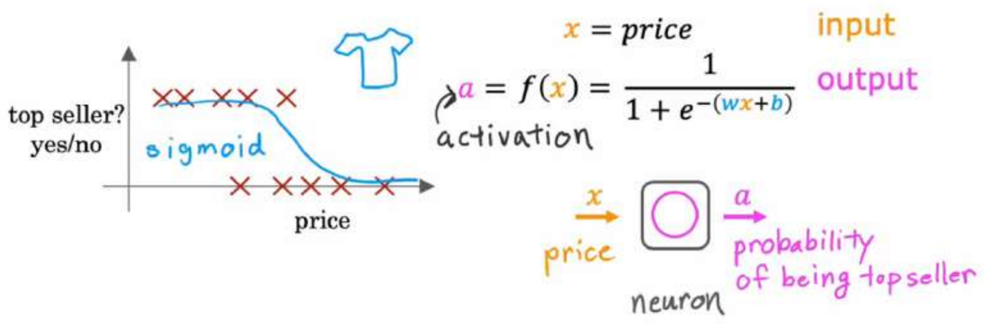
Course 2 - Machine Learning Specialisation: Advanced Learning Algorithms

Neural Networks

Demand prediction: predicting whether a product will sell

e.g. have collected data of different T-shirts sold at different prices and which have been top sellers (yes/no)

- Apply a sigmoid function to this. For deep learning, switch f(x) for a ("activation" - how much a neuron is sending a high output to other neurons downstream).
- This logistic regression algorithm can be thought of as a simplified model of a neuron (takes input x, and outputs a, the probability of being a top seller).
- Now need to wire together these neurons.

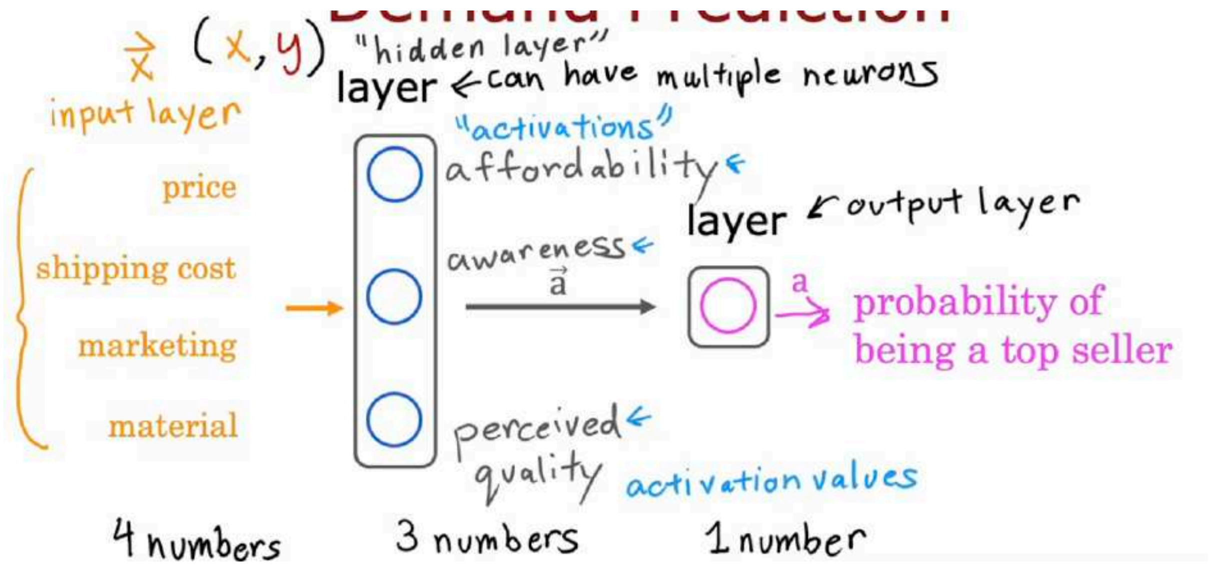


Using four features to predict whether t-shirt will be a top-seller: price, shipping cost, marketing, material. We are going to use an artificial neuron to predict outputs ("activations") of affordability (price + shipping cost), another to evaluate awareness (based on marketing), and another to evaluate perceived quality (material, and price (people think price = quality)).

We then wire the outputs of these three neurons to another neuron that takes these inputs and outputs the probability of being a top seller.
- i.e. we have grouped these neurons into a layer (layers can have a single or multiple neurons)

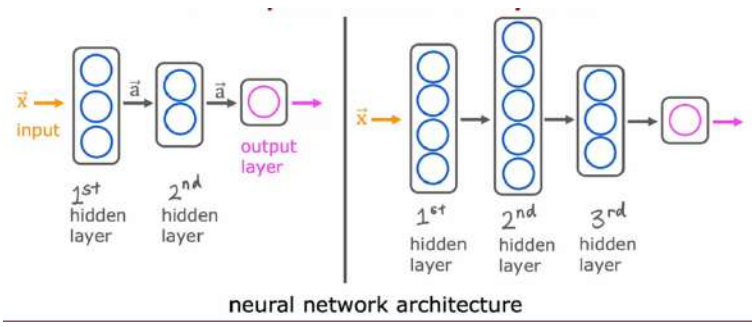
Don't want to manually assign which neurons to which features. In practice, each neuron will have access to every value from previous layer, and hopefully learn which features to ignore and which to focus on.
- Write input features as a vector x

Vector x is fed to layer in middle, which computes three activation values, which in turn become another vector a, which is fed to the final layer, which outputs the desired probability.



If you cover up the input layer, this is just logistic regression; however, rather than using the initial features, it is using new and hopefully improved "awarenesses". i.e. **it is a version of logistic regression that can learn the best features to create/ use.**
- i.e. it is a sort of automated feature engineering

Multiple hidden layers ("Neural Network Architecture")

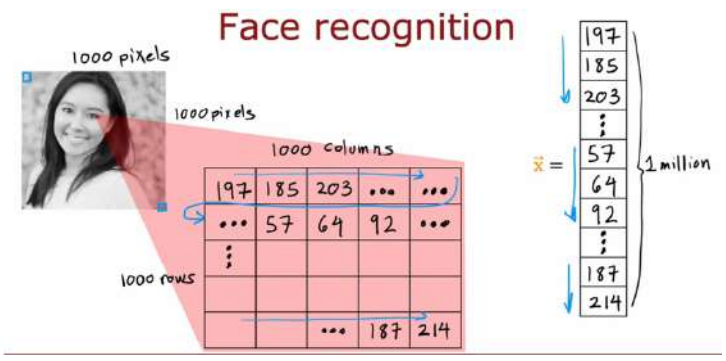


"Multilayer perceptron" = multiple hidden layers

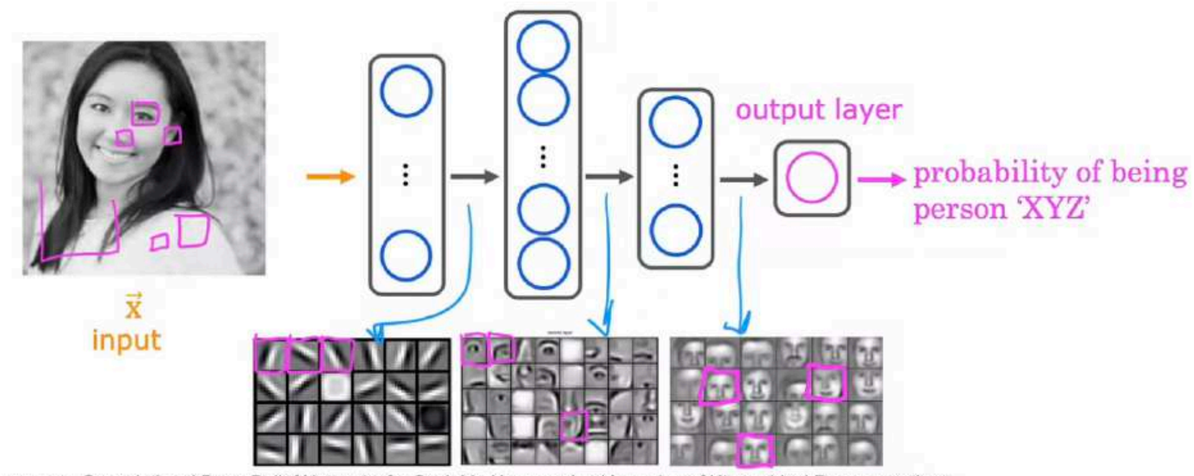
Recognising images: Computer vision

A 1000x1000 pixel image is represented in a computer by a 1000x1000 matrix of pixel intensity values (from 0 to 255)

Unrolling the pixel intensity values into a feature vector would give you **one million** features...



Building a neural network for face recognition:
- In the earliest neurons, they are often looking at short lines or segments
- In later neurons, they might learn to look for more complex lines i..e parts of phases (e.g. nose)
- In next hidden layer, neural network is aggregating parts of faces to detect presence or absence of larger, coarser face shapes, and then finally detecting how much the face corresponds to different face shapes
- It learnt to look for these features in edge hidden layer independently
- With each hidden layer, than are looking at bigger regions in the image: **activtions are higher level features**

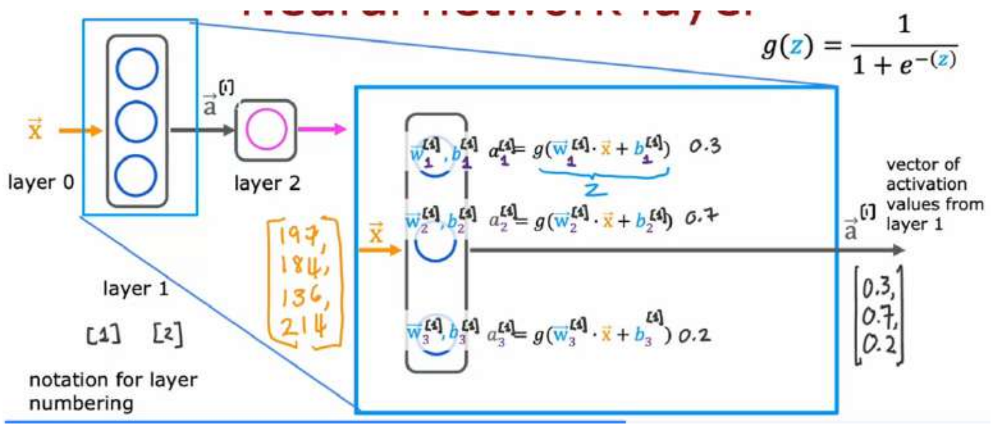


source: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations by Honglak Lee, Roger Grosse, Ranganath Andrew Y. Ng

<u>Neural Network Layer</u>

Computation of Layer 1

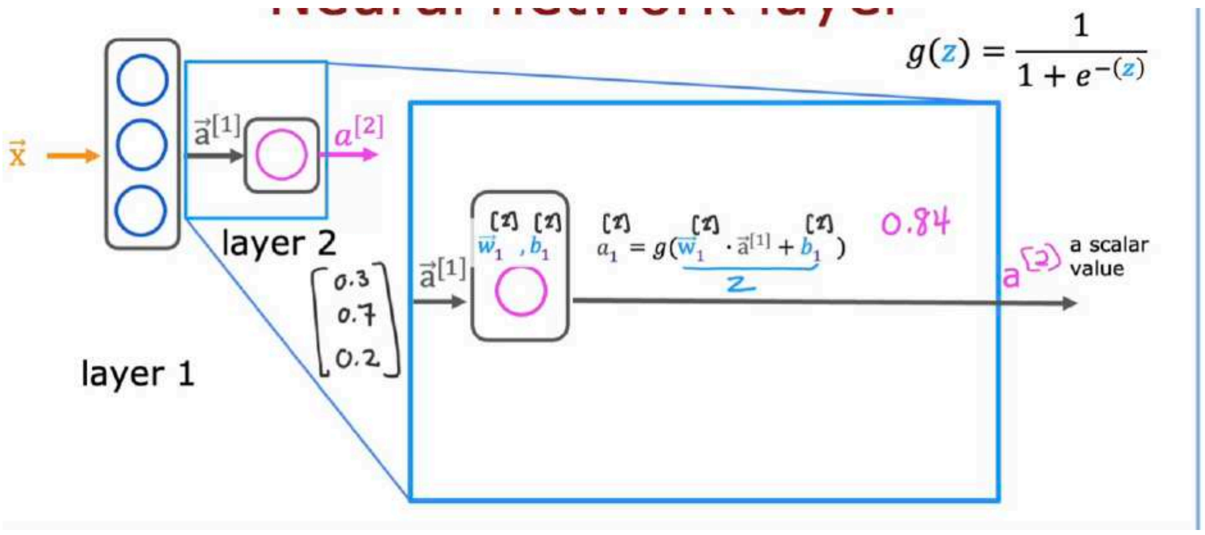
Hidden layer inputs four numbers to three neurons; each neuron implements a logistic regression function (with two parameters w and b). It will output some activation a where a = g(w.x + b)



$g(z) = \frac{1}{1+e^{-(z)}}$
$\vec{x}$
layer 0
$\vec{a}^{[1]}$
layer 2
layer 1
[1] [2]
notation for layer numbering
197, 184, 136, 214
$\vec{w}_1^{[1]}, b_1^{[1]}$ $a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]})$ 0.3
z
$\vec{w}_2^{[1]}, b_2^{[1]}$ $a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]})$ 0.7
$\vec{w}_3^{[1]}, b_3^{[1]}$ $a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]})$ 0.2
vector of activation values from layer 1
$\vec{a}^{[1]}$
0.3, 0.7, 0.2

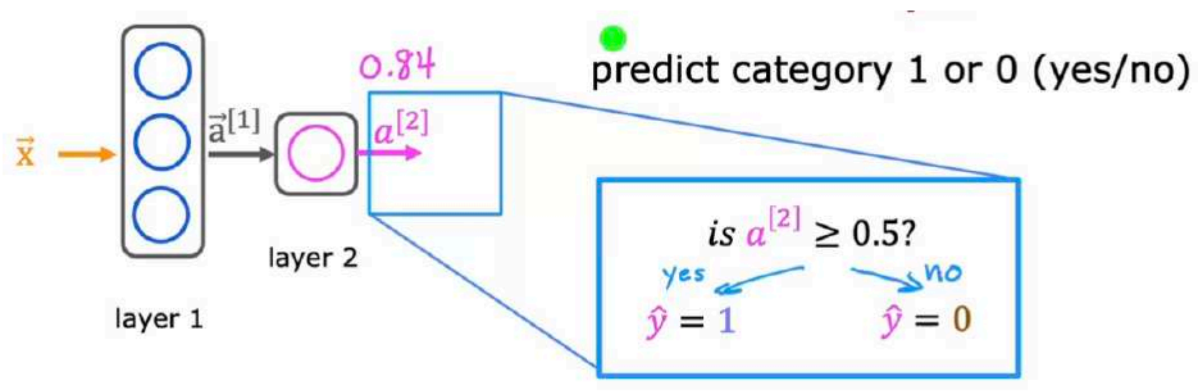
Computation of layer 2

$a^{[1]}$ is output from 1 and input to 2.

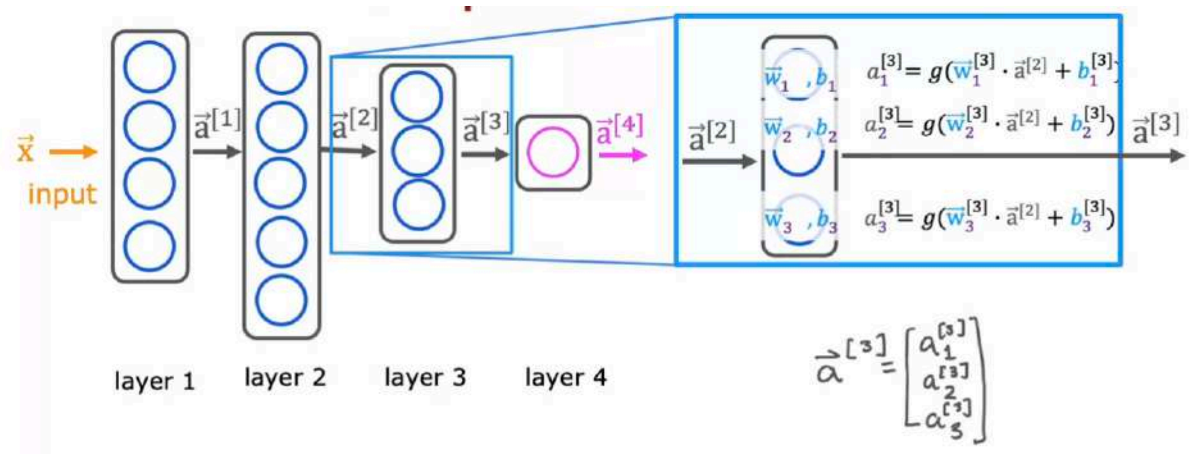


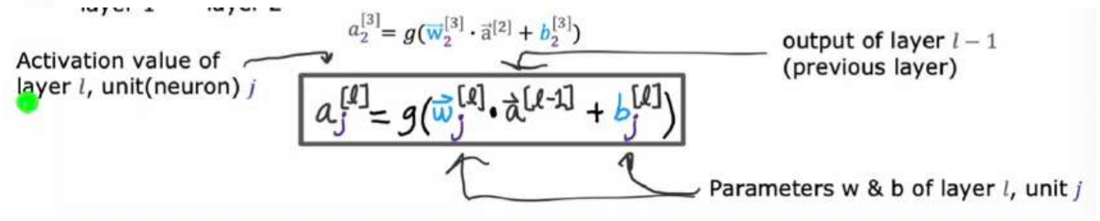
$g(z) = \frac{1}{1+e^{-(z)}}$
$\vec{x}$
$\vec{a}^{[1]}$
$a^{[2]}$
layer 2
layer 1
0.3 0.7 0.2
$\vec{a}^{[1]}$
$\vec{w}_1^{[2]}, b_1^{[2]}$
$a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]})$
z
0.84
$a^{[2]}$ a scalar value

Once the neural network has completed $a^{[2]}$, there is an optional final step, to make a binary prediction using a threshold.



More complex neural networks


$\vec{x}$
input
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]}$
$\vec{a}^{[4]}$
layer 1
layer 2
layer 3
layer 4
$\vec{w}_1, b_1$
$\vec{w}_2, b_2$
$\vec{w}_3, b_3$
$a_1^{[3]} = g(\vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]})$
$a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]})$
$a_3^{[3]} = g(\vec{w}_3^{[3]} \cdot \vec{a}^{[2]} + b_3^{[3]})$
$\vec{a}^{[3]} = \begin{bmatrix} a_1^{[3]} \\ a_2^{[3]} \\ a_3^{[3]} \end{bmatrix}$

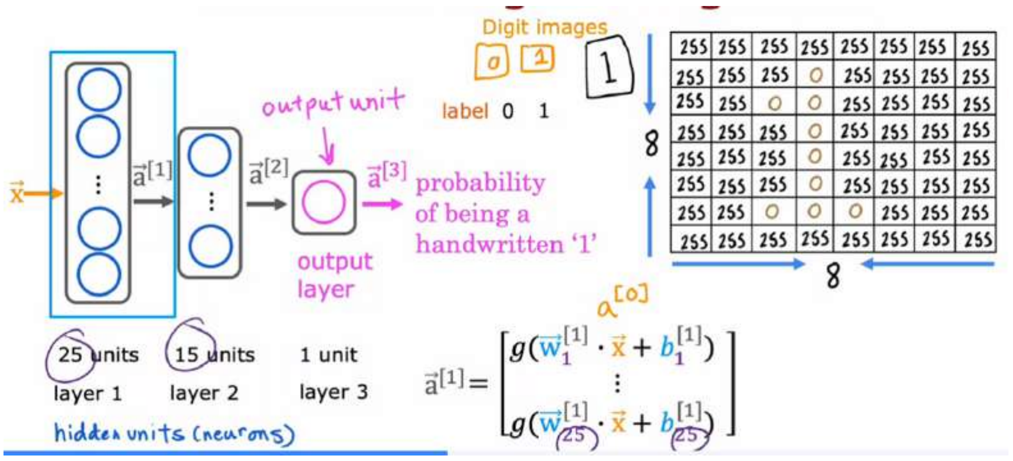

$a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]})$
Activation value of layer l, unit(neuron) j
output of layer $l-1$ (previous layer)
$a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})$
Parameters w & b of layer l, unit j

where g is the sigmoid or "activation" function.

$x = a^{[0]}$

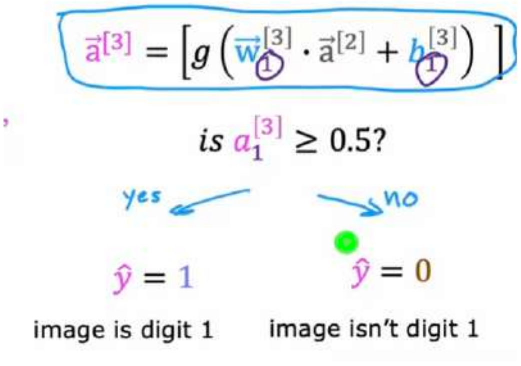
Inference: making predictions (forward propagation)

e.g. Handwritten digit recognition, differentiating between 0 and 1,=. Use an 8x8 matrix, where 255 represents white and 0 black.


Digit images
label 0 1
$\vec{x}$
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]}$
output unit
probability of being a handwritten '1'
output layer
25 units
15 units
1 unit
layer 1
layer 2
layer 3
hidden units (neurons)
8
8
$a^{[0]}$
$\vec{a}^{[1]} = \begin{bmatrix} g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]}) \\ \vdots \\ g(\vec{w}_{25}^{[1]} \cdot \vec{x} + b_{25}^{[1]}) \end{bmatrix}$

255	255	255	255	255	255	255	255
255	255	255	0	255	255	255	255
255	255	0	0	255	255	255	255
255	255	255	0	255	255	255	255
255	255	255	0	255	255	255	255
255	255	255	0	255	255	255	255
255	255	0	0	0	255	255	255
255	255	255	255	255	255	255	255

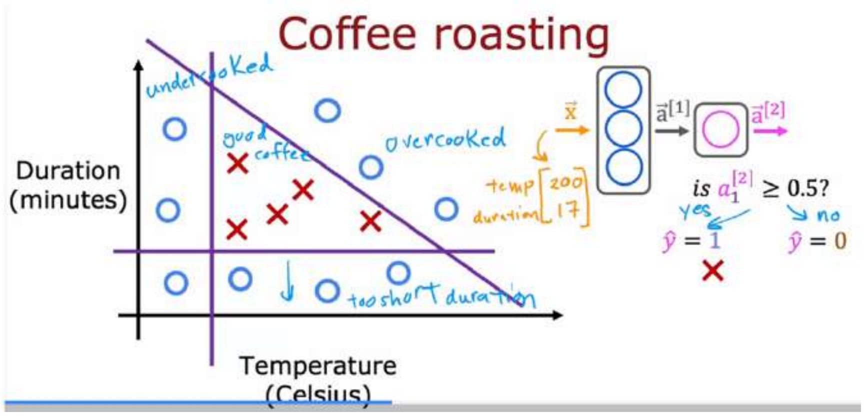
$$\vec{a}^{[2]} = \begin{bmatrix} g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}) \\ \vdots \\ g(\vec{w}_{15}^{[2]} \cdot \vec{a}^{[1]} + b_{15}^{[2]}) \end{bmatrix}$$



This is called forward propagation: making computations from left to right (vs back propagation, which is used for learning).

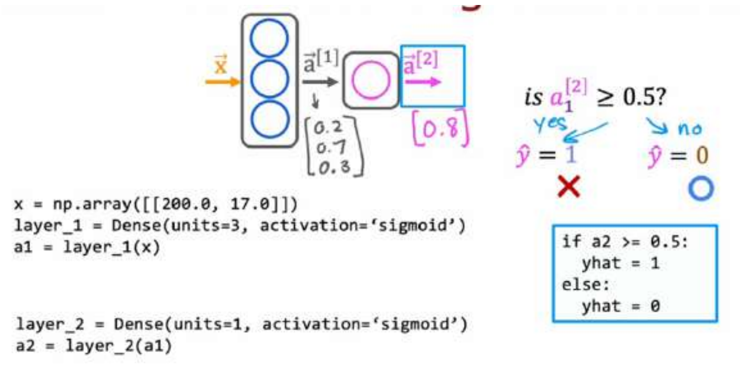
Inference in Code (TensorFlow)

- Coffee roasting: you control the temperature and the duration. There is a sweet spot.



