

20/7

Advanced Learning Algorithms

Week 1 - Neural Networks.

↳ Inference.

Week 2 → Training.

Week 3 → How to build ML systems.

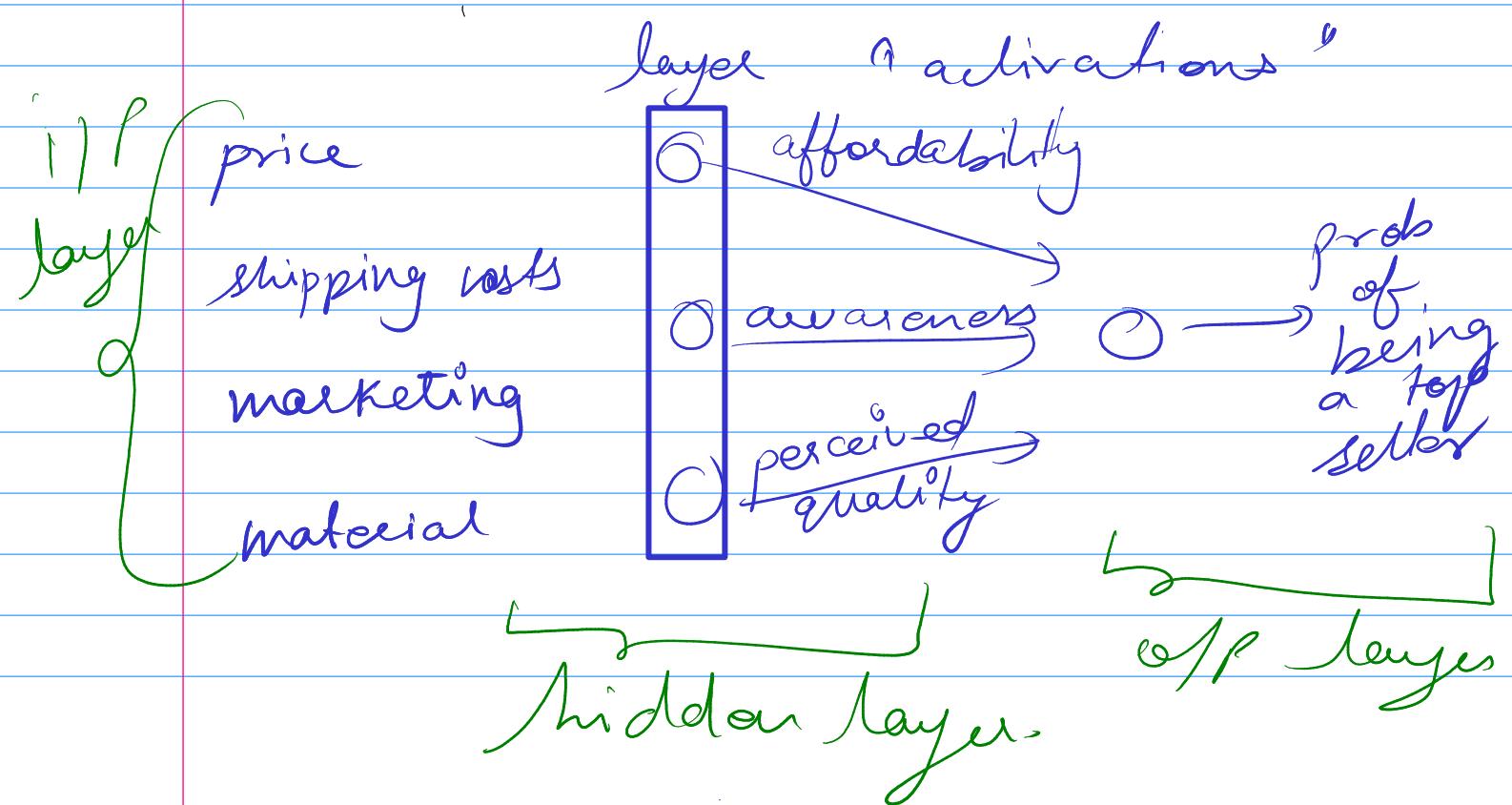
Week 4 → Decision Trees.

Week 1

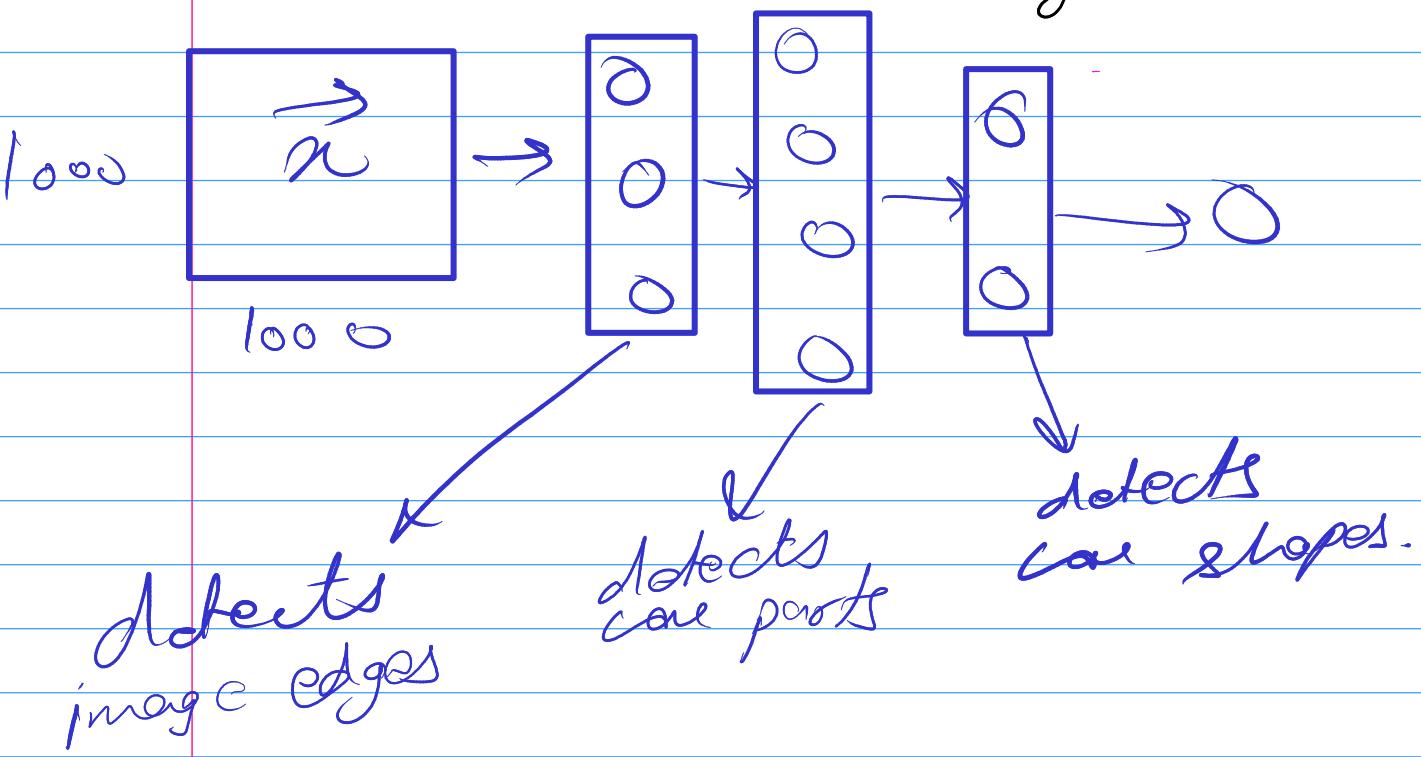
Demand Prediction

$$Q = f(x) = \frac{1}{1 + e^{-(w_0 + w_1 x_1)}}$$

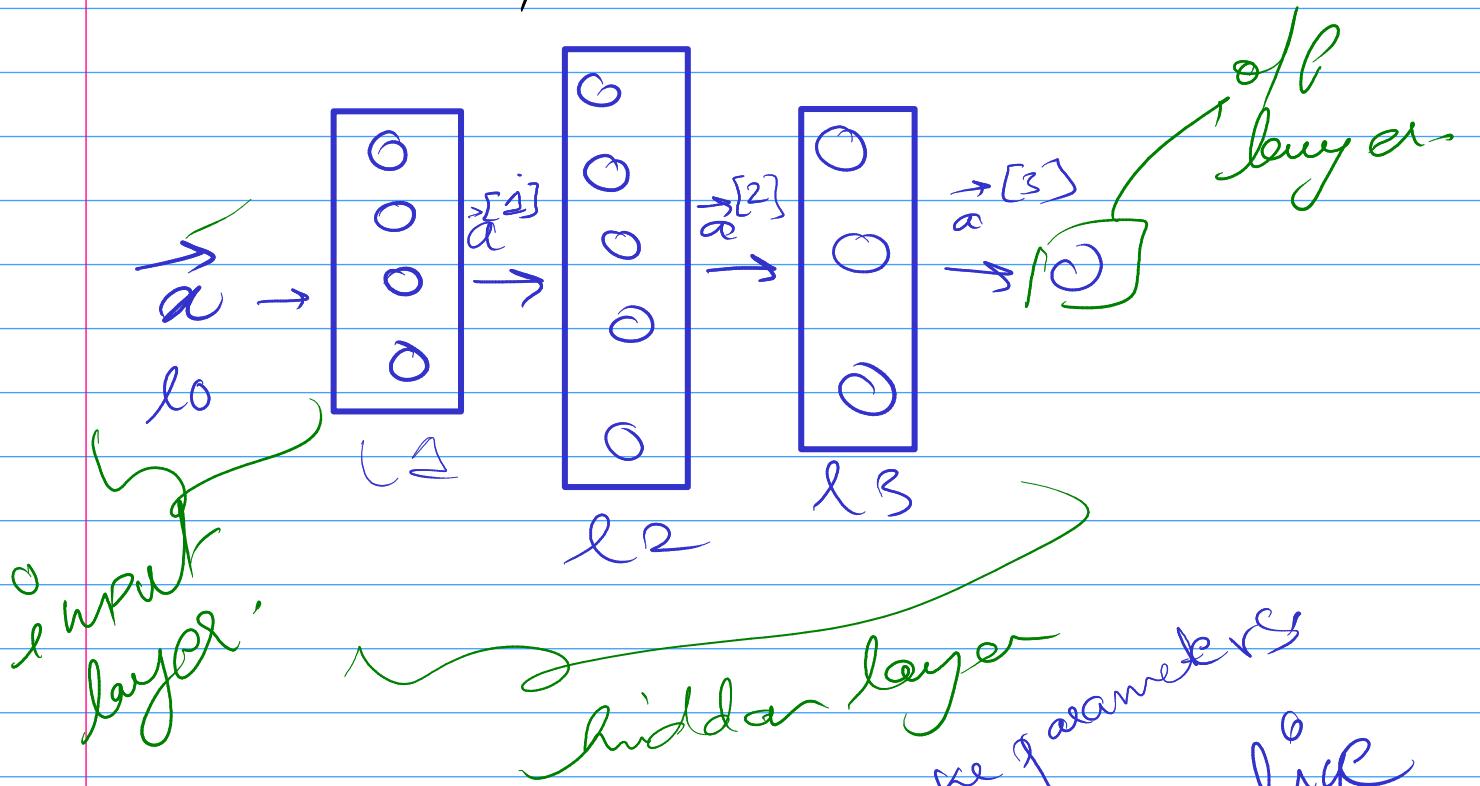
activation function.



Car Classification using ANN



More Complex Neural Networks.



Layer 3 zoomed

$$w_1, b_1 \quad a_1^{[3]} = g(w_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]})$$

$$w_2, b_2 \quad a_2^{[3]} = g(w_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]})$$

~~General formula~~

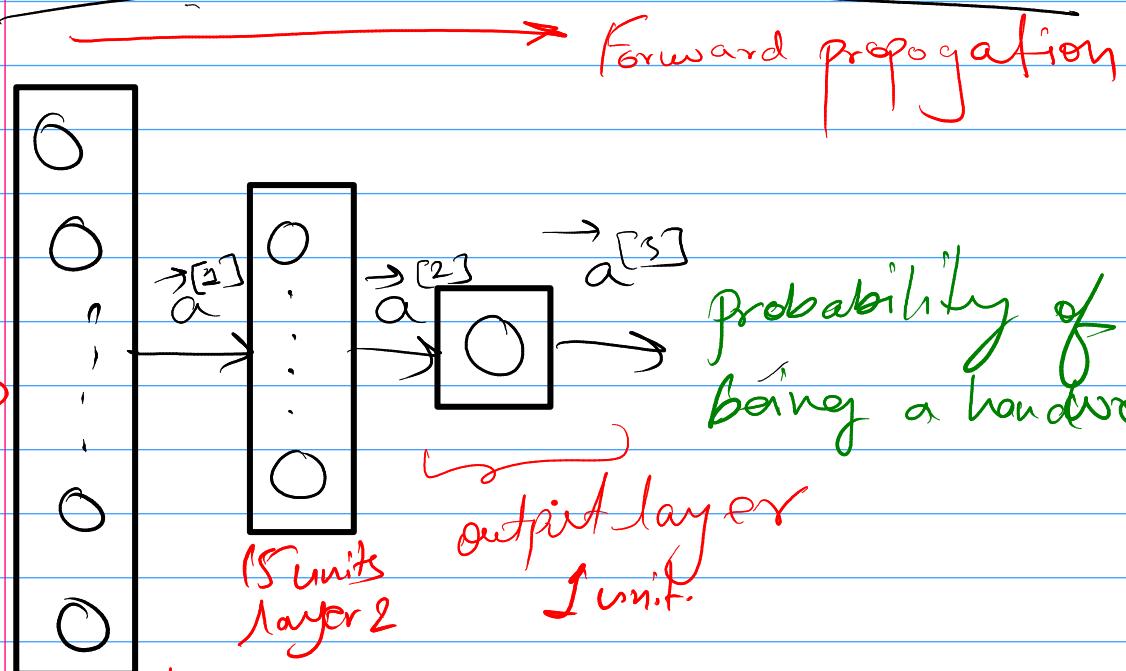
$$\vec{a}_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})$$

$l \rightarrow$ denotes layer l .

$j \rightarrow$ denotes unit j {Neuron no.}

$g \rightarrow$ Sigmoid function

Inference: making predictions -



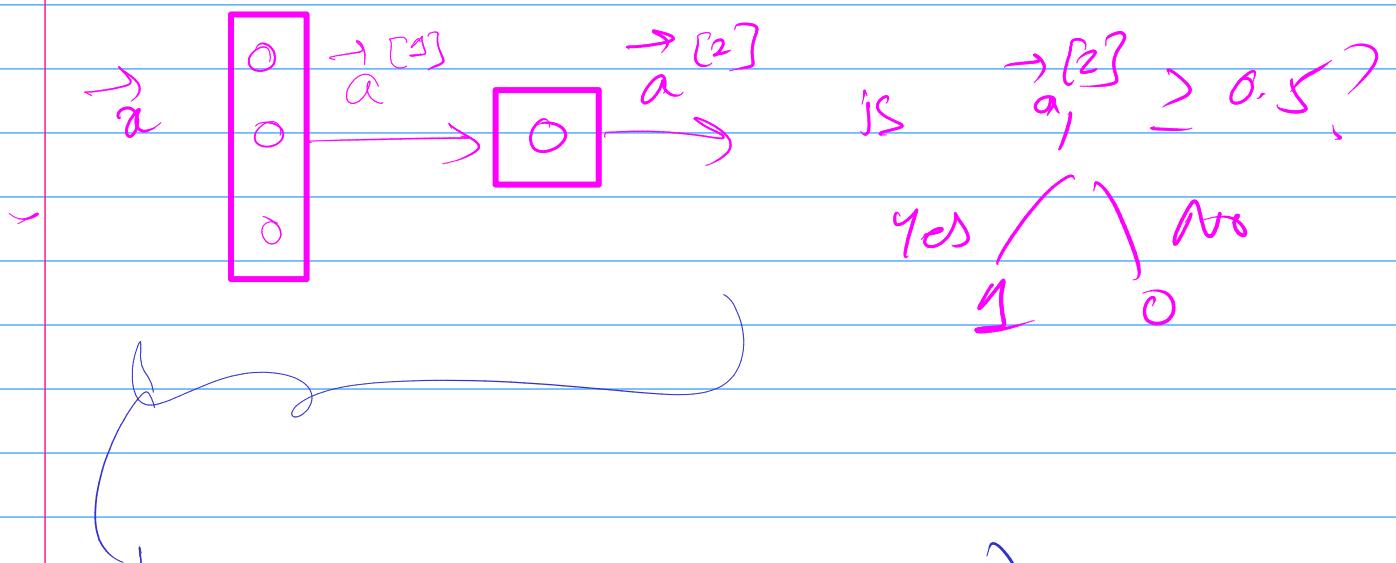
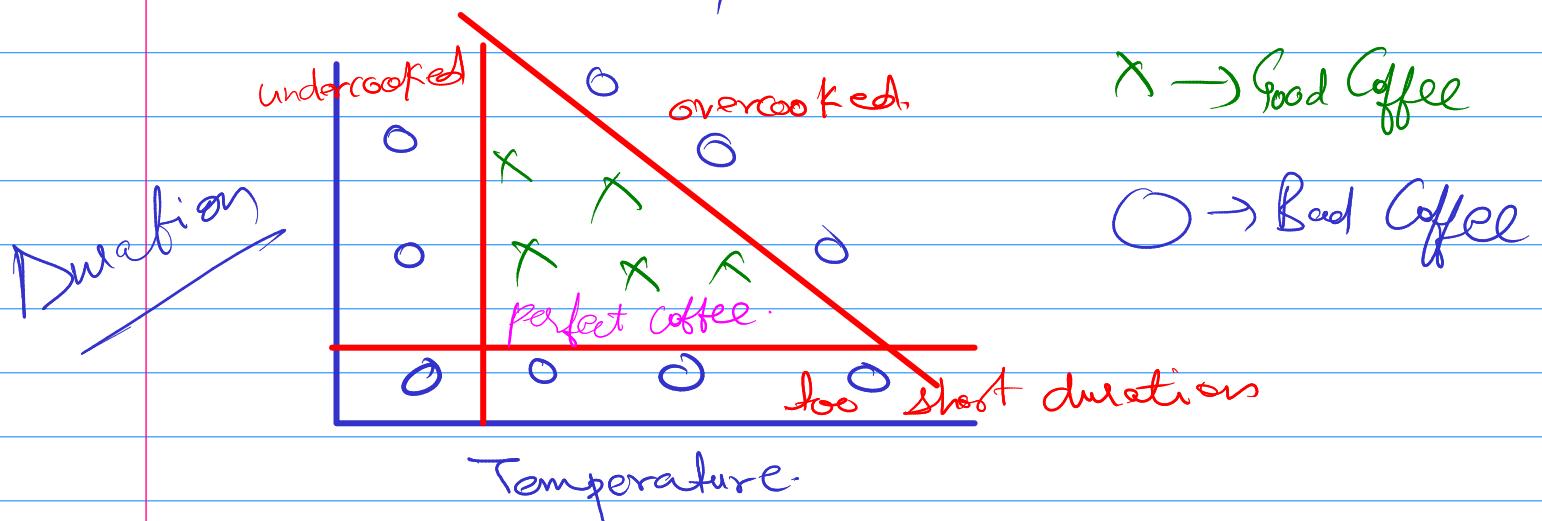
$$a^{[1]} = \begin{bmatrix} g(\vec{\omega}_1 \cdot \vec{x} + b_1^{[1]}) \\ \vdots \\ g(\vec{\omega}_{25} \cdot \vec{x} + b_{25}^{[1]}) \end{bmatrix}$$

(sigmoid)

Tensorflow Implementation

Coffee Roasting Example -

Imagine we have raw coffee beans and we want to roast it. The roasting params we can control are duration & temperature.



