AI CAMP

Introduction to Deep learning

Prepared by : Oussama BOUSSAHLA

school of ai Algiers

AI CAMP

Neural Network Intuition

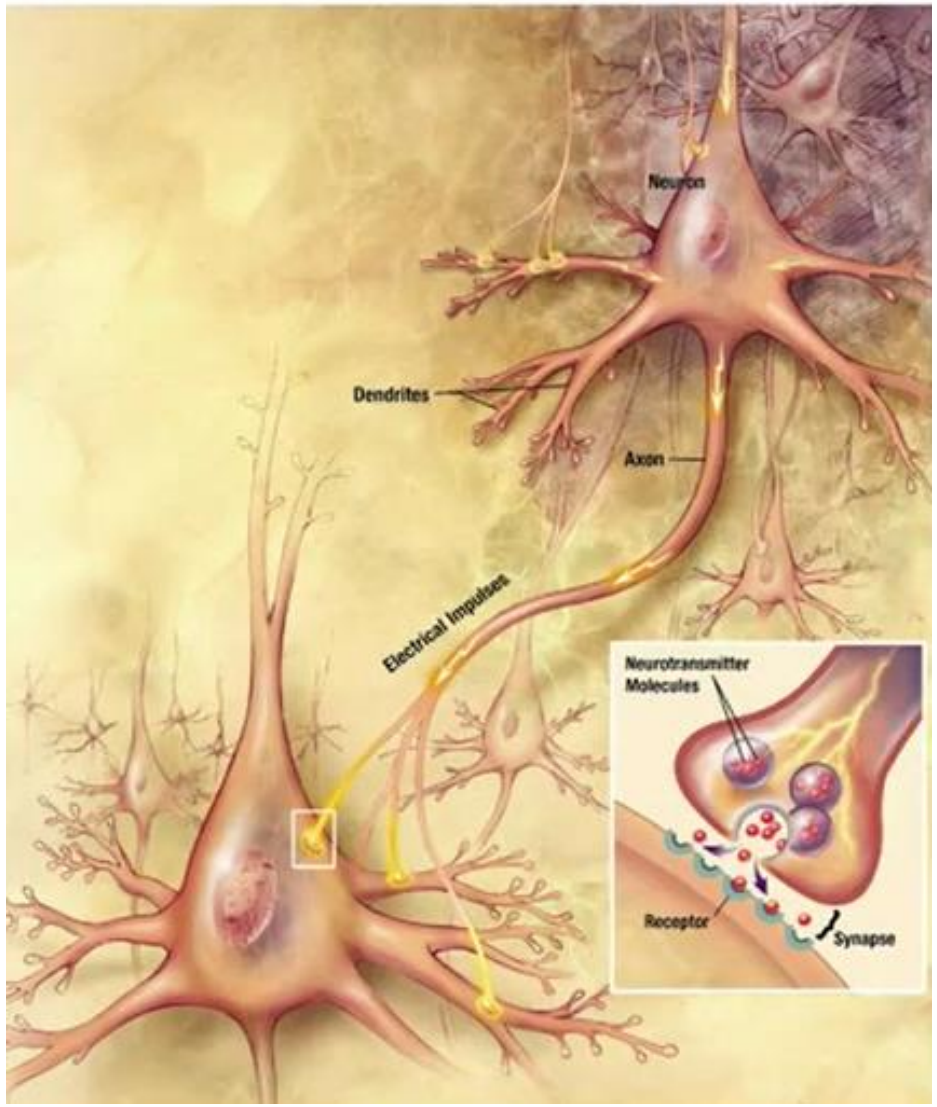
school of ai Algiers

Neural Network

- Origins: Algorithms that try to mimic the brain (1950).
- Used in the 1980’s and early in 1990’s (Hand written digit recognation).
- Fell out of favor in the late 1990’s.
- Resurgence from around 2005 (Deep-learning).

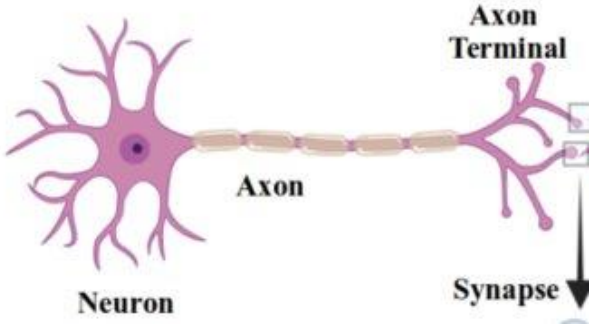
Speech images text

Neurons and the Brain



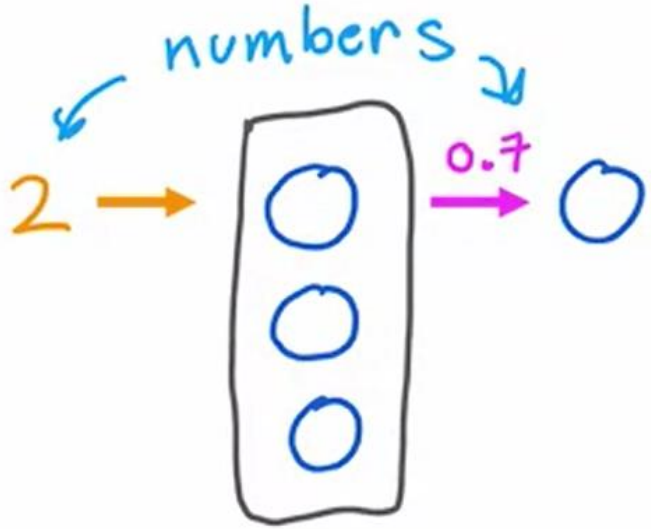
Neurons and the Brain

Biological neuron



Simplified mathematical model of a neuron

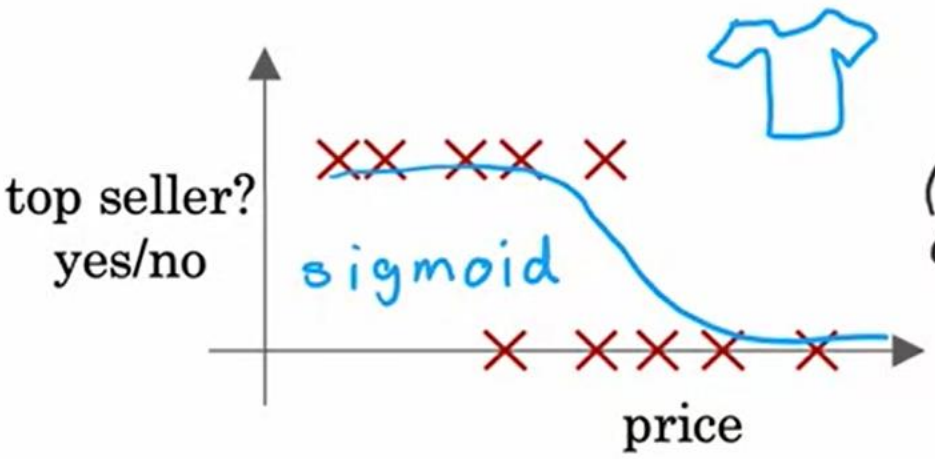
inputs outputs



Why now?

