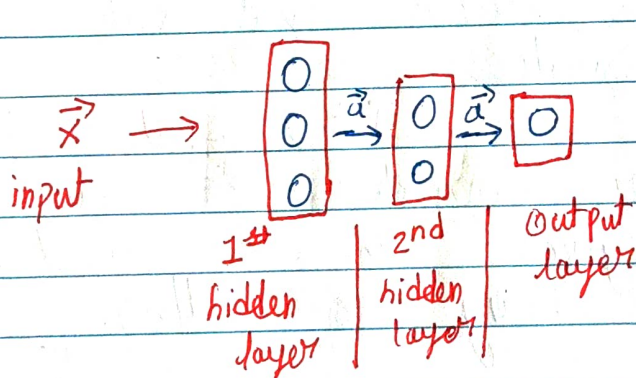
The layer in the middle acts as an activation layer for the last layer perceptron. all the perceptron in the second layer are called "activations" for the next neuron or next layer neuron.

Let's say we have, the ~~four~~ input factors ~~are~~, Price, shipping cost, marketing & material & then the hidden layer neurons are, affordability, awareness & perceived equality, now we can say that affordability is the product of price & shipping cost

& awareness is the product of marketing but if we have a lot's of input variables then we won't be able to do this stuff manually so insted,

we give the neural network all the values ex $\Rightarrow$ we will give "affordability" all the input values like price, shipping cost, marketing & material & let the neural network decide what to choose for the best prediction by setting parameters.

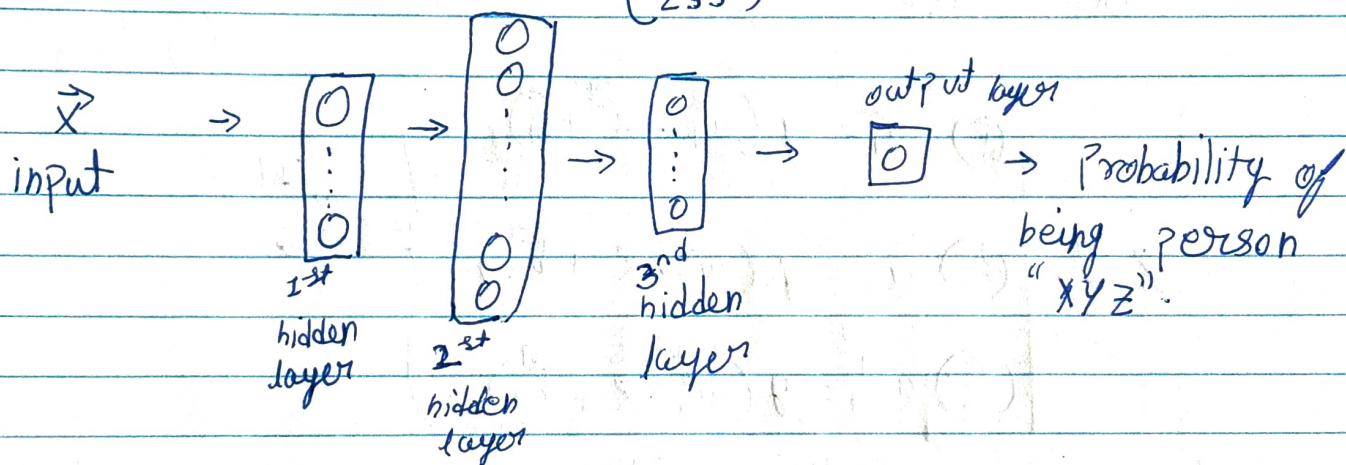
$\Rightarrow$ Here in the neural network you don't actually need to manually engineer the features. it can learn to make it's own features



Now, you need to decide how many hidden layers you want & how many perceptrons in those hidden layers.

Neural Networks for computer vision application.

Let's assume if you have a 1000×1000 pixel black & white image then the input will be ~~in~~ 1 million data points each data point will denote the brightness of the pixel ranging from (0-255).
if a pixel is completely Black then it's ~~("00")~~ ("0")
& if a pixel is ~~("00")~~ then it's white ("255")



1st hidden layer $\rightarrow$ In this layer you notice that the model is focusing on the corners & edges of the image

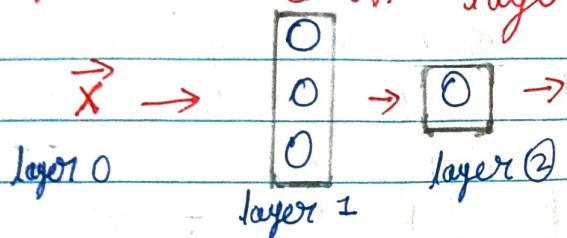
2nd hidden layer $\rightarrow$ In this layer you notice that the model is identifying the face parts like $\therefore$ nose, ears, eyes & eyebrows.

3rd hidden layer $\rightarrow$ In this layer you notice that the model is trying to add parts of the face & detecting a larger, coarser face shapes.

$\Rightarrow$ and all of this is done by the neural network no one tells it to detect edges & stuff.

Neural Networks beyond intuition.

Neural Network layer $\Rightarrow$



So, for the hidden layer the computation goes like,

$$\vec{x} \rightarrow \left\{ \begin{array}{l} \textcircled{\begin{smallmatrix} w_1 \\ b_1 \end{smallmatrix}} \quad a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]}) \\ \textcircled{\begin{smallmatrix} w_2 \\ b_2 \end{smallmatrix}} \quad a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]}) \\ \textcircled{\begin{smallmatrix} w_3 \\ b_3 \end{smallmatrix}} \quad a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]}) \end{array} \right\} \rightarrow \vec{a}^{[1]}$$

Here, the activations weight's & biases are applied with a superscript to denote the layer number

ex $\Rightarrow a_1^{[1]}$ Here $[1] \rightarrow$ is a superscript

$$\vec{a}^{[1]} \rightarrow \left\{ \begin{array}{l} \textcircled{\begin{smallmatrix} w_1 \\ b_1 \end{smallmatrix}} \quad a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}) \end{array} \right\} \rightarrow a^{[2]}$$

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

Once the neural network has completed $a^{[2]}$ let's say it's "0.84", if $a^{[2]} \geq 0.5$?

Yes $\swarrow$ $\hat{y} = 1$ No $\searrow$ $\hat{y} = 0$

In general, you can write,

Activation value of layer l , unit (neuron) j

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

Here, $\vec{w}_j^{[l]}$ & $b_j^{[l]}$ are parameters
w & b of layer l , unit j . & g is an activation
function.

Inference: Making predictions (forward propagation)

⇒ Same shit bruv you give input from
input layers it gets g through
hidden layers generally the size of
hidden layers decrease. & yeah let's say
~~whether~~ you want to detect if a digit
is 1 or 0 then last output layer will
give you the probability of digit being
zero or 1.

you can apply a condition too.
like if,

prediction ≥ 0.5 ?

yes

no

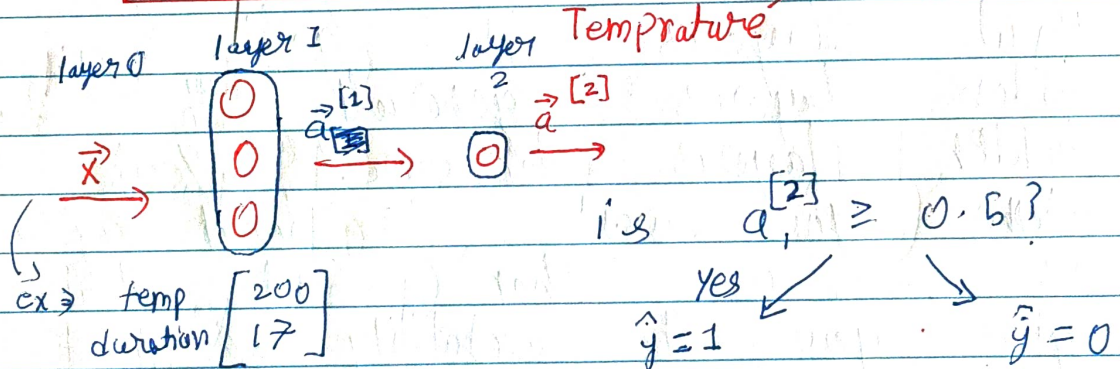
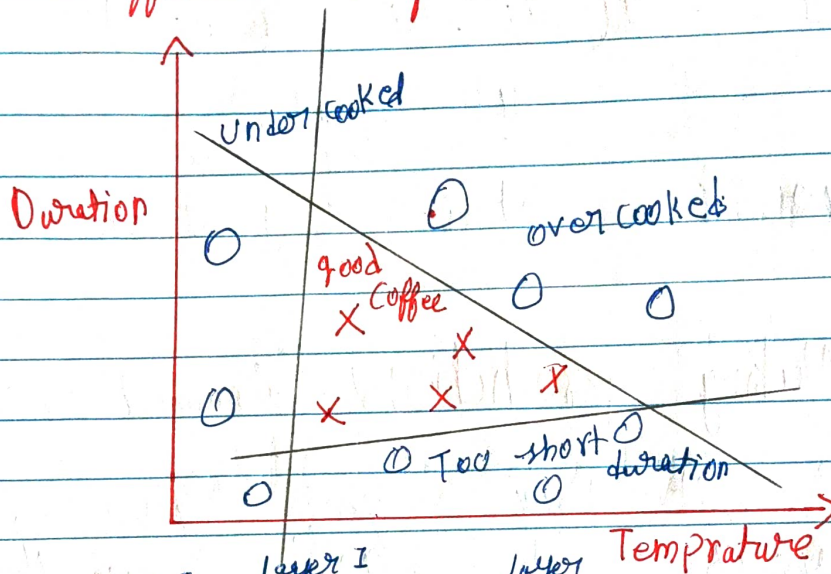
$\hat{y} = 1$

$\hat{y} = 0$

That's it. go next

Inference in code (using tensorflow)

ex $\Rightarrow$ Coffee roasting



Here are the steps in code to do it.

```
x = np.array([[200.0, 17.0]])
```

```
layer_1 = Dense(units=3, activation='sigmoid')  
a1 = layer_1(x)
```

```
layer_2 = Dense(units=1, activation='sigmoid')  
a2 = layer_2(a1) # This will give you probability  
                  whether the coffee brewed  
                  is good or not.
```

Data in tensorflow

You will study about the tensor data type which is used by tensorflow later,

In numpy,

```
x = np.array([[200, 17]])
```

Here, it's a 2d array or matrix with 1 row & 2 columns.

```
x = np.array([[200],  
               [17]])
```

Here, it's a 2d array with 2 rows & 1 column.

```
x = np.array([200, 17])
```

Here, it's a "1d" array or simply a vector.