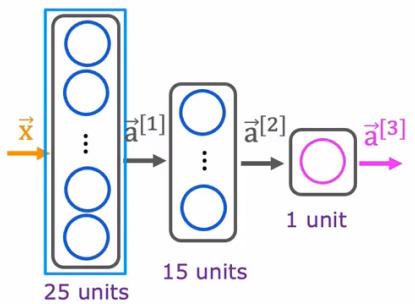
$\vec{a} = \text{np.array}([200.0, 17.0])$
 $\text{layer_1} = \text{Dense}(\text{units}=3, \text{activation}=\text{'sigmoid'})$

$$a_1 = \text{layer_1}(\vec{a})$$

layer 2 = Dense(units='1', activation='sigmoid')
 $a_2 = \text{layer}_2(a_1)$

if $a_2 \geq 0.5$:
yhat = 1

else:
yhat = 0



$a = \text{np.array}([0, 0, \dots, 245, 0, \dots])$
 $l-1 = \text{Dense}(\text{units}=25, \text{activation}=\text{'sigmoid'})$
 $a_1 = l-1(a)$

$l-2 = \text{Dense}(\text{units}=15, \text{activation}=\text{'sigmoid'})$
 $a_2 = l-2(a_1)$

$l-3 = \text{Dense}(\text{units}=1, \text{activation}=\text{'sigmoid'})$
 $a_3 = l-3(a_2)$

if $a_3 \geq 0.5$:
yhat = 1

else :

yhat = 0

How is data represented in Tensorflow.

Note about numpy arrays

3 columns
2 rows
2 x 3 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

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

```
[[1, 2, 3],  
 [4, 5, 6]]
```

$$\begin{bmatrix} 0.1 & 0.2 \\ -3 & -4 \\ -0.5 & -0.6 \\ 7 & 8 \end{bmatrix}$$

```
x = np.array([[0.1, 0.2],  
             [-3.0, -4.0],  
             [-0.5, -0.6],  
             [7.0, 8.0]])
```

```
[[0.1, 0.2],  
 [-3.0, -4.0],  
 [-0.5, -0.6],  
 [7.0, 8.0]]
```

DeepLearning.AI Stanford ONLINE

Andrew Ng

2, 3 is the first row of this matrix and

4, 5,

Note about numpy arrays

Very
2D
matrix

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

→ [200 17]

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

→ [200]
→ [17]

1×2 row vector

2×1 column vector

This is
only an
array.

→ $x = np.array([200, 17])$

just a 1D array.
1D "Vector"

no columns, but
it's just a list of numbers.

DeepLearning.AI Stanford ONLINE

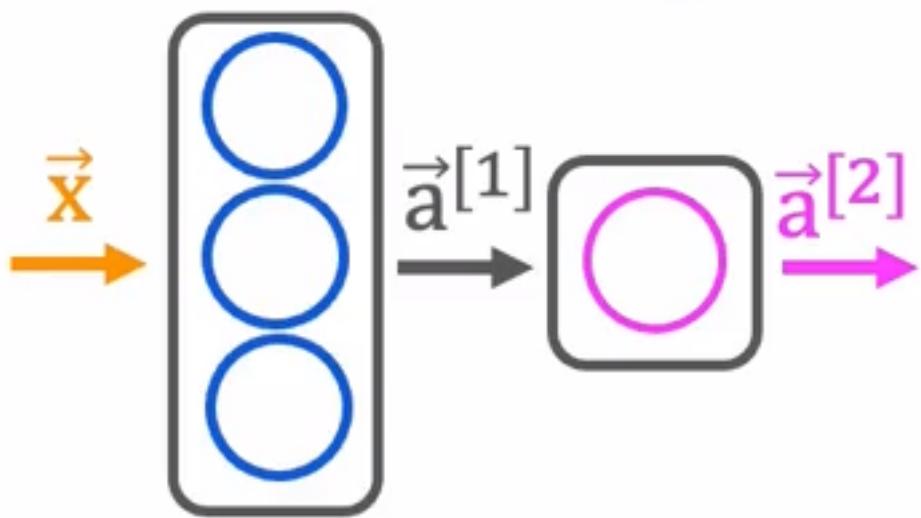
Andrew Ng

Note:

In tensorflow the convention is to use matrices to represent data

— x —

Building a neural network.



layer_1 = Dense(units=3, activation='sigmoid')
layer_2 = Dense(units=1, activation='sigmoid')

model = Sequential([layer_1, layer_2])

Tells Tensorflow to sequentially string together these layers into a neural network.

$x = np.array([[200.0, 17.0],$
 $[120.0, 5.0],$
 $[425.0, 20.0],$
 $[212.0, 18.0]])$

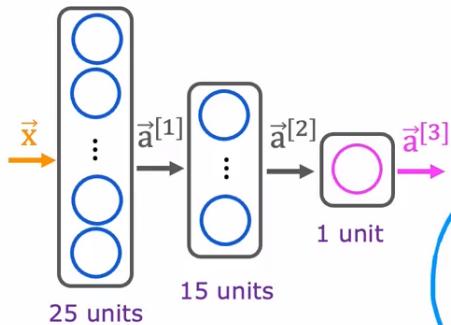
$y = np.array([1, 0, 0, 1])$

model.compile(...)
model.fit(x,y)

model.predict(x_new)

Another example

Digit classification model



```
→ layer_1 = Dense(units=25, activation="sigmoid")
→ layer_2 = Dense(units=15, activation="sigmoid")
→ layer_3 = Dense(units=1, activation="sigmoid")
→ model = Sequential([layer_1, layer_2, layer_3])
model.compile(...)

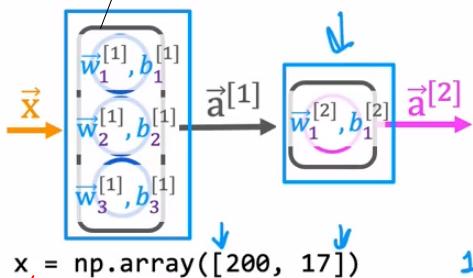
x = np.array([[0..., 245, ..., 17],
              [0..., 200, ..., 184]])
y = np.array([1,0])

model.fit(x,y) ← more about this next week!
→ model.predict(x_new)
```

X new and similar to what you saw before
with the coffee classification network

Forward prop in a single layer -

forward prop (coffee roasting model)



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

1D arrays

$$w_1^{[2]} \quad w_{2-1}$$

Break em down to steps

① $w1_1 = \text{np.array}([1, 2])$ ② $w1_2 = \text{np.array}([-3, 4])$ ③ $w1_3 = \text{np.array}([5, -6])$
 ④ $b1_1 = \text{np.array}([-1])$ $b1_2 = \text{np.array}([1])$ $b1_3 = \text{np.array}([2])$
 $z1_1 = \text{np.dot}(w1_1, x) + b1_1$ $z1_2 = \text{np.dot}(w1_2, x) + b1_2$ $z1_3 = \text{np.dot}(w1_3, x) + b1_3$
 $a1_1 = \text{sigmoid}(z1_1)$ $a1_2 = \text{sigmoid}(z1_2)$ $a1_3 = \text{sigmoid}(z1_3)$
 $a1 = \text{np.array}([a1_1, a1_2, a1_3])$

DeepLearning.AI

Stanford ONLINE

Fed to layer 2 as input

Andrew Ng

General Implementation of Fwd. propagation.

def dense(a_{in} , w , b):

 units = $w.shape[0]$ all cols

$a_{out} = np.zeros(units)$

 for j in range(units):

$w = W[:, j]$ traverse all rows of jth column.

$z = np.dot(w, a_{in}) + b[j]$

$a_{out} = g(z)$

 return a_{out} .

def parameters from prev layer activations
and sends it to next layer.

def sequential(a):

$a_1 = \text{dense}(a, w_1, b_1)$

$a_2 = \text{dense}(a_1, w_2, b_2)$

$a_3 = \text{dense}(a_2, w_3, b_3)$

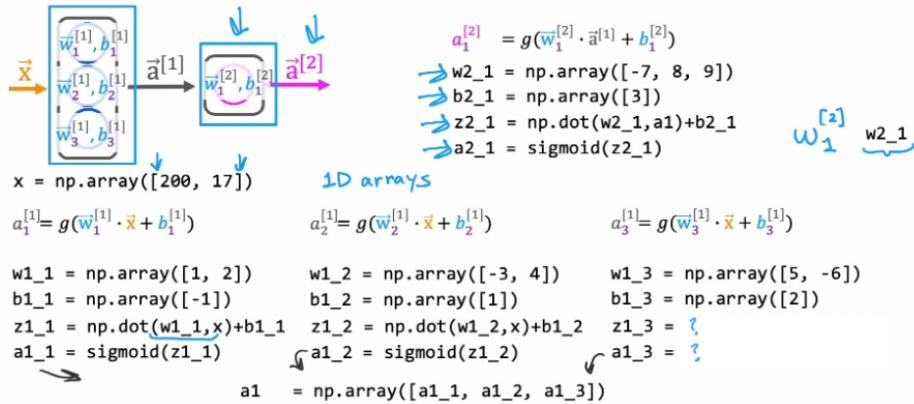
$a_4 = \text{dense}(a_3, w_4, b_4)$

$f_{\text{out}} = a_4$

return f

1 point

forward prop (coffee roasting model)



- According to the lecture, how do you calculate the activation of the third neuron in the first layer using NumPy?

$z_{1-3} = \text{np.dot}(w1_3, x) + b1_3$

$a1_3 = \text{sigmoid}(z_{1-3})$

Is there a path to AGI?

AI

ANI

AGI

Artificial Narrow Intelligence

An AI system that does a particular task very well. It's narrowed down to do a task of a set of tasks.

Artificial General Intelligence

Do anything a human can do.

Vectorization

How neural networks are implemented efficiently -

Using For Loops.

```

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):
    units = W.shape[1]
    a_out = np.zeros(units)
    for j in range(units):
        w = W[:,j]
        z = np.dot(w, a_in) + b[j]
        a_out[j] = g(z)
    return a_out
  
```

[1,0,1]

✓S

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

29

Flow of using TensorFlow

load_data()

check the dimensions of the data

model = Sequential[

Input Shape,

Layers

- - -

- - -

], name = "my-model")

)

↓

model.summary()

↓

Examine the weights-

↓

w1, b1 = layer_1.get_weights()

↓

model.compile(

loss = ff.keras.losses.BinaryCrossentropy(),

optimizer = ff.keras.optimizers.Adam(0.001),

)

↓

model.fit(

X, Y

, epochs = 50

)

\downarrow
prediction = model.predict($X[6].reshape(1, 400)$)

prediction = model.predict($X[500].reshape(1, 400)$)

\downarrow
if prediction >= 5:
yhat = 1

else:
yhat = 0



Note: Confusion b/w w and $w[i]$

Dot Product Requirement

`np.dot(a_in, w) → dono ka shape (n,) hona chahiye.`

- Agar tum `w[i]` likhoge → `w[i]` ek scalar ban jayega (sirf ek weight pick ho gaya) → pura dot product kharab ho jayega.
- Dot product hamesha **poora vector** ke saath hota hai → isliye `w = w[:, i]` lena zaroori hai.

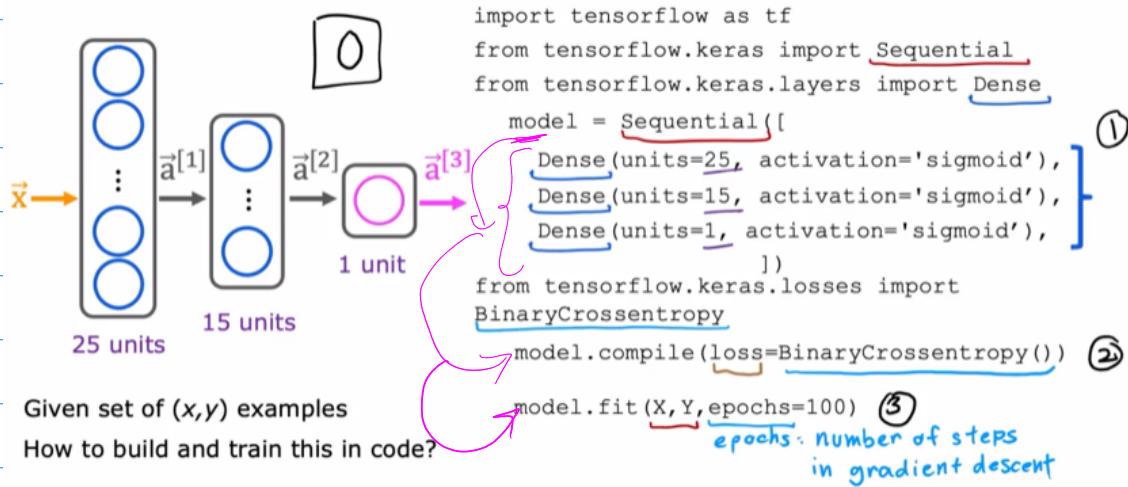
Rule of Thumb

- Poora column of weights (ek neuron ke liye) → `w[:, i]` (vector)
- Ek single scalar weight (sirf ek feature ka ek weight) → `w[i, j]` ya `w[i]` (scalar)

Week 2

Tensorflow Implementations

Train a NN in tf (code)



Model Training Steps

① Define the model given \vec{w}, \vec{b} & \vec{x} .

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

In logistic regression,

$$z = \text{np.dot}(w, x) + b$$

$$f_x = 1 / (1 + \text{np.exp}(-z))$$

②

Specify the loss and cost.

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

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

In Logistic Regression

$$\begin{aligned} \text{loss} = & -y * \text{np.log}(f_x) \\ & - (1-y) * \text{np.log}(1-f_x) \end{aligned}$$

③

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

$$w = w - \alpha * dj_{dw}$$

$$b = b - \alpha * dj_{db}$$

The 3 steps in model training in Neural Networks

Step 1: $\text{model} = \text{Sequential}([\text{Dense}(\dots), \dots, \dots])$ } Describes the whole architecture of nn

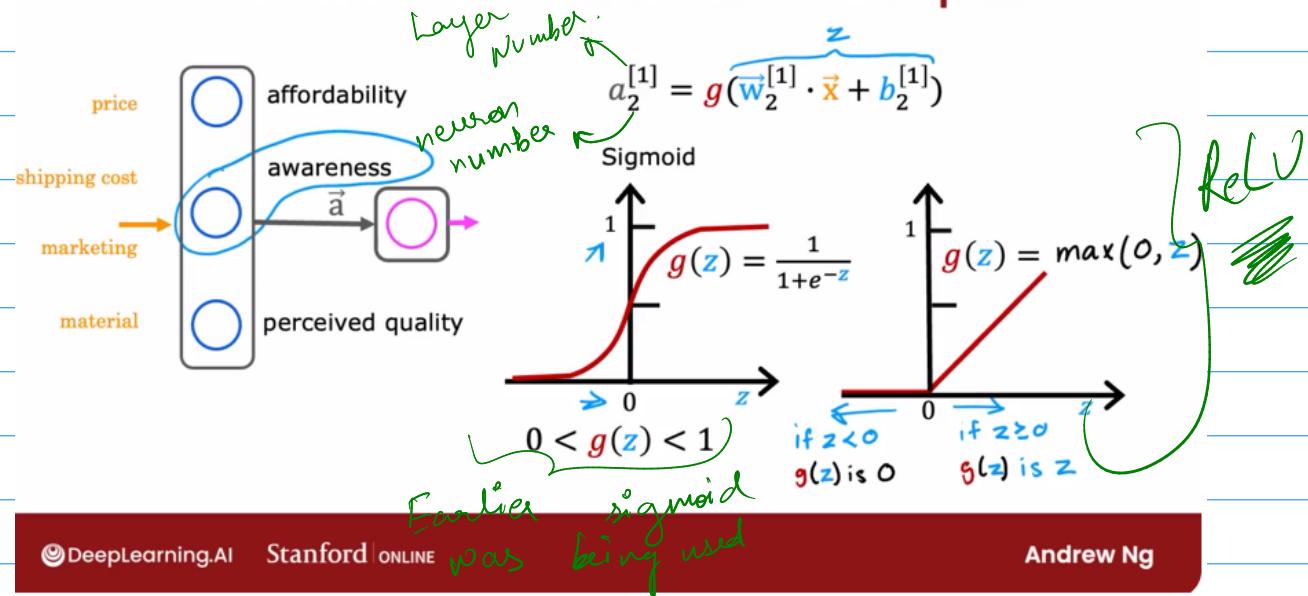
Step 2: $\text{model.compile}(\text{loss}=\dots)$

Step 3: $\text{model.fit}(X, Y, \text{epochs}=100)$

(This performs backprop in background)

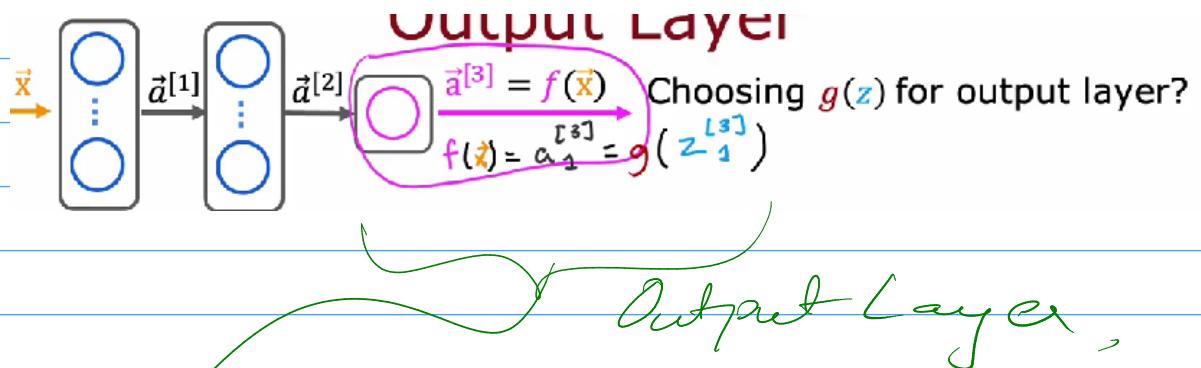
Alternatives to the Sigmoid Function.

