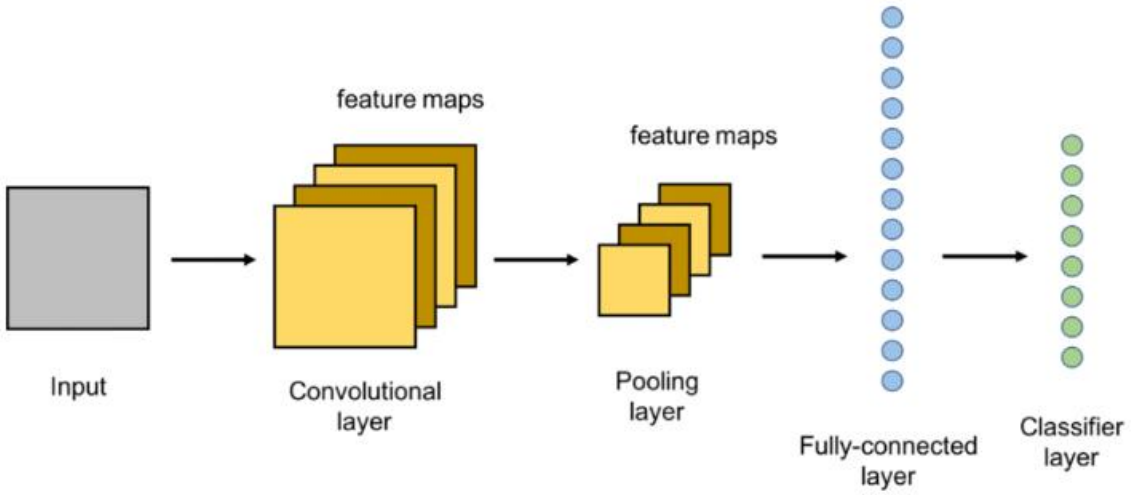
Unidad 5. Redes neuronales convolucionales



"Programación de Inteligencia Artificial"

Curso de Especialización en Inteligencia Artificial y Big Data

Contenido

1. Introducción a las Redes Neuronales Convolucionales

Las redes neuronales convolucionales (CNN, por sus siglas en inglés: Convolutional Neural Networks) son un tipo especializado de red neuronal artificial diseñada para procesar datos con una estructura similar a una cuadrícula, como las imágenes. Se han convertido en la piedra angular de la visión por computador y han revolucionado aplicaciones en campos como la clasificación de imágenes, la detección de objetos y la segmentación semántica.

Las CNN se inspiran en la organización del sistema visual de los mamíferos, particularmente en el trabajo pionero de Hubel y Wiesel en la corteza visual primaria. Estas redes explotan la estructura espacial de los datos de entrada mediante operaciones de convolución, que permiten extraer características jerárquicas con un menor número de parámetros en comparación con las redes densamente conectadas.

1.1. ¿Qué son las CNN? Diferencias entre redes neuronales densas y convolucionales.

Las redes neuronales tradicionales, también llamadas redes densas o fully connected (FC), conectan cada neurona de una capa con todas las neuronas de la siguiente. Aunque esto les permite modelar relaciones complejas, tiene un gran inconveniente cuando se aplican a imágenes: la cantidad de parámetros crece exponencialmente con el tamaño de la imagen, lo que hace que el entrenamiento sea ineficiente y propenso al sobreajuste, y se pierde la estructura espacial.

Las CNN abordan la resolución de este problema utilizando tres ideas clave:

- **Convolución y filtros**: En lugar de conectar todas las neuronas con todas las entradas, las CNN aplican filtros (“kernels”) que recorren la imagen extrayendo características locales.
- **Compartición de pesos**: Un mismo filtro se usa en toda la imagen, reduciendo drásticamente la cantidad de parámetros y mejorando la generalización.
- **Submuestreo (pooling)**: reduce la dimensionalidad de las características extraídas, haciéndolas más robustas a pequeñas variaciones y reduciendo el costo computacional.

Mientras que una red densa aprendería características sin considerar la relación espacial entre los píxeles, una CNN capta patrones como bordes, texturas y formas de manera jerárquica. Por ejemplo, en una red convolucional entrenada para el reconocimiento facial, las primeras capas pueden detectar bordes, las

intermedias estructuras como ojos y nariz, y las finales patrones específicos que identifican a una persona.

1.2. Aplicaciones prácticas de las CNN en la actualidad

Las CNN han impulsado avances en diversas aplicaciones del mundo real, destacándose en problemas donde el reconocimiento de patrones visuales es clave:

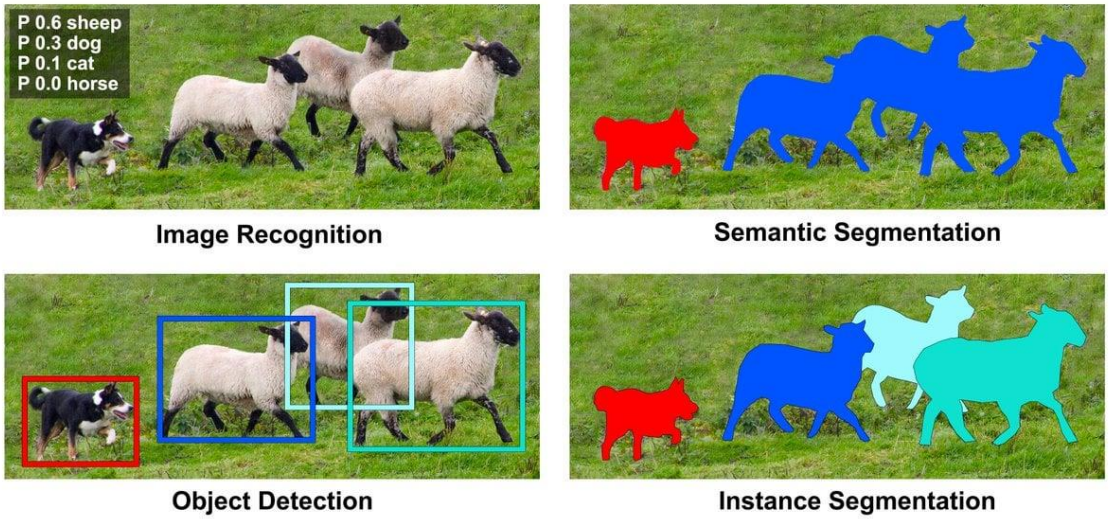
- **Clasificación de imágenes**: Modelos como AlexNet, VGG y ResNet han demostrado un rendimiento excepcional en conjuntos de datos como ImageNet, permitiendo etiquetar imágenes con alta precisión.

- **Detección de objetos**: Algoritmos como YOLO (You Only Look Once) y Faster R-CNN identifican y localizan múltiples objetos en una imagen en tiempo real, con aplicaciones en seguridad, automoción y análisis de vídeo.

- **Segmentación semántica**: Modelos como U-Net y DeepLab permiten dividir una imagen en regiones específicas, crucial para medicina (segmentación de órganos en imágenes médicas) y vehículos autónomos.

- **Reconocimiento facial**: CNNs son la base de los sistemas de autenticación biométrica en smartphones y aplicaciones de seguridad.

- **Generación de imágenes**: Redes generativas adversarias (GANs) basadas en CNN pueden crear imágenes sintéticas realistas, utilizadas en diseño, entretenimiento y restauración de imágenes históricas.

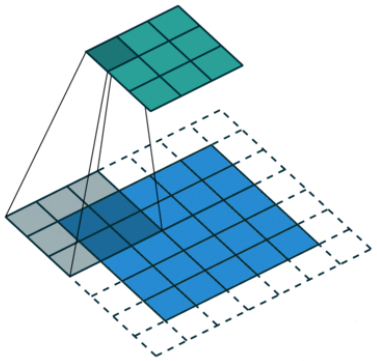


Las redes neuronales convolucionales continúan evolucionando y adaptándose a nuevos desafíos, consolidándose como una herramienta fundamental en inteligencia artificial y aprendizaje profundo.

2. Conceptos básicos de las CNN

2.1. Operación de convolución: definición y ejemplo visual.

La convolución es la operación fundamental en las CNN. Se basa en aplicar un filtro o kernel sobre una imagen de entrada para extraer características locales, patrones como bordes, texturas o formas. El kernel es una matriz de pesos que se desliza sobre la imagen, multiplicando sus valores por los píxeles correspondientes y sumando el resultado. De forma más simple, cuando cada píxel de la imagen de salida es una función matemática de los píxeles cercanos (incluido él mismo) de la imagen de entrada, el kernel es esa función. Esto genera una nueva representación conocida como **mapa de características (feature map)**. En la imagen vemos una operación de convolución con stride = 2.



Una operación de convolución básica que se aplica a una imagen bidimensional I como entrada, usando un kernel o filtro K bidimensional y que nos da como resultado una nueva imagen S sería, por ejemplo:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n = I(m,n)K(i-m, j-n)$$

Parámetros clave en la convolución

- **Tamaño del kernel**: define la dimensión del filtro (ej. 3x3, 5x5). Tamaños más pequeños capturan detalles locales, mientras que los grandes detectan estructuras más generales.

- **Stride (paso)**: es el número de píxeles que se mueve el kernel en cada iteración sobre la imagen. Un stride mayor reduce el tamaño del mapa de características.

- **Padding (relleno)**: determina si se agregan ceros alrededor de la imagen para mantener el tamaño de la salida. Tipos:
    - *Valid Padding*: sin ceros, reduce el tamaño del output.
    - *Same Padding*: agrega ceros para mantener el tamaño de la imagen.

