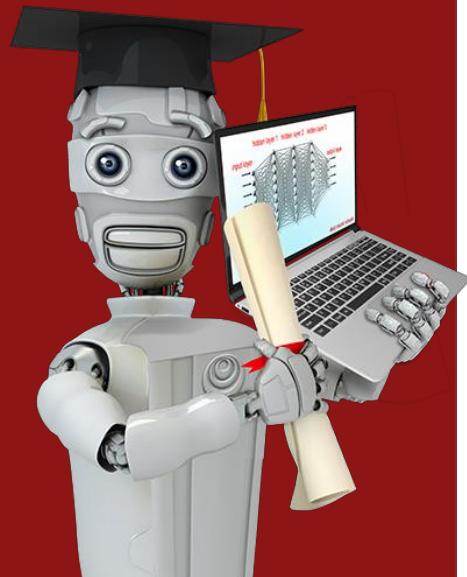


# Copyright Notice

These slides are distributed under the Creative Commons License.

[DeepLearning.AI](#) makes these slides available for educational purposes. You may not use or distribute these slides for commercial purposes. You may make copies of these slides and use or distribute them for educational purposes as long as you cite [DeepLearning.AI](#) as the source of the slides.

For the rest of the details of the license, see <https://creativecommons.org/licenses/by-sa/2.0/legalcode>

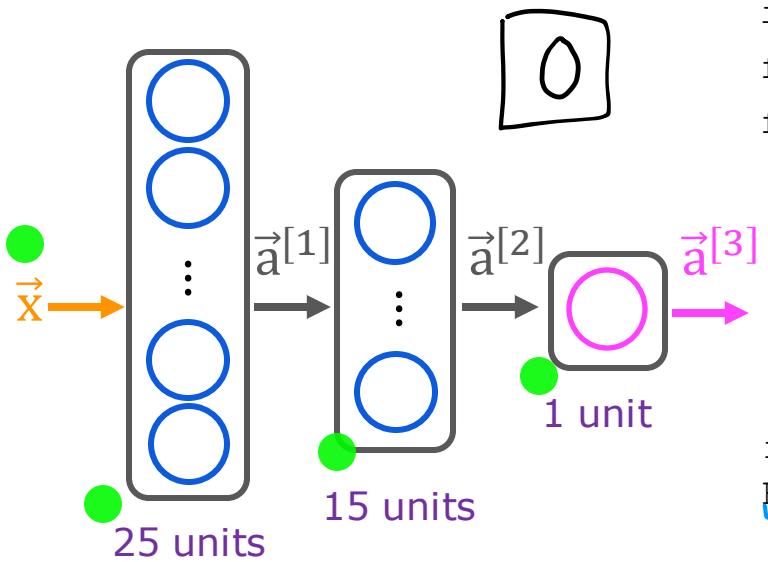


# Neural Network Training

---

TensorFlow  
implementation

# Train a Neural Network in TensorFlow



Given set of  $(x, y)$  examples

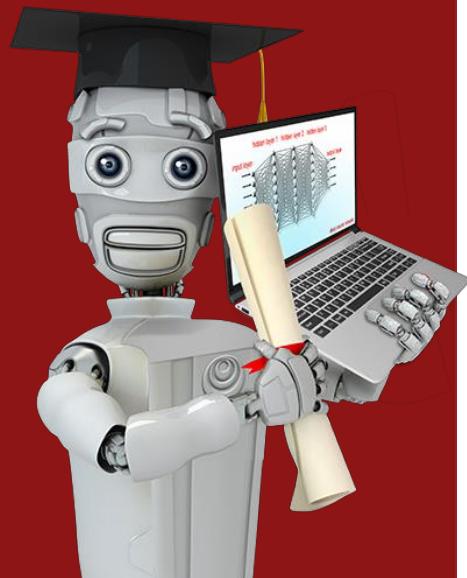
How to build and train this in code?

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

model.compile(loss=BinaryCrossentropy())

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

③  
epochs: number of steps  
in gradient descent



# Neural Network Training

---

## Training Details

# Model Training Steps

TensorFlow

①

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

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

②

specify loss and cost

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

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

③

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

logistic regression

$$\begin{aligned} z &= np.dot(w, x) + b \\ f_x &= 1 / (1 + np.exp(-z)) \end{aligned}$$

logistic loss

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

$$\begin{aligned} w &= w - \alpha * dj_dw \\ b &= b - \alpha * dj_db \end{aligned}$$

neural network

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

binary cross entropy

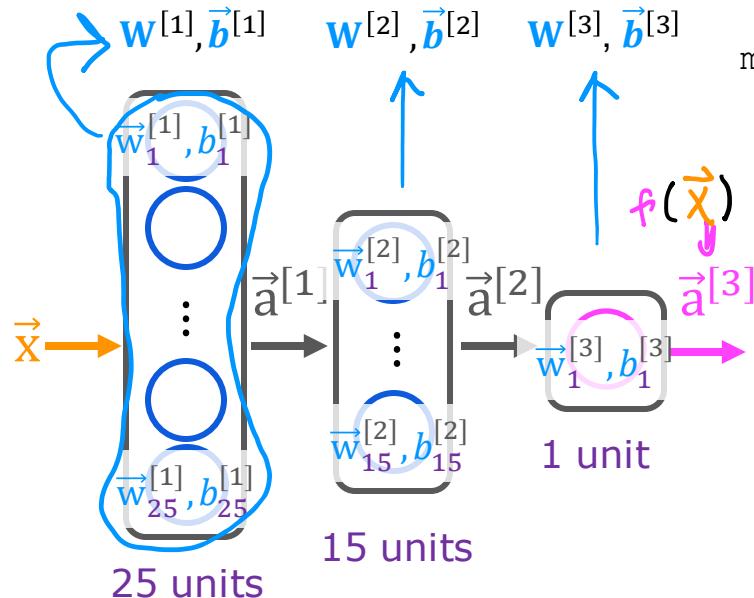
```
model.compile(  
    loss=BinaryCrossentropy())
```

```
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 functions

Mnist digit classification problem

binary classification

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

compare prediction vs. target

logistic loss

also known as binary cross entropy

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

regression

(predicting numbers  
and not categories)

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

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

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

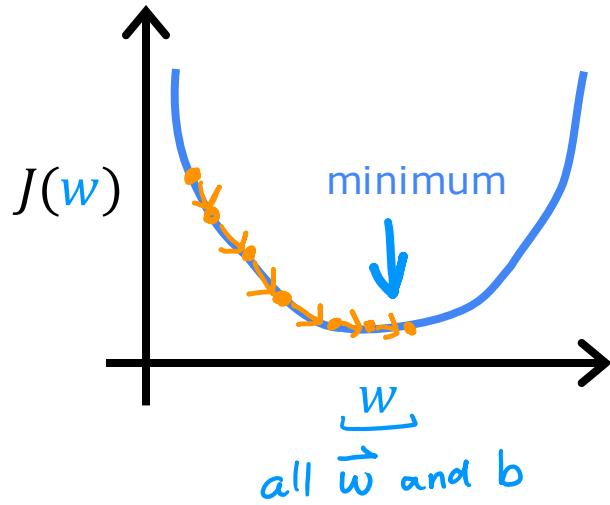
$f_{\mathbf{W}, \mathbf{B}}(\vec{x})$

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



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

# 3. Gradient descent



