# REDES NEURONALES CONVOLUCIONALES 1

Las redes neuronales convolucionales (RNC) o CNN (Convolutional Neural Networks) surgieron del estudio de la corteza visual del cerebro y se utilizan en el reconocimiento de imágenes por ordenador desde los 80. En los últimos 10 años, gracias al aumento de la potencia computacional, la cantidad de datos de entrenamiento disponibles y las mejoras encontradas para el entrenamiento de redes profundas (evitar desvanecimiento del gradiente, regularización, redes preentrenadas...), las redes convolucionales han logrado alcanzar un rendimiento sobrehumano en algunas tareas visuales complejas.

Ahora mismo, podemos encontrarlas detrás de muchos ámbitos:

* **Reconocimiento de Imágenes**: son muy utilizadas para clasificar y reconocer objetos en imágenes.
* **Detección de Objetos**: Pueden utilizarse para detectar la presencia y ubicación de objetos en una imagen.
* **Segmentación Semántica**: pueden utilizarse para asignar a cada píxel de una imagen una etiqueta que representa la clase a la que pertenece. Esto es útil en aplicaciones como la identificación de objetos en imágenes médicas.
* **Reconocimiento Facial:** se aplican en sistemas de reconocimiento facial para identificar y verificar rostros en imágenes o videos.
* **Procesamiento de Vídeo**: son utilizadas para analizar y entender secuencias de video, permitiendo tareas como el seguimiento de objetos en movimiento.

Las redes neuronales convolucionales cuentan con bloques de construcción que ya conoces como las capas completamente conectadas y las funciones de activación, pero también presenta dos bloques nuevos: **las capas convolucionales y las capas de pooling**.

*¿Por qué no utilizar simplemente una red neuronal profunda con capas completamente conectadas para las tareas de reconocimiento de imágenes?* Por desgracia, aunque funciona bien para imágenes pequeñas (por ejemplo, MNIST), no funcionan bien para imágenes grandes, debido al número enorme de parámetros que requiere. Por ejemplo, una imagen de 100 x 100 píxeles tiene 10.000 píxeles y, si la primera capa tiene solo 1.000 neuronas (lo cual ya restringe mucho la cantidad de información transmitida a la siguiente capa), eso supone un total de 10 millones de conexiones y eso es solo la primera capa. **Las RNC solucionan este problema utilizando capas parcialmente conectadas y los pesos compartidos.**

## CAPAS CONVOLUCIONALES

Las capas de convolución son componentes fundamentales de las redes convolucionales (CNN). Estas capas son responsables de realizar operaciones de convolución en los datos de entrada para extraer características relevantes. La operación de convolución implica deslizar un filtro (también conocido como kernel) sobre la entrada y realizar productos escalares locales entre los elementos del filtro y la región correspondiente de la entrada. Este proceso ayuda a detectar patrones y características específicas en los datos.

En resumen, son esenciales para aprender y extraer características relevantes de datos estructurados en cuadrículas, como imágenes. Estas capas son capaces de capturar patrones locales y composiciones de patrones que son fundamentales en tareas de visión por computadora y otras aplicaciones relacionadas con datos espaciales.

<https://youtu.be/V8j1oENVz00?si=4XpCnmmLePGuwQA4> (13:24min, empieza en el minuto 3:30)

<https://www.youtube.com/watch?v=4sWhhQwHqug&t=310s> (23:40min)

Algunos conceptos relacionados con las redes convolucionales:

|  |  |
| --- | --- |
|  | **Relleno de ceros:** Se refiere a agregar píxeles de valor cero alrededor de los bordes de una imagen o entrada de datos antes de aplicar ciertas operaciones, como la convolución. El relleno de ceros es útil para asegurar que la información en los bordes de la entrada se tenga en cuenta durante las operaciones. |
|  | **Paso de avance:** al aplicar un paso de avance (stride) mayor que 1 (por ejemplo 2) en operaciones como la convolución o el pooling, se reduce la resolución espacial de la salida, lo que lleva a una representación más compacta. |
|  | En el contexto de una red neuronal convolucional, se **aplican filtros diferentes durante la operación de convolución a una entrada para generar mapas de características diferentes**. Cada filtro detecta patrones específicos en la entrada, y cada mapa de características resalta la presencia de esos patrones. |
|  | Dos capas convolucionales con múltiples filtros (kernels) cada una, procesando una imagen en color con tres canales de color; cada capa convolucional genera como salida un mapa de características por filtro. |

### CAPAS CONVOLUCIONALES CON KERAS

#### CARGAR Y PROCESAR IMAGENES

En primer lugar, vamos a **cargar y preprocesar** un par de imágenes de muestra, utilizando la función **load\_sample\_image()** de Scikit-Learn y **las capas de preprocesamiento** CenterCrop y Rescaling de Keras (en el penúltimo apartado de este documento tienes más información sobre las capas de preprocesamiento):

images = load\_sample\_images()["images"]

images = tf.keras.layers.CenterCrop(height=70, width=120)(images)

images = tf.keras.layers.Rescaling(scale=1 / 255)(images)

|  |  |
| --- | --- |
| Imágenes originales | Imágenes transformadas |
|  |  |

Vamos a echar un vistazo a **la forma de los datos:**

images.shape

TensorShape([2, 70, 120, 3])

*¿Qué significan todas estas dimensiones?* Hay dos imágenes de muestra, lo que explica la primera dimensión con el 2. Después, cada imagen es de 70 x 120, puesto que ese es el tamaño que hemos especificado al crear la capa CenterCrop (las imágenes originales eran de 427 x 640), esto explica la segunda dimensión y la tercera. Y, por último, cada píxel alberga un valor por canal de color y hay tres (rojo, verde y azul), lo que explica la última dimensión con el 3.

#### CREAR CAPA CONVOLUCIONAL

Ahora, vamos a **crear una capa convolucional 2D** y a introducirle estas imágenes para ver qué sale. Para esto, Keras ofrece un objeto capa Convolution2D, con alias Conv2D. Vamos a crear una capa convolucional con 32 filtros, cada uno con un tamaño de 7 x 7 (usando kernel\_size=7, que es equivalente a utilizar kernel\_size=(7,7)), y a aplicar esta capa a nuestro pequeño lote de dos imágenes:

conv\_layer = tf.keras.layers.Conv2D(filters=32, kernel\_size=7)

fmaps = conv\_layer(images)

Ahora **vamos a fijarnos en la forma de la salida**:

fmaps.shape

>>>TensorShape([2, 64, 114, 32])

La forma de la salida es similar a la de la entrada, con **dos diferencias importantes**:

* Hay **32 canales en vez de 3**. Esto se debe a que establecemos filters=32, así que obtenemos 32 mapas de características de salida.
* **Tanto la altura como la anchura se han reducido en 6 píxeles**. Esto se debe al hecho de que la capa **Conv2D no utiliza ningún relleno de ceros por defecto**, lo que significa que perdemos algunos píxeles a los lados de los mapas de características de salida, dependiendo del tamaño de los filtros. En este caso, puesto que el tamaño del kernel es 7, perdemos 6 píxeles en horizontal y 6 píxeles en vertical (es decir, 3 píxeles en cada lado).

Si en vez de eso **configuramos padding="same"**, **las entradas se rellenan con ceros suficientes en todos los lados** para garantizar que los mapas de característica de salida acaban teniendo el mismo tamaño que las entradas:

conv\_layer = tf.keras.layers.Conv2D(filters=32, kernel\_size=7, padding="same")

fmaps = conv\_layer(images)

fmaps.shape

>>>TensorShape([2, 70, 120, 32])

Por lo que los valores de salida serían los mismos que los de entrada, exceptuando el último que pasa a ser 32 en vez de 3. Esto se debe a que establecemos 32 filtros como hemos explicado previamente.

Estas dos opciones de relleno se ilustran en la siguiente figura (por una cuestión de simplicidad, aquí solo se muestra la dimensión horizontal, pero, se aplica la misma lógica a la dimensión vertical).

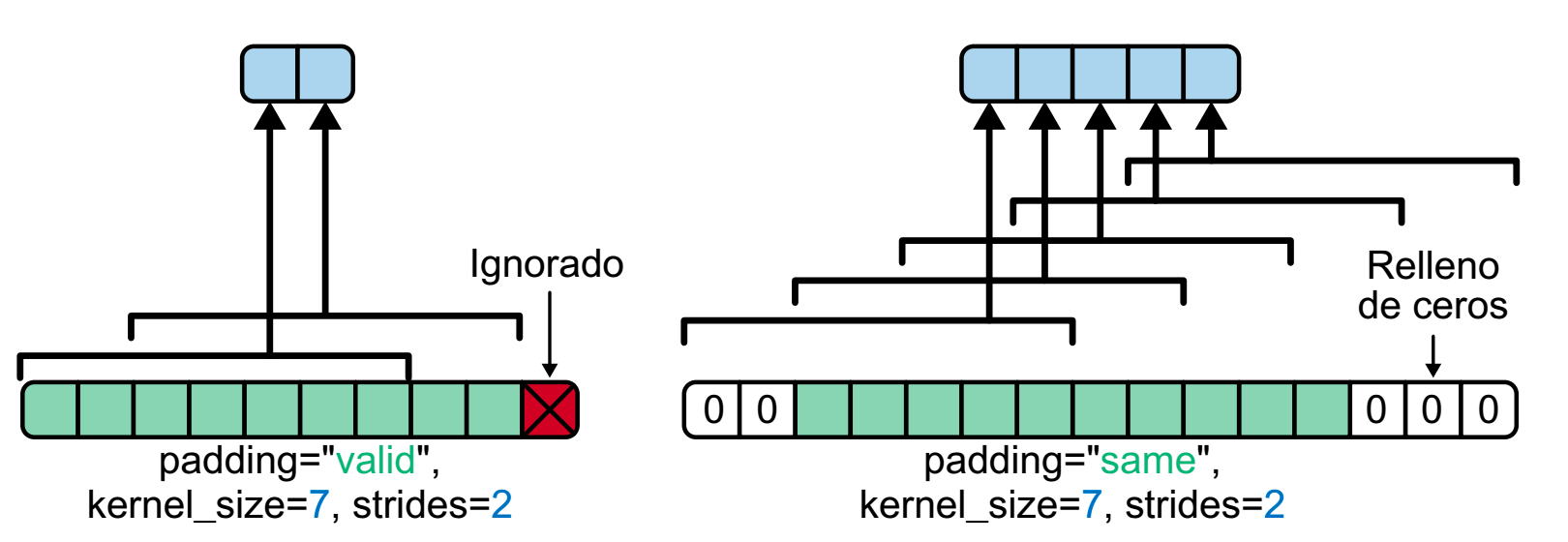
El atributo strides indica el paso con el que se desplaza el filtro a lo largo de las dimensiones espaciales de la entrada durante la operación de convolución. Es decir, especifica el número de píxeles que el filtro se mueve horizontalmente y verticalmente en cada paso.

