

### 3. Fundamentos de redes neuronales

#### 3.1 Introducción

Este capítulo está diseñado para guiarlos a través de los conceptos fundamentales y las estructuras que forman la base de las redes neuronales modernas. Al desentrañar estos elementos, adquirirán no solo una comprensión teórica, sino también práctica de cómo funcionan estas redes y cómo se pueden aplicar para resolver problemas complejos.

Comenzaremos nuestra exploración con el perceptrón, el bloque de construcción más básico de una red neuronal. Aquí, se sumergirán en la comprensión de los pesos y la combinación lineal, elementos esenciales que permiten a una red procesar y aprender de los datos de entrada. También abordaremos las funciones de activación, un componente crucial que añade la capacidad de modelar decisiones no lineales y complejas.

A continuación, avanzaremos hacia redes con varias salidas, explorando cómo se pueden construir para realizar tareas múltiples simultáneamente. Esta sección proporcionará una visión más profunda de cómo las redes neuronales pueden ser versátiles y potentes en el manejo de una variedad de desafíos en el procesamiento de datos.

Profundizaremos aún más en el tema al discutir las redes de varias capas. Estas estructuras más complejas permiten abordar problemas que una neurona simple no puede manejar, abriendo un mundo de posibilidades en términos de modelado y capacidad de predicción.

Finalmente, llegaremos al perceptrón multicapa, una forma de red neuronal que combina múltiples capas de perceptrones para formar una arquitectura capaz de aprender y modelar relaciones extremadamente complejas en los datos. Esta sección será crucial para entender cómo las redes neuronales se han convertido en una herramienta tan poderosa en el campo de la inteligencia artificial.

Cada tema está acompañado de ejercicios matemáticos detallados y ejemplos de programación en Python utilizando NumPy. Estos ejercicios están diseñados para solidificar su comprensión teórica y mejorar sus habilidades prácticas. Mediante la resolución de estos ejercicios, no solo fortalecerán su comprensión de los conceptos, sino que también desarrollarán una intuición práctica que es esencial para aplicar estas técnicas en el mundo real.

Al final de este capítulo, habrán adquirido no solo un conocimiento profundo de los fundamentos

de las redes neuronales, sino también la experiencia práctica necesaria para implementarlos en sus propios proyectos. Este conocimiento y habilidad son esenciales para cualquier aspirante a científico de datos, ingeniero de software o investigador en el campo de la inteligencia artificial.

### 3.2 Modelo McCulloch y Pitts

El modelo de McCulloch y Pitts, concebido por Warren McCulloch, neurocientífico, y Walter Pitts, lógico matemático, en 1943 [10], representa uno de los fundamentos teóricos de las redes neuronales y la inteligencia artificial. Este modelo es una simplificación abstracta de las neuronas biológicas, propuesta para entender cómo podrían las neuronas del cerebro generar patrones complejos de pensamiento a partir de operaciones simples.

En esencia, el modelo describe una neurona como una unidad de procesamiento binario, que recibe entradas, las procesa y luego produce una salida. Las entradas se conectan a los receptores exitatorios o inhibitorio de la neurona. Los receptores exitatorios contribuyen a que la neurona pueda producir una excitación, mientras que las inhibitorias mantienen la neurona apagada.

La neurona de McCulloch y Pitts (MyP) opera sumando las señales en los receptores exitatorios. Si la suma total de dichas señales alcanza un cierto umbral, la neurona se activa y emite una señal (1); de lo contrario, permanece inactiva (0).

Este modelo introdujo la idea de que las redes de tales neuronas podrían implementar cualquier función de cálculo lógico, al permitirles organizarse en circuitos que realizan operaciones lógicas básicas como AND, OR y NOT. Así, se estableció un puente entre la biología del cerebro y los principios de la computación, sugiriendo que redes complejas de estas unidades podrían teóricamente simular aspectos del pensamiento humano [10].

A pesar de su simplicidad y las limitaciones inherentes en comparación con la complejidad de las neuronas biológicas reales, el modelo de McCulloch y Pitts ha tenido un impacto duradero en el desarrollo de las redes neuronales artificiales estableciéndose como el primer modelo de una neurona artificial [14].

#### 3.2.1 Reglas de operación

Para formalizar el modelo de McCulloch y Pitts, representaremos una neurona con la letra  $C$ . La neurona se configura con tres elementos: la entrada exitatoria, la entrada inhibitoria y el umbral, es decir:

$$C = (E, I, u) \quad (3.1)$$

La entrada exitatoria,  $E = \{e_1, \dots, e_n\}$ , está compuesta por un conjunto de  $n$  elementos binarios mientras que la entrada inhibitoria contiene  $m$  elementos,  $I = \{i_1, \dots, i_m\}$ . Note que las entradas pueden también ser conjuntos vacíos. Tanto las entradas como la salida,  $s$ , están limitadas a valores binarios, es decir,  $e, i \in \{0, 1\}$ . Por su parte, el umbral  $u$  puede tomar valores naturales, es decir,  $u \in \mathbb{N}$ .

Dado lo anterior, la salida de  $C$  se calcula usando las siguientes reglas:

1. En caso de que alguna de las entradas inhibitorias esté activa la neurona no se excita, es decir,  $C(E, I, u) = 0 \quad \text{if } \exists i_i = 1; i_i \in I$
2. La neurona se excita si la integral de sus entradas exitatorias es igual o superior al umbral, es decir,  $C(E, I) = 1 \quad \text{if } \sum i_i \geq u; i_i \in I$
3. En cualquier otro caso la neurona permanece sin excitación.

Para representar las neuronas de forma gráfica utilizaremos la notación propuesta por Marvin Minsky [13], donde se utiliza un medio círculo para representar la neurona y una línea para su axón, además las entradas exitatorias se representan con flechas y las inhibitorias con líneas terminadas en círculo. Observe la figura 3.1

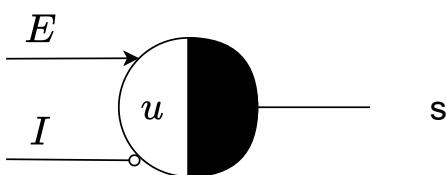
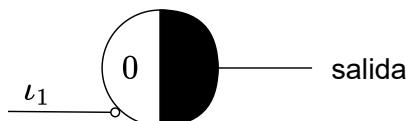


Figure 3.1: Diagrama general de una neurona de McCulloch y Pitts. Las entradas exitatorias se representan con una flecha. Las entradas inhibitorias se representan con líneas terminadas en un círculo. El umbral se representa en el centro del diagrama.

■ **Example 3.1** Suponga una neurona MyC que tiene cero entradas exitatorias, una inhibitoria y el umbral esta establecido en cero. Es decir,  $C = (E = \emptyset, I = \{i_1\}, u = 0)$ . Dibuje el diagrama correspondiente y determine la tabla de posibles salidas y analice el comportamiento.

SOLUCION: El diagrama de dicha neurona solo tiene una entrada conectada tal como se muestra a continuación:

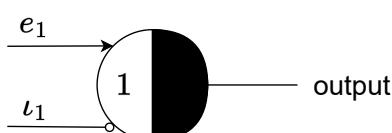


Para hallar las salidas usaremos los dos posibles valores de la entrada. Cuando  $i_1$  es igual a cero. La salida se calcula a partir de la sumatoria sobre  $E$ , como no hay elementos en  $E$  dicha sumatoria será cero. Esto se compara con el umbral,  $u$ , dado que la sumatoria iguala al umbral, la salida de la neurona es uno. Para el segundo caso, cuando  $i_1 = 1$ . La salida se ve inhabilitada y por tanto la salida sera cero. Esto se resumen en la tabla siguiente.

