4. REDES NEURONALES CONVOLUCIONALES

En teoría, una red neuronal artificial con una sola capa oculta como las estudiadas en el capítulo anterior puede aproximar o modelar arbitrariamente bien una función continua de *n* variables reales, siempre y cuando tenga un número suficiente de unidades o neuronas a las cuales se les haya asignado una ponderación adecuada (Hornik, 1989). Es decir que en principio una red neuronal poco profunda, pero muy amplia puede modelar cualquier función por compleja que sea (Csáji, 2001).

Sin embargo, una red neuronal de dichas características tenderá a sobre ajustarse rápidamente a los datos de entrenamiento. Entre mayor sea la amplitud de la red (mayor número de neuronas en la capa oculta), mayor será la capacidad de memorización de la red, por lo cual se comportará muy bien con los datos de entrenamiento, pero no muy bien con datos que no haya visto, como los datos de validación o prueba.

Como soluciones a este problema de sobreajuste, se podría pensar en aplicar alguna de las reglas para reducción de varianza, como aumentar el número de ejemplos de entrenamiento (que no en todos los casos es fácil de realizar), o también aumentar la complejidad del modelo.

Para aumentar la complejidad del modelo, sin aumentar el número de neuronas de la capa oculta, la opción sería aumentar el número de capas ocultas del modelo. Esto permitiría a la red aprender a caracterizar los datos a diferentes niveles de abstracción, con estructuras más diversas. Precisamente el incremento en la complejidad (profundidad) ayuda a que el modelo en este caso generalice el comportamiento de los datos de entrada, en lugar de aprenderlos de memoria. A su vez, tener un modelo con una mayor profundidad puede reducir la amplitud del modelo, lo que equivale a un menor número de neuronas en cada capa oculta. La importancia de esto último radica en que el número de parámetros de la red puede reducirse, acelerando su entrenamiento. En conclusión, un balance entre profundidad y amplitud (priorizando profundidad contra amplitud) puede ser provechoso en cuanto a evitar un sobreajuste excesivo y controlar el número de parámetros de la red.

Prohibida su reproducción o difusión total o parcial | © 2023 |

De cualquier forma, es importante considerar aquí que las redes tipo perceptrón multicapa procesan datos que estén estructurados como un vector. En caso de alimentar una red de este tipo con datos de una dimensión mayor, como imágenes, será necesario aplanar los datos para adaptarlos a la capa de entrada de la red (por ejemplo, mediante una capa *Flatten* en *Keras/tensorflow*). Respecto a la forma de procesar imágenes con este tipo de redes, se pueden precisar dos observaciones importantes. La primera tiene que ver con luego de aplanar los datos, la información espacial o de textura de la imagen se pierde, al igual que su información espectral o de color. La segunda tiene que ver con el costo computacional, asociado al número de neuronas que utilizan las capas totalmente interconectadas de estas arquitecturas. A partir de estas dos observaciones se ha propuesto como alternativa el uso de redes neuronales convolucionales, que se describen a continuación.

Operaciones de correlación cruzada y convolución en 2D

Como se comentó anteriormente, una red basada en capas totalmente interconectadas se caracteriza por una gran cantidad de parámetros entrenables, dado que el número de parámetros entre capas se determina por el producto entre los tamaños (o número de neuronas) de las dos capas. De aquí que, si el número de parámetros de la red es considerablemente alto, su aprendizaje puede resultar inviable o requerir una gran cantidad de datos.

Como alternativa, se tiene el uso de capas convolucionales que procesan la información de entrada como si de un filtro espacial se tratara y mantienen la estructura de la información de entrada (por ejemplo, sin aplanar los datos), con dos objetivos en mente. El primero consiste en que la información de entrada se procese por pequeñas áreas o vecindades (principio de localización), mientras que el segundo involucra que todas las zonas o áreas de la imagen sean tratadas de la misma forma (invariancia a traslaciones). Por ejemplo, si en una imagen se desea identificar un determinado objeto, la red debe ser de capaz de identificarlo independientemente de la posición o tamaño que tenga.

Partiendo de la estructura de una capa totalmente interconectada ($A=XW+b$, Ecuación (49)), estudiada en el capítulo anterior, se procederá a definir una capa convolucional. En este caso, para cumplir los dos criterios anteriores, la imagen de entrada X se procesaría en una vecindad alrededor de un píxel (principio de localización) y dicha vecindad se desplazaría en toda la imagen a lo largo de filas y columnas (invarianza a traslaciones). A su vez, la operación en cada región o

Prohibida su reproducción o difusión total o parcial | © 2023 |

vecindad alrededor de un píxel se aplicaría con ciertos valores o pesos que corresponden a los valores del filtro (K), sumados a valores de ajuste o *bias* (b). De esta forma, el valor de salida para cada píxel o valor de entrada procesado en una posición específica (i,j) corresponde a:

$$[\boldsymbol{A}]_{\boldsymbol{i},\boldsymbol{j}} = \sum_{-m}^{m} \sum_{-n}^{n} [K]_{mnij} \, [\boldsymbol{X}]_{i+m,j+n} + [b]_{ij} \tag{71}$$

Donde la dimensión de la ventana a procesar es ($2m+1 \times 2n+1$). Sin embargo, si se desea que todas las zonas de la imagen se procesen de la misma forma, tanto el *bias* como los valores de ponderación del filtro o *kernel* (K) deberían ser iguales para toda la imagen. De esta forma, la ecuación se simplifica de la siguiente forma:

$$[\boldsymbol{A}]_{\boldsymbol{i},\boldsymbol{j}} = \sum_{-m}^{m} \sum_{-n}^{n} [K]_{mn} \, [\boldsymbol{X}]_{i+m,j+n} + b \tag{72}$$

Lo anterior significa que mediante una operación de este tipo es posible que un modelo aprenda patrones que estén presentes en cualquier parte de los datos de entrada, lo que no sucede con operaciones tipo FC. Esta característica hace que las redes convolucionales funcionen mejor en el modelamiento de imágenes respecto a arquitecturas basadas en perceptrón multicapa. Para dar cumplimento al principio de localización, la zona procesar, es decir, las dimensiones de la ventana ($2m+1 \times 2n+1$) se tienen que limitar a un valor pequeño en comparación con las dimensiones de los datos de entrada. Es típico utilizar en redes convolucionales recientes, dimensiones de filtro de (3×3) o (5×5). De esta forma, los valores que debe aprender la red se limitan a los valores del filtro, y su número es reducido en comparación con el número de parámetros que se tienen para una capa FC.

