Resumen del capítulo: Redes neuronales convolucionales

Convolución

Las redes neuronales completamente conectadas no pueden trabajar con imágenes grandes. Si no hay suficientes neuronas, la red no podrá encontrar todas las conexiones, y si hay demasiadas, la red estará sobreajustada. La convolución es una solución simple para esto.

Para encontrar los elementos más importantes para la clasificación, **la convolución** aplica las mismas operaciones a todos los pixeles.

Comencemos con una **convolución unidimensional**. Ya que no hay imágenes unidimensionales, observemos esta operación en una secuencia.

Vamos a definir *w* como pesos y *s* como secuencia.

La convolución (*c*) se hace así: los pesos (*w*) se mueven a lo largo de la secuencia (*s*) y se calcula el producto escalar para cada posición en la secuencia. La longitud *n* del vector de los pesos nunca es mayor que la longitud *m* del vector de la secuencia; de otro modo, no habría una posición a la que se pudiera aplicar la convolución.

Vamos a expresar una operación de convolución unidimensional con una fórmula:

$$C_k = \sum_{t=0}^{n-1} S_{k+t} W_t$$

Aquí, *t* es el índice para calcular el producto escalar y *k* es un valor entre 0 y (m - n + 1).

Se escoge el número (m - n + 1) para que los pesos no excedan a la secuencia.

Vamos a expresar una convolución unidimensional en Python:

```
def convolve(sequence, weights):
    convolution = np.zeros(len(sequence) - len(weights) + 1)
    for i in range(convolution.shape[0]):
        convolution[i] = np.sum(weights * sequence[i:i + len(weights)])
```

Ahora vamos a ver cómo funciona la **convolución bidimensional** (2D). Tomemos una imagen bidimensional *s* con un tamaño de *m*×*m* pixeles y una matriz de peso de *n*×*n* pixeles. La matriz es el **kernel** (núcleo) de la convolución.

El kernel se mueve en la imagen de izquierda a derecha y de arriba a abajo. Sus pesos se multiplican por cada pixel en todas las posiciones. Los productos se suman y se registran como los pixeles resultantes.

Para ver esto en acción, vamos a convolucionar las siguientes matrices:

```
s = [[1, 1, 1, 0, 0],
     [0, 1, 1, 1, 0],
     [0, 0, 1, 1, 1],
     [0, 0, 1, 1, 0],
     [0, 1, 1, 0, 0]]

w = [[1, 0, 1],
     [0, 1, 0],
     [1, 0, 1]]
```

La convolución bidimensional se puede expresar mediante esta fórmula:

$$C_{k1,\,k2} = \sum_{t_1=0}^{n} \sum_{t_2=0}^{n} S_{k1+t1,\,k2+t2} W_{t1,\,t2}$$

Puedes encontrar los contornos de esta imagen usando la convolución. Los contornos horizontales se pueden encontrar mediante la convolución con el siguiente kernel:

```
np.array([[-1, -2, -1],
          [ 0,  0,  0],
          [ 1,  2,  1]])
```

Este kernel fue descubierto por los científicos estadounidenses Irwin Sobel y Gary Feldman, y su propósito específico es encontrar los contornos de una imagen. Sirve para resaltar los contornos mejor que el kernel anterior de esta lección.

Usa el siguiente kernel para encontrar contornos verticales:

```
np.array([[-1, 0, 1],
          [-2, 0, 2],
          [-1, 0, 1]])
```