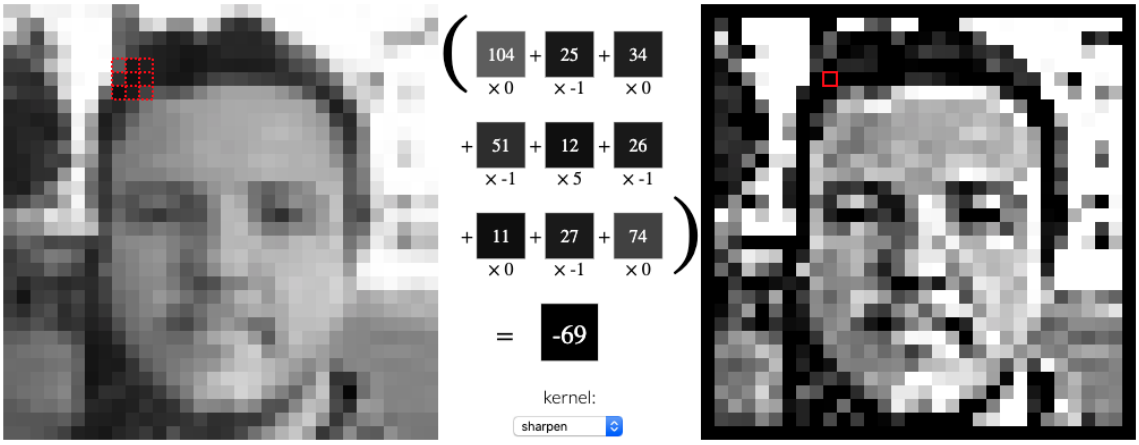
- Cantidad de filtros: determina la profundidad del mapa de características resultante. Cada filtro detecta diferentes características de la imagen, por

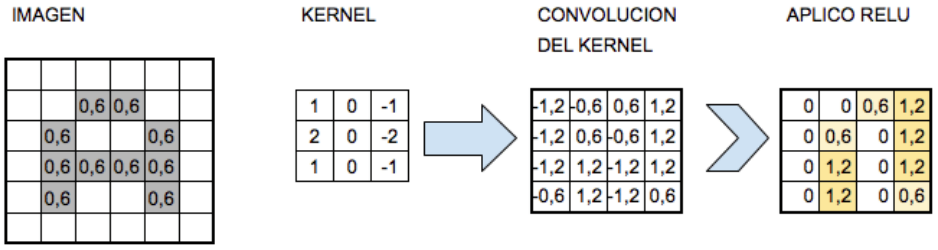
lo que un mayor número de filtros permite capturar una mayor variedad de patrones.

Mapas de características (feature maps) y evolución en las capas.

Cada capa convolucional genera un conjunto de **mapas de características** que capturan patrones específicos. En las primeras capas, los filtros detectan bordes y texturas simples. A medida que avanzamos en la red, se identifican estructuras más complejas como formas y objetos.

El proceso de convolución y pooling se repite en múltiples capas, permitiendo que la red aprenda representaciones jerárquicas de las imágenes. Este mecanismo es clave para el éxito de las CNN en tareas de visión por computador.

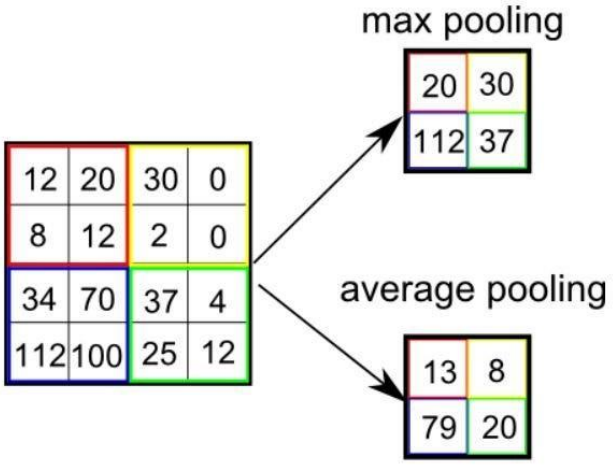




2.2. Operación de pooling: max pooling vs average pooling.

El pooling es una técnica de reducción de dimensionalidad (subsampling) que disminuye la cantidad de información, manteniendo las características más relevantes. Se divide la imagen convolucionada en recuadros de tamaño N x N, y de cada recuadro se toma un único valor. Existen principalmente dos técnicas:

- **Max Pooling**: Selecciona el valor máximo en una región de la imagen, preservando las características más intensas y eliminando ruido.
- **Average Pooling**: Calcula el promedio de los valores en una región, generando una representación más suave.



Operaciones de pooling de 2x2

La etapa de pooling ofrece una **doble ventaja**: reduce el volumen de datos al seleccionar un valor representativo de cada zona, reduciendo coste computacional (cuanto mayor es esta zona, mayor es la reducción, pero también la pérdida de detalle) y, a la vez, independiza al modelo de las variaciones en la imagen, haciéndolo tolerante a las distorsiones.

3. Arquitectura de una CNN

Las redes neuronales convolucionales (CNN) están compuestas por una serie de capas especializadas que trabajan en conjunto para extraer características y realizar tareas como la clasificación de imágenes. La arquitectura de una CNN está diseñada para aprovechar la estructura espacial de los datos de entrada mediante operaciones de convolución y reducción de dimensiones, organizando sus capas de forma jerarquizada.

3.1. Capas típicas en una CNN: capas convolucionales y capas de pooling

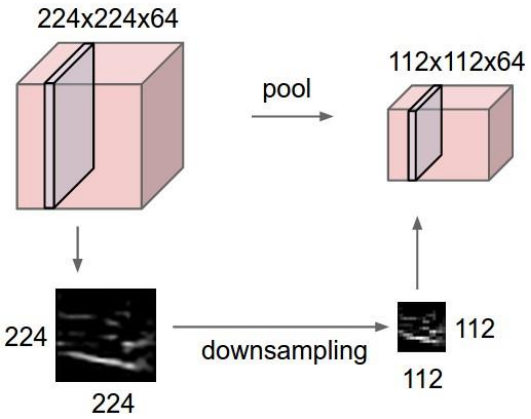
Una CNN típica está formada por varias capas interconectadas que transforman la imagen de entrada en una representación más abstracta. Las capas principales incluyen:

- **Capa de entrada:** es la 1ª capa de la red y recibe la imagen o los datos que queremos procesar. Generalmente debemos proporcionar imágenes representadas como tensores con la forma (altura, anchura, canales). Por ejemplo, una imagen en escala de grises de 28×28 píxeles se

representaría como (28, 28, 1), mientras que una imagen RGB sería (28, 28, 3) (3 canales por RGB).

- **Capas convolucionales**: son la base de las CNN. Aplican filtros (kernels) sobre la imagen para extraer características como bordes, texturas y patrones. Cada neurona en esta capa está conectada a una región local de la entrada en lugar de a todas las neuronas de la capa anterior, como en una red densa.

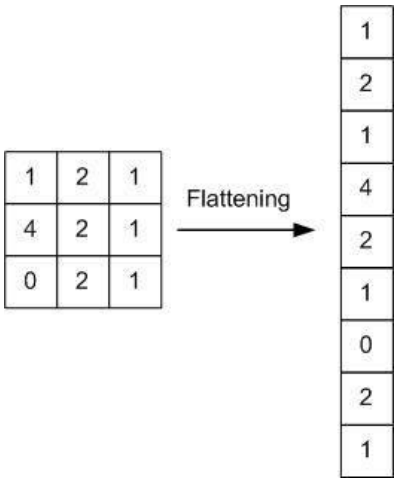
- **Capas de pooling (subsampling o downsampling)**: su objetivo es reducir la dimensionalidad de los mapas de características generados en las capas convolucionales, disminuyendo la cantidad de parámetros y mejorando la tolerancia a variaciones menores en la imagen.



- **Capas de activación**: introducen no linealidad en la red, permitiendo que ésta aprenda relaciones complejas. Algunas funciones de activación comunes son:

    - **ReLU** (Rectified Linear Unit): f(x) = max(0, x), que introduce sparsity y acelera el entrenamiento.
    - **Sigmoid**: f(x) = 1 / (1 + e^(-x)), utilizada en tareas donde se requiere una salida probabilística.
    - **Tanh**: f(x) = (e^x - e^(-x)) / (e^x + e^(-x)), similar a Sigmoid pero con valores en el rango [-1,1].

- **Capa de aplanamiento** (flattening): es un paso crucial que transforma los datos multidimensionales en un formato lineal. Tras las etapas de convolución y pooling, que extraen y resumen las características de la imagen, el flattening convierte las matrices resultantes en un vector unidimensional. Este proceso "aplana" cada matriz de características, colocando sus valores en fila, y luego concatena todos estos vectores en uno solo. El resultado es un vector largo que contiene todas las características relevantes extraídas de la imagen original, preparando los datos para su procesamiento en las capas completamente conectadas de la red neuronal, que forman el clasificador.



- **Capas densas (fully connected)**: representan la fase de clasificación, se encuentran en la parte final de la arquitectura y están completamente conectadas. Transforman la información extraída por las capas convolucionales en una representación apta para la tarea final, como la clasificación binaria o categórica.

3.2. Profundidad, stride y padding: impacto en el diseño y el rendimiento.

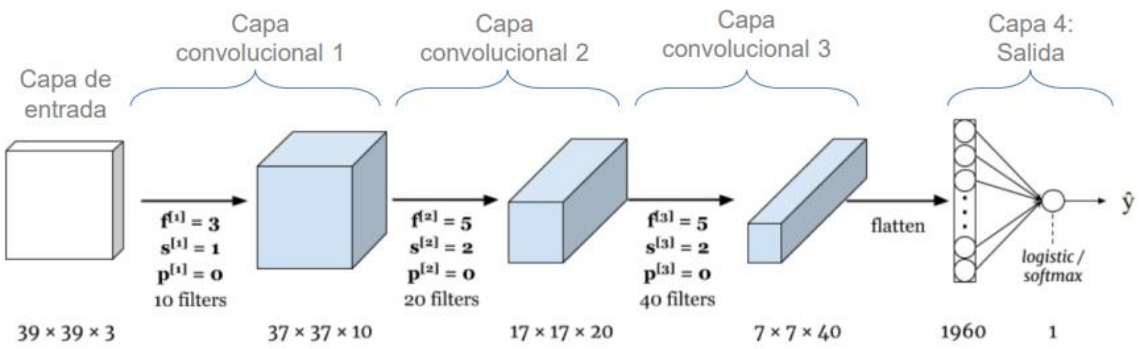
El diseño de una CNN depende de varios parámetros que afectan su rendimiento y capacidad de aprendizaje:

- **Profundidad**: se refiere a la cantidad de capas convolucionales y de pooling en la red. Redes más profundas pueden capturar características más abstractas, pero requieren más datos y poder computacional para su entrenamiento.