Performance
Large NN
Medium NN
Small NN
Taraditional AI
Amount of data

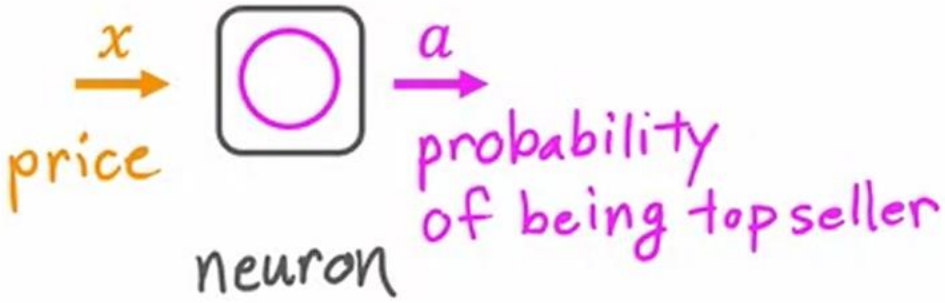
Demand Prediction



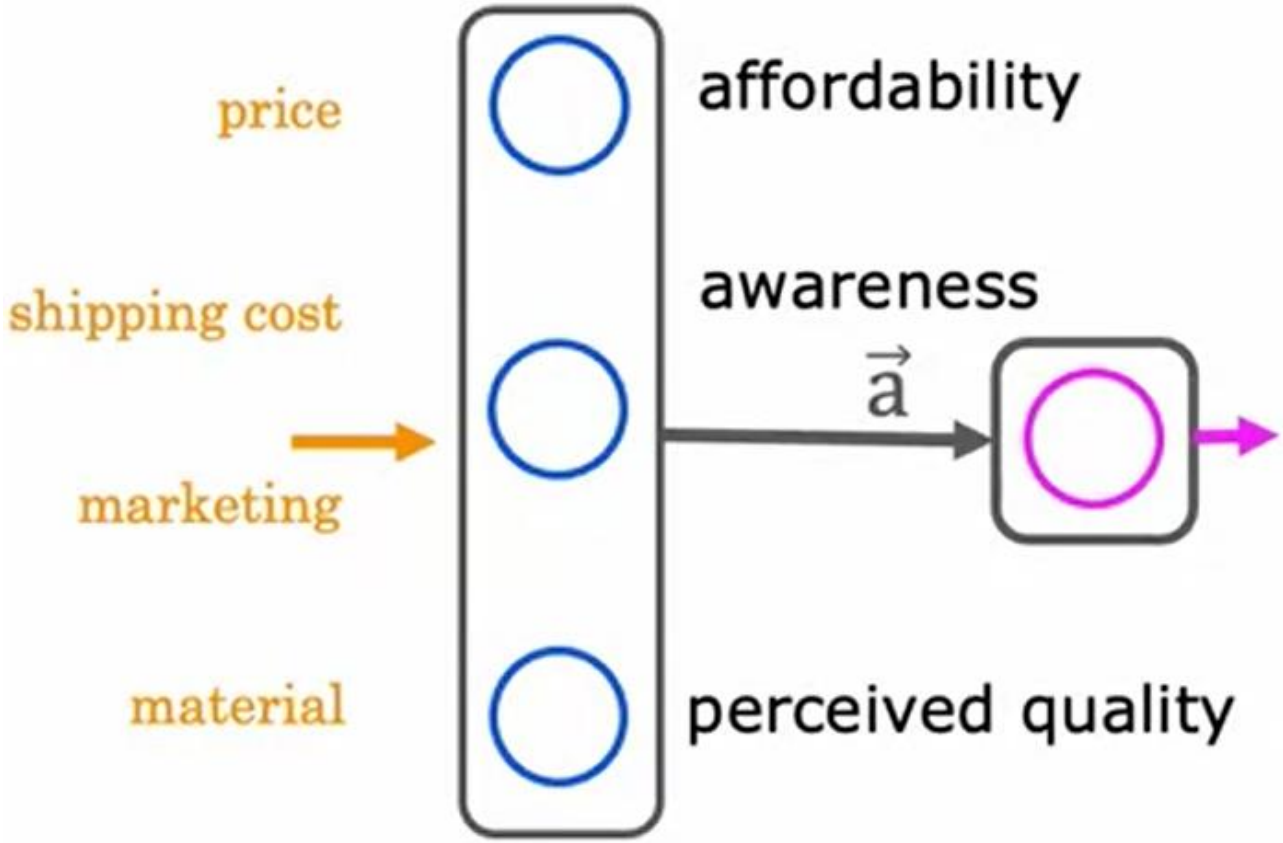
$x = price$ input

$a = f(x) = \dfrac{1}{1 + e^{-(wx+b)}}$ output

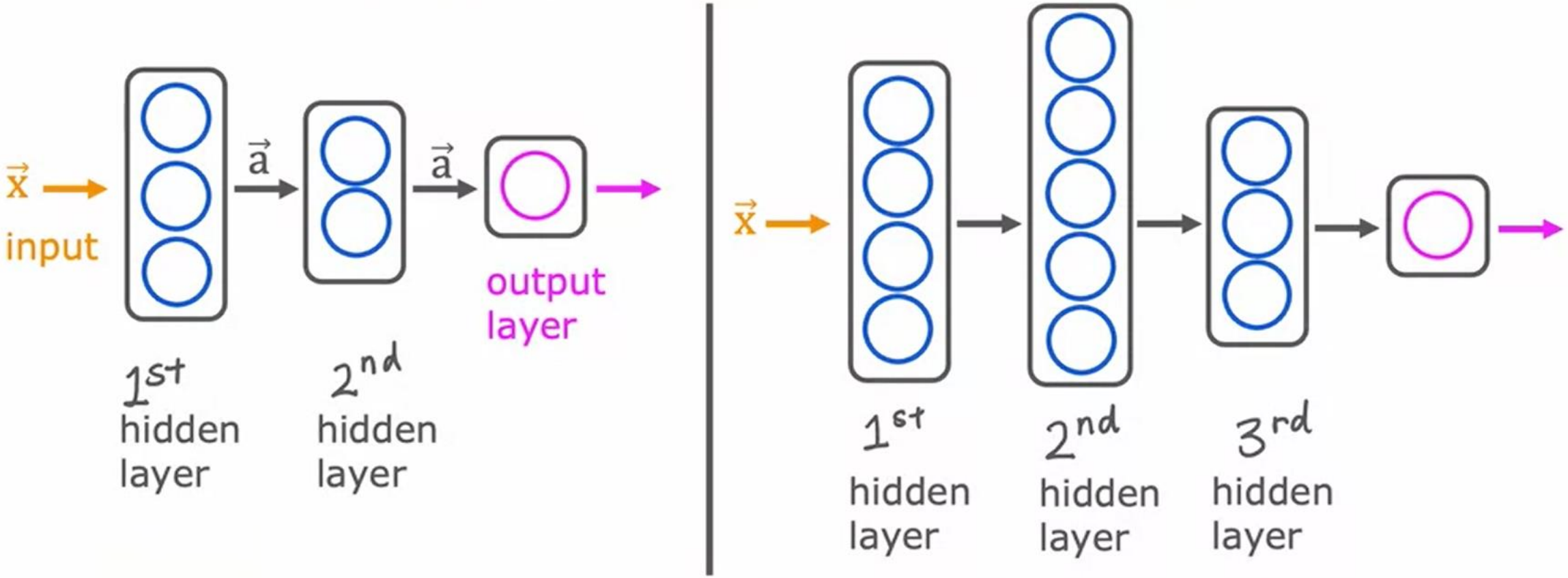
activation



Demand Prediction



Multiple hidden layers



Example: Recognizing Images

1000 pixels

1000 pixels

1000 colums

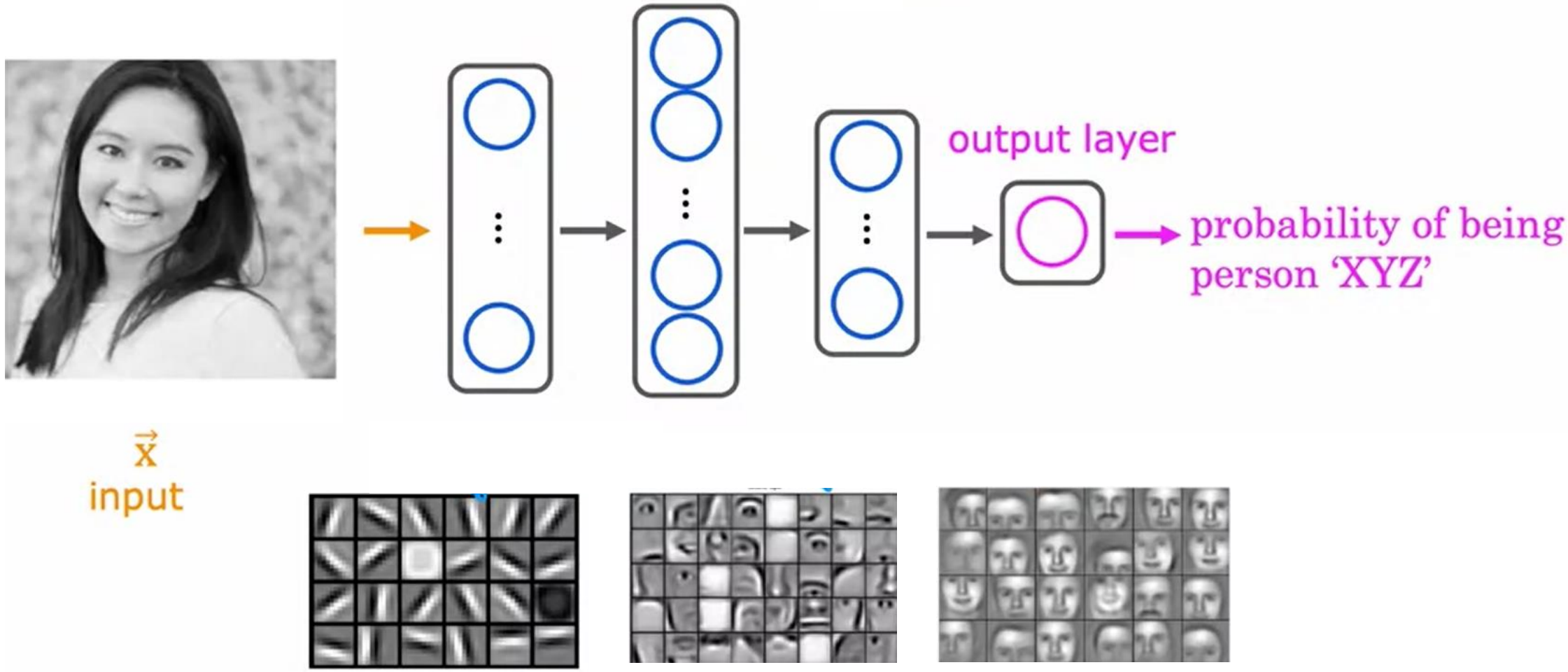
1000 rows

179	158	99	…	….
		57	64	48
	……			
	121	….		
….			45	21

$\vec{x} =$

179
158
…
…
57
….
45
21

1 milllion

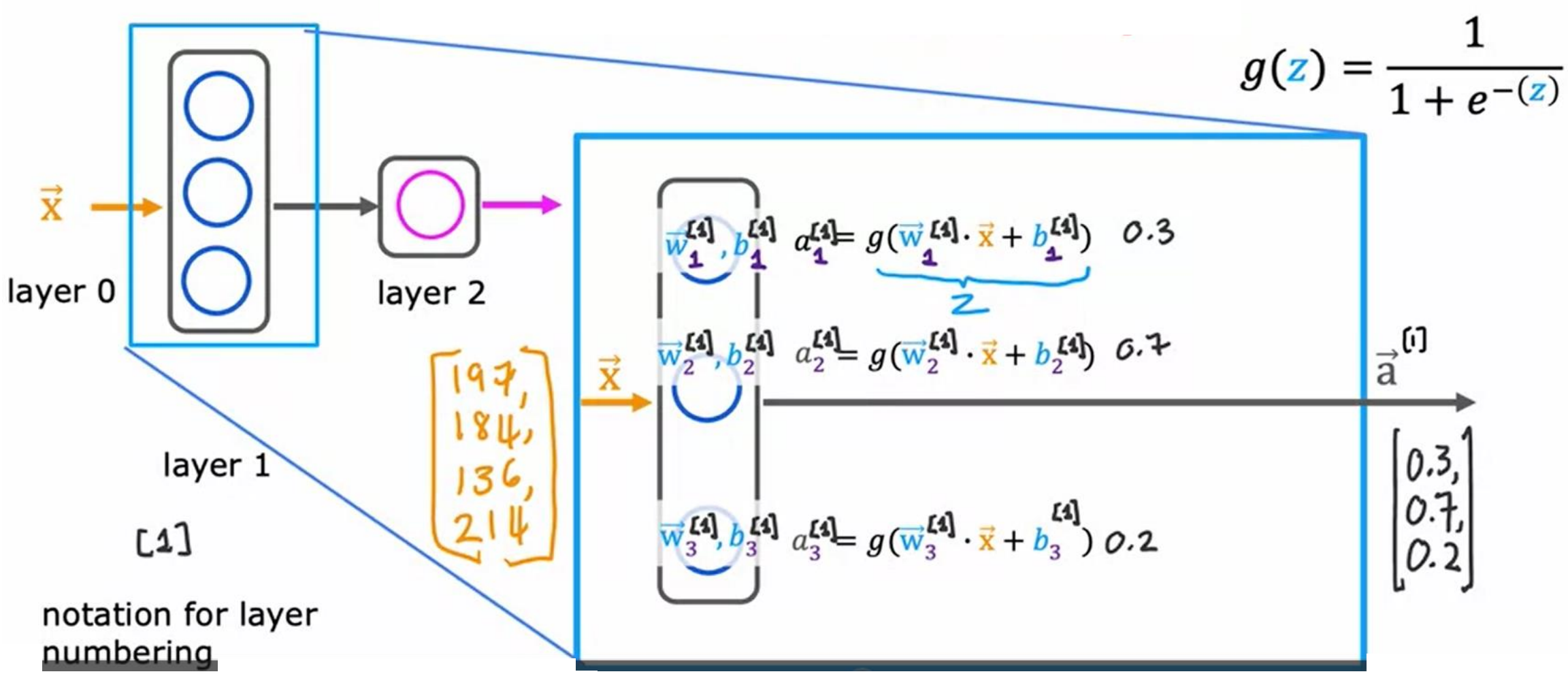

output layer
probability of being person 'XYZ'
$\vec{x}$
input

AI CAMP

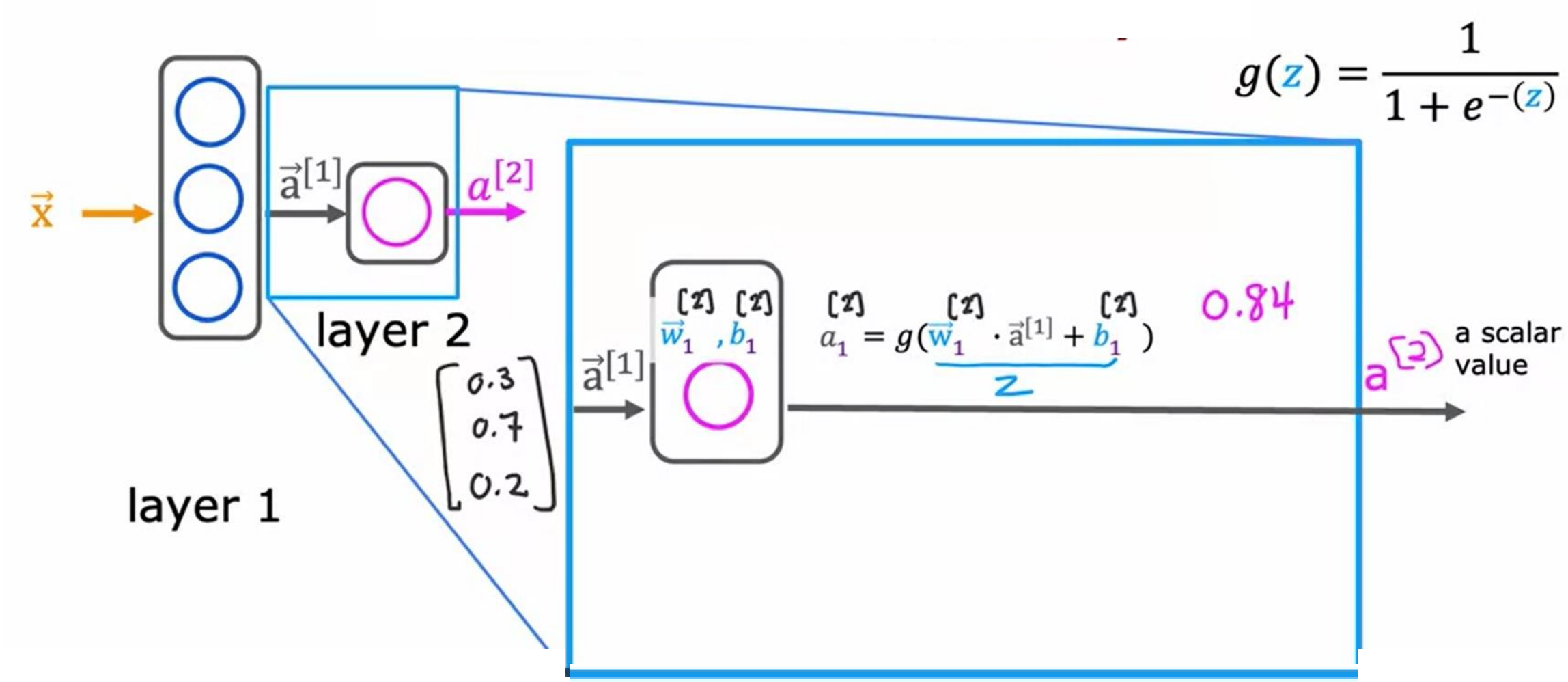
Neural Network Model

school of ai Algiers

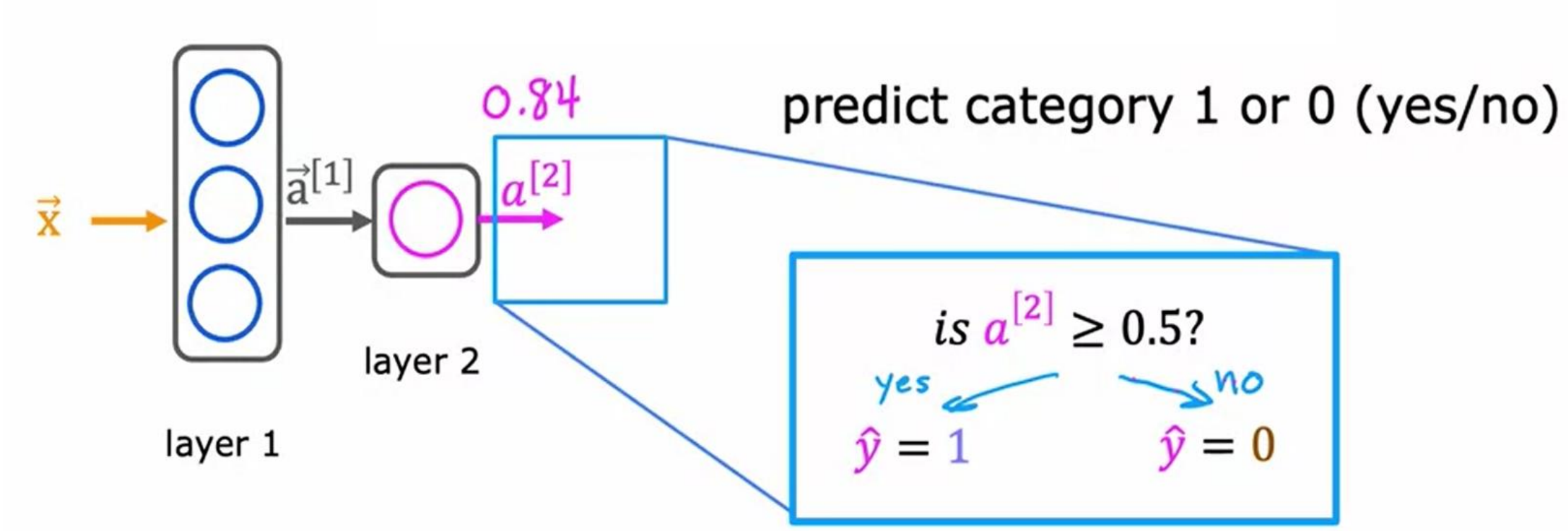
Neural network layer



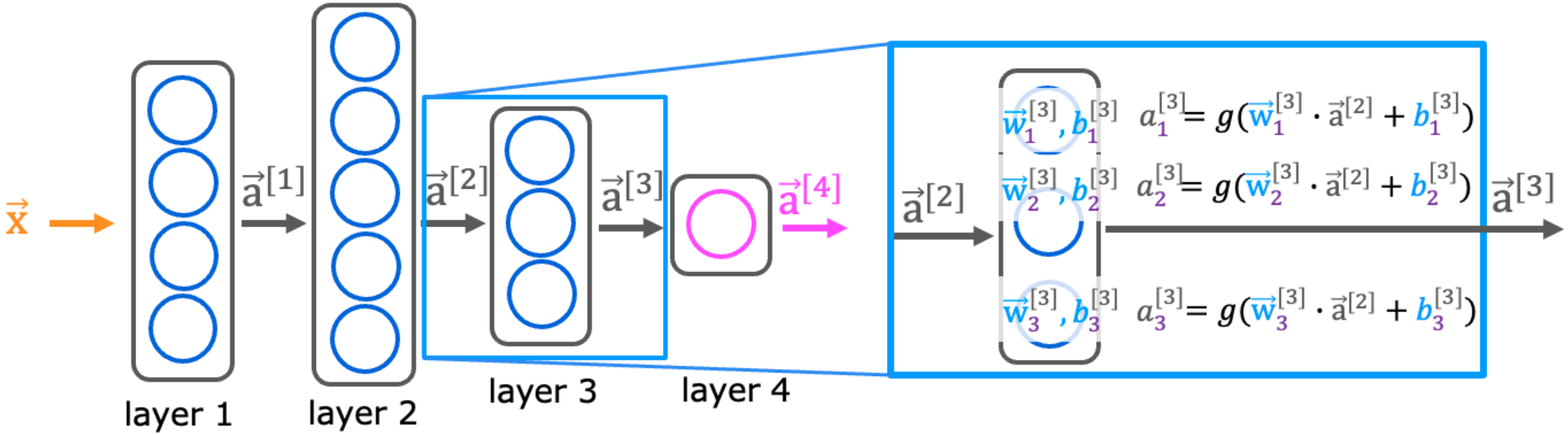
Neural network layer



Neural network layer

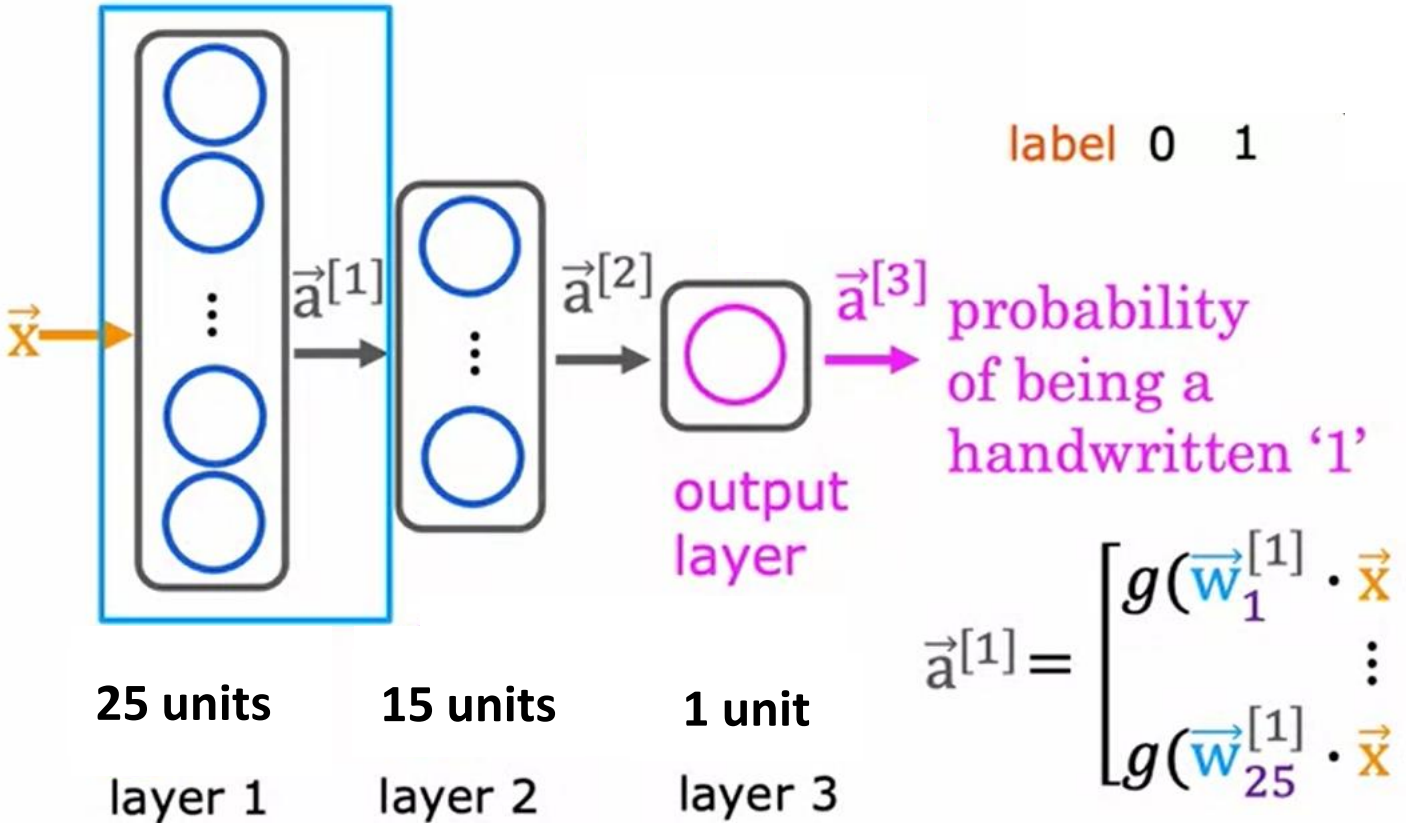


More complex neural networks



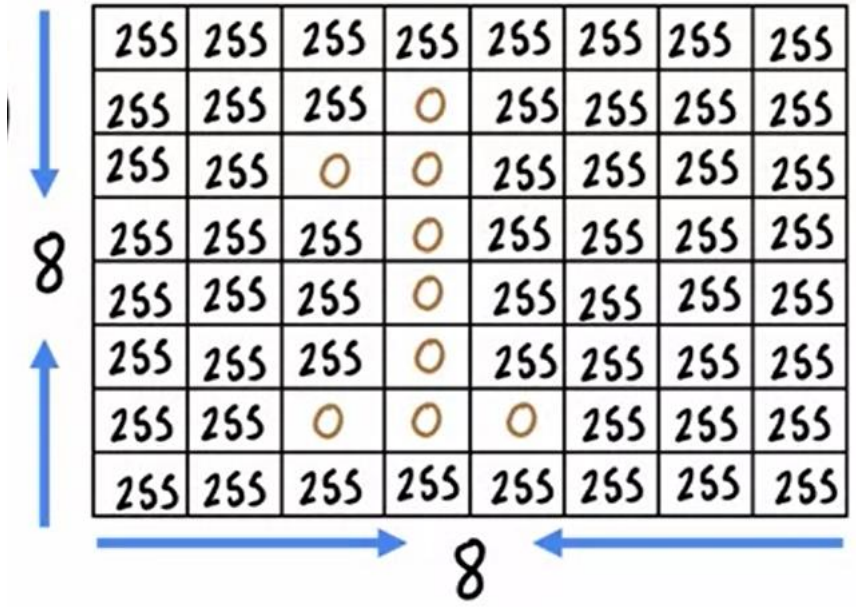
$$\boxed{a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})}$$

Inference: making predictions (Forward propagation)
Example: Handwritten digit recognition