Building the model using TensorFlow

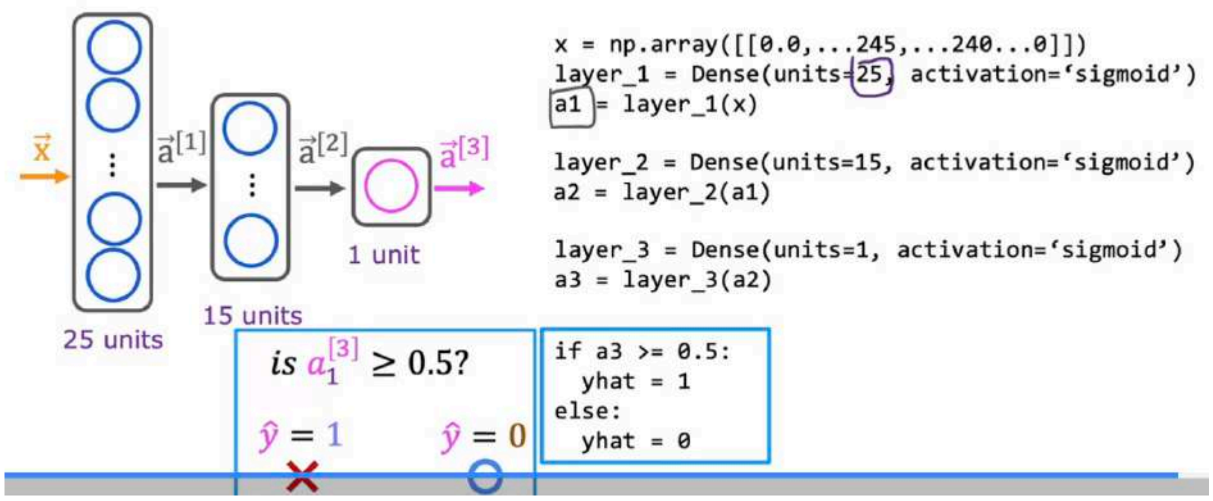
Set x to be an array of two numbers, the input temperature (200) and time (17 mins).
- Then create a layer with 3 units, using sigmoid activation
- Then computer a1 by applying layer 1 to the value of x



```
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)
```

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

Looking back at the model for digit classification
- x is a numpy array of list of intensity values

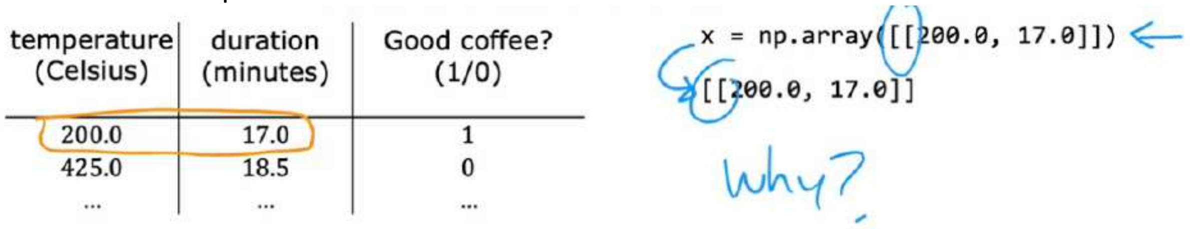


```
x = np.array([[0.0,...245,...240...0]])
layer_1 = Dense(units=25, activation='sigmoid')
a1 = layer_1(x)

layer_2 = Dense(units=15, activation='sigmoid')
a2 = layer_2(a1)

layer_3 = Dense(units=1, activation='sigmoid')
a3 = layer_3(a2)
```

```
if a3 >= 0.5:
  yhat = 1
else:
  yhat = 0
```

Data in TensorFlow

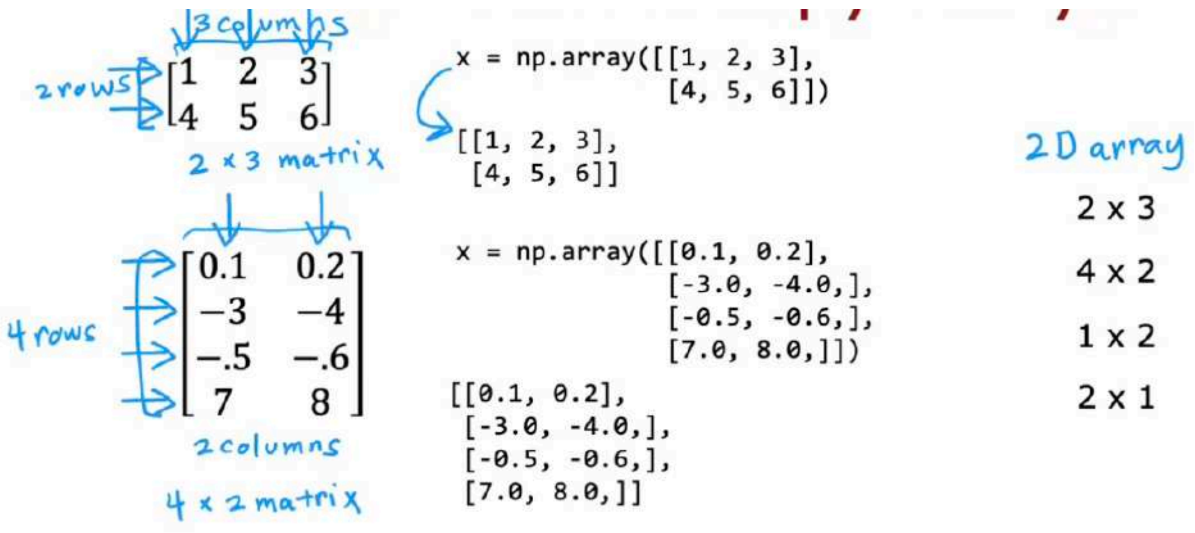
- Use of double square bracket



temperature (Celsius)	duration (minutes)	Good coffee? (1/0)
200.0	17.0	1
425.0	18.5	0
...	...	...

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

Matrices in code:



$$\begin{array}{c|ccc} & & \text{3 columns} & \\ \text{2 rows} & 1 & 2 & 3 \\ & 4 & 5 & 6 \end{array}$$

2 x 3 matrix

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

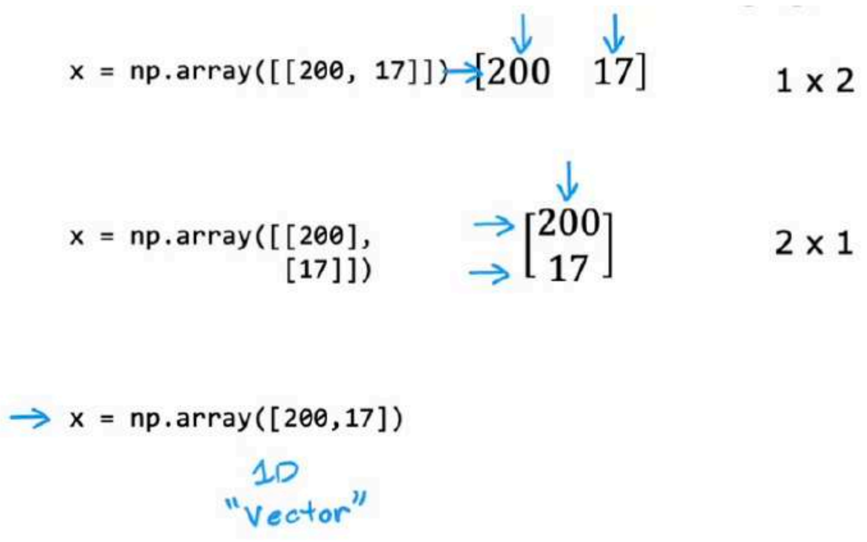
$$\begin{array}{c|cc} \text{4 rows} & 0.1 & 0.2 \\ & -3 & -4 \\ & -.5 & -.6 \\ & 7 & 8 \\ \hline & \text{2 columns} & \end{array}$$

4 x 2 matrix

```
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,]]
```

2D array
2 x 3
4 x 2
1 x 2
2 x 1

Note about numpy arrays:



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

$[200 \quad 17]$ 1 x 2

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

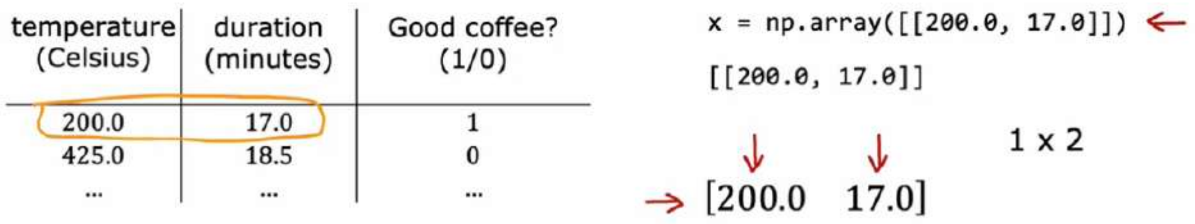
$\begin{bmatrix}200\\17\end{bmatrix}$ 2 x 1

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

1D
"Vector"

With linear and logistic regression, we used 1D vectors. In TensorFlow, use matrices instead of 1D arrays.

Going back to first example... (Feature vectors)



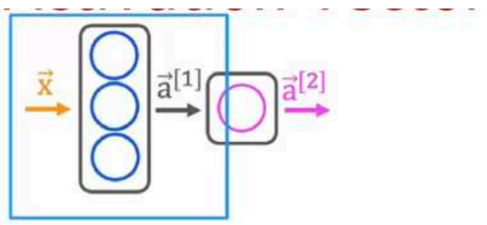
temperature (Celsius)	duration (minutes)	Good coffee? (1/0)
200.0	17.0	1
425.0	18.5	0
...	...	...

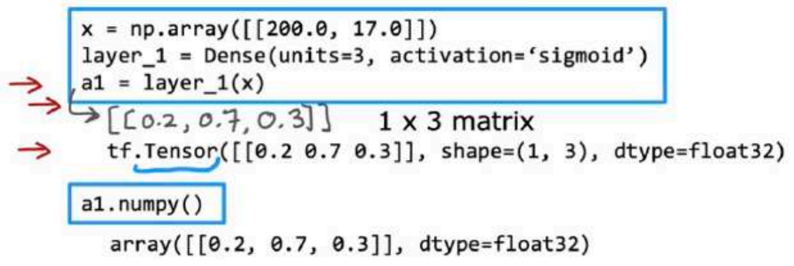
```
x = np.array([[200.0, 17.0]])
[[200.0, 17.0]]
```

$[200.0 \quad 17.0]$ 1 x 2

Not a 1D array, but a 1x2 matrix.

Activation vector





```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
```

[[0.2, 0.7, 0.3]] 1 x 3 matrix

```
tf.Tensor([[0.2 0.7 0.3]], shape=(1, 3), dtype=float32)
```

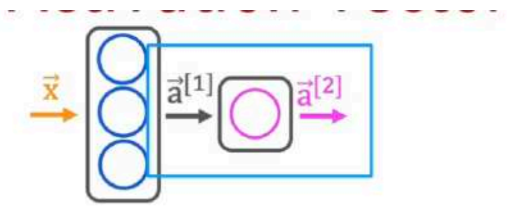
```
a1.numpy()
```

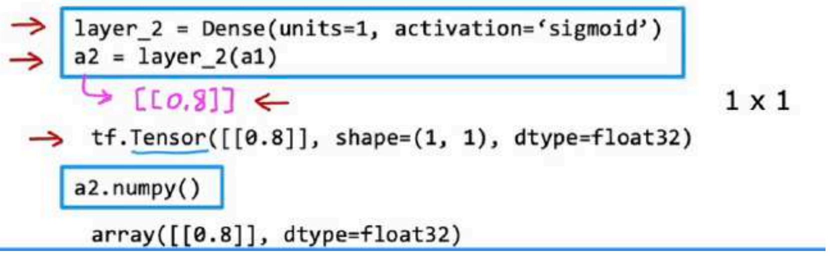
```
array([[0.2, 0.7, 0.3]], dtype=float32)
```

float32 = a number that can have a decimal point represented by 32 bits of memory in computer

Tensor = a tensor is a datatype that allows storage and efficient computations of matrices. Way of representing matrices.

a1.numpy() then takes same data and returns it in form of **numpy array** (rather than as a tensor flow matrix)





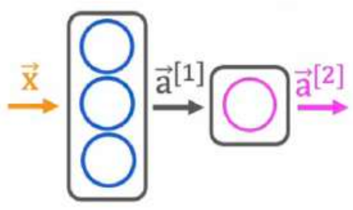
```
→ layer_2 = Dense(units=1, activation='sigmoid')
→ a2 = layer_2(a1)
    ↳ [[0.8]] ←
→ tf.Tensor([[0.8]], shape=(1, 1), dtype=float32)
  a2.numpy()
    array([[0.8]], dtype=float32)
```

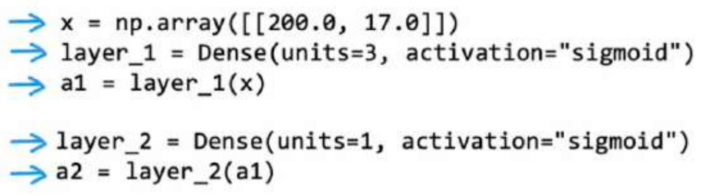
1 x 1

1 x 1 matrix

Once again, can convert from a tensorflow tensor to a numpy matrix using a2.numpy().

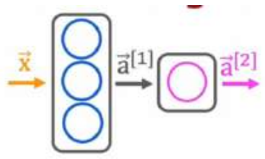
Overall, for forward propagation:

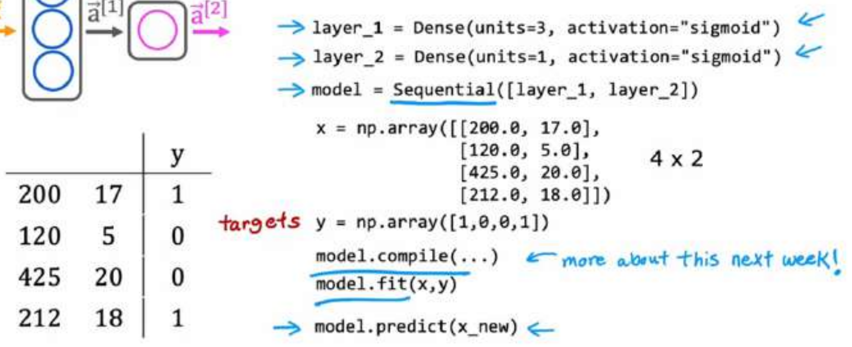




```
→ x = np.array([[200.0, 17.0]])
→ layer_1 = Dense(units=3, activation="sigmoid")
→ a1 = layer_1(x)

→ layer_2 = Dense(units=1, activation="sigmoid")
→ a2 = layer_2(a1)
```

Tensorflow also has a different way of building a neural network. Can ask tensorflow to create a neural network by **sequentially** stringing together the two layers you have created:





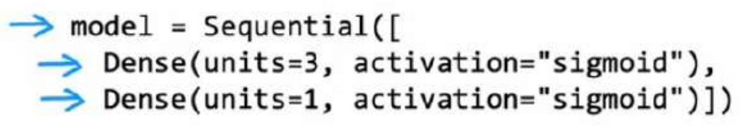
		y
200	17	1
120	5	0
425	20	0
212	18	1

```
→ layer_1 = Dense(units=3, activation="sigmoid") ←
→ layer_2 = Dense(units=1, activation="sigmoid") ←
→ model = Sequential([layer_1, layer_2])
   x = np.array([[200.0, 17.0],
                 [120.0, 5.0],
                 [425.0, 20.0],
                 [212.0, 18.0]])
targets y = np.array([1,0,0,1])
   model.compile(...)  ← more about this next week!
   model.fit(x,y)
→ model.predict(x_new) ←
```

4 x 2

To carry out forward propagation and output a2, use model.predict(x_new)

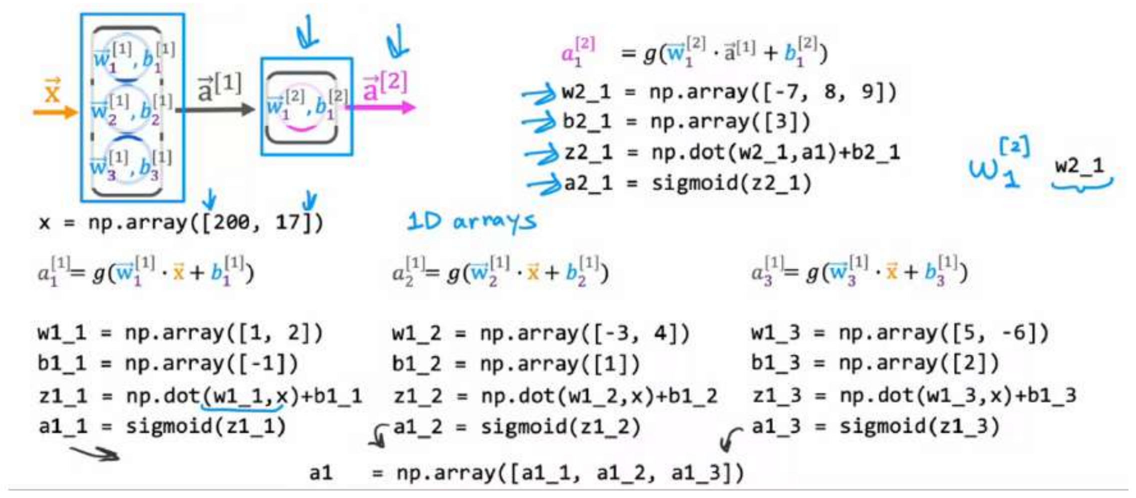
By convention, to simplify the code, we can simply the first bit by this:



```
→ model = Sequential([
   → Dense(units=3, activation="sigmoid"),
   → Dense(units=1, activation="sigmoid")])
```

Building forward prop in a single layer from scratch

NB using 1D arrays here so only one square bracket.



$\vec{x}$
$\vec{w}_1^{[1]}, b_1^{[1]}$
$\vec{w}_2^{[1]}, b_2^{[1]}$
$\vec{w}_3^{[1]}, b_3^{[1]}$
$\vec{a}^{[1]}$
$\vec{w}_1^{[2]}, b_1^{[2]}$
$\vec{a}^{[2]}$
1D arrays
$w_1^{[2]}$
w2_1

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

```
w2_1 = np.array([-7, 8, 9])
b2_1 = np.array([3])
z2_1 = np.dot(w2_1,a1)+b2_1
a2_1 = sigmoid(z2_1)
```

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

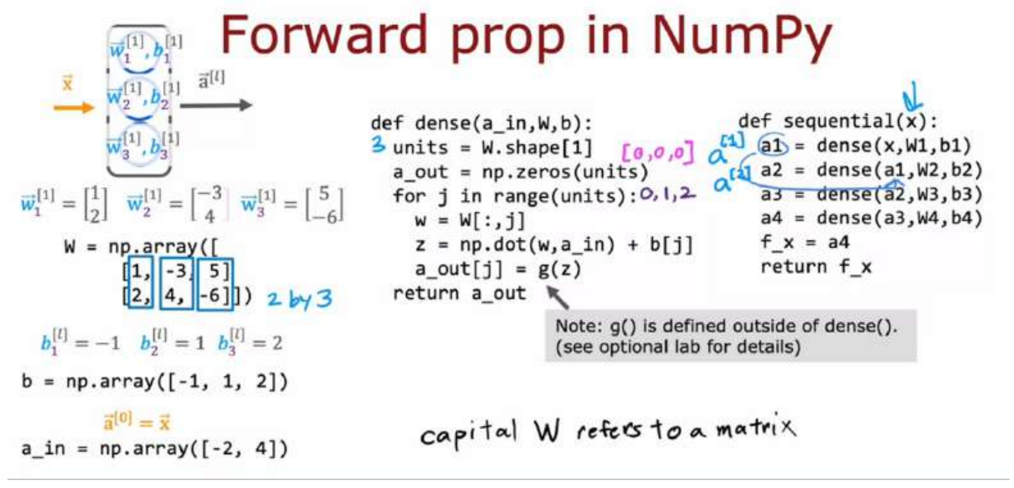
$a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]})$ $a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]})$ $a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]})$

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

                      a1 = np.array([a1_1, a1_2, a1_3])
```

General implementation of forward propagation

Forward prop in NumPy

- Dense function inputs activations from previous layer and the w and b parameters stacked into columns and 1d arrays respectively, and outputs activations from current layer
- w = W [:, j] - pulls out first column in first loop, and second column in second loop
- NB use capital W to refer to matrix and lower case w to refer to vectors and scalars



$\vec{x}$
$\vec{w}_1^{[1]}, b_1^{[1]}$
$\vec{w}_2^{[1]}, b_2^{[1]}$
$\vec{w}_3^{[1]}, b_3^{[1]}$
$\vec{a}^{[l]}$

Forward prop in NumPy

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

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

$b_1^{[l]} = -1$ $b_2^{[l]} = 1$ $b_3^{[l]} = 2$

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

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

```
a_in = np.array([-2, 4])
```

```
def dense(a_in,W,b):
  3 units = W.shape[1]      [0,0,0]
    a_out = np.zeros(units)
    for j in range(units): 0,1,2
      w = W[:,j]
      z = np.dot(w,a_in) + b[j]
      a_out[j] = g(z)
    return a_out
```

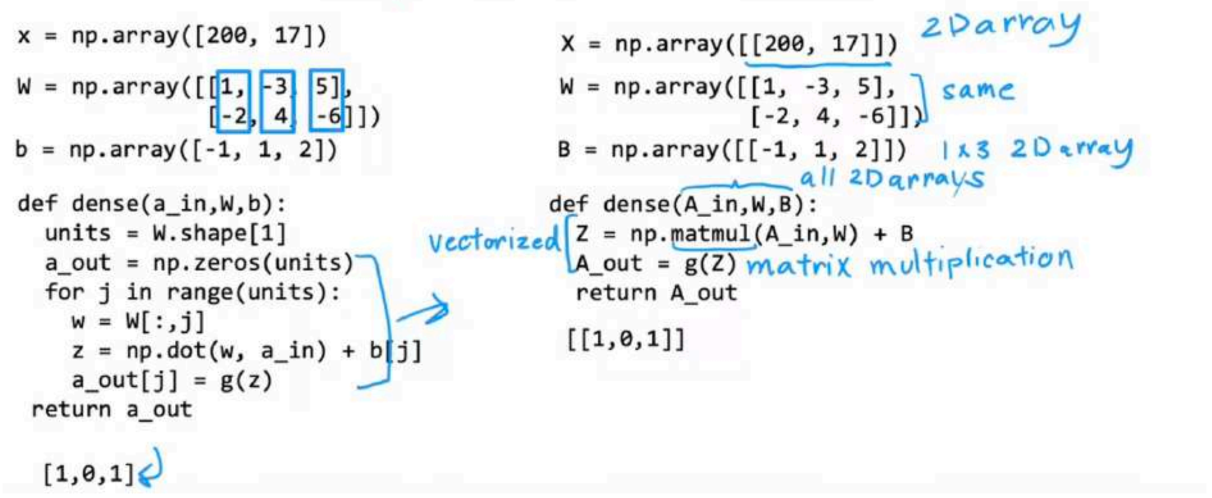
```
def sequential(x):
  a[1] a1 = dense(x,W1,b1)
  a[2] a2 = dense(a1,W2,b2)
       a3 = dense(a2,W3,b3)
       a4 = dense(a3,W4,b4)
       f_x = a4
       return f_x
```

Note: g() is defined outside of dense().
(see optional lab for details)

capital W refers to a matrix

Vectorisation

For loops vs vectorization (vectorised implementation of forward prop)



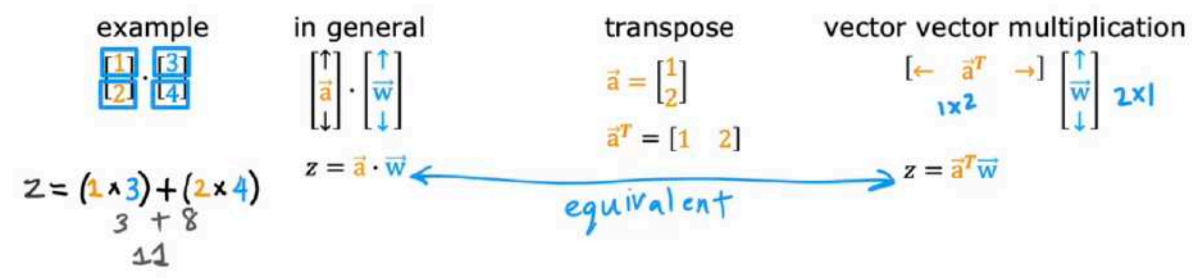
```
x = np.array([200, 17])