Tensorflow implementation

Building a neural network.

example :-

- ① `layer-1 = Dense(units=25, activation="sigmoid")`
- ② `layer-2 = Dense(units=15, activation="sigmoid")`
- ③ `layer-3 = Dense(units=1, activation="sigmoid")`
- ④ `model = Sequential([layer-1, layer-2, layer-3])`
- ⑤ `model.compile(...)`
- ⑥ `x = np.array([[0..., 245, ..., 17],
[0..., 200, ..., 184]])`
- ⑦ `y = np.array([1, 0])`
- ⑧ `model.fit(x, y)`
- ⑨ `model.predict(x_new)`

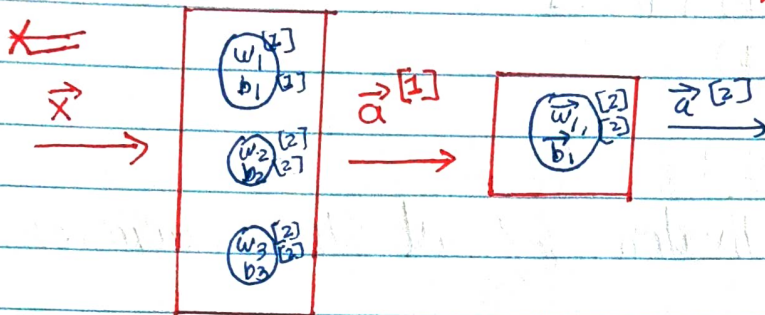
Here, in line ① we create a layer of neural network with 25 perceptrons having sigmoid activation fn.

in line ④ we make or join all the layers. Or string all the layers, "use of the 'sequential fn'".

in line ⑤ there is this compile fn which you will study about later.

In ⑧ line we fit the data & prediction for that data.
 In ⑨ line we generate inference from that data.

[forward prop (coffee roasting model example)]



$x = \text{np.array}([200, 17])$

$$a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]})$$

$w_{1-1} = \text{np.array}([1, 2])$

$b_{1-1} = \text{np.array}([-1])$

$z_{1-1} = \text{np.dot}(w_{1-1}, x) + b_{1-1}$

$a_{1-1} = \text{sigmoid}(z_{1-1})$

$w_{1-2} = \text{np.array}([-3, 4])$

$b_{1-2} = \text{np.array}([1])$

$z_{1-2} = \text{np.dot}(w_{1-2}, x) + b_{1-2}$

$a_{1-2} = \text{sigmoid}(z_{1-2})$

$w_{1-3} = \text{np.array}([5, -6])$

$b_{1-3} = \text{np.array}([2])$

$z_{1-3} = \text{np.dot}(w_{1-3}, x) + b_{1-3}$

$a_{1-3} = \text{sigmoid}(z_{1-3})$

$a_1 = \text{np.array}([a_{1-1}, a_{1-2}, a_{1-3}])$

$$a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]})$$

$$w_{2-1} = \text{np.array}([-7, 8, 9])$$

$$b_{2-1} = \text{np.array}([3])$$

$$z_{2-1} = \text{np.dot}(w_{2-1}, a_1) + b_{2-1}$$

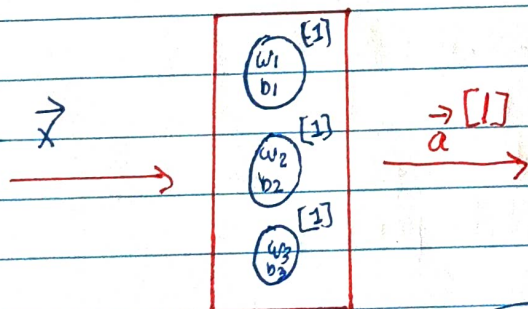
$$a_{2-1} = \text{Sigmoid}(z_{2-1})$$

in the code above,

a_1 is the activation of all the neurons in layer 1.

a_{2-1} is the output layer.

Forward prop more general implementation in python



$$\vec{w}_1^{[1]} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \vec{w}_2^{[1]} = \begin{bmatrix} -3 \\ 4 \end{bmatrix}, \quad \vec{w}_3^{[1]} = \begin{bmatrix} 5 \\ -6 \end{bmatrix}$$

$$b_1^{[1]} = -1, \quad b_2^{[1]} = 1, \quad b_3^{[1]} = 2$$

$$\textcircled{1} \quad w = \text{np.array}(\begin{bmatrix} 1, -3, 5 \\ 2, 4, -6 \end{bmatrix})$$

② $b = \text{np.array}([-1, 1, 2])$

$$\vec{a}^{[0]} = \vec{x}$$

③ $a_in = \text{np.array}([-2, 4])$

④ $\text{def dense}(a_in, w, b):$

⑤ $units = w.shape[1]$

⑥ $a_out = \text{np.zeros}(units)$

⑦ $\text{for } j \text{ in range}(units):$

⑧ $w = w[:, j]$

⑨ $z = \text{np.dot}(w, a_in) + b[j]$

⑩ $a_out[j] = g(z)$

⑪ $\text{return } a_out$

Note: $g()$ is defined outside of $\text{dense}()$ - (see optional lab).

⑫ $\text{def sequential}(x):$

⑬ $a_1 = \text{dense}(x, w_1, b_1)$

⑭ $a_2 = \text{dense}(a_1, w_2, b_2)$

⑮ $a_3 = \text{dense}(a_2, w_3, b_3)$

⑯ $a_4 = \text{dense}(a_3, w_4, b_4)$

⑰ $f_x = a_4$

⑱ $\text{return } f_x$

here Capital W refers to a matrix of weights

Code explanation:-

In line ① we have an 2d array (2x3) which stores all the weights & biases of the layer,

The fn `dense` takes the creates an empty array which is equal to the size shape of the columns of weights array,

In the for loop of this fn we take the weights of a single perceptron & does dot product & after the use of fn "`g()`" we get the activation from that neuron & populate the activation array of the particular layer.

The fn `sequential` calls the fn `dense` four times which means we are inferring from a neural net which has 3 hidden layers & 1 output layer.

⇒ Same code using vectorization technique.

```
X = np.array([[200, 17]])
```

```
W = np.array([[1, -3, 5],  
              [-2, 4, -6]])
```

```
B = np.array([[-1, 1, 2]])
```

```
def dense(A_in, W, B):
```

```
    z = np.matmul(A_in, W) + B
```

```
    A_out = g(z)
```

```
    return A_out
```

⇒ In the code above the fn "ne-matmul" does matrix multiplication.
& g() is a sigmoid fn.

Week 2 :- Advanced learning Algorithms :- Neural Network training

Training a model in tensorflow :- Code :-

```
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'), ])
```

```
from tensorflow.keras.losses import BinaryCrossentropy
```

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

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

explanation of the code above :-

⇒ in the code above we create an object called model with 3 Dense layers & all having sigmoid activation fn.

⇒ after that we compile the model with the use of BinaryCrossentropy's loss which is used in Binary classification problems.

⇒ The fn model.fit performs gradient descent on the neural network taking the input X, Y & epochs.

⇒ (model training steps)

① Specify how to compute output given input x & parameters w, b (define model).

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

Neural network ⇒

model = Sequential ([
Dense (....)
Dense (....)
Dense (....)
])

② Specify loss & cost

$$L(f_{\vec{w}, b}(\vec{x}), y) \quad 1 \text{ example}$$

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

Neural networks ⇒

model.compile (

loss = Binary Cross entropy (1))

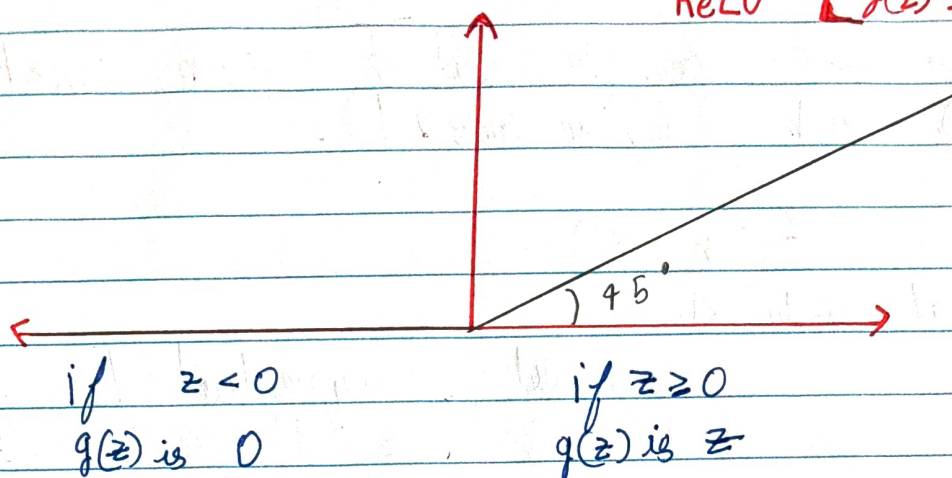
③ Train on data to

minimize $J(\vec{w}, b)$

Neural network ⇒

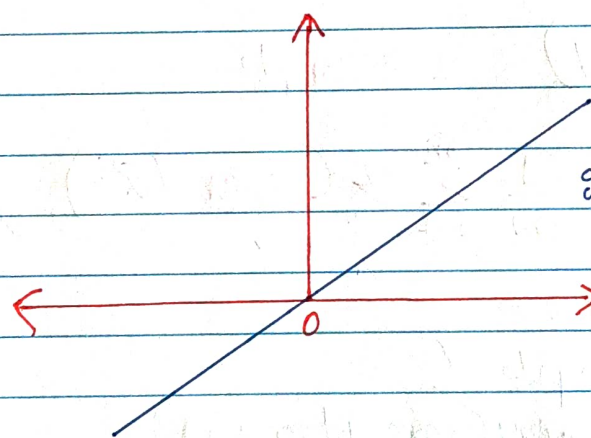
model.fit (x, y, epochs = 100)

Alternatives to the sigmoid activation function.
ReLU $[g(z) = \max(0, z)]$



ReLU $\rightarrow$ Rectified linear unit

$\Rightarrow$ Linear activation fn



$$g(z) = z$$

whatever the input is it will be the output, it's seen in Linear regression

Choosing activation fn.

For the output layers $\therefore$

for binary classification problems
 $\rightarrow$ go for sigmoid fn

for regression problems
 $\rightarrow$ go for linear fn

for regression when the output has to be strictly greater than 0 ($y > 0$).

$\rightarrow$ go for ReLU fn

for hidden layers generally we use ReLU fn.