Pero ¿por qué se habla de capas convolucionales? Recordando la Ecuación de convolución discreta entre dos funciones que se estudia en un curso de señales, se tiene que:

$$y[i,j] = f[i,j] * g[i,j] = \sum_{-\infty}^{\infty} \sum_{-\infty}^{\infty} f[a,b] \, g[i-a, j-b] \tag{73}$$

Prohibida su reproducción o difusión total o parcial | © 2023 |

Si comparamos la Ecuación (72) con la de convolución en 2D, vemos que tienen una estructura similar. Sin embargo, existen dos diferencias. La primera es la presencia del *bias* como parámetro de ajuste del aprendizaje, y la segunda es que no se invierte la segunda señal. Esto último implica, que la operación mostrada en la Ecuación (72), no corresponde estrictamente a una operación de convolución, sino a una operación de correlación cruzada. Más allá de esto, en la literatura se popularizó el nombre de redes convolucionales, que hoy en día se mantiene.

Para ilustrar la operación de correlación cruzada (es decir, la operación que realizan las capas convolucionales), se creará en *python* la siguiente matriz de entrada:

```
import numpy as np
import cv2
Entrada = np.array([
    [0.0, 5.0, 0.0, 0.0, 9.0, 4.0, 0.0],
    [1.0, 6.0, 0.0, 0.0, 8.0, 3.0, 0.0],
    [2.0, 7.0, 0.0, 0.0, 7.0, 2.0, 0.0],
    [3.0, 8.0, 0.0, 0.0, 6.0, 1.0, 0.0],
    [4.0, 9.0, 0.0, 0.0, 5.0, 0.0, 0.0]
])
Entrada
```

```
array([[0., 5., 0., 0., 9., 4., 0.],
       [1., 6., 0., 0., 8., 3., 0.],
       [2., 7., 0., 0., 7., 2., 0.],
       [3., 8., 0., 0., 6., 1., 0.],
       [4., 9., 0., 0., 5., 0., 0.]])
```

Y se operará con el siguiente filtro o *kernel*:

```
kernel = np.array([
    [1.0, -2.0, 3.0],
    [4.0, -5.0, 6.0],
    [-7.0, 8.0, -9.0],
    ])
kernel
```


Prohibida su reproducción o difusión total o parcial | © 2023 |

```
array([[ 1., -2.,  3.],
       [ 4., -5.,  6.],
       [-7.,  8., -9.]])
```

Como ejemplo se muestra la operación de los datos del recuadro rojo, con los datos del *kernel*, multiplicando elemento por elemento y sumando sus resultados. Es decir, 0*1 + 5*(-2) + 0*3 + 1*4 + 6*(-5) + 0*6 + 2*(-7) + 7*(8) + 0*(-9) = 6. Para operar sobre los bordes se realiza una operación conocida como *padding*, que se explicará más adelante. El resultado de la operación completa se muestra a continuación.

```
Salida = cv2.filter2D(src=Entrada, ddepth=-
1, kernel=kernel)
Salida
```

```
array([[-16.,   5., -16.,   6.,   9., -14.,   4.],
       [-21.,   6., -20.,  12.,  10., -15.,  14.],
       [-22.,   5., -22.,  12.,   9., -14.,  16.],
       [-23.,   4., -24.,  12.,   8., -13.,  18.],
       [ -8.,   1., -12.,  -6.,   5., -10., -12.]])
```

En conclusión, la convolución realiza un producto punto entre el filtro y la imagen de entrada en una posición específica, sumando el resultado para obtener un único valor. Luego desplaza el filtro una posición y repite el mismo procedimiento. Para ilustrar un ejemplo de convolución para una imagen, se utilizarán tres tipos de filtros que permiten extraer los bordes de una imagen. Particularmente, los bordes horizontales (0º), bordes verticales (90º) y los bordes diagonales (45º). Los filtros se han definido manualmente de la siguiente forma:

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

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

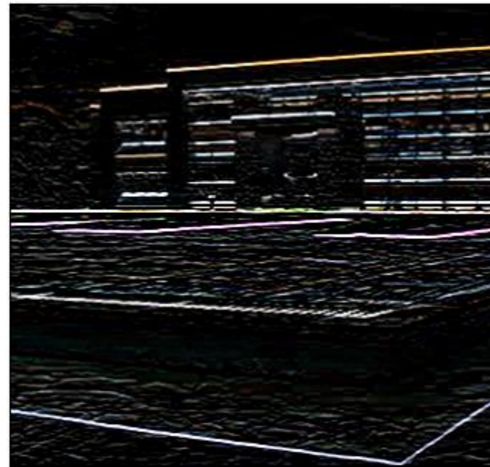

Prohibida su reproducción o difusión total o parcial | © 2023 |

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

La imagen de prueba y las tres imágenes filtradas se muestran a continuación:

```
image = cv2.imread("umng.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

horizontal = cv2.filter2D(image, -
1, filtro_horizontal)
vertical = cv2.filter2D(image, -
1, filtro_vertical)
diagonal_edges2 = cv2.filter2D(image, -
1, filtro_diagonal)
```

	
Imagen original	Imagen filtrada (bordes horizontales)