$i_1$	$s$
0	1
1	0

■

■ **Example 3.2** Calcule la tabla de salidas de una neurona de MyP definida por las entradas  $E = [e_1], I = [i_1]$  y un umbral  $u = 1$ . De acuerdo al diagrama siguiente:



SOLUCION: Utilizando las reglas de operación podemos observar que el único caso donde la neurona produce una salida es cuando la entrada inhibitoria esta en cero y la entrada exitatoria es igual a uno, dado que esta cantidad iguala al umbral establecido a uno. Por lo tanto, la tabla de salidas se escribiría como:

$e_1, i_1$	$s$
0, 0	0
0, 1	0
1, 0	1
1, 1	0

■

### 3.2.2 Parámetros

El modelo de MyP se puede configurar de diversas maneras. Si suponemos que dicho modelo tienen  $n$  entradas, el parámetro que podemos configurar es la selección de la conexión. Es decir, para cada entrada especificaremos si es una entrada exitatoria o inhibitoria. Por lo cual haremos  $n$

especificaciones. Adicionalmente debemos establecer el valor del umbral. Formalmente, para este modelo los parámetros se especifican como:

$$\theta = \{c_1, \dots, c_n, u\} \quad (3.2)$$

, donde  $c_i$  puede tomar uno de dos valores. Es decir  $c_i \in \{\text{'exitatoria'}, \text{'inhibitoria'}\}$  y  $n \in \mathbb{N}^+$ . Por lo tanto, podemos decir que el total de parámetros del modelo MyP es  $n + 1$ . Usualmente esto se escribe como:

$$|\theta| = n + 1 \quad (3.3)$$

### 3.2.3 Implementación

Para implementar una neurona de McCulloch y Pitts en una función de Python podemos seguir el enfoque codificado en el cuadro 3.1 y explicado a continuación. En caso de que exista al menos una entrada inhibitoria activa se debe retornar cero directamente. En caso contrario, se debe calcular la suma de las entradas exitatorias. Si la suma es mayor o igual que el umbral, la función retorna 1; de lo contrario la función retorna cero.

```
def neuronaMyP(E, I, u):
    for inhibitoria in I:
        if inhibitoria == 1:
            return 0

    integracion = 0
    for exitatoria in E:
        integracion = integracion + exitatoria

    if integracion >= u:
        return 1
    else:
        return 0
```

Table 3.1: Implementación en Python de la neurona de McCulloch y Pitts.

### 3.2.4 Ejercicios

**Exercise 3.1** Dada la neurona de MyP,  $C = (E = [x_1, x_2], I = [i_1], u = 1)$ , obtenga su diagrama gráfico y la tabla de salidas. ■

**Exercise 3.2** Diseñe una neurona de MyC de tal forma que cuando al menos dos de tres entradas se encuentren activas la salida se active. La salida debe estar apagada para cualquier otro caso. ■

### 3.2.5 Limitaciones del modelo

El modelo de McCulloch y Pitts fue una de las primeras aproximaciones teóricas para simular cómo las neuronas en el cerebro podrían operar en términos lógicos. Este modelo introdujo la noción de la neurona como un dispositivo binario, que realiza cálculos basados en funciones booleanas. A pesar de ser pionero, el modelo tenía varias limitaciones, lo que llevó a su reemplazo o evolución hacia modelos más sofisticados como el de Rosenblatt, denominado perceptrón, que estudiaremos en la siguiente lección.

El modelo de McCulloch y Pitts trataba a las neuronas como dispositivos extremadamente simples que solo podían disparar o no disparar y no incorporaba ninguna forma de aprendizaje o ajuste de los pesos sinápticos. Esto lo hacía poco flexible para simular procesos más complejos del cerebro.

El por su parte el modelo del perceptrón introdujo la idea de que una red de neuronas podía aprender ajustando los pesos de las conexiones sinápticas entre neuronas. Esto se lograba mediante un proceso de retroalimentación basado en ejemplos y correcciones en función de errores. Esto provocó que el modelo de McCulloch y Pitts perdiera aplicación práctica.

### 3.3 Perceptrón

El perceptrón es la unidad con la que se puede construir una red neuronal artificial. Éste se basa en el modelo lógico de umbral propuesto por Rosenblatt a finales de los años cincuentas [16]. De cierta forma, el perceptrón se asemeja a una neurona biológica (ver figura 3.2), que cuenta con dendritas para adquirir información, un núcleo para procesarla y un axón para enviarla. Sin embargo, en el perceptrón, la información se representa en forma de números, y el procesamiento se realiza mediante operaciones matemáticas básicas. Este modelo hipotético del sistema nervioso puede configurarse para separar correctamente dos clases contenidas en un conjunto de datos linealmente separable.

El perceptrón inicial fue un dispositivo físico construido con potenciómetros, sumadores, multiplicadores y demás dispositivos eléctricos [17]; y diseñado para reconocer patrones simples, como figuras geométricas. Este dispositivo utilizaba una matriz de celdas fotoeléctricas como "ojos" para observar los patrones, y ajustaba sus "pesos" internos en respuesta a los errores cometidos durante la clasificación.

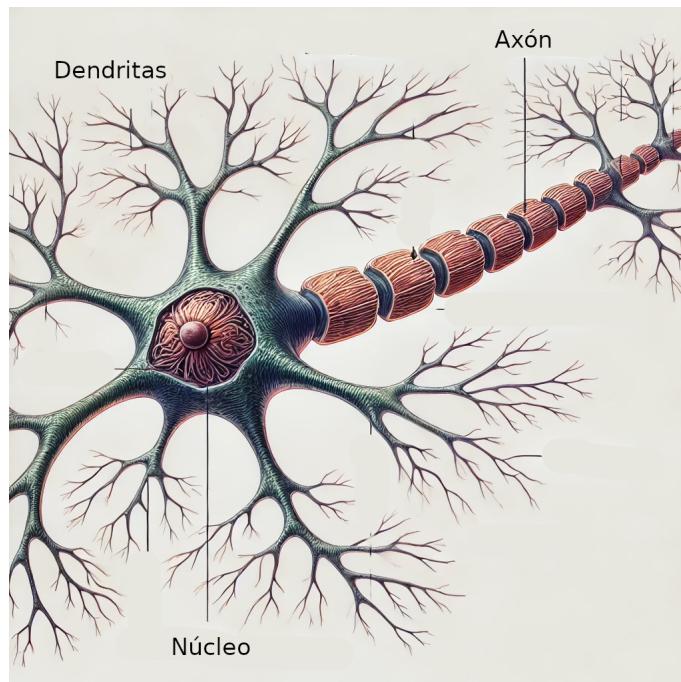


Figure 3.2: Diagrama simplificado de una neurona biológica.

### 3.3.1 Modelo del perceptrón

El perceptrón es una función que mapea un conjunto de valores de entrada reales a una de dos posibles clases; al igual que el modelo de MyP podemos decir que el perceptrón se activa o no. La figura 3.3 muestra los componentes del modelo: entrada ( $X$ ), pesos ( $W$ ), sesgo ( $b$ ), función de activación ( $f$ ) y salida ( $\hat{y}$ ).

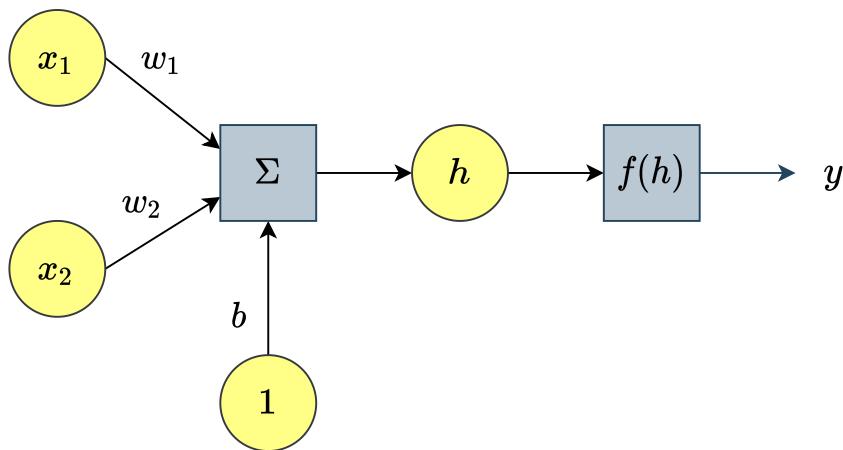


Figure 3.3: Representación gráfica extendida del perceptrón. Se pueden distinguir dos procesos la suma de la combinación de los pesos con las entradas y el mapeo usado por la función de activación.

La entrada del perceptrón,  $X$ , es un conjunto de valores cuyo dominio se encuentra en los reales. Dicha información puede provenir de sensores en tiempo real, bases de datos (*datasets*), entradas a mano, etc. Por ejemplo, para predecir si una casa se venderá, la entrada puede ser el conjunto de características de la casa junto con su precio. Formalmente,

$$X = [x_1, x_2, \dots, x_n]^T, \quad (3.4)$$

donde  $X$  es un vector que contiene  $n$  valores numéricos reales, es decir  $x \in \mathbb{R}$ .