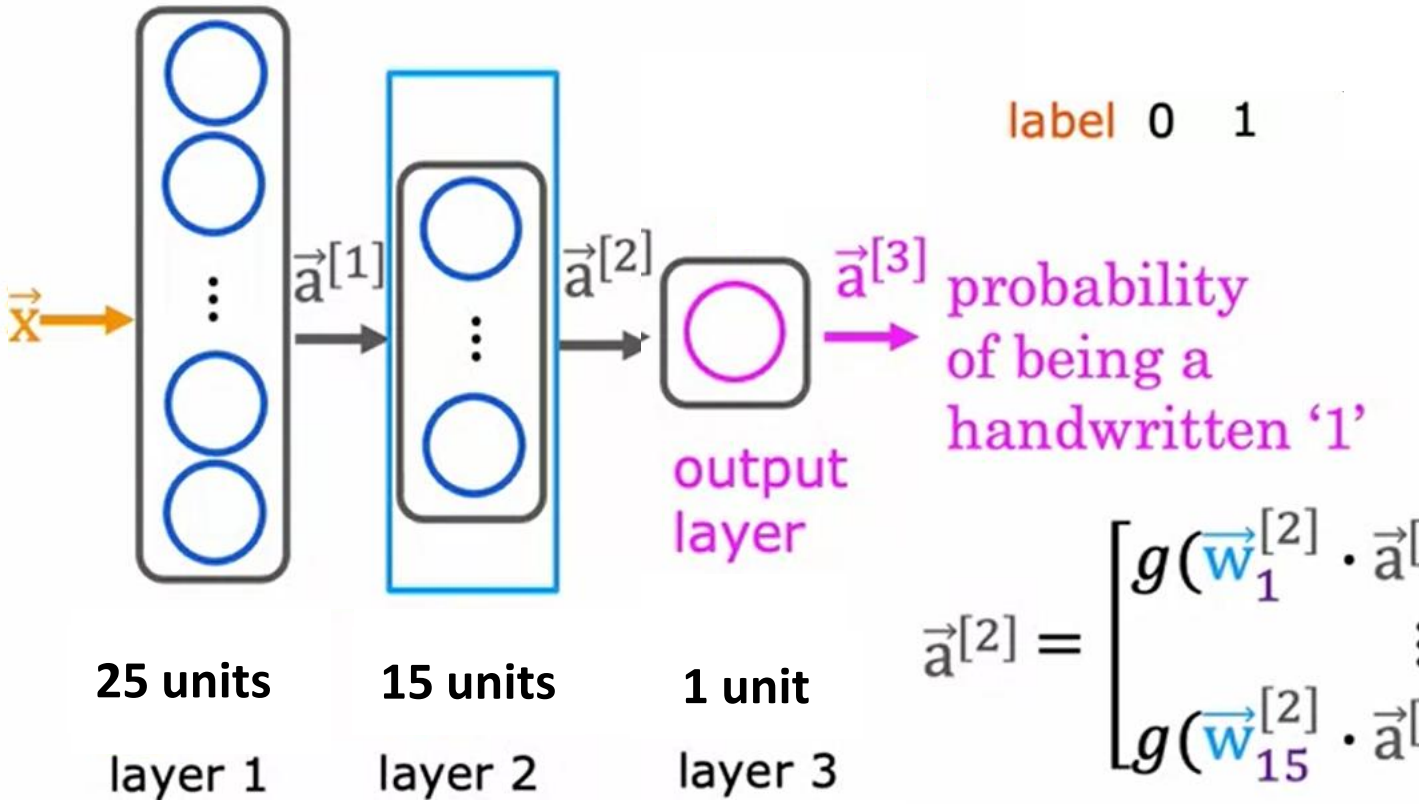
Digit images

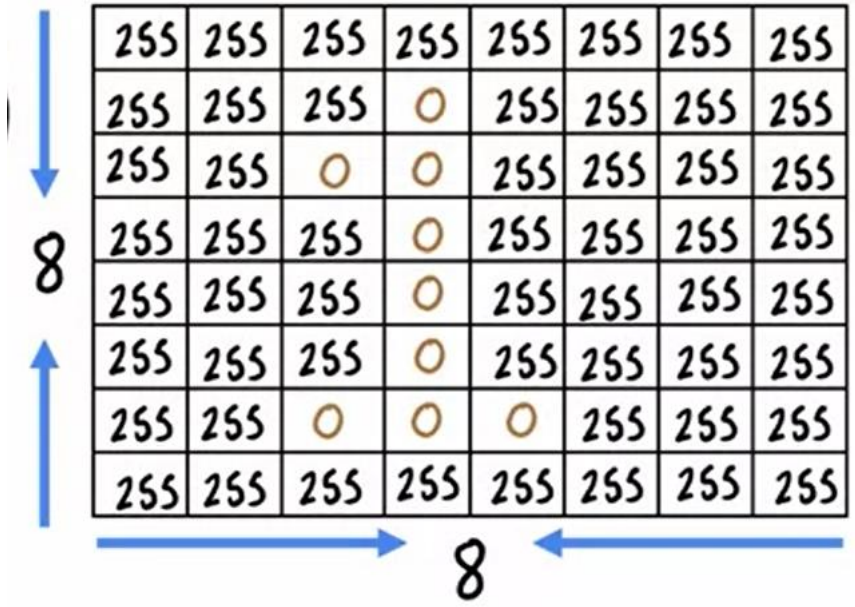
label 0 1



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

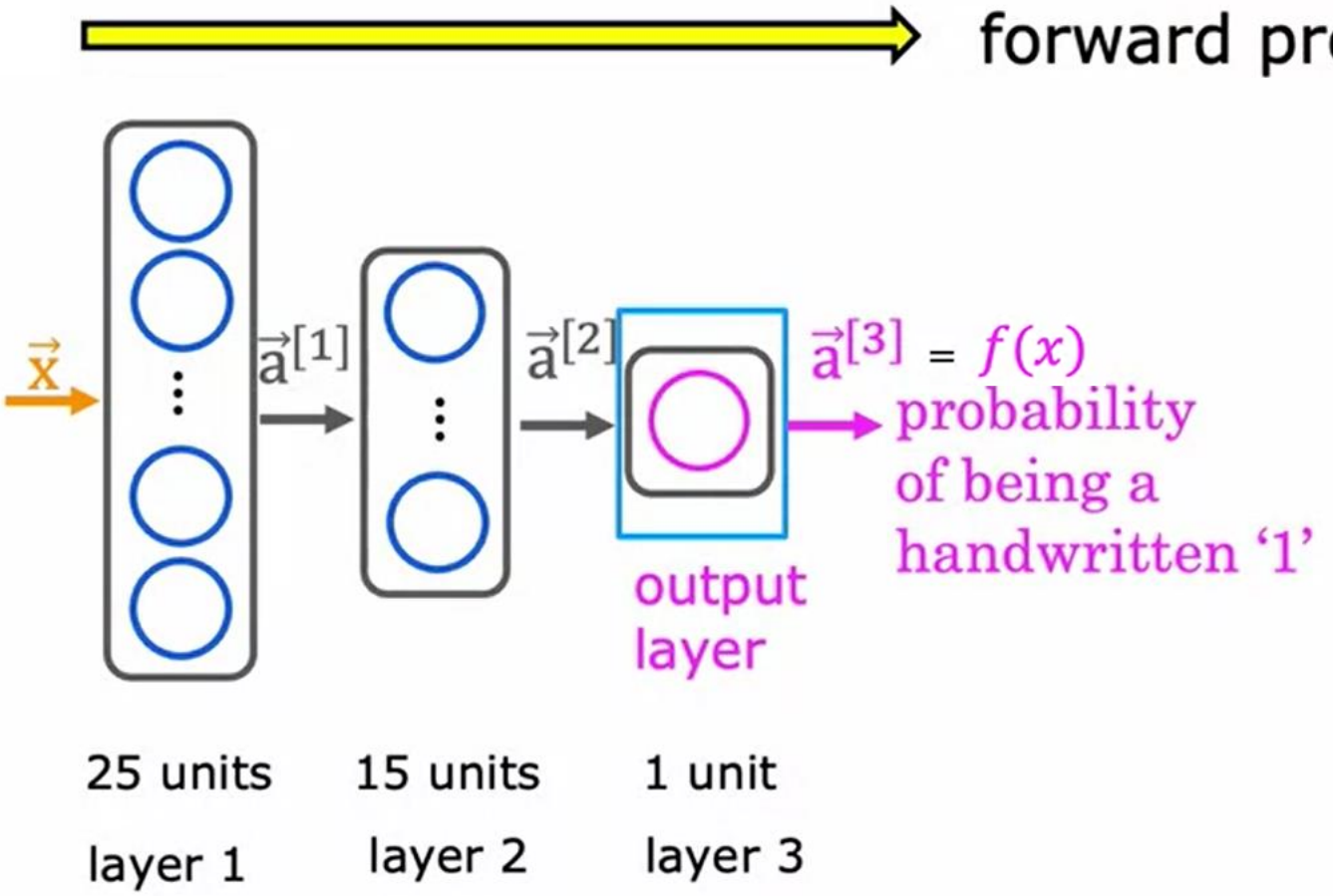
Inference: making predictions (Forward propagation)
Example: Handwritten digit recognition





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

Inference: making predictions (Forward propagation)
Example: Handwritten digit recognition



$$\vec{a}^{[3]} = \left[g\left(\vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]} \right) \right]$$

$$is\ a_1^{[3]} \geq 0.5?$$

yes → $\hat{y} = 1$ image is digit 1

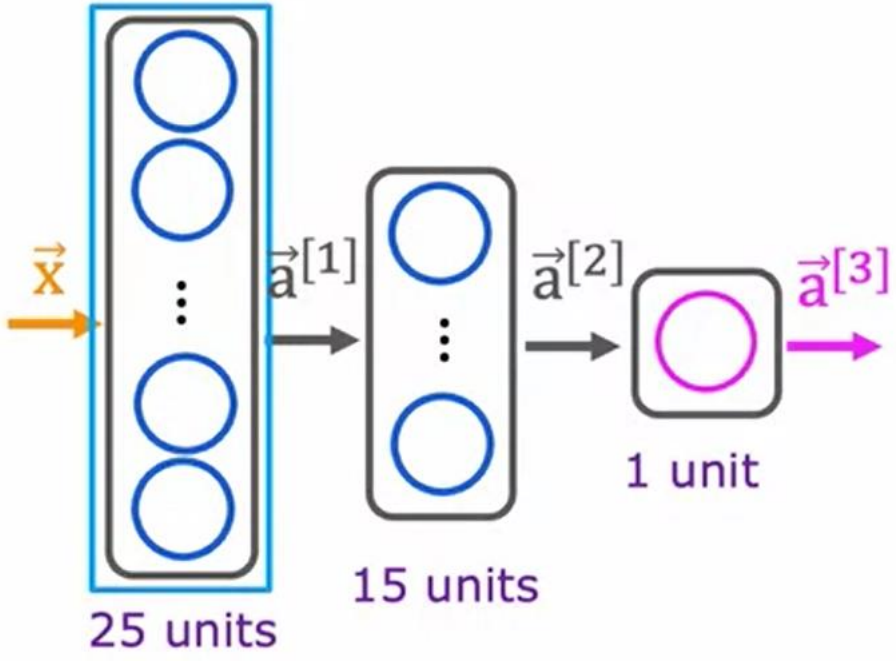
no → $\hat{y} = 0$ image isn't digit 1

AI CAMP

TensorFlow implementation

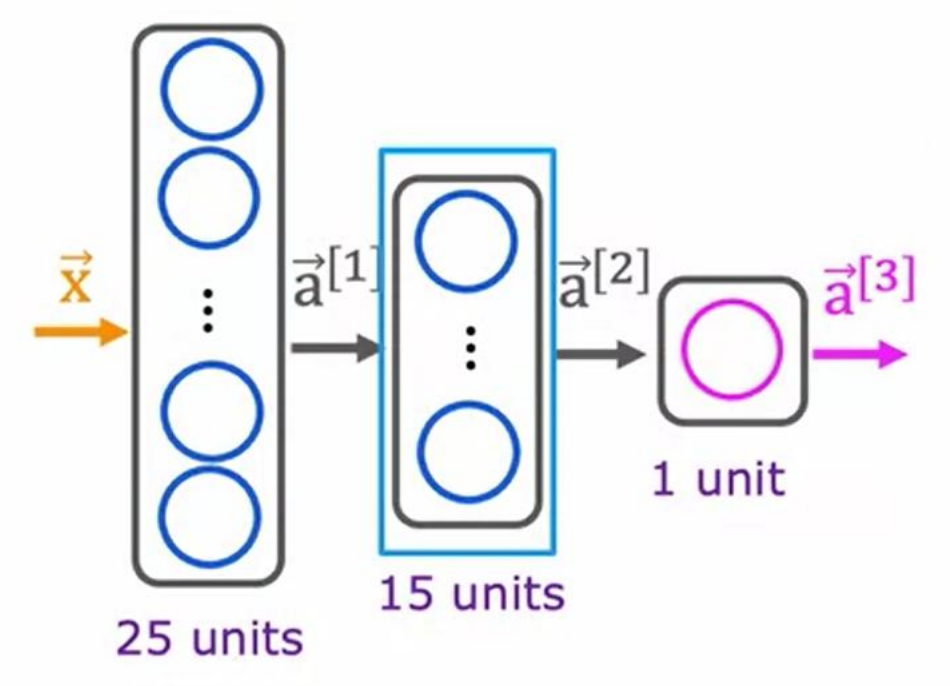
school of ai Algiers

Inference in Code
Model for digit classification




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

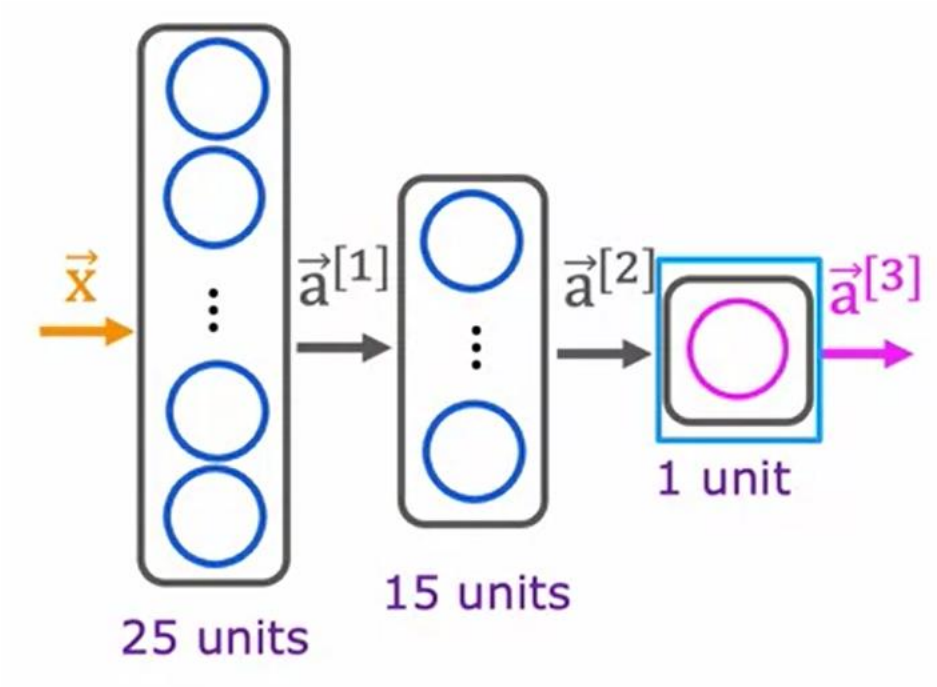
Inference in Code
Model for digit classification




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

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

Inference in Code
Model for digit classification

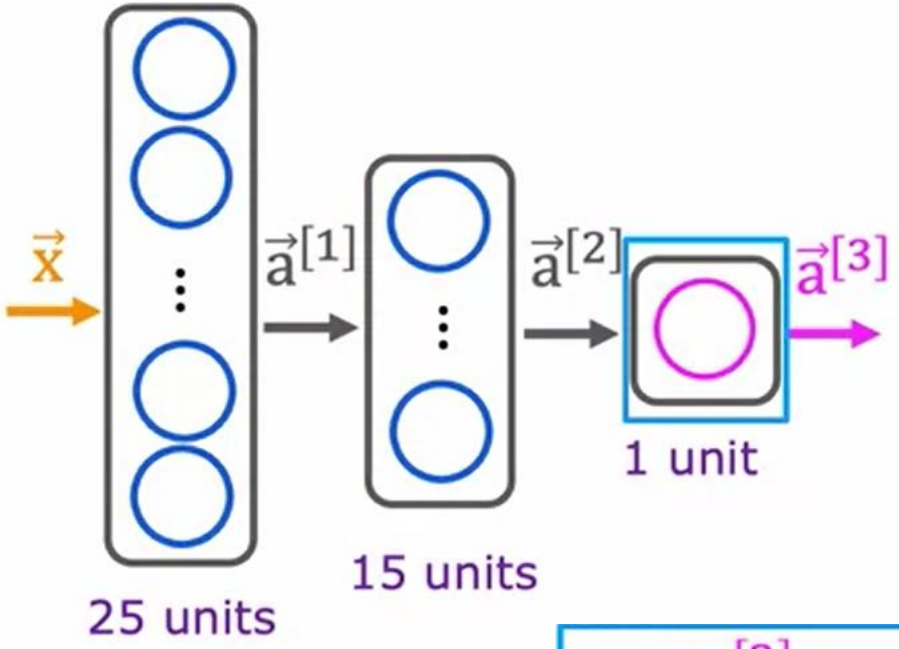



```
x = np.array([[0.0,...245,...240...0]])
layer_1 = Dense(units=25, activation='sigmoid')
a1 =
```

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

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

Inference in Code
Model for digit classification




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

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

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

$is\ a_1^{[3]} \geq 0.5?$

$\hat{y} = 1$ ✕ $\hat{y} = 0$ ○

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

Data in TensorFlow

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]]
```

Note about NumPy arrays

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

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

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

$$\begin{bmatrix} 0.1 & 0.2 \\ -3 & -4 \\ -.5 & -.6 \\ 7 & 8 \end{bmatrix}$$

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

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

Note about NumPy arrays

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

$$\begin{bmatrix} 200 & 17 \end{bmatrix}$$

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

$$\begin{bmatrix} 200 \\ 17 \end{bmatrix}$$

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

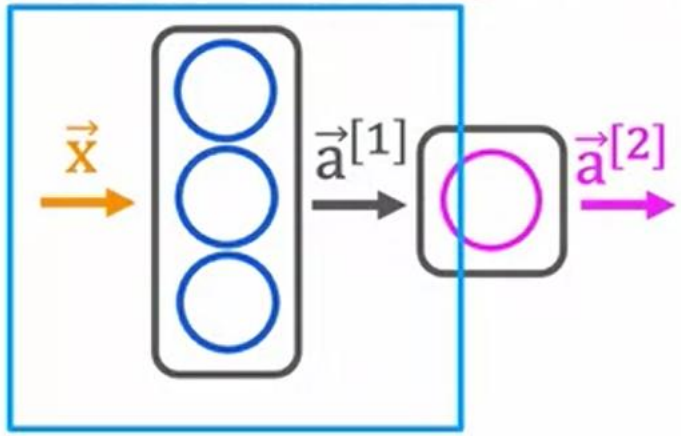
Data in TensorFlow

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]]
```

