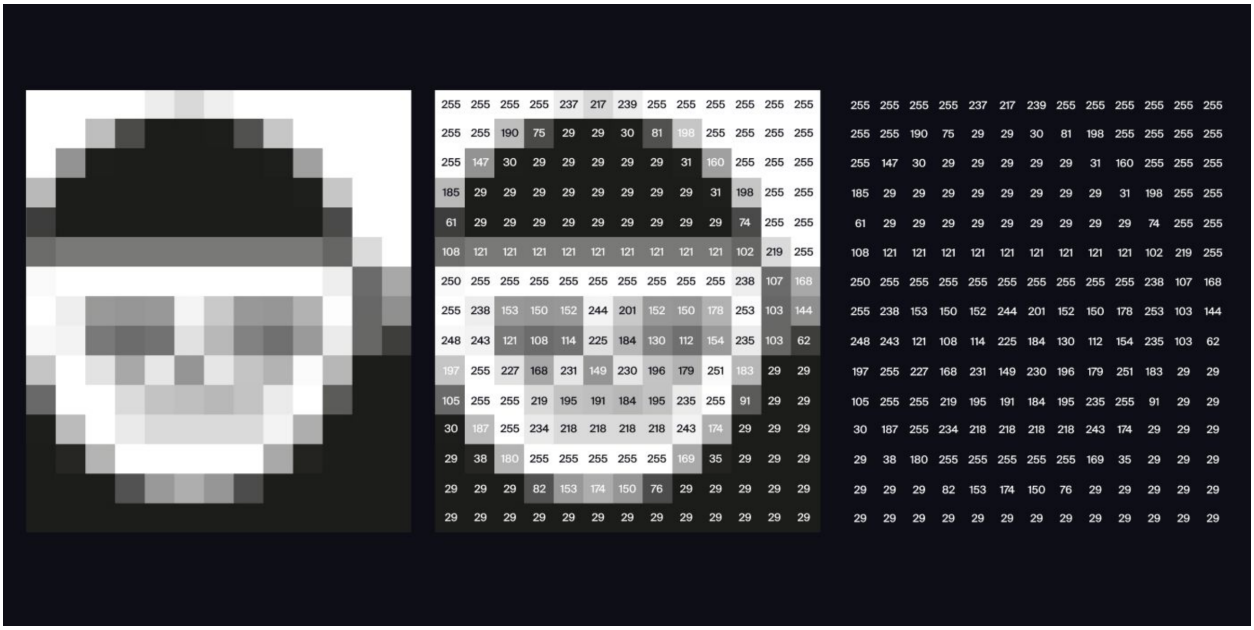
Resumen del capítulo: Redes totalmente conectadas

Tareas con redes neuronales

Supongamos que la observación es una foto y cada pixel es una característica de esa observación.

Aquí hay un ejemplo: la imagen de la izquierda es una imagen en blanco y negro de baja resolución. La del medio es la misma imagen, pero con tonos de gris de 0 (negro) a 255 (blanco). El rango de valores de brillo de pixeles es 0-255, es decir, el posible valor de característica. La imagen de la derecha contiene solo los números del rango sin la imagen.



Vamos a convertir los valores de pixel en un vector para obtener nuestras características:

```
[255, 255, 255, 255, 237, 217, 239, 255, 255, 255, 255, 255, 255, 255, 255, 190, 75, 29, 2
9, 30, 81, 198, 255, 255, 255, 255, 255, 147, 30, 29, 29, 29, 29, 29, 31, 160, 255, 255, 2
55, 185, 29, 29, 29, 29, 29, 29, 29, 29, 31, 198, 255, 255, 61, 29, 29, 29, 29, 29, 29, 2
9, 29, 29, 74, 255, 255, 108, 121, 121, 121, 121, 121, 121, 121, 121, 121, 102, 219, 255,
 250, 255, 255, 255, 255, 255, 255, 255, 255, 255, 238, 107, 168, 255, 238, 153, 150, 152,
```

```
244, 201, 152, 150, 178, 253, 103, 144, 248, 243, 121, 108, 114, 225, 184, 130, 112, 154,
 235, 103, 62, 197, 255, 227, 168, 231, 149, 230, 196, 179, 251, 183, 29, 29, 105, 255, 25
5, 219, 195, 191, 184, 195, 235, 255, 91, 29, 29, 30, 187, 255, 234, 218, 218, 218, 218, 2
43, 174, 29, 29, 29, 29, 38, 180, 255, 255, 255, 255, 255, 169, 35, 29, 29, 29, 29, 29, 2
9, 82, 153, 174, 150, 76, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29,
 29, 29]
```

Si el tamaño de cada imagen del conjunto de datos es de 1920x1080 pixeles, entonces cada imagen se describe mediante 2,073,600 características (1920 multiplicado por 1080). Los algoritmos clásicos como la potenciación del gradiente no pueden administrar el entrenamiento con tantas funciones.

Veamos qué tienen en común las imágenes y los textos:

1. Ambos tienen información redundante.
2. Las características vecinas están relacionadas entre sí.

Librería Keras

Hablemos de **Keras**, una librería de red neuronal de código abierto. De hecho, Keras es más una interfaz para trabajar con otra biblioteca más compleja, **TensorFlow.** Otra librería popular de redes neuronales es **PyTorch**, la cual ya conoces. Es posible que te haya parecido fácil trabajar con esta, ya que usaste un modelo previamente entrenado. De todas formas, tanto TensorFlow como PyTorch son difíciles para los programadores principiantes.

Vamos a escribir una regresión lineal en Keras. Una regresión lineal es una red neuronal, pero con una sola neurona:

```
# importa Keras
from tensorflow import keras

# crea el modelo
model = keras.models.Sequential()
# indica cómo está organizada la red neuronal
model.add(keras.layers.Dense(units=1, input_dim=features.shape[1]))
# indica cómo está entrenada la red neuronal
model.compile(loss='mean_squared_error', optimizer='sgd')

# entrena el modelo
model.fit(features, target)
```

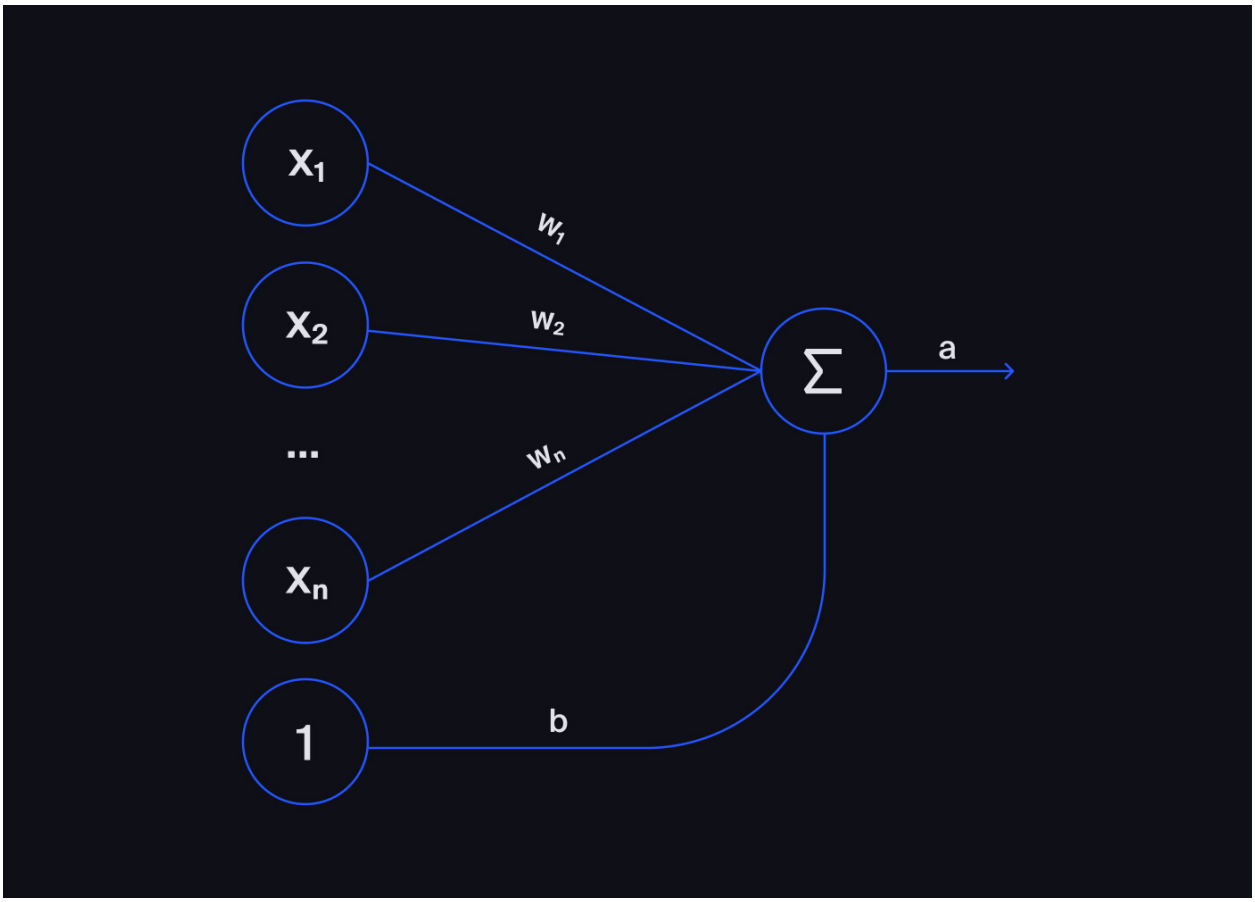
Analicemos el código línea por línea. La primera línea importa Keras desde la librería *tensorflow*. En la plataforma, usamos TensorFlow v2.1.0.

```
from tensorflow import keras
```

La siguiente línea inicializa el modelo (es decir, la red neuronal que construiremos). Establezcamos la clase del modelo en `Sequential`. Esta clase se utiliza para modelos con capas secuenciales. Una **capa** es un conjunto de neuronas que comparten la misma entrada y salida.

```
model = keras.models.Sequential()
```

Nuestra red consistirá en una sola neurona (o el valor en una salida). Tiene *n* entradas, cada una multiplicada por su propio peso. Por ejemplo, x_1 se multiplica por w_1. Hay una entrada más y siempre es igual a la unidad. Su peso se designa como **b** (sesgo). Lo que constituye el proceso de entrenamiento de la red neuronal es la selección de pesos *w* y *b*. Después de sumar todos los productos de los valores de las entradas y los pesos, la respuesta de la red neuronal (*a*) se envía a la salida.