La salida del perceptrón se calcula en dos etapas: una combinación lineal y una activación. En la combinación lineal, cada elemento de entrada tiene asociado un peso,  $w$ , que indica la importancia de esa entrada:

$$W = [w_1, w_2, \dots, w_n], \quad (3.5)$$

donde  $w \in \mathbb{R}$ , de tal forma que la combinación lineal de pesos y entradas se describe como una sumatoria de los productos:

$$\bar{h} = W \cdot X = \sum_{i=1}^n w_i \cdot x_i \quad (3.6)$$

Es común también adicionar a la ecuación 3.6 un valor de sesgo. El sesgo o *bias*,  $b$ , permite hacer desplazamientos a la función de activación. De forma similar a la ecuación de una recta  $y = mx + b$ ,  $m$  nos indica la pendiente que presenta la recta y el valor  $b$  indica cual será su desplazamiento; esto se verá aplicado de la misma forma. Por lo tanto, la forma canónica de la combinación lineal se escribe como:

$$h = W \cdot X + b = w_1 \cdot x_2 + \dots w_n \cdot x_n + b \quad (3.7)$$

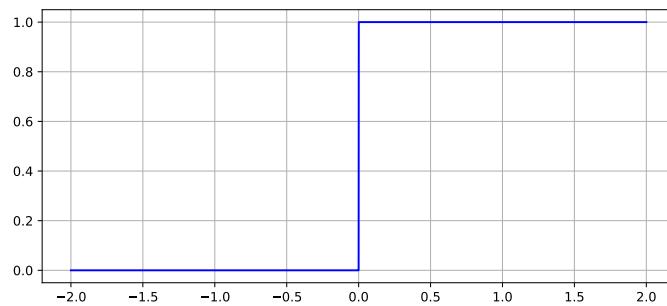


Figure 3.4: Gráfica de función escalón (*step*).

Si incorporamos una entrada adicional  $x_{n+1} = 1$  que se multiplique por el parámetro  $b$  tendríamos,

$$h = w_1 \cdot x_1 + \cdots + b \cdot 1 \quad (3.8)$$

esto nos lleva a una representación similar a la ecuación (3.6). Es por ello que en varias ocasiones durante el libro obviaremos el sesgo de las redes neuronales y únicamente escribiremos la combinación lineal como la ecuación (3.6).

En el segundo paso del cálculo de la salida, la función de activación es la encargada de determinar si se activa o no el perceptrón. Rosenblatt asumió que la salida presenta un comportamiento "todo o nada", por lo que utilizó la función escalón unitario (ver figura 3.4). Esta función se define de la siguiente forma:

$$f(h) = \begin{cases} 0 & \text{if } h < a \\ 1 & \text{if } h \geq a \end{cases}, \quad (3.9)$$

donde  $a$  es una constante que indica el valor de  $h$  en el que la función cambia de 0 a 1. Usualmente se considera que  $a = 0$ .

Por lo tanto, la salida es:

$$\hat{y} = f(h), \quad (3.10)$$

reemplazando (3.7) en (3.9) podemos escribir la salida,  $\hat{y}$ , como:

$$\hat{y} = f(WX + b) \quad (3.11)$$

Con lo establecido hasta el momento, es posible definir una formulación completa del comportamiento del perceptrón:

$$f(x_1, x_2, \dots, x_n) = \begin{cases} 0 & \text{if } \sum w_i \cdot x_i + b < 0 \\ 1 & \text{if } \sum w_i \cdot x_i + b \geq 0 \end{cases} \quad (3.12)$$

**■ Example 3.3** Suponga un perceptrón que acepta entradas de longitud tres y usa la función escalón. Si el perceptrón tiene configurados sus parámetros como  $W = [0.7, -0.3, 0.4]$  y  $b = 0$ . Calcule la salida cuando la entrada es  $X = [8, 5, 6]$ . Solución: primero se calcula el valor intermedio:

$$h = 0.7 \cdot 8 + (-0.3) \cdot 5 + 0.4 \cdot 6 + 0 = 6.5,$$

y a continuación se pasa por la función escalón para obtener la salida:

$$\hat{y} = f(6.5) = 1$$

■

```
def combinacion_lineal (x , w , b):
    suma = 0
    for w , x in zip (w , x ):
        suma = suma + w * x
    suma = suma + b
    return suma
```

Table 3.2: Implementación de la función  $h$ , también conocida como combinación lineal.

```
def escalon(h):
    if h >= 0:
        return 1
    else:
        return 0
```

Table 3.3: Código de la función escalón.

### Acerca de los parámetros

En el modelo de Rosenblatt los pesos y el sesgo se denominan parámetros y se representan como:

$$\theta = \{w_1, \dots, w_n, b\} \quad (3.13)$$

donde  $w_i \in \mathbb{R}$  y  $b \in \mathbb{R}$ . En el resto del libro un modelo que tiene un conjunto de parámetros específico se escribirá como la función:  $\Phi_\theta$ . Por lo anterior, podemos especificar que en un perceptrón simple con  $n$  entradas y una sola salida. La cantidad de parámetros es:

$$|\theta| = n + 1, \quad (3.14)$$

donde los símbolos  $||$  indican la cardinalidad del conjunto.

### 3.3.2 Selección de parámetros

Para que el perceptrón pueda tener diferentes comportamientos se requiere obtener un conjunto de pesos específico para cada comportamiento. Por ejemplo, para diferencias perros y gatos se requerirá un conjunto específico, el cual no será el mismo para diferenciar señales de tránsito. Al proceso de obtener los parámetros de forma automática se le denomina aprendizaje. El aprendizaje del perceptrón originalmente se realizaba mediante un algoritmo simple de ajuste de pesos, conocido como la regla de aprendizaje del perceptrón [3]. Este algoritmo ajusta los pesos del modelo en función de los errores cometidos en las predicciones. Si la predicción es correcta, los pesos se mantienen; si es incorrecta, los pesos se ajustan para mejorar la predicción en futuras iteraciones. El objetivo es encontrar un conjunto de pesos que permita separar los datos de entrada en dos categorías. Dado que actualmente la forma más común de entrenar los perceptrones es con el uso de descenso por gradiente, la regla de aprendizaje no se tratará en este libro. En cambio, en la sección 4.2, se abordarán los detalles del aprendizaje con el descenso por gradiente.

### 3.3.3 Implementación

Para la implementación de la función  $h$  haremos uso de la función `zip` que nos permitirá empaquetar cada entrada con su respectivo peso mientras se recorren los elementos. Veáse el código 3.2. Para la función de activación escalón simplemente usaremos condicionales.

De esta forma el proceso de inferencia frontal de un perceptrón se puede implementar de forma completa con el código del cuadro 3.4. En el código se implementa el perceptrón como una función la cual recibe entre los parámetros la función de activación. Esto es posible por que Python trata a

```

import numpy as np

def perceptron(W, X, b, activacion):
    h = combinacion_lineal(W, X, b)
    return activacion(h)

inputs = np.array([0.7, -0.3])
weights = np.array([0.1, 0.8])
bias = 0.5

# Pase frontal
activacion = escalon
output = perceptron(weights, inputs, bias, activacion)
print('Output : ')
print(output)

```

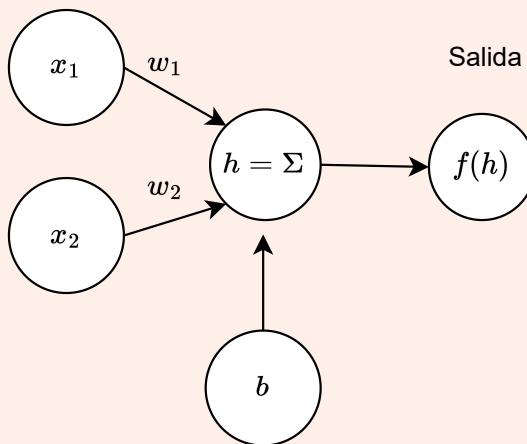
Table 3.4: Implementación del perceptrón

las funciones como objetos de primera clase. Esto significa que las funciones en Python pueden ser asignadas a variables, pasadas como argumentos a otras funciones, y retornadas desde otras funciones.

### 3.3.4 Ejercicios

**Exercise 3.3** Suponga el perceptrón de la siguiente figura, el cual recibe entradas de longitud igual a dos y cuya función de activación es la función escalón.

