

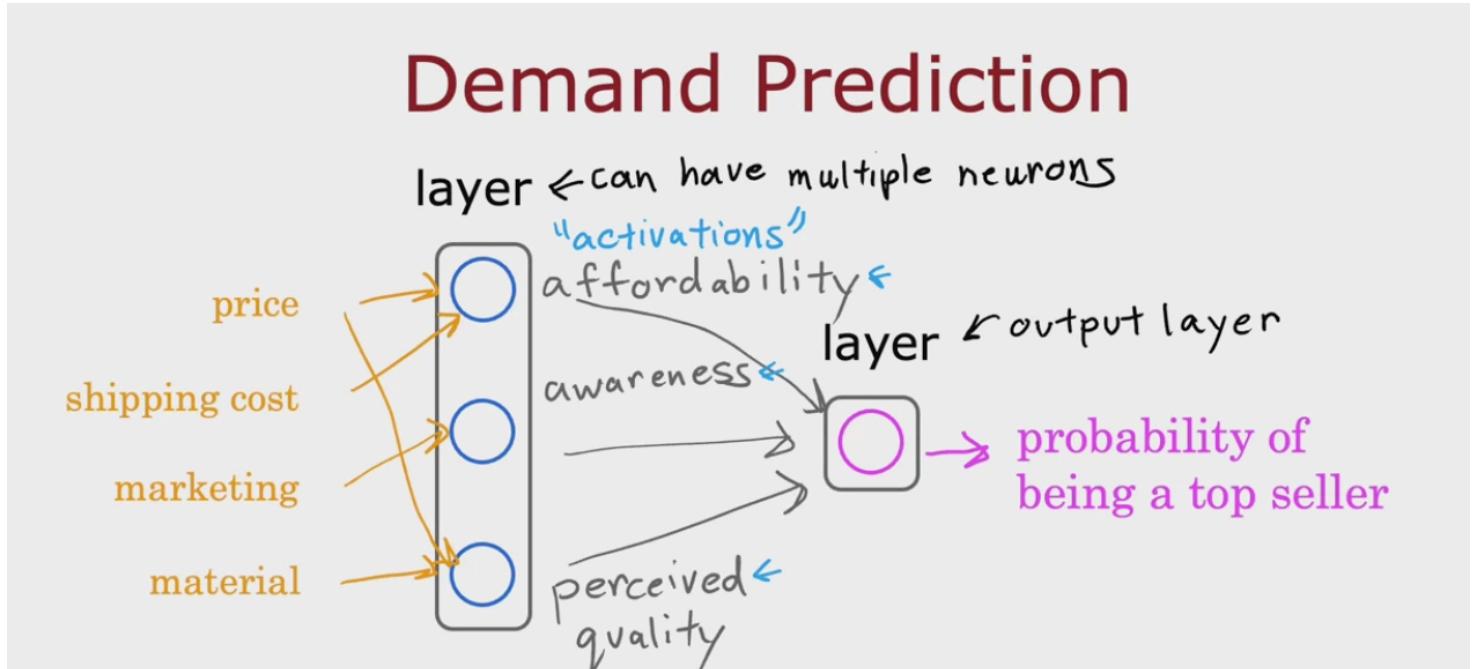
# Advanced Learning Algorithms - Week 1&2

Neural networks: Mimic brain. Speech recognition - images - NLP

Neuron

some # of inputs, a number as output

Traditional AI, not capable of handling this much amount of data - Everything is digital now.  
so neural networks.



it just looks like logistic regression, but it does not use ready features - price, shipping cost... - but it creates other features, maybe better ones.

feature engineering

How many hidden layers? How many neurons in which?

Face recognition - 1000x1000 pixels - can there be a model of neural network that takes a vector with 1m parameters.

what each neuron outputs:

$$a[j] = g(w.a[j-1] + b)$$

Handwritten digit recognition:

forward computation: run something over neural network, running it.

