Deep Learning

Inteligencia Artificial

Universidad EAFIT

August 25, 2025

UNIVERSIDAD EAFIT
Escuela de Ciencias Aplicadas e Ingeniería

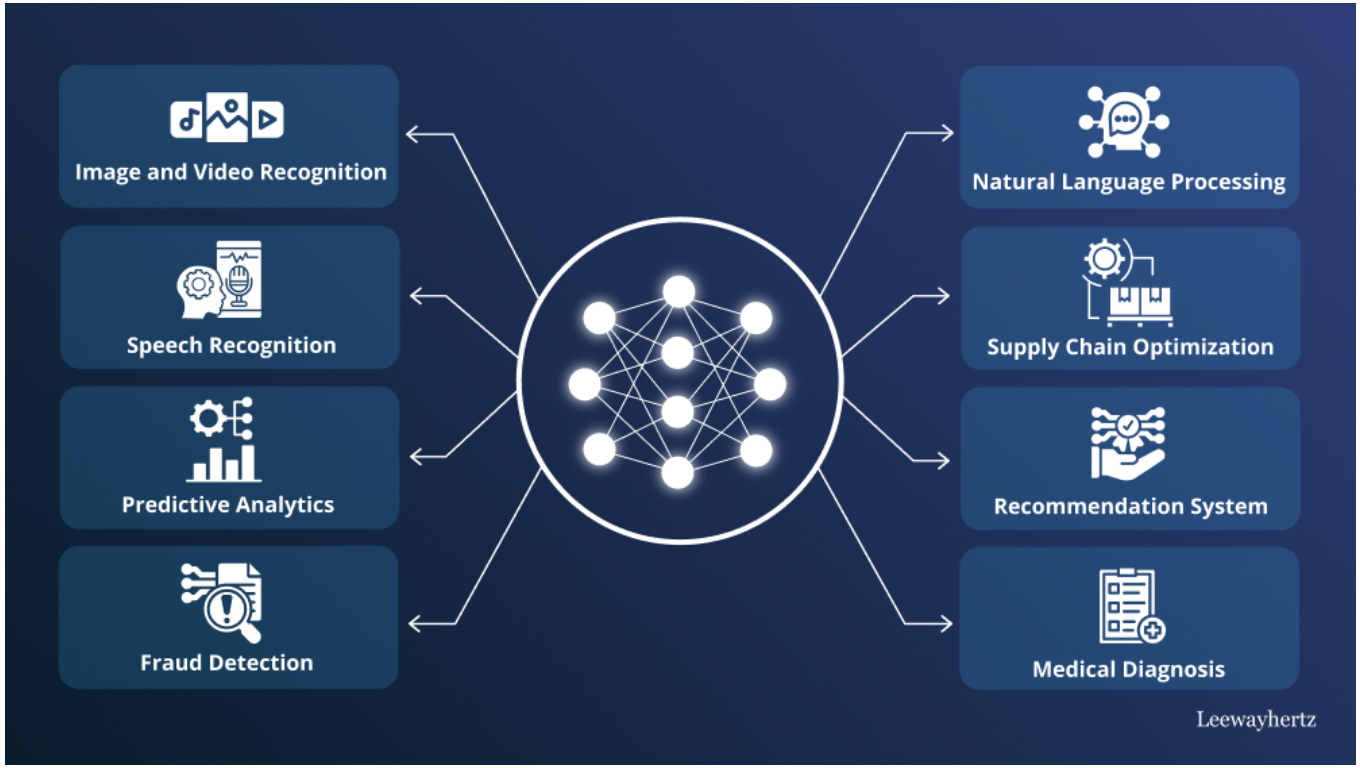
Agenda del día (3 Horas)

1. **Parte 1: Fundamentos (60 min)**
   - ¿Qué es el Deep Learning?
   - El Cerebro vs. La Red Neuronal
   - El Perceptrón: La neurona artificial
   - Funciones de Activación
2. **Parte 2: Construcción de una Red Neuronal (ANN) (60 min)**
   - Introducción a Keras y TensorFlow
   - El Dataset MNIST: "Hola Mundo" del Deep Learning
   - Codificando una Red Neuronal Densa
   - Compilación, Entrenamiento y Evaluación
3. **Parte 3: Ejercicios (60 min)**
   - Ejercicios prácticos
   - Conclusiones y recursos adicionales

Parte 1

Fundamentos de Deep Learning

Parte 1: ¿Qué es el Deep Learning?



