Sistemas Inteligentes

Práctica 2

Visión Artificial y Aprendizaje

Noel Martínez Pomares
48771960T

RECUPERACIÓN DE LA PRÁCTICA

Índice

1. Parte I -> Aprende las bases de un MLP 2
1.1. Resolviendo una función booleana mediante un "Multi Layer Perceptron" 2
1.1.1. Diseñando una red MLP con funciones de activación sigmoidea 3
1.1.2. Implementando en python una función y = forward((a,b,c,d)) 5
1.2. Modelando, entrenando y probando la red en Keras 6
1.3. Analizando el entrenamiento y comparando con la red ajustada a mano 7
1.4. Aplica backpropagation manualmente 10

2. Parte II -> Entrena un MLP mediante Deep Learning usando Keras 31
2.1. Procesamiento de los datos 31
2.2. Implementando la red en keras 32
2.3. Probando el modelo 35
2.4. Mejorando la red 38

3. Referencias bibliográficas 47

1. Parte I -> Aprende las bases de un MLP

1.1. Resolviendo una función booleana mediante un "Multi Layer Perceptron"

Para la creación del MLP se ha proporcionado a cada estudiante una función booleana, distintas todas entre sí. Mi función booleana asignada es la siguiente:

$$(\bar{a} \wedge b \wedge c \wedge \bar{d}) \vee (\bar{a} \wedge b \wedge \bar{c} \wedge \bar{d}) \vee (\bar{a} \wedge \bar{b} \wedge \bar{c} \wedge d) \vee (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge b \wedge \bar{c} \wedge d) \vee (a \wedge b \wedge \bar{c})$$

La función booleana nos permite saber cuál debe ser la salida esperada por nuestra red neuronal; esta salida se puede saber gracias a la tabla de la verdad que nos proporciona la función. La tabla de la verdad correspondiente a mi función booleana es la siguiente:

Terminos				Neurona 1	Neurona 2	Neurona 3	Neurona 4	Neurona 5	Neurona 6	
a	b	c	d	(-a ^ b ^ c ^ -d)	(-a ^ b ^ -c ^ -d)	(-a ^ -b ^ -c ^ d)	(-a ^ b ^ -c)	(a ^ b ^ -c ^ d)	(a ^ b ^ -c)	OR de las 6 neuronas
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0
0	1	0	0	0	1	0	1	0	0	1
0	1	0	1	0	0	0	1	0	0	1
0	1	1	0	1	0	0	0	0	0	1
0	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	1	1
1	1	0	1	0	0	0	0	1	1	1
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

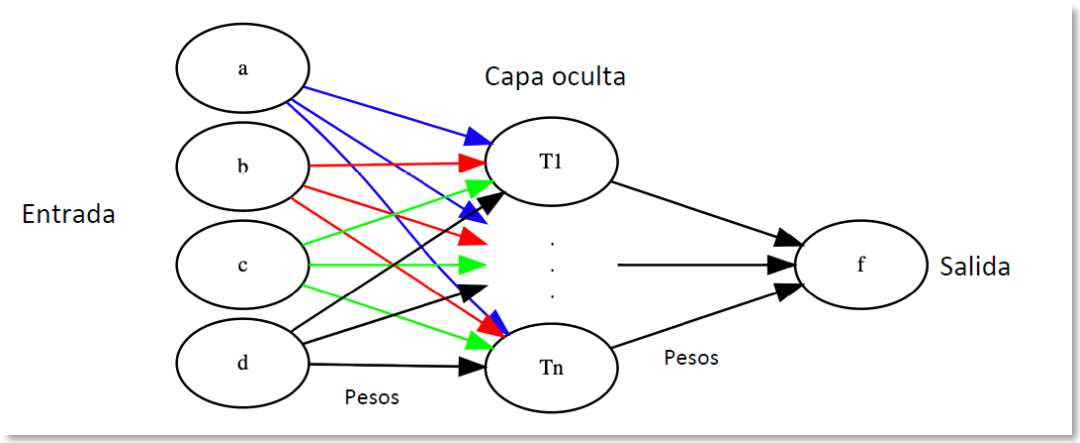
Sabemos que las entradas a, b, c, d deben tomar valores entre 0 y 1; esto implica tener 16 entradas distintas (2^4).

Los T1…Tn, de la función booleana, son el número de neuronas que hay en cada capa oculta; en mi caso se me han asignado 6 términos diferentes por lo que las capas ocultas tendrán 6 neuronas.

La capa de entrada debe tener 4 neuronas (los términos a, b, c, d) y la capa de salida debe tener una sola neurona pues solo puede ser 1 o 0.

Se deben preparar las matrices de pesos de la entrada para las capas y la matriz de los "bias".

El diagrama de la red es el siguiente:



1.1.1. Diseñando una red MLP con funciones de activación sigmoidea

Para que la red neuronal imite el comportamiento de la función booleana, la neurona f necesita activarse si cualquiera de las capas ocultas lo hace (imitando el comportamiento de los OR).

Dada la función de activación $a_f = \sigma (W_f * a^{(1)} + b_f)$; la activación de la neurona f depende de la multiplicación de los pesos por los valores de activación de la capa anterior, además sumando el bias y aplicando la función sigmoidea al resultado.

La función sigmoidea devuelve valores cercanos a 0 para los valores menores a -5 y devuelve valores cercanos a 1 para los valores mayores de 5. Si la neurona no se debe activar, debemos conseguir que la evaluación de la función de activación devuelva un valor cercano a -5 o inferior, y un valor cercano o superior a 5 si debe activarse.

Para ajustar los pesos debemos usar la formula sigmoidea (w1x1+w2x2+w3x3+w4x4+b).

$$(\overline{a} \wedge b \wedge c \wedge \overline{d})$$

Esta primera neurona se activará cuando a=0, b=1, c=1 y d=0. Para conseguir esto, le doy un valor negativo muy bajo a "a" y "d", para que cuando se activen resten lo suficiente para dejar la neurona lo más negativa posible; les doy un valor de -80. Para "b" y "c" les doy un valor de 20.

La neurona se activa cuando sea 0110; si usamos la formula sigmoidea tendríamos:

(0*-80 + 1*20 + 1*20 + 0*-80 + bias); que si se resuelve se queda -> (40+bias). Como queremos que se active (tener un valor cercano a 1), el bias será -30: así seria 40-30 -> 10.

Voy a usar esta metodología para calcular los pesos de las siguientes neuronas, excepto la 4 y la 6, que voy a usar otra metodología.

$$(\overline{a} \wedge b \wedge \overline{c} \wedge \overline{d})$$

Para que se active debe ser 0100, por lo que "a", "c" y "d" tendrán el valor -80, mientras que "b" tendrá el valor 20, y el bias será -10:

(0*-80 + 1*20 + 0*-80 + 0*-80 -10) -> 10.

$$(\overline{a} \wedge \overline{b} \wedge \overline{c} \wedge d)$$

Para que se active debe ser 0001, por lo que "a", "b" y "c" tendrán el valor -80, mientras que "d" tendrá el valor 20, y el bias será -10:

(0*-80 + 0*-80 + 0*-80 + 1*20 -10) -> 10.

$$(\bar{a} \wedge b \wedge \bar{c})$$

Para esta neurona y la última voy a introducir un pequeño cambio con respecto a las demás, pues el valor "d" no se especifica. Entonces, "a" y "c" serán -80, "b" será 40, "d" será 20 y el bias será -30. Esto se debe a que como no se especifica el valor de "d", le damos un valor positivo, pero como sabemos que "b" debe ser obligatoriamente 1, le damos más valor que a "d".

(0*-80 + 1*40 + 0*-80 + 1*20 - 30) -> 10.

$$(a \wedge b \wedge \bar{c} \wedge d)$$

Para que se active debe ser 1101, por lo que "a", "b" y "d" tendrán el valor 20, mientras que "c" tendrá el valor -80, y el bias será -50:

(1*20 + 1*20 + 0*-80 + 1*20 - 50) -> 10.

$$(a \wedge b \wedge \bar{c})$$

Mismo método que la cuarta neurona. "c" será -80, "a" y "b" serán 30, "d" será 10, y el bias -60.

(1*30 + 1*30 + 0*-80 + 1*10 - 60) -> 10.

Para el ajuste de la segunda matriz de pesos, he usado en todas las neuronas el mismo peso (30) y el mismo bias (-10); pues si ninguna neurona se activa, el resultado será -10 y la activación estará cercana a 0. Pero si 1 o más se activan, el valor mínimo que habrá será de 20 y la activación será cercana a 1.

Pesos 1: [[-80,20,20,-80], [-80,20,-80,-80], [-80,-80,-80,20], [-80,40,-80,20], [20,20,-80,20], [30,30,-80,10]]

Bias 1: [-30, -10, -10, -30, -50, -60]

Pesos 2: [30, 30, 30, 30, 30, 30]

Bias 2: [-10]

1.1.2. Implementando en python una función y = forward((a,b,c,d))

Comienza con la implementación de la función de activación simoidea en código:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Se crea la función "forward" que recibe como parámetro una tupla formada por una combinación de 4 números (0s y 1s). Se transforma la tupla a formato array para poder trabajar con las matrices de los pesos.

Se define la matriz de pesos de la primera capa, los bias de la primera capa, la matriz de pesos de la capa de salida y su correspondiente bias.

```
def forward(tupla):

    tupla = np.asarray(tupla)

    w1 = np.array([[-80,20,20,-80],
                   [-80,20,-80,-80],
                   [-80,-80,-80,20],
                   [-80,40,-80,20],
                   [20,20,-80,20],
                   [30,30,-80,10]])

    Bias = np.array([-30, -10, -10, -30, -50, -60])
    BiasOculta = np.array([-10])
    w2 = np.array([30, 30, 30, 30, 30, 30])
    CapaOculta = []
```

Ahora se hace un bucle sobre los 6 nodos de la primera capa y se calcula el valor de activación de cada uno de ellos. Se multiplica la entrada por los pesos correspondientes y se suma el bias correspondiente. Luego, se aplica la función sigmoidea a este resultado para obtener el valor de activación del nodo. Finalmente, se agrega este valor de activación a la lista CapaOculta.

Una vez terminado el bucle, se convierte la lista CapaOculta en un array para poder realizar operaciones matriciales con los pesos w2 y el bias BiasOculta. Se multiplica la activación de la primera capa por los pesos de la capa de salida y se suma el bias de la capa de salida. Por último, se aplica la función sigmoidea a este resultado para obtener la salida de la red neuronal.

```
for i in range(6):
    auxiliar = w1[i]
    x = sum(tupla * auxiliar)
    resultado = sigmoid(x + Bias[i])
    CapaOculta.append(resultado)

CapaOculta = np.asarray(CapaOculta)
output = sigmoid(sum(CapaOculta * w2) + BiasOculta[0])
output = output

return format(output)
```

Prueba de funcionamiento para las 16 posibles entradas

```
for i in range(16):
    print(forward((INPUTS[i])))
```

```
[0 0 0 0] -> 4.5521689622150855e-05
[0 0 0 1] -> 0.9999999979388463
[0 0 1 0] -> 4.5459737006960635e-05
[0 0 1 1] -> 4.5397868702434395e-05
[0 1 0 0] -> 1.0
[0 1 0 1] -> 0.9999999979416518
[0 1 1 0] -> 0.9999999979360372
[0 1 1 1] -> 4.5397868702434395e-05
[1 0 0 0] -> 4.539786870268929e-05
[1 0 0 1] -> 4.5459739817817976e-05
[1 0 1 0] -> 4.5397868702434395e-05
[1 0 1 1] -> 4.5397868702434395e-05
[1 1 0 0] -> 0.9933161972234091
[1 1 0 1] -> 1.0
[1 1 1 0] -> 4.5397868702434395e-05
[1 1 1 1] -> 4.5397868702434395e-05
```

1.2. Modelando, entrenando y probando la red en Keras

Primero se crean 2 vectores para guardar los inputs y los outputs de la red:

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

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

Se inicia la variable Model como un objeto de la clase Sequential(); clase que permite crear el MLP con Keras añadiéndole capas.

```
def keras(X, Y, EPOCHS, BATCH):
    model = Sequential()
```

Se usa el método ".add()" para añadir capas al objeto Model; se usa el constructor Dense para crear las capas. La primera capa (oculta) debe contener los inputs de la red; esto se especifica con el parámetro input_shape, que indica la dimensión de la entrada de la red. Se especifica el número de outputs, que en mi caso son 6. El siguiente ".add()" crea la capa de salida.

La función de activación que se va a usar se indica con el parámetro activation; en este caso se usa la función de activación sigmoid.

```
model = Sequential()
model.add(Dense(6, input_shape = (4,), activation = 'sigmoid'))
model.add(Dense(1, activation = 'sigmoid'))
```

Con el método ".compile()" se configura el modelo para el entrenamiento. El método usa el parámetro loss (función de pérdida), el parámetro optimizer (optimizador que se va a usar en el entrenamiento) y el parámetro metrics (métricas que evaluarán el modelo).

```
model.compile(loss = 'mean_squared_error', optimizer = 'adam', metrics = ['acc'])
```

Para realizar el entrenamiento de la red se usa el método ".fit()"; que entrena el modelo para un número fijo de épocas (epochs). Recibe los inputs y los outputs esperados, además de los epochs y el batch_size. Para empezar, voy a usar 6000 epochs y 16 de batch_size.

```
model.fit(X, Y, epochs = EPOCHS, batch_size = BATCH)
```

Con el método ".predict()" se generan predicciones de salida para las muestras de entrada.

```
z = model.predict(X)

for i, j in zip(Y, z):
    print('{} => {}'.format(i, j))
```

Prueba de la red

```
[0] => [0.00957282]
[1] => [0.9866108]
[0] => [0.00354138]
[0] => [0.00356471]
[1] => [0.9922666]
[1] => [0.99224246]
[1] => [0.9867063]
[0] => [0.0097317]
[0] => [0.00360547]
[0] => [0.00417255]
[0] => [0.00352157]
[0] => [0.00352729]
[1] => [0.99185455]
[1] => [0.99198794]
[0] => [0.00443939]
[0] => [0.00381717]
```

Se usan 6000 epochs y 16 de batch_size:

```
loss: 5.6833e-05 - acc: 1.0000
```

1.3. Analizando el entrenamiento y comparando con la red ajustada a mano

¿Cuántos pasos de entrenamiento necesitas para que aprenda correctamente la función? ¿Te parecen muchos?

Al principio desconocía qué cambiaba al modificar la cantidad de epochs; en la primera prueba le di un valor de 300 a los epochs y la red me devolvió un resultado pésimo. Fui probando, aumentando cada vez más los epochs; hasta que llegué a los valores de 6000 epochs y 16 batch_size, pues son 16 entradas diferentes a la red y aumentar el batch_size a un valor mayor a este, no tendría ningún efecto.

Sí, me parece una cantidad excesivamente elevada; además de tardar casi 2 minutos en completar el entrenamiento.

¿Qué es el parámetro loss y optimizer de la red? ¿Crees que afectan al entrenamiento?

El parámetro **loss** determina el método que se utiliza para compilar y determinar el error de la red. Indica cuan mala ha sido una predicción en un ejemplo concreto (si la predicción del modelo es perfecta, loss es 0). Guía la actualización de los pesos durante el entrenamiento.

El parámetro **optimizer** determina el tipo de optimizador que usa el compilador junto con la tasa de aprendizaje o learning_rate.

Un optimizador es un algoritmo que minimiza la función de pérdida durante el entrenamiento y se utiliza para saber qué pesos poner en cada capa de la red neuronal.

Creo que tanto el parámetro loss como el parámetro optimizer afectan al rendimiento, pues hay diversos tipos de optimizadores y maneras de calcular errores diferentes. Hay que escoger dichos tipos según las entradas, salidas, capas, funciones de activación, etc… que se deben utilizar.

¿Qué es el learning rate (LR) en una red neuronal? ¿Se puede ajustar el LR en Keras?

El **Learning Rate** indica que tanto se actualizan los pesos en cada iteración (en un rango $[0, 1]$); es decir, cada vez que se realiza una iteración en el proceso de entrenamiento se deben actualizar los pesos de entrada para poder dar cada vez una mejor aproximación. También se define como la "velocidad" a la que aprende la red neuronal.

Si el valor es demasiado alto, el modelo puede oscilar y no converger a una solución óptima; y si es demasiado bajo, el entrenamiento puede ser muy lento.