How TensorFlow represents data?:

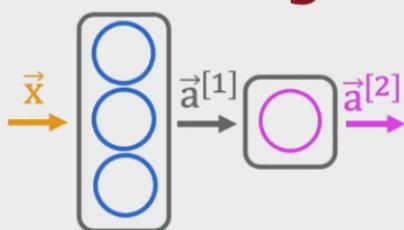
numpy array 2x3 (2 rows, 3 columns):

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

tf.Tensor( [ [ ] ] ) -> we can say this is a matrix. (More general than matrix)

How to deploy on tf:

## Building a neural network architecture



```
→ layer_1 = Dense(units=3, activation="sigmoid") ←  
→ layer_2 = Dense(units=1, activation="sigmoid") ←  
→ model = Sequential([layer_1, layer_2])
```

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

targets

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

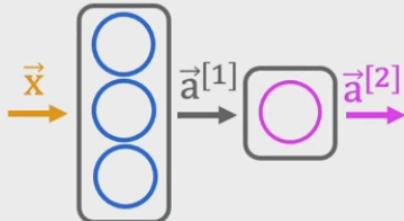
```
y = np.array([1,0,0,1])
```

```
model.compile(...)
```

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

← more about this next week!

## Building a neural network architecture



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

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

targets

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

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

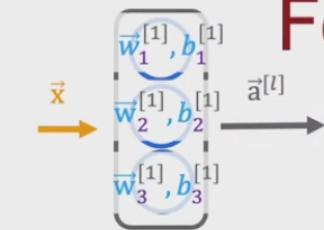
```
model.compile(...)
```

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

← more about this next week!

```
→ model.predict(x_new) ←
```

# Forward prop in NumPy



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

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

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

```
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] [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):
    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
```

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

capital W refers to a matrix

units :  $w.shape[1] = \#$  of columns

$a_{out} : [0,0,0]$

Vectorization:

## 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]]) 2D array  
W = np.array([[1, -3, 5], same  
[-2, 4, -6]])  
B = np.array([-1, 1, 2]) 1x3 2D array

vectorized  
def dense(A\_in, W, B):  
 Z = np.matmul(A\_in, W) + B  
 A\_out = g(Z) matrix multiplication  
 return A\_out

Dot product:

$1 \times 2 \cdot 2 \times 1 = 1 \times 1$

Transpose:  $2 \times 1 \rightarrow 1 \times 2$

# Vector matrix multiplication

$$\vec{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\vec{a}^T = [1 \ 2] \quad w = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix}$$

$$z = \vec{a}^T w \quad [ \leftarrow \vec{a}^T \rightarrow ] \quad \begin{bmatrix} \overset{\uparrow}{w_1} & \overset{\uparrow}{w_2} \\ \downarrow & \downarrow \end{bmatrix}$$

1 by 2

$$z = [\vec{a}^T \vec{w}_1 \quad \vec{a}^T \vec{w}_2]$$

$$(1 * 3) + (2 * 4) \\ 3 + 8 \\ 11$$

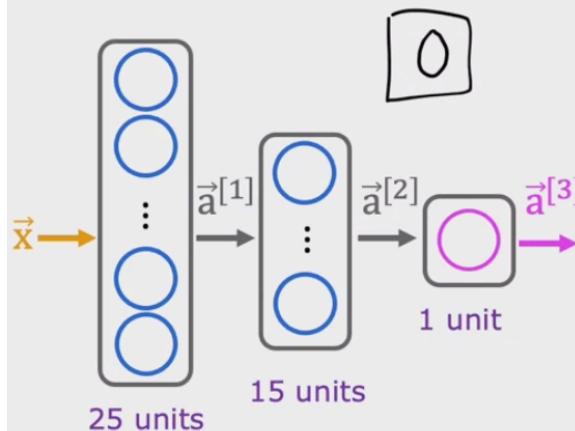
$$(1 * 5) + (2 * 6) \\ 5 + 12 \\ 17$$

$$z = [11 \ 17]$$

BinaryCrossEntropy() loss function

How to train a model on tf:

## 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
```

①

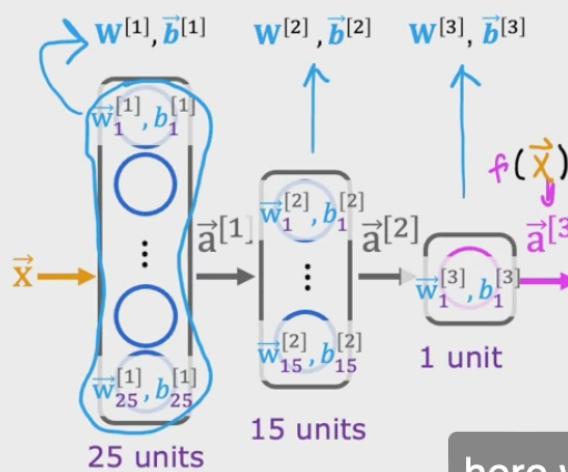
②

③

# 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'),
])
```

here we have written w and b

when we initialize the model, we are not giving exact values to w and b, but we are also initialize them so that we can train the model and the model will decide itself on those values.