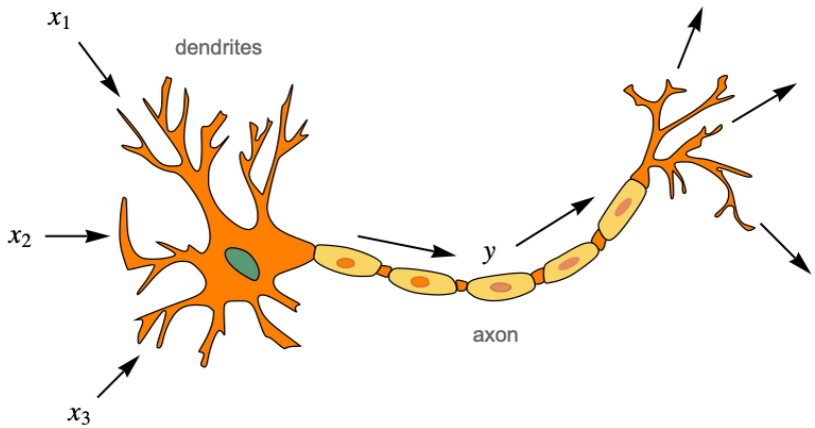
Deep Learning es un subcampo del Machine Learning basado en redes neuronales artificiales con múltiples capas.

Parte 1: ¿Qué es el Deep Learning?

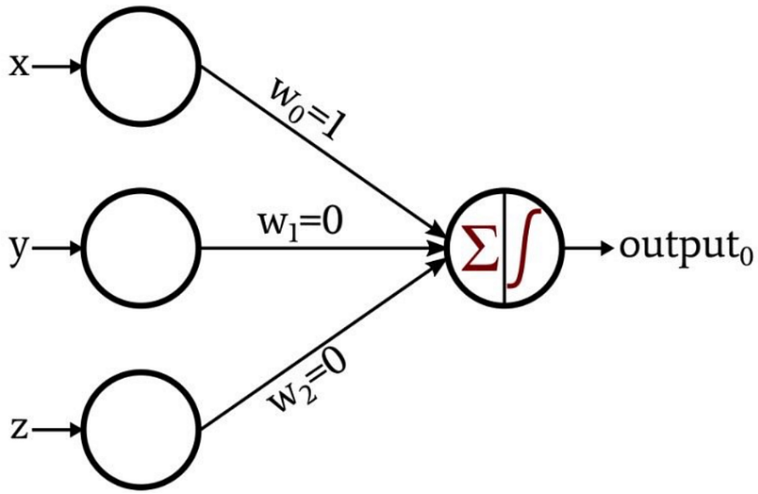
- **Recap- Machine Learning:** Algoritmos que aprenden de los datos.
- **Redes Neuronales:** Modelos inspirados en el cerebro humano.
- **Deep (Profundo):** Se refiere al uso de múltiples capas (generalmente > 2 capas ocultas) en la red para aprender jerarquías de características.

Inspiración: El Cerebro Humano

Neurona Biológica



Neurona Artificial (Perceptrón)



Inspiración: El Cerebro Humano

Neurona Biológica

- Recibe señales a través de las **dendritas**.
- Procesa las señales en el **soma**.
- Si la señal es suficientemente fuerte, se dispara un impulso a través del **axón**.

Neurona Artificial (Perceptrón)

- Recibe **entradas** $(x_1, x_2, \dots)$.
- Cada entrada tiene un **peso** $(w_1, w_2, \dots)$.
- Se calcula una suma ponderada.
- Una **función de activación** decide la salida.

El Perceptrón: La Neurona Artificial

La operación de una neurona se puede describir matemáticamente.

1. **Suma Ponderada (z):** Se multiplican las entradas (x_i) por sus pesos (w_i) y se suma un término de sesgo (b).

$$z = (w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n) + b$$

En notación vectorial:

$$z = \mathbf{w}^T \mathbf{x} + b$$

2. **Función de Activación (σ):** La suma ponderada z pasa a través de una función no lineal para producir la salida final (y).