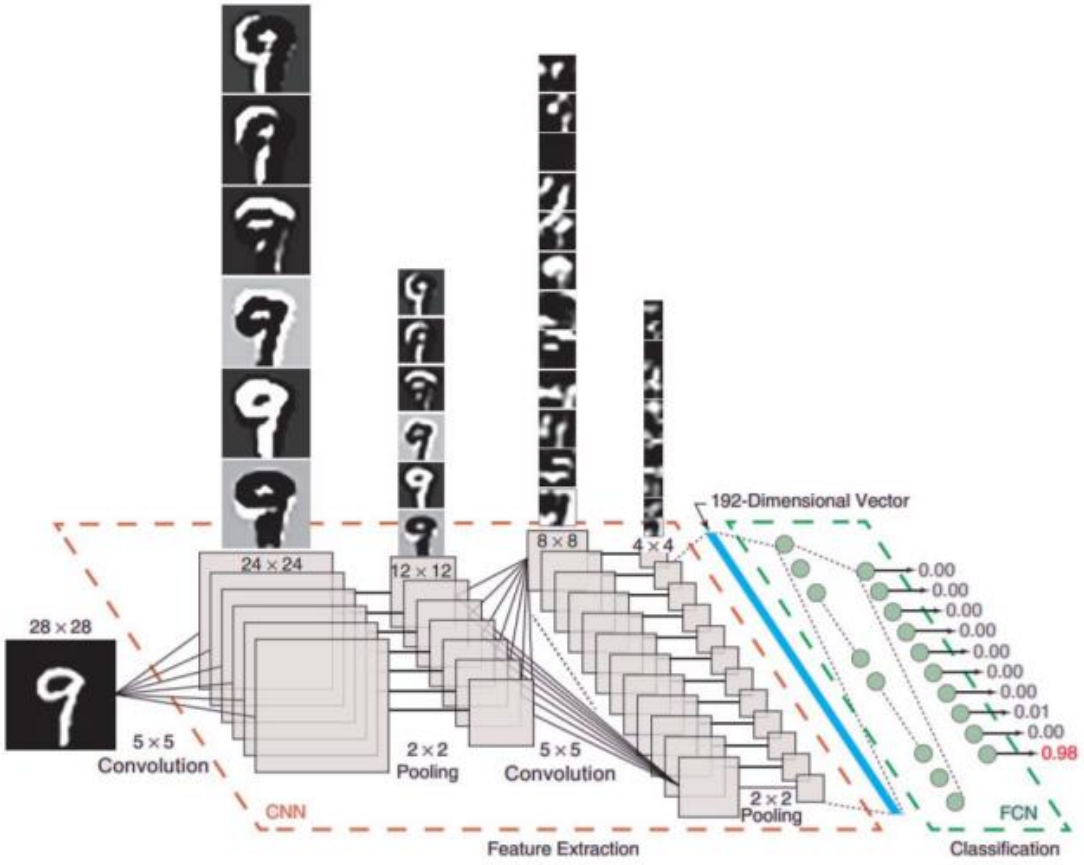
- **Stride**: un stride mayor reduce el tamaño del mapa de características, pero puede perder información importante.

- **Padding**:

Veamos un ejemplo de red con 3 capas convolucionales:



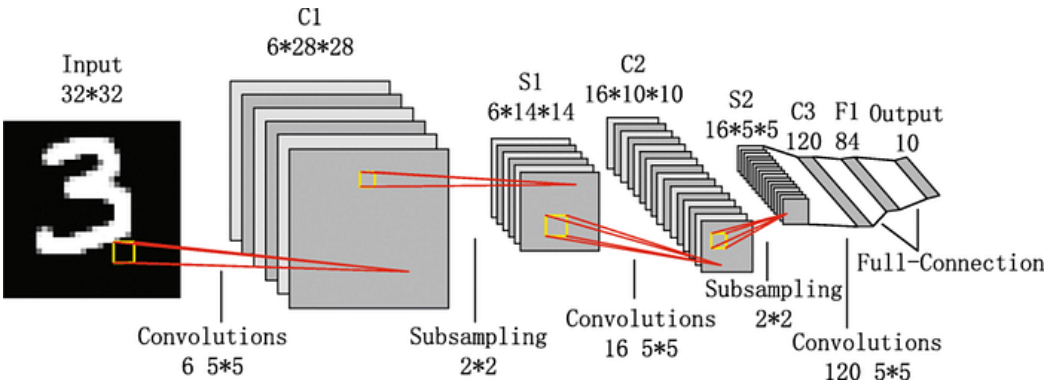
Ahora veamos los efectos de la arquitectura de la red:



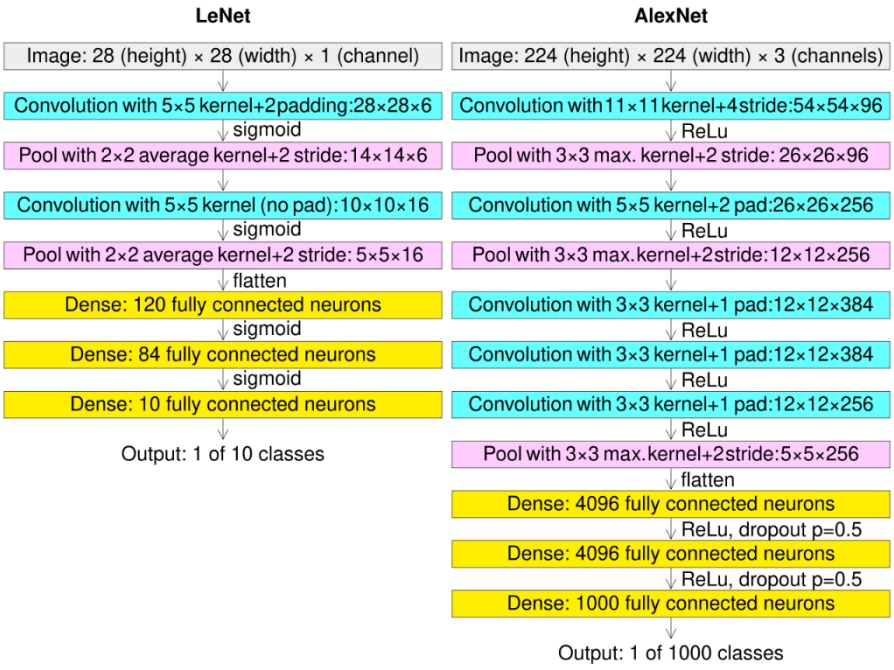
3.3. Arquitecturas populares

Existen varias arquitecturas populares de CNN, cada una con características específicas optimizadas para diferentes tareas:

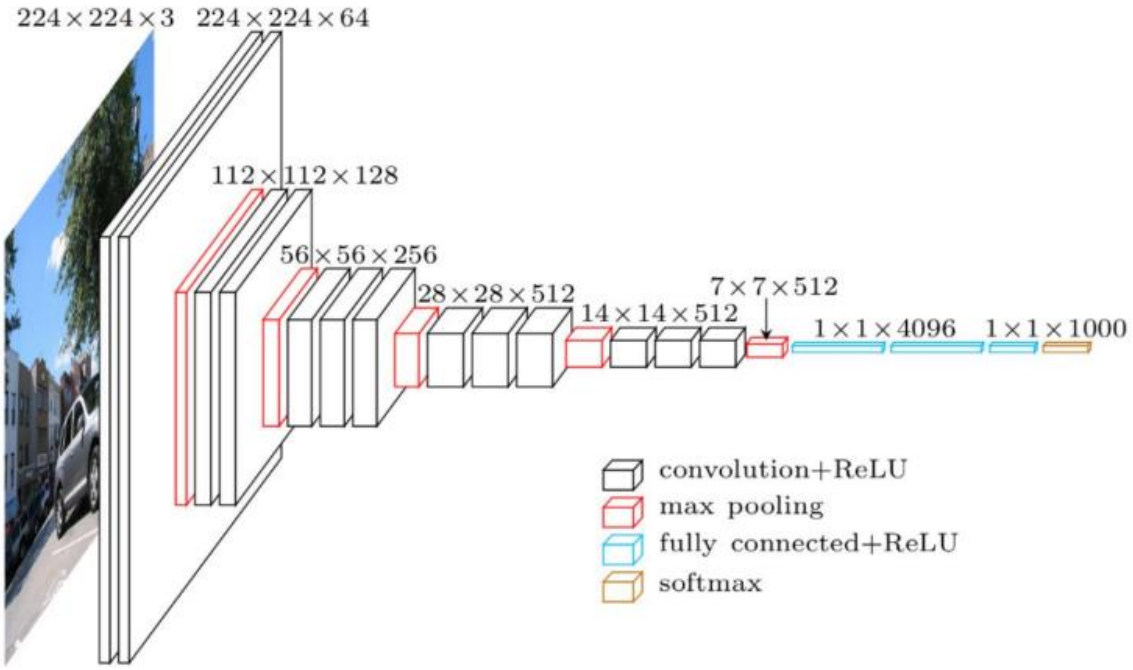
- **LeNet-5**: una de las primeras CNN, diseñada para la clasificación de dígitos escritos a mano en el dataset MNIST. Consta de 2 capas convolucionales seguidas de capas de pooling y 3 capas densas al final.