## 2. Loss and cost functions

handwritten 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())

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

$\mathbf{w}^{[1]}, \mathbf{w}^{[2]}, \mathbf{w}^{[3]}$

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

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

from tensorflow.keras.losses import  
BinaryCrossentropy

Keras

from tensorflow.keras.losses import  
MeanSquaredError

That's the loss function  
and the cost function.

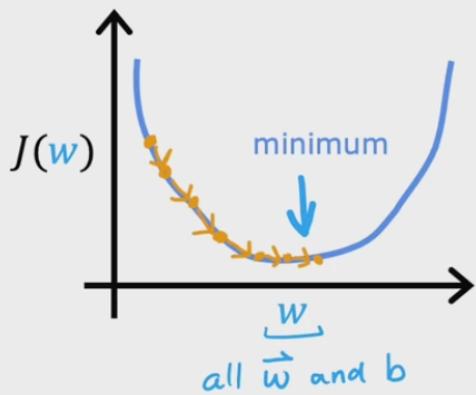
Andrew Ng

Loss func: how distant is the model's prediction from the actual values.

Cost function = average error over all the training examples.

- **Binary Cross-Entropy (or Logistic Loss)** is used in binary classification problems (i.e., problems where there are only two possible outcomes like 0 or 1, yes or no). It works well for classification problems because it compares the probability that the model assigns to the correct class.
- **Mean Squared Error (MSE)** is commonly used for **regression** tasks, where the goal is to predict a continuous number (e.g., predicting house prices). It calculates the squared difference between the predicted value and the actual value.

### 3. Gradient descent



repeat {

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

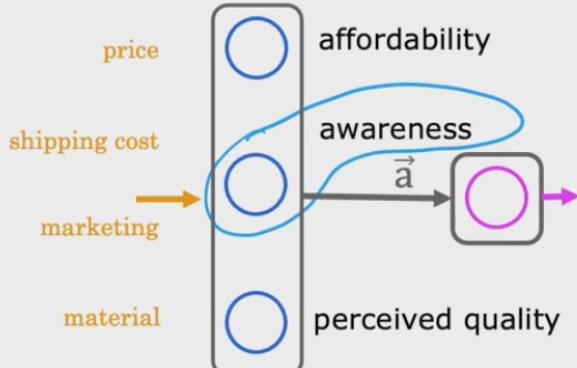
$$b_j^{[l]} = b_j^{[l]} - \alpha \frac{\partial}{\partial b_j} J(\vec{w}, b)$$

} Compute derivatives  
for gradient descent  
using "backpropagation"

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

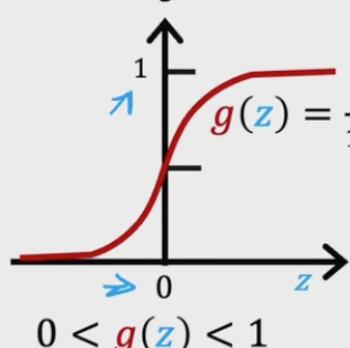
Activation Functions:

### Demand Prediction Example

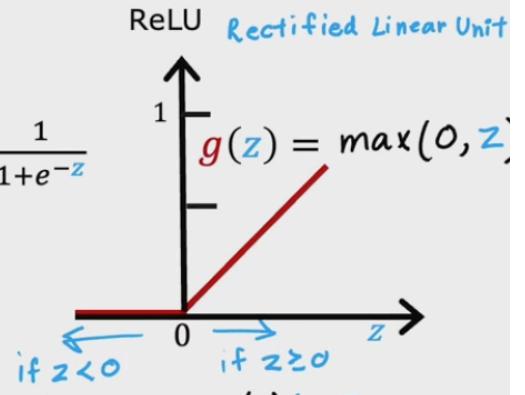


$$a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]})$$

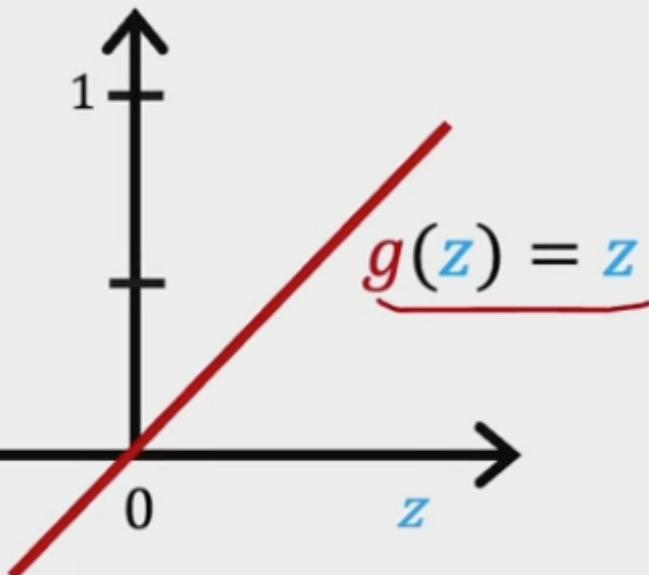
Sigmoid



ReLU



# Linear activation function



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

( -same as- not using any activation function)

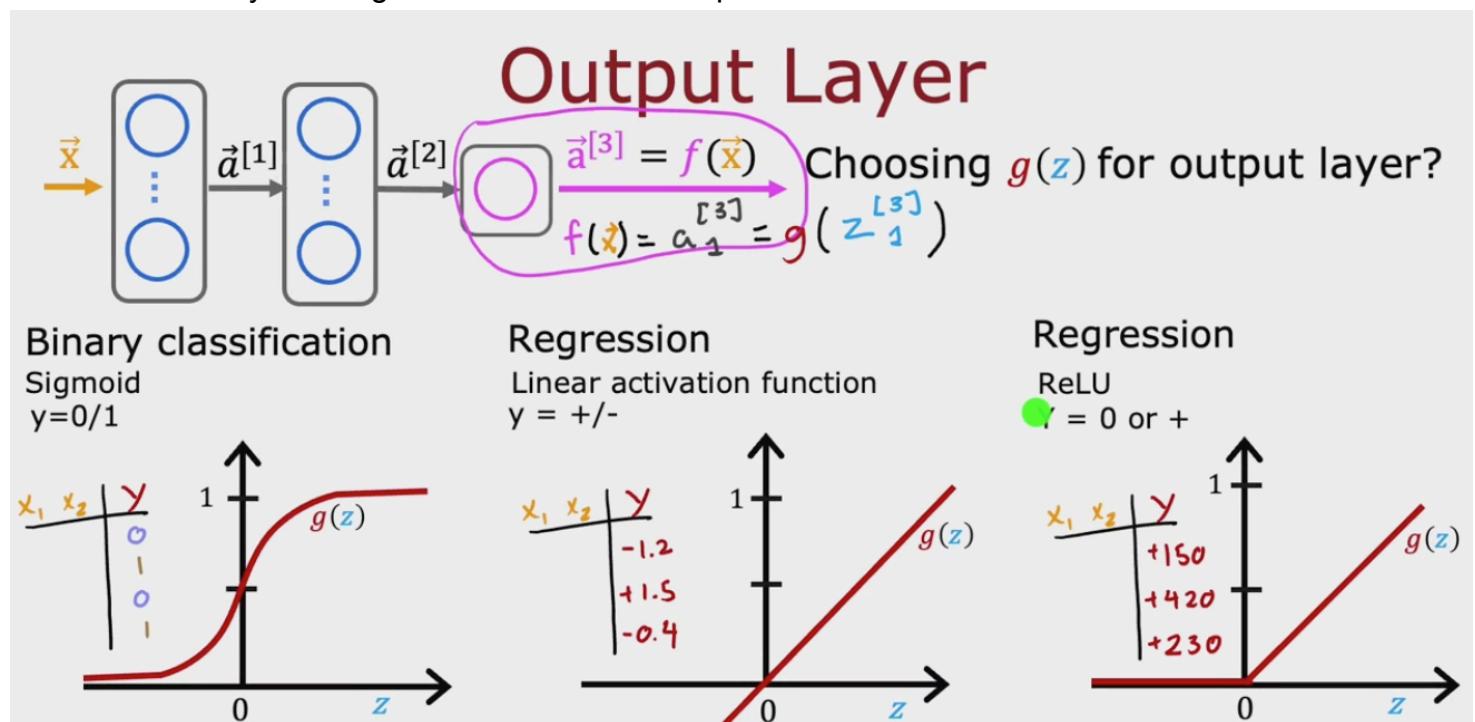
Choosing the activation function:

## Output Layer

Binary Classification: Sigmoid - nearly always a natural choice

Regression: Linear activation function

Cases where only non negative values can be outputs: ReLU



## Hidden Layers

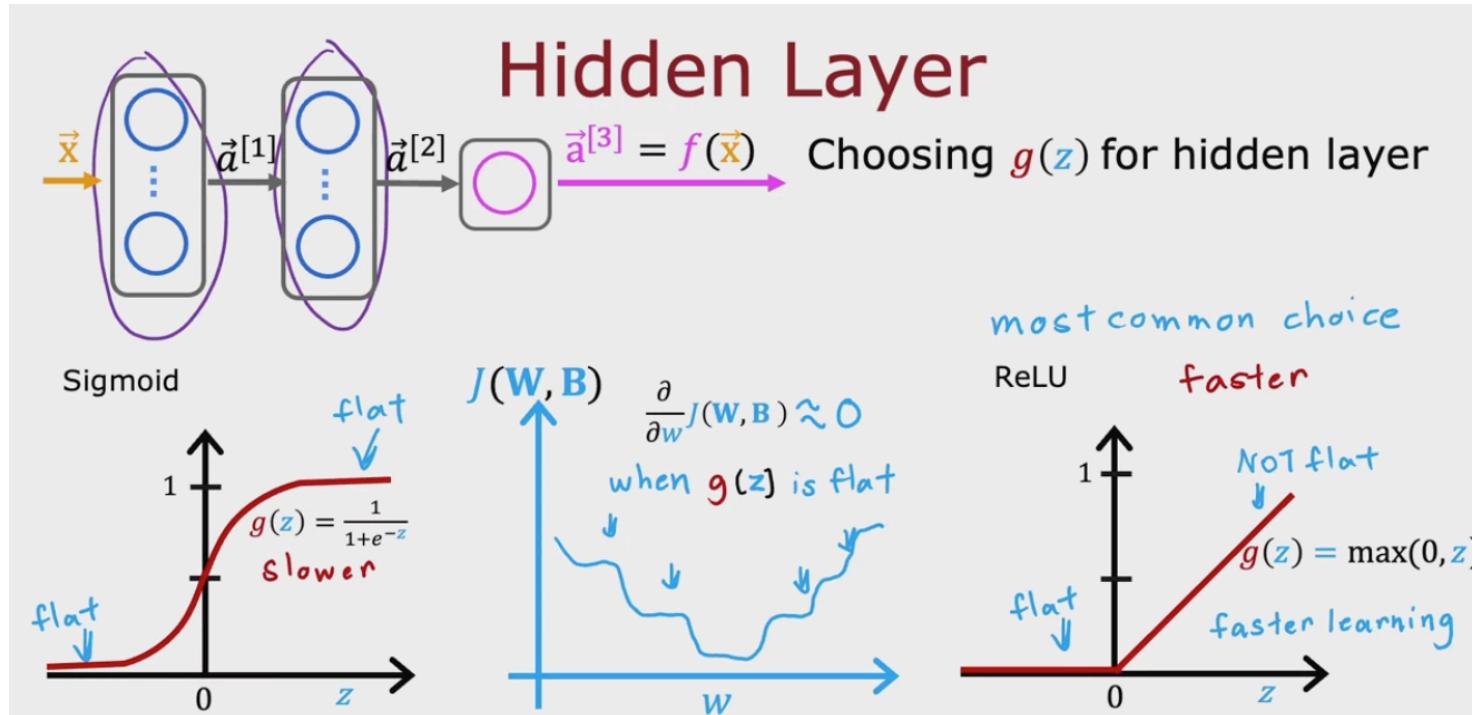
ReLU most common for hidden layers.

Early history: sigmoid. but now hardly ever.

## Why?

A bit faster :  $g(z) = \max(0, z)$

at sigmoid, flat on two sides. but in ReLU, flat in only one side. Effects on Gradient Descent. So neural networks may learn faster in ReLU.



there are other activation functions where some cases they work better, note.

## Multiclass Classification

There is no 2 options - good coffee, bad coffee - there are not 2 digits to draw - 0 and 1 - but all numbers.

