Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

**Trabajo fin de grado**

# Guía Completa de Deep Learning

*Complete Guide to Deep Learning*

Eduardo José Barrios García

Basado en el Libro ”IA y Deep Learning” de Jordi Torres

# Table of contents

# 1 Introducción al deep learning

## 1.1 ¿Qué es el Deep Learning?

El Deep Learning es una rama del aprendizaje automático (*Machine Learning*) que utiliza redes neuronales profundas para modelar patrones complejos en los datos. Inspirado en la estructura del cerebro humano, permite que los modelos aprendan representaciones jerárquicas y abstractas directamente a partir de los datos, sin necesidad de diseñar manualmente las características.

Este enfoque se ha popularizado en los últimos años debido a la mejora de las capacidades computacionales, el acceso a grandes volúmenes de datos y los avances en algoritmos de optimización. Como resultado, Deep Learning se ha aplicado con éxito en áreas como:

- **Medicina**: Detección de enfermedades mediante imágenes médicas, como estudios sobre glaucoma.
- **Automoción**: Vehículos autónomos que procesan información en tiempo real.
- **Entretenimiento**: Sistemas de recomendación en plataformas como Netflix.
- **Finanzas**: Predicción de mercados y detección de fraudes.

## 1.2 Entorno de Trabajo

### 1.2.1 Herramientas de trabajo

Para desarrollar proyectos de Deep Learning se pueden utilizar varios entornos de trabajo, siendo **Google Colab** y el **entorno local con GPU** dos de las opciones más destacadas:

- **Google Colab**: Es un servicio en la nube que permite ejecutar notebooks de Python sin necesidad de configuraciones locales complejas.[8] Ofrece recursos de GPU y TPU gratuitos, lo que facilita el entrenamiento de modelos complejos.
- **Entorno Local con GPU**: Para usuarios que prefieren más control sobre el hardware o trabajan con datos sensibles, configurar un entorno local con GPU es una alternativa válida. Esto permite personalizar la instalación y aprovechar al máximo los recursos disponibles.

Cada opción tiene ventajas y desventajas: **Google Colab** es ideal para experimentar rápidamente, mientras que un entorno local es más eficiente para grandes volúmenes de datos y permite mayor personalización.

### 1.2.2 Planificación íntegra del proyecto

Para la gestión del proyecto se van a usar dos herramientas:

1. Un sitio específico de documentación, a modo de cuaderno de bitácora, donde se irá subiendo todo el contenido de investigación y notas día a día sobre el proyecto, disponible en:

   **https://edujbarrios.github.io/My-TFG-Logbook/docs/Documentation%20Overview/intro**.

2. Una Herramienta de gestión de proyectos llamada **Projectmad**, una herramienta desarrollada por mí mismo cuyo objetivo es ser una alternativa a **Trello**, incluso con más capacidades y mejoras en interfaz, para hacerlo lo más intuitivo posible a la vez que contiene todos los requisitos necesarios para hacer un seguimiento completo a un proyecto. Su código es cerrado por ahora, pero planeo publicarlo a lo largo del transcurso de este TFG.

## 1.3 TensorFlow, Keras y PyTorch

### 1.3.1 TensorFlow

**TensorFlow** es una plataforma popular para desarrollar modelos de Deep Learning. Creada por Google, se caracteriza por su flexibilidad y capacidad para producción e investigación. TensorFlow ofrece optimizaciones específicas para GPUs y permite implementar arquitecturas avanzadas.

#### 1.3.1.1 TensorFlow Playground

TensorFlow Playground es una herramienta visual que facilita la experimentación con redes neuronales. Permite observar el impacto de diferentes hiperparámetros y funciones de activación de manera interactiva.

### 1.3.2 Keras

**Keras** es una API de alto nivel que corre sobre TensorFlow, proporcionando una manera rápida y sencilla de construir modelos. Es ideal para prototipado rápido y facilita la experimentación con arquitecturas diversas.