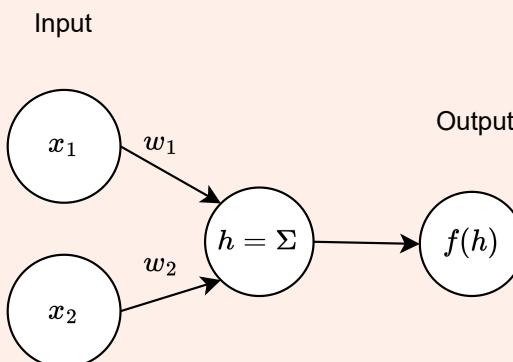
Entrada



1. Si  $W = [0.2, -0.5]$  y  $b = 0.3$ , determine la salida del perceptrón,  $\hat{y}$ , para las siguientes entradas:
  - (a)  $X^1 = [0.4, 0.8]$ ,
  - (b)  $X^2 = [0.2, 0.1]$ ,
  - (c)  $X^3 = [0.8, 0.7]$ ,
2. Determine la cantidad de parámetros que tiene el perceptrón.

**Exercise 3.4** A partir del siguiente perceptrón de dos entradas cuya función de activación,  $f(h)$ ,

es la función escalón y el sesgo  $b$  es nulo:



1. Suponga una entrada  $X = [0.8, 0.2]$  con pesos  $W = [1.0, 0.8]$ . Calcule el logit ( $h$ ) y la salida del perceptrón ( $\hat{y}$ ).
2. Suponga que tenemos un conjunto de datos de dos ejemplos de un determinado fenómeno. El conjunto contiene la entrada y el valor de salida de acuerdo a la siguiente tabla:

Ejemplo	$x_1$	$x_2$	Objetivo $y$
1	-0.5	-1	0
2	-1	0.5	1

Determine por prueba y error qué valores para  $W$  satisfacen lo asentado en dicha tabla.

■

### 3.3.5 Limitaciones del perceptrón

Aunque el perceptrón fue un avance significativo en su momento, pronto se encontró con limitaciones. En 1969, Marvin Minsky y Seymour Papert publicaron "Perceptrons" [12], un libro que demostraba matemáticamente que los perceptrones simples no podían resolver problemas no lineales, como el problema del XOR. Esto llevó a una reducción en el interés y la financiación de la investigación en redes neuronales durante varios años, un período conocido como el "invierno de la IA".

## 3.4 Redes neuronales simples

Una red neuronal simple es una extensión del perceptrón de Rosenblatt la diferencia radica en que ésta utiliza una función de activación continua y derivable que reemplaza a la función escalón. Por ejemplo, la función sigmoide fue la más utilizada en los primeros años. Teóricamente la función de activación debe ser derivable en todo el dominio sin embargo, como observaremos más adelante esta derivación puede sustituirse por una derivación numérica en los casos que la derivación analítica no está disponible. En esta sección, revisaremos algunas funciones de activación populares.

### 3.4.1 Funciones de activación

Las funciones de activación se pueden agrupar de forma general en dos conjuntos denominados comúnmente como:

- Funciones de activación discretas: Estas acotan los valores de salida de las neuronas a un conjunto finito de valores, generalmente usan valores binarios, 0 o 1.
- Funciones de activación continuas: De la misma forma, con estas es posible acotar los valores de salida y estos pueden tomar cualquier valor dentro del intervalo, generalmente los intervalos son  $[0, 1]$  o  $[-1, 1]$ . Dentro de este conjunto se pueden utilizar distintos tipos de

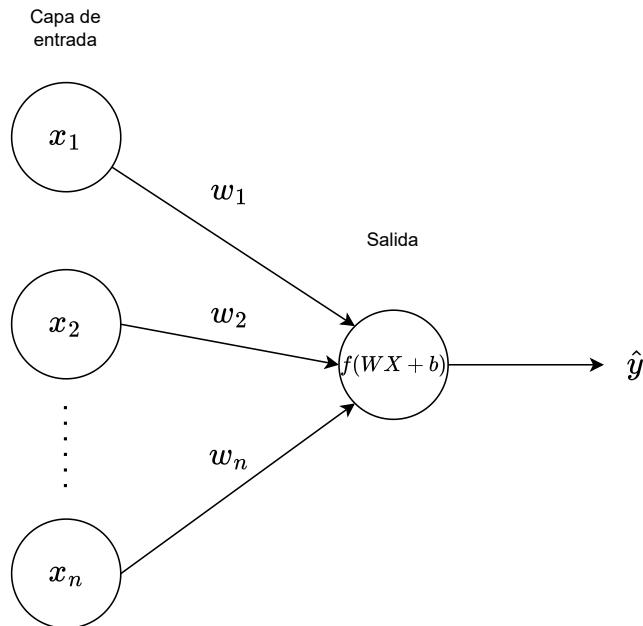


Figure 3.5: Diagrama simplificado de una neurona artificial. La capa de entrada esta compuesta de los elementos que componen el vector  $X$  y la capa de salida solo tiene un elemento. Note que a diferencia del diagrama del perceptrón. En este diagrama simplificado se condensan la combinación lineal y la función de activación en un mismo nodo.

funciones ya sean lineales<sup>1</sup> o no lineales<sup>2</sup>.

### Función lineal

Una de las funciones más sencillas a la que se proporciona un valor de entrada ( $x$ ) y a su salida ( $y$ ) produce un valor proporcional a la entrada. Ver figura 3.6. La función se escribe:

$$f_{lineal}(h) = h \quad (3.15)$$

---

<sup>1</sup>Funciones lineales en pytorch

<sup>2</sup>Funciones no lineales en pytorch



Figure 3.6: Función lineal

Una de las características más relevantes de estas funciones es que al colocar varias neuronas lineales en serie la salida siempre será un valor lineal. Desde una perspectiva más formal, las neuronas lineales únicamente pueden funcionar como regresores lineales [21].

### Función sigmoide

Comúnmente, el uso de funciones de activación no lineales y derivables es necesario para buscar soluciones a problemas más complejos que no pueden ser resueltos haciendo uso de funciones lineales. La función sigmoide se utiliza a menudo como una función de activación en las redes neuronales para introducir no linealidad en la red y esto a su vez permite que las neuronas de la red sean activadas de manera no lineal en función de la entrada, lo que les concede la capacidad de aprender patrones complejos en los datos de entrada. El hecho de que esta y otras funciones de activación sean derivables toma relevancia ya que algunas estrategias en el entrenamiento emplean algoritmos basados en la optimización de gradientes.

La sigmoide se calcula como:

$$f_{sigmoid}(h) = \frac{1}{1 + e^{-h}} \quad (3.16)$$

Uno de los inconvenientes que puede presentar esta función es que tiende saturarse en sus extremos, lo que puede dificultar el entrenamiento de las redes.

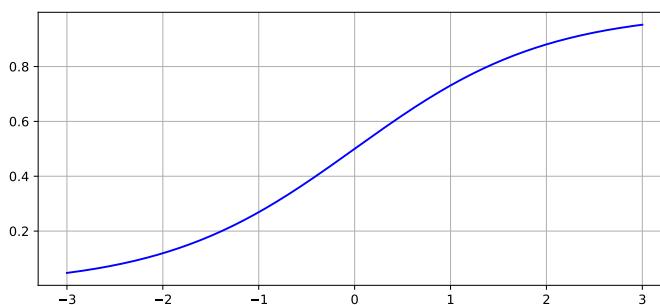


Figure 3.7: Función sigmoide

### Función tangente hiperbólica

La función tangente hiperbólica, también conocida como "tanh", es una función matemática no lineal que toma una entrada ( $h$ ) y produce una salida ( $y$ ) en el rango de -1 a 1 (ver figura 3.8). La función tanh se define como:

$$f_{tanh}(h) = \frac{e^h - e^{-h}}{e^h + e^{-h}} \quad (3.17)$$

A diferencia de la función sigmoide, la función tanh es simétrica alrededor del origen, lo que significa que tiene una salida negativa para entradas negativas y una salida positiva para entradas positivas. Esto puede ser útil en algunos casos en los que se desea una salida simétrica en torno a cero.