Para poder ajustar el Learning Rate en Keras usamos el parámetro optimizer y la clase Adam de Keras. Primero ajustamos el Learning Rate con la clase Adam y luego la asignamos al parámetro optimizer. Quedaría de la siguiente manera:

```
adam = Adam(learning_rate=0.004)
model.compile(loss = 'mean_squared_error', optimizer = adam, metrics = ['acc'])
```

Haz pruebas para intentar minimizar el número de pasos de entrenamiento requeridos. Comenta los resultados obtenidos.

Antes de las pruebas debo aclarar que el valor del batch_size va a ser ten todo momento de 16, pues como se ha explicado antes, un valor mayor a este no va a mejorar en nada la red.

Prueba con 4000 epochs:

```
[0] => [0.01558671]
[1] => [0.98108727]
[0] => [0.00733849]
[0] => [0.00755282]
[1] => [0.99027175]
[1] => [0.99014926]
[1] => [0.9822124]
[0] => [0.01749936]
[0] => [0.00755114]
[0] => [0.01131946]
[0] => [0.00716261]
[0] => [0.00721073]
[1] => [0.9883636]
[1] => [0.9868249]
[0] => [0.00995292]
[0] => [0.00811627]
```

Se obtienen los mismos resultados que con 6000 epochs (hay un ahorro de 2000 epochs).

Prueba con 1000 epochs y 0.01 de Learning Rate:

```
[0] => [0.04262182]
[1] => [0.9551754]
[0] => [0.02656115]
[0] => [0.02690474]
[1] => [0.9797131]
[1] => [0.9794114]
[1] => [0.9566401]
[0] => [0.04518517]
[0] => [0.02682584]
[0] => [0.02817664]
[0] => [0.02646213]
[0] => [0.02651453]
[1] => [0.9759926]
[1] => [0.9765206]
[0] => [0.02857368]
[0] => [0.02800254]
```

```
adam = Adam(learning_rate=0.01)
```

Se obtienen buenos resultado y la red ha pasado de 6000 epochs a 1000 epochs.

He podido mejorar la red al máximo cambiando el valor del Learning Rate a 0.1. Con este valor la red consigue el entrenamiento correcto con tan solo 150 epochs.

```
adam = Adam(learning_rate=0.1)
```

```
[0] => [0.01599731]
[1] => [0.9926858]
[0] => [0.01553448]
[0] => [0.01554022]
[1] => [0.99778867]
[1] => [0.9977875]
[1] => [0.99506605]
[0] => [0.01640544]
[0] => [0.01553314]
[0] => [0.0155373]
[0] => [0.01553317]
[0] => [0.01553643]
[1] => [0.99776816]
[1] => [0.9976706]
[0] => [0.01554707]
[0] => [0.01554282]
```

He podido pasar de 6000 epochs a 150 epochs.

Compara los pesos y bias aprendidos con los que pusiste a mano ¿Se parecen? ¿Ha aprendido la misma función? Justifica tu respuesta.

Ahora compruebo en Keras. Uso el siguiente código para ver los bias y los pesos luego del entrenamiento:

```
Keras_w1 = model.layers[0].get_weights()[0]
Keras_b1 = model.layers[0].get_weights()[1]
Keras_w2 = model.layers[1].get_weights()[0]
Keras_b2 = model.layers[1].get_weights()[1]
```

Pesos

```
Pesos capa 1:
[[-2.9570324   1.2338989  -0.732742    2.3562176  -2.7135057   0.77369756]
 [ 1.7511498  -0.20291156  6.2621703  -3.413946    4.2330546  -2.8242784 ]
 [-5.3576913   5.345085   -0.46440345  1.2513056  -0.8317666   2.8066025 ]
 [-2.9922953  -4.6492796  -6.161918   -2.5539446   3.5985007   1.4067093 ]]
```

```
Pesos capa 2:
[[ 4.0193014]
 [-4.922775 ]
 [ 4.3755383]
 [-3.2146087]
 [ 3.8527505]
 [-3.4953935]]
```

Bias

```
Bias capa 1:
[ 4.8911943   2.4281511  -3.2400777   0.4651968  -1.2219833  -0.02169824]
```

```
Bias capa 2:
[-0.19753914]
```

No se parecen en nada; pues yo he ajustado los pesos guiándome con la función booleana y la red neuronal no tiene esta información (busca otros datos en los que basarse para hacerlo).

Keras busca entrenar la red mediante las posibles entradas y las salidas que se esperan según qué entrada se meta.

1.4. Aplica backpropagation manualmente

Aplica backpropagation manualmente a tu red para un par de entradas de entrenamiento y explica su funcionamiento de manera clara y detallada. Se debe hacer una traza paso a paso, dado unos pesos determinados, explicando cómo se realizan los cálculos y como se ajustan los pesos. Se valorará la didáctica utilizada para explicar convenientemente el proceso realizado, así como la presentación del apartado. Comprueba que el ajuste mejora el desempeño de la red.

Ejemplo detallado

Ejemplo para la entrada $(0,0,0,1)$, salida 1, y con los pesos y bias de mi red explicada antes.

Con este ejemplo voy a probar que he seleccionado correctamente los valores.

Datos iniciales

Pesos de la capa de entrada:

[-80,20,20,-80]
[-80,20,-80,-80]
[-80,-80,-80,20]
[-80,40,-80,20]
[20,20,-80,20]
[30,30,-80,10]

Bias de la capa de entrada:

[-30,-10,-10,-30,-50,-60]

Pesos de la capa de salida:

[30,30,30,30,30,30]

Bias de la capa de salida:

[-10]

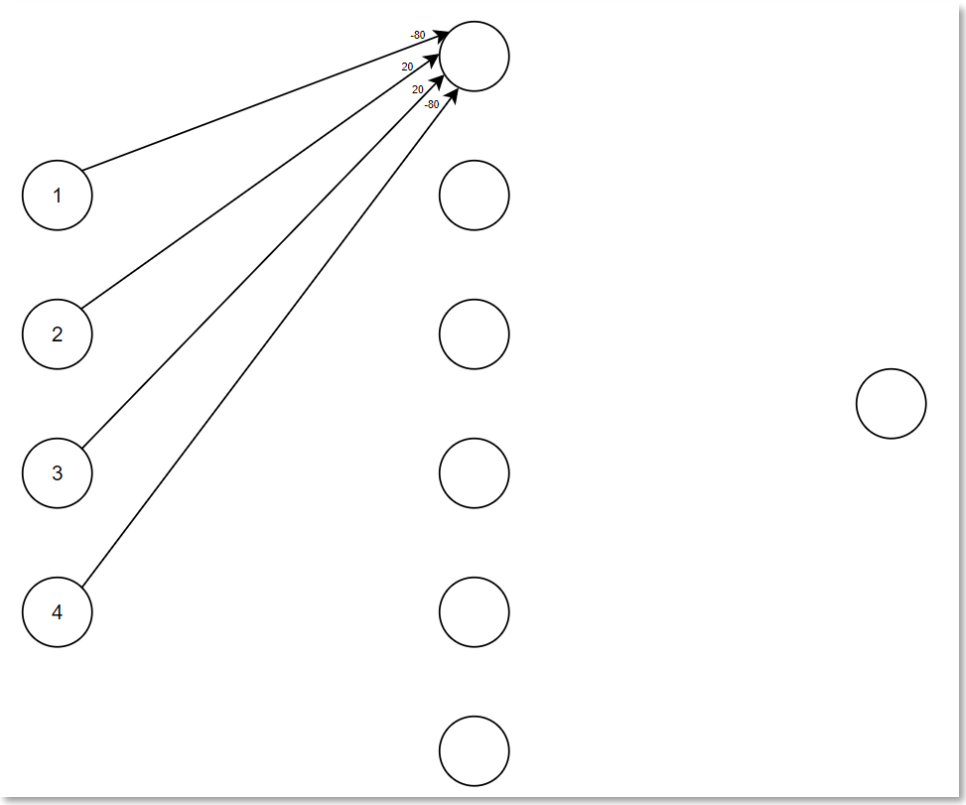
Entradas de la red:

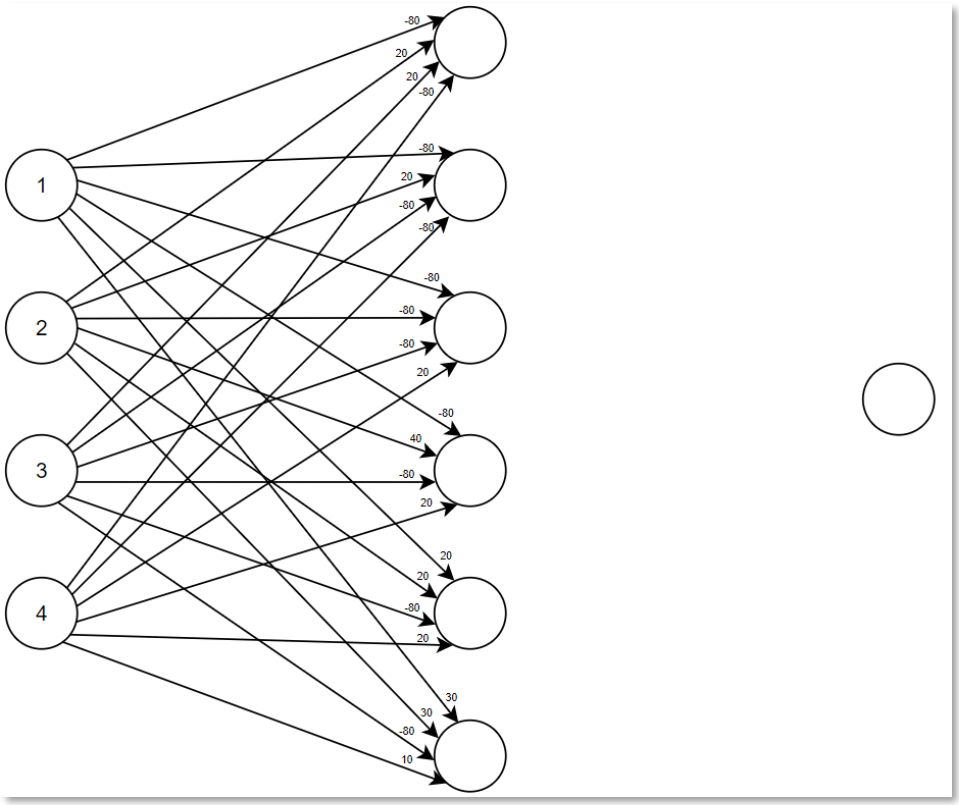
[(0,0,0,0),(0,0,0,1),(0,0,1,0),(0,0,1,1),(0,1,0,0),(01,0,1),(0,1,1,0),(0,1,1,1),
(1,0,0,0),(1,0,0,1),(1,0,1,0),(1,0,1,1),(1,1,0,0),(1,1,0,1),(1,1,1,0),(1,1,1,1)]

Salidas esperadas:

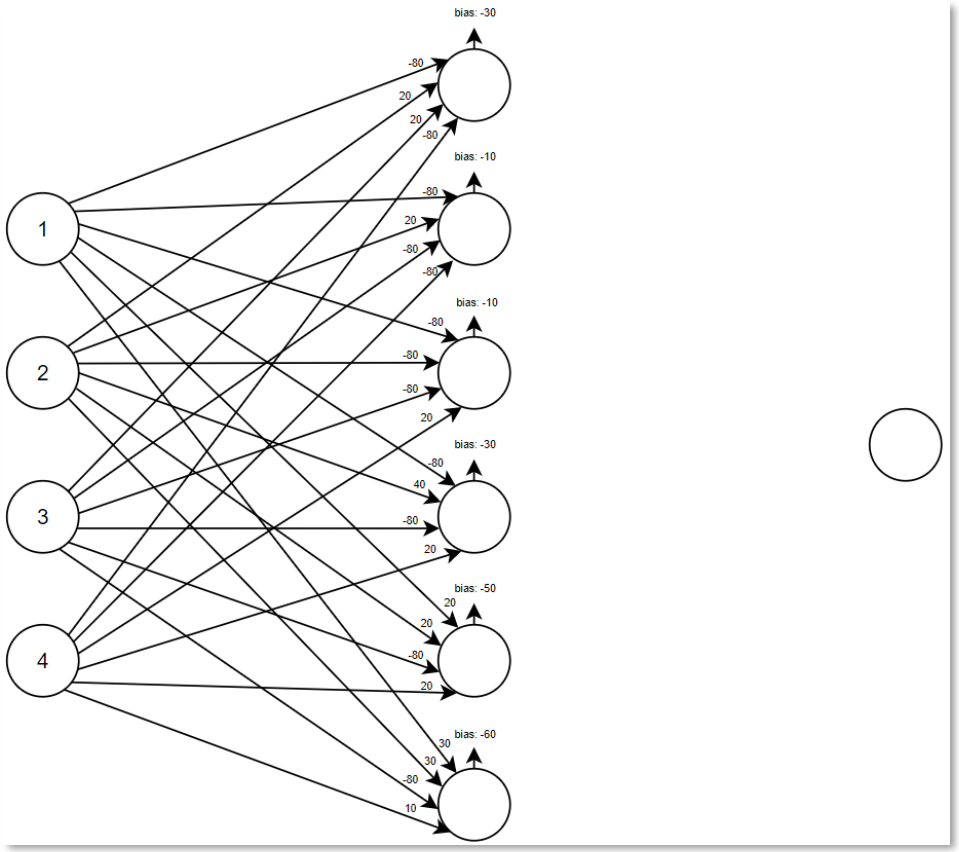
[[0],[1],[0],[0],[1],[1],[1],[0],[0],[0],[0],[0],[1],[1],[0],[0]]

Se detallan los pesos de la capa de entrada.

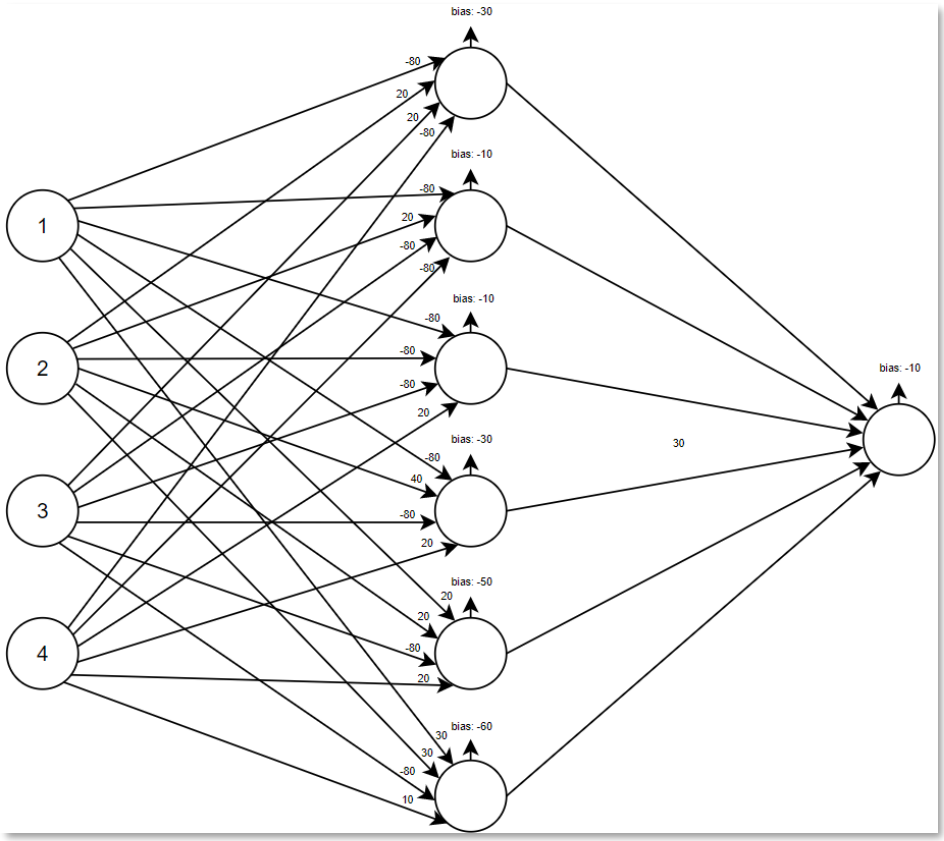




Se indican los bias correspondientes para cada neurona en la capa oculta.

