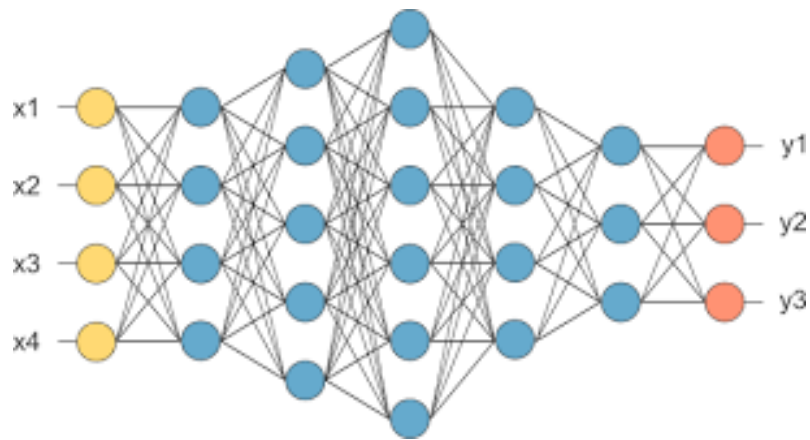


Unidad 4. Redes neuronales artificiales



“Programación de Inteligencia Artificial”
Curso de Especialización en Inteligencia Artificial
y Big Data
-
IES Serra Perenxisa

Contenido

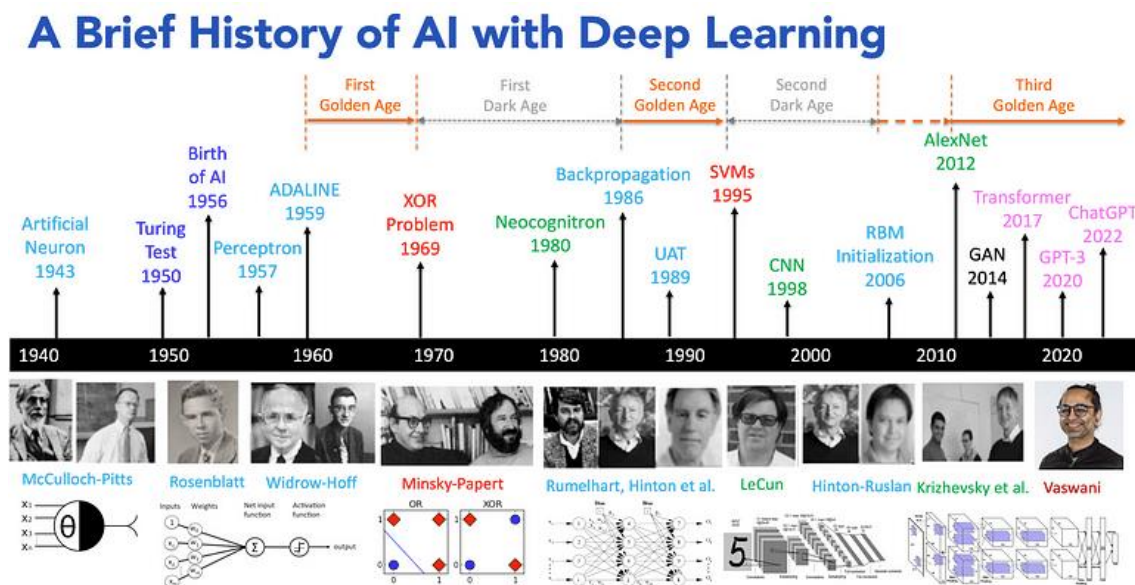
1. Introducción a las redes neuronales artificiales	3
1.1. Breve historia y evolución de las redes neuronales	3
1.2. Motivación, casos de uso y aplicaciones en la industria	3
2. Conceptos básicos de las redes neuronales	5
2.1. Neurona artificial: estructura y función	5
2.2. Funcionamiento de una red neuronal simple	6
3. Arquitecturas de redes neuronales	8
3.1. Capas de entrada, ocultas y de salida	9
3.2. Tipos de arquitecturas (shallow vs. deep networks)	10
4. Funciones de activación	11
4.1. Listado de funciones de activación	11
4.2. Selección de la función de activación por capa	12
5. Proceso de entrenamiento de redes neuronales	13
6. Regularización y prevención del overfitting	18
7. Evaluación del rendimiento de redes neuronales	21
7.1. Métricas de evaluación comunes	21
7.2. Visualización de la convergencia y curvas de aprendizaje	22
7.3. Técnicas para mejorar el rendimiento y ajuste de hiperparámetros	23
8. Implementación práctica de una red neuronal simple	25
8.1. TensorFlow y Keras	25
8.2. Construcción de una red básica para un problema de clasificación	25
8.3. Clases, métodos y parámetros de keras	29
8.4. Guardado y carga de modelos entrenados	31
9. Retos y limitaciones de las redes neuronales	32
9.1. Problemas de <i>vanishing</i> y <i>exploding</i> gradients	32
9.2. Coste computacional y necesidad de recursos	33
9.3. Redes neuronales profundas y Deep Learning	33
10. Tipos de redes neuronales	35

1. Introducción a las redes neuronales artificiales

1.1. Breve historia y evolución de las redes neuronales

Desde sus inicios en la década de 1940, las redes neuronales artificiales han evolucionado significativamente. El modelo de McCulloch y Pitts marcó el primer intento de representar el funcionamiento de una neurona mediante un enfoque matemático, estableciendo las bases teóricas para esta tecnología. Más adelante, en 1958, Frank Rosenblatt desarrolló el perceptrón, una red neuronal simple capaz de realizar tareas de clasificación lineal. Sin embargo, las limitaciones de estos primeros modelos, como la incapacidad del perceptrón para resolver problemas no linealmente separables (como el problema de XOR), ralentizaron el progreso en este campo.

En los años 80, la introducción del algoritmo de retropropagación del error por Rumelhart, Hinton y Williams supuso un avance crucial, permitiendo entrenar redes multicapa y superar limitaciones anteriores. A pesar de esto, el desarrollo de las redes neuronales enfrentó un período de estancamiento debido a la falta de poder computacional y datos suficientes. No fue hasta el siglo XXI, con el auge del big data, los avances en hardware (como GPUs) y la aparición de arquitecturas más avanzadas como las redes convolucionales (CNN) y recurrentes (RNN), que las redes neuronales experimentaron un renacimiento, convirtiéndose en una herramienta esencial en inteligencia artificial y aprendizaje automático.



1.2. Motivación, casos de uso y aplicaciones en la industria

Las redes neuronales artificiales se inspiran en el funcionamiento del cerebro humano, imitando la forma en que las neuronas biológicas procesan información.

Una de sus principales ventajas es la capacidad de aprender representaciones útiles directamente de los datos, eliminando la necesidad de un diseño manual exhaustivo de características. Gracias a esta flexibilidad, han sido clave en dominios como la visión por computador, el procesamiento del lenguaje natural (NLP) y el análisis de series temporales, donde las relaciones subyacentes son altamente complejas y multidimensionales.

Optimizadas para entornos computacionales y potenciadas por avances en hardware y algoritmos de optimización, estas redes han transformado numerosas industrias con innumerables aplicaciones: el reconocimiento facial, la detección de objetos y el diagnóstico médico precoz mediante el análisis de imágenes como radiografías y resonancias magnéticas, la comprensión y generación de lenguaje natural, traducción automática, análisis de sentimientos y el desarrollo de chatbots avanzados. Su capacidad para abordar problemas complejos y su adaptabilidad las convierten en herramientas imprescindibles para la innovación tecnológica.

Otras aplicaciones destacadas incluyen:

- **Finanzas:** Detección de fraudes, previsión de mercados y análisis de riesgos.
- **Industria manufacturera:** Mantenimiento predictivo, optimización de procesos y control de calidad automatizado.
- **Automoción:** Desarrollo de vehículos autónomos, donde las redes neuronales procesan datos de sensores y cámaras para tomar decisiones en tiempo real.
- **Marketing:** Segmentación de clientes, sistemas de recomendación, personalización de contenido y análisis predictivo de comportamiento del consumidor.
- **Salud y medicina:** Análisis de ADN y descubrimiento de medicamentos. Monitorización de pacientes en tiempo real a través de dispositivos IoT.

En un contexto más reciente, las redes neuronales también son la base de tecnologías como la generación de contenido en inteligencia artificial (IA generativa), permitiendo la creación de imágenes, texto, música y más, a partir de descripciones o datos iniciales. Estos casos de uso ilustran cómo esta tecnología se ha convertido en una herramienta imprescindible para resolver problemas complejos y mejorar procesos en múltiples industrias, empujando los límites de lo que es posible en la transformación digital.

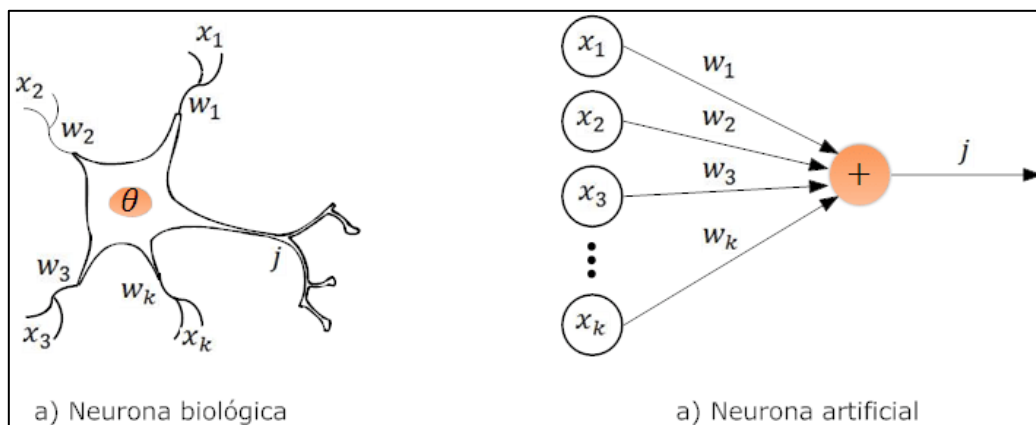
2. Conceptos básicos de las redes neuronales

2.1. Neurona artificial: estructura y función

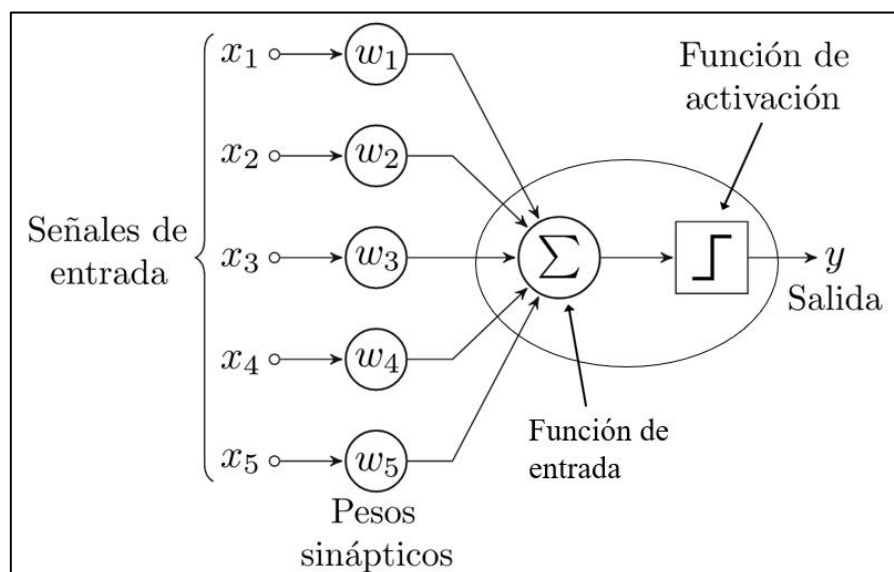
Una red neuronal artificial está formada por un conjunto de unidades básicas denominadas neuronas, que se encuentran interconectadas entre sí, y que son, básicamente, un modelo matemático que intenta emular la función de una neurona cerebral humana.