```
repeat {  
     $w_j^{[l]} = w_j^{[l]} - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$   
     $b_j^{[l]} = b_j^{[l]} - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$   
}  
} Compute derivatives  
for gradient descent  
using "back propagation"
```

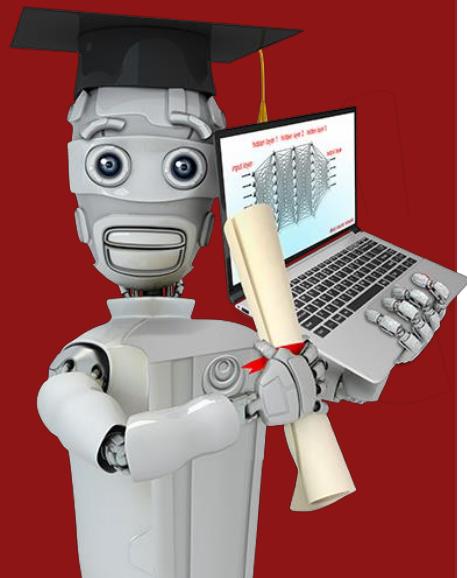
`model.fit(X, y, epochs=100)`

# Neural network libraries

Use code libraries instead of coding "from scratch"



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

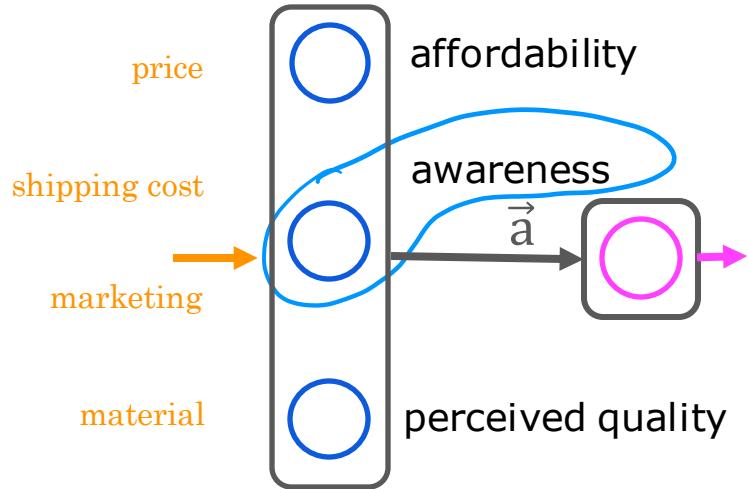


# Activation Functions

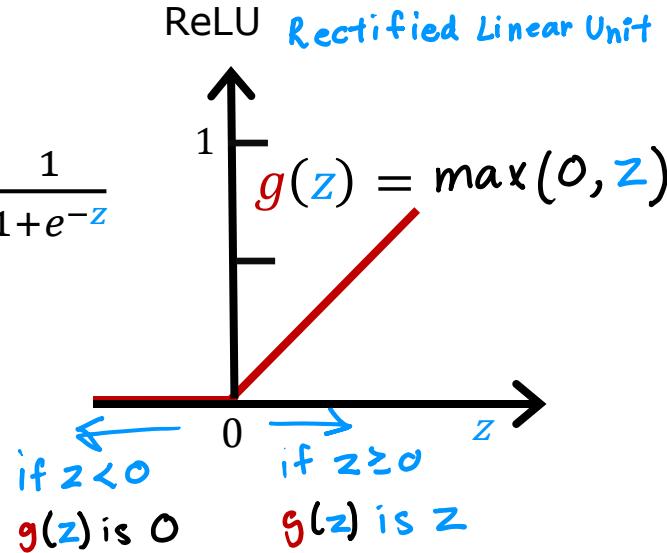
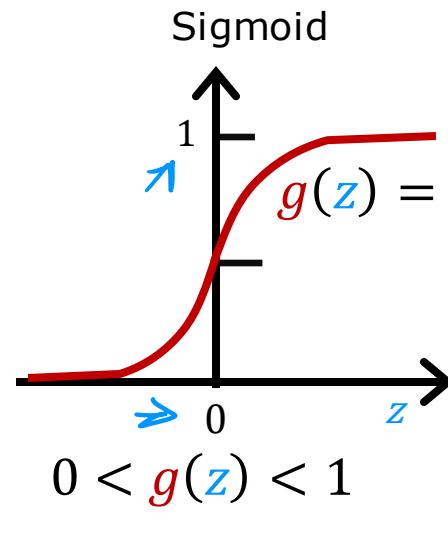
---

Alternatives to the  
sigmoid activation

# Demand Prediction Example



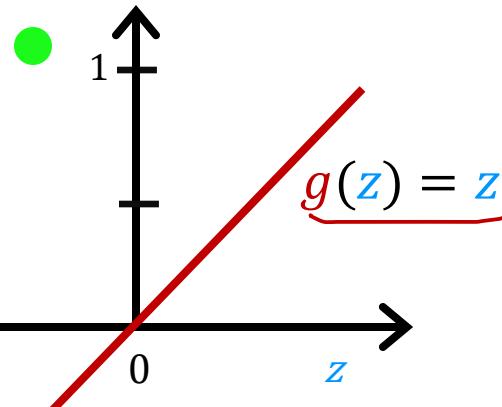
$$a_2^{[1]} = g(\overrightarrow{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]})$$



# Examples of Activation Functions

"No activation function"

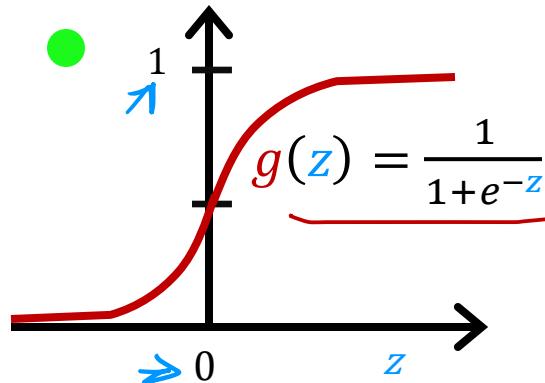
Linear activation function



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