Gradient descent

work faster on ReLU than Sigmoid.

code $\Rightarrow$

```
from tf.keras.layers import Dense  
model = Sequential ([
```

```
#layer 1 Dense(units=25, activation = 'relu'),
```

```
#layer 2 Dense(units=15, activation = 'relu'),
```

```
#layer 3 Dense(units=1, activation = 'sigmoid')  
])
```

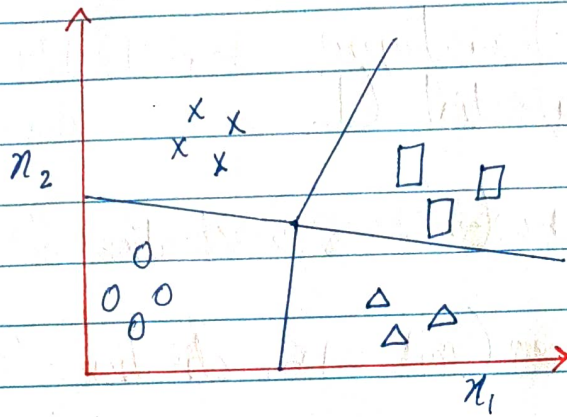
Why do we use activation fn's ?

$\Rightarrow$ So, it turns out that if you don't use any activation fn the model "Neural Network" will just be another linear regression model, it can't fit non-linear data that's why we use activation fn's, like if we just have a sigmoid fn at the output, it will just be an logistic regression model,

that is why if you want it to fit complex data use activation fn's like - ReLU, LeakyReLU.

Multi class :- USING NEURAL NETWORK.

It is the type of classification in which you have more than one two classes to fit the output variable in.



Softmax Regression :-

Let's say we have 4 possible outcomes :-

$$z_1 = \vec{w}_1 \cdot \vec{x} + b_1 \quad a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=1|\vec{x})$$

$$z_2 = \vec{w}_2 \cdot \vec{x} + b_2 \quad a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=2|\vec{x})$$

$$z_3 = \vec{w}_3 \cdot \vec{x} + b_3 \quad a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=3|\vec{x})$$

$$z_4 = \vec{w}_4 \cdot \vec{x} + b_4 \quad a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=4|\vec{x})$$

General case :-

$$z_j = \vec{w}_j \cdot \vec{x} + b_j, \quad j = 1, \dots, N$$

Parameters $w_1, w_2, \dots, w_N$
 $b_1, b_2, \dots, b_N$

$$a_j = \frac{e^{z_j}}{\sum_{k=1}^N e^{z_k}} = P(y=j|\vec{x})$$

note: - $a_1 + a_2 + a_3 + \dots + a_N = 1$

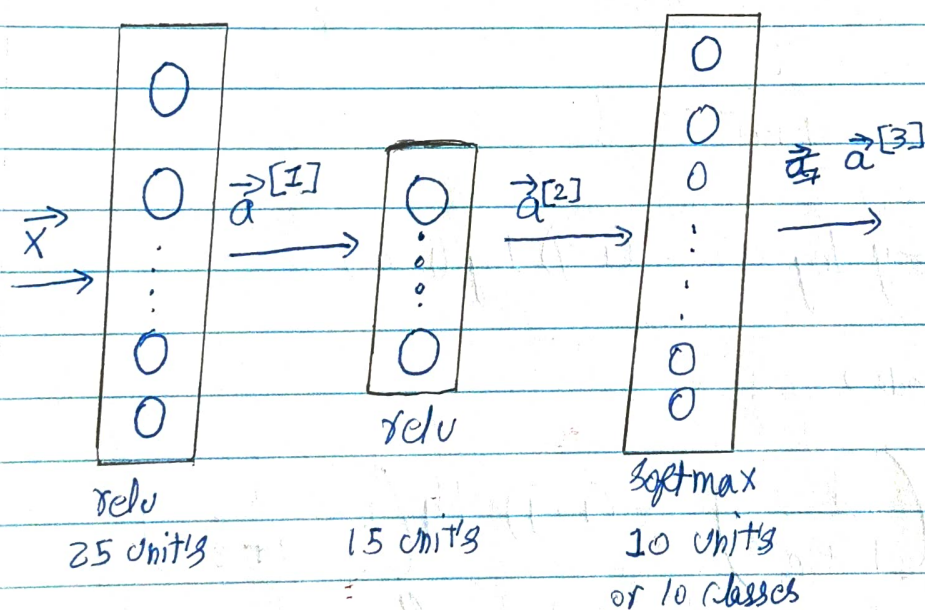
⇒ Cost function for Softmax regression

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

ex →

$$\text{loss} = -\log a_j, \text{ if } y = j$$

Neural Network with Softmax output :-



$$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]}$$

⋮

$$z_{10}^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]}$$

$$a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \dots + e^{z_{10}^{[3]}}} = P(y=1 | \vec{x})$$

$$a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{e^{z_1^{[3]}} + \dots + e^{z_{10}^{[3]}}} = P(y=10 | \vec{x})$$

The difference btw softmax activation & other activations like ReLU, sigmoid & linear is that their activation a_1 only depends upon z_1 & a_2 only depends upon z_2 & so on, but in softmax activation, a_1 will depend upon $z_1, z_2, \dots, z_N$ & that goes for every activation in the output layer or softmax layer.

Implementation of softmax $\therefore$ Code

Let's take a look at more numerically accurate implementation of logistic loss:

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

original loss $\therefore$

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

more accurate loss (in code)

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

if you expand the eqn, Then TensorFlow can rearrange terms in this expression & come up with more numerically accurate way.

Code -:

```
//model    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= 'Linear') ])
```

```
// loss    from tensorflow.keras.layers import
           SparseCategoricalCrossentropy.
```

```
model.compile(..., loss = SparseCategoricalCrossentropy (from-
logits = True))
```

```
// Fit     model.fit(x, y, epochs=100)
```

```
// Predict .    logits = model(x)
```

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


Explanation of the code

So, in the code, what's peculiar is that we have used a linear output in the activation in the third dense layer. This is because we don't want the approximation to give us slightly less accurate prediction.

In `model.compile(...)` we are using `from_logits=True`. We are telling the model, please apply softmax inside the loss fn automatically, because logits are the raw output of the neural net's without any activation fn.

In `logits = model(x)`, we are giving an input `x` (To generate inference), & we know that the model outputs raw logits. So we use `tf.nn.softmax(...)` to convert it into probabilities.

→ Multi-label classification :-

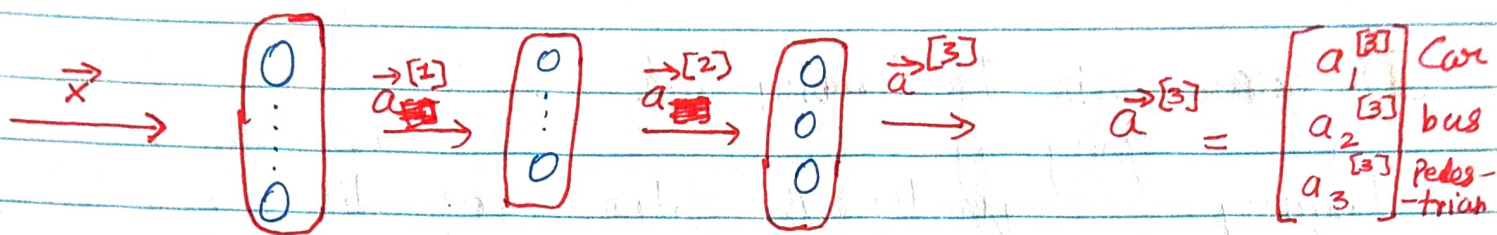
In this type of classification you can have multiple labels to a single image.

Let's say you are building a machine learning model to tell you how to what are the things in the image, like for a self-driving car's,

	image 1	image 2	image 3
Is there a car?	yes $\begin{bmatrix} 1 \end{bmatrix}$	no $\begin{bmatrix} 0 \end{bmatrix}$	yes $\begin{bmatrix} 1 \end{bmatrix}$
Is there a bus?	no $\begin{bmatrix} 0 \end{bmatrix}$	no $\begin{bmatrix} 0 \end{bmatrix}$	yes $\begin{bmatrix} 1 \end{bmatrix}$
Is there a pedestrian?	yes $\begin{bmatrix} 1 \end{bmatrix}$	yes $\begin{bmatrix} 1 \end{bmatrix}$	no $\begin{bmatrix} 0 \end{bmatrix}$

To tackle this problem, what you can do is.

→ train one neural network with 3 outputs.



Now, in the output layer $\vec{a}^{[3]}$ we can use the sigmoid activation fn for each perceptron, which will tell us if the image has a car, a bus & a pedestrian, it will give us a vector ~~st~~ having all of these values in form of zero or one.

Advanced optimizations

→ So it turns out that there is this method called "Adam" Algorithm which helps change the learning algorithm mid gradient descent which helps in faster convergence, & Yeah

this is how it works.

→ (i) increase in learning algorithm $\therefore$ So when the convergence goes in the same direction for a specific amount of time also goes like hmm maybe i should increase the learning rate because it's taking too damn time & the direction isn't even changing so it should be right "right?"

(ii) decrease in the learning rate $\therefore$ If the cost keeps going up & down & the direction is changing frequently the algo goes like hmm maybe i should decrease the learning rate 'cause idk where the fk i'm goin'.

Adam $\therefore$ Adaptive Moment estimation

$\rightarrow$ It not just assign 1 " α " but ~~a~~ a different α for each weight

$$\text{like } \therefore w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$

$$w_n = w_n - \alpha_n \frac{\partial}{\partial w_n} J(\vec{w}, b)$$

In Code $\therefore$

model $\therefore$

```
model = Sequential([  
    tf.keras.layers.Dense(units=25, activation='sigmoid'),  
    tf.keras.layers.Dense(units=15, activation='sigmoid'),  
    tf.keras.layers.Dense(units=10, activation='linear')  
])
```

Compile $\therefore$

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

fit :-

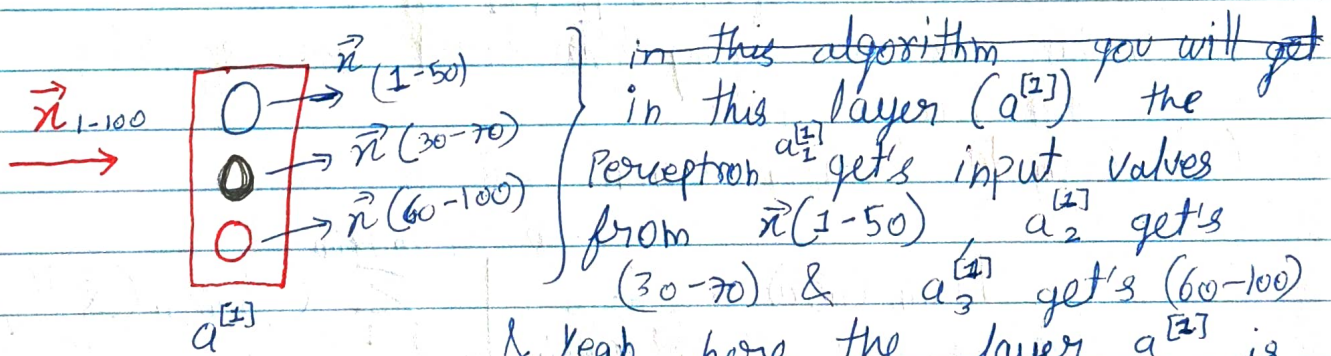
model.fit(x, y, epochs=100)

Convolutional LAYERS

So, we know about the Dense layers in which every perceptron will get the activation every activation from the previous layers,

but in **Convolutional layers**, each perceptron gets only a part of the previous activations.

it is done because it provides faster computation & needs less training data meaning less prone to overfitting.