Cada neurona procesa los valores que recibe como entrada a través de las conexiones con otras neuronas, aplicando una función específica llamada función de entrada. El resultado de este procesamiento se pasa por una función de activación que genera un nuevo valor, siendo este último la salida de la neurona.

Veamos una comparativa entre una neurona humana y una artificial:



Ahora profundizaremos en la estructura de una neurona artificial:



Una neurona artificial está formada por los siguientes elementos:

- **Entradas** (x_n): datos que recibe la neurona. Las entradas suelen ser valores numéricos, como los píxeles de una imagen, mediciones o datos preprocesados.
- **Pesos** (w_n): cada entrada está asociada a un peso, que indica la importancia relativa de esa entrada en el cálculo. Los pesos son parámetros ajustables durante el entrenamiento de la red neuronal, y su valor se optimiza para mejorar el rendimiento del modelo. A uno de los pesos se le suele conocer como “bias”.
- **Función de entrada**: combina los valores de las entradas ponderadas por sus respectivos pesos. Aunque hay diversos tipos, suele emplearse la suma ponderada:

$$z = \sum_{i=1}^n w_i x_i$$

- **Función de activación**: transforma la entrada neta (z) en la salida final de la neurona. Esta transformación introduce no linealidad, permitiendo que la red neuronal represente relaciones complejas entre las entradas y las salidas. Una habitual suele ser la ReLU (más adelante veremos este punto en mayor profundidad):

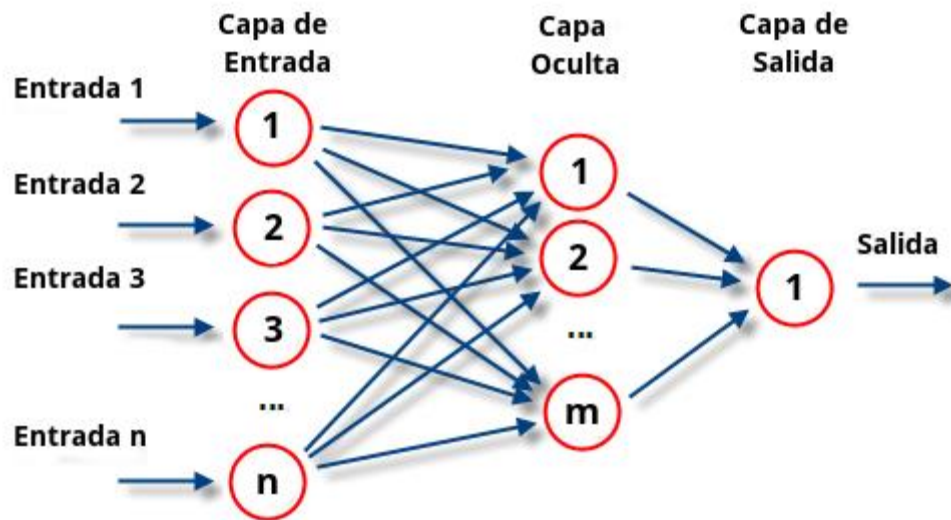
$$f(z) = \max(0, z)$$

- **Salida** (y): la salida final de la neurona se calcula aplicando la función de activación al resultado de la función de entrada.

$$y = f(z) = f\left(\sum_{i=1}^n w_i x_i\right)$$

2.2. Funcionamiento de una red neuronal simple

De la misma forma que las neuronas biológicas se agrupan en el cerebro formando redes complejas, debemos agrupar las neuronas artificiales en estructuras de mayor nivel y capacidad llamadas redes neuronales (artificiales). Una red neuronal simple puede describirse como un modelo matemático donde las neuronas están organizadas en capas: una capa de entrada (que no tiene neuronas), una (o varias) capa(s) oculta(s) y una capa de salida.



El funcionamiento de una red neuronal comienza cuando los datos de entrada se entregan a la capa de entrada. Cada uno de estos valores se procesan de forma sucesiva hasta llegar al resultado (salida), siendo la salida de una capa la entrada de la siguiente y así sucesivamente.

3. Arquitecturas de redes neuronales

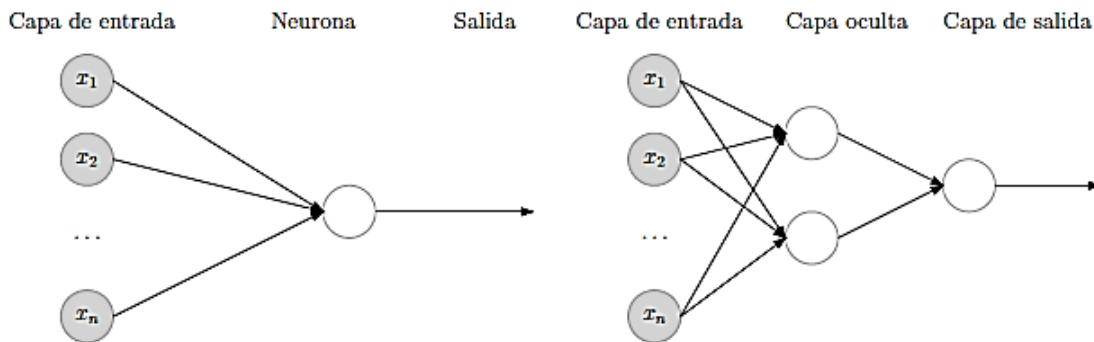
Una red neuronal **feedforward** es una de las arquitecturas más simples dentro del aprendizaje profundo. Su principal característica es que la información fluye en una única dirección, desde la capa de entrada hacia la capa de salida, pasando por las capas ocultas (si las hay). No existen ciclos ni conexiones hacia atrás, lo que la distingue de otras arquitecturas como las redes recurrentes.

El **perceptrón multicapa** (MLP, por sus siglas en inglés) es una extensión del perceptrón simple (aunque el perceptrón hoy en día casi no se utiliza). Incorpora una o más capas ocultas entre la entrada y la salida, lo que le permite modelar relaciones no lineales en los datos. Cada neurona dentro de estas capas aplica una función de activación, como ReLU o sigmoid, que introduce no linealidad al modelo.

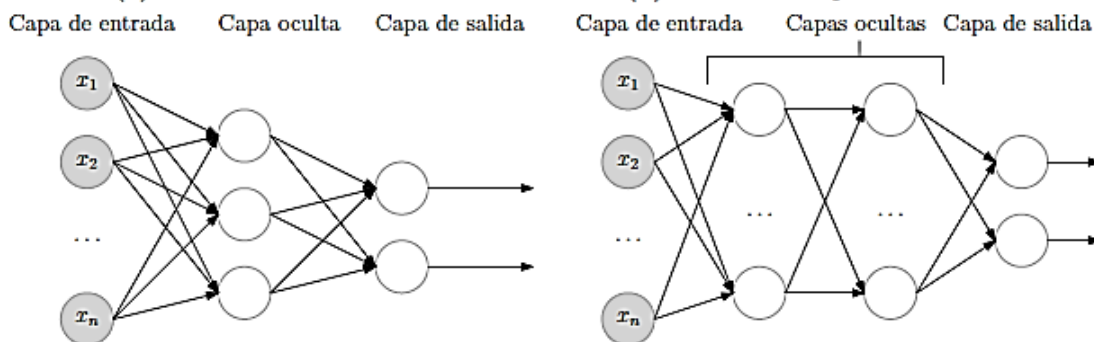
Características clave del modelo *feedforward*:

- **Capas ocultas:** permiten capturar patrones complejos y no lineales.
- **Backpropagation:** el aprendizaje se realiza ajustando los pesos mediante algoritmos de optimización basados en gradientes, como el descenso por gradiente estocástico (SGD).

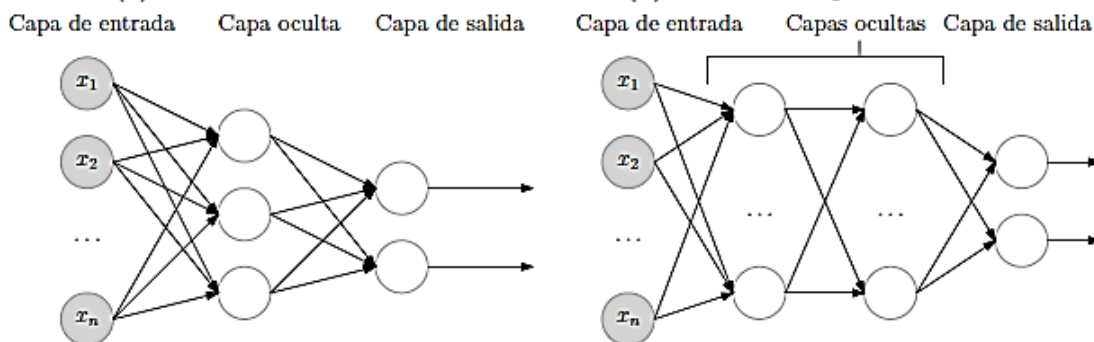
Veamos unos ejemplos gráficos:



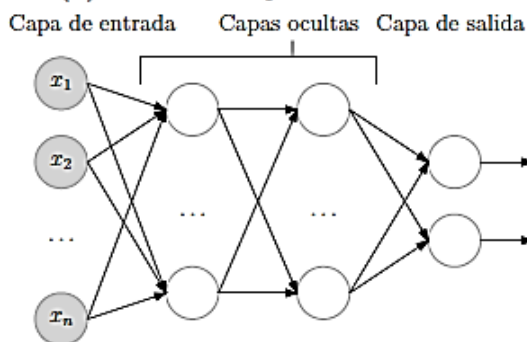
(a) Red con una neurona



(b) Red monocapa con una salida



(c) Red monocapa con múltiples salidas

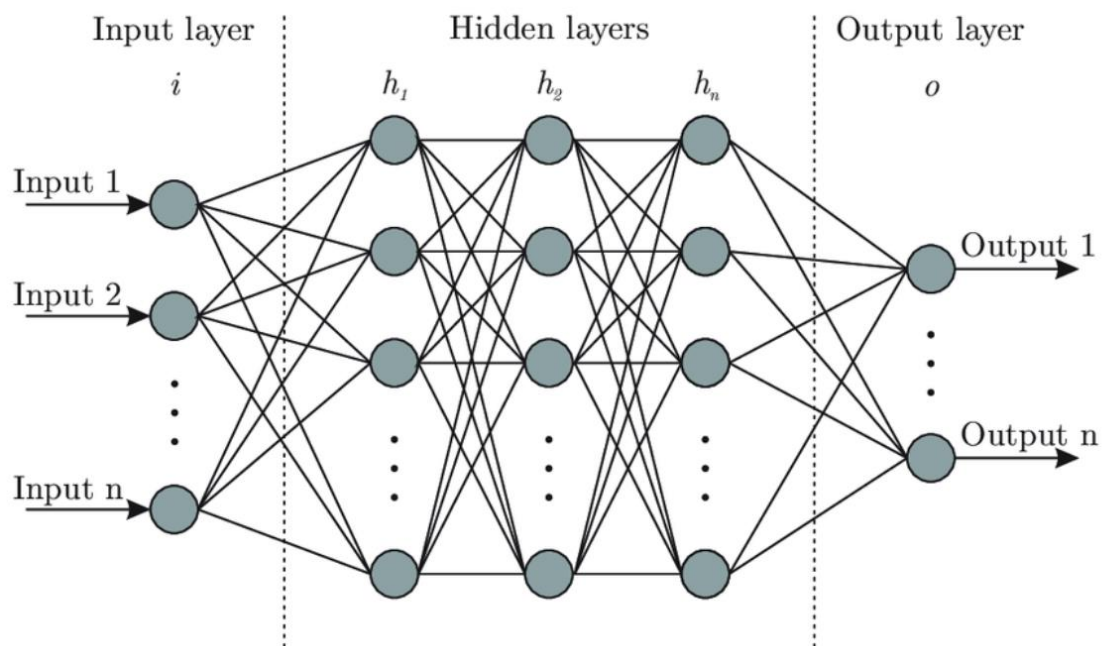


(d) Red multicapa con múltiples salidas

3.1. Capas de entrada, ocultas y de salida

A continuación, se describe el rol de cada capa:

1. **Capa de entrada:** recibe los datos de entrada y los pasa a la siguiente capa. El número de entradas en esta capa corresponde al número de características de los datos.
2. **Capas ocultas:** procesan la información mediante combinaciones lineales de los pesos y funciones de activación. Cuantas más capas y neuronas, mayor capacidad para modelar patrones complejos.
3. **Capa de salida:** genera el resultado final del modelo. El número de neuronas depende del tipo de tarea: una para regresión, varias para clasificación multiclase. El formato de salida, además, depende del problema:
 - **Regresión:** Un único valor continuo o un vector de valores.
 - **Clasificación binaria:** Una probabilidad (usando Sigmoides).
 - **Clasificación multiclase:** Una distribución de probabilidades (usando Softmax).



Ejemplo de redes neuronales con diferentes configuraciones

- **Red básica:** Una capa oculta con 10 neuronas.
- **Red intermedia:** Dos capas ocultas con 20 y 15 neuronas, respectivamente.
- **Red compleja:** Cinco capas ocultas con 50 neuronas cada una.

3.2. Tipos de arquitecturas (shallow vs. deep networks)

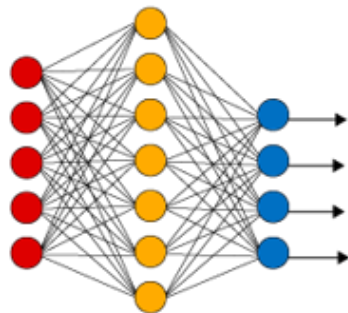
- **Redes poco profundas (1 o 2 capas ocultas)**

Las redes poco profundas son más fáciles de entrenar y requieren menos potencia computacional. Sin embargo, tienen una capacidad limitada para capturar patrones muy complejos.

- **Redes profundas (muchas capas): Deep Learning**

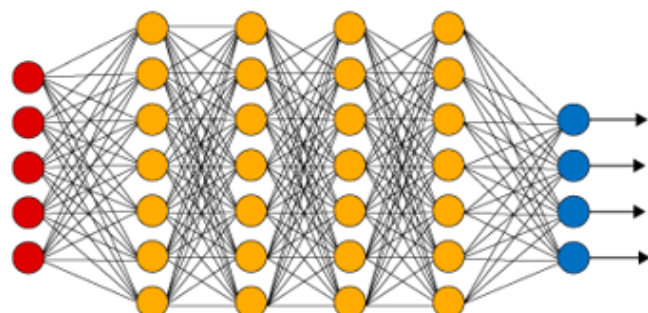
Las redes profundas (deep neural networks) son capaces de aprender representaciones jerárquicas de los datos, lo que las hace ideales para problemas complejos como el reconocimiento de imágenes o el procesamiento del lenguaje natural; además, son aptas para grandes volúmenes de datos. Sin embargo, requieren muchos más recursos computacionales, y tiene riesgo de ciertos problemas como el desvanecimiento o explosión de gradientes.

Simple Neural Network



● Input Layer

Deep Learning Neural Network



● Hidden Layer

● Output Layer

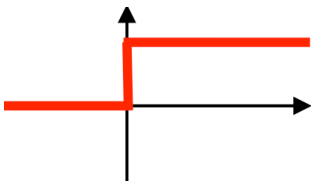
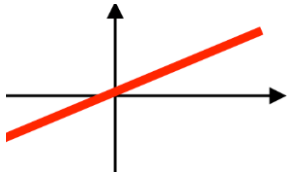
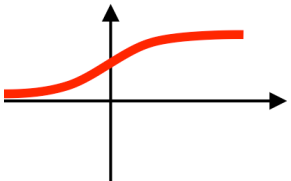
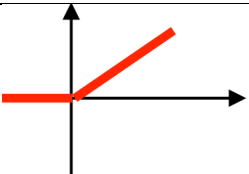
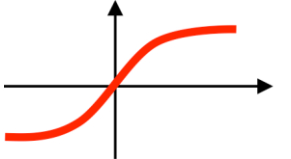
4. Funciones de activación

Las funciones de activación son una parte vital de las redes neuronales, ya que permiten que los modelos capturen relaciones complejas y no lineales entre las entradas y las salidas. Son importantes porque:

1. **Introducen no linealidad en las decisiones de la red:** permiten que la red aprenda patrones como curvas, discontinuidades y distribuciones no triviales; es decir, patrones en problemas complejos.
2. **Importancia para representar funciones complejas:** facilitan que la red neuronal aprenda relaciones jerárquicas y abstractas. Permiten que cada capa capture características específicas antes de transmitir las a la siguiente capa.

4.1. Listado de funciones de activación

Cada función de activación tiene características y aplicaciones específicas:

Función	Formulación	Representación
Escalón (usada en el Perceptrón)	$f(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$	
Lineal	$f(z) = \beta x$	
Sigmoide o logística	$f(z) = \frac{1}{1 + e^{-z}}$	
ReLU (Rectified Linear Unit)	$f(z) = \max(0, z)$	
Tangente hiperbólica (tanh)	$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	

4.2. Selección de la función de activación por capa

La elección de la función de activación depende de la tarea y la ubicación en la arquitectura:

1. **Capas ocultas:** usualmente se elige **ReLU** por su eficiencia computacional y eficacia en redes profundas. Para evitar dying ReLU, se pueden usar variantes como Leaky ReLU o ELU.
2. **Capa de salida:**
 - **Sigmoide:** Clasificación binaria (e.g., presencia/ausencia de una condición).
 - **Softmax:** Clasificación multiclase (e.g., identificar una categoría entre varias).
 - **Lineal:** Problemas de regresión.
3. **Cuando las salidas requieren valores centrados:**
 - **tanh** puede ser más útil que Sigmoide, ya que sus salidas están centradas en cero.

5. Proceso de entrenamiento de redes neuronales

El entrenamiento de una red neuronal es el proceso mediante el cual la red ajusta sus parámetros (pesos y bias) para minimizar el error en sus predicciones y mejorar su desempeño en una tarea específica. Este proceso se lleva a cabo en varias etapas principales, que incluyen la inicialización de los parámetros, la propagación hacia adelante, el cálculo del error, la retropropagación y la actualización de los parámetros. A continuación, se describe cada una de estas etapas.

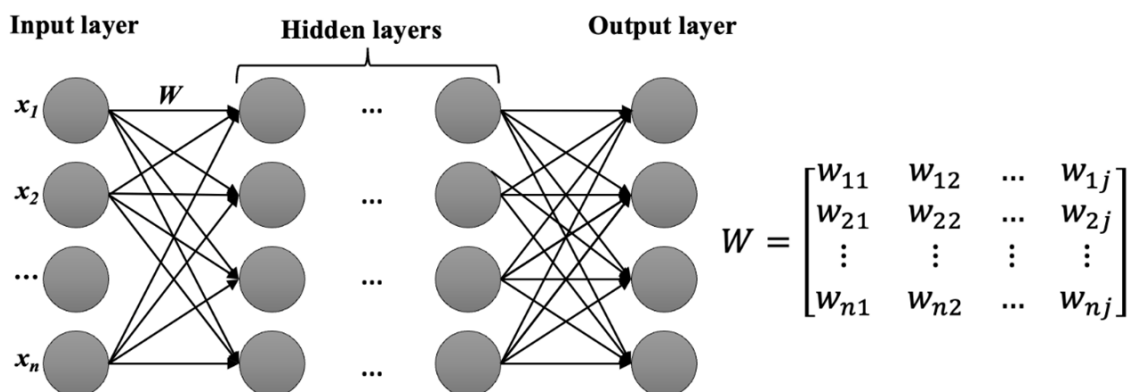
A. Inicialización de pesos

La inicialización de los pesos es un paso crucial en el entrenamiento de redes neuronales y tiene un impacto significativo en la velocidad de convergencia y el rendimiento general del modelo. Se refiere al proceso de asignar valores iniciales a los pesos de la red antes de que comience el entrenamiento. La inicialización del peso adecuado ayuda a garantizar que el proceso de aprendizaje sea eficiente y efectivo, lo que permite que el modelo aprenda patrones a partir de los datos sin caer en errores comunes, como gradientes que desaparecen o explotan.

Se han desarrollado varias técnicas para la inicialización del peso, cada una con sus ventajas y desventajas. Uno de los métodos más utilizados es la inicialización de Xavier (o Glorot), que establece los pesos en función del número de neuronas de entrada y salida. Este método es particularmente eficaz para funciones de activación como la tangente sigmoidea o hiperbólica (tanh). Otra técnica popular es la inicialización He, que está diseñada para capas que utilizan funciones de activación ReLU (Unidad lineal rectificada).

