

5. Redes neuronales básicas

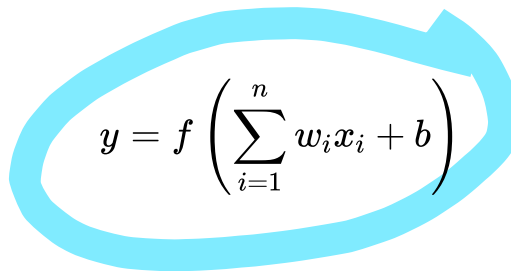
Fundamentos de Redes Neuronales

Las redes neuronales artificiales son modelos computacionales inspirados en el funcionamiento del cerebro humano. Se componen de unidades básicas llamadas **neuronas artificiales**, que están organizadas en capas y se conectan entre sí mediante **pesos**. Su principal propósito es aprender representaciones y patrones en los datos a través de un proceso de entrenamiento.

Neuronas artificiales y perceptrón

Modelo matemático de una neurona artificial

Una neurona artificial recibe un conjunto de entradas, las multiplica por un conjunto de pesos, les suma un sesgo y aplica una función de activación para obtener una salida:


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

Donde:

- x_i son las entradas.
- w_i son los pesos asociados a cada entrada.
- b es el sesgo.
- f es la función de activación.
- y es la salida de la neurona.

Esta estructura es la base de cualquier red neuronal.

Diferencias entre perceptrón simple y perceptrón multicapa (MLP)

- **Perceptrón simple:** Tiene una única capa de neuronas y solo puede resolver problemas linealmente separables (ej. la compuerta lógica AND).

- **Perceptrón multicapa (MLP):** Contiene múltiples capas de neuronas (entrada, ocultas y salida), lo que le permite aprender patrones más complejos y resolver problemas no lineales.
-

Funciones de activación

Las funciones de activación introducen no linealidad en la red neuronal, permitiendo que aprenda relaciones más complejas.

1. ReLU (Rectified Linear Unit)

- Fórmula: $f(x) = \max(0, x)$
- Es la más utilizada en redes profundas porque evita el problema del desvanecimiento del gradiente.

2. Sigmoid

- Fórmula: $f(x) = \frac{1}{1+e^{-x}}$
- Se usa en la última capa de modelos de clasificación binaria.
- Problema: Saturación en valores extremos, lo que afecta el aprendizaje.

3. Tanh (Tangente hiperbólica)

- Fórmula: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Rango de salida: $[-1, 1]$
- Se usa en capas ocultas cuando los valores pueden ser negativos.

4. Softmax

- Fórmula:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- Se usa en la capa de salida para clasificación multiclase.
-

Arquitectura de redes neuronales

Las redes neuronales están compuestas por diferentes capas:

1. Capa de entrada

- Recibe los datos de entrada sin procesamiento.

- Cada neurona representa una característica del conjunto de datos (ej., píxeles de una imagen).

2. Capas ocultas

- Procesan la información mediante operaciones matemáticas y funciones de activación.
- Cuantas más capas ocultas haya, mayor capacidad tiene la red para capturar patrones complejos.

3. Capa de salida

- Genera la predicción final del modelo.
- Su número de neuronas depende del número de clases a predecir.
- Usa funciones de activación adecuadas (sigmoide para clasificación binaria, softmax para multiclase).

Pesos y sesgos

- **Pesos (w)**: Definen la importancia de cada conexión entre neuronas.
- **Sesgos (b)**: Permiten desplazar la función de activación para mejorar la representación del modelo.

Proceso de Entrenamiento y Retropropagación en Redes Neuronales

El entrenamiento de una red neuronal se basa en ajustar sus pesos y sesgos para minimizar el error entre las predicciones y los valores reales. Este proceso se lleva a cabo mediante la **propagación hacia adelante** y la **retropropagación del error**, utilizando funciones de pérdida y algoritmos de optimización.

Propagación hacia adelante

En la **propagación hacia adelante**, la información fluye desde la capa de entrada hasta la capa de salida, pasando por las capas ocultas. Cada neurona realiza las siguientes operaciones:

1. Cálculo de la suma ponderada

Cada neurona recibe entradas, las multiplica por sus respectivos pesos y suma el sesgo:

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

2. Aplicación de la función de activación

Se usa una función de activación para introducir no linealidad:

$$a = f(z)$$

3. Paso a la siguiente capa

La salida de cada neurona se convierte en la entrada de la siguiente capa, hasta llegar a la capa de salida, donde se obtiene la predicción final.

Función de pérdida y optimización

Para que la red neuronal aprenda, es necesario medir qué tan lejos está la predicción del valor real. Esto se logra con una **función de pérdida**, que calcula el error del modelo.

Funciones de pérdida más comunes

- **Error Cuadrático Medio (MSE - Mean Squared Error)**

Utilizado en problemas de regresión. Calcula el promedio de los errores al cuadrado:

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

- **Entropía cruzada (Cross-Entropy Loss)**

Usada en clasificación, mide la diferencia entre dos distribuciones de probabilidad:

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

Donde:

- y_i es la etiqueta real (0 o 1).
- \hat{y}_i es la probabilidad predicha por el modelo.