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]
```

```
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

[[1,0,1]]
```

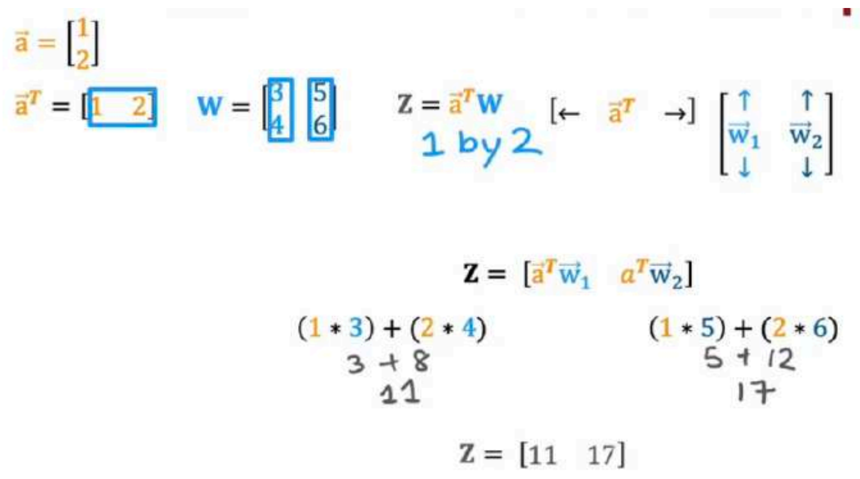
Matrix multiplication

Dot products and transposes

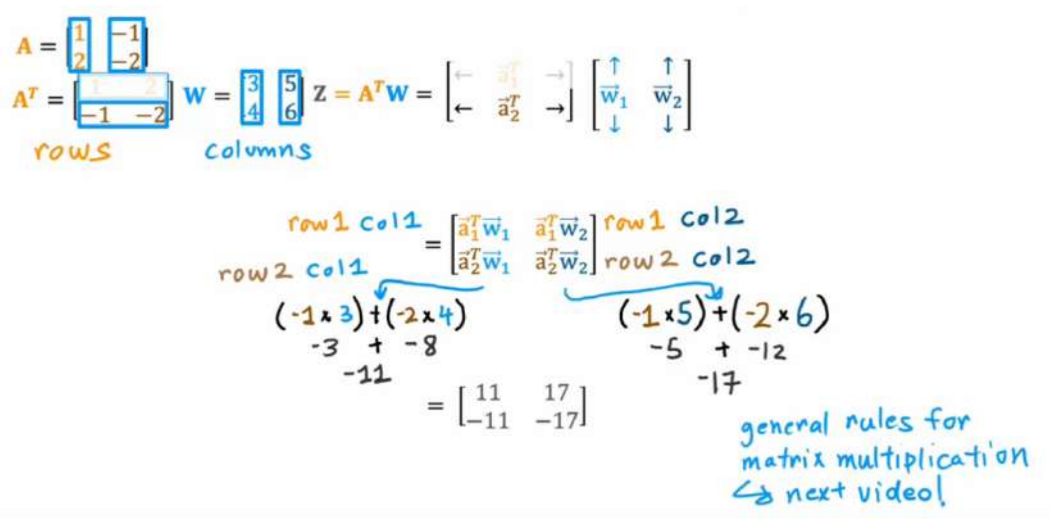
There are two ways of writing dot products:


example
in general
transpose
vector vector multiplication
$\begin{bmatrix}1\\2\end{bmatrix} \cdot \begin{bmatrix}3\\4\end{bmatrix}$
$z = (1 \times 3) + (2 \times 4)$
$3 + 8$
11
$z = \vec{a} \cdot \vec{w}$
$\vec{a} = \begin{bmatrix}1\\2\end{bmatrix}$
$\vec{a}^T = \begin{bmatrix}1 & 2\end{bmatrix}$
1x2
2x1
equivalent
$z = \vec{a}^T \vec{w}$

Vector-matrix multiplication


$\vec{a} = \begin{bmatrix}1\\2\end{bmatrix}$
$\vec{a}^T = \begin{bmatrix}1 & 2\end{bmatrix}$
$\mathbf{W} = \begin{bmatrix}3 & 5\\4 & 6\end{bmatrix}$
$\mathbf{Z} = \vec{a}^T \mathbf{W}$
1 by 2
$\mathbf{Z} = \begin{bmatrix}\vec{a}^T\vec{w}_1 & a^T\vec{w}_2\end{bmatrix}$
$(1 * 3) + (2 * 4)$
$3 + 8$
11
$(1 * 5) + (2 * 6)$
$5 + 12$
17
$\mathbf{Z} = \begin{bmatrix}11 & 17\end{bmatrix}$

Matrix-matrix multiplication



$$\mathbf{A} = \begin{bmatrix}1\\2\end{bmatrix} \begin{bmatrix}-1\\-2\end{bmatrix}$$

$$\mathbf{A}^T = \begin{bmatrix}1 & 2\\-1 & -2\end{bmatrix} \quad \mathbf{W} = \begin{bmatrix}3 & 5\\4 & 6\end{bmatrix} \quad \mathbf{Z} = \mathbf{A}^T\mathbf{W} = \begin{bmatrix}\leftarrow & \vec{a}_1^T & \rightarrow\\ \leftarrow & \vec{a}_2^T & \rightarrow\end{bmatrix}\begin{bmatrix}\uparrow & \uparrow\\ \vec{w}_1 & \vec{w}_2\\ \downarrow & \downarrow\end{bmatrix}$$

rows columns

$$= \begin{bmatrix}\vec{a}_1^T\vec{w}_1 & \vec{a}_1^T\vec{w}_2\\ \vec{a}_2^T\vec{w}_1 & \vec{a}_2^T\vec{w}_2\end{bmatrix}$$

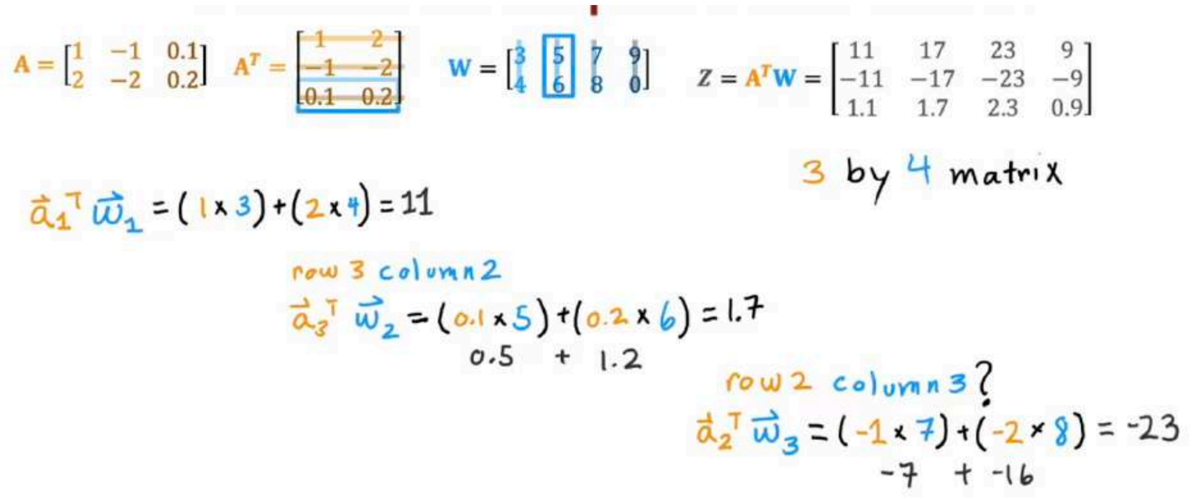
row 1 col 1 row 1 col 2 row 2 col 1 row 2 col 2

$$(-1\times3)+(-2\times4)$$
$$-3 + -8$$
$$-11$$

$$(-1\times5)+(-2\times6)$$
$$-5 + -12$$
$$-17$$

$$= \begin{bmatrix}11 & 17\\-11 & -17\end{bmatrix}$$

general rules for matrix multiplication ↳ next video!



$$\mathbf{A} = \begin{bmatrix}1 & -1 & 0.1\\2 & -2 & 0.2\end{bmatrix} \quad \mathbf{A}^T = \begin{bmatrix}1 & 2\\-1 & -2\\0.1 & 0.2\end{bmatrix} \quad \mathbf{W} = \begin{bmatrix}3 & 5 & 7 & 9\\4 & 6 & 8 & 0\end{bmatrix} \quad Z = \mathbf{A}^T\mathbf{W} = \begin{bmatrix}11 & 17 & 23 & 9\\-11 & -17 & -23 & -9\\1.1 & 1.7 & 2.3 & 0.9\end{bmatrix}$$

3 by 4 matrix

$$\vec{a}_1^T\vec{w}_1 = (1\times3)+(2\times4) = 11$$

row 3 column 2

$$\vec{a}_3^T\vec{w}_2 = (0.1\times5)+(0.2\times6) = 1.7$$
$$0.5 + 1.2$$

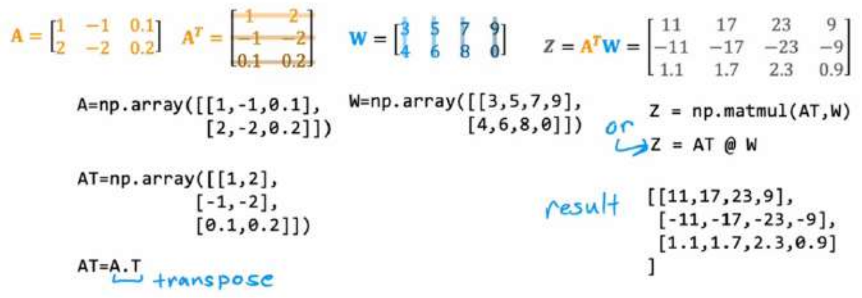
row 2 column 3?

$$\vec{a}_2^T\vec{w}_3 = (-1\times7)+(-2\times8) = -23$$
$$-7 + -16$$

- Can only take dot products between vectors that are the same length, so need number of columns of first matrix to be equal to number of rows of second matrix. Output has same number of rows as A^T and same number of columns as W.

<u>Matrix multiplication code</u>

Matrix multiplication in NumPy



$$\mathbf{A} = \begin{bmatrix}1 & -1 & 0.1\\2 & -2 & 0.2\end{bmatrix} \quad \mathbf{A}^T = \begin{bmatrix}1 & 2\\-1 & -2\\0.1 & 0.2\end{bmatrix} \quad \mathbf{W} = \begin{bmatrix}3 & 5 & 7 & 9\\4 & 6 & 8 & 0\end{bmatrix} \quad Z = \mathbf{A}^T\mathbf{W} = \begin{bmatrix}11 & 17 & 23 & 9\\-11 & -17 & -23 & -9\\1.1 & 1.7 & 2.3 & 0.9\end{bmatrix}$$

```
A=np.array([[1,-1,0.1],
            [2,-2,0.2]])

AT=np.array([[1,2],
             [-1,-2],
             [0.1,0.2]])

AT=A.T
```

transpose

```
W=np.array([[3,5,7,9],
            [4,6,8,0]])
```

```
Z = np.matmul(AT,W)
```

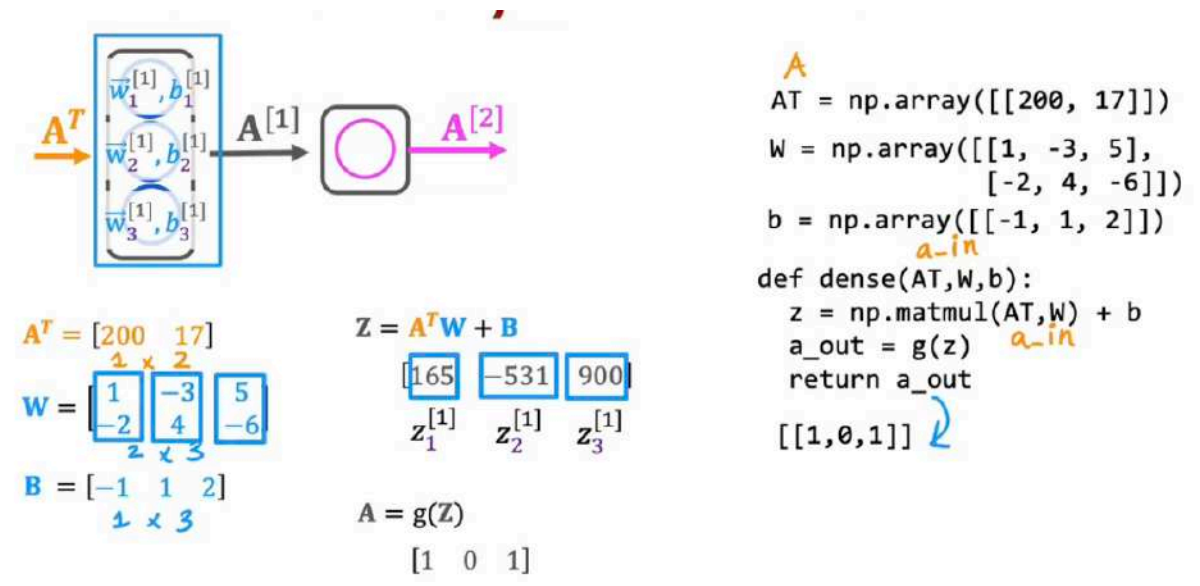
or

```
Z = AT @ W
```

result

```
[[11,17,23,9],
 [-11,-17,-23,-9],
 [1.1,1.7,2.3,0.9]
]
```

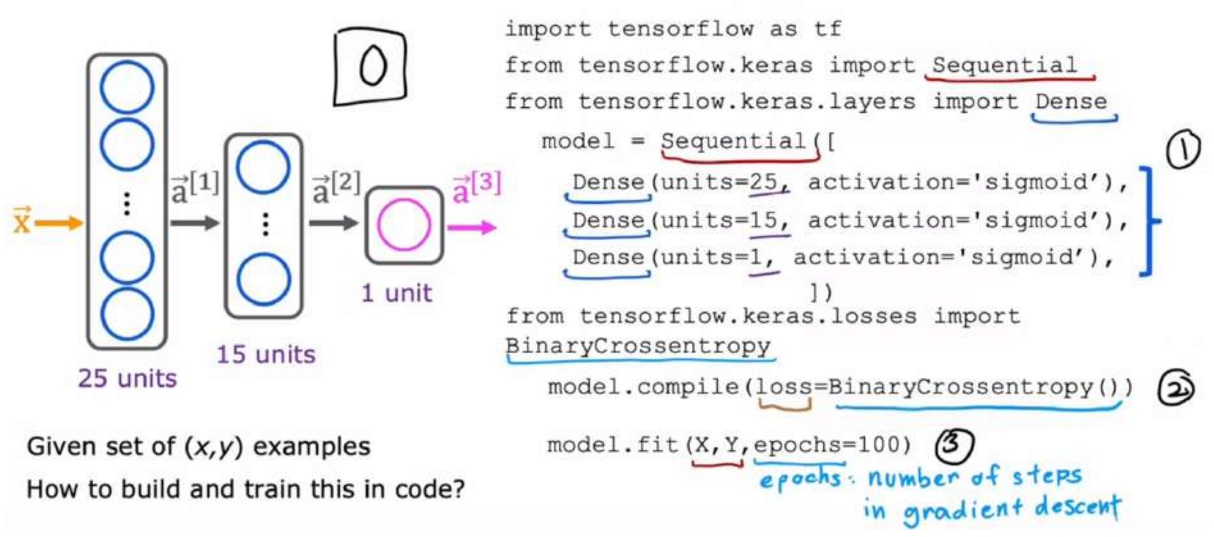
Dense layer vectorized



A^T
$\vec{w}_1^{[1]}, b_1^{[1]}$
$\vec{w}_2^{[1]}, b_2^{[1]}$
$\vec{w}_3^{[1]}, b_3^{[1]}$
$A^{[1]}$
$A^{[2]}$
$A^T = [200 \quad 17]$
1×2
$W = \begin{bmatrix} 1 & -3 & 5 \\ -2 & 4 & -6 \end{bmatrix}$
2×3
$B = [-1 \quad 1 \quad 2]$
1×3
$Z = A^T W + B$
$[165 \quad -531 \quad 900]$
$z_1^{[1]} \quad z_2^{[1]} \quad z_3^{[1]}$
$A = g(Z)$
$[1 \quad 0 \quad 1]$
A
a_in
a_in

```
AT = np.array([[200, 17]])
W = np.array([[1, -3, 5],
              [-2, 4, -6]])
b = np.array([[-1, 1, 2]])
def dense(AT,W,b):
  z = np.matmul(AT,W) + b
  a_out = g(z)
  return a_out
[[1,0,1]]
```

<u>Train a Neural Network in TensorFlow</u>



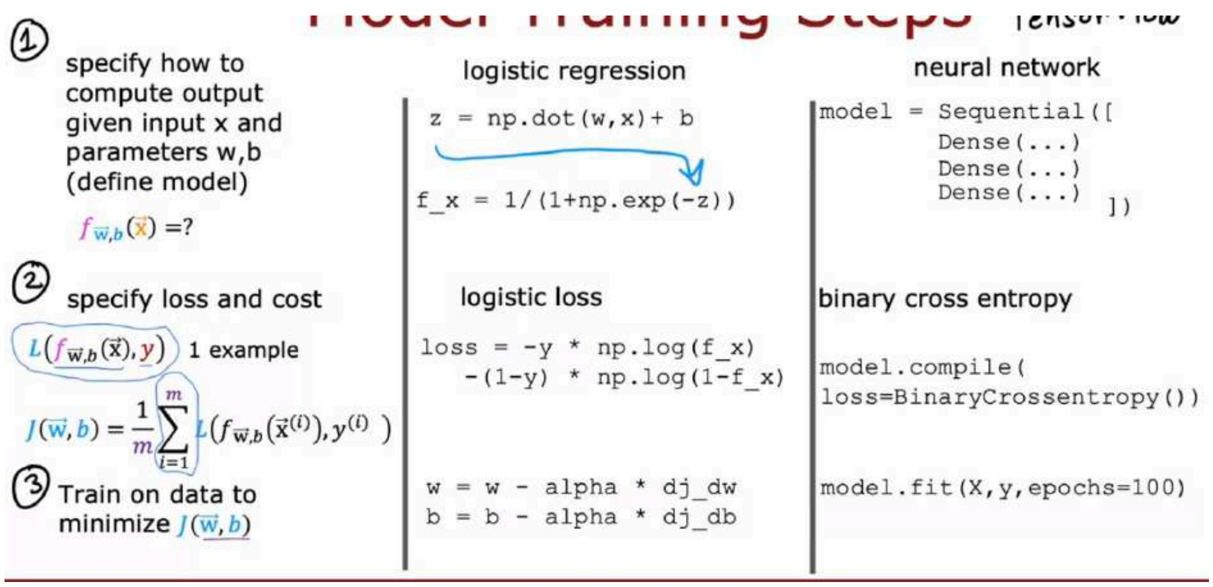
```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
  model = Sequential([
    Dense(units=25, activation='sigmoid'),
    Dense(units=15, activation='sigmoid'),
    Dense(units=1, activation='sigmoid'),
                ])
from tensorflow.keras.losses import
BinaryCrossentropy
  model.compile(loss=BinaryCrossentropy())

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

Given set of (x,y) examples
How to build and train this in code?

In detail

e.g.1. how to train a logistic regression model
1) Specify input to output function
2) Specify loss and cost functions (to see how well logistic regression is doing on a single training example x,y)
3) Use gradient descent to minimise J(w,b)

Same three steps used in training a neural network in TensorFlow:



	logistic regression	neural network

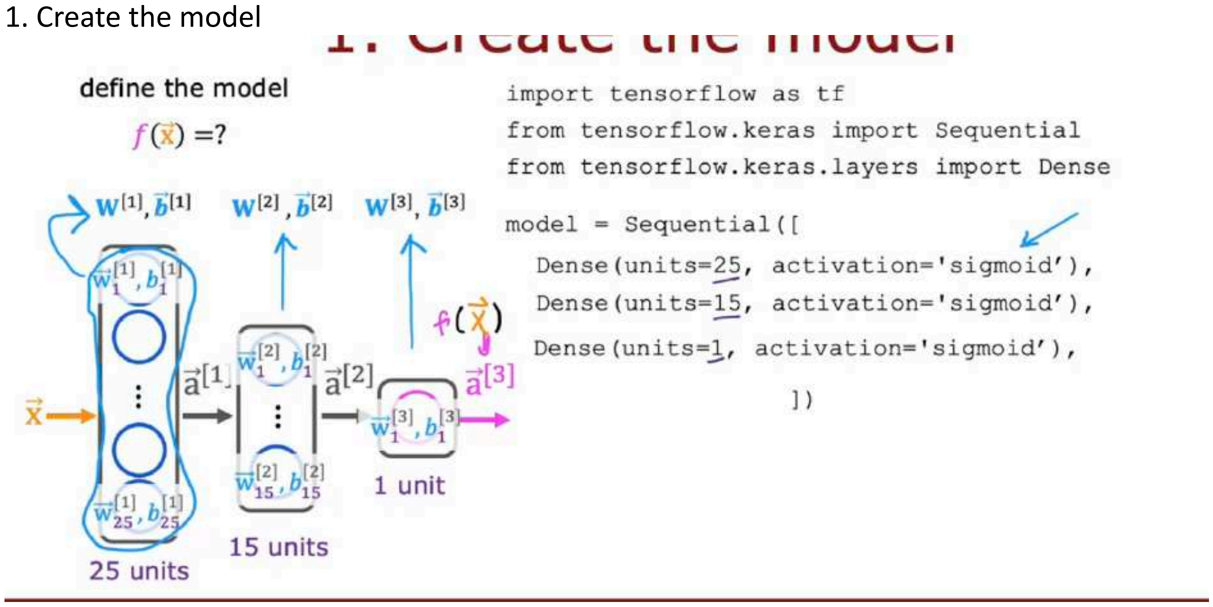
① specify how to compute output given input x and parameters w,b (define model) $f_{\vec{w},b}(\vec{x})=?$	`z = np.dot(w,x)+ b`
	 `f_x = 1/(1+np.exp(-z))`
		`model = Sequential([`
		 `Dense(...)`
		 `Dense(...)`
		 `Dense(...)  ])`

② specify loss and cost
 $L(f_{\vec{w},b}(\vec{x}),y)$ 1 example
 $J(\vec{w},b)=\frac{1}{m}\sum_{i=1}^{m}L(f_{\vec{w},b}(\vec{x}^{(i)}),y^{(i)})$
	logistic loss
	 `loss = -y * np.log(f_x)`
	 `-(1-y) * np.log(1-f_x)`
		binary cross entropy
		 `model.compile(`
		 `loss=BinaryCrossentropy())`

③ Train on data to minimize $J(\vec{w},b)$	`w = w - alpha * dj_dw`
	 `b = b - alpha * dj_db`
		`model.fit(X,y,epochs=100)`

TensorFlow in more detail:

1. Create the model

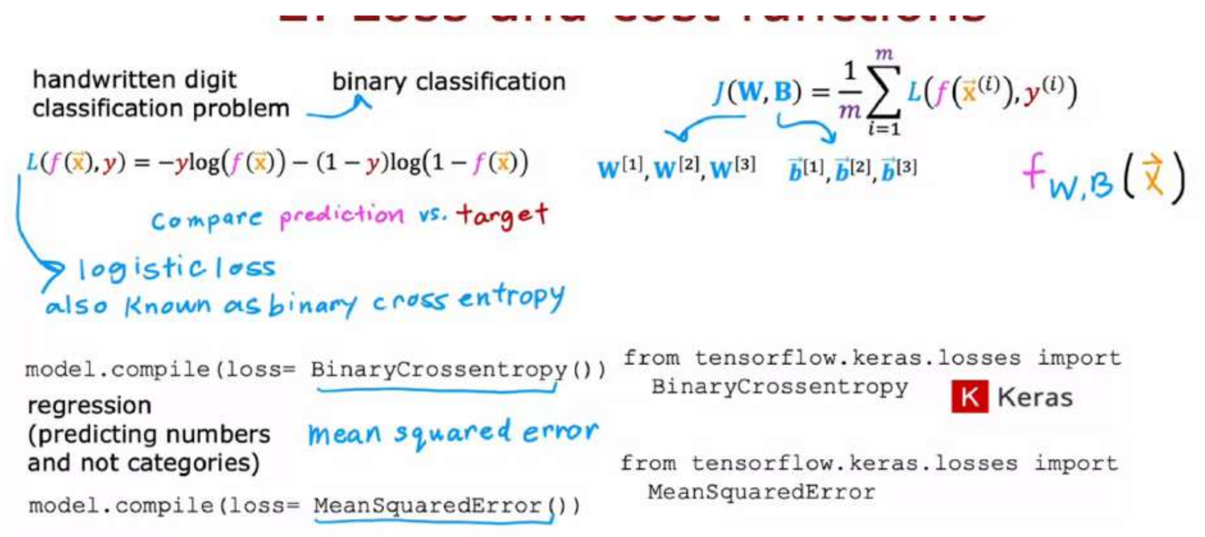


define the model

$f(\vec{x})=?$

```
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'),
                ])
```

2. Loss and cost functions
- Specify loss function, which also defines the cost function
- Can use different loss functions for classification and regression
- In top right, you see cost function



handwritten digit classification problem → binary classification

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

$\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{W}^{[3]}$ $\vec{b}^{[1]}, \vec{b}^{[2]}, \vec{b}^{[3]}$ $f_{W,B}(\vec{x})$

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

Compare prediction vs. target

logistic loss
also known as binary cross entropy

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

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

Keras

regression
(predicting numbers and not categories)

mean squared error

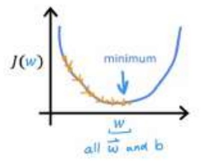
```
model.compile(loss= MeanSquaredError())
```

```
from tensorflow.keras.losses import
  MeanSquaredError
```

3. Gradient descent
- Minimise cost function
- Compute derivatives using back propagation, using model.fit

3.Gradient Descent

- Minimise cost function
- Compute derivatives using back propagation, using model.fit (X,y,epochs = 100)



repeat {

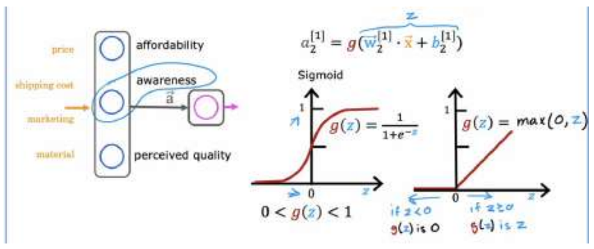
$$w_j^{[l]} = w_j^{[l]} - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

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

}

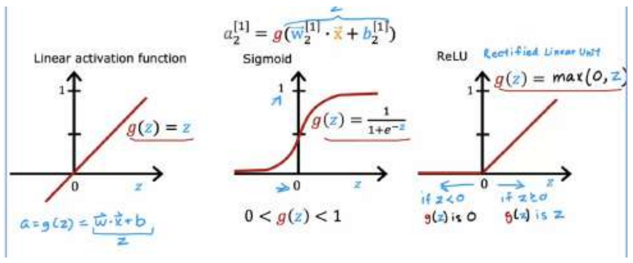
Alternatives to the Sigmoid Acivation

- Recall demand prediction example:



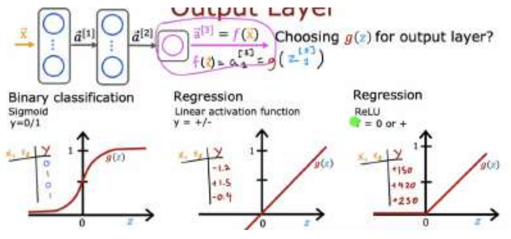
Using sigmoid assumes awareness is binary (aware or not). But it is unlikely to be binary - we can model it as a non-negative number, from 0 up to a very large value. Here, we can use a ReLU (Rectified Linear Unit) function.

