Práctica 6
Redes Neuronales Convolucionales

TÉCNICAS ESTADÍSTICAS PARA EL APRENDIZAJE II

Máster Universitario en Estadística Computacional
y Ciencia de Datos para la Toma de Decisiones

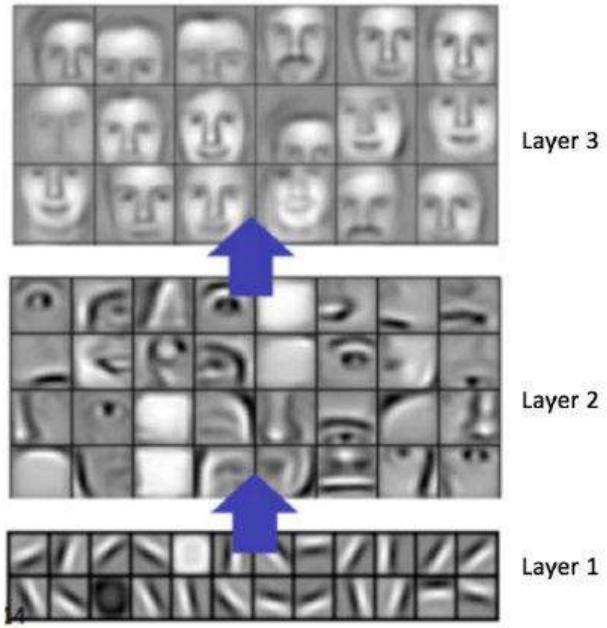
UNIVERSITAS
Miguel Hernández

Redes convolucionales o convolutivas
CNN [*Convolutional Neural Network*]

Empleadas en **visión artificial** para reconocer objetos en imágenes y localizarlos.

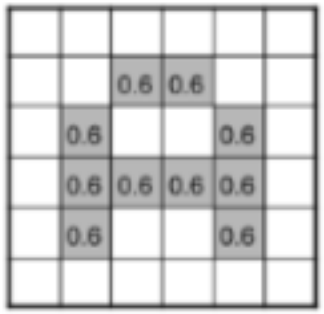
Son redes multicapa en las que algunas capas realizan una **operación de convolución** en lugar de la tradicional multiplicación matricial de entradas por pesos.

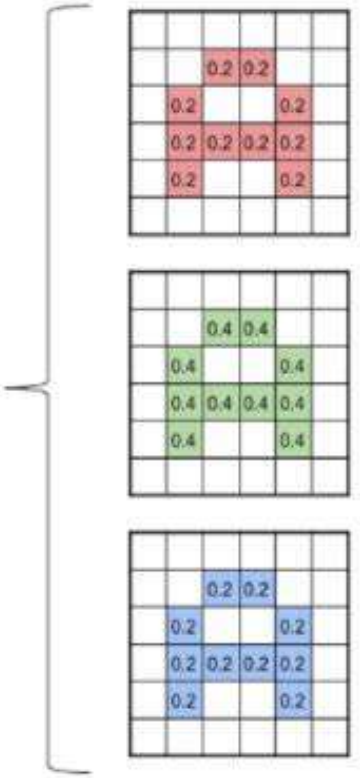
Las primeras capas tienden a aprender **características simples** (como bordes), mientras que **características más complejas** (formas básicas) se aprenden a medida que pasamos a capas más profundas.



El reconocimiento de imágenes comienza por la pixelización y asignación de intensidad.

Una imagen es una matriz de pixeles. El valor de los pixeles va de 0 a 255 y se normaliza para la red neuronal de 0 a 1:





Si la imagen es a color, estará compuesta de tres canales: rojo, verde y azul

Hay dos tipos de capas que definen a las redes neuronales convolucionales, que pueden expresarse como grupos de neuronas especializadas en dos operaciones: **convolución** y *pooling*.