La inicialización tiene en cuenta la cantidad de neuronas de entrada y escala los pesos en consecuencia, lo que ayuda a mitigar los problemas relacionados con la desaparición de gradientes.

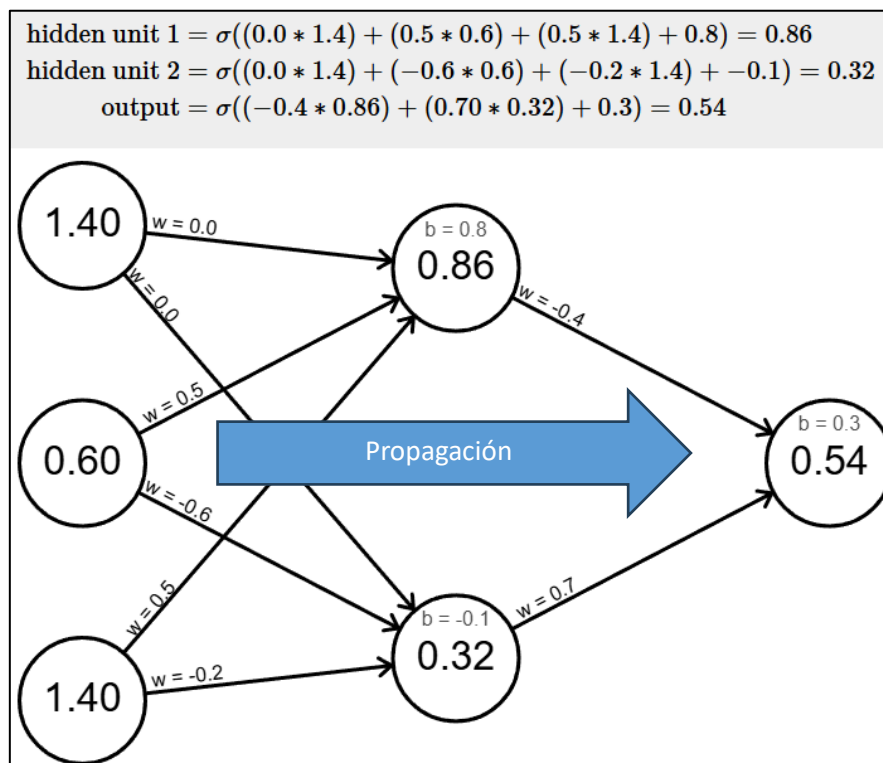
La siguiente imagen ilustra una equivalencia algebraica (matriz) de los pesos (w):



B. Propagación hacia adelante (Forward Propagation)

La propagación hacia adelante es el proceso mediante el cual una red neuronal procesa las entradas y genera una salida (predicción) a través de sus capas. Es el primer paso en el entrenamiento y la evaluación de una red neuronal. La salida de una capa se convierte en la entrada de la siguiente. Este proceso se repite para todas las capas de la red. En la última capa, la salida puede representarse de diferentes formas, dependiendo del problema.

Ejemplo gráfico:



C. Cálculo del error y función de pérdida

El cálculo del error nos permite evaluar qué tan lejos están las predicciones de la red de los valores reales. Este error se cuantifica mediante una función de pérdida, que es una métrica que queremos minimizar durante el proceso de entrenamiento. Veamos las funciones de pérdida comunes:

- **Error Cuadrático Medio** (Mean Squared Error, MSE): se utiliza en problemas de regresión.

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

- **Entropía Cruzada** (Cross-Entropy): usada en problemas de clasificación, especialmente en clasificación binaria y multiclase.

$$\text{Binary Cross Entropy} = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$$

La selección de la función de pérdida a emplear dependerá del tipo de problema a resolver, por tanto, tenemos:

- **Clasificación binaria:** binary cross-entropy
- **Clasificación multiclase:** categorical cross-entropy, sparse categorical cross-entropy
- **Regresión:** MSE, MAE, MAPE, Huber loss.

D. Propagación hacia atrás (Backpropagation) y ajuste de pesos

La propagación hacia atrás es el proceso mediante el cual las redes neuronales ajustan sus pesos y sesgos para minimizar el error calculado por la función de pérdida (se produce un ajuste mayor cuanto más error ha cometido la red en la salida).

Consiste en un método iterativo que utiliza el gradiente descendente para actualizar los pesos de la red en función del error. Este proceso se divide en dos fases principales: propagación del error hacia atrás y ajuste de los pesos.

Veamos las fases con mayor detalle:

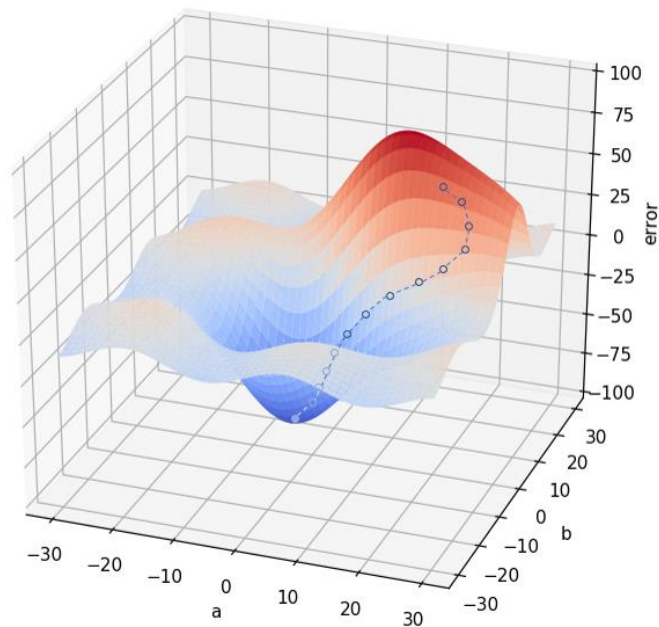
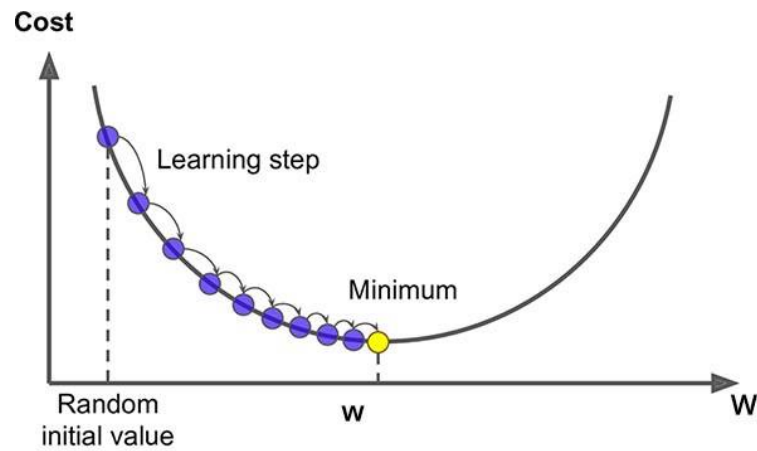
1. Cálculo del error en la salida: la salida de la red (y_{pred}) se compara con el valor real (y_{real}) usando una función de pérdida (L). Por ejemplo:

$$L = \frac{1}{n} \sum_{i=1}^n (y_{\text{real},i} - y_{\text{pred},i})^2$$

2. Cálculo del gradiente de la función de pérdida: el gradiente indica cómo cambiar los pesos para minimizar el error. Se calcula derivando la función de pérdida respecto a los pesos. Para un peso w :

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y_{\text{pred}}} \cdot \frac{\partial y_{\text{pred}}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Las siguientes imágenes ilustran el funcionamiento del descenso de gradiente en 2 casos:



3. Retropropagación del gradiente

El gradiente calculado se propaga desde la capa de salida hacia las capas anteriores. Esto se realiza en dos pasos:

- calcular el error en la capa actual:

$$\delta = \frac{\partial L}{\partial z}$$

- propagarlo hacia la capa anterior.

$$\delta_{\text{anterior}} = \delta \cdot W$$

4. Actualización de los pesos y sesgos

Los pesos y sesgos se actualizan utilizando el gradiente y una tasa de aprendizaje (η).

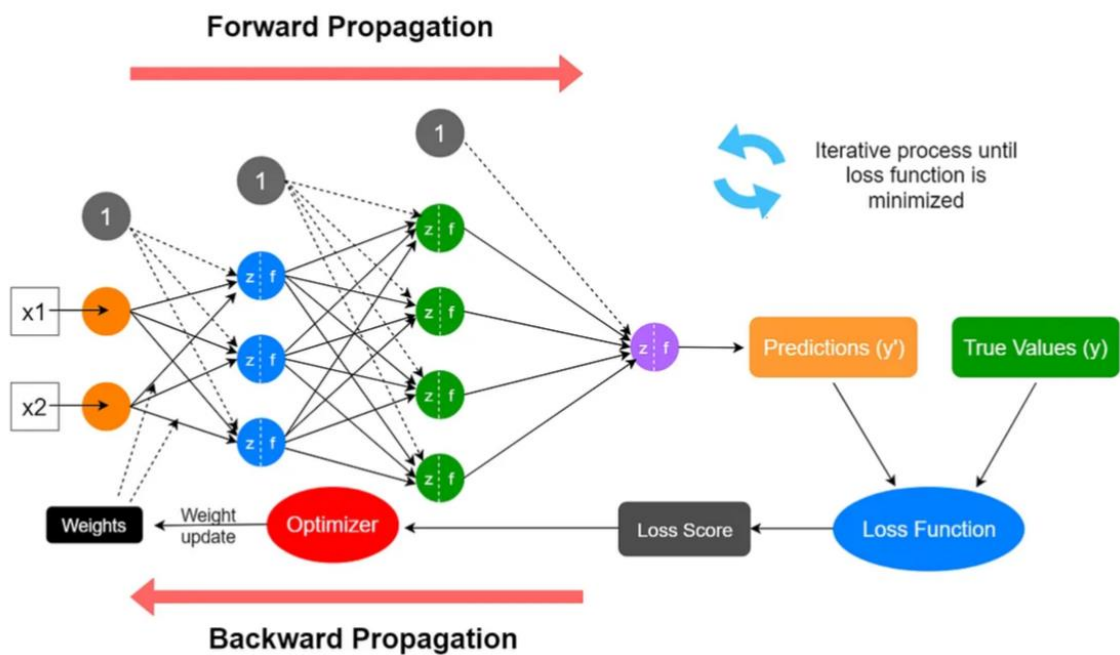
- Pesos:

$$w_{\text{nuevo}} = w_{\text{actual}} - \eta \cdot \frac{\partial L}{\partial w}$$

- Sesgos:

$$b_{\text{nuevo}} = b_{\text{actual}} - \eta \cdot \frac{\partial L}{\partial b}$$

Veamos un diagrama esquemático del proceso completo:



6. Regularización y prevención del overfitting

El overfitting y el underfitting son dos problemas comunes al entrenar modelos de aprendizaje automático, especialmente redes neuronales. En este apartado vamos a profundizar en sus causas, formas de detección y técnicas para prevenir el overfitting mediante regularización y buenas prácticas en la validación del modelo. Causas:

- **Overfitting** ocurre cuando el modelo tiene demasiada capacidad (es decir, demasiados parámetros) y se ajusta excesivamente a los datos de entrenamiento, capturando ruido o patrones irrelevantes. Esto resulta en un bajo error en el conjunto de entrenamiento, pero un alto error en el conjunto de validación o prueba.
- **Underfitting** sucede cuando el modelo tiene poca capacidad y no puede capturar los patrones relevantes en los datos. Esto se traduce en un alto error tanto en los datos de entrenamiento como en los de validación o prueba.