Prohibida su reproducción o difusión total o parcial | © 2023 |

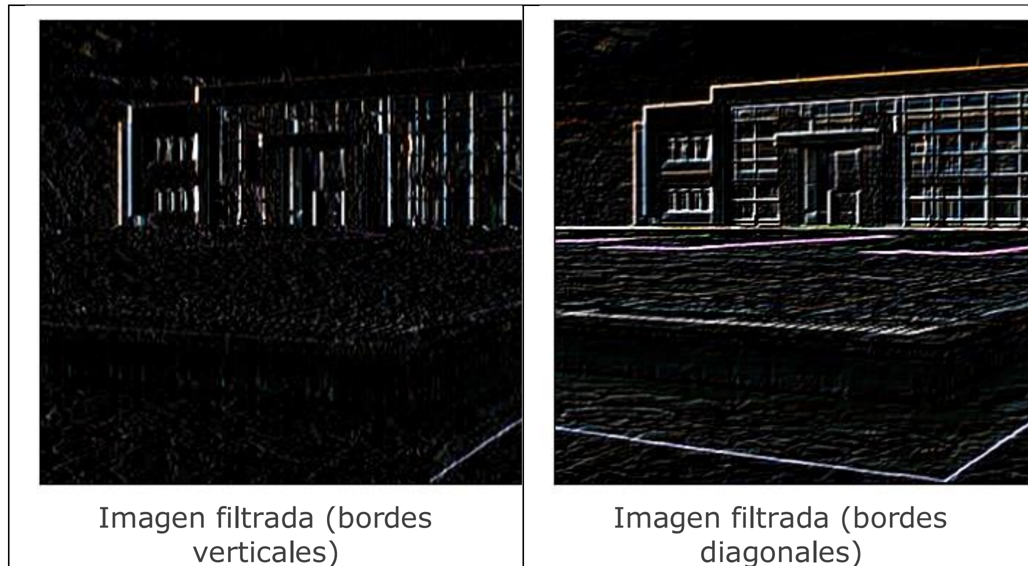


Imagen filtrada (bordes verticales)	Imagen filtrada (bordes diagonales)

Figura 23. Ejemplo de aplicación de filtros espaciales en una imagen con máscaras predefinidas.

El ejemplo anterior involucra el uso de tres filtros pre-diseñados para extraer una característica específica. En la práctica, para capas convolucionales la idea es utilizar de manera simultánea más de un filtro, de tal forma que cada uno de ellos extraiga características específicas. Además, el valor que tendrán los coeficientes del filtro será aprendido durante el entrenamiento de la red, por lo cual no es necesario diseñarlos previamente, pero tampoco se tiene la posibilidad de definir sus características específicas. Para que la red aprenda los coeficientes del filtro, se realiza un procedimiento como el que se ha explicado en capítulos anteriores, es decir, se construye la capa convolucional, se inicializan los valores del filtro, se define una función de pérdida sobre la cual se calcula el gradiente y se actualizan los valores del filtro.

Para crear una capa convolucional en *frameworks* como *tensorflow*, los principales argumentos que hay especificar al instanciar la clase serán el número de filtros, el tamaño del *kernel* o filtro (vecindad), el algoritmo de inicialización y dos características adicionales: *strides* y *padding* que se explicarán a continuación.

```
# https://keras.io/api/layers/convolution_layers/convolution2d/
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
```


Prohibida su reproducción o difusión total o parcial | © 2023 |

```
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs)
```

En el caso de no especificar algunos argumentos, la herramienta tomará su valor predeterminado. Para agregar una capa convolucional a un modelo secuencial en keras/tensorflow, se realiza de forma similar a la adición de una capa Densa. Es importante tener en cuenta la estructura de los datos que ingresarán a la capa convolucional, ya que, si se utiliza una convolución en 2D, los datos ingresarán con estructura de filas y columnas, por lo cual no es necesario realizar un aplanamiento (*flatten*) previo.

```
model = Sequential([Conv2D(32, (3, 3),
                           activation= 'relu',
                           input_shape=(28, 28, 1)
                           ),
                    MaxPooling2D((2, 2)),
                    Flatten(),
                    Dense(10, activation='softmax')])
```

Al ser la primera capa del modelo, por cuestiones prácticas en este ejemplo se ha incluido la dimensión de los datos de entrada. Posterior a la capa convolucional se ha incluido una capa de *pooling*, que se explicará en este mismo capítulo. Luego de las capas convolucionales, es común que el modelo incluya capas Densas, por loscual, antes de dichas capas se incluye una capa *Flatten*.

Prohibida su reproducción o difusión total o parcial | © 2023 |

Padding y *Stride*

Como se comentó anteriormente, para poder realizar la convolución entre el filtro y la imagen de entrada, es necesario realizar una operación adicional, de tal forma que los pixeles de los bordes puedan coincidir con el píxel central del *kernel*, ya que es común que los filtros en capas convolucionales sean de dimensión impar (1×1, 3×3, 5×5, etc.). Para esto, se agregan píxeles de relleno alrededor del límite de la imagen de entrada, lo que se conoce como *padding*. Para completar estos valores, es posible agregar, por ejemplo, ceros, o repetir el mismo valor de los pixeles del borde. Tomando el ejemplo anterior, la imagen con *padding* agregando ceros quedaría de la siguiente forma:

0	5	0	0	9	4	0
1	6	0	0	8	3	0
2	7	0	0	7	2	0
3	8	0	0	6	1	0
4	9	0	0	5	0	0

0	0	0	0	0	0	0	0	0
0	0	5	0	0	9	4	0	0
0	1	6	0	0	8	3	0	0
0	2	7	0	0	7	2	0	0
0	3	8	0	0	6	1	0	0
0	4	9	0	0	5	0	0	0
0	0	0	0	0	0	0	0	0

Figura 24. Ejemplo de *padding* de dos filas y dos columnas en una matriz

En este ejemplo, se han agregado dos filas y dos columnas, por lo cual al realizar la convolución entre la imagen y el filtro 3×3, el resultado será de las mismas dimensiones que la imagen de entrada. Más allá de este caso, es importante decir que el valor de *padding* tendrá incidencia en las dimensiones del dato de salida, de acuerdo con la siguiente Ecuación:

$$(I_x - k_x + p_x + 1) \times \left(I_y - k_y + p_y + 1\right) \qquad (74)$$

Donde x es el número de filas, y es el número de columnas, I es la imagen de entrada, k es el *kernel* y p representa padding. Si los datos del ejemplo se procesan con el *padding* señalado (dos filas y dos columnas añadidas) y un filtro 3×3, las dimensiones del dato de salida serán:

$$(5 - 3 + 2 + 1) \times (7 - 3 + 2 + 1) = 5 \times 7 \qquad (75)$$

Prohibida su reproducción o difusión total o parcial | © 2023 |

Que en este caso corresponde a la misma dimensión de la imagen original, dado que se ha utilizado $p_x = k_x - 1$ y $p_y = k_y - 1$.

En cuanto al *stride*, este corresponde al número de filas y número de columnas que se desplaza el kernel en cada operación. Hasta aquí, los ejemplos mostrados involucraron un *stride* 1, sin embargo, en la práctica es posible utilizar un desplazamiento mayor. Para ilustrar este concepto, la siguiente figura muestra el desplazamiento del *kernel* con un valor de 1, y un valor de 2 (Saravia, 2021).