### 1.3.3 PyTorch

**PyTorch**, desarrollado por Facebook's AI Research lab (FAIR), es otro framework ampliamente utilizado. Su enfoque dinámico permite una mayor flexibilidad durante el entrenamiento y la depuración de modelos. PyTorch utiliza gráficos computacionales dinámicos, lo que lo hace especialmente adecuado para tareas como redes recurrentes.

## 1.4 Otras Librerías y recursos importantes usados en el proyecto.

### 1.4.1 Librería NumPy

**NumPy** es una biblioteca esencial para la computación numérica en Python y juega un rol crucial en proyectos de Deep Learning. Permite trabajar con arreglos multidimensionales (tensores) y realizar operaciones matemáticas avanzadas de manera eficiente.

#### 1.4.1.1 Tensores

Un **tensor** generaliza conceptos como escalares, vectores y matrices a un número arbitrario de dimensiones:

- Un escalar es un tensor de orden 0.
- Un vector es un tensor de orden 1.
- Una matriz es un tensor de orden 2.

### 1.4.2 Manipulación de Tensores

NumPy permite diversas operaciones sobre tensores:

- **Remodelación**: Cambiar su forma con `reshape()`.
- **Transposición**: Intercambiar ejes usando `transpose()`.
- **Selección de elementos**: Usar indexación avanzada para seleccionar valores específicos.

## 1.5 Otra libería 1

## 1.6 Otra libreria 2

## 1.7 Otra Librería 3

# 2 Fundamentos del Deep Learning

## 2.1 Redes Neuronales Densamente Conectadas

### 2.1.1 Una Neurona Artificial

Una neurona artificial es la unidad básica de una red neuronal, inspirada en las neuronas biológicas. Su función es recibir un conjunto de entradas, ponderarlas, aplicar una función de activación, y producir una salida.

#### 2.1.1.1 Introducción a la Terminología y Notación Básica

La neurona se caracteriza por: - **Entradas** ($x$): Representan los datos de entrada a la neurona. - **Pesos** ($w$): Cada entrada está asociada a un peso que determina la influencia de esa entrada en la salida de la neurona. - **Bias** ($b$): Un término constante que permite a la neurona ajustar la salida independientemente de las entradas.

La salida de una neurona $y$ se calcula mediante la ecuación:

$$y = f\left(\sum_i w_i x_i + b\right)$$

donde $f$ es la función de activación.

#### 2.1.1.2 Algoritmos de Regresión

En el contexto de redes neuronales, los algoritmos de regresión se utilizan para ajustar los pesos y el bias de manera que se minimice el error entre las predicciones de la red y los valores reales. La función de error comúnmente utilizada es el error cuadrático medio (MSE):

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

#### 2.1.1.3 Una Neurona Artificial Simple

Una neurona artificial simple realiza una operación lineal sobre sus entradas y aplica una función de activación no lineal. Esto le permite aproximar relaciones complejas en los datos.

La función de esta neurona artificial puede expresarse mediante:

$$z = b + \sum_i x_i w_i$$

donde: - $x_i$ son las entradas, - $w_i$ son los pesos asociados a cada entrada, - $b$ es el término bias.

La salida $y$ depende de la función de activación aplicada al valor de $z$. Por ejemplo, usando una **función escalón**:

$$y = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{si } z < 0 \end{cases}$$

#### 2.1.1.4 Función Sigmoid

Otra función de activación común es la **función sigmoid**, que retorna un valor entre 0 y 1 para cualquier entrada. Es útil en problemas de clasificación binaria, donde queremos interpretar la salida como una probabilidad:

$$y = \frac{1}{1 + e^{-z}}$$

Esta función aplana el valor de salida, limitándolo entre 0 y 1.

## 2.2 Redes Neuronales

Las redes neuronales están compuestas por múltiples neuronas interconectadas. Estas conexiones permiten a la red aprender representaciones de los datos y resolver problemas complejos.

#### 2.2.0.1 Perceptrón

El **perceptrón** es la forma más básica de red neuronal y se utiliza principalmente para problemas de clasificación binaria. Su función es aprender una línea de decisión que separe las clases en el espacio de características.

