# Advance Deep Learning with Keras

## ¿Por qué Keras?

* Keras está diseñado para acelerar el desarrollo, la capacitación y la validación de modelos de Deep Learning.
* TensorFlow de Google, una popular biblioteca de Deep Learning de código abierto, utiliza Keras como una API de alto nivel para su biblioteca. Comúnmente se llama tf.keras.
* En la industria de la tecnología, Google, Netflix, Uber y NVIDIA utilizan Keras.
* Al usar Keras, aumentaremos la productividad al ahorrar tiempo en la implementación del código, que en su lugar se puede dedicar a tareas más críticas, como formular mejores algoritmos de aprendizaje profundo.
* La red MLP (*Multilayer Perceptron*) se usará para construir un clasificador simple usando tf.keras.

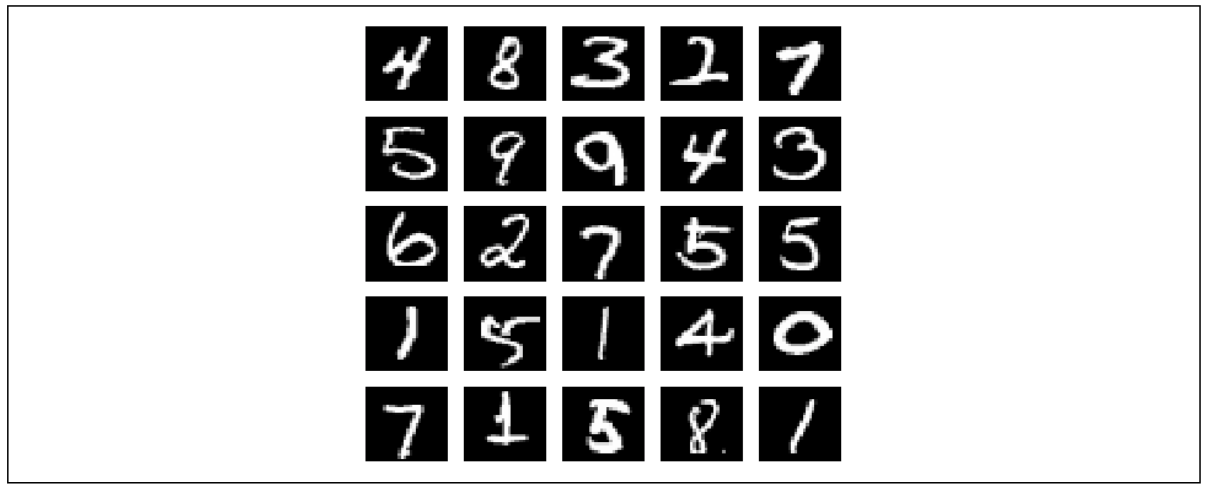
## Multilayer Perceptron (MLP)

Supongamos que el objetivo es crear una red neuronal para identificar números a partir de dígitos escritos a mano. Este es un trabajo clásico de las redes clasificadoras que se puede entrenar mediante regresión logística.

Antes de discutir el modelo clasificador MLP, es esencial que comprendamos el conjunto de datos MNIST. MNIST se utiliza para explicar y validar muchas teorías de Deep Learning porque las 70.000 muestras que contiene son pequeñas, pero suficientemente ricas en información.

## El conjunto de datos MNIST

MNIST es una colección de dígitos escritos a mano que van del 0 al 9. Tiene un conjunto de entrenamiento de 60.000 imágenes y 10.000 imágenes de prueba que se clasifican en categorías o etiquetas.



Ejemplo de imágenes de la tabla de datos MNIST. Cada imagen en escala de grises es de 28 × 28 píxeles

En la figura anterior, se pueden ver imágenes de muestra de los dígitos del MNIST, cada una con un tamaño de 28 x 28 píxeles, en escala de grises.

Antes de discutir el modelo de clasificador MLP, es esencial tener en cuenta que, si bien los datos del MNIST consisten en tensores bidimensionales, deben reformarse según el tipo de capa de entrada. La siguiente Figura muestra cómo se cambia la forma de una imagen en escala de grises de 3 × 3 para las capas de entrada MLP:

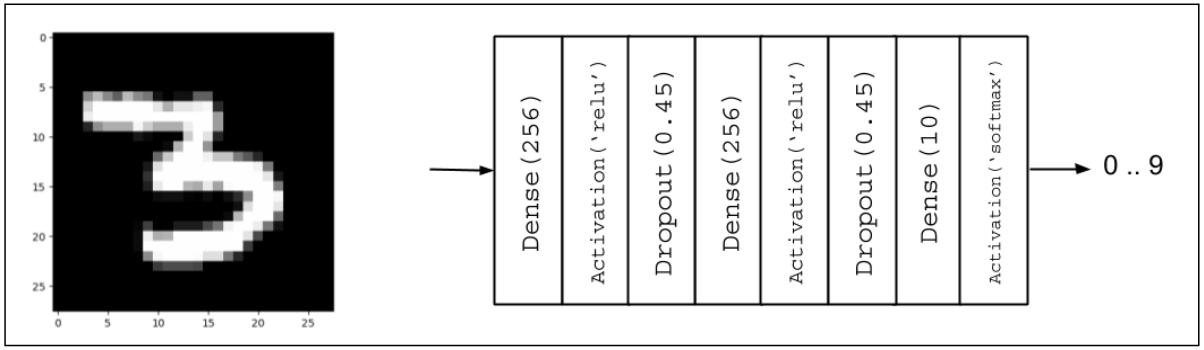
### Imagen que contiene Gráfico Descripción generada automáticamente

## El modelo de clasificación de dígitos MNIST

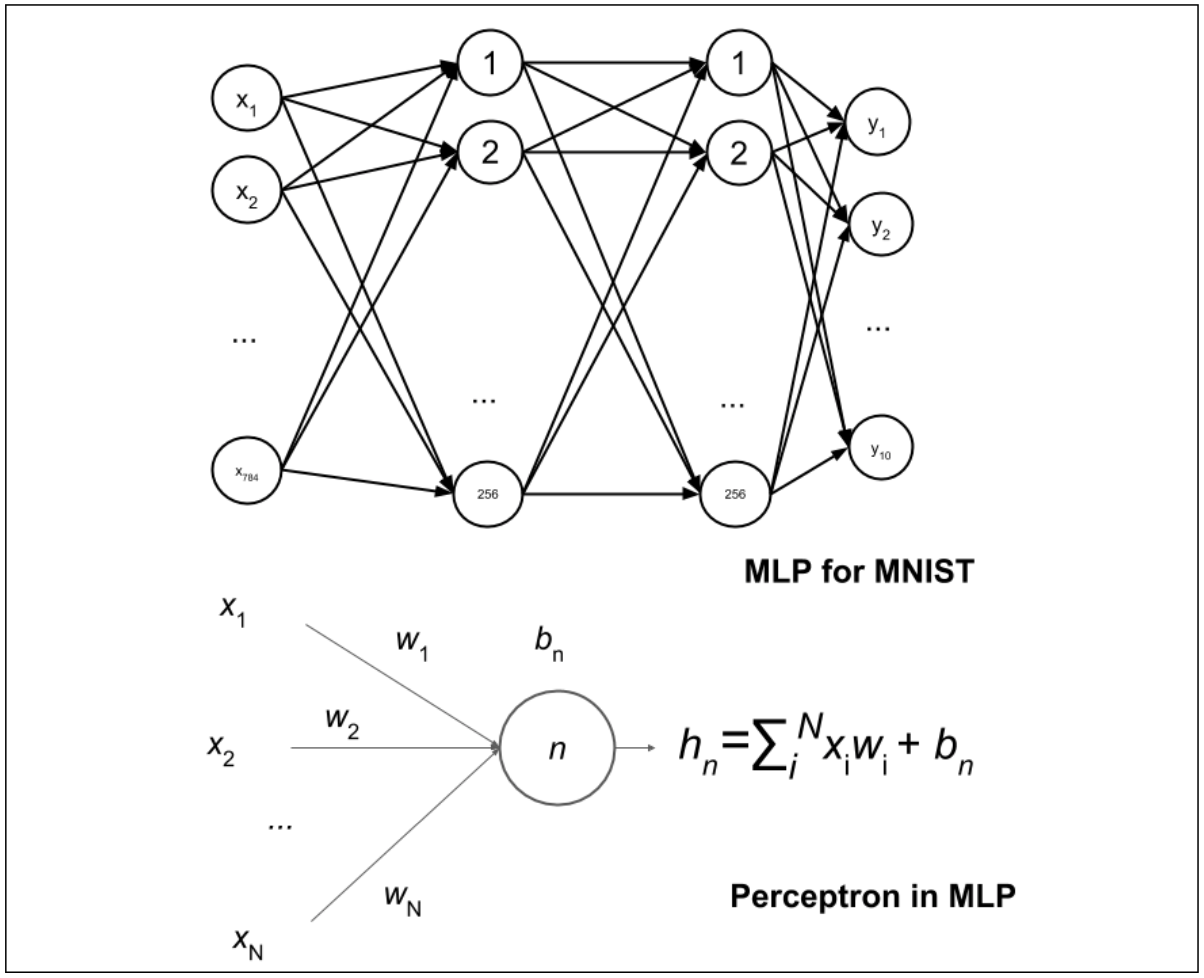
El modelo MLP propuesto que se muestra en la Figura 1.3.3 se puede utilizar para la clasificación de dígitos MNIST. Cuando las unidades o perceptrones están expuestos, el modelo MLP es una red completamente conectada, como se muestra en la Figura 1.3.4. También mostraremos cómo la salida del perceptrón se calcula a partir de las entradas en función de los pesos, wi, y el sesgo, bn, para la n-ésima unidad. Se ilustra la implementación de tf.keras correspondiente

en el Listado 1.3.2:

Para la clasificación de dígitos MNIST se utilizará el siguiente modelo MLP:



Cuando las unidades o perceptrones están expuestos, el modelo MLP es una red completamente conectada, como se muestra en la siguiente figura. En esta imagen los wi corresponden a los pesos y los bn al sesgo de la n-ésima unidad.



La implementación en tf.keras del modelo anterior sería la siguiente:

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Activation, Dropout

from tensorflow.keras.utils import to\_categorical, plot\_model

from tensorflow.keras.datasets import mnist

# carga la tabla de datos mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# calcula el número de etiquetas

num\_labels = len(np.unique(y\_train))

# conviete a código disyuntivo

y\_train = to\_categorical(y\_train)

y\_test = to\_categorical(y\_test)

# dimensiones de la imagen

image\_size = x\_train.shape[1]

input\_size = image\_size \* image\_size

# cambia el tamaño y normaliza

x\_train = np.reshape(x\_train, [-1, input\_size])

x\_train = x\_train.astype('float32') / 255

x\_test = np.reshape(x\_test, [-1, input\_size])

x\_test = x\_test.astype('float32') / 255

# parámetros de la red

batch\_size = 128

hidden\_units = 256

dropout = 0.45

# el modelo es un MLP de 3 capas con ReLU y “abandono” después de cada capa

model = Sequential()

model.add(Dense(hidden\_units, input\_dim=input\_size))

model.add(Activation('relu'))

model.add(Dropout(dropout))

model.add(Dense(hidden\_units))

model.add(Activation('relu'))

model.add(Dropout(dropout))

model.add(Dense(num\_labels))

# salida de cada vector one-hot (disyuntivo)

model.add(Activation('softmax'))

model.summary()

plot\_model(model, to\_file='mlp-mnist.png', show\_shapes=True)

# función de pérdida para un vector one-hot

# uso de adam optimizer

# la precisión es una buena métrica para las tareas de clasificación model.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

# entrena la red

model.fit(x\_train, y\_train, epochs=20, batch\_size=batch\_size)

# valida el modelo en el conjunto de datos de prueba para determinar la generalización

\_, acc = model.evaluate(x\_test,

y\_test,

batch\_size=batch\_size,

verbose=0)

print("\nTest accuracy: %.1f%%" % (100.0 \* acc))

## Construir un modelo usando MLP y Keras

Después de la preparación de los datos sigue construir el modelo, el modelo anterior está compuesto de 3 capas MLP, en Keras una capa MLP se denomina ***densa***. Tanto la primera como la segunda capa MLP son de la misma naturaleza, con 256 unidades cada una, seguidas de la activación y desactivación de la **Unidad Lineal Rectificada** (ReLU). Se eligen 256 unidades ya que 128, 512 y 1024 unidades tienen métricas de rendimiento más bajas. Esto pues, a 128 unidades, la red converge rápidamente pero tiene una precisión de prueba menor. El número adicional de unidades para 512 o 1024 no aumenta significativamente la precisión de la prueba.

El número de unidades es un hiperparámetro. Controla la capacidad de la red. La capacidad es una medida de la complejidad de la función que la red puede aproximar. Por ejemplo, para polinomios, el grado es el hiperparámetro. A medida que aumenta el grado, también aumenta la capacidad de la función.

Como se muestra en las siguientes líneas de código, el modelo clasificador se implementa utilizando la API secuencial de Keras. Esto es suficiente si el modelo requiere una entrada y una salida procesadas por una secuencia de capas. Por simplicidad, usaremos esto por ahora; sin embargo, más adelante, se presentará la API funcional de Keras para implementar modelos avanzados de aprendizaje profundo que requieren estructuras más complejas, como múltiples entradas y salidas.