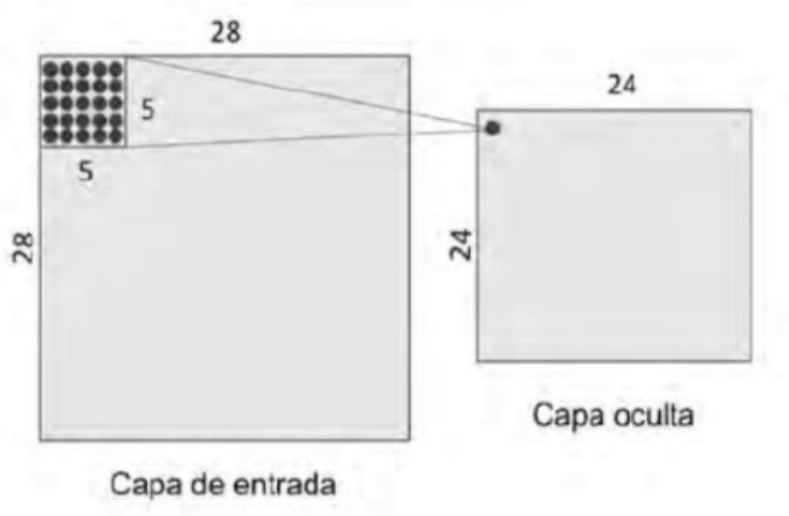
Operación de convolución

La diferencia fundamental entre una capa densamente conectada y una capa especializada en la operación de convolución (**capa convolucional**) es que la capa densa aprende patrones globales en su espacio global de entrada, mientras que la capa convolucional **aprende patrones locales** dentro de la imagen en pequeñas ventanas de dos dimensiones.

Además, las capas convolucionales pueden **aprender jerarquías espaciales de patrones**. Por ejemplo, una primera capa convolucional puede aprender elementos básicos como aristas, y una segunda capa convolucional puede aprender patrones compuestos de elementos básicos aprendidos en la capa anterior. Y así sucesivamente hasta ir aprendiendo patrones muy complejos.

A diferencia de las redes neuronales densamente conectadas, no **se conectan** todas **las neuronas de entrada con** todas las neuronas del primer nivel de neuronas ocultas; solo se hace por **pequeñas zonas localizadas** del espacio de las neuronas de entrada que almacenan los píxeles de la imagen.

Ejemplo: en el caso de MNIST o Fashion-MNIST, como entrada en nuestra red neuronal podemos imaginar un espacio de neuronas de dos dimensiones 28x28. Si utilizamos una ventana 5x5 podemos pensar que la ventana va recorriendo toda la capa de 28x28 que contiene la imagen:



A continuación, la ventana se desliza una posición hacia la derecha para *conectar* las 5x5 neuronas de la capa de entrada incluidas en esta ventana con la segunda neurona de la capa oculta. Y así, sucesivamente, va recorriendo todo el espacio de la capa de entrada, de izquierda a derecha y de arriba abajo.

Si tenemos una entrada de 28x28 píxeles y una ventana de 5x5, nos define un espacio de 24x24 neuronas en la primera capa oculta, debido a que la ventana solo se puede desplazar 23 neuronas hacia la derecha y 23 hacia abajo antes de chocar con el lado derecho (o inferior) de la imagen de entrada.

Aquí hemos supuesto que la ventana hace movimientos de avance de 1 píxel en cada paso, tanto en horizontal como en vertical, cuando empieza una nueva fila (en cada paso la nueva ventana se solapa con la anterior, excepto en esta línea de píxeles que hemos avanzado). No obstante,

- Se pueden usar **diferentes longitudes de pasos** de avance (el parámetro llamado ***stride***).

- Se puede aplicar una **técnica de relleno de ceros alrededor del margen de la imagen** para mejorar el resultado del barrido que se realiza con la ventana que se va deslizando. El parámetro para definir este relleno recibe el nombre de ***padding***.

Para *conectar* cada neurona de la capa oculta con las 25 neuronas que le corresponden de la capa de entrada usaremos un valor de sesgo b y una matriz de pesos W de tamaño 5x5, que llamaremos **filtro** (***kernel*** o ***filter***).

Lo particular y muy importante de las redes convolucionales es que se usa el mismo filtro (la misma matriz W de pesos y el mismo sesgo b) para todas las neuronas de la capa oculta (en nuestro caso para las 24x24=576 neuronas de la primera capa oculta), pasando de 14400 parámetros que tendrían que ser ajustados (5x5x24x24) a 25 (5x5) parámetros más los sesgos b.

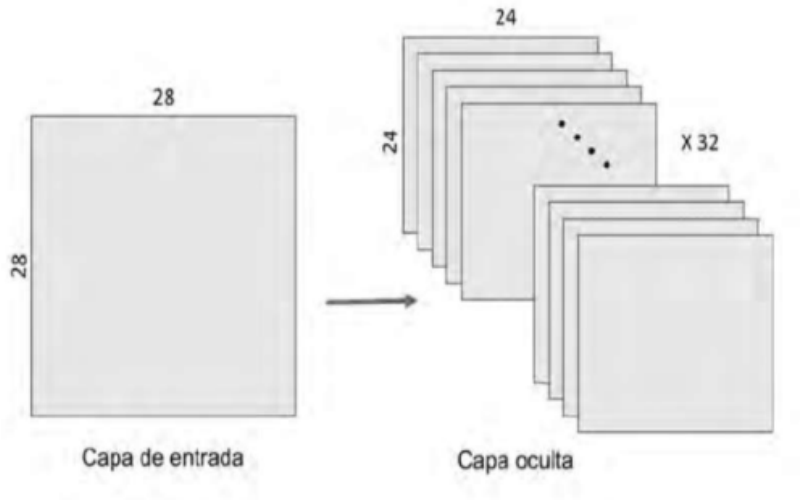
En resumen, una convolución es el tratamiento de una matriz de entrada por otra que llamamos filtro.