Activation vector



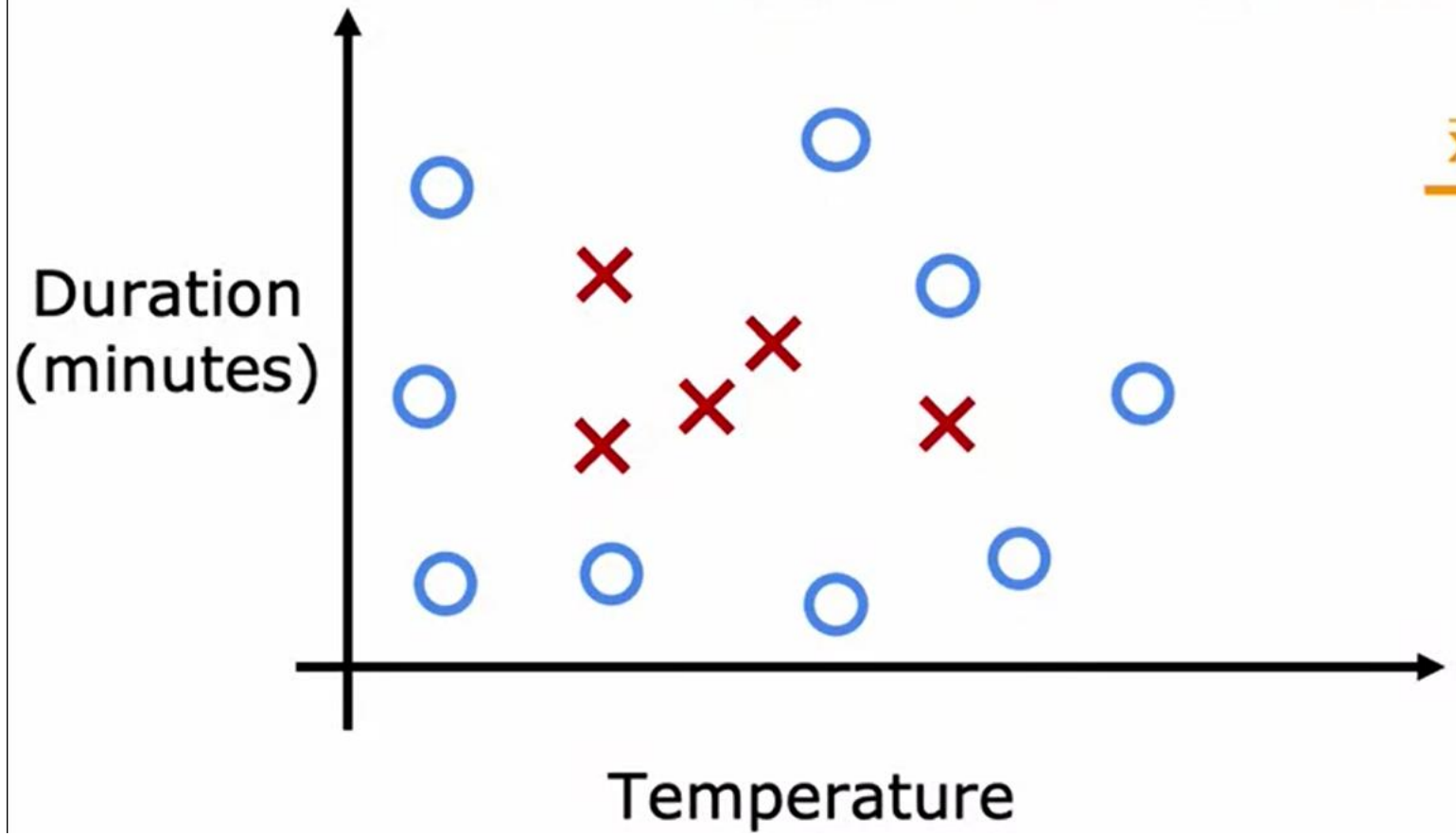

```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
```

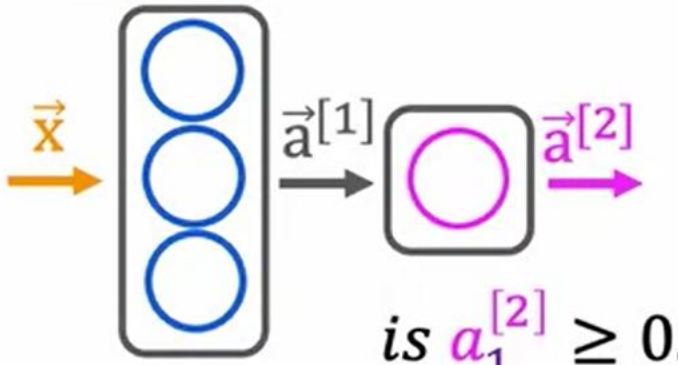
```
tf.Tensor([[0.2 0.7 0.3]], shape=(1, 3), dtype=float32)
```

```
a1.numpy()
```

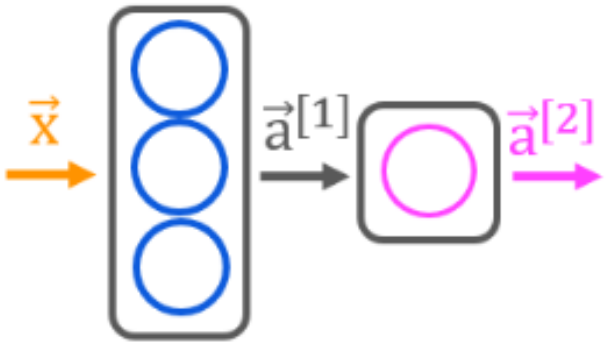
```
array([[0.2, 0.7, 0.3]], dtype=float32)
```

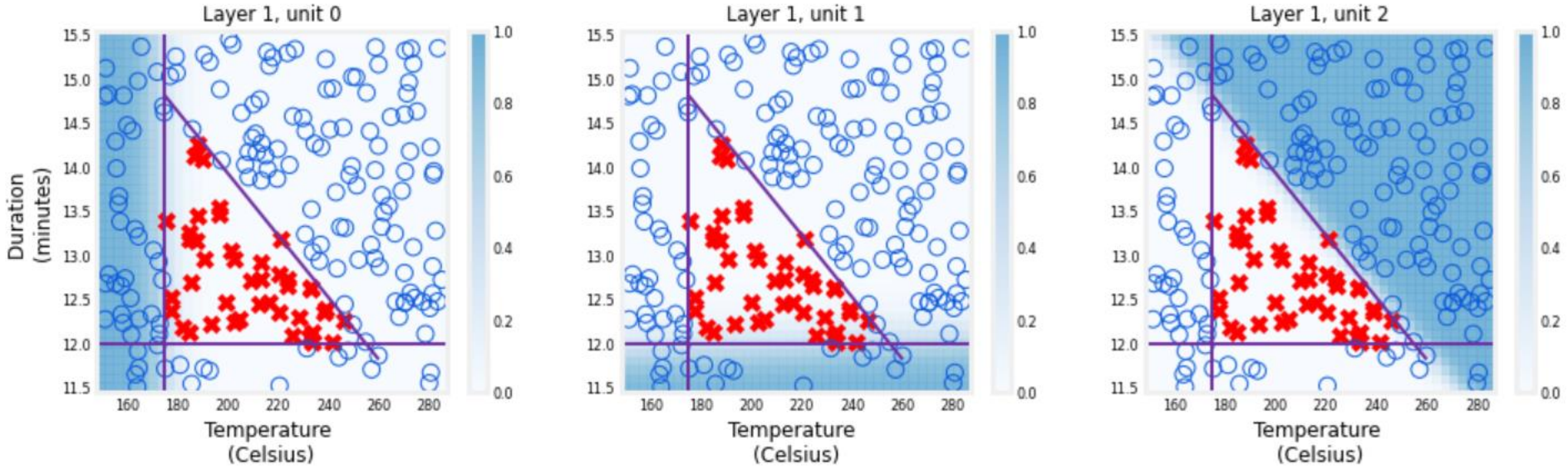
Coffee roasting

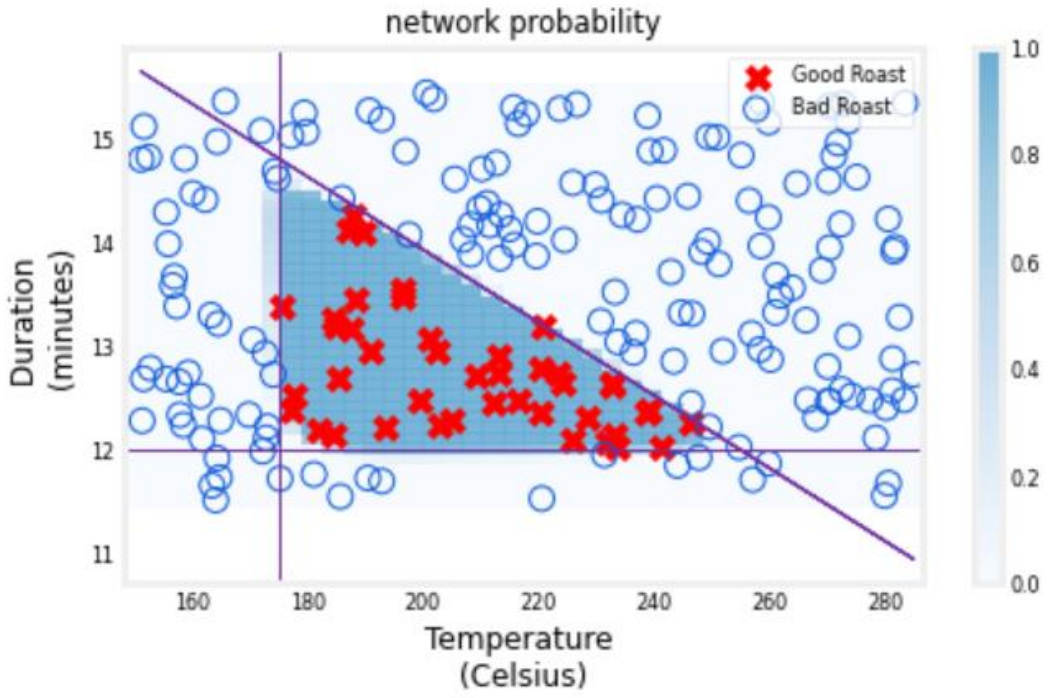


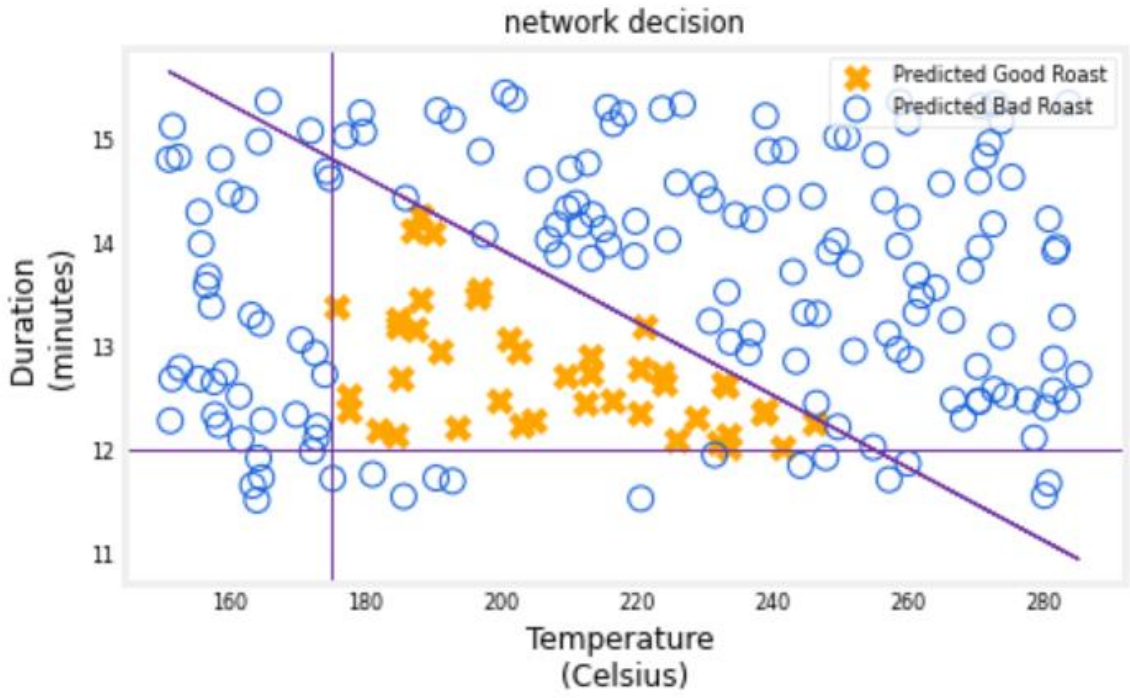


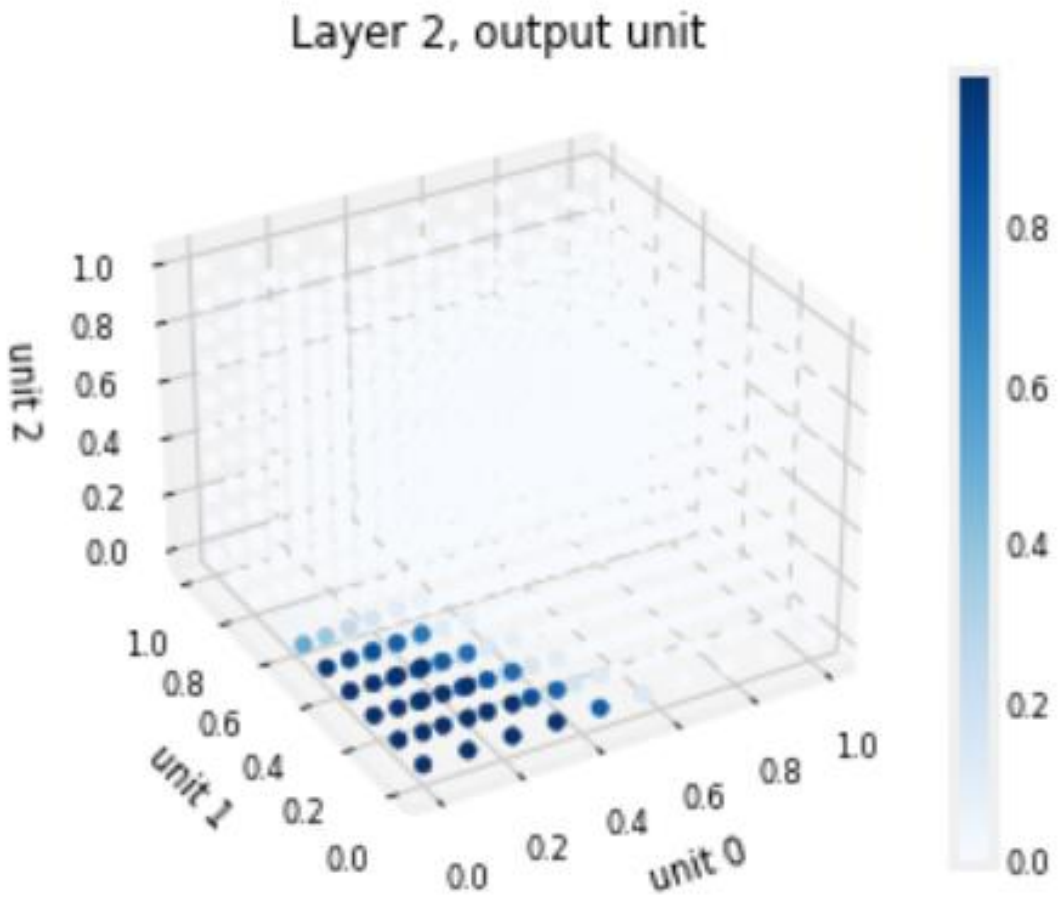
$is\ a_1^{[2]} \geq 0.5?$


$\vec{x}$
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$

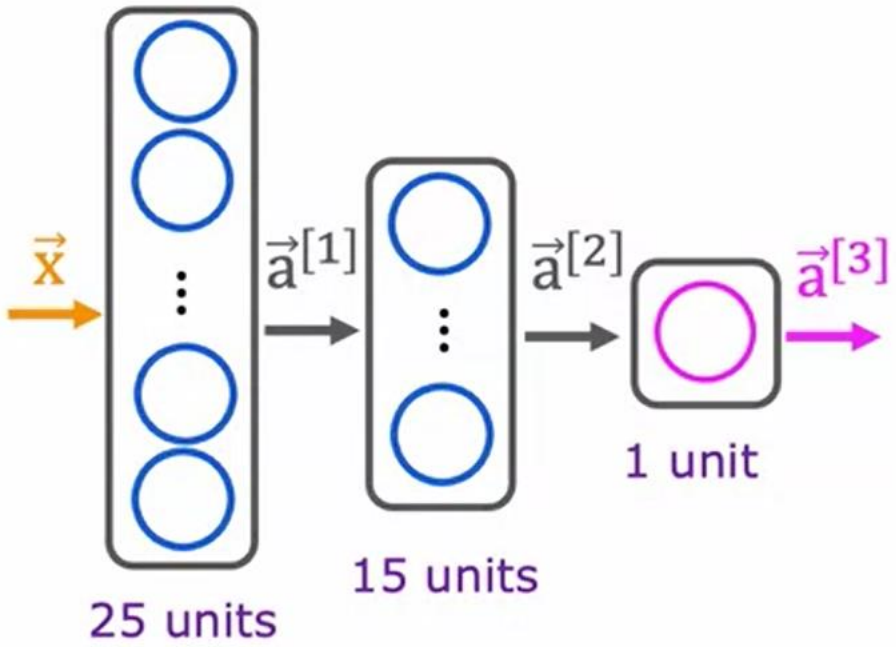

Layer 1, unit 0
Layer 1, unit 1
Layer 1, unit 2
Duration (minutes)
Temperature (Celsius)
15.5
15.0
14.5
14.0
13.5
13.0
12.5
12.0
11.5
160
180
200
220
240
260
280
1.0
0.8
0.6
0.4
0.2
0.0


network probability
Good Roast
Bad Roast
Duration (minutes)
Temperature (Celsius)
11
12
13
14
15
160
180
200
220
240
260
280
0.0
0.2
0.4
0.6
0.8
1.0


network decision
Predicted Good Roast
Predicted Bad Roast
Duration (minutes)
Temperature (Celsius)
11
12
13
14
15
160
180
200
220
240
260
280