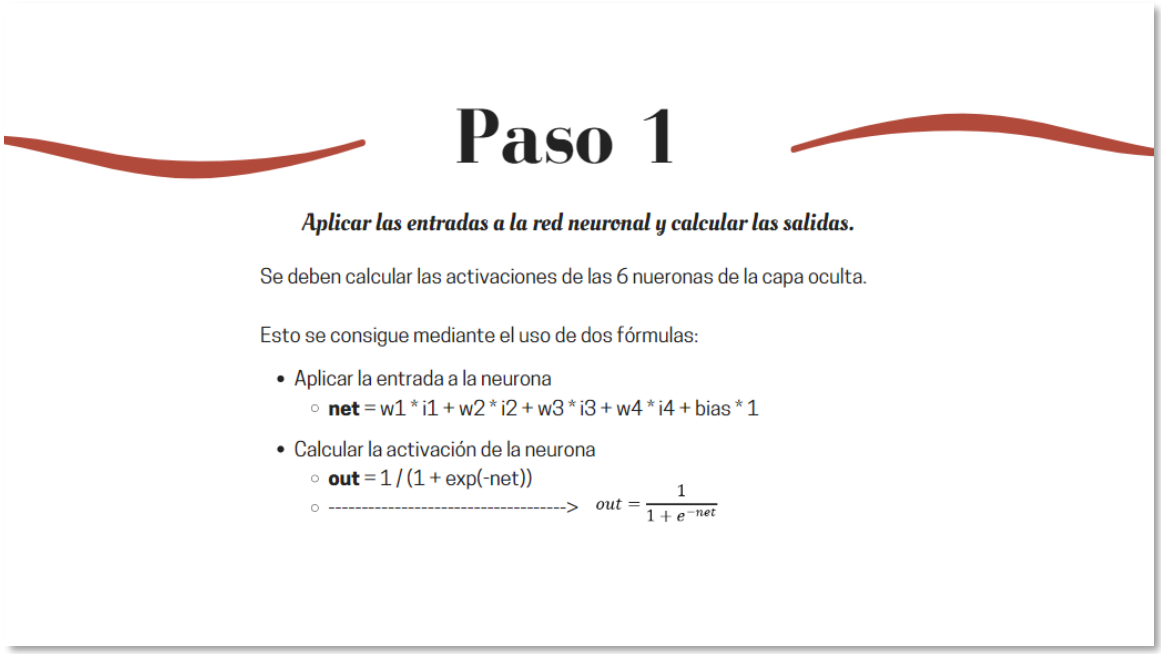


Se detallan los pesos de la capa de salida, y también su bias.



Pasos:

1. Aplicar las entradas a la red neuronal.
2. Calcular el error entre la salida esperada y la salida obtenida.
3. Propagar el error hacia atrás a través de la red.
4. Calcular los gradientes para cada uno de los pesos y los bias.
5. Ajustar los pesos y los bias en la dirección opuesta al gradiente para reducir el error.
6. Calcular la nueva salida de la red, junto con su error.
7. Repetir los pasos 1 a 6 varias veces hasta que el error sea lo suficientemente pequeño.



Aplicamos la entrada $(0,0,0,1)$ a la red neuronal y calculamos las salidas:

Primero, calculamos la activación para cada una de las 6 neuronas de la capa oculta:

Para la primera neurona:

- **net** = 0 * -80 + 0 * 20 + 0 * 20 + 1 * -80 + -30 = -110
- **out** = 1 / (1 + e^{-net}) = 1 / (1 + e^{110}) = $1.68*10^{-48} \approx 0$

Para la segunda neurona:

- **net** = 0 * -80 + 0 * 20 + 0 * -80 + 1 * -80 + -10 = -90
- **out** = 1 / (1 + e^{-net}) = 1 / (1 + e^{90}) = $8.19*10^{-40} \approx 0$

Para la tercera neurona:

- **net** = 0 * -80 + 0 * -80 + 0 * -80 + 1 * 20 + -10 = 10
- **out** = 1 / (1 + e^{-net}) = 1 / (1 + e^{-10}) = 0.9999546

Para la cuarta neurona:

- **net** = 0 * -80 + 0 * 40 + 0 * -80 + 1 * 20 + -30 = -10
- **out** = 1 / (1 + e^{-net}) = 1 / (1 + e^{10}) ≈ 0.00004539786

Para la quinta neurona:

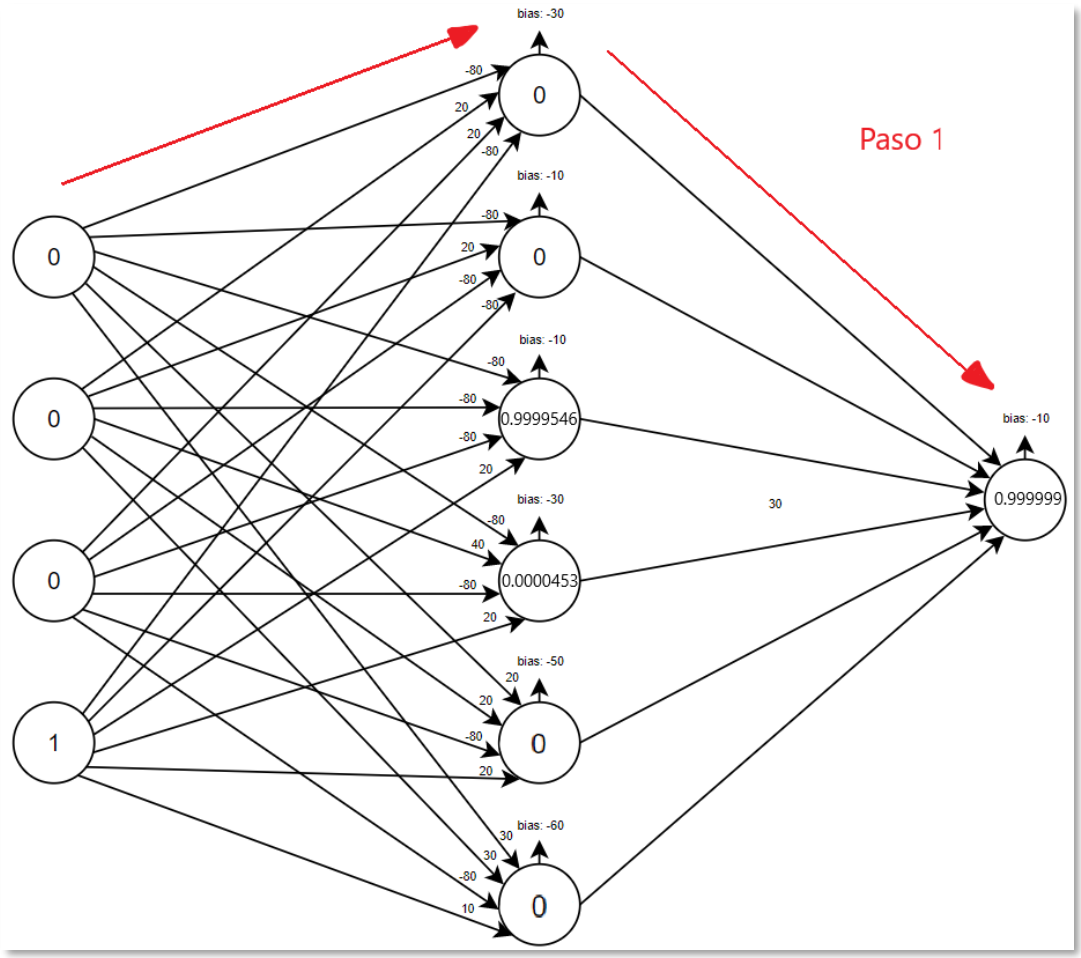
- **net** = 0 * 20 + 0 * 20 + 0 * -80 + 1 * 20 + -50 = -30
- **out** = 1 / (1 + e^{-net}) = 1 / (1 + e^{30}) = $9.35*10^{-14} \approx 0$

Para la sexta neurona:

- **net** = 0 * 30 + 0 * 30 + 0 * -80 + 1 * 10 + -60 = -50
- **out** = 1 / (1 + e^{-net}) = 1 / (1 + e^{50}) = $1.92*10^{-22} \approx 0$

Calculamos la salida de la red:

- **net** = 0 * 30 + 0 * 30 + 0.9999546 * 30 + 0.00004539786 * 30 + 0 * 30 + 0 * 30 + -10 = 19.99999994
- **out** = 1 / (1 + e^{-net}) = 1 / (1 + $e^{-19.99999994}$) = 0.9999999979



Paso 2

Calcular el error entre la salida esperada y la salida obtenida.

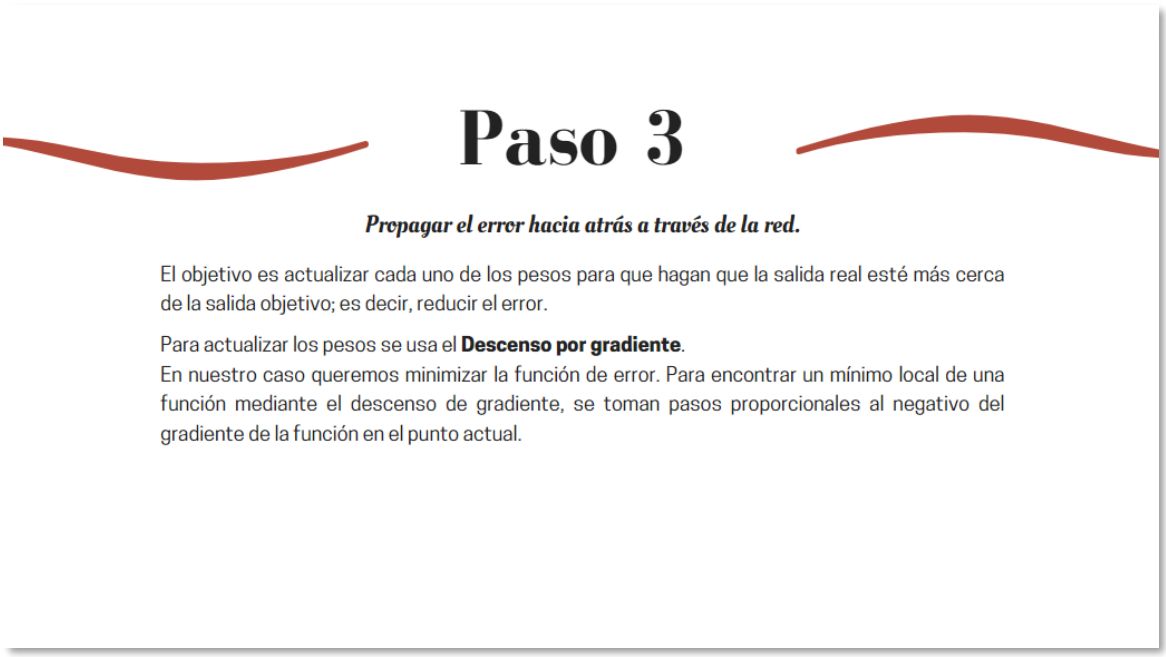
Calcular el error total

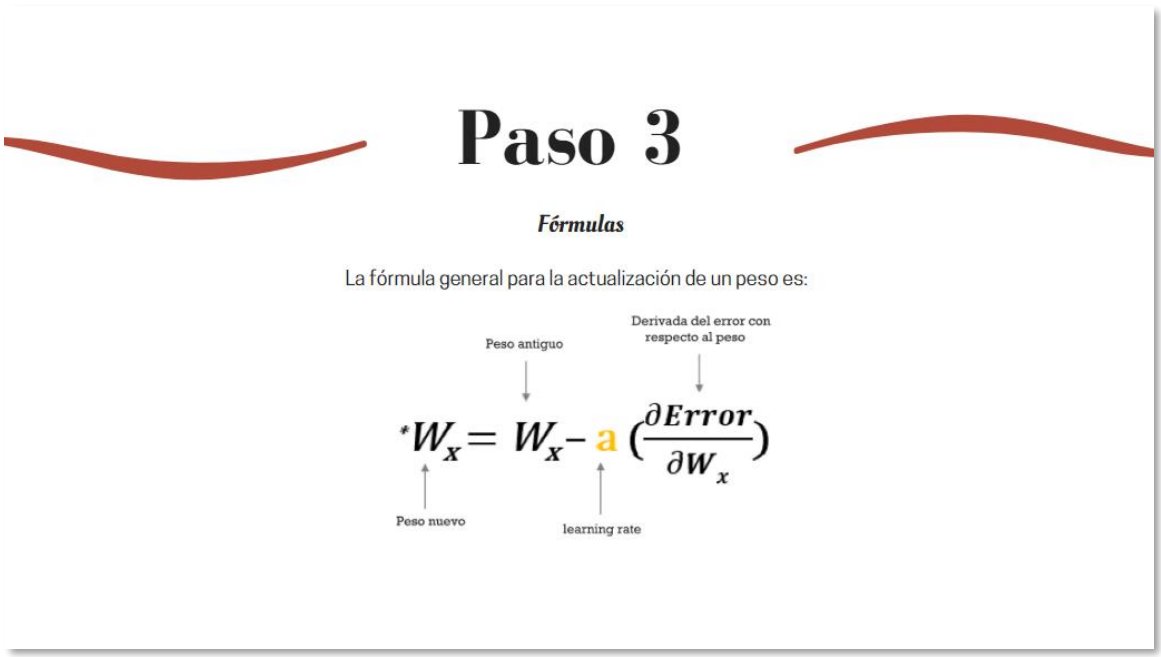
- ----------> $E_{total} = \sum \frac{1}{2}(output - target)^2$

"target" es el resultado ideal (el que queremos) y "output" es el que hemos conseguido en la salida de la red.

La salida esperada para $(0,0,0,1)$ es 1. Como solo tenemos una salida, el sumatorio de la fórmula lo obviamos.

$E_{total} = \frac{1}{2} (0.9999999979-1)^2 = 2.205x10^{-18}$





Fórmulas

Se evalua la derivada del error con la regal de la cadena.

