

```
In [1]: import torch  
import matplotlib.pyplot as plt
```

Modelos Neuronales I

Autor: Jorge García González (Universidad de Málaga)

Última Actualización: 25/09/2025

Asignatura: Programación para la Inteligencia Artificial

Hasta ahora hemos hablado del aprendizaje y lo hemos planteado simplemente como el ajuste de parámetros de un modelo. Dicho ajuste lo hemos realizado valiéndonos de las propiedades del gradiente del error que comete el modelo respecto a dichos parámetros para unos datos concretos. Esa estrategia es a grandes rasgos lo que se conoce como el método del Descenso del Gradiente Estocástico (Stochastic Gradient Descent).

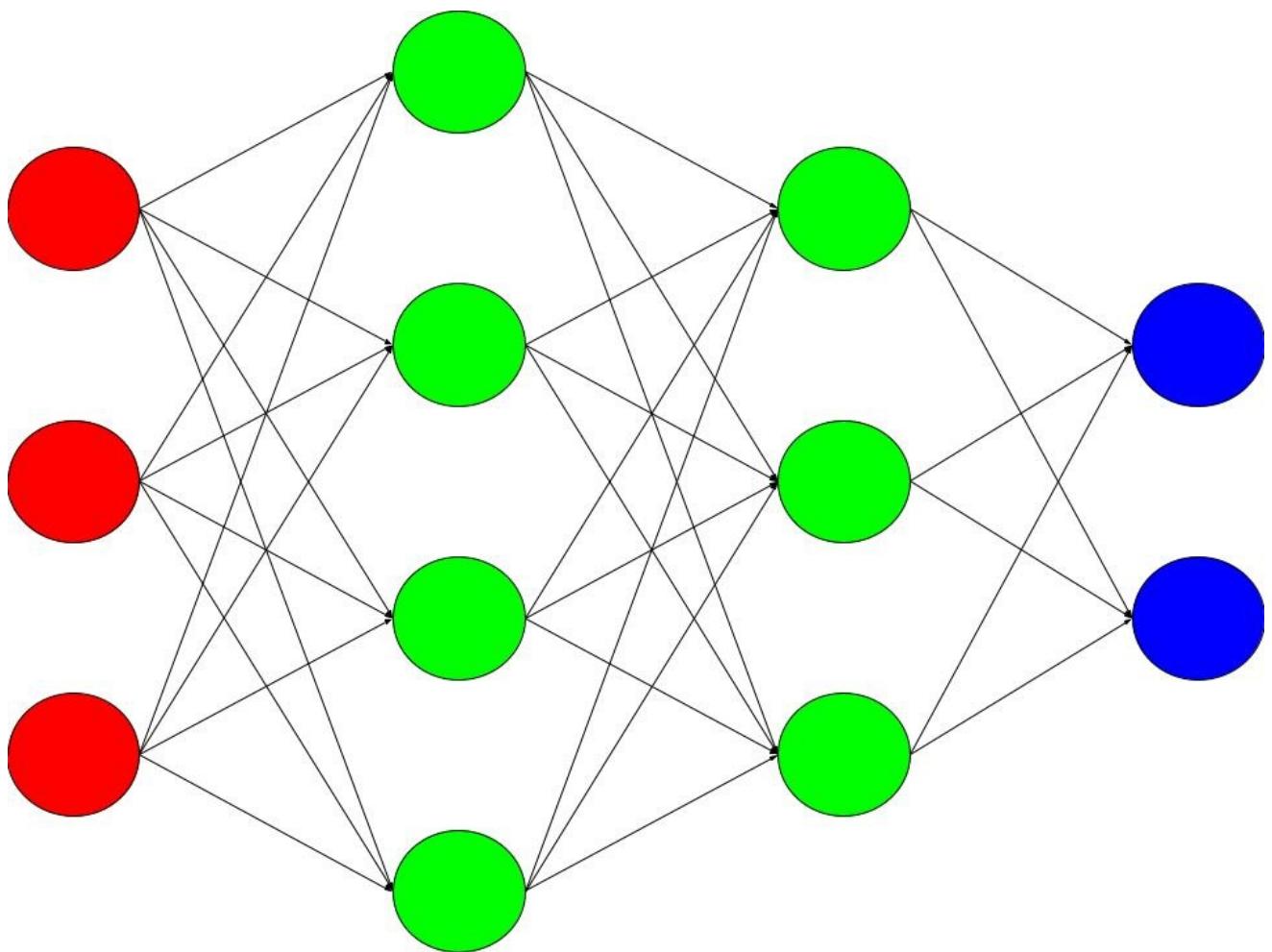
Ahora vamos a centrarnos en el modelo a optimizar. De nuevo vamos a rescatar algunas funciones de sesiones anteriores:

```
In [26]: def get_data_from_a_line(m, b, n, sigma=2, range=50):  
    """  
        Función que devuelve datos a partir de una recta. Recordemos que la ecuación de la recta está definida por y=m*x+b.  
        Args:  
            m: inclinación de la recta.  
            b: ordenada en el origen de la recta.  
            n: número de datos que queremos obtener.  
            sigma: desviación típica para el ruido que añadiremos a los datos (2 por defecto).  
            range: extremo del rango [0,rango] en el eje x en el que queremos situar los datos (50 por defecto).  
    """  
    x = range*torch.randn(n) # Generamos los datos en el eje x.  
    y = m*x+b # Obtenemos la posición correspondiente en y.  
    noise_x = torch.normal(torch.zeros_like(y), sigma*torch.ones_like(y)) # Generamos un vector de ruido gaussiano en x.  
    noise_y = torch.normal(torch.zeros_like(y), sigma*torch.ones_like(y)) # Generamos un vector de ruido gaussiano en y.  
    x_with_noise = x + noise_x  
    y_with_noise = y + noise_y  
  
    data = torch.cat((x_with_noise[:,None], y_with_noise[:,None]), dim=1)  
    return data  
  
def line_model(m, b, inputs):  
    """  
        Función para modelar una recta.  
        Args:  
            m: inclinación de la recta.  
            b: ordenada en el origen de la recta.  
            inputs: vector con los valores de entrada para ser calculados por el modelo.  
    """  
    return m*inputs+b
```

Hasta ahora hemos trabajado con un modelo lineal. Sirve para modelar una recta en 2 dimensiones. Cualquier recta en 2 dimensiones, pero solo rectas en 2 dimensiones. Por poder, se puede intentar ajustar un modelo lineal a cualquier conjunto de datos y minimizar el error, pero lo máximo que conseguiríamos sería una recta que quede más o menos centrada respecto a los datos. Nada que consideremos un ajuste razonable.

Por eso hace falta un modelo más potente. Más *flexible*. Y es ahí donde entran los modelos neuronales.

Llamamos Redes Neuronales Artificiales (RNA o ANN por sus siglas en inglés) a los modelos que combinan neuronas artificiales formando un grafo. Un ejemplo típico es el de un perceptrón:



Ahora no es relevante qué representan los círculos rojos, verdes y azules. Lo relevante es que en conjunto representan un grafo en el que hay unas unidades de cómputo que llamamos neuronas y las salidas de unas unidades de cómputo se convierten en las entradas de otras.

Una neurona artificial es una función matemática que se inspira en el funcionamiento de las neuronas naturales. Es un modelo bioinspirado, pero el objetivo no es modelar el funcionamiento biológico. Las neuronas artificiales no están limitadas por modelar las neuronas naturales, sino que toman ideas de su funcionamiento. De ahora en adelante nos referiremos siempre a neuronas artificiales, así que por simplicidad omitiremos el *artificiales* y simplemente hablaremos de neuronas.

Como hemos dicho, una neurona es una función matemática. Hay diversos tipos de estas funciones en la bibliografía (lineales, convolucionales, LSTM, etc) con utilidades distintas. Nosotros nos vamos a centrar en la neurona clásica. También denominada neurona lineal o densa, que se define como:

$$y = a(\sum w_i x_i - b)$$

Siendo w un vector de pesos, x un vector de entradas, b un valor umbral (también llamado desviación o *bias*) y a una función de activación. Los parámetros que se ajustan son w y b .

Antes de entrar en la arquitectura, vamos a pensar detenidamente en la fórmula de la neurona clásica. Por ahora supongamos que a es la función identidad $f(x)=x$ (la ignoramos). Nos queda la función $y=\sum w_i x_i - b$. Supongamos además que los vectores w y x tienen una única componente. Podemos reducirlo a $y=wx-b$. ¡Un momento! ¡Esa es la ecuación de la recta en 2D! Si la extendemos un poco tenemos $y=w_i x_i + w_j x_j - b$ que representa un plano y así podemos seguir extendiendo dimensiones. Si a es la función identidad, nuestra neurona no es más que una función lineal y una función lineal se caracteriza por tener siempre la misma inclinación. Vamos a verlo en 2D.