Layer 2, output unit
unit 2
1.0
0.8
0.6
0.4
0.2
0.0
unit 1
1.0
0.8
0.6
0.4
0.2
0.0
unit 0
0.0
0.2
0.4
0.6
0.8
1.0
0.8
0.6
0.4
0.2
0.0

Building a Neural Network
Exp: Digit classification model




```
layer_1 = Dense(units=25, activation="sigmoid")
layer_2 = Dense(units=15, activation="sigmoid")
layer_3 = Dense(units=1, activation="sigmoid")

model = Sequential([layer_1, layer_2, layer_3])
model.compile(...)

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

model.fit(x,y)

model.predict(x_new)
```

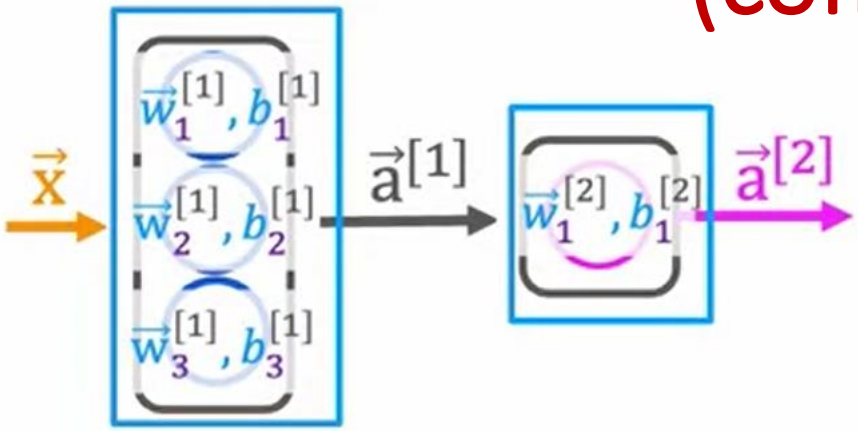
AI CAMP

Neural Network Implementation in Python

school of ai Algiers

Forward Prop in a single layer (coffee roasting example)

In a single layer



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

```
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)
```

w2_1

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

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

```
w1_1 = np.array([1, 2])
b1_1 = np.array([-1])
z1_1 = np.dot(w1_1,x)+b1_1
a1_1 = sigmoid(z1_1)
```

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

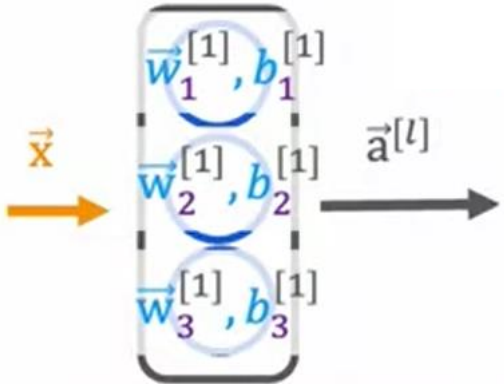
```
w1_2 = np.array([-3, 4])
b1_2 = np.array([1])
z1_2 = np.dot(w1_2,x)+b1_2
a1_2 = sigmoid(z1_2)
```

$a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]})$

```
w1_3 = np.array([5, -6])
b1_3 = np.array([2])
z1_3 = np.dot(w1_3,x)+b1_3
a1_3 = sigmoid(z1_3)
```

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

General implementation of forword propagation



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

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

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

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

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

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

```
def dense(a_in,W,b):
  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
```

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

AI CAMP

How neural netwoks are implemented efficiently?

school of ai Algiers

Matrix Multiplication

for loops vs. vectorization

```
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]]
```

AI CAMP

Artificial General Intelligence

school of ai Algiers

Lab : 1st assignment

AI CAMP

BREAK
5 min

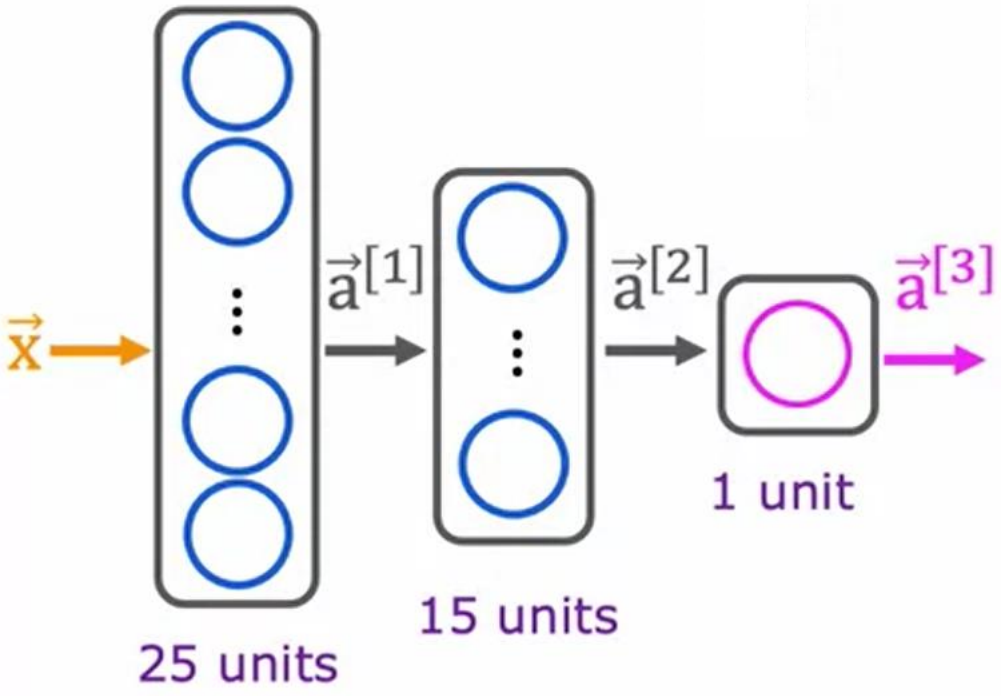
school of ai
Algiers

AI CAMP

Neural network training

school of ai Algiers

Neural network training



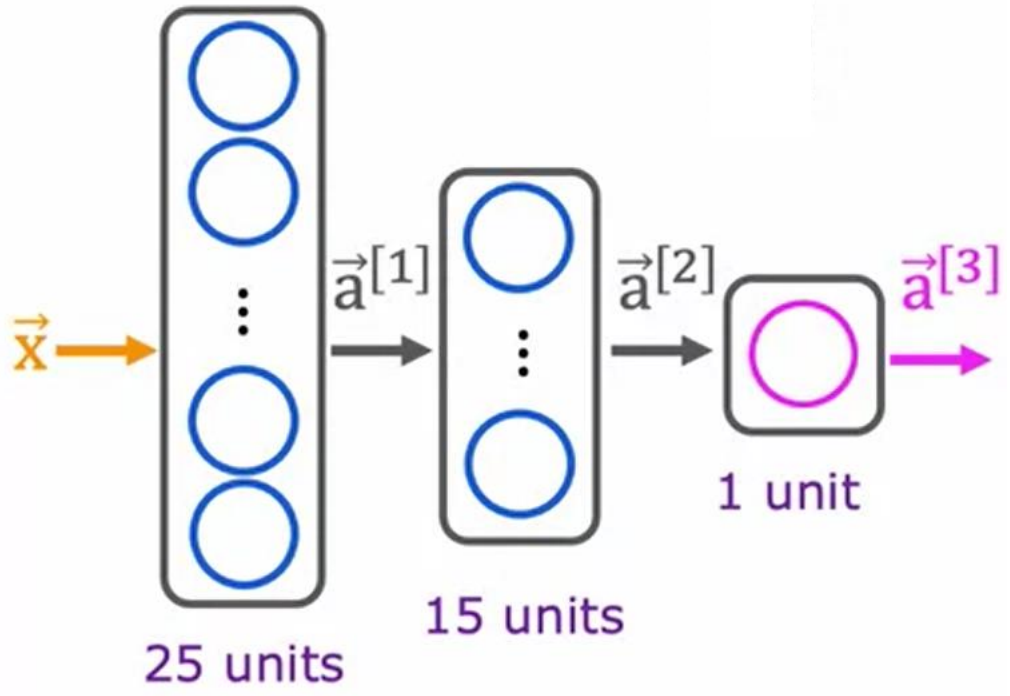

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
   Dense(units=25, activation='sigmoid'),
   Dense(units=15, activation='sigmoid'),
   Dense(units=1, activation='sigmoid'),
                    ])
from tensorflow.keras.losses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

Training details




```
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?

Model training steps

specify how to compute output given input x and parameters w,b (define model)

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

specify loss and cost

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

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

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

logistic regression