### Función activación lineal rectificada (ReLU)

La función de activación activación lineal rectificada (ReLU, por sus siglas en inglés *Rectified Linear Unit*) es una función no lineal, que toma una entrada ( $h$ ) y produce una salida ( $y$ ) de valor cero si la entrada es negativa, y si la entrada es positiva, devuelve la entrada misma, ver figura 3.9. La función ReLU se define como:

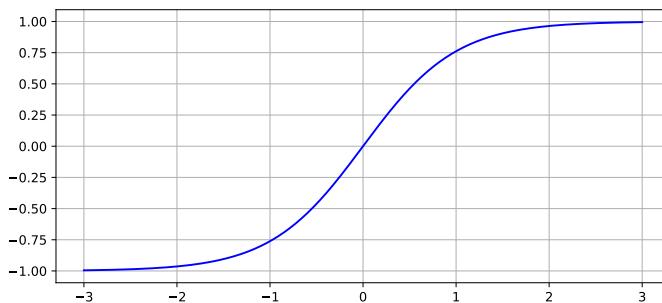


Figure 3.8: Función tangente hiperbólica

$$f_{ReLU}(h) = \begin{cases} 0 & \text{if } h < 0 \\ h & \text{if } h \geq 0 \end{cases} \quad (3.18)$$

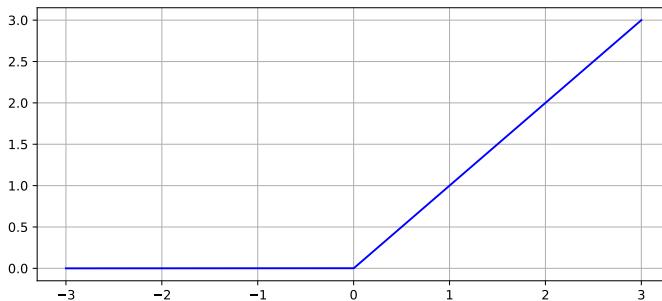


Figure 3.9: Funcion ReLU

Existen formas alternativas de definir la función ReLU:

$$f_{ReLU}(h) = h^+ = \max(0, h) \quad (3.19)$$

Una de las ventajas es que al no saturarse en sus extremos como la función sigmoide, permitiendo que fluya mayor cantidad de información a través de las neuronas; sin embargo, para los valores negativos esta función no permite que las neuronas se activen, que puede verse como una perdida de información.

### Función Leaky ReLU

La función Leaky ReLU es una variante de la función ReLU diseñada para solucionar uno de los problemas de ReLU, conocido como neurona muerta. Este problema ocurre cuando, para valores de entrada negativos, la ReLU devuelve cero, lo que puede llevar a que ciertas neuronas nunca se activen ni contribuyan al aprendizaje, ya que sus gradientes se vuelven cero.

En Leaky ReLU, en lugar de devolver cero para entradas negativas, se permite un pequeño valor negativo. La función es:

$$f_{\text{Leaky ReLU}}(x) = \max(\alpha x, x), \quad (3.20)$$

donde  $\alpha$  es un pequeño valor positivo (por ejemplo, 0.01) que se multiplica por los valores negativos de la entrada. Esto asegura que siempre haya un pequeño gradiente, incluso cuando la entrada sea negativa, evitando que las neuronas queden inactivas.

### Función de unidad lineal exponencial (ELU)

La función ELU (*Exponential Linear Unit*) es una variante de la función Leaky ReLU que introduce una no linealidad suave para los valores negativos. Su objetivo es mejorar el aprendizaje de redes profundas al reducir el problema de neurona muerta y hacer que la red sea más robusta.

La función ELU se define como:

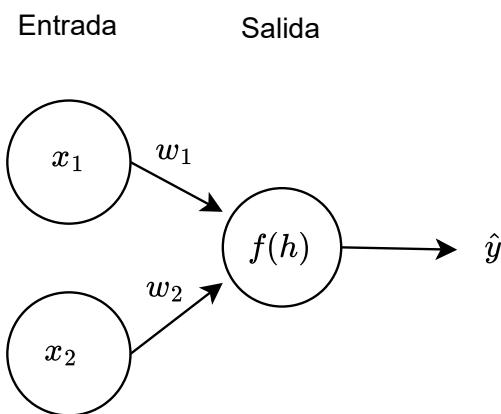
$$f_{\text{ELU}(x)} = \begin{cases} x, & \text{si } x > 0 \\ \alpha(e^x - 1), & \text{si } x \leq 0 \end{cases}, \quad (3.21)$$

donde  $\alpha$  es un parámetro positivo que controla la saturación para entradas negativas (usualmente,  $\alpha = 1$ ).

Esta función tiene ventaja sobre ReLU en ciertos escenarios donde las redes profundas son propensas a problemas de gradientes o inestabilidad en las activaciones.

#### 3.4.2 Ejemplos

■ **Example 3.4** A partir del diagrama siguiente:



donde  $W = [0.9, -0.5]$ ,  $b = -0.6$  y la función de activación es una función tangente hiperbólica. Calcule las salidas de la red para el siguiente conjunto de datos:

ID	$x_1$	$x_2$
1	0	3
2	1	5
3	7	6

SOLUCION: Para cada uno de los ejemplos en el conjunto se realiza:

$$\hat{y}^i = f_{\tanh}(w_1 \cdot x_1^i + w_2 \cdot x_2^i + b)$$

lo que implica que para el primer ejemplo,  $i = 1$ , se calcule como:

$$\hat{y}^1 = f_{\tanh}(w_1 \cdot x_1^1 + w_2 \cdot x_2^1 + b)$$

por tanto,

$$\hat{y}^1 = f_{\tanh}(0.9 \cdot 0 + (-0.5) \cdot 3 - 0.6)$$

$$\hat{y}^1 = -0.635$$

```
def sigmoide(h):
    result = 1/(1+np.exp(-h))
    return result
```

Table 3.5: Implementación de la función sigmoide en Python.

```
def combinacion_lineal (X , W , b):
    h = np.dot(W,X) + b
    return h

def neurona(W, X, b, activacion):
    h = combinacion_lineal(W, X, b)
    return activacion(h)
```

Table 3.6: Implementación de una neurona artificial simple en Python.

Siguiendo la misma metodología obtenemos para los ejemplos restantes:

$$\hat{y}^2 = -0.975$$

$$\hat{y}^3 = 0.993$$

■

### 3.4.3 Implementación

Una forma de implementar una neurona simple es mediante el código 3.6 que separa los cálculos en funciones para la combinación lineal y la función de activación. Con respecto a la combinación lineal se usa la función dot de NumPy; ésta función realiza un producto escalar entre dos *arrays*. Es importante destacar que el comportamiento de esta función varía ligeramente dependiendo de la dimensión y la forma de los *arrays* con los que se trabaja. Para vectores 1D, numpy.dot devuelve el producto escalar (también conocido como producto punto) de los vectores. Por ejemplo, si a y b son dos vectores 1D, numpy.dot(a, b) calculará el producto escalar de a y b. Para matrices 2D (y arrays multidimensionales), numpy.dot realiza una multiplicación de matrices. En este caso, el número de columnas de la primera matriz debe ser igual al número de filas de la segunda matriz. Por ejemplo, si A es una matriz de dimensiones ( $m \times n$ ) y B es una matriz de dimensiones ( $n \times p$ ), entonces numpy.dot(A, B) resultará en una nueva matriz de dimensiones ( $m \times p$ ). Para combinaciones de vectores y matrices, la función dot aplicará las reglas correspondientes de multiplicación vector-matriz o matriz-vector, dependiendo del orden en que se proporcionen los vectores y las matrices.

La función de activación se deja libre en el código 3.6. Sin embargo, como ejemplo se provee la función sigmoide en el código 3.5. Finalmente, la implementación se prueba en el código 3.7.

### 3.4.4 Ejercicios

**Exercise 3.5** Usando la red simple del siguiente diagrama:

```
import numpy as np

def main():
    print("Red neuronal simple")

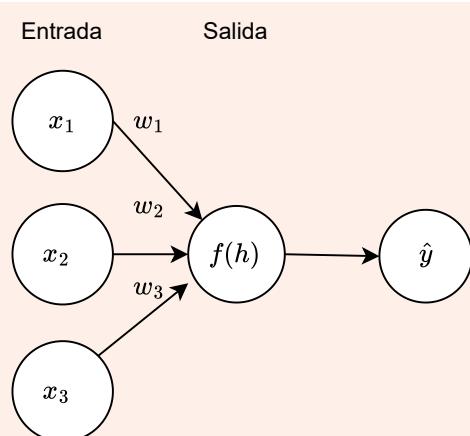
    entradas = np.array([1, 1])
    print("X: ", entradas)
    pesos = np.array([.2, .2])
    print("W: ", pesos)
    sesgo = 1
    print("b: ", sesgo)

    # inferencia o pase frontal
    activacion = sigmoide
    output = neurona(pesos, entradas, sesgo, activacion)

    print("Resultado: ", output)

if __name__ == "__main__":
    main()
```

Table 3.7: Programa que verifica el funcionamiento de una neurona artificial simple.



donde

$$W = [1.0, -0.9, 0.4], X = [4.0, 6.0, 8.0]^T, b = 0.1.$$

Responda las siguientes preguntas.