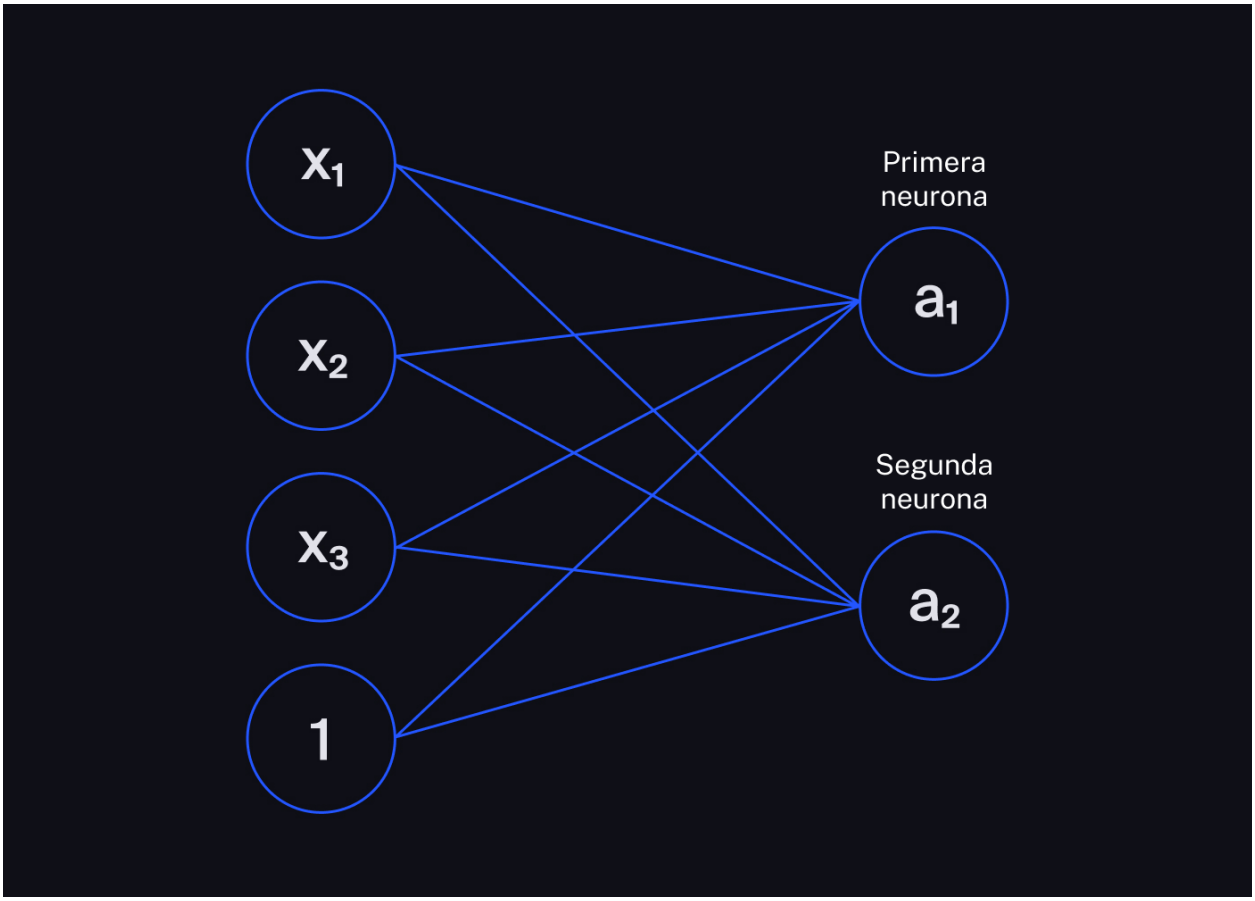


El comando `keras.layers.Dense()` crea una capa de neuronas. "Dense" (densa) significa que cada entrada estará conectada a cada neurona o salida. El parámetro `units` establece la cantidad de neuronas en la capa, mientras que `input_dim` establece la cantidad de entradas en la capa. Observa que este parámetro no considera el sesgo.

Para crear una capa para nuestra red, escribe:

```
# toma el número de entradas del conjunto de entrenamiento
keras.layers.Dense(units=1, input_dim=features.shape[1])
```

La capa totalmente conectada establecida por el comando `keras.layers.Dense(units=2, input_dim=3)` se ve así:



Para agregar la capa totalmente conectada al modelo, llama al método `model.add()`:

```
model.add(keras.layers.Dense(units=1, input_dim=features.shape[1]))
```

Echa un vistazo a esta línea:

```
 model.compile(loss='mean_squared_error', optimizer='sgd')
```

Lo que hace es preparar el modelo para el entrenamiento. Después de este comando, la estructura de la red ya no se puede cambiar. Especifica el ECM como la función de pérdida de la tarea de regresión para el parámetro `loss`. Establece el método de descenso de gradiente para el parámetro `optimizer='sgd'`. Recuerda, las redes neuronales se entrenan con SGD.

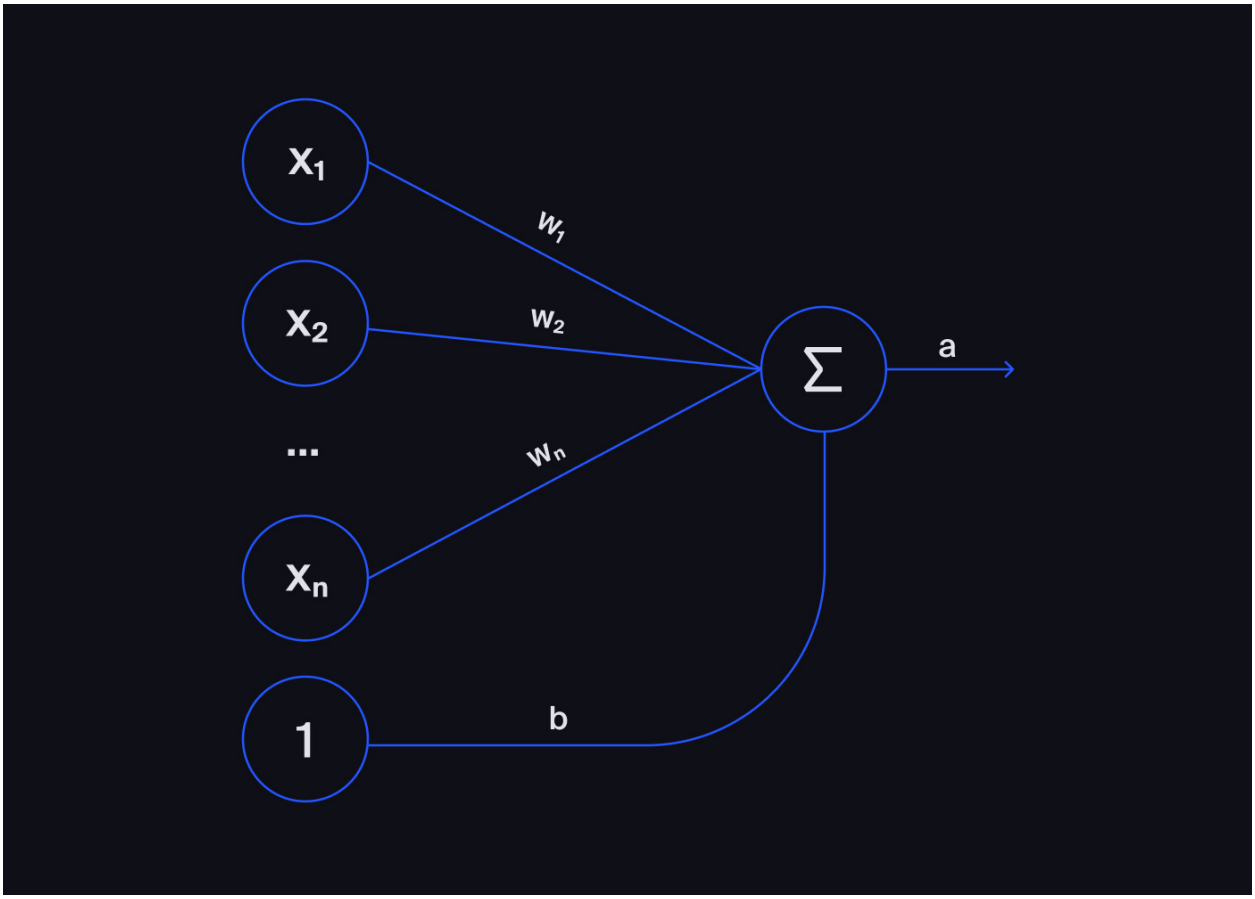
Ahora vamos a ejecutar el entrenamiento:

```
model.fit(features, target)
```