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

Sigmoid

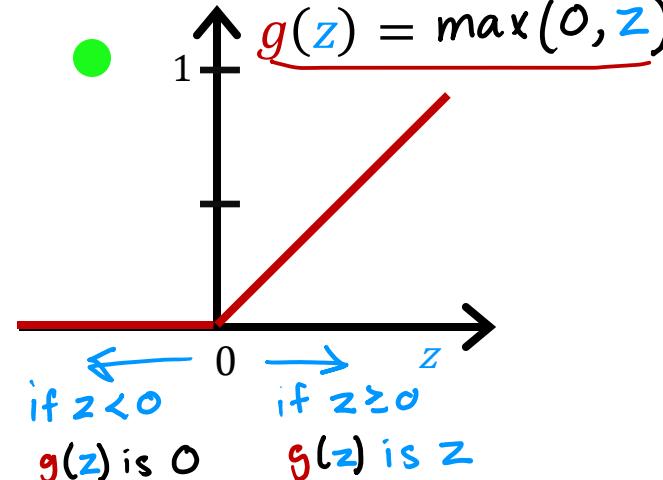


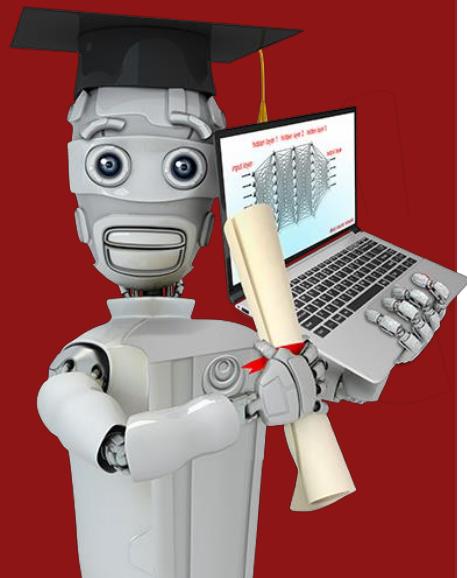
$$0 < g(z) < 1$$

Later: softmax activation

ReLU

Rectified Linear Unit



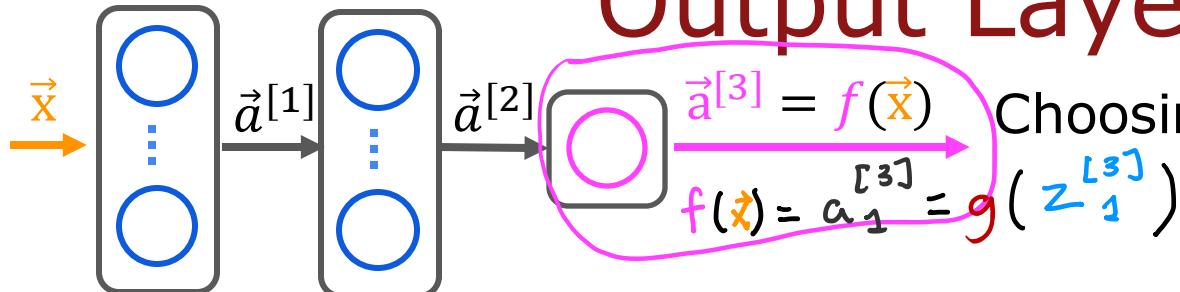


# Activation Functions

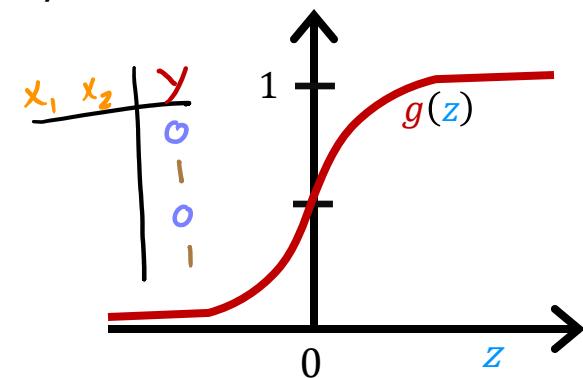
---

## Choosing activation functions

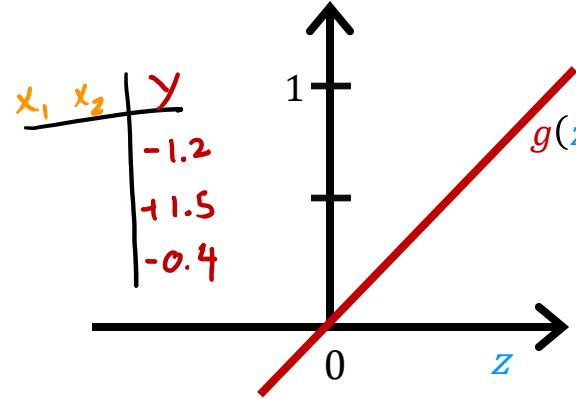
# Output Layer



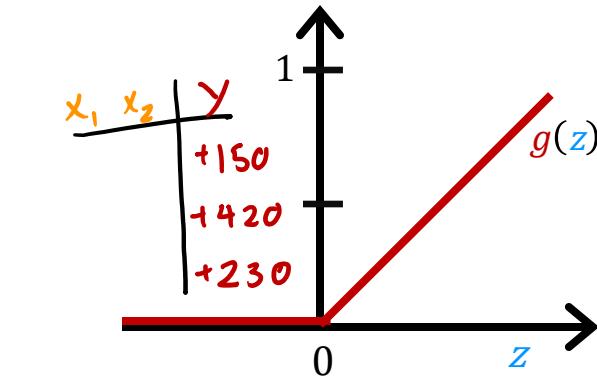
- Binary classification  
Sigmoid  
 $y=0/1$



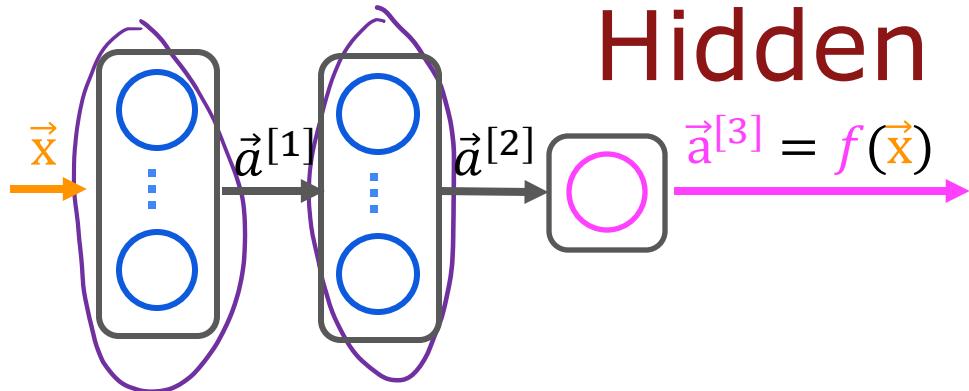
- Regression  
Linear activation function  
 $y = +/-$



- Regression  
ReLU  
 $Y = 0 \text{ or } +$

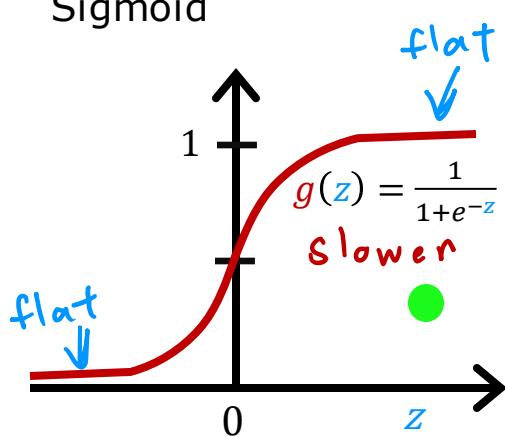


# Hidden Layer

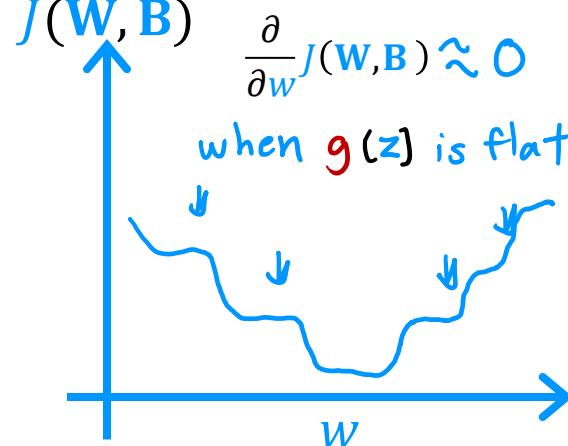


Choosing  $g(z)$  for hidden layer

Sigmoid

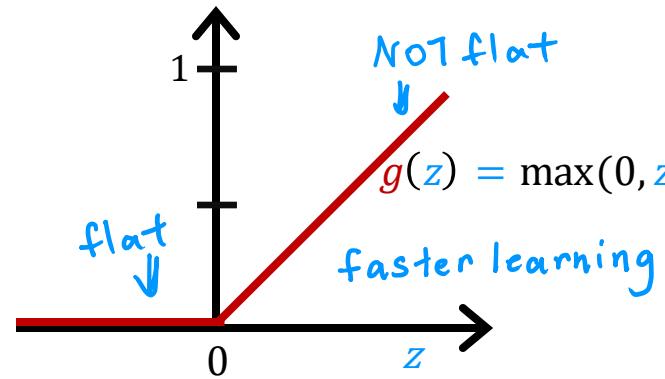


$J(W, B)$

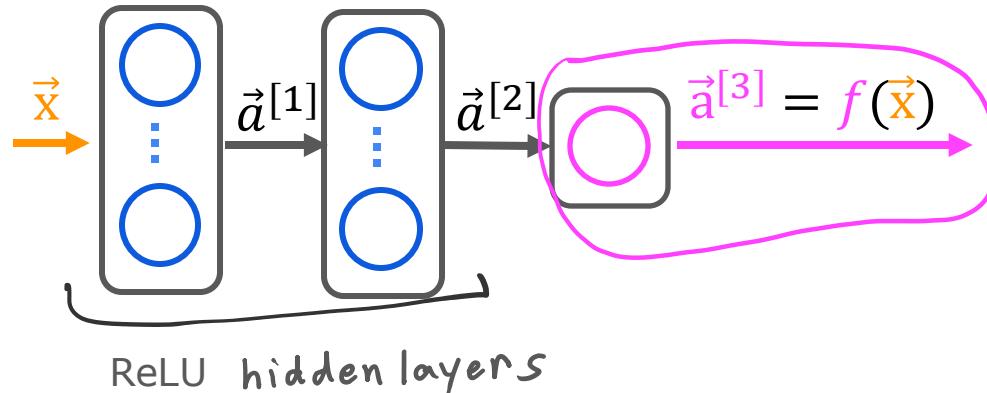


ReLU

faster

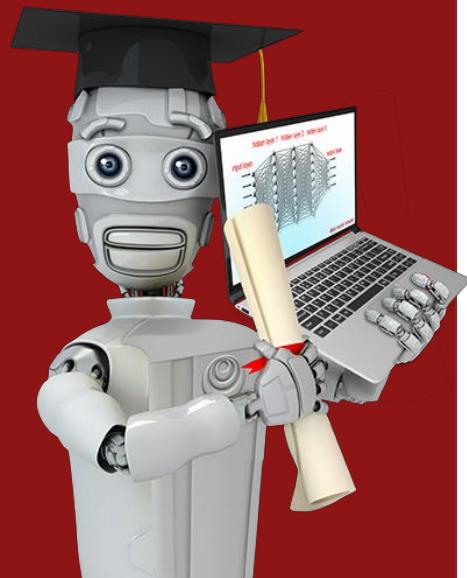


# Choosing Activation Summary