$$y = \sigma(z) = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

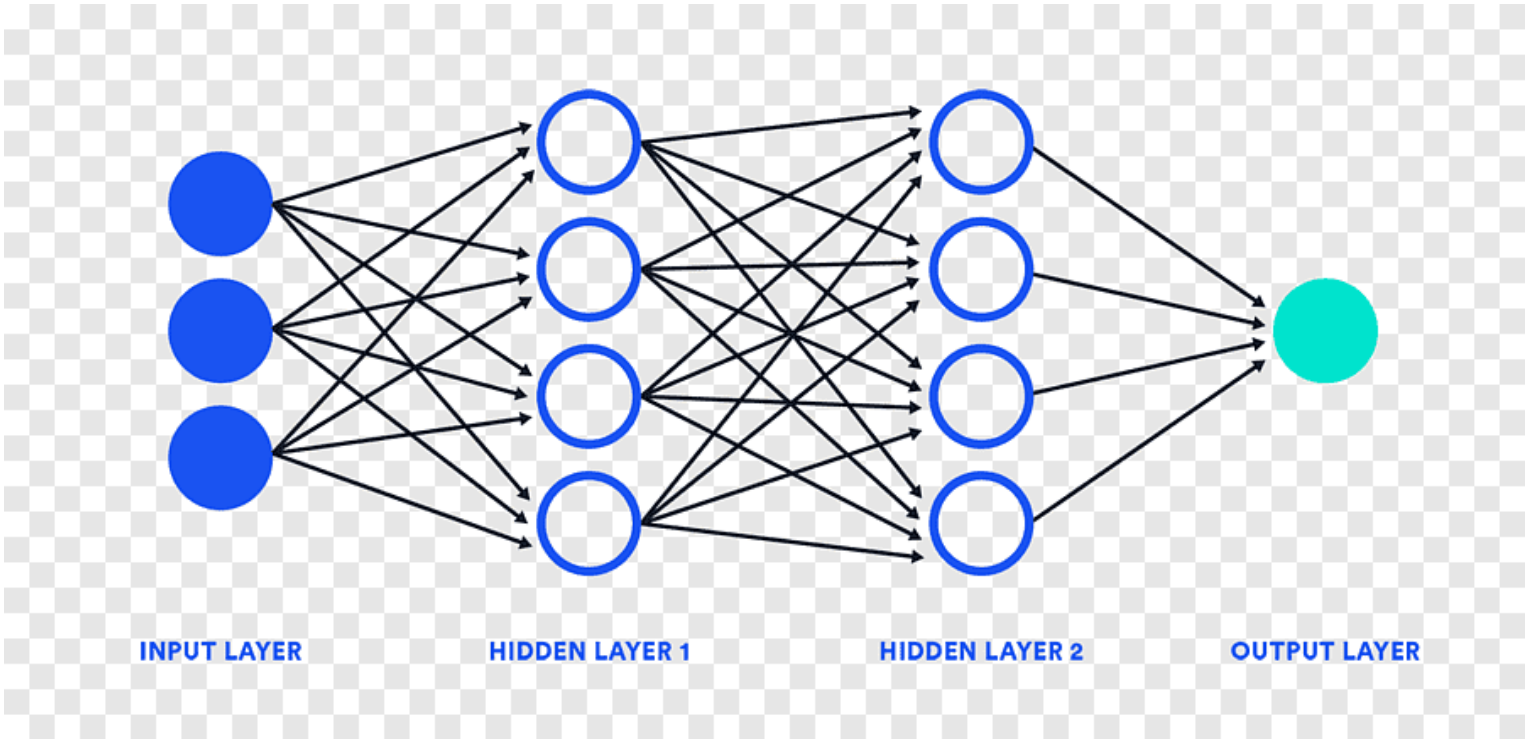
El Perceptrón: La Neurona Artificial

¿Por qué el sesgo (bias)?

El sesgo b permite que la función de activación se desplace hacia la izquierda o la derecha, lo cual es crítico para que la red aprenda correctamente. Es como la ordenada al origen en una recta $y = mx + b$.

De una Neurona a una Red

Una red neuronal se construye apilando neuronas en **capas**.