Un filtro definido por una matriz W y un sesgo b **solo permite detectar una característica concreta en una imagen**. Por tanto, para poder realizar el reconocimiento de imágenes se propone usar varios filtros a la vez, uno para cada característica que queramos detectar. Por eso **una capa convolucional** completa en una red neuronal convolucional **incluye varios filtros**.

En general, las capas convolucionales operan sobre lo que se conoce como **mapas de características** (*feature maps* en inglés), con dos ejes espaciales de altura y anchura (*height* y *width*), además de un eje de canal (*channels*) también llamado profundidad (*depth*). Para una imagen de color RGB, la dimensión del eje *depth* es 3, pues la imagen tiene tres canales: rojo, verde y azul. Para una imagen en blanco y negro, la dimensión del eje *depth* es 1 (nivel de gris).

En nuestro ejemplo, si utilizáramos 32 filtros, donde cada filtro se define con una matriz W de pesos compartida de 5x5 y un sesgo b:

Capa de entrada
(28, 28, 1)
(*height, width, depth*)

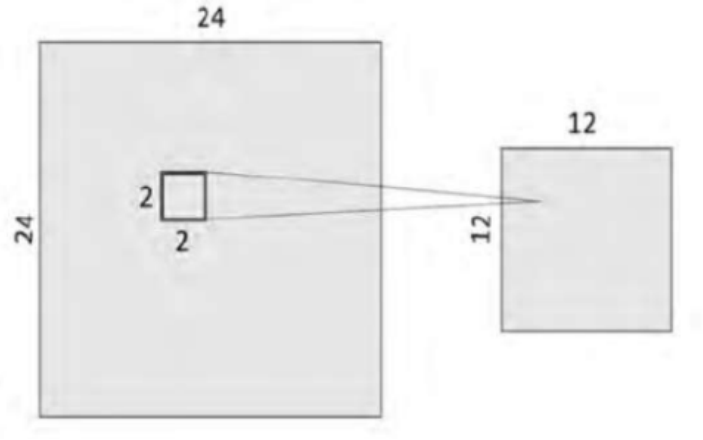


Capa oculta
(24, 24, 32)
(*height, width, depth*)

Operación de *pooling*

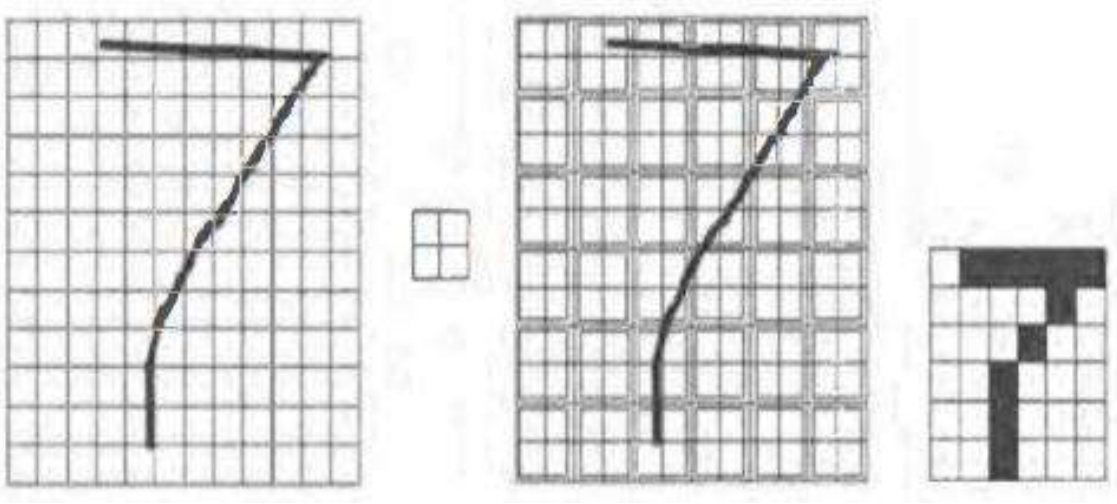
Las redes neuronales convolucionales acompañan a la capa de convolución con unas **capas de *pooling*** (agrupación), que suelen ser aplicadas inmediatamente después de las capas convolucionales y que, intuitivamente, hacen una simplificación de la información recogida por la capa convolucional y crean una versión condensada de la información contenida en esta capa.

Hay varias maneras de condensar la información, pero la más habitual es la conocida como ***max-pooling***, que se queda con el valor máximo de los que hay en una ventana de entrada de tamaño, por ejemplo, 2x2:



La transformación de *pooling* mantiene la relación espacial.