Most commonly used activation functions:



Choosing Activation Functions

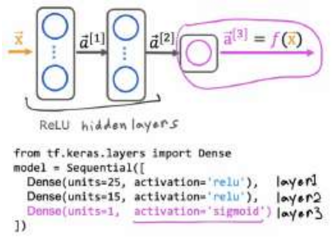
1) Choosing activation function for output layer


Output Layer
$\vec{x}$
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]} = f(\vec{x})$
Choosing $g(z)$ for output layer?
$f(\vec{x}) = a_1^{[3]} = g(z_1^{[3]})$
Binary classification
Sigmoid
y=0/1
Regression
Linear activation function
y = +/-
Regression
ReLU
y = 0 or +

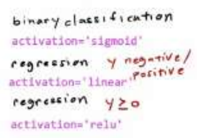
Often, there is a natural choice.
Binary classification problem (y=0 or 1) → Sigmoid
Regression problem e.g. stock price prediction —> Linear activation function
If y can only take non-negative values (house price) → ReLU

2) Hidden Layer

- ReLU generally used, as is easier to compute and goes flat only in one part, compared to two in sigmoid. This results in cost function being flatter, leading to gradient descent being slower



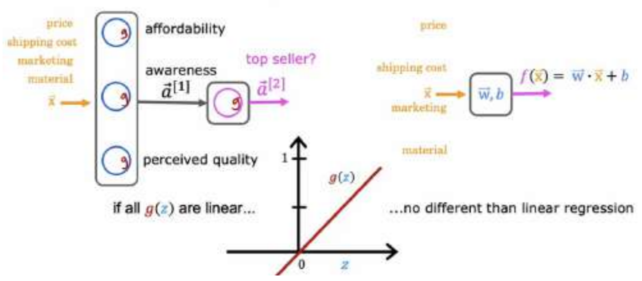
```
from tf.keras.layers import Dense
model = Sequential([
  Dense(units=25, activation='relu'),    layer1
  Dense(units=15, activation='relu'),    layer2
  Dense(units=1,  activation='sigmoid')  layer3
])
```



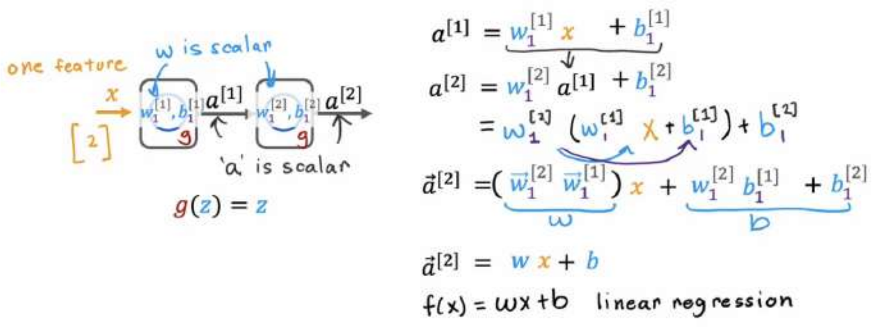
binary classification
activation='sigmoid'
regression y negative/positive
activation='linear'
regression y ≥ 0
activation='relu'

Why do we need activation functions?

If we used a linear activation for all nodes in this neural network, it will become no different from just linear regression. This would defeat the point of using a neural network.



Simpler example: a linear fn of a linear fn is itself a linear fn



$$a^{[1]} = \underbrace{w_1^{[1]} x + b_1^{[1]}}$$

$$a^{[2]} = w_1^{[2]} a^{[1]} + b_1^{[2]}$$

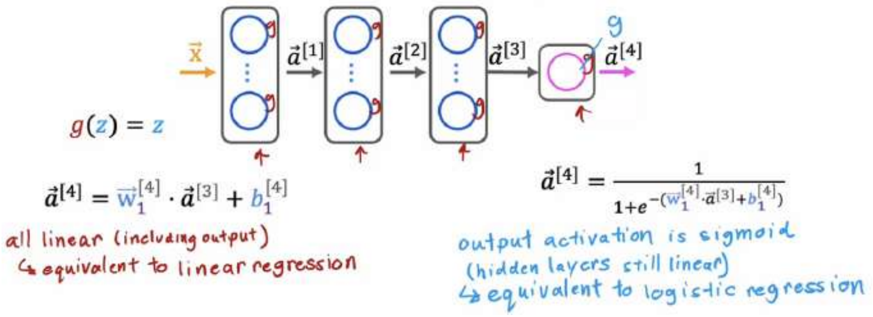
$$= w_1^{[2]} (w_1^{[1]} x + b_1^{[1]}) + b_1^{[2]}$$

$$\vec{a}^{[2]} = (\underbrace{\vec{w}_1^{[2]} \vec{w}_1^{[1]}}_{w}) x + \underbrace{w_1^{[2]} b_1^{[1]} + b_1^{[2]}}_{b}$$

$$\vec{a}^{[2]} = w x + b$$

$$f(x) = wx + b$$ linear regression

With hidden layers, if all are linear activation fns, you get linear regression. If the output activation is sigmoid, with hidden layers still linear, you just have logistic regression.


g(z) = z
all linear (including output)
↳ equivalent to linear regression
output activation is sigmoid
(hidden layers still linear)
↳ equivalent to logistic regression

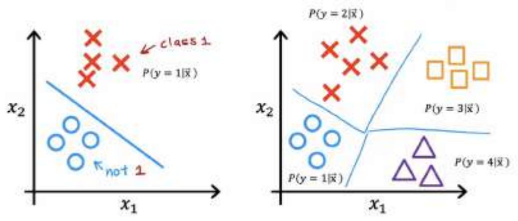
$$\vec{a}^{[4]} = \vec{w}_1^{[4]} \cdot \vec{a}^{[3]} + b_1^{[4]}$$

$$\vec{a}^{[4]} = \frac{1}{1+e^{-(\vec{w}_1^{[4]} \cdot \vec{a}^{[3]} + b_1^{[4]})}}$$

Rule of thumb: **Don't use linear activations in hidden layers (use ReLU)**

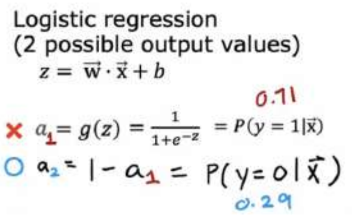
Multiclass Classification

- Classification problem with more than two possible outputs
- With two-class classification, can use logistic regression to estimate decision boundary (softmax); more complex if multiclass



Softmax

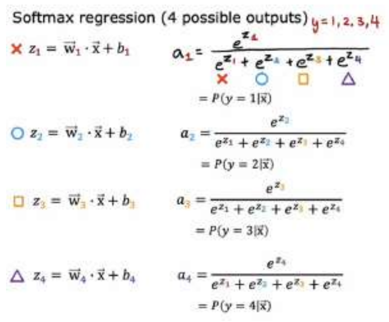
Logistic regression applies when y can take on either 0 or 1 as output. (Softmax is a generalisation of logistic regression - if you do softmax on 2 possible outputs, it reduces to logistic regression).



Logistic regression
(2 possible output values)
$z = \vec{w} \cdot \vec{x} + b$
0.71
✕ $a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$
○ $a_2 = 1 - a_1 = P(y=0|\vec{x})$
0.29

Generalising this to softmax, where y can take on 4 possible outputs:

- a1 is interpreted as the estimate of the chance of y = 1, given the input features x; a2 is the estimated chance of y=2, etc.



Softmax regression (4 possible outputs) y=1, 2, 3, 4
✕ $z_1 = \vec{w}_1 \cdot \vec{x} + b_1$
$a_1 = \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=1|\vec{x})$
✕ ○ □ △
○ $z_2 = \vec{w}_2 \cdot \vec{x} + b_2$
$a_2 = \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=2|\vec{x})$
□ $z_3 = \vec{w}_3 \cdot \vec{x} + b_3$
$a_3 = \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=3|\vec{x})$
△ $z_4 = \vec{w}_4 \cdot \vec{x} + b_4$
$a_4 = \frac{e^{z_4}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} = P(y=4|\vec{x})$

Generalising this to N possible outputs where y = 1,2,3,...N:

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

parameters $w_1, w_2, \ldots, w_N$ $b_1, b_2, \ldots, b_N$

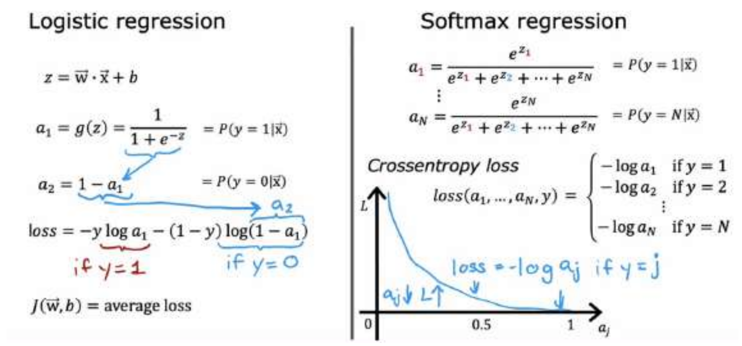
$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y = j|\vec{x})$$

Where aj is the estimate that y = j. Note that a1 + a2 + … + aN = 1

Cost Function

For softmax, the loss for if the algorithm outputs y =1, the loss is the negative log of the probability that y=1 according to algorithm. I.e. if y = j, loss = -logaj.

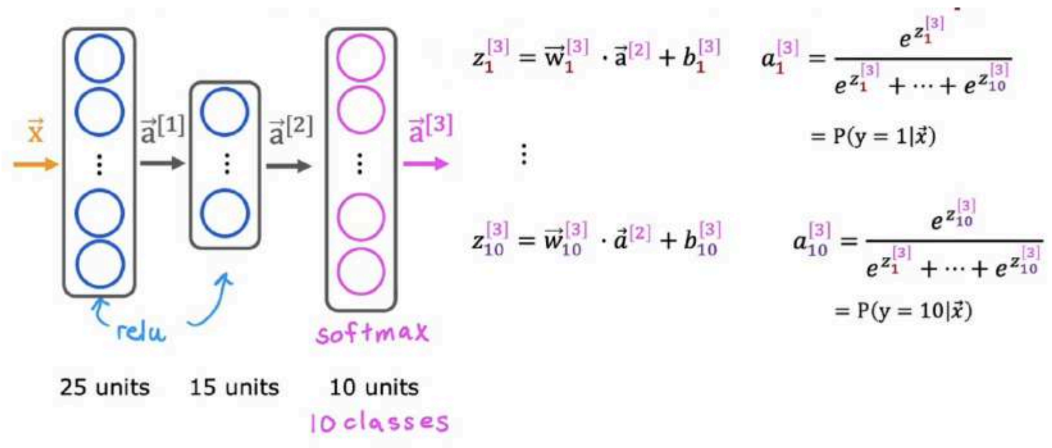
- If aj very close to 1, loss is very small - smaller aj means bigger loss, incentivising the algorithm to make aj as close to 1 as possible - we want the chance of y being the value said by the algorithm to be large.
- As y can only take on one value in every training example, end up computing -logaj only for one value of aj (whatever the actual value of y = j in that training example e.g. if y=2, computer -loga2).



Logistic regression
$z = \vec{w} \cdot \vec{x} + b$
$a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$
$a_2 = 1 - a_1 = P(y=0|\vec{x})$
a_2
$\text{loss} = -y \log a_1 - (1-y)\log(1-a_1)$
if y=1
if y=0
$J(\vec{w}, b) = \text{average loss}$
Softmax regression
$a_1 = \frac{e^{z_1}}{e^{z_1}+e^{z_2}+\cdots+e^{z_N}} = P(y=1|\vec{x})$
$a_N = \frac{e^{z_N}}{e^{z_1}+e^{z_2}+\cdots+e^{z_N}} = P(y=N|\vec{x})$
Crossentropy loss
$loss(a_1, \ldots, a_N, y) = \begin{cases} -\log a_1 & \text{if } y=1 \\ -\log a_2 & \text{if } y=2 \\ \vdots \\ -\log a_N & \text{if } y=N \end{cases}$
L
aj↓ L↑
loss = -log aj if y=j
0
0.5
1
a_j

Neural Network with Softmax output (for multiclass classification)

P(y=1|x) refers to chance y is equal to 1.



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

$$\vdots$$

$$z_{10}^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]} \qquad a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y=10|\vec{x})$$

Note that each a in softmax depends on all z, not just a1 depending on z1 as in logistic regression:

logistic regression

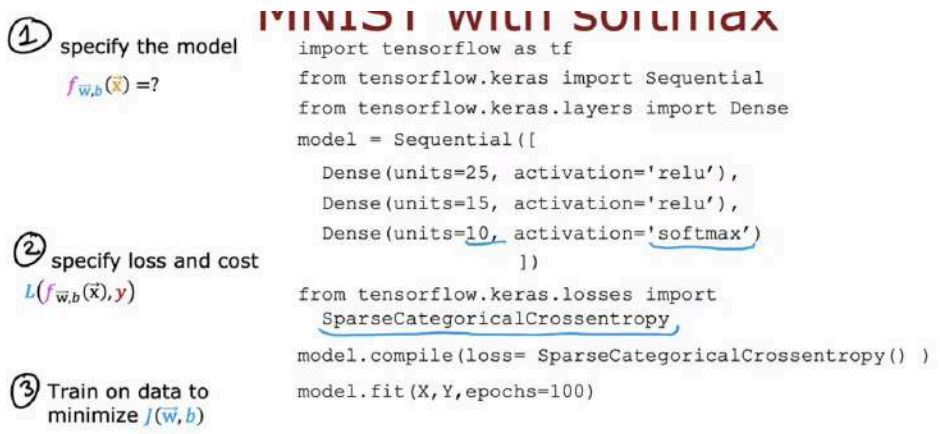
$$a_1^{[3]} = g(z_1^{[3]}) \quad a_2^{[3]} = g(z_2^{[3]})$$

softmax

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

Implementation in TensorFlow (3 steps)

- NB for softmax, use a different crossentropy fn (compared to binary crossentropy in logistic regression)



MNIST with softmax

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

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

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

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

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

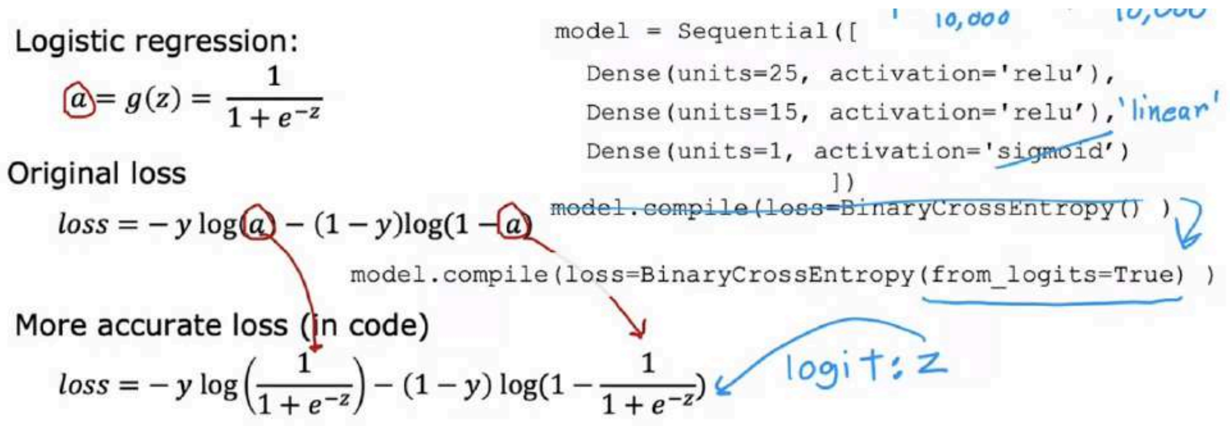
```
model.fit(X,Y,epochs=100)
```

NB there is a better version written later.

Improved Implementation of SoftMax

In Logistic regression:

- To compute loss function, first compute a and then compute (binary crossentropy) loss:



Logistic regression:

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

Original loss

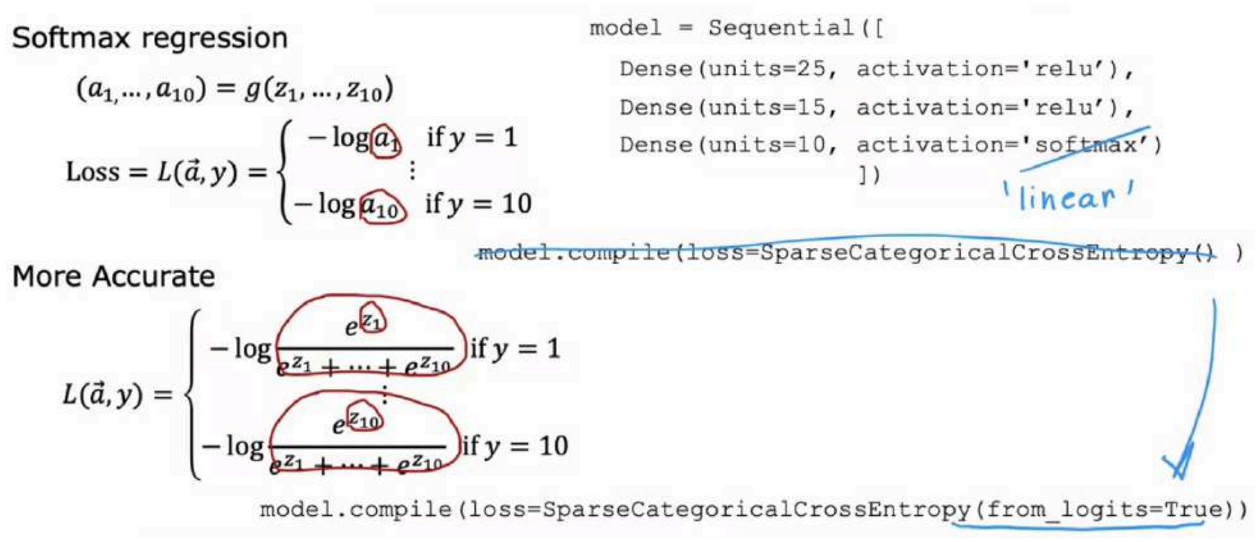
$$loss = -y\log(\boxed{a}) - (1-y)\log(1-\boxed{a})$$

More accurate loss (in code)

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

```
model = Sequential([                                   10,000        10,000
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'), 'linear'
  Dense(units=1, activation='~~sigmoid~~')
                   ])
~~model.compile(loss=BinaryCrossEntropy()~~ )
model.compile(loss=BinaryCrossEntropy(from_logits=True) )
```

This reduces numerical roundoff errors. Applying this to softmax, instead specify formula to give tensorflow ability to rearrange terms and calculate more accurately…



Softmax regression

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

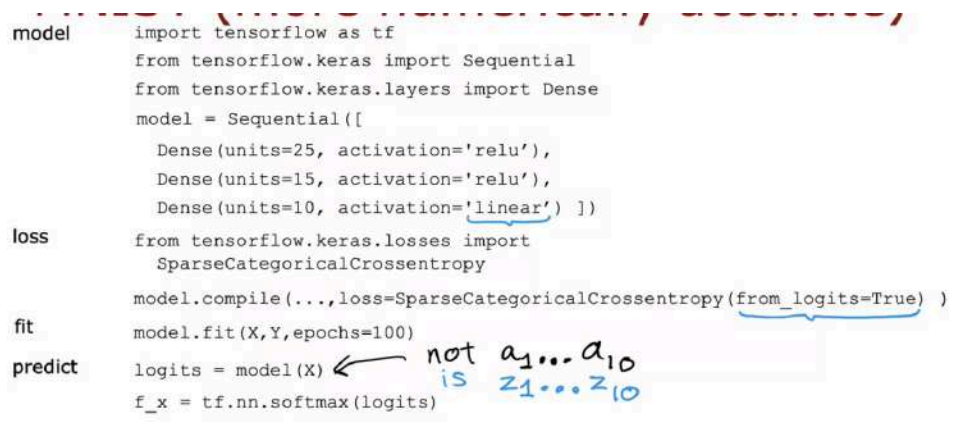
$$\text{Loss} = L(\vec{a}, y) = \begin{cases} -\log \boxed{a_1} & \text{if } y = 1 \\ \vdots & \\ -\log \boxed{a_{10}} & \text{if } y = 10 \end{cases}$$

More Accurate

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

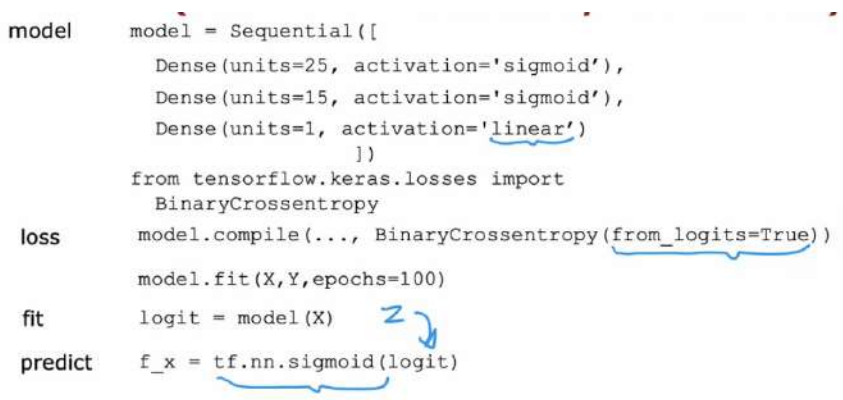
```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='~~softmax~~')
                    ])
                         'linear'
~~model.compile(loss=SparseCategoricalCrossEntropy()~~ )

model.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))
```

We have now changed the neural network to use a linear activation function rather than softmax, so the neural network's final layer no longer outputs a1…a10 but rather z1…z10.



```
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
          model.compile(...,loss=SparseCategoricalCrossentropy(from_logits=True) )
fit       model.fit(X,Y,epochs=100)
predict   logits = model(X) <— not a1... a10
                               is z1... z10
          f_x = tf.nn.softmax(logits)
```

Similarly, for logistic regression have to change the code…



```
model      model = Sequential([
             Dense(units=25, activation='sigmoid'),
             Dense(units=15, activation='sigmoid'),
             Dense(units=1, activation='linear')
                                    ])
           from tensorflow.keras.losses import
             BinaryCrossentropy
loss       model.compile(..., BinaryCrossentropy(from_logits=True))

           model.fit(X,Y,epochs=100)
fit        logit = model(X)        z
predict    f_x = tf.nn.sigmoid(logit)
```

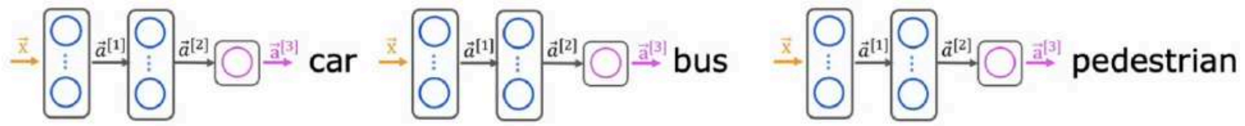
Classification with multiple outputs (multilabel)

- Could be multiple labels associated with each image e.g. is there a car? Is there a bus? Is there a pedestrian? (associated with a single input image x, there are three possible labels)

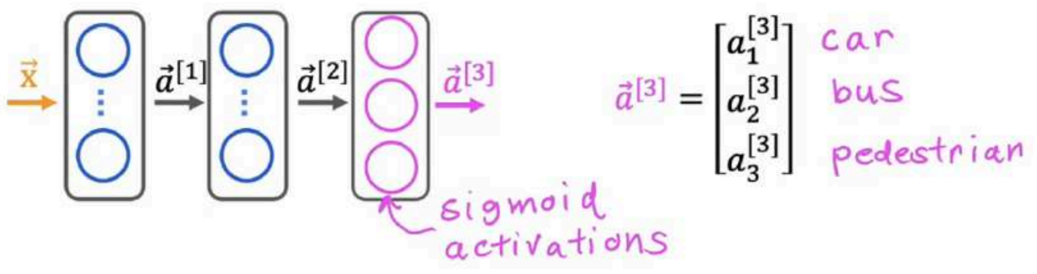
$y = \begin{bmatrix}1\\0\\1\end{bmatrix}$ $y = \begin{bmatrix}0\\0\\1\end{bmatrix}$ $y = \begin{bmatrix}1\\1\\0\end{bmatrix}$

Is there a car?	yes	no	yes
Is there a bus?	no	no	yes
Is there a pedestrian	yes	yes	no

Could do three separate neural networks…



Alternatively, train one neural network with three outputs

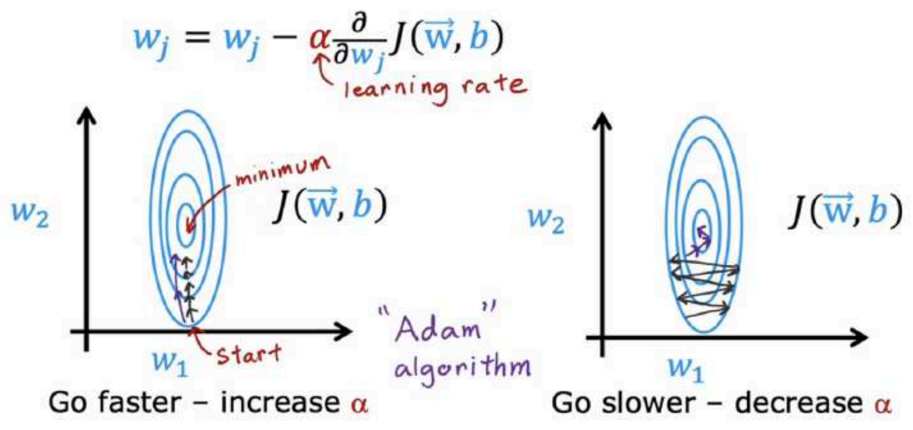


$$\vec{a}^{[3]} = \begin{bmatrix}a_1^{[3]}\\a_2^{[3]}\\a_3^{[3]}\end{bmatrix}$$ car, bus, pedestrian

NOT THE SAME AS MULTICLASS CLASSIFICATION.