& Yeah here the layer $a^{[1]}$ is a convolutional layer.

"A neuron gets to look at a limited window of the input"

Evaluating your model

⇒ Split your data in 70% & 30%, the major value will be training set the minor value will be test set.

Here are the steps:-

(1) fit parameters by minimizing cost fn $J(\vec{w}, b)$

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

(2) Compute test error:

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

in this we are using the test ~~exam~~ examples

(3) Compute training error:

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

So, see if the training error is ~~much higher~~ ^{smaller} than the test error, the model is likely overfitting.

In logistic Regression:— for classification problem.

calculate fraction of the test set & the fraction of train set that the algorithm has misclassified & if

$\{ \text{fraction}_{\text{test}} > \text{fraction}_{\text{train}} \}$, the model might

be overfitting

Model Selection.

Step ① Split your data into 3 parts, 1 major portion & 2 minor portions

60% → training data

20% → test data

20% → Cross validation data

Step ② → decide how many hyper parameters there should be, like, $d=1$, $\hat{y} = wx + b$
 $d=2$, $\hat{y} = wx_1 + \dots + wx_2 + b$

$\vdots$

$d=10$, $\hat{y} = wx_1 + \dots + wx_{10} + b$

To do this, use cross validation data & the model which gives the lowest cost will be the model we choose.

Step ③ → Use test data to evaluate the model

if $\text{test}_{\text{cost}} \approx \text{validation}_{\text{cost}} \rightarrow$ your model selection procedure worked well.

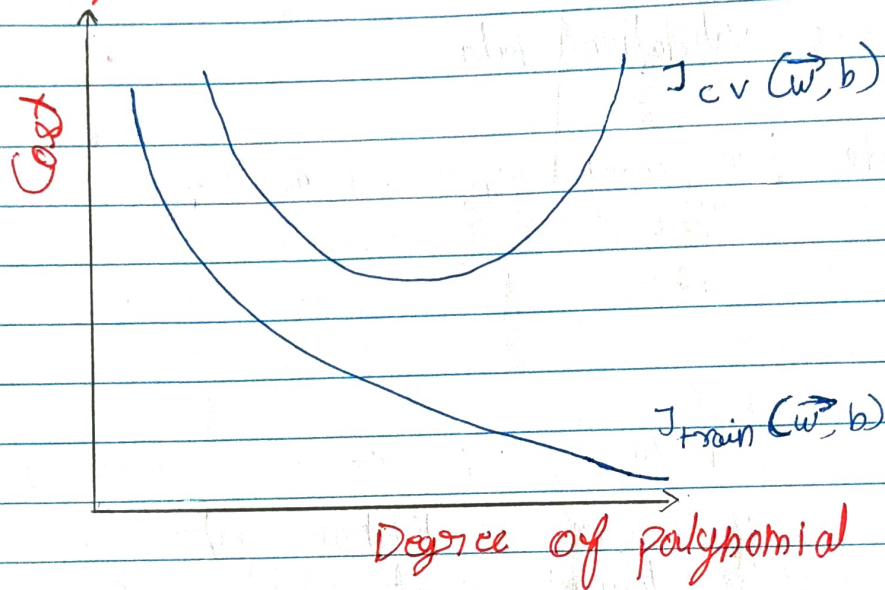
if $\text{test} \ll \text{validation} \rightarrow$ lucky/unrepresentative split.

if $\text{test} \gg \text{validation} \rightarrow$ you may have overfit your validation process.

⇒ "You can also use this method in neural networks to decide what model to choose & based on numbers of like what

is the number of dense layer you want in your model & what is the number of perceptron units per layer.

Diagnosing bias & variance.



High bias (underfit)

J_{train} will be high
($J_{\text{train}} \approx J_{\text{cv}}$)

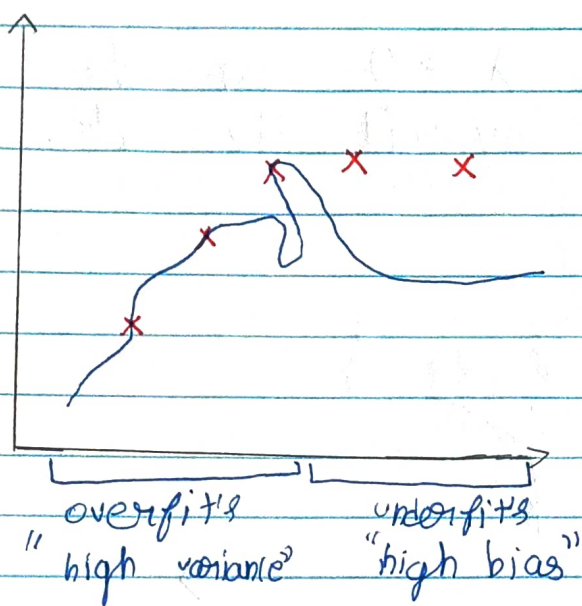
High variance (overfit)

($J_{\text{cv}} \gg J_{\text{train}}$)

High bias & high variance

$\left\{ \begin{array}{l} J_{\text{train}} \text{ will be high} \\ \& J_{\text{cv}} \gg J_{\text{train}} \end{array} \right\}$

It means that the model will be overfitting some part of the data & underfitting other part of the data.

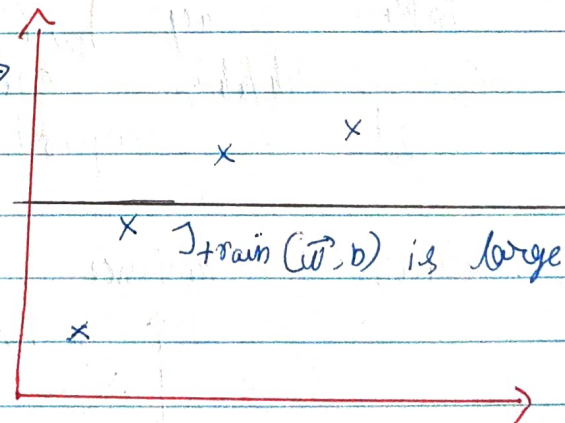


Finding the right value of " λ " using "cross validation error" in Linear regression $\therefore$

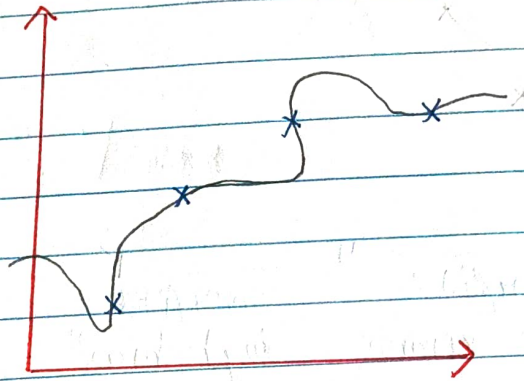
$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^m (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n w_j^2$$

if " λ " regularization parameter is too high then the weight's become too small &

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



when " λ " is too low $\lambda \approx 0$ then the weights don't change & they overfit the data.



To choose the model

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

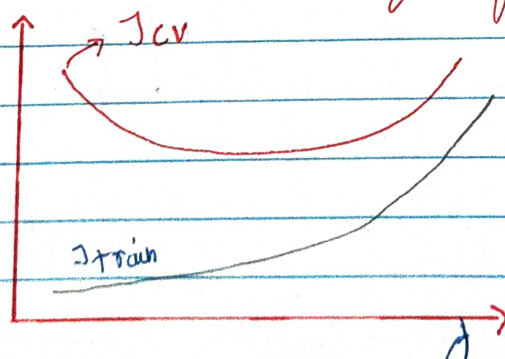
(1) Try $\lambda = 0 \rightarrow \min_{\vec{w}, b} J(\vec{w}, b) \rightarrow w^{(1)}, b^{(1)} \rightarrow J_{cv}(w^{(1)}, b^{(1)})$

(2) Try $\lambda = 0.01$

(12) Try $\lambda \approx 10$

Let's say $J_{cv}(w^{(5)}, b^{(5)})$ has the lowest cross validation error so you pick that model & report $J_{test}(w^{(5)}, b^{(5)})$.

$\rightarrow$ Bias & Variance as a fn of " λ ".



Establishing a baseline level of performance

What is the level of error you can reasonably hope to get to?

- Human level performance
- Competing algorithms performance
- Guess based on experience-

So, let's say,

Ex-1

- i Baseline Performance $\therefore 10.6\%$ $\uparrow 0.2\%$
- ii Training error (J_{train}) $\therefore 10.8\%$ $\uparrow 4.0\%$
- iii Cross validation error (J_{cv}) $\therefore 14.8\%$

Now, you have to look what is the difference between Baseline performance & Training error, ~~Baseline perfo~~ Training error & Cross validation error, ~~≡ ≡~~

Now, we see that training error & Baseline performance are almost similar which is expected but

training error & Cross validation error (J_{cv}) are $(4.0)\%$ different, so we conclude that the model has high variance.

Ex-2

- (i) BP - 10.6
- (ii) J_{train} - 15.0%
- (iii) J_{cv} - 15.5%

Here, J_{train} & J_{cv} are actually quite same but BP & J_{train} have a huge difference so,

the conclusion is the model has high bias.

Ex-3

- (i) BP $\rightarrow 10.6\%$ $\uparrow 4.4\%$
- (ii) J_{train} $\rightarrow 15.0\%$ $\uparrow 4.7\%$
- (iii) J_{cv} $\rightarrow 19.7\%$ $\downarrow$

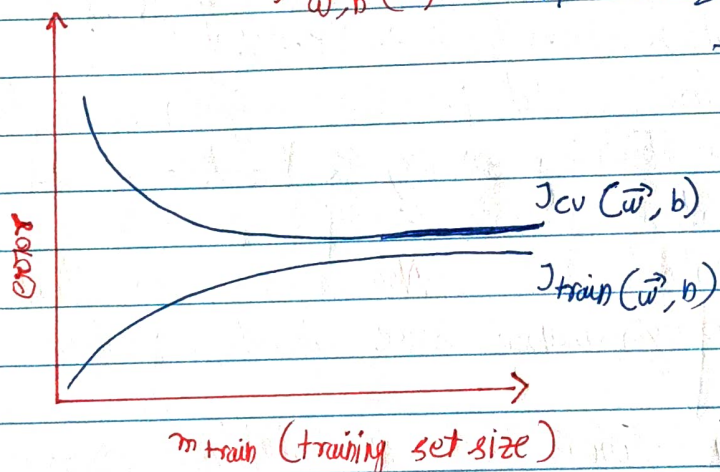
} LOL, your algo has both high bias & high Variance

Learning Curves in bias & Variance

J_{train} = training error

J_{cv} = Cross validation error

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



as training set size increases, the J_{cv} decreases.