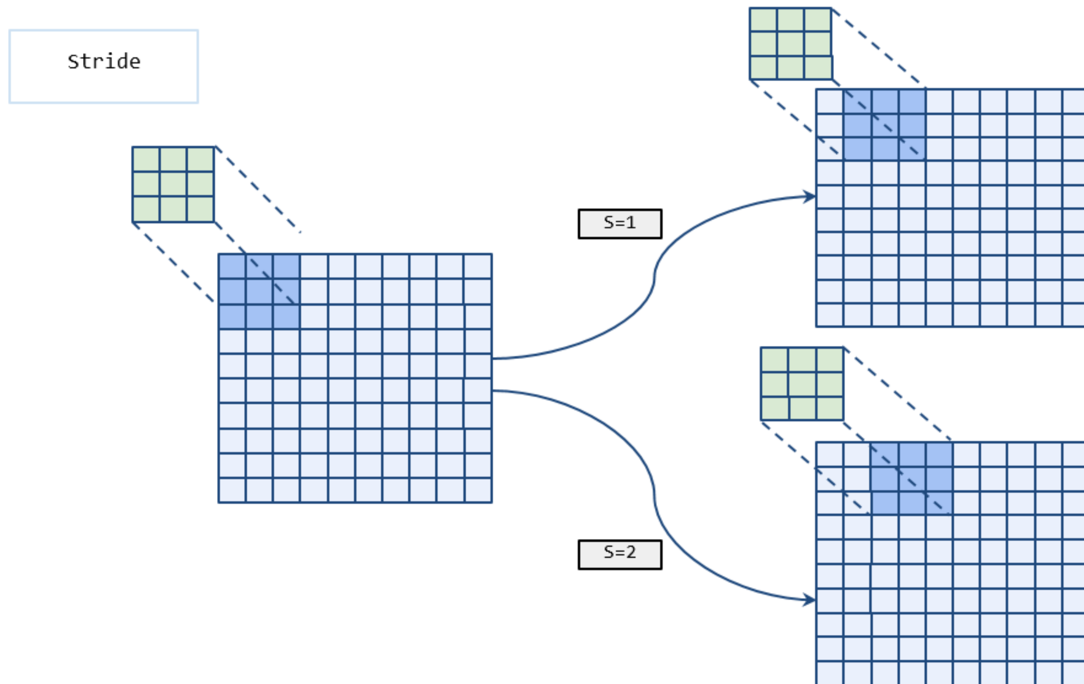


Figura 25. Ejemplo de dos valores de *stride* diferentes en una operación de convolución.

Al igual que para el *padding*, los valores que se definan en el *stride* inciden directamente en las dimensiones de los datos de salida, que están dadas por:

$$\lfloor (I_x - k_x + p_x + s_x)/s_x \rfloor \times \lfloor (I_y - k_y + p_y + s_y)/s_y \rfloor \quad (76)$$

Donde s corresponde a *stride*, y los corchetes medios corresponden a una operación de valor entero. Continuando con el ejemplo, se utilizará un *stride* 2 tanto en filas como en columnas, por lo cual las dimensiones de los datos de salida serán:

$$\lfloor (5 - 3 + 2 + 2)/2 \rfloor \times \lfloor (7 - 3 + 2 + 2)/2 \rfloor = 3 \times 4 \quad (77)$$

Prohibida su reproducción o difusión total o parcial | © 2023 |

Para definir el *stride* en *keras/tensorflow* es necesario especificar un valor entero (si el *stride* en las dos dimensiones es igual) o una tupla o lista de 2 enteros (si el *stride* en filas y columnas es diferente). Estos valores, por lo tanto, detallan el desplazamiento del filtro de convolución a lo alto y ancho. En el caso del padding, se dispone de dos opciones: "*valid*" y "*same*". La primera opción significa que no aplica *padding*; por su parte, "*same*" entrega como resultado un *padding* uniforme ya sea a izquierda/derecha o arriba/abajo de tal forma que garantiza que la salida tenga la misma dimensión que la entrada[19].

Ejemplo de red neuronal convolucional en Python

A continuación, se muestra un modelo de red neuronal convolucional que se entrenará con el dataset MNIST, y que involucra tres capas convolucionales y una capa FC.

En primer lugar, se importan los módulos y librerías necesarias:

```
# Carga de librerías
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Carga de módulos
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
, Conv2D, MaxPooling2D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
```

Conjunto de datos MNIST, normalizando datos entre 0 y 1, redimensionando a un tensor de 4 dimensiones (muestras, alto, ancho, canales), y convirtiendo las etiquetas a una representación categórica (*one-hot encoding*).

```
# Dataset
```

[19] https://keras.io/api/layers/convolution_layers/convolution2d/

Prohibida su reproducción o difusión total o parcial | © 2023 |

```
(x_train, y_train ), (x_test , y_test) = mnist.loa
d_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = x_train.reshape( (x_train.shape[0], 28,
28, 1))
x_test = x_test.reshape( (x_test.shape[0], 28, 28,
 1))

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Hiperparámetros: tamaño de lote de 32, tasa de aprendizaje 0.01 y entrenamiento por 10 épocas. Modelo secuencial con 3 capas convolucionales con filtros 3×3, un *stride* 1×1, y *padding* "*valid*". Para las tres capas convolucionales, el primer argumento que se especifica es el número de filtros que tendrá la capa, valor que usualmente se define como una potencia de 2 y que por lo general aumenta capa tras capa.