```
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  
])  
or 'linear'  
or 'relu'
```

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

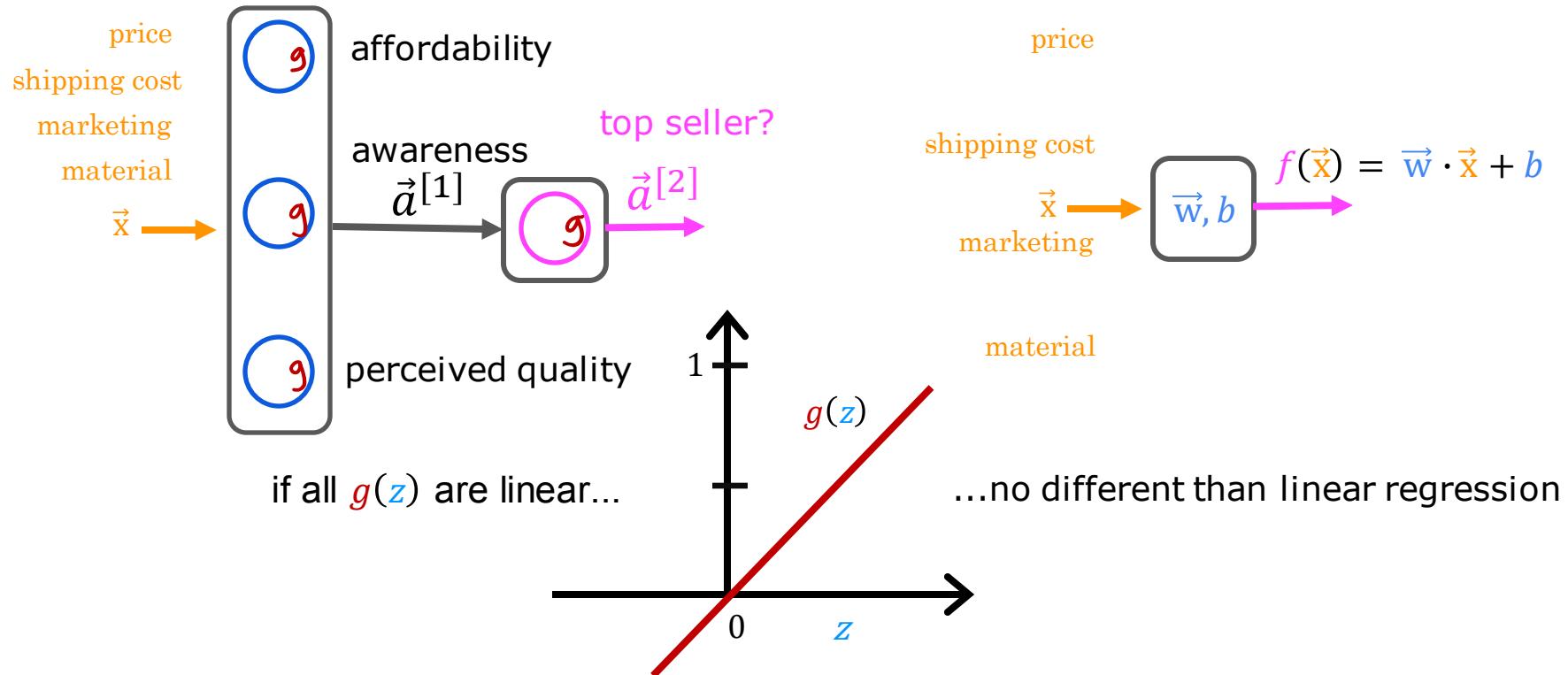


# Activation Functions

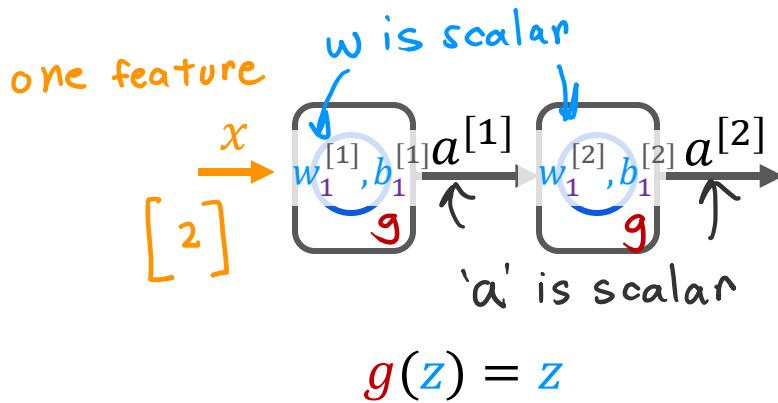
---

Why do we need activation functions?

# Why do we need activation functions?



# Linear Example

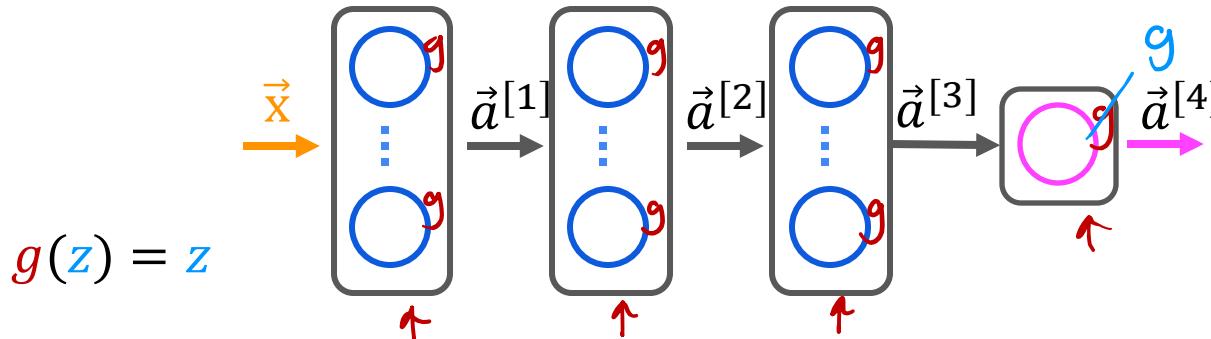


$$\begin{aligned}a^{[1]} &= \underbrace{w_1^{[1]} x}_{\downarrow} + b_1^{[1]} \\a^{[2]} &= w_1^{[2]} a^{[1]} + b_1^{[2]} \\&= w_1^{[2]} (w_1^{[1]} x + b_1^{[1]}) + b_1^{[2]} \\\vec{a}^{[2]} &= (\underbrace{\vec{w}_1^{[2]} \vec{w}_1^{[1]}}_{\omega}) x + \underbrace{w_1^{[2]} b_1^{[1]}}_{b} + b_1^{[2]}\end{aligned}$$

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

$$f(x) = w x + b \text{ linear regression}$$

# Example



$$g(z) = z$$

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

all linear (including output)  
↳ equivalent to linear regression

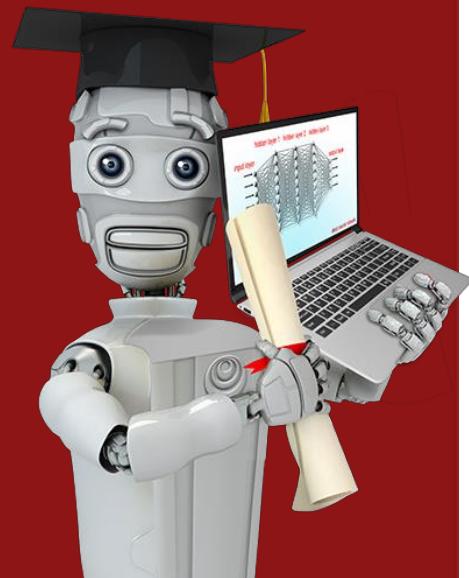
$$\vec{a}^{[4]} = \frac{1}{1+e^{-(\vec{w}_1^{[4]}\cdot\vec{a}^{[3]}+b_1^{[4]})}}$$

output activation is sigmoid  
(hidden layers still linear)  
↳ equivalent to logistic regression

Don't use linear activations in hidden layers (use ReLU)

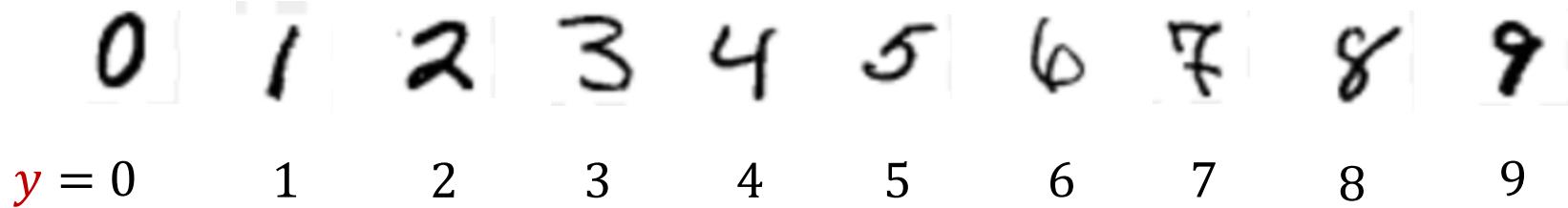
# Multiclass Classification

---



## Multiclass

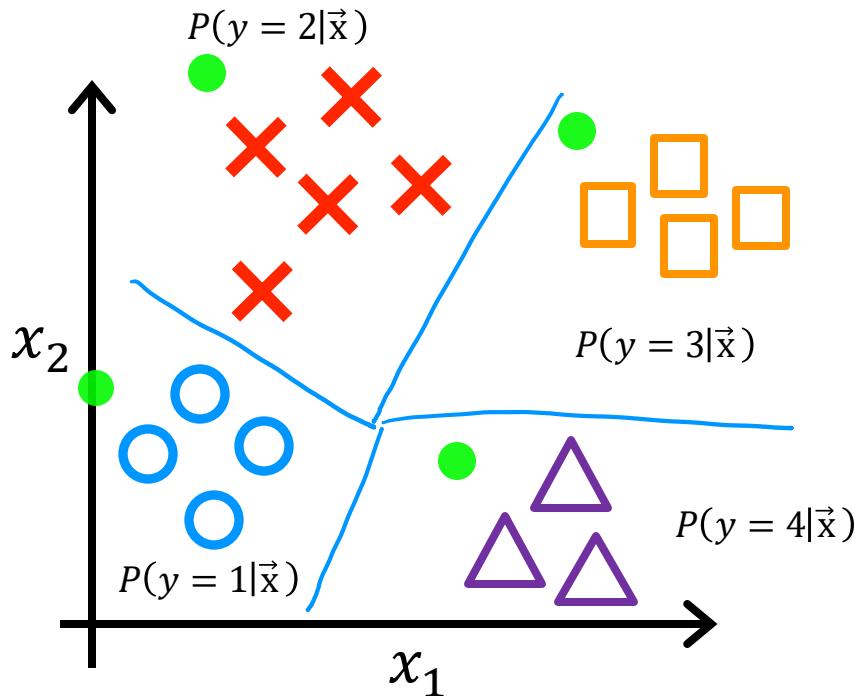
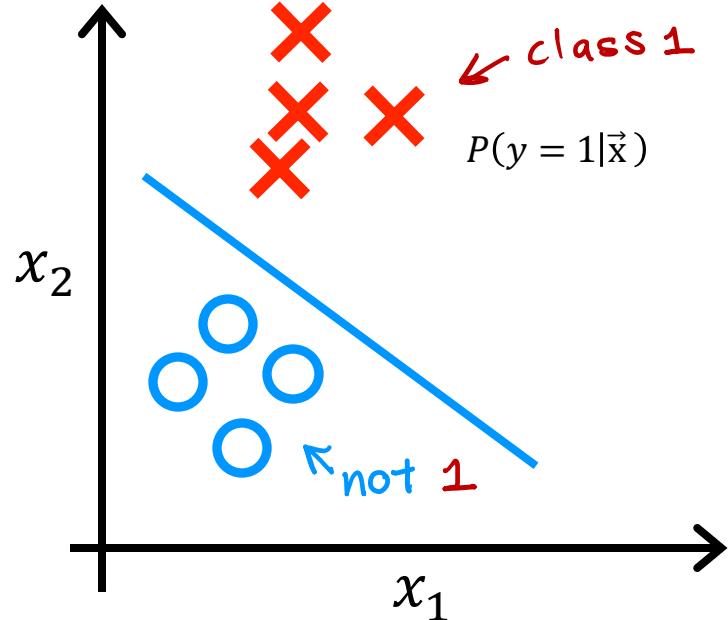