Diagrama

Descripción generada automáticamente

*Las dos opciones de relleno, cuando strides=1*

Si el paso de avance es mayor que 1 (en cualquier dirección), el tamaño de la salida no será igual al tamaño de la entrada, ni siquiera con padding="same". Por ejemplo, si configuramos strides=2, los mapas de características de salida serán de 35 x 60: reducidos a la mitad tanto vertical como horizontalmente.



*Con pasos de avance mayores que 1, la salida es mucho más pequeña cuando se usa el relleno "same" (y puede que el relleno "valid" ignore algunas entradas).*

Al igual que una capa Dense, una capa Conv2D alberga todos los pesos de la capa, incluyendo los kernels y los sesgos. En el contexto de una capa convolucional:

* **Kernels (filtros):** son los filtros que se aplican a regiones locales de la entrada para extraer características. Cada kernel tiene sus propios pesos, que se aprenden durante el entrenamiento. La capa convolucional puede tener múltiples kernels.
* **Sesgos:** cada kernel tiene un sesgo asociado, que es un término independiente que se suma después de aplicar el kernel a la entrada. Los sesgos también se aprenden durante el entrenamiento.

Estos **pesos son accesibles** a través del método get\_weights( ):

kernels, biases = conv\_layer.get\_weights()

kernels.shape

(7, 7, 3, 32)

biases.shape

(32,)

**La matriz kernels es 4D**, y su forma es **[altura\_kernel, anchura\_kernel, canales\_entrada, canales\_salida]**. **La matriz biases es 1D**, con la forma **[canales\_salida].** El número de canales de salida es igual al número de mapas de características de salida, que también es igual al número de filtros.

Observa que la altura y la anchura de las imágenes de entrada no aparece en la forma del kernel. Eso significa que podemos introducir imágenes de cualquier tamaño en esta capa, siempre y cuando sean al menos igual de grandes que los kernels, y tengan el número adecuado de canales (tres en este caso).

Por último, por lo general conviene especificar una función de activación (como ReLU) al crear una capa Conv2D, y también especificar el inicializador del kernel correspondiente (como la inicialización He). Es por la misma razón que para las capas Dense.

Como puedes ver, las capas convolucionales tienen bastantes hiperparámetros:

* filters
* kernel\_size
* padding
* strides
* activation
* kernel\_initializer
* ...

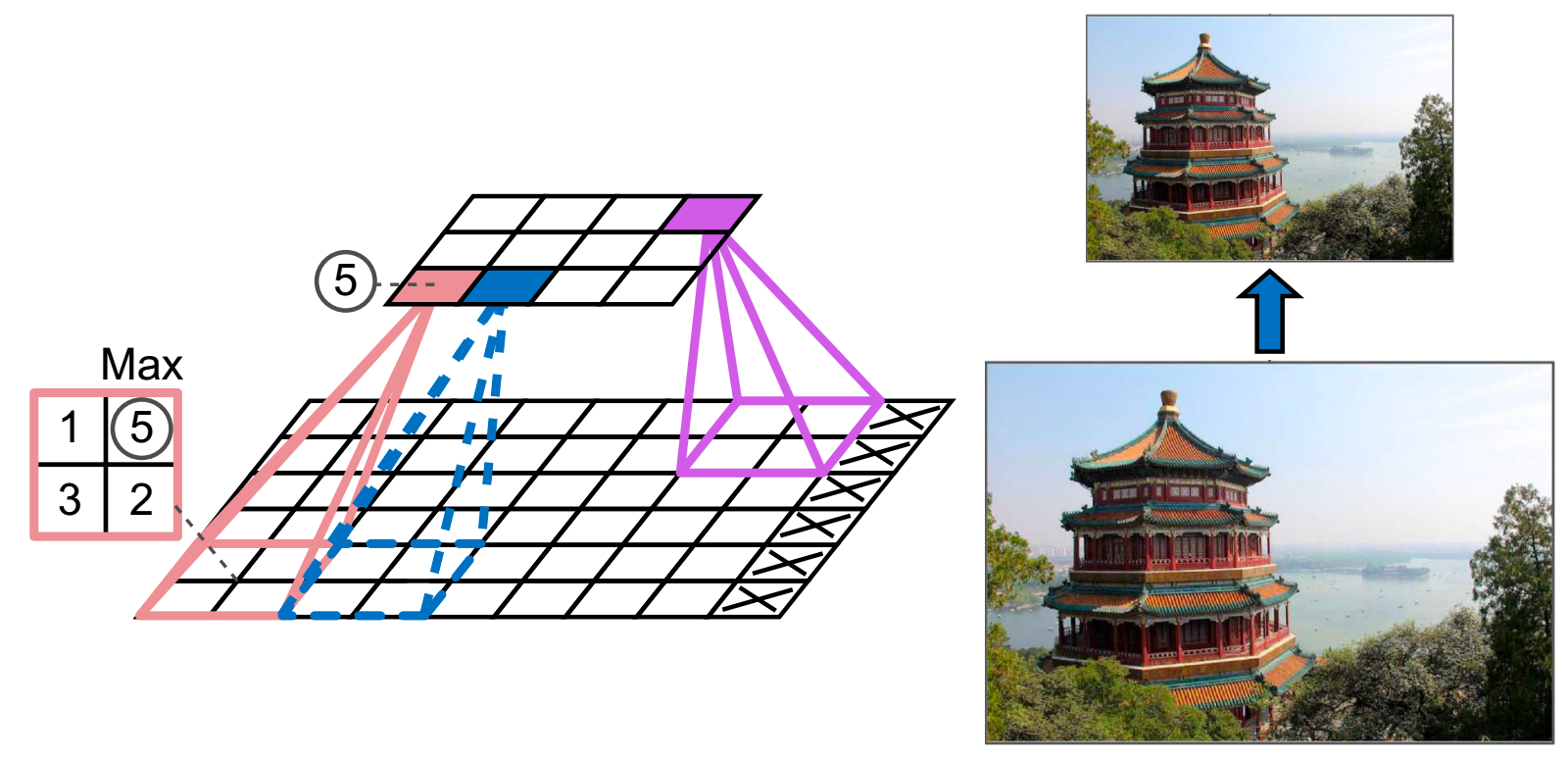
Las capas convolucionales requieren una cantidad enorme de RAM, en especial durante el entrenamiento.

Durante la inferencia (esto es, al hacer una predicción para una nueva instancia), la RAM ocupada por una capa puede liberarse en cuanto se haya calculado la siguiente capa, así que solo necesitamos la RAM requerida por dos capas consecutivas. Pero, en el entrenamiento, hay que conservar todo lo computado en el paso hacia delante para el paso hacia atrás, así que la cantidad de RAM necesaria es (como mínimo) la cantidad de RAM total que requieren todas las capas.

## CAPAS DE POOLING

El objetivo de las capas de pooling es **submuestrear (es decir, reducir) la imagen para disminuir la carga computacional, el uso de memoria y el número de parámetros** (limitando así el riesgo de sobreajuste).

Al igual que ocurre en las capas convolucionales, cada neurona de una capa de pooling está conectada a las salidas de un número limitado de neuronas en la capa anterior, ubicadas en un campo receptivo rectangular pequeño. Hay que definir su tamaño, el paso de avance y el tipo de relleno, igual que antes. Sin embargo, **una neurona de pooling no tiene pesos**; todo lo que hace es agregar las entradas utilizando una función de agregado como la máxima o la media.

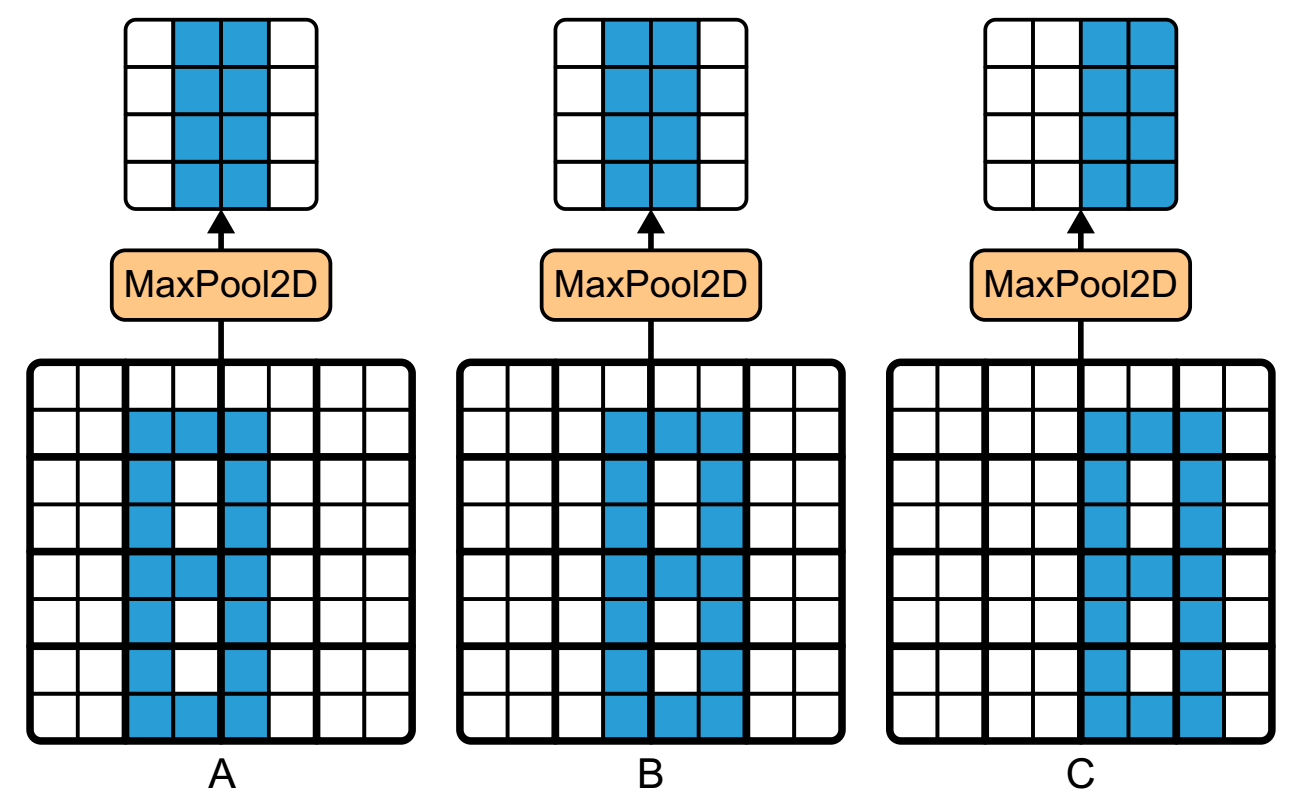


*Capa max pooling (kernel de pooling de 2 x 2, paso de avance de 2, sin relleno)*