#### 2.2.0.2 Perceptrón Multicapa

El **perceptrón multicapa** (MLP) consiste en varias capas de neuronas. Las capas intermedias entre la entrada y la salida se conocen como **capas ocultas** y permiten a la red aprender representaciones más abstractas.

#### 2.2.0.3 Perceptrón Multicapa para Clasificación

En problemas de clasificación, el MLP asigna cada entrada a una categoría específica. Utiliza una función de activación como **softmax** en la capa de salida para convertir las puntuaciones en probabilidades.

### 2.2.1 Función de Activación Softmax

La función **softmax** es utilizada en la capa de salida de redes neuronales para problemas de clasificación multiclase. Convierte los valores de salida en probabilidades que suman 1:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

donde $z_i$ es la salida lineal de la neurona $i$ y la suma se realiza sobre todas las neuronas en la capa de salida.

Softmax ayuda a interpretar los resultados en tareas de clasificación, indicando la probabilidad de que una entrada pertenezca a cada clase.

## 2.3 Redes Neuronales en Keras

### 2.3.1 Precarga de los Datos en Keras

Antes de construir una red neuronal en Keras, es fundamental preparar y cargar los datos. Keras facilita este proceso mediante módulos como `keras.datasets`, que incluye conjuntos de datos comunes como MNIST, CIFAR-10, entre otros. Estos conjuntos de datos se pueden cargar directamente, lo que simplifica el proceso de experimentación.

```
from tensorflow.keras.datasets import mnist

# Cargar el conjunto de datos de ejemplo Mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

### 2.3.2 Preprocesado de Datos de Entrada en una Red Neuronal

El preprocesado es crucial para que el modelo funcione correctamente. En redes neuronales, los datos suelen normalizarse para que sus valores estén en un rango específico, generalmente entre 0 y 1. Este proceso puede incluir también la codificación de etiquetas en formato one-hot si es necesario para tareas de clasificación.

```
# Normalizar las imágenes a valores entre 0 y 1
train_images = train_images / 255.0
test_images = test_images / 255.0
```

### 2.3.3 Definición del Modelo

La definición del modelo en Keras se hace mediante la creación de una secuencia de capas utilizando `Sequential` o el API funcional. Este paso implica definir las capas, el número de neuronas en cada capa, y las funciones de activación.

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

# Definir el modelo
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='sigmoid'),
    Dense(10, activation='softmax')
])
```

### 2.3.4 Configuración del Proceso de Aprendizaje

Una vez definido el modelo, es necesario configurarlo para el entrenamiento, lo que implica especificar el optimizador, la función de pérdida y las métricas de evaluación.

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

### 2.3.5 Entrenamiento del Modelo

El entrenamiento del modelo se realiza usando el método `fit`, donde se pasan los datos de entrenamiento, el número de épocas y el tamaño de lote.

```
model.fit(train_images, train_labels, epochs=5, batch_size=32)
```

### 2.3.6 Evaluación del Modelo

Después del entrenamiento, es importante evaluar el rendimiento del modelo en el conjunto de datos de prueba. Esto permite verificar su capacidad de generalización.

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print("Precisión en el conjunto de prueba:", test_acc)
```

### 2.3.7 Generación de Predicciones

Una vez que el modelo ha sido entrenado y evaluado, se pueden realizar predicciones sobre nuevos datos. Este paso es esencial para la implementación del modelo en un entorno de producción.