# MNIST example



$x \curvearrowright 7 \quad y = 7$

multiclass classification problem:  
target  $y$  can take on more than two possible values

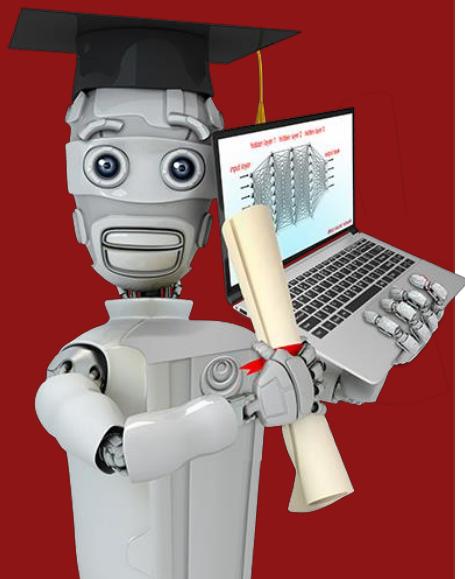
# Multiclass classification example



# Multiclass Classification

---

## Softmax



## Logistic regression (2 possible output values)

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

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

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

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

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

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

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

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

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

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

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

**✗** **○** **□** **△**  
 $= P(y=1|\vec{x})$  **0.30**

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

$$a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$
 $= P(y=2|\vec{x})$  **0.20**

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

$$a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$
 $= P(y=3|\vec{x})$  **0.15**

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

$$a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$
 $= P(y=4|\vec{x})$  **0.35**