Demand Prediction Example

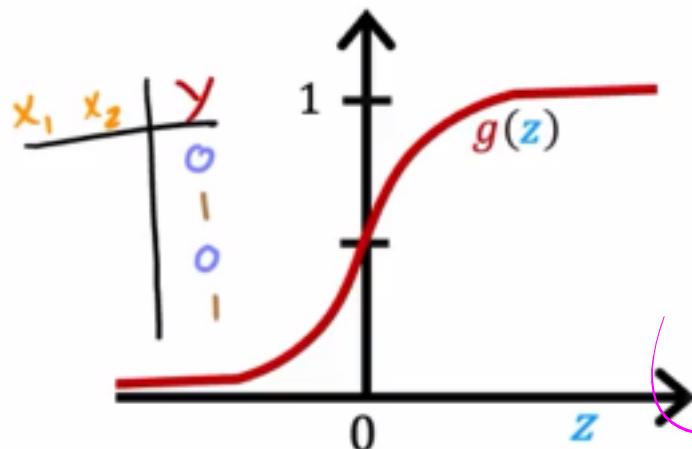


They are activation functions

Choosing an AF for Output Layer.



(Q1)
if its a Binary classification problem then
we will use Sigmoid AF.

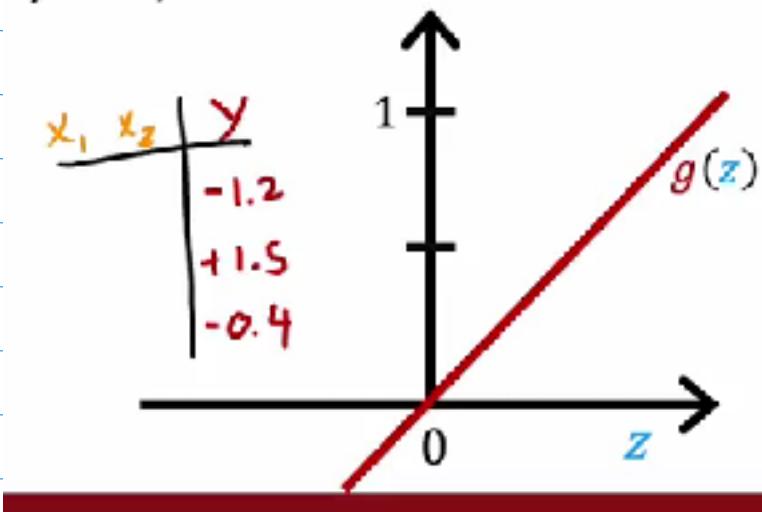


Sigmoid
Activation
Function,

If it's a regression task where the model has to predict certain value(s) then we would use Linear Activation func?

Regression

Linear activation function
 $y = +/-$



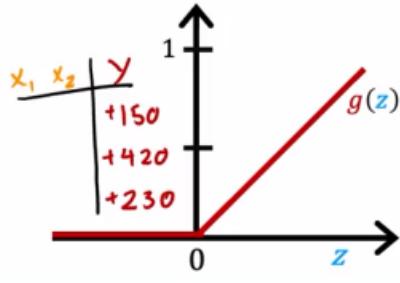
In regression if the predicted values could be +ve or -ve then we will use Linear AR.

Regression case 2: Where the predicted values is only positive or 0, like for example predicting price of house - Price of a house can't be -ve.

In such cases we use ReLU AF,

Regression

ReLU
 $y = 0 \text{ or } +$



$$\text{ReLU} = \max(0, z)$$

Activation function used in hidden layers.

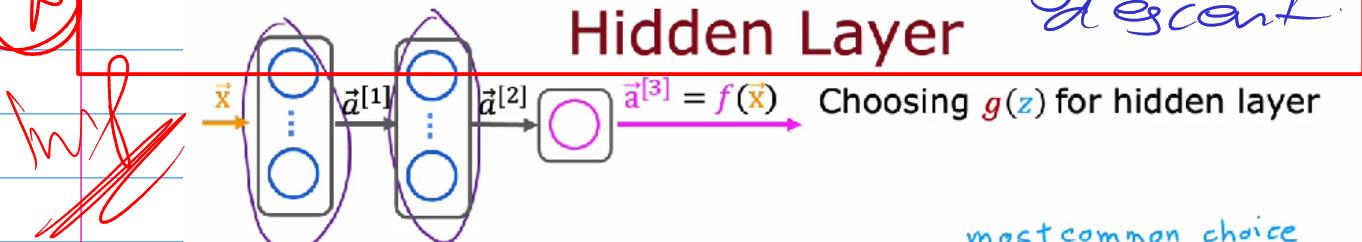
The most commonly used AF in hidden layers is **ReLU**.

2 reasons to choose ReLU over Sigmoid for hidden layers:-

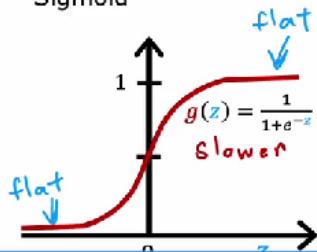
① It's faster.

② Sigmoid is flat on both ends while ReLU is only flat at one end.

more no. of flat == slower gradient descent

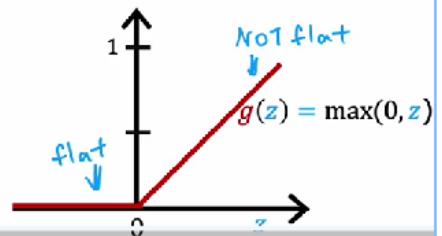


Sigmoid



most common choice

ReLU faster



4:39 / 8:24

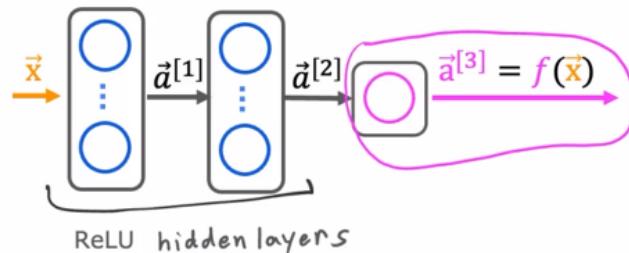


1.25x



Andrew Ng

Choosing Activation Summary



```
from tensorflow import keras
model = Sequential([
    Dense(units=25, activation='relu'), layer1
    Dense(units=15, activation='relu'), layer2
    Dense(units=1, activation='sigmoid') layer3
])
    or 'linear'
    or 'relu'
```

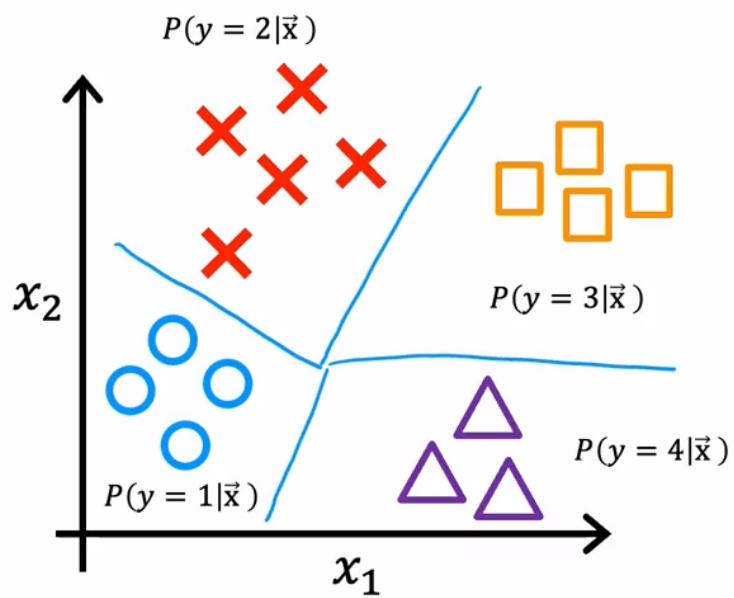
Annotations: layer1 is labeled 'hidden', layer2 is labeled 'hidden', and layer3 is labeled 'output'.

binary classification
activation='sigmoid'
regression y negative/
positive
activation='linear'
regression $y \geq 0$
activation='relu'

Note: Avoid using linear activation function in hidden layers, instead use ReLU.

Multi Class Classification

It is a classification problem where the target 'y' can take on more than 2 possible values



Softmax

Softmax regression (4 possible outputs) $y=1, 2, 3, 4$

$$\text{X } z_1 = \vec{w}_1 \cdot \vec{x} + b_1 \quad a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

X O \square Δ

$$= P(y=1|\vec{x}) \quad 0.30$$

$$\text{O } z_2 = \vec{w}_2 \cdot \vec{x} + b_2 \quad a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

$$= P(y=2|\vec{x}) \quad 0.20$$

$$\square z_3 = \vec{w}_3 \cdot \vec{x} + b_3 \quad a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

$$= P(y=3|\vec{x}) \quad 0.15$$

$$\Delta z_4 = \vec{w}_4 \cdot \vec{x} + b_4 \quad a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

$$= P(y=4|\vec{x}) \quad 0.35$$

]} Softmax
Formula.

Softmax Regression General Formula

Softmax regression
(N possible outputs) $y=1, 2, 3, \dots, N$

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

parameters w_1, w_2, \dots, w_N
 b_1, b_2, \dots, b_N

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

} Info

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

all probabilities sum up to 1.



Cost

Logistic regression

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

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

$$a_2 = \underbrace{1 - a_1}_{\text{blue bracket}} = P(y = 0 | \vec{x})$$

$$\text{loss} = -y \underbrace{\log a_1}_{\text{if } y=1} - (1-y) \underbrace{\log(1-a_1)}_{\text{if } y=0}$$

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

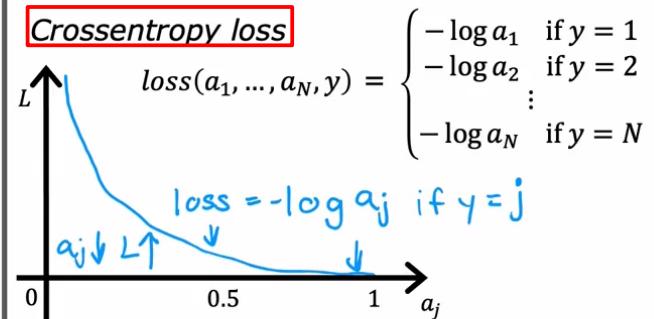
Softmax regression

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \dots + e^{z_N}} = P(y=1|\vec{x})$$

$$a_N = \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \dots + e^{z_N}} = P(y = N | \vec{x})$$

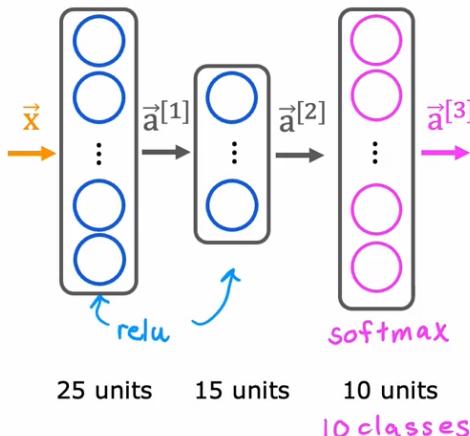
Crossentropy loss

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



Neural Network w/ Software Output -

Neural Network with Softmax output



$$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]} \quad a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \dots + e^{z_{10}^{[3]}}} = P(y=1|\vec{x})$$

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

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

gives

the prob
at the

of 99

any 1

there is

MNIST with softmax

① specify the model

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

② specify loss and cost

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

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

```
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')
])
from tensorflow.keras.losses import
    SparseCategoricalCrossentropy
model.compile(loss= SparseCategoricalCrossentropy())
model.fit(X, Y, epochs=100)
```

Note: better (recommended) version later.

Don't use the version shown here!

Sparse Categorical Cross Entropy

Fixed & sparse
no. of categories
to predict
like 0 to 9

These are more than 2
categories to predict

Improved implementation of Softmax -

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

$$\text{Loss} = L(\vec{a}, y) = \begin{cases} -\log \frac{e^{z_y}}{e^{z_1} + e^{z_2} + \dots + e^{z_{10}}} & \text{if } y=1 \\ -\log \frac{e^{z_{y+10}}}{e^{z_1} + e^{z_2} + \dots + e^{z_{10}}} & \text{if } y \neq 1 \end{cases}$$

model = Sequential ([

Dense(25, activation='relu'),

Dense(15, activation='relu'),

Dense(10, activation='linear')

])

model.compile(loss='sparse_categorical_crossentropy', from_logits=True)

This implementation is equivalent to using
Softmax in op layer. Why this? Cuz this is more numerically accurate.

Why not to use Softmax + Cross Entropy Log-

1. Usual Softmax + Cross-Entropy Loss

- Normally:

$$\text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

and cross-entropy loss:

$$L = -\log(\text{Softmax}(z)_y)$$

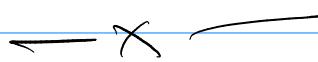
- This means you:

1. Compute softmax → probabilities.
2. Take log of the probability of the correct class.

2. Issue with Direct Softmax + Log

- If z_i is very large or very negative, e^{z_i} can overflow (∞) or underflow (0).
- Taking log of very small numbers → $-\infty$, causing numerical instability.

This is not recommended and not a good industry practice.



3. Numerically Stable Version

- Instead of:

$$-\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

- We compute directly:

✓

$$L = -z_y + \log\left(\sum_j e^{z_j}\right)$$

- And before exponentiating, we subtract the max logit to avoid overflow:

$$\log\left(\sum_j e^{z_j - \max z}\right) + \max z$$

- This is what libraries do internally when you set `from_logits=True`.

2nd approach: recommended

This doesn't cause numerical instability, underflow, overflow.

This is the good practice to follow.

What is logits?

What is `from_logits=True`?

- **Logits** = raw outputs of the last layer (no activation).
- If you already applied `Softmax` before passing to the loss function, you pass `from_logits=False`.
- If your final layer is `linear` (no softmax), you tell the loss function `from_logits=True`, and it internally:
 1. Applies softmax in a numerically stable way.
 2. Calculates cross-entropy in one step.

Summary

- **Yes, more accurate:** Because it avoids overflow/underflow by combining softmax + log in one step.
- `from_logits=True` : Means you are passing **raw scores (logits)**, and the library will handle softmax internally.

Improved and Correct Code

MNIST (more numerically accurate)

```
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.losses import
      SparseCategoricalCrossentropy
fit model.compile(..., loss=SparseCategoricalCrossentropy(from_logits=True)) ✓
predict logits = model(X) ← its not outputting  $a_1 \dots a_{10}$ 
                           instead its outputting  $z_1 \dots z_{10}$ 
f_x = tf.nn.softmax(logits)
```

model Sequential(layers) — ①

↓

model.compile(loss) — ②

↓

model.fit(X, Y, epochs) — ③

↓

logits = model(X) — ④

↓

f_x = tf.nn.softmax(logits), — ⑤

A

Multi Class v/s Multi Label Classification -

Multi Class: Only one output for a given i/p. Eg: Detect a digit.

Multi Label: Multiple outputs for a given i/p -

Eg: Detect a Car, Bus, Person in a single image.



Imp:-

```
def softmax(z):
    ez = np.exp(z)
    sm = ez / np.sum(ez)
    return(sm)
```

The loss function associated with Softmax, the cross-entropy loss, is:

$$\text{loss} = -\log(a_{\text{true-label}})$$

$$L(\mathbf{a}, y) = \begin{cases} -\log(a_1), & \text{if } y = 1. \\ \vdots \\ -\log(a_N), & \text{if } y = N \end{cases} \quad (3)$$

Where y is the target category for this example and \mathbf{a} is the output of a softmax function. In particular, the values in \mathbf{a} are probabilities that sum to one.

Recall: In this course, Loss is for one example while Cost covers all examples.