```
predictions = model.predict(test_images)
```

### 2.3.8 Flujo completo

Por resumir los pasos en texto, el flujo completo con un modelo en Keras incluye:

1. **Precarga de los datos**: Cargar el conjunto de datos.
2. **Preprocesado de datos**: Normalizar y ajustar los datos.
3. **Definición del modelo**: Crear la arquitectura de la red.
4. **Configuración del proceso de aprendizaje**: Establecer optimizador, función de pérdida y métricas.
5. **Entrenamiento del modelo**: Ajustar los pesos en función de los datos de entrenamiento.
6. **Evaluación del modelo**: Medir el rendimiento del modelo en datos no vistos.
7. **Generación de predicciones**: Utilizar el modelo entrenado para predecir resultados en nuevos datos.

## 2.4 Fundamentos Prácticos de Redes Neuronales

El entrenamiento de una red neuronal es un proceso iterativo donde el modelo ajusta sus pesos para minimizar el error en las predicciones. Este proceso se basa en optimizar los pesos de las conexiones neuronales para que la salida de la red se aproxime lo más posible a los valores deseados.

### 2.4.1 Visión Global

En términos generales, el proceso de aprendizaje implica:

1. **Inicialización de pesos**: Los pesos se establecen con valores iniciales (generalmente aleatorios).
2. **Propagación hacia adelante (Forward Pass)**: Se calculan las salidas de la red en función de las entradas y los pesos actuales.
3. **Cálculo de la pérdida**: La diferencia entre la salida real y la salida deseada se mide utilizando una función de pérdida.

### 2.4.2 Proceso Iterativo de Aprendizaje de una Red Neuronal

El aprendizaje en redes neuronales es un proceso iterativo. La red ajusta sus pesos mediante múltiples pasos (o épocas) en los que se calcula el error y se ajustan los pesos para reducir este error.

### 2.4.3 Piezas Clave del Proceso de Backpropagation

El **backpropagation** o retropropagación es el proceso mediante el cual se calcula el gradiente de la función de pérdida respecto a los pesos de la red. Esta información se utiliza para ajustar los pesos y reducir el error en futuras predicciones. La retropropagación es esencial para el aprendizaje supervisado en redes neuronales.

## 2.5 Descenso del Gradiente

El descenso del gradiente es el algoritmo central en el ajuste de los pesos de una red neuronal. Se basa en la idea de moverse en la dirección opuesta al gradiente de la función de pérdida para minimizar el error.

### 2.5.1 Algoritmo Básico de Descenso del Gradiente

En el algoritmo básico de descenso del gradiente, los pesos de un modelo se ajustan iterativamente en la dirección opuesta al gradiente de la función de pérdida. Este gradiente indica cómo cambian los errores del modelo en función de los pesos actuales. La actualización de los pesos se realiza multiplicando el gradiente por una tasa de aprendizaje y restando este valor de los pesos actuales. Este proceso se repite hasta que la función de pérdida se minimice suficientemente o se cumpla otro criterio de parada.

### 2.5.2 Tipos de Descenso del Gradiente

Existen varias variantes del descenso del gradiente:

- **Descenso del Gradiente Estocástico (SGD)**: Actualiza los pesos para cada muestra individual. Es ruidoso, pero más rápido [32].
- **Mini-Batch Gradient Descent**: Actualiza los pesos en pequeños lotes, combinando las ventajas del SGD y el descenso por lotes [33].
- **Descenso del Gradiente por Lotes**: Calcula el gradiente sobre todo el conjunto de datos antes de actualizar los pesos, pero es más lento en términos computacionales.

## 2.6 Función de Pérdida

La función de pérdida mide la discrepancia entre las predicciones de la red y los valores reales. Existen diferentes tipos de funciones de pérdida dependiendo del tipo de problema:

- **Error Cuadrático Medio (MSE)**: Común en problemas de regresión.
- **Entropía Cruzada**: Común en problemas de clasificación.

La función de pérdida guía el entrenamiento, indicando cuán lejos están las predicciones de los valores reales.

## 2.7 Optimizadores

Los optimizadores son algoritmos que se utilizan para actualizar los pesos de la red, basándose en el gradiente de la función de pérdida. Algunos de los optimizadores más comunes incluyen:

- **SGD (Descenso de Gradiente Estocástico)**: Una implementación simple del descenso del gradiente.