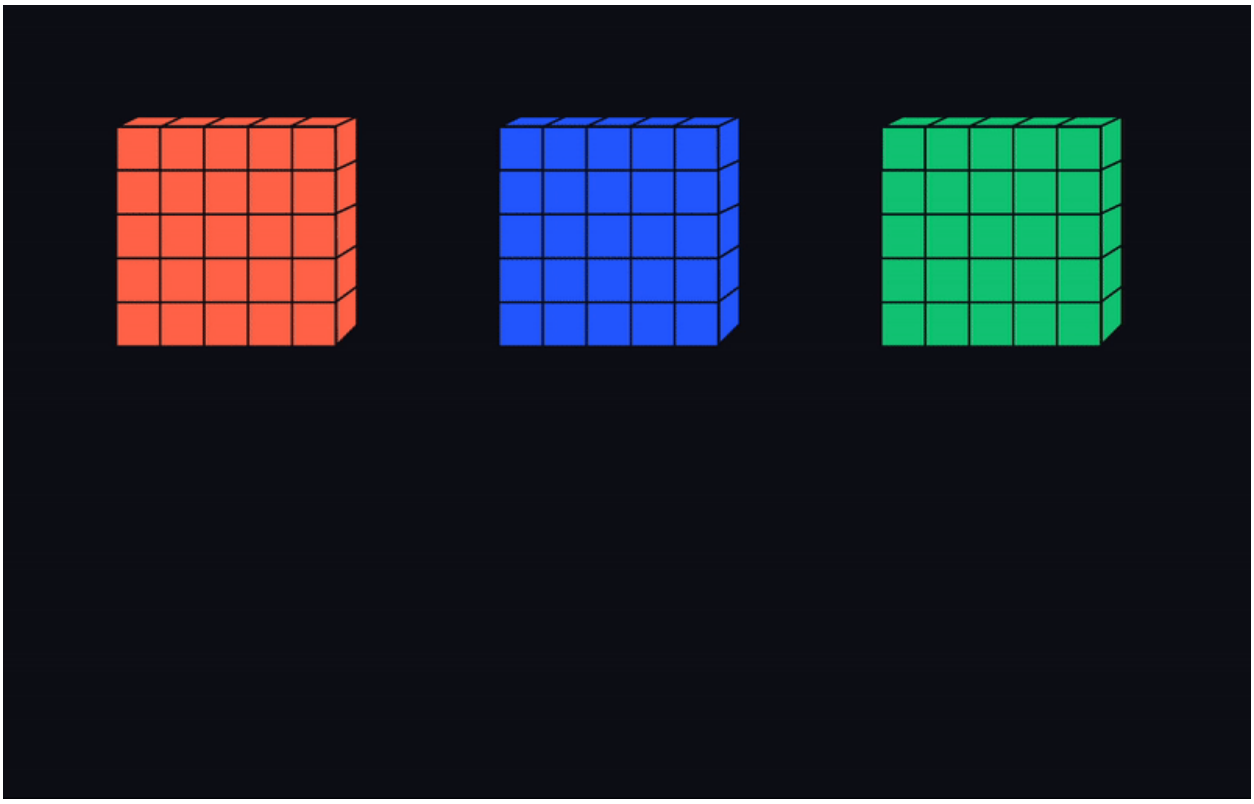
Capas convolucionales

Hablemos de las capas convolucionales y el papel que desempeñan en las redes neuronales convolucionales (CNN). Las **capas convolucionales** aplican una operación de convolución a las imágenes de input y hacen la mayor parte del cómputo dentro de la red.

Una capa convolucional consiste en **filtros** (conjuntos de pesos) que se pueden personalizar y entrenar, los cuales se aplican a la imagen. En esencia, un filtro es una matriz cuadrada con tamaño de $K \times K$ pixeles.