En este ejemplo, utilizamos un kernel de pooling de 2 x 2, con un paso de avance de 2 y sin relleno. Solo el valor de entrada máximo de cada campo receptivo llega a la siguiente capa, mientras que las otras entradas se descartan. Por ejemplo, en el campo receptivo inferior izquierdo de la figura, los valores de entrada son 1, 5, 3 y 2, así que solo el valor máximo, 5, se propaga a la siguiente capa. Como el paso de avance es de 2, la imagen de salida tiene la mitad de altura y la mitad de anchura que la imagen de entrada (redondeando hacia abajo, porque no usamos relleno).

Además de *reducir los cálculos, el uso de memoria y el número de parámetros*, una capa max pooling también introduce cierto nivel de **invariancia traslacional**. Esto hace referencia a la capacidad de un modelo o método para mantener su desempeño o capacidad de reconocimiento de patrones incluso cuando las características u objetos en la imagen se desplazan ligeramente en posición.

Por ejemplo, en la siguiente imagen partimos de tres imágenes A, B y C (B y C son iguales a A pero desplazadas). Pasamos las tres imágenes por una capa max pooling con un kernel de 2 x 2 y paso de avance de 2 y vemos que las imágenes A y B son idénticas (invariancia traslacional) y que la de C está desplazada un píxel hacia la derecha (pero sigue habiendo una invariancia del 50 %).



*lnvariancia en traslaciones pequeñas.*

Al insertar una capa max pooling cada pocas capas en una CNN, es posible obtener **cierto nivel de invariancia traslacional a una escala mayor**.

Además, el max pooling **ofrece una pequeña cantidad de invariancia rotacional y una leve invariancia de escala**. Una invariancia así (incluso aunque sea limitada) puede ser útil en casos donde la predicción no debería depender de esos detalles, como las tareas de clasificación.

Sin embargo, el max pooling también tiene inconvenientes. Es evidente que es una operación muy destructiva: incluso con un kernel muy pequeño de 2 x 2 y un paso de avance de 2, la salida será dos veces más pequeña en ambas direcciones (así que esta área será cuatro veces más pequeña), dejando fuera el 75% de los valores de entrada. Además, en algunas aplicaciones, la invariancia no es deseable.

### CAPAS DE POOLING CON KERAS

#### MAX POOLING

El siguiente código crea una capa MaxPooling2D, con alias MaxPool2D, utilizando un kernel de 2 x 2. Los pasos de avance se establecen por defecto como el tamaño del kernel, así que esta capa utiliza un paso de avance de 2 (horizontal y verticalmente). Por defecto, utiliza relleno "valid" (es decir, sin relleno):

max\_pool = tf.keras.layers.MaxPool2D(pool\_size=2)

#### AVERAGE POOLING

Para crear una capa de average pooling, solo hay que utilizar AveragePooling2D, con alias AvgPool2D, en vez de MaxPool2D.

max\_pool = tf.keras.layers.AvgPool2D(pool\_size=2)

Como cabría esperar, funciona exactamente igual que una capa max pooling, salvo por el hecho de que **calcula la media en vez del máximo**.

Las capas de average pooling eran muy populares, pero, ahora, se utiliza sobre todo *capas de max pooling, ya que suelen tener mejor rendimiento*. Puede parecer sorprendente, puesto que, por lo general, al calcular la media se pierde menos información que al calcular el máximo, pero, por otra parte, el max pooling preserva solo las características más fuertes y se deshace de las intrascendentes, así que la siguiente capa recibe una señal más limpia con la que trabajar. Además, *el max pooling ofrece una invariancia traslacional más fuerte que el average pooling y requiere menos cálculos*.

#### GLOBAL AVERAGE POOLING

Otro tipo de capa de pooling que se utiliza a menudo en las arquitecturas modernas es la capa global average pooling, que funciona de manera muy diferente: todo lo que hace es **calcular la media de cada mapa de características completo (es como una capa average pooling utilizando un kernel de pooling con las mismas dimensiones espaciales que las entradas)**. Eso significa que solo genera como salida un solo número por mapa de características y por instancia. Aunque, por supuesto, esto es muy destructivo (se pierde la mayor parte de la información en el mapa de características), puede ser útil justo antes de la capa de salida.

Para crear una capa así, solo tienes que utilizar la clase GlobalAveragePooling2D, con alias GlobalAvgPool2D:

global\_avg\_pool = tf.keras.layers.GlobalAvgPool2D()

## ARQUITECTURAS DE CNN

Las arquitecturas de CNN apilan unas pocas capas convolucionales, después, una capa de pooling, a continuación, otras pocas capas convolucionales, luego otra capa de pooling, y así sucesivamente. La imagen va haciéndose cada vez más pequeña a medida que progresa a través de la red, pero, por lo general, también va haciéndose cada vez más profunda (es decir, con más mapas de características), gracias a las capas convolucionales. En la parte superior de la pila, se añade una red prealimentada corriente, compuesta por capas completamente conectadas (+ReLU) y la capa final genera como salida la predicción (por ejemplo, una capa softmax que genera como salida probabilidades de clases estimadas).

Diagrama

Descripción generada automáticamente

*Arquitectura típica de RNC*

Un error común es utilizar kernels de convolución que son demasiado grandes. Por ejemplo, en vez de utilizar una capa convolucional con un kernel de 5 x 5, apila dos capas con 3 x 3 kernels: utilizará menos parámetros, requerirá menos cálculos y, habitualmente, tendrá mejor rendimiento. La primera capa convolucional es una excepción: por lo general, suele tener un kernel grande (por ejemplo, 5 x 5), normalmente con un paso de avance de 2 o más. Esto reducirá la dimensión espacial de la imagen sin perder demasiada información y, puesto que la imagen de entrada tiene solo tres canales en general, no será demasiado costoso.

Veamos cómo implementar una CNN básica para trabajar con el conjunto de datos Fashion MNIST:

DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel\_size=3, padding="same", activation="relu", kernel\_initializer="he\_normal")

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(28, 28, 1)),

DefaultConv2D(filters=64, kernel\_size=7),

tf.keras.layers.MaxPool2D(),

DefaultConv2D(filters=128),

DefaultConv2D(filters=128),

tf.keras.layers.MaxPool2D(),

DefaultConv2D(filters=256),

DefaultConv2D(filters=256),

tf.keras.layers.MaxPool2D(),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(units=128, activation="relu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dropout(0.5),

tf.keras.layers.Dense(units=64, activation="relu",kernel\_initializer="he\_normal"),

tf.keras.layers.Dropout(0.5),

tf.keras.layers.Dense(units=10, activation="softmax")

])

Vamos a repasar este código:

* **Utilizamos la función functools.partial() para definir DefaultConv2D, que actúa como Conv2D, pero con argumentos predeterminados diferentes:** un tamaño de kernel pequeño de 3, relleno "same", la función de activación ReLU y su correspondiente inicializador He.
* A continuación**, creamos el modelo con Sequential**.
* Su primera capa es una **capa Input** para indicar las dimensiones de los datos de entrada. Las imágenes son de 28 x 28 píxeles, con un solo canal de color (es decir, escala de grises). Cuando carguemos el conjunto de datos Fashion MNIST, hay que asegurarse de que cada imagen tiene esta forma: puede que haya que usar np.reshape() o np.expanddims() para añadir la dimensión de los canales
* La segunda cada es una capa **DefaultConv2D** con 64 filtros bastante grandes (7 x 7). Utiliza el paso de avance predeterminado de 1 porque las imágenes de entrada no son muy grandes.
* Después, **añadimos una capa de max pooling** que utiliza el tamaño de reducción predeterminado de 2, así que divide cada dimensión espacial entre 2.
* Después, **repetimos la misma estructura dos veces: dos capas convolucionales seguidas de una capa max pooling.** Para imágenes más grandes, podríamos repetir esta estructura varias veces más. El número de repeticiones es un hiperparámetro que se puede ajustar.
* **Podemos observar que el número de filtros se duplica a medida que subimos por la CNN hacia la capa de salida (al principio es 64, luego 128, después 256):** tiene sentido que crezca, puesto que, a menudo, el número de características de bajo nivel es bastante bajo (por ejemplo, círculos pequeños, líneas horizontales), pero hay muchas maneras de combinarlas para crear características de nivel superior. Es habitual duplicar el número de filtros después de cada capa de pooling: puesto que una capa de pooling divide cada dimensión espacial entre 2, podemos permitirnos duplicar el número de mapas de características en la siguiente capa sin miedo a hacer explotar el número de parámetros, el uso de memoria o la carga computacional.
* A continuación, **está la red completamente conectada, compuesta por dos capas densas ocultas y una capa de salida densa.** Puesto que es una tarea de clasificación con 10 clases, la capa de salida tiene 10 neuronas, y usa la función de activación softmax. Ten en cuenta que hay que aplastar las entradas justo antes de la primera capa densa, ya que espera una matriz 1D de características para cada instancia. También tenemos dos capas dropout, con una tasa de dropout del 50% cada una, para reducir el sobreajuste.