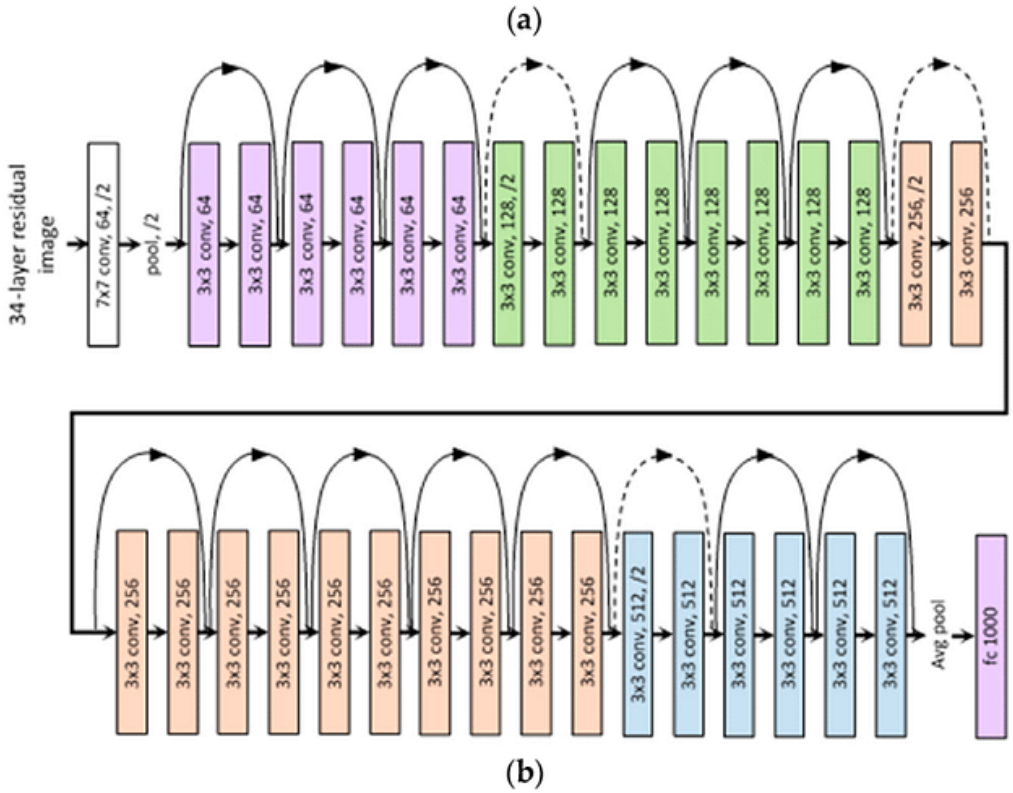
- **AlexNet**: Introducida en 2012, fue la primera en demostrar el poder de las CNN en ImageNet. Utiliza ReLU, dropout y múltiples GPUs para mejorar el rendimiento. Veamos una comparativa de LeNet-5 y AlexNet:



- **VGG16/VGG19**: Proponen una arquitectura profunda con muchas capas convolucionales pequeñas (3x3), seguidas de capas de pooling y densas. Son fáciles de interpretar, pero computacionalmente costosas.

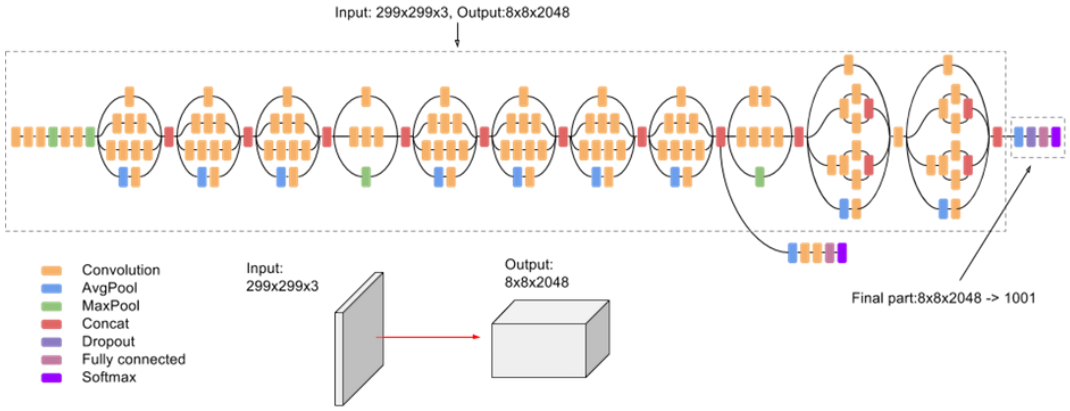


- **ResNet** (Residual Networks): Introduce conexiones residuales que permiten entrenar redes extremadamente profundas (hasta 152 capas) evitando el problema del desvanecimiento del gradiente. Veamos el ejemplo de ResNet-34:

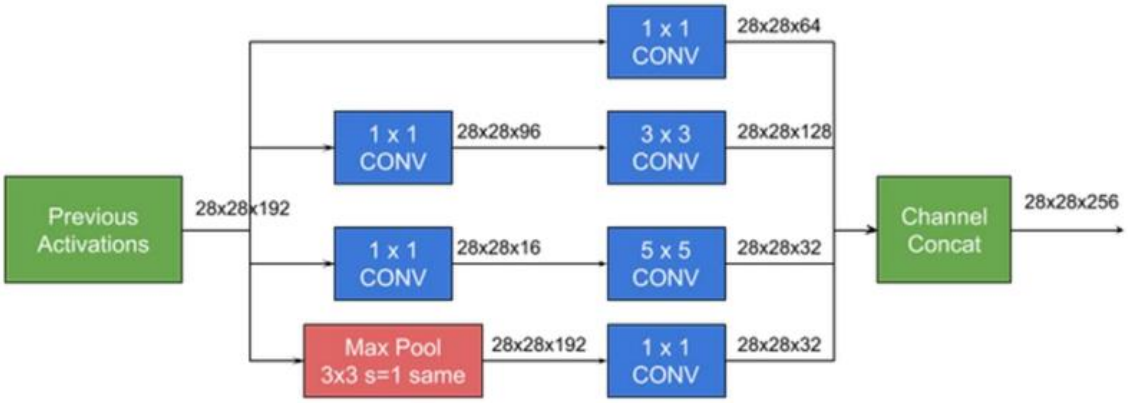


- **Inception** (GoogLeNet): usa módulos Inception que combinan convoluciones de diferentes tamaños en una misma capa, optimizando

el uso de parámetros y mejorando la eficiencia computacional. En la siguiente imagen podemos ver la arquitectura de Inception-v3.



Y en esta otra podemos ver el detalle de un “bloque” en Inception:



Cada una de estas arquitecturas ha representado un avance significativo en la visión por computador y sigue siendo la base para modelos modernos utilizados en aplicaciones del mundo real.

4. Implementación de CNN con Keras/TensorFlow

En este apartado se presenta una guía práctica para implementar una red neuronal convolucional (CNN) utilizando Keras. Se abordarán desde la configuración del entorno hasta el entrenamiento y evaluación del modelo sobre un conjunto de datos estándar.

4.1. Preparación del entorno de trabajo

Antes de comenzar con la implementación, es necesario asegurarse de que Keras y TensorFlow estén correctamente instalados. Se recomienda utilizar un entorno virtual para mantener organizadas las dependencias. También se pueden utilizar entornos como Google Colab, que ya incluyen estas bibliotecas preinstaladas.

4.1. Carga y procesamiento de las imágenes de entrada

Para poder procesar las imágenes de entrada como datos de entrenamiento es ampliamente conocido y utilizado el método `image_dataset_from_directory` del paquete keras.utils. Este método acepta, entre otros, los siguientes parámetros:

- La carpeta de donde tomar las imágenes (en una ruta absoluta o relativa).
- **`labels`**: indica qué etiquetas se deben tomar para clasificar las imágenes de entrada. Un valor muy habitual es inferred, para que se deduzcan de las subcarpetas existentes dentro de la carpeta principal indicada en el parámetro anterior.
- **`label_mode`**: indica el modo de etiquetado que se aplicará. Algunos valores habituales son binary (para clasificadores binarios medidos con entropía cruzada binaria), categorical (para clasificadores categóricos medidos con entropía cruzada categórica), o int (para otras clasificaciones medidas con otras métricas).
- **`class_names`**: sólo en el caso de que el parámetro labels sea inferred, podemos indicar aquí en forma de lista los valores de las categorías que queremos identificar (y que deben coincidir con los nombres de las subcarpetas existentes).
- **`color_mode`**: puede tomar los valores grayscale, rgb (valor por defecto) o rgba
- **`batch_size`**: tamaño de los paquetes de imágenes que se formarán para pasarlos al modelo. Por defecto es 32
- **`image_size`**: tupla indicando el tamaño de las imágenes de entrada (altura y anchura, en ese orden)

Aquí vemos un ejemplo donde indicamos que las imágenes de entrenamiento están en una subcarpeta train en la carpeta actual. Las queremos proporcionar con un tamaño de 224x224, y dentro de esa carpeta hay dos subcarpetas con las categorías perros y gatos (clasificación binaria). La carpeta test tiene una distribución similar para el test:

```
from keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    '.\\train',
    image_size=(224, 224),
    labels='inferred',
    label_mode='binary',
    class_names=['perros', 'gatos']
)

test_dataset = image_dataset_from_directory(
    '.\\test',
    image_size=(224, 224),
    labels='inferred',
    label_mode='binary',
    class_names=['perros', 'gatos']
)
```

El método devuelve un objeto de tipo tf.data.Dataset, el cual contiene tensores de tipo tf.Tensor.

Además, resulta conveniente **normalizar** los valores de la imagen en una escala de 0 a 1; para ello tenemos disponible la clase Rescaling del paquete keras.layers. Por otra parte, con el objetivo de prevenir el overfitting y mejorar las métricas, suele ser importante aplicar la técnica del ***data augmentation***, la cual permite generar múltiples datos a partir de uno en particular. En este caso, a partir de una imagen podemos generar otra similares aplicando algunas transformaciones: voltear, rotar, ampliar... Para esto tenemos disponibles las clases RandomFlip, RandomRotation, RandomZoom y/o RandomTranslation, entre otras. Veamos un ejemplo combinado aplicando ambas cosas al conjunto de entrenamiento, y sólo el escalado al de test:

```
from keras.layers import Rescaling, RandomFlip, RandomRotation,
RandomZoom, RandomTranslation

normalization = Rescaling(1./255)

data_augmentation = Sequential([
    RandomFlip("horizontal"),       # Volteo horizontal
    RandomRotation(0.2),            # Rotaciones 20% de 360º
    RandomZoom(0.1),                # Zoom hasta 10%
    RandomTranslation(0.1, 0.1)     # Desplaz. vertical y horizontal
])
```

```
train_dataset = train_dataset.map(lambda x, y:
(data_augmentation(normalization(x)), y))
test_dataset = test_dataset.map(lambda x, y: (normalization(x), y))
```

4.2. Construcción y compilación de una CNN básica con Keras

Además de capas ya conocidas, como son “Input” y “Dense”, vamos a añadir capas específicas para implementar las arquitecturas convolucionales, como son:

- **Conv2D**: capa convolucional.
- **MaxPooling2D**: capa para realizar el pooling.
- **Flatten**: aplanado previo a la fase de capas Dense.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense

model = keras.Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

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

Este modelo consiste en 2 capas convolucionales con ReLU como función de activación, seguidas de capas de pooling y una capa completamente conectada (la tradicional capa oculta de neuronas vista en la unidad anterior) para la clasificación. Algunos aspectos relevantes a comentar:

- En las capas Conv2D especificamos el número de filtros distintos de convolución que se aplicarán.
- En las capas MaxPooling2D definimos el tamaño de la matriz de *pooling* (2x2, en este caso).
- Tras el aplanado (Flatten), pasamos al clasificador que se trata de una red neuronal tradicional mediante las capas Dense. La primera que añadamos tomará las conexiones de la capa *Flatten*. Dependiendo del tamaño final de esa capa deberemos darle un tamaño intermedio adecuado.