Ejemplo para un peso "W6" conectado a una capa oculta "h2":

$$\frac{\partial Error}{\partial W_6} = \frac{\partial Error}{\partial\ \text{output}} * \frac{\partial\ \text{output}}{\partial W_6}$$

Regla de la cadena

$$\frac{\partial Error}{\partial W_6} = \frac{\partial \frac{1}{2}(\ \text{output}\ -\ \text{target}\ )^2}{\partial\ \text{output}} * \frac{\partial\ \text{output}}{\partial W_6}$$

$$\frac{\partial Error}{\partial W_6} = (\ \text{output}\ -\ \text{target}\ ) * (h_2)$$

$$\frac{\partial Error}{\partial W_6} = \Delta h_2$$

$\Delta =$ output $-$ target ← **Delta**

Fórmulas

Ejemplo para un peso "W3" conectado a la entrada "E2", a la capa oculta "h2" y dicha capa oculta tiene el peso "W6":

$$\frac{\partial Error}{\partial W_1} = \frac{\partial Error}{\partial\ \text{output}} * \frac{\partial\ \text{output}}{\partial\ \text{hl}} * \frac{\partial\ \text{hl}}{\partial W_1}$$

Regla de la cadena

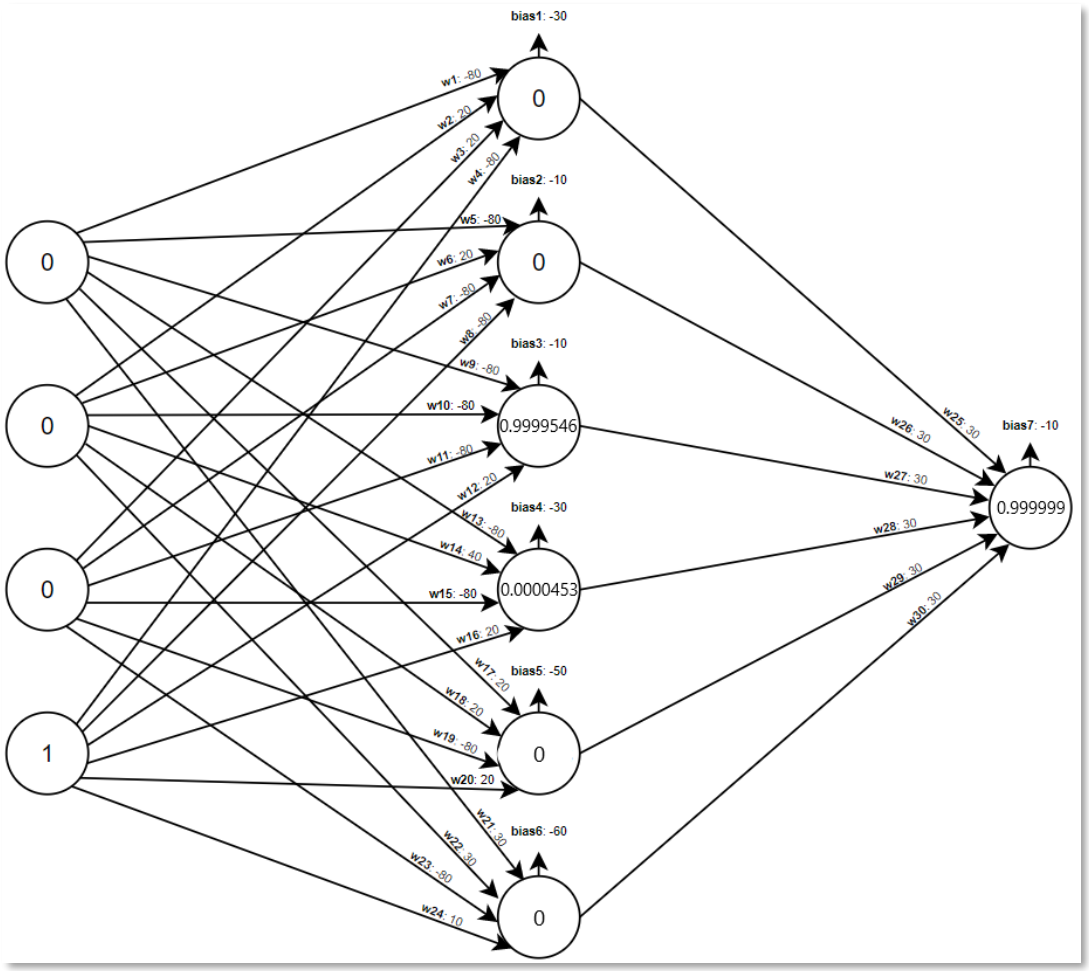
$$\frac{\partial Error}{\partial W_1} = \frac{\partial \frac{1}{2}(\ \text{output}\ -\ \text{target}\ )^2}{\partial\ \text{output}} * \frac{\partial\ \text{output}}{\partial\ \text{hl}} * \frac{\partial\ \text{hl}}{\partial W_1}$$

$$\frac{\partial Error}{\partial W_1} = (\ \text{output}\ -\ \text{target}\ ) * (w_5 i_1)$$

$$\frac{\partial Error}{\partial W_1} = \Delta\, w_5 i_1$$

$\Delta =$ output $-$ target ← **Delta**

Se numeran los pesos y los bias para el cálculo de sus nuevos valores.



Ajustar pesos

Capa **entrada**:

*w1: w1 - a (i1 · Δw25)
*w2: w2 - a (i2 · Δw25)
*w3: w3 - a (i3 · Δw25)
*w4: w4 - a (i4 · Δw25)

*w5: w5 - a (i1 · Δw26)
*w6: w6 - a (i2 · Δw26)
*w7: w7 - a (i3 · Δw26)
*w8: w8 - a (i4 · Δw26)

*w9: w9 - a (i1 · Δw27)
*w10: w10 - a (i2 · Δw27)
*w11: w11 - a (i3 · Δw27)
*w12: w12 - a (i4 · Δw27)

*w13: w13 - a (i1 · Δw28)
*w14: w14 - a (i2 · Δw28)
*w15: w15 - a (i3 · Δw28)
*w16: w16 - a (i4 · Δw28)

*w17: w17 - a (i1 · Δw29)
*w18: w18 - a (i2 · Δw29)
*w19: w19 - a (i3 · Δw29)
*w20: w20 - a (i4 · Δw29)

*w21: w21 - a (i1 · Δw30)
*w22: w22 - a (i2 · Δw30)
*w23: w23 - a (i3 · Δw30)
*w24: w24 - a (i4 · Δw30)

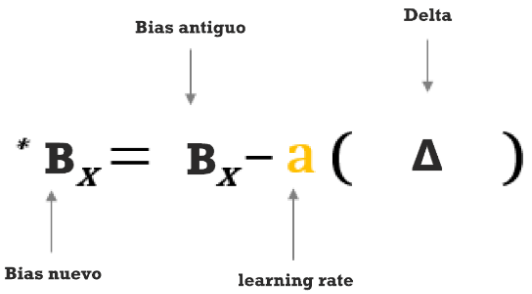
Capa **salida**:

*w25: w25 - a (h1 · Δ)
*w26: w26 - a (h2 · Δ)
*w27: w27 - a (h3 · Δ)
*w28: w28 - a (h4 · Δ)
*w29: w29 - a (h5 · Δ)
*w30: w30 - a (h6 · Δ)

w = peso
a = learning rate
h = neurona capa oculta
i = entrada
Δ = delta

Bias

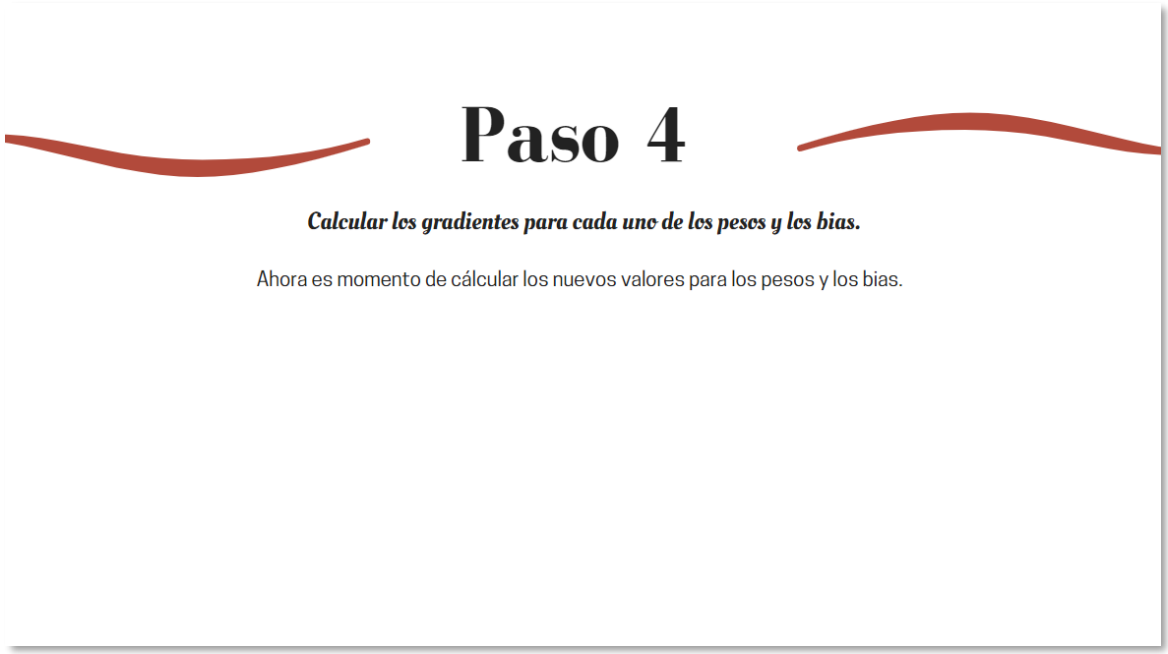
La fórmula general para la actualización de un bias es:



Ajustar bias

Capa **entrada**:

*bias1: bias1- a (Δ)
*bias2: bias2- a (Δ)
*bias3: bias3- a (Δ)
*bias4: bias4- a (Δ)
*bias5: bias5- a (Δ)
*bias6: bias6- a (Δ)
*bias7: bias7- a (Δ)



Δ = output – target = 0.9999999979– 1 = -0.0000000021

Learning rate: 0.5

Capa oculta:

- ***w25**: w25 - a (h1 · Δ) = 30 – 0.5 * (0 * -0.0000000021) = 30
- ***w26**: w26 - a (h2 · Δ) = 30 – 0.5 * (0 * -0.0000000021) = 30
- ***w27**: w27 - a (h3 · Δ) = 30 – 0.5 * (0.9999546 * -0.0000000021) = 30
- ***w28**: w28 - a (h4 · Δ) = 30 – 0.5 * (0.0000453 * -0.0000000021) = 30
- ***w29**: w29 - a (h5 · Δ) = 30 – 0.5 * (0 * -0.9999544783) = 30
- ***w30**: w30 - a (h6 · Δ) = 30 – 0.5 * (0 * -0.9999544783) = 30

- ***bias7**: bias7- a (Δ) = -10 – 0.5 * (-0.0000000021) = -9.999999999 (NUEVO)

Capa entrada:

- ***w1**: w1 - a (i1 · Δw25) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w2**: w2 - a (i1 · Δw25) = 20 – 0.5 * (0 * -0. 0000000021* 30) = 20
- ***w3**: w3 - a (i1 · Δw25) = 20 – 0.5 * (0 * -0. 0000000021* 30) = 20
- ***w4**: w4 - a (i1 · Δw25) = -80 – 0.5 * (1 * -0. 0000000021* 30) = -79.9999 (NUEVO)

- ***w5**: w5 - a (i2 · Δw26) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w6**: w6 - a (i2 · Δw26) = 20 – 0.5 * (0 * -0. 0000000021* 30) = 20
- ***w7**: w7 - a (i2 · Δw26) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w8**: w8 - a (i2 · Δw26) = -80 – 0.5 * (1 * -0. 0000000021* 30) = -79.9999 (NUEVO)

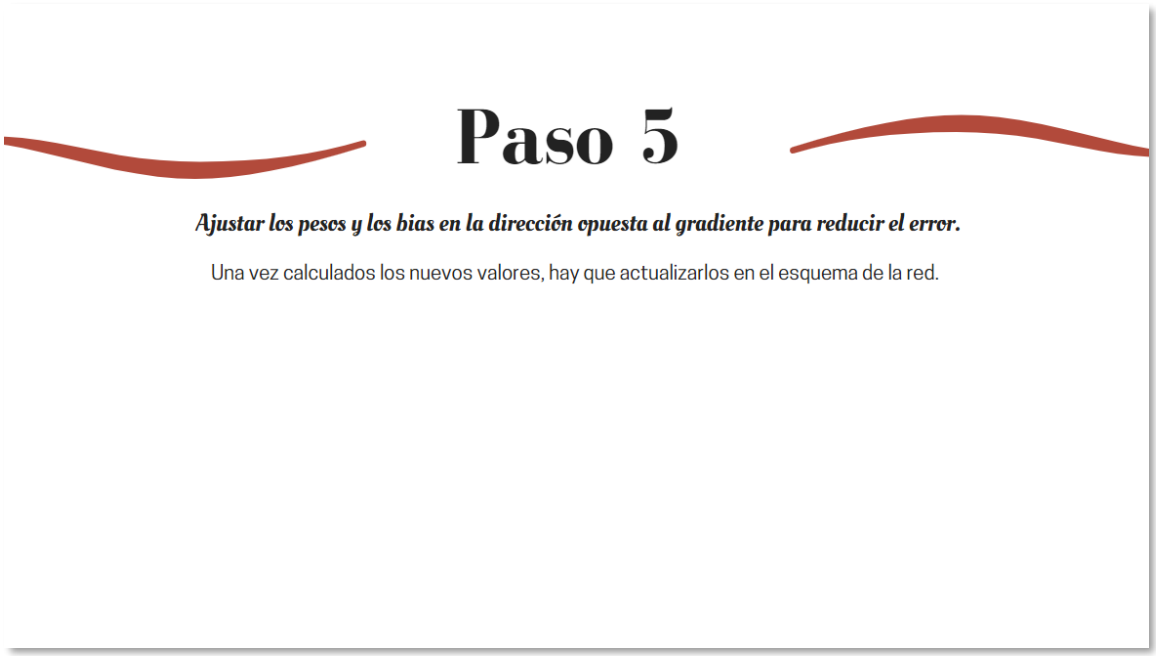
- ***w9**: w9 - a (i3 · Δw27) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w10**: w10 - a (i3 · Δw27) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w11**: w11 - a (i3 · Δw27) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w12**: w12 - a (i3 · Δw27) = 20 – 0.5 * (1 * -0. 0000000021* 30) = 20.00000003 (NUEVO)

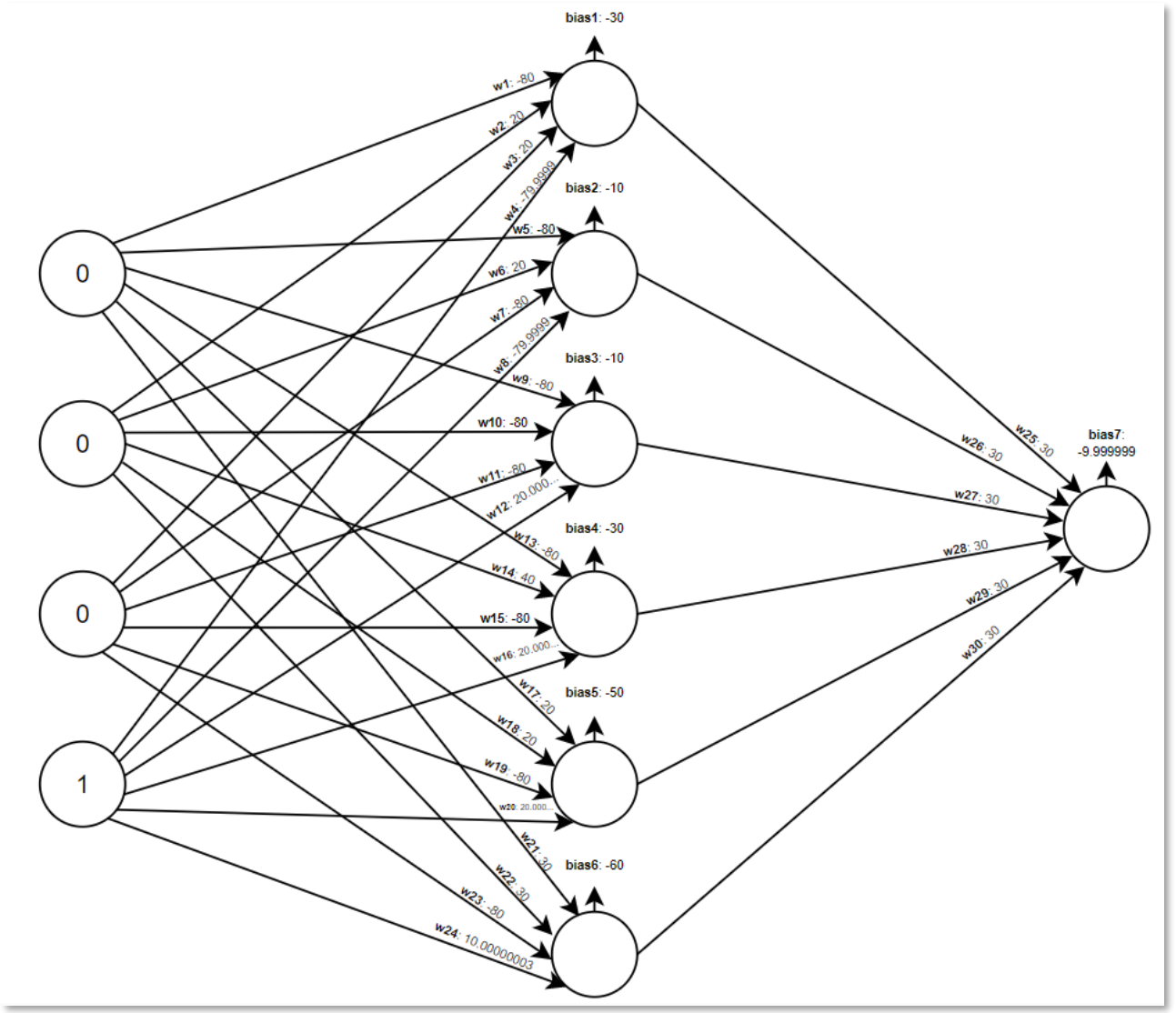
- ***w13**: w13 - a (i4 · Δw28) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w14**: w14 - a (i4 · Δw28) = 40 – 0.5 * (0 * -0. 0000000021* 30) = 40
- ***w15**: w15 - a (i4 · Δw28) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w16**: w16 - a (i4 · Δw28) = 20 – 0.5 * (1 * -0. 0000000021* 30) = 20.00000003 (NUEVO)

- ***w17**: w17 - a (i5 · Δw29) = 20 – 0.5 * (0 * -0. 0000000021* 30) = 20
- ***w18**: w18 - a (i5 · Δw29) = 20 – 0.5 * (0 * -0. 0000000021* 30) = 20
- ***w19**: w19 - a (i5 · Δw29) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w20**: w20 - a (i5 · Δw29) = 20 – 0.5 * (1 * -0. 0000000021* 30) = 20.00000003 (NUEVO)

- ***w21**: w21 - a (i6 · Δw30) = 30 – 0.5 * (0 * -0. 0000000021* 30) = 30
- ***w22**: w22 - a (i6 · Δw30) = 30 – 0.5 * (0 * -0. 0000000021* 30) = 30
- ***w23**: w23 - a (i6 · Δw30) = -80 – 0.5 * (0 * -0. 0000000021* 30) = -80
- ***w24**: w24 - a (i6 · Δw30) = 10 – 0.5 * (1 * -0. 0000000021* 30) = 10.00000003 (NUEVO)

- ***bias1**: bias1- a (Δ) = -30 – 0.5 * (-0. 0000000021) = -30
- ***bias2**: bias2- a (Δ) = -10 – 0.5 * (-0. 0000000021) = -10
- ***bias3**: bias3- a (Δ) = -10 – 0.5 * (-0. 0000000021) = -10
- ***bias4**: bias4- a (Δ) = -30 – 0.5 * (-0. 0000000021) = -30
- ***bias5**: bias5- a (Δ) = -50 – 0.5 * (-0. 0000000021) = -50
- ***bias6**: bias6- a (Δ) = -60 – 0.5 * (-0. 0000000021) = -60





Paso 6

Calcular la nueva salida de la red, junto con su error.

