In [ ]:

```
import numpy as np
import torch
from torch import nn, optim
import random
```

La "neurona"

Representación matemática

$$y = W * x + b$$

Por ejemplo:

$$0.2 = 0.05 * 4$$

In [ ]:

```
x = 4
W = 0.05

y = W * x

print("y =", y)
```

$$1.5 = W * 6$$

Partamos del mismo problema pero suponiendo que no conocemos el valor de W

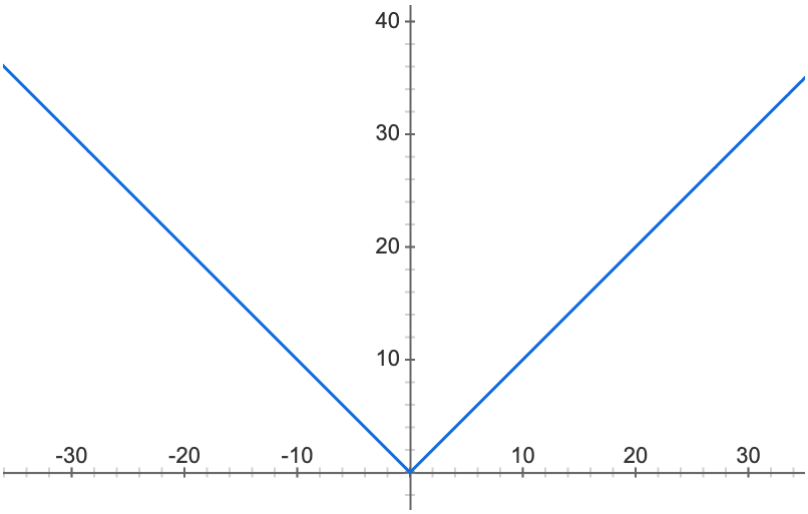
In [ ]:

```
x = 6
W = random.uniform(0, 0.5)

y = W * x
print("W =", W)
print("y =", y)
```

Queríamos que el resultado de la ecuación fuese 1.5, pero tenemos 0.389...

Empecemos por calcular por cuanto nos hemos equivocado, usando el valor absoluto ya que tiene un mínimo



El valor absoluto nos da una pendiente de 1 en valores positivos, y una pendiente de -1 en valores negativos, podriamos usar eso como medida de cuanto nos hemos equivocado en W.

Sin embargo, queremos algo más preciso, así que multiplicamos la pendiente del error (-1 o 1) por la pendiente de la ecuación inicial en base a W, que es x = 6

In [ ]:

```
y = W * x
error = y - 1.5

print("Queremos 1.5, pero y =", y)
print("Nuestro error es de", abs(error))
```

Vamos a intentar modificar W en base a cuánto nos hemos equivocado

In [ ]:

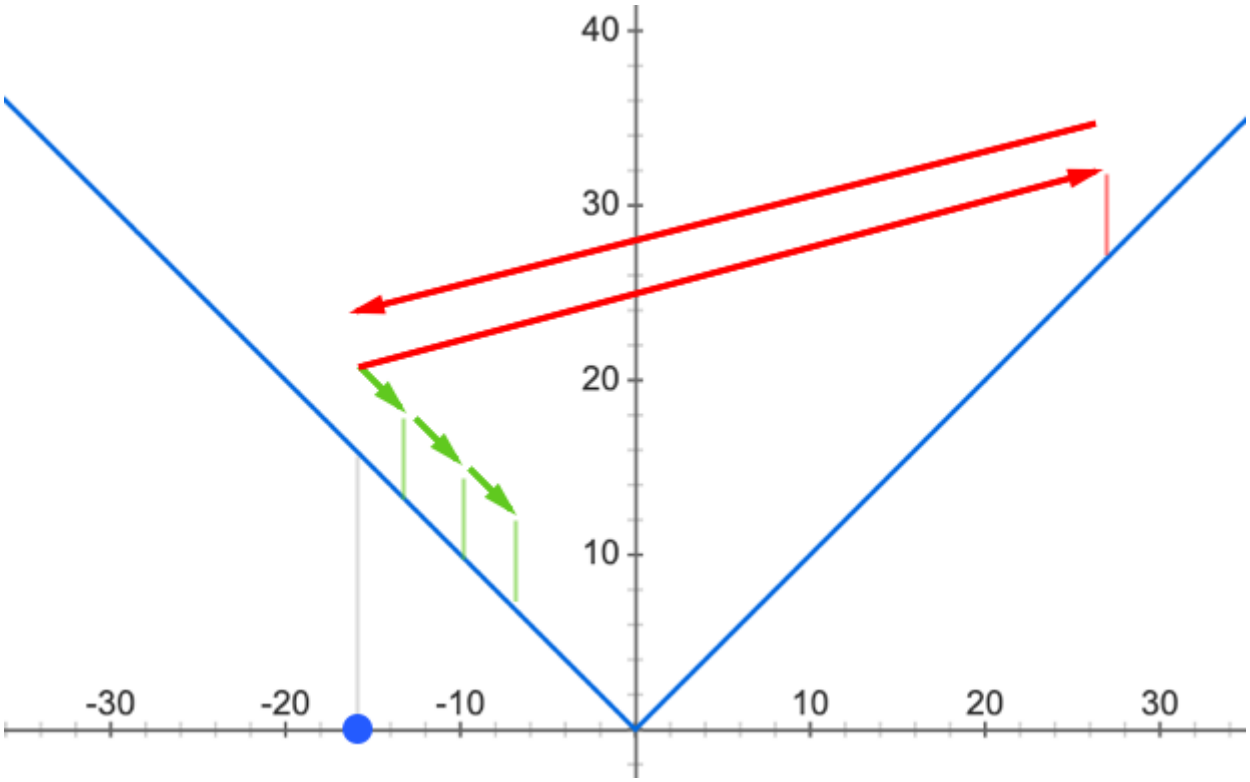
```
y = W * x
print("y = ", y)

pendiente_error = -1 if error < 0 else 1 # -1 para valores negativos, 1 para val
ores positivos
pendiente_w = x * pendiente_error

nuevo_W = W - pendiente_w * 0.001
y = nuevo_W * x

print("Tras modificar W, y =", y)
```