El número de filtros por capa es totalmente experimental. Podemos probar hasta que encontremos resultados satisfactorios. Es habitual definir pocos filtros en las primeras capas (donde el tamaño de las imágenes aún puede ser grande) y aumentarlo en capas posteriores, donde ya se han reducido con operaciones de *pooling*.

En la fase del clasificador, podemos añadir tantas capas intermedias como queramos, y finalizar con una capa de salida que tendrá tantas neuronas como elementos queramos clasificar:

- Si sólo queremos distinguir entre 2 estados, podemos colocar una neurona de salida con función de activación sigmoide. La función de coste, si es un problema de clasificación, podría ser la entropía cruzada binaria (binary_crossentropy).
- Si queremos distinguir entre más de 2 categorías, añadiremos una neurona por cada categoría, con activación softmax. En este caso, la función de coste al compilar la red será la entropía cruzada categórica (categorical_crossentropy). En el ejemplo la capa de salida tiene 10 neuronas porque querríamos clasificar una imagen entre 10 posibles estados (reconocer dígitos del 0 al 9, por ejemplo).

4.3. Entrenamiento, evaluación y visualización de resultados.

Una vez construida la arquitectura de la CNN, el siguiente paso es entrenarla utilizando los datos previamente cargados y procesados. En este apartado, se explicará cómo llevar a cabo el entrenamiento del modelo, evaluar su rendimiento y visualizar los resultados obtenidos.

El entrenamiento de la CNN se realiza con el método fit(), donde se le pasan los datos de entrenamiento, el número de épocas y, opcionalmente, los datos de validación. Durante este proceso, el modelo ajustará sus pesos a través de la retropropagación y optimización, buscando minimizar la función de pérdida definida.

```
history = model.fit(
    train_dataset,
    validation_data=test_dataset,
    epochs=20,
    batch_size=32
)
```

Tras el entrenamiento, es crucial evaluar el modelo para medir su rendimiento sobre el conjunto de prueba. Esto nos permite verificar si el modelo ha aprendido correctamente o si ha caído en sobreajuste (overfitting). También podemos

analizar cómo ha progresado el modelo durante el entrenamiento, mostrando de forma gráfica las métricas almacenadas en history.

```
# Evaluación del modelo en el conjunto de test
loss, accuracy = model.evaluate(test_dataset)

print(f"Pérdida en test: {loss:.4f}")
print(f"Precisión en test: {accuracy:.4f}")

# Visualización de la evolución del entrenamiento
import matplotlib.pyplot as plt

# Extraer los valores del historial
epochs_range = range(epochs)
train_loss = history.history['loss']
val_loss = history.history['val_loss']
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, train_loss, label='Pérdida de entrenamiento')
plt.plot(epochs_range, val_loss, label='Pérdida de validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.title('Evolución de la Pérdida')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_acc, label='Precisión de entrenamiento')
plt.plot(epochs_range, val_acc, label='Precisión de validación')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.title('Evolución de la Precisión')

plt.show()
```

4.4. Uso del modelo en predicciones

Una vez que el modelo ha sido entrenado, se puede utilizar para realizar predicciones sobre nuevas imágenes. Para ello, es necesario cargar la imagen, procesarla de la misma manera que las imágenes de entrenamiento y pasarla al modelo.

```
import numpy as np
from tensorflow.keras.preprocessing import image

# Cargar la imagen y preprocesarla
img = image.load_img("imagen_nueva.jpg", target_size=(224, 224))
img_array = image.img_to_array(img)
```

```
img_array = np.expand_dims(img_array, axis=0)  # Añadir a batch
img_array /= 255.0  # Normalización

# Realizar la predicción
prediccion = model.predict(img_array)
clase_predicha = np.argmax(prediccion)
print(f"La imagen pertenece a la clase: {clase_predicha}")
```

5. Mejora del rendimiento, técnicas avanzadas y optimización

Las redes neuronales convolucionales pueden ser modelos muy potentes, pero su rendimiento depende de diversos factores como la arquitectura, el conjunto de datos y la forma en que se entrenan. Para mejorar su eficacia y evitar problemas como el overfitting o la convergencia lenta, se pueden aplicar varias técnicas de optimización. En este apartado, exploraremos estrategias clave para mejorar la generalización del modelo y acelerar el proceso de entrenamiento.

5.1. Tratamiento del overfitting mediante regularización

El overfitting es muy habitual en redes convolucionales, recordemos que éste ocurre cuando el modelo se ajusta demasiado a los datos de entrenamiento y no generaliza bien a datos nuevos. Algunas técnicas para mitigar esto son:

- **Dropout**: desactiva aleatoriamente un porcentaje de neuronas en cada iteración para evitar la dependencia excesiva en ciertas conexiones.
- **Regularización L1/L2**: agrega penalizaciones a los pesos para evitar valores excesivos

Veamos la sintaxis de ambas técnicas en el siguiente código:

```
from tensorflow.keras.layers import Dropout
from tensorflow.keras.regularizers import l2

model.add(Dropout(0.5))

model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.01)))
```

También podemos emplear **BatchNormalization**.

Batch Normalization (BN) es una técnica que **normaliza las activaciones** de cada capa antes de pasarlas a la siguiente, ayudando a estabilizar y acelerar el entrenamiento. Se encarga de ajustar la media y la varianza de las activaciones, lo que reduce el problema del *vanishing gradient* y permite usar tasas de aprendizaje más altas.

Ventajas de Batch Normalization:

- Reduce la **sensibilidad** del modelo a la inicialización de pesos.
- Permite entrenar con **learning rates más altos**.
- Reduce la dependencia de técnicas como Dropout en algunas arquitecturas.
- Actúa como una forma de regularización, reduciendo el overfitting.

```
from tensorflow.keras.layers import BatchNormalization, Conv2D

model.add(Conv2D(64, (3,3), activation='relu'))
model.add(BatchNormalization())
```

No siempre es necesario usar **Batch Normalization (BN)**, aunque es una técnica muy útil en muchos casos. Su uso depende del tipo de modelo, la arquitectura y los datos con los que trabajamos.

Cuándo usar Batch Normalization:

1. **Modelos profundos**. En redes con muchas capas, ayuda a estabilizar el entrenamiento y a acelerar la convergencia. Reduce el problema del *vanishing/exploding gradient* en redes muy profundas.
2. **Cuando el entrenamiento es inestable.** Si la pérdida oscila demasiado o el modelo tarda en converger, BN puede ayudar a regularizar y estabilizar el aprendizaje. Permite usar tasas de aprendizaje más altas sin que el modelo diverja.
3. **Si el dataset tiene variabilidad en las entradas**. En imágenes con iluminaciones o contrastes distintos, ayuda a reducir la sensibilidad del modelo a estas diferencias. Normaliza las activaciones para que las distribuciones sean más homogéneas.
4. **En capas convolucionales**. BN funciona especialmente bien en **redes convolucionales profundas** al mejorar la propagación de los gradientes. Normalizar activaciones después de Conv2D mejora la estabilidad.
5. **Si no queremos depender tanto de Dropout**. BN actúa como una forma de regularización, por lo que en algunas arquitecturas permite reducir o incluso prescindir de Dropout.

Cuándo NO usar Batch Normalization

1. **En redes pequeñas o poco profundas**. Si el modelo ya converge rápido y sin problemas, añadir BN puede ser innecesario y añadir sobrecarga computacional.
2. **Si usamos capas de normalización en la entrada (Layer Normalization, Instance Normalization, etc.)** Por ejemplo, en redes que ya usan **Layer Normalization (LN)** en lugar de BN, como en Transformers.
3. **En modelos con batch size muy pequeño**. BN calcula estadísticas por batch (media y varianza), por lo que en entrenamientos con batch sizes muy pequeños (<8), las estimaciones pueden ser inestables. En estos casos, es mejor usar **Group Normalization (GN)** o **Layer Normalization (LN)**.
4. **En redes recurrentes (RNN, LSTM, GRU)**. BN no se suele usar en redes recurrentes porque las estadísticas de normalización cambian en cada

timestep y afectan negativamente al entrenamiento. Alternativas: Layer Normalization o técnicas específicas para RNNs.

5.2. Data augmentation: rotación, escalado, traslación, etc.

Uso de ImageDataGenerator o keras.preprocessing.
El incremento artificial del número de imágenes es importante, pero en algunos casos hay que tener cuidado con las técnicas aplicadas porque pueden ser contraproducentes (ejemplo: confundir un 9 con un 6).

5.3. Ajuste de la tasa de aprendizaje

La tasa de aprendizaje (learning_rate) controla el tamaño de los pasos en la actualización de los pesos. Un valor demasiado alto puede hacer que el modelo no converja, mientras que un valor demasiado bajo puede hacer que el entrenamiento sea lento.