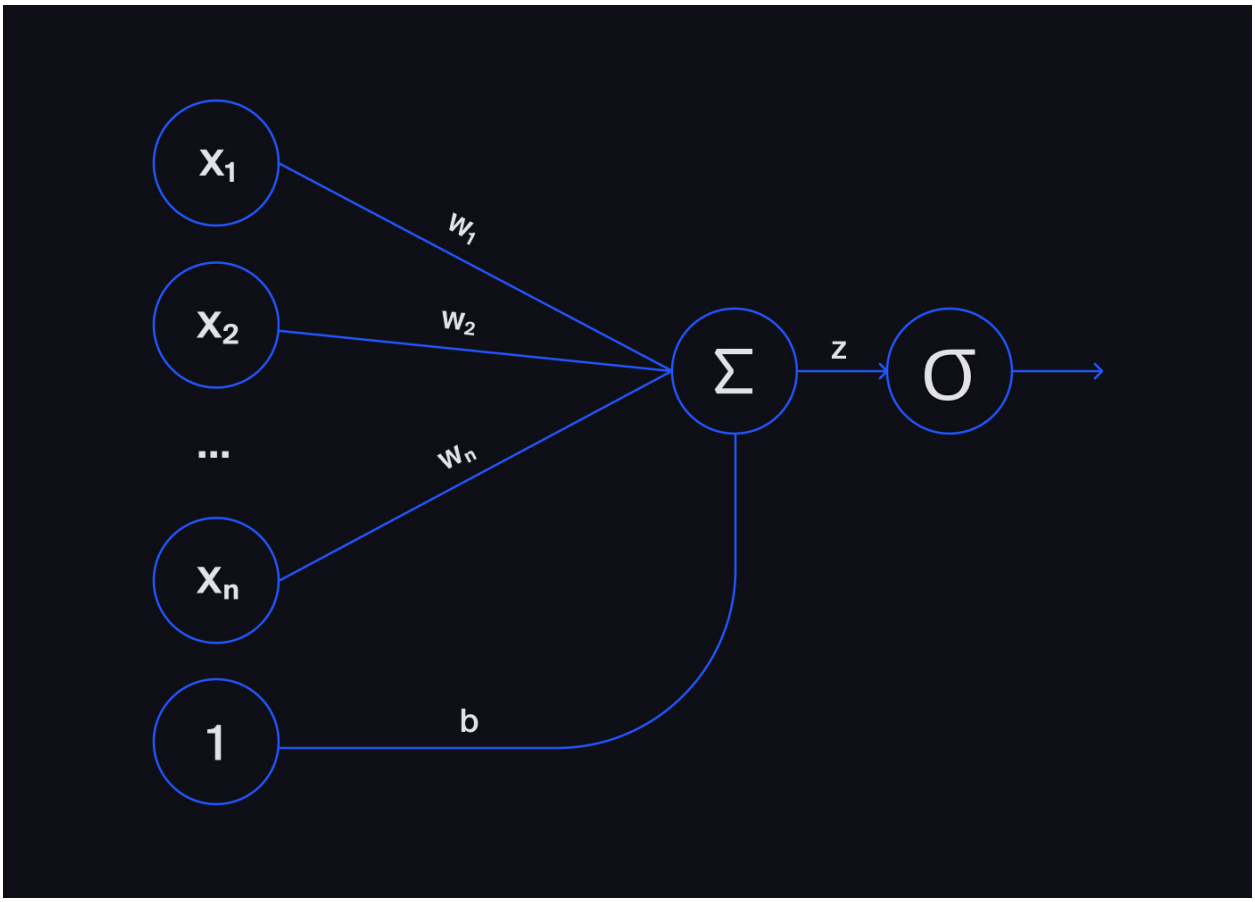
Regresión logística

La regresión lineal es una red neuronal con una sola neurona. Se puede decir lo mismo sobre la regresión logística. Si las observaciones tienen solo dos clases, la diferencia entre la regresión lineal y la regresión logística es casi imperceptible. Necesitamos agregar un elemento adicional.

Así es como se ve la regresión lineal:



Y así es como se ve la regresión logística:

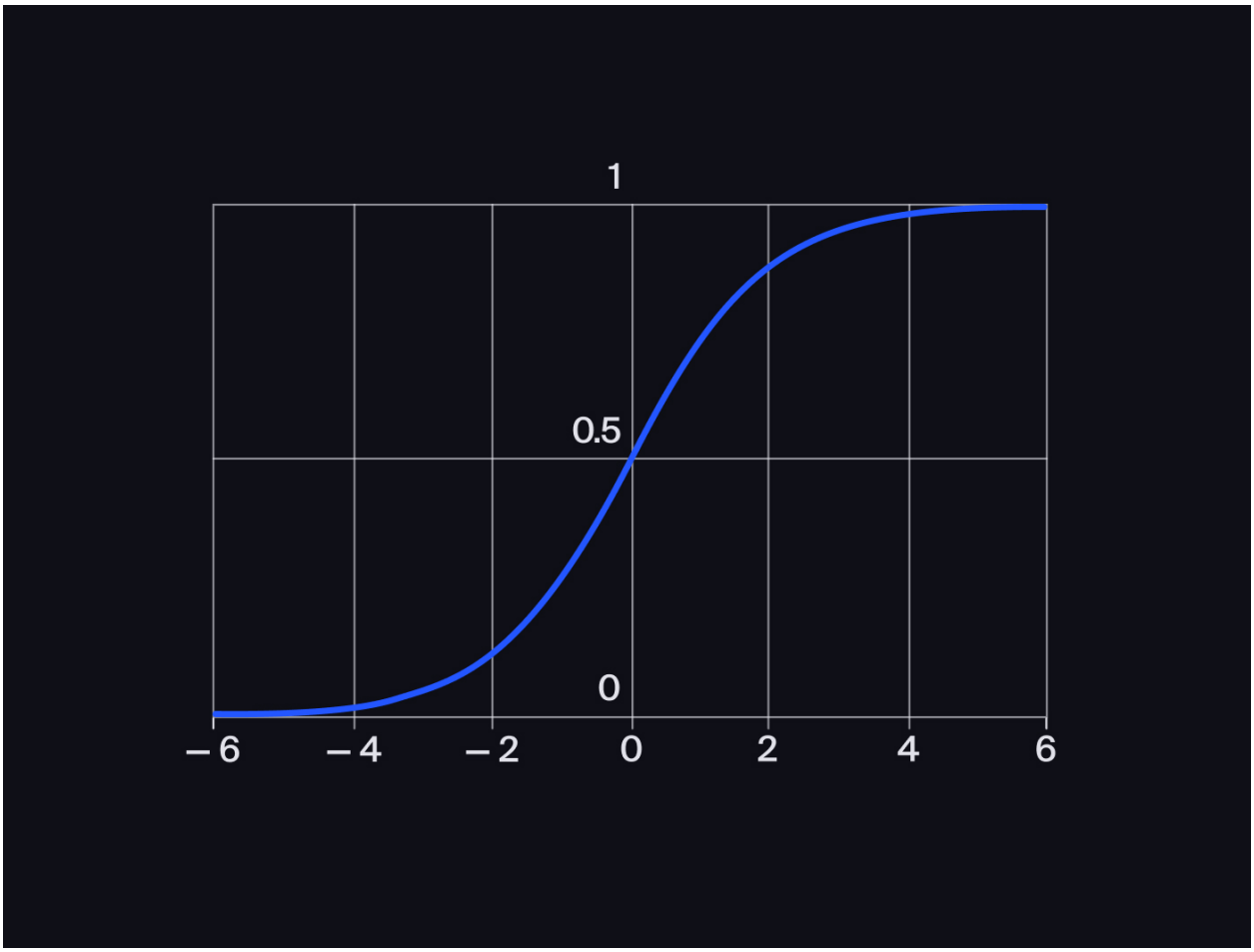


El último diagrama tiene la **función sigmoide** o función de activación, que toma cualquier número real como entrada y devuelve un número en el rango de 0 (sin activación) a 1 (activación).

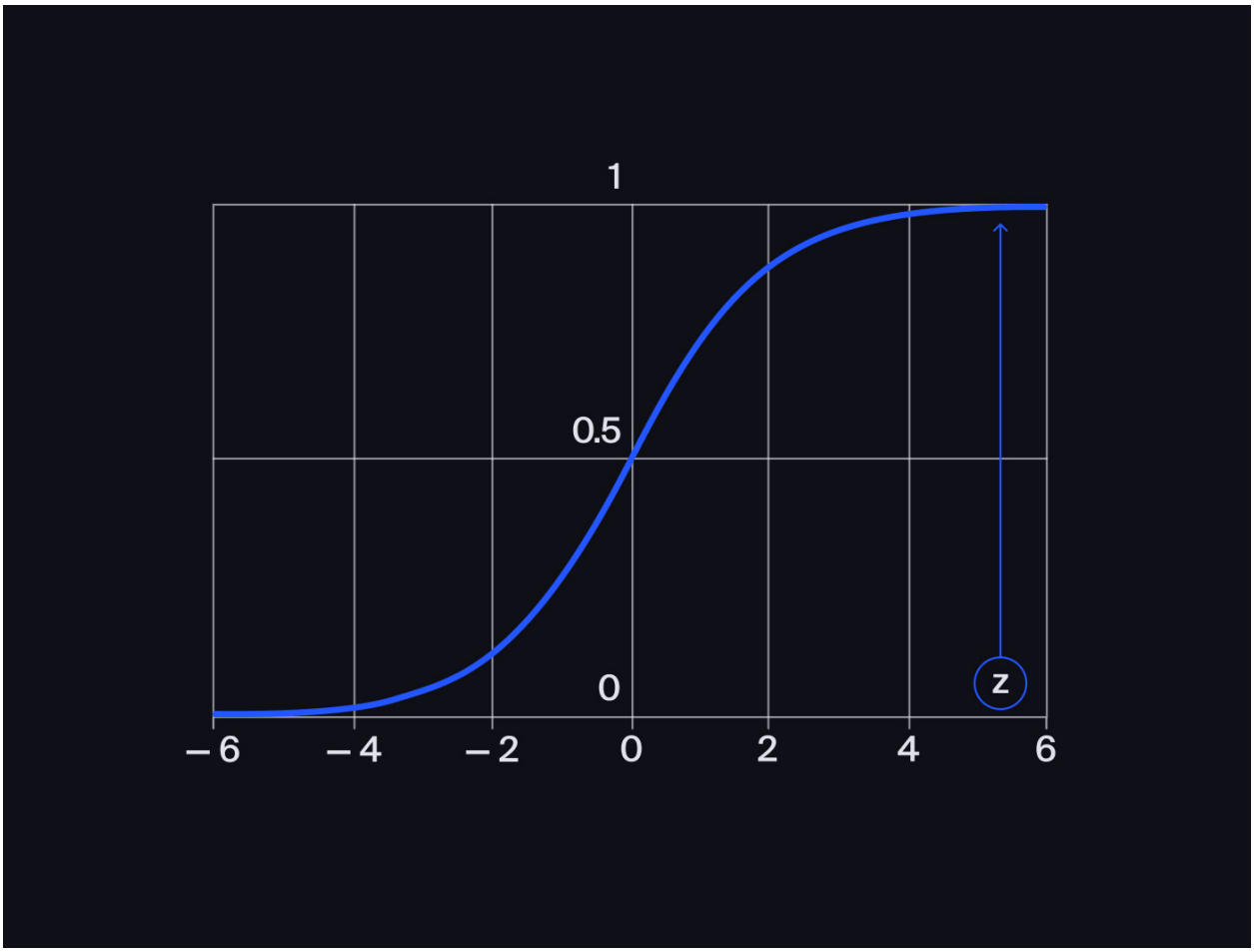
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