1. Calcule la salida de la red,  $\hat{y}$ , suponiendo como función de activación la función sigmoide.
  2. Cambie la función de activación por una función tangente hiperbólica y calcule de nuevo la salida.
  3. Cambie las entradas al siguiente vector y aplique la función ReLU.

$$X = \begin{bmatrix} 5.0 \\ 6.0 \\ 4.0 \end{bmatrix}$$

**Exercise 3.6** Implemente en python y numpy una red neuronal simple que utilice como función de activación:

1. la función tangente hiperbólica.
2. la función ELU.

**Exercise 3.7** Modifique el código 3.7 para que el vector de entrada sea introducido por el usuario (el tamaño y los valores deben ser introducidos por el usuario) y los pesos sean inicializados de forma aleatoria.

### 3.5 Redes neuronales de varias salidas

Una red neuronal simple con varias salidas se forma de dos o más neuronas que comparten la misma entrada y estas a su vez tienen su propia salida. Estas arquitecturas de redes se limitan a recibir las señales de entrada y redistribuyéndolas a la salida. La figura 3.10 muestra un diagrama extendido de la red de varias salidas para un ejemplo de tres entradas y dos salidas.

Las redes neuronales de una sola capa con varias salidas, a pesar de su simplicidad estructural, albergan una serie de ventajas que las hacen adecuadas para ciertos contextos y aplicaciones. Estas redes, por su construcción, pueden abordar simultáneamente múltiples tareas de predicción, lo que resulta especialmente útil cuando se requieren varias respuestas a partir de un mismo conjunto de datos de entrada. Imagina, por ejemplo, un sistema que, a partir de datos meteorológicos, predice no solo la probabilidad de lluvia, sino también la temperatura y la humedad; este sería un escenario ideal para implementar una red de este tipo.

Podemos descomponer la red en dos capas: la capa de entrada es la correspondiente al vector de entrada  $X$  y una capa de salida,  $\hat{Y}$ , que calcula las predicciones. Cada nodo de entrada,  $x_i$ , se conecta a cada neurona de salida  $\hat{y}$  a través de un peso  $w_{i,j}$ . A esta configuración se le denomina capa completamente conectada (*fully connected*) y más tarde la usaremos como bloque de construcción para las redes de varias capas.

Esta red tiene un conjunto de pesos, un conjunto de sesgos y una función de activación. Los pesos ahora los representaremos con matrices de la siguiente forma:

$$W = \begin{bmatrix} w_{1,1} & \dots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \dots & w_{m,n} \end{bmatrix} \quad (3.22)$$

donde  $m$  es la cantidad de entradas y  $n$  la cantidad de salidas. Con respecto a los sesgos se requiere uno por cada salida y se especifican como:

$$B = [b_1, \dots, b_n] \quad (3.23)$$

De esta forma el valor intermedio se especifica como:

$$H = XW + B \quad (3.24)$$

donde,  $X$  representa el vector de entradas. Note que la ecuación (3.24) invierte  $X$  y  $W$  con respecto a la definición de la neurona simple. Esto se requiere debido a la nueva definición de los pesos como matriz, delo contrario no es posible la multiplicación matricial.

La capa de salida la podemos escribir de forma extendida como:

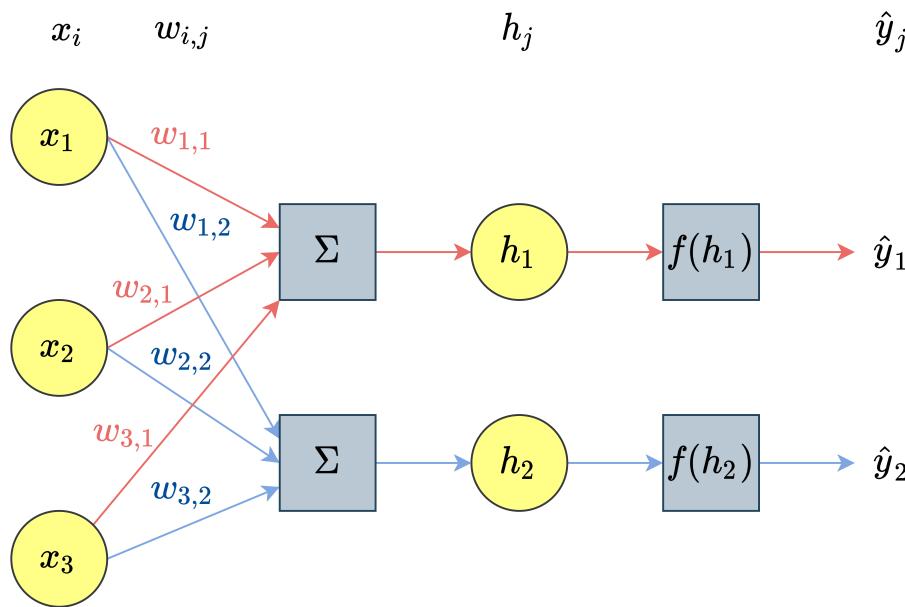


Figure 3.10: Diagrama extendido de una red neuronal de varias salidas.

$$\hat{y}_j = f \left( \sum_{i=1}^m x_i \cdot w_{i,j} + b_j \right), \quad (3.25)$$

donde  $\hat{y}_j$  es la salida de la  $j$ -ésima neurona. O podemos escribir de forma simplificada usando notación matricial:

$$\hat{Y} = f(H) = \begin{bmatrix} f(h_1) \\ \vdots \\ f(h_n) \end{bmatrix} \quad (3.26)$$

ahora, la función de activación recibe el vector intermedio, y retorna las activations o logits. La función retorna tantas activations como entradas haya tenido, este comportamiento se conoce como *broadcasting* de una función. Es observable de la ecuación anterior que la misma función de activación se aplica a todas las salidas.

### 3.5.1 Parámetros

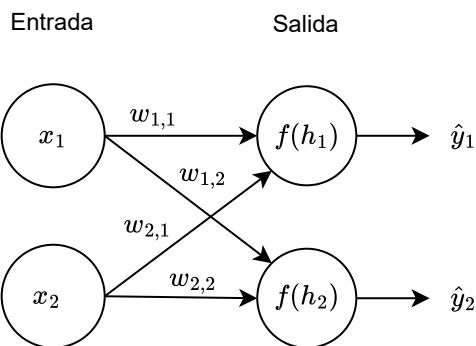
Con respecto a la cantidad de parámetros esta se incrementará tantas veces como salidas tenga la red. Por lo tanto, para las redes neuronales con varias salidas la cantidad de parámetros se calcula como:

$$|\theta| = (m+1)n, \quad (3.27)$$

donde  $m$  es la cantidad de nodos de entrada y  $n$  la de nodos de salida.

### 3.5.2 Ejemplos

■ **Example 3.5** Suponga una red neuronal de una capa con dos nodos de entrada y dos salidas de acuerdo al siguiente diagrama:



donde

$$X = [8, 4],$$

$$W = \begin{bmatrix} -0.1 & 0.6 \\ -0.8 & 0.3 \end{bmatrix}$$

$$b = [-0.7, 0.3]$$

Calcule la salida de la red.