Softmax:

<b>Logistic regression</b> (2 possible output values) $z = \vec{w} \cdot \vec{x} + b$ $a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1 \vec{x})$ $a_2 = 1 - a_1 = P(y=0 \vec{x})$	<b>Softmax regression (4 possible outputs)</b> $y=1, 2, 3, 4$ $z_1 = \vec{w}_1 \cdot \vec{x} + b_1$ $a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$ $= P(y=1 \vec{x})$ $a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$ $= P(y=2 \vec{x})$ $a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$ $= P(y=3 \vec{x})$ $a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$ $= P(y=4 \vec{x})$
<b>Softmax regression</b> (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 $w_1, w_2, \dots, w_N$ $b_1, b_2, \dots, b_N$ $a_j = \frac{e^{z_j}}{\sum_{k=1}^N e^{z_k}} = P(y=j \vec{x})$	<span style="background-color: #ccc; padding: 2px;">another variable k to index</span>

if  $n=2$  in softmax  $\rightarrow$  then this becomes logistic regression, so softmax is a generalization for logistic regression.

# Cost

## Logistic regression

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

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

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

$$\text{loss} = -y \log a_1 - (1 - y) \log(1 - a_1)$$

if  $y=1$

if  $y=0$

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

## Softmax regression

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

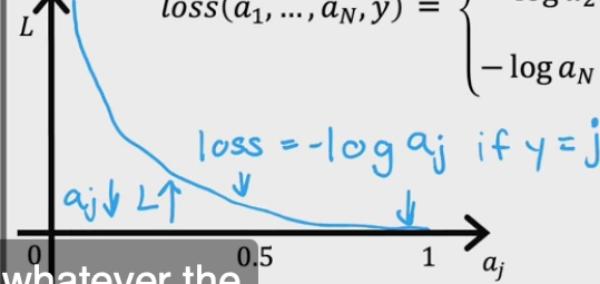
$$\vdots$$

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

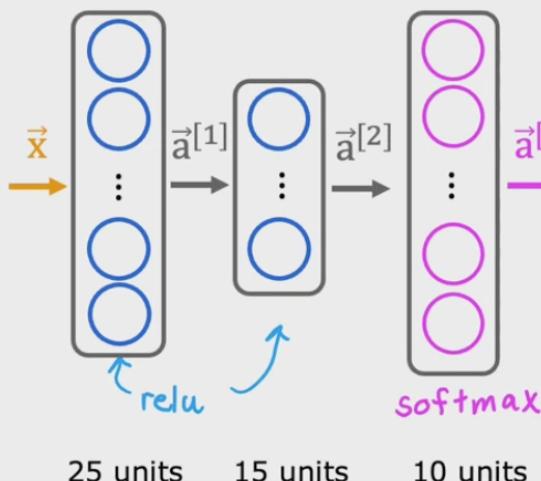
### Crossentropy loss

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

Because whatever the



## Neural Network with Softmax output



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

$$\vdots$$

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

### logistic regression

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

If you want to implement the neural network that I've shown here on,  $\vec{z}_1^{[3]}, \vec{z}_2^{[3]}, \dots, \vec{z}_{10}^{[3]}$

How is the neural network with multiple outs? Ex: 10 outs, 10 units in the final layer - output layer. now with softmax activation function.

roundoff errors, minimizing this error

$$x = 2/10000$$

(after making some subtraction and addition, it becomes something different)

$$\{ x = .18f \}$$

# Numerical Roundoff Errors

More numerically accurate implementation of logistic loss:

Logistic regression:

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

Original loss

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

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

More accurate loss (in code)

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

to have a little bit less

this is worse in softmax

## More numerically accurate implementation of softmax

Softmax regression

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

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

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

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.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))
except that it is a little bit
```

but here, you do not actually get the probability. to do so, use the following:

# logistic regression (more numerically accurate)