```
z = np.dot(w,x)+ b

f_x = 1/(1+np.exp(-z))
```

logistic loss

```
loss = -y * np.log(f_x)
   -(1-y) * np.log(1-f_x)
```

```
w = w - alpha * dj_dw
b = b - alpha * dj_db
```

neural network

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

binary cross entropy

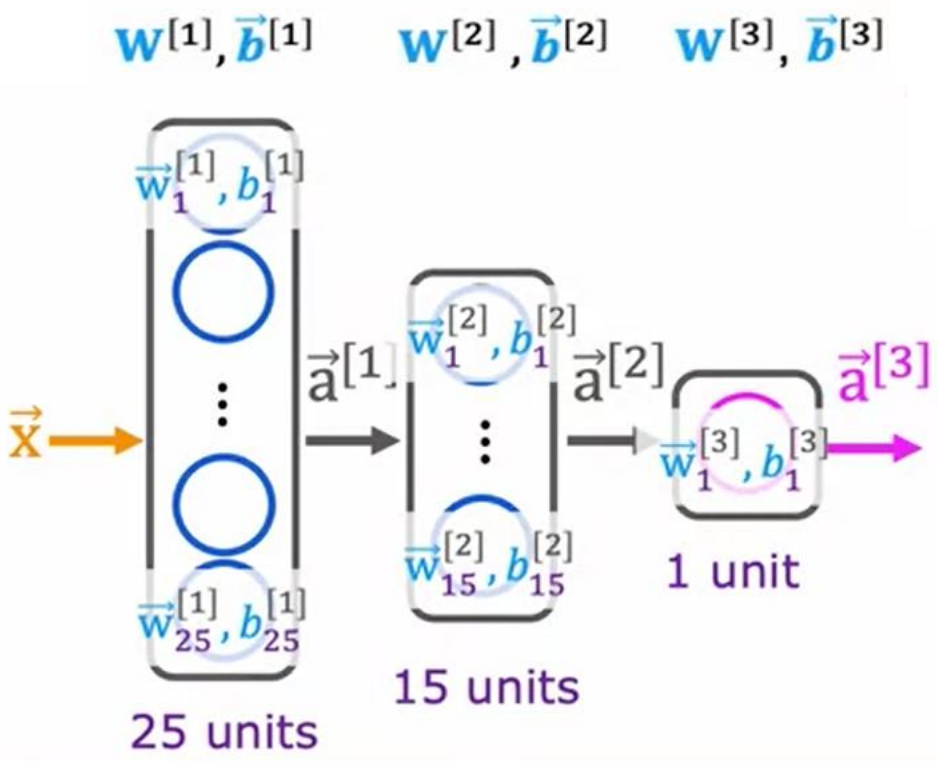
```
model.compile(
loss=BinaryCrossentropy())
```

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

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 function

handwritten digit classification problem

binary classification

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

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

$\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{W}^{[3]}$ $\vec{b}^{[1]}, \vec{b}^{[2]}, \vec{b}^{[3]}$

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

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

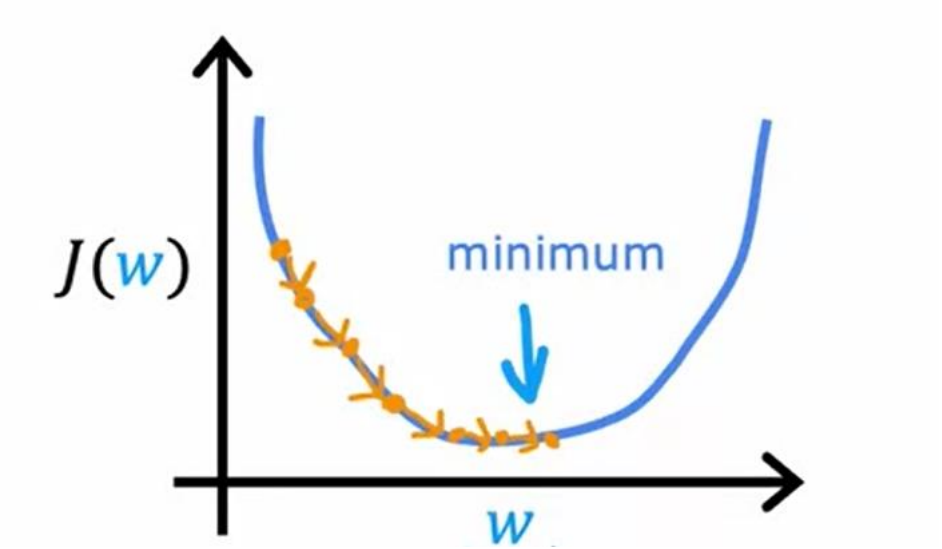
Keras

regression
(predicting numbers
and not categories)

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

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

3. Gradient descent



repeat {

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

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

}

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

Neural Network libraries

Use code libraries instead of coding "from scratch"

TensorFlow PyTorch

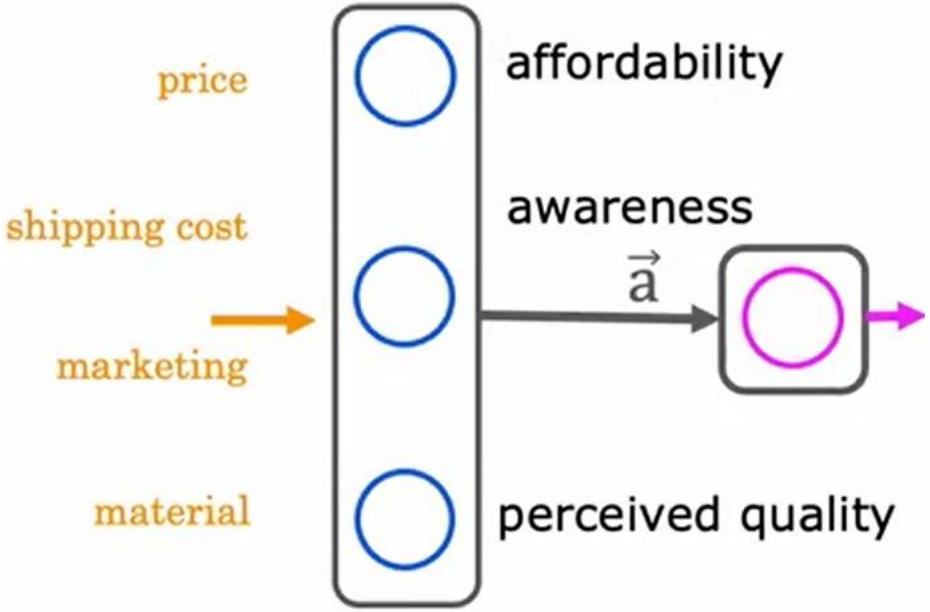
Good to understand the implementation
(for tuning and debugging).

AI CAMP

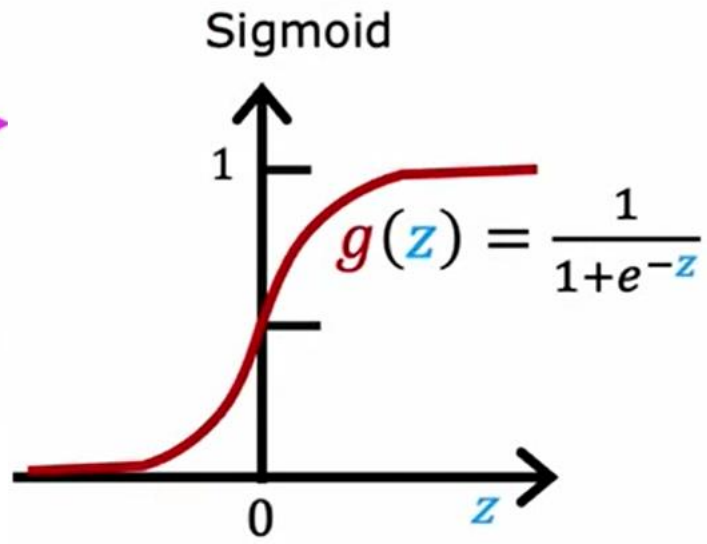
Activation functions

school of ai Algiers

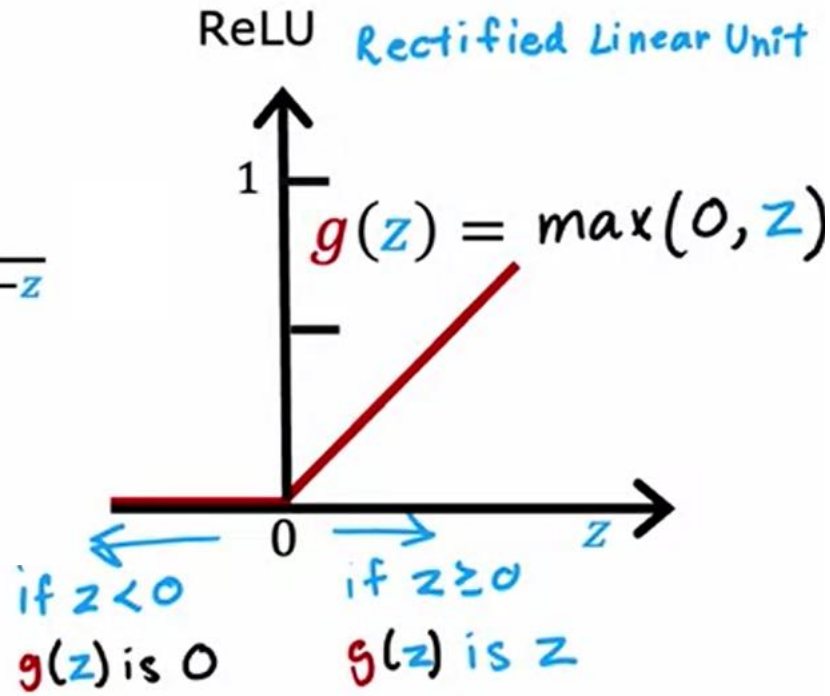
Alternatives to the sigmoid function
Demand prediction example



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



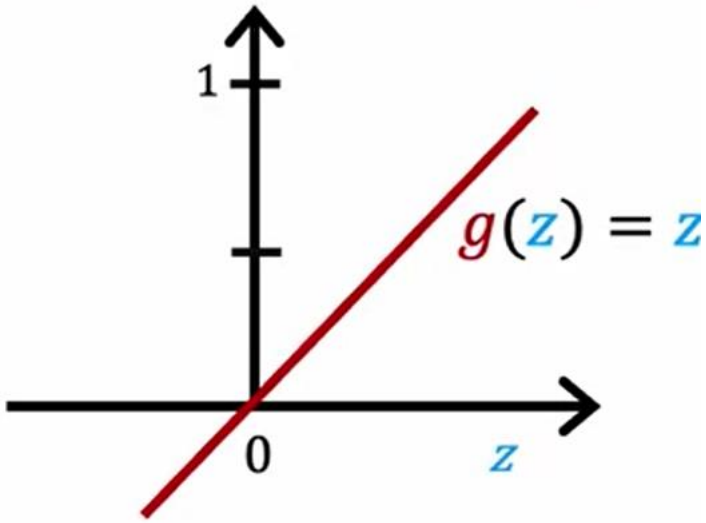
$0 < g(z) < 1$



Examples of activation functions

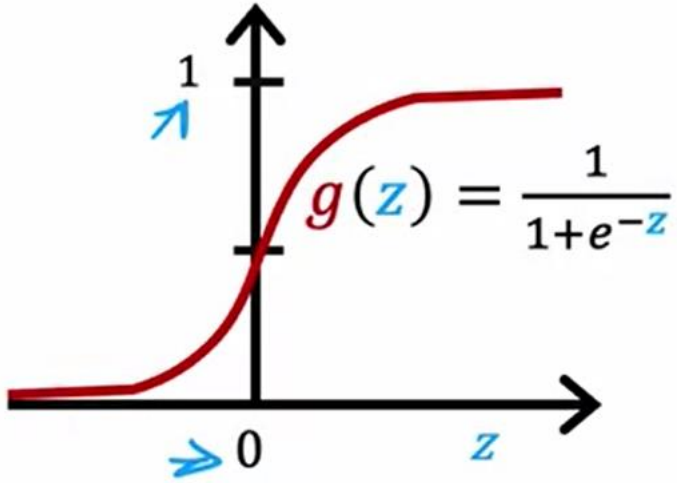
"No activation function"

Linear activation function



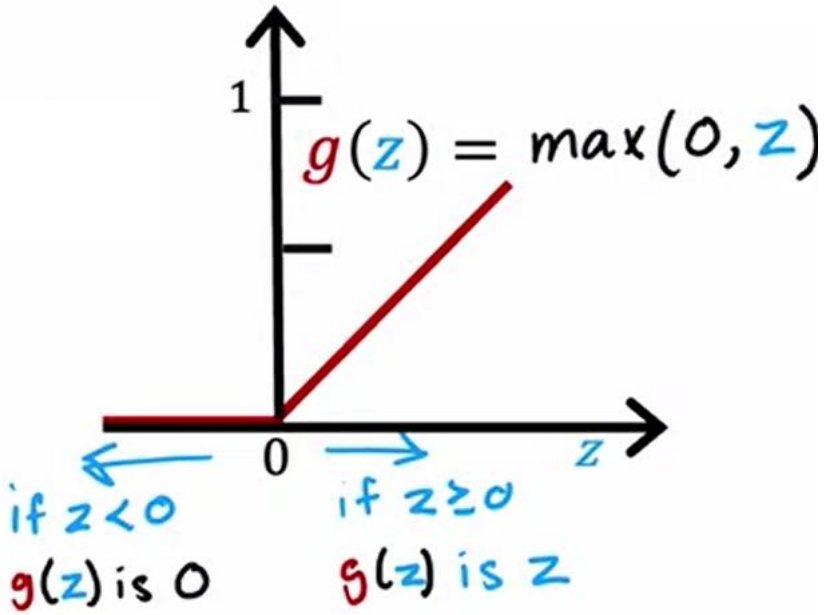
$a = g(z) = \underbrace{\vec{w} \cdot \vec{x} + b}_{z}$

Sigmoid



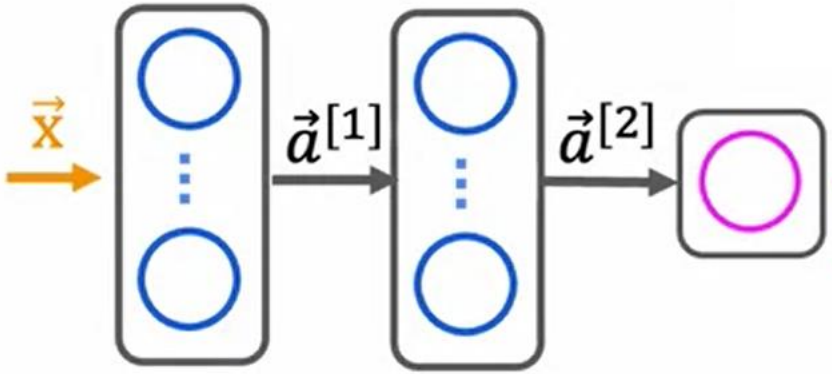
$0 < g(z) < 1$

ReLU "Rectified Linear Unit"



Choosing activation functions

Output layer

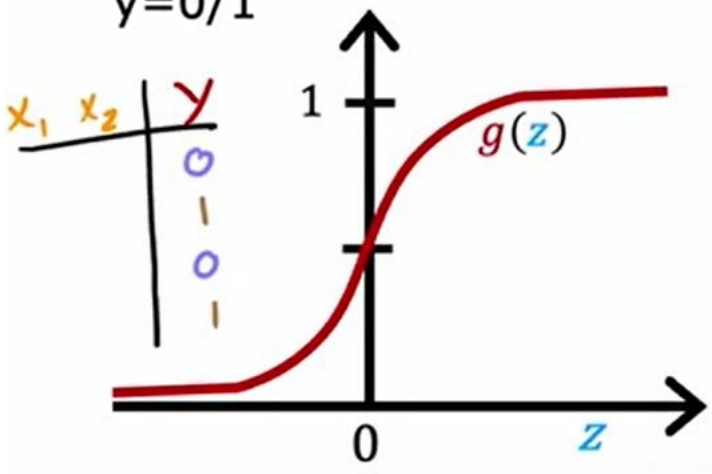


$\vec{a}^{[3]} = f(\vec{x})$

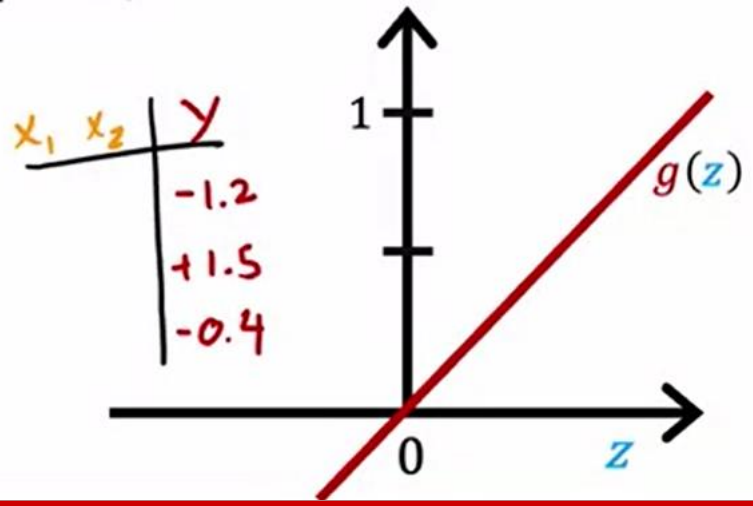
$f(\vec{x}) = a_1^{[3]} = g(z_1^{[3]})$

Choosing $g(z)$ for output layer?

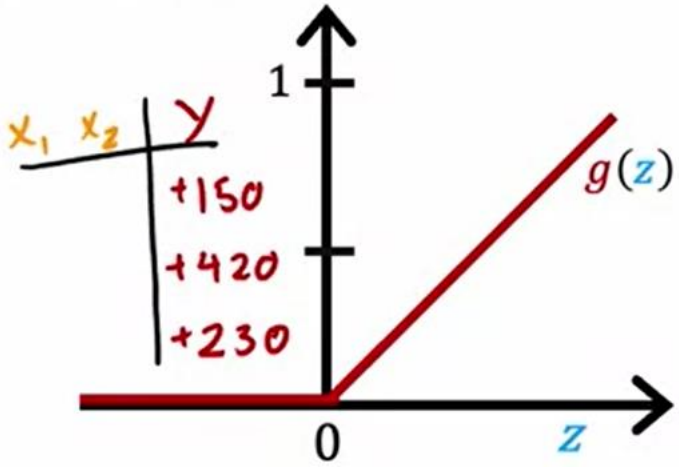
Binary classification
Sigmoid
y=0/1



Regression
Linear activation function
y = +/-

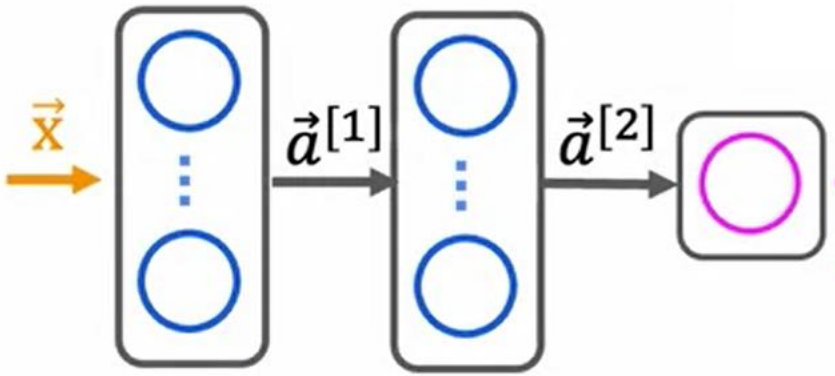


Regression
ReLU
Y = 0 or +



Choosing activation functions

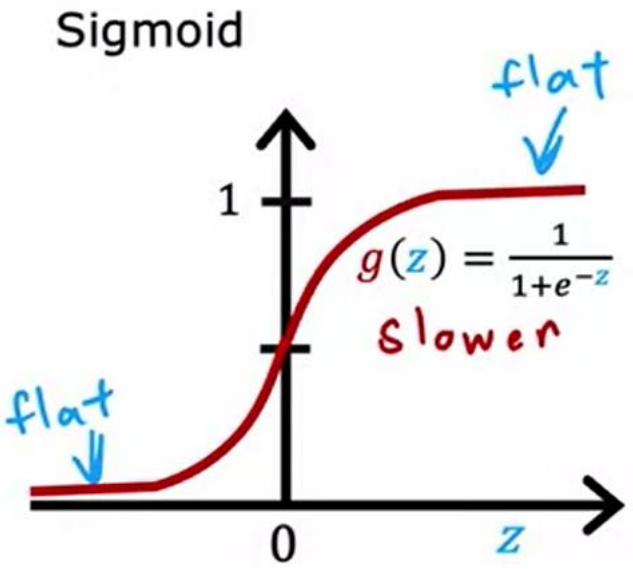
Hidden layer

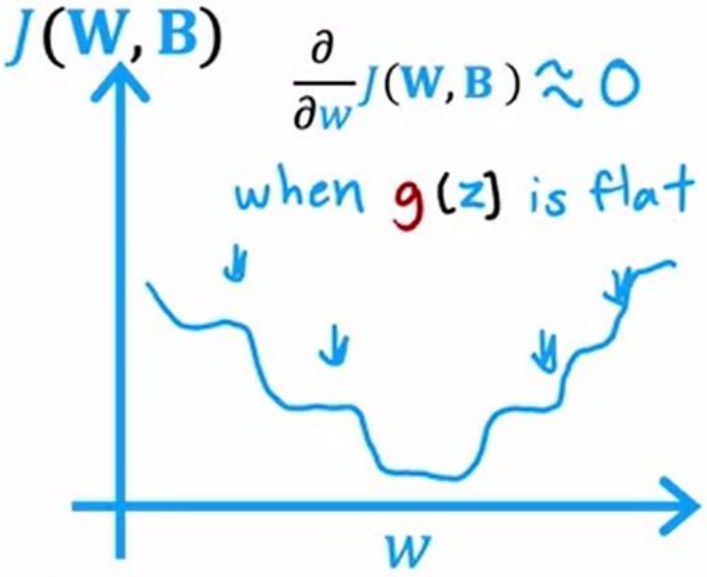


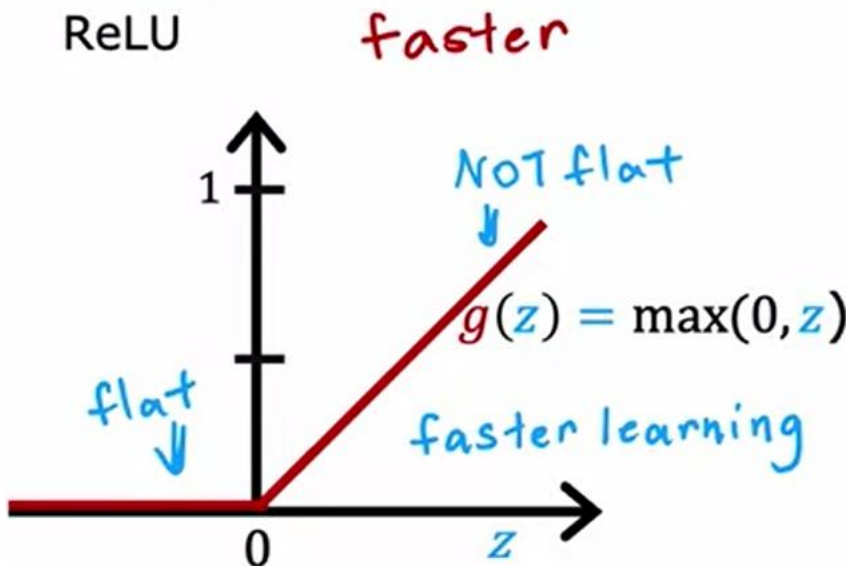
$\vec{a}^{[3]} = f(\vec{x})$

$f(\vec{x}) = a_1^{[3]} = g(z_1^{[3]})$

Choosing $g(z)$ for hidden layer

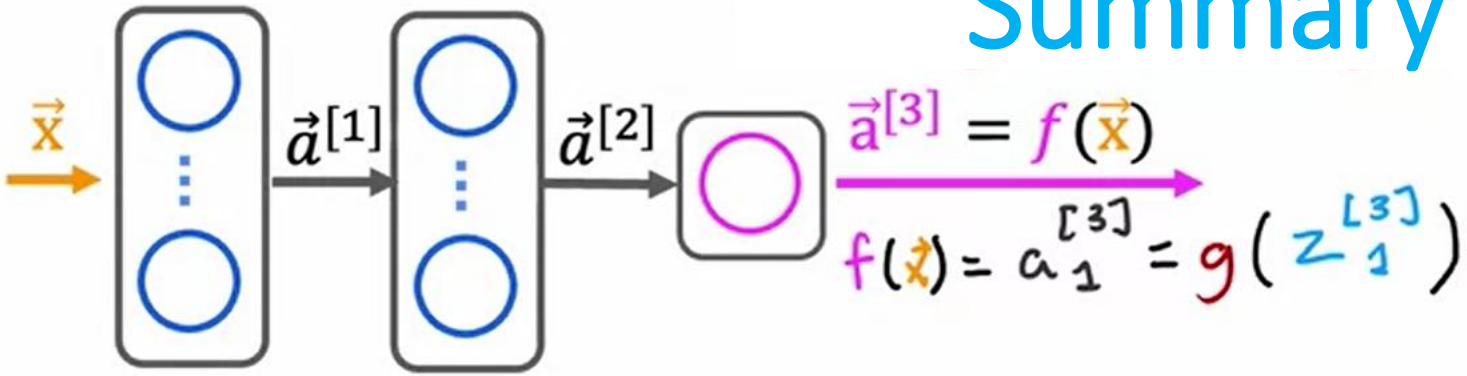






activation functions

Summary



ReLU

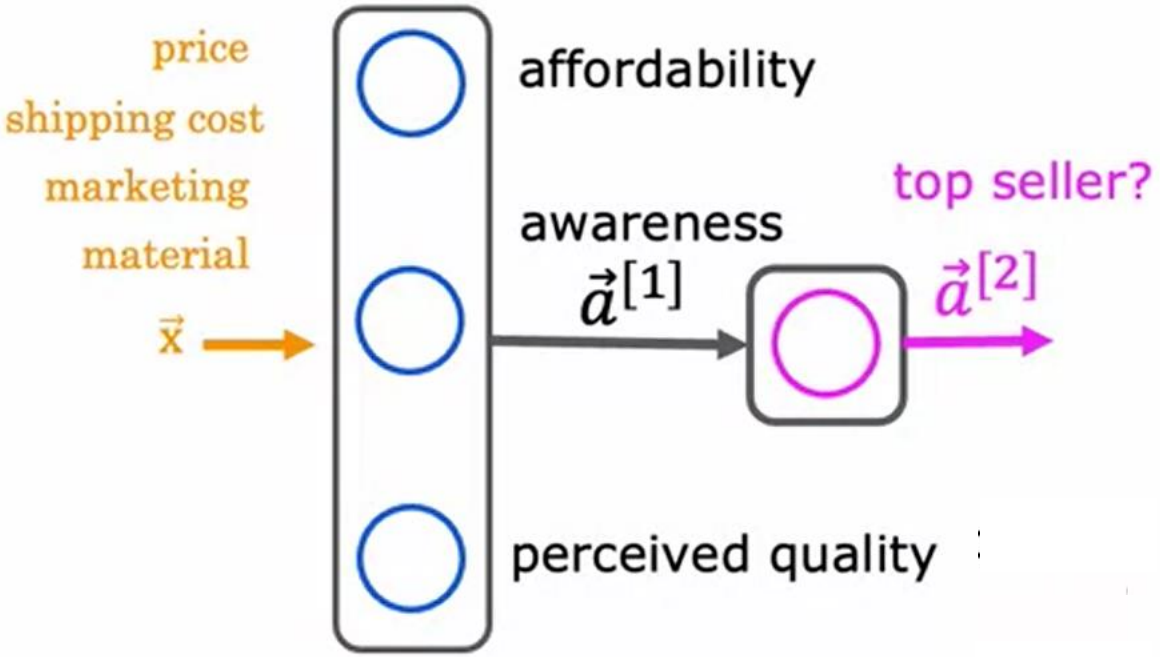
activation='sigmoid'

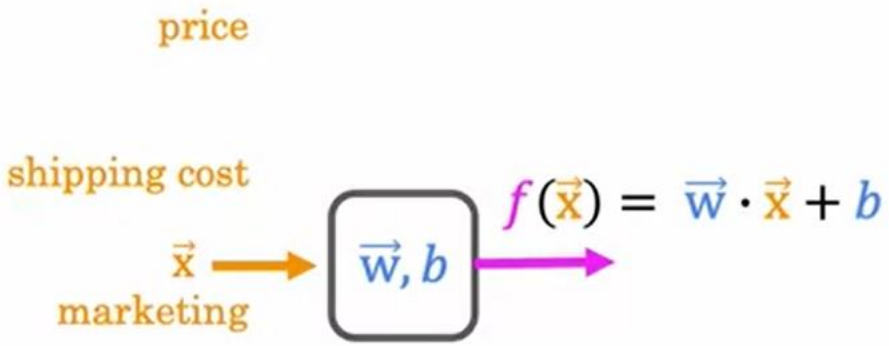
activation='linear

activation='relu'

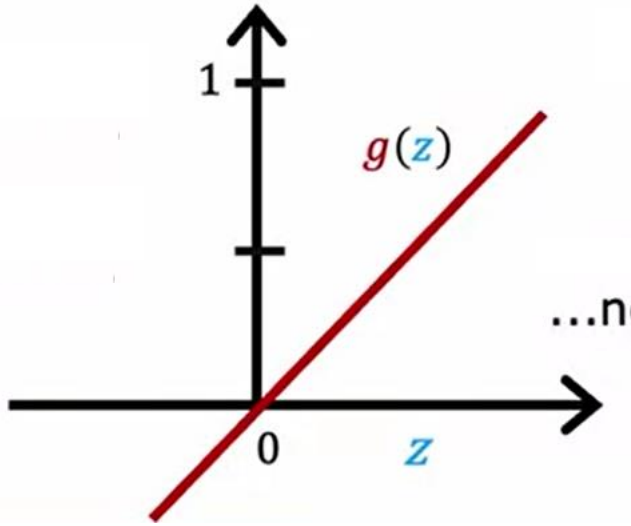
```
from tf.keras.layers import Dense
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=1,  activation='sigmoid')
])
```

Why do we need activation functions?



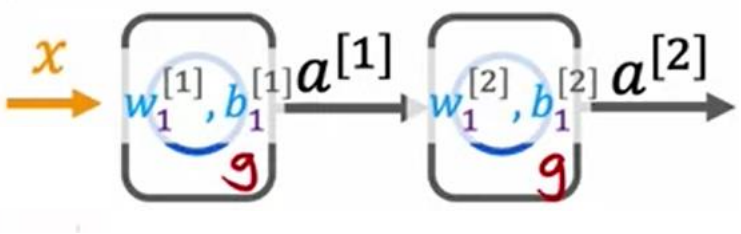


if all $g(z)$ are linear...



...no different than linear regression

Linear example



$$g(z) = z$$

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

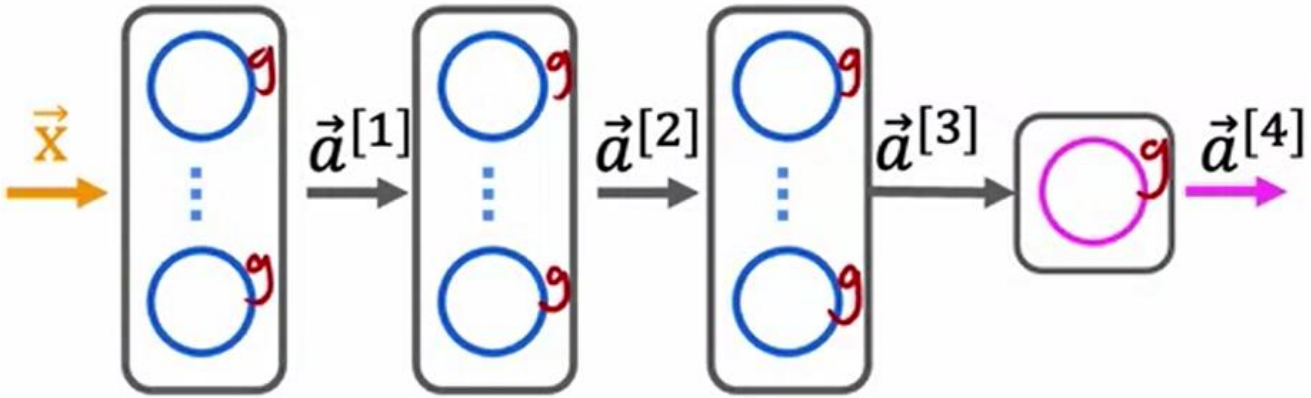
$$a^{[2]} = w_1^{[2]}\, a^{[1]} + b_1^{[2]}$$

$$=$$

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

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

example



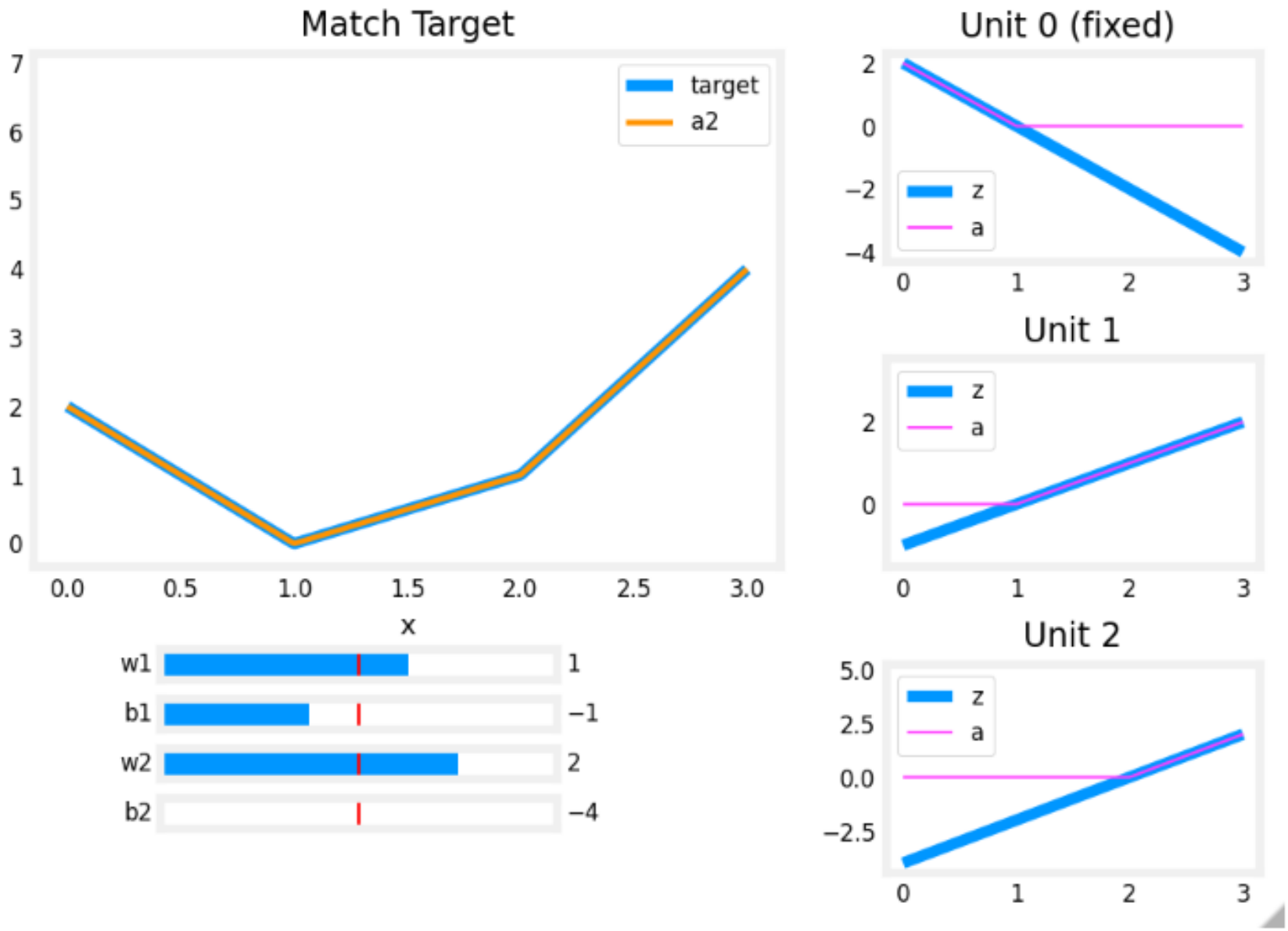
$g(z) = z$

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

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

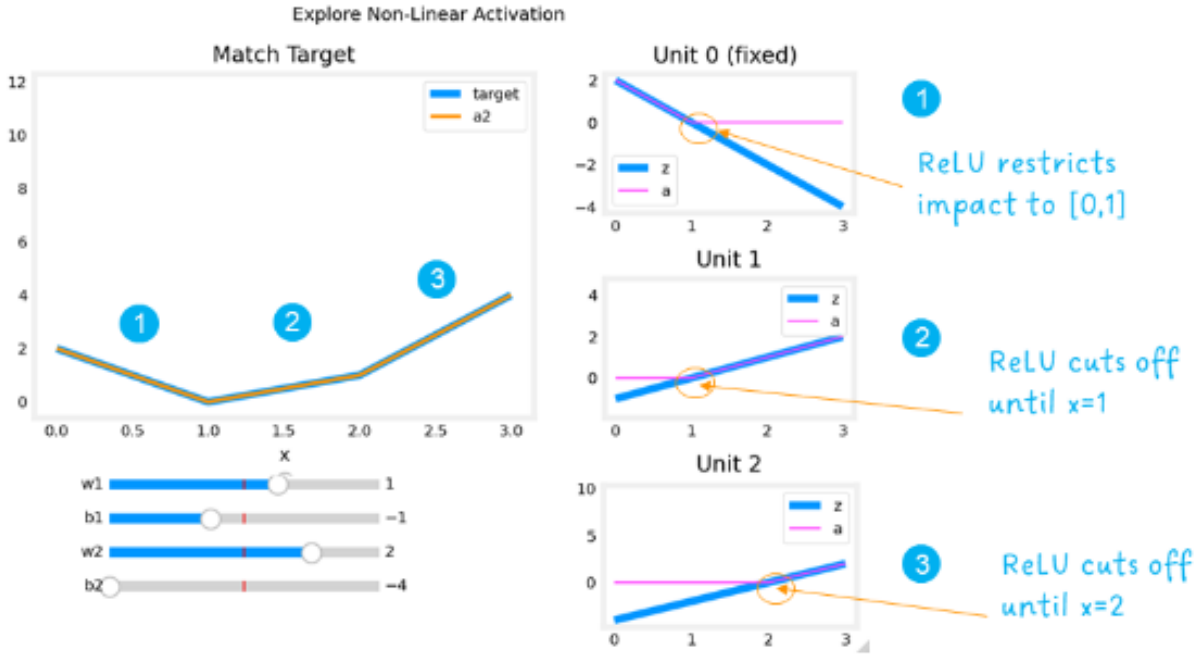