En la fórmula, *e* es el número de Euler, que es aproximadamente igual a 2.718281828.

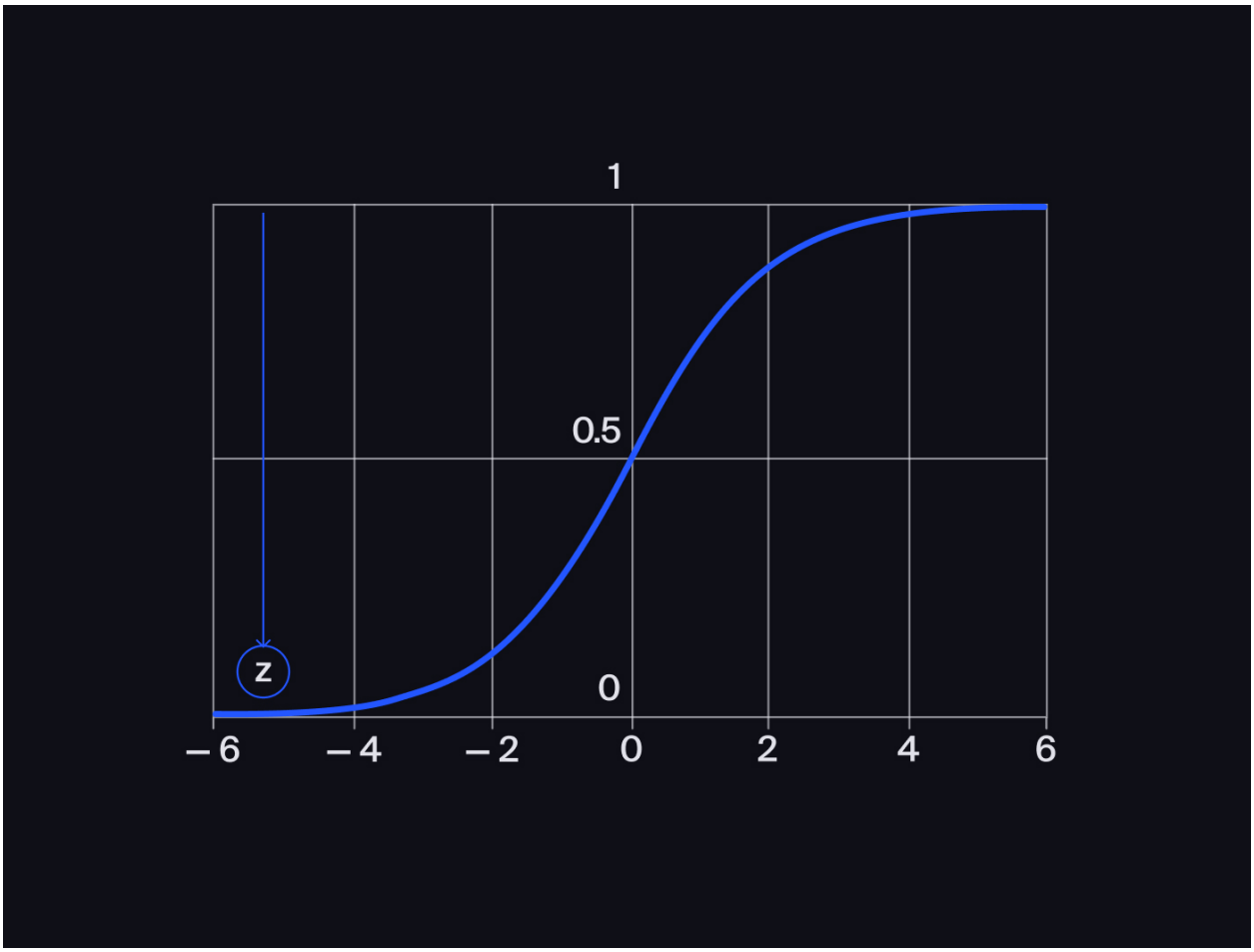
Este número en el rango de 0 a 1 se puede interpretar como una predicción de una red neuronal sobre si la observación pertenece a la clase positiva o a la clase negativa.



Si la suma de los productos de los valores de las entradas y los pesos (z) es muy grande, entonces, en la salida sigmoide, obtenemos un número cercano a la unidad:



Pero si, por el contrario, la suma es un número negativo grande, entonces la función devuelve un número cercano a cero:



La función de pérdida varía dependiendo del tipo de red neuronal. El ECM se usa en tareas de regresión, mientras que la **entropía cruzada binaria (BCE)** es la adecuada para una clasificación binaria. No podemos usar la métrica de *exactitud* porque no tiene un producto, lo cual hace que sea imposible trabajar para SGD.

La BCE se calcula de la siguiente manera:

$$BCE = -\log(p)$$

En la fórmula, p es la probabilidad de respuesta correcta. La base del logaritmo no importa porque el cambio de la base es la multiplicación de la función de pérdida por la constante, que no cambia el mínimo.

Si el objetivo = 1, entonces la probabilidad de respuesta correcta es:

$$p = \sigma(z)$$

Si el objetivo = 0, entonces p es:

$$p = (1 - \sigma(z))$$

Para comprender mejor la función BCE, observa su gráfico:

Si la probabilidad de respuesta correcta *p* es aproximadamente igual a la unidad, entonces *log(p)* es un número positivo cercano a cero. Por lo tanto, el error es pequeño. Si la probabilidad de respuesta correcta *p≈0*, entonces *log(p)* es un número positivo grande. Por lo tanto, el error también es grande.

Regresión logística en Keras

Para obtener una regresión logística, solo necesitamos cambiar el código de regresión lineal en dos lugares:

1. Aplica la función de activación a la capa totalmente conectada:

```
keras.layers.Dense(units=1, input_dim=features_train.shape[1],
                   activation='sigmoid')
```

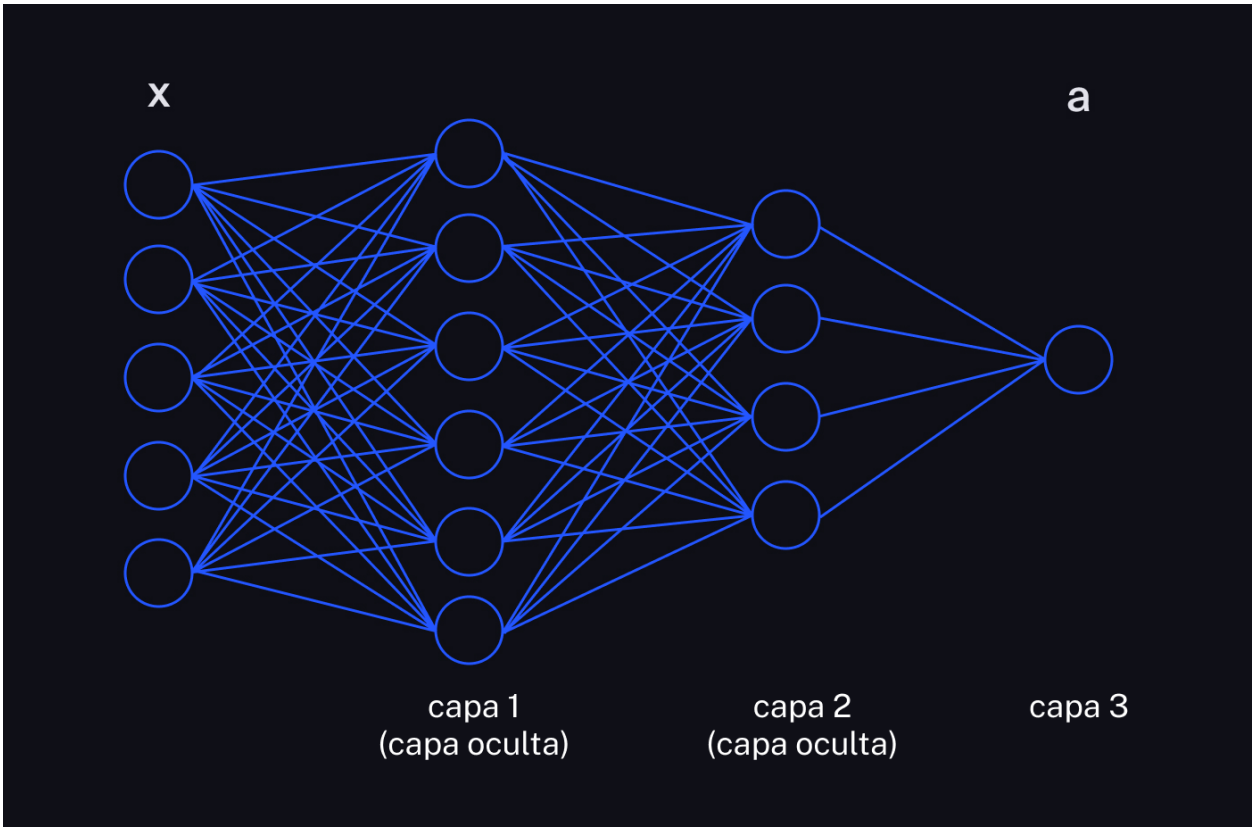
2. Cambia la función de pérdida del ECM a `binary_crossentropy`:

```
model.compile(loss='binary_crossentropy', optimizer='sgd')
```