Diagnóstico a partir de las curvas de aprendizaje: las curvas de aprendizaje muestran la evolución de la pérdida (o una métrica de rendimiento) durante el entrenamiento para los conjuntos de entrenamiento y validación. Son útiles para identificar overfitting o underfitting:

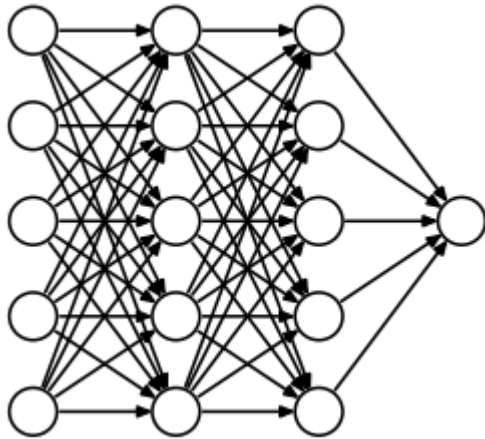
- **Overfitting:** La pérdida de entrenamiento sigue disminuyendo, pero la pérdida de validación se estabiliza o aumenta.
- **Underfitting:** La pérdida de entrenamiento y validación se mantienen altas y no mejoran significativamente.

Relación entre tamaño del modelo y calidad del ajuste:

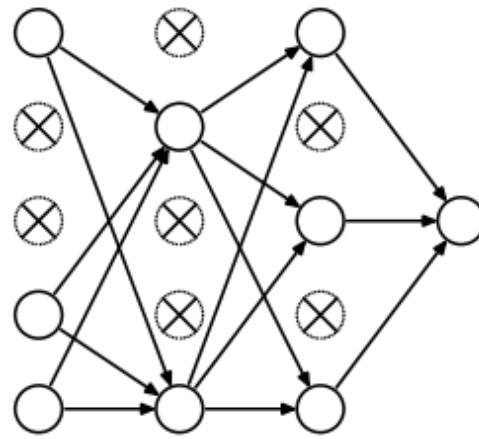
- Los modelos grandes (con más parámetros) tienden a sobreajustar si los datos de entrenamiento son limitados.
- Los modelos pequeños pueden subajustar al no ser capaces de aprender patrones complejos en los datos.
- Encontrar un equilibrio es crucial: el modelo debe ser lo suficientemente grande como para aprender los patrones relevantes, pero no tan grande que capture ruido.

Técnicas de regularización

1. **Dropout:** es una técnica que apaga aleatoriamente un porcentaje de neuronas en cada capa durante el entrenamiento. Esto fuerza al modelo a aprender representaciones más robustas. Dropout actúa como un ensamble implícito de redes neuronales al entrenar varias subredes diferentes dentro de un único modelo.



Red neuronal estándar



Tras aplicar Dropout

Ejemplo:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

model = Sequential([
    Dense(128, activation='relu', input_shape=(784,)),
    Dropout(0.5), # Apaga el 50% de las neuronas de las 128
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])
```

Dropout(0.5) se aplica a la capa oculta **inmediatamente anterior**, es decir, a la capa Dense(128, activation='relu')

Valores a usar en las capas densas (Dense): 0.2 - 0.5

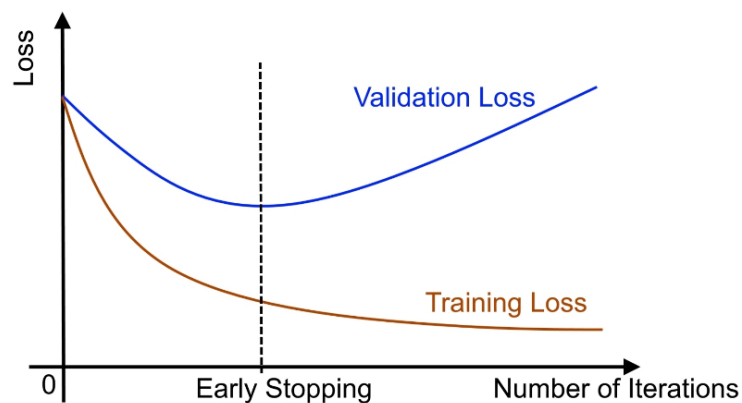
Reglas prácticas: empieza con 0.2 - 0.5 y ajusta según la validación. Si el modelo sobreajusta, sube el Dropout. Si el modelo aprende muy lento, bájalo. Se recomienda usar en 1 o 2 capas ocultas, generalmente tras las capas más grandes.

2. **Early Stopping:** detiene el entrenamiento cuando la métrica de validación deja de mejorar durante un número determinado de épocas (patience). Previene que el modelo continúe ajustándose a los datos de entrenamiento cuando ya no mejora en la validación.

Ejemplo:

```
from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss', patience=5)
model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=50,
callbacks=[early_stopping])
```



3. **Batch Normalization:** normaliza los valores intermedios de las capas (activaciones) para que tengan media cero y desviación estándar uno. Esto acelera el entrenamiento al permitir usar tasas de aprendizaje más altas y actúa como una técnica de regularización, reduciendo la dependencia del modelo a inicializaciones específicas.

Ejemplo:

```
from keras.layers import BatchNormalization

model = Sequential([
    Dense(128, activation='relu', input_shape=(784,)),
    BatchNormalization(),
    Dense(64, activation='relu'),
    BatchNormalization(),
    Dense(10, activation='softmax')
])
```

7. Evaluación del rendimiento de redes neuronales

Evaluar el rendimiento de una red neuronal es clave para medir su eficacia y ajustar el modelo según las necesidades del problema. La validación y evaluación de redes neuronales siguen principios muy similares a los de los modelos tradicionales de machine learning (ML) que ya se han trabajado en la unidad anterior, como el hecho de que las métricas de evaluación (precisión, F1-score, MSE, etc.) son las mismas para cada problema específico (clasificación, regresión, etc.). La diferencia principal radica en las particularidades de las redes neuronales, que introducen aspectos adicionales a tener en cuenta:

- a) **Entrenamiento iterativo en batches y múltiples épocas:** las redes neuronales se entrenan en iteraciones (épocas), ajustando los pesos progresivamente. Esto requiere introducir curvas de pérdida/métricas y callbacks como el Early Stopping para decidir cuándo detenerse. Los modelos tradicionales de ML, como árboles de decisión o SVM, suelen ajustarse en una sola pasada por los datos.
- b) **Mayor sensibilidad al overfitting:** las redes neuronales, debido a su alta capacidad (número de parámetros), son más propensas a sobreajustarse a los datos de entrenamiento. Para evitarlo se emplean técnicas específicas como: regularización con dropout, L1/L2; y aumento de datos mediante sobremuestreo, rotaciones, escalados en imágenes o ruido en texto.
- c) **Diseño de la arquitectura:** la evaluación no solo mide la capacidad del modelo para aprender, sino también la calidad de su arquitectura en base a cosas como el número de capas, su tipo (Convolucionales, LSTM, etc.) y también hiperparámetros como learning rate o tamaño del batch.

7.1. Métricas de evaluación comunes

Dependiendo del tipo de problema (clasificación o regresión), se utilizan diferentes métricas. Hay que tener en cuenta que estas métricas son muy similares a las tratadas en el tema anterior:

- A. Métricas para Clasificación:** precisión (accuracy), recall (Sensibilidad), F1-Score, AUC-ROC. Matriz de confusión.
- B. Métricas para Regresión:** RMSE (Root Mean Squared Error), MAE (Mean Absolute Error): R^2 (Coeficiente de Determinación).
- C. Rendimiento:** también es importante valorar el tiempo de entrenamiento.

7.2. Visualización de la convergencia y curvas de aprendizaje

Es fundamental monitorizar el entrenamiento analizando las curvas de pérdida y precisión, así como otras para detectar problemas como sobreajuste o subajuste.

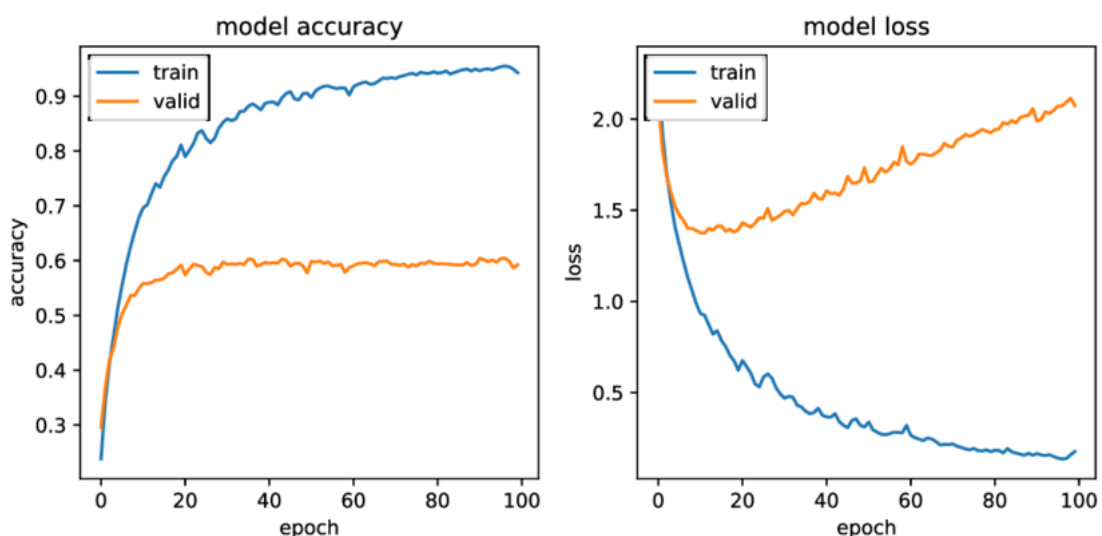
Curvas de Error: representan cómo cambia la pérdida durante el entrenamiento y la validación. Veamos un ejemplo:

```
history = model.fit(x_train, y_train, validation_data=(x_val,
y_val), epochs=20)
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Curvas de Métricas: ayudan a evaluar la evolución de métricas como accuracy o F1-score. Veamos un ejemplo añadiendo la curva al código anterior:

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Veamos unas muestras visuales de ambas gráficas:



En este caso se muestra un comportamiento anómalo, la red no está rindiendo bien. La precisión en validación se estanca, alejada de la de entrenamiento, y la pérdida de validación aumenta.