Note in (3) above, only the line that corresponds to the target contributes to the loss, other lines are zero. To write the cost equation we need an 'indicator function' that will be 1 when the index matches the target and zero otherwise.

$$1\{y == n\} = \begin{cases} 1, & \text{if } y == n. \\ 0, & \text{otherwise.} \end{cases}$$

Now the cost is:

$$J(\mathbf{w}, b) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{j=1}^N 1\{y^{(i)} == j\} \log \frac{e^{z_j^{(i)}}}{\sum_{k=1}^N e^{z_k^{(i)}}} \right] \quad (4)$$

Where m is the number of examples, N is the number of outputs. This is the average of all the losses.

Indicator ka role

- Har example ke liye ek **one-hot** indicator hota hai:
 - Example 1 → [1, 0, 0]
 - Example 2 → [0, 1, 0]
 - Example 3 → [0, 0, 1]
- Formula mein:

$$\sum_j 1\{y = i\} \log(a_j) \rightarrow \text{sirf true label pick hota hai.}$$

Bottom line:

- Softmax se probability nikal
- True label ki probability lo
- $-\log(\text{true prob})$ ka average le lo

Key Takeaway -

Advanced Optimization

Adam: Adaptive Moment Estimation

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

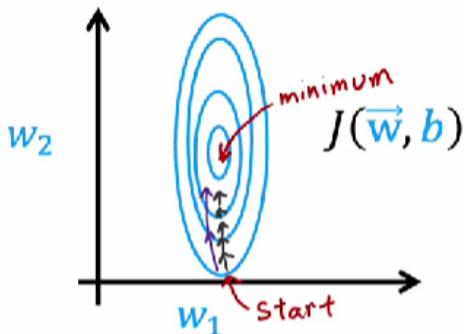
⋮

$$w_{10} = w_{10} - \alpha_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$

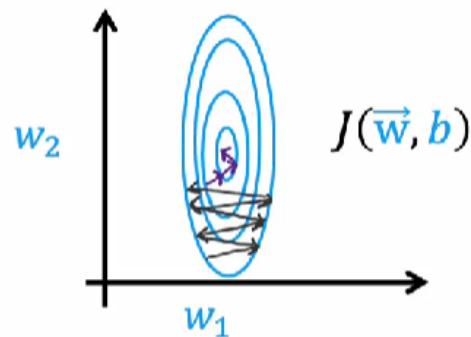
$$b = b - \alpha_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$

if there
are 11 params
including b
then there
would be
11 α 's -

Adam Algorithm Intuition



If w_j (or b) keeps moving
in same direction,
increase α_j .



If w_j (or b) keeps oscillating,
reduce α_j .

MNIST Adam

MNIST Adam

model

```
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

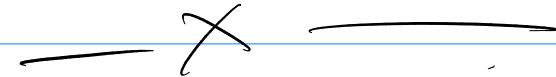
```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

fit

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

$\alpha = 10^{-3}$
initial learning rate

Adam is works than traditional Gradient Descent algorithm and is the defacto standard on how practitioners train their neural networks!

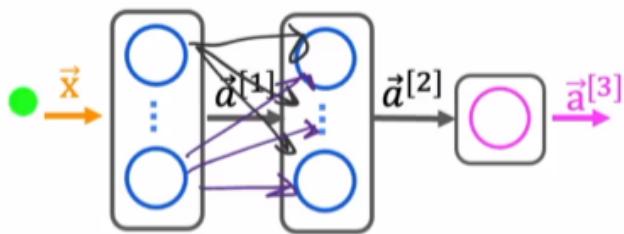


Additional Layer Types -

RECAP

1. Dense -

Dense Layer



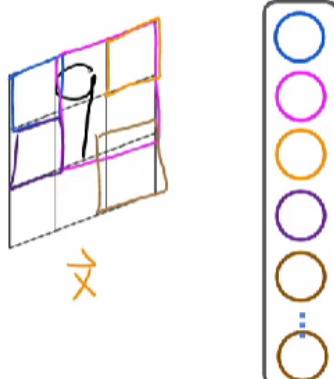
Each neuron output is a function of all the activation outputs of the previous layer.

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

New Layers

→ Convolutional Layer

Convolutional Layer



Each neuron only looks at part of the previous layer's outputs.

Why?

- Faster computation
- Need less training data
(less prone to overfitting)

Conv Neural Network

What Conv Layer?

A layer that only looks at a part of the data and not the whole data

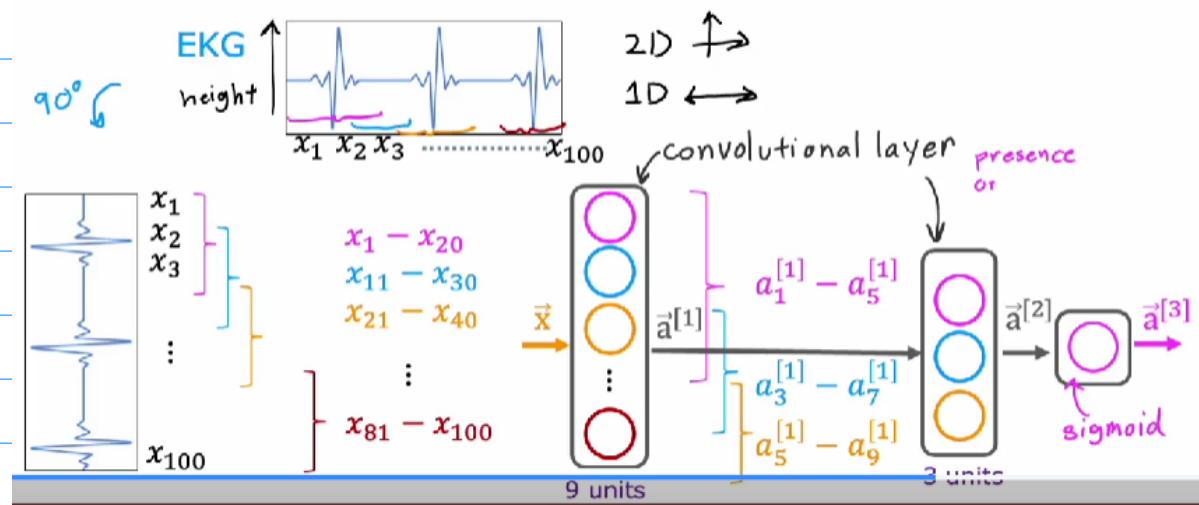
{ Combine Conv. layers.

You get a neural network, which is faster than Dense Neural Network,

which consists of layers each that look at a part of data and then collectively look at the whole

data. This is called a Convolutional Neural Network.

Convolutional Neural Network



CNN's are good for spatial data recognition.

Backpropagation.

$$J(w) = w^2$$

$$= 9$$

let $w = 3$

If $w \uparrow$ by E causes $J(w) \uparrow$ by $K \neq E$

then $\frac{\partial J(w)}{\partial w} = K$

$$\frac{\partial}{\partial w}(w^2) = 2w$$

More Derivative Examples

$w = 3$

$J(w) = w^2 = 9$

$w \uparrow 0.001$

$J(w) = J(3.001) = 9.006001$

$\frac{\partial}{\partial w} J(w) = 6$

$J(w) \uparrow 6 \times 0.001$

$w = 2$

$J(w) = w^2 = 4$

$w \uparrow 0.001$

$J(w) = J(2.001) = 4.004001$

$\frac{\partial}{\partial w} J(w) = 4$

$J(w) \uparrow 4 \times 0.001$

$\downarrow 0.006$

$w = -3$

$J(w) = w^2 = 9$

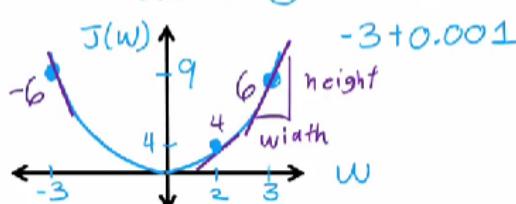
$w \uparrow 0.001$

$J(w) = J(-2.99) = 8.994001$

$\frac{\partial}{\partial w} J(w) = -6$

$J(w) \downarrow 6 \times 0.001$

$J(w) \uparrow -6 \times 0.001$



Calculus	w	$\frac{\partial J(w)}{\partial w}$
$\frac{\partial}{\partial w} J(w) = 2w$	3	$2 \times 3 = 6$
	2	$2 \times 2 = 4$
	-3	$2 \times -3 = -6$



DeepLearning.AI



12:46 / 22:56



1.25x Andrew Ng



A note on derivative notation

If $J(w)$ is a function of one variable (w),

$$d \quad \frac{d}{dw} J(w)$$

If $J(w_1, w_2, \dots, w_n)$ is a function of more than one variable,

$$\partial \quad \frac{\partial}{\partial w_i} J(w_1, w_2, \dots, w_n)$$

Computation Graph

$$\vec{x} \rightarrow o \xrightarrow{a}$$

Let $w=2, b=8$
 $x=-2, y=2$

$$a = wa + b$$
$$J(w, b) = \frac{1}{2}(a - y)^2$$

$$w \xrightarrow{2} c = wn \xrightarrow{-y} q = c + b \xrightarrow{y} d = ay \xrightarrow{z} J = \frac{1}{2}d^2 \xrightarrow{?} J$$



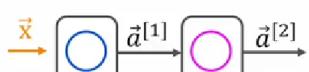
Forward propagation.

Neural Network Example

$$x = 1 \quad y = 5$$

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

ReLU activation



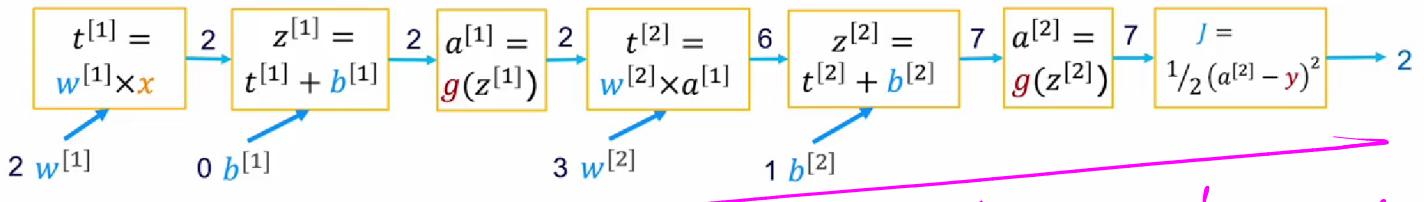
$$w^{[2]} = 3, b^{[2]} = 1$$

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

$$a^{[1]} = g(w^{[1]} x + b^{[1]}) = \underbrace{w^{[1]} x}_{z^{[1]}} + \underbrace{b^{[1]}}_{z^{[1]}} = 2 \times 1 + 0 = 2$$

$$a^{[2]} = g(w^{[2]} a^{[1]} + b^{[2]}) = \underbrace{w^{[2]} a^{[1]}}_{z^{[2]}} + \underbrace{b^{[2]}}_{z^{[2]}} = 3 \times 2 + 1 = 7$$

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



Computation Graph for forward propagation

Week 3

This week we will focus on -

Machine learning diagnostic

Diagnostic:

A test that you run to gain insight into what is/isn't working with a learning algorithm, to gain guidance into improving its performance.

Diagnostics can take time to implement
but doing so can be a very good use of your time.

$m_{train} \rightarrow$ no. of training examples [split dataset into Train & Test]

$m_{test} \rightarrow$ no. of test examples

Train/Test procedure for linear Regression

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

(cost)

m_{train} → no. of training example

pred. → predicted value
actual → actual value

regularization term → regularization term

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

Compute test error:

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

Predicted o/p

True o/p

Compute training error:

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

Cost function

Train/test procedure for classification problem

0 / 1

Fit parameters by minimizing $J(\vec{w}, b)$ to find \vec{w}, b

E.g.,

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

main
cost
loss
mtrain

Compute test error:

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

loss

reg

Compute train error:

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

Test cost
Train cost

A better way is calculate the fraction of test and train set that has been misclassified.

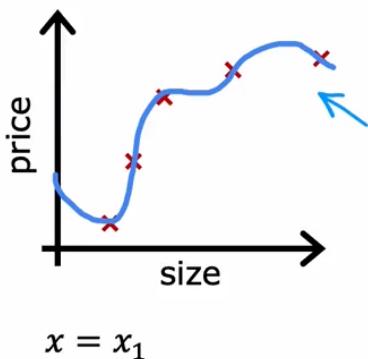
J_{test} → Fraction of test set that's misclassified.

J_{train} → Fraction of train set that's misclassified.

Evaluating & choosing models

- Model Selection & training/cross validation
/ test sets

Model selection (choosing a model)



Once parameters \vec{w}, b are fit to the training set, the training error $J_{\text{train}}(\vec{w}, b)$ is likely lower than the actual generalization error.

$J_{\text{test}}(\vec{w}, b)$ is better estimate of how well the model will generalize to new data compared to $J_{\text{train}}(\vec{w}, b)$.

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

Train Set vs Validation Set vs Test Set

"Final judgement ke liye use karo" ka matlab:

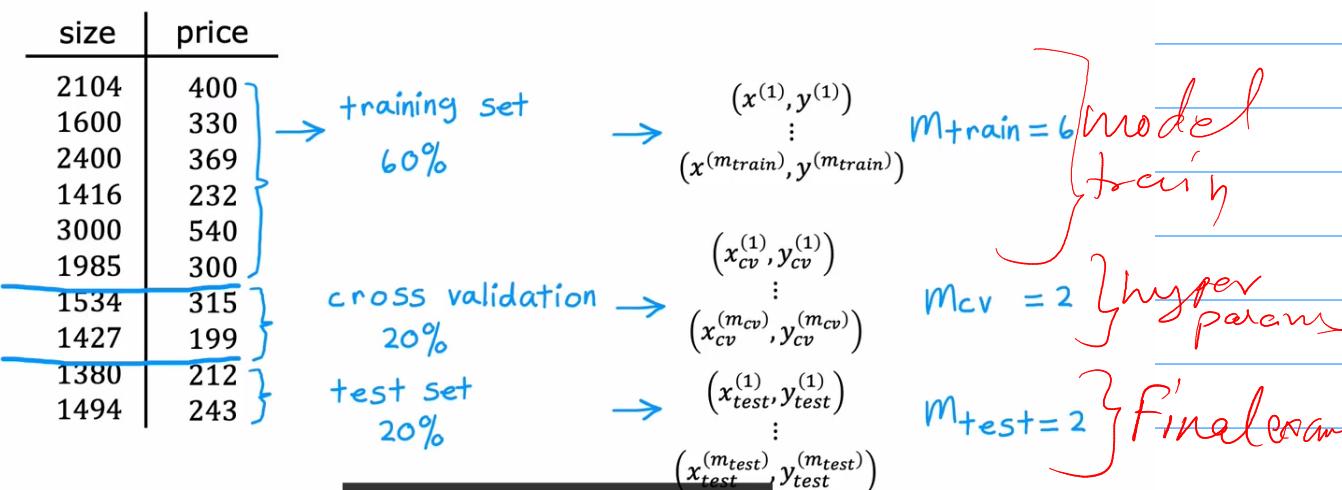
- **Training set:** Ispe tum model ko train karte ho (weights w , b learn karte ho).
- **Validation set:** Ispe tum **hyperparameters** (jaise polynomial degree d , learning rate, regularization strength, etc.) choose karte ho. Matlab model kis shape/complexity ka hoga yeh decide karte ho.
- **Test set:** Yeh ek dum alag rakha jata hai. Ispe **kuch bhi tune nahi karna**.
- Jab tum final model select kar chuke (hyperparameters fix ho gaye), tab test set ek baar run karke real-world performance estimate karte ho.

Matlab:

- Test set ek **final exam** hai.
- Agar tumne test set ka use **practice karne** (hyperparameter choose karne) ke liye bhi kar diya, toh exam ka answer tumhe pehle se pata hoga → aur tumhe lagega ki tum topper ho, but **real life** (new data) me utna score nahi aayega.

We will split our data into 3 parts -

Training/cross validation/test set



Training/cross validation/test set

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

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

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

Procedure

Train the model on



Model Selection \rightarrow Validation Set J_{cv}

Est. generalization error using test set J_{test} \rightarrow Final Exam

When to use what dataset:

Stage-wise use of datasets

1. Stage 1 → Model ko train karna (weights seekhna)

- **Dataset:** Training set
- **Purpose:** Model ke parameters (weights, biases) ko optimize karna.
- **Output:** Ek trained model (given architecture & hyperparameters ke saath).
- **Kya nahi karte:** architecture decide nahi karte is stage pe — sirf ek architecture ke liye train karte ho.

2. Stage 2 → Model design choose karna (architecture & hyperparameters tune karna)

- **Dataset:** Validation (or Cross-validation) set
- **Purpose:**
 - Compare karo: kaunsa **architecture** (e.g., layers/neurons) better perform karta hai?
 - Kaunsa **hyperparameter** (learning rate, regularization) best hai?
- **Why validation and not training?**
 - Training error sirf yeh batata hai ki model ne training data kitna yaad kiya.
 - Validation error batata hai ki unseen data pe model generalize kar raha ya nahi.
- **Modern ML (industry):** Architecture bhi mostly validation pe decide karte hain.

3. Stage 3 → Final unbiased performance check

- **Dataset:** Test set
- **Purpose:** Jab architecture + hyperparameters fix ho gaye, tab **ek baar** use karna to check **true unseen performance**.
- **Why single use?** Kyunki agar repeatedly test pe tune karte ho, woh bhi training jaise ho jayega → biased result.



Quick diagram idea

kotlin

Copy Edit

```
Raw data
|
Split ----> Training set (weights seekhna)
|
---> Validation set (architecture & hyperparameter tuning)
|
---> Test set (final unbiased check)
```

Stage Mapping in Code

1. Stage 1 (Training weights):

```
python  
  
model.fit(x_train, y_train)
```

 Copy  Edit

→ Weights learn karte hain.

2. Stage 2 (Architecture choice):

```
python  
  
val_loss, val_acc = model.evaluate(x_val, y_val)
```

 Copy  Edit

→ Kis architecture ka generalization best hai, yeh validation set se pata chalta hai.

3. Stage 3 (Final unbiased check):

```
python  
  
test_loss, test_acc = best_model.evaluate(x_test, y_test)
```

 Copy  Edit

→ Sirf ek baar final check ke liye.