Si damos pasos muy grandes nos pasamos y nunca llegamos al mínimo, así que mejor dar pasos pequeñitos



In [ ]:

```
for i in range(100):
    y = W * x
    error = y - 1.5
    pendiente_error = -1 if error < 0 else 1
    pendiente_w = x * pendiente_error
    W = W - pendiente_w * 0.0001

y = W * x
print("Tras 100 pasos, y =", y, "y W =", W)
```

Y así tenemos nuestra primera neurona!!!

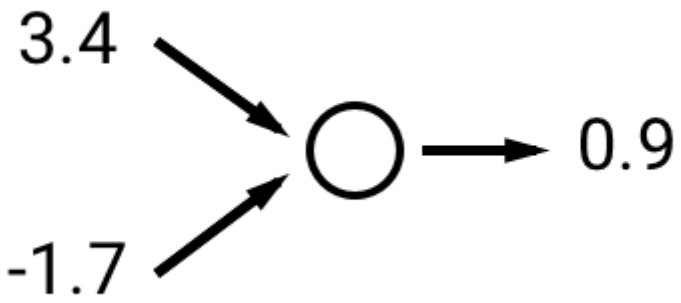
Capas de neuronas

Hemos visto una neurona con **un** valor de entrada y **un** valor de salida que equivale a una ecuación lineal:

$$y = W * x + b$$

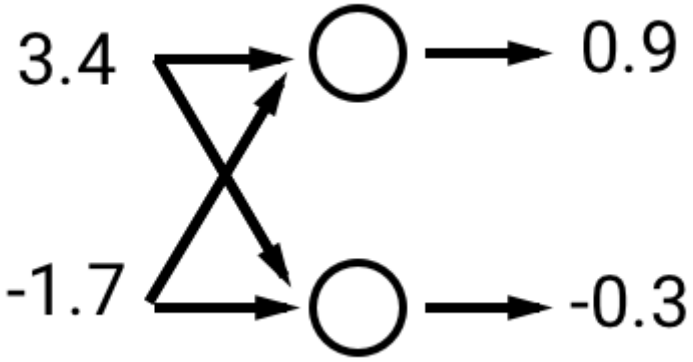
Pero... ¿Cómo sería una neurona con varios valores de entrada?
Sencillamente tendremos varios x y varios W.

$$y = (W1 * x1) + (W2 * x2) + \ldots + (Wn * xn) + b$$



¿Y si queremos varios valores de salida?
Como hemos visto antes, usamos una ecuación lineal que nos permite tener varios x, pero solo un y, por lo que para tener varios valores de salida necesitamos... ¡ MAS NEURONAS ! Lo que significa más ecuaciones:

$$y1 = (W11 * x1) + (W21 * x2) + \ldots + (Wn1 * xn) + b1$$
$$y2 = (W12 * x1) + (W22 * x2) + \ldots + (Wn2 * xn) + b2$$
$$\ldots$$
$$ym = (W1m * x1) + (W2m * x2) + \ldots + (Wnm * xn) + bm$$