- **Adam**: Un optimizador que ajusta la tasa de aprendizaje durante el entrenamiento y suele converger más rápido.
- **RMSprop**: Optimiza el descenso del gradiente adaptando la tasa de aprendizaje en función de las medias cuadráticas de gradientes pasados.

Estos optimizadores permiten un ajuste más preciso y eficiente de los pesos, acelerando la convergencia del modelo y mejorando su rendimiento.

## 2.8 Parámetros e Hiperparámetros en Redes Neuronales

### 2.8.1 Parametrización de los Modelos

La parametrización de los modelos de redes neuronales es fundamental para ajustar el rendimiento del modelo. Los parámetros y los hiperparámetros determinan cómo la red procesa los datos y aprende.

#### 2.8.1.1 Motivación por los Hiperparámetros

Los hiperparámetros permiten ajustar el modelo para optimizar su rendimiento en una tarea específica. Elegir adecuadamente los hiperparámetros puede mejorar significativamente la precisión del modelo y su capacidad de generalización.

#### 2.8.1.2 Parámetros e Hiperparámetros

- **Parámetros**: Son los valores que la red aprende durante el entrenamiento, como los pesos y los sesgos.
- **Hiperparámetros**: Son valores que configuran el modelo y el proceso de entrenamiento, como la tasa de aprendizaje y el tamaño del lote.

#### 2.8.1.3 Grupos de Hiperparámetros

Los hiperparámetros pueden clasificarse en diferentes grupos según su propósito y el aspecto del modelo que afectan, como los hiperparámetros de arquitectura y los de optimización.

### 2.8.2 Hiperparámetros Relacionados con el Algoritmo de Aprendizaje

Algunos hiperparámetros importantes en el proceso de entrenamiento de una red neuronal incluyen:

#### 2.8.2.1 Número de Epochs

El número de épocas determina cuántas veces la red procesará todo el conjunto de datos durante el entrenamiento.

#### 2.8.2.2 Batch Size

El tamaño de lote determina cuántas muestras se procesan juntas antes de actualizar los pesos. Los tamaños de lote más pequeños pueden proporcionar una mayor precisión, pero también requieren más tiempo de procesamiento.

#### 2.8.2.3 Learning Rate y Learning Rate Decay

La tasa de aprendizaje controla el tamaño de los pasos en la actualización de los pesos. La **decadencia de la tasa de aprendizaje** ajusta el valor de la tasa de aprendizaje a medida que el entrenamiento avanza.

#### 2.8.2.4 Momentum

El momentum permite que el modelo mantenga la dirección del gradiente en actualizaciones sucesivas, lo cual ayuda a evitar oscilaciones y acelera la convergencia.

#### 2.8.2.5 Inicialización de los Pesos de los Parámetros

La inicialización adecuada de los pesos puede mejorar la velocidad de convergencia y la estabilidad del modelo durante el entrenamiento.

### 2.8.3 Funciones de Activación

Las funciones de activación añaden no linealidad al modelo, permitiendo que la red neuronal aprenda y represente relaciones complejas. Ejemplos de funciones de activación incluyen **ReLU**, **sigmoid** y **tanh**.

## 2.9 Redes Neuronales Convolucionales

### 2.9.1 Introducción a las Redes Neuronales Convolucionales

Las redes neuronales convolucionales (CNNs) son una clase de redes neuronales profundas diseñadas para procesar datos estructurados en forma de cuadrículas, como las imágenes. Las CNNs han demostrado ser muy efectivas en tareas de visión por computadora debido a su capacidad para aprender representaciones espaciales jerárquicas.

### 2.9.2 Componentes Básicos de una Red Neuronal Convolucional

#### 2.9.2.1 Operación de Convolución

La operación de convolución es el núcleo de las CNNs. Consiste en aplicar filtros (o kernels) sobre las entradas para extraer características de diferentes regiones de la imagen. La salida de una operación de convolución es un mapa de características que resalta patrones específicos en los datos de entrada

#### 2.9.2.2 Operación de Pooling

La operación de *pooling* se utiliza para reducir la dimensionalidad de los mapas de características, lo que disminuye el costo computacional y ayuda a evitar el sobreajuste. El *max pooling* es una técnica común, en la que se selecciona el valor máximo en cada región de la característica extraída.

### 2.9.3 Implementación de un Modelo Básico en Keras

Las CNNs se pueden implementar en Keras utilizando capas como `Conv2D` y `MaxPooling2D`. En esta sección, crearemos un modelo básico en Keras y veremos cómo entrenarlo y evaluarlo.