Advanced Optimisation

- There exist even better training models than gradient descent. “Adam algorithm” automatically adjusts learning rate to increase or decrease alpha, optimising grad. descent

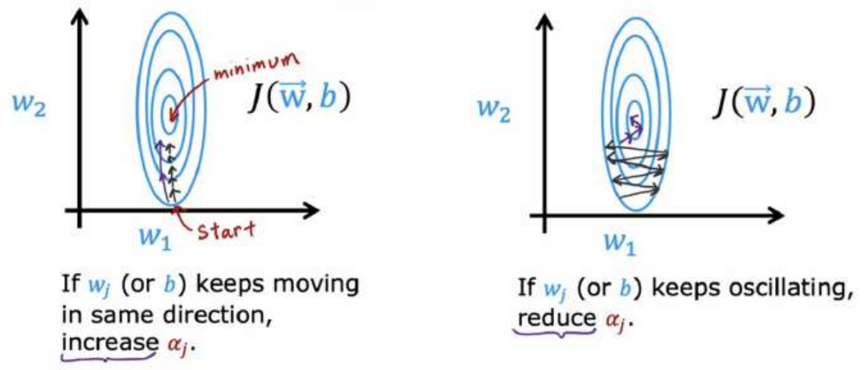

$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$
learning rate
w_2
minimum
$J(\vec{w}, b)$
start
w_1
"Adam" algorithm
Go faster – increase α
w_2
$J(\vec{w}, b)$
w_1
Go slower – decrease α

- It does not have just one alpha, uses a different learning rate for every parameter of your model:

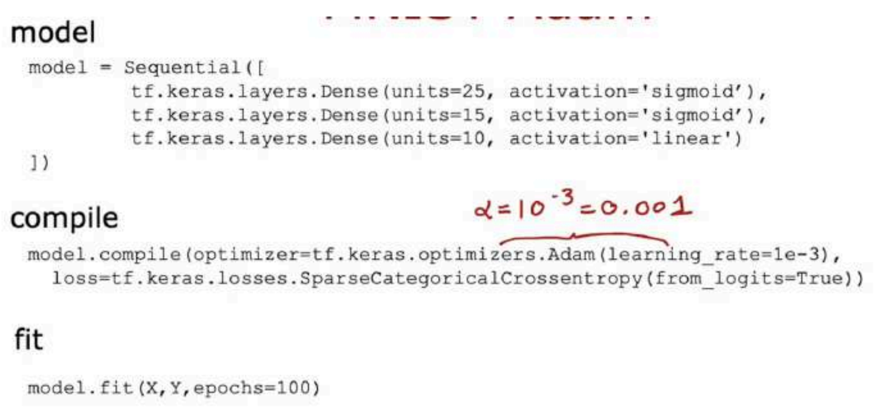
Adam: Adaptive Moment estimation not just one α

$$w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$
$$\vdots$$
$$w_{10} = w_{10} - \alpha_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$
$$b = b - \alpha_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$

- Intuition of adam algorithm:


w_2
minimum
$J(\vec{w}, b)$
w_1
start
If w_j (or b) keeps moving in same direction, increase α_j.
w_2
$J(\vec{w}, b)$
w_1
If w_j (or b) keeps oscillating, reduce α_j.

- Implementing this in code: (VERY STANDARD FOR TRAINING NETWORKS)



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

$\alpha=10^{-3}=0.001$

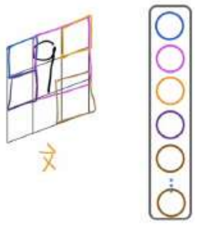
```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

fit

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

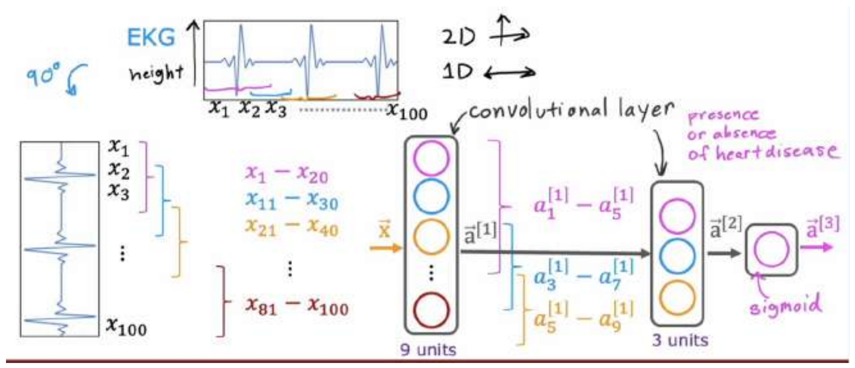
Most practitioners will use Adam rather than gradient descent.

Additional Layer Types



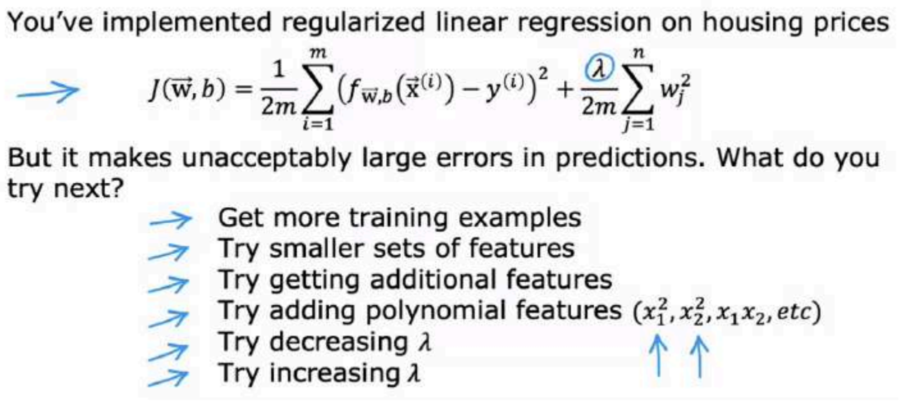
- All so far have been **dense** layers - each neuron output is a function of all the activation outputs of the previous layer.
- **Convolutional layer** - Each neuron only looks at part of the previous layer's outputs. Leads to faster computation + needs less training data (less prone to overfitting)

E.g. for looking at a heart beat signal, where the different hidden units in the convolutional layer each look at a different part of the signal. Then in the second layer only looks at a certain number of the activations.



Week 3

1) *Debugging a learning algorithm*


You've implemented regularized linear regression on housing prices

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

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

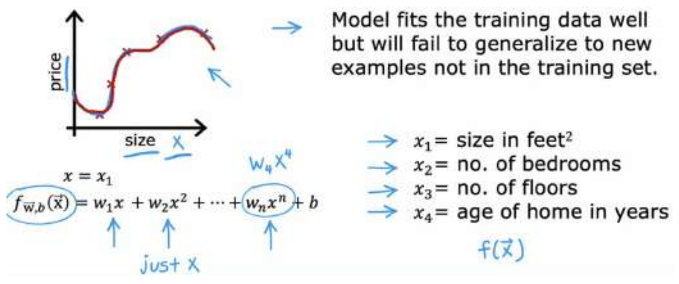
- $\rightarrow$ Get more training examples
- $\rightarrow$ Try smaller sets of features
- $\rightarrow$ Try getting additional features
- $\rightarrow$ Try adding polynomial features $(x_1^2, x_2^2, x_1 x_2, etc)$
- $\rightarrow$ Try decreasing λ $\uparrow \uparrow$
- $\rightarrow$ Try increasing λ

Machine Learning Diagnostics

Evaluating a model

E.g. housing prices

If we have two features, easy to just plot - here, we see that line is v. curvy so will not generalise well. With four features etc., however, cannot plot easily so need another systematic way to evaluate strength of model.



$\rightarrow$ Model fits the training data well but will fail to generalize to new examples not in the training set.

$x = x_1$

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

$w_4 x^4$

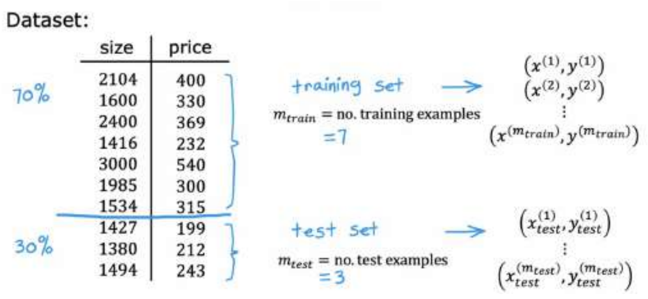
just x

- $\rightarrow$ x_1 = size in feet²
- $\rightarrow$ x_2 = no. of bedrooms
- $\rightarrow$ x_3 = no. of floors
- $\rightarrow$ x_4 = age of home in years

$f(\vec{x})$

Solution:

1) Use a training and test set


Dataset:

	size	price
70%	2104	400
	1600	330
	2400	369
	1416	232
	3000	540
	1985	300
	1534	315
30%	1427	199
	1380	212
	1494	243

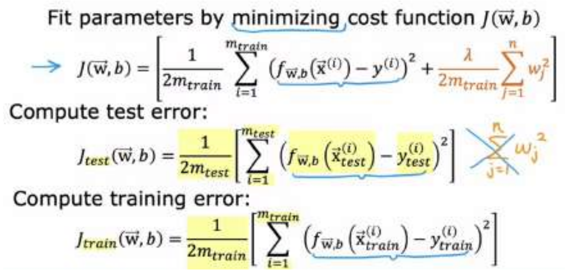
training set $\rightarrow$ $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$, $\vdots$, $(x^{(m_{train})}, y^{(m_{train})})$

m_{train} = no. training examples $= 7$

test set $\rightarrow$ $(x_{test}^{(1)}, y_{test}^{(1)})$, $\vdots$, $(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

m_{test} = no. test examples $= 3$

2) Train/ test procedure for linear regression (with squared error cost)
   - Training error is how well algorithm is doing on training set


Fit parameters by minimizing cost function $J(\vec{w}, b)$

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

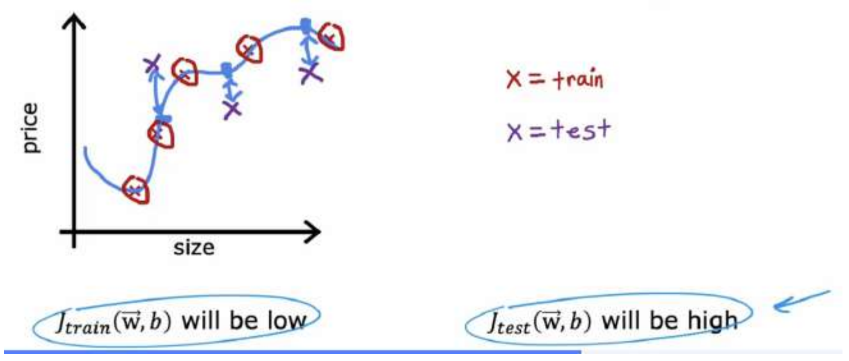
Compute test error:

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

Compute training error:

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

- Value of Jtest is a good measure of how well model is generalising. E.g. if it fits training set sell but generalises poorly, Jtrain will be low but Jtest will be high:



How do we apply this to a classification problem?

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

E.g.,

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

Compute test error:

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

Compute train error:

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

But more commonly for classification problems, instead of using logistic loss to measure error, instead measure what is the **fraction of the test set and the fraction of the train set that the algorithm has misclassified.**

$$\hat{y} = \begin{cases} 1 \text{ if } f_{\vec{w},b}(\vec{x}^{(i)}) \geq 0.5 \\ 0 \text{ if } f_{\vec{w},b}(\vec{x}^{(i)}) < 0.5 \end{cases}$$

count $\hat{y} \neq y$

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

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

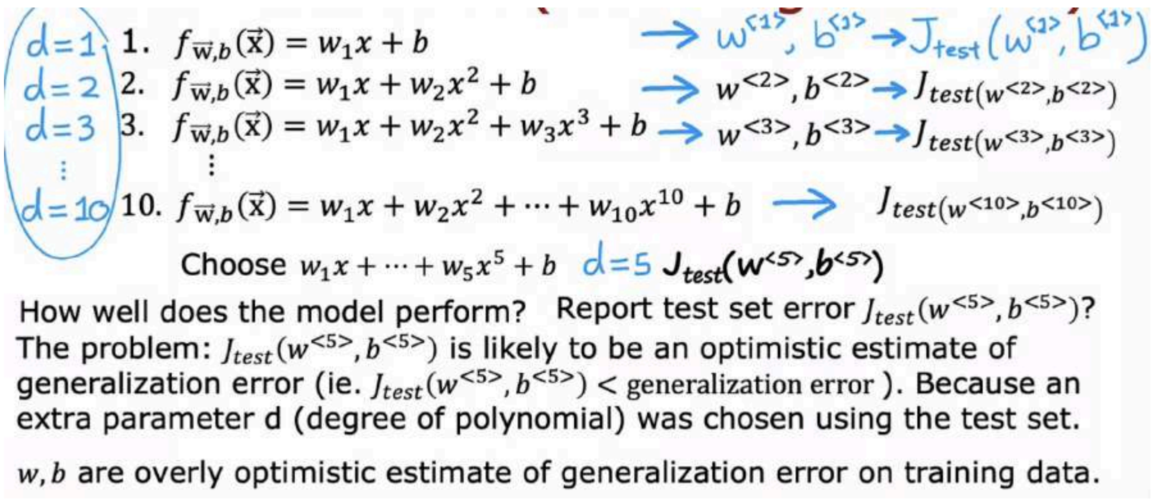
I.e. algo makes a prediction 1 or 0 on each example; count up number of examples where y hat is not equal to the correct output y.

<u>Model selection and training/ cross validation/ test sets</u>

How can we automatically choose a model?

- Once parameters w,b are fit to training set, the training error Jtrain is likely lower than the actual generalisation error. Thus, Jtest is a better estimate of how well the model will generalise to new data compared to Jtrain

Possible Option: Look at all Jtest and see which gives lowest value



$d=1$ 1. $f_{\vec{w},b}(\vec{x}) = w_1x + b \rightarrow w^{<1>}, b^{<1>} \rightarrow J_{test}(w^{<1>}, b^{<1>})$

$d=2$ 2. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + b \rightarrow w^{<2>}, b^{<2>} \rightarrow J_{test}(w^{<2>}, b^{<2>})$

$d=3$ 3. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + w_3x^3 + b \rightarrow w^{<3>}, b^{<3>} \rightarrow J_{test}(w^{<3>}, b^{<3>})$

$\vdots$

$d=10$ 10. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + \cdots + w_{10}x^{10} + b \rightarrow J_{test}(w^{<10>}, b^{<10>})$

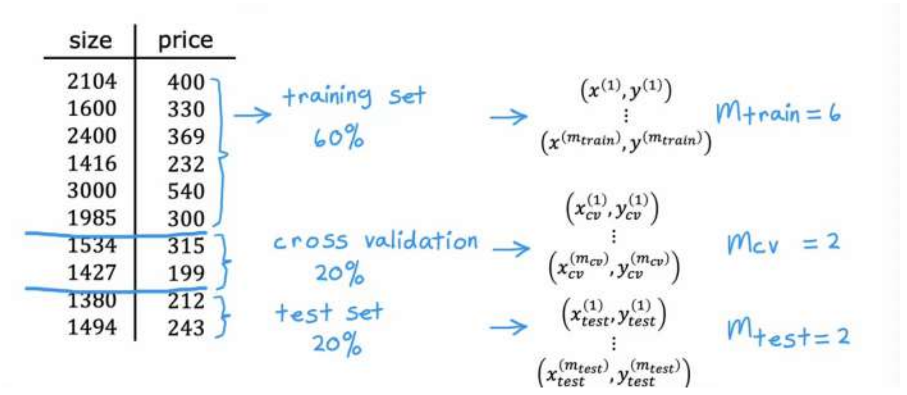
Choose $w_1x + \cdots + w_5x^5 + b$ $d=5$ $J_{test}(w^{<5>}, b^{<5>})$

How well does the model perform? Report test set error $J_{test}(w^{<5>}, b^{<5>})$?
The problem: $J_{test}(w^{<5>}, b^{<5>})$ is likely to be an optimistic estimate of generalization error (ie. $J_{test}(w^{<5>}, b^{<5>}) <$ generalization error). Because an extra parameter d (degree of polynomial) was chosen using the test set.

w, b are overly optimistic estimate of generalization error on training data.

- Have essentially fitted an extra parameter using the test set. Thus, this is a flawed method.

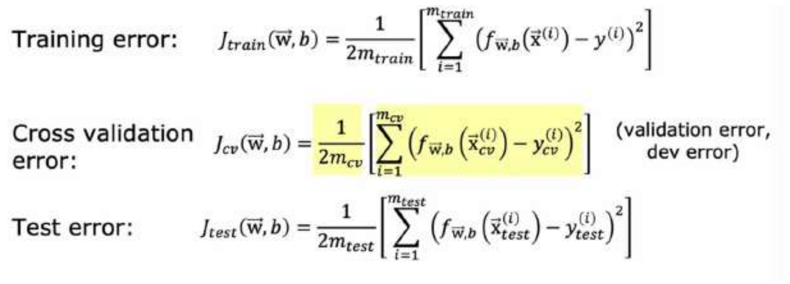
Modification of procedure:
- Split into three: training set, cross validation (/development/dev/validation) set, and test set



size	price
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199
1380	212
1494	243

training set
60%
$(x^{(1)}, y^{(1)}) \vdots (x^{(m_{train})}, y^{(m_{train})})$
$m_{train} = 6$
cross validation
20%
$(x_{cv}^{(1)}, y_{cv}^{(1)}) \vdots (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$
$m_{cv} = 2$
test set
20%
$(x_{test}^{(1)}, y_{test}^{(1)}) \vdots (x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$
$m_{test} = 2$

- Then compute the training error, cross validation error, and test error:

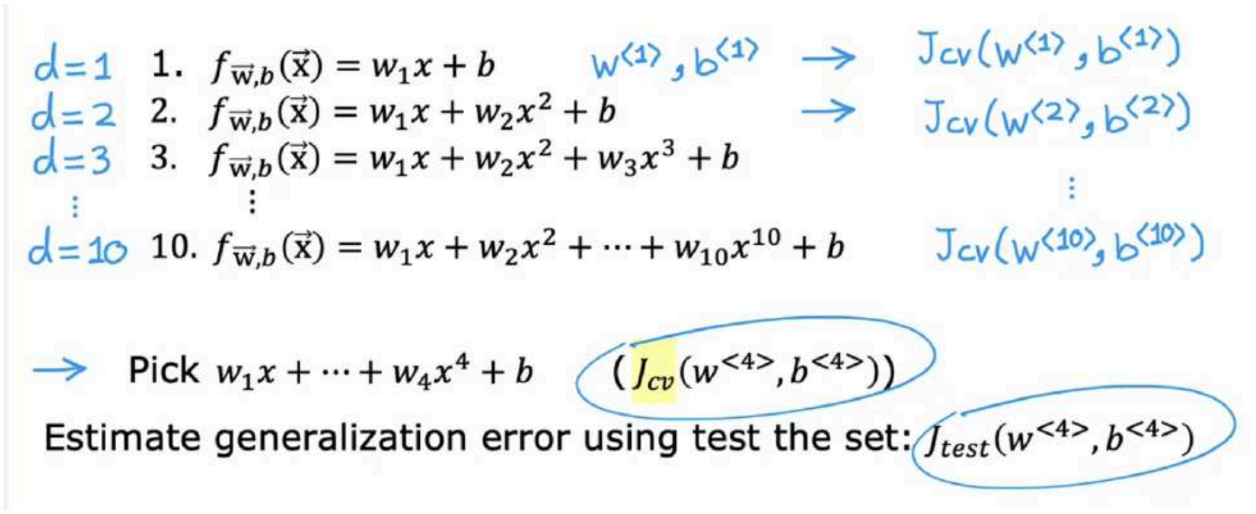


Training error: $$J_{train}(\vec{w}, b) = \frac{1}{2m_{train}}\left[\sum_{i=1}^{m_{train}} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2\right]$$

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

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

- Then can go about model selection, using the cross validation set to see which model has the lowest cross validation error:



$d=1$ 1. $f_{\vec{w},b}(\vec{x}) = w_1x + b$ $\quad w^{\langle 1\rangle}, b^{\langle 1\rangle} \rightarrow J_{cv}(w^{\langle 1\rangle}, b^{\langle 1\rangle})$

$d=2$ 2. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + b$ $\quad \rightarrow J_{cv}(w^{\langle 2\rangle}, b^{\langle 2\rangle})$

$d=3$ 3. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + w_3x^3 + b$

$\vdots$

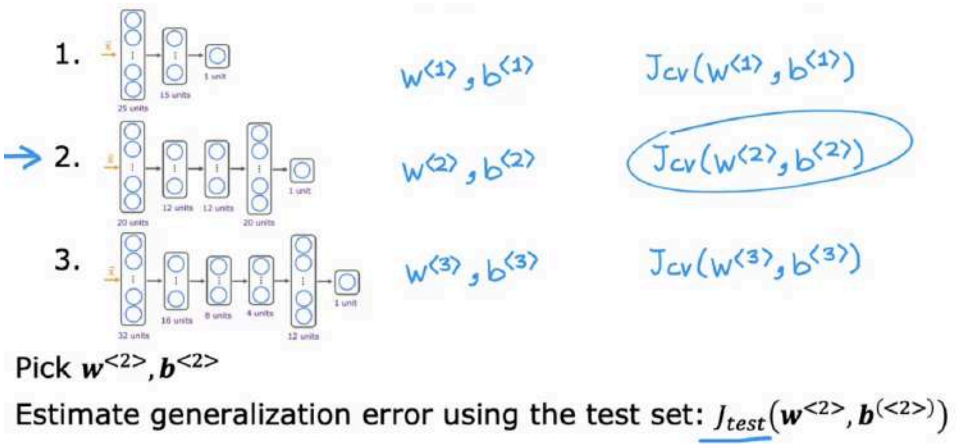
$d=10$ 10. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + \cdots + w_{10}x^{10} + b$ $\quad J_{cv}(w^{\langle 10\rangle}, b^{\langle 10\rangle})$

$\rightarrow$ Pick $w_1x + \cdots + w_4x^4 + b$ $\quad (J_{cv}(w^{<4>}, b^{<4>}))$

Estimate generalization error using test the set: $J_{test}(w^{<4>}, b^{<4>})$

- Jtest here is a fair estimate of the generalisation error because up until this point you have not fit w, b, or d to the test set

This model selection also applies to neural network architecture selection…



1. $w^{\langle 1\rangle}, b^{\langle 1\rangle}$ $\quad J_{cv}(w^{\langle 1\rangle}, b^{\langle 1\rangle})$

$\rightarrow$ 2. $w^{\langle 2\rangle}, b^{\langle 2\rangle}$ $\quad J_{cv}(w^{\langle 2\rangle}, b^{\langle 2\rangle})$

3. $w^{\langle 3\rangle}, b^{\langle 3\rangle}$ $\quad J_{cv}(w^{\langle 3\rangle}, b^{\langle 3\rangle})$

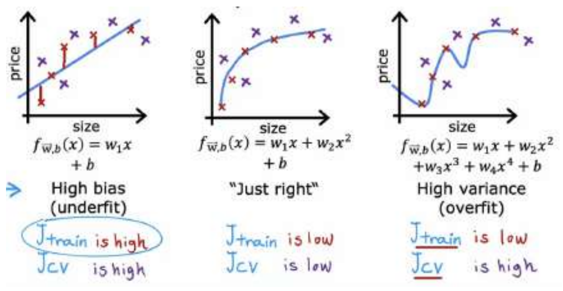
Pick $\mathbf{w}^{<2>}, \mathbf{b}^{<2>}$

Estimate generalization error using the test set: $\underline{J_{test}}(\mathbf{w}^{<2>}, \mathbf{b}^{(<2>)})$

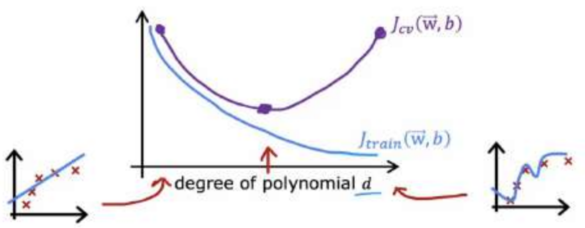
- As this is classification, evaluate using fraction of y hat = y

Diagnosing Bias and Variance

- Good guidance on what to try next when something is working poorly.
- Jtrain & Jcv high → high bias; Jtrain low, Jcv high → high variance



Understand bias and variance:



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 $J_{cv} \gg J_{train}$

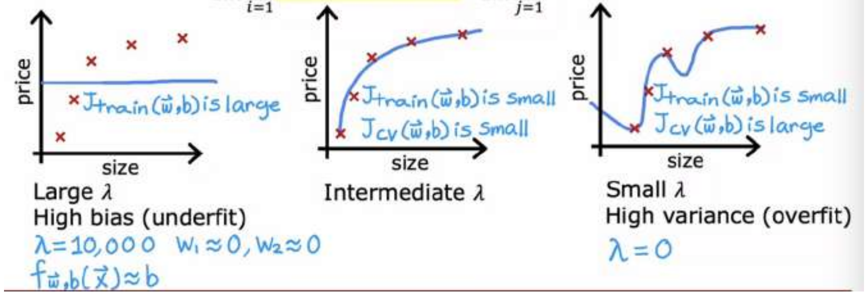
__Regularization and bias/variance__

- How choice of regularisation parameter lamda affects bias and variance
- Trade off of keeping w small (large lambda) and fitting well (low lambda)

Linear regression with regularisation:

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

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

- 

How do we choose a good value of lambda (similar to cross validation)?

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

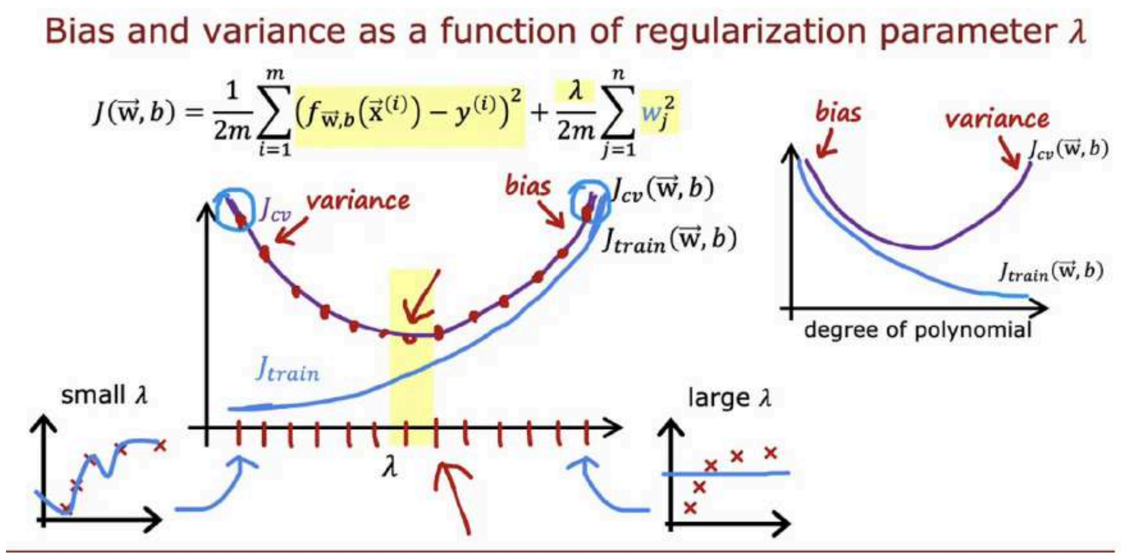
→ 1. Try $\lambda = 0$ → $\min_{\vec{w},b} J(\vec{w}, b)$ → $w^{<1>}, b^{<1>}$ → $J_{cv}(w^{<1>}, b^{<1>})$
→ 2. Try $\lambda = 0.01$ → → $w^{<2>}, b^{<2>}$ → $J_{cv}(w^{<2>}, b^{<2>})$
→ 3. Try $\lambda = 0.02$ → $J_{cv}(w^{<3>}, b^{<3>})$
→ 4. Try $\lambda = 0.04$
→ 5. Try $\lambda = 0.08$ $J_{cv}(w^{<5>}, b^{<5>})$
⋮
→ 12. Try $\lambda \approx \underline{10}$ → $w^{<12>}, b^{<12>}$ → $J_{cv}(w^{<12>}, b^{<12>})$

Pick $\underline{w^{<5>}, b^{<5>}}$
Report test error: $\underline{J_{test}(w^{<5>}, b^{<5>})}$

Looking at how bias and variance vary as a function of lambda:

– Jtrain: will increase, as in optimisation cost fn, the larger lambda, the more it is trying to keep w small, so the worse it does on the training set, so the train error increases

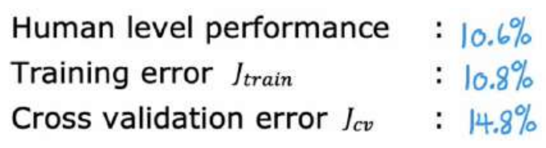
- Jcv overfits on left, and underfits on right


Bias and variance as a function of regularization parameter λ

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

Establishing a baseline level of performance

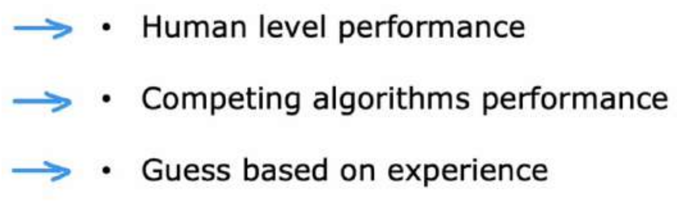
E.g. speech recognition (training error = % of incorrect transcriptions)



Human level performance	:	10.6%
Training error J_{train}	:	10.8%
Cross validation error J_{cv}	:	14.8%

Algo doing quite well on training set, but cross validation higher - so has a variance problem.

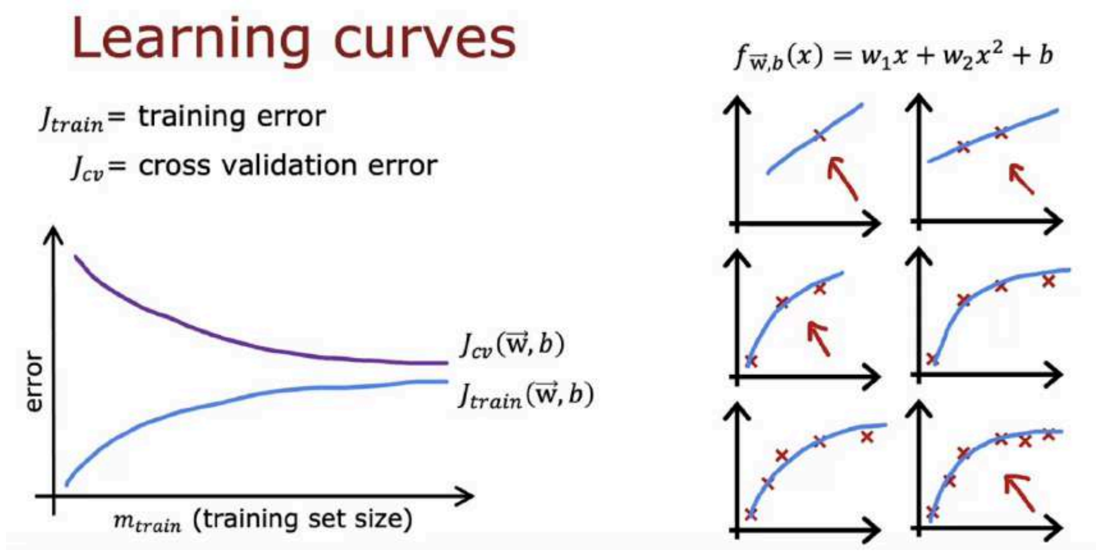
Baseline level of performance:



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

- Gap between baseline and training error large = bias problem
- Gap between training error and cross validation error large = variance problem
- Can have both problems simultaneously

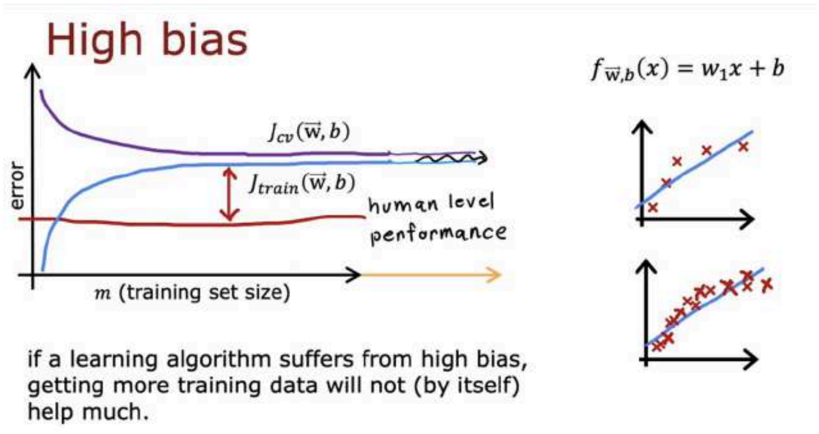
Learning Curves



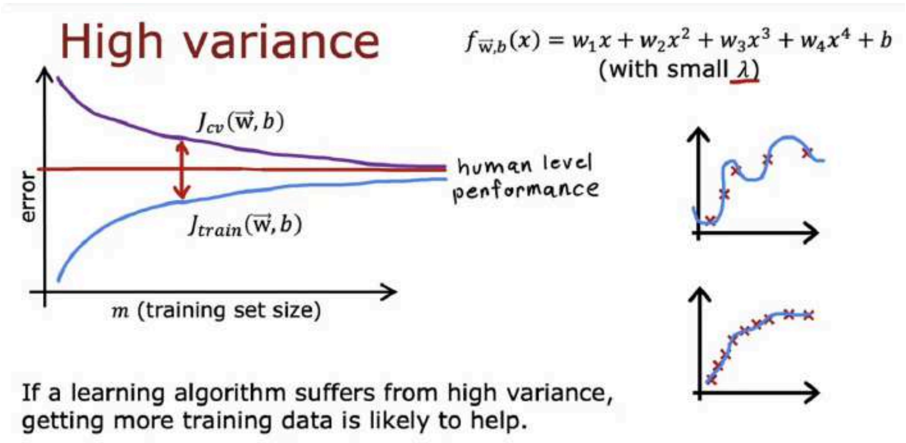
- More training examples → harder to fit all training examples perfectly → Jtrain increases
- Jcv > Jtrain, as expect to do at least slightly better on training set than cross validation set

For high bias:

- Plateaus because fitting a straight line won't change up with more examples - it's such a simple model

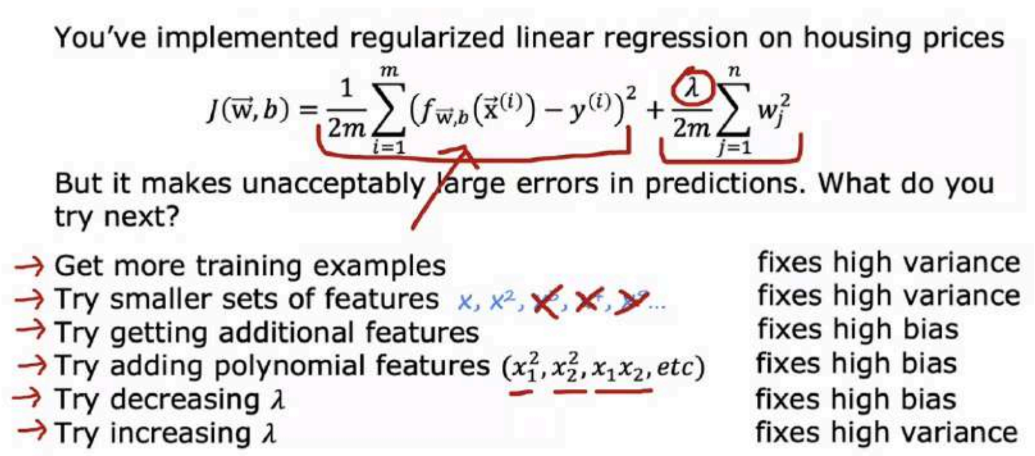


For high variance:



Deciding what to do next

Debugging a learning algorithm



You've implemented regularized linear regression on housing prices

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

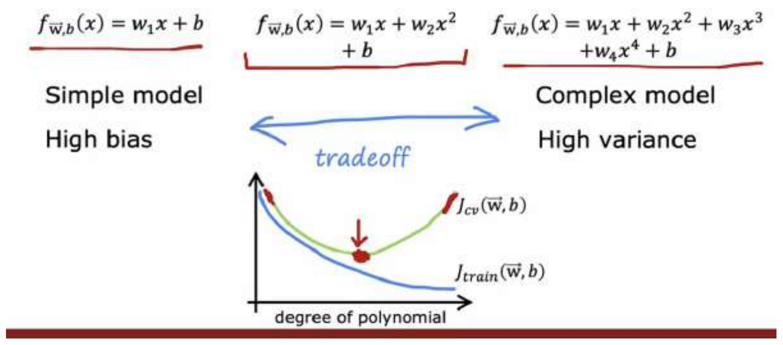
But it makes unacceptably large errors in predictions. What do you try next?

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

Bias/ variance and neural networks

The bias variance tradeoff

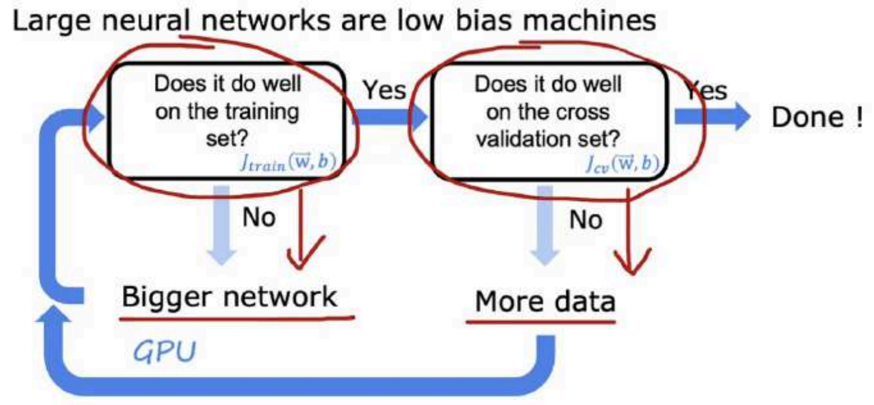
- Go in the middle to pick model with lowest cross-validation error



$f_{\vec{w},b}(x) = w_1x + b$
$f_{\vec{w},b}(x) = w_1x + w_2x^2 + b$
$f_{\vec{w},b}(x) = w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$
Simple model
High bias
Complex model
High variance
tradeoff
$J_{cv}(\vec{w},b)$
$J_{train}(\vec{w},b)$
degree of polynomial

Neural networks offer us a way out of this:

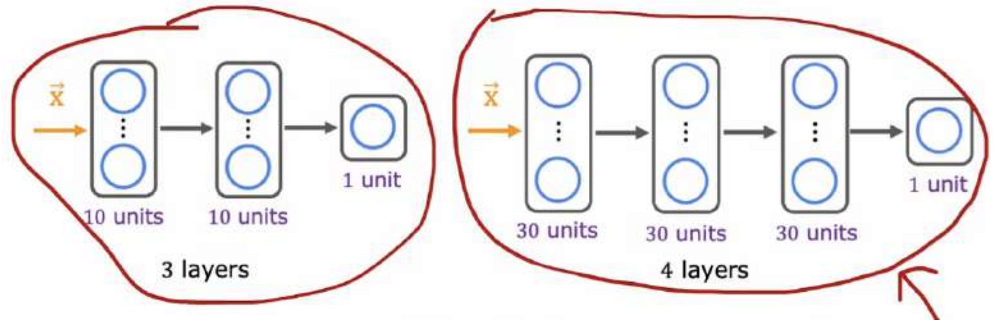
- **Large neural networks are low bias machines (don't need to trade off)**


Large neural networks are low bias machines

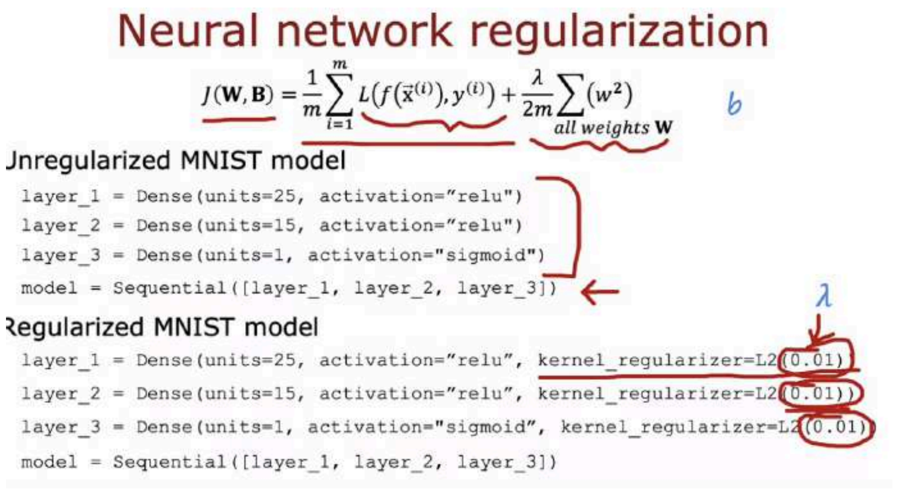
Does it do well on the training set? $J_{train}(\vec{w},b)$
Yes
Does it do well on the cross validation set? $J_{cv}(\vec{w},b)$
Yes
Done !
No
No
Bigger network
More data
GPU

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 (does not overfit)**



To regularise it practically in TensorFlow:



Neural network regularization

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

Jnregularized 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])
```