Don't use linear activations in hidden layers

Explore Non-Linear Activation



The goal of this exercise is to appreciate how the ReLU's non-linear behavior provides the needed ability to turn functions off until they are needed. Let's see how this worked in this example. The plots on the right contain the output of the units in the first layer.

Starting at the top, unit 0 is responsible for the first segment marked with a 1. Both the linear function z and the function following the ReLU a are shown. You can see that the ReLU cuts off the function after the interval [0,1]. This is important as it prevents Unit 0 from interfering with the following segment.



Unit 1 is responsible for the 2nd segment. Here the ReLU kept this unit quiet until after x is 1. Since the first unit is not contributing, the slope for unit 1, $w_1^{[1]}$, is just the slope of the target line. The bias must be adjusted to keep the output negative until x has reached 1. Note how the contribution of Unit 1 extends to the 3rd segment as well.

Unit 2 is responsible for the 3rd segment. The ReLU again zeros the output until x reaches the right value.The slope of the unit, $w_2^{[1]}$, must be set so that the sum of unit 1 and 2 have the desired slope. The bias is again adjusted to keep the output negative until x has reached 2.

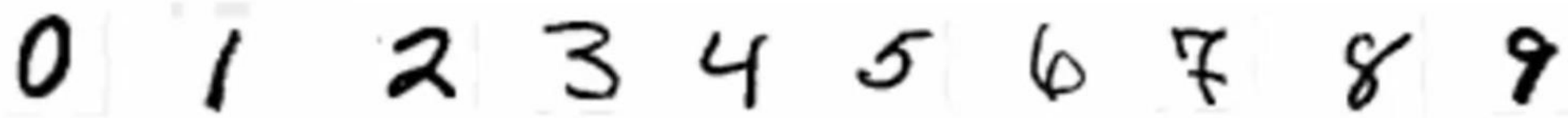
The "off" or disable feature of the ReLU activation enables models to stitch together linear segments to model complex non-linear functions.

AI CAMP

Multiclass classification

school of ai Algiers

Multiclass classification
MNIST example



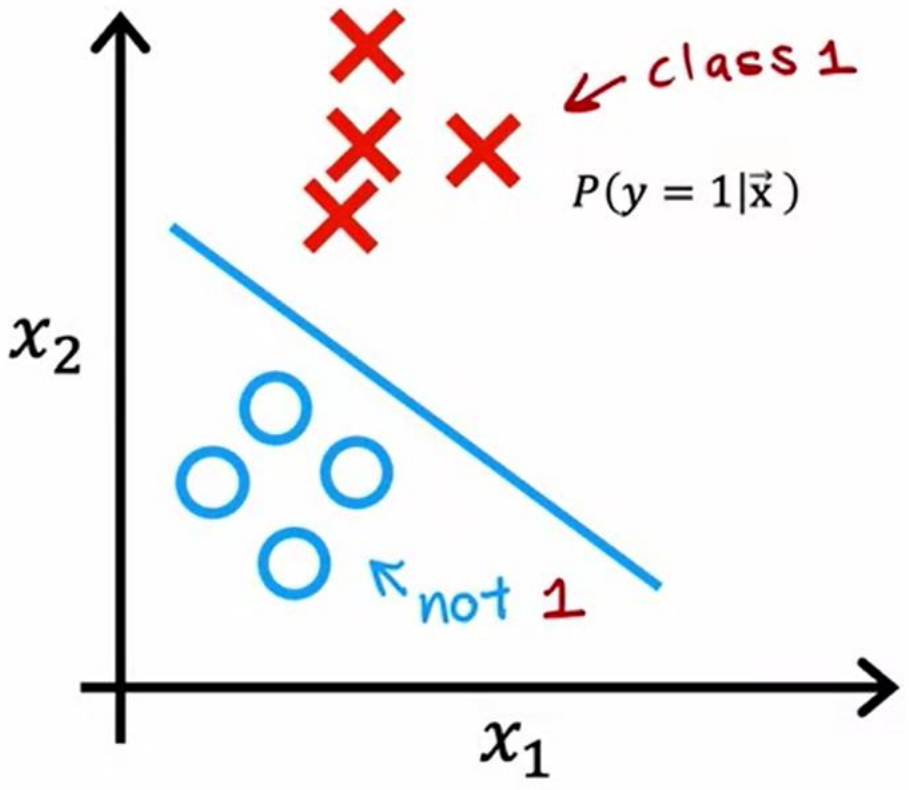
$y = 0$ 1 2 3 4 5 6 7 8 9

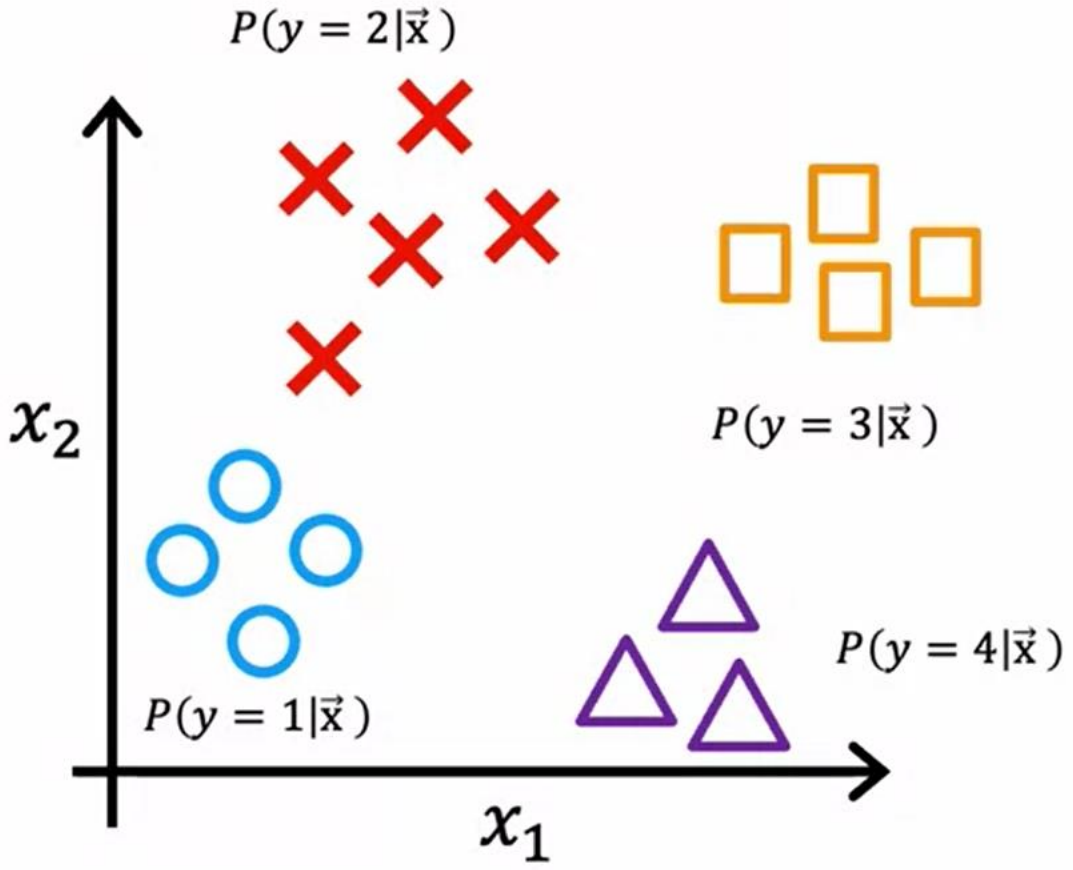
$y = 7$

Multiclass classification problem:

Target y can take more than two possible values

Multiclass classification





Logistic regression
(2 possible output values)

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

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

○

Softmax regression
(N possible outputs)

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

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

Softmax regression (4 possible outputs)

✕ $z_1 = \vec{w}_1 \cdot \vec{x} + b_1$

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

✕ ○ □ △

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

○ $z_2 = \vec{w}_2 \cdot \vec{x} + b_2$

$$a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

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

□ $z_3 = \vec{w}_3 \cdot \vec{x} + b_3$

$$a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

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

△ $z_4 = \vec{w}_4 \cdot \vec{x} + b_4$

$$a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

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

Cost

Logistic regression

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

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

$$a_2 = 1 - a_1 \quad = P(y = 0|\vec{x})$$

$$loss = -y \log a_1 - (1 - y) \log(1 - a_1)$$

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

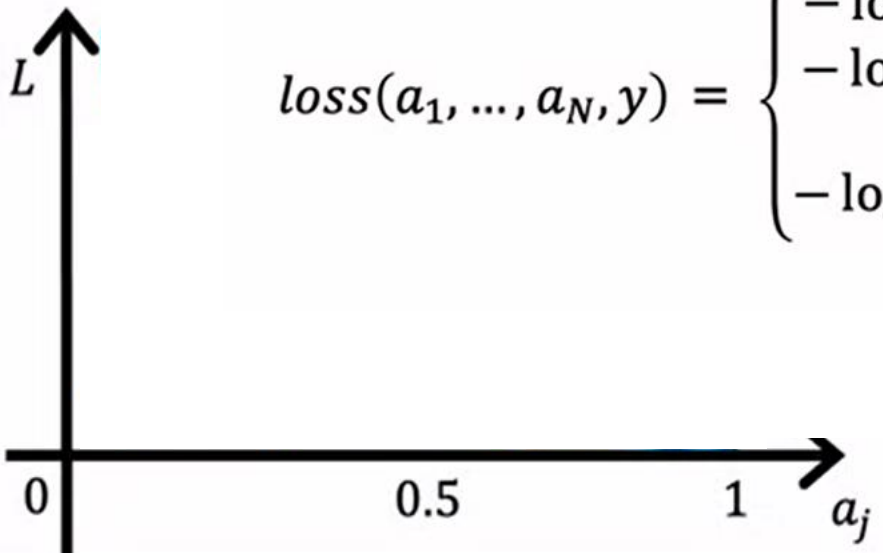
Softmax regression

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

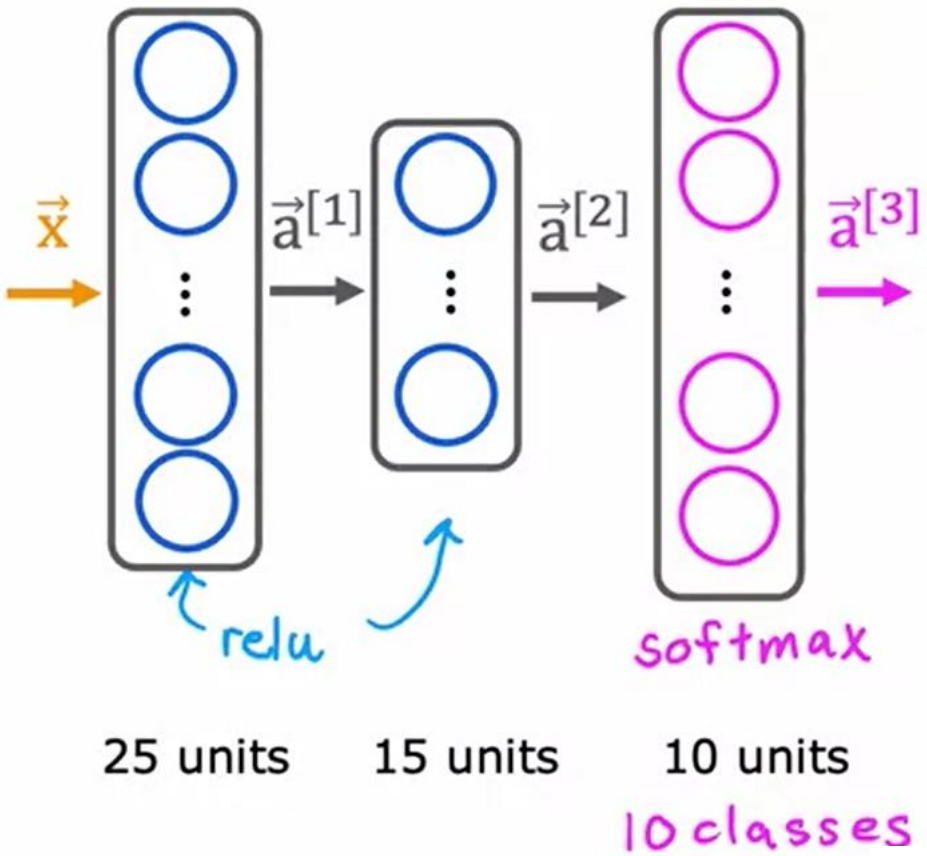
$$\vdots$$

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

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



Neural Network with Softmax output



$$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]} \qquad a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}}$$

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

$$\vdots$$

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

$$= P(y = 10|\vec{x})$$

logistic regression

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

softmax

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

MNIST with softmax

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')
                 ])
```