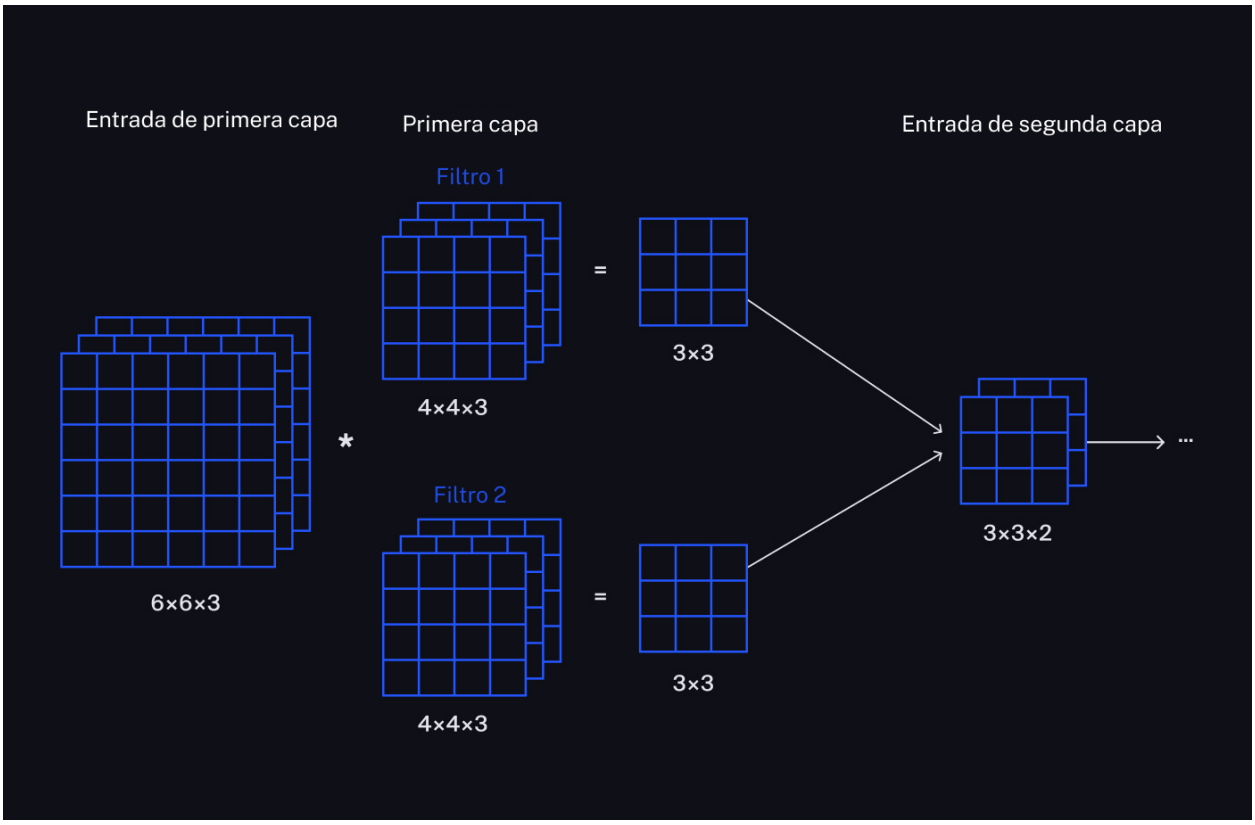
Si el input es una imagen de color, se agrega al filtro una tercera dimensión, la **profundidad**. En este caso, el filtro ya no es una matriz, sino un tensor, o una matriz multidimensional.

Veamos un ejemplo. A continuación puedes ver tres canales de color: rojo, azul y verde. Un filtro de 3x3x3 (tres pixeles de ancho, alto y profundidad) se mueve a través de la imagen input en cada canal, realizando una operación de convolución. No se mueve a través de la tercera dimensión y cada color tiene pesos diferentes. En consecuencia, las imágenes resultantes se doblan hacia el resultado final de la convolución.



Una capa convolucional puede tener varios filtros, cada uno de los cuales devuelve una imagen bidimensional que se puede reconvertir a una tridimensional. En la siguiente capa convolucional, la profundidad de los filtros será igual al número de filtros de la capa anterior.

El asterisco (*) indica una operación de convolución.



Las capas convolucionales contienen menos parámetros que las capas completamente conectadas, lo que hace que sea más fácil entrenarlas.

Echemos un vistazo a las configuraciones de la capa convolucional:

1. *Padding* o relleno. Esta configuración coloca ceros en los bordes de la matriz (padding cero) de modo que los pixeles más alejados participen en la convolución al menos la misma cantidad de veces que los pixeles centrales. Esto evita que se pierda información importante. Los ceros agregados también participan en la convolución y el tamaño del padding determina el ancho del padding cero.

2. **Striding** o **stride** (paso). Esta configuración desplaza el filtro en más de un pixel y genera una imagen de output más pequeña.

Vamos a calcular el tamaño del tensor de output para la capa convolucional. Si la imagen inicial tiene un tamaño de W×W×D, un filtro K×K×D, padding (P) y un paso (S), entonces la nueva imagen de tamaño *W'* se puede determinar de la siguiente manera:

$$W' = \frac{(W - K + 2P)}{S} + 1$$

Usa este visualizador en GitHub para ver cómo se comportan las convoluciones con diferentes parámetros.