SOLUCION: Siguiendo la ecuación (3.24) obtenemos:

$$H = [8, 4] \begin{bmatrix} -0.1 & 0.6 \\ -0.8 & 0.3 \end{bmatrix} + [-0.7, 0.3]$$

$$H = [?, ?]$$

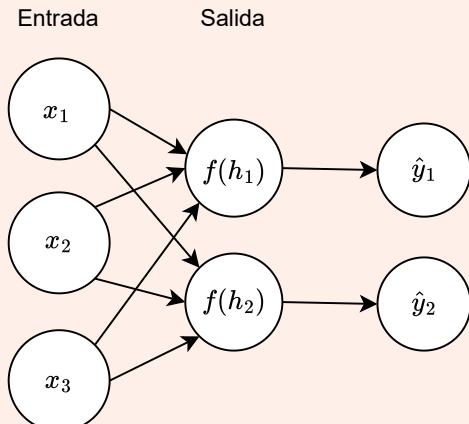
y pasando por la función de activación, ecuación (3.26)

$$\hat{Y} = [?, ?]$$

■

### 3.5.3 Ejercicios

**Exercise 3.8** Tomando en cuenta el siguiente diagrama de red neuronal de una capa con tres entradas y dos salidas sin sesgos:



, donde

$$X = [0, 4, 1],$$

$$W = \begin{bmatrix} 0.5 & 0.6 \\ 0.9 & 0.5 \\ 0.3 & 0.9 \end{bmatrix},$$

Calcule el vector de predicciones,  $\hat{Y}$ ,

1. usando una función de activación sigmoide
2. usando una función de activación ReLU

**Exercise 3.9** Diseñe una red neuronal que a un solo valor de entrada obtenga en la salida  $\hat{Y} = [2x, -5x]$ . Es decir, especifique los parámetros  $W, B$  y la función de activación a utilizar. ■

## 3.6 Redes neuronales multicapa

Las redes neuronales que estudiamos previamente; denominadas de una sola capa, enfrentan una serie de limitaciones debido a su arquitectura simplificada. Principalmente, su habilidad para procesar información se ve restringida a funciones y relaciones lineales. Esto significa que si los datos o el problema que se intenta resolver presentan una complejidad que requiere el discernimiento de patrones no lineales, el perceptrón simple se queda corto. Su estructura no le permite modelar la rica diversidad de relaciones que podrían existir entre los datos de entrada y salida, limitando su utilidad a escenarios donde la relación es directa y sin complicaciones. Esta característica intrínseca las hace inadecuados para tareas más complejas del aprendizaje automático, como el reconocimiento de imágenes o el procesamiento del lenguaje natural, donde la habilidad para capturar y modelar interacciones no lineales entre los datos es crucial.

En esta sección, extenderemos las redes neuronales para que puedan tener varias capas de profundidad. Las redes neuronales multicapa, conocidas también como perceptrones multicapa (*MLP* por sus siglas en inglés) [3], son una potente evolución en el campo de la inteligencia artificial y el aprendizaje automático, ofreciendo ventajas clave que superan ampliamente las capacidades de los modelos más simples de una sola capa. Una de sus principales fortalezas radica en su habilidad para capturar y aprender patrones no lineales complejos, algo esencial para abordar una amplia gama de problemas del mundo real que van desde el reconocimiento de imágenes hasta la comprensión del lenguaje natural. Esta capacidad se debe a su arquitectura profunda, compuesta por múltiples capas de neuronas interconectadas, donde cada capa puede extraer y refinar características de los datos de entrada, construyendo una representación cada vez más rica y detallada a medida que la información avanza a través de la red.

### 3.6.1 Capas ocultas

Las capas ocultas como se muestra en la figura 3.11, son aquellas que están contenidas entre las capas de entrada y salida en una red neuronal. Cada capa oculta contiene: una matriz de pesos, un vector de sesgos y una función de activación que se aplica a todas las neuronas de dicha capa.

Como se presentó anteriormente, los pesos pueden estar contenidos en un vector; ahora en los perceptrones multicapa, existen múltiples conexiones sinápticas entre las distintas capas de la red, mismos que se pueden contener en un matriz y estos se etiquetan de la siguiente forma:  $w_{i,j}$  donde  $i$  representa la unidad de entrada y  $j$  denota el perceptrón al que esta asociada esa conexión <sup>3</sup>.

<sup>3</sup>El tamaño de la matriz de pesos es dado por el numero de entradas  $m$  y el numero de neuronas siguientes a las que se conecta  $n$ , así la matriz de pesos tendrá dimensiones  $m \times n$

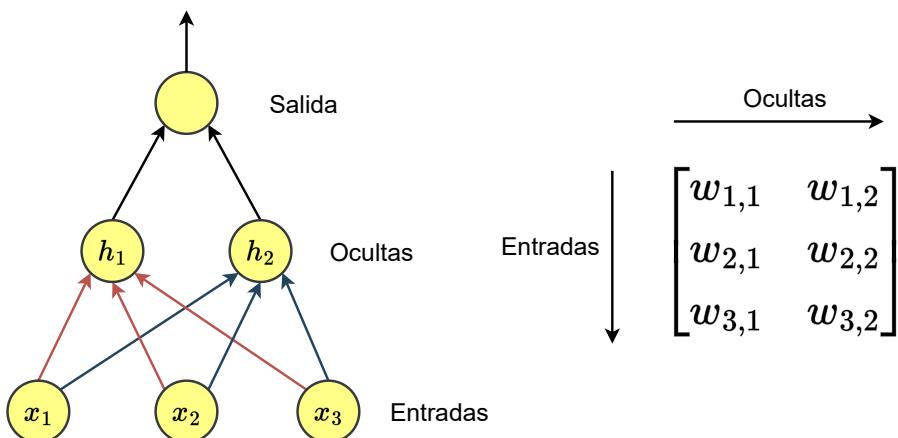


Figure 3.11: Red neuronal multicapa (izquierda) y matriz de pesos (derecha)

$$W_k = \begin{bmatrix} w_{1,1} & \dots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \dots & w_{m,n} \end{bmatrix} \quad (3.28)$$

Nótese que se agrega el super-índice  $k$  a la matriz de pesos para indicar la capa oculta. De la misma forma el sesgo ahora se especifica como:

$$B_k = [b_1, \dots, b_n] \quad (3.29)$$

De esta forma el valor intermedio se especifica como:

$$H_k = A_{k-1}W_k + B_k \quad (3.30)$$

donde,  $A_{k-1}$  representa la salida de la capa anterior.

La función de activación ahora recibe un vector de predicciones y retorna las activations, tantas como entradas haya tenido.

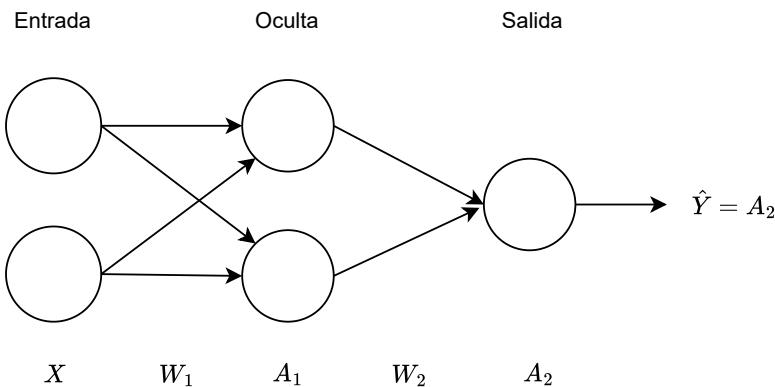
$$A_k = f_k(H_k) \quad (3.31)$$

Note que en la ecuación (3.31) la función de activación se complementa con un superíndice, esto implica que en cada capa se puede usar una función de activación distinta.

Para la capa de salida el valor de  $A$  se convierte en la predicción. Es decir si la capa  $k$ -ésima es la salida entonces:

$$\hat{Y} = A_k \quad (3.32)$$

■ **Example 3.6** Partiendo del siguiente diagrama de red neuronal multicapa con una capa oculta:



y usando los siguientes parámetros:

$$W_1 = \begin{bmatrix} 0.3 & 0.2 \\ -0.8 & 0.9 \end{bmatrix}$$

$$B_1 = [-0.8, 0.6]$$

$$f_1 = f_{\text{sigmoide}}$$

$$W_2 = \begin{bmatrix} -0.2 \\ 0.7 \end{bmatrix}$$

$$B_2 = [-0.1]$$

$$f_2 = f_{\text{ReLU}}$$

Calcule la salida de la red si se introduce:

$$X = [1, 2]$$

Solución: Usando,  $A_0 = X$ , y las ecuaciones (3.30) (3.31) se calcula

$$H_1 = [1, 2] \begin{bmatrix} 0.3 & 0.2 \\ -0.8 & 0.9 \end{bmatrix} + [-0.8, 0.6]$$

$$H_1 = [-2.1, 2.6]$$

$$A_1 = f_{\text{sigmoide}}([-2.1, 2.6])$$

$$A_1 = [0.109, 0.930]$$

lo que nuevamente se usa para calcular

$$H_2 = [0.109, 0.930] \begin{bmatrix} -0.2 \\ 0.7 \end{bmatrix} + [-0.1]$$

$$H_2 = [0.529]$$

$$A_2 = f_{\text{ReLU}}([0.529])$$

$$A_2 = [0.5292]$$

Finalmente,

$$\hat{Y} = A_2$$

$$\hat{Y} = [0.5292]$$

■

### 3.6.2 Implementación

En el código 3.8 se implementa una red neuronal multicapa de una capa oculta. El código utiliza NumPy, una librería de Python que facilita la manipulación de matrices y operaciones matemáticas.

