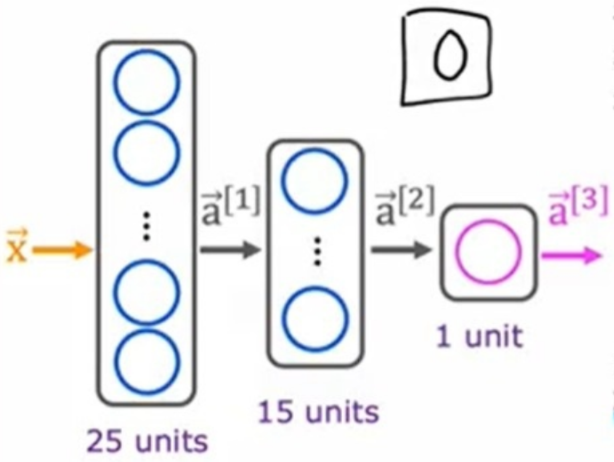
# Training a Neural Network in TensorFlow

## Train a Neural Network in TensorFlow




```
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)
```

specify model ①

compile with loss fn ②

fit model ③

epochs: number of steps in gradient descent

Given set of $(x,y)$ examples

How to build and train this in code?

earlier logistic reg:-

1) given $w, b$ compute output

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

$f\_x = g(z)$ $\quad \frac{1}{1+e^{-z}}$

2) specify loss & cost

$-y * \log(f\_x) - (1-y)\log(1-f\_x)$

cost func:

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

3) Train on data to minimize $J(w,b)$

$w = w - \alpha * dj\_dw \qquad b = b - \alpha * dj\_db$

| | 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`<br>`f_x = 1/(1+np.exp(-z))` | `model = Sequential([`<br>`Dense(...)`<br>`Dense(...)`<br>`Dense(...)`<br>`])` |
| ② specify loss and cost<br>$L(f_{\vec{w},b}(\vec{x}), y)$ 1 example<br>$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$ | logistic loss<br>`loss = -y * np.log(f_x)`<br>`-(1-y) * np.log(1-f_x)` | binary cross entropy<br>`model.compile(`<br>`loss=BinaryCrossentropy())` |
| ③ Train on data to minimize $J(\vec{w}, b)$ | `w = w - alpha * dj_dw`<br>`b = b - alpha * dj_db` | `model.fit(X,y,epochs=100)` |

( gradient descent)

(model.compile
(loss = MeanSquaredError)

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

$$W^{[1]}, W^{[2]}, W^{[3]} \ldots \qquad b^{[1]}, b^{[2]}, b^{[3]}$$

repeat;

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

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

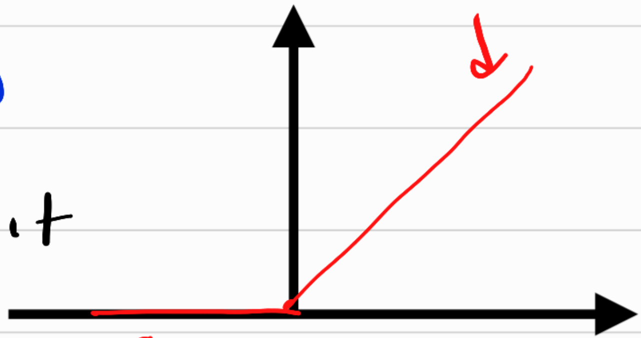
model.fit(x, y, epochs=100)

$\longrightarrow$ compute derivatives for gradient descent using "back propogation"

Alternatives to sigmoid ;

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

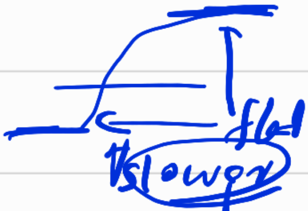
Rectified Linear Unit — ReLU




choosing :-

Output layer

binary classification → sigmoid




regression problem → linear activation
( eg :- stock price - , + )
(+) or (-)

Regression (+) → ReLU → most common choice

Hidden Layer → ReLU

if all activations were linear
→ equivalent to linear regression

if all act. were linear except output logistic

→ logistic regression