# Cost

## Logistic regression

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

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

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

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

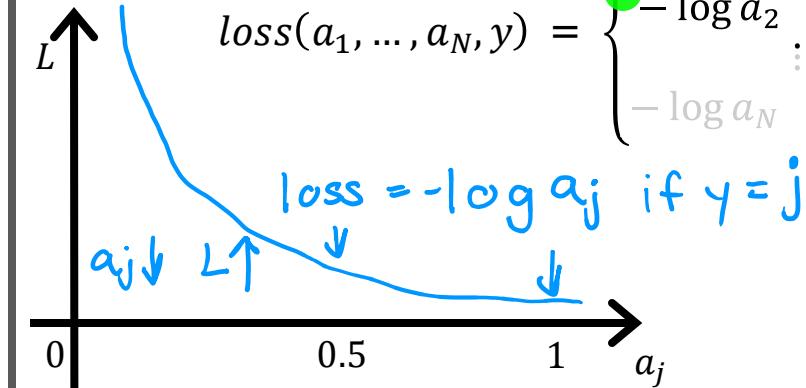
$$J(\vec{w}, b) = \text{average loss}$$

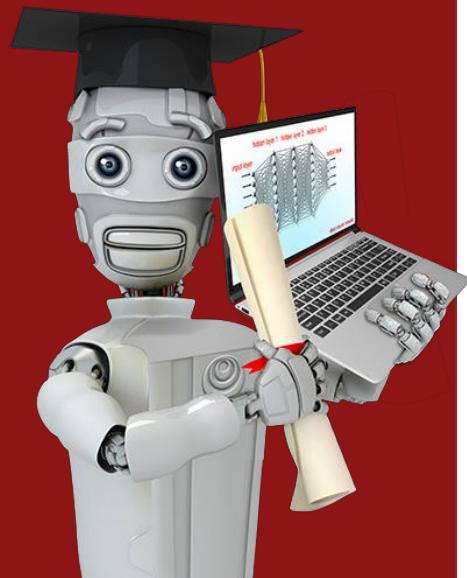
## Softmax regression

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

### Crossentropy loss

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



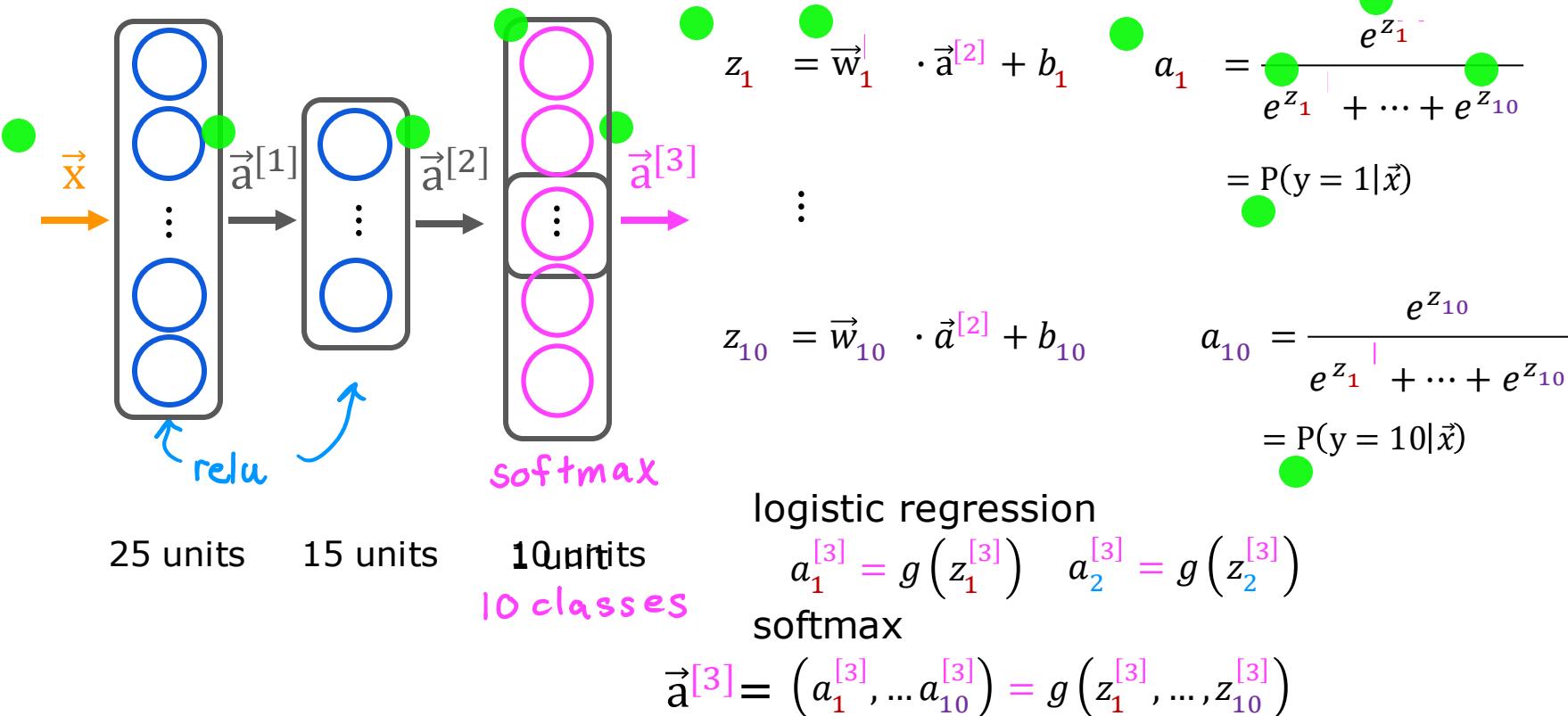


# Multiclass Classification

---

Neural Network with  
Softmax output

# Neural Network with Softmax output



# 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

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

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

③

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

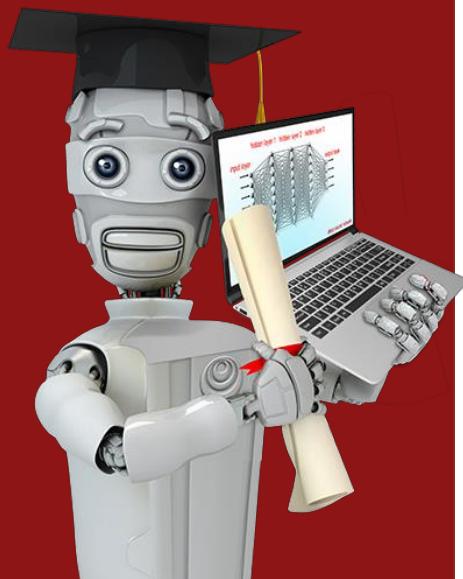
Note: better (recommended) version later.

*Don't use the version shown here!*

# Multiclass Classification

---

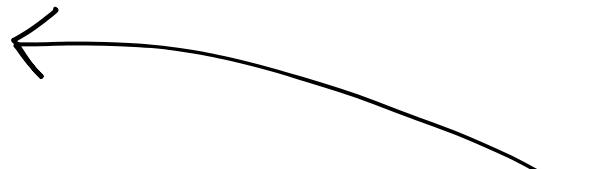
Improved implementation  
of softmax



# Numerical Roundoff Errors

option 1

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



option 2

$$x = \underbrace{\left(1 + \frac{1}{10,000}\right)} - \underbrace{\left(1 - \frac{1}{10,000}\right)} =$$

# Numerical Roundoff Errors

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=10, activation='sigmoid')
])
```