```
# Hiperparámetros
batch_s, lr, num_epochs = 32, 0.01, 10
optimizerf = tf.keras.optimizers.SGD(learning_rate
=lr)

# Modelo CNN
model = Sequential([
                    Conv2D(32, (3, 3),
                           activation= 'relu',
                           input_shape=(28, 28, 1)
,
                           strides=(1, 1),
                           padding="valid"),
                    MaxPooling2D(),
                    Conv2D(64, (3, 3),
                           activation= 'relu',
                           strides=(1, 1),
                           padding="valid"),
```


Prohibida su reproducción o difusión total o parcial | © 2023 |

```
                    MaxPooling2D(),
                    Conv2D(128, (3, 3),
                           activation= 'relu',
                           strides=(1, 1),
                           padding="valid"),
                    MaxPooling2D(),

                    Flatten(),
                    Dense(10, activation='softmax'
)])
```

Una vez se ha creado el modelo, es necesario compilarlo (`compile`) previo a su entrenamiento (`fit`):

```
model.compile(
  optimizer=optimizerf,
  loss= 'categorical_crossentropy',
  metrics=['CategoricalAccuracy']
)

history = model.fit(
    x_train,
    y_train,
    epochs=1,
    batch_size=batch_s,
    validation_data=(x_test, y_test))
```

```
1875/1875 [==============================] - 17s 5ms/step - loss: 0.5368 - categorical_accuracy: 0.8567
```

Al observar el resumen del modelo, se percibe el bajo número de parámetros que involucran las capas convolucionales en comparación con las capas FC:

```
model.summary()
```


Prohibida su reproducción o difusión total o parcial | © 2023 |

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_19 (Conv2D)          (None, 26, 26, 32)        320

 max_pooling2d_16 (MaxPoolin  (None, 13, 13, 32)       0
 g2D)

 conv2d_20 (Conv2D)          (None, 11, 11, 64)        18496

 max_pooling2d_17 (MaxPoolin  (None, 5, 5, 64)         0
 g2D)

 conv2d_21 (Conv2D)          (None, 3, 3, 128)         73856

 max_pooling2d_18 (MaxPoolin  (None, 1, 1, 128)        0
 g2D)

 flatten_8 (Flatten)         (None, 128)               0

 dense_9 (Dense)             (None, 10)                1290

=================================================================
Total params: 93,962
Trainable params: 93,962
Non-trainable params: 0
```

Una vez se tiene el modelo entrenado, es posible realizar predicción como se explicó en el capítulo anterior.

Inicialización de parámetros

Tal como se explicó en el capítulo 2, es necesario que los parámetros que se empiezan a actualizar en la primera iteración de un ciclo de entrenamiento tengan un valor definido, al cual se le pueda restar un término que depende del gradiente. Sin embargo, esta no es la única razón o característica involucrada en el uso de un método de inicialización. Particularmente, los métodos de inicialización en aprendizaje automático deben tratar de acotar o prevenir que el gradiente de un parámetro tome un valor excesivamente alto[20], pero tampoco

[20] https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

Prohibida su reproducción o difusión total o parcial | © 2023 |

que tomen un valor excesivamente pequeño que impida que el gradiente identifique una tendencia de cambio en los datos[21].

Adicionalmente, es necesario que los algoritmos de inicialización eviten que los parámetros dentro de una capa sean iguales, dado que los mismos parámetros con unas mismas entradas podrían producir un mismo valor de activación hacia la siguiente capa. Es decir, que diferentes unidades o filtros de una misma capa podrían aprender lo mismo de los datos. Esto ocasionaría que el gradiente respecto a los parámetros sea igual para todas las unidades o filtros, y que al final el modelo se comporte como si tuviera una sola neurona o un solo filtro. Este comportamiento se evita con un método de inicialización que asigne valores diferentes o valores aleatorios, particularmente en los pesos o en los filtros de la capa (no es necesario para el *bias*).

Los métodos de inicialización disponibles en *keras/tensorlfow* se pueden consultar en https://www.tensorflow.org/api_docs/python/tf/keras/initializers.

El método de inicialización general asigna valores de pesos aleatorios con una distribución normal y valores de *bias* en cero. Métodos alternativos como el propuesto por Xavier (Glorot, 2010), permite utilizar una distribución normal de media cero y desviación estándar dependiente del número de unidades del tensor de entrada y de salida, o también propone el uso de una distribución uniforme, donde los valores límite también dependen de los mismos dos argumentos.

Otro método de inicialización popular es el propuesto por He (He K. Z., 2015), propone alternativas para definir la desviación estándar de la distribución uniforme (manteniendo la desviación estándar en cero), y también para el valor límite de la distribución uniforme. En este caso, sólo utiliza el número de unidades del tensor de entrada.

Convolución para múltiples canales de entrada

Al utilizar redes neuronales convolucionales, es común que la entrada presente más de un canal, por ejemplo, al utilizar imágenes RGB. Este tipo de imágenes se define por su dimensión espacial mediante alto y ancho, y por su profundidad o resolución espectral que equivale al número de canales (ej. 3 para imágenes RGB).

[21] https://en.wikipedia.org/wiki/Vanishing_gradient_problem

Prohibida su reproducción o difusión total o parcial | © 2023 |

Es importante señalar que os ejemplos mostrados hasta aquí, trataron con imágenes de un solo canal, por lo cual el *kernel* para procesar dicho dato es de un canal también. Al tener imágenes de más de un canal, el *kernel* necesario para procesar los datos tendrá el mismo número de canales de los datos de entrada. De esta forma, cada canal del *kernel* procesa el respectivo canal de los datos de entrada, para finalmente sumar sus resultados y entregar un único valor, como lo muestra la siguiente figura (Saravia, 2021).

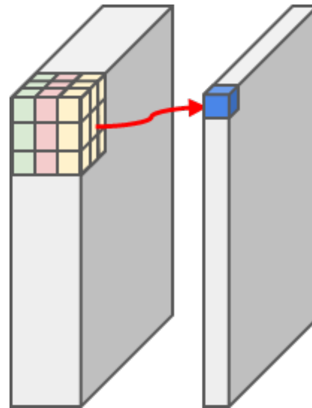


Figura 26. Ilustración de proceso de convolución para datos de entrada de varios canales y *kernel* del mismo número de canales.

En la Figura, los datos de entrada (en color gris) poseen tres canales, por lo cual el filtro (3×3) presenta también tres canales (ilustrados en verde, rojo y amarillo). Además, el filtro en cada posición entregará un único valor de salida (ilustrado en color azul), por lo cual típicamente el dato de salida contendrá solo un canal. Es decir que el resultado en cada canal de salida se calcula a partir del *kernel* de convolución correspondiente a ese canal de salida calculado sobre todos los canales del tensor de entrada (Zhang, 2021). En caso de definir un número de filtros mayor a 1, el número de canales del dato de salida será igual al número de filtros, tal como sucede en las capas convolucionales 1×1.