specify loss and cost

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

```
from tensorflow.keras.losses import
  SparseCategoricalCrossentropy

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

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

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

Note: better (recommended) version later.

Improved implementation of softmax

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')
                 ])
```

specify loss and cost

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

```
from tensorflow.keras.losses import
  SparseCategoricalCrossentropy

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

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

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

Note: better (recommended) version later.

Numerical round off errors

option 1

$$x = \frac{2}{10{,}000}$$

option 2

Numerical round off errors

```
In [1]: x1 = 2.0 / 10000
        print(f"{x1:.18f}") # print 18 digits to the right of decimal point

        0.000200000000000000

In [2]: x2 = 1 + (1/10000) - (1 - 1/10000)
        print(f"{x2: .18f}")

        0.000199999999999978

In [ ]:
```

More numerically accurate implementation of logistic loss:

Logistic regression:

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

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=1, activation='sigmoid')
                    ])
model.compile(loss=BinaryCrossEntropy() )
```

Original loss

$$loss = -y\log(a) - (1-y)\log(1-a)$$

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

More accurate loss (in code)

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

More numerically accurate implementation of softmax

Softmax regression

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

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

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='softmax')
                  ])
```

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

More Accurate

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

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

logistic regression (more numerically accurate)

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)
```

predict

```
f_x = tf.nn.sigmoid(logit)
```

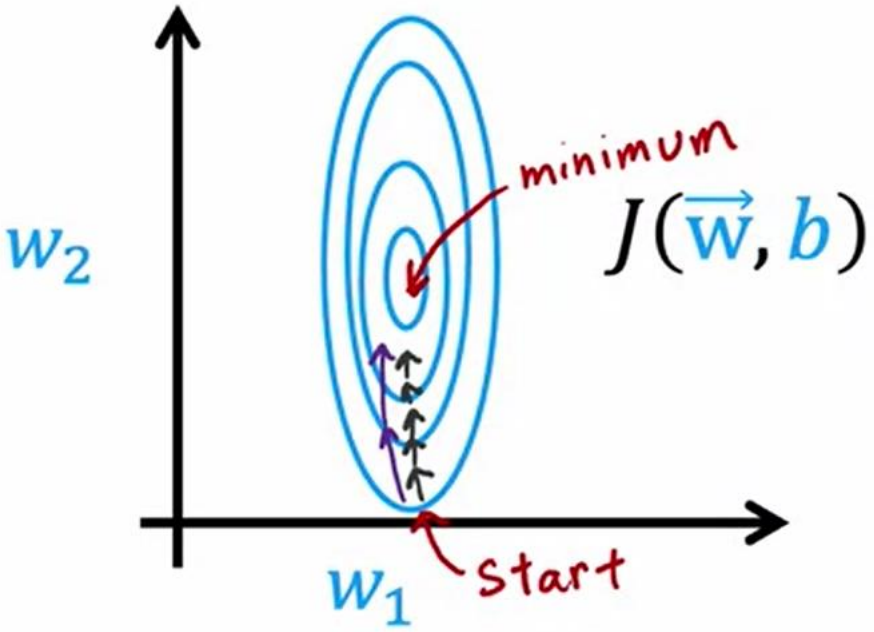
AI CAMP

Advanced Optimization

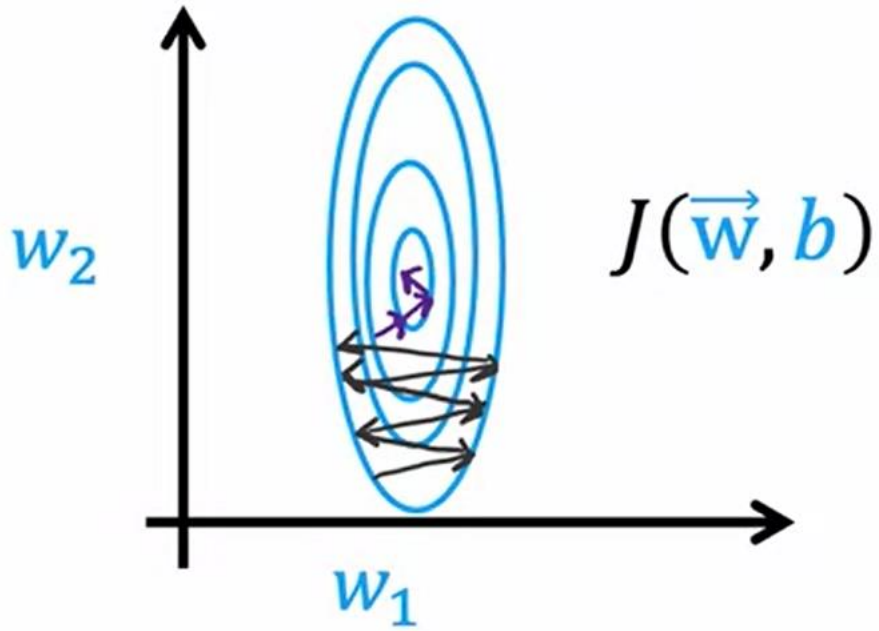
school of ai Algiers

Gradient Descent

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



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



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

Adam: Adaptive Moment estimation

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

$$\vdots$$

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

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

MNIST Adam

model

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

compile

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

fit

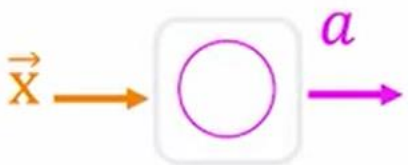
```
model.fit(X,Y,epochs=100)
```

AI CAMP

Back propagation

school of ai Algiers

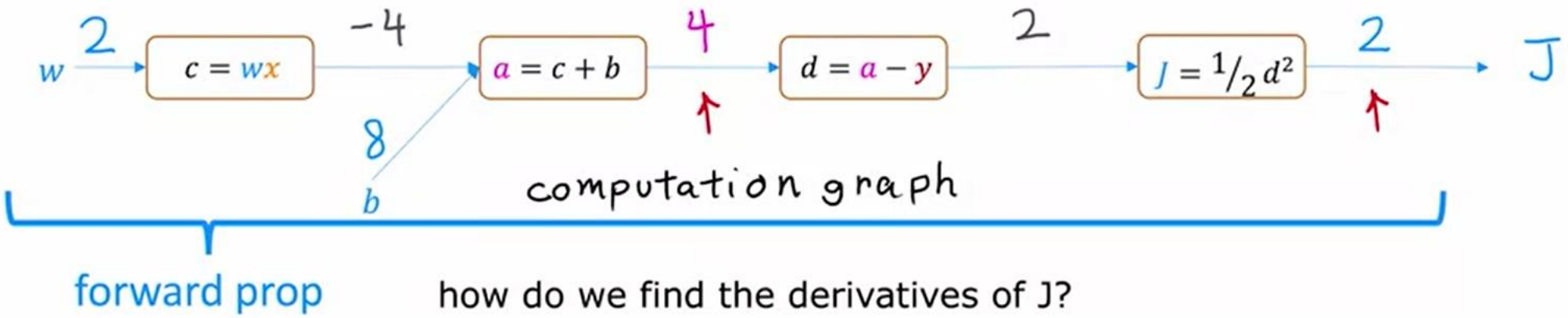
Small Neural Network Example



$w = 2 \quad b = 8 \quad x = -2 \quad y = 2$

$a = wx + b$ linear activation $a = g(z) = z$

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



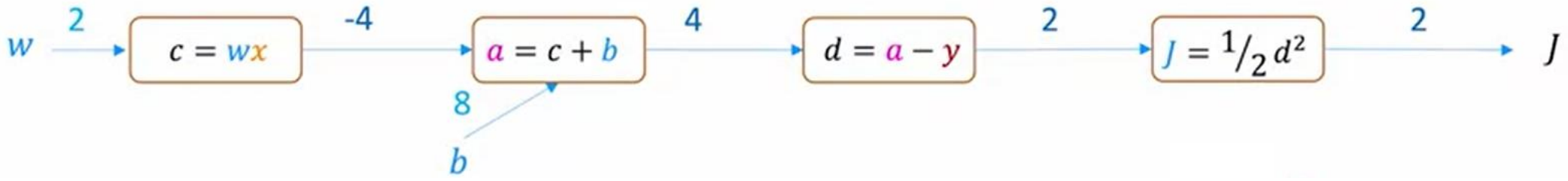
how do we find the derivatives of J?

Computing the Derivatives

Forward prop →

Back prop ←

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

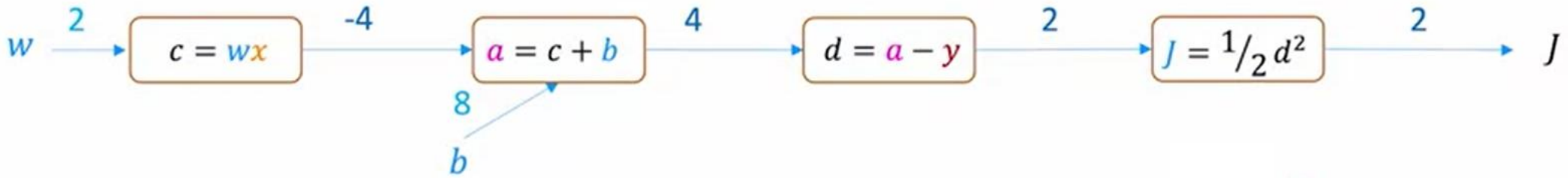


Computing the Derivatives

Forward prop →

Back prop ←

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



Neural Network Example

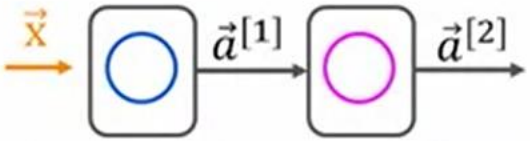
$x = 1 \quad y = 5$

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

ReLU activation

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

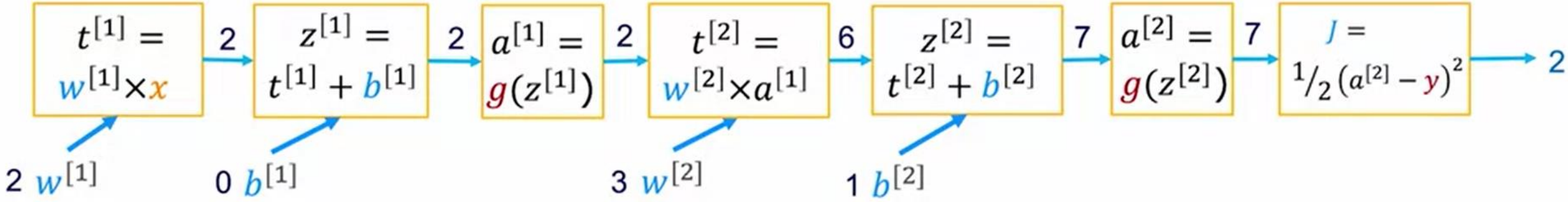
$g(z) = \max(0, z)$



$$a^{[1]} = g(w^{[1]}x + b^{[1]}) = \overbrace{w^{[1]}x + b^{[1]}}^{z^{[1]}} = 2\times1 + 0 = 2$$

$$a^{[2]} = g(w^{[2]}a^{[1]} + b^{[2]}) = \overbrace{w^{[2]}a^{[1]} + b^{[2]}}^{z^{[2]}} = 3\times2 + 1 = 7$$

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



AI CAMP

Thank you !!

Don't forget to solve the quiz and to share your score with us

Quiz link: https://forms.gle/6FSd1ReBnYP2wXpn7

school of ai Algiers