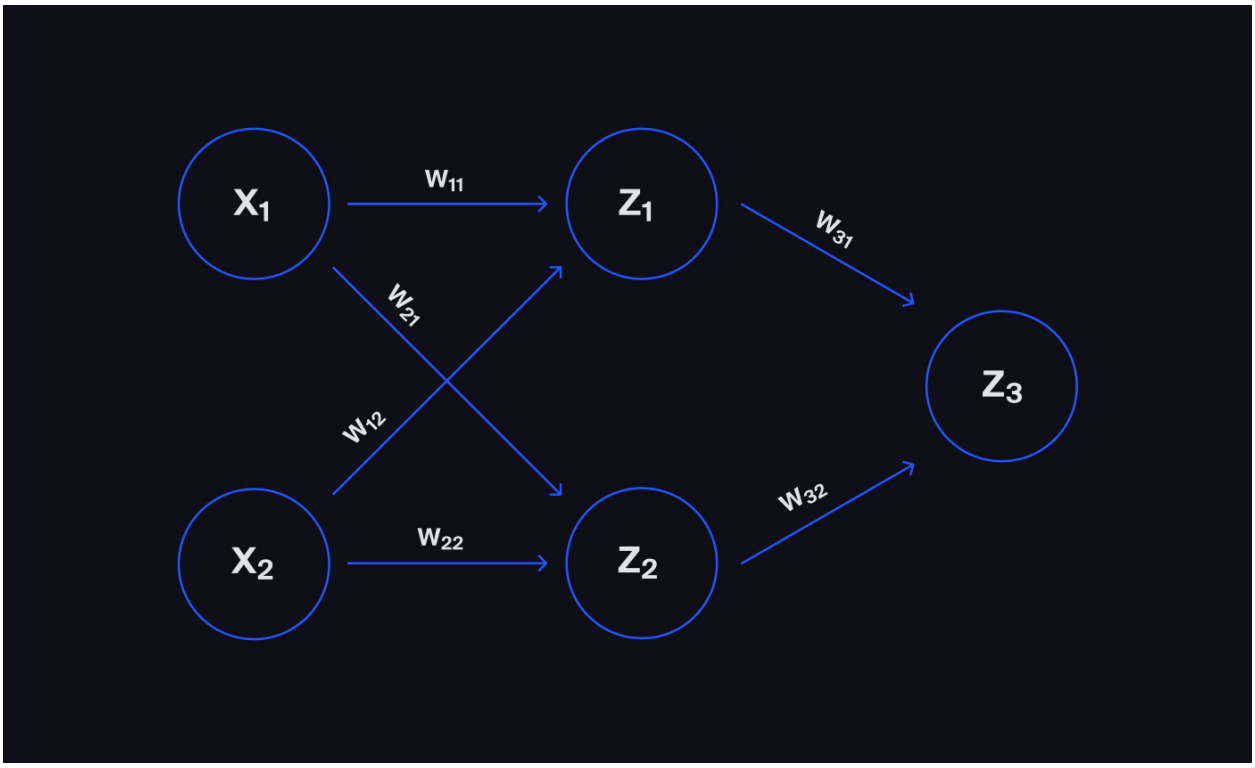
Redes neuronales totalmente conectadas

Es hora de ver más de cerca **las redes neuronales totalmente conectadas**. En estas redes, las neuronas en cada capa están conectadas con las de las capas previas.

A continuación se muestra un ejemplo de una red totalmente conectada. Debes tener en cuenta que todas las capas, excepto las capas de entrada y salida, se llaman **capas ocultas**:



Vamos a analizar la estructura de esta red totalmente conectada. Cada neurona, excepto la última, es seguida por una función de activación. ¿Te preguntas por qué? Echa un vistazo al ejemplo de dicha red sin el peso b:



Para obtener z, suma todos los productos de los valores de entrada y los pesos:

```
z1 = x1 * w11 + x2 * w12
z2 = x1 * w21 + x2 * w22
z3 = z1 * w31 + z2 * w32
```

Obtenemos:

```
z3 = (x1 * w11 + x2 * w12) * w31 + (w21 * x1 + w22 * x2) * w32
```

Saca a x_1 y x_2 de los corchetes:

```
z3 = x1 * (w11 * w31 + w21 * w32) + x2 * (w12 * w31 + w22 * w32)
```

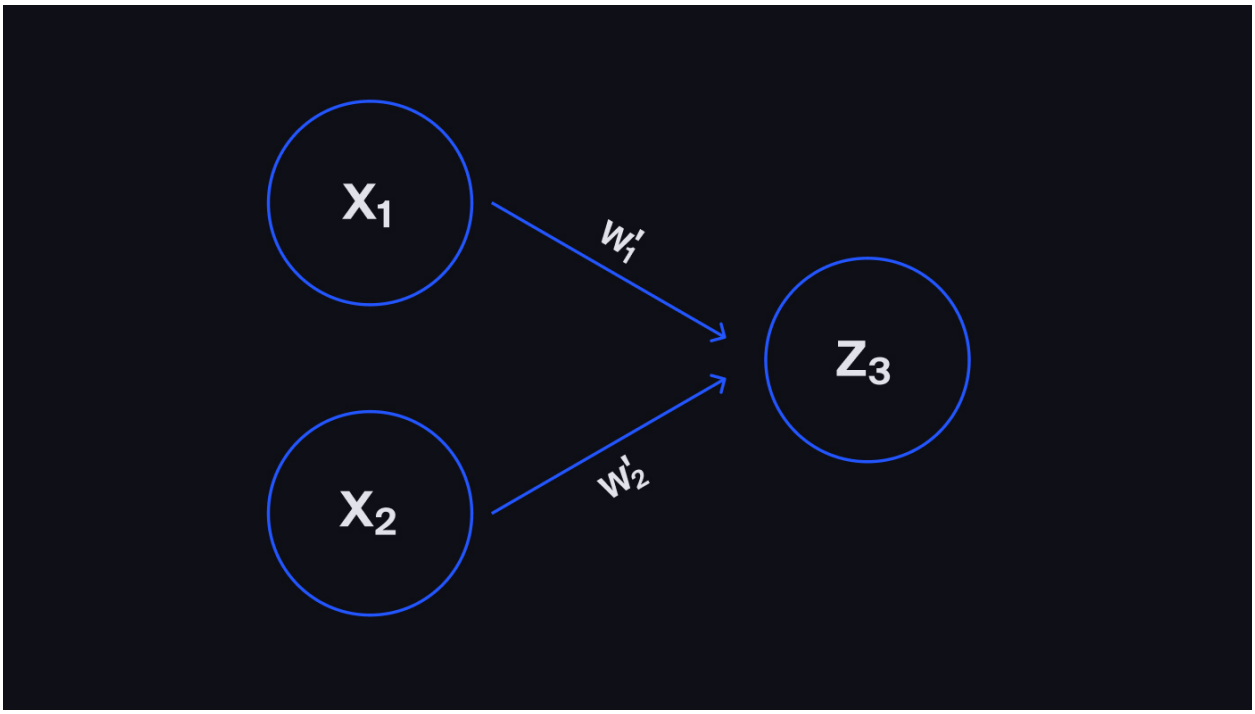
Por comodidad, usaremos una nueva notación, w_1 y w_2:

```
w1' = w11 * w31 + w21 * w32
w2' = w12 * w31 + w22 * w32
```

Obtenemos:

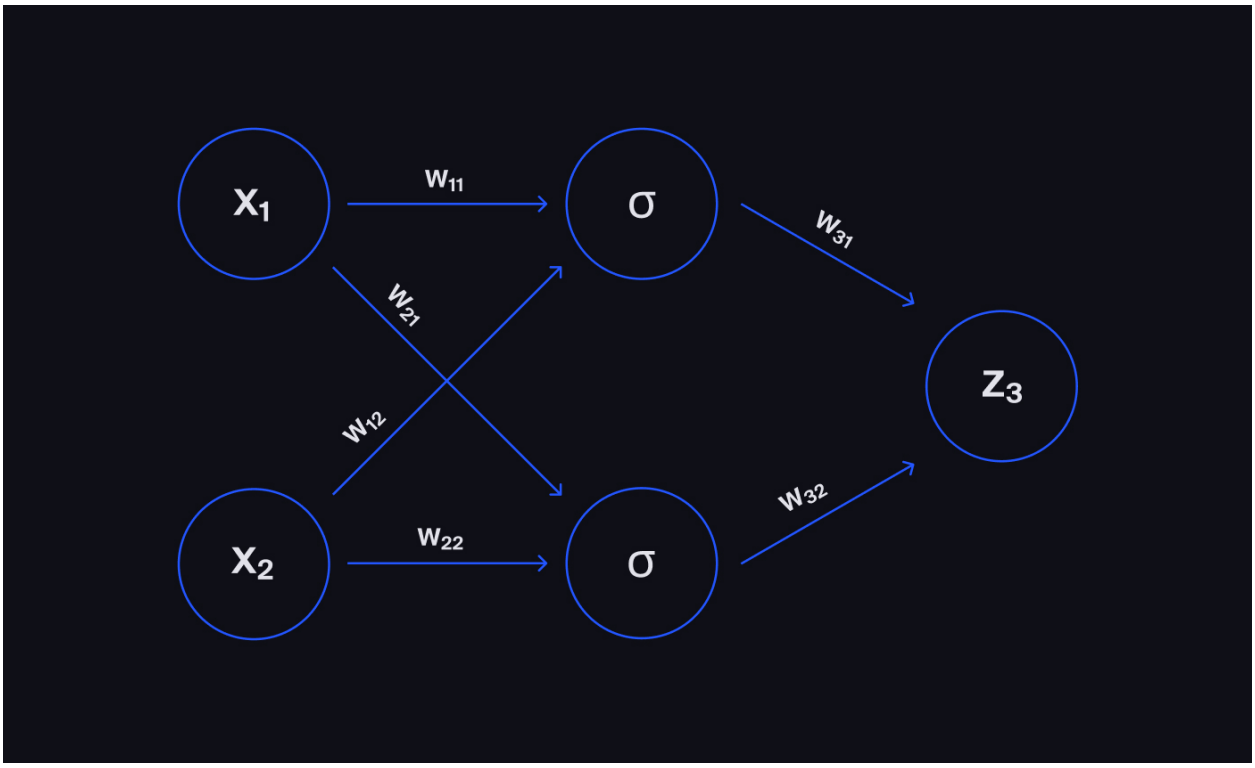
```
z3 = x1 * w1' + x2 * w2'
```

Aquí está el diagrama de esta fórmula:



¿Te parece familiar? ¡Una red multicapa es una red de una sola neurona!

Para solucionar esto, vamos a reintroducir el sigmoide y ver cómo cambia la red:



Escribe la fórmula para esto:

```
z1 = σ(x1 * w11 + x2 * w12)
z2 = σ(x1 * w21 + x2 * w22)
z3 = z1 * w31 + z2 * w32
```

Obtenemos:

```
z3 =  σ(x1 * w11 + x2 * w12) * w31 + σ(w21 * x1 + w22 * x2) * w32
```

Debido a los sigmoides, no podemos sacar a x_1 y x_2 de los corchetes. Esto significa que ahora no estamos tratando con una sola neurona. Los sigmoides permiten hacer que la red sea más compleja.

Cómo se entrenan las redes neuronales

De forma similar a la regresión lineal, las redes multicapa están entrenadas con un descenso de gradiente. Hay parámetros que son los pesos de las neuronas en cada

capa totalmente conectada. Y el objetivo de entrenamiento es encontrar los parámetros que dan como resultado la función de pérdida mínima.

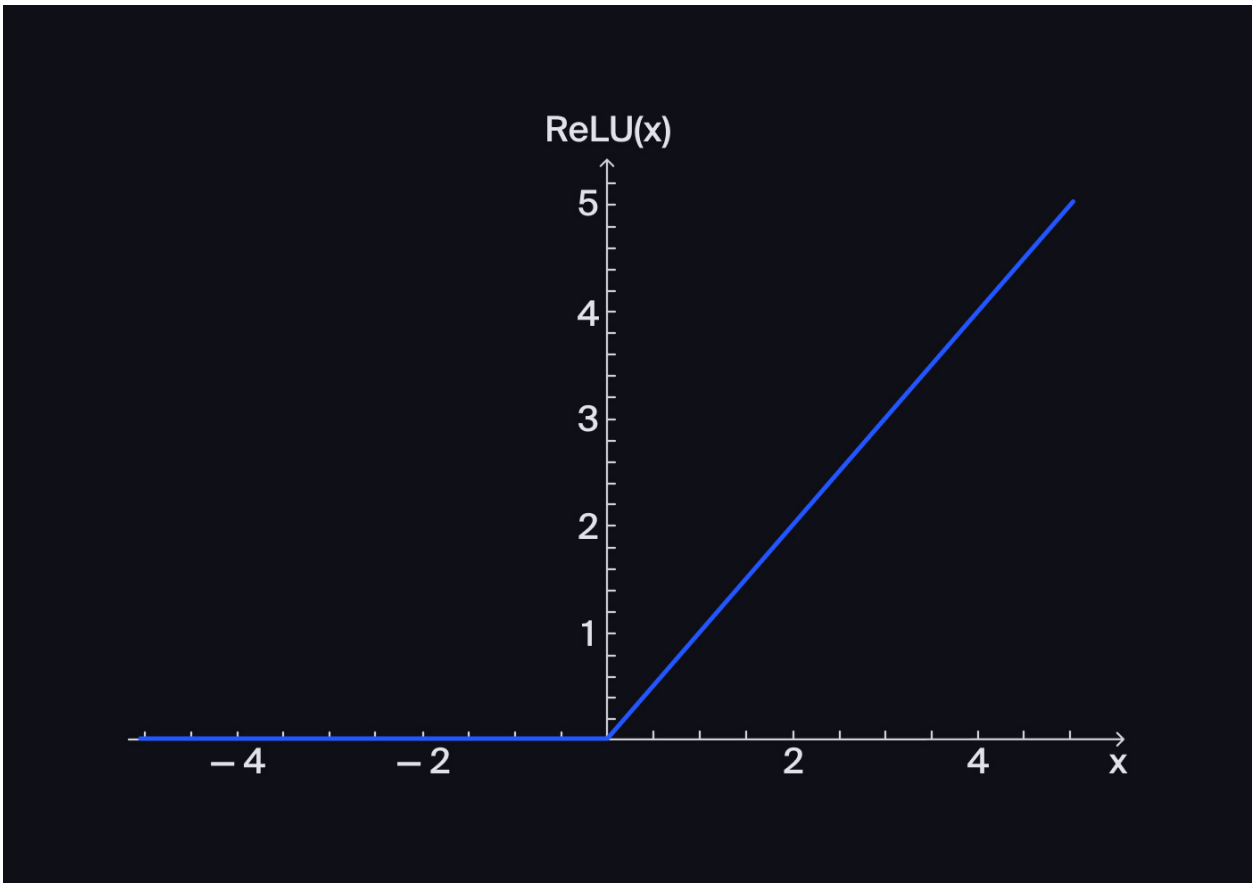
¿Cómo cambia una red neuronal cuando agregamos neuronas y capas? Para averiguarlo, resuelve algunas tareas en el sitio web **TensorFlow Playground** (materiales en inglés). Esta plataforma te permite entrenar pequeñas redes neuronales en datos de modelo con dos características para tareas de clasificación.

Puedes elegir el conjunto de datos en la parte izquierda de la interfaz *TensorFlow Playground*. En el medio tienes la estructura de tu red y en el lado derecho tienes el resultado del entrenamiento. Puedes controlar el proceso de entrenamiento del modelo desde el panel superior y cambiar el valor de la época, la tasa de aprendizaje o la función de activación.

Observa esta red. Tiene muchas capas y una función de activación sigmoide. Intenta entrenar la red. Verás que no aprenderá sin importar el valor de paso que establezcas. Vamos a averiguar por qué.

A medida que aumenta el número de capas, el entrenamiento se vuelve menos eficiente. Entre más capas haya en la red, menos señal habrá de la entrada a la salida de la red. Esto se llama **señal de fuga** y es causada por el sigmoide, que convierte grandes valores en más pequeños una y otra vez.

Para deshacerte del problema puedes intentar otra función de activación, por ejemplo, **ReLU** (*unidad lineal rectificada*). Así es como se ve:



Y aquí está la fórmula de ReLU:

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

ReLU lleva todos los valores negativos a 0 y deja los valores positivos sin cambios.

Cambia la función de activación de sigmoide a *ReLU* en esta red (materiales en inglés). Asegúrate de que ahora la red neuronal esté aprendiendo correctamente. Debido a que se cambió la función de activación, el área que separa los puntos es poligonal. La forma depende de cómo se inicializaron los pesos de la red.