Para empezar, vamos a crear algunas funciones que permitan usar polinomios y generar datos a partir de un polinomio.

```

# El cat concatena
# para un solo tensor.
# Finalmente sumamos los
# generados.

return y

return apply_polynomial

def get_data_from_a_polynomial(poly, n, sigma=2, range=50):
    """
    Función que devuelve datos a partir de una recta. Recordemos que la ecuación de la recta está definida por y = mx + b.

    Args:
        poly: Función que aplica un polinomio.
        n: número de datos que queremos obtener.
        sigma: desviación típica para el ruido que añadiremos a los datos (2 por defecto).
        range: extremo del rango [0,rango] en el eje x en el que queremos situar los datos (50 por defecto).
    """
    x = range*torch.rand(n)
    y = poly(x)
    noise_x = torch.normal(torch.zeros_like(y), sigma*torch.ones_like(y))
    noise_y = torch.normal(torch.zeros_like(y), sigma*torch.ones_like(y))
    x_with_noise = x + noise_x
    y_with_noise = y + noise_y

    data = torch.cat((x_with_noise[:,None], y_with_noise[:,None]), dim=1)
    return data

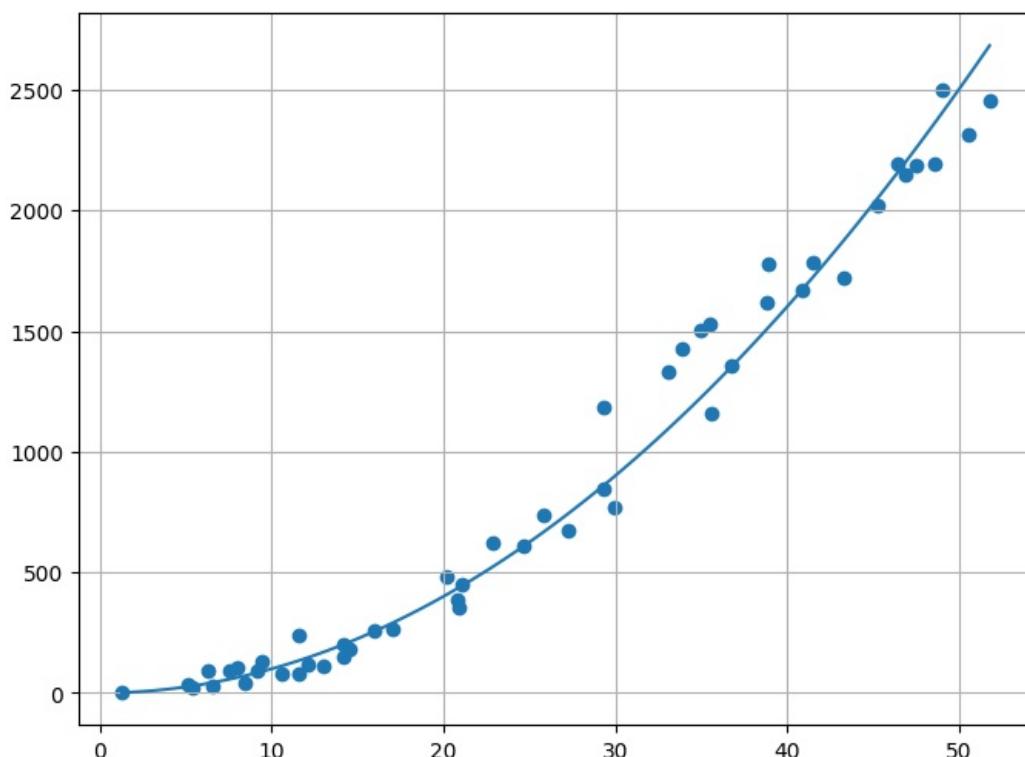
def draw_2D_function(function, range=[-4,4], function_label = None, title = None, data=None):
    plt.figure(figsize=(8,6))
    if data is not None:
        plt.scatter(data[:,0], data[:,1])
        x = torch.linspace(data[:,0].min(), data[:,0].max(), 100)
    else:
        x = torch.linspace(range[0], range[1], 100)
    y = function(x)
    plt.plot(x, y, label=function_label)

    if title is not None:
        plt.title(title)
    if function_label is not None:
        plt.legend()
    plt.grid(True)
    plt.show()

```

Nota: Sí. La función generate_polynomial genera otras funciones. Con esas funciones puedo generar polinomios de los que generar datos con un poco de ruido como hacíamos con las rectas. Por ejemplo, del polinomio $y=x^2$:

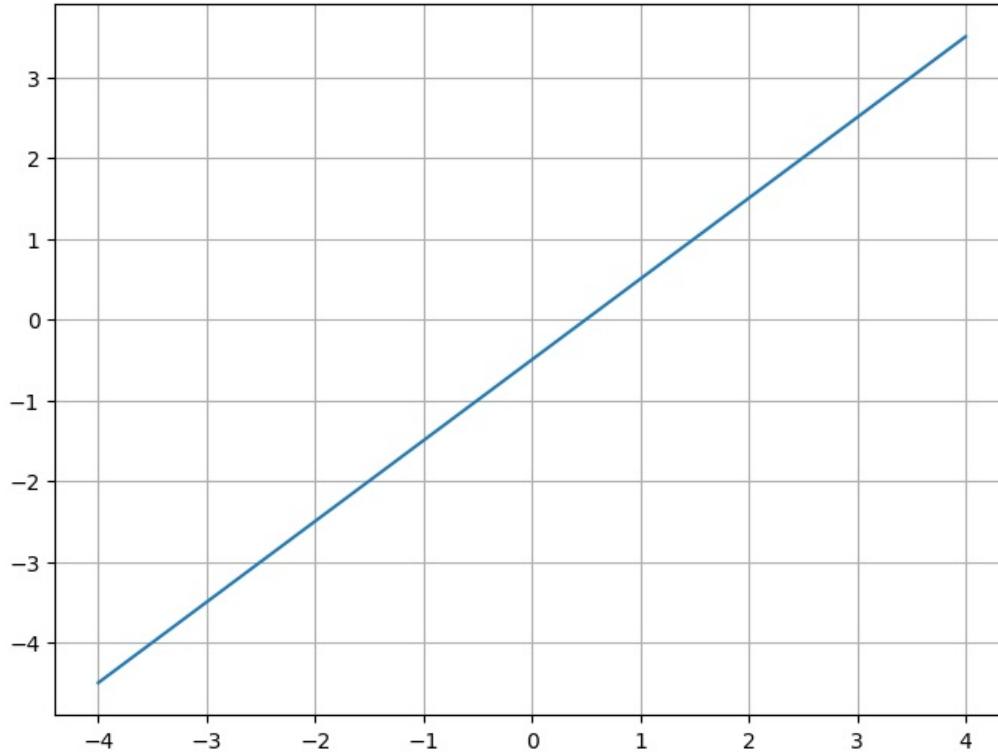
```
In [28]: polynomial_coefficients = [1,0,0]
polynomial = generate_polynomial(polynomial_coefficients)
data = get_data_from_a_polynomial(polynomial, 50)
draw_2D_function(polynomial, data=data)
```



¿Cómo puedo representar esa curva con un modelo lineal de la forma $y=mx+b$? ¡La inclinación de ese modelo va a ser siempre m ! Su derivada respecto a x es constante, lo que quiere decir que la función no se *dobra* en ningún lugar. No puede tener curvas. Lo mismo pasa con todo modelo lineal con más variables. La inclinación es siempre la misma. Además, combinar muchos modelos lineales da como resultado otro modelo lineal.