El código define una red neuronal con las siguientes características:

```
N_input = 4
N_hidden = 3
N_output = 2
```

que indican el número de neuronas en cada capa. El tamaño de las capas es importante para determinar las dimensiones de las matrices de pesos que conectan las capas.

Los pesos entre las capas se inicializan aleatoriamente usando una distribución normal con una media de 0 y una desviación estándar de 0.1. Existen dos matrices de pesos:

```
W_1 = np.random.normal(mean, scale=stdev, size=(N_input, N_hidden))
W_2 = np.random.normal(mean, scale=stdev, size=(N_hidden, N_output))
```

#### Pase frontal

Entrada a la red: Se define una entrada aleatoria  $X$ , que tiene cuatro valores correspondientes a las cuatro neuronas de la capa de entrada.

Cálculo de la capa oculta: Multiplicación de entrada por pesos: El producto entre la entrada  $X$  y la matriz de pesos  $W_1$  se calcula usando la función **np.dot**. Este producto da como resultado una activación lineal en la capa oculta  $H_1$ .

```
H_1 = np.dot(X, W_1)
```

Función de activación: Luego, a esta activación lineal  $H_1$  se le aplica la función de activación sigmoide, lo que da lugar a las salidas activadas de la capa oculta  $A_1$ . Este paso no solo ayuda a introducir no linealidad, sino que también normaliza los valores.

```
A_1 = sigmoid(H_1)
```

El proceso se repite para la siguiente capa ahora usando  $A_1$  como entrada.

```
H_1 = np.dot(A_1, W_2)
H_1 = np.dot(A_1, W_2)
Y_prediccion = sigmoid(H_1)
```

Finalmente, el código imprime los resultados.

### 3.6.3 La base de muchos avances recientes

A pesar de que el perceptrón multicapa es una arquitectura con una larga historia en el campo de la inteligencia artificial, su importancia no ha mermado y ahora es base fundamental dentro del aprendizaje profundo. Su aplicación se extiende a diversos dominios, incluyendo su rol crucial en la etapa de clasificación de las redes neuronales convolucionales (CNNs)[8], donde contribuye a interpretar las características visuales extraídas para la identificación de objetos. Además, se integra dentro de los avanzados mecanismos de atención en los modelos de transformers [20],

```

def main():
    # Ejemplo de red multicapa implementada en Python

    # Tamano de la red
    N_input = 4
    N_hidden = 3
    N_output = 2

    #Definir matrices de pesos, inicializados de forma aleatoria
    mean = 0.0
    stdev = 0.1
    W_1 = np.random.normal(mean, scale=stdev, size=(N_input, N_hidden))
    W_2 = np.random.normal(mean, scale=stdev, size=(N_hidden, N_output))

    # Probaremos con una entrada aleatoria
    X = np.random.randn(4)

    # Ejecutar pase frontal de la red
    H_1 = np.dot(X, W_1)
    A_1 = sigmoid(H_1)

    print('Salida de la capa oculta:')
    print(A_1)

    H_2 = np.dot(A_1, W_2)
    Y_prediccion = sigmoid(H_2)

    # Imprimir prediccion
    print('Prediccion de la red:')
    print(Y_prediccion)

if __name__ == "__main__":
    main()

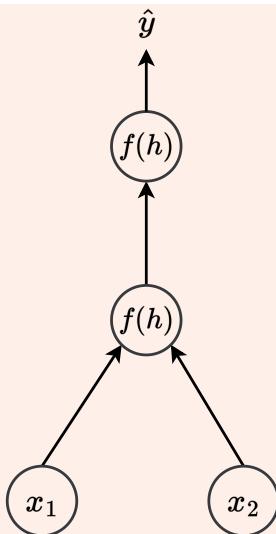
```

Table 3.8: Implementación de una red neuronal de una capa oculta. La implementación se hace de forma estructurada usando Python y NumPy.

facilitando procesos complejos como la comprensión y generación de lenguaje natural. Otro ámbito de aplicación innovador es en el desarrollo de representaciones implícitas, como se ejemplifica en las *Neural Radiance Fields* (NeRF) [11], donde el perceptrón multicapa ayuda a modelar escenas 3D complejas con un nivel de detalle y realismo impresionantes. Esta versatilidad demuestra que, lejos de quedar obsoleto, el perceptrón multicapa sigue adaptándose y encontrando nuevas aplicaciones en la vanguardia de la investigación y el desarrollo tecnológico.

### 3.6.4 Ejercicios

**Exercise 3.10** Tomando en cuenta la siguiente red neuronal multicapa:



donde:

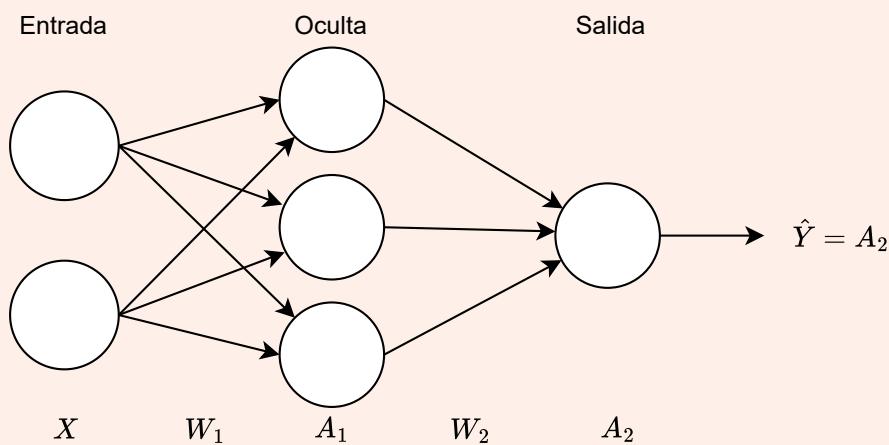
$$X = [0.1, 0.2], W_1 = \begin{bmatrix} -0.5 \\ 0.4 \end{bmatrix}, W_2 = [0.3], B_1 = [-1], B_2 = [8]$$

y suponiendo funciones de activación:

$$f_1 = \text{sigmoid}, f_2 = \tanh$$

1. Calcule la salida de la capa oculta,  $A_1$ .
2. Calcule la predicción de la red,  $\hat{Y}$ .

**Exercise 3.11** Suponga una red neuronal de dos entradas, una capa intermedia de tres neuronas y una salida de dos. Tal como se ilustra en la siguiente figura:



Donde,

$$W_1 = \begin{bmatrix} -0.5 & -0.2 & 0.2 \\ 0.4 & -0.1 & 0.9 \end{bmatrix}, W_2 = \begin{bmatrix} 0.3 \\ 0.0 \\ 0.5 \end{bmatrix}, B_1 = [-0.8, -0.5, 0.0], B_2 = [-0.2]$$

y suponiendo funciones de activación:

$$f_1 = \text{ReLU}, f_2 = \text{sigmoide}$$

Determine las predicciones de la red para el siguiente conjunto de datos de entrada:

X	Y
[0.4, -0.9]	0
[0.9, 0.8]	1

■

### 3.7 Acordeón de redes neuronales

#### Definiciones:

Fórmula de la neurona simple

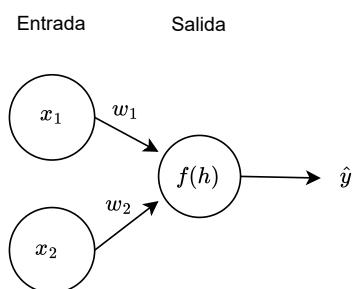
$$\hat{y} = f(WX + b)$$

Forma canónica de la combinación lineal:

$$h = W \cdot X + b = w_1 \cdot x_2 + \dots w_n \cdot x_n + b$$

#### Arquitecturas:

Red simple:



Red multiples salidas: Red multicapa:

#### Funciones de activación:

Lineal

$$f_{linear}(x) = x$$

Escalón

$$f_{step}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Sigmoide

$$f_{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Tangente hiperbólica

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

Rectified Linear Unit (ReLU)

$$f_{relu}(x) = \max(0, x)$$