7.3. Técnicas para mejorar el rendimiento y ajuste de hiperparámetros

El ajuste de hiperparámetros es un proceso clave en la optimización de modelos de Machine Learning, ya que seleccionar valores adecuados para parámetros como la tasa de aprendizaje, el tamaño del lote o el número de neuronas puede mejorar significativamente el rendimiento del modelo. Existen diferentes enfoques para realizar esta tarea, cada uno con sus ventajas y desventajas.

Librerías Populares:

- **Keras Tuner con RandomSearch:** en lugar de probar todas las combinaciones posibles, este método selecciona valores aleatorios dentro del espacio de búsqueda definido. Es más eficiente que Grid Search cuando hay muchos hiperparámetros, ya que no evalúa combinaciones irrelevantes.

```
import tensorflow as tf
from keras_tuner import RandomSearch
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# Definir función para construir el modelo
def build_model(hp):
    model = Sequential([
        Dense(units=hp.Int('units', min_value=32, max_value=512,
step=32), activation='relu'),
        Dense(1)
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(
            learning_rate=hp.Choice('learning_rate', [0.001, 0.01])
        ),
        loss='mean_squared_error',
        metrics=['mae']
    )
    return model

# Crear el sintonizador de hiperparámetros
tuner =
RandomSearch(build_model,objective='val_loss',max_trials=5,directory='
my_dir',project_name='tuning1'
)

# Ejecutar la búsqueda de hiperparámetros
tuner.search(x_train, y_train, epochs=10, validation_split=0.2)

best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"Mejor cantidad de neuronas en capa oculta:
{best_hps.get('units')}")
print(f"Mejor learning rate: {best_hps.get('learning_rate')}")
```

- **Optuna:** librería de optimización de hiperparámetros que se enfoca en la eficiencia, utilizando un enfoque de optimización basado en "optimización bayesiana". Es más eficiente que Grid Search y Random Search en cuanto a la exploración del espacio de búsqueda, ya que no prueba todas las combinaciones posibles, sino que selecciona las combinaciones más prometedoras basándose en los resultados anteriores.

```
import optuna
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
import numpy as np

# Definir la función objetivo que Optuna optimizará
def objective(trial):
    # Espacios de búsqueda de hiperparámetros (esto son ejemplos)
    units = trial.suggest_int('units', 32, 512, step=32)
    lr = trial.suggest_categorical('learning_rate', [0.001, 0.01, 0.1])

    model = Sequential([
        Dense(units=units, activation='relu', input_shape=(10,)),
        Dense(1)
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
        loss='mean_squared_error',
        metrics=['mae']
    )

    model.fit(x_train, y_train, epochs=10, validation_split=0.2,
              verbose=0)

    # Obtener la pérdida en la validación para optimizar
    val_loss = model.history.history['val_loss'][-1]
    return val_loss

# Crear estudio de Optuna y ejecutar la optimización
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=5) # 5 intentos de prueba

# Mostrar los mejores hiperparámetros encontrados
print(f'Best hyperparameters: {study.best_params}')

# Extraer los mejores hiperparámetros
best_units = study.best_params['units']
best_lr = study.best_params['learning_rate']
# Construir y entrenar el modelo con esos hiperparámetros
final_model = Sequential([
    Dense(units=best_units, activation='relu', input_shape=(10,)),
    Dense(1)
])
```


8. Implementación práctica de una red neuronal simple

8.1. TensorFlow y Keras

TensorFlow (TF) es un framework de código abierto especialmente diseñado para construir y entrenar modelos de aprendizaje profundo (Deep Learning). Aunque es conocido por su flexibilidad y potencia, su API de bajo nivel puede ser muy compleja. TensorFlow soporta computación distribuida (CPUs, GPUs, TPUs), tiene una gran comunidad y abundantes recursos y es capaz de conseguir una optimización automática para hardware específico.

Dada la complejidad que tiene la API de TF, se utiliza **Keras**, una API de alto nivel, integrada en TF, que facilita la construcción y entrenamiento de redes neuronales de manera sencilla y estructurada. Algunas de sus características más relevantes son:

- Diseño modular: los componentes (capas, optimizadores, etc.) son intercambiables.
- Compatibilidad con múltiples frameworks backend (TensorFlow, Theano, CNTK, aunque TensorFlow es el estándar ahora).
- Sencillez: proporciona una sintaxis intuitiva para trabajar con modelos.

Otro framework importante es **PyTorch**, conocido por su flexibilidad, potencia y dinamismo, pero que no veremos en esta unidad

8.2. Construcción de una red básica para un problema de clasificación

Vamos a trabajar con un dataset **tabular**, ideal para problemas de clasificación o regresión. Usaremos el dataset **Pima Indians Diabetes** de UCI Machine Learning Repository, que es ampliamente utilizado para demostrar conceptos en aprendizaje automático. Este dataset contiene datos relacionados con la diabetes y se utiliza para predecir si una persona tiene diabetes o no.

El dataset contiene:

- **8 características** numéricas como: Embarazos, Glucosa, Presión arterial, Grosor de la piel, Insulina, Índice de masa corporal (IMC), DiabetesPedigree (historial familiar), Edad
- **1 objetivo (target):** Si la persona tiene diabetes (1) o no (0).

En primer lugar, vamos a cargar el dataset:

```
import pandas as pd
import numpy as np

# Cargar los datos
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
columns = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
           'Insulin', 'BMI', 'DiabetesPedigree', 'Age', 'Outcome']

data = pd.read_csv(url, header=None, names=columns)
data.head()
```

Ahora haremos un pequeño preproceso. Es importante separar las características, por una parte, de la columna target, por otra. También debemos normalizar porque hay atributos en escalas muy diferentes (Glucose vs Pregnancies) y porque así mejoraremos la convergencia del modelo. Finalmente dividimos en conjuntos de entrenamiento y prueba:

```
X = data.iloc[:, :-1].values # Todas las columnas excepto 'Outcome'
y = data.iloc[:, -1].values  # Solo la columna 'Outcome'

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X = scaler.fit_transform(X)

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2, random_state=42)
```

Ahora podemos construir nuestro modelo de red neuronal, compilarlo y visualizar la configuración antes de entrenar:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

# Creamos el modelo de red neuronal
model = Sequential([
    Input(shape=(X_train.shape[1],)), # Definir forma de la entrada
    Dense(16, activation='relu'),      # 1a capa oculta
    Dense(8, activation='relu'),       # 2a capa oculta
    Dense(1, activation='sigmoid')     # Salida
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Mostramos configuración
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 16)	144
dense_1 (Dense)	(None, 8)	136
dense_2 (Dense)	(None, 1)	9

Total params: 289 (1.13 KB)

Trainable params: 289 (1.13 KB)

Non-trainable params: 0 (0.00 B)

Vemos la salida en pantalla donde se resume la configuración de la red que hemos preparado, especialmente interesante saber la cantidad total de parámetros para el entrenamiento (289).

En este momento podemos lanzar el entrenamiento:

```
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2, verbose=1)
```

La parte final de la salida del entrenamiento tendrá este aspecto:

```
Epoch 49/50
16/16 _____ 0s 4ms/step - accuracy:
0.8028 - loss: 0.4312 - val_accuracy: 0.7724 - val_loss: 0.4610
Epoch 50/50
16/16 _____ 0s 3ms/step - accuracy:
0.8284 - loss: 0.3878 - val_accuracy: 0.7642 - val_loss: 0.4622
```

Cuando finalice el entrenamiento podemos evaluarlo y visualizar determinadas métricas, así como la evolución por cada época.

```
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Pérdida: {loss}, Precisión: {accuracy}")

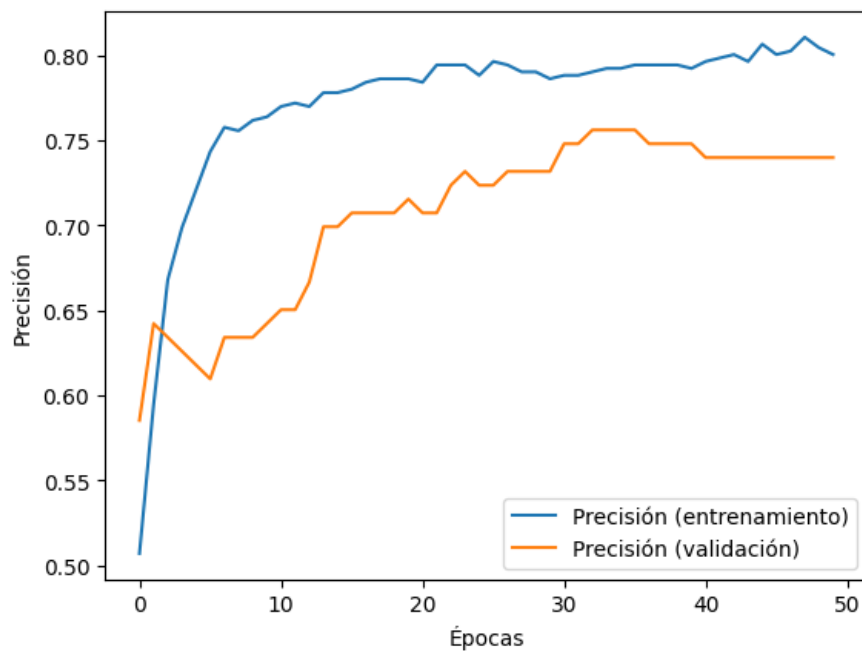
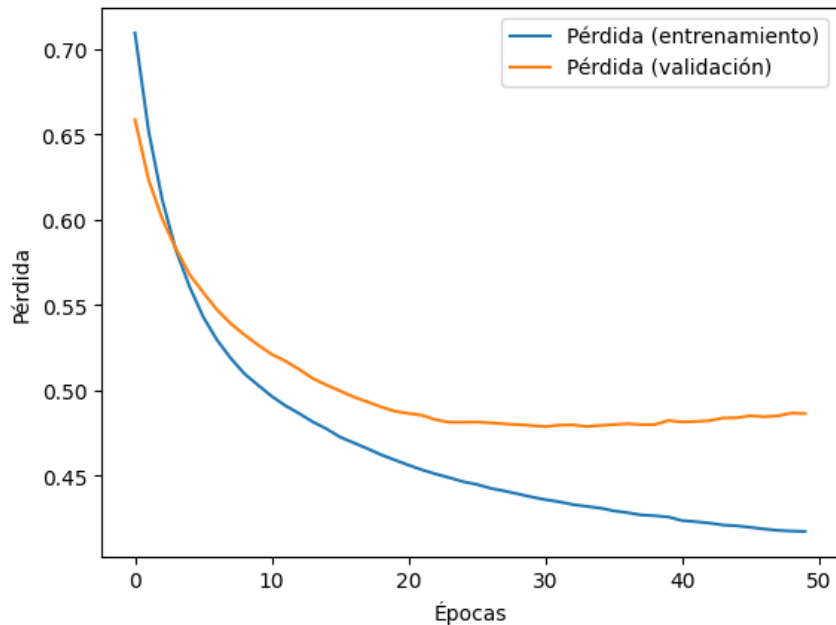
import matplotlib.pyplot as plt

# Pérdida
plt.plot(history.history['loss'], label='Pérdida (entrenamiento)')
plt.plot(history.history['val_loss'], label='Pérdida (validación)')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.show()

# Precisión
plt.plot(history.history['accuracy'], label='Precisión
(entrenamiento)')
plt.plot(history.history['val_accuracy'], label='Precisión
(validación)')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.show()
```

La salida esperada será similar a la siguiente:

5/5 ————— 0s 2ms/step - accuracy: 0.7598 - loss: 0.5239
Pérdida: 0.5436650514602661, Precisión: 0.7597402334213257



Vemos algunas cosas interesantes:

- En primer lugar, la salida del método evalúate.
- La **gráfica de pérdida** muestra una disminución a medida que avanzan las épocas, sin embargo, a partir de la época 30, aproximadamente, la pérdida en validación aumenta mientras disminuye en entrenamiento. Eso es indicativo de overfitting, debería haberse detenido el entrenamiento ahí. Eso se resolverá con un callback como EarlyStopping.