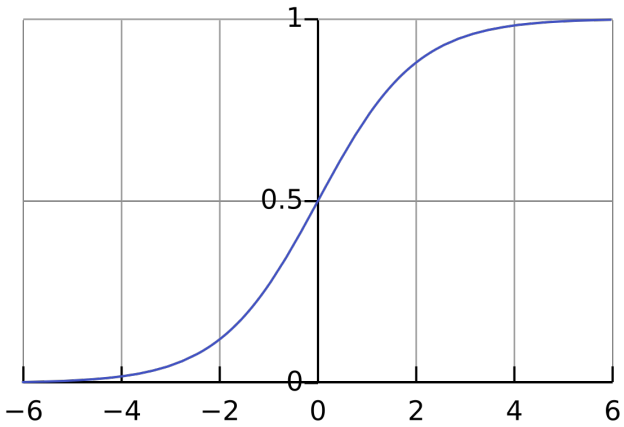


De una Neurona a una Red

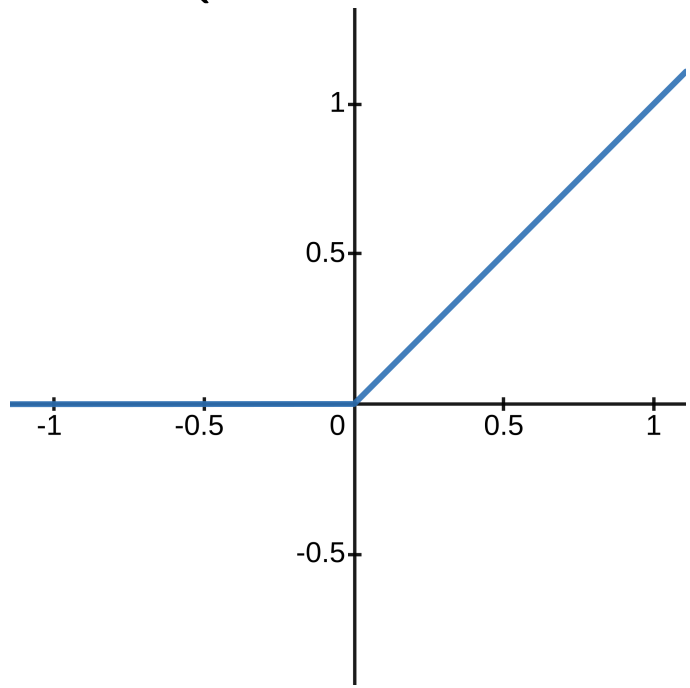
- **Capa de Entrada (Input Layer):** Recibe los datos iniciales (ej: los píxeles de una imagen).
- **Capas Ocultas (Hidden Layers):** Capas intermedias donde ocurre el aprendizaje de patrones complejos. ¡Aquí es donde reside la "profundidad"!
- **Capa de Salida (Output Layer):** Produce el resultado final (ej: la probabilidad de que la imagen sea un '7').

Funciones de Activación: El "Interruptor"

Sigmoide



ReLU (Rectified Linear Unit)



Funciones de Activación: El "Interruptor"

La función de activación decide si una neurona debe "dispararse" o no. Introduce no-linealidad en el modelo, permitiendo aprender relaciones complejas.

Sigmoide

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

- Salida entre 0 y 1.
- Usada históricamente, pero puede sufrir de "desvanecimiento de gradiente".

ReLU (Rectified Linear Unit)

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

- ¡La más popular!
- Muy eficiente computacionalmente.
- Evita problemas de gradiente.

Función de Activación para la Capa de Salida: Softmax

Cuando tenemos un problema de clasificación multiclase (como MNIST, con 10 dígitos), usamos **Softmax**.

- Convierte un vector de números (logits) en un vector de probabilidades.
- La suma de todas las probabilidades es igual a 1.

Para un vector de salida $Z = (z_1, z_2, \ldots, z_K)$, la probabilidad para la clase i es:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

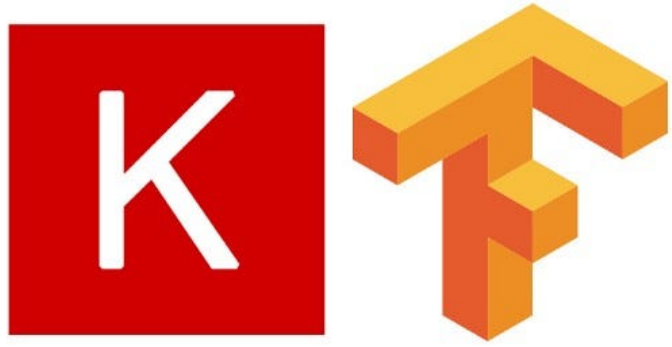
Ejemplo: Si la red produce logits `[1.2, 0.9, 0.4]` para las clases [gato, perro, pájaro], Softmax los convierte en probabilidades como `[0.48, 0.36, 0.16]`.

Predicción: Gato

Parte 2

Construcción de una Red Neuronal (ANN)

Parte 2: Herramientas



Parte 2: Herramientas

TensorFlow

- Una librería de código abierto para computación numérica y machine learning a gran escala.
- Desarrollada por Google.
- Proporciona las bases (operaciones con tensores, ejecución en GPU, etc.).