Bias and Variance

Underfit Model

- It has high bias and is inaccurate
- $J_{\text{train}} \& J_{\text{cv}}$ is high.

If a model has significantly high bias & high variance, its the characteristic of an underfit model.

Overfit Model

- It has high variance and is highly accurate on training data but performs poorly on CV data & test data.
 - J_{train} is low but J_{cv} is significantly high in comparison to J_{train} .
-) This is the characteristic of an overfit model.

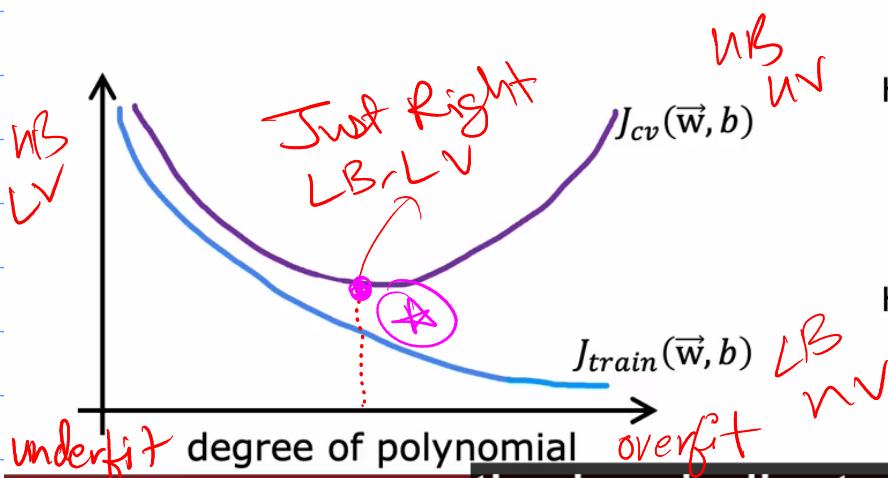
Just right

Both J_{train} & J_{cv} is low - Meaning the model is well fit and performs well.

$J_{\text{train}} \downarrow, J_{\text{cv}} \downarrow \rightarrow$ Characteristic of good model.

Diagnosing bias and variance

How do you 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 and high variance

J_{train} will be high
and $J_{cv} \gg J_{train}$

DeepLearning.AI

Stanford | ONLINE

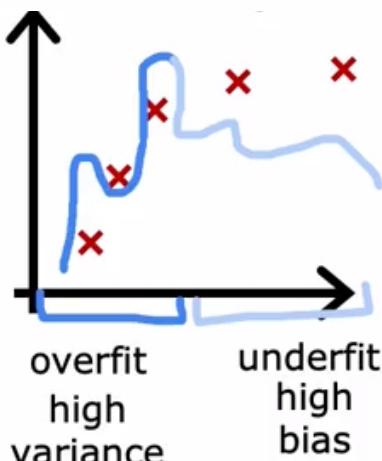
Andrew Ng

the key indicator will
be if J_{train} is high.

In what case do we get WB & UV?

When the model fits a part of the training data very well (Overfits) and doesn't fit the other half of the training set (Underfit) in such cases we get WB, UV

getting WB, UV meaning your model is quite bad.

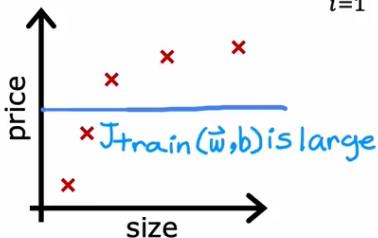


Regularization & Bias/Variance

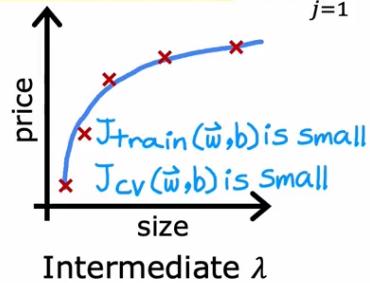
Linear regression with regularization

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

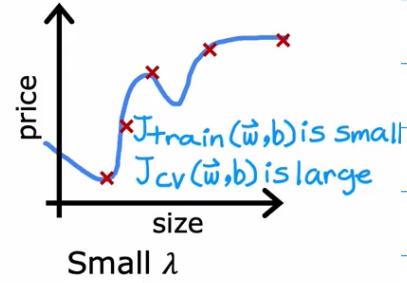
$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^m (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n w_j^2$$



Large λ
High bias (underfit)
 $\lambda = 10,000$ $w_1 \approx 0, w_2 \approx 0$
 $f_{\vec{w}, b}(\vec{x}) \approx b$



Intermediate λ



Small λ
High variance (overfit)
 $\lambda = 0$

DeepLearning.AI

Stanford ONLINE

Let's take a look at
how we could do so.

Andrew Ng

How to choose the correct value of λ ?

Let's say model is as follows:-

$$f(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$$

Try the following values of λ :-

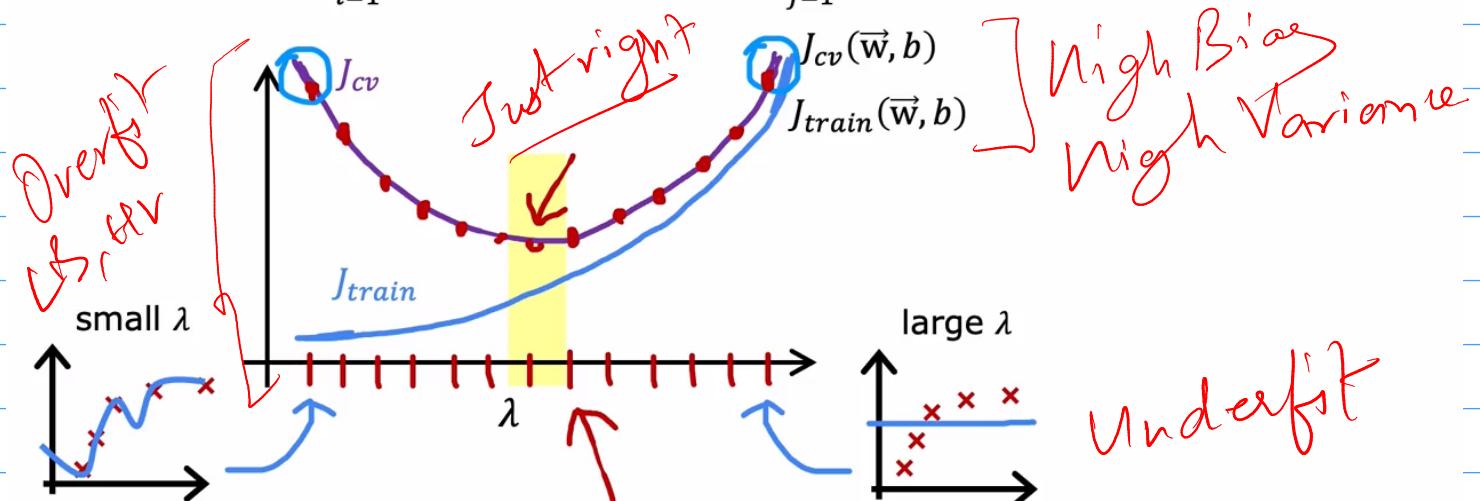
$$0, 0.01, 0.02, 0.04, 0.08, \dots, 10$$

Calculate the J_{λ} for each λ to figure out the suitable λ for your model.

pick λ with low $J_{\lambda}(w, b)$

Bias and variance as a function of regularization parameter λ

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



Establishing a Baseline level of Performance

By setting a Baseline we can judge if the learning algorithm is under or over by looking at J_{train} & J_{cv}

Speech recognition example



Human level performance

Training error J_{train}

Cross validation error J_{cv}



higher than a human level of performance,

Human level performance is 10.6%
& machine is 10.8%. This means
quite accurate since if human is
at 10.6%, then you can't expect
machine to be less or than
human error right.
Human error is just 2%
In this example more than
human performance

TLDR when we see Train error we have to benchmark it against human level performance to decide if model is LB or HB

In above eg $J_{train} = 10.4\%$ & $J_{cv} = 14.8\%$

$\leftarrow 4\% \rightarrow$
difference →

This 4% difference is quite high & this is inclined toward a variance problem.

Establishing a baseline level of performance

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

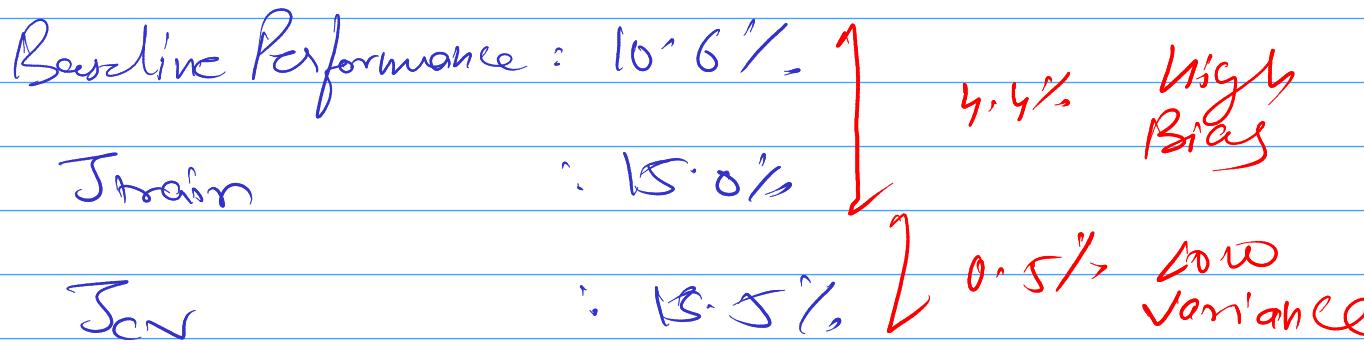
- • Human level performance
- • Competing algorithms performance
- • Guess based on experience

Bias/variance examples

Baseline performance	: 10.6%	$\downarrow 0.2\%$	Low Bias
Training error (J_{train})	: 10.8%	$\downarrow 4.0\%$	High Variance
Cross validation error (J_{cv})	: 14.8%		

Hence this is an overfitted model

lets say:



Then this is an underfitted model.

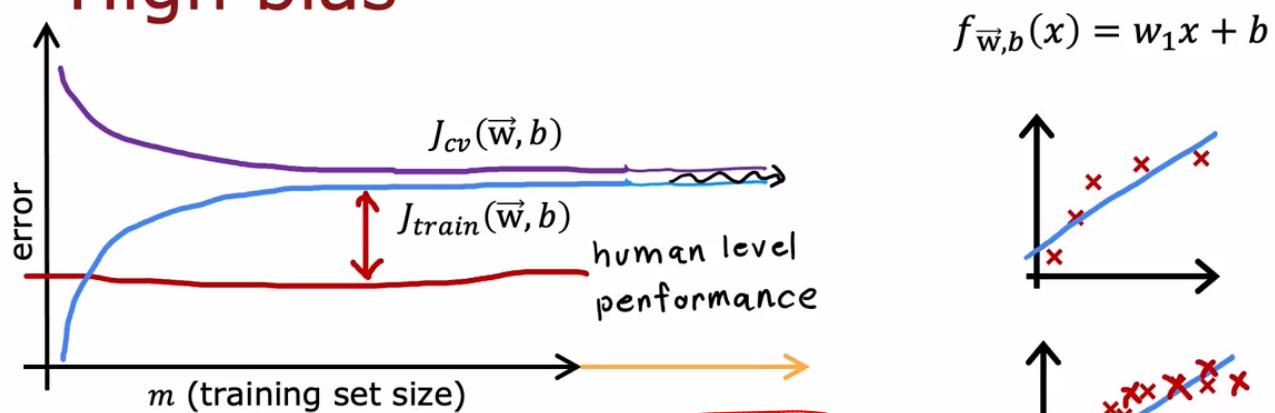


Always compare to baseline performance

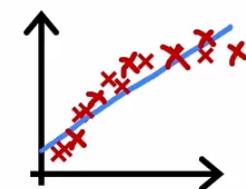
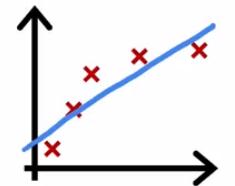
Learning Curve

Learning Curve tells us about one learning algo's as of what experience it has with the training examples.

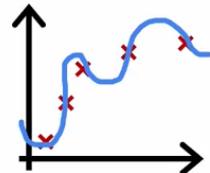
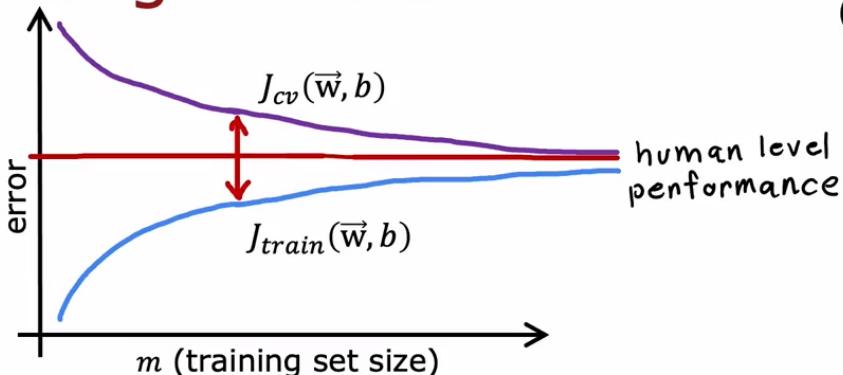
High bias



if a learning algorithm suffers from high bias, getting more training data will not (by itself) help much.



High variance



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

Debugging a learning algorithm

Debugging a learning algorithm

You've implemented regularized linear regression on housing prices

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

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

- Get more training examples fixes high variance
- Try smaller sets of features $x, x^2, \cancel{x^3}, \cancel{x^4}, \cancel{x^5}, \dots$ fixes high variance
- Try getting additional features fixes high bias
- Try adding polynomial features $(x_1^2, x_2^2, x_1 x_2, \text{etc})$ fixes high bias
- Try decreasing λ fixes high bias
- Try increasing λ fixes high variance

If you find your algo has high variance then primarily you should get more training examples or dial down the complexity of the model (By using lesser features or increase λ)

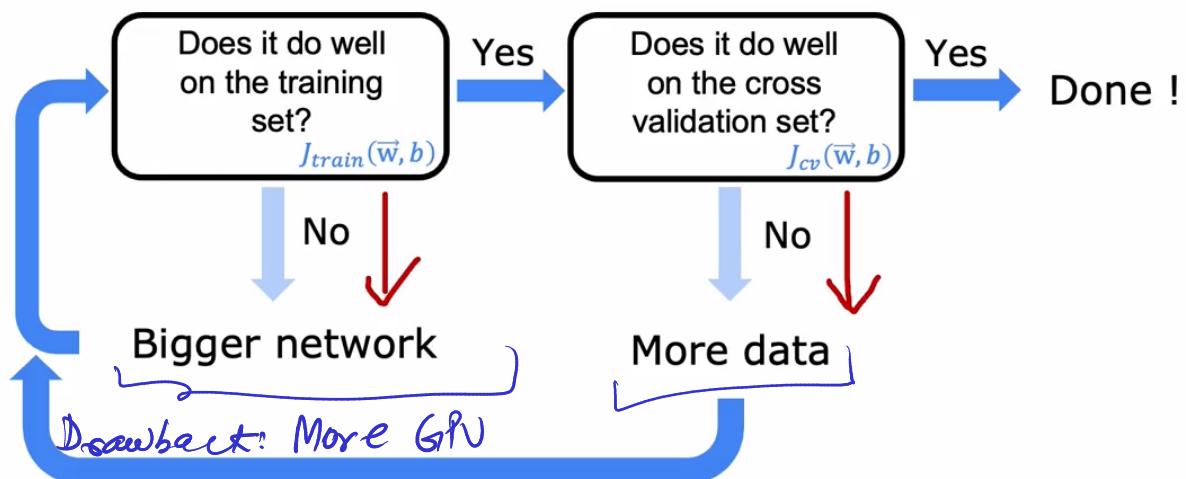
P If you find that your algorithm isn't even doing good on the training set, its NB, the primarily you should dial up the complexity (Add more features or decrease λ).

Bias/Variance of Neural Networks

Large Neural Networks trained on medium to large datasets are low bias machines -

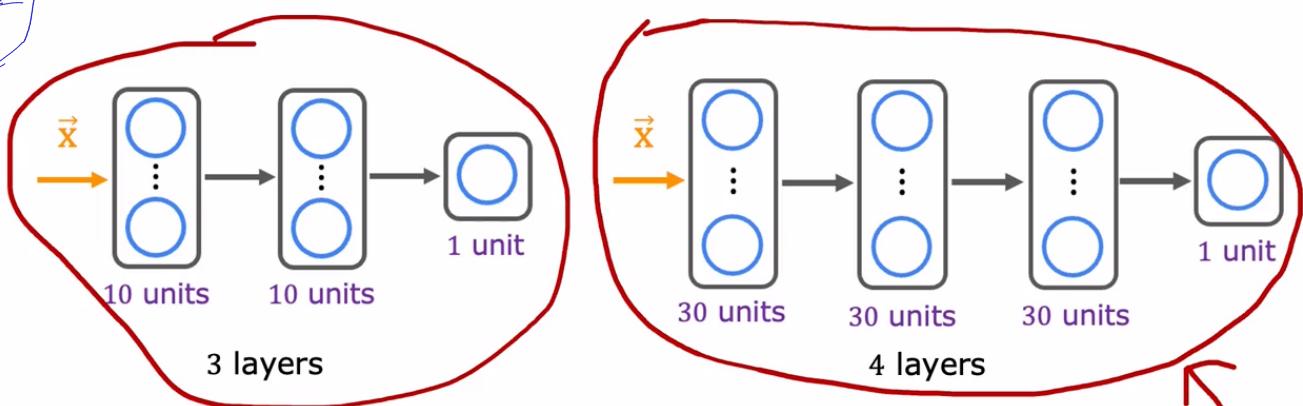
Neural networks and bias variance

Large neural networks are low bias machines



validation set and hopefully will also generalize to new examples as well.