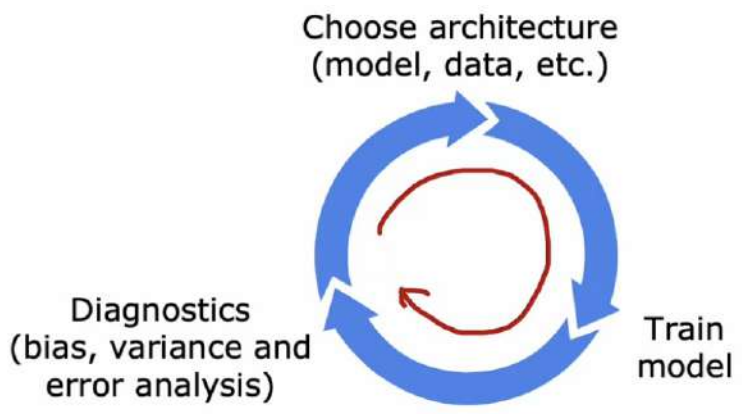
Regularized MNIST model

λ

```
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])
```

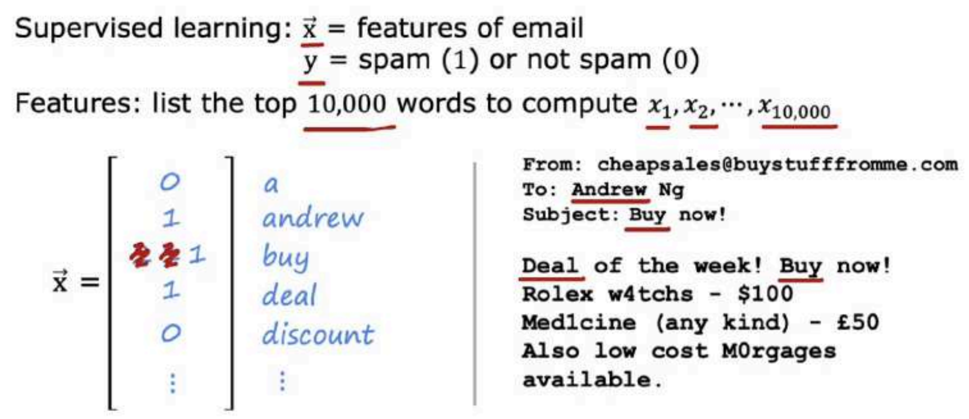
- Often you fight variance problems rather than bias problems with a large neural network

<u>Iterative loop of ML development</u>



Building a spam classifier

- Text classification
- Set words that appear to 1, ones that do not to 0
- Can then train model to predict y from x

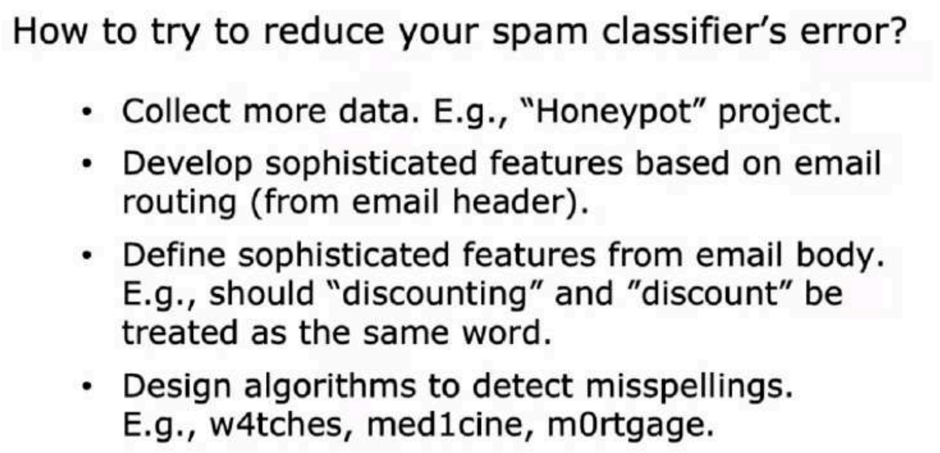


Supervised learning: $\vec{x}$ = features of email
y = spam (1) or not spam (0)

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

$$\vec{x} = \begin{array}{cc} \begin{bmatrix} 0 \\ 1 \\ \cancel{2}\cancel{2}1 \\ 1 \\ 0 \\ \vdots \end{bmatrix} & \begin{array}{l} a \\ andrew \\ buy \\ deal \\ discount \\ \vdots \end{array} \end{array}$$