```
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)
```

the logistic function in order

- 1. Softmax Activation Function:** When using the softmax activation function directly in the final layer of a neural network (especially for classification problems), you convert the raw logits (pre-activation values) into probabilities. This works fine most of the time, but softmax involves calculating exponentials of the logits, which can lead to numerical issues (like overflow or underflow) when the logits are very large or very small. This is particularly problematic for high-dimensional input data or deep networks where extreme values in logits can destabilize training.
- 2. Alternative with Linear Activation:** Instead of using the softmax activation directly, you can use a **linear activation** (i.e., no activation function) for the final layer, which outputs the raw logits. Then, during the loss calculation, you apply softmax **inside the loss function** (such as SparseCategoricalCrossEntropy with from\_logits=True). By doing this, TensorFlow (or the framework you're using) handles the softmax computation internally in a more numerically stable way, often leveraging techniques like log-sum-exp to prevent overflow and underflow.

## Why Linear Activation Helps:

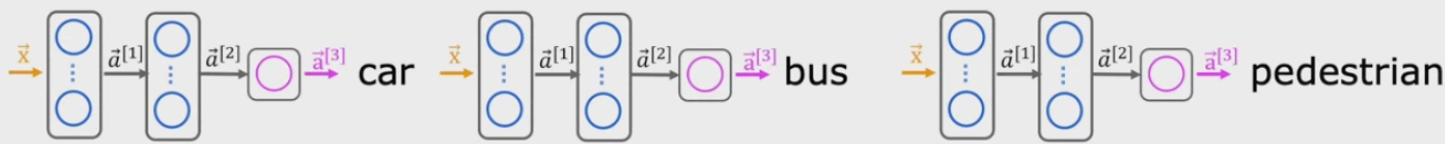
- **From Logits=True:** When you use a linear activation followed by a cross-entropy loss with from\_logits=True, the softmax and the logarithm are combined into one numerically stable operation. This prevents large exponents from being computed explicitly, which can avoid rounding errors.
- **Log-sum-exp Trick:** The underlying loss function uses the “log-sum-exp” trick, which is mathematically equivalent to softmax but avoids extreme values that cause numerical problems.

## Multi-Label Classification Problem

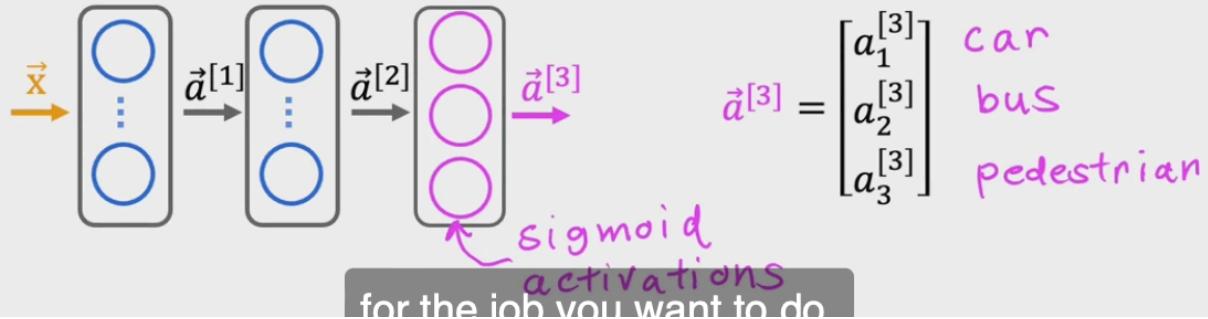
Is there a car in the image? A person? Ambulance? - More than one questions neural network should answer.

**Can use more than one neural network. (One for each)**

# Multi-label Classification



Alternatively, train one neural network with three outputs

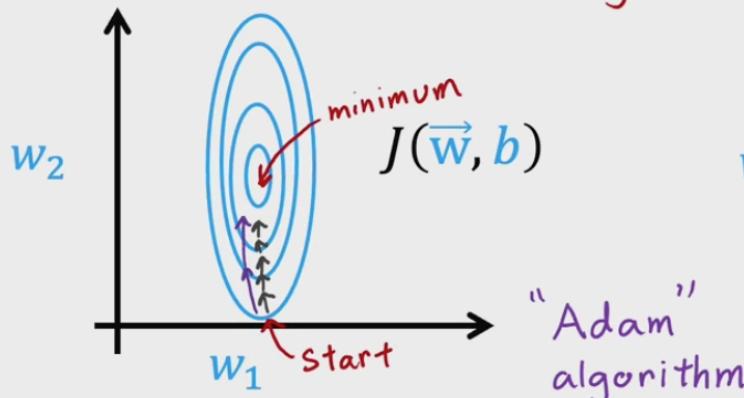


Training your neural network much faster than gradient descent

## Gradient Descent

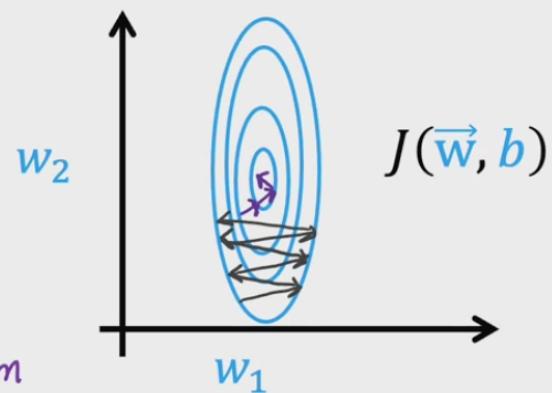
$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

learning rate



Go faster – increase  $\alpha$

Depending on how gradient



Go slower – decrease  $\alpha$

"Adam" algorithm decides the alpha step for the gradient descent.

Uses different learning rates for each of the parameters. (lets say 10 units, 11 learning rate alpha's - one for the  $b$  constant)

## MNIST Adam