Se pueden utilizar estrategias como:

- Tasa de aprendizaje constante: Se fija un valor y se mantiene durante todo el entrenamiento.
- Decaimiento de la tasa de aprendizaje: Se reduce progresivamente a medida que avanza el entrenamiento para evitar oscilaciones.
- Scheduler de tasa de aprendizaje: Keras proporciona el callback ReduceLROnPlateau, que ajusta dinámicamente la tasa cuando la métrica de validación no mejora.

```
from tensorflow.keras.callbacks import ReduceLROnPlateau

lrScheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3, verbose=1)
```

5.4. Optimización del rendimiento: Hiperparámetros clave: learning rate, optimizadores, batch size.

5.4. Uso de callbacks: EarlyStopping

Este callback, que en español se traduce como “parada anticipada”, ya se ha presentado en la unidad anterior. En las CNN es más necesario todavía, así que vamos a recordar su estructura. La parada anticipada resuelve el problema de cuántas épocas debemos entregar nuestro modelo antes de que sus métricas empeoren, especialmente, cuando se produce overfitting. Lo que hace es detener el entrenamiento si la métrica no mejora, a partir de la configuración que

hayamos realizado; para ello se usa la clase de callback de keras llamado EarlyStopping(). La clase contiene entre otros los siguientes parámetros:

- `monitor`: La métrica a monitorizar. Por defecto monitoriza `val_loss`.
- `min_delta`: El mínimo valor que consideramos como una mejora. Si mejora menos que este valor lo consideraremos como que no ha mejorado. Por defecto si valor es 0.
- `patience`: Cuántas épocas puede estar sin mejorar antes de que paremos el entrenamiento. Por defecto vale 0.
- `mode`: Indica el tipo de métrica que es. Es decir si la métrica es mejor cuanto mayor valor tiene (`max`) o la métrica es mejor cuanto menor valor tiene (`min`). Por defecto es valor es `auto` y keras sabrá el tipo de la métrica por el nombre que tiene.
- `restore_best_weights`: Si es True, restaurará los pesos del modelo a la mejor versión del entrenamiento, es decir, la versión que tuvo el mejor valor en la métrica monitorizada.

```
from tensorflow.keras.callbacks import EarlyStopping

earlystopping = EarlyStopping(
    monitor='val_loss',  # Métrica a controlar
    min_delta=0.001,     # Mejora mínima significativa
    patience=10,         # Esperar 10 épocas sin mejora antes de parar
    mode='min',          # Monitorizar si la pérdida disminuye
    restore_best_weights=True # Restaurar los mejores pesos
)
```

6. Estructuras de datos en TensorFlow

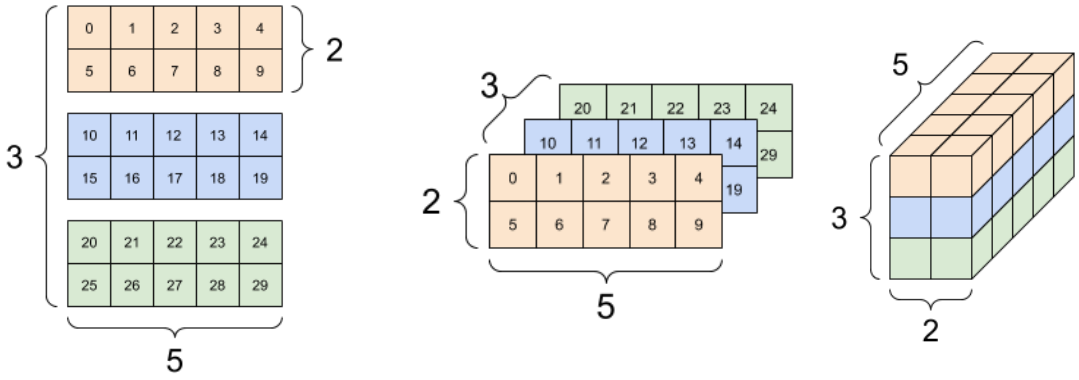
En TensorFlow, el procesamiento de datos se basa en unas estructuras fundamentales como son los tensores y su agrupación en los `tf.data.Dataset`. De hecho, recibe su nombre precisamente de ahí. Comprender estas estructuras es esencial para trabajar con modelos de deep learning de manera eficiente.

6.1. Tensores y estructuras tensoriales en TF

Los **tensores** son la estructura de datos fundamental en redes neuronales convolucionales (CNNs) y en general en el deep learning; suelen entenderse como matrices multidimensionales con tipo de datos uniforme (como en NumPy). Son generalizaciones de los vectores y matrices a dimensiones superiores. En TensorFlow y PyTorch, los tensores son las unidades de almacenamiento y procesamiento de datos básicas dentro de la red. Veamos tipos concretos:

- **Escalar (0D)**: un solo número, por ejemplo, `5.0`.
- **Vector (1D)**: una lista o array de números, por ejemplo, `[1, 2, 3]`.
- **Matriz (2D)**: una tabla de números.
- **Tensor (3D o más)**: una estructura multidimensional, por ejemplo, un conjunto de imágenes en formato `(batch, height, width, channels)`.

Hay muchas formas de visualizar un tensor con más de dos ejes. Veamos un ejemplo para representar un tensor de 3 ejes, de la forma: [3, 2, 5]



En el contexto de imágenes, un tensor suele tener la forma `(batch_size, altura, anchura, canales)`, donde:

- `batch_size`: Número de imágenes procesadas simultáneamente.
- `altura` y `anchura`: Dimensiones de la imagen.
- `canales`: Número de canales (1 para imágenes en escala de grises, 3 para RGB).

En TensorFlow, los tensores son **inmutables** y optimizados para ejecutarse en CPU y también en GPU. Se pueden crear de diferentes formas y tienen una relación muy estrecha con los arrays de NumPy.

Un `tf.Tensor` es similar a un `numpy.ndarray`, pero con la ventaja de estar optimizado para su ejecución en entornos de hardware acelerado como GPUs y TPUs. Además, los tensores pueden participar en operaciones diferenciales, lo que los hace fundamentales para el entrenamiento de modelos de deep learning.

Se pueden crear de diferentes formas:

```
import tensorflow as tf
import numpy as np

# Crear tensores a partir de listas/arrays de NumPy
tensor_1d = tf.constant([1.0, 2.0, 3.0])
tensor_2d = tf.constant([[1.0, 2.0], [3.0, 4.0]])

# Convertir un array de NumPy a tensor
tensor_from_numpy = tf.convert_to_tensor(np.array([[5, 6], [7, 8]]),
dtype=tf.float32)

# Recuperar un numpy.ndarray desde un tf.Tensor
numpy_array = tensor_from_numpy.numpy()
```

6.2. Evolución hacia Datasets

Trabajar con grandes volúmenes de datos, como por ejemplo para entrenar una CNN, requiere estructuras eficientes. TensorFlow introduce **tf.data.Dataset**, que permite manipular datos de manera escalable y optimizada, agrupando muchas imágenes en formato de tensores.

Las principales formas de crear un **Dataset** son:

1. Desde una lista de tensores con **from_tensor_slices**
2. Desde archivos o directorios con **image_dataset_from_directory**

Veamos el primer caso:

```
import tensorflow as tf

datos = tf.constant([[1, 2], [3, 4], [5, 6]])
etiquetas = tf.constant([0, 1, 0])
dataset = tf.data.Dataset.from_tensor_slices((datos, etiquetas))

for elemento in dataset.take(1):
    print(elemento)
```

```
dataset = tf.data.Dataset.from_tensor_slices((
    tf.random.uniform((10, 128, 128, 3)),  # 10 imágenes
    tf.random.uniform((10, 1))  # 10 etiquetas
))

# Inspeccionar la estructura
dataset_spec = tf.nest.map_structure(lambda x:
tf.TensorSpec.from_tensor(x), next(iter(dataset)))
print(dataset_spec)
```

Salida:

```
(<tf.Tensor: shape=(2,), dtype=int32, numpy=array([1, 2], dtype=int32)>,
 <tf.Tensor: shape=(), dtype=int32, numpy=0>)
```

Ahora el segundo caso.

Para cargar imágenes desde directorios, se usa `image_dataset_from_directory`, que devuelve un `tf.data.Dataset` con tuplas `(x, y)`, donde cada tupla es un batch con:

- `x`: Un tensor de imágenes con la forma `(batch_size, image_size[0], image_size[1], canales)`.
- `y`: Un tensor de etiquetas de clase, con formato dependiente del `label_mode`.

```
dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "dataset/train",
    image_size=(224, 224),
    batch_size=32
)

for images, labels in dataset.take(1):
    print(type(images))  # <class 'tensorflow.Tensor'>
    print(images.shape)  # (32, 224, 224, 3)
```

El tipo `tf.data.Dataset` es útil porque maneja la carga de datos en paralelo y permite preprocesar imágenes de manera eficiente antes de alimentar el modelo.

Esta estructura está formada por **batches de tensores**, donde:

- `images` es un tensor con las imágenes normalizadas.
- `labels` es un tensor con las etiquetas correspondientes.

Conversión a NumPy

Si se necesita trabajar con NumPy, se pueden convertir los datos:

```
import numpy as np

data_numpy = np.array([x.numpy() for x, y in dataset])
labels_numpy = np.array([y.numpy() for x, y in dataset])
```

También se puede usar `tfds.as_numpy()`:

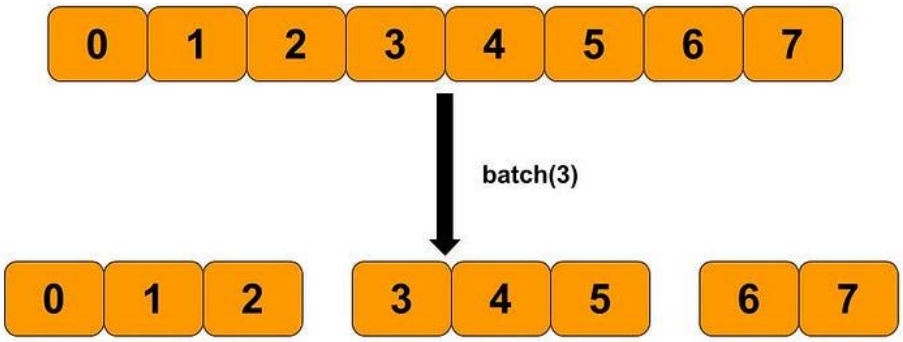
```
import tensorflow_datasets as tfds

data_numpy = list(tfds.as_numpy(dataset))
```

6.3. Métodos importantes de tf.data.Dataset

a) `batch(batch_size)`: agrupar en lotes. Divide los datos en lotes de tamaño `batch_size`, útil para el entrenamiento en modelos de aprendizaje profundo.

```
dataset_batched = dataset.batch(2)
for batch in dataset_batched:
    print(batch)
```



b) `shuffle(buffer_size)`: mezclar los datos. Reorganiza los datos de manera aleatoria dentro de un buffer de tamaño `buffer_size` para evitar sesgos en el entrenamiento.

```
dataset_shuffled = dataset.shuffle(buffer_size=3)
for elemento in dataset_shuffled:
    print(elemento)
```

c) `map(funcion)`: transformar los datos. Aplica una función a cada elemento del dataset. Es útil para normalizar datos o aplicar cualquier preprocesamiento.