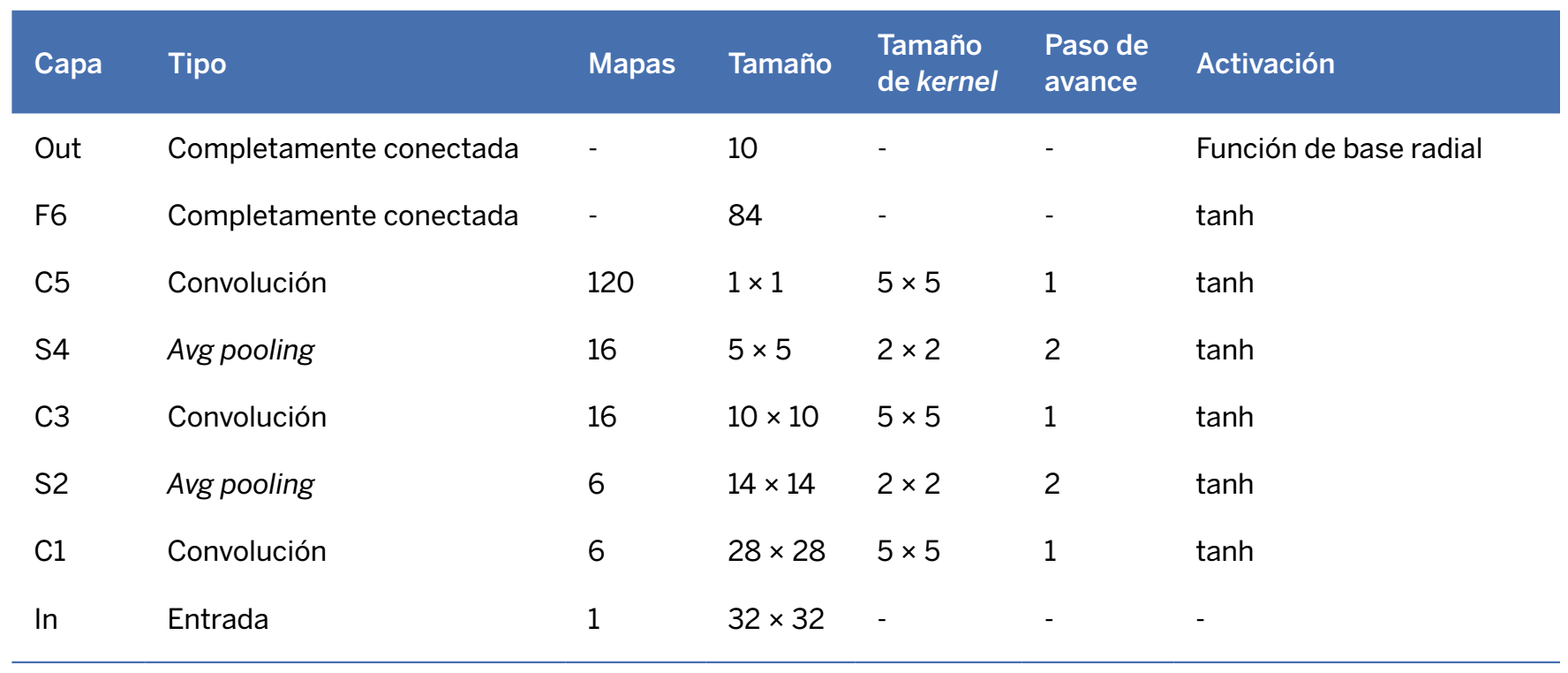
Si compilas este modelo utilizando la pérdida *"sparse\_categorical\_crossentropy"* y ajustas el modelo al conjunto de entrenamiento Fashion MNIST, debería alcanzar una exactitud del 92% en el conjunto de prueba. No es lo mejor, pero está bastante bien.

Hay muchas variantes de esta arquitectura fundamental, lo que ha llevado a avances increíbles en el campo. Una buena medida de este progreso es la tasa de error en competiciones como el reto ILSVRC de lmageNet. En esta competición, la tasa de error del top cinco para la clasificación de imágenes bajó de más del 26% a menos del 2,3% en solo seis años. Las imágenes son bastante grandes (por ejemplo, 256 píxeles de altura) y hay 1.000 clases, algunas de las cuales son muy sutiles (intenta distinguir entre 102 razas de perro). Fijarnos en la evolución de las ganadoras es una buena forma de entender cómo funcionan las CNN y cómo progresa la investigación sobre el deep learning.

Primero, vamos a echar un vistazo a la arquitectura LeNet-5 clásica (1998) y, después, a varias ganadoras del reto ILSVRC: AlexNet (2012), GoogLeNet (2014), ResNet (2015) y SENet (2017). Por el camino, también veremos algunas arquitecturas más, incluyendo Xception, ResNeXt, DenseNet, MobileNet, CSPNet y EfficientNet.

### LENET-5

La arquitectura LeNet-5 (1998) es una piedra angular en la historia de las redes neuronales convolucionales (CNN) y ha sido fundamental para el desarrollo de la visión por computadora. Además, es quizás, la arquitectura de CNN más conocida y se ha utilizado mucho para el reconocimiento de dígitos escritos a mano (MNIST).

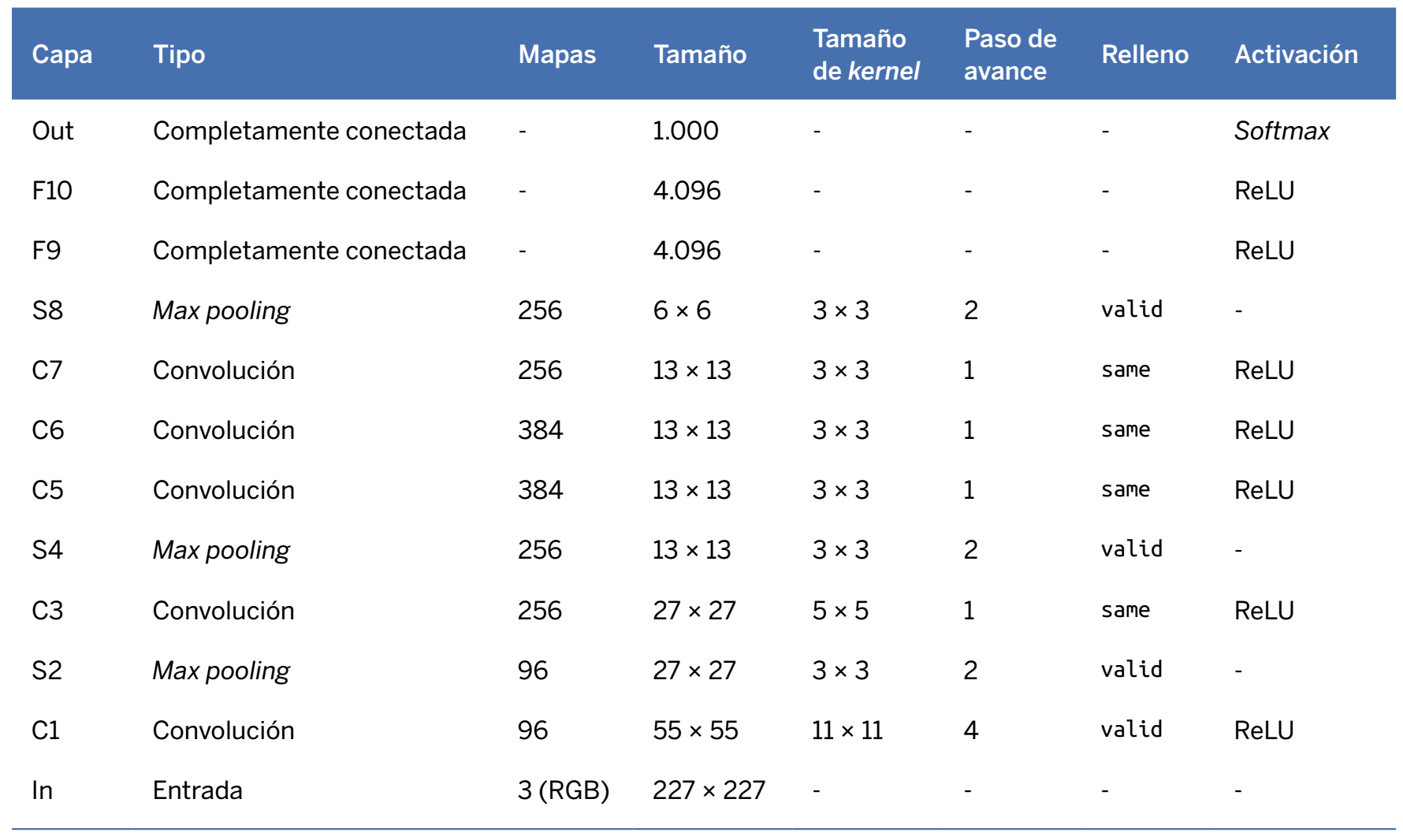


*Arquitectura LeNet-5*

Como vemos, tiene un aspecto bastante similar a nuestro modelo Fashion MNIST: una pila de capas convolucionales y capas de pooling, seguida de una red densa. Quizá la diferencia principal con las CNN de clasificación más modernas sean las funciones de activación: hoy en día, utilizaríamos ReLU en vez de tanh y softmax en vez de la función de base radial. Aunque fue revolucionaria en su tiempo, LeNet-5 puede ser limitada para tareas más complejas debido a su simplicidad en comparación con arquitecturas más modernas.

### ALEXNET

La arquitectura de RNC AlexNet ganó el reto ILSVRC en 2012 con un margen muy amplio: consiguió una tasa de error del top cinco del 17%, mientras que el segundo mejor participante consiguió solo el 26%. Es similar a LeNet-5, pero mucho más grande y profunda, y fue la primera en apilar capas convolucionales unas directamente encima de otras, en vez de apilar una capa de pooling encima de cada capa convolucional.



*Arquitectura AlexNet*

Para reducir el sobreajuste, los autores utilizaron dos técnicas de regularización. En primer lugar, aplicaron dropout con una tasa de dropout del 50 % durante el entrenamiento a las salidas de las capas F9 y F10. En segundo lugar, llevaron a cabo un aumento de datos desplazando de manera aleatoria las imágenes de entrenamiento con distintas desviaciones, poniéndolas en horizontal y cambiando las condiciones de iluminación (este tema se trata en el último apartado del documento)

AlexNet utiliza también un paso de normalización inmediatamente después del paso ReLU de las capas C1 y C3, denominado "normalización de respuesta local" (LRN, local response normalization): las neuronas activadas con más fuerza inhiben otras neuronas situadas en la misma posición en mapas de características vecinos. Esta activación competitiva se ha observado en las neuronas biológicas. Esto impulsa a diferentes mapas de características a especializarse, separándolos y obligándolos a explorar un rango más amplio de características, mejorando, finalmente, la generalización.