Capas convolucionales en Keras

Vamos a crear una capa convolucional llamada **Conv2D.**

```
keras.layers.Conv2D(filters, kernel_size, strides, padding, activation)
```

Esto es lo que significan todos los parámetros:

- **filters:** el número de filtros, que corresponde al tamaño del tensor de output.
- **Kernel_size:** las dimensiones espaciales del filtro *K*. Todo filtro es un tensor de tamaño *K*×*K*×*D*, donde *D* es igual a la profundidad de la imagen de input.
- **strides**: un paso (o stride) determina cuán lejos se desplaza el filtro sobre la matriz de input. De forma predeterminada, está configurado en 1.
- **padding:** este parámetro define el ancho del padding cero. Hay dos tipos de padding: *valid* (válido) y *same* (igual). El tipo por defecto es *valid* y es igual a cero. *Same* establece el tamaño del padding automáticamente de modo que el ancho y la altura del tensor de output sea igual al ancho y la altura del tensor de input.
- **activation:** esta función se aplica inmediatamente después de la convolución. Puedes usar las funciones de activación que ya conoces: `'relu'` y `'sigmoid'`. De manera predeterminada, este parámetro es *None* (esto significa que la activación está deshabilitada).

Para que los resultados de la capa convolucional sean compatibles con la capa completamente conectada, debes conectar una nueva capa llamada **Flatten** ("aplanar"), que convierte al tensor multidimensional en unidimensional.

Por ejemplo, una imagen de input con el tamaño 32x32x3 pasa por una capa convolucional, luego por una totalmente conectada, y entre ellas se debe colocar una capa Flatten:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, Flatten, Dense


model = Sequential()

# este tensor mide (None, 32, 32, 3)
# la primera dimensión define diferentes objetos
# está configurada en None porque desconocemos el tamaño del lote

model.add(Conv2D(filters=4, kernel_size=(3, 3), input_shape=(32, 32, 3)))

# este tensor mide (None, 30, 30, 4)

model.add(Flatten())

# este tensor mide (None, 3600)
```

```
# donde 3600 = 30*30*4

model.add(Dense(...))
```

Arquitectura LeNet

Con las técnicas de **pooling** (agrupación), puedes reducir el número de los parámetros del modelo. Un ejemplo de esto es la operación **Max Pooling*, que se hace así:

1. Se determina el tamaño del kernel (por ejemplo, 2x2).

2. El kernel comienza a moverse de izquierda a derecha y de arriba a abajo; en cada cuadro de cuatro pixeles hay un pixel con el valor máximo.

3. El pixel con el máximo valor se mantiene, mientras que los que le rodean desaparecen.

4. El resultado es una matriz formada solo por los pixeles con los valores máximos.

En Keras también puedes usar la operación **AveragePooling**. Estas son las principales diferencias entre dichas técnicas:

- *MaxPooling* devuelve el valor máximo de pixel del grupo de pixel dentro de un canal. Si la imagen de input tiene un tamaño de $W \times W$, entonces el tamaño de la imagen de output es W/K, donde K es el tamaño del kernel.

- *AveragePooling* devuelve el valor promedio de un grupo de pixeles dentro de un canal.

En Keras, la operación AveragePooling se escribe así:

```
keras.layers.AveragePooling2D(pool_size=(2, 2), strides=None, padding='valid', ...)
```

Veamos cada parámetro:

- **Pool_size:** tamaño de pooling (agrupación). Cuanto más grande sea, más pixeles vecinos se involucran.
- **strides**: un paso (o stride) determina cuán lejos se desplaza el filtro sobre la matriz de input. Si se especifica *None*, el paso es igual al tamaño de pooling.
- **padding:** este parámetro define el ancho del padding cero. El tipo predeterminado del padding es *valid*, que es igual a cero. *Same* establece el tamaño del padding automáticamente.

Los parámetros de *MaxPooling2D* son similares a estos.

Ahora tenemos todas las herramientas para crear una arquitectura popular para clasificar imágenes con tamaño de 20-30 pixeles: **LeNet**. La red toma su nombre de su creador, Yann André LeCun, desarrollador de la tecnología de compresión de imágenes *DjVu* y el encargado del Laboratorio de inteligencia artificial de Facebook.

LeNet se estructura de la siguiente manera:

1. La red comienza con dos o tres capas de 5x5 alternando con *AveragePooling* con un tamaño de 2x2. Estas reducen gradualmente la resolución espacial y recopilan toda la información de la imagen en una pequeña matriz de unos 5 píxeles.
2. El número de filtros aumenta de capa a capa para evitar la pérdida de información importante.
3. Hay una o dos capas totalmente conectadas al final de la red. Estas recopilan y clasifican todas las características.

Así es como creamos LeNet en Keras:

```
model = Sequential()
```

```
model.add(Conv2D(6, (5, 5), padding='same', activation='tanh',
                 input_shape=(28, 28, 1)))
model.add(AvgPool2D(pool_size=(2, 2)))

model.add(Conv2D(16, (5, 5), padding='valid', activation='tanh'))
model.add(AvgPool2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(120, activation='tanh'))

model.add(Dense(84, activation='tanh'))

model.add(Dense(10, activation='softmax'))

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

El algoritmo Adam

El descenso de gradiente estocástico (*SGD* en inglés) no es el algoritmo más eficiente para entrenar una red neuronal. Si el paso es muy pequeño, el entrenamiento podría tardar demasiado. Si es demasiado grande, puede que no logre el entrenamiento mínimo requerido. El algoritmo **Adam** automatiza la selección del paso. Escoge diferentes parámetros para diferentes neuronas, lo que acelera el entrenamiento del modelo.

Para entender cómo funciona este algoritmo, mira esta visualización creada por Emilien Dupont de la Universidad de Oxford. Se muestra cuatro algoritmos: SGD** a la izquierda, el algoritmo Adam a la derecha y entre ellos hay otros dos algoritmos similares a Adam (de los cuales no hablaremos en detalle). De los cuatro, *Adam* es la forma más rápida de encontrar el mínimo.

Vamos a escribir el *algoritmo Adam* en *Keras*:

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

Hay que determinar la clase algoritmo para configurar los hiperparámetros:

```
from tensorflow.keras.optimizers import Adam
optimizer = Adam()

model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',
```

```
              metrics=['acc'])El algoritmo *Adam* es la tasa de aprendizaje. Esta es la pa
rte del descenso de gradiente donde comienza el algoritmo. Se escribe así:
```

```
optimizer = Adam(lr=0.01)
```

La tasa de aprendizaje predeterminada es 0.001. A veces reducirla puede ralentizar el aprendizaje, pero eso mejora la calidad del modelo en general.

Generadores de datos

Las matrices se almacenan en la RAM, no en el disco duro del PC. Ahora bien, ¿qué pasaría si necesitaras crear una matriz de terabytes de imágenes? ¡Qué cosa tan difícil de siquiera imaginar! Después de todo, los recursos de RAM son limitados.

Para lidiar con una cantidad tan grande de imágenes, necesitas implementar la carga de datos dinámica.

La librería Keras tiene una herramienta útil para esto, **ImageDataGenerator** (materiales en inglés):

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

La clase ImageDataGenerator forma lotes con imágenes y etiquetas de clase de acuerdo con las fotos que están en las carpetas. Vamos a ponerla a prueba.

```
# Generador de datos
datagen = ImageDataGenerator()
```

Para extraer datos de una carpeta, llama a la función **flow_from_directory()**.

```
datagen_flow = datagen.flow_from_directory(
    # carpeta con el conjunto de datos
    '/dataset/',
    # tamaño de la imagen objetivo,
    target_size=(150, 150),
    # tamaño del lote
    batch_size=16,
    # modo de clase
    class_mode='sparse',
```

```
    # configurar un generador de números aleatorios
    seed=54321)
```

```
Se encontraron 1683 imágenes pertenecientes a 12 clases.
```

El generador de datos encontró 12 clases (carpetas) y un total de 1683 imágenes.