Vamos a verlo en una prueba sencilla:

```
In [29]: def wrapper_function(inputs):
    return line_model(4,1, inputs) + line_model(-3,-1.5, inputs)
draw_2D_function(wrapper_function)
```



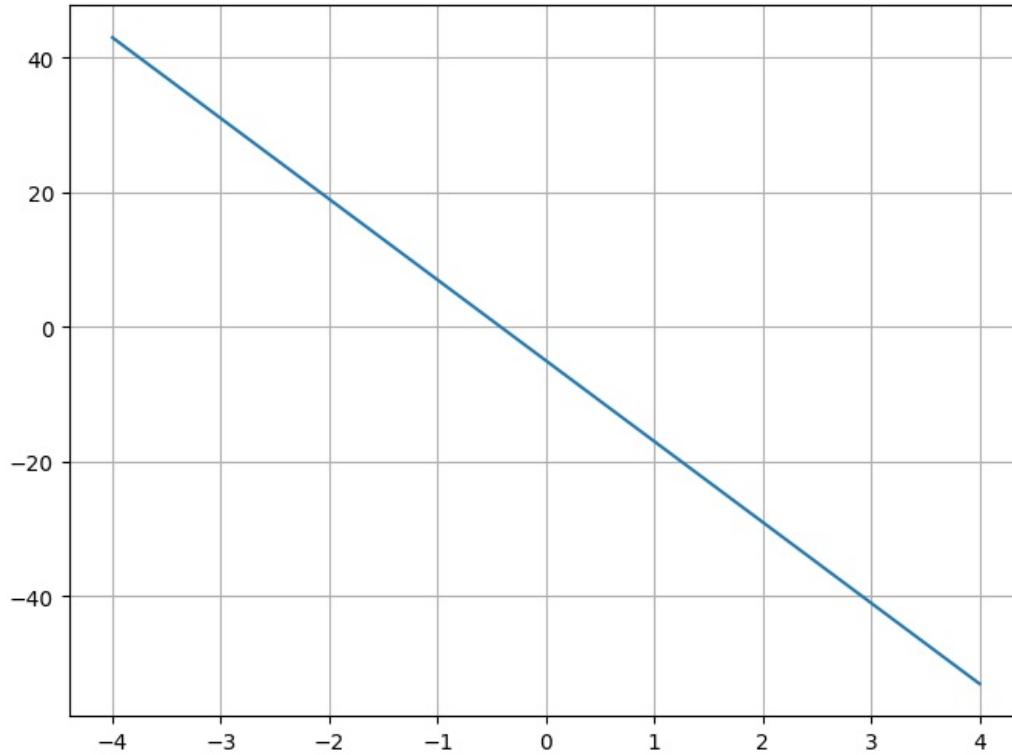
Nota: un wrapper en programación es una función intermedia que sirve para hacer compatibles distintas funciones. Una función intermedia, por así decirlo.

Si cambiamos los valores de m y b de los modelos lineales anteriores el resultado siempre es otro modelo lineal. Al fin y al cabo:

$$y = (m_1x + b_1) + (m_2x + b_2) = (m_1 + m_2)x + (b_1 + b_2)$$

Si intentamos combinarlos secuencialmente (que la salida de uno sea la entrada de otro que es lo mismo que hacer la composición de ambos).

```
In [30]: def wrapper_function(inputs):
    return line_model(4,1, line_model(-3,-1.5, inputs))
draw_2D_function(wrapper_function)
```



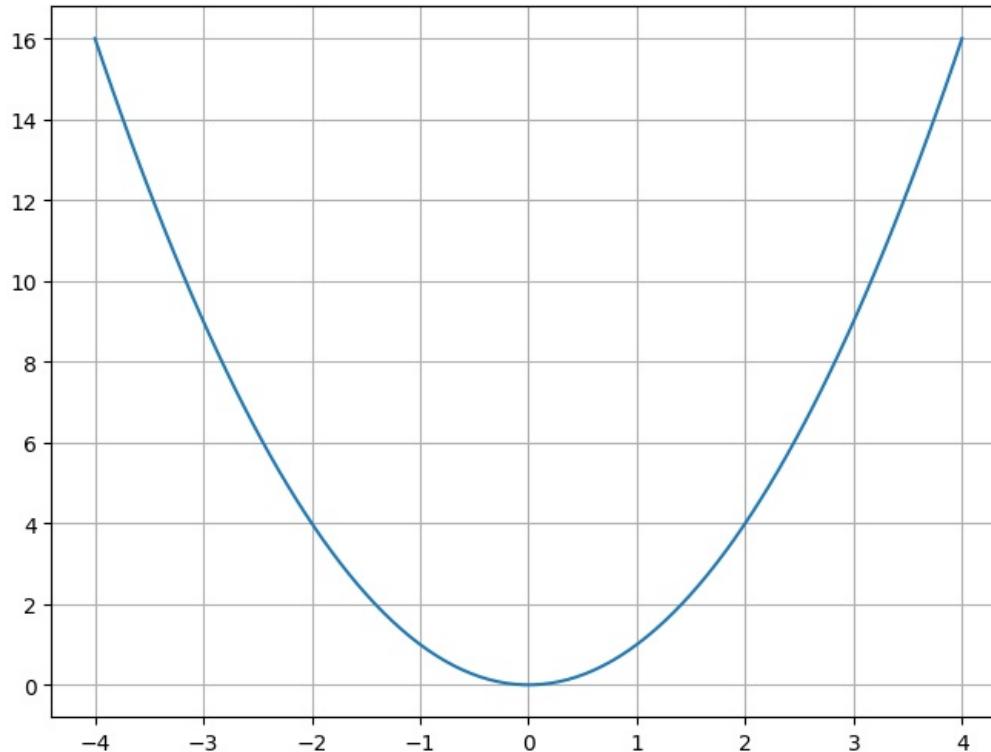
¡Vuelve a ser otra línea recta!

$$y = (m_1(m_2x + b_2) + b_1) = m_1m_2x + m_1b_2 + b_1$$

Si sumar modelos lineales y componerlos tiene como resultado otros modelos lineales, ¿Dónde está el truco de las redes? Por mucho que cree un grafo con muchas neuronas clásicas, el resultado siempre iba a ser una recta. ¿Qué les otorga flexibilidad? El truco está en el elemento que hemos dejado *inutilizado*, la función de activación.

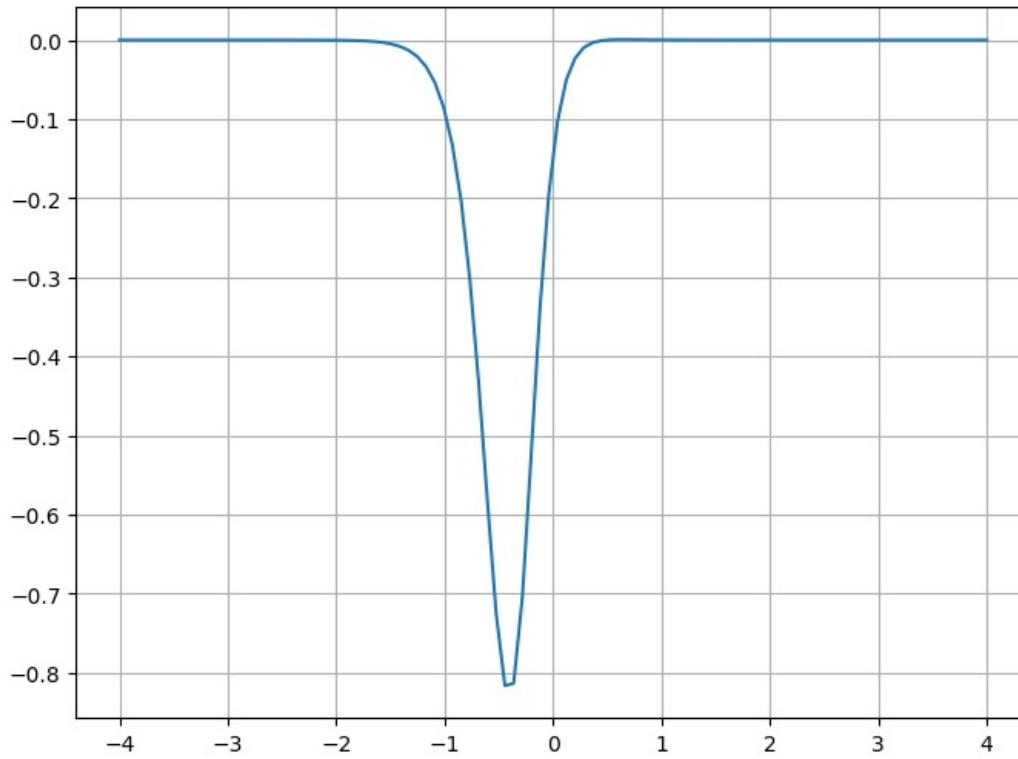
La función $y=x^2$ tiene esta forma:

```
In [31]: polynomial_coefficients = [1, 0, 0]
polynomial = generate_polynomial(polynomial_coefficients)
draw_2D_function(polynomial)
```