### GOOGLENET

La arquitectura GoogLeNet ganó el reto ILSVRC de 2014 llevando la tasa de error a menos del 7%. Este excelente rendimiento se debió, en gran parte, al hecho de que la red era mucho más profunda que las RNC anteriores. Eso fue posible gracias a unas subredes denominadas "módulos lnception", que permiten a GoogLeNet utilizar parámetros de manera mucho más eficiente que las arquitecturas anteriores: en realidad, GoogLeNet tiene un número de parámetros 10 veces menor que AlexNet.

Diagrama

Descripción generada automáticamente

*Módulo lnception*

La notación "3 x 3 + 1(S)" significa que la capa utiliza un kernel de 3 x 3, un paso de avance de 1 y relleno "same". La señal de entrada se introduce primero en cuatro capas diferentes en paralelo. Todas las capas convolucionales utilizan la función de activación ReLU. **Las capas convolucionales superiores utilizan tamaños de kernel diferentes (1 x 1, 3 x 3 y 5 x 5)**, lo que permite capturar patrones a diferentes escalas. Todas las capas utilizan un paso de avance de 1 y relleno "same" (incluso la capa max pooling), así que todas las salidas tienen la misma altura y anchura que sus entradas. Eso hace que sea posible concatenar todas las entradas a lo largo de la dimensión de profundidad en la capa de concatenación de profundidad (depth concat) final (es decir, apilar los mapas de características de las cuatro capas convolucionales superiores). Puede implementarse utilizando la capa Concatenate de Keras, utilizando la configuración predeterminada axis=-1.

Quizá te preguntes por qué los **módulos lnception tienen capas convolucionales con kernels de 1 x 1**. Sin duda, estas capas no pueden capturar ninguna característica porque solo se fijan en un píxel cada vez, ¿no? En realidad, estas capas sirven para tres fines:

* Aunque no pueden capturar patrones espaciales, pueden capturar patrones a lo largo de la dimensión de profundidad (es decir, a través de los canales).
* Cada par de capas convolucionales ([1 x 1, 3 x 3] y [1 x 1, 5 x 5]) actúa como una sola capa convolucional potente, capaz de capturar patrones más complejos.

Puedes imaginar el módulo lnception completo como una capa convolucional hipervitaminada, capaz de producir mapas de características que capturan patrones complejos a varias escalas.

Veamos ahora la arquitectura de la RNC GoogLeNet.

Diagrama

Descripción generada automáticamente

*Arquitectura GoogleNet.*

El número de mapas de características que produce cada capa convolucional y cada capa de pooling aparece antes que el tamaño del kernel. La arquitectura es tan profunda que hay que representarla en tres columnas, pero, en realidad, GoogLeNet es una pila alta, que incluye nueve módulos lnception (las cajas con una peonza). Los seis números de los módulos lnception representan el número de mapas de características generados por cada capa convolucional del módulo. Todas las capas convolucionales utilizan la función de activación ReLU.

Vamos a examinar esta red:

* Las dos primeras capas dividen la altura y la anchura de la imagen entre 4 (así que su área se divide entre 16) para reducir la carga computacional. La primera capa utiliza un tamaño de kernel grande, 7 x 7.
* Después, la capa de normalización de respuesta local garantiza que las capas anteriores aprenden una amplia variedad de características.
* Siguen dos capas convolucionales. Puedes imaginar este par como una sola capa convolucional más inteligente.
* De nuevo, una capa de normalización de respuesta local garantiza que las capas anteriores capturan una amplia variedad de patrones.
* A continuación, una capa max pooling reduce la altura y la anchura de la imagen dividiéndolas entre 2, otra vez para acelerar los cálculos.
* Luego viene la "columna vertebral" (backbone) de la CNN: una pila alta de nueve módulos lnception, intercalados con un par de capas max pooling para reducir la dimensionalidad y dar más velocidad a la red.
* Después, la capa global average pooling genera como salida la media de cada mapa de características: esto deja fuera cualquier información espacial restante, lo cual está bien porque, en este punto, no queda mucha información espacial. En realidad, se suele esperar que las imágenes de entrada de GoogleNet sean de 224 x 224 píxeles, por lo que, después de 5 capas max pooling, cada una de las cuales divide la altura y la anchura entre 2, los mapas de características se reducen a 7 x 7. Además, se trata de una tarea de clasificación, no de localización, así que no importa dónde está el objeto. Gracias a la reducción de dimensionalidad que aporta esta capa, no es necesario tener muchas capas completamente conectadas en la parte superior de la CNN (como en AlexNet), lo que reduce de forma considerable el número de parámetros de la red y limita el riesgo de sobreajuste.
* Las últimas capas son evidentes: dropout para regularización, una capa completamente conectada con 1.000 unidades (porque hay 1.000 clases) y una función de activación softmax para generar como salida probabilidades de clases estimadas.

La arquitectura GoogLeNet original incluía dos clasificadores auxiliares colocados encima del tercer y el sexto módulo lnception. Ambos estaban compuestos por una capa average pooling, una capa convolucional, dos capas completamente conectadas y una capa de activación softmax. Durante el entrenamiento, su pérdida (reducida en un 70%) se añadía a la pérdida global. El objetivo era luchar contra el problema del desvanecimiento de gradientes y regularizar la red, pero, más adelante, se demostró que su efecto era relativamente pequeño. Después, los investigadores de Google propusieron muchas variantes de la arquitectura GoogLeNet, incluidas lnception-v3 e lnception-v4, que usaban módulos lnception un poco

diferentes para conseguir un rendimiento aún mejor.

### VGGNET

La subcampeona del reto ILSVRC de 2014 fue VGGNet, **es conocida por su estructura profunda y uniforme. Presenta configuraciones como VGG16 y VGG19 (que llegan llegar a un total de 16 o 19 capas), con capas convolucionales seguidas de capas de pooling y capas totalmente conectadas**. Utiliza extensivamente convoluciones 3x3 y la función de activación ReLU. A pesar de su simplicidad, VGGNet ha influido en el diseño de CNNs.

### RESNET

ResNet o Redes Residuales, ganó el reto ILSVRC de 2015 con una tasa de error de menos del 3,6%. La variante ganadora utilizó una CNN extremadamente profunda, compuesta por 152 capas (otras variantes tenían 34, 50 y 101capas). Se confirmó la tendencia general: los modelos estaban haciéndose cada vez más profundos y cada vez tienen menos parámetros. La clave para poder entrenar una red neuronal así es usar conexiones de salto (también llamadas conexiones de acceso directo): la señal que se introduce en una capa también se añade a la salida de una capa situada más arriba en la pila.

Cuando se entrena una red neuronal, el objetivo es hacer que modele una función objetivo h(x). Si añadimos la entrada x a la salida de la red (es decir, añadimos una conexión de salto), la red se verá obligada a modelar f(x) = h(x) - x en vez de h(x). Esto se denomina "aprendizaje residual" y acelera el aprendizaje.

Diagrama

Descripción generada automáticamente

*Aprendizaje residual*

Ahora, vamos a echar un vistazo a la arquitectura ResNet. Empieza y acaba igual que GoogleNet (pero sin capa dropout) y, en medio, no es más que una pila muy profunda de unidades residuales. **Cada unidad residual está compuesta por dos capas convolucionales, con normalización de lotes (BN) y activación ReLU, usando kernels de 3 x 3 y preservando las dimensiones espaciales** (paso de avance de 1, relleno "same").

**El número de mapas de características se duplica cada pocas unidades residuales**, **al tiempo que su altura y su anchura se reducen a la mitad** (usando una capa convolucional con paso de avance de 2). **Cuando ocurre esto, las entradas no pueden añadirse de forma directa a las salidas de las unidades residuales, porque no tienen la misma forma.** Para solucionar este problema, las entradas se pasan a través de una capa convolucional de 1 x 1 con paso de avance de 2 y el número adecuado de mapas de características de salida.

Diagrama

Descripción generada automáticamente

*Arquitectura ResNet*

Diagrama

Descripción generada automáticamente

*Conexión de salto al cambiar el tamaño y la profundidad de los mapas de características.*

Existen diferentes variaciones de la arquitectura, con diferentes números de capas. ResNet-34 es una ResNet con 34 capas (contando solo las capas convolucionales y la capa completamente conectada).

Las redes residuales más profundas, como ResNet-152, utilizan unidades residuales un poco diferentes. En vez de usar dos capas convolucionales 3 x 3 con, digamos, 256 mapas de características, utilizan tres capas convolucionales: primero, una capa convolucional de 1 x 1 con solo 64 mapas de características (cuatro veces menos), después, una capa de 3 x 3 con 64 mapas de características y, por último, otra capa convolucional de 1 x 1 con 256 mapas de características (64 multiplicado por 4) que restaura la profundidad original.

### XCEPTION

Otra variante de la arquitectura GoogleNet que merece la pena destacar es Xception (cuyo nombre viene de **Extreme lnception**). Al igual que lnception-v4, **fusiona las ideas de GoogleNet y ResNet, pero sustituye los módulos lnception por un tipo especial de capa denominada "capa convolucional separable por profundidad"**. Mientras que una capa convolucional corriente utiliza filtros que intentan capturar al mismo tiempo patrones espaciales (por ejemplo, un óvalo) y patrones de canales cruzados (por ejemplo, boca + nariz + ojos = cara), **una capa convolucional separable hace la suposición fuerte de que los patrones espaciales y los patrones de canales cruzados pueden modelarse por separado**. Así, se compone de dos partes: la primera parte aplica un solo filtro espacial a cada mapa de características de entrada y, después, la segunda parte busca exclusivamente patrones de canales cruzados; se trata solo de una capa convolucional corriente con filtros de 1 x 1.