#### 2.9.3.1 Arquitectura Básica de una Red Neuronal Convolucional

Una arquitectura básica de CNN consiste en una secuencia de capas de convolución y *pooling*, seguida de una o más capas densas para la clasificación.

### 2.9.4 Hiperparámetros de la Capa Convolucional

Algunos de los hiperparámetros clave de una capa convolucional son:

#### 2.9.4.1 Tamaño y Número de Filtros

El tamaño de los filtros (kernels) determina el área de la entrada que cada neurona examina, mientras que el número de filtros define cuántas características se extraen en cada capa.

#### 2.9.4.2 Padding

El *padding* controla si se añade un borde de ceros alrededor de la entrada antes de aplicar la convolución, lo cual permite preservar el tamaño espacial de los datos.

#### 2.9.4.3 Stride

El *stride* o paso de la convolución determina el número de posiciones que el filtro se desplaza en cada paso. Valores más altos de *stride* resultan en una mayor reducción del tamaño del mapa de características.

### 2.9.5 Capas y Optimizadores

#### 2.9.5.1 Capas Adicionales y Optimización

Para mejorar el rendimiento, se pueden añadir capas adicionales y optimizadores específicos. `Adam` es un optimizador comúnmente utilizado en CNNs.

#### 2.9.5.2 Capas de Dropout y BatchNormalization

Las capas de *Dropout* ayudan a reducir el sobreajuste al desactivar neuronas aleatoriamente durante el entrenamiento, mientras que *BatchNormalization* normaliza las activaciones, acelerando el aprendizaje [59].

#### 2.9.5.3 Decaimiento del Ratio de Aprendizaje

El decaimiento de la tasa de aprendizaje es una técnica que reduce gradualmente la tasa de aprendizaje durante el entrenamiento, permitiendo al modelo converger más suavemente hacia el mínimo de la función de pérdida [60].

## 2.10 Técnicas de Prevención del Sobreentrenamiento

El sobreentrenamiento ocurre cuando un modelo se ajusta demasiado a los datos de entrenamiento, perdiendo su capacidad de generalizar a datos nuevos. Existen varias técnicas para mitigar el sobreentrenamiento, especialmente útiles cuando se dispone de conjuntos de datos pequeños.

### 2.10.1 Modelos a partir de Conjuntos de Datos Pequeños

Trabajar con conjuntos de datos pequeños es un desafío en Deep Learning, ya que puede llevar al sobreentrenamiento. Algunas estrategias incluyen:

- **Recolección de datos adicionales**: Si es posible, se recomienda ampliar el conjunto de datos.
- **Data Augmentation**: Aumentar la variabilidad de los datos mediante transformaciones.

### 2.10.2 Visualización del Comportamiento del Entrenamiento

La visualización de métricas como la pérdida y la precisión durante el entrenamiento puede ayudar a detectar el sobreentrenamiento. Si la precisión en el conjunto de entrenamiento sigue mejorando mientras que la precisión en el conjunto de validación se estanca o disminuye, el modelo está sobreentrenado.

### 2.10.3 Técnicas de Prevención del Sobreentrenamiento

Algunas técnicas comunes para prevenir el sobreentrenamiento incluyen:

#### 2.10.3.1 Regularización (L1 y L2)

La regularización L1 y L2 agregan un término de penalización a la función de pérdida, lo que limita el crecimiento de los pesos y reduce el sobreajuste.

#### 2.10.3.2 Dropout

El *Dropout* es una técnica que desactiva aleatoriamente neuronas durante el entrenamiento, lo que obliga a la red a aprender representaciones más generales.

#### 2.10.3.3 Early Stopping

La técnica de *Early Stopping* detiene el entrenamiento cuando el rendimiento en el conjunto de validación deja de mejorar, evitando que el modelo se ajuste demasiado.

## 2.11 Data Augmentation y Transfer Learning

### 2.11.1 Data Augmentation

El *Data Augmentation* es una técnica que genera variaciones de los datos de entrada, aumentando efectivamente el tamaño del conjunto de datos y mejorando la generalización del modelo.

#### 2.11.1.1 Transformaciones de Imágenes

Algunas transformaciones comunes incluyen:

- **Rotación**: Girar la imagen en un ángulo aleatorio.
- **Escalado**: Cambiar el tamaño de la imagen.
- **Traslación**: Mover la imagen en el plano XY.
- **Flip**: Voltear la imagen horizontal o verticalmente.

#### 2.11.1.2 Configuración de ImageGenerator

Keras proporciona la clase `ImageDataGenerator` para aplicar *Data Augmentation* de manera eficiente.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)
```