Ahora se debe calcular la nueva salida de la red y su error, utilizando la red con los nuevos valores para los pesos y los bias.

Primero, calculamos la activación para cada una de las 6 neuronas de la capa oculta:

Para la primera neurona:

- **net** = 0 * -80 + 0 * 20 + 0 * 20 + 1 * -79.9999 + -30 = -109.9999
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{109.9999}) = 1.68*10^{-48} \approx 0$

Para la segunda neurona:

- **net** = 0 * -80 + 0 * 20 + 0 * -80 + 1 * -79.9999 + -10 = -89.9999
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{89.9999}) = 8.19*10^{-40} \approx 0$

Para la tercera neurona:

- **net** = 0 * -80 + 0 * -80 + 0 * -80 + 1 * 20.00000003 + -10 = 10.00000003
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{-10.00000003}) = 0.9999546021$

Para la cuarta neurona:

- **net** = 0 * -80 + 0 * 40 + 0 * -80 + 1 * 20.00000003 + -30 = -9.99999997
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{9.99999997}) = 0.00004539787$

Para la quinta neurona:

- **net** = 0 * 20 + 0 * 20 + 0 * -80 + 1 * 20.00000003+ -50 = -29.99999997
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{29.99999997}) = 9.35*10^{-14} \approx 0$

Para la sexta neurona:

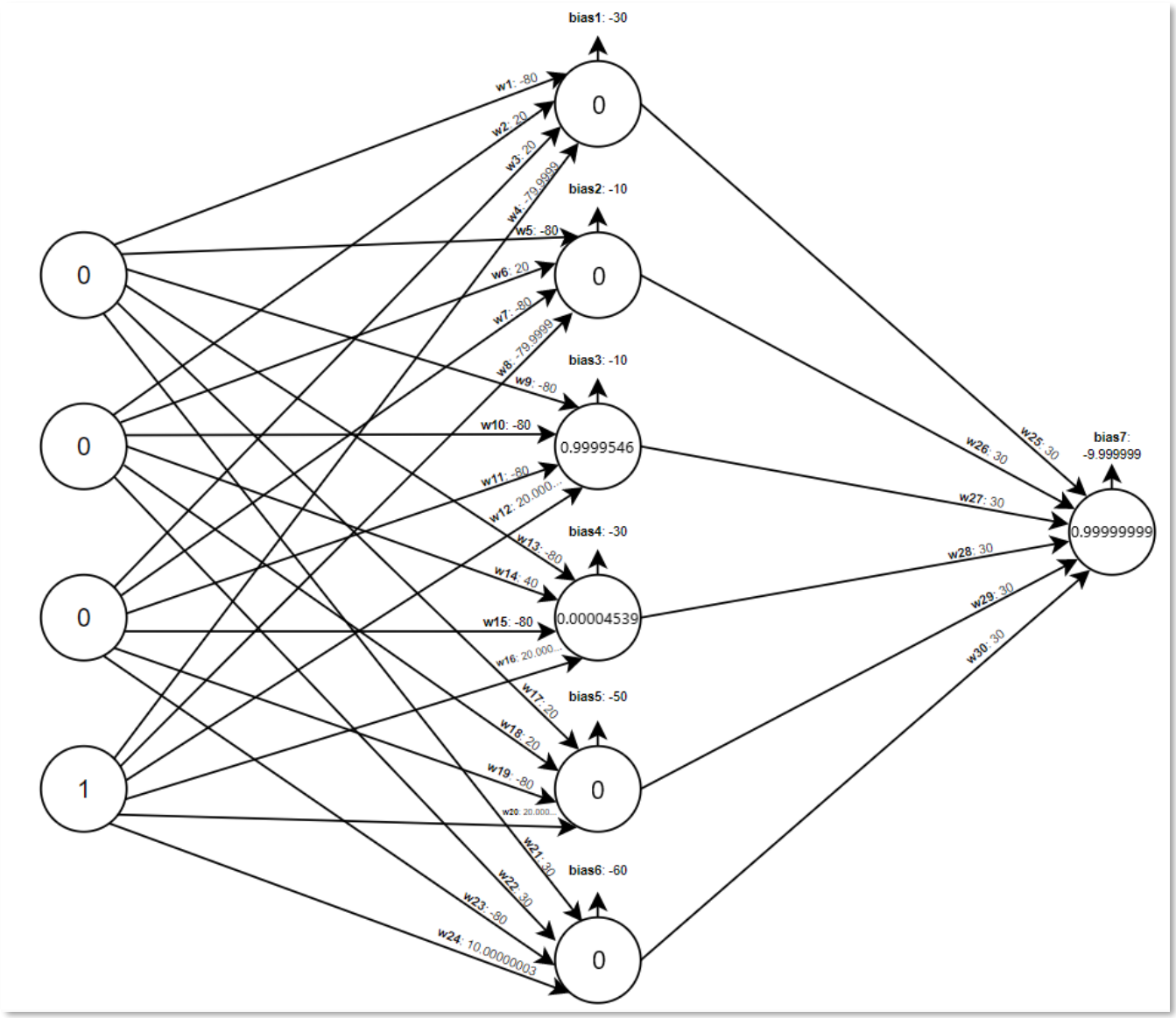
- **net** = 0 * 30 + 0 * 30 + 0 * -80 + 1 * 10.00000003 + -60 = -49.99999997
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{49.99999997}) = 1.92*10^{-22} \approx 0$

Calculamos la salida de la red:

- **net** = 0 * 30 + 0 * 30 + 0.9999546021 * 30 + 0.00004539787 * 30 + 0 * 30 + 0 * 30 + -10 = 20
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{-20}) = 0.9999999980$

Calculamos el error:

$\mathbf{E_{total}} = ½ (0.9999999980-1)^2 = 2*10^{-18}$



Paso 7

Repetir los pasos 1 a 6 varias veces hasta que el error sea lo suficientemente pequeño.

Ejemplo rápido

Ejemplo para la entrada $(0,0,1,1)$, con salida 0, y con valores de pesos y bias que no se corresponden con los de mi red.

Voy a probar que se ajustan bien los pesos/bias y se reduce el error en la red.

Datos iniciales

Pesos de la capa de entrada:
[-80,20,-10,-80]
[-80,20,-10,-10]
[-80,-60,-40,50]
[-80,30,-80,30]
[20,20,-80,60]
[30,20,-50,10]

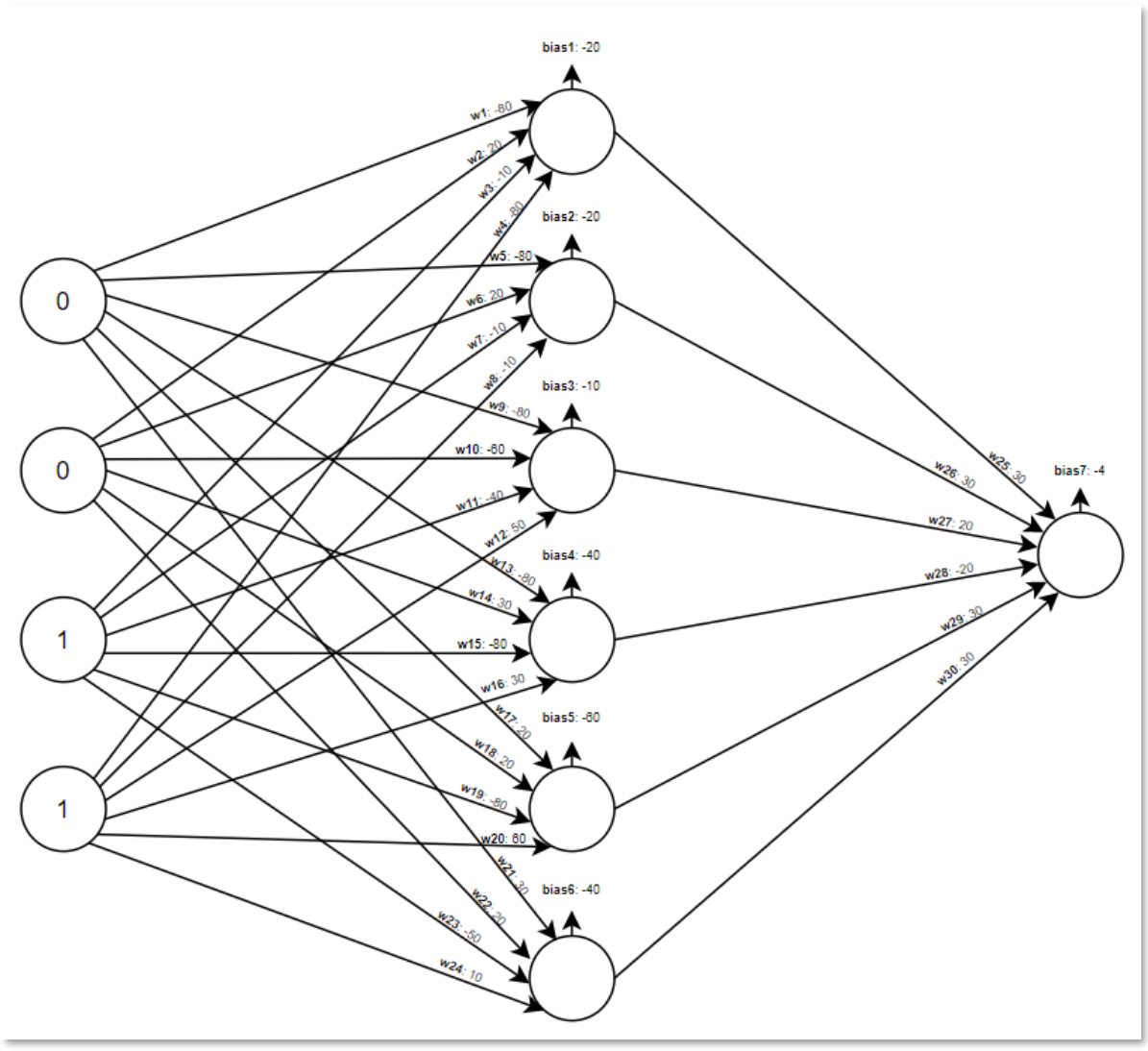
Bias de la capa de entrada:
[-20,-20,-10,-40,-60,-40]

Pesos de la capa de salida:
[30,30,20,-20,30,30]

Bias de la capa de salida:
[-4]

Entradas de la red:
[(0,0,0,0),(0,0,0,1),(0,0,1,0),(0,0,1,1),(0,1,0,0),(01,0,1),(0,1,1,0),(0,1,1,1),
(1,0,0,0),(1,0,0,1),(1,0,1,0),(1,0,1,1),(1,1,0,0),(1,1,0,1),(1,1,1,0),(1,1,1,1)]

Salidas esperadas:
[[0],[1],[0],[0],[1],[1],[1],[0],[0],[0],[0],[0],[1],[1],[0],[0]]



Aplicamos la entrada $(0,0,1,1)$ a la red neuronal y calculamos las salidas.

Primero, calculamos la activación para cada una de las 6 neuronas de la capa oculta:

Para la primera neurona:

- **net** = 0 * -80 + 0 * 20 + 1 * -10 + 1 * -80 + -20 = -110
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{110}) = 1.68*10^{-48} \approx 0$

Para la segunda neurona:

- **net** = 0 * -80 + 0 * 20 + 1 * -10 + 1 * -10 + -20 = -40
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{40}) = 4.24*10^{-18} \approx 0$

Para la tercera neurona:

- **net** = 0 * -80 + 0 * -60 + 1 * -40 + 1 * 50 + -10 = 0
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{0}) = 0.5$

Para la cuarta neurona:

- **net** = 0 * -80 + 0 * 30 + 1 * -80 + 1 * 30 + -40 = -90
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{90}) = 8.19*10^{-40} \approx 0$

Para la quinta neurona:

- **net** = 0 * 20 + 0 * 20 + 1 * -80 + 1 * 60 + -60 = -80
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{80}) = 1.80*10^{-35} \approx 0$

Para la sexta neurona:

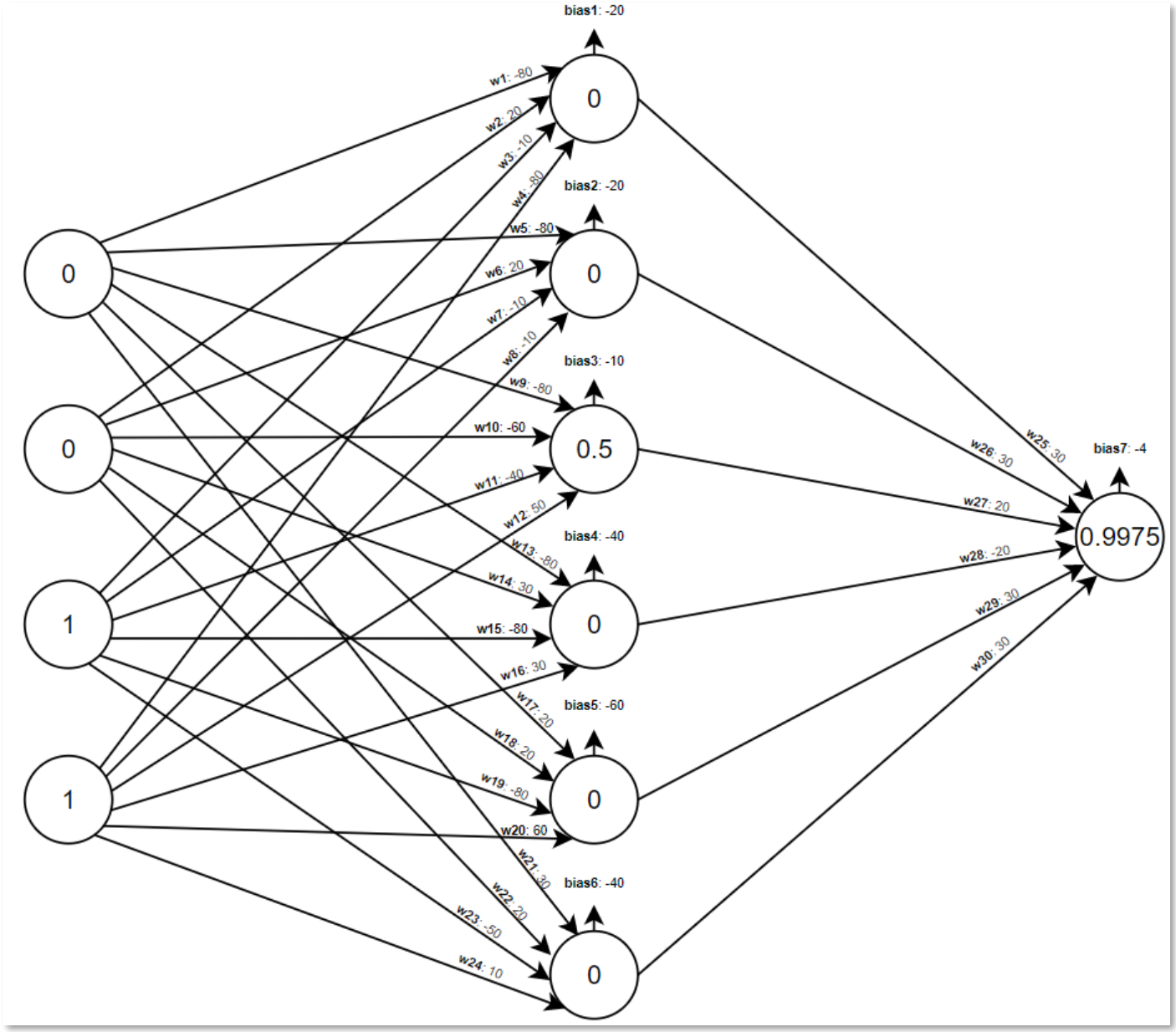
- **net** = 0 * 30 + 0 * 20 + 1 * -50 + 1 * 10 + -40 = -80
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{80}) = 1.80*10^{-35} \approx 0$

Calculamos la salida de la red:

- **net** = 0 * 30 + 0 * 30 + 0.5 * 20 + 0 * -20 + 0 * 30 + 0 * 30 + -4 = 6
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{-6}) = 0.9975$

La salida esperada para $(0,0,1,1)$ es 0. Como solo tenemos una salida, el sumatorio de la fórmula lo obviamos.