Echemos un vistazo a los argumentos:

- `target_size=(150, 150)`: argumento con el ancho y alto objetivo de la imagen. Las carpetas pueden contener imágenes de distintos tamaños, pero las redes neuronales necesitan que todas tengan las mismas dimensiones.
- `batch_size=16`: el número de imágenes en los lotes. Cuantas más imágenes haya, más eficaz será el entrenamiento del modelo. No cabrán demasiadas imágenes en la memoria de la GPU, así que 16 es el valor perfecto para iniciar.
- `class_mode='sparse'`: argumento que indica el modo de output de la etiqueta de clase. `sparse` significa que las etiquetas corresponderán al número de la carpeta.

  Puedes saber cómo se relacionan los números de clase con los nombres de las carpetas de esta manera:

  ```
  # índices de clase
  print(datagen_flow.class_indices)
  ```

Llamar al método `datagen.flow_from_directory(...)` devolverá un objeto a partir del cual se pueden obtener los pares "imagen-etiqueta" usando la función *next()*:

```
features, target = next(datagen_flow)

print(features.shape)
```

El resultado es un tensor de cuatro dimensiones con dieciséis imágenes de 150x150 y tres canales de colores.

Para entrenar al modelo con estos datos, vamos a pasar el objeto `datagen_flow` al método *fit()***. La época no debería ser demasiado larga. Para limitar el tiempo de

entrenamiento, hay que especificar el número de lotes de conjuntos de datos en el parámetro `steps_per_epoch`:

```
model.fit(datagen_flow, steps_per_epoch=len(datagen_flow))
```

El método `fit()` debe contener conjuntos de entrenamiento y validación. Para esto, hay que crear dos generadores de datos para cada conjunto.

```
# indica que el conjunto de validación contiene
# 25% de objetos aleatorios
datagen = ImageDataGenerator(validation_split=0.25)

train_datagen_flow = datagen.flow_from_directory(
    '/datasets/fruits_small/',
    target_size=(150, 150),
    batch_size=16,
    class_mode='sparse',
    # indica que este es el generador de datos para el conjunto de entrenamiento
    subset='training',
    seed=54321)

val_datagen_flow = datagen.flow_from_directory(
    '/datasets/fruits_small/',
    target_size=(150, 150),
    batch_size=16,
    class_mode='sparse',
    # indica que este es el generador de datos para el conjunto de validación
    subset='validation',
    seed=54321)
```

Ahora se inicia el entrenamiento así:

```
model.fit(train_datagen_flow,
          validation_data=val_datagen_flow,
          steps_per_epoch=len(train_datagen_flow),
          validation_steps=len(val_datagen_flow))
```

Aumento de datos de imágenes

Si hay muy pocas muestras de entrenamiento, la red podría sobreentrenarse. El **aumento** se utiliza para expandir artificialmente un conjunto de datos mediante la transformación de las imágenes existentes y evitar el sobreentrenamiento. Los cambios

solo se aplican a los conjuntos de entrenamiento, mientras que los de prueba y validación se quedan igual.

El aumento transforma la imagen original, pero aún preserva sus características principales. Por ejemplo, se puede rotar o reflejar la imagen.

Hay varios tipos de aumento:

- Rotar
- Reflejar
- Cambiar el brillo y el contraste
- Estirar y comprimir
- Difuminar y enfocar
- Agregar ruido

Puedes aplicar más de un tipo de aumento a una imagen.

Sin embargo, el aumento puede causar problemas. Por ejemplo, la clase de la imagen puede cambiar o el resultado podría terminar como una pintura impresionista debido a las alteraciones. Esto afecta la calidad del modelo.

Puedes evitar estos problemas si sigues estas recomendaciones:

- No hagas aumento en los conjuntos de prueba y validación para evitar distorsionar los valores de las métricas.
- Agrega aumentos de forma gradual, uno a la vez, y pon atención a la métrica de calidad del conjunto de validación.
- Siempre deja sin cambiar algunas imágenes del conjunto de datos.