# model is a 3-layer MLP with ReLU and dropout after each layer

model = Sequential()

model.add(Dense(hidden\_units, input\_dim=input\_size))

model.add(Activation('relu'))

model.add(Dropout(dropout))

model.add(Dense(hidden\_units))

model.add(Activation('relu'))

model.add(Dropout(dropout))

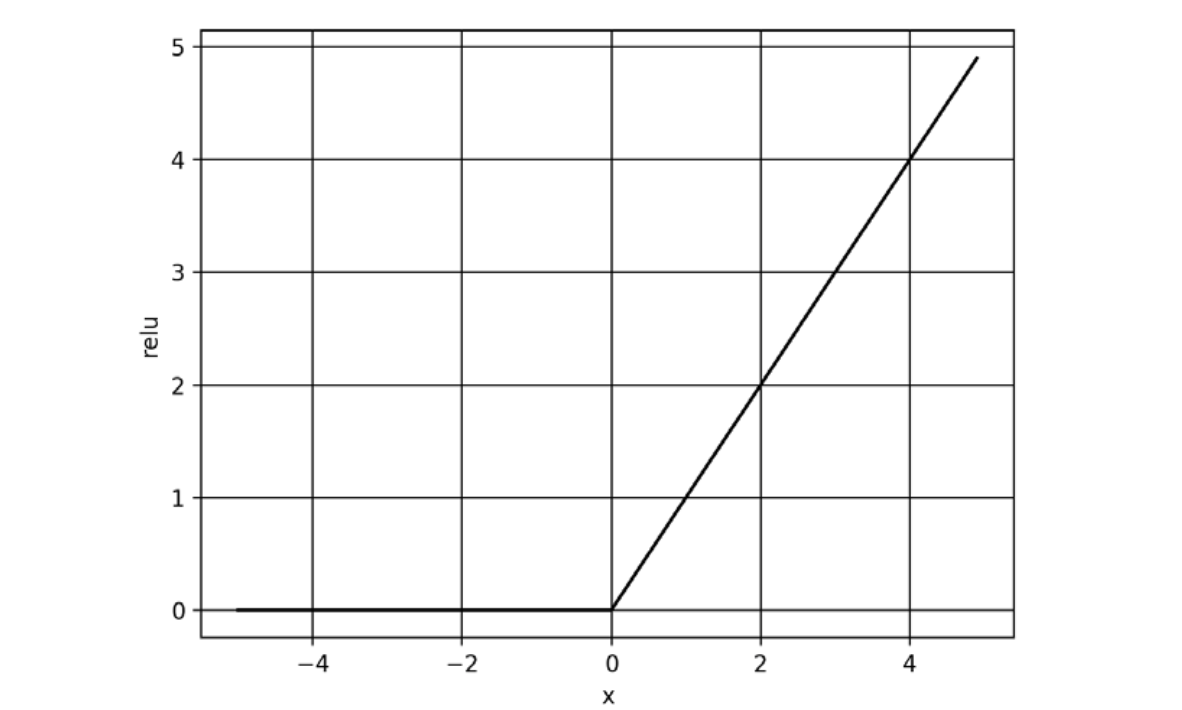
model.add(Dense(num\_labels))

# this is the output for one-hot vector model.

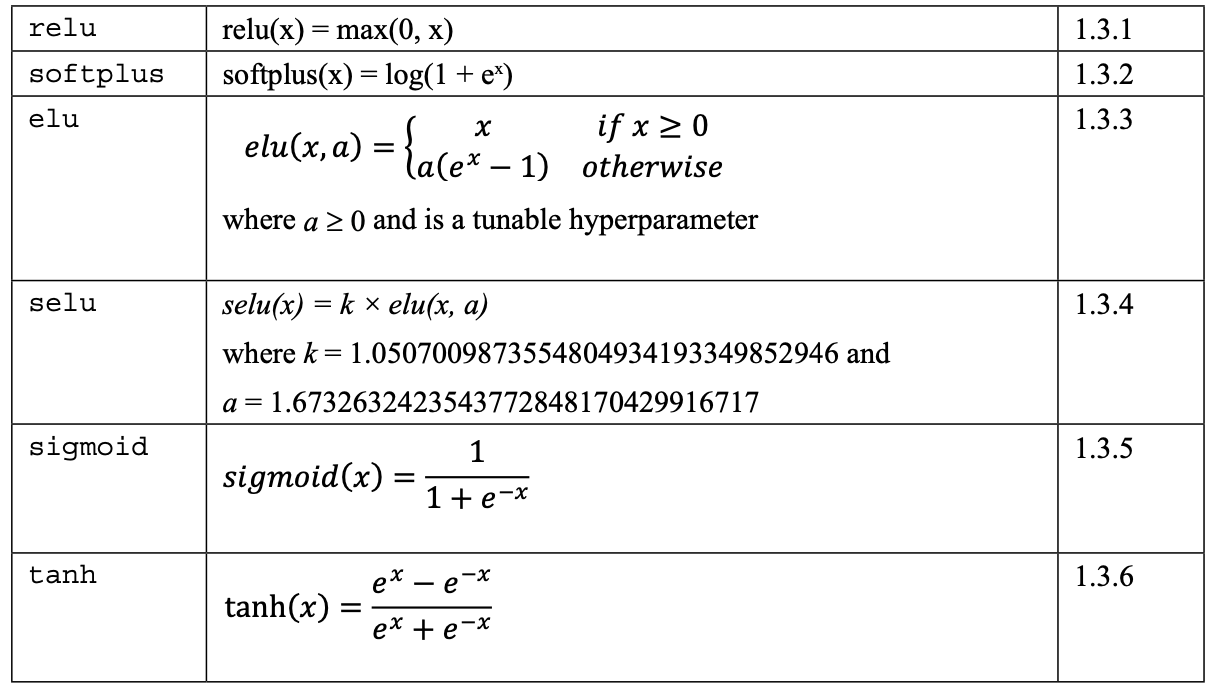
add(Activation('softmax'))