Diagrama

Descripción generada automáticamente

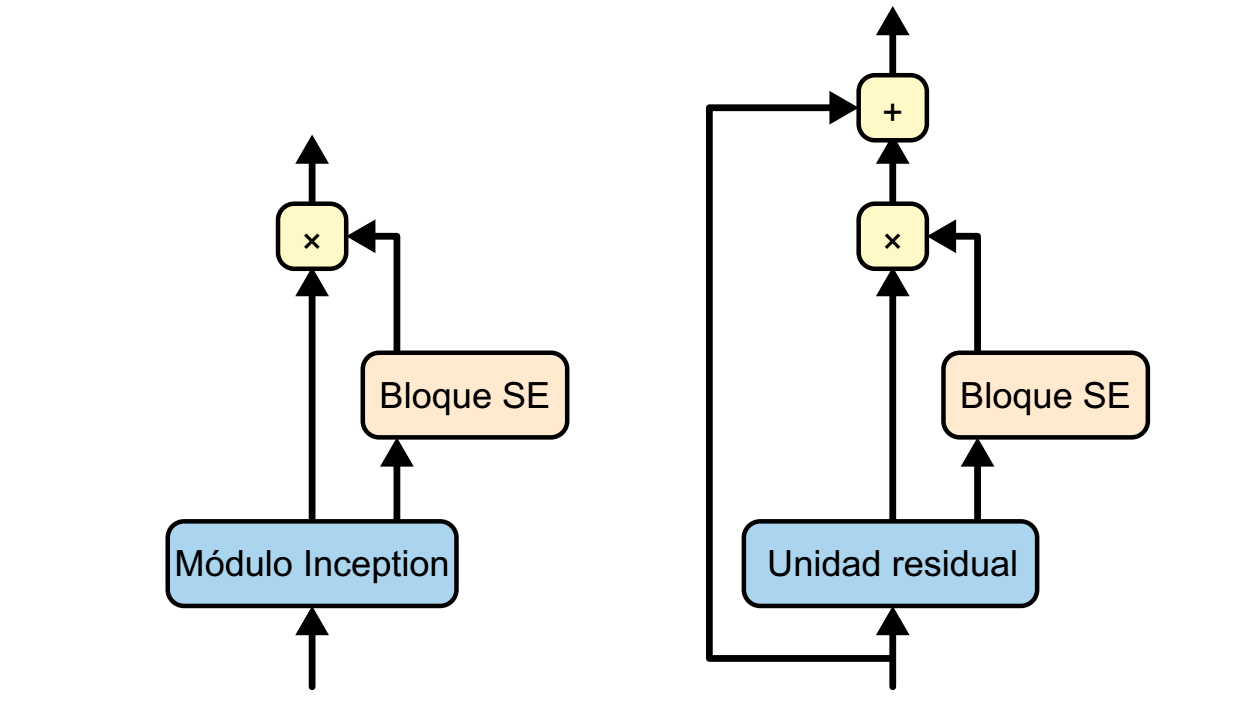
*Capa convolucional separable por profundidad.*

Puesto que las capas separables convolucionales solo tienen un filtro espacial por canal de entrada, hay que evitar utilizarlas después de capas con demasiados canales, como la capa de entrada. Por esa razón, la arquitectura Xception empieza con 2 capas convolucionales corrientes, pero, después, el resto de la arquitectura usa solo convoluciones separables (34 en total), además de algunas capas max pooling y las capas finales habituales (una capa global average pooling y una capa de salida densa).

Las capas convolucionales separables utilizan menos parámetros, menos memoria y menos cálculos que las capas convolucionales corrientes y, a menudo, tienen mejor rendimiento. Se pueden utilizar por defecto, salvo después de capas con pocos canales (como el canal de entrada). En Keras, usa SeparableConv2D en vez de Conv2D: es un reemplazo directo.

### SENET

SENet fue la ganadora del reto ILSVRC en 2017, con un 2,25% de error. **Esta arquitectura amplía arquitecturas existentes, como las redes lnception y las redes residuales, y mejora su rendimiento**. La mejora se debe al hecho de que una SENet **añade una red neuronal pequeña, llamada "bloque SE", a cada módulo lnception o unidad residual en la arquitectura original.**



*Módulo SE-lnception (izquierda) y unidad SE-ResNet (derecha)*

**Un bloque SE analiza la salida de la unidad a la que está unido, centrándose en exclusiva en la dimensión de profundidad (no busca ningún patrón espacial) y aprende qué características suelen ser más activas juntas**. Después, utiliza esta información para volver a calibrar los mapas de características. Por ejemplo, un bloque SE puede aprender que las bocas, las narices y los ojos suelen aparecen juntos en las imágenes: si ves una boca y una nariz, deberías esperar ver también los ojos. Así pues, si el bloque ve una activación fuerte en los mapas de características de boca y nariz, pero solo una activación débil en el mapa de características de ojos, dará impulso al mapa de características de los ojos (de forma más exacta, reducirá mapas de características irrelevantes). Si los ojos se confunden de algún modo con otra cosa, esta nueva calibración del mapa de características ayudará a resolver la ambigüedad.

Diagrama

Descripción generada automáticamente

*Un bloque SE lleva a cabo una nueva calibración del mapa de características.*

Un bloque SE está compuesto por solo tres capas: una capa global average pooling, una capa densa oculta que utiliza la función de activación ReLU y una capa de salida densa que utiliza la función de activación sigmoide.

Diagrama

Descripción generada automáticamente

*Arquitectura del bloque SE.*

Como antes, la capa global average pooling calcula la activación media para cada mapa de características: por ejemplo, si su entrada contiene 246 mapas de características, generará como salida 256 números representando el nivel total de respuesta para cada filtro. La siguiente capa es donde tiene lugar la "reducción": esta capa tiene bastante menos de 256 neuronas; por lo general, 16 veces menos que el número de mapas de características (por ejemplo, 16 neuronas), así que los 256 números se comprimen en un vector pequeño (por ejemplo, 16 dimensiones). Por último, la capa de salida genera un vector con la nueva calibración que contiene un número por mapa de características (por ejemplo, 256), cada uno entre 0 y 1. Entonces, los mapas de características se multiplican por este vector con la nueva calibración, de manera que las características irrelevantes (con una puntuación baja en la nueva calibración) se reducen, mientras que las características relevantes (con una puntuación cercana a 1 en la nueva calibración) se dejan sin tocar.

### OTRAS ARQUITECTURAS DESTACABLES

Hay muchas otras arquitecturas de RNC para explorar: RedNeXt

* **ResNeXt**: ResNeXt mejora las unidades residuales en ResNet. Mientras que las unidades residuales en los mejores modelos ResNet contienen solo 3 capas convolucionales cada una, las unidades residuales de ResNeXt están compuestas por muchas pilas paralelas (por ejemplo, 32 pilas), con 3 capas convolucionales cada una. Sin embargo, las dos primeras capas de cada pila solo utilizan unos pocos filtros (por ejemplo, solo cuatro), así que el número total de parámetros sigue siendo el mismo que en ResNet. Después, las salidas de todas las pilas se suman y el resultado se pasa a la siguiente unidad residual (junto con la conexión de salto).
* **DenseNet**: una DenseNet está compuesta de varios bloques densos, cada uno formado por unas pocas capas convolucionales densamente conectadas. Esta arquitectura consiguió una exactitud excelente al tiempo que utilizaba pocos parámetros en comparación. ¿Qué significa "densamente conectada"? La salida de cada capa se introduce como entrada a todas las capas después de ella dentro del mismo bloque. Por ejemplo, la capa 4 de un bloque toma como entrada la concatenación por profundidad de las salidas de las capas 1, 2 y 3 en ese bloque. Los bloques densos están separados para algunas capas de transición.
* **MobileNet**: las MobileNets son modelos optimizados diseñados para ser ligeros y rápidos, lo que hace que sean populares para aplicaciones para móviles y web. Se basan en capas convolucionales separables por profundidad, como Xception. Los autores propusieron distintas variantes, sacrificando un poco de exactitud a cambio de modelos más rápidos y pequeños.
* **CSPNet**: una CSPNet es similar a una DenseNet, pero parte de la entrada de cada bloque denso se concatena directamente con la salida de ese bloque, sin pasar por el bloque.

:

* **EfficientNet**: los autores propusieron un método para escalar de manera eficiente cualquier RNC, al aumentar conjuntamente la profundidad (número de capas), anchura (número de filtros por capa) y resolución (tamaño de la imagen de entrada) de una manera sistematizada. Esto se denomina escalado compuesto. Utilizaron la búsqueda de arquitecturas neuronales para encontrar una buena arquitectura para una versión reducida de lmageNet (con menos imágenes y más pequeñas) y, después, utilizaron el escalado compuesto para crear versiones cada vez más grandes de esta arquitectura. Cuando aparecieron los modelos EfficientNet, superaron ampliamente el rendimiento de todos los modelos existentes, en todos los presupuestos de computación, siguen estando entre los mejores modelos que hay hoy en día.
* …

## ELEGIR LA ARQUITECTURA DE CNN ADECUADA

Con tantas arquitecturas de RNC, ¿cómo elegir la que es mejor para nuestro proyecto?

Bueno, depende de lo que te importe más: ¿Exactitud? ¿Tamaño del modelo (por ejemplo, para el despliegue en un dispositivo móvil)? ¿Velocidad de inferencia en la CPU? ¿En la GPU? A continuación, se muestra una lista de aspectos a tener en cuenta:

1. **Tamaño y complejidad del conjunto de datos:** la cantidad de datos disponibles y la complejidad de las imágenes influyen en la elección. **Grandes conjuntos de datos** y datos complejos pueden beneficiarse de **arquitecturas más profundas como ResNet o EfficientNet**.
2. **Naturaleza del problema:** la naturaleza del problema, ya sea **clasificación**, **detección de objetos, segmentación**, etc., puede influir en la elección de la arquitectura. Algunas arquitecturas son más adecuadas para tareas específicas. Por ejemplo, para tareas de **clasificación ResNet, EfficienteNet** o **DenseNet** pueden ser las más efectivas. En el caso de **detección de objetos**, podrían considerarse **Faster R-CNN, YOLO o SSD**. Si hablamos de **segmentación**, podrían ser **U-Net** o **SegNet**.
3. **Recursos computacionales disponibles:** la **capacidad computacional** que tienes disponible también es un factor crucial, la familia **EfficientNet** puede ser una buena opción. Arquitecturas más profundas tienden a requerir más recursos para entrenamiento e inferencia.
4. **Transfer Learning: la capacidad de realizar transfer learning con modelos preentrenados** puede ser un factor determinante. Algunas arquitecturas, como **VGGNet y ResNet**, son comúnmente utilizadas para transfer learning.
5. **Rendimiento y eficiencia:** evalúa el **rendimiento y la eficiencia** de la arquitectura en términos de **precisión y velocidad de entrenamiento/inferencia**. Algunas arquitecturas pueden ser más eficientes en ciertos aspectos. Por ejemplo, en este caso, podrían ser **EfficienNet y MobileNetV3**.
6. **Innovaciones recientes:** considera las innovaciones recientes en arquitecturas, ya que hay desarrollos constantes en este campo. Modelos como **EfficientNet han introducido mejoras significativas en eficiencia.**
7. **Simplicidad vs. Profundidad:** evalúa la complejidad de la arquitectura en **relación con la complejidad del problema**. **Problemas más simples** pueden beneficiarse de arquitecturas más simples como **VGGNet**, mientras que **problemas complejos** pueden requerir modelos **más profundos como ResNet**.
8. **Requerimientos de tiempo real:** Si tienes **restricciones de tiempo real**, debes considerar arquitecturas que **equilibren eficacia y velocidad**, como **MobileNet**.