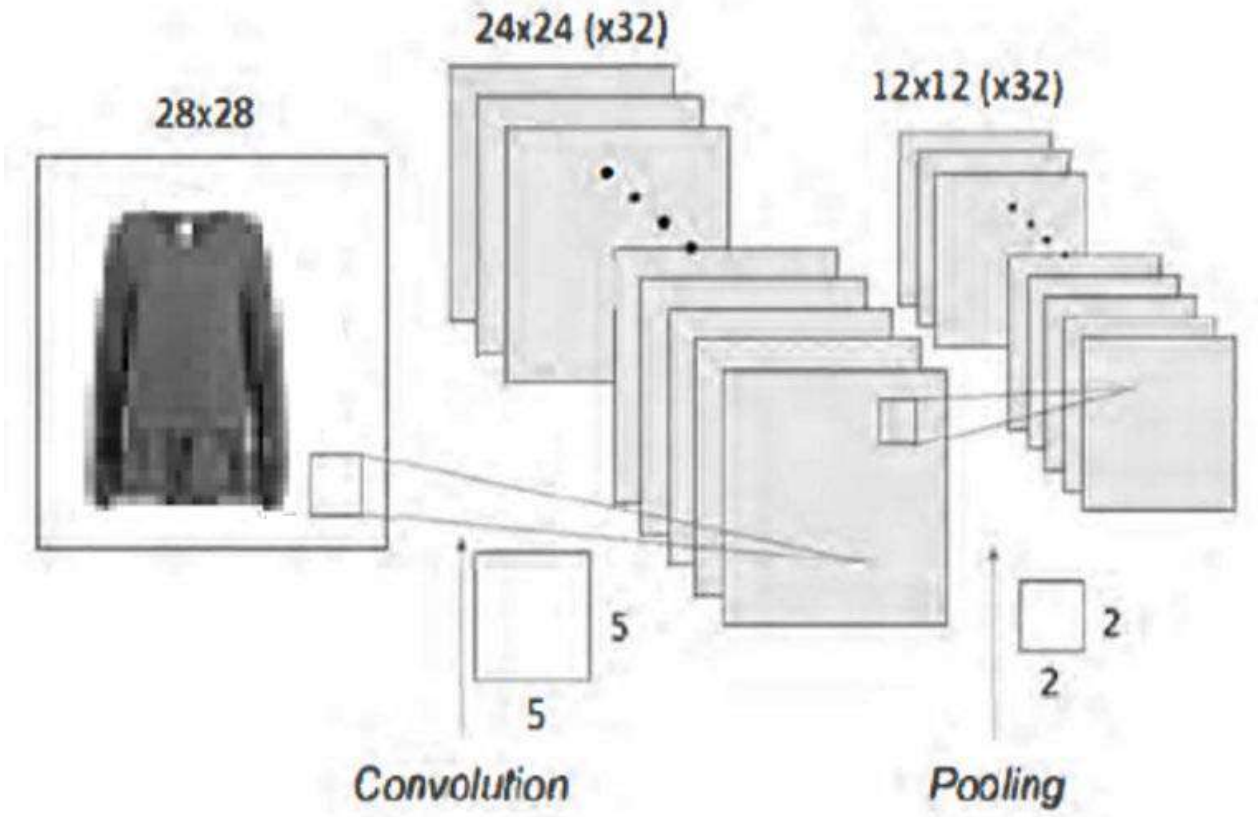
Ejemplo: supongo la siguiente matriz 12x12 donde tenemos representado un 7 y aplicamos una operación de *max-pooling* con una ventana 2x2, con lo que obtenemos una matriz 6x6 donde se mantiene la representación del número 7.



Las **capas de *pooling*** se indican en **Keras** de la siguiente forma:
MaxPooling2D(pool_size=(2,2))

https://keras.io/api/layers/pooling_layers/max_pooling2d/

Como hemos comentado, la capa convolucional alberga más de un filtro y, por tanto, como aplicamos el *max-pooling* a cada uno de estos filtros separadamente, la capa de *pooling* contendrá tantos filtros de *pooling* como filtros convolucionales había.



Los **principales hiperparámetros** de las **capas convolucionales** son:

- Número de filtros (habitualmente 32 o 64, *filters*)
- Tamaño de la ventana del filtro (habitualmente 3x3 o 5x5, *kernel_size* = (*window_height*, *window_width*))
- Relleno (*padding*).
- Tamaño del paso de avance (*strides*).