- La **gráfica de precisión** muestra cómo la precisión va en aumento a medida que avanzan las épocas, tanto en entrenamiento como en validación. Aunque en validación es algo inferior, lo cual es normal. Si la precisión en entrenamiento aumentara, pero la de validación no, estaríamos ante overfitting.

8.3. Clases, métodos y parámetros de keras

Vamos a analizar con más detalle algunos objetos y funciones utilizados por fases:

A. **Creación de modelos:** definir la estructura y capas de la red.

- **Secuencial:** ideal en redes simples, sirve para arquitecturas lineales.
- **Funcional:** más versátil, permite arquitecturas complejas con múltiples entradas y/o salidas, o conexiones entre capas no lineales.
- **Subclassing:** heredando de Model, permite crear arquitecturas personalizadas útiles en diseños avanzados o dinámicos.

B. **Compilación de modelos (función compile):** especificar cómo el modelo debe ser optimizado, cómo calcular la pérdida y qué métricas se usarán para evaluar su rendimiento. Argumentos:

- **optimizer (obligatorio):** algoritmo que ajustará los pesos en el entrenamiento. Valores: adam, sgd, rmsprop, adagrad, adamax, nadam. Habitualmente se usa adam, podemos escribirlo directamente o emplear optimizadores personalizados desde "keras.optimizers". Parámetros del optimizador:
 - **learning_rate:** tasa de aprendizaje (típicamente 0.001)
 - **momentum** (en SGD): acelera la convergencia.
 - **decay:** para reducir la tasa de aprendizaje con el tiempo.
- **loss (obligatorio):** función de pérdida para calcular las discrepancias entre las predicciones y los valores reales. Valores habituales:
 - **Clasificación:** 'binary_crossentropy', 'categorical_crossentropy' y 'sparse_categorical_crossentropy'.
 - **Regresión:** 'mean_squared_error', 'mean_absolute_error' y 'huber'.
 - **Otras:** 'kullback_leibler_divergence' y 'poisson'
- **metrics:** no afectan al entrenamiento, especifica métricas para evaluar el modelo.
 - **Clasificación:** accuracy, precisión, recall, AUC.
 - **Regresión:** mean_squared_error, mean_absolute_error.

C. **Entrenamiento del modelo (función fit):** implementa el proceso de aprendizaje iterativo, ajustando los pesos del modelo para minimizar la función de pérdida, y devolviendo un objeto “History” que contiene información útil sobre el entrenamiento del modelo, como las métricas y la pérdida.

```
history = model.fit(x=None, y=None, batch_size=None, epochs=1,
verbose=1, callbacks=None, validation_split=0.0,
validation_data=None, shuffle=True, class_weight=None,
sample_weight=None, initial_epoch=0, steps_per_epoch=None,
validation_steps=None, validation_batch_size=None,
validation_freq=1, max_queue_size=10, workers=1,
use_multiprocessing=False)
```

Parámetros más destacados:

- **x**: datos de entrada.
- **y**: valores reales correspondientes a los datos de entrada x.
- **batch_size**: tamaño del lote por iteración, es decir, cuántas muestras se procesarán antes de actualizar los pesos. Por defecto es 32, se debe aumentar a 64 o 128 para conjuntos grandes de datos.
- **epochs**: número de épocas, es decir, pasadas o iteraciones por todos los datos de entrenamiento.
- **verbose**: 0 (no muestra información), 1 (muestra una barra de progreso por epoch) y 2 (muestra información sin línea de progreso).
- **callbacks**: objetos de tipo Callback para personalizar y controlar el entrenamiento. Es típico el uso de EarlyStopping.
- **shuffle**: determina si se reorganizan aleatoriamente los datos antes de cada época. Por defecto es True.

La variable “history” es el objeto de tipo History que contiene un atributo principal llamado “history”, el cual es un diccionario que contiene las métricas y la pérdida.

8.4. Guardado y carga de modelos entrenados

Cuando se entrena un modelo, es fundamental poder guardar su estado para reutilizarlo en futuras predicciones sin necesidad de volver a entrenarlo. Esto ahorra tiempo y recursos computacionales, además de permitir compartir modelos entre diferentes entornos.

Las dos estrategias principales en Keras para guardar un modelo son:

- Guardar el modelo completo (estructura y parámetros).
- Guardar solo los pesos (en redes neuronales).

A continuación, veremos ambas estrategias:

```
# Guardar modelo compilado y entrenado
modelo.save('nombre_modelo.keras')

# Cargar el modelo a una variable
modeloCargado = keras.models.load_model('nombre_modelo.keras')
# -----
# Guardar sólo los pesos
modelo.save_weights('pesos.h5')

# Cargar solo los pesos en un modelo con la misma arquitectura
# Primero tenemos que definir la arquitectura
modelo_nuevo = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(1, activation='sigmoid')
])

# Luego podemos cargar los pesos guardados
modelo_nuevo.load_weights('pesos.h5')
```

Atención: existe en Python una librería de serialización llamada “**pickle**”, PERO no se recomienda almacenar una red neuronal así, aunque técnicamente sea posible. Esto se debe a que pickle no garantiza la compatibilidad entre diferentes versiones de librerías como TensorFlow o PyTorch. Además, puede haber problemas si el entorno donde se carga el modelo no es exactamente el mismo que el original.

Sin embargo, es preferible usar otros formatos más eficientes, especialmente para serializar grandes volúmenes de datos, como son `.h5` y `.parquet`

9. Retos y limitaciones de las redes neuronales

Aunque las redes neuronales han demostrado ser una herramienta extremadamente poderosa, también presentan desafíos significativos que deben abordarse para garantizar su eficacia. A continuación, se explican algunos de los retos más importantes junto con estrategias para mitigarlos.

9.1. Problemas de *vanishing* y *exploding* gradients

En redes neuronales profundas, los gradientes juegan un papel crucial en el entrenamiento, ya que determinan cómo se ajustan los pesos a través del proceso de backpropagation. Sin embargo, en redes muy profundas, los gradientes pueden volverse extremadamente pequeños (*vanishing gradients*) o extremadamente grandes (*exploding gradients*), lo que puede impedir un entrenamiento eficaz.

1. **Vanishing Gradients:** ocurre cuando los gradientes se reducen a valores cercanos a cero mientras retropropagan a través de las capas. Como resultado, las primeras capas de la red apenas reciben actualizaciones significativas, lo que limita su capacidad de aprendizaje. Esto suele suceder con funciones de activación que comprimen los valores, como la sigmoide o la tangente hiperbólica, y cuando la inicialización de los pesos no es adecuada. Las soluciones comunes son:
 - Usar funciones de activación como ReLU (Rectified Linear Unit), que no saturan para valores positivos.
 - Inicialización de pesos mediante métodos como He o Xavier, diseñados específicamente para redes profundas.
 - Normalización mediante técnicas como Batch Normalization, que estabilizan la distribución de las activaciones en cada capa.
2. **Exploding Gradients:** los gradientes pueden crecer exponencialmente mientras se retropropagan, causando actualizaciones de pesos muy grandes. Esto puede llevar a una red inestable o a la aparición de NaN en los cálculos. Las soluciones comunes son:
 - **Clipping de gradientes:** Establecer un límite máximo para el valor de los gradientes durante el entrenamiento.
 - Regularización, como Dropout o L2, para limitar la complejidad del modelo.

9.2. Coste computacional y necesidad de recursos

El entrenamiento y la implementación de redes neuronales, especialmente las profundas, requieren recursos computacionales significativos. Esto incluye no solo el hardware, sino también tiempo y energía, factores que pueden convertirse en limitaciones prácticas.

1. Comparación entre CPU, GPU y TPU:

- **CPU (Unidad Central de Procesamiento):** son unidades de propósito general que pueden manejar una variedad de tareas. Son adecuadas para redes neuronales simples, pero su capacidad para realizar cálculos en paralelo es limitada, lo que ralentiza el entrenamiento.
- **GPU (Unidad de Procesamiento Gráfico):** están diseñadas para realizar cálculos paralelos en grandes cantidades, lo que las hace ideales para el entrenamiento de redes neuronales. En comparación con las CPUs, las GPUs son significativamente más rápidas para este uso.
- **TPU (Unidad de Procesamiento Tensor):** han sido diseñadas específicamente para cargas de trabajo de aprendizaje automático, y ofrecen aún más optimización que las GPUs en ciertos escenarios, como grandes modelos de deep learning.

2. **Impacto del coste computacional y soluciones:** los entrenamientos prolongados pueden ser prohibitivamente caros. Modelos más grandes y complejos aumentan la necesidad de memoria y almacenamiento.

Soluciones:

- Uso de técnicas como la poda de redes para reducir su tamaño y complejidad.
- Adopción de frameworks de optimización distribuida para aprovechar múltiples dispositivos en paralelo.
- Transferencia de aprendizaje, que permite usar modelos preentrenados en tareas específicas, reduciendo el tiempo y los recursos necesarios.

9.3. Redes neuronales profundas y Deep Learning

Las redes neuronales profundas son la base del deep learning, una rama del aprendizaje automático que se centra en el uso de modelos complejos de redes

neuronales que poseen múltiples capas para representar datos de alta complejidad. Aunque esta profundidad permite aprender patrones extremadamente sofisticados, también introduce nuevos retos.

1. Relación entre redes profundas y deep learning:

- En las redes profundas, cada capa adicional puede aprender características más abstractas y de mayor nivel.
- Deep learning se basa en arquitecturas como CNNs, RNNs y transformers, que aprovechan esta profundidad para resolver problemas complejos en imágenes, texto y audio.