Neural networks and regularization



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

So long as the regularization has chosen appropriately.

Tldr: Choose a large NN with an appropriate regularization term.

Lasso (L1) vs Ridge (L2) v/s Elastic Net (L1 + L2)

1) Use Lasso (L1)

When:

- High-dimensional data (features >> samples) → e.g. genomics, text, one-hot categorical encoding
- Want feature selection automatically → kuch features irrelevant hai, unko model khud 0 kar dega
- Interpretability important hai → chhota sparse model chahiye jo clearly bole "ye features important hain"
- Storage ya latency constraint → sparse weights = lightweight model

Tradeoff:

- Jab correlated features hain, Lasso randomly ek rakhega aur dusre ko zero karega (unstable selection).

2) Use Ridge (L2)

When:

- Almost all features relevant hain → aur unka magnitude control karna hai, par drop nahi karna
- Multicollinearity issue hai (features correlated hain) → Ridge sabko thoda shrink karega instead of dropping
- Deep Learning & Neural Nets → Ridge smoothness deta hai, optimization easier hai, isliye ye default hai
- Numeric stability chahiye (overfitting ka risk high hai, but interpretability important nahi hai)

Tradeoff:

- Sparse nahi banata → har feature ka contribution rahega.

3) Elastic Net (L1 + L2 mix)

When:

- Feature selection bhi chahiye AND correlated features ko saath handle karna hai
- Kaggle ke competitions ya applied ML me ye common hai because:
 - L1 → sparse bana deta hai
 - L2 → correlated features me stability deta hai

Summary Shortcut

- Feature selection important? → Lasso (L1)
- Sab feature rakho but control karo? → Ridge (L2)
↓
- Both chahiye → Elastic Net

1) Lasso (L1) – Push to 0

Real-life scenarios:

1. Finance (Credit Scoring)

- Tumhare paas 200 variables hain (income, spending habits, location, 100 dummy features, etc.).
- Tum chahte ho ki **kaunse features loan default predict karne me sabse zyada important hai**.
- **Lasso** automatically irrelevant features (e.g., "favourite Netflix genre") ko **0** kar dega.
- Output: sirf top 10–15 impactful variables bache → explain karna easy.

2. Healthcare (Disease Prediction)

- Genomic data me 20,000+ gene expression features.
- Tum chahte ho ki **kuch hi genes select ho jo cancer risk ke liye critical hain**.
- Lasso → sparse output deta hai → interpret karna easy.

3. IoT / Sensor Data

- 300 sensors ka data leke machine failure predict karna.
- Lasso automatically useless sensors ignore karega → cheaper sensor system ban sakta hai.

2) Ridge (L2) – Near to 0

Real-life scenarios:

1. Image Classification (Deep Learning)

- CNN me 1 million parameters.
- Tum chahte ho ki **sab features contribute kare, par overfitting na ho**.
- Ridge (L2) = weights ko shrink karega par kisi ko zero nahi karega → deep learning me default hai.

2. Marketing Sales Prediction

- Tumhare paas 100 related marketing channels (TV, FB, Insta, etc.) ke spends ka data hai.
- Sare correlated hain, tum kisi ko drop nahi kar sakte (sab important hai).
- Ridge use karo → sabka thoda impact rakhta hai, par magnitude control kar ke stable predictions deta hai.

3. Weather Forecasting

- Features like temperature, humidity, pressure, wind, etc. – sab naturally important hain.
- Ridge ensures koi bhi feature hataya na jaye, bas unka effect smooth ho.

3) Elastic Net (L1 + L2)

Real-life scenarios:

1. E-commerce Recommendations

- Bahut saare user features aur product features (kuch irrelevant bhi).
- Elastic Net → irrelevant features drop karega (L1) and correlated features ko saath handle karega (L2).

2. Stock Price Prediction (High Dimensional)

- 500+ technical indicators, mostly correlated.
- Elastic Net feature selection + stability dono deta hai → robust model.

Quick Mental Model

- **Lasso** = jab tumhe "kaunse factors important hain" jaan'na hai (interpretability).
- **Ridge** = jab tumhe **sab features use karna hai** but stable prediction chahiye.
- **Elastic Net** = jab tumhe **dono ka mix chahiye** (real-world tabular data me common).

Regularized MNIST Model

Neural network regularization

$$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m} \sum_{i=1}^m L(f(\vec{x}^{(i)}), y^{(i)}) + \frac{\lambda}{2m} \sum_{\text{all weights } \mathbf{W}} (\mathbf{w}^2)$$