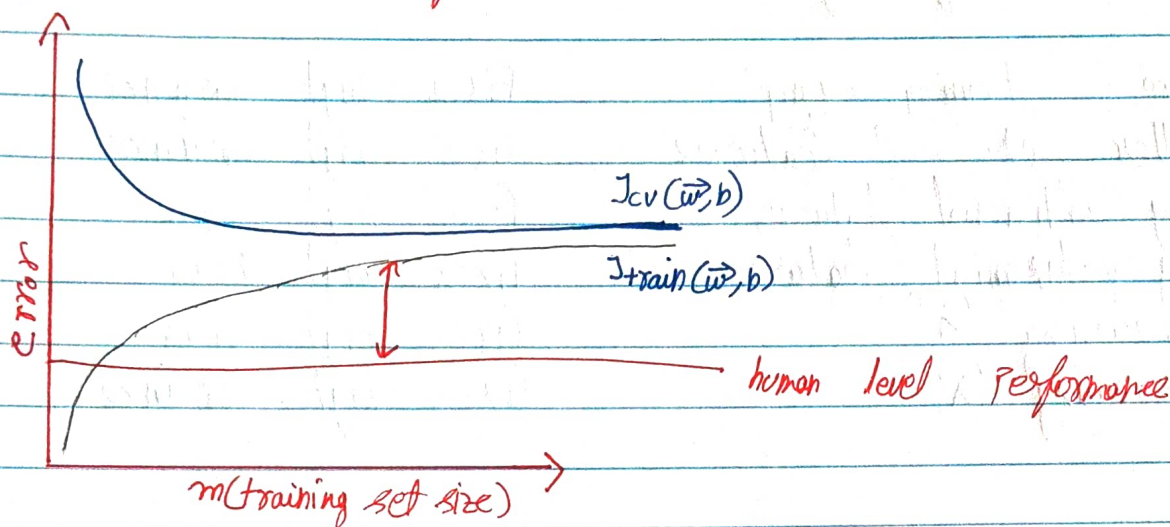
as ~~the~~ J_{train} increases,

this is because well if you train a model on lots of examples it

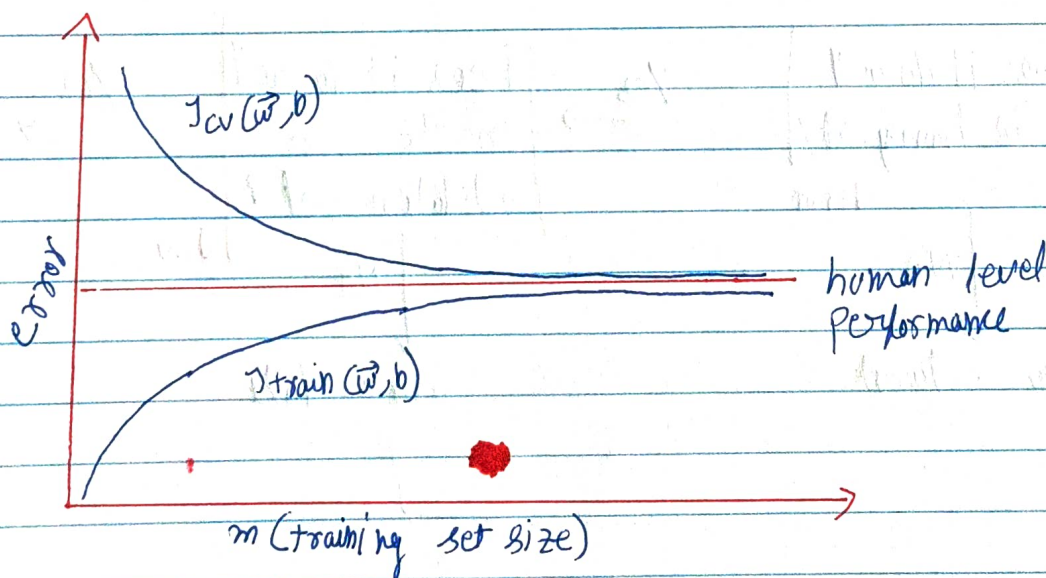
gets good, but as you

increase the training set size J_{train} ~~decreases~~ increases because, it's easy to fit in few examples perfectly but it's not possible if you increase the examples.

High bias (Learning curves)



⇒ if a learning algorithm suffers with high bias, getting more training data will not help much.



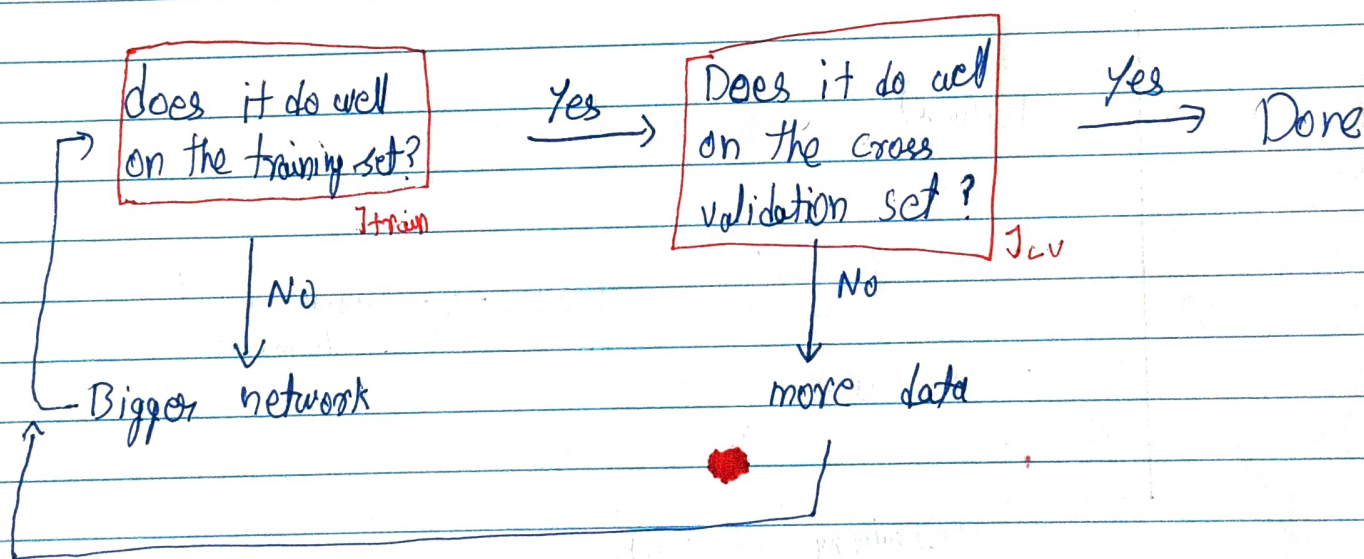
⇒ If a learning algo suffers from high variance getting more training data is likely to help.

Debugging a learning algorithm

- Get more training examples
- Try smaller set of features
- Try getting additional features
- Try adding polynomial features
- Try decreasing " λ "
- Try increasing " λ "

Fixes high variance
fixes high variance
fixes high bias
fixes high bias
fixes high bias
fixes high variance

Neural Networks & Bias, Variance



⇒ A large neural network will usually do as well or better than smaller one so long as regularization is chosen appropriately.

Neural Network regularization using tensorflow :-

→ Unregularized MNIST Model

layer-1 = Dense(units = 25, activation = 'relu')

layer-2 = Dense(units = 15, activation = 'relu')

layer-3 = Dense(units = 1, activation = 'sigmoid')

model = Sequential([layer-1, layer-2, layer-3])

→ Regularized MNIST Model

layer-1 = Dense(units = 25, activation = 'relu', kernel_regularizer = L2(0.01))

layer-2 = Dense(units = 15, activation = 'relu', kernel_regularizer = L2(0.01))

layer-3 = Dense(units = 1, activation = 'sigmoid', kernel_regularizer = L2(0.01))

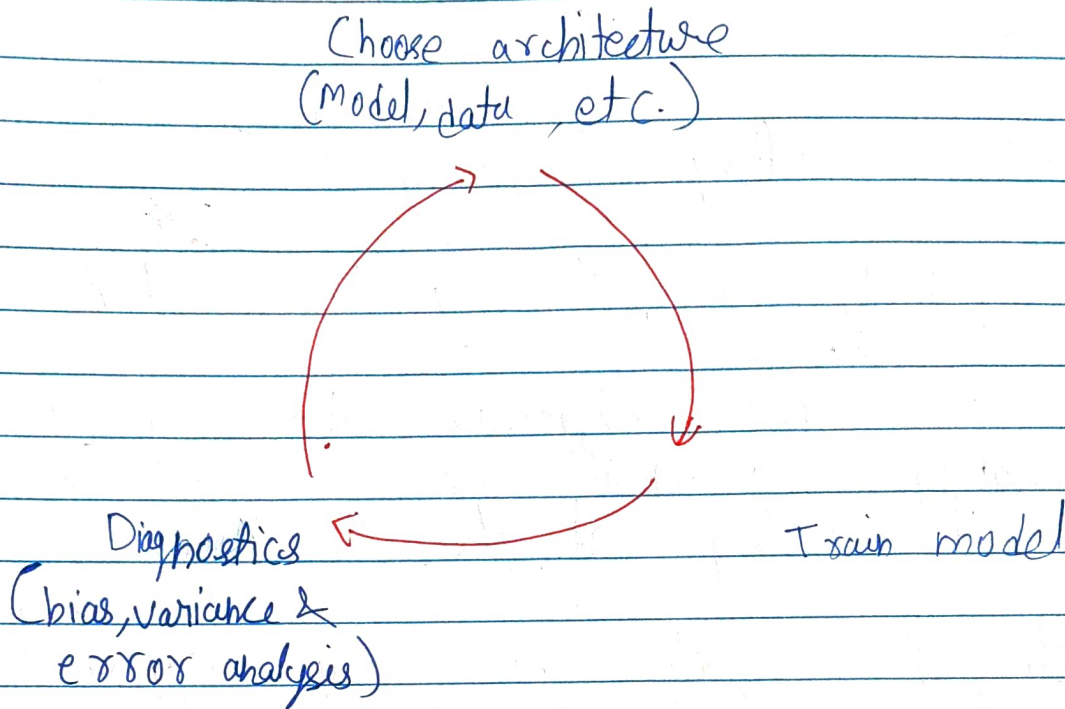
model = Sequential([layer-1, layer-2, layer-3])

⇒ Code explanation :- Regularized MNIST Model

in the code above,

The term "kernel" means weights matrix.
in "kernel_regularizer = L2(0.01)" we are implementing L2 regularization with $\lambda = 0.01$.