Capas de convolución 1×1

Algunos modelos de aprendizaje profundo utilizan capas convolucionales de dimensión 1. Debido a esto, este tipo de capa procesará los datos solo en la dimensión de los canales. Es decir, no tiene en cuenta la información espacial de una vecindad alrededor de un píxel, solo tiene en cuenta su profundidad. Una de sus principales aplicaciones es modificar el número de canales de los datos, dado que cada capa convolucional 1×1 determinará un número de canales de salida.

Prohibida su reproducción o difusión total o parcial | © 2023 |

De aquí que el número de parámetros de la capa está dado solamente por el producto entre el número de canales de entrada multiplicado por el número de canales de salida. La ilustración de la operación de una capa convolucional 1×1 de un canal se ilustra en la siguiente Figura (Saravia, 2021).

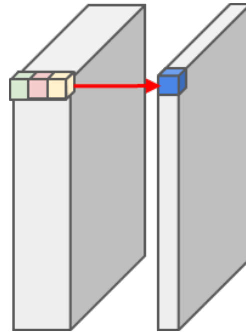


Figura 27. Ilustración de proceso de convolución 1×1.

Es importante tener en cuenta que este tipo de capa permite implementar una capa FC, dado que se aplica sobre un solo píxel.

Capas de *pooling*

Con el fin de operar capas convolucionales a diferentes resoluciones e impedir que los patrones que aprende la red sean sensibles a la ubicación y dimensiones de los datos, se suele utilizar capas de agrupación o *pooling*, que realizan una especie de submuestreo de los datos, por lo cual entregan datos con menor número de filas y de columnas.

Estos operadores son deterministas, es decir, que ante una misma entrada entregan una misma salida. Por lo general involucran una operación de selección del valor máximo de un área (*max pooling*) o del valor medio de los datos en el área (*average pooling*). Su operación se ilustra a continuación para una operación de *max pooling* (Saravia, 2021):

Prohibida su reproducción o difusión total o parcial | © 2023 |

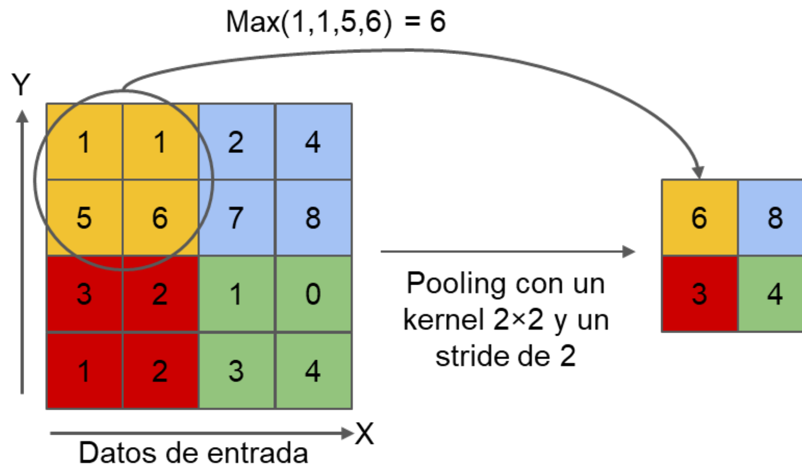


Figura 28. Ilustración de proceso de *max pooling* con *stride* de 2.

Para el ejemplo mostrado, es posible especificar también un valor de *padding* y *stride*. Es común utilizar filtros 2×2, con *padding* "*same*" y con un *stride* de 2. Estos valores garantizan que el ancho y alto de los datos de salida sean la mitad respecto a los datos de entrada. En esta operación, el número de canales de salida equivale al número de canales de los datos de entrada.

Finalmente, los tipos de capas de *pooling* disponibles en *Keras/tensorflow* se listan a continuación.

- MaxPooling1D layer
- MaxPooling2D layer
- MaxPooling3D layer
- AveragePooling1D layer
- AveragePooling2D layer
- AveragePooling3D layer
- GlobalMaxPooling1D layer
- GlobalMaxPooling2D layer
- GlobalMaxPooling3D layer
- GlobalAveragePooling1D layer

Guardar y cargar modelos

Un modelo creado en Keras/tensorflow abarca la arquitectura (capas y su interconexión), parámetros (valores de pesos y *bias*), optimizador y el conjunto

Prohibida su reproducción o difusión total o parcial | © 2023 |

de pérdidas y métricas, los cuales se definen al compilar el modelo. De esta forma, una vez un modelo ha sido entrenado, es posible exportar tanto su arquitectura como sus parámetros o su configuración, para su posterior utilización o implementación. Una de las alternativas en keras/tensorflow es el método `save`[22] que que permite tanto exportar en formato json o en formato h5. El formato h5, genera un archivo de estado con claves de directorio para las capas y sus pesos.

Posteriormente, el comando `load`[23] permite cargar ya sea el modelo o sólo sus pesos (en formato hdf5). El siguiente ejemplo ilustra la utilización de estos dos comandos.

```
from keras.models import load_model

# Guardar el modelo y los pesos
model.save('alexnet.h5')
model.save_weights('alexnetw.hdf5')

# Cargar el modelo y los pesos
model = load_model('alexnet.h5')
model.load_weights('alexnetw.hdf5')
```

Puntos de control de modelos (Model ckeck point)

Durante el entrenamiento de un modelo, bajo un criterio dado (un número de época, un valor de métrica, etc.), es posible realizar un determinado tipo de acción conocida como *callback*. Este tipo de acción es útil para procesos como registrar logs o estadísticas del modelo o sus métricas, guardar versiones del modelo, detener el entrenamiento de forma anticipada (*early stopping*).