Las **capas convolucionales** se indican en **Keras** de la siguiente forma:

Conv2D(filters, (window_height, window_width),
padding="valid", strides=(1,1))

https://keras.io/api/layers/convolution_layers/convolution2d/

Padding

Si queremos obtener una salida de las mismas dimensiones de entrada; podemos usar para ello el hiperparámetro *padding* en las capas convolucionales. Con *padding* podemos agregar ceros (*zero-padding* en inglés) alrededor de las imágenes de entrada antes de hacer deslizar la ventana por ella.

En Keras, este relleno con ceros en la capa Conv2D se configura con el **argumento *padding***, que puede tener dos valores: *same*, que implica que se añadan tantas filas y columnas de ceros como sea necesario para que la salida tenga la misma dimensión que la entrada, y *valid*, que implica no hacer *padding* (que es el valor por defecto de este argumento en Keras).

Stride

stride nos indica el número de pasos en que se mueve la ventada de los filtros. En nuestro ejemplo el *stride* era de 1, el valor por defecto.

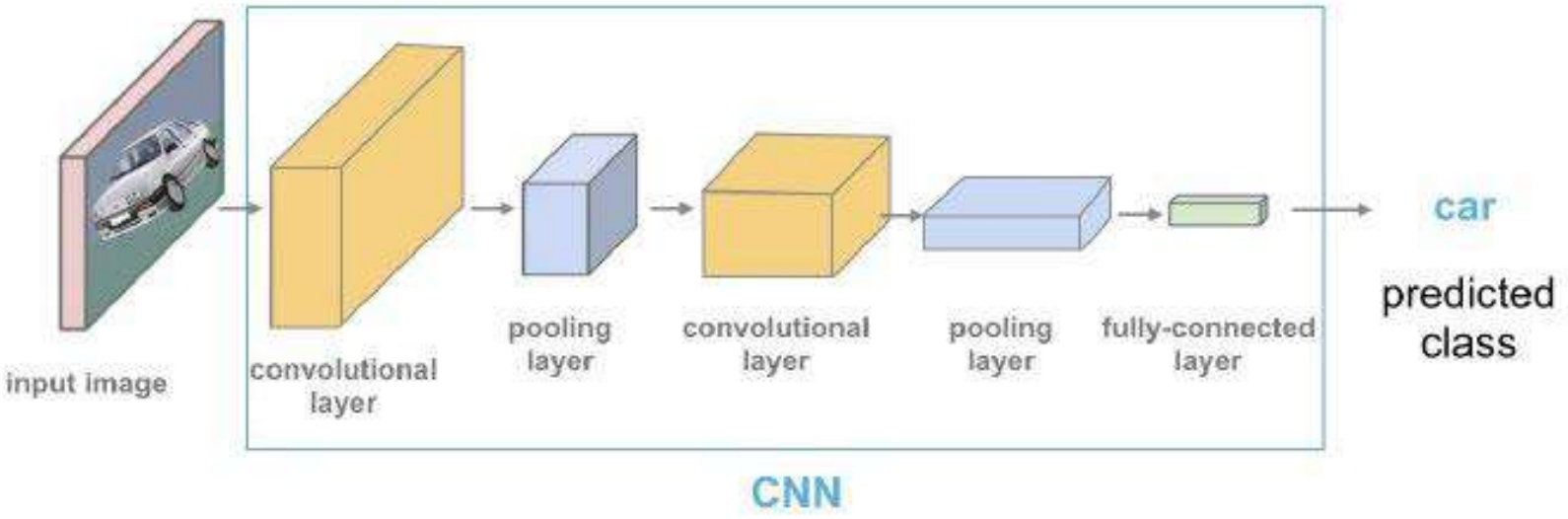
Valores de *stride* grandes hacen decrecer el tamaño de la información que pasará a la siguiente capa.

En la práctica, los *strides* son raramente utilizados en convolucionales para reducir los tamaños; para ello se usan las operaciones de *pooling*.

En Keras, el *stride* en la capa conv2D se configura con el **argumento *stride*** que tiene por defecto el valor *strides*=(1, 1) que indica por separado el avance en las dos dimensiones.

Antes de la capa de salida de la red, por ejemplo de una capa *softmax* en un problema de clasificación multiclase que hará la clasificación final, hay que *aplanar* los datos, lo que en Keras se hace con la **capa *Flatten*.**

Además, una red convolucional puede tener **más de una etapa Conv2D + MaxPooling2D**, lo que permite tanto aumentar la capacidad de la red como reducir aún más el tamaño de los mapas de características, de modo que no sean demasiado grandes cuando lleguemos a la capa *Flatten*.