Iterative loop of ML development



Error analysis $\Rightarrow$

Let's take an example of spam email classifier

$n_{cv} = 500$, examples in Cross validation set

Algorithm misclassifies 100 of them

$\rightarrow$ Manually examine the 100 examples & categorize them based on common traits

~~etc~~

- Pharmaceutical emails : 21
- Deliberate misspelling (watches, medicine) : 3
- Unusual routing : 7
- Steal passwords (Phishing) : 18
- Spam message in embedded image : 5

So in the example above, you see pharma emails are "21", which tells you that the model is doing bad on these emails, whereas it works on "Deliberate misspellings" & only has "3" misclassified emails.

So pick up the problem which has bigger net impact.

The categories above aren't mutually exclusive meaning an email can be part of Pharma & Unusual email routing at the same time,

so if you have misclassified examples are way bigger like 1000, just randomly sample a 100 from them & analyse it.

So let's say as of our example Pharmaceutical emails are highly misclassified so let's see what we can do.

1) Collect more data

2) add more features like adding specific names of drugs & pharma products

& then you can work on the second biggest problem like "Steal passwords (Phishing)",

often the above points will help inspire you for what might be useful,

Error analysis can be a bit harder for tasks that humans aren't good at. for ex - if you're trying to predict

What ads someone will click on the websites. well, i can't predict what someone will click on. Error analysis there actually tends to be more difficult.

Adding data

→ Add more data of everything.
↳ This might be very slow & ~~comp~~ expensive.
an alternative way.

→ Add more data of the types where error analysis has indicated it might help.

in the context of the previous example we would try to add more data on Pharma^{spam} examples.

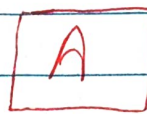
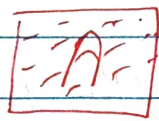
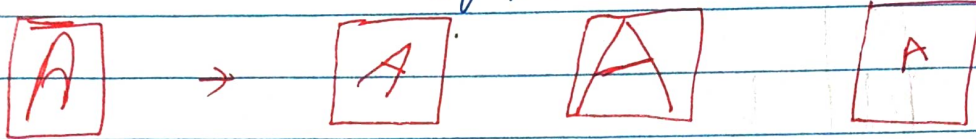
e.g. → go to unlabeled data & find more examples of Pharma related Spam & then will boost the learning algorithms performance much more than adding all sorts of data.

beyond getting brand new training examples (x, y) ,
another technique: Data Augmentation

Data Augmentation :-

Augmentation: modifying an existing training example to create a new training example.

ex → let's say you have an image of letter A
then you can → rotate, shrink, distort, enlarge the image but the image will still be A of the letter



→ ~~so~~ invert the image.

this idea also works for speech recognition.

ex →

- ① original audio
- ② original audio + crowd noise
- ③ original audio + car noise
- ④ original audio + Audio on bad cellphone connection

Note:- distortion introduced should be representative of the type of noise/distortion in the test set.
→ it doesn't help to add purely random/meaningless noise to your data.

Data synthesis

Synthesis: Using artificial data inputs to create a new training example.

ex → Artificial data synthesis for photo OCR.

OCR → Optical character recognition, it converts images of texts into machine readable & editable text data.

ex → you look at a billboard & you see text & then you can recognise the text from the image of billboard,

to generate synthetic data, you can go to your text editor get all the letters in each of the photo fonts & make them have different contrasts & that way you will get the image of the alphabets to train on synthetically without taking pictures of the billboards.

Transfer learning :- Using data from different tasks

Let's say you have a neural network for image recognition which recognizes images like, Cat's, dog, cars, people -- etc.

& you want to train a ML model that recognizes Digits from (zero - 9),

in transfer learning you can just take the hidden layers of the cat's, dog --- etc & update their weights & biases to get a new neural network

~~Transfer learning: Using data from a different task.~~

that will predict the probability of an image being a certain digit.

Now you have two ways of doing that

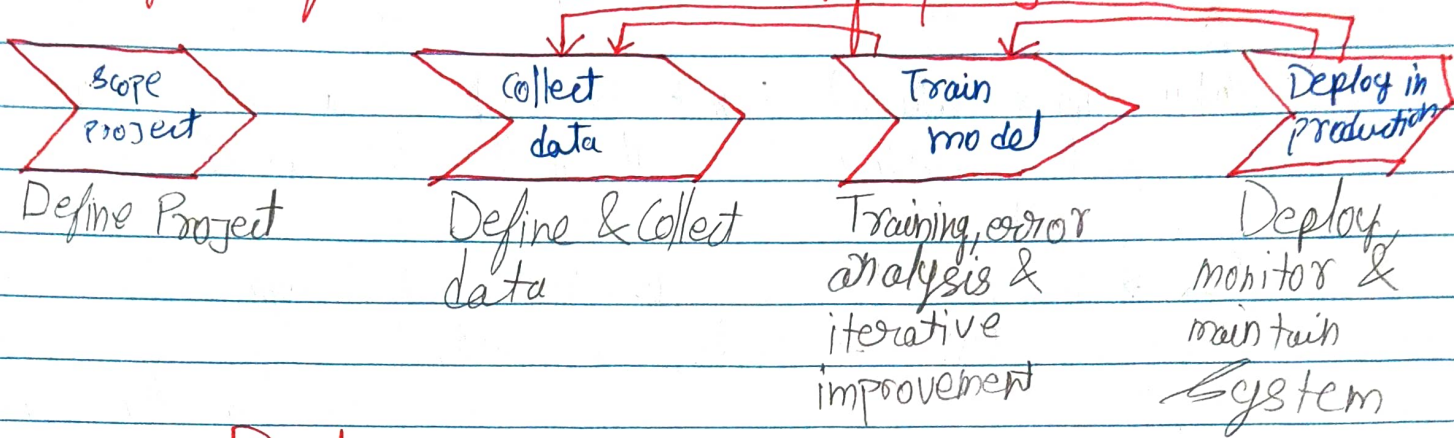
- ① Only train output layers parameters :- only when you have too small dataset.
- ② train all parameters :- when you have a little bit more data set.

→ So, while training the data on cat's & dog's & blah blah images The process is called "Supervised Pretraining" & Training that same neural net to detect digits is "fine tuning"

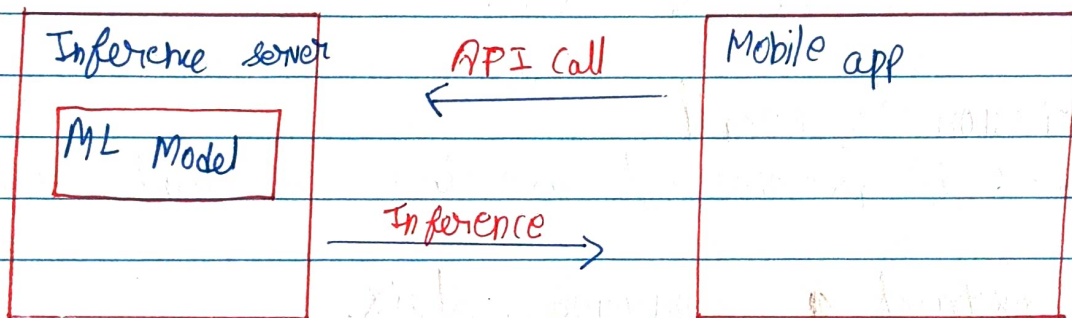
⇒ Why does transfer learning work?

⇒ So, you know that all image recognition models work like, their first hidden layer searches for edges, second searches or looks out for corners & third for curves & basic shapes & then a vivid full image. So yeah basically we take all the hidden layers that lookout for edges, corners, curves/basic shapes & then after a little bit of fine tuning it's done.

Full cycle of a machine learning project



Deployment



→ Software engineering may be needed for

- Ensure reliable & efficient predictions
- Scaling
- Logging
- System monitoring
- Model updates

ML OPs
machine learning
operations

Error metrics for skewed datasets

- ① when your dataset is ~~is~~ skewed, accuracy isn't a good marker to tell whether the model is doing good or not.

ex → if there are 1% patient who diagnose +ve & 99% -ve in the dataset,

So a dumb model who predicts -ve all the time will have accuracy of 99%.

Precision & recall

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

Let's construct a confusion matrix.

		Actual class	
		1	0
Predicted class	1	True +ve 15	False +ve 5
	0	False -ve 10	True -ve 70

True +ve → when the label is +ve & Prediction is +ve

False +ve → when the label is -ve but the prediction is +ve

Precision → (Of all the patients where we predicted $y=1$, what fraction actually have the rare disease?)

$$\Rightarrow \frac{\text{True +ve}}{\# \text{ predicted +ve}} = \frac{\text{True +ve}}{\text{True +ve} + \text{False +ve}} = \frac{15}{15+5} = 0.75$$

Recall → (Of all patients that actually have the rare disease, what fraction did we correctly detect as having it?)

$$\frac{\text{True +ve}}{\# \text{ actual +ve}} = \frac{\text{True +ve}}{\text{True +ve} + \text{False -ve}} = \frac{15}{15+10} = 0.6$$

Trading off Precision & Recall

Predict 1 if $f_{\vec{w},b}(\vec{x}) \geq 0.7$

Predict 0 if $f_{\vec{w},b}(\vec{x}) < 0.7$

⇒ Suppose we want to predict $y=1$ (rare disease) only if very confident,

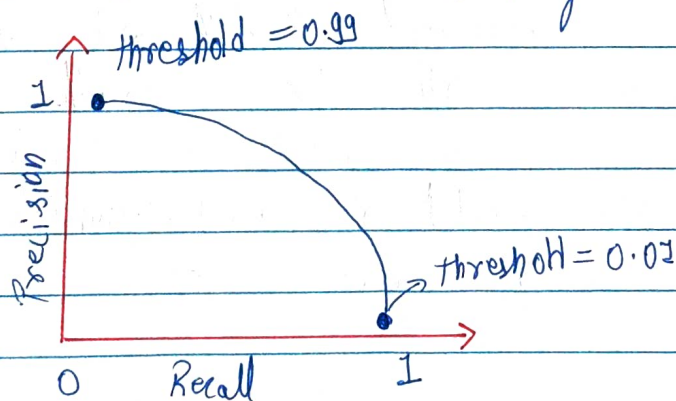
"higher Precision, lower recall"

so we can increase the threshold from 0.7 to 0.8 or even but it will decrease the recall.