Dado que una capa densa es una operación lineal, una secuencia de capas densas solo puede aproximarse a una función lineal. El problema es que la clasificación de dígitos MNIST es inherentemente un proceso no lineal. La inserción de una activación relu entre las capas densas permitirá que una red MLP modele asignaciones no lineales. relu o ReLU es una función no-lineal simple. Es muy parecido a un filtro que permite que las entradas positivas pasen sin cambios mientras sujeta todo lo demás a cero. Matemáticamente, relu se expresa en la siguiente ecuación y se representa en la Figura 1.3.5:

*ReLU = max(0,x)*



Hay otras funciones no lineales que se pueden utilizar, como elu, selu, softplus, sigmoid y tanh. Sin embargo, relu es la función más utilizada y es computacionalmente eficiente debido a su simplicidad. Las funciones sigmoidea y tanh se utilizan como funciones de activación en la capa de salida y se describirán más adelante. La siguiente tabla muestra la ecuación para cada una de estas funciones de activación:



Aunque hemos completado las capas clave del modelo de clasificador MLP, no hemos abordado el tema de la generalización o la capacidad del modelo para funcionar más allá del conjunto de datos del entrenamiento. Para abordar este problema, presentaremos a continuación la regularización.

## Regularización

Una red neuronal tiene la tendencia a memorizar sus datos de entrenamiento, especialmente si contiene capacidad más que suficiente. En tales casos, la red falla catastróficamente cuando se somete a los datos de prueba. Este es el caso clásico de la red que no logra generalizar. Para evitar esta tendencia, el modelo utiliza una capa o función de regularización. Una capa de regularización común es la Dropout (deserción).

¿??

## Función de activación y pérdida de salida

La capa de salida tiene 10 unidades seguidas de una capa de activación softmax. Las 10 unidades corresponden a las 10 posibles etiquetas, clases o categorías. La activación de softmax se puede expresar matemáticamente, como se muestra en la siguiente ecuación:

*Softmax(*

La ecuación se aplica a todas las salidas *N = 10*, para i = 0, 1 ... 9 para la predicción final. La idea de softmax es sorprendentemente simple. Aplasta las salidas en probabilidades normalizando la predicción. Aquí, cada salida prevista es una probabilidad de que el índice sea la etiqueta correcta de la imagen de entrada dada. La suma de todas las probabilidades para todas las salidas es 1,0. Por ejemplo, cuando la capa softmax genera una predicción, será un tensor 1D de 10 dim que puede verse como el siguiente resultado:

|  |  |  |  |
| --- | --- | --- | --- |
| **[3.57351579e-11** | **7.08998016e-08** | **2.30154569e-07** | **6.35787558e-07** |
| **5.57471187e-11** | **4.15353840e-09** | **3.55973775e-16** | **9.99995947e-01** |
| **1.29531730e-09** | **3.06023480e-06]** |  |  |