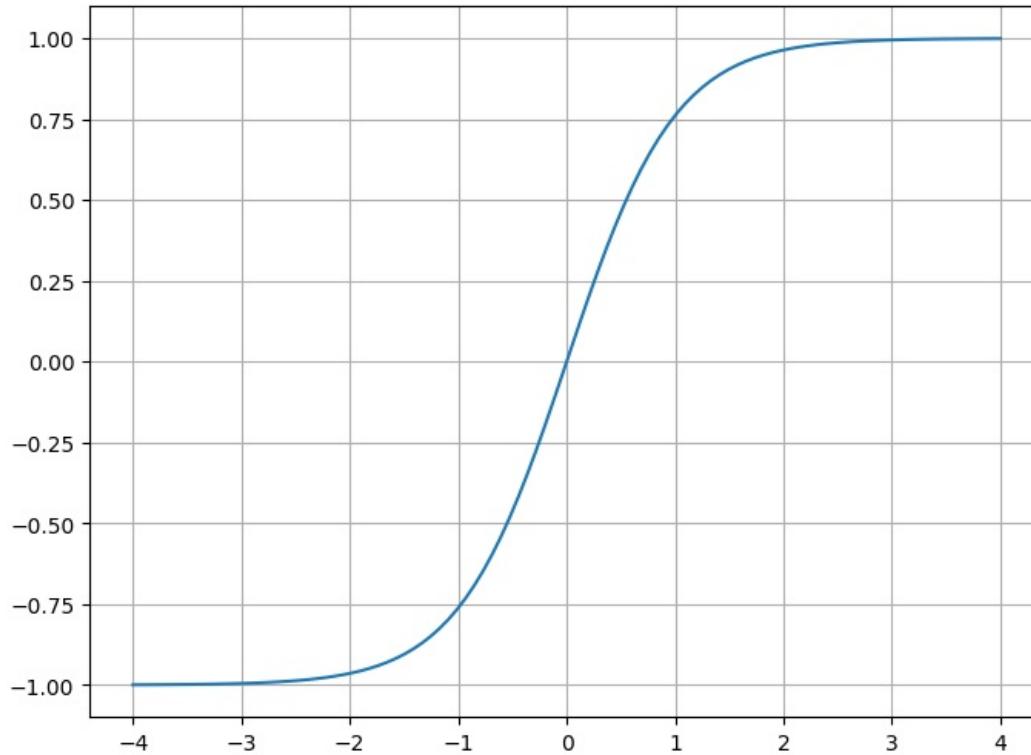
Si combinamos un par de modelos lineales a los que añadimos unas funciones de activación (de las que ahora hablaremos), obtenemos algo así:

```
In [32]: def wrapper_function(inputs):
    return torch.tanh(line_model(4, 1, inputs)) + torch.tanh(line_model(-3, -1.5, inputs))
draw_2D_function(wrapper_function)
```



Vale. No son la misma función. Ni siquiera tienen la misma forma, pero se parecen más. Y lo más importante, ¡No es una línea recta! La función de activación permite representar algo más que líneas rectas. Concretamente la función que hemos utilizado es la tangente hiperbólica (abreviado típicamente a tanh) Y es una función de activación habitual que tiene la forma:

In [33]: `draw_2D_function(torch.tanh)`



Su fórmula es:

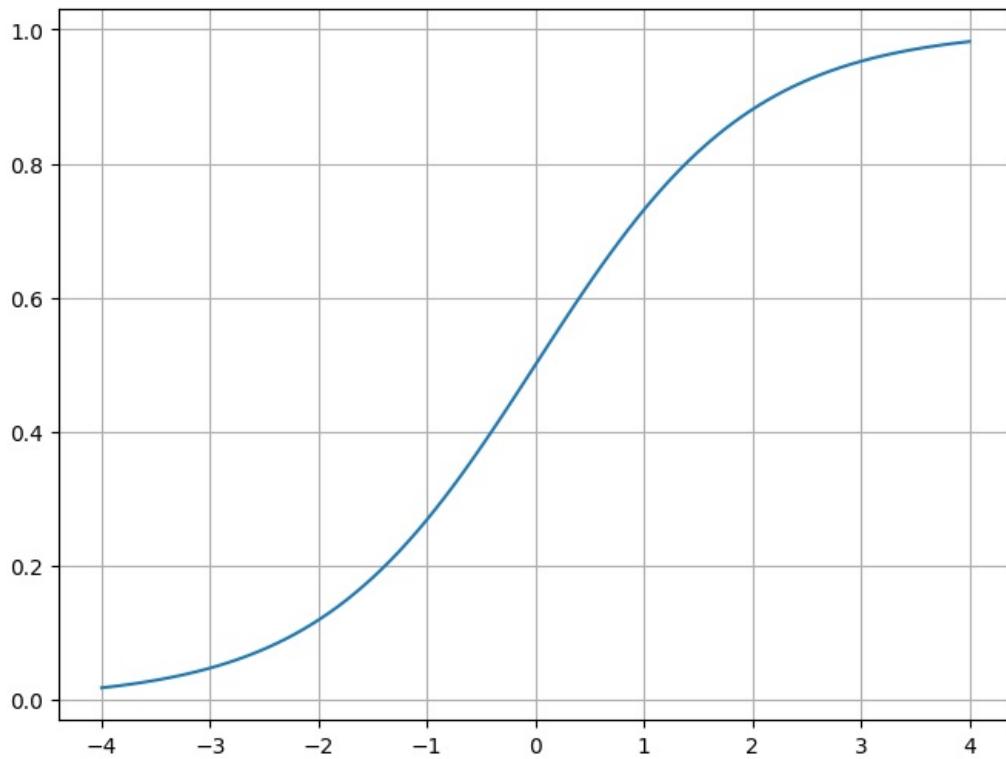
$$\tanh(x) = (e^x - e^{-x}) / (e^x + e^{-x})$$

Y como características destacadas diremos que es derivable en todo su dominio y su resultado está en $(-1, 1)$.

vamos a ver otras funciones de activación relevantes.

La función logística (típicamente llamada sigmoide, aunque técnicamente es un caso particular de función sigmoide):

In [34]: `draw_2D_function(torch.sigmoid)`

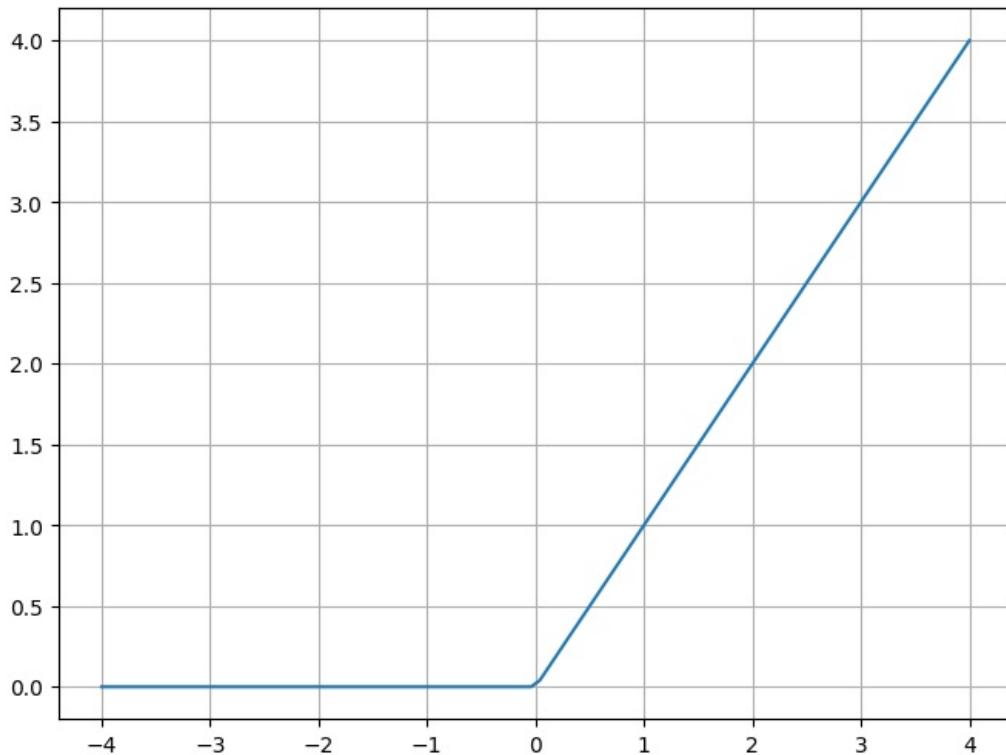


$$\text{sigmoid}(x) = 1/(1+e^{-x})$$

También derivable y con resultados en (0,1).