La opción que permite guardar periódicamente el modelo o sus pesos durante el entrenamiento se conoce como *model ckeckpoint*[24]. Como criterio para guardar el modelo o sus pesos en disco, puede definirse que sea al final de cada época, o transcurridos un número dado de lotes, o también, definir una métrica a monitorear, de tal forma que el modelo solo se exporta cuando el rendimiento

[22] https://www.tensorflow.org/guide/keras/serialization_and_saving
[23] https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

[24] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

Prohibida su reproducción o difusión total o parcial | © 2023 |

mejora. Un ejemplo de utilización se ilustra en el siguiente código, en donde, luego de importar el método, se define una ruta donde se guardará el modelo y el nombre de archivo que puede configurarse para utilizar variables del entrenamiento. En ese caso, la métrica a monitorear es "*accuracy*". El *checkpoint* creado se instancia durante el entrenamiento como un *callback*.

```
# Callbacks - ModelCheckPoint
from tensorflow.keras.callbacks import
ModelCheckpoint
filepath="weights-improvement-{epoch:02d}-
{accuracy:.2f}.h5"
checkpoint = ModelCheckpoint(filepath,
                             monitor='accuracy',
                             verbose=1,
                             save_best_only=True,
                             mode='max')
callbacks_list = [checkpoint]

history = model.fit(train_ds,
                    validation_data=val_ds,
                    epochs=65,
                    verbose=1,
                    callbacks=callbacks_list)
```

Predicción y evaluación con modelos entrenados

Como se ha comentado hasta aquí, una vez creado el modelo, es posible configurarlo especificando una función de pérdida y métricas de evaluación a través del método *compile()*, o entrenarlo mediante el método *fit()*. Con el método entrenado, se pueden realizar predicciones sobre nuevos datos mediante el método *predict()*. En este caso, se genera predicciones de salida para las muestras de entrada, es decir el modelo realiza una inferencia sobre estos nuevos datos y entrega una salida, que por ejemplo corresponde a la clase a la cual pertenece la imagen de entrada. Los métodos aplicables a un modelo en keras/tensorflow pueden revisarse en https://www.tensorflow.org/api_docs/python/tf/keras/Model.

Prohibida su reproducción o difusión total o parcial | © 2023 |

Al aplicar el método *predict(),* el cálculo se realiza por lotes, por lo cual no es necesario aplicarlo dentro de bucles o lazos de lectura de datos. Esto significa que el método está diseñado para funcionar con entradas a gran escala, puede funcionar desde una sola imagen, hasta un conjunto o lote de imágenes. Cuando se instancia el método de predicción, es necesario especificar como mínimo los datos de entrada (tensor, arreglo, etc). De manera complementaria, es posible especificar aspectos como el tamaño del lote, el número de pasos o *callbacks*. La forma general de aplicación del método de predicción en clasificación, depende del tipo de clasificador: binario o multiclase. En el primer caso, por lo general se tendrá una función de activación sigmoide en la última capa FC, por lo cual discriminar los valores de salida comparándolos con 0.5 servirá para determinar la clase (0, 1).

```
result = (model.predict(x) > 0.5).astype("int32")
```

Al tener un clasificador multiclase, donde la función de activación de la última capa FC es de tipo *softmax*, se suele utilizar la función `argmax` de `NumPy`, que devuelve los índices de los valores máximos a lo largo de un eje. Este valor máximo corresponde a la clase del dato de entrada.

```
result = np.argmax(model.predict(img), axis=-1
```

Más allá de la predicción, es posible evaluar el rendimiento de un modelo mediante el método `evaluate()`. Este método devuelve los valores de pérdida y métricas del modelo en modo de prueba. Al igual que en la predicción, este método opera por lotes. Los principales argumentos a especificar cuando se evalúa un modelo corresponden a los datos a evaluar y a sus etiquetas. Es posible especificar también el tamaño del lote o los pesos de las muestras de prueba para ponderar la función de pérdida, entre otros. Estos argumentos y su valor predeterminado se muestran a continuación.

```
evaluate(x=None, y=None, batch_size=None,
verbose=1, sample_weight=None, steps=None,
callbacks=None, max_queue_size=10, workers=1,
use_multiprocessing=False, return_dict=False)
```


Prohibida su reproducción o difusión total o parcial | © 2023 |

Arquitectura LeNet

LeNet es una arquitectura de aprendizaje automático que utiliza capas convolucionales y que fue propuesta por Yann LeCun, Léon Bottou, Yoshua Bengio, y Patrick Haffner en 1998, consolidando un gran trabajo desarrollado en los años anteriores a su publicación. Es importante destacar que esta arquitectura se propuso mucho antes de la explosión del aprendizaje profundo iniciada a partir del 2012. En su momento, este modelo presentó resultados competitivos con respecto a métodos de aprendizaje de máquina consolidados como lo son las máquinas de soporte vectorial (SVM).

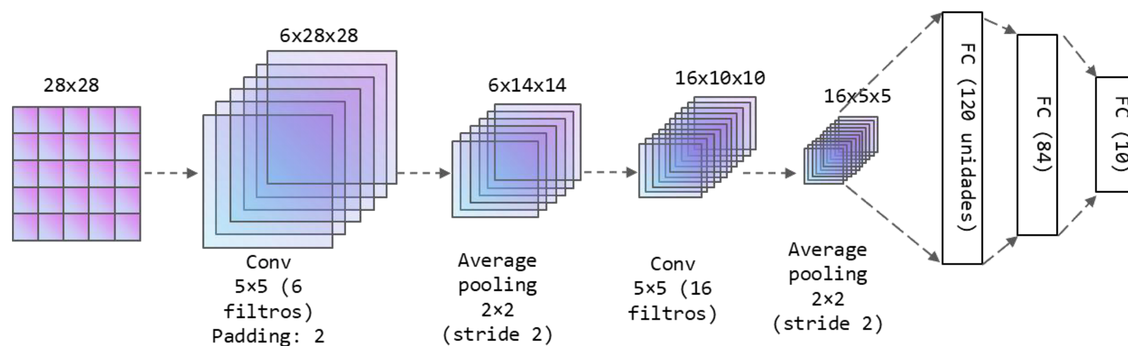


Figura 29. Arquitectura LeNet.

Como se aprecia en la Figura, la arquitectura LeNet implementa dos capas convolucionales con filtros 5×5 que utilizan función de activación sigmoide (LeCun Y. B., 1998). Estas capas convolucionales aumentan la profundidad de los datos, primero con seis filtros y luego con dieciséis filtros. A la par se reducen las dimensiones de los datos en alto y ancho, mediante capas de *pooling* que promedia los datos de una vecindad 2×2 y que reduce el alto y ancho a la mitad (reducción de dimensiones a la cuarta parte). Estas primeras capas permiten la extracción de características de los datos.