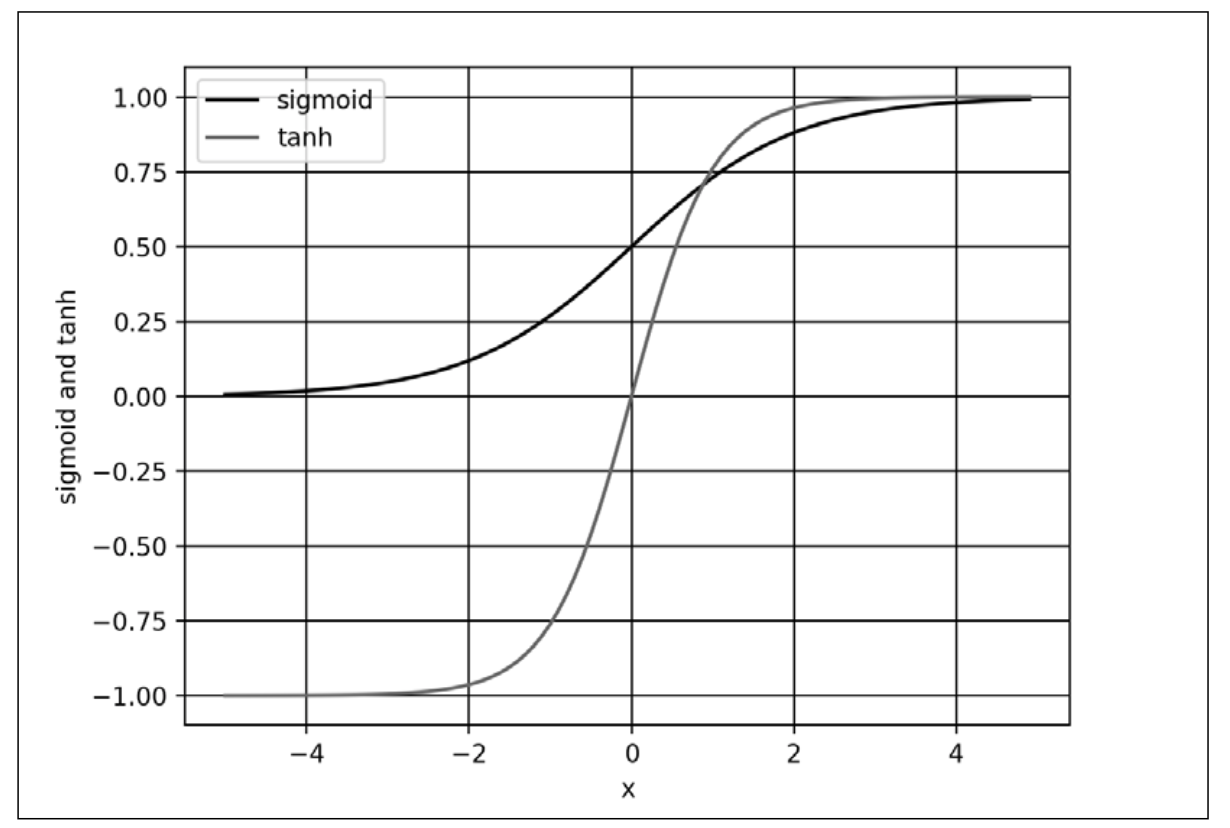
El tensor de salida de la predicción sugiere que la imagen de entrada será 7 dado que su índice tiene la probabilidad más alta (**9.99995947e-01**). El método numpy.argmax() se puede utilizar para determinar el índice del elemento con el valor más alto.

Hay otras opciones de capa de activación de salida, como lineal, sigmoid o tanh. La activación linear es una función de identidad. Copia su entrada a su salida. La función sigmoid se conoce más específicamente como **sigmoide logística**. Esto se utilizará si los elementos del tensor de predicción se mapearán de forma independiente entre 0.0 y 1.0. La suma de todos los elementos del tensor predicho no está restringida a 1.0 a diferencia de softmax. Por ejemplo, sigmoid se usa como la última capa en la predicción de sentimientos (de 0.0 a 1.0, 0.0 es malo y 1.0 es bueno) o en la generación de imágenes (0.0 se asigna al nivel de píxel 0 y 1.0 se asigna al píxel 255).

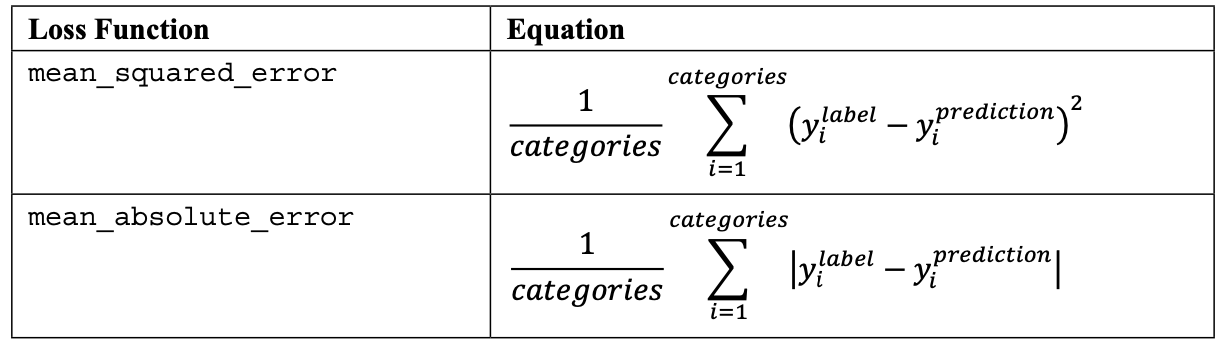
La función tanh mapea su entrada en el rango de -1.0 a 1.0. Esto es importante si la salida puede oscilar tanto en valores positivos como negativos. La función tanh se utiliza más popularmente en la capa interna de redes neuronales recurrentes, pero también se ha utilizado como activación de la capa de salida. Si se usa tanh para reemplazar sigmoid en la activación de salida, los datos usados deben escalarse adecuadamente. Por ejemplo, en lugar de ir escalando cada píxel de escala de grises en el rango [0.0 1.0] usando , se le asigna en el rango [-1.0 a 1.0] usando .