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

Deal of the week! Buy now!
Rolex w4tchs - $100
Med1cine (any kind) - £50
Also low cost M0rgages
available.
```

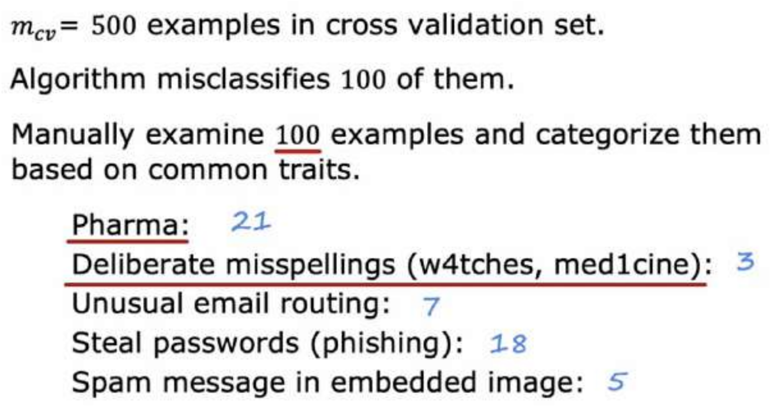


How to try to reduce your spam classifier's error?

- Collect more data. E.g., "Honeypot" project.
- Develop sophisticated features based on email routing (from email header).
- Define sophisticated features from email body. E.g., should "discounting" and "discount" be treated as the same word.
- Design algorithms to detect misspellings. E.g., w4tches, med1cine, m0rtgage.

<u>*Error Analysis*</u>

- Manually examine errors: (if very large misclassified set, randomly sample them)



m_{cv} = 500 examples in cross validation set.

Algorithm misclassifies 100 of them.

Manually examine 100 examples and categorize them based on common traits.

Pharma:	21
Deliberate misspellings (w4tches, med1cine):	3
Unusual email routing:	7
Steal passwords (phishing):	18
Spam message in embedded image:	5

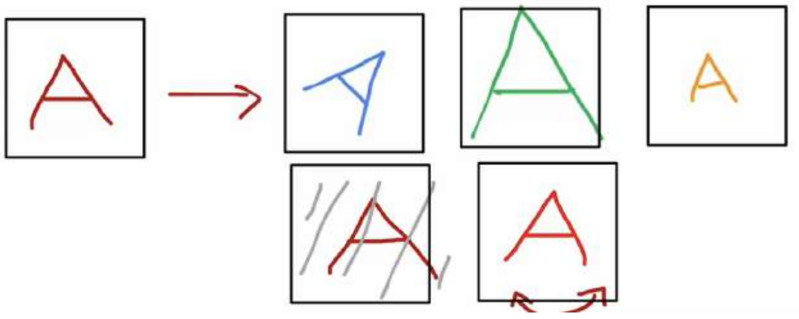
- Identify where the major errors are occurring - here, pharmaceutical spam emails. Specifically target these e.g. look at more data features here of these specific email types

Adding Data

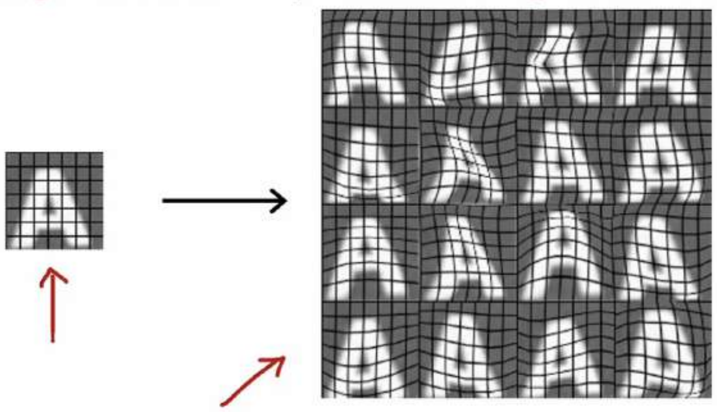
- **Add more data of the types where error analysis has indicated it might help.**

E.g. go to unlabeled data and find more examples of pharma related spam.

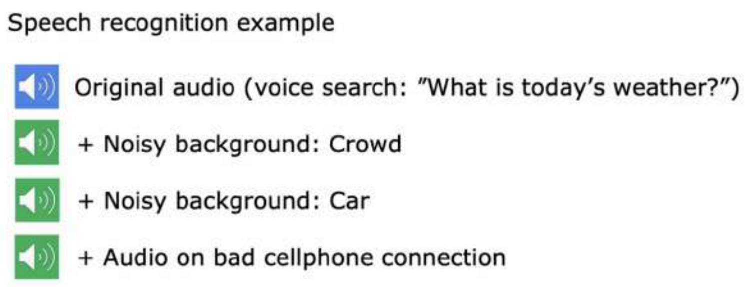
1) **Data augmentation:** modifying an existing training example to create a new training example.



- Can also do data augmentation by introducing distortions



- Can also apply data augmentation to speech data
    - Put noisy backgrounds or crackles etc. (to take one audio clip to create three training examples)



Speech recognition example

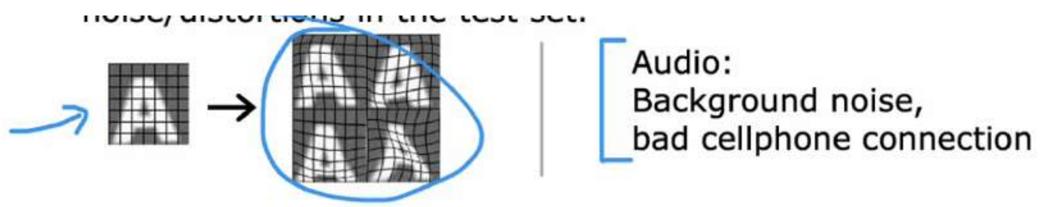
🔊 Original audio (voice search: "What is today's weather?")

🔊 + Noisy background: Crowd

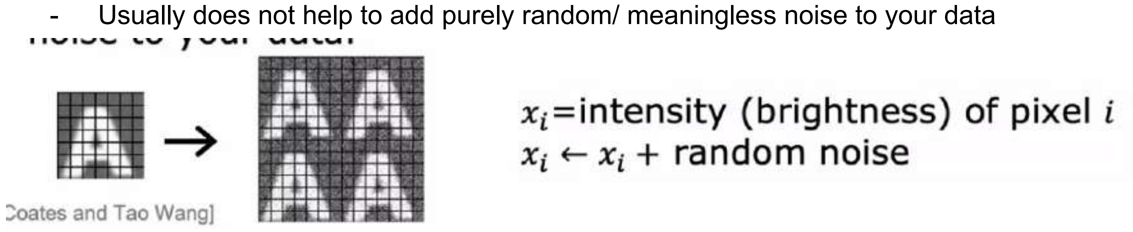
🔊 + Noisy background: Car

🔊 + Audio on bad cellphone connection

Note that distortions introduced should be representative of the type of noise/ distortions in the test set.

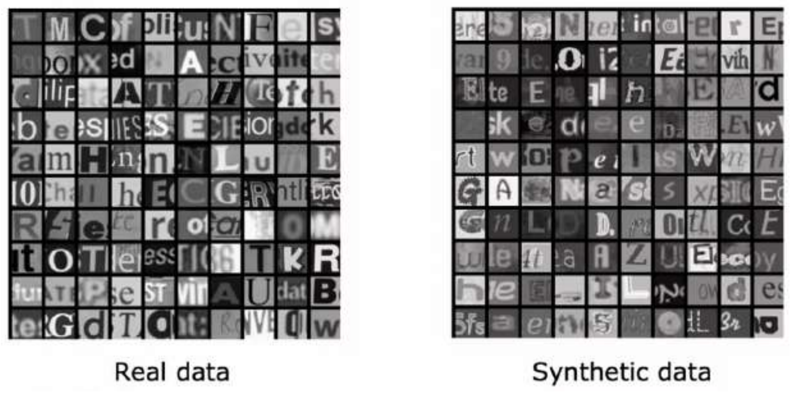


- Usually does not help to add purely random/ meaningless noise to your data

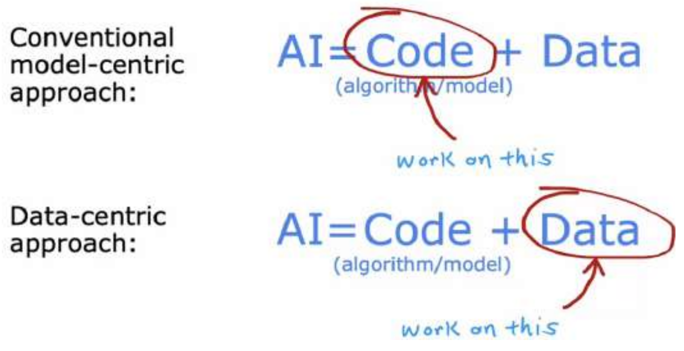

x_i=intensity (brightness) of pixel i
$x_i \leftarrow x_i$ + random noise
Coates and Tao Wang]

2) **Data synthesis**: using artificial data inputs to create a new training example. Largely used for computer vision.

E.g. photo OCR. Can create artificial data by using different fonts and screenshot it with different backgrounds.



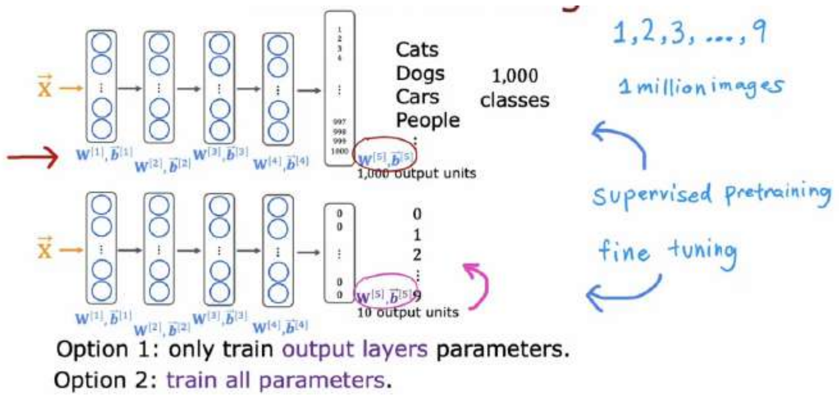
These are all types of data engineering:



When there is not much data, can use:

Transfer learning: Using data from a different task

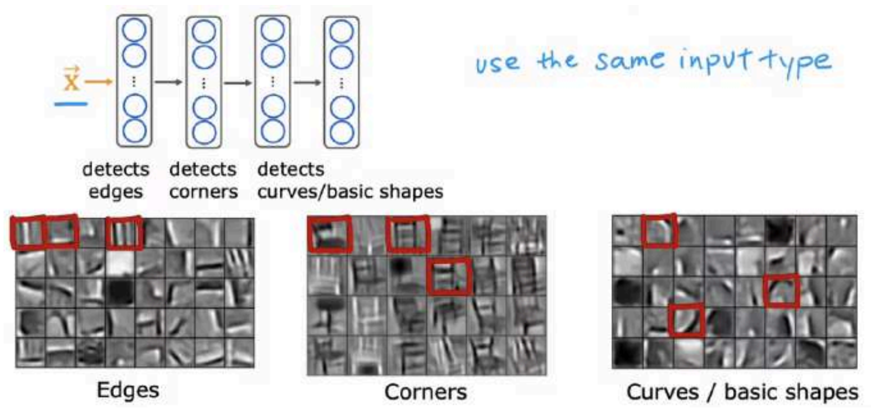
- Want to recognise handwritten digits of 1 to 9; find 1 million images, start to train learning algorithm on 1000 classes
- For transfer learning, make a copy of this neural network with the same w1, b1, w2, b2 etc. but replace the output layer with only 10 output units and a new w5,b5
- So two options for training this neural network's parameters
    - Option 1: only train output layers parameters, and only update w5,b5 to lower cost function (best for a small training set)
    - Option 2: train all parameters in the network; first four layers would be initialised using previous values (best for a larger training set)



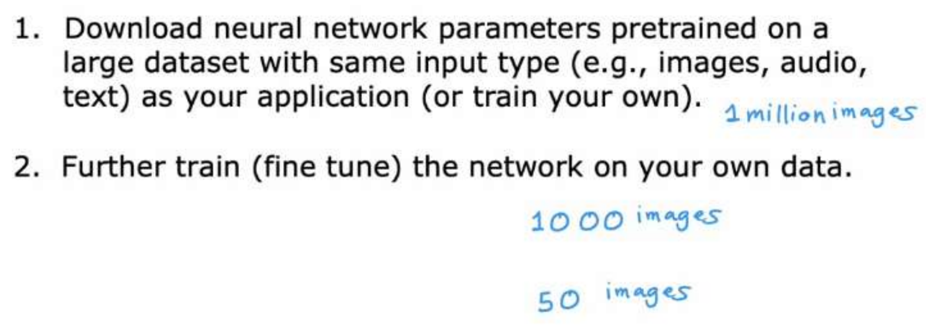
- Can use this to make the most of pre-trained neural networks online

Why does transfer learning work?

- In image recognition, first level might learn to detect edges, the second to detect corders, the third to detect curves/basic shapes etc.
- Restriction: need to use same input type as the input the network was pretrained on



Summary of transfer learning:



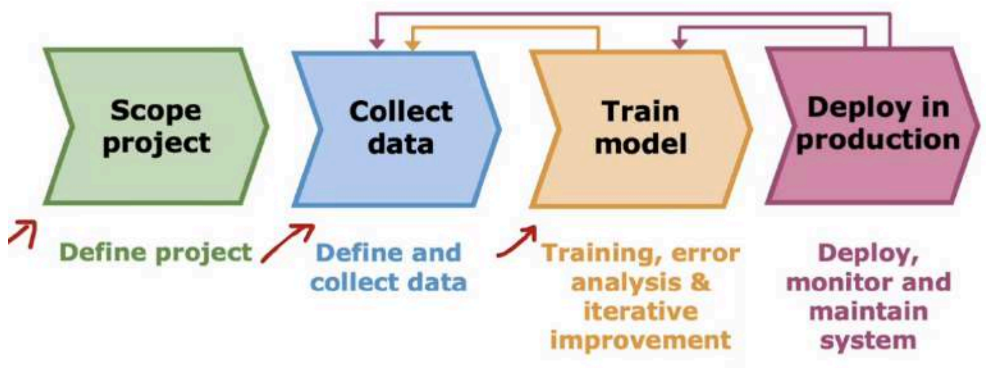
1. Download neural network parameters pretrained on a large dataset with same input type (e.g., images, audio, text) as your application (or train your own). 1 million images
2. Further train (fine tune) the network on your own data.

1000 images

50 images

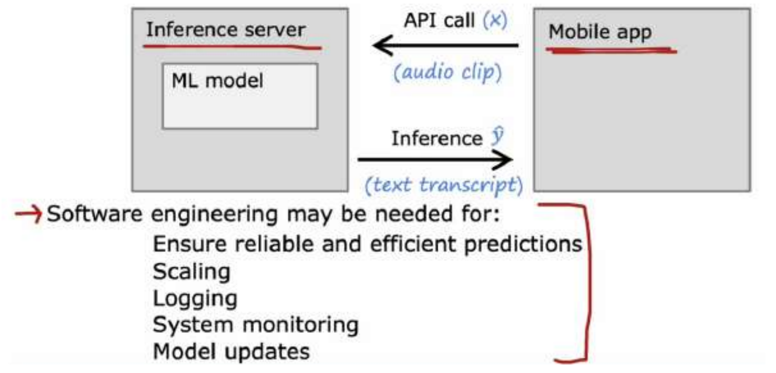
Full Cycle of a Machine Learning Project

Using the example of speech recognition…



What is deployment?

- Implement ML model in an inference server. If team has implemented a mobile ask, when a user talks to app, mobile app can make an API call to parse to inference server, which can then apply the ML model and return the inference prediction



Software engineering may be needed for:
- Ensure reliable and efficient predictions
- Scaling
- Logging
- System monitoring
- Model updates

MLOps - how to systematically build, deploy, and maintain ML systems.

<u>Fairness, bias, and ethics</u>

Bias:

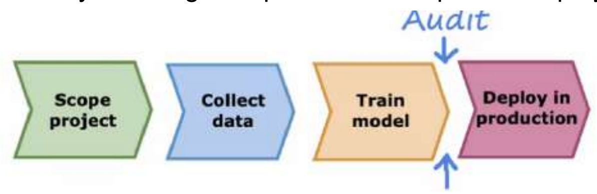
- Hiring tool that discriminates against women
- Facial recognition system matching dark skinned individuals to criminal mugshots
- Biased bank loan approvals
- Toxic effect of reinforcing negative stereotypes

Adverse use cases of ML algorithms:

- Buzzfeed released deepfake of Obama
- Social media spreading toxic/incendiary speech through optimizing for engagement
- Generating fake content for commercial and political purposes
- Using ML to build harmful products, commit fraud etc.
    - Spam vs anti-spam: fraud vs anti-fraud

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



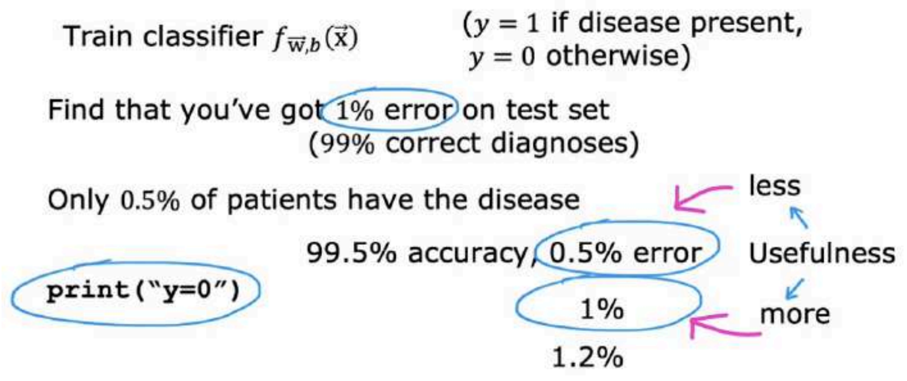
- Develop mitigation plan (if applicable) and, after deployment, monitor for possible harm.

<u>Error metrics for skewed datasets</u>

For situations where ratio of positive to negative examples is v. skewed, usual error metrics like accuracy do not work that well.

E.g. Rate disease classification

If you just printed y=0 all the time, would have 99.5% accuracy in this case. Accuracy is relevant:



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

Find that you've got 1% error on test set (99% correct diagnoses)

Only 0.5% of patients have the disease

99.5% accuracy, 0.5% error

`print("y=0")`

1%

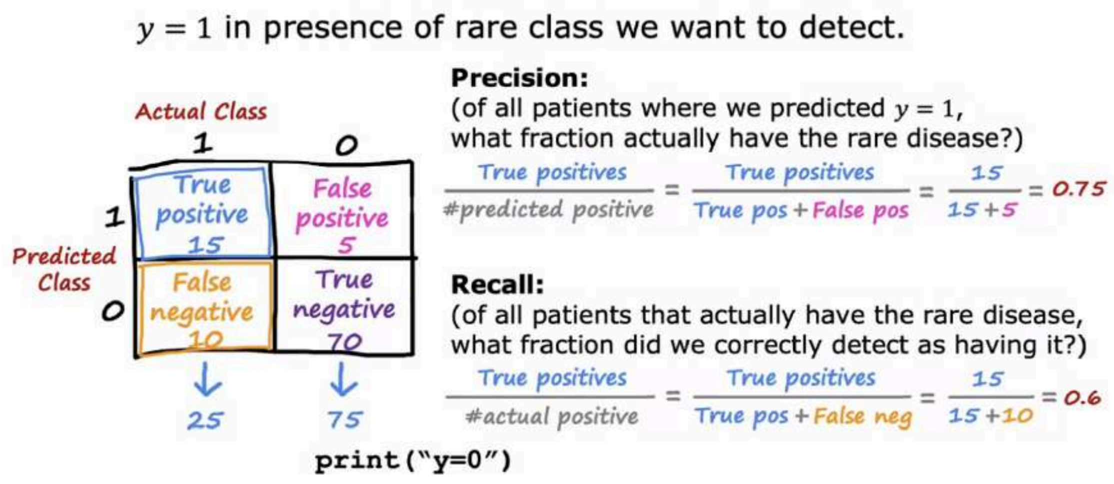
1.2%

less

Usefulness

more

Precision/ recall



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

		Actual Class 1	Actual Class 0
Predicted Class	1	True positive 15	False positive 5
	0	False negative 10	True negative 70
		25	75

print("y=0")

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

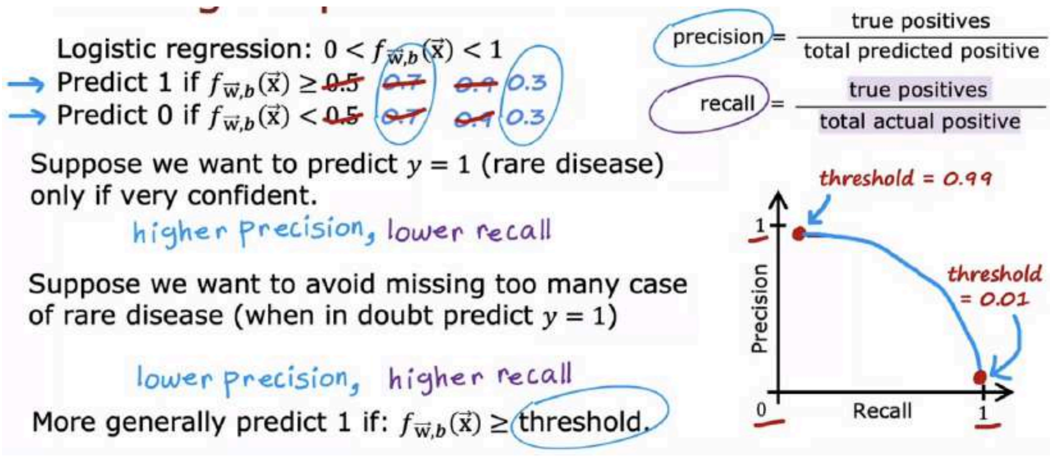
$$\frac{\text{True positives}}{\#\text{predicted positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False pos}} = \frac{15}{15+5} = 0.75$$

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

$$\frac{\text{True positives}}{\#\text{actual positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False neg}} = \frac{15}{15+10} = 0.6$$

Trading off precision and recall

- Ideally have high precision and high recall, but often there is a tradeoff



Logistic regression: $0 < f_{\vec{w},b}(\vec{x}) < 1$

→ Predict 1 if $f_{\vec{w},b}(\vec{x}) \geq$ ~~0.5~~ ~~0.7~~ ~~0.9~~ 0.3

→ Predict 0 if $f_{\vec{w},b}(\vec{x}) <$ ~~0.5~~ ~~0.7~~ ~~0.9~~ 0.3

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

higher precision, lower recall

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

lower precision, higher recall

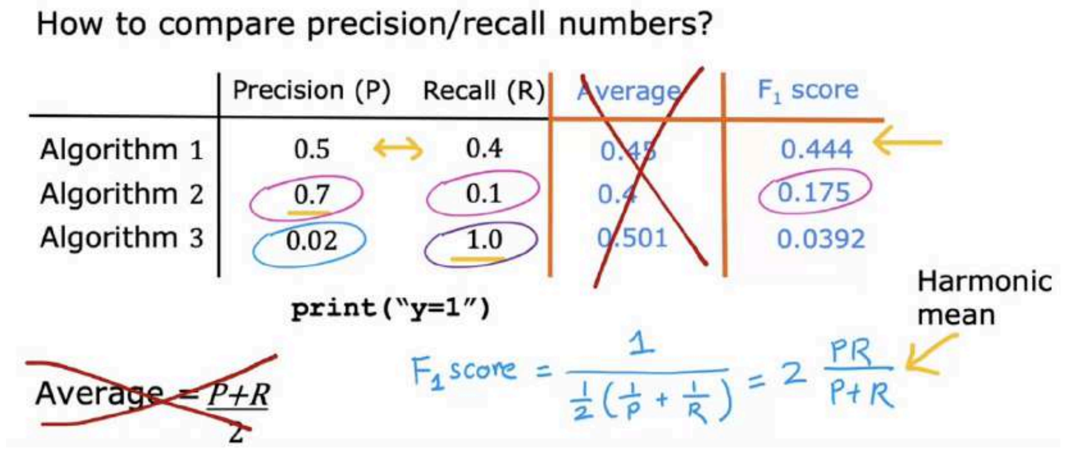
More generally predict 1 if: $f_{\vec{w},b}(\vec{x}) \geq$ threshold.

$$\text{precision} = \frac{\text{true positives}}{\text{total predicted positive}}$$

$$\text{recall} = \frac{\text{true positives}}{\text{total actual positive}}$$

Often need to manually pick the threshold. If you want to automatically trade off precision and recall, can use an “F1 score”:

- Taking average of P and R is not a good method. F1 emphasises very low scores:



How to compare precision/recall numbers?

	Precision (P)	Recall (R)	~~Average~~	F_1 score
Algorithm 1	0.5	0.4	~~0.45~~	0.444
Algorithm 2	0.7	0.1	~~0.4~~	0.175
Algorithm 3	0.02	1.0	~~0.501~~	0.0392

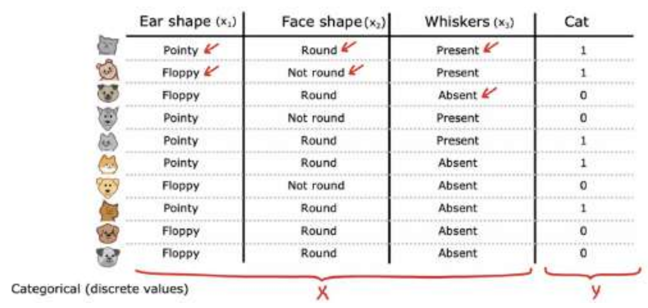
```
print("y=1")
```

~~$\text{Average} = \frac{P+R}{2}$~~

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

Harmonic mean

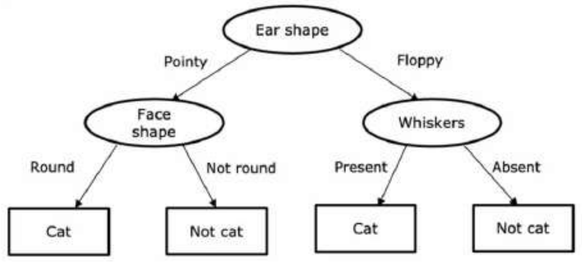
Week 4: Decision Trees



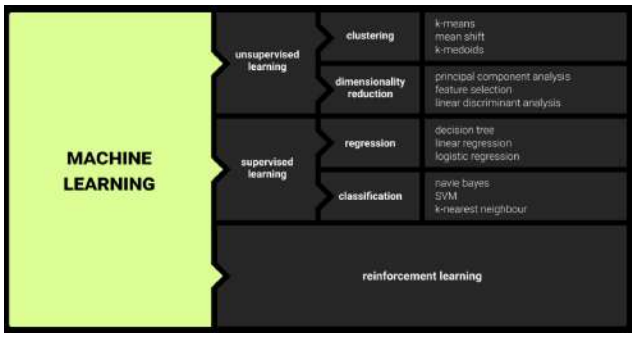
	Ear shape (x_1)	Face shape (x_2)	Whiskers (x_3)	Cat
	Pointy	Round	Present	1
	Floppy	Not round	Present	1
	Floppy	Round	Absent	0
	Pointy	Not round	Present	0
	Pointy	Round	Present	1
	Pointy	Round	Absent	1
	Floppy	Not round	Absent	0
	Pointy	Round	Absent	1
	Floppy	Round	Absent	0
	Floppy	Round	Absent	0

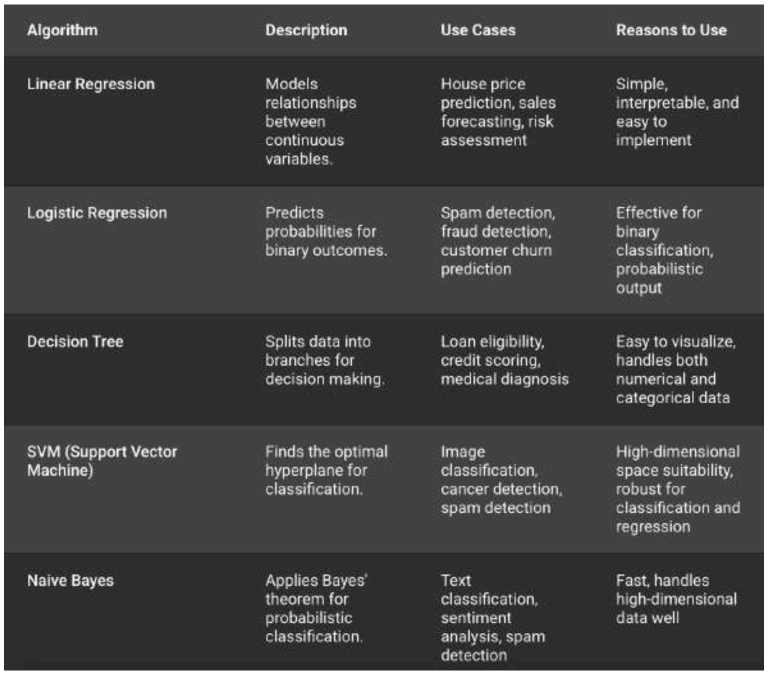
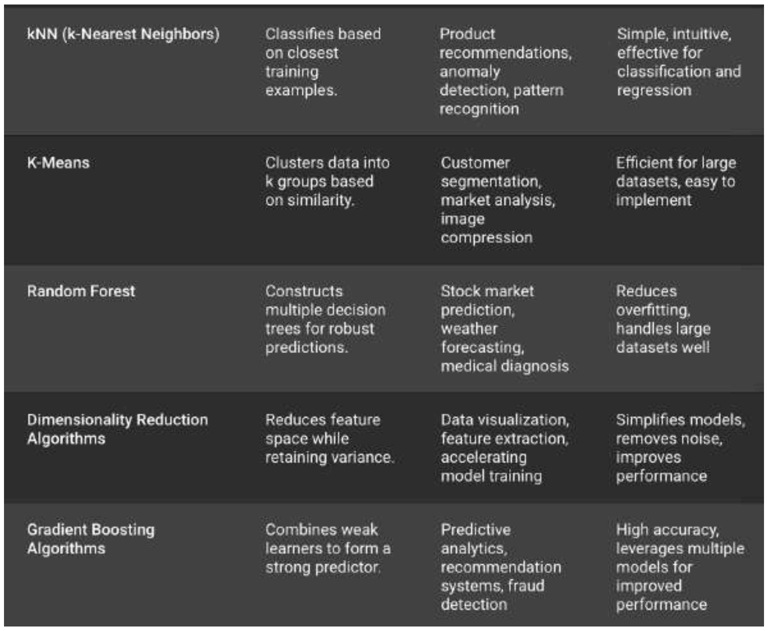
Categorical (discrete values)

X y



Root node at top; middle nodes are decision nodes; notes at bottom are called leaf nodes.



Algorithm	Description	Use Cases	Reasons to Use
Linear Regression	Models relationships between continuous variables.	House price prediction, sales forecasting, risk assessment	Simple, interpretable, and easy to implement
Logistic Regression	Predicts probabilities for binary outcomes.	Spam detection, fraud detection, customer churn prediction	Effective for binary classification, probabilistic output
Decision Tree	Splits data into branches for decision making.	Loan eligibility, credit scoring, medical diagnosis	Easy to visualize, handles both numerical and categorical data
SVM (Support Vector Machine)	Finds the optimal hyperplane for classification.	Image classification, cancer detection, spam detection	High-dimensional space suitability, robust for classification and regression
Naive Bayes	Applies Bayes' theorem for probabilistic classification.	Text classification, sentiment analysis, spam detection	Fast, handles high-dimensional data well
kNN (k-Nearest Neighbors)	Classifies based on closest training examples.	Product recommendations, anomaly detection, pattern recognition	Simple, intuitive, effective for classification and regression
K-Means	Clusters data into k groups based on similarity.	Customer segmentation, market analysis, image compression	Efficient for large datasets, easy to implement
Random Forest	Constructs multiple decision trees for robust predictions.	Stock market prediction, weather forecasting, medical diagnosis	Reduces overfitting, handles large datasets well
Dimensionality Reduction Algorithms	Reduces feature space while retaining variance.	Data visualization, feature extraction, accelerating model training	Simplifies models, removes noise, improves performance
Gradient Boosting Algorithms	Combines weak learners to form a strong predictor.	Predictive analytics, recommendation systems, fraud detection	High accuracy, leverages multiple models for improved performance

__Decision Tree: Learning Process__

- Use an algorithm to decide what root nodes etc. to use
- Looking to maximise purity (percentage correctness)

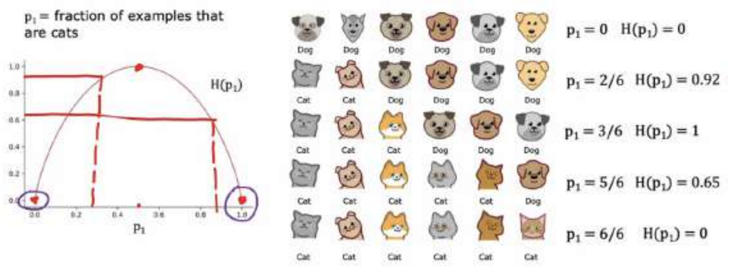
Decision 1: How to choose what feature to split on at each node?

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 (root node is at Depth 0)
- When improvements in purity score are below a threshold
- When number of examples in a node is below a certain threshold

Decision Trees: Measuring purity

E.g. p1 = fraction of examples that are cats; y-axis is entropy



Entropy function (log is to base 2):

$$p_0 = 1 - p_1$$

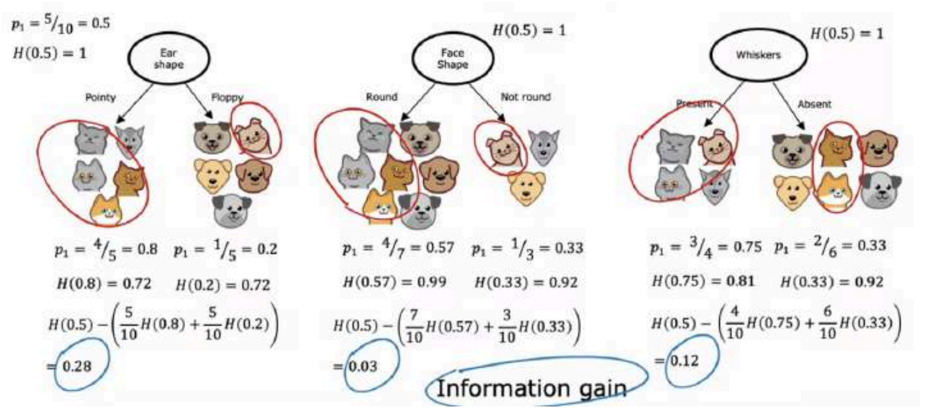
$$H(p_1) = -p_1 log_2(p_1) - p_0 log_2(p_0)$$
$$= -p_1\, log_2(p_1) - (1 - p_1) log_2(1 - p_1)$$

Note: "$0 \log(0)$" $= 0$

- Even if 0log(0) is technically negative infinity

Choosing a split: Information gain

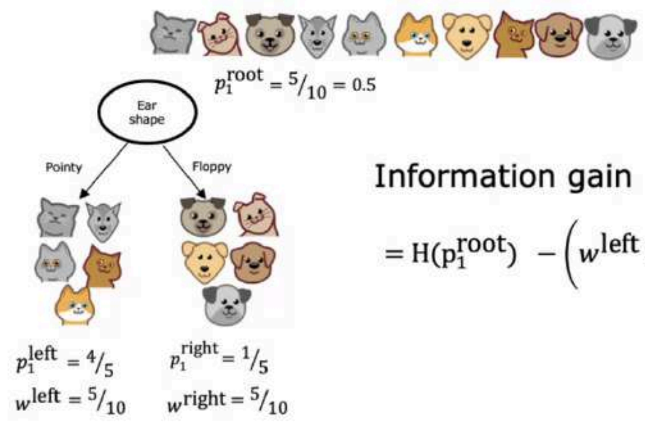
- What choice of feature reduces entropy, *H*, the most (reduction of entropy = "information gain")
- Take a weighted average, based on how many examples + entropy (a large dataset with high entropy is worse than a small one with high entropy)
- Overall, compute weighted reduction in entropy compared to not splitting at all (p1 = 5/10 = 0.5, so H at root node is 1) [as one of the ways to decide whether to split is size of reduction in entropy - not much point in splitting if decrease is v small]



$p_1 = 5/10 = 0.5$
$H(0.5) = 1$
Ear shape
Pointy
Floppy
$p_1 = 4/5 = 0.8$ $p_1 = 1/5 = 0.2$
$H(0.8) = 0.72$ $H(0.2) = 0.72$
$H(0.5) - \left(\frac{5}{10}H(0.8) + \frac{5}{10}H(0.2)\right)$
$= 0.28$
$H(0.5) = 1$
Face Shape
Round
Not round
$p_1 = 4/7 = 0.57$ $p_1 = 1/3 = 0.33$
$H(0.57) = 0.99$ $H(0.33) = 0.92$
$H(0.5) - \left(\frac{7}{10}H(0.57) + \frac{3}{10}H(0.33)\right)$
$= 0.03$
Information gain
$H(0.5) = 1$
Whiskers
Present
Absent
$p_1 = 3/4 = 0.75$ $p_1 = 2/6 = 0.33$
$H(0.75) = 0.81$ $H(0.33) = 0.92$
$H(0.5) - \left(\frac{4}{10}H(0.75) + \frac{6}{10}H(0.33)\right)$
$= 0.12$

__Information Gain__

P1 - number of positive examples on left or right, w = fraction of examples that went to right or left branch



$p_1^{\text{root}} = 5/10 = 0.5$
Ear shape
Pointy
Floppy
$p_1^{\text{left}} = 4/5$
$w^{\text{left}} = 5/10$
$p_1^{\text{right}} = 1/5$
$w^{\text{right}} = 5/10$

Information gain

$$= H(p_1^{\text{root}}) - \left(w^{\text{left}} H\left(p_1^{\text{left}}\right) + w^{\text{right}} H\left(p_1^{\text{right}}\right)\right)$$

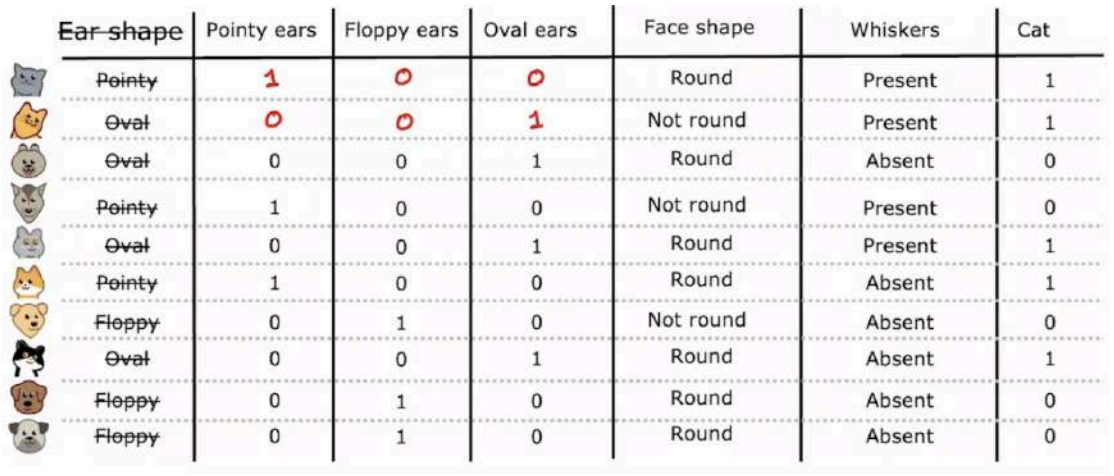
__Putting it together__

Overall process:
- Start with all examples at root node
- Calculate info gain for all poss features, and pick one with highest info gain
- Split dataset according to selected feature, and create left and right branches of the treee
- 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
    - Info gain from additional splits is less than threshold
    - When number of examples in a node is below a threshold

Recursive splitting: to build a decision tree at the root, you build smaller decision trees at the left and right branches.

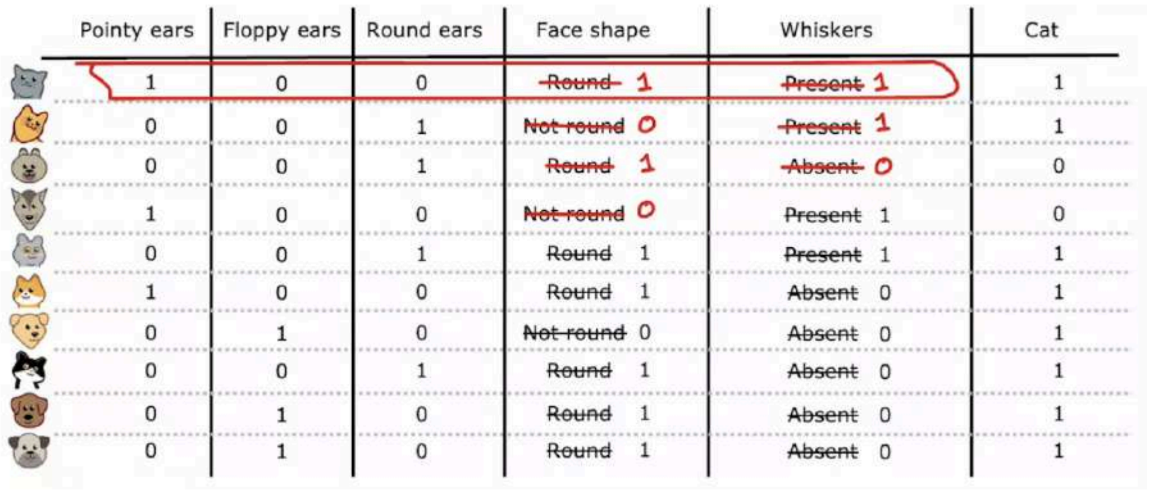
<u>Using one-hot encoding of categorical features:</u> ***<u>For features that can take on more values than yes or no</u>***

- Split into different features, keeping each to taking on two values. ***If a categorical feature can take on k values, create k binary features (0 or 1 valued).***



~~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

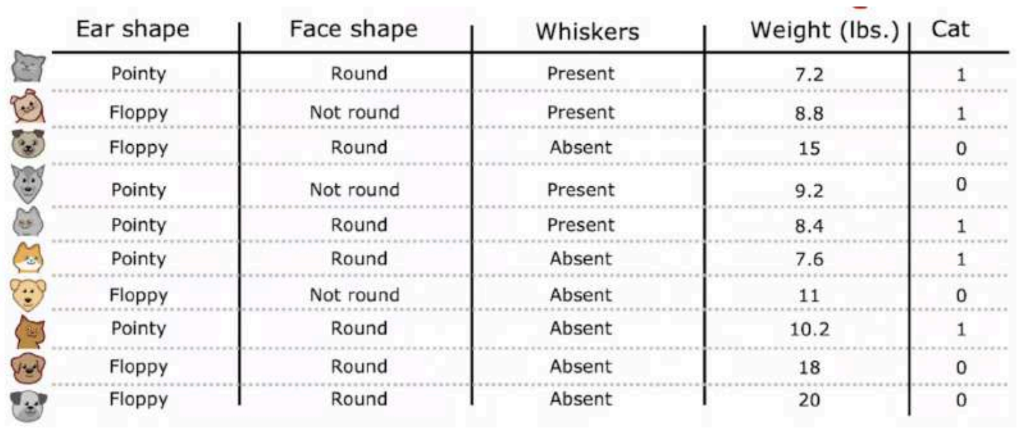
This also works for training neural networks (not just for decision trees - neural networks expect numbers as inputs):



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

<u>Continuous Valued Features</u>

- Using decision trees to work with not only discrete values, but continuous too (e.g. weight)

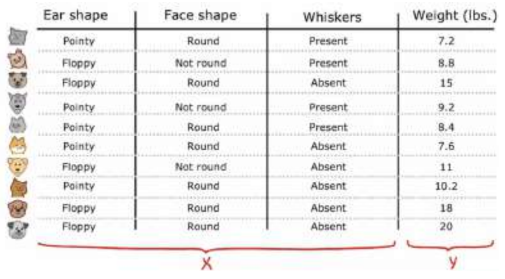


Ear shape	Face shape	Whiskers	Weight (lbs.)	Cat
Pointy	Round	Present	7.2	1
Floppy	Not round	Present	8.8	1
Floppy	Round	Absent	15	0
Pointy	Not round	Present	9.2	0
Pointy	Round	Present	8.4	1
Pointy	Round	Absent	7.6	1
Floppy	Not round	Absent	11	0
Pointy	Round	Absent	10.2	1
Floppy	Round	Absent	18	0
Floppy	Round	Absent	20	0

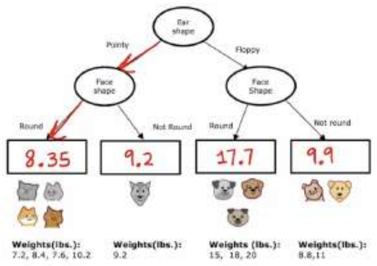
- For a continuous value, pick a threshold which gives the best information gain (calculate information gain for lots of possible thresholds)
- Convention: sort all of examples according to weight/ value of features, and take all the values that are the midpoints between the sorted list of training examples as the values for consideration as a threshold
- If info gain is better from splitting on a given value of this threshold than any other feature, then you decide to split this node at this feature. I.e. there are two thresholds: algorithm has to choose which feature to use first with info gain (considering the different thresholds for continuous ones)

Decision trees in Regression

- So far, only in classification algorithms
- Can also use in regression: using ear shape etc. to predict the weight

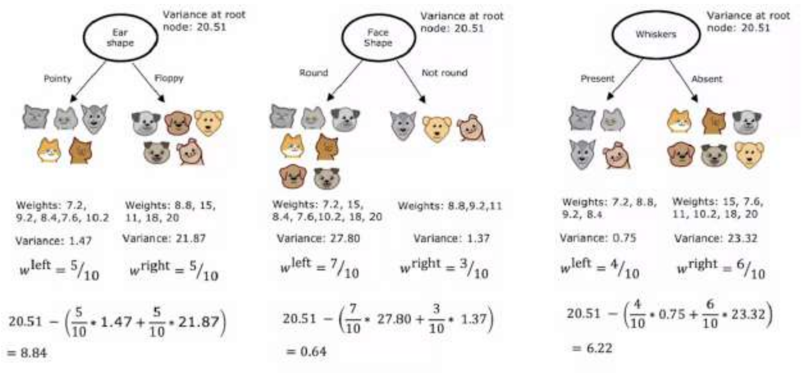


Ear shape	Face shape	Whiskers	Weight (lbs.)
Pointy	Round	Present	7.2
Floppy	Not round	Present	8.8
Floppy	Round	Absent	15
Pointy	Not round	Present	9.2
Pointy	Round	Present	8.4
Pointy	Round	Absent	7.6
Floppy	Not round	Absent	11
Pointy	Round	Absent	10.2
Floppy	Round	Absent	18
Floppy	Round	Absent	20



So, how to choose a split?:

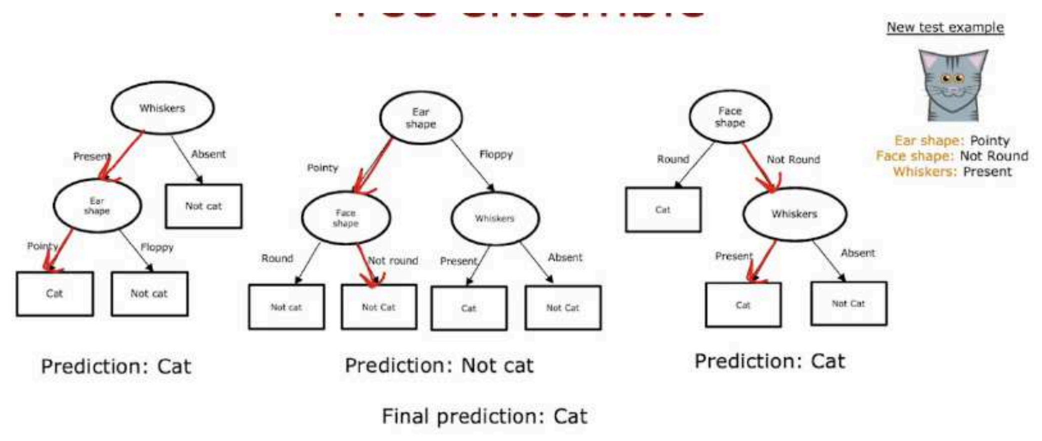
- Instead of reducing entropy, **try to reduce the weighted average variance** (how widely a set of numbers varies)



Then build a new decision tree based on these five examples.

Using Multiple Decision Trees: A tree ensemble

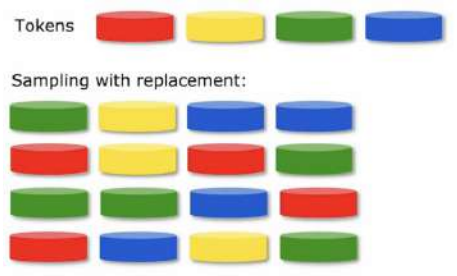
- **Trees are highly sensitive to small changes of the data** (can cause a different split at the root and thus a totally different tree)
- Use many trees, and run all on a new example. Get them to then vote



Having lots of trees makes the algo less sensitive to changes in data. But how to come up with plausible but different decision trees - use **sampling with replacement**.

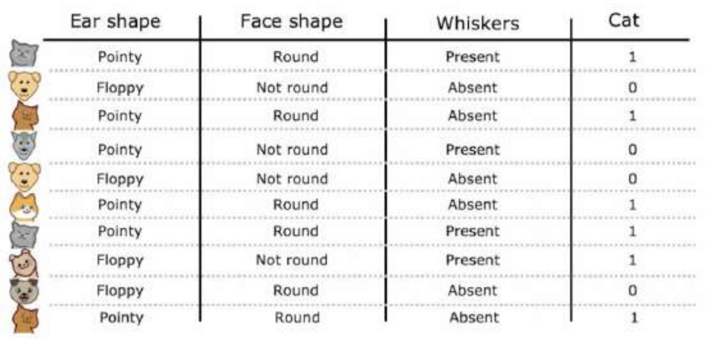
Sampling with replacement

[Putting the token back - if not, would get the same four tokens every time.]



Applying this to decision trees:

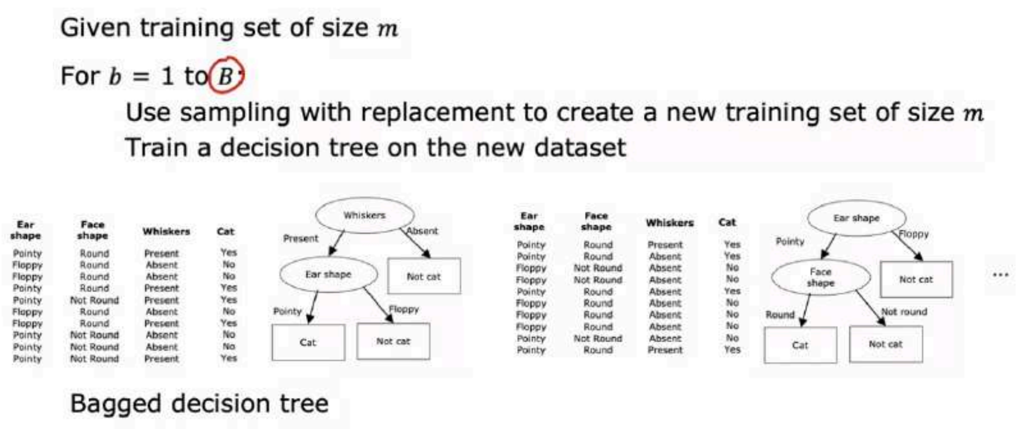
- Construct random training sets that are all slightly different from original training set. Put the training examples in a theoretical bag and pick one out, put it back in, and pick again (will get some repeats).



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

Random Forest Algorithm

1. 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
Floppy	Round	Absent	No
Pointy	Round	Present	Yes
Pointy	Not Round	Present	Yes
Floppy	Round	Absent	No
Floppy	Round	Present	Yes
Pointy	Not Round	Absent	No
Pointy	Not Round	Absent	No
Pointy	Not Round	Present	Yes

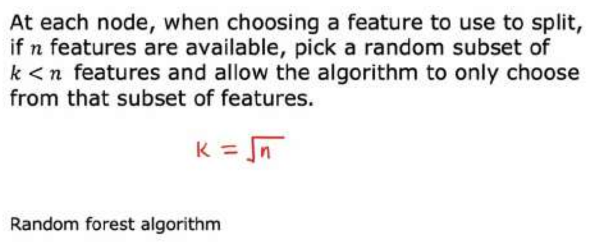
Ear shape	Face shape	Whiskers	Cat
Pointy	Round	Present	Yes
Pointy	Round	Absent	Yes
Floppy	Not Round	Absent	No
Floppy	Not Round	Absent	No
Pointy	Round	Absent	Yes
Floppy	Round	Absent	No
Floppy	Round	Absent	No
Floppy	Round	Absent	No
Pointy	Not Round	Absent	No
Pointy	Round	Present	Yes

Bagged decision tree

- B is normally around 100 max - higher than that slows down computation and also does not improve much
- Each tree then votes

2. Modification: Randomizing the feature choice

- To avoid root note split being often the same



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.

$K = \sqrt{n}$

Random forest algorithm

- This is more robust than a single decision tree because sampling with replacement procedure causes the algorithm to explore a lot of small changes in the data already and is averaging over all these changes, and so small changes to training set are less likely to affect overall output of random first

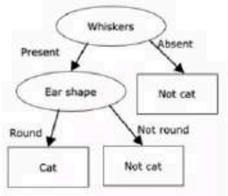
XGBoost

1. Generating a tree sample
    - For every time through this loop, other than first time, when sampling - instead of picking from all m examples from equal probability, make it more likely to pick examples that the previously trained trees do poorly on [FOCUSING ON WHAT WE ARE NOT DOING WELL ON]