Redes neuronales totalmente conectadas en Keras

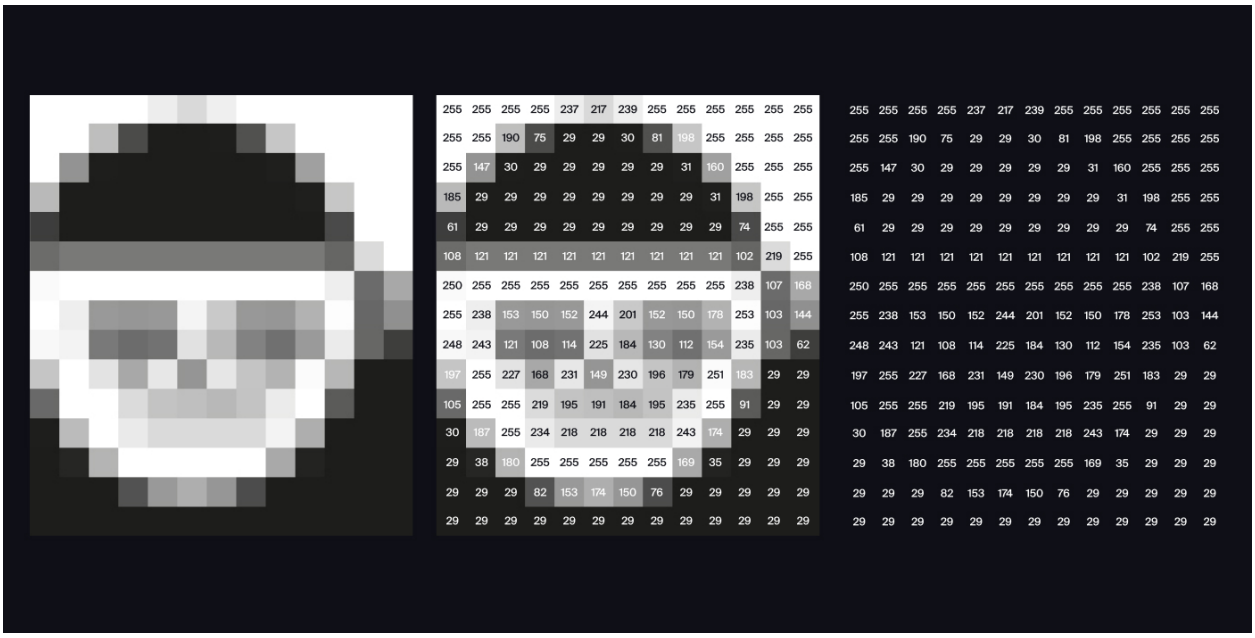
Las capas totalmente conectadas en *Keras* se pueden crear llamando a *Dense()*. Para construir una red multicapa totalmente conectada, debes agregar una capa totalmente conectada varias veces. Deja la primera capa oculta con diez neuronas y así la segunda capa de salida tendrá una neurona.

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(units=10, input_dim=features_train.shape[1],
                             activation='sigmoid'))
model.add(keras.layers.Dense(units=1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(features_train, target_train, epochs=10, verbose=2,
          validation_data=(features_valid, target_valid))
```

Trabajar con imágenes en Python

Ya sabes que las imágenes son conjuntos de números. Si la imagen es en blanco y negro, cada pixel almacena un número de 0 (negro) a 255 (blanco).



Vamos a abrir esta imagen con las herramientas de la librería **PIL** (Python Imaging Library). Podremos trabajar con ella como una matriz *NumPy*:

```
import numpy as np
from PIL import Image
```

```
image = Image.open('image.png')
image_array = np.array(image)
print(image_array)
```

```
[[255 255 255 255 237 217 239 255 255 255 255 255 255]
 [255 255 190  75  29  29  30  81 198 255 255 255 255]
 [255 147  30  29  29  29  29  29  31 160 255 255 255]
 [185  29  29  29  29  29  29  29  29  31 198 255 255]
 [ 61  29  29  29  29  29  29  29  29  29  74 255 255]
 [108 121 121 121 121 121 121 121 121 121 102 219 255]
 [250 255 255 255 255 255 255 255 255 255 238 107 168]
 [255 238 153 150 152 244 201 152 150 178 253 103 144]
 [248 243 121 108 114 225 184 130 112 154 235 103  62]
 [197 255 227 168 231 149 230 196 179 251 183  29  29]
 [105 255 255 219 195 191 184 195 235 255  91  29  29]
 [ 30 187 255 234 218 218 218 218 243 174  29  29  29]
 [ 29  38 180 255 255 255 255 255 169  35  29  29  29]
 [ 29  29  29  82 153 174 150  76  29  29  29  29  29]
 [ 29  29  29  29  29  29  29  29  29  29  29  29  29]]
```

¡Obtenemos una matriz bidimensional!

Llama a `plt.imshow()` (muestra de imagen) para trazar la imagen.

```
plt.imshow(image_array)
```

Puedes trazar la imagen en blanco y negro agregando el argumento `cmap='gray'`. Para agregar una barra de color a la imagen, debes llamar a `plt.colorbar()`.

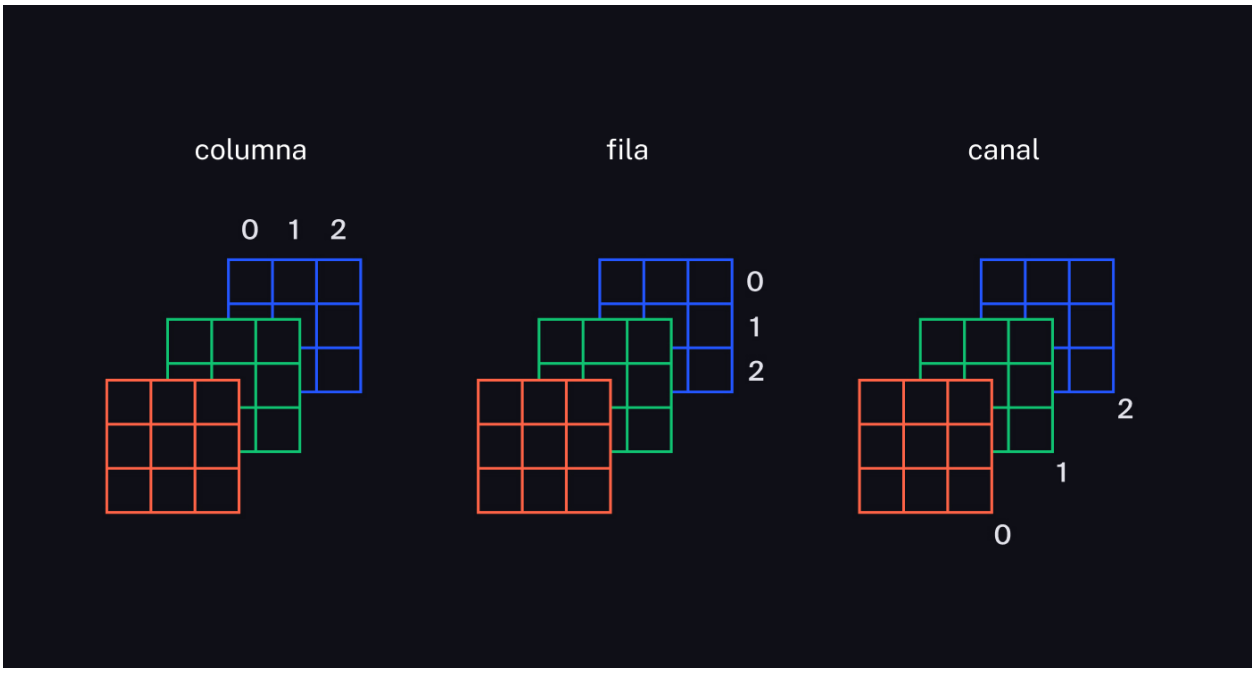
```
plt.imshow(image_array, cmap='gray')
plt.colorbar()
```

Por lo general, las redes neuronales aprenden mejor cuando reciben imágenes en el rango de 0 a 1 como entrada. Para llevar la escala [0, 255] a [0, 1], divide todos los valores de la matriz bidimensional entre 255.

```
image_array = image_array / 255.
```

Imágenes en color

Las imágenes en color o **imágenes RGB** consisten en tres canales: **rojo**, **verde** y **azul**. De hecho, estas imágenes son matrices tridimensionales con celdas que contienen números enteros en el rango de 0 a 255.



En *NumPy*, las matrices tridimensionales trabajan de la misma forma que las bidimensionales.

Compara cómo se crean:

```
np.array([[0, 255],
          [255, 0]])
```

```
np.array([[[0, 255, 0], [128, 0, 255]],
          [[12, 89, 0], [5,  89, 245]]])
```

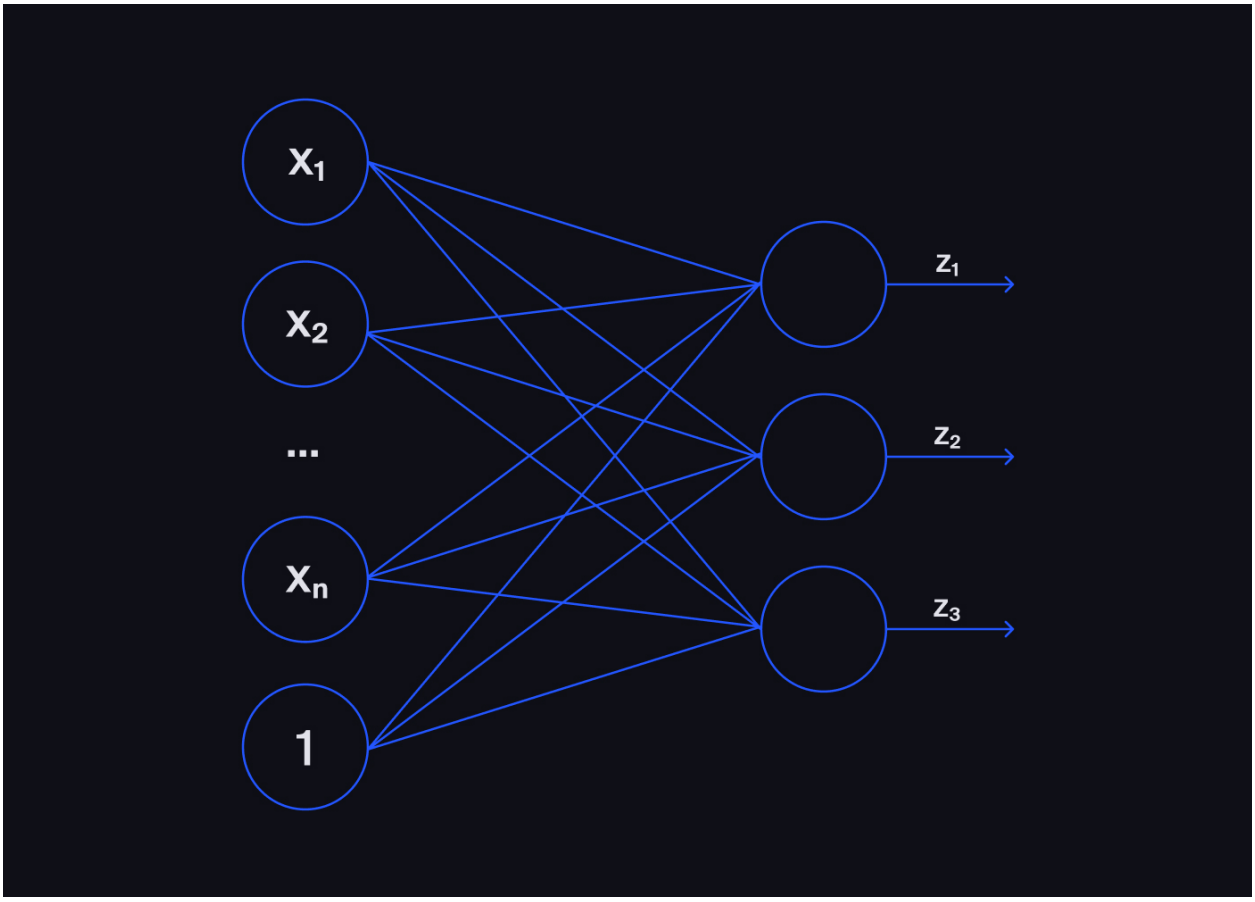
En una matriz tridimensional obtenida de una imagen, todo es casi igual. La primera coordenada es la ID de fila y la segunda es la ID de columna. Sin embargo, aquí también tenemos una nueva tercera coordenada, la cual indica el canal.

Entonces, una matriz tridimensional es como una matriz bidimensional de una imagen en blanco y negro. La única diferencia es que cada pixel de dicha matriz almacena tres números que representan el brillo de cada uno de los tres canales: rojo, verde y azul.

Clasificación multiclase

Vamos a explorar la **clasificación multiclase** y ver cómo funciona. Esta clasificación implica que las observaciones pertenecen a una de varias clases en vez de a una de dos clases.

Supongamos que tenemos tres clases. Aquí está la regresión logística representada como una red neuronal:



Obtenemos una red totalmente conectada con la capa de salida que contiene no una, sino tres neuronas. Cada neurona es responsable de su propia clase. Si el valor en la salida *z1* es un número positivo grande, la red establecerá la observación en la clase "1".

¿Cómo calculamos la función de pérdida? Recuerda la entropía cruzada binaria:

$$BCE = -\log(p)$$

Si el valor objetivo es 1, entonces la probabilidad de respuesta correcta es:

$$p = \sigma(z)$$

Si el valor objetivo es 0, el valor p es:

$$p = (1 - \sigma(z))$$

Tenemos tres clases, pero esto no afectará el cálculo de la función de pérdida. Y se llamará **CE** (entropía cruzada):

$$CE = -\log(p)$$

En la fórmula, p es la probabilidad de respuesta correcta devuelta por la red.

¿De dónde sacamos la probabilidad? Antes los sigmoides la obtenían. ¿Qué pasa si ponemos un sigmoide después de cada neurona en la capa de salida?

```
p_1 (first class probability) = σ(z_1)
p_2 (second class probability) = σ(z_2)
p_3 (third class probability) = σ(z_3)
```

Todas las probabilidades están en el rango de 0 a 1, pero la suma de las tres no necesariamente es igual a la unidad. Si suponemos que la observación pertenece a una sola clase, esperamos obtener esto:

```
p_1 + p_2 + p_3 = 1
```

La función de activación que se adapta a este caso se llama **SoftMax**, que toma varias salidas de las redes y devuelve probabilidades que son todas iguales a uno.

$$\text{SoftMax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Así es como calculamos las probabilidades:

```
p1 = SoftMax(z1) = e^z1 / (e^z1 + e^z2 + e^z3)
p2 = SoftMax(z2) = e^z2 / (e^z1 + e^z2 + e^z3)
p3 = SoftMax(z3) = e^z3 / (e^z1 + e^z2 + e^z3)
```

Ahora p_1 varía en el rango de 0 a 1.

Ten en cuenta que si z_1 es significativamente mayor que z_2 y z_3, entonces en la fórmula `e^z1 / (e^z1 + e^z2 + e^z3)` el numerador es aproximadamente igual al denominador, es decir, $p_1 \approx 1$.

Si z_1 es significativamente menor que z_2 o z_3, entonces en la fórmula `e^z1 / (e^z1 + e^z2 + e^z3)` el numerador es mucho menor que el denominador, es decir, $p_1 \approx 0$.

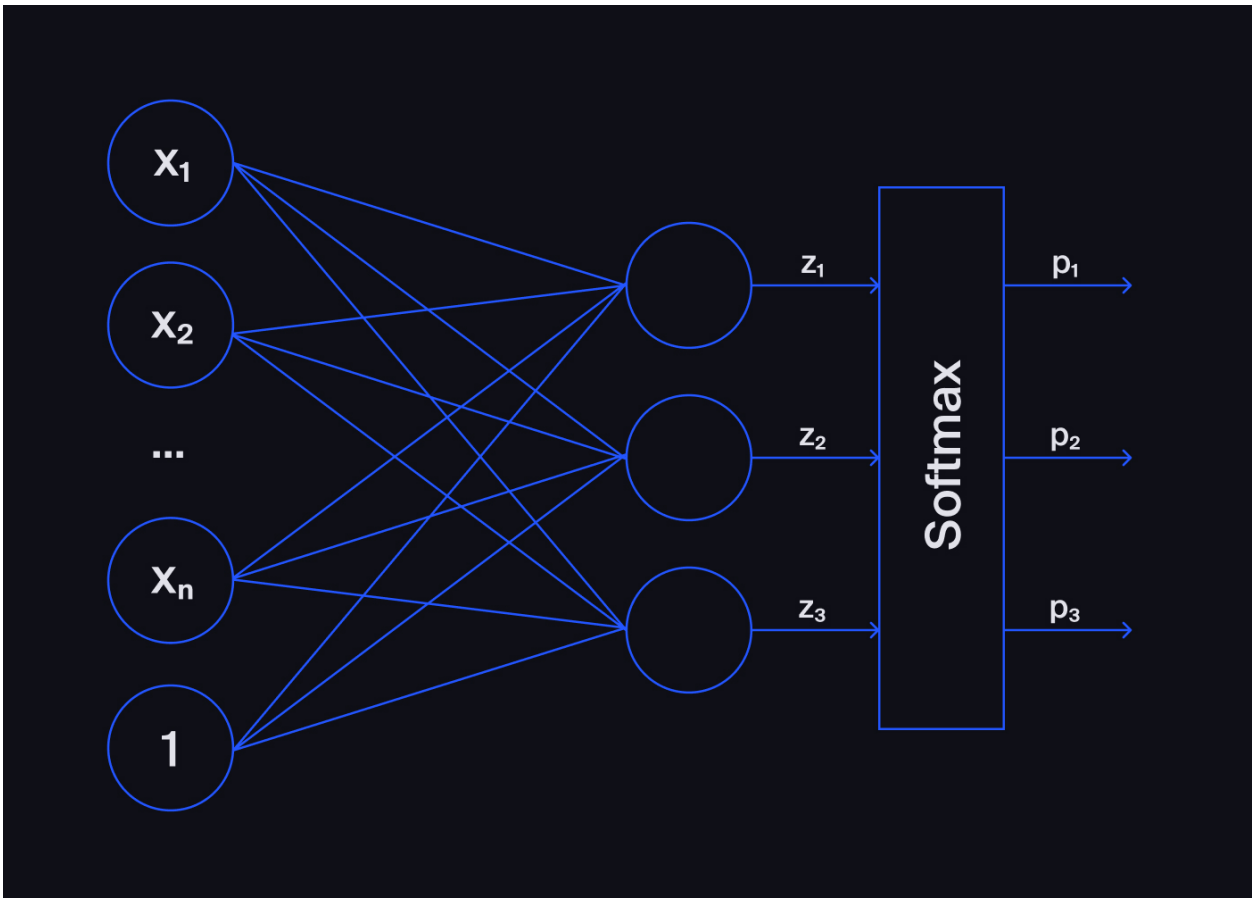
Ahora las probabilidades son iguales a uno:

```
p1 + p2 + p3 = SoftMax(z1) + SoftMax(z2) + SoftMax(z3) =

= e^z1 / (e^z1 + e^z2 + e^z3) + e^z2 / (e^z1 + e^z2 + e^z3) + e^z3 / (e^z1 + e^z2 + e^z3)
 =

= (e^z1 + e^z2 + e^z3) / (e^z1 + e^z2 + e^z3) = 1
```

Aquí está el diagrama de nuestras redes neuronales con la función de activación *SoftMax*:



¿Por qué el bloque *SoftMax* del diagrama depende de todas las salidas de la red? Porque realmente necesitamos todos los resultados para encontrar todas las probabilidades.

Si hay más de tres clases, el número de neuronas en la capa de salida debe ser igual al número de clases, y todas las salidas deben estar conectadas a *SoftMax*.

Las probabilidades de *SoftMax* en la etapa de entrenamiento pasarán a la entropía cruzada, la cual calculará el error. La función de pérdida se minimizará utilizando el método de descenso de gradiente. La única condición para que funcione el descenso de gradiente es que la función tenga una derivada para todos los parámetros: pesos y sesgo de la red neuronal.

Veamos cómo cambia el código. Esta es la inicialización de la última capa para la clasificación binaria:

```
Dense(units=1, activation='sigmoid'))
```

Y esta es la inicialización para la clasificación multiclase:

```
Dense(units=3, activation='softmax'))
```