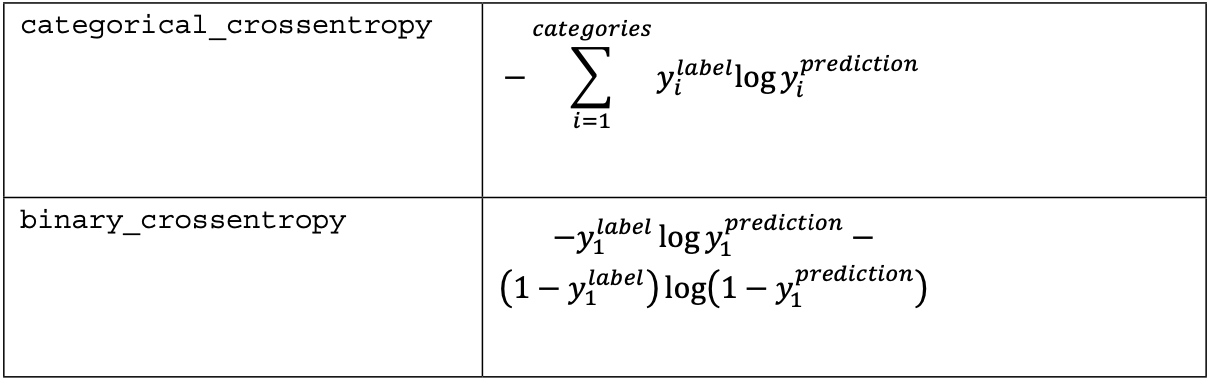
En el siguiente gráfico se muestra las funciones sigmoid y tanh.

Matemáticamente, sigmoidea puede ser expresada como:



Qué tan lejos está el tensor predicho del vector de verdad de un one-hot terreno se llama **pérdida**. Un tipo de función de pérdida es mean\_squared\_error (**MSE**), o el promedio de los cuadrados de las diferencias entre el objetivo o la etiqueta y la predicción. En el ejemplo actual, estamos usando categorical\_crossentropy. Es el negativo de la suma del producto del objetivo o etiqueta y el logaritmo de la predicción por categoría. Hay otras funciones de pérdida que están disponibles en Keras, como mean\_absolute\_error y binary\_crossentropy. La siguiente tabla resume las funciones de pérdida comunes.





La elección de la función de pérdida no es arbitraria sino que debe ser un criterio que el modelo está aprendiendo. Para la clasificación por categoría, categorical\_crossentropy o mean\_squared\_error es una buena opción después de la capa de activación de softmax. La función de pérdida binary\_crossentropy se usa normalmente después de la capa de activación sigmoid, mientras que mean\_squared\_error es una opción para la salida tanh.

## Optimización

Con la optimización, el objetivo es minimizar la función de pérdida. La idea es que, si la pérdida se reduce a un nivel aceptable, el modelo ha aprendido indirectamente la función que asigna entradas a salidas. Las métricas de rendimiento se utilizan para determinar si un modelo ha aprendido la distribución de datos subyacente. La métrica predeterminada en Keras es la **pérdida**. Durante el entrenamiento, la validación y las pruebas, otras métricas como la **precisión** también se pueden incluir. La precisión es el porcentaje, o fracción, de predicciones correctas basadas en la verdad del terreno. En Deep Learning , hay muchas otras métricas de rendimiento. Sin embargo, depende de la aplicación del modelo. En la literatura, las métricas de rendimiento del modelo entrenado en el conjunto de **datos de prueba** se informan para su comparación con otros modelos de deep learning.

En Keras, hay varias opciones para optimizar. Los optimizadores más utilizados son el **descenso de gradiente estocástico (SGD)**, los **momentos adaptativos (Adam)** y la **propagación cuadrática media de raíz (RMSprop)**. Cada optimizador presenta parámetros ajustables como la tasa de aprendizaje, el impulso y el decaimiento. Adam y RMSprop son variaciones de SGD con tasas de aprendizaje adaptativo. En la red de clasificadores propuesta, se utiliza Adam ya que tiene la mayor precisión de prueba.