⇒ Suppose we want to avoid missing too many cases of rare disease (when in doubt predict $y=1$)

"So we can decrease the threshold from 0.7 to 0.4 or 0.3, but it will decrease the precision"

"Lower Precision, higher recall"



The correct way to pick right trade off between Precision & Recall is to do manually but if you want to do it automatically, you can do it with the help of "F1 Score"

let's say you have 3 algorithms

	Precision (P)	Recall (R)	F ₁ Score
Algo 1	0.5	0.4	0.444
Algo 2	0.7	0.1	0.175
Algo 3	0.02	1.0	0.0392

$$F_1 \text{ score} = \frac{1}{\frac{1}{P} + \frac{1}{R}}$$

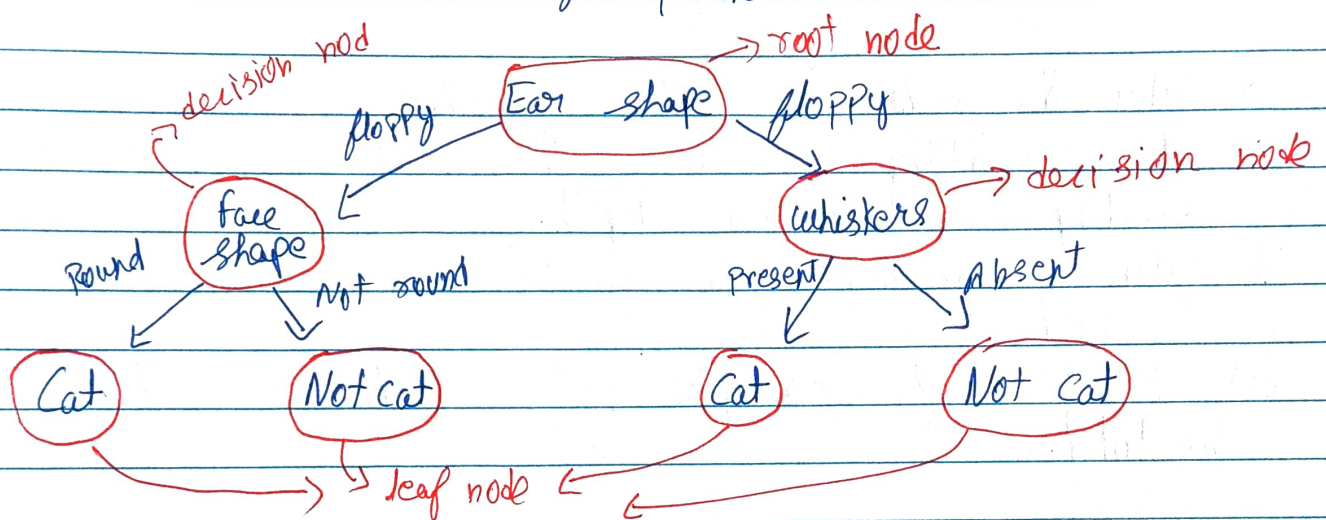
We take F1 score because it accounts for small values, meaning if either of the two values Precision & Recall are very small the F1 score will be small.

It reduces the chance that our model is a dummy which only predicts one single value like `Print("y=1")`.

Decision Trees

→ A supervised ML-model that is represented as a tree structure, it follows certain conditions to reach the outcome.

ex → A cat image prediction model



Root node :- The main node where you start your inference generation/prediction generation

decision node :- The conditional node where your model decides what is what like yes or no

leaf nodes are outcome nodes

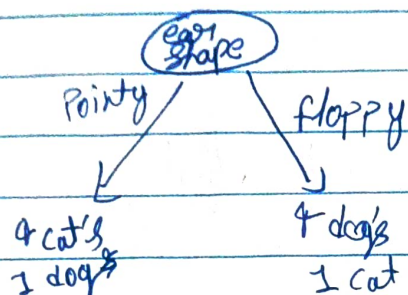
Decision tree learning process

→ Decision ① How to choose what features to split on at each node?

⇒ The one that maximizes purity or minimizes impurity.

5 dogs & 5 cat's

ex →



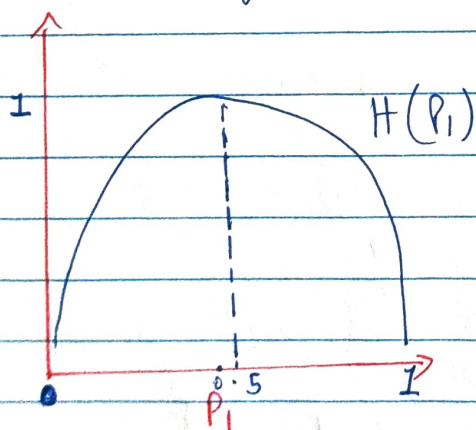
in this example this one "car shape" maximizes purity that's why we use it.

Decision ② ∴ when do you stop splitting?

- when a node is 100% one class
- when splitting a node will result in the tree exceeding a maximum depth.
- when improvements in purity score are below a threshold
- when no of examples in a node is below a threshold.

Entropy as a measure of impurity

P_1 = Fraction of examples that are cats



ex → when cat's are 3 & dogs are 3 $H(P_1) = \frac{3}{6} = \frac{1}{2}$, which on the curves gives highest impurity.

Cat → 5, dog → 1, $P(P_1) = 5/6 = 0.65$

Cat → 1, dog → 0, $H(P_1) = 0$, $P_1 = \frac{1}{1} = 1$

Now,

P_1 = fraction of examples that are cats
 $P_0 = 1 - P_1$

$$\{ H(P_1) = -P_1 \log_2(P_1) - P_0 \log_2(P_0) \} \text{ This is called as entropy function}$$

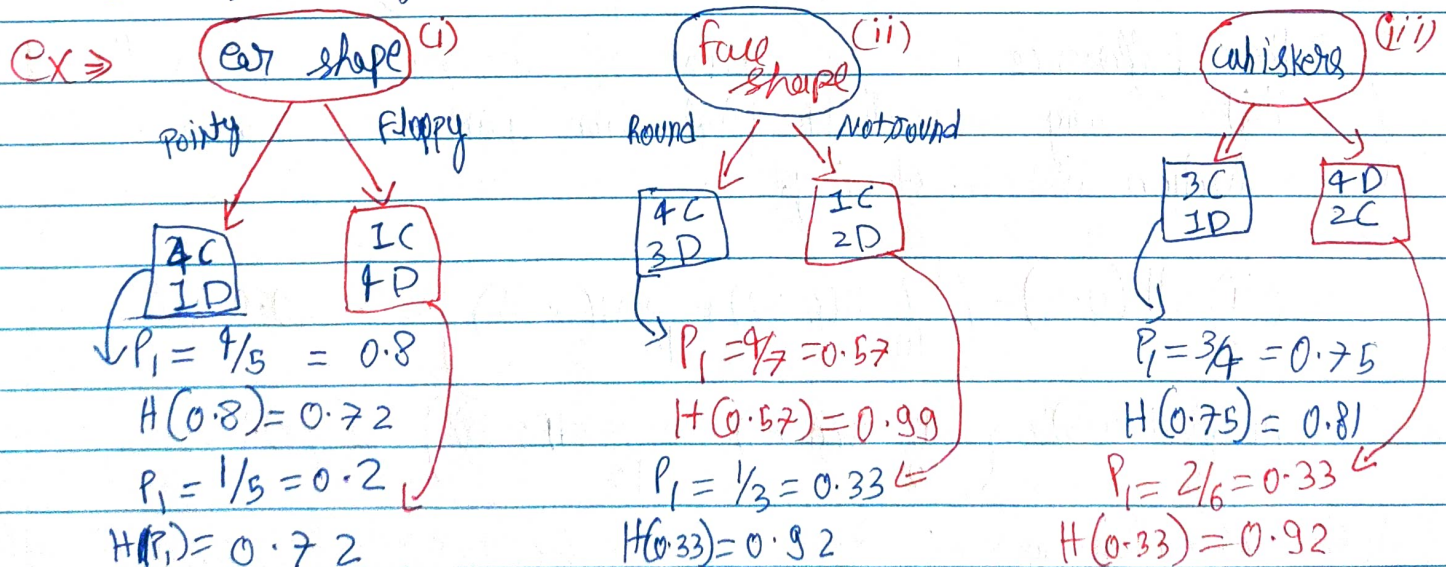
if one of P_1 or P_0 is zero
 then $\log(0)$ is not defined
 but we take, $-P_1 \log_2(P_1)$
 $\Rightarrow -0 \log_2(0) = 0$

Choosing a Split: Information gain

→ The reduction of entropy is called information gain

Choosing a split ∴ a split is a decision on which we will be splitting our data.

C → Cat, D → Dog



rather than looking at individual entropy value & deciding what split to choose what we will do is, take a

weighted average of the split

for, ex (i)

$$\frac{\overset{\text{no of examples in the split}}{5}}{10} H(0.8) + \frac{5}{10} H(0.2)$$

→ Total no of examples before splitting

$$(ii) \quad \frac{7}{10} H(0.57) + \frac{3}{10} H(0.33)$$

$$(iii) \quad \frac{4}{10} H(0.75) + \frac{6}{10} H(0.33)$$

we calculate, the difference between the entropy of before the split & entropy after the split.

$$\text{like } \Rightarrow (i) \quad H(0.5) - \left(\frac{5}{10} H(0.8) + \frac{5}{10} H(0.2) \right) = 0.28$$

⇒ The difference is the "amount of disorder removed" by the split. That's why we call it information gain - it measures the reduction in uncertainty.

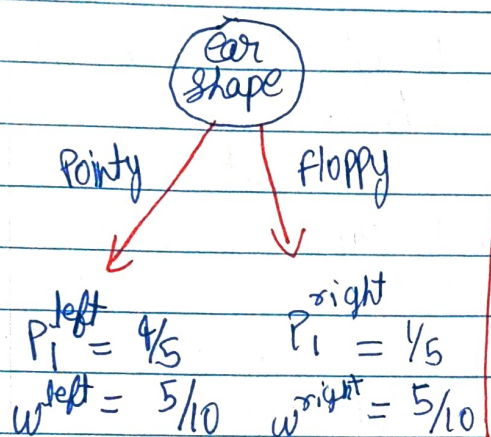
$$(ii) \quad H(0.5) - \left(\frac{7}{10} H(0.57) + \frac{3}{10} H(0.33) \right) = 0.03$$

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

{ I.G (Information gain) = 0.28, 0.03, 0.12 } respectively

so, in the example (i) we get the biggest entropy reduction of "0.28", so we will choose example (i) of car shape.