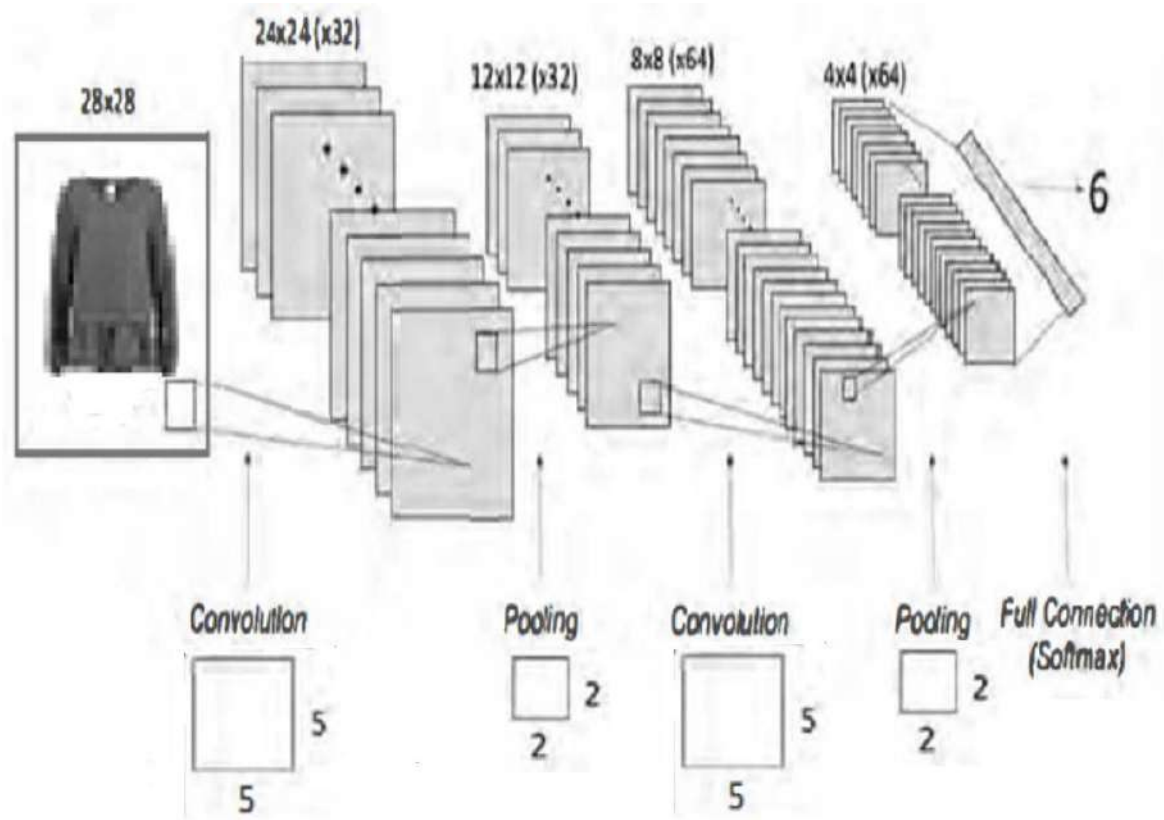


En esta práctica vamos a trabajar con el **conjunto de datos Fashion–MNIST** haciendo uso de redes neuronales convolucionales.

Fashion-MNIST es un conjunto de datos de las imágenes de los artículos de Zalando, una tienda de moda online alemana especializada en venta de ropa y zapatos. El conjunto de datos contiene 70000 imágenes en escala de grises en 10 categorías. Las imágenes muestran prendas individuales de ropa en baja resolución (28x28 píxeles). Se usan 60000 imágenes para entrenar la red y 10000 imágenes para evaluar la precisión con la que la red aprende a clasificar las imágenes.

Las etiquetas son una matriz de enteros, que van de 0 a 9, y corresponden a la clase de ropa que representa la imagen.

Construiremos la siguiente red neuronal convolucional:




```
modelo = Sequential([
        Input(shape=(28,28,1)),
        Conv2D(32, (5, 5), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(64, (5, 5), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(10, activation='softmax')
])
```

La profundidad de los mapas de características suele aumentar progresivamente en la red (aquí de 32 a 64), mientras que sus dimensiones tienden a reducirse a medida que nos adentramos en las capas ocultas de la red neuronal (aquí de 24x24 a 4x4). Esto se observa en casi todas las redes convolucionales.

Despues de *Flatten* se podrían añadir mas capas densas, por ejemplo

```
Dense(64, activation="relu")
```

Tener que entrenar un modelo de clasificación de imágenes con muy **pocos datos** es una situación común.