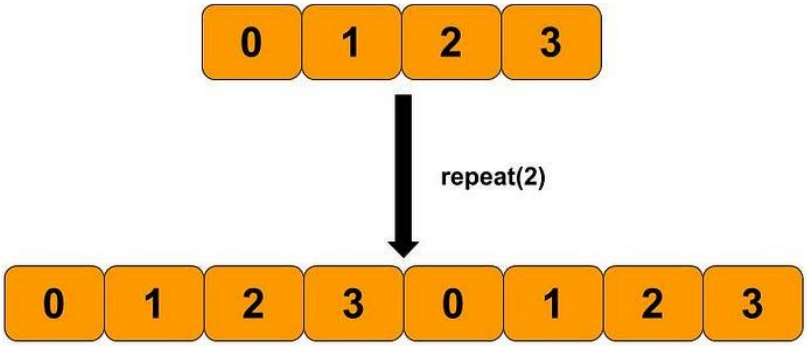
```
def escalar_x_y(x, y):
    return x / 10.0, y

dataset_scaled = dataset.map(escalar_x_y)
```

```
for elemento in dataset_scaled:
      print(elemento)
```

d) `repeat(count)`: repetir el dataset. Duplica los datos `count` veces. Si se omite el parámetro, se repite indefinidamente.

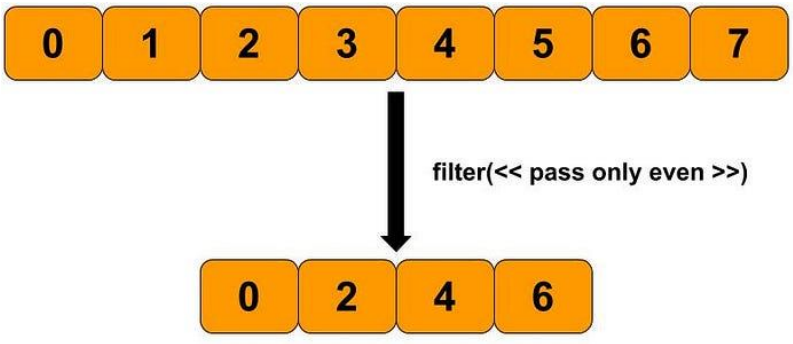
```
dataset_repeated = dataset.repeat(2)
for elemento in dataset_repeated:
      print(elemento)
```



e) `filter(predicate)`: filtrar los datos. Retiene solo los elementos del dataset que cumplen con una condición dada.

```
def filtrar_pares(x, y):
    return tf.reduce_any(x % 2 == 0)

dataset_filtrado = dataset.filter(filtrar_pares)
for elemento in dataset_filtrado:
    print(elemento)
```



f) `concatenate(dataset)`: concatenar datasets. Une dos datasets en una sola secuencia, útil cuando se combinan conjuntos de datos de diferentes fuentes.

```
dataset_extra = tf.data.Dataset.from_tensor_slices(([[7, 8]],
[1]))
dataset_concatenado = dataset.concatenate(dataset_extra)
for elemento in dataset_concatenado:
    print(elemento)
```

g) `zip(dataset)`: combinar múltiples datasets. Empareja elementos de múltiples datasets en tuplas, útil para combinar características distintas en un solo dataset.

```
dataset_1 = tf.data.Dataset.from_tensor_slices([1, 2, 3])
dataset_2 = tf.data.Dataset.from_tensor_slices(['a', 'b', 'c'])
dataset_zip = tf.data.Dataset.zip((dataset_1, dataset_2))

for elemento in dataset_zip:
    print(elemento)
```

h) `take(count`): tomar un número limitado de elementos. Permite extraer un número determinado de elementos del dataset, útil para depuración o pruebas rápidas.

7. Técnicas de tratamiento de imagen

El preprocesamiento de imágenes es una etapa fundamental en el desarrollo de redes neuronales convolucionales. Su objetivo es optimizar la calidad de las imágenes para mejorar el rendimiento del modelo. Existen diversas librerías, como son Pillow/PIL y OpenCV (cv2); en este apartado trabajaremos con OpenCV, ya que es una de las librerías más utilizadas para estas tareas debido a su eficiencia y facilidad de uso. El preprocesamiento de imágenes incluye una serie de transformaciones destinadas a normalizar y estandarizar los datos de entrada, mejorando la eficiencia y precisión del modelo. Algunas de las técnicas más comunes incluyen:

- Cambio de tamaño (Resizing): Ajustar todas las imágenes a un tamaño uniforme para que sean compatibles con la arquitectura del modelo.
- Conversión de escala de grises: Para modelos que no requieren información de color.
- Normalización: Escalar los valores de píxeles al rango [0, 1] o [-1, 1] para mejorar la estabilidad del entrenamiento.
- Conversión a tensores: Transformar imágenes en arrays numéricos compatibles con los frameworks de IA.

Importancia del Preprocesamiento en CNN

Las imágenes pueden venir en distintos formatos, resoluciones y condiciones de iluminación, lo que puede afectar la capacidad de generalización del modelo. Algunas razones por las cuales es importante el preprocesamiento son:

- Normalización de datos.
- Reducción del ruido.
- Corrección de iluminación y contraste.
- Asegurar un tamaño uniforme de las imágenes.
- Resaltar características importantes.

Operaciones de Preprocesamiento con OpenCV

Carga y Visualización de Imágenes

```
import cv2
import matplotlib.pyplot as plt

# Cargar la imagen en escala de grises
imagen = cv2.imread('imagen.jpg', cv2.IMREAD_GRAYSCALE)

# Mostrar imagen con OpenCV (requiere ventana emergente)
cv2.imshow('Imagen', imagen)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Mostrar con Matplotlib (evita problemas con colores)
plt.imshow(imagen, cmap='gray')
plt.axis('off')
plt.show()
```

Redimensionamiento de Imágenes

Las CNN requieren un tamaño de imagen fijo. OpenCV permite redimensionar imágenes con cv2.resize(). Lo habitual es necesitarlo cuando las imágenes provienen de distintas fuentes con tamaños variables, o cuando se usa un modelo preentrenado que requiere dimensiones específicas.

```
dim = (224, 224)  #Tamaño típico para modelos como VGG16 o ResNet
imagen_redimensionada = cv2.resize(imagen, dim,
interpolation=cv2.INTER_AREA)
```

Normalización e Igualación de Histograma

Normalizar los valores de los píxeles mejora la estabilidad numérica del modelo. También podemos mejorar el contraste mediante ecualización. Debe usarse cuando hay grandes variaciones en la iluminación de las imágenes y para mejorar el contraste en imágenes con tonos similares.

```
imagen_normalizada = imagen_redimensionada / 255.0

imagen_ecualizada = cv2.equalizeHist(imagen)
```

Eliminación de Ruido con Filtros

Para reducir ruido en imágenes ruidosas, se pueden aplicar filtros:

```
# Filtro Gaussiano
imagen_suavizada = cv2.GaussianBlur(imagen, (5,5), 0)
# Filtro de mediana
imagen_median = cv2.medianBlur(imagen, 5)
```

Detección de Bordes con Canny

Los bordes resaltan características importantes para una CNN.

```
bordes = cv2.Canny(imagen, 100, 200)
plt.imshow(bordes, cmap='gray')
plt.axis('off')
plt.show()
```

Transformaciones Geométricas: Rotación y Traslación

```
# Rotación de 45 grados alrededor del centro
(h, w) = imagen.shape[:2]
M = cv2.getRotationMatrix2D((w//2, h//2), 45, 1.0)
imagen_rotada = cv2.warpAffine(imagen, M, (w, h))
```

Integración con TensorFlow/Keras

Para usar imágenes preprocesadas en Keras, se pueden convertir en tensores:

import tensorflow as tf

imagen_tensor = tf.convert_to_tensor(imagen_normalizada, dtype=tf.float32)

8. Uso de modelos preentrenados (Transfer Learning)

El aprendizaje por transferencia (Transfer Learning) es una técnica en el ámbito del aprendizaje profundo donde un modelo previamente entrenado en un conjunto de datos grande y genérico se reutiliza para un problema específico con un conjunto de datos más pequeño. En lugar de entrenar una red neuronal convolucional (CNN) desde cero, se aprovechan las características aprendidas por un modelo ya existente y se adaptan al nuevo problema.

El concepto detrás del Transfer Learning se basa en la idea de que las primeras capas de una CNN aprenden características generales de las imágenes, como bordes y texturas, mientras que las capas más profundas capturan patrones específicos del conjunto de datos en el que fueron entrenadas. Esto permite reutilizar la red como extractor de características y personalizar las últimas capas según la nueva tarea.

Casos de uso destacados

- **Clasificación de imágenes con datos limitados**: cuando se dispone de un conjunto de datos pequeño, utilizar un modelo preentrenado permite evitar el sobreajuste.
- **Detección de objetos**: modelos preentrenados como YOLO o Faster R-CNN permiten reconocer objetos en imágenes sin necesidad de entrenar desde cero.
- **Medicina y diagnóstico por imágenes**: se han desarrollado modelos basados en Transfer Learning para detectar enfermedades a partir de imágenes de rayos X o resonancias magnéticas.
- **Visión en robots y vehículos autónomos**: se aplican modelos preentrenados para la detección de peatones, señales de tráfico y otros objetos en el entorno.

8.1. Introducción a modelos preentrenados de Transfer Learning

Existen varias arquitecturas de modelos preentrenados que se han desarrollado y optimizado para tareas de visión por computador. Algunos de los más utilizados incluyen:

VGG16

- Tiene 16 capas de profundidad con filtros de convolución de tamaño 3x3 y capas completamente conectadas al final.
- Es fácil de entender y modificar, pero tiene un alto costo computacional debido a su gran número de parámetros.
- Tiene variantes como la VGG19.