```
Given training set of size $m$
For $b$ = 1 to $\underline{B}$:
    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 to pick misclassified examples from previously trained trees
    Train a decision tree on the new dataset
```

$\underbrace{1, 2, \dots, b-1}$ ↖ b

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



Ear shape	Face shape	Whiskers	Prediction
Pointy	Round	Present	Cat ✓
Floppy	Not Round	Present	Not cat ✗ ←
Floppy	Round	Absent	Not cat ✓
Pointy	Not Round	Present	Not cat ✓
Pointy	Round	Present	Cat ✓
Pointy	Round	Absent	Not cat ✗ ←
Floppy	Not Round	Absent	Not cat ✓
Pointy	Round	Absent	Not cat ✗ ←
Floppy	Round	Absent	Not cat ✓
Floppy	Round	Absent	Not cat ✓

XGBoost is:

- 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)
  - Instead of sampling with replacement, assigns different weights to different training examples so does not need to generate a load of randomly chosen training sets, making it a bit more efficient than using a sampling with replacement procedure

Using XGBoost:

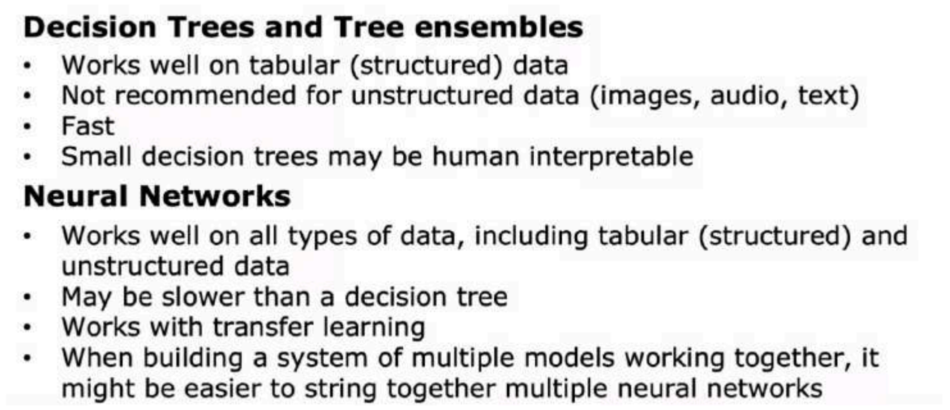
Classification

```
→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 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
    - Can train neural networks all together using gradient descent, but can only train one decision tree at a time