$\mathbf{E_{total}} = \frac{1}{2} (0.9975-0)^2 = 0.497503125$



Δ = output – target = 0.9975–0 = 0.9975

Learning rate: 0.5

Capa oculta:

- ***w25**: w25 - a (h1 · Δ) = 30 – 0.5 * (0 * 0.9975) = 30
- ***w26**: w26 - a (h2 · Δ) = 30 – 0.5 * (0 * 0.9975) = 30
- ***w27**: w27 - a (h3 · Δ) = 20 – 0.5 * (0.5 * 0.9975) = 19.75 (NUEVO)
- ***w28**: w28 - a (h4 · Δ) = -20 – 0.5 * (0 * 0.9975) = -20
- ***w29**: w29 - a (h5 · Δ) = 30 – 0.5 * (0 * 0.9975) = 30
- ***w30**: w30 - a (h6 · Δ) = 30 – 0.5 * (0 * 0.9975) = 30

- ***bias7**: bias7- a (Δ) = -4 – 0.5 * (0.9975) = -4.49875 (NUEVO)

Capa entrada:

- ***w1**: w1 - a (i1 · Δw25) = -80 – 0.5 * (0 * 0.9975 * 30) = -80
- ***w2**: w2 - a (i1 · Δw25) = 20 – 0.5 * (0 * 0.9975 * 30) = 20
- ***w3**: w3 - a (i1 · Δw25) = -10 – 0.5 * (1 * 0.9975 * 30) = -24.9625 (NUEVO)
- ***w4**: w4 - a (i1 · Δw25) = -80 – 0.5 * (1 * 0.9975 * 30) = -94.9625 (NUEVO)

- ***w5**: w5 - a (i2 · Δw26) = -80 – 0.5 * (0 * -0.9975 * 30) = -80
- ***w6**: w6 - a (i2 · Δw26) = 20 – 0.5 * (0 * 0.9975 * 30) = 20
- ***w7**: w7 - a (i2 · Δw26) = -10 – 0.5 * (1 * 0.9975 * 30) = -24.9625 (NUEVO)
- ***w8**: w8 - a (i2 · Δw26) = -10 – 0.5 * (1 * 0.9975 * 30) = -24.9625 (NUEVO)

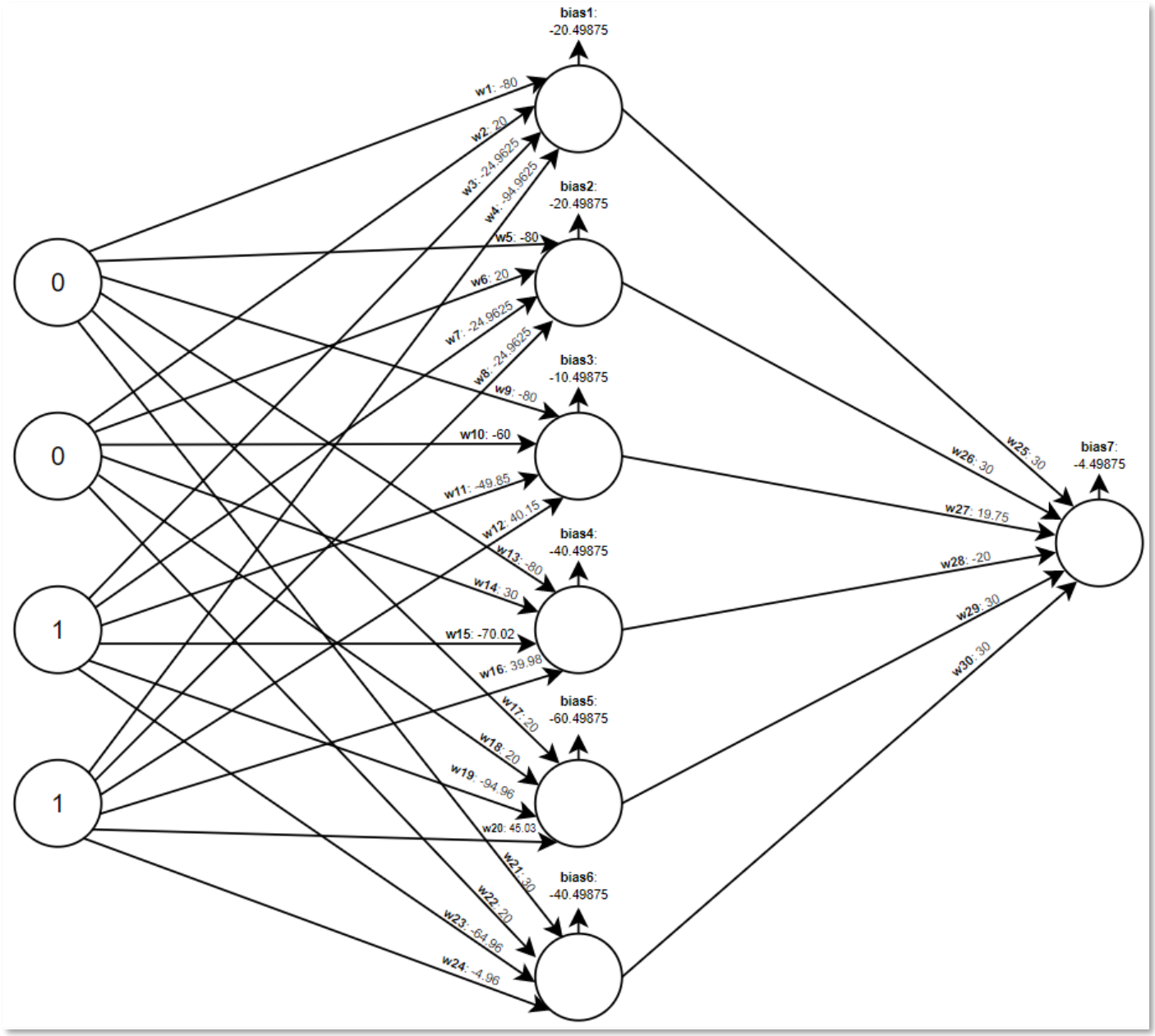
- ***w9**: w9 - a (i3 · Δw27) = -80 – 0.5 * (0 * 0.9975 * 19.75) = -80
- ***w10**: w10 - a (i3 · Δw27) = -60 – 0.5 * (0 * 0.9975 * 19.75) = -60
- ***w11**: w11 - a (i3 · Δw27) = -40 – 0.5 * (1 * 0.9975 * 19.75) = -49.85 (NUEVO)
- ***w12**: w12 - a (i3 · Δw27) = 50 – 0.5 * (1 * 0.9975 * 19.75) = 40.15 (NUEVO)

- ***w13**: w13 - a (i4 · Δw28) = -80 – 0.5 * (0 * 0.9975 * -20) = -80
- ***w14**: w14 - a (i4 · Δw28) = 30 – 0.5 * (0 * 0.9975 * -20) = 30
- ***w15**: w15 - a (i4 · Δw28) = -80 – 0.5 * (1 * 0.9975 * -20) = -70.02 (NUEVO)
- ***w16**: w16 - a (i4 · Δw28) = 30 – 0.5 * (1 * 0.9975 * -20) = 39.98 (NUEVO)

- ***w17**: w17 - a (i5 · Δw29) = 20 – 0.5 * (0 * 0.9975 * 30) = 20
- ***w18**: w18 - a (i5 · Δw29) = 20 – 0.5 * (0 * 0.9975 * 30) = 20
- ***w19**: w19 - a (i5 · Δw29) = -80 – 0.5 * (1 * 0.9975 * 30) = -94.96 (NUEVO)
- ***w20**: w20 - a (i5 · Δw29) = 60 – 0.5 * (1 * 0.9975 * 30) = 45.03 (NUEVO)

- ***w21**: w21 - a (i6 · Δw30) = 30 – 0.5 * (0 * 0.9975 * 30) = 30
- ***w22**: w22 - a (i6 · Δw30) = 20 – 0.5 * (0 * 0.9975 * 30) = 20
- ***w23**: w23 - a (i6 · Δw30) = -50 – 0.5 * (1 * 0.9975 * 30) = -64.96 (NUEVO)
- ***w24**: w24 - a (i6 · Δw30) = 10 – 0.5 * (1 * 0.9975 * 30) = -4.96 (NUEVO)

- ***bias1**: bias1- a (Δ) = -20 – 0.5 * (0.9975) = -20.49875 (NUEVO)
- ***bias2**: bias2- a (Δ) = -20 – 0.5 * (0.9975) = -20.49875 (NUEVO)
- ***bias3**: bias3- a (Δ) = -10 – 0.5 * (0.9975) = -10.49875 (NUEVO)
- ***bias4**: bias4- a (Δ) = -40 – 0.5 * (0.9975) = -40.49875 (NUEVO)
- ***bias5**: bias5- a (Δ) = -60 – 0.5 * (0.9975) = -60.49875 (NUEVO)
- ***bias6**: bias6- a (Δ) = -40 – 0.5 * (0.9975) = -40.49875 (NUEVO)



Aplicamos la entrada $(0,0,1,1)$ a la red neuronal y calculamos las salidas.

Primero, calculamos la activación para cada una de las 6 neuronas de la capa oculta:

Para la primera neurona:

- **net** = 0 * -80 + 0 * 20 + 1 * -24.9625 + 1 * -94.9625 + -20.49875 = -140.42
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{140.42}) = 1.03*10^{-61} \approx 0$

Para la segunda neurona:

- **net** = 0 * -80 + 0 * 20 + 1 * -24.9625 + 1 * -24.9625 + -20.49875 = -70.42375
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{70.42375}) = 2.60*10^{-31} \approx 0$

Para la tercera neurona:

- **net** = 0 * -80 + 0 * -60 + 1 * -49.85 + 1 * 40.15 + -10.49875 = -20.19
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{20.19}) = 0.0000000017$

Para la cuarta neurona:

- **net** = 0 * -80 + 0 * 30 + 1 * -70.02 + 1 * 39.98 + -40.49875 = -70.54
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{70.54}) = 2.31*10^{-31} \approx 0$

Para la quinta neurona:

- **net** = 0 * 20 + 0 * 20 + 1 * -94.96 + 1 * 45.03 + -60.49875 = -110.43
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{110.43}) = 1.09*10^{-48} \approx 0$

Para la sexta neurona:

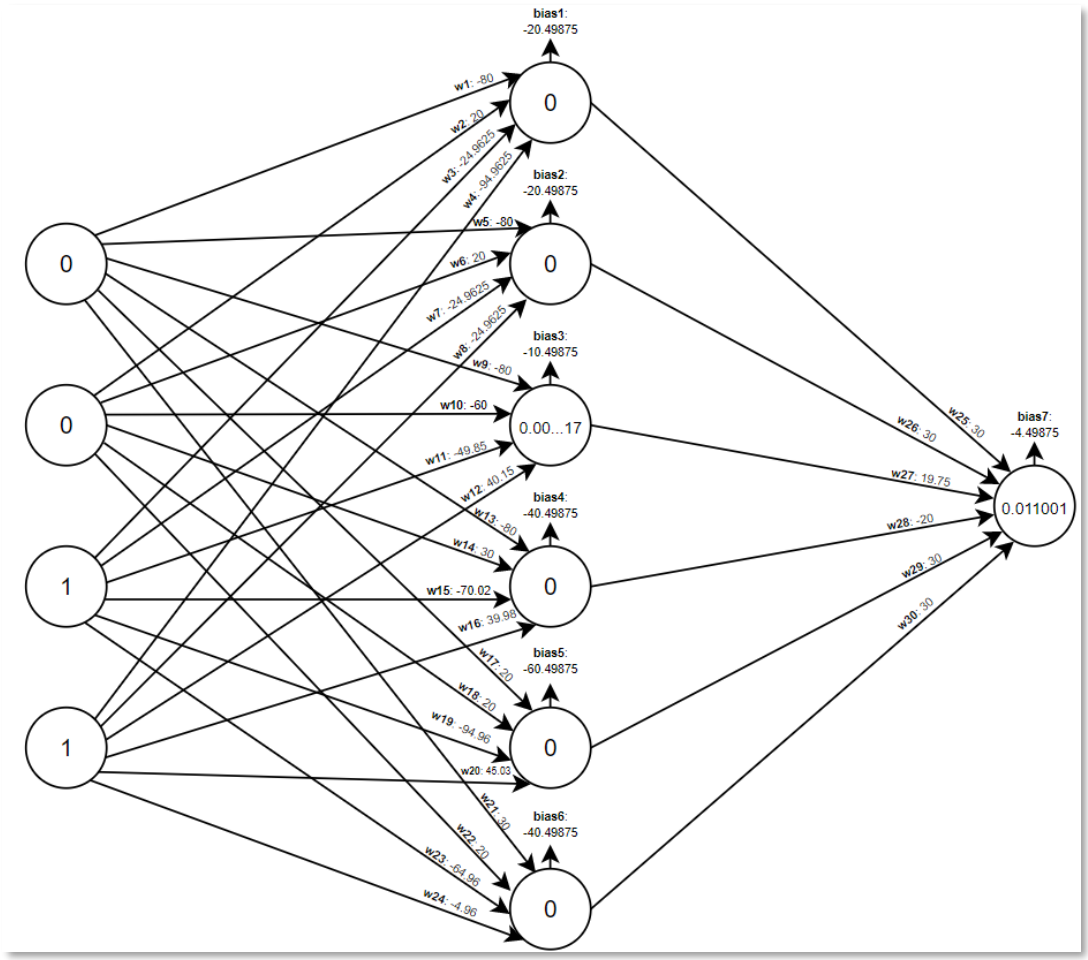
- **net** = 0 * 30 + 0 * 20 + 1 * -64.96 + 1 * -4.96 + -40.49875 = -110.418
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{110.418}) = 1.11*10^{-48} \approx 0$

Calculamos la salida de la red:

- **net** = 0 * 30 + 0 * 30 + 0.0000000017 * 19.75 + 0 * -20 + 0 * 30 + 0 * 30 + -4.49875 = -4.4987
- **out** = $1 / (1 + e^{-net}) = 1 / (1 + e^{4.4987}) = 0.011001077$

La salida esperada para (0,0,1,1) es 0. Como solo tenemos una salida, el sumatorio de la fórmula lo obviamos.