Keras

- Una API de alto nivel para construir y entrenar modelos de deep learning.
- ¡Corre sobre TensorFlow!
- Enfocada en la facilidad de uso y la experimentación rápida.

El Dataset MNIST: "Hola Mundo" del Deep Learning

El Dataset MNIST: "Hola Mundo" del Deep Learning

- Una colección de 70,000 imágenes de dígitos escritos a mano (0-9).
- **60,000 imágenes** para entrenamiento.
- **10,000 imágenes** para pruebas.
- Cada imagen es de **28x28 píxeles** en escala de grises.
- El objetivo es construir un modelo que pueda clasificar correctamente estas imágenes.

Paso 1: Cargar y Preparar los Datos

Keras nos facilita enormemente la carga de MNIST.

```
import tensorflow as tf
from tensorflow import keras

# Cargar el dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.
    mnist.load_data()

# x_train tiene una forma de (60000, 28, 28)
# y_train tiene una forma de (60000,)

print("Forma de x_train:", x_train.shape)
print("Forma de y_train:", y_train.shape)
```

Objetivo: Predecir la etiqueta y_{train} (un número de 0 a 9) a partir de la imagen x_{train}.

Paso 2: Preprocesamiento de Datos

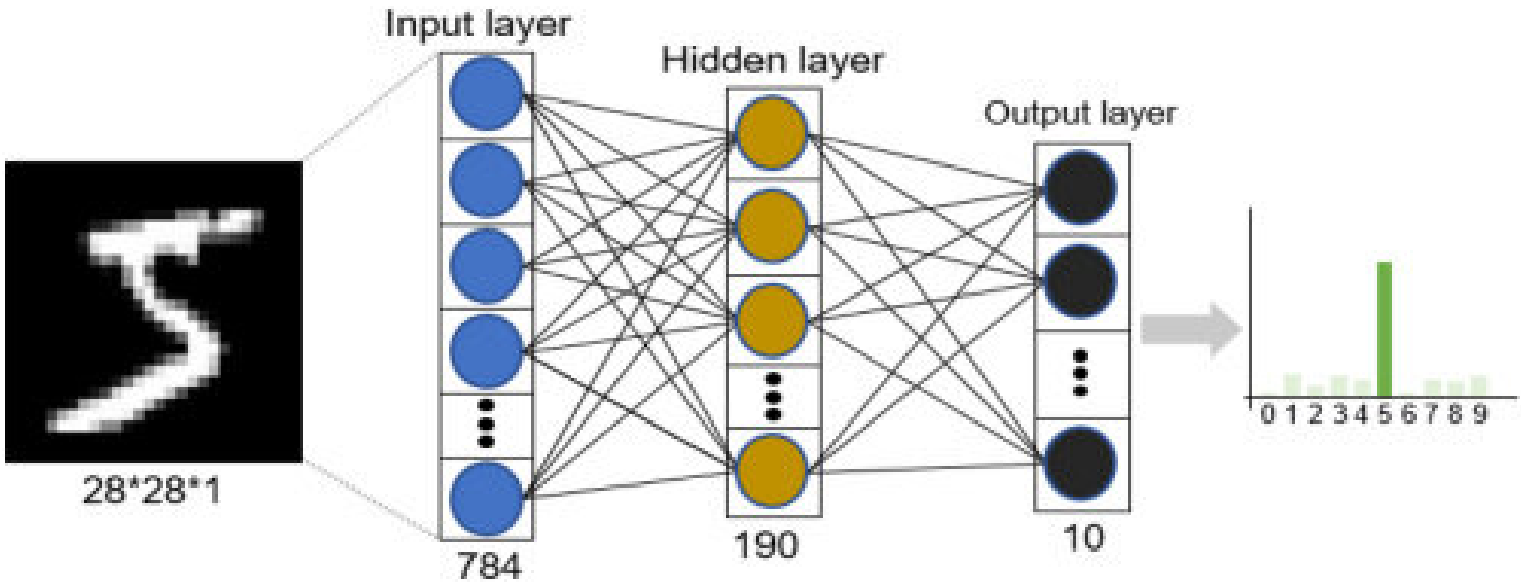
Las redes neuronales funcionan mejor con datos normalizados y con la forma correcta.