#### 2.11.1.3 Código del Caso de Estudio

En un caso de estudio, podemos aplicar *Data Augmentation* a un conjunto de datos de imágenes para mejorar el rendimiento del modelo en datos de prueba.

### 2.11.2 Transfer Learning

El *Transfer Learning* permite reutilizar un modelo preentrenado en una nueva tarea, ahorrando tiempo y mejorando el rendimiento cuando los datos son limitados.

#### 2.11.2.1 Transfer Learning

Consiste en utilizar un modelo preentrenado en una tarea similar y adaptarlo a una nueva tarea mediante un ajuste fino de los últimos parámetros.

#### 2.11.2.2 Feature Extraction

La extracción de características implica congelar las capas iniciales del modelo y utilizar las salidas de estas capas como características de entrada para una nueva tarea.

#### 2.11.2.3 Fine-Tuning

El *fine-tuning* o ajuste fino desbloquea las últimas capas del modelo y las ajusta en el nuevo conjunto de datos para mejorar la precisión. Este punto va a ser crucuial de cara a mi investigación en el TFG, para realizar modelos "finetuneados" que sean capaces de qnalizar imágenes de glaucoma.

## 2.12 CAPÍTULO 12: Arquitecturas Avanzadas de Redes Neuronales

### 2.12.1 API Funcional de Keras

La API funcional de Keras permite construir modelos más complejos, como redes con múltiples entradas o salidas y capas que no están en secuencia.

#### 2.12.1.1 Modelo Secuencial

El modelo secuencial es útil para redes simples, donde las capas están apiladas una tras otra.

#### 2.12.1.2 Modelos Complejos

La API funcional permite la creación de arquitecturas complejas, como redes residuales y modelos con conexiones personalizadas.

### 2.12.2 Redes Neuronales Preentrenadas

Las redes neuronales preentrenadas permiten reutilizar arquitecturas populares que ya han sido entrenadas en grandes conjuntos de datos como ImageNet.

#### 2.12.2.1 Redes Neuronales con Nombre Propio

Modelos como **ResNet**, **VGG**, y **Inception** están disponibles como modelos preentrenados en Keras.

#### 2.12.2.2 Acceso a Redes Preentrenadas con la API Keras

Keras permite cargar modelos preentrenados fácilmente.

```
from tensorflow.keras.applications import ResNet50

model = ResNet50(weights='imagenet')
```

### 2.12.3 Uso de Redes Preentrenadas con Keras

#### 2.12.3.1 Conjunto de Datos CIFAR-10

El conjunto de datos CIFAR-10 es una colección de imágenes en 10 clases, comúnmente utilizado para probar redes preentrenadas.

#### 2.12.3.2 Red Neuronal ResNet50

**ResNet50** es una red profunda basada en bloques residuales, ideal para tareas de clasificación de imágenes complejas.

#### 2.12.3.3 Red Neuronal VGG19

**VGG19** es una red con 19 capas que se utiliza ampliamente en tareas de visión por computadora.

Con estos modelos, podemos realizar tareas de clasificación avanzada y mejorar la precisión utilizando la técnica de *Transfer Learning*.

# 3 Referencias

Se listarán aquí, y obviamente dentro de las descripciones textuales que asñi lo requieranm sin embargo en este documento no se ha precisado la utilización de referencias, pues el objeto de este resumen se basa principalmente en el libro ***IA y Deep Learning de Jordi Torres.***