2. Desafíos específicos del deep learning:

- **Sobreajuste:** Las redes profundas tienen una enorme capacidad de aprendizaje, lo que puede llevarlas a memorizar datos en lugar de generalizar. Esto requiere técnicas como la regularización y el uso adecuado de conjuntos de validación.
- **Requerimientos de datos:** El deep learning es altamente dependiente de grandes volúmenes de datos. Sin un conjunto de datos suficientemente grande y representativo, los modelos pueden ser propensos al sesgo y al mal rendimiento.
- **Explicabilidad:** A medida que las redes se hacen más profundas y complejas, entender cómo toman decisiones se vuelve más difícil, lo que plantea problemas en aplicaciones críticas como la medicina o el derecho.

Ventajas de las redes profundas:

- Capacidad para modelar relaciones no lineales y dependencias complejas.
- Excelentes resultados en aplicaciones donde otras técnicas de aprendizaje automático fallan, como el reconocimiento facial o la traducción automática.

10. Tipos de redes neuronales

Las redes neuronales han evolucionado significativamente desde sus primeras implementaciones, adaptándose a diversos tipos de problemas. Aunque comparten principios fundamentales, cada tipo de red está diseñado para abordar necesidades específicas y optimizar el rendimiento en tareas concretas. A continuación, exploramos los tipos más relevantes y sus características principales:

Redes Neuronales Feedforward (FFNN)

Las redes neuronales feedforward son la arquitectura más básica y ampliamente utilizada, siendo, de hecho, la utilizada en esta unidad como modelo. En estas redes, las señales fluyen desde la capa de entrada hacia la capa de salida, pasando por una o más capas ocultas. Cada neurona recibe información de las capas anteriores, la procesa mediante una función de activación y transmite el resultado a la siguiente capa. Este diseño, sin bucles ni retroalimentación, es ideal para resolver problemas generales de clasificación y regresión.

Estas redes son apropiadas para tareas donde no existe una relación temporal o secuencial en los datos, como predecir el precio de un producto basándose en características demográficas o clasificar correos electrónicos como spam o no spam. Su simplicidad las hace fáciles de entrenar y entender, aunque su capacidad para modelar relaciones complejas puede ser limitada en comparación con arquitecturas más avanzadas.

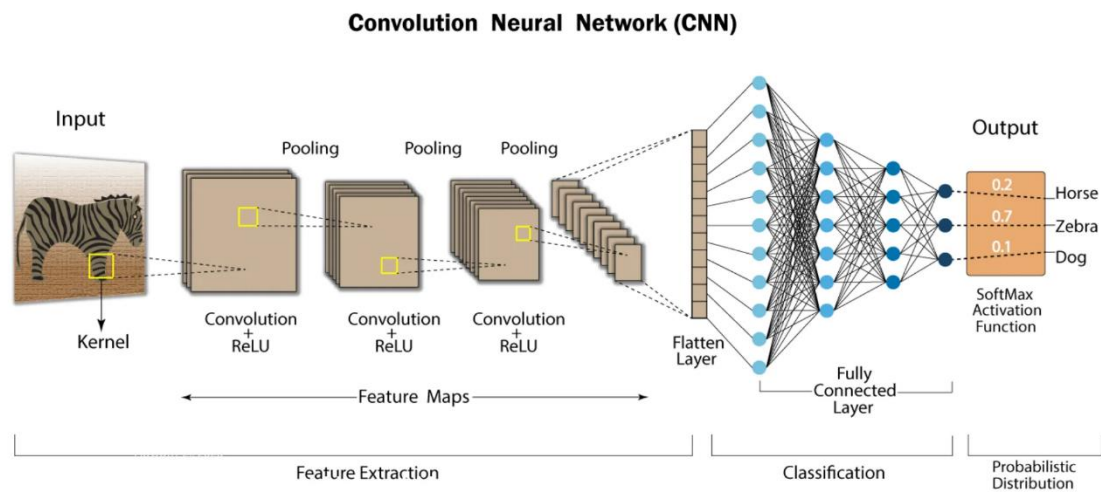
Redes Neuronales Convolucionales (CNN)

Las redes convolucionales son particularmente efectivas para procesar datos con estructura espacial, como imágenes y video. A diferencia de las redes tradicionales, las CNN emplean operaciones de convolución para detectar patrones locales, como bordes, texturas o formas simples, que luego combinan para identificar estructuras más complejas.

El núcleo de una CNN se basa en capas convolucionales, que aplican filtros a pequeños fragmentos de los datos de entrada para extraer características significativas. Estas capas se complementan con capas de pooling, que reducen la dimensionalidad de los datos al mismo tiempo que preservan la información esencial. Finalmente, las capas completamente conectadas procesan las características extraídas para realizar predicciones o clasificaciones.

Las CNN han revolucionado campos como la visión por computador, siendo fundamentales en aplicaciones como la detección de objetos, el reconocimiento facial y la segmentación de imágenes médicas. Su diseño permite manejar la

complejidad de las imágenes mientras mantiene un rendimiento computacional eficiente.



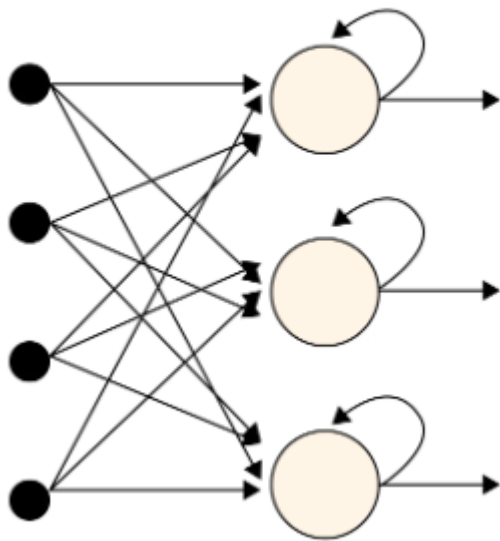
Redes Neuronales Recurrentes (RNN)

Las redes neuronales recurrentes están diseñadas para manejar datos secuenciales o temporales, como series de tiempo, texto o audio. A diferencia de las feedforward, las RNN tienen conexiones recurrentes que les permiten almacenar información de pasos anteriores en una memoria interna. Esta capacidad de "recordar" información previa las convierte en herramientas potentes para modelar dependencias temporales.

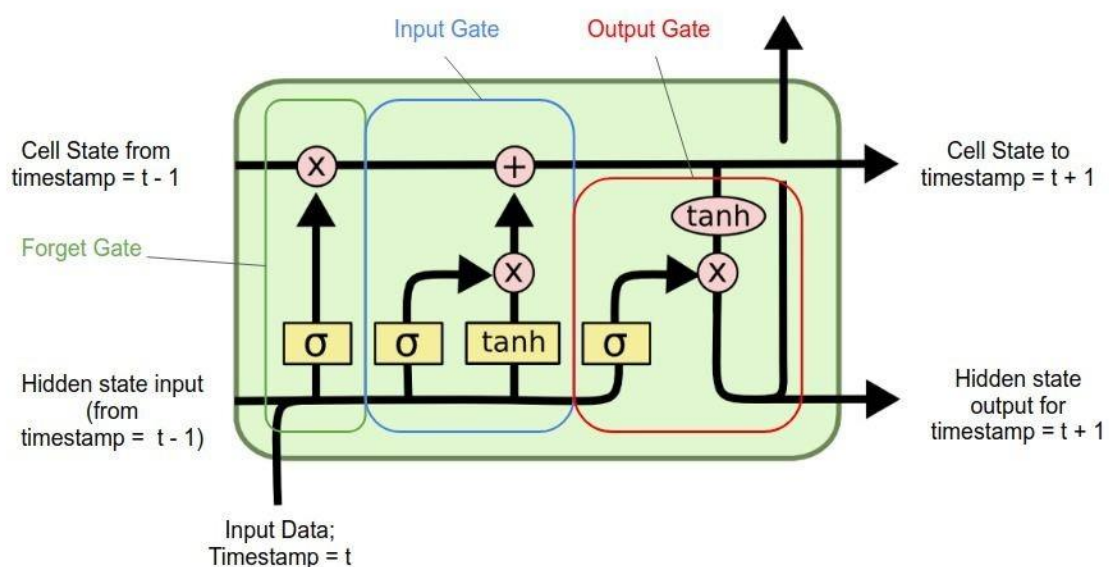
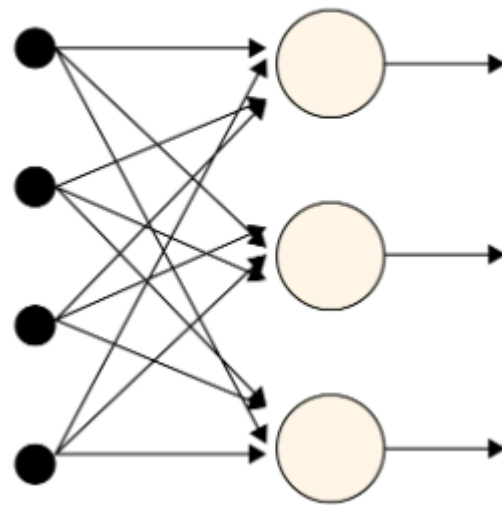
Sin embargo, las RNN enfrentan retos significativos como el desvanecimiento y la explosión del gradiente. Estos problemas dificultan el aprendizaje cuando las secuencias son largas, ya que los gradientes (que guían el ajuste de los pesos) se vuelven demasiado pequeños o demasiado grandes. Para superar estas limitaciones, se han desarrollado variantes como las LSTM (Long Short-Term Memory) y las GRU (Gated Recurrent Unit), que incluyen mecanismos avanzados de control para retener información relevante y descartar la irrelevante.

Entre sus aplicaciones más comunes se encuentran el análisis de sentimiento en textos, la predicción de valores en series temporales financieras y el reconocimiento de voz.

(a) Recurrent Neural Network



(b) Feed-Forward Neural Network



Menciones Breves a Otras Variantes

Además de las arquitecturas más comunes, existen redes especializadas que han abierto nuevas posibilidades en áreas específicas:

- **GANs** (Generative Adversarial Networks): estas redes constan de dos componentes principales: un generador, que crea datos sintéticos, y un discriminador, que evalúa la autenticidad de los datos generados. Compiten entre sí en un proceso que mejora progresivamente la calidad de los datos generados. Las GANs han sido cruciales en la creación de imágenes realistas, la generación de contenido creativo y el aumento de datos para entrenar otras redes.

- **Autoencoders:** estas redes se utilizan principalmente para compresión y reducción de dimensionalidad. Un autoencoder aprende a representar datos en una forma comprimida y luego reconstruirlos desde esa representación. Son útiles en tareas como la detección de anomalías, donde los datos fuera de lo normal no pueden ser reconstruidos con precisión, y en la eliminación de ruido en señales o imágenes.