Algoritmos de optimización

Para minimizar la función de pérdida, se usan algoritmos de optimización que ajustan los pesos de la red neuronal.

1. Descenso de Gradiente (Gradient Descent)

- Calcula la derivada de la función de pérdida con respecto a los pesos.
- Ajusta los pesos en la dirección opuesta al gradiente.
- Fórmula de actualización de pesos:

$$w = w - \alpha \frac{\partial L}{\partial w}$$

Donde α es la tasa de aprendizaje.

2. Adam (Adaptive Moment Estimation)

- Combina **momentum** y **RMSProp** para un mejor ajuste.
- Se adapta a cada peso individualmente, acelerando la convergencia.
- Es la opción más utilizada en la práctica.

Retropropagación del Error

La **retropropagación** es el proceso que ajusta los pesos y sesgos para reducir el error de la red neuronal.

1. Cálculo del error en la capa de salida

Se mide la diferencia entre la predicción y el valor real usando la función de pérdida.

2. Propagación del error hacia atrás

Se usa la **regla de la cadena** para calcular la contribución de cada peso al error total:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

3. Ajuste de pesos y sesgos

Cada peso se actualiza utilizando el algoritmo de optimización seleccionado (como Adam o Descenso de Gradiente).

4. Repetición del proceso

Se repite la propagación hacia adelante y la retropropagación en múltiples **épocas** hasta que la red converge y el error se minimiza.

Concepto de Gradiente y Precisión del Modelo

El **gradiente** es la derivada de la función de pérdida con respecto a los pesos. Cuanto más pronunciada sea la pendiente, mayor será el ajuste necesario. El objetivo es encontrar el punto donde el gradiente es **cercano a cero**, indicando que la red ha aprendido correctamente.

Este proceso es la clave del **aprendizaje profundo**, permitiendo a las redes neuronales reconocer patrones y realizar predicciones precisas.

Introducción a TensorFlow y Keras

En esta sección, exploraremos **TensorFlow** y **Keras**, dos de las librerías más utilizadas para construir redes neuronales en Python. Veremos sus diferencias, cuándo usar cada una y cómo construir un modelo básico en Keras.

TensorFlow vs Keras

¿Qué es TensorFlow?

TensorFlow es una biblioteca de código abierto desarrollada por **Google** para computación numérica y aprendizaje automático. Ofrece herramientas para:

- Definir modelos de redes neuronales.
- Entrenar y optimizar modelos de deep learning.
- Implementar modelos en dispositivos móviles y entornos de producción.

¿Qué es Keras?

Keras es una API de alto nivel que se integra con TensorFlow y simplifica la construcción de redes neuronales. Antes, Keras podía funcionar con diferentes backends (Theano, CNTK), pero desde TensorFlow 2.0, se ha convertido en su API oficial.

Diferencias clave entre TensorFlow y Keras

Característica	TensorFlow	Keras
Nivel de abstracción	Bajo (más flexible)	Alto (más fácil de usar)
Facilidad de uso	Mayor control, más complejo	Intuitivo y rápido de prototipar
Velocidad	Más optimizado para producción	Más adecuado para prototipos
Compatibilidad	Soporte para modelos en producción	Integrado con TensorFlow

¿Cuándo usar cada uno?

- **Usar Keras** cuando se necesita rapidez en el desarrollo y facilidad de uso.
- **Usar TensorFlow puro** cuando se requiere más control sobre el modelo, optimización avanzada o personalización de capas.

Estructura básica de un modelo en Keras

1. Definición de capas

Un modelo en Keras se construye con una estructura **secuencial** o con una API funcional. La más sencilla es la secuencial:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Definir un modelo secuencial
modelo = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)), # Capa de entrada con 128 neuronas
    layers.Dense(64, activation='relu'), # Capa oculta con 64 neuronas
    layers.Dense(10, activation='softmax') # Capa de salida para clasificación en 10 clases
])
```

2. Compilación del modelo

Antes de entrenar, el modelo debe ser compilado con:

- Una **función de pérdida**: Indica cómo de lejos están las predicciones de los valores reales.
- Un **optimizador**: Ajusta los pesos de la red para minimizar la pérdida.
- **Métricas**: Evalúan el desempeño del modelo.

```
modelo.compile(  
    optimizer='adam',  
    loss='sparse_categorical_crossentropy', # Para clasificación multiclase  
    metrics=['accuracy']  
)
```

3. Entrenamiento del modelo

Para entrenar el modelo, usamos el método `fit()`, que recibe:

- Los datos de entrada y etiquetas.
- Número de **épocas** (cantidad de veces que se verá el conjunto de datos).
- Tamaño del **lote** (cantidad de muestras procesadas en cada iteración).

```
modelo.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val))
```

4. Evaluación del modelo

Después del entrenamiento, el modelo se evalúa en datos de prueba:

```
test_loss, test_acc = modelo.evaluate(X_test, y_test)  
print(f'Precisión en datos de prueba: {test_acc:.2f}')
```

Conclusión

Keras es una herramienta poderosa que permite definir modelos de deep learning de manera rápida y sencilla dentro de TensorFlow. Su API de alto nivel facilita la experimentación con arquitecturas neuronales, mientras que TensorFlow proporciona la base robusta para optimización y despliegue en producción.