La función Linear Rectificada (abreviada ReLU por Rectified Linear Unit):

In [35]: `draw_2D_function(torch.relu)`

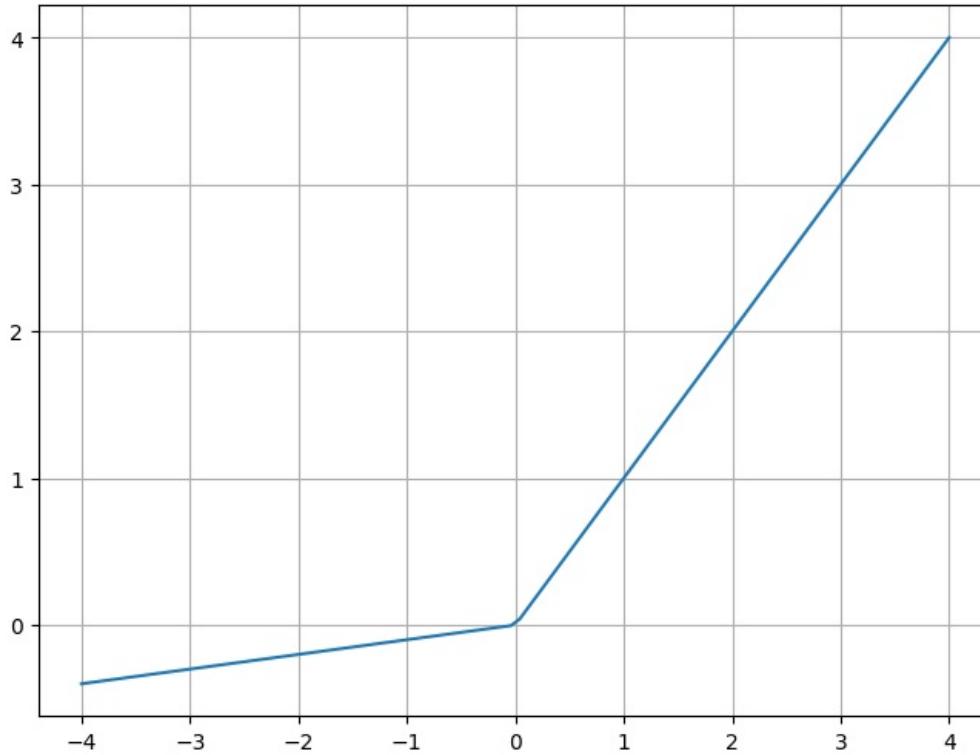


$$\text{ReLU}(x) = \max(0, x)$$

Esta no es derivable en todo su dominio ya que su derivada no está definida en el 0, pero ya hemos visto que eso no siempre es un problema (en ese caso se suele elegir arbitrariamente que la derivada sea 0 o 1). Su resultado está en $[0, +\infty)$.

La LeakyReLU:

In [36]: `leaky_relu = torch.nn.LeakyReLU(negative_slope=0.1)
draw_2D_function(leaky_relu)`



$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative_slope} * \min(0, x)$$

En este caso la función vuelve a no ser derivable en todo el dominio, pero con la ReLU se puede elegir un valor arbitrario (1 o la inclinación negativa) para la derivada en el 0. Su resultado está en $(-\infty, +\infty)$.

Nota histórica: Cuando la función de activación es la función paso ($f(x) = 1$ si $x \geq 1$ y 0 si $x < 0$), a esta neurona se la llama neurona de McCulloch-Pitts (los que la propusieron en 1943) y de ese paso de *activación* cuando el sumatorio supera el umbral viene la terminología de activación y umbral.

Hay muchas otras funciones de activación (algunas más veremos en el futuro), pero estas son las más importantes para cualquier problema de regresión en Aprendizaje Profundo.

¿Qué tienen en común?

- No son funciones lineales. ¡Ya hemos visto que componer funciones lineales tiene como resultado otra función lineal! Justamente queremos que las funciones de activación rompan esa linealidad para permitir a las redes neuronales ajustarse a funciones no lineales.
- Son derivables o los puntos en los que no son derivables los podemos evitar asignando valores razonables sin provocar problemas numéricos. Recordemos que nuestra estrategia de entrenamiento se basa en el gradiente y para eso tenemos que poder obtener derivadas parciales.
- Tienen al menos un rango *sensible*. Rangos en los que cambios no triviales en la entrada provocan cambios no triviales en la salida.

¿Qué se puede observar en algunas de ellas?

Varias tienen rangos no *sensibles*. Son asíntotas en las que cambios en la entrada provocan poco o ningún cambio en la salida. ¡En esas asíntotas la derivada es 0! Si aplicamos la regla de la cadena con una función de activación en su rango insensible, el gradiente se convierte en 0. ¡No podemos ajustar nada! A este problema se le llama el Problema de Desvanecimiento de Gradiente (*Vanishing Gradient Problem*) y es un problema relativamente frecuente a la hora de entrenar modelos. Por eso hay que tener cuidado con el rango de valores en el que se inicializan los modelos y en el que están los datos. Si una neurona se inicializa con parámetros en valores que operados por los datos saquen a la función de su rango sensible, esa neurona no va a poder aprender.

Para terminar este cuaderno vamos a actualizar nuestro ejemplo algoritmo de aprendizaje usando la neurona lineal de PyTorch y esta vez vamos a optimizar ambos parámetros.

```
In [42]: true_m, true_b = [0.5, 0]
data = get_data_from_a_line(true_m, true_b, 50)

learning_rate = 1e-3
steps = 80
loss_fn = torch.nn.L1Loss()

model = torch.nn.Linear(1,1)

m = model.weight
b = model.bias
line_evolution_m = [m[0].clone().detach()] # A las listas no añadimos el parámetro, sino una copia
```

```

line_evolution_b = [b.clone().detach()] # de sacar del grafo usando un detach()

optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate
)

x_true = data[:,0][:,None]
y_true = data[:,1][:,None]

for step in range(steps):
    y_pred = model(x_true) # Forward
    optimizer.zero_grad()
    loss = loss_fn(y_pred, y_true)
    loss.backward() # Backward
    m = model.weight
    b = model.bias
    grad = [m.grad, b.grad]
    print(f"----- step {step}")
    print(f"m,b actual: {m,b}")
    print(f"Valor medio de y: {y_true.mean()}")
    print(f"Valor medio de y predicho: {y_pred.mean()}")
    print(f"Gradiente: {grad}")
    optimizer.step()

    line_evolution_m.append(m[0].clone().detach())
    line_evolution_b.append(b.clone().detach())

plt.figure(figsize=(8,6))
colors = plt.cm.winter(torch.linspace(0,1,steps+1))
for index, (m, b) in enumerate(zip(line_evolution_m, line_evolution_b)):
    plt.plot([0,x_true.max()], [m*0+b,m*x_true.max()+b], c=colors[index])
    plt.scatter(x_true, data[:,1], marker='o', c='r', label="data")
plt.plot([0,x_true.max()], [true_m*0+true_b,true_m*x_true.max()+true_b], c='r', label="True line")
plt.legend()
plt.grid(True)
plt.show()

```

```

Linear(in_features=1, out_features=1, bias=True)
----- step 0
m,b actual: (Parameter containing:
tensor([[0.2846]], requires_grad=True), Parameter containing:
tensor([-0.2452], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 5.584117889404297
Gradiente: [tensor([[ -8.3109]]), tensor([-0.8000])]
----- step 1
m,b actual: (Parameter containing:
tensor([[0.3029]], requires_grad=True), Parameter containing:
tensor([-0.2444], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 5.959984302520752
Gradiente: [tensor([[ -7.1880]]), tensor([-0.7200])]
----- step 2
m,b actual: (Parameter containing:
tensor([[0.3201]], requires_grad=True), Parameter containing:
tensor([-0.2437], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 6.312768459320068
Gradiente: [tensor([[ -7.1880]]), tensor([-0.7200])]
----- step 3
m,b actual: (Parameter containing:
tensor([[0.3373]], requires_grad=True), Parameter containing:
tensor([-0.2430], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 6.665553092956543
Gradiente: [tensor([[ -6.6845]]), tensor([-0.6800])]
----- step 4
m,b actual: (Parameter containing:
tensor([[0.3540]], requires_grad=True), Parameter containing:
tensor([-0.2423], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 7.007986545562744
Gradiente: [tensor([[ -5.8273]]), tensor([-0.6000])]
----- step 5
m,b actual: (Parameter containing:
tensor([[0.3698]], requires_grad=True), Parameter containing:
tensor([-0.2417], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 7.332779407501221
Gradiente: [tensor([[ -5.8273]]), tensor([-0.6000])]
----- step 6
m,b actual: (Parameter containing:
tensor([[0.3856]], requires_grad=True), Parameter containing:

```

```
tensor([-0.2411], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 7.657574653625488
Gradiente: [tensor([[-15.4961]]), tensor([-0.5600])]
----- step 7
m,b actual: (Parameter containing:
tensor([[0.4011]], requires_grad=True), Parameter containing:
tensor([-0.2405], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 7.975543022155762
Gradiente: [tensor([[-12.3938]]), tensor([-0.4800])]
----- step 8
m,b actual: (Parameter containing:
tensor([[0.4135]], requires_grad=True), Parameter containing:
tensor([-0.2401], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 8.229889869689941
Gradiente: [tensor([[-12.3938]]), tensor([-0.4800])]
----- step 9
m,b actual: (Parameter containing:
tensor([[0.4259]], requires_grad=True), Parameter containing:
tensor([-0.2396], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 8.484234809875488
Gradiente: [tensor([[-10.8748]]), tensor([-0.4400])]
----- step 10
m,b actual: (Parameter containing:
tensor([[0.4368]], requires_grad=True), Parameter containing:
tensor([-0.2391], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 8.707425117492676
Gradiente: [tensor([[-10.0273]]), tensor([-0.4000])]
----- step 11
m,b actual: (Parameter containing:
tensor([[0.4468]], requires_grad=True), Parameter containing:
tensor([-0.2387], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 8.913216590881348
Gradiente: [tensor([[-8.4914]]), tensor([-0.3200])]
----- step 12
m,b actual: (Parameter containing:
tensor([[0.4553]], requires_grad=True), Parameter containing:
tensor([-0.2384], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.087468147277832
Gradiente: [tensor([[-7.3076]]), tensor([-0.2800])]
----- step 13
m,b actual: (Parameter containing:
tensor([[0.4626]], requires_grad=True), Parameter containing:
tensor([-0.2381], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.237430572509766
Gradiente: [tensor([[-6.6297]]), tensor([-0.2400])]
----- step 14
m,b actual: (Parameter containing:
tensor([[0.4692]], requires_grad=True), Parameter containing:
tensor([-0.2379], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.373468399047852
Gradiente: [tensor([[-6.6297]]), tensor([-0.2400])]
----- step 15
m,b actual: (Parameter containing:
tensor([[0.4759]], requires_grad=True), Parameter containing:
tensor([-0.2377], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.509505271911621
Gradiente: [tensor([[-4.6737]]), tensor([-0.1600])]
----- step 16
m,b actual: (Parameter containing:
tensor([[0.4805]], requires_grad=True), Parameter containing:
tensor([-0.2375], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.605398178100586
Gradiente: [tensor([[-3.6435]]), tensor([-0.1200])]
----- step 17
m,b actual: (Parameter containing:
tensor([[0.4842]], requires_grad=True), Parameter containing:
tensor([-0.2374], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.680146217346191
Gradiente: [tensor([[-3.6435]]), tensor([-0.1200])]
----- step 18
m,b actual: (Parameter containing:
```

```
tensor([[0.4878]], requires_grad=True), Parameter containing:  
tensor([-0.2373], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.754897117614746  
Gradiente: [tensor([[-3.6435]]), tensor([-0.1200])]  
----- step 19  
m,b actual: (Parameter containing:  
tensor([[0.4915]], requires_grad=True), Parameter containing:  
tensor([-0.2371], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.829647064208984  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 20  
m,b actual: (Parameter containing:  
tensor([[0.4914]], requires_grad=True), Parameter containing:  
tensor([-0.2371], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.827424049377441  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 21  
m,b actual: (Parameter containing:  
tensor([[0.4912]], requires_grad=True), Parameter containing:  
tensor([-0.2371], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.825200080871582  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 22  
m,b actual: (Parameter containing:  
tensor([[0.4911]], requires_grad=True), Parameter containing:  
tensor([-0.2370], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.822975158691406  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 23  
m,b actual: (Parameter containing:  
tensor([[0.4910]], requires_grad=True), Parameter containing:  
tensor([-0.2370], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.820752143859863  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 24  
m,b actual: (Parameter containing:  
tensor([[0.4909]], requires_grad=True), Parameter containing:  
tensor([-0.2369], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.81852912902832  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 25  
m,b actual: (Parameter containing:  
tensor([[0.4908]], requires_grad=True), Parameter containing:  
tensor([-0.2369], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.816306114196777  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 26  
m,b actual: (Parameter containing:  
tensor([[0.4907]], requires_grad=True), Parameter containing:  
tensor([-0.2369], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.814082145690918  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 27  
m,b actual: (Parameter containing:  
tensor([[0.4906]], requires_grad=True), Parameter containing:  
tensor([-0.2368], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.811858177185059  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 28  
m,b actual: (Parameter containing:  
tensor([[0.4905]], requires_grad=True), Parameter containing:  
tensor([-0.2368], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.8096342086792  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 29  
m,b actual: (Parameter containing:  
tensor([[0.4904]], requires_grad=True), Parameter containing:  
tensor([-0.2367], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.807411193847656  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 30
```

```
m,b actual: (Parameter containing:  
tensor([[0.4903]], requires_grad=True), Parameter containing:  
tensor([-0.2367], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.805188179016113  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 31  
m,b actual: (Parameter containing:  
tensor([[0.4901]], requires_grad=True), Parameter containing:  
tensor([-0.2367], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.802964210510254  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 32  
m,b actual: (Parameter containing:  
tensor([[0.4900]], requires_grad=True), Parameter containing:  
tensor([-0.2366], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.800740242004395  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 33  
m,b actual: (Parameter containing:  
tensor([[0.4899]], requires_grad=True), Parameter containing:  
tensor([-0.2366], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.798517227172852  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 34  
m,b actual: (Parameter containing:  
tensor([[0.4898]], requires_grad=True), Parameter containing:  
tensor([-0.2365], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.796293258666992  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 35  
m,b actual: (Parameter containing:  
tensor([[0.4897]], requires_grad=True), Parameter containing:  
tensor([-0.2365], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.79407024383545  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 36  
m,b actual: (Parameter containing:  
tensor([[0.4896]], requires_grad=True), Parameter containing:  
tensor([-0.2365], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.791847229003906  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 37  
m,b actual: (Parameter containing:  
tensor([[0.4895]], requires_grad=True), Parameter containing:  
tensor([-0.2364], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.78962230682373  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 38  
m,b actual: (Parameter containing:  
tensor([[0.4894]], requires_grad=True), Parameter containing:  
tensor([-0.2364], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.787398338317871  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 39  
m,b actual: (Parameter containing:  
tensor([[0.4893]], requires_grad=True), Parameter containing:  
tensor([-0.2363], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.785175323486328  
Gradiente: [tensor([[-1.8044]]), tensor([-0.0800])]  
----- step 40  
m,b actual: (Parameter containing:  
tensor([[0.4911]], requires_grad=True), Parameter containing:  
tensor([-0.2363], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.822216987609863  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]  
----- step 41  
m,b actual: (Parameter containing:  
tensor([[0.4909]], requires_grad=True), Parameter containing:  
tensor([-0.2362], requires_grad=True))  
Valor medio de y: 9.680885314941406  
Valor medio de y predicho: 9.819993019104004  
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
```

```
----- step 42
m,b actual: (Parameter containing:
tensor([[0.4908]], requires_grad=True), Parameter containing:
tensor([-0.2362], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.817769050598145
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 43
m,b actual: (Parameter containing:
tensor([[0.4907]], requires_grad=True), Parameter containing:
tensor([-0.2361], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.815546035766602
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 44
m,b actual: (Parameter containing:
tensor([[0.4906]], requires_grad=True), Parameter containing:
tensor([-0.2361], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.813323020935059
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 45
m,b actual: (Parameter containing:
tensor([[0.4905]], requires_grad=True), Parameter containing:
tensor([-0.2361], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.8110990524292
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 46
m,b actual: (Parameter containing:
tensor([[0.4904]], requires_grad=True), Parameter containing:
tensor([-0.2360], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.80887508392334
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 47
m,b actual: (Parameter containing:
tensor([[0.4903]], requires_grad=True), Parameter containing:
tensor([-0.2360], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.806652069091797
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 48
m,b actual: (Parameter containing:
tensor([[0.4902]], requires_grad=True), Parameter containing:
tensor([-0.2359], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.804429054260254
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 49
m,b actual: (Parameter containing:
tensor([[0.4901]], requires_grad=True), Parameter containing:
tensor([-0.2359], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.802206039428711
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 50
m,b actual: (Parameter containing:
tensor([[0.4900]], requires_grad=True), Parameter containing:
tensor([-0.2359], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.799980163574219
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 51
m,b actual: (Parameter containing:
tensor([[0.4898]], requires_grad=True), Parameter containing:
tensor([-0.2358], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.797757148742676
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 52
m,b actual: (Parameter containing:
tensor([[0.4897]], requires_grad=True), Parameter containing:
tensor([-0.2358], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.79553508758545
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 53
m,b actual: (Parameter containing:
tensor([[0.4896]], requires_grad=True), Parameter containing:
tensor([-0.2357], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.793310165405273
```