En la segunda parte del modelo se tienen capas tipo *Fully connected*, que reducen la dimensionalidad de los datos hasta llegar al número de clases del problema (10), dado que esta arquitectura se evaluó sobre el dataset MNIST. Por consiguiente, las capas FC realizan la clasificación.

La implementación de la arquitectura LeNet como modelo secuencial en *keras/tensorflow* se puede implementar de la siguiente forma:

```
model = Sequential([
```


Prohibida su reproducción o difusión total o parcial | © 2023 |

```
                    Conv2D(filters=6, kernel_size=5,
                          activation='sigmoid', paddi
ng='same'),
                    AvgPool2D(pool_size=2, strides=2)
,
                    Conv2D(filters=16, kernel_size=5,
 activation='sigmoid'),
                    AvgPool2D(pool_size=2, strides=2)
,
                    Flatten(),
                    Dense(120, activation='sigmoid'),
                    Dense(84, activation='sigmoid'),
                    Dense(10)])
```

Se resalta que dada la fecha en la cual se presentó esta arquitectura, las funciones de activación fueron sigmoide y las capas de *pooling* son de promedio. Hoy en día es más común encontrar arquitecturas que utilicen función de activación ReLU y *max pooling*.

Aumento de datos

Como se mencionó en el capítulo anterior, una de las primeras alternativas para combatir el sobre ajuste de un modelo es aumentar la cantidad de muestras del conjunto de datos de entrenamiento. Sin embargo, no en todos los casos es fácil recolectar más datos, ni tampoco aumentar su diversidad.

Es aquí donde se habla de técnicas de *data augmentation*, es decir, literalmente aumento de datos, aunque no hay que confundirse con el concepto literal de esta frase. Para esto, las técnicas de *data augmentation* no están enfocadas a aumentar el número de muestras del conjunto de datos, sino de aumentar la diversidad de estas. Este aumento de diversidad puede facilitar que el modelo aprenda a reconocer de una mejor forma los patrones de los datos de entrenamiento, lo que conlleva a una mejor generalización del modelo, y en consecuencia, una reducción en el sobre ajuste.

Para aumentar la diversidad de los datos es posible aplicar un pre-procesamiento a los datos de entrenamiento que modifiquen ciertas características de las imágenes. Por ejemplo, es posible aplicar un zoom a la imagen, para que los

Prohibida su reproducción o difusión total o parcial | © 2023 |

objetos de interés se vean más grandes, o también se pueden rotar o desplazar las imágenes haciendo que los datos presenten diversos contextos que siguen siendo válidos. En este caso, la imagen original se reemplaza por la imagen modificada, conservando el número de ejemplos de entrenamiento.

Operaciones típicas de pre-procesamiento disponibles en los frameworks de aprendizaje profundo incluyen las siguientes:

- Rotación
- Desplazamiento (a lo ancho o a lo largo)
- Modificación de brillo
- Inclinación
- Zoom
- Efecto espejo (a nivel vertical u horizontal)

En cada una de estas operaciones es posible especificar el grado de modificación de la imagen. Para la rotación, por ejemplo, el rango de grados en los cuales de forma aleatoria se modificará la imagen.

La implementación de técnicas de *data augmentation* suele realizarse desde el momento de la lectura de datos de entrenamiento, con opciones como el *Image Data Generator* de *Keras*. Esta opción permite implementar operaciones de pre-procesamiento de datos, o leer datos desde un dataframe, o desde una carpeta a los cuales se les aplicarán de forma aleatoria las técnicas de aumento de datos comentadas anteriormente. En el caso de leer los datos desde un directorio, lo deseable es que el directorio principal contenga subcarpetas con imágenes de cada una de las clases.

Como ejemplo se mostrará la implementación de *data augmentation* con el *Image Data Generator* de *keras*, aplicado al dataset MNIST. Para su aplicación, se leerá el dataset desde un directorio de entrenamiento que contiene diez subcarpetas correspondientes a cada clase, y que cada subcarpeta contiene imágenes de ejemplo de cada clase, como se muestra a continuación.

Prohibida su reproducción o difusión total o parcial | © 2023 |

Figura 30. Estructura de conjuntos de datos MNIST (10 clases) con estructura por carpetas

En primer lugar, se define la ruta que contiene los datos:

```
import pathlib
data_dir = "/content/MNIST/trainingSet"
data_dir = pathlib.Path(data_dir)
```

A partir de lo anterior, es posible leer los datos por lotes especificando las operaciones de pre-procesamiento que se aplicarán a los datos, en el generador (`train_datagen`). Para el ejemplo, se aplicarán operaciones de inclinación (shear), zoom. Se ha desactivado la operación de reflejo horizontal dado que no es viable aplicarla para un conjunto de datos que contienen los dígitos del cero al nueve. Aquí se especifica cómo se realizará la distribución de los datos (en entrenamiento y validación). En este caso se utilizan el 80% de los datos para entrenamiento y el 20% para validación. En consecuencia, con el generador creado, es posible generar los contenedores de los datos de entrenamiento (`train_generator`) y de validación (`val_generator`). Para poder crearlos, se especifica el directorio done están los datos, las dimensiones que tendrán las imágenes leídas, el tamaño del lote, el tipo de etiqueta (categórica para el ejemplo) y se especifica cuál es el subconjunto que se está generando.

Prohibida su reproducción o difusión total o parcial | © 2023 |

```
from tensorflow.keras.preprocessing.image import I
mageDataGenerator
img_height = 28
img_width = 28

train_datagen = ImageDataGenerator(
        #rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        #horizontal_flip=True,
        validation_split=0.2)

train_generator = train_datagen.flow_from_director
y(
        'MNIST/trainingSet',
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='categorical',
        subset='training')

val_generator = train_datagen.flow_from_directory(
        'MNIST/trainingSet',
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='categorical',
        subset='validation')
```

```
Found 33604 images belonging to 10 classes.
Found 8396 images belonging to 10 classes.
```

Al ejecutar el script de aumento de datos, se observa el número de imágenes utilizado en cada subconjunto (entrenamiento y validación) y el número de clases identificado que corresponde al número de subcarpetas del directorio. Estos contenedores pueden utilizarse para procesos posteriores como el entrenamiento, de la siguiente forma:

```
model.fit(
        train_generator,
        epochs=5,
        validation_data=val_generator,)
```


Prohibida su reproducción o difusión total o parcial | © 2023 |