1. **Aplanar las imágenes:** Convertimos cada imagen de 28x28 en un vector plano de 784 píxeles ($28 \times 28 = 784$).
2. **Normalizar los píxeles:** Los valores de los píxeles van de 0 a 255. Los escalamos para que estén entre 0 y 1. Esto ayuda a que el entrenamiento sea más estable.

```
1  # Aplanar las imagenes: de (28, 28) a (784,)
2  x_train = x_train.reshape(60000, 784)
3  x_test = x_test.reshape(10000, 784)
4
5  # Normalizar los valores de pixeles a un rango de 0 a 1
6  x_train = x_train.astype('float32') / 255
7  x_test = x_test.astype('float32') / 255
8
9  # Nota: No preprocesamos las etiquetas (y_train, y_test)
       aun.
10 # Keras lo puede hacer internamente.
```

Paso 3: Diseñar la Arquitectura del Modelo

Vamos a construir un modelo secuencial simple con tres capas:



Paso 3: Diseñar la Arquitectura del Modelo

Vamos a construir un modelo secuencial simple con tres capas:

- **Capa de Entrada (implícita):** Recibirá nuestros vectores de 784 píxeles.
- **Capa Oculta 1:** Una capa **Densa** con 512 neuronas y activación **ReLU**.
- **Capa de Salida:** Una capa **Densa** con 10 neuronas (una para cada dígito) y activación **Softmax** para obtener probabilidades.

Paso 3: Codificando el Modelo en Keras

Usamos la API Sequential de Keras, que nos permite apilar capas una tras otra.

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

model = Sequential([
    # Capa de entrada (definida en la primera capa)
    # 784 entradas (pixeles)
    Dense(512, activation='relu', input_shape=(784,)),

    # Capa de salida
    # 10 salidas (una por digito), con softmax
    Dense(10, activation='softmax')
])

# Ver un resumen de la arquitectura
model.summary()
```

Paso 3: Codificando el Modelo en Keras

Usamos la API `Sequential` de Keras, que nos permite apilar capas una tras otra.