General Information gain



Information gain

$$\Rightarrow H(P_i^{\text{root}}) - (w^{\text{left}} H(P_i^{\text{left}}) + w^{\text{right}} H(P_i^{\text{right}}))$$

Putting it all together

Decision tree learning

- Start with all examples at the root node
- Calculate information gain for all possible features, & pick the one with the highest information gain.
- Split dataset according to selected feature, & create left & right branches of the tree.
- keep repeating split process & until stopping criteria is met:
 - ↳ when a node is 100% one class
 - ↳ when splitting a node will result in the tree exceeding a maximum depth.
 - ↳ Information gain from additional splits is less than threshold.
 - ↳ when number of examples in a node is below a threshold

⇒ Decision trees use recursive algorithm ~~to~~ like creating a sub division of decision trees & putting them together to get a bigger decision tree

Using One-hot encoding of Categorical features

In the examples we have seen so far, each of the feature could take on only one of two possible values.

But what if a feature takes multiple values ex → Port type

→ Micro USB

→ USB

→ Type C

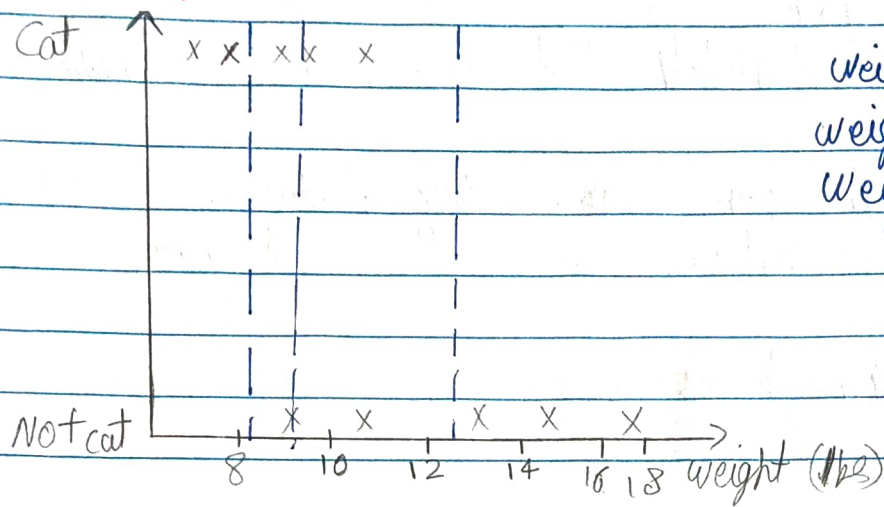
⇒ Charging type	Micro USB	USB	Type C
① → Micro USB	1	0	0
② → USB	0	1	0
③ → Type C	0	0	1
④ → Micro USB	1	0	0
⑤ → Type C	0	0	1
⑥ → Type C	0	0	1

But, we drop a column after doing one hot encoding to avoid multicollinearity.

in one hot encoding we take a feature with multiple input values, & divide each input into a sub-feature that can take a value 0 or 1.

in the above example, we divide charging type column into 3 sub columns that take on the values 0 or 1

Splitting on continuous variable



$$\left. \begin{array}{l} \text{weight} \leq 8 \\ \text{weight} \leq 9 \\ \text{weight} \leq 12 \end{array} \right\} \begin{array}{l} \text{IG} = 0.24 \\ \text{"} = 0.61 \\ \text{"} = 0.90 \end{array}$$

ex	car shape	weight (lbs)
	round	7.2
	Pointy	8.8
	floppy	15
	Pointy	9.2
	Pointy	8.4
	round	7.6
	Round	11
	floppy	10.2
	Floppy	18
	round	20

When you have continuous variables you can decide how which value to split your dataset on by choosing the value of the weight that has the biggest information gain.

General method,

- Sort all the examples by their weight.
- Look at midpoints between each pair of consecutive examples
- if you have 10 examples, you test 9 thresholds

⇒ for each threshold

↳ compute info gain.

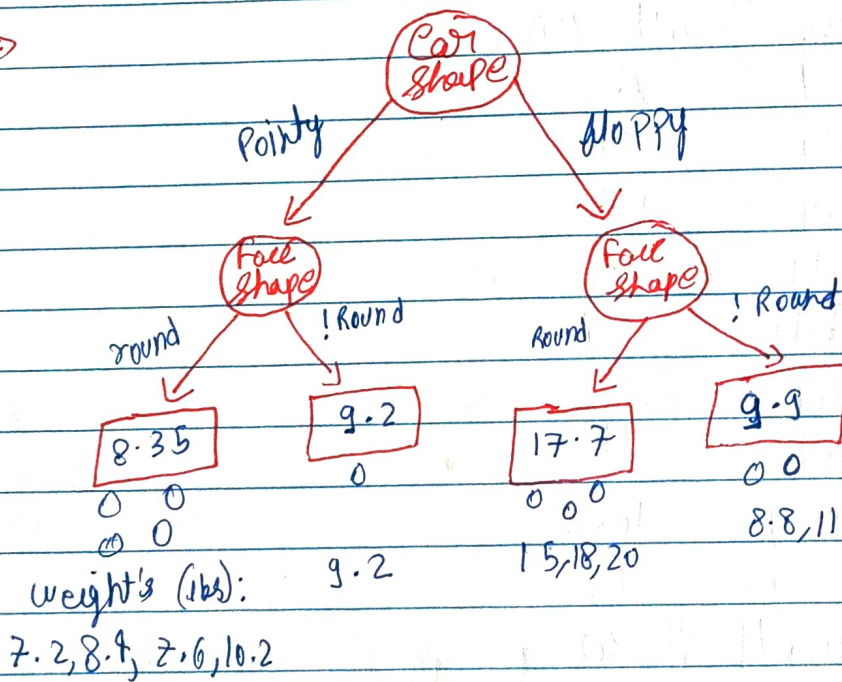
↳ Pick the threshold that gives the maximum

→ Compare with info gains from other features.

→ Split on whichever feature gives the best gains

Decision Trees ∴ Regression trees
Learning

⇒



{ in the leaf node we will have the average of
all the weights in that split }

{ btw it's ok if the condition is in both the decision
node for example in the node above we have "face
shape" on both decision nodes }

Choosing a Split

↳ it's same as we did before but this time we will use variance instead of entropy.

variance at root node ↳ 20.51		↳ 20.51		↳ 20.51	
car shape pointy weight = 7.2, 9.2, 8.4, 7.6, 10.2 variance 1.47 $w_{\text{left}} = 5/10$ floppy 8.8, 15, 11, 18, 20 variance 21.87 $w_{\text{right}} = 5/10$		face shape Round 7.2, 15, 8.4, 7.6, 10.2, 18, 20 variance 27.80 $w_{\text{left}} = 7/10$! Round weights 8.8, 9.3, 11 variance 1.37 $w_{\text{right}} = 3/10$		whiskers 7.2, 8.8, 9.2, 8.4 variance 0.75 $w_{\text{left}} = 4/10$ 15, 7.6, 11, 10.2, 18, 20 variance 23.32 $w_{\text{right}} = 6/10$	
Reduction in variance $20.51 - \left(\frac{5}{10} \times 1.47 + \frac{5}{10} \times 21.87 \right)$ $= 8.84$		$20.51 - \left(\frac{7}{10} \times 27.80 + \frac{3}{10} \times 1.37 \right)$ $= 0.64$		$20.51 - \left(\frac{4}{10} \times 0.75 + \frac{6}{10} \times 23.32 \right)$ $= 6.22$	
weighted average of variance					

→ Here the ~~root~~ root node with "ear shape" has the most reduction in ~~the~~ variance so we will choose that

Steps → ① → calculate the variance for each split of the decision
 → Calculate the variance before the split or the root node.

Step → ② → find the weighted average of the variance

Step → ③ → Subtract the weighted average mean from root mean.

Step → ④ → Select the highest number, for & that will be the split.

Using multiple decision trees:-

Having multiple decision trees for same type prediction & having them do the prediction & choose the prediction with maximum number of prediction quantity.

→ It's done because a decision is very sensitive to the training data, if you change even a few example it would be some entirely new "Tree"

The collection of Trees is called:- "Tree ensemble"

ex → "let's say there are 5 trees & 4 of them predict it's a cat & 1 predicts it's not a cat, then we conclude based on majority decision that it's a cat."

Tree ensembles = Sampling with replacement

So what we do is take a bunch of data and create a training set, ~~and~~ & then sample from it, using replacement & do it multiple times to get a pool of sampled data,

Now using that sampled data let's say you sampled 3 times took 3 sets of data by sampling the original data & now you

have 3 datasets as same size of the original dataset, so you can use these 3 datasets which ^{are now} sampled with the replacement to train 3 different decision trees model to predict or inference or prediction, & now you have a "Tree ensemble of 3".

Random forest Algorithm

↳ This is a tree ensemble algorithm

Given training set of size m

For $b=1$ to B : # Here B is the no of decision trees
use sampling with replacement to create a new training set of size m that we are going to create it's generally good to take a no btw 64 to 128.

m Train a decision tree on the new dataset

{ one specific instance creation of tree ensemble is sometimes also called a bagged decision tree }

Sometimes you end up with always using the same split at the root node & very similar split near the root node.

like you have " B " no of decision trees in the random forest algorithm they all almost all start with lets, say ear shape in the root node which is not desired

To tackle this we do,

Randomizing the feature choice

At each node, when choosing a feature to use to split, if n features are available, pick a random ~~sub~~ subset of " k ", " $< n$ " features & allow the algorithm to only choose from that subset of " k " features.

to choose k we typically use the formulae,

$$k = \sqrt{n}$$

XGBoost $\therefore$ eXtreme Gradient Boost

Boosted trees intuition,

Given training set of size m .

For $b=1$ to B :

Use sampling with replacement to create a new training set of size m ,

But instead of picking from all examples with equal $(1/m)$ probability, make it more likely ~~like~~ to pick misclassified examples from previously trained trees.

$\rightarrow$ in simpler words, rather than looking at all the training examples we focus more attention on the subset of examples it is not ~~get~~ doing well on & get the new decision tree, the next decision tree to try to do well on them

XGBoost (eXtreme Gradient Boosting)

- Open source implementation of boosted trees
- fast efficient implementation
- Good choice of default splitting criteria & criteria for when to stop splitting
- Built in regularization to prevent overfitting
- highly competitive algorithm for machine learning competitions (eg: kaggle competitions)
- Rather than doing sampling with replacement XGBoost actually assigns different weights to different training examples, so it doesn't need to generate a lot of randomly chosen training sets, it makes a little bit more efficient.

Code :-

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 & neural networks?

Decision Trees & Tree ensembles

- works well on tabular (structured data)
- Not recommended for unstructured data (image, audio, text)

→ Fast to train

→ small decision trees may be human interpretable

Neural networks

- Works well on all types of data, including tabular (structured) & unstructured data (images, audio etc)
- Maybe slower than a decision tree
- Works with transfer learning
- when building a system of multiple models $\neq$ working together, it might be easier to string together multiple neural networks.