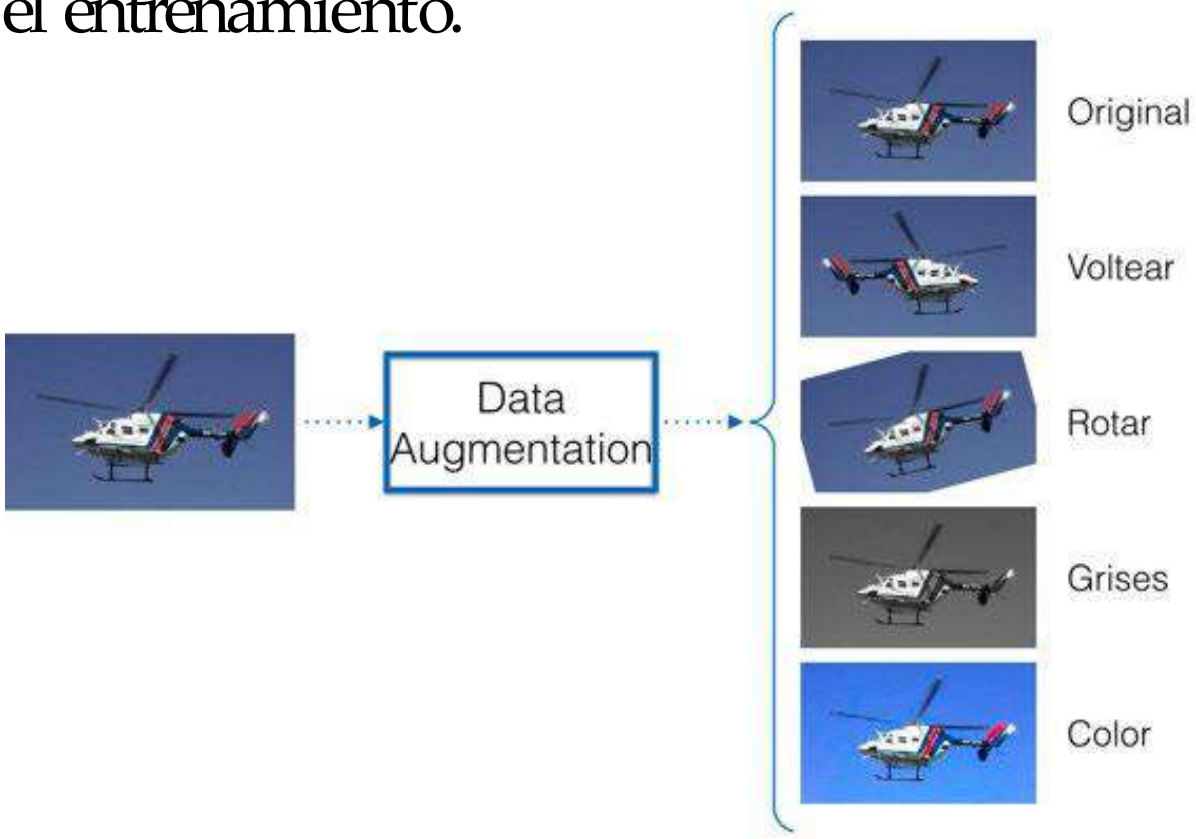
En este tipo de situaciones el principal problema será el sobreajuste, ya que, al disponer de pocos datos, las arquitecturas suelen tender a conseguir muy buenos resultados para el entrenamiento pero no tan buenos para las muestras de validación y test.

Una posibilidad para disminuir este problema es aplicar aumento de datos (*Data Augmentation*) generando más datos de entrenamiento a partir de nuestros datos disponibles.

En el caso de las imágenes, esto se consigue aplicando una serie de **transformaciones aleatorias a la imagen** que producen nuevas imágenes de aspecto creíble, de modo que, en el momento del entrenamiento, nuestro modelo nunca verá exactamente la misma imagen. Esta idea simple, pero potente, ayuda a exponer al modelo a más aspectos de los datos y a generalizar mejor.

Data Augmentation

Al aplicar *Data Augmentation* es importante tener en cuenta el contexto del conjunto de datos de entrenamiento y conocer el dominio del problema, para no generar imágenes que nunca podrían encontrarse en realidad ya que de esta manera estaríamos empeorando el entrenamiento.



Transfer Learning

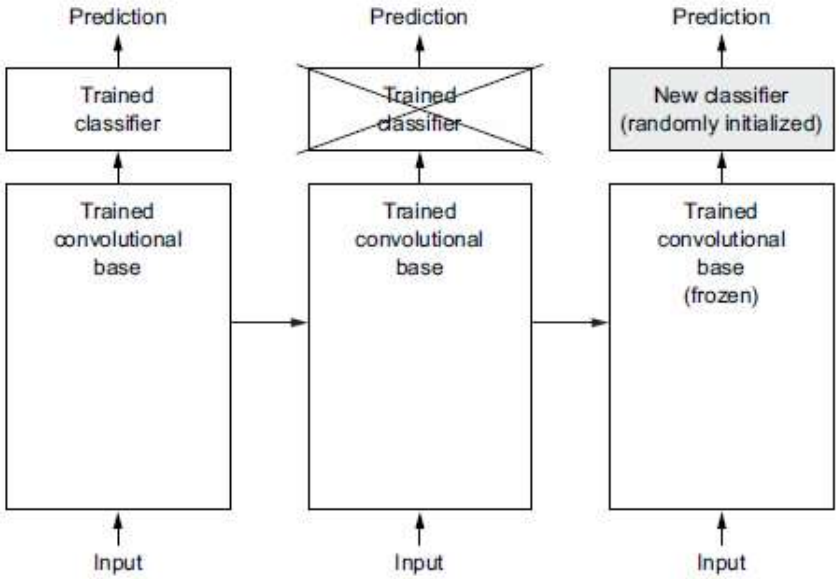
En vez de entrenar una red neuronal desde cero, que implica disponer de una gran cantidad de datos y puede requerir mucho tiempo de computación, podemos **partir de una red preentrenada**.