```
model.summary()
```

Este comando es muy útil para ver el número de capas, la forma de sus salidas y la cantidad de parámetros (pesos y sesgos) que el modelo necesita aprender.

Paso 4: Compilar el Modelo - La Función de Pérdida

Antes de entrenar, debemos "compilar" el modelo. Esto requiere definir tres cosas. La primera es la **función de pérdida** (loss function).

- Mide qué tan "mal" está funcionando el modelo. Compara las predicciones del modelo con los valores reales.
- El objetivo del entrenamiento es **minimizar** esta función.
- Para clasificación multiclase, usamos **Categorical Cross-Entropy**.

Paso 4: Compilar el Modelo - La Función de Pérdida

La ecuación para Cross-Entropy Categórica es:

$$L(y, \hat{y}) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$$

Donde:

- K es el número de clases (10 en nuestro caso).
- y_i es 1 si la clase i es la verdadera, y 0 en otro caso.
- $\hat{y}_i$ es la probabilidad predicha por el modelo para la clase i.

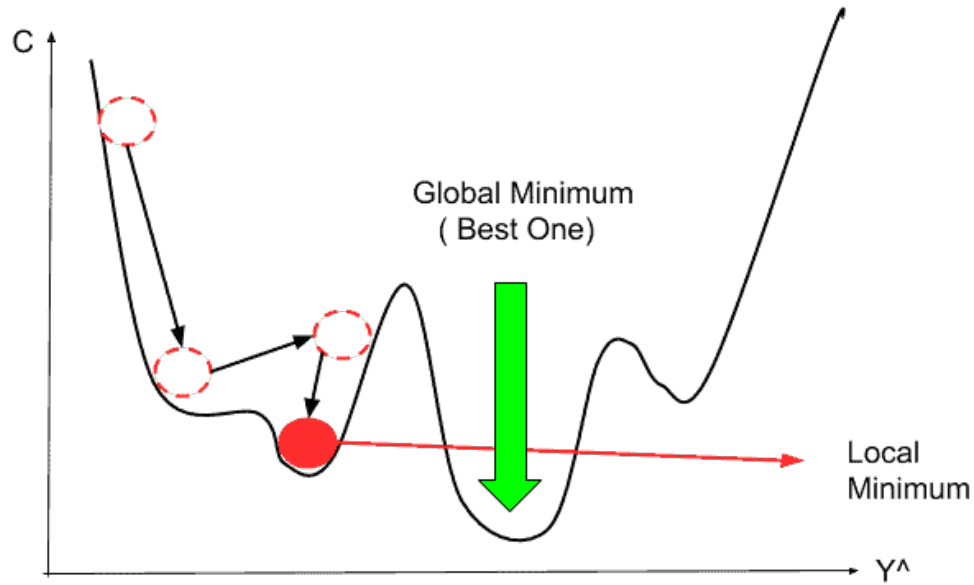
Paso 4: Compilar el Modelo - El Optimizador

El segundo componente es el **optimizador**.

- Es el algoritmo que ajusta los pesos (w) y sesgos (b) de la red para minimizar la función de pérdida.
- Funciona usando un método llamado **descenso de gradiente**. Imagina bajar una montaña buscando el punto más bajo, dando pasos en la dirección de la pendiente más pronunciada.
- **Adam** (Adaptive Moment Estimation) es un optimizador muy popular y eficiente que funciona bien en la mayoría de los casos.

Paso 4: Compilar el Modelo - El Optimizador

El segundo componente es el **optimizador**.



Visualización del descenso de gradiente en una superficie de pérdida.

Paso 4: Compilar el Modelo - La Métrica

El tercer y último componente es la **métrica**.

- Se usa para **monitorear y evaluar** el rendimiento del modelo durante el entrenamiento y las pruebas.
- A diferencia de la función de pérdida, la métrica no se usa para optimizar el modelo, solo para que nosotros, los humanos, entendamos qué tan bien está funcionando.
- Para problemas de clasificación, la métrica más común es la **precisión** (accuracy).

Precisión (Accuracy):

$$\text{Accuracy} = \frac{\text{Número de predicciones correctas}}{\text{Número total de predicciones}}$$

Paso 4: Código de Compilación

Ahora juntamos los tres componentes con el método `model.compile()`.

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

¿Por qué sparse_categorical_crossentropy?

Usamos esta variante de la cross-entropy porque nuestras etiquetas (y_{train}) son números enteros (0, 1, 2,...). Si fueran vectores "one-hot" (ej: [0,0,1,0,0...]), usaríamos `categorical_crossentropy`. Keras se encarga de la conversión por nosotros.

Paso 5: Entrenar el Modelo

¡Llegó el momento de entrenar! Usamos el método `model.fit()`.

- **epochs:** Una época es una pasada completa por todo el conjunto de datos de entrenamiento.
- **batch_size:** En lugar de procesar todas las 60,000 imágenes a la vez (lo cual consumiría mucha memoria), las procesamos en lotes (batches).
- **validation_data:** Después de cada época, el modelo se evaluará con datos que no ha visto (el conjunto de prueba) para monitorear su rendimiento real.

Paso 5: Entrenar el Modelo

```
history = model.fit(
    x_train,
    y_train,
    epochs=5,
    batch_size=128,
    validation_data=(x_test, y_test)
)
```

Entendiendo la Salida del Entrenamiento

Mientras el modelo entrena, verás algo como esto por cada época:

```
Epoch 1/5
469/469 [==============================] - 4s 8ms/step
- loss: 0.2845 - accuracy: 0.9193
- val_loss: 0.1384 - val_accuracy: 0.9587
```

- **loss:** El valor de la función de pérdida en los datos de entrenamiento. ¡Queremos que baje!
- **accuracy:** La precisión en los datos de entrenamiento. ¡Queremos que suba!
- **val_loss:** La pérdida en los datos de validación (test). Es un indicador clave.
- **val_accuracy:** La precisión en los datos de validación. **Para saber generalización del modelo.**

Visualizando el Historial de Entrenamiento

Es una buena práctica graficar la precisión y la pérdida para entender mejor el proceso de entrenamiento.

```
import matplotlib.pyplot as plt

# Graficar la precision del entrenamiento y validacion
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Paso 6: Evaluar el Rendimiento Final

Una vez entrenado, evaluamos el rendimiento final del modelo en el conjunto de prueba con `model.evaluate()`.

```
1 test_loss, test_acc = model.evaluate(x_test, y_test,
      verbose=2)
2
3 print(f'\nTest accuracy: {test_acc:.4f}')
```

Salida esperada:

```
313/313 - 1s - loss: 0.0782 - accuracy: 0.9765
Test accuracy: 0.9765
```

¡Con un modelo muy simple, hemos logrado una precisión de más del **97%**! Esto significa que el modelo clasifica correctamente 97 de cada 100 dígitos escritos a mano que nunca antes había visto.