$\mathbf{E_{total}} = ½ (0.011001077-0)^2 = 0.00006051184$



2. Parte II -> Entrena un MLP mediante Deep Learning usando Keras

2.1. Procesamiento de los datos

Busca información sobre el conjunto/base de datos MNIST y explica con detalle para que sirve y su importancia.

El dataset **MNIST** es un conjunto de datos de aprendizaje automático muy popular en el campo de la visión por computadora. Contiene un conjunto de entrenamiento de 60.000 imágenes representando dígitos manuscritos del 0 al 9, y otro conjunto de pruebas con 10.000 muestras adicionales. Los dígitos están normalizados en tamaño y centrados en una imagen de tamaño fijo (28x28 píxeles).

Es muy importante pues se utiliza para la preparación de diferentes sistemas de manejo de imágenes, en el reconocimiento de patrones y en el aprendizaje automático. En nuestro caso, se va a utilizar para caso entrenar una red neuronal en diferenciar los números del 0 al 9.

¿Cuántas neuronas necesitamos en la capa de entrada? ¿Y en la capa de salida?

Para la **capa de entrada** tenemos 128 neuronas (indicadas en el enunciado). Las imágenes tienen una dimensión de 784 (28x28 píxeles), esto se indica en el parámetro "input_shape".

La **capa de salida** debe tener 10 neuronas, pues son las salidas que queremos. Se va a tratar un problema de clasificación y los valores a clasificar son los valores del 0 al 9.

¿Cómo dividirás el conjunto de entrenamiento/test/validación?

Para dividir el conjunto de datos se utiliza la función ".load_data()" de MNIST. Devuelve el conjunto de datos de entrenamiento/validación de 60000 imágenes (x_train, y_train) y 10000 de pruebas (x_test, y_test). Si no se ha hecho previamente, Keras descargará el conjunto de datos; pero si ya lo hizo, lo buscará en el cache local.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Los datos de validación se separarán en la función "model.fit()" con el parámetro validation_split. Se usará el valor 0.1 en dicho parámetro, por lo que se escogerán 6000 ejemplos para la validación (10% del conjunto de entrenamiento).

¿Cómo preprocesarás los datos de entrada?

Los datos vienen preprocesados por defecto. Una vez que obtenemos las imágenes, estas se encuentran almacenadas en un vector (60000, 28, 28), mientras que las salidas se encuentran en otro vector con diferente dimensión (60000, 1).

¿Cómo transformarás la imagen para poder entrenarla con una red MLP?

Lo primero es convertir los datos de entrada en vectores. Se aplana cada imagen; es decir, se redimensiona a 784 = 28 * 28 * 1 features. Se redimensionan los datos con la función ".reshape()".

```
X_train = X_train.reshape((X_train.shape[0], 28 * 28 * 1))
X_test = X_test.reshape((X_test.shape[0], 28 * 28 * 1))
```

También se deben normalizar los datos de entrada para evitar problemas. Para poder trabajar con las imágenes necesitamos que solo tengan valores entre 0 y 1 (al principio las imágenes tendrán valores entre 0 y 255).

Se convierten en floats de 32-bits con la función ".astype()". Se dividen entre 255.0, pues es el valor máximo que pueden tomar las imágenes.

```
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

Para preparar los datos de salida se deben convertir las salidas de su versión numérica a una versión one-hot encoded. La transformación es la siguiente:

```
0 --> [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1 --> [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2 --> [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3 --> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4 --> [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5 --> [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6 --> [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7 --> [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8 --> [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9 --> [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

La clase LabelBinarizer() permite hacer esta conversión. Se crea un objeto de dicha clase y se usa su método ".fit_transform()" para hacer la transformación de las salidas:

```
label_binarizer = LabelBinarizer()
y_train = label_binarizer.fit_transform(y_train)
y_test = label_binarizer.fit_transform(y_test)
```

2.2. Implementando la red en keras

Diseño de la red

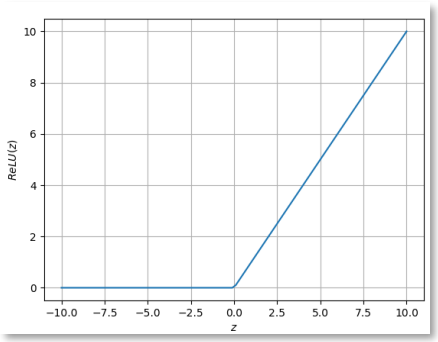
```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,), activation = 'relu'))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
```

```
adam = Adam()
model.compile(loss = 'categorical_crossentropy', optimizer = adam, metrics = ['accuracy'])
model.fit(X_train, y_train, epochs = EPOCHS, batch_size = BATCH, validation_split = 0.1, validation_data = (X_test, y_test), shuffle = True)
```

¿Qué es la función de activación relu?

La función **relu** se define como $f(x)=\max(0, x)$. Es una de las funciones de activación más usadas hoy en día y se utiliza en las redes neuronales para obtener resultados de menor error que los generados con la función sigmoidea.

Transforma los valores introducidos eliminando los valores negativos y dejando los positivos tal y como entran. Genera una salida igual a cero cuando la entrada es negativa, y una salida igual a la entrada cuando dicha entrada sea positiva.



¿Cómo varia de la sigmoidea vista en teoría?

La función **sigmoidea** transforma los valores introducidos a una escala $(0,1)$, donde los valores altos tienden de manera asintótica a 1 y los valores muy bajos tienden de manera asintótica a 0.

Es derivable, esto permite hacer de forma directa el Backpropagation en las redes neuronales. Tiende a saturar/matar el gradiente, tiene una lenta convergencia y presenta un tiempo de computación alto.

La relu no presenta saturación, todo lo contrario que en la función sigmoidea. Esto hace que el algoritmo del gradiente descendente converja más rápidamente, facilitando el entrenamiento.

A diferencia de la función sigmoidea, relu se denomina función por partes, porque la mitad de la salida es lineal (la salida positiva) mientras que la otra mitad no es lineal. La función relu también es mucho menos exigente computacionalmente que la función sigmoidea.

Las dos funciones coinciden en que no pueden tener valores negativos, pero la sigmoidea tampoco puede tener valores superiores a 1; al contrario que la función relu, que los valores mayores de 0 se quedan igual.

La función sigmoidea tiene un rango $[0, 1]$, y la función relu un rango $[0, +\infty]$.

¿Qué consigue la función de activación softmax de la última capa?

La función softmax divide los resultados en probabilidades del 0 al 1, consiguiendo que la suma de los valores finales de 1. Esto le sirve para ver el número más probable (con más valor).

¿Se podría usar una función de activación relu en la última capa?

No se puede utilizar la función de activación relu en la última capa puesto que es necesario que la red neuronal devuelva un valor en un rango limitado para su entrenamiento y esta función no produce una distribución de probabilidades sobre las diferentes clases.

El parámetro "loss='categorical_crossentropy'" requiere datos del 0 al 1, y la función relu no devuelve dicho formato de valores.

¿Qué función de activación crees que es mejor?

La función relu en general es más rápida y eficiente que las demás. Tiene un gran rendimiento en las capas ocultas. Su inconveniente es qué puede acabar con demasiados resultados en 0.

La función softmax es la más útil para la última capa de salida cuando se trata con una red configurada para dar salida a valores N, uno por cada clase en la tarea de clasificación.

La sigmoidea es útil en modelos en los que tenemos que predecir la probabilidad como un resultado.

¿Por qué crees que se suelen utilizar más relu que su alternativa sigmoidea?

La función relu se utiliza más que la función sigmoidea pues no presenta saturación, es más fácil de implementar computacionalmente y tiene una mayor simplicidad.

La función sigmoidea imita mejor el funcionamiento de las neuronas humanas, pero al entrenar redes neuronales es mejor la función relu, pues facilita su entrenamiento.

La función sigmoidea; al contrario que la función relu, satura/mata el gradiente y tiene una lenta convergencia.

Salida de la red

Una vez que se obtienen las predicciones de la red, se muestran los datos de salida con la siguiente función que crea un informe de texto donde se muestran las principales métricas de clasificación.

Se usa la función ".argmax()" para obtener el índice que devuelve la mayor probabilidad.

```
print(classification_report(y_test.argmax(axis=1), predictions.argmax(axis=1), target_names=[str(x) for x in label_binarizer.classes_]))
```

Explica que son y para qué sirven los parámetros batch_size y validation_split.

El parámetro **batch_size** es el número de datos que tiene cada iteración de un ciclo (epoch). Si dividimos los ciclos en iteraciones con un numero de datos más pequeño ya no es necesario cargar todos los datos en la memoria al mismo tiempo y la red neuronal se entrena más rápido.

Si se especifica el parámetro batch_size cada ciclo (epoch) tendrá más ejecuciones internas.

El parámetro **validacion_split** (valor flotante entre 0 y 1) indica la fracción de los datos de entrenamiento que se usará como datos de validación. El modelo separará esta fracción de los datos de entrenamiento, no los entrenará, evaluará la pérdida y cualquier métrica del modelo en estos datos al final de cada época. Los datos de validación se seleccionan de las últimas muestras en los datos x e y proporcionados, antes de barajar.

Por ejemplo, si el validation_split es 0.1, significa que el 10% de los datos de entrenamiento se van a utilizar para la validación.

¿Conoces algún otro algoritmo de optimización distinto adam? Comenta brevemente el funcionamiento de otro distinto.

El algoritmo **Descenso de gradiente estocástico** (SGD) es el algoritmo de optimización más simple. Ajustar los pesos del modelo en la dirección opuesta al gradiente del error con respecto a los pesos, utilizando un tamaño de paso de forma fija.

AdaGrad es un algoritmo que escala/adapta el valor de cada peso para cada dimensión con respecto al gradiente acumulado en cada iteración.

2.3. Probando el modelo

¿Qué error de clasificación obtienes para los datos de test? ¿Y qué valor de perdida de la función de coste? Para este apartado gasta la función de keras model.evaluate.

El valor de perdida es el valor loss y el error de clasificación es el valor -> 1 - accuracy.

Se realiza la evaluación con 15 epochs y 128 de batch_size:

```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,), activation = 'relu'))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))

adam = Adam()
model.compile(loss = 'categorical_crossentropy', optimizer = adam, metrics = ['accuracy'])
model.fit(X_train, y_train, epochs = EPOCHS, batch_size = BATCH, validation_split=0.1, validation_data = (X_test, y_test))

predictions = model.predict(X_test)
print()
print(classification_report(y_test.argmax(axis=1), predictions.argmax(axis=1), target_names=[str(x) for x in label_binarizer.classes_]))
print()
evaluacion = model.evaluate(X_test, y_test, batch_size = BATCH)
print()
print("test loss, test acc:", evaluacion)
print()
```

Resultados de la evaluación:

```
79/79 [==============================] - 0s 5ms/step - loss: 0.1158 - accuracy: 0.9752

test loss, test acc: [0.1158452183008194, 0.9751999974250793]
```

Nos devuelve un valor de 0.115 para el valor de perdida y valor de 0.03 (1-0.97) para el error de clasificación.

Escoge 1000 elementos al azar del conjunto de entrenamiento y comprueba que tasa de error de clasificación obtienes. ¿Coincide con la métrica anterior? (por aquí)

Seleccionamos varios conjuntos de elementos aleatorios de 1000 elementos del conjunto de entrenamiento, así podemos probar de una mejor manera; y con más caso, como varía la métrica con respecto a la anterior.

Se seleccionan los conjuntos de valores aleatorios del conjunto de entrenamiento:

```
newArrayX1 = X_train[40000:41000]
newArrayY1 = y_train[40000:41000]

newArrayX2 = X_train[30000:31000]
newArrayY2 = y_train[30000:31000]

newArrayX3 = X_train[20000:21000]
newArrayY3 = y_train[20000:21000]
```

```
adam = Adam()
model.compile(loss = 'categorical_crossentropy', optimizer = adam, metrics = ['accuracy'])
model.fit(newArrayX1, newArrayY1, epochs = EPOCHS, batch_size = BATCH, validation_split = 0.1, validation_data = (X_test, y_test))
```

Resultado para el conjunto $\mathrm{newArrayX1}$ y $\mathrm{newArrayY1}$:

```
79/79 [==============================] - 1s 11ms/step - loss: 0.4050 - accuracy: 0.8889

test loss, test acc: [0.405032217502594, 0.8888999819755554]
```

Resultado para el conjunto $\mathrm{newArrayX2}$ y $\mathrm{newArrayY2}$:

```
79/79 [==============================] - 0s 3ms/step - loss: 0.3857 - accuracy: 0.8890

test loss, test acc: [0.3856920301914215, 0.8889999985694885]
```

Resultado para el conjunto $\mathrm{newArrayX3}$ y $\mathrm{newArrayY3}$:

```
79/79 [==============================] - 0s 3ms/step - loss: 0.4095 - accuracy: 0.8743

test loss, test acc: [0.40948519110679626, 0.8743000030517578]
```

Se aprecia como los resultados obtenidos son peores que los anteriores pues se ha entrenado con un número sumamente menor de datos de entrenamiento.

En general se obtiene un valor de pérdida del 0.40 y un error de clasificación de 0.12 (1-0.87).

Escoge 1000 elementos al azar del conjunto de test y comprueba que tasa de error de clasificación obtienes. ¿Coincide con la métrica anterior?

Se seleccionan los conjuntos de valores aleatorios del conjunto de test:

```
newTestX1 = X_test[5000:6000]
newTestY1 = y_test[5000:6000]

newTestX2 = X_test[4000:5000]
newTestY2 = y_test[4000:5000]

newTestX3 = X_test[3000:4000]
newTestY3 = y_test[3000:4000]
```

```
adam = Adam()
model.compile(loss = 'categorical_crossentropy', optimizer = adam, metrics = ['accuracy'])
model.fit(X_train, y_train, epochs = EPOCHS, batch_size = BATCH, validation_split = 0.1, validation_data = (newTestX1, newTestY1))

predictions = model.predict(newTestX1)
print()
print(classification_report(newTestY1.argmax(axis = 1), predictions.argmax(axis = 1), target_names = [str(x) for x in label_binarizer.classes_]))
print()
evaluacion = model.evaluate(newTestX1, newTestY1, batch_size = BATCH)
```

Resultado para el conjunto newTestX1 y newTestY1:

```
8/8 [==============================] - 0s 5ms/step - loss: 0.0920 - accuracy: 0.9840

test loss, test acc: [0.09201616048812866, 0.984000027179718]
```

Resultado para el conjunto newTestX2 y newTestY2:

```
8/8 [==============================] - 0s 4ms/step - loss: 0.1267 - accuracy: 0.9680

test loss, test acc: [0.12673646211624146, 0.9679999947547913]
```

Resultado para el conjunto newTestX2 y newTestY2:

```
8/8 [==============================] - 0s 3ms/step - loss: 0.1027 - accuracy: 0.9710

test loss, test acc: [0.10274914652109146, 0.9710000157356262]
```

En este caso como tenemos los 60000 datos de entrada, pero solamente 1000 datos en el conjunto de pruebas en vez de 10000, obtenemos mejores resultados en toda la clasificación.

Se obtiene un valor de pérdida del 0.1 y un valor de error de clasificación del 0.02 (1-0.98).

2.4. Mejorando la red

Intentar mejorar el resultado de la clasificación cambiando la red o su entrenamiento. Se permiten utilizar cualquier mejora, pero:

- **Se deben realizar varias pruebas con una red vanilla MLP. Se puede sacar la máxima nota haciendo pruebas con este tipo de red en exclusiva.**
- **Se pueden probar además otras arquitecturas más adaptadas al uso de imágenes. En este caso se deben realizar varias pruebas para mejorar lo máximo posible la arquitectura escogida (a parte de las del apartado a).**

Debes explicar las técnicas utilizadas, el proceso seguido y la mejora conseguida con respecto al conjunto de test. Se debe informar tanto de las pruebas realizadas que no consigan mejoría como las que sí. Se valorará el número de pruebas realizada, la justificación de porqué se han realizado las pruebas (y que estrategia se ha seguido para diseñarlas) así como las referencias bibliográficas utilizadas.

En el caso de "red vanilla MLP" se refiere a una implementación básica y sin modificaciones adicionales.

Cuando se habla de mejorar el resultado de la clasificación, se está hablando de mejorar la tasa de precisión de la red. Para comprobar esta tasa, voy a usar el método ".evaluate()" de los aparatados anteriores.