~~model.compile(loss=BinaryCrossEntropy())~~

Original loss

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

~~model.compile(loss=BinaryCrossEntropy(from\_logits=True))~~

More accurate loss (in code)

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

logit: z

# More numerically accurate implementation of softmax

## Softmax regression

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

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

More Accurate

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

```
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=SparseCrossEntropy(from\_logits=True))

# MNIST (more numerically accurate)

model

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

loss

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

fit

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

predict

```
logits = model(X) ← not  $a_1 \dots a_{10}$ 
f_x = tf.nn.softmax(logits) is  $z_1 \dots z_{10}$ 
```

# 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

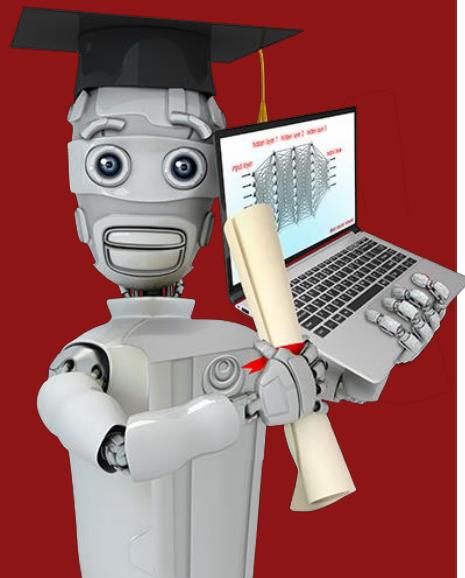
```
logit = model(X)
```

$z$

predict

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





# Multi-label Classification

---

Classification with  
multiple outputs  
(Optional)

# Multi-label Classification



Is there a car?

$$\begin{array}{ll} \text{yes} & y = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

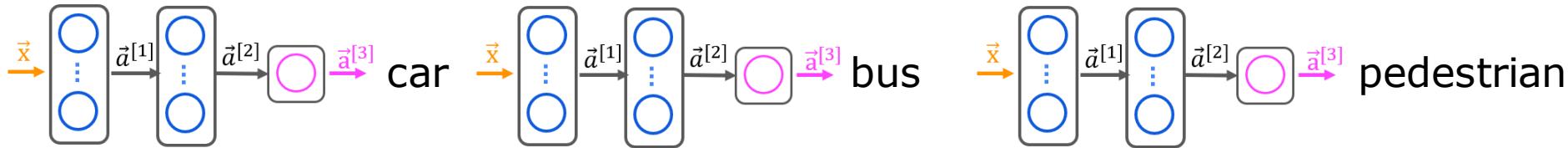
Is there a bus?

$$\begin{array}{ll} \text{no} & y = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

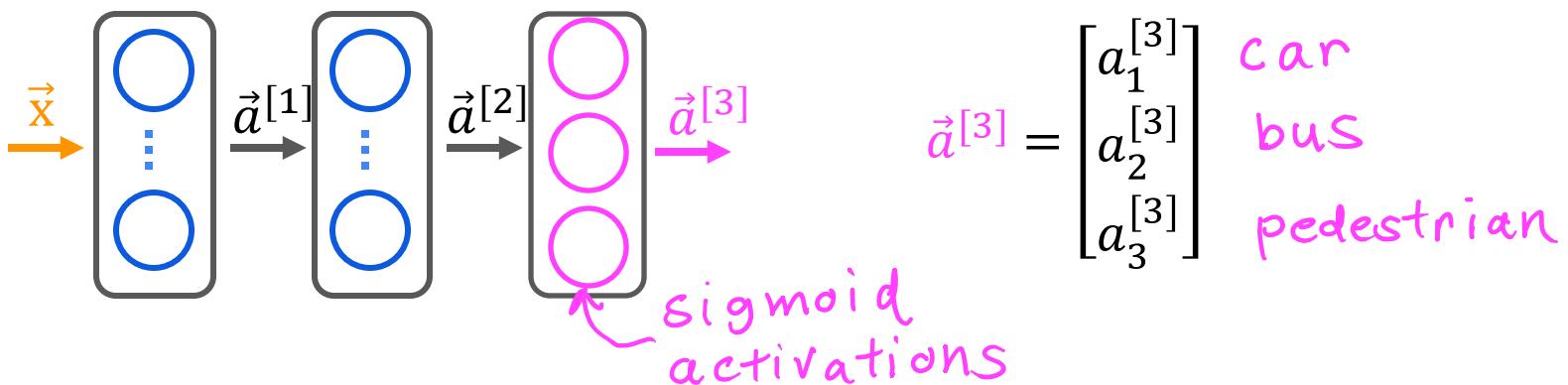
Is there a pedestrian?

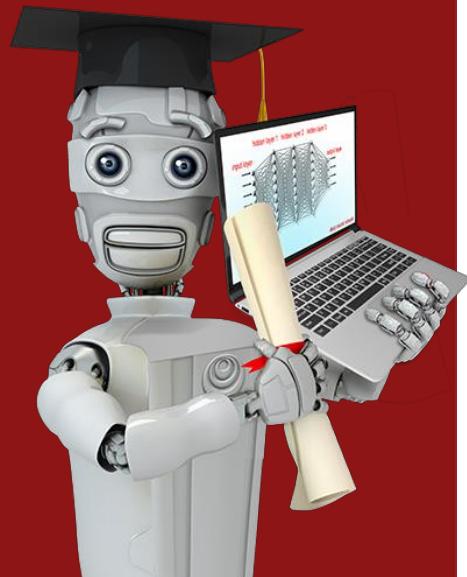
$$\begin{array}{ll} \text{yes} & y = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \end{array}$$

# Multiple classes



Alternatively, train one neural network with three outputs





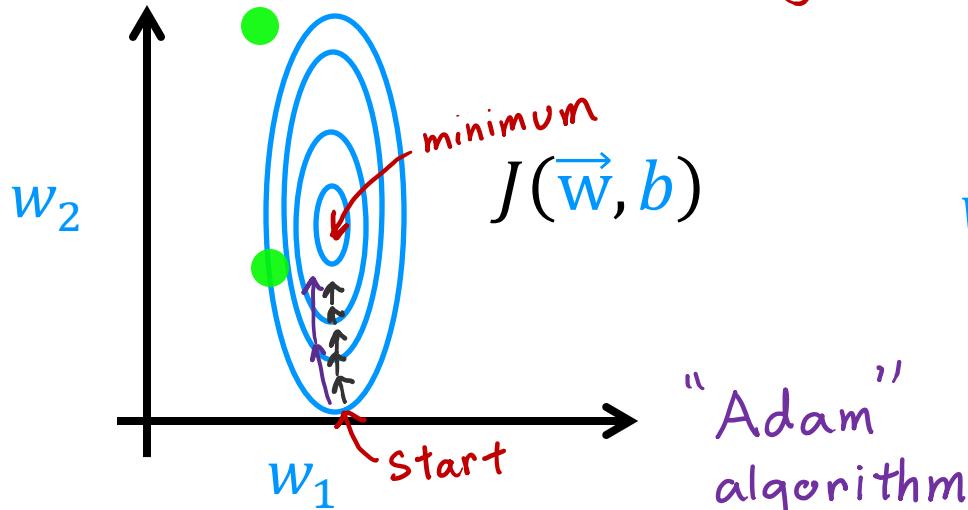
## Additional Neural Network Concepts

# Advanced Optimization

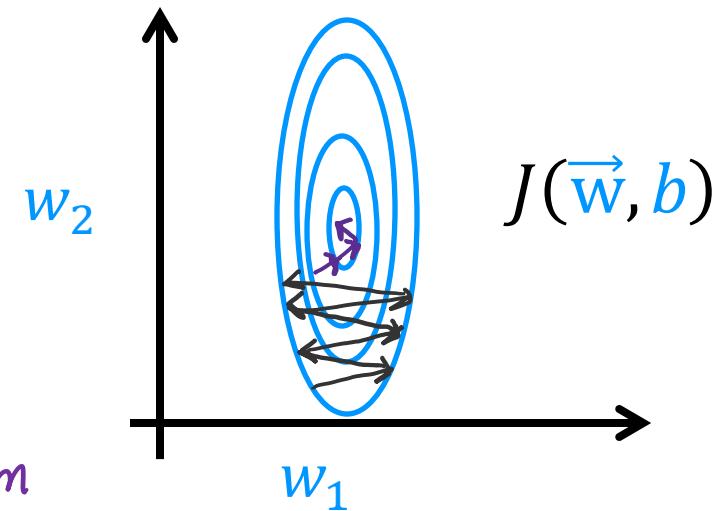
# Gradient Descent

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

↑  
learning rate



Go faster – increase  $\alpha$



Go slower – decrease  $\alpha$

# Adam Algorithm Intuition

Adam: Adaptive Moment estimation    *not just one  $\alpha$*

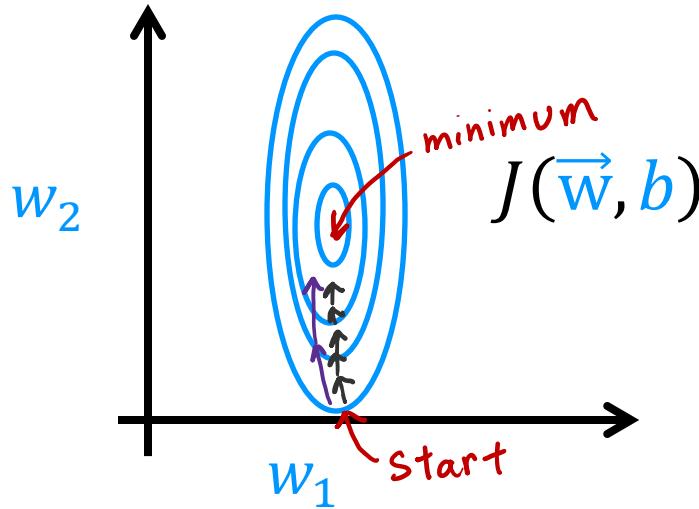
$$w_1 = w_1 - \underbrace{\alpha_1}_{\text{red}} \frac{\partial}{\partial w_1} J(\vec{w}, b)$$

⋮

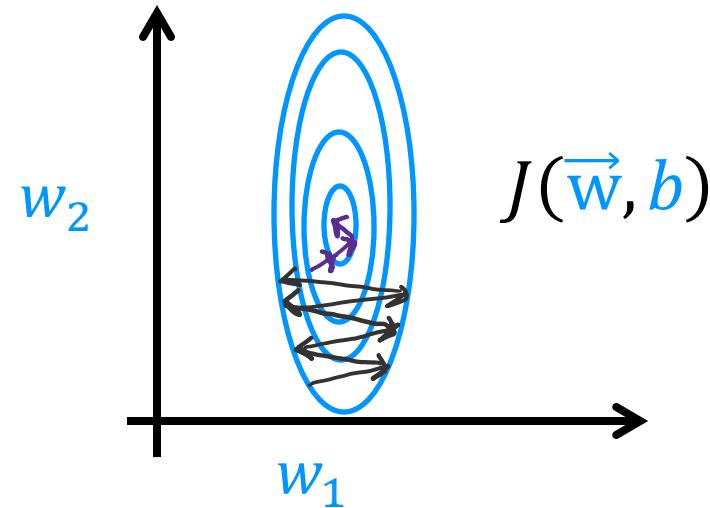
$$w_{10} = w_{10} - \underbrace{\alpha_{10}}_{\text{red}} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$

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

# Adam Algorithm Intuition



If  $w_j$  (or  $b$ ) keeps moving in same direction, increase  $\alpha_j$ .



If  $w_j$  (or  $b$ ) keeps oscillating, reduce  $\alpha_j$ .

# 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')
])
```