Cada una de estas neuronas recibe todos los inputs y devuelve un único output, y tenemos una forma más bonita de representarlas, usando matrices:

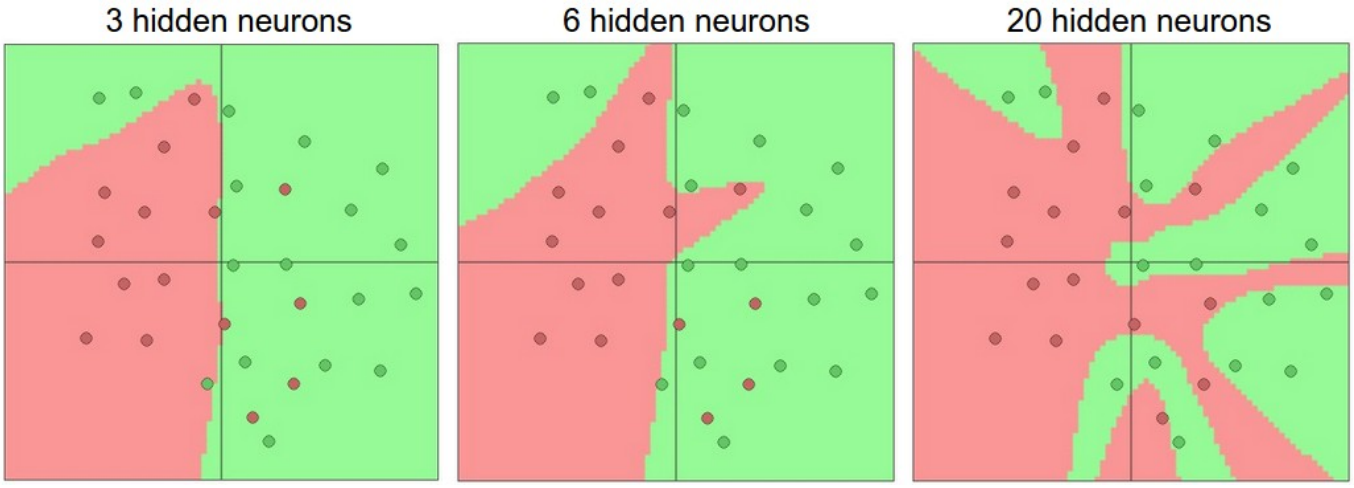
$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \end{bmatrix} + \begin{bmatrix} b \\ b \end{bmatrix}$$

De 'Shallow' a 'Deep'

Ya sabemos cómo crear una capa de neuronas usando ecuaciones lineales, con multiples inputs y multiples outputs.

Sin embargo, al ser operaciones lineales las relaciones que la red será capaz de extraer de los inputs serán lineales, así que para solucionar esto vamos a encadenar varias capas de neuronas, pasando los outputs de una de nuestras capas a la siguiente, y añadiendo una operación no-lineal entre ellas.

Cuantas más capas añadamos, más compleja es la representación de los datos que podemos obtener:



Todo esto ya parece muy dificil de hacer... No hay algo que me lo de ya hecho?

Por supuesto, a la hora de trabajar con redes neuronales no definimos las operaciones una a una, ni tenemos que calcular nosotros mismos el error, sino que usamos frameworks que nos abstraen de todo esto.

Por ejemplo, probemos con PyTorch y su maravilloso autograd

In [ ]:

```
X = torch.tensor([[-1, 3, 5]]).float()
Y = torch.tensor([[0.7, 1.3, 2.1]]).float()

W1 = torch.randn(3, 6, requires_grad=True)
W2 = torch.randn(6, 3, requires_grad=True)

optimizer = optim.SGD([W1, W2], lr=0.01)

for i in range(3000):
    pred = torch.sigmoid(X.mm(W1)).mm(W2)
    error = torch.abs(Y - pred).mean()

    optimizer.zero_grad()
    error.backward()
    optimizer.step()

torch.sigmoid(X.mm(W1)).mm(W2)
```

O lo que es todavía más facil, en vez de definir nosotros las dos matrices W, podemos usar capas pre-definidas de PyTorch!!!

In [ ]:

```
X = torch.tensor([[-1, 3, 5]]).float()
Y = torch.tensor([[0.7, 1.3, 2.1]]).float()

network = nn.Sequential(
    nn.Linear(3, 6),
    nn.Sigmoid(),
    nn.Linear(6, 3)
)

optimizer = optim.SGD(network.parameters(), lr=0.01)

for i in range(3000):
    pred = network(X)
    error = torch.abs(Y - pred).mean()

    optimizer.zero_grad()
    error.backward()
    optimizer.step()

network(X)
```