### ARQUITECTURAS CON KERAS

**Para cada modelo Keras ofrece una clase**, tenemos la lista completa en <https://keras.io/api/applications>.

Tabla

Descripción generada automáticamente

…

Para cada modelo, la tabla muestra el nombre de la clase de Keras, el tamaño del modelo en MB, la precisión de la validación del top 1 (porcentaje de veces que la clase predicha con la mayor probabilidad es la correcta) y el top 5 (porcentaje de veces que la clase verdadera se encuentra dentro de las cinco principales predicciones realizadas por el modelo) en el conjunto de datos lmageNet, el número de parámetros (millones) y el tiempo de inferencia (tiempo que tarda el modelo en hacer una predicción) en la CPU y la GPU en milisegundos, utilizando lotes de 32 imágenes en un hardware de una potencia razonable.

## IMPLEMENTAR UNA CNN RESNET-34 USANDO KERAS

La mayoría de las arquitecturas de RNC descritas hasta ahora pueden implementarse de forma bastante natural usando Keras.

Vamos a implementar una ResNet-34 desde cero con Keras: es una ResNet con 34 capas (contando solo las capas convolucionales y la capa completamente conectada) que contiene 3 unidades residuales que generan como salida 64 mapas de características, 4 unidades residuales con 128 mapas, 6 unidades residuales con 256 mapas y 3 unidades residuales con 512 mapas.

Primero, vamos a crear una capa ResiduaUnit:

Diagrama

Descripción generada automáticamente

*ResidualUnit*

DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel\_size=3, strides=1, padding="same",

kernel\_initializer="he\_normal", use\_bias=False)

class ResidualUnit(tf.keras.layers.Layer):

def \_\_init\_\_(self, filters, strides=1, activation="relu", \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.activation = tf.keras.activations.get(activation)

self.main\_layers = [

DefaultConv2D(filters, strides=strides),

tf.keras.layers.BatchNormalization(),

self.activation,

DefaultConv2D(filters),

tf.keras.layers.BatchNormalization()

]

self.skip\_layers = []

if strides > 1:

self.skip\_layers = [

DefaultConv2D(filters, kernel\_size=1, strides=strides),

tf.keras.layers.BatchNormalization()

]

def call(self, inputs):

Z = inputs

for layer in self.main\_layers:

Z = layer(Z)

skip\_Z = inputs

for layer in self.skip\_layers:

skip\_Z = layer(skip\_Z)

return self.activation(Z + skip\_Z)

En el constructor, creamos todas las capas que vamos a necesitar: las capas principales son las de la derecha del diagrama y las capas de salto son las de la izquierda (solo son necesarias si el paso de avance es mayor que 1). Después, en el método call(), hacemos que las entradas pasen por las capas principales y las capas de salto (si las hay) y sumamos ambas salidas y aplicamos la función de activación.

Ahora, podemos crear una ResNet-34 usando un modelo Sequential, puesto que, en realidad, es solo una larga secuencia de capas; podemos tratar cada unidad residual como una sola capa ahora que tenemos la clase ResidualUnit.

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(224, 224, 3)),

DefaultConv2D(64, kernel\_size=7, strides=2),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Activation("relu"),

tf.keras.layers.MaxPool2D(pool\_size=3, strides=2, padding="same"),

])

prev\_filters = 64

for filters in [64] \* 3 + [128] \* 4 + [256] \* 6 + [512] \* 3:

strides = 1 if filters == prev\_filters else 2

model.add(ResidualUnit(filters, strides=strides))

prev\_filters = filters

model.add(tf.keras.layers.GlobalAvgPool2D())

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(10, activation="softmax"))

La única parte complicada en este código es el bucle que añade las capas ResidualUnit al modelo: como hemos explicado antes, las 3 primeras unidades residuales tienen 64 filtros, las 4 unidades residuales siguientes tienen 128 filtros, y así sucesivamente. En cada iteración, debemos configurar el paso de avance como 1 cuando el número de filtros es el mismo que en la unidad residual anterior, o, si no, lo configuramos como 2; después, añadimos la ResidualUnit y, por último, actualizamos prev\_filters.

Implementar las otras arquitecturas de RNC es un poco más largo, pero no es mucho más difícil. Sin embargo, Keras incluye muchas de esas arquitecturas integradas, así que ¿por qué no usarlas?

## UTILIZAR MODELOS PREENTRENADOS DESDE KERAS

En general, no tenemos que implementar modelos estándar de forma manual, puesto que hay redes preentrenadas listas para usar con una sola línea de código en el paquete tf.keras.applications.

Podemos cargar el modelo ResNet-50, preentrenado con lmageNet, con la siguiente línea de código:

model = tf.keras.applications.ResNet50(weights="imagenet")

Esto creará un modelo ResNet-50 y descargará sus pesos preentrenados con el conjunto de datos lmageNet. Para utilizarlo, primero tenemos que asegurarnos de **que las imágenes tienen el tamaño adecuado**. Un modelo ResNet-50 espera imágenes de 224 x 224 píxeles (otros modelos pueden esperar otros tamaños, como 299 x 299), así que vamos a usar la **capa Resizing de Keras para redimensionar dos imágenes de muestra:**

images = load\_sample\_images()["images"]

images\_resized = tf.keras.layers.Resizing(height=224, width=224,crop\_to\_aspect\_ratio=True)(images)

Los modelos preentrenados asumen que las imágenes se preprocesan de una manera específica. En algunos casos, pueden esperar que las entradas se escalen de 0 a 1, o de -1 a 1, etc. Cada modelo ofrece una función preprocess\_input() que podemos utilizar para preprocesar nuestras imágenes. Estas funciones asumen que los valores de los píxeles originales van de 0 a 255:

inputs = tf.keras.applications.resnet50.preprocess\_input(images\_resized)

Y\_proba = model.predict(inputs)

Y\_proba.shape

(2, 1000)

Como siempre, la salida Y\_proba es una matriz con una fila por imagen y una columna por clase (en este caso, hay 1.000 clases). **Para mostrar las K mejores predicciones**, **incluyendo el nombre de clase y la probabilidad estimada de cada clase predicha**, utilizamos la **función decode\_predictions**():

top\_K = tf.keras.applications.resnet50.decode\_predictions(Y\_proba, top=3)

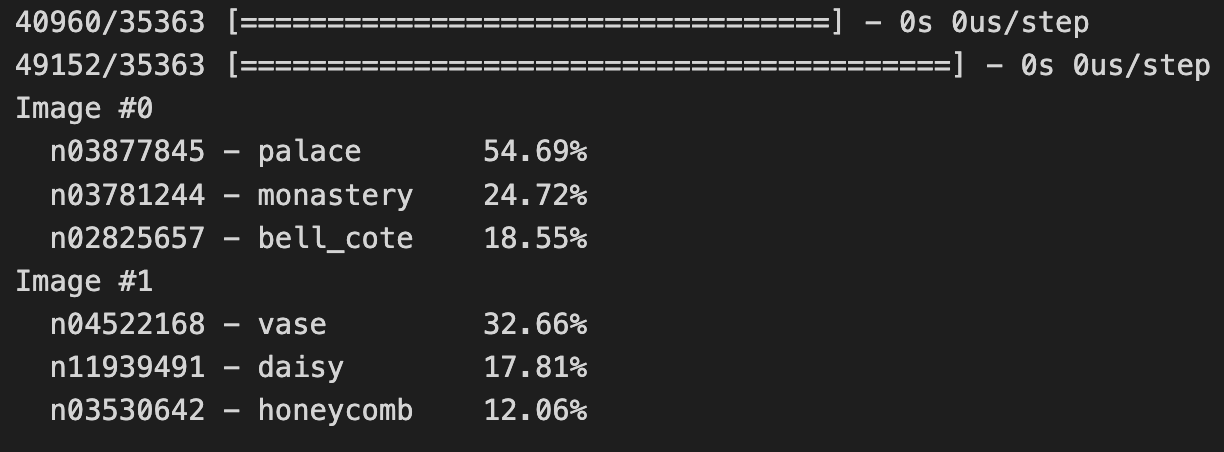
for image\_index in range(len(images)):

print(f"Image #{image\_index}")

for class\_id, name, y\_proba in top\_K[image\_index]:

print(f" {class\_id} - {name:12s} {y\_proba:.2%}")

La salida tiene este aspecto:



Las clases correctas son palace (palacio) y dahlia (dalia), así que el modelo es correcto para la primera imagen, pero erróneo para la segunda. Sin embargo, eso se debe a que dahlia no es una de las 1.000 clases de lmageNet. Teniendo eso en mente, vase (jarrón) es una suposición razonable (¿quizá la flor está en un jarrón?), y daisy (margarita) tampoco es una mala opción.

Es muy fácil crear un clasificador de imágenes bastante bueno usando un modelo preentrenado.

Pero ¿qué pasa si quieres utilizar un clasificador de imágenes para clases de imágenes que no forman parte de lmageNet? En ese caso, aún puedes beneficiarte de los modelos predeterminados si los utilizas para realizar un aprendizaje por transferencia.

## MODELOS PREENTRENADOS PARA APRENDIZAJE POR TRANSFERENCIA

Si quieres crear un clasificador de imágenes, pero no tienes suficientes datos para entrenarlo desde cero, a menudo es buena idea reutilizar las capas inferiores de un modelo predeterminado. Por ejemplo, vamos a entrenar un modelo para clasificar imágenes de flores, reutilizando un modelo Xception preentrenado.