Aumentos en Keras

En *ImageDataGenerator* hay muchas maneras de agregar aumentos de imagen. Por defecto, están deshabilitadas. Hagamos un giro vertical:

```
datagen = ImageDataGenerator(validation_split=0.25,
                             rescale=1./255,
                             vertical_flip=True)
```

Deben crearse diferentes generadores para los conjuntos de entrenamiento y validación:

```
train_datagen = ImageDataGenerator(
    validation_split=0.25,
    rescale=1./255,
    horizontal_flip=True)

validation_datagen = ImageDataGenerator(
    validation_split=0.25,
    rescale=1./255)

train_datagen_flow = train_datagen.flow_from_directory(
    '/dataset/',
    target_size=(150, 150),
    batch_size=16,
    class_mode='sparse',
    subset='training',
    seed=54321)

val_datagen_flow = validation_datagen.flow_from_directory(
    '/dataset/',
    target_size=(150, 150),
    batch_size=16,
    class_mode='sparse',
    subset='validation',
    seed=54321)
```

Configura los objetos *train_datagen_flow* y *val_datagen_flow* al mismo valor de `seed` para evitar que estos conjuntos compartan elementos.

ResNet en Keras

Importa *ResNet50* desde Keras (50 indica el número de capas de la red).

```
from tensorflow.keras.applications.resnet import ResNet50

model = ResNet50(input_shape=None,
                 classes=1000,
                 include_top=True,
                 weights='imagenet')
```

Echemos un vistazo a los argumentos:

- `input_shape`: tamaño de la imagen de input. Por ejemplo, `(640, 480, 3)`.

- `classes=1000`: el número de neuronas en la última capa totalmente conectada donde sucede la clasificación.
- `weights='imagenet'`: la inicialización de pesos. *ImageNet* es el nombre de una gran base de datos de imágenes que se usaba al entrenar la red para que clasificara imágenes en 1000 clases. Si comienzas a entrenar la red en ImageNet y luego continúas con la tarea, el resultado será mucho mejor que si solo entrenaras la red desde cero. Escribe `weights=None` para inicializar los pesos al azar.
- `include_top=True`: indica que hay dos capas al final de ResNet (*GlobalAveragePooling2D* y *Dense*). Si la configuras en *False*, estas dos capas no estarán presentes.

Vamos a repasar las capas mencionadas anteriormente:

1. *GlobalAveragePooling2D* funciona como ventana para todo el tensor. Como *AveragePooling2D,* devuelve el valor promedio de un grupo de pixeles dentro de un canal. Usamos *GlobalAveragePooling2D* para tomar el promedio de la información a través de la imagen con el objetivo de obtener un pixel con mayor número de canales (por ejemplo, 512 para ResNet50).

2. *Dense* es la capa totalmente conectada responsable de la clasificación.

Aprendamos cómo usar una red que ha sido entrenada previamente en *ImageNet*. Para adaptar *ResNet50* a nuestra tarea, vamos a quitarle la parte superior y reconstruirla.

```
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Sequential

backbone = ResNet50(input_shape=(150, 150, 3),
                    weights='imagenet',
                    include_top=False)

model = Sequential()
model.add(backbone)
model.add(GlobalAveragePooling2D())
model.add(Dense(12, activation='softmax'))
```

`Backbone` (red troncal) es lo que queda de ResNet50.

Pero aquí está la cosa: digamos que hay un conjunto de datos muy pequeño que solo contiene 100 imágenes y dos clases. Si entrenas ResNet50 en este conjunto de datos, puedes tener la certeza de que se va a entrenar en exceso porque tiene demasiados

parámetros (¡cerca de 23 millones!). La red acabará por obtener predicciones perfectas en el conjunto de entrenamiento, pero también tendrá predicciones aleatorias en el conjunto de prueba.

Para evitar esto, tenemos que "congelar" una parte de la red: algunas capas se quedarán con pesos de *ImageNet* y no entrenarán con descenso de gradiente. Vamos a entrenar una o dos capas totalmente conectadas en la parte superior de la red. De esta manera, se reducirá el número de parámetros de la red, pero se conservará la arquitectura.

Congelemos la red:

```
backbone = ResNet50(input_shape=(150, 150, 3),
                    weights='imagenet',
                    include_top=False)

# congela ResNet50 sin la parte superior
backbone.trainable = False

model = Sequential()
model.add(backbone)
model.add(GlobalAveragePooling2D())
model.add(Dense(12, activation='softmax'))
```

No congelamos la capa totalmente conectada encima de *backbone* para que la red pudiera aprender.

Congelar la red te permite evitar el exceso de entrenamiento e incrementar la tasa de aprendizaje de la red (el descenso de gradiente no necesitará contar derivadas para las capas congeladas).