Como hemos comprobado en el apartado anterior, la tasa de predicción de la red sin mejorar es de un valor de 0.115 para el valor de perdida y valor de 0.03 (1-0.97) para el error de clasificación; es decir, un 97% de aciertos en la clasificación.

Pruebas con una Red Vanilla MLP

Número de neuronas en las capas

Primeramente, voy a probar a cambiar el número de neuronas en las capas de la red, aumentándolas o disminuyéndolas, a ver si se consigue un cambio en la clasificación.

Con capas de 256 neuronas (128x2):

```
test loss, test acc: [0.07848817110061646, 0.9828000068664551]
```

Con capas de 64 neuronas (128/2):

```
test loss, test acc: [0.09105904400348663, 0.9761999845504761]
```

Con capas de 1024 neuronas (128x8):

```
test loss, test acc: [0.09379648417234421, 0.9805999994277954]
```

Una vez hechas las pruebas he podido mejorar muy poco la red con respecto a la red inicial. Aumentando el número de neuronas en las capas, la red mejorar muy brevemente; mientras que, si se disminuye el número, empeora la tasa de precisión.

Número de capas en la red

La siguiente prueba se basa en aumentar el número de capas ocultas que contiene la red, para ver si obtenemos unos mejores resultados al hacer que los datos de entrada pasen por más capas ocultas hasta llegar al resultado final.

Prueba 1:

```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,), activation = 'relu'))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
```

```
test loss, test acc: [0.08791486918926239, 0.9789000153541565]
```

Resultado muy similar al de la red inicial, sin cambios en el número de neuronas.

Prueba 2:

```
model =  Sequential()
model.add(Dense(256, input_shape=(28 * 28 * 1,), activation = 'relu'))
model.add(Dense(256, activation = 'relu'))
model.add(Dense(256, activation = 'relu'))
model.add(Dense(256, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
```

```
test loss, test acc: [0.08505858480930328, 0.982200026512146]
```

Cambiando el número de capas ocultas no se ha conseguido ninguna mejora significante.

Cambio de hiperparámetros

Ahora voy a cambiar el número del batch_size y de las epochs; esto lo hago para comprobar si con una mayor cantidad de entrenamiento y en porciones más pequeñas, consigue mejorar.

Prueba aumentando el valor batch_size considerablemente. La red tendrá un mayor rendimiento y se entrenará a una velocidad mayor:

```
#    (epochs, batch_size)
MNIST(15,        500)
```

```
test loss, test acc: [0.07741749286651611, 0.9782999753952026]
```

Prueba disminuyendo el valor batch_size considerablemente. Al contrario que la prueba anterior, el entrenamiento tardará más tiempo en realizarse puesto que las epochs contarán con más datos que entrenar:

```
#    (epochs, batch_size)
MNIST(15,        32)
```

```
test loss, test acc: [0.13864034414291382, 0.9743000268936157]
```

Aumentado las epochs:

```
#    (epochs, batch_size)
MNIST(30,        128)
```

```
test loss, test acc: [0.13506007194519043, 0.9781000018119812]
```

Disminuyendo las epochs:

```
#    (epochs, batch_size)
MNIST(5,         128)
```

```
test loss, test acc: [0.08882596343755722, 0.9725000262260437]
```

Cambiando el valor de los hiperparámetros no se consigue ninguna mejora notable.

Función de activación

Ahora se voy a probar diferentes funciones de activación parecidas a la del modelo inicial. La función actual es la ReLU.

LeakyReLU:

```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,)))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(128))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(10, activation = 'softmax'))
```

Donde "alpha" es un parámetro que controla la pendiente de la rectificación negativa; es decir, cuando una entrada es negativa, en lugar de ser rectificada a cero como en la función ReLU, es rectificada a un valor "alpha" veces la entrada negativa.

```
test loss, test acc: [0.08155564963817596, 0.9799000024795532]
```

PReLU:

```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,)))
model.add(PReLU())
model.add(Dense(128))
model.add(PReLU())
model.add(Dense(10, activation = 'softmax'))
```

```
test loss, test acc: [0.10559829324483871, 0.9750999808311462]
```

La diferencia entre ambas funciones es el uso del parámetro "alpha". En la función LeakyReLU declara el valor antes de comenzar el entrenamiento, mientras que en la función PReLU, este valor va cambiando según se vaya desarrollando el entrenamiento.

No se ha conseguido ninguna mejora significativa con respecto a la tasa de precisión de la red, puesto que las dos funciones de activación son derivadas de la función ReLU.

Learning Rate

Utilizando el optimizador indicado en la práctica (Adam), voy a probar a cambiar el valor del learning rate para ver si mejora la red. El valor del learning rate del optimizador Adam es 0.001 por defecto.

Prueba 1:

```
adam = Adam(learning_rate=0.01)
```

```
test loss, test acc: [0.1782258152961731, 0.9751999974250793]
```

Prueba 2:

```
adam = Adam(learning_rate=0.1)
```

```
test loss, test acc: [1.8276222944259644, 0.2071000039577484]
```

Ha ocurrido un error.

Prueba 3:

```
adam = Adam(learning_rate=0.0001)
```

```
test loss, test acc: [0.10354520380496979, 0.9681000113487244]
```

Prueba 4:

```
adam = Adam(learning_rate=0.00001)
```

```
test loss, test acc: [0.29846954345703125, 0.9193000197410583]
```

Al cambiar el learning rate no mejora la red; es más, empeora la red si los valores se alejan mucho del valor por defecto que tiene Adam. Incluso puede llegar a dar errores.

Optimizador

Voy a probar a cambiar el tipo de optimizador.

RMSprop (su learning rate es 0.001 por defecto) utiliza una técnica de regularización para prevenir que los pesos de la red no exploten o se hagan muy pequeños:

```
optimizador = RMSprop()
model.compile(loss = 'categorical_crossentropy', optimizer = optimizador, metrics = ['accuracy'])
```

```
test loss, test acc: [0.10128507763147354, 0.978600025177002]
```

Adagrad (tiene un learning rate de 0.001 por defecto) utiliza una técnica de adaptación de aprendizaje para ajustar el tamaño de los pasos de descenso de gradiente:

```
optimizador = Adagrad()
model.compile(loss = 'categorical_crossentropy', optimizer = optimizador, metrics = ['accuracy'])
```

```
test loss, test acc: [0.3338167369365692, 0.9067999720573425]
```

Con RMSprop se obtienen unos resultados parecidos al optimizador Adam, pero Adagrad empeora totalmente el resultado.

Técnicas de regularización

Voy a probar a usar técnicas de regularización, pues se puede reducir el overfitting y mejorar la precisión de clasificación. Voy a probar el Dropout y la regularización L1/L2.

Dropout: durante cada epoch, un subconjunto de las neuronas de entrada se apaga, lo que evita que las neuronas se vuelvan demasiado dependientes de otras neuronas específicas en la red.

```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,), activation = 'relu'))
model.add(Dropout(0.05))
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.05))
model.add(Dense(10, activation = 'softmax'))
```

```
test loss, test acc: [0.08898285776376724, 0.9793999791145325]
```

Regularización L1: tanto L1 como L2 agregan una penalización a la función de pérdida en función de la magnitud de los pesos de la capa y favorece la selección de características importantes.

```
from keras.regularizers import l1, l2
```

```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,), activation = 'relu', kernel_regularizer=l1(0.01)))
model.add(Dense(128, activation = 'relu', kernel_regularizer=l1(0.01)))
model.add(Dense(10, activation = 'softmax'))
```

```
test loss, test acc: [0.9357690215110779, 0.8716999888420105]
```

Regularización L2:

```
model =  Sequential()
model.add(Dense(128, input_shape=(28 * 28 * 1,), activation = 'relu', kernel_regularizer=l2(0.01)))
model.add(Dense(128, activation = 'relu', kernel_regularizer=l2(0.01)))
model.add(Dense(10, activation = 'softmax'))
```

```
test loss, test acc: [0.26958659291267395, 0.9621999859809875]
```

Como se puede ver, ninguna técnica logra mejorar la red.

Pruebas con otras arquitecturas

Red Neuronal Convolucional

Es un tipo de red neuronal que se ha demostrado ser muy efectiva en la clasificación de imágenes.

Se basa en el uso de capas de filtros convolucionales y pooling para detectar características locales y abstractas en los datos de entrada.

Su implementación en nuestra red vanilla sería:

```
model = Sequential()
model.add(Conv2D(32, kernel_size = (3, 3), activation = 'relu', input_shape = (28, 28, 1)))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
```

La primera capa es una capa convolucional 2D que tiene 32 filtros de 3x3 y la función de activación ReLU. La capa convolucional usa los filtros para extraer características de la imagen de entrada, como bordes y texturas, y detectarlas en cualquier parte de la imagen. Y al utilizar varios filtros, la capa convolucional puede aprender características/patrones en paralelo.

La segunda capa es una capa de agrupación máxima (MaxPooling2D) que reduce la dimensión de la imagen de características a la mitad. En este caso, la ventana de agrupación tiene un tamaño de 2x2.

Y la tercera capa es una capa de aplanamiento (Flatten) que convierte la imagen de características en un vector unidimensional.

Prueba de la Red Neuronal Convolucional:

```
test loss, test acc: [0.04986335337162018, 0.9868000149726868]
```

Este tipo de arquitectura sí que consigue mejorar la red; no mucho, pero si notablemente.

Red Neuronal Recurrente

Una Red Neuronal Recurrente es un tipo arquitectura que agrega capas de tipo recurrente con respecto al modelo vanilla.

```
model = Sequential()
model.add(SimpleRNN(128, input_shape = (28, 28, 1), activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
```

Este tipo de capas utilizan una sola unidad de memoria recurrente. Funcionan tomando una secuencia de datos de entrada y procesando cada elemento de la secuencia uno por uno, utilizando la unidad recurrente para mantener una memoria interna del estado de la secuencia hasta ese momento.

El resultado de una capa SimpleRNN es una secuencia de salida, donde cada elemento de la secuencia es la salida de la capa para el correspondiente elemento de entrada de la secuencia de entrada.

Prueba de la Red Neuronal Recurrente:

```
test loss, test acc: [0.07742419093847275, 0.9779999852180481]
```

Este tipo de arquitectura no consigue mejorar la red; obtengo peores resultados.

ResNet (Residual Network)

Las ResNet son una arquitectura de red neuronal profunda que utiliza bloques residuales para mejorar el flujo de información y reducir el impacto de la desaparición del gradiente.

Los bloques residuales utilizan conexiones de salto para agregar la información original a la información transformada, esto facilita la optimización de la red y evita que la información original se pierda.

```
# Capa de entrada
input_layer = Input(shape = (28 * 28 * 1,))

# Bloque residual
x = Dense(784, activation = 'relu')(input_layer)
x = Dense(784, activation = 'relu')(x)
residual = x

# Añadir la suma residual a la entrada
x = Add()([input_layer, residual])
x = Dense(10, activation = 'softmax')(x)

# Modelo
model = Model(inputs = input_layer, outputs = x)
```

Se define una capa de entrada de la red neuronal con una forma de entrada de (28 * 28 * 1), como hacemos en el modelo vanilla.

Luego se define un bloque residual con dos capas "Dense" que utilizan la función de activación ReLU del modelo inicial. Las dos capas tienen una salida de 784 nodos (28x28). La primera se agrega a la capa de entrada, y la segunda se agrega a las capas de salida.

La suma residual consiste en la suma de la salida de la segunda capa densa y la entrada original. La suma residual tiene como objetivo agregar la información adicional obtenida en la capa densa al conjunto de datos de entrada original.

Finalmente, se define una capa densa de salida con una salida de 10 nodos, que corresponde al número de clases de salida diferentes.

Prueba de la Residual Network:

```
test loss, test acc: [0.08720794320106506, 0.9818999767303467]
```

Con esta arquitectura no obtengo mejores resultados; es más, son muy parecidos a los del modelo vanilla.

DenseNet

Una DenseNet es una arquitectura de red neuronal convolucional profunda que se caracteriza por tener conexiones densas entre las capas.

Cada capa recibe la entrada de todas las capas anteriores y pasa su salida a todas las capas siguientes. Esto permite que la red aproveche la información de características útiles en todas las capas anteriores; la red se vuelve más profunda y con mejor rendimiento.

```
input_layer = Input(shape = (28 * 28 * 1,))

x = Dense(128, activation = 'relu')(input_layer)
x = BatchNormalization()(x)
x = Dense(128, activation = 'relu')(x)

x = concatenate([input_layer, x])
output_layer = Dense(10, activation = 'softmax')(x)

model = Model(inputs = input_layer, outputs = output_layer)
```

La red utiliza una combinación de capas densas, normalización por lotes y concatenación para aprender características útiles de las imágenes.

En primer lugar, se define la capa de entrada "input_layer" con una forma de (28 * 28 * 1); esta capa se pasa a una capa densa con 128 neuronas y una función de activación ReLU.

Ahora se normaliza la activación de la capa anterior; esto se consigue agregando una capa de normalización por lotes ("BatchNormalization()"), que reducirá el sesgo y la varianza. Después, se agrega otra capa densa con 128 neuronas y una función de activación ReLU.

A continuación, se utiliza la función "concatenate" para concatenar la capa de entrada original con la salida de la segunda capa densa. Esto se hace para permitir que la información fluya tanto a través de las capas densas como a través de la entrada original, lo que puede ayudar a la red a aprender características más útiles y a tener una mejor capacidad de generalización.

Finalmente, la salida de la capa concatenada se pasa a una capa densa de salida con 10 neuronas y una función de activación softmax.

Prueba de la DenseNet:

```
test loss, test acc: [0.11047445237636566, 0.9754999876022339]
```

Se consiguen peor resultados que el modelo original.

VGG (Visual Geometry Group)

VGG es una arquitectura de red neuronal convolucional que utiliza capas de convolución y agrupamiento repetidas varias veces.

Al combinar VGG con MLP, se crea una red neuronal convolucional más compleja y potente que puede aprender características más complejas y abstractas de las imágenes, lo que mejora la precisión de la clasificación.

```
model =  Sequential()
model.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', input_shape = (28, 28, 1)))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Conv2D(64, kernel_size = (3,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
```

Primeramente, se agrega una capa de convolución 2D con 32 filtros y un tamaño de kernel de 3x3, utilizando la función de activación ReLU. Estas capas ya las he explicado anteriormente.

Se agrega una capa de agrupamiento (pooling) 2D con un tamaño de pool de 2x2. Explicada anteriormente también.

Luego se agrega otra capa de convolución 2D con 64 filtros y un tamaño de kernel de 3x3, utilizando la función de activación ReLU. Junto a esta, se agrega otra capa de agrupamiento 2D con un tamaño de pool de 2x2.

Se utiliza "Flatten()" para aplanar los mapas de características obtenidos en la última capa de convolución para convertirlos en un vector; es decir, convierte la imagen de características en un vector unidimensional

Se agrega una capa densa con 128 neuronas, utilizando la función de activación ReLU.

Por último, se agrega una capa de salida densa con 10 neuronas correspondientes a las 10 posibles clases de clasificación de las imágenes, utilizando la función de activación softmax para obtener una distribución de probabilidad sobre las clases.

Prueba de la DenseNet:

```
test loss, test acc: [0.02984323725104332, 0.9925000071525574]
```

Con este tipo de arquitectura sí que se consigue mejorar en gran medida la red neuronal, consiguiendo una tasa de precisión del 99,2%.

Conclusión

Una vez que he probado a cambiar todos los aspectos diferentes que afectan al comportamiento de la red, no he logrado mejorar significativamente la red en vanilla.

Pero usando la arquitectura VGG, si he podido llegar a mejorar la red.

3. Referencias bibliográficas

Backpropagation:

- https://neptune.ai/blog/backpropagation-algorithm-in-neural-networks-guide
- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
- https://stackoverflow.com/questions/3775032/how-to-update-the-bias-in-neural-network-backpropagation

MNIST:

- https://medium.com/metadatos/t%C3%A9cnicas-de-regularizaci%C3%B3n-b%C3%A1sicas-para-redes-neuronales-b48f396924d4
- https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/
- https://lamaquinaoraculo.com/computacion/redes-neuronales-recurrentes/
- https://paperswithcode.com/method/densenet
- https://datascientest.com/es/vgg-que-es-este-modelo-daniel-te-lo-cuenta-todo
- http://blog.hadsonpar.com/2021/08/crear-una-red-neuronal-convolucional-en.html