Paso 7: Hacer una Predicción

Usemos nuestro modelo para predecir un solo dígito del conjunto de prueba.

```python
import numpy as np

# Tomamos la primera imagen del conjunto de test
sample_image = x_test[0]
# El modelo espera un lote de imagenes, asi que anadimos una dimension
sample_image = np.expand_dims(sample_image, axis=0)

# Realizar la prediccion
predictions = model.predict(sample_image)
predicted_class = np.argmax(predictions[0])
true_class = y_test[0]

print(f"Prediccion del modelo: {predicted_class}")
print(f"Clase real: {true_class}")
```

La salida de predictions es un array de 10 probabilidades de Softmax. np.argmax nos da el índice del valor más alto, que corresponde al dígito predicho.

Resumen del Código Completo

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

# 1. Cargar y preprocesar datos
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255

# 2. Construir el modelo
model = Sequential([
    Dense(512, activation='relu', input_shape=(784,)),
    Dense(10, activation='softmax')
])

# 3. Compilar el modelo
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# 4. Entrenar el modelo
model.fit(x_train, y_train, epochs=5, batch_size=128)

# 5. Evaluar el modelo
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc:.4f}')
```

Parte 3

Ejercicios con ANN

Ejercicios Prácticos

Intenta modificar el código para ver cómo cambian los resultados.

1. **Cambia el número de épocas:**
   - ¿Qué pasa si entrenas por solo 1 época?
   - ¿Y si entrenas por 20 épocas? ¿Mejora la precisión? ¿Aparece el sobreajuste (overfitting)?
2. **Modifica la arquitectura de la red:**
   - Cambia el número de neuronas en la capa oculta. Prueba con 128 o 1024. ¿Cómo afecta esto a la `val_accuracy` y al tiempo de entrenamiento?
   - Añade una segunda capa oculta. Por ejemplo: `Dense(256, activation='relu')`. ¿Un modelo más profundo es siempre mejor?

Ejercicios Prácticos (Continuación)

3. **Prueba un optimizador diferente:**
   - En `model.compile()`, cambia `optimizer='adam'` por `optimizer='sgd'` (Descenso de Gradiente Estocástico) o `optimizer='rmsprop'`.
   - ¿Convergen más rápido o más lento? ¿Alcanzan la misma precisión?
4. **Cambia la función de activación:**
   - En la capa oculta, reemplaza `activation='relu'` por `activation='sigmoid'`.
   - ¿Cómo impacta esto en la precisión final y la velocidad del entrenamiento?

Objetivo del ejercicio

El objetivo no es solo obtener la máxima precisión, sino desarrollar una intuición sobre cómo cada hiperparámetro afecta el comportamiento del modelo.

Un Concepto Clave: Sobreajuste (Overfitting)

El sobreajuste ocurre cuando un modelo aprende "demasiado bien" los datos de entrenamiento, incluyendo el ruido y los detalles específicos.

- **Señal de Overfitting:** La precisión en el entrenamiento (accuracy) sigue subiendo, pero la precisión en la validación (val_accuracy) se estanca o empieza a bajar.
- El modelo pierde su capacidad de **generalizar** a datos nuevos.
- **Causas comunes:** Un modelo demasiado complejo para los datos, o entrenar por demasiadas épocas.

Conclusiones

- El Deep Learning se basa en **redes neuronales profundas** para aprender patrones complejos directamente de los datos.
- Una neurona artificial calcula una **suma ponderada** y aplica una **función de activación**.
- **Keras** ofrece una interfaz simple y poderosa para construir, compilar y entrenar modelos de Deep Learning.
- El proceso clave es: **Definir la arquitectura -¿ Compilar (loss, optimizer, metrics) -¿ Entrenar (fit) -¿ Evaluar**.
- ¡Con unas pocas líneas de código, creaste un modelo capaz de reconocer dígitos con alta precisión!

Próximos Pasos en Deep Learning

Este fue solo el comienzo. Revisar a futuro:

- **Redes Neuronales Convolucionales (CNNs):** El estándar de oro para tareas de visión por computador. Son expertas en encontrar patrones espaciales en imágenes.
- **Redes Neuronales Recurrentes (RNNs):** Diseñadas para trabajar con datos secuenciales como texto o series de tiempo.
- **Técnicas de Regularización:** Métodos para combatir el overfitting, como Dropout.
- **Transfer Learning:** Usar modelos pre-entrenados en grandes datasets para resolver problemas con menos datos.

¿Preguntas?

Gracias por su atención

Universidad EAFIT - August 25, 2025