```
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 the model

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

# Fit the model

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

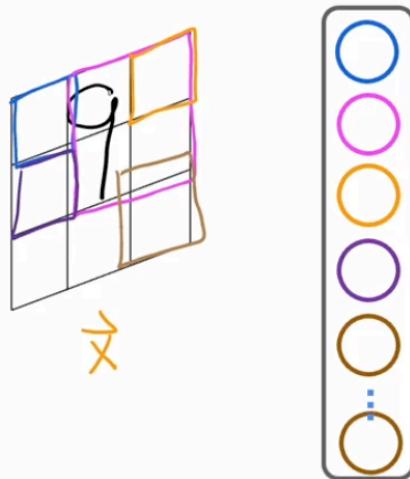
become a standard instead of gradient descent - much faster.

## Additional Layer Types

**Dense Layer:** Each neuron output is a function of all the activation outputs of the previous layer.

**Convolutional Layer:**

## Convolutional Layer



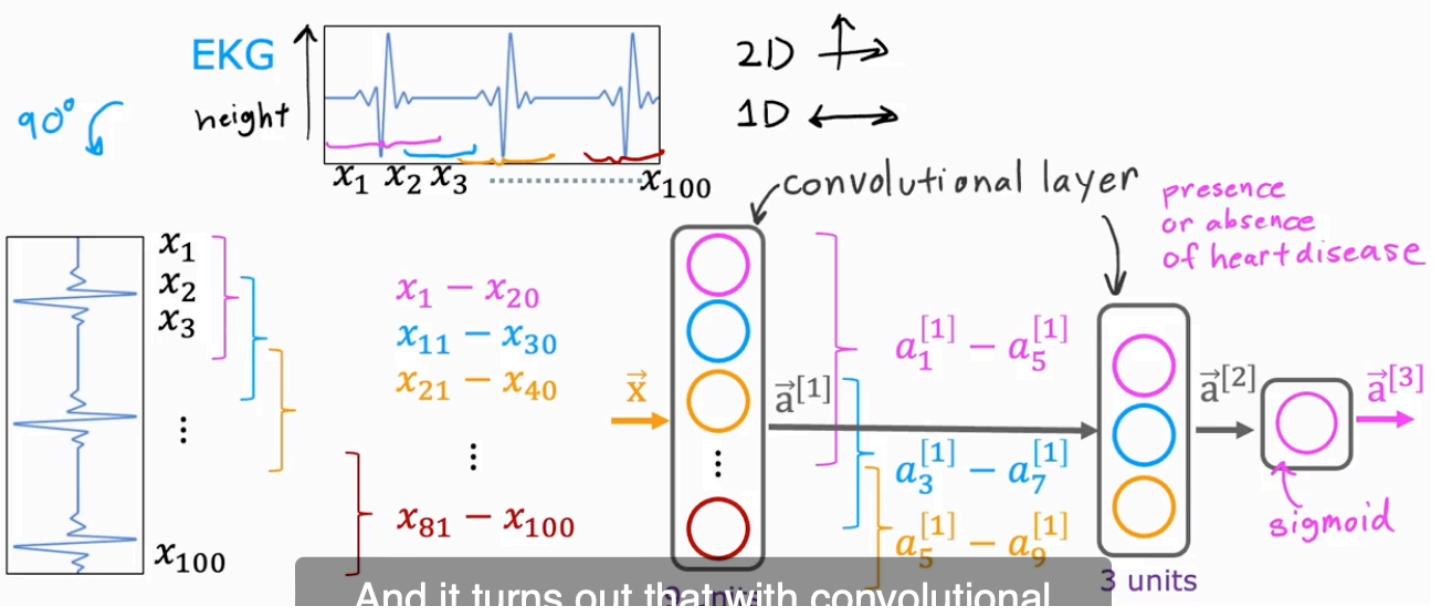
Each neuron only looks at part of the previous layer's outputs.

Why?

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

When we talk about practical tips for

## Convolutional Neural Network



- May look into here more later.