ResNet50

- Introduce el concepto de Residual Learning para resolver el problema del gradiente desaparecido en redes profundas.
- Utiliza skip connections para permitir que la información fluya sin degradarse en capas profundas.
- Es más eficiente que VGG16 en términos de precisión y uso de parámetros.

InceptionV3

- Utiliza bloques de convoluciones con diferentes tamaños de filtros en paralelo para capturar múltiples niveles de características en una imagen.
- Optimiza el uso de los recursos computacionales mediante módulos Inception que reducen el número de operaciones.
- Es una opción popular para tareas de clasificación de imágenes.

MobileNet

- Diseñado para dispositivos móviles con recursos computacionales limitados.
- Utiliza convoluciones separables en profundidad para reducir la cantidad de cálculos necesarios.
- Es más liviano y rápido que otras arquitecturas, aunque con una ligera pérdida de precisión.

Keras proporciona múltiples modelos preentrenados en el conjunto de datos ImageNet. Algunos de los más utilizados incluyen:

```
from tensorflow.keras.applications import VGG16, ResNet50, InceptionV3

# Carga de VGG16 con pesos de ImageNet
vgg_model = VGG16(weights='imagenet')

resnet_model = ResNet50(weights='imagenet')

inception_model = InceptionV3(weights='imagenet')
```

Los modelos preentrenados se pueden usar como extractores de características o ajustarse (fine-tuning) para una tarea específica.

8.2. Implementación práctica de Transfer Learning

Cuando se usa un modelo preentrenado, una opción es **congelar las capas convolucionales** y agregar nuevas capas densas para la clasificación específica del problema.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.applications import VGG16
from tensorflow.keras.optimizers import Adam

# Cargar VGG16 sin la capa de clasificación final
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(224, 224, 3))

# Congelar las capas convolucionales
for layer in base_model.layers:
    layer.trainable = False

# Agregar capas densas personalizadas
x = Flatten()(base_model.output)
x = Dense(256, activation='relu')(x)
x = Dense(1, activation='sigmoid')(x)  # Para clasificación binaria

# Crear el modelo final y compilar
model = Model(inputs=base_model.input, outputs=x)
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Adaptación de los datos a modelos preentrenados

Los modelos preentrenados en Keras han sido entrenados en ImageNet, lo que significa que esperan entradas con un formato y preprocesamiento específicos:

- **Tamaño de imagen:** Cada modelo tiene un tamaño de entrada esperado.
  - VGG16, ResNet50, InceptionV3: 224x224 píxeles
  - InceptionV3, Xception: 299x299 píxeles

- **Normalización de píxeles:** algunos modelos requieren normalización específica:
  - VGG16 y ResNet50 esperan valores de píxeles en el rango [0, 255], pero suelen utilizar preprocess_input() de Keras para normalizarlos.
  - InceptionV3 y MobileNet usan una normalización de [-1, 1].

Ejemplo de preprocesamiento para VGG16:

```
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.preprocessing.image import load_img,
img_to_array
import numpy as np

# Cargar imagen y ajustar tamaño
img = load_img('imagen.jpg', target_size=(224, 224))
img_array = img_to_array(img)
```

```
img_array = np.expand_dims(img_array, axis=0)

# Aplicar el preprocesamiento específico
img_array = preprocess_input(img_array)
```

Fine-tuning: ajuste de las últimas capas y cuándo hacerlo

Para mejorar el rendimiento, se pueden descongelar algunas capas superiores y reentrenarlas con una tasa de aprendizaje baja. Este método permite que la red ajuste las características aprendidas en ImageNet a la nueva tarea sin perder por completo los patrones generales aprendidos en las capas profundas.

```
# Descongelar algunas capas superiores
for layer in base_model.layers[-4:]:  # Descongelar 4 últimas capas
    layer.trainable = True

# Compilar nuevamente con una tasa de aprendizaje baja
model.compile(optimizer=Adam(learning_rate=0.00001),
loss='binary_crossentropy', metrics=['accuracy'])
```

La estrategia de congelación de capas depende del tamaño del dataset y de su similitud con ImageNet:

1. **Congelar todas las capas (Feature Extraction)**
   - Si el conjunto de datos nuevo es pequeño.
   - Si las clases del nuevo problema son similares a las de ImageNet.
   - Se agregan nuevas capas densas al final para la clasificación específica.

   ```
   for layer in base_model.layers:
       layer.trainable = False
   ```

2. **Descongelar algunas capas superiores (Fine-Tuning)**
   - Si el conjunto de datos es lo suficientemente grande para evitar sobreajuste.
   - Si el problema es diferente de ImageNet y requiere aprender características más específicas.
   - Se recomienda descongelar solo las últimas capas convolucionales.

   ```
   # Descongelar las últimas 10 capas
   for layer in base_model.layers[-10:]:
       layer.trainable = True
   ```

Ajuste de hiperparámetros en Fine-Tuning

Al realizar *fine-tuning*, es importante ajustar correctamente los hiperparámetros:

1. **Tasa de aprendizaje (learning rate)**
   - Para *fine-tuning*, se recomienda una tasa de aprendizaje baja (1e-5 o 1e-6). Ejemplo:

```
from tensorflow.keras.optimizers import Adam
model.compile(optimizer=Adam(learning_rate=1e-5),
loss='categorical_crossentropy', metrics=['accuracy'])
```

2. **Regularización para evitar sobreajuste**
   - **Dropout:** Apagar aleatoriamente algunas neuronas en la capa densa.
   - **L2 Regularization:** Añadir penalización a los pesos excesivos.

```
from tensorflow.keras.layers import Dropout
x = Dense(256, activation='relu', kernel_regularizer='l2')(x)
x = Dropout(0.5)(x)
```

3. **Tamaño del batch**
   - Fine-tuning requiere un batch pequeño (16 o 32) para evitar sobreajuste y mejorar la convergencia.

8.3. Ventajas de usar modelos preentrenados, comparativa y limitaciones

1. **Ahorro de tiempo y recursos:** entrenar una CNN desde cero requiere grandes volúmenes de datos y potencia computacional.
2. **Mejora del rendimiento en conjuntos de datos pequeños:** los modelos preentrenados han aprendido representaciones útiles de imágenes en general.
3. **Evita el sobreajuste:** especialmente útil cuando el dataset específico no es suficientemente grande.
4. **Facilidad de implementación:** herramientas como Keras y TensorFlow permiten cargar y ajustar modelos preentrenados con pocas líneas de código.
5. **Adaptabilidad:** se pueden ajustar parcialmente para nuevas tareas sin perder la generalización obtenida en ImageNet.

Comparación con el entrenamiento desde cero

Característica	Transfer Learning	Sin TL
Datos requeridos	Pocos	Muchos
Tiempo de entrenamiento	Corto	Largo
Computación necesaria	Baja	Alta
Rendimiento inicial	Alto (en general)	Bajo (requiere ajustes)
Posibilidad de adaptación	Alta	Media

Como vemos, Transfer Learning es una estrategia poderosa para construir modelos eficientes en visión por computador sin la necesidad de enormes recursos computacionales y grandes conjuntos de datos. Su uso se ha extendido en múltiples aplicaciones, desde diagnóstico médico hasta la industria automotriz y la seguridad.

Limitaciones del Transfer Learning

Aunque Transfer Learning es una estrategia poderosa, presenta algunas limitaciones que debemos tener presentes:

- **Si el dominio de datos es muy diferente de ImageNet, la transferencia de aprendizaje puede ser limitada.**
  - Por ejemplo, un modelo preentrenado en ImageNet (fotos naturales) puede no ser óptimo para imágenes médicas o satelitales.
  - En estos casos, puede ser mejor reentrenar la red desde cero o usar un modelo preentrenado en un dataset más relevante (ej., modelos preentrenados en radiografías para diagnóstico médico).

- **Algunas arquitecturas preentrenadas son computacionalmente costosas.**
  - Modelos como VGG16 o ResNet50 pueden ser pesados para dispositivos con baja capacidad.
  - Modelos más ligeros como MobileNet o EfficientNet pueden ser una mejor alternativa.

9. Fundamentos de CNN con PyTorch

Cuando se trabaja con redes neuronales convolucionales, tanto PyTorch como Keras son frameworks populares que ofrecen diferentes enfoques para el desarrollo de modelos de deep learning. A continuación, vamos a introducir a PyTorch explicando algunas de las principales diferencias entre ambos:

1. Filosofía y enfoque

- Keras está diseñado para ser una API de alto nivel, enfocada en la facilidad de uso y la rapidez en la implementación de modelos. Se basa en TensorFlow y utiliza una estructura más declarativa.
- PyTorch es más flexible y basado en programación imperativa, lo que permite mayor control sobre el flujo de ejecución y facilita la depuración, aunque ello entraña mayor código y complejidad de implementación.

2. Definición de modelos

- Keras usa un enfoque basado en Sequential o en el API funcional para definir modelos de forma estructurada y clara.
- PyTorch requiere definir explícitamente una clase que herede de nn.Module, proporcionando mayor flexibilidad en la estructura del modelo.

3. Entrenamiento y optimización

- Keras utiliza métodos predefinidos como `model.fit()` para entrenar los modelos con un flujo simplificado.
- PyTorch requiere una gestión más explícita del bucle de entrenamiento y de la optimización. Además, suele ser más eficiente en el uso de recursos computacionales que Keras.

4. Flexibilidad y personalización

- Keras es más fácil de usar para arquitecturas estándar, pero puede ser más restrictivo al intentar modificar detalles internos del modelo.
- PyTorch otorga mayor flexibilidad para modificar arquitecturas y personalizar el flujo de entrenamiento, lo que lo hace preferido para investigación y modelos más avanzados.

En conclusión, si se busca rapidez en el desarrollo y facilidad de uso, Keras es la mejor opción. Sin embargo, si se requiere mayor personalización y control sobre el entrenamiento y los datos, así como un mejor uso de recursos, PyTorch es más adecuado. Veamos a continuación un código de ejemplo en PyTorch:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32,
shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32,
shuffle=False)

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16,
kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(32 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 32 * 8 * 8)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
for epoch in range(num_epochs):
    running_loss = 0.0
    for images, labels in trainloader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
```

```
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    print(f'Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}')

correct = 0
total = 0
with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy: {accuracy:.2f}%')
```