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])
```



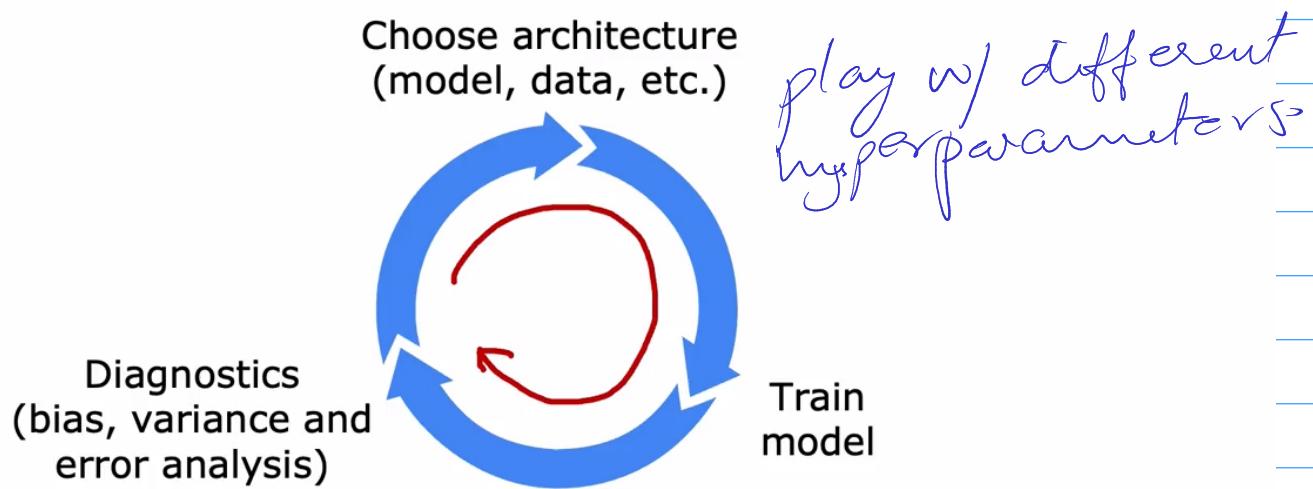
Then the regularization term for a neural network looks like pretty much what

Andrew Ng

Machine Learning development process -

Iterative loop of ML Development -

Iterative loop of ML development



In this lesson we will learn about error analysis which has a second key set of ideas for gaining insights about what architecture choices might be fruitful.

Error handling is one of the important ways to run error diagnostics.

Error Handling.

Error handling is the process of looking at your model's mistakes, figuring out what happened and using that information to improve the model.

Why Error Analysis?

- To understand where your ML model is making mistakes.
- Helps decide what to improve (data, model, features, etc.).
- Data-driven approach → avoid random guessing.

Steps in Error Analysis

1. Collect Misclassified Examples
 - Take a random sample of errors (e.g., 100–1000 misclassified points).
 - Look for patterns.
2. Manually Examine Errors
 - Check what kind of errors are common.
 - Categorize them (e.g., blurry image, wrong label, unusual angle, etc.).
3. Quantify Error Types
 - Example:
 - 43% = bad lighting
 - 29% = incorrect labels
 - 15% = unusual objects
 - This tells where to focus.
4. Decide Priorities
 - If 50% errors are due to mislabeled data → fix labeling.
 - If 40% due to unusual backgrounds → collect more such data.

Important Tips

- Look at actual errors, not overall accuracy.
- Use % of error categories to decide next steps (data cleaning, data augmentation, model architecture).
- Revisit error analysis regularly when model improves.

Outcome?

→ Clear actionable insights.

→ Faster model improvements with less random trial & error.

Error Analysis in a nutshell?

Look at where your ML model is going wrong - find pattern in mistakes, and use that info to decide what to fix first

Collect some of the wrong predictions



Manually check why they went wrong



Group and count error types



Prioritize fixes based on which error type happens the most.

Bug Analysis = MANUALLY checking your model's wrong predictions

Adding data

Add more data of everything. E.g., "Honeypot" project.

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

Pharma spam

E.g., Go to unlabeled data and find more examples of Pharma related spam.

Beyond getting brand new training examples (x, y), another technique: Data augmentation

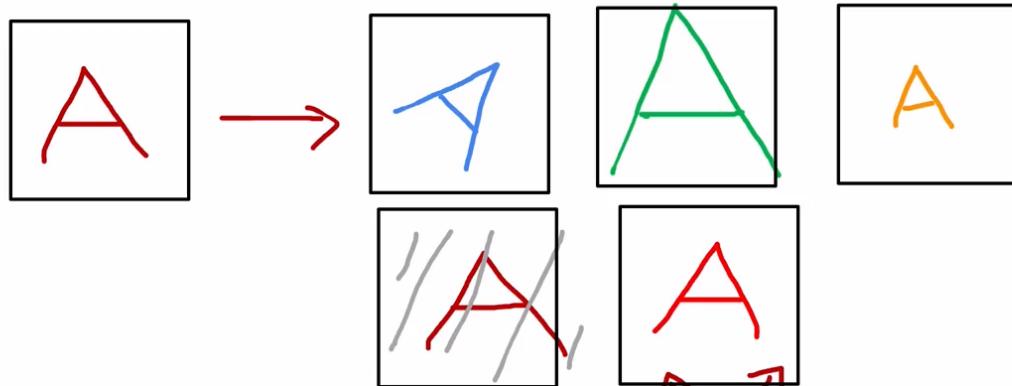
DeepLearning.AI

audio data that can increase your
training set size significantly.

Andrew Ng

Data augmentation

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



DeepLearning.AI

these would be ways of taking
a training example X, Y.

Andrew Ng

Data augmentation for speech

Speech recognition example

- 🔊 Original audio (voice search: "What is today's weather?")
- 🔊 + Noisy background: Crowd
- 🔊 + Noisy background: Car
- 🔊 + Audio on bad cellphone connection

DeepLearning.AI STANFORD ONLINE

Andrew Ng

examples here, one with crowd background
noise, one with car background noise and

Transfer Learning

Transfer Learning is when you take a model that has already learnt something useful in one task and then reuse it to help with another task — usually when you don't have much data for your own task.

💡 Nutshell Definition:

Transfer Learning is the technique of using a pre-trained model (on a large dataset) as a starting point for a new but related task, especially when you have limited data.

Apne kaam ke liye kam data hai ?

↓
Solution

Dusre task pe trained model ultra aur apne kaam ke liye sweat kar.

💡 Core Idea:

1. Train a model on a **BIG** dataset (like 1 million images of cats, dogs, cars, etc.)
2. Use the learned **layers** (especially earlier ones) as a starting point for a new task (like recognizing handwritten digits 0-9).
3. Just change the **final output layer** and **fine-tune** the rest — that's transfer learning.

Solution: TRANSFER LEARNING

Step 1 : Use a Pre-trained Model

Someone trained a deep neural net on 1 million images with 1000 classes (like cats, cars, etc.).

That model learned:

- Layer 1 → Detect edges
- Layer 2 → Corners
- Layer 3 → Shapes
- Layer 4 → Combinations of shapes
- Layer 5 → Final classification to 1000 classes

These early layers learn generic features that are useful for ANY image task (even digit recognition).

Step 2 : Copy Layers (Except Final One)

You copy:

- w^1, b^1
- w^2, b^2
- w^3, b^3
- w^4, b^4

But throw away the final layer (w^5, b^5) and replace it with a new output layer of 10 units for digits 0–9.

🛠 Two Fine-Tuning Options:

◆ Option 1: Freeze base, train only final layer

- Early layers (w^1 to w^4) → Keep fixed
- Train only new layer (w^5, b^5)
- Good when you have very little data

(Prevents overfitting, since fewer parameters are trained)

💡 But Why Does This Even Work?

Bro, cats and digits are totally different! 😱

Valid point. But...

- The **early layers** of CNNs don't care about the *specific object*
- They just learn to detect **edges, textures, patterns**
- These low-level features are **common** across many visual tasks
- So, when you re-use these layers, you're starting from a **smart place** instead of random weights

🔍 Terminology:

Term	Meaning
Pre-training	Training on big dataset (like ImageNet)
Fine-tuning	Adjusting the model on your small dataset
Transfer Learning	The whole process: reusing & tuning

okay sure
so lets imagine you are building an image classification algo
but ohhhh nooooooo, you dont have much training examples

here comes TL to the rescueeeeeeeeeeee 😊😊😊😊

you: ehhh, TL whats that, and how would that help meeee???

well TL is a technique where you can use a pre-trained model,
a model who is trained on extensive amount of similar kind of data and hence has its
foundation layers solid for lets say image classification in your case.

you: but how would the model know about my images???

well you directly dont copy paste the algo and model. you take its foundational
or early layers, the foundational layers of the algo all learn the similar things
or properties of a general image like edges, corners, texture, basic shapes and its the
later layers that hand pick the objects. so you can use the foundation layers of a
pre trained model and add layers of your model that fine tune it to your dataset and tweak
the foundational layers if you want to further double down and improve its performance

you: ohhhhhh wooooooooo rlyyyyyyyyyy?? 😳😳😳

yessssssirrr, just take the heavy work already done by a pre trained model
and attach your work to it and tweak, fine-tune and polish it according to your needs
and boom you have achieved what you wanted without you doing much heavy lifting

NOTE! The input types must be same for both

the networks for transfer learning to work

So:

- If your model is pre-trained on **images**, it expects **images** again
- If it's pre-trained on **text**, it only works for **text-based tasks**
- You **can't** use an image-trained model for audio or vice versa (unless you want your model to hallucinate 😱)

Transfer learning summary

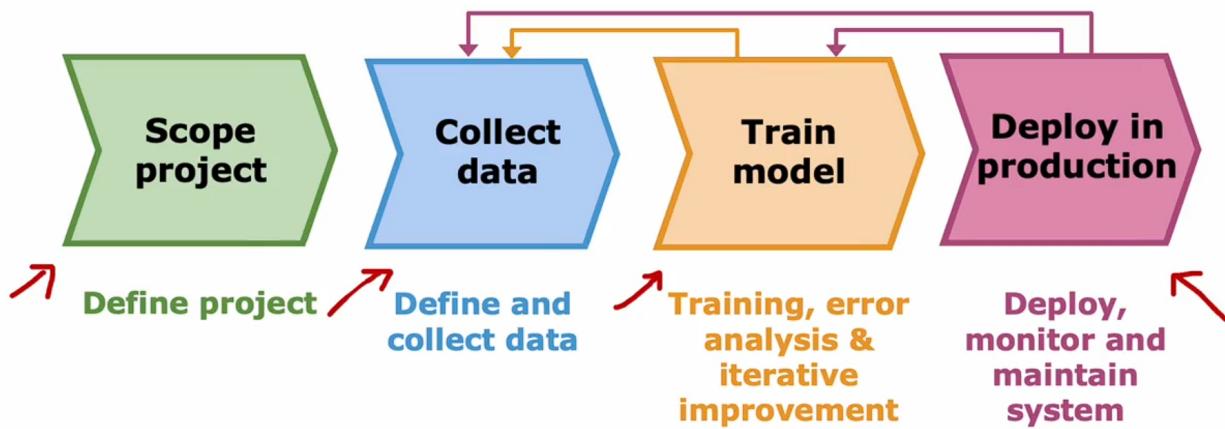
1. Download neural network parameters pretrained on a large dataset with same input type (e.g., images, audio, text) as your application (or train your own). *1 million images*
2. Further train (fine tune) the network on your own data.

10 00 images

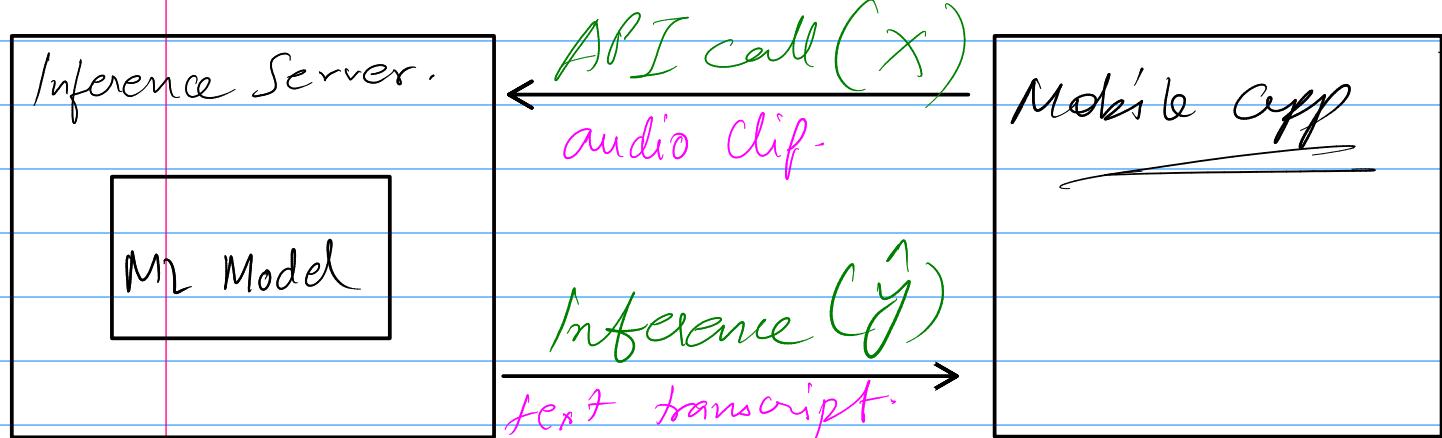
50 images

Full Cycle of a Machine Learning project

Full cycle of a machine learning project



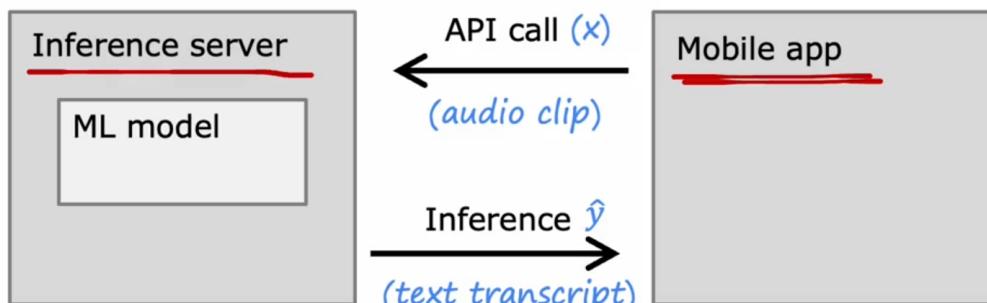
Deployment



Tumhare model ek inference server mey hogा।
Mobile app waqai apni tumhare server se
communicate karega and model ko
invoke karega with the input you gave.

model evaluate karega and apna prediction bhejega via the server to your mobile application as an output

Deployment



→ Software engineering may be needed for:

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

MLOps
machine learning operations

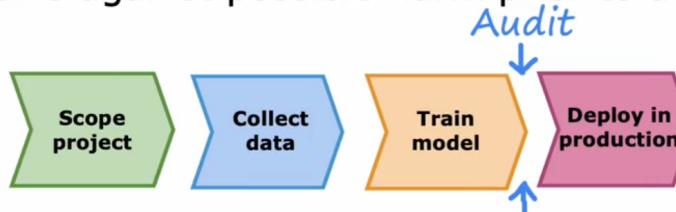


Guidelines

Get a diverse team to brainstorm things that might go wrong, with emphasis on possible harm to vulnerable groups.

Carry out literature search on standards/guidelines for your industry.

Audit systems against possible harm prior to deployment.



Develop mitigation plan (if applicable), and after deployment, monitor for possible harm.

Confusion Matrix

Actual Class		1	0
Predicted Class	1	True positive 15 TP	False positive 5 FP
	0	False negative 10 FN	True negative 70 TN

$$\text{Recall} = \frac{TP}{TP+FN} \downarrow 25 \qquad \downarrow 75$$

$$\text{Precision} = \frac{TP}{TP+FP}$$

Precision:

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

$$\frac{\text{True positives}}{\#\text{predicted positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False pos}} = \frac{15}{15+5} = 0.75$$

Recall:

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

$$\frac{\text{True positives}}{\#\text{actual positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False neg}}$$

1 Precision — model ka "clean shooter" hone ka scale 🎯

- Tells you: **When the model says "Yes/Positive", can you trust it?**
- High Precision → Model doesn't raise false alarms often.
- Low Precision → Model says "Positive" too often, even when it's wrong.

Example:

Spam detection

- Model predicts an email is spam.
- **Precision** = Of all emails marked as spam, how many were actually spam?
- High precision → Hardly any important mail got sent to spam.
- Low precision → Important mails also land in spam → user angry.

2 Recall — model ka "catch them all" ability 🔎

- Tells you: **Does the model miss real positives?**
- High Recall → Catches almost every positive case.
- Low Recall → Misses a lot of positives.

Example:

Cancer detection

- Model predicts whether someone has cancer.
- **Recall** = Of all patients who actually have cancer, how many did the model correctly flag?
- High recall → Almost every sick person is detected.
- Low recall → Some sick people go undetected → dangerous.

3 What they say about the learning algorithm

- **High Precision, Low Recall** → Model is *very picky*. It only says "Positive" when it's super sure → fewer false alarms, but misses many real positives. (*Like a strict teacher who rarely gives full marks.*)
- **Low Precision, High Recall** → Model is *very generous*. Flags a lot of positives → catches most true positives but also many false alarms. (*Like a CCTV motion sensor that triggers even if a leaf moves.*)
- **High Precision, High Recall** → Ideal → Model is both accurate when it predicts and catches almost everything.
- **Low Precision, Low Recall** → Trash → Can't predict correctly and misses a lot.

Precision v/s Recall Tradeoff

Alright, boss — simple tareeke se samajh le precision-recall tradeoff ka pura scene:

1 Tradeoff ka funda (simple)

- Precision ↑, Recall ↓ → Model becomes **picky** → predicts "Positive" only when super sure → fewer false alarms, but more actual positives get missed.
- Recall ↑, Precision ↓ → Model becomes **generous** → predicts "Positive" more often → catches more real positives, but also creates more **false alarms**.

Matlab:

- Agar tu **tight filter** lagata hai → kam galat hits, but miss zyada.
- Agar tu **loose filter** lagata hai → zyada catch, but galat bhi zyada.

2 Real-life me ka scene

In practice, tu **business needs** ke hisaab se decide karta hai balance:

- **Critical safety / health** → Recall ko priority dete hain (e.g., cancer detection, security breaches)
→ Thoda galat alarm chale toh chalega, bas koi important case miss na ho.
- **User experience / trust** → Precision ko priority dete hain (e.g., spam filter, face unlock)
→ Galat alarm bohot irritating hota hai, so model should be sure before saying "Positive."

3 Example table to visualize tradeoff

Precision	Recall	Model Behavior	Risk
High	Low	Picky sniper 🎩	Misses many true positives
Low	High	Over-enthusiastic guard 🚫	Too many false alarms
High	High	Perfect utopia 🚗	Rare in real life
Low	Low	Trash model 🗑️	Useless

Case A — Balanced (reasonable tradeoff)

Predictions: model finds a lot of true positives and some false positives.

Confusion matrix:

Actual \ Predicted	Positive	Negative	Total (actual)
Positive	TP = 80	FN = 20	100
Negative	FP = 40	TN = 860	900
Total (pred)	120	880	1000

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP}) = 80 / (80 + 40)$$

- Step 1: $80 + 40 = 120$
- Step 2: $80 / 120 = \text{reduce by } 40 \rightarrow 2 / 3$
- Long division: $2 \div 3 = 0.666666\ldots \rightarrow \text{round} = 0.6667$
- As percent: 66.67%

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN}) = 80 / (80 + 20)$$

- Step 1: $80 + 20 = 100$
- Step 2: $80 / 100 = 0.80 \rightarrow 80.00\%$

$$F1 = 2 * (\text{precision} * \text{recall}) / (\text{precision} + \text{recall})$$

Use fractions: precision = $2/3$, recall = $4/5$.

- Multiply: $(2/3) * (4/5) = 8/15$
- Numerator: $2 * (8/15) = 16/15$
- Denominator: $(2/3 + 4/5) = (10/15 + 12/15) = 22/15$
- Division: $(16/15) \div (22/15) = 16/22 = 8/11$
- Decimal: $8 \div 11 = 0.7272727\ldots \rightarrow 0.7273 \rightarrow 72.73\%$

Takeaway (A): decent recall (80%) and okay precision (66.7%). Good if you want a balanced model.

*Maximize F1 score
if you want a balanced
model.*

Case B — Picky (high precision, low recall)

Model only predicts "Positive" when very sure.

Confusion matrix:

Actual \ Predicted	Positive	Negative	Total (actual)
Positive	TP = 40	FN = 60	100
Negative	FP = 5	TN = 895	900
Total (pred)	45	955	1000

$$\text{Precision} = 40 / (40 + 5)$$

- Step 1: $40 + 5 = 45$
- Step 2: $40 / 45 = \text{reduce by } 5 \rightarrow 8 / 9$
- Long division: $8 \div 9 = 0.888888\ldots \rightarrow \text{round} = 0.8889 \rightarrow 88.89\%$

$$\text{Recall} = 40 / (40 + 60)$$

- Step 1: $40 + 60 = 100$
- Step 2: $40 / 100 = 0.40 \rightarrow 40.00\%$

F1 using fractions: precision = $8/9$, recall = $2/5$.

- Multiply: $(8/9) * (2/5) = 16/45$
- Numerator: $2 * 16/45 = 32/45$
- Denominator: $8/9 + 2/5 = (40/45 + 18/45) = 58/45$
- Division: $(32/45) \div (58/45) = 32/58 = \text{reduce} \rightarrow 16/29$
- Decimal: $16 \div 29 = 0.55172413\ldots \rightarrow 55.17\%$

Takeaway (B): Very reliable positive predictions (88.9%), but misses a lot of real positives (only 40% recall).

Good when *false positives are costly*.

Case C — Generous (high recall, low precision)

Model flags almost everything as suspicious.

Confusion matrix:

Actual \ Predicted	Positive	Negative	Total (actual)