Es decir, podemos descargar un modelo ya entrenado previamente en un gran conjunto de datos y usar sus parámetros (miles o millones) como punto de partida para posteriormente continuar entrenando el modelo con el conjunto de datos (más pequeño) que tengamos para una tarea determinada.

Recordemos que en una **red neuronal convolucional** cada capa va aprendiendo diferentes niveles de abstracción siendo las primeras capas las encargadas de aprender características más genéricas, por lo que podemos **volver a usar** fácilmente esas **primeras capas**, ya que las características aprendidas son aplicables a otros problemas.

Transfer Learning

A la base convolucional que ya se encuentra entrenada le añadimos las capas finales que se requieran para hacer, por ejemplo, un clasificador a nuestra medida del problema que estamos tratando. Este nuevo clasificador sí que requiere entrenamiento y, en general, los pesos de sus parámetros se inicializarán de forma aleatoria.



Keras nos permite aplicar esta técnica de una manera muy fácil y con pocas líneas de código, indicando qué capas son entrenables (*trainable layers*) y qué capas no (*frozen layers*).

Transfer Learning

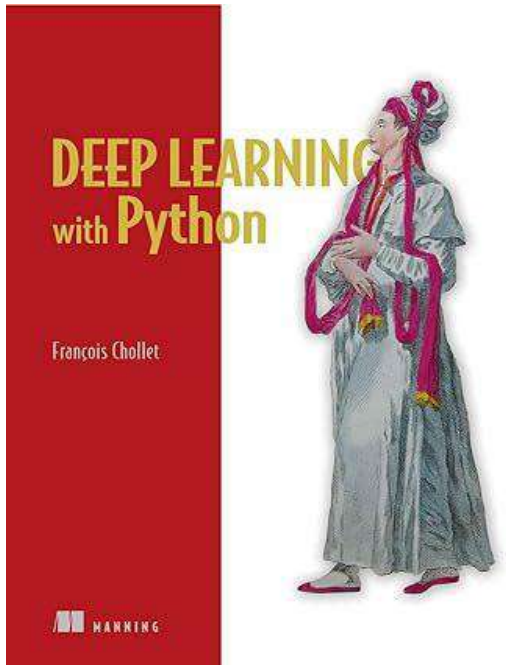
Por ejemplo, un modelo muy popular es la **red neuronal VGG16**, que es una red de 16 capas propuesta por Karen Simonyan y Andrew Zisserman en su artículo '*Very Deep Convolutional Networks for Large-Scale Image Recognition*'.

VGG16, que está disponible en Keras, se ha entrenado con un conjunto de datos de ImageNet, que dispone de 1,4 millones de imágenes en 1000 clases diferentes.

Supongamos que queremos resolver un problema de clasificación de imágenes de perros y gatos y disponemos solo de unos cientos de imágenes de perros y gatos. Aplicar *Data Augmentation* podría funcionar pero, puesto que ImageNet contiene muchas clases de animales, incluidas diferentes especies de gatos y perros, podríamos esperar un mejor rendimiento en el problema de clasificación de perros y gatos haciendo uso de la red neuronal VGG16.

Las técnicas de *Data Augmentation* y *Transfer Learning*, entre otras, están detalladas con ejemplos prácticos en el libro ***Deep Learning with Python***, en concreto en el capítulo 8 "*Introduction to deep learning for computer vision*".

Además, podemos encontrar técnicas más avanzadas en el capítulo 9 "*Advanced deep learning for computer vision*"



Ejercicio 1.

Utiliza otras configuraciones y verifica si la red utilizada en la práctica se comporta mejor.

Por ejemplo, puedes probar a cambiar el tamaño de la venta, el número de filtros, añadir capas densas, cambiar el optimizador, probar con un número de épocas diferente, utilizar una tasa de aprendizaje adaptativa, añadir técnicas de regularización…

Evalúa tu nuevo modelo en el conjunto de test y guarda el modelo con la función *save()* de Keras.