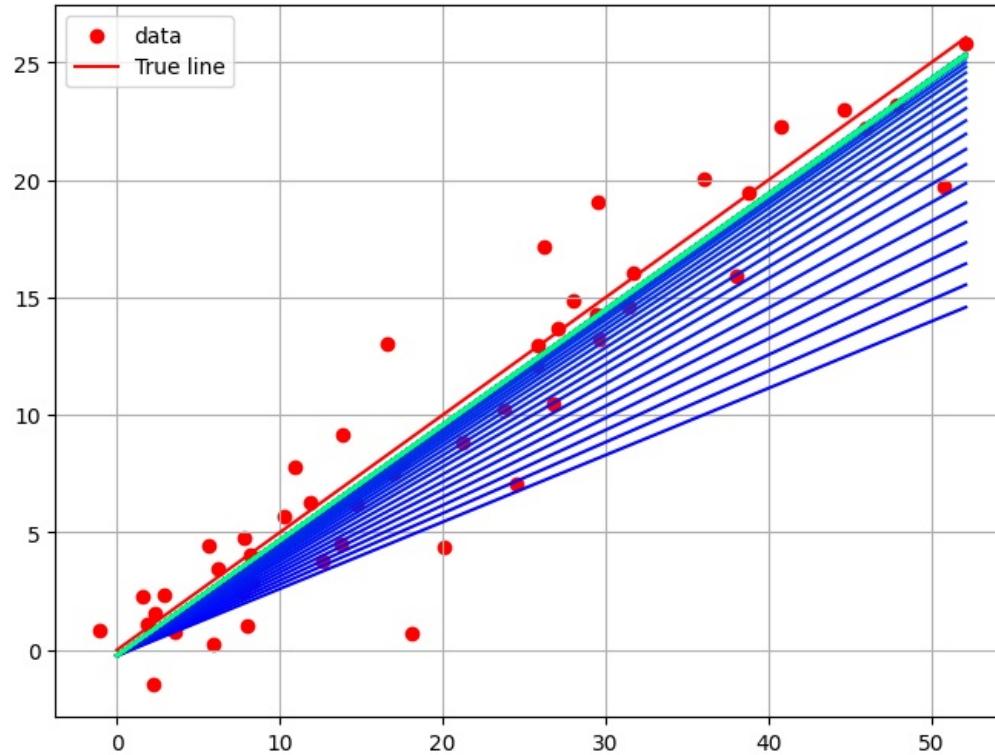
```
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 54
m,b actual: (Parameter containing:
tensor([[0.4895]], requires_grad=True), Parameter containing:
tensor([-0.2357], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.791086196899414
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 55
m,b actual: (Parameter containing:
tensor([[0.4894]], requires_grad=True), Parameter containing:
tensor([-0.2357], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.788863182067871
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 56
m,b actual: (Parameter containing:
tensor([[0.4893]], requires_grad=True), Parameter containing:
tensor([-0.2356], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.786640167236328
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 57
m,b actual: (Parameter containing:
tensor([[0.4892]], requires_grad=True), Parameter containing:
tensor([-0.2356], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.784416198730469
Gradiente: [tensor([[-1.8044]]), tensor([-0.0800])]
----- step 58
m,b actual: (Parameter containing:
tensor([[0.4910]], requires_grad=True), Parameter containing:
tensor([-0.2355], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.821457862854004
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 59
m,b actual: (Parameter containing:
tensor([[0.4909]], requires_grad=True), Parameter containing:
tensor([-0.2355], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.819232940673828
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 60
m,b actual: (Parameter containing:
tensor([[0.4908]], requires_grad=True), Parameter containing:
tensor([-0.2354], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.817008972167969
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 61
m,b actual: (Parameter containing:
tensor([[0.4907]], requires_grad=True), Parameter containing:
tensor([-0.2354], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.814785957336426
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 62
m,b actual: (Parameter containing:
tensor([[0.4905]], requires_grad=True), Parameter containing:
tensor([-0.2353], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.8125638961792
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 63
m,b actual: (Parameter containing:
tensor([[0.4904]], requires_grad=True), Parameter containing:
tensor([-0.2353], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.810338973999023
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 64
m,b actual: (Parameter containing:
tensor([[0.4903]], requires_grad=True), Parameter containing:
tensor([-0.2353], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.808115005493164
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 65
m,b actual: (Parameter containing:
tensor([[0.4902]], requires_grad=True), Parameter containing:
tensor([-0.2352], requires_grad=True))
Valor medio de y: 9.680885314941406
```

```
Valor medio de y predicho: 9.805891990661621
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 66
m,b actual: (Parameter containing:
tensor([[0.4901]], requires_grad=True), Parameter containing:
tensor([-0.2352], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.803668022155762
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 67
m,b actual: (Parameter containing:
tensor([[0.4900]], requires_grad=True), Parameter containing:
tensor([-0.2351], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.801444053649902
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 68
m,b actual: (Parameter containing:
tensor([[0.4899]], requires_grad=True), Parameter containing:
tensor([-0.2351], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.79922103881836
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 69
m,b actual: (Parameter containing:
tensor([[0.4898]], requires_grad=True), Parameter containing:
tensor([-0.2351], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.796998023986816
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 70
m,b actual: (Parameter containing:
tensor([[0.4897]], requires_grad=True), Parameter containing:
tensor([-0.2350], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.794774055480957
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 71
m,b actual: (Parameter containing:
tensor([[0.4895]], requires_grad=True), Parameter containing:
tensor([-0.2350], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.792551040649414
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 72
m,b actual: (Parameter containing:
tensor([[0.4894]], requires_grad=True), Parameter containing:
tensor([-0.2349], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.790327072143555
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 73
m,b actual: (Parameter containing:
tensor([[0.4893]], requires_grad=True), Parameter containing:
tensor([-0.2349], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.788104057312012
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 74
m,b actual: (Parameter containing:
tensor([[0.4892]], requires_grad=True), Parameter containing:
tensor([-0.2349], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.785880088806152
Gradiente: [tensor([[-1.8044]]), tensor([-0.0800])]
----- step 75
m,b actual: (Parameter containing:
tensor([[0.4910]], requires_grad=True), Parameter containing:
tensor([-0.2348], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.822920799255371
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 76
m,b actual: (Parameter containing:
tensor([[0.4909]], requires_grad=True), Parameter containing:
tensor([-0.2347], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.820697784423828
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 77
m,b actual: (Parameter containing:
tensor([[0.4908]], requires_grad=True), Parameter containing:
tensor([-0.2347], requires_grad=True))
```

```

Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.818473815917969
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 78
m,b actual: (Parameter containing:
tensor([[0.4907]], requires_grad=True), Parameter containing:
tensor([-0.2347], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.816250801086426
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]
----- step 79
m,b actual: (Parameter containing:
tensor([[0.4906]], requires_grad=True), Parameter containing:
tensor([-0.2346], requires_grad=True))
Valor medio de y: 9.680885314941406
Valor medio de y predicho: 9.814026832580566
Gradiente: [tensor([[0.1105]]), tensor([-0.0400])]

```



En esta versión de nuestro algoritmo de aprendizaje hay varios cambios. Para empezar, nuestro modelo lo inicializamos con un `torch.nn.Linear(1,1)`. Dentro de `torch.nn` están generalmente las utilidades específicas de Redes Neuronales (Neural Networks). `torch.nn.Linear()` es la clase que representa una capa de neuronas lineales. Ya hablaremos de capas más adelante. Por ahora basta entender que el primer argumento indica el tamaño de entrada (en nuestro caso 1 ya que solo hay una variable) y el segundo argumento indica el tamaño de salida (que coincide con el número de neuronas en la capa) y en nuestro caso es 1 porque queremos representar una única función lineal. Nótese que no le indicamos la función de activación. Por defecto asume que no hay. Nuestra `m` y nuestra `b` la tenemos que extraer ahora de la neurona (su único peso y su bias).

Lo segundo llamativo es que ya no tenemos explícitamente nuestra regla de aprendizaje. Lo hemos sustituido por añadir el optimizador SGD (Stochastic Gradient Descent). Un optimizador es una clase que gestiona el aprendizaje de la red. Se le indican los parámetros de cuyo aprendizaje tiene que ocuparse y el ratio de aprendizaje y cuando llega el momento de actualizar esos parámetros se le indica que ejecute un paso de aprendizaje (`.step()`). Esto enmascara y simplifica el código, además de facilitar hacer cambios rápidos alterando el algoritmo de aprendizaje. No tenemos que preocuparnos por ejecutar la actualización sin afectar el grafo de computación y llamando a `zero_grad()` los gradientes de todos los parámetros se resetearán a 0. El optimizador se encarga. Ya hablaremos más de optimizadores en el futuro.

Vale. Ya hemos visto cómo usar una neurona lineal en PyTorch y optimizarla para ajustar la recta, pero ¿qué pasa con la función $y=x^2$? ¿Permite la neurona lineal con una función de activación adecuada ajustarla?