Positive	TP = 95	FN = 5	100
Negative	FP = 300	TN = 600	900
Total (pred)	395	605	1000

$$\text{Precision} = 95 / (95 + 300)$$

- Step 1: $95 + 300 = 395$
- Step 2: $95 / 395 = \text{divide numerator \& denominator by } 5 \rightarrow 19 / 79$
- Long division ($19 \div 79$) — digit by digit to 6 decimals:
 - 79 into 19 → 0, remainder 19 → decimal: 0.
 - $190 \rightarrow 79 \times 2 = 158 \rightarrow \text{remainder } 32 \rightarrow \text{decimal digit } 2 \rightarrow 0.2$
 - $320 \rightarrow 79 \times 4 = 316 \rightarrow \text{remainder } 4 \rightarrow \text{decimal digit } 4 \rightarrow 0.24$
 - $40 \rightarrow 79 \times 0 = 0 \rightarrow \text{remainder } 40 \rightarrow \text{digit } 0 \rightarrow 0.240$
 - $400 \rightarrow 79 \times 5 = 395 \rightarrow \text{remainder } 5 \rightarrow \text{digit } 5 \rightarrow 0.2405$
 - $50 \rightarrow 79 \times 0 = 0 \rightarrow \text{remainder } 50 \rightarrow \text{digit } 0 \rightarrow 0.24050$
 - Continue → pattern yields $\approx 0.240506\dots$
- As percent: 24.05% (approx)

$$\text{Recall} = 95 / (95 + 5)$$

- Step 1: $95 + 5 = 100$
- Step 2: $95 / 100 = 0.95 \rightarrow 95.00\%$

$$\text{F1 (use fractions): precision} = 19/79, \text{recall} = 19/20.$$

- Multiply: $(19/79) * (19/20) = 361 / 1580$
- Numerator: $2 * 361/1580 = 722 / 1580$
- Sum: $19/79 + 19/20 = (19 \times 20) / (79 \times 20) + (19 \times 79) / (79 \times 20) = (380 + 1501) / 1580 = 1881 / 1580$
- Division: $(722/1580) \div (1881/1580) = 722 / 1881$
- Long division $722 \div 1881 = 0.3838383838\dots \rightarrow 0.3838 \rightarrow 38.38\%$

Takeaway (C): You catch almost every real positive (95% recall) but most positive predictions are false (precision $\approx 24\%$). Good when *missing a positive is very costly*, and you can afford many false alarms (e.g., triage with human review).

1 What is F1?

The F1 score is the harmonic mean of precision and recall:

$$F1 = 2 \times \frac{(\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})}$$

- It's high only when both precision and recall are high.
- If either precision or recall is low, F1 drops sharply.
- It's a single number to judge a model when you care about both equally.

2 Why harmonic mean?

The harmonic mean punishes imbalances more than the normal average:

- Example:
 - Avg of 90% and 10% (arithmetic mean) = 50%
 - But $F1 = \sim 18\% \rightarrow$ tells you "nah, this model sucks for balance."
 - This stops you from being fooled by one high number.

3 Example values & comparison

Let's compare four models:

Model	Precision	Recall	F1 Score	What it tells us	🔗
A	0.90	0.90	0.90	Excellent balance — catches most positives, and rarely wrong.	
B	0.95	0.50	0.655	Very picky → high precision but misses half of actual positives.	
C	0.50	0.95	0.655	Very generous → catches almost all positives but half are false alarms.	
D	0.60	0.60	0.60	Meh performance — average in both, nothing stands out.	

Relative interpretation

- A vs B/C: A is clearly better *overall* (balanced high numbers).
- B & C have the same F1 (0.655) even though one favors precision, the other recall → F1 ignores the "direction" of imbalance, just punishes the imbalance itself.
- D: Lower than B/C means it's weaker overall — both precision and recall are mediocre.
- B vs C: F1 says they're equally "balanced" in the *mathematical* sense, but *use case decides which is better* (e.g., spam filter → pick B, cancer detection → pick C).

4 What F1 really tells you

- Think of it as the final **combined score** when precision and recall are **team members** — if one is lazy, the team's score tanks.
- **High F1 (≥ 0.8)** → Good in both areas.
- **Medium (0.6–0.8)** → Tradeoffs exist; okay but not perfect.
- **Low (< 0.6)** → Either precision, recall, or both are too weak.

Golden Rule to Follow (Must Practice)

Always calculate the F1 score of your model.

Exactly 🤪 — that's the perfect TL;DR:

1. **Check F1** → Quick health check.
2. **If F1 is low** →
 - Look at **precision vs recall**.
 - If **both low** → model is trash, needs major work.
 - If **one high, one low** → could still be good depending on the business case (precision-heavy vs recall-heavy use case).
3. **If both high** → You've got a well-balanced, strong model. 🚀

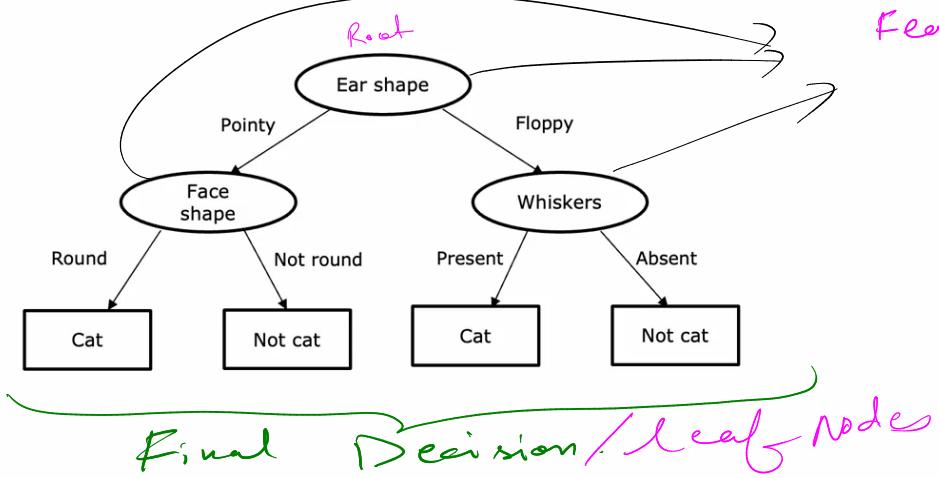
That's literally the precision-recall-F1 decision tree in one line.

Note! Precision, Recall, F1 all these are for
Classification use cases!

Week —

Decision Tree Models.

Decision Tree

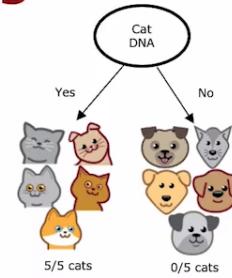
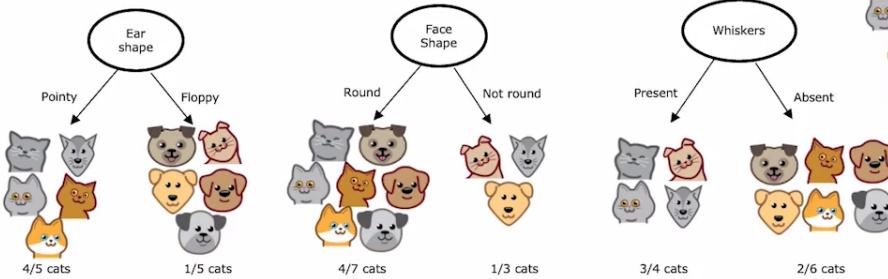


How to develop Decision Trees

Decision Tree Learning

Decision 1: How to choose what feature to split on at each node?

Maximize purity (or minimize impurity)



DeepLearning.AI Stanford ONLINE

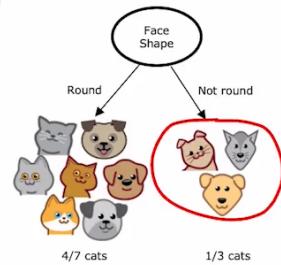
Andrew Ng

the greatest purity of

Decision Tree Learning

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 number of examples in a node is below a threshold



DeepLearning.AI Stanford ONLINE

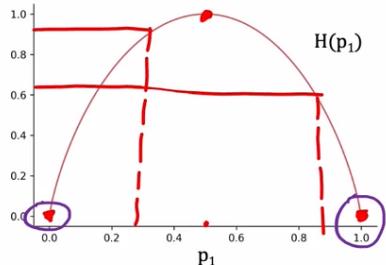
Andrew Ng

if you decided not to split

Measuring Purity

Entropy as a measure of impurity

p_1 = fraction of examples that are cats



Dog	Cat	Dog	Dog	Dog	Dog	Dog	$p_1 = 0 \quad H(p_1) = 0$
Cat	Cat	Dog	Dog	Dog	Dog	Dog	$p_1 = 2/6 \quad H(p_1) = 0.92$
Cat	Cat	Cat	Dog	Dog	Dog	Dog	$p_1 = 3/6 \quad H(p_1) = 1$
Cat	Cat	Cat	Dog	Dog	Dog	Dog	$p_1 = 5/6 \quad H(p_1) = 0.65$
Cat	$p_1 = 6/6 \quad H(p_1) = 0$						

DeepLearning.AI

Stanford ONLINE

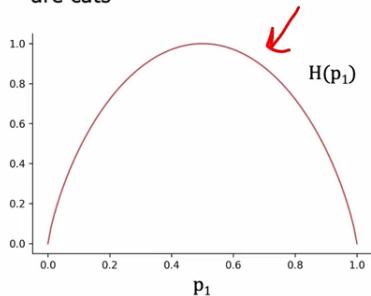
Andrew Ng

zero impurity or this would be

Entropy Formula

Entropy as a measure of impurity

p_1 = fraction of examples that are cats



$$p_0 = 1 - p_1$$

$$\begin{aligned} H(p_1) &= -p_1 \log_2(p_1) - p_0 \log_2(p_0) \\ &= -p_1 \log_2(p_1) - (1 - p_1) \log_2(1 - p_1) \end{aligned}$$



DeepLearning.AI

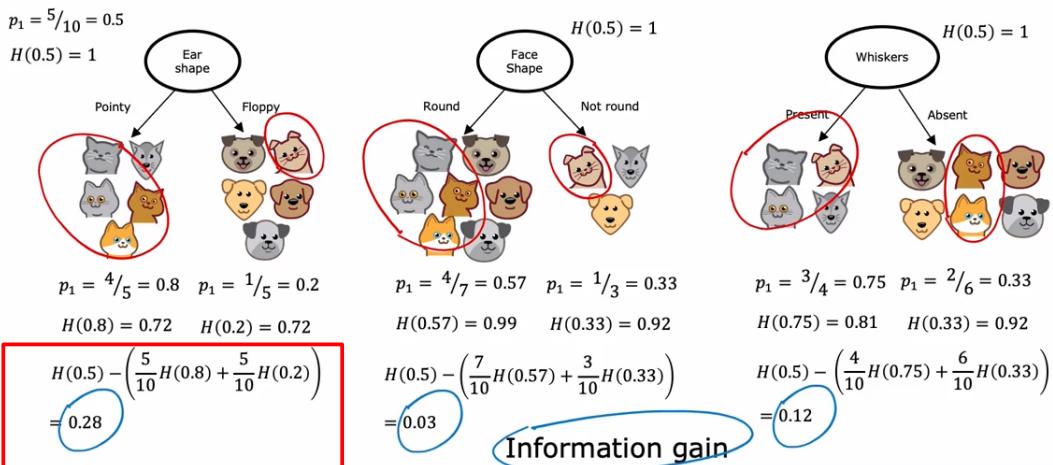
Stanford ONLINE

Andrew Ng

that it will be exactly this function on the left.

Choosing the correct splitting Feature - Information Gain

Choosing a split



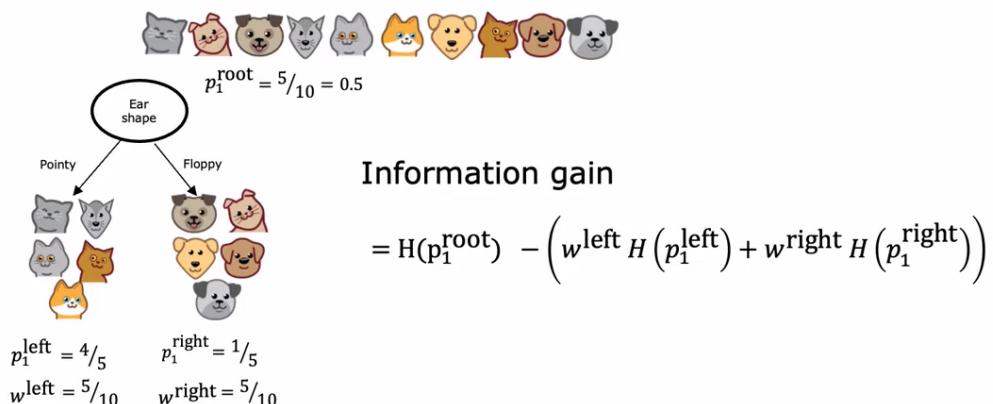
DeepLearning.AI Stanford ONLINE

Andrew Ng

and what it measures
is the reduction in

Maximum Information Gain is chosen as Splitting Node

Information Gain



Decision Tree Learning

- Start with all examples at the root node
- Calculate information gain for all possible features, and pick the one with the highest information gain
- Split dataset according to selected feature, and create left and right branches of the tree
- 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 maximum depth
 - Information gain from additional splits is less than threshold
 - When number of examples in a node is below a threshold

©DeepLearning.AI

Stanford University Online

You will keep on repeating
the splitting process until

Andrew Ng

One hot encoding of Categorical Features

One hot encoding

Ear shape	Pointy ears	Floppy ears	Oval ears	Face shape	Whiskers	Cat
Pointy	1	0	0	Round	Present	1
Oval	0	0	1	Not round	Present	1
Oval	0	0	1	Round	Absent	0
Pointy	1	0	0	Not round	Present	0
Oval	0	0	1	Round	Present	1
Pointy	1	0	0	Round	Absent	1
Floppy	0	1	0	Not round	Absent	0
Oval	0	0	1	Round	Absent	1
Floppy	0	1	0	Round	Absent	0
Floppy	0	1	0	Round	Absent	0

on only one of two possible values,
either 0 or 1.

Andrew Ng

If a categorical feature can take on k values, then we have to create k binary features that take on values 0 or 1 as seen in the screenshot above.

One hot encoding and neural networks

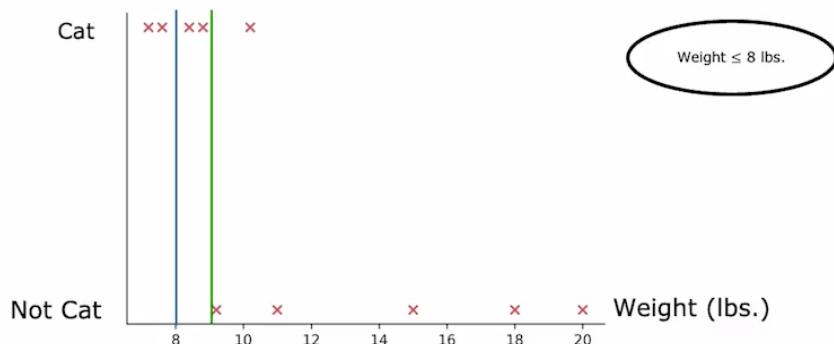
Pointy ears	Floppy ears	Round ears	Face shape	Whiskers	Cat
1	0	0	Round 1	Present 1	1
0	0	1	Not round 0	Present 1	1
0	0	1	Round 1	Absent 0	0
1	0	0	Not round 0	Present 1	0
0	0	1	Round 1	Present 1	1
1	0	0	Round 1	Absent 0	1
0	1	0	Not round 0	Absent 0	1
0	0	1	Round 1	Absent 0	1
0	1	0	Round 1	Absent 0	1
0	1	0	Round 1	Absent 0	1

one for whiskers and
encoded as a list of these five features.

Andrew Ng

Continued Valued Features

Splitting on a continuous variable

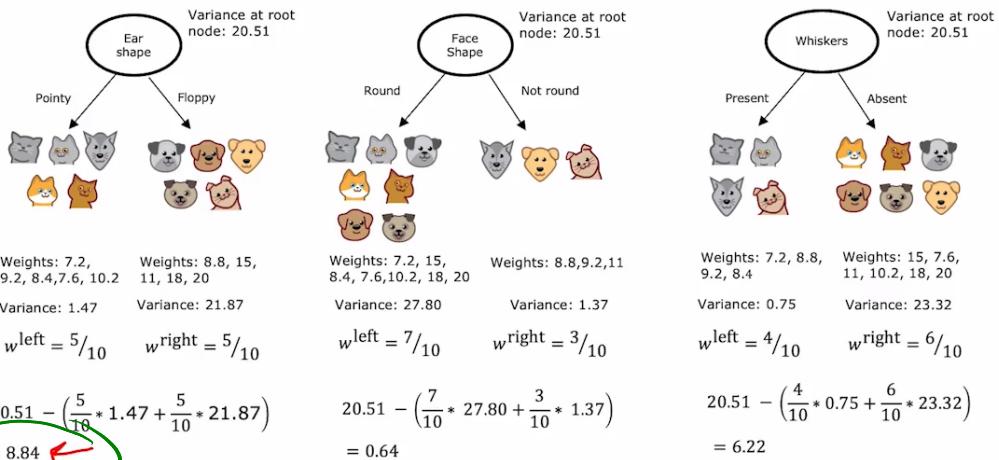


$$H(0.5) - \left(\frac{2}{10} H\left(\frac{2}{2}\right) + \frac{8}{10} H\left(\frac{3}{8}\right) \right) = 0.24 \quad \text{IG when threshold is } w \leq 8 \text{ lbs}$$

$$H(0.5) - \left(\frac{4}{10} H\left(\frac{4}{4}\right) + \frac{6}{10} H\left(\frac{1}{6}\right) \right) = 0.61 \quad \text{IG when threshold is } w \leq 9.2 \text{ lbs.}$$

Regression Trees

Choosing a split



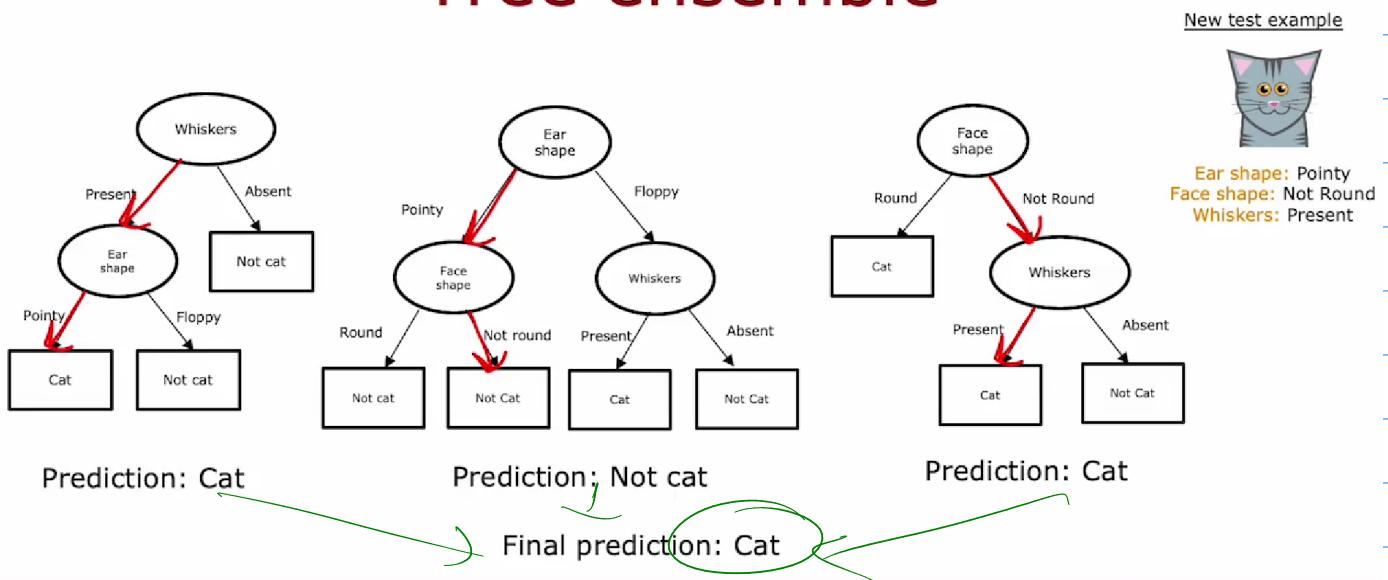
→ Largest reduction in variance would give you the splitting feature.

Tree Ensembles

Using multiple Decision Trees

A drawback of using only one DT is that it's really sensitive to any change in the data. And one solution to make the algorithm less sensitive to data/make it more robust is to use multiple decision trees. This is known as a tree ensemble.

Tree ensemble

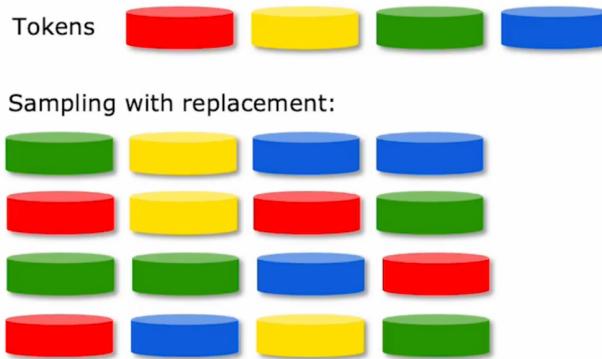


Since 2/3 decision trees classified it as a cat hence the final decision is that its a Cat.

Sampling w/ replacement

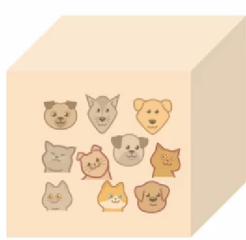
To build ensemble of trees we need a technique called Sampling w/ replacement.

Sampling with replacement



Below is an example that demonstrates sampling with replacement. Notice how we picked 10 random samples w/ replacement from the main dataset.

Sampling with replacement



	Ear shape	Face shape	Whiskers	Cat
1	Pointy	Round	Present	1
0	Floppy	Not round	Absent	0
1	Pointy	Round	Absent	1
0	Pointy	Not round	Present	0
0	Floppy	Not round	Absent	0
1	Pointy	Round	Absent	1
1	Pointy	Round	Present	1
1	Floppy	Not round	Present	1
0	Floppy	Round	Absent	0
1	Pointy	Round	Absent	1

The process of sampling w/ replacement

lets you construct a new training set
that's a little bit similar to but pretty
different to the original training set.

This would be the key building block
to building an ensemble of trees.

Random Forest Algorithm

Random Forest is one powerful tree
ensemble algorithm that works much
better than using a single decision
tree.

Generating a tree sample

Given training set of size m

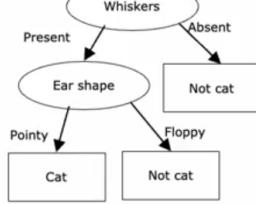
For $b = 1$ to B :

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

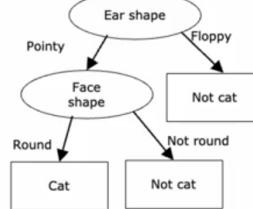
Train a decision tree on the new dataset

Ear shape	Face shape	Whiskers	Cat
Pointy	Round	Present	Yes
Floppy	Round	Absent	No
Pointy	Round	Absent	No
Pointy	Round	Present	Yes
Pointy	Not Round	Present	Yes
Floppy	Not Round	Absent	No
Floppy	Not Round	Present	Yes
Pointy	Not Round	Absent	No
Pointy	Not Round	Absent	Present
Pointy	Not Round	Present	Yes

Sample for DT



Ear shape	Face shape	Whiskers	Cat
Pointy	Round	Present	Yes
Floppy	Round	Absent	No
Pointy	Round	Absent	No
Pointy	Round	Present	Yes
Pointy	Not Round	Present	Yes
Floppy	Not Round	Absent	No
Floppy	Not Round	Present	Yes
Pointy	Not Round	Absent	No
Pointy	Not Round	Absent	Present
Pointy	Not Round	Present	Yes



Sample 2 for DT

you end up with a somewhat different decision tree.

Toh baat aisi hai - Let's say we have a training set that has m training examples. Let B denote the no. of samples with replacement generated.

So basically we will generate B no. samples each having m training example. Each sample from the generated B samples would be used to create a Decision tree.

B samples \rightarrow B DT's.

Now enter the test sample.

It would go thru all B DT's and all DT's would give their prediction.

Let's say $8/10 \rightarrow$ Cat $2/10 \rightarrow$ Not Cat

Never find answer due to majority
could be AT

Note: Setting 'B' to a high number wont
degrade the performance but beyond
a certain point you end up with
diminishing returns and it doesn't
get that much better when B is
larger than say 100 or 80.

If we set B=1000 it just increases
compute and doesn't affect performa-
nance.

This is also called Bagged Ensemble
Alg.

Randomizing the feature choice

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.

$k = \sqrt{n}$ for very large samples

→ Random Forest Algorithm.

Boosted Decision Tree Algo.

XG Boost

1. Normal Decision Tree

- Imagine a tree asking *yes/no questions* step by step:
"Is salary > 50k? → Yes? Then check age > 30? → etc."
- At the end, you get a prediction (class or number).
- Problem: If the tree is too deep → it **overfits**. If it's shallow → it **underfits**.

2. Boosted Trees Intuition

- Instead of training **one big perfect tree**, boosting says:
 "Let's build a series of small dumb trees, each one fixing mistakes of the previous one."

Example:

- Tree 1: Makes rough guesses (lots of errors).
- Tree 2: Looks at where Tree 1 failed and tries to improve *just those cases*.
- Tree 3: Improves on errors left by Tree 2.
- ...repeat this many times.

In the end: all small trees' predictions are added together → strong final model.

This is why it's called "**boosting**" → you boost weak learners into a strong one.

3. XGBoost Intuition

XGBoost is just "Boosted Trees on steroids" 

- Uses **gradient boosting** → instead of randomly correcting errors, each new tree is trained using the *gradient of the loss function* (i.e., it mathematically knows in which direction to fix mistakes).
- Has optimizations like:
 - **Regularization** → avoids overfitting.
 - **Parallelization** → trains trees faster.
 - **Handling sparse data** (missing values).
 - **Pruning** → throws away useless branches.

So compared to vanilla decision trees:

- Tree = One-shot prediction, can overfit easily.
- Boosting = Many weak trees chained, correcting each other.
- XGBoost = Super-optimized gradient boosting that is fast + accurate.

👉 Think of it like this:

- **Decision Tree** = one student giving an exam answer.
- **Boosted Trees** = a group project, where each new student fixes the mistakes of the previous student.
- **XGBoost** = same group project, but with toppers who also use math to figure out exactly *where and how* to improve the answer, while finishing faster.

XGBoost (eXtreme Gradient Boosting)

- Open source implementation of boosted trees
- Fast efficient implementation
- Good choice of default splitting criteria and criteria for when to stop splitting
- Built in regularization to prevent overfitting
- Highly competitive algorithm for machine learning competitions (eg: Kaggle competitions)

~~XG Boost Implementation~~

Using XGBoost

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 vs Neural Networks

Decision Trees and Tree ensembles

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, audio, text)
- Fast
- Small decision trees may be human interpretable

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

Finish