Primero, vamos a cargar el conjunto de datos de las flores utilizando TensorFlow Datasets:

dataset, info = tfds.load("tf\_flowers", as\_supervised=True, with\_info=True)

dataset\_size = info.splits["train"].num\_examples

>>>3670

class\_names = info.features["label"].names

>>>['dandelion', 'daisy', 'tulips', 'sunflowers', 'roses']

n\_classes = info.features["label"].num\_classes

>>>5

Podemos obtener información sobre el conjunto de datos configurando with\_info=True. Obtenemos el tamaño del conjunto de datos y los nombres de las clases. Por desgracia, solo hay un conjunto de datos "train" para el entrenamiento, no hay conjunto de prueba ni de validación, así que tenemos que dividir el conjunto de entrenamiento.

Vamos a llamar otra vez a tfds.load(), pero, esta vez, vamos a tomar el primer 10 % del conjunto de datos para las pruebas, el siguiente 15 % para la validación y el 75 % restante para el entrenamiento:

test\_set\_raw, valid\_set\_raw, train\_set\_raw = tfds.load(

"tf\_flowers",

split=["train[:10%]", "train[10%:25%]", "train[25%:]"],

as\_supervised=True)

Los tres conjuntos de datos contienen imágenes individuales y necesitamos agruparlos en lotes, pero, primero, tenemos que asegurarnos de que todos tienen el mismo tamaño, o la distribución en lotes fallará. Para esto, podemos utilizar una capa Resizing. También debemos llamar a la función tf.keras.applications.xception.preprocess\_input() para preprocesar las imágenes de manera apropiada para el modelo Xception. Por último, vamos a mezclar el conjunto de entrenamiento y utilizar la precarga (que mientras está procesando uno vaya cargando el siguiente en memoria):

batch\_size = 32

preprocess = tf.keras.Sequential([

tf.keras.layers.Resizing(height=224, width=224, crop\_to\_aspect\_ratio=True),

tf.keras.layers.Lambda(tf.keras.applications.xception.preprocess\_input)

])

train\_set = train\_set\_raw.map(lambda X, y: (preprocess(X), y))

train\_set = train\_set.shuffle(1000, seed=42).batch(batch\_size).prefetch(1)

valid\_set = valid\_set\_raw.map(lambda X, y: (preprocess(X), y)).batch(batch\_size)

test\_set = test\_set\_raw.map(lambda X, y: (preprocess(X), y)).batch(batch\_size)

Cada lote contiene 32 imágenes, todas ellas de 224 x 224 píxeles, con valores de píxel que van de -1 a 1.

Puesto que el conjunto de datos no es muy grande, ayudaría realizar un aumento de datos. Vamos a crear un modelo de aumento de datos que incrustaremos en nuestro modelo definitivo. Durante el entrenamiento, de forma aleatoria, volteará las imágenes horizontalmente, las rotará un poco y ajustará el contraste (es la secuencia para aplicarlo, pero no lo vamos a aplicar en este modelo):

data\_augmentation = tf.keras.Sequential([

tf.keras.layers.RandomFlip(mode="horizontal", seed=42),

tf.keras.layers.RandomRotation(factor=0.05, seed=42),

tf.keras.layers.RandomContrast(factor=0.2, seed=42)

])

Ahora, vamos a cargar un modelo Xception, preentrenado con lmageNet. **Excluimos la parte superior** de la red configurando **include\_top=False**. **Esto excluye la capa global average pooling y la capa de salida densa.** Después, añadimos nuestra propia capa global average pooling (introduciéndole la salida del modelo base), seguida de una capa de salida densa con una unidad por clase, usando la función de activación softmax. Por último, envolvemos todo esto en un Model de Keras:

base\_model = tf.keras.applications.xception.Xception(weights="imagenet", include\_top=False)

avg = tf.keras.layers.GlobalAveragePooling2D()(base\_model.output)

output = tf.keras.layers.Dense(n\_classes, activation="softmax")(avg)

model = tf.keras.Model(inputs=base\_model.input, outputs=output)

Suele ser buena idea congelar los pesos de las capas preentrenadas, al menos al principio del entrenamiento:

for layer in base\_model.layers:

layer.trainable = False

Por último, podemos compilar el modelo y empezar el entrenamiento:

optimizer = tf.keras.optimizers.SGD(learning\_rate=0.1, momentum=0.9)

model.compile(loss="sparse\_categorical\_crossentropy", optimizer=optimizer, metrics=["accuracy"])

history = model.fit(train\_set, validation\_data=valid\_set, epochs=3)

Después de entrenar el modelo durante unas pocas repeticiones, su exactitud de validación debería ser un poco superior al 80 % y, luego, dejar de mejorar. Eso significa que, ahora, las capas superiores están bastante bien entrenadas y estamos listos para descongelar algunas de las capas superiores del modelo base y, después, continuar el entrenamiento. Por ejemplo, vamos a descongelar las capas 56 y superiores (ese es el principio de la unidad residual 7 de 14):

for layer in base\_model.layers[56:]:

layer.trainable = True

Hay que compilar el modelo cada vez que se congelen o descongelen capas. Conviene utilizar una tasa de aprendizaje mucho más baja para evitar dañar los pesos preentrenados:

optimizer = tf.keras.optimizers.SGD(learning\_rate=0.01, momentum=0.9)

model.compile(loss="sparse\_categorical\_crossentropy", optimizer=optimizer, metrics=["accuracy"])

history = model.fit(train\_set, validation\_data=valid\_set, epochs=10)

Este modelo debería llegar a una exactitud de alrededor del 90% en el conjunto de prueba, en solo unos minutos de entrenamiento (con una GPU). Si ajustamos los hiperparámetros, bajamos la tasa de aprendizaje y lo entrenamos durante algo más de tiempo, debería ser capaz de alcanzar entre un 95% y un 97%.

Pero la visión por ordenador es más que la clasificación. Por ejemplo, ¿qué pasa si también quieres saber dónde está la flor en una fotografía?

## CAPAS DE PREPROCESAMIENTO

Las capas de preprocesamiento de Keras ofrecen una variedad de operaciones que se pueden aplicar a los datos antes de ser ingresados en un modelo de red neuronal. Estas capas **facilitan la integración del preprocesamiento de datos en los flujos de trabajo de desarrollo de modelos** y pueden ser parte de un modelo de Keras o usarse de forma independiente.

Entre las capas de preprocesamiento de Keras, podemos encontrar:

* **Preprocesamiento de texto**
  + **TextVectorization**: Convierte el texto en una representación numérica que una red neuronal puede procesar. Tokeniza el texto y convierte los tokens en índices numéricos.
* **Preprocesamiento de características numéricas**
  + **Normalization**: Normaliza las características numéricas (por ejemplo, escalando los datos a tener media 0 y varianza 1) para acelerar el aprendizaje.
  + **Discretization**: Convierte variables numéricas continuas en intervalos categóricos, lo que puede ser útil para modelos que trabajan mejor con características categóricas.
* **Preprocesamiento de características categóricas**
  + **CategoryEncoding**: Codifica las características categóricas como un tensor numérico, a menudo utilizando codificación one-hot o codificaciones densas.
  + **Hashing**: Utiliza una función de hashing para convertir entradas de tamaño variable en salidas de tamaño fijo, útil para el procesamiento de texto y categóricas.
  + **HashedCrossing**: Combina características y aplica una función de hashing, útil para crear interacciones de características.
  + **StringLookup**: Mapea cadenas a índices numéricos, similar a un codificador de etiquetas para datos de texto.
  + **IntegerLookup**: Similar a StringLookup, pero para mapear enteros a índices numéricos.
* **Preprocesamiento de imágenes**
  + **Resizing**: Cambia el tamaño de las imágenes a dimensiones específicas.
  + **Rescaling**: Ajusta los valores de los píxeles de las imágenes, normalmente para llevar los valores de píxeles al rango [0, 1].
  + **CenterCrop**: Recorta la parte central de las imágenes para lograr un tamaño específico.
  + **RandomCrop**: Recorta aleatoriamente una sección de las imágenes en cada lote.
  + **RandomFlip**: Voltea las imágenes horizontal o verticalmente de manera aleatoria.
  + **RandomTranslation**: Mueve las imágenes aleatoriamente en el eje vertical u horizontal.
  + **RandomRotation**: Rota las imágenes un número aleatorio de grados.
  + **RandomZoom**: Aplica un zoom aleatorio a las imágenes.
  + **RandomContrast**: Ajusta el contraste de las imágenes de manera aleatoria.
  + **RandomBrightness**: Ajusta el brillo de las imágenes de manera aleatoria.

Ejemplo de normalización de características numéricas:

# Suponiendo que `data` es un array de NumPy con datos de entrada

normalizer = Normalization(axis=-1)

normalizer.adapt(data)

# Ahora `normalizer` puede ser usado como la primera capa de un modelo

Ejemplo de aumento de imagen:

# Crear capas de aumento de imagen

random\_flip = RandomFlip("horizontal")

random\_rotation = RandomRotation(0.2)

# Estas capas se pueden incluir en un modelo secuencial

## AUMENTO DE DATOS

**El aumento de manera artificial de datos incrementa el tamaño del conjunto de entrenamiento al generar muchas variantes realistas de cada instancia de entrenamiento**. Eso reduce el sobreajuste, lo que lo convierte en una **técnica de regularización**.

Las instancias generadas deberían ser tan realistas como sea posible: añadir ruido blanco. sin más, no servirá: las modificaciones deberían poder aprenderse (el ruido blanco no puede). Por ejemplo, puedes desplazar, rotar y redimensionar cada imagen del conjunto de entrenamiento varias veces y añadir las imágenes resultantes a ese conjunto. Para ello, puedes utilizar **capas de procesamiento de datos de Keras (por ejemplo, RandomCrop, RandomRotation, etc.)**. **Eso obliga al modelo a ser más tolerante respecto a las variaciones en la posición, la orientación y el tamaño de los objetos en las imágenes**. Para producir un modelo que sea más tolerante respecto a las condiciones de iluminación, podemos generar de manera similar muchas imágenes con varios contrastes. En general, también podemos poner las imágenes en horizontal (salvo el texto y otros objetos asimétricos). Al combinar estas transformaciones, podemos aumentar mucho el tamaño del conjunto de entrenamiento.

Imagen que contiene foto, exterior, posando, cuarto

Descripción generada automáticamente

*Generación de nuevas instancias de entrenamiento a partir de otras existentes.*

El aumento de datos también resulta útil cuando tenemos un conjunto de datos desequilibrado: podemos utilizarlo para generar más muestras de las clases menos frecuentes. **Esto se denomina SMOTE (synthetic minority oversampling technique).**