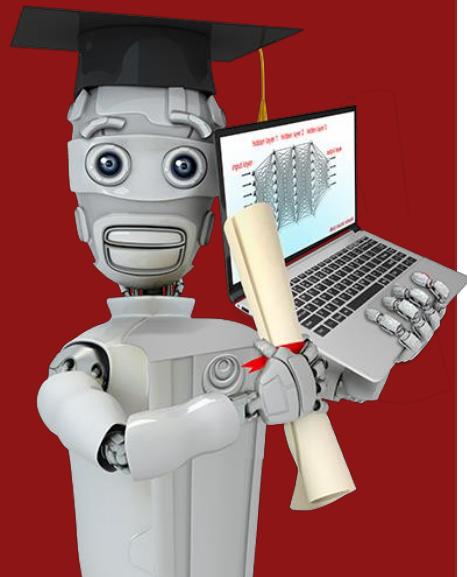
compile

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

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

fit

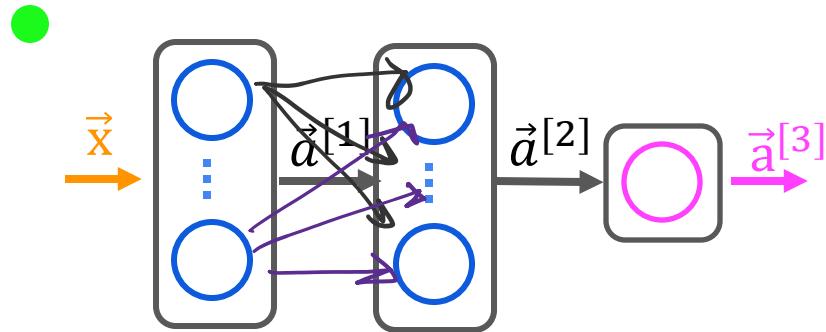
```
model.fit(X, Y, epochs=100)
```



## Additional Neural Network Concepts

### Additional Layer Types

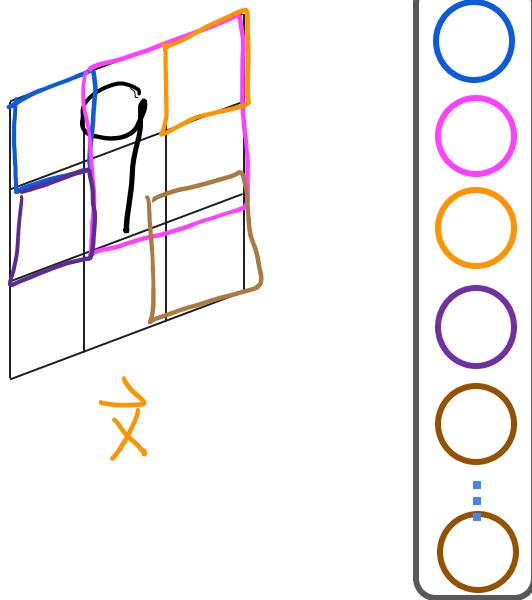
# Dense Layer



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

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

# Convolutional Layer



Each Neuron only looks at part of the previous layer's inputs.

Why?

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

# Convolutional Neural Network