Depende. Ninguna función de activación de las que hemos comentado permite ajustar una función con esa forma. El rango de salida de la función sigmoide o la tangente hiperbólica no es el mismo que el de $y=x^2$ así que esperar que sirva para ajustarla no tiene mucho sentido. ¿Y usando una ReLU o LeakyReLU? El rango de salida de la ReLU es el mismo que el de $y=x^2$. Ambas van de 0 a infinito. No cuesta nada probar...

```

In [3]: polynomial_coefficients = [1,0,0]
polynomial = generate_polynomial(polynomial_coefficients)
data = get_data_from_a_polynomial(polynomial, 50, sigma=0.5, range=10)

learning_rate = 1e-3

```

```

steps = 8000
loss_fn = torch.nn.L1Loss()

model = torch.nn.Sequential(
    torch.nn.Linear(1,1),
    torch.nn.ReLU(),
)

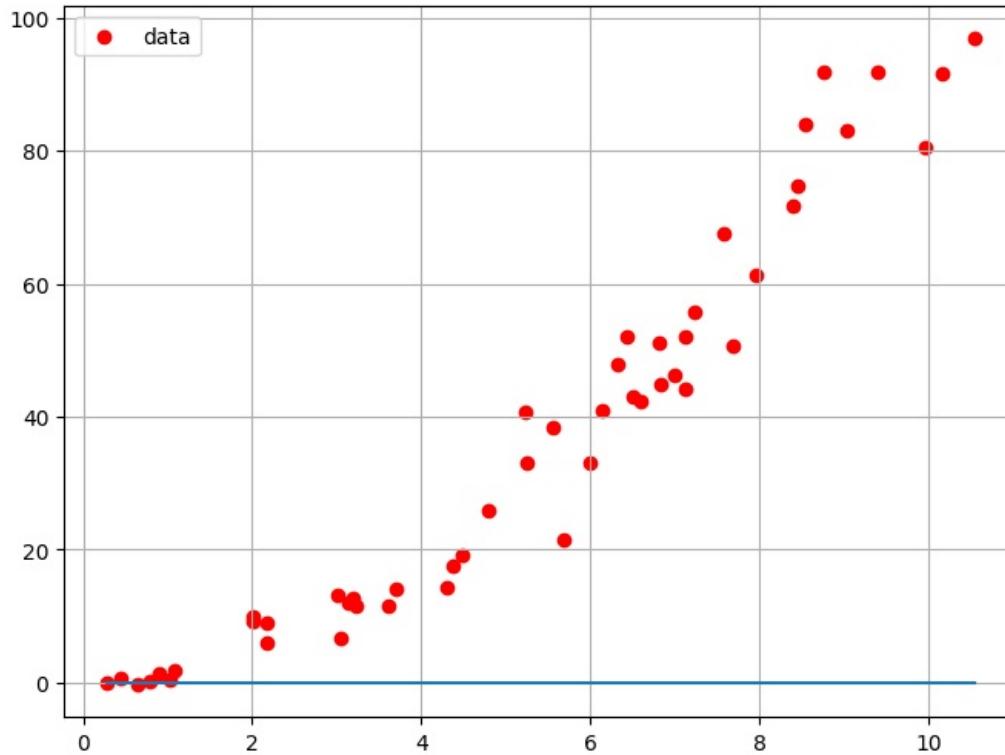
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate
)

x_true = data[:,0][:,None]
y_true = data[:,1][:,None]

for step in range(steps):
    y_pred = model(x_true) # Forward
    optimizer.zero_grad()
    loss = loss_fn(y_pred, y_true)
    loss.backward() # Backward
    optimizer.step()

plt.figure(figsize=(8,6))
x = torch.linspace(x_true.min(), x_true.max(), 100)[:,None]
y = model(x).detach()
plt.plot(x, y)
plt.scatter(data[:,0], data[:,1], marker='o', c='r', label="data")
plt.legend()
plt.grid(True)
plt.show()

```



Pues parece que la ReLU no nos soluciona el problema. De hecho.. ¡Ha aprendido una recta! ¿Y la LeakyReLU?

```

In [48]: polynomial_coefficients = [1,0,0]
polynomial = generate_polynomial(polynomial_coefficients)
data = get_data_from_a_polynomial(polynomial, 50, sigma=0.5, range=10)

learning_rate = 1e-3
steps = 8000
loss_fn = torch.nn.L1Loss()

model = torch.nn.Sequential(
    torch.nn.Linear(1,1),
    torch.nn.LeakyReLU(),
)

optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate
)

x_true = data[:,0][:,None]

```

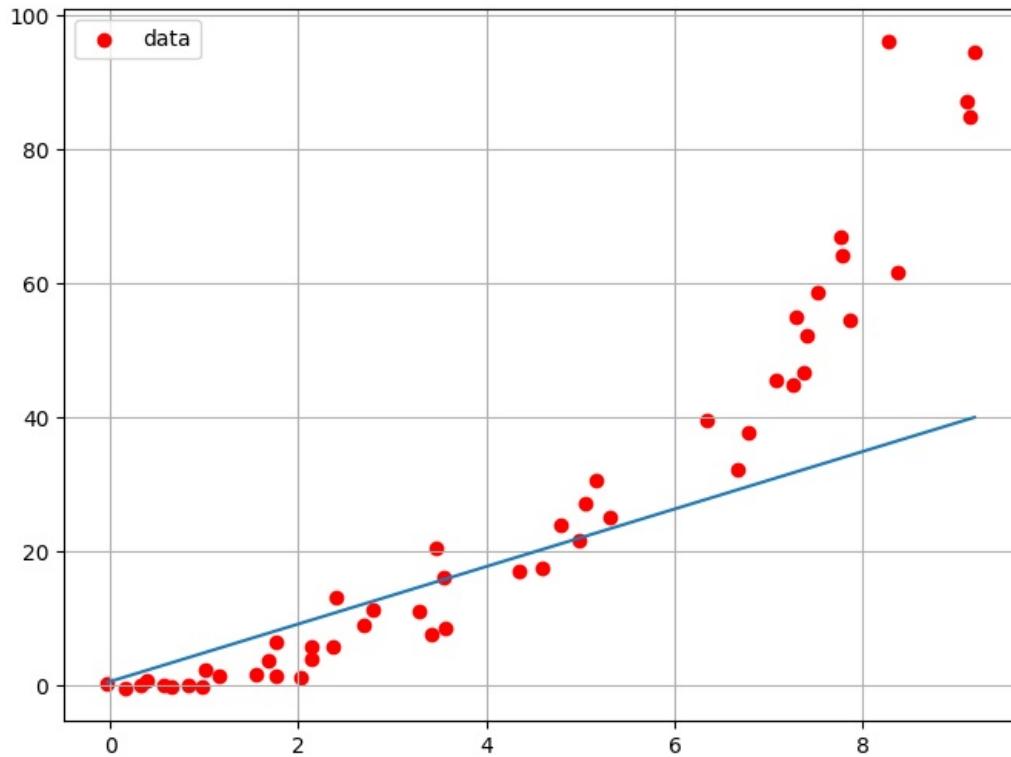
```

y_true = data[:,1][:,None]

for step in range(steps):
    y_pred = model(x_true)                                # Forward
    optimizer.zero_grad()
    loss = loss_fn(y_pred, y_true)
    loss.backward()                                       # Backward
    optimizer.step()

plt.figure(figsize=(8,6))
x = torch.linspace(x_true.min(), x_true.max(), 100)[:,None]
y = model(x).detach()
plt.plot(x, y)
plt.scatter(data[:,0], data[:,1], marker='o', c='r', label="data")
plt.legend()
plt.grid(True)
plt.show()

```



Otra dichosa recta. ¿Pero no habíamos dicho que el truco estaba en la función de activación? Una función de activación no es más que una función matemática, así que podríamos usar como función de activación $a(x)=x^2$ y así conseguir la forma que tenemos. Sería una opción, pero esa forma de razonar solo tiene sentido si sabemos que los datos se ajustan bien a $y=x^2$. ¿Cómo lo sabemos? En este caso porque hemos creado los datos artificialmente a partir de ese polinomio, pero en un caso real no sabemos a qué función se corresponden los datos. Necesitamos un modelo flexible que pueda ajustar funciones desconocidas. La flexibilidad de las redes no está en usar una única neurona, sino en usar muchas combinadas. De eso vamos a hablar en el siguiente cuaderno.

Conclusiones:

- La neurona clásica es una función lineal que necesita una función de activación no lineal para poder aprender algo que no sean rectas.
- Hay muchas funciones de activación muy variadas con distintos rangos de salida que condicionarán la misma.
- Las funciones de activación tienen unos rangos en las que son útiles, pero pueden quedar inutilizadas fuera de esos rangos porque el gradiente en esos puntos se vuelve 0 y no se pueda seguir aprendiendo.
- PyTorch ya incluye la neurona clásica implementada (entre otras).
- PyTorch enmascara los pasos del algoritmo de actualización de los pesos para simplificar el código mediante los optimizadores.
- Una sola neurona no es muy flexible.