SGD se considera el optimizador más fundamental. Es una versión más simple del descenso de gradientes en cálculo. En el descenso de gradiente (GD), al trazar la curva de una función cuesta abajo se encuentra el valor mínimo, muy parecido a caminar cuesta abajo en un valle hasta llegar al fondo

Explico nuevamente el descenso del gradiente??

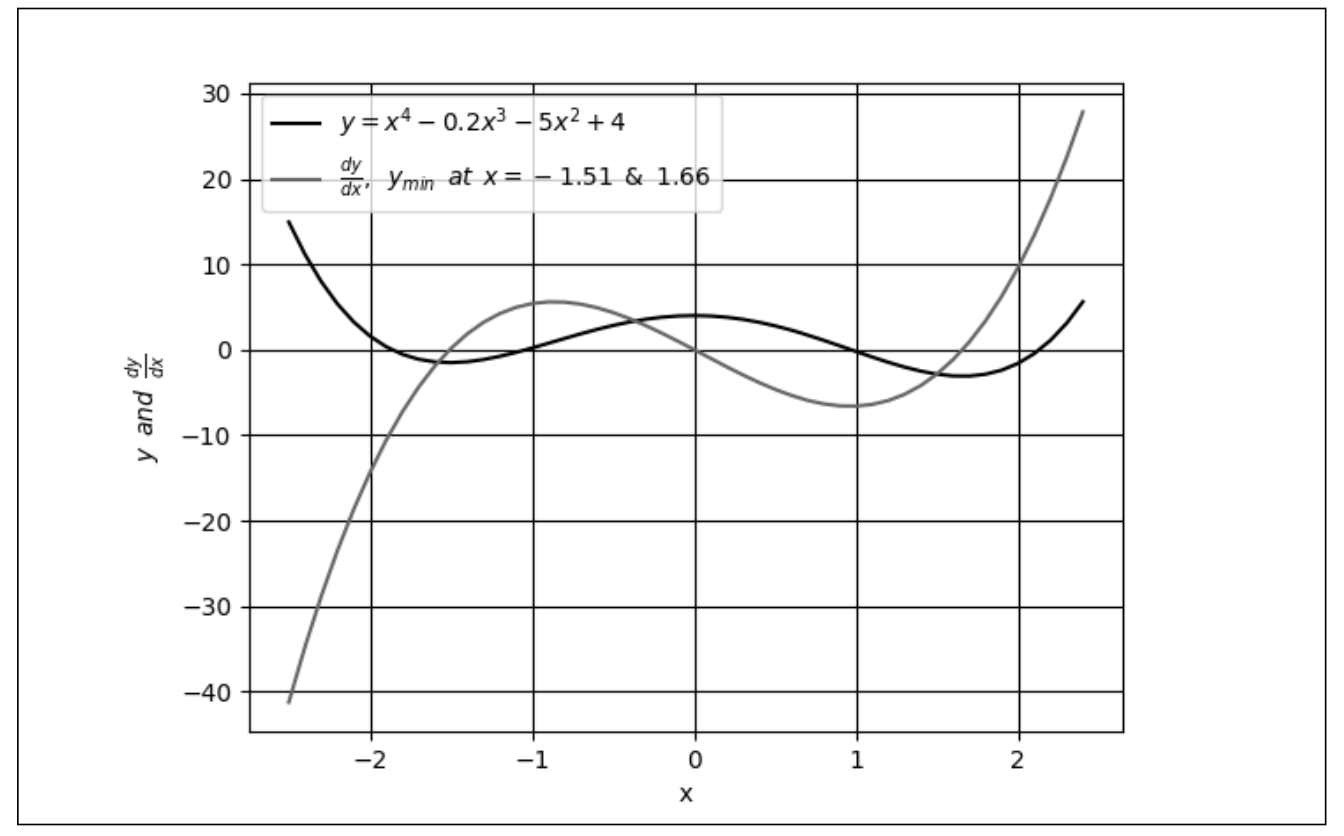
Supongamos que x es el parámetro (peso) que se está ajustando para encontrar el valor mínimo de y (función de pérdida). Comenzando en un punto arbitrario de x = -0.5 el gradiente -2.0. El algoritmo GD impone que x se actualice ax = -0,5 - z (-2,0). El nuevo valor es igual al valor anterior, más el opuesto del gradiente escalado por z. El pequeño número z se refiere a la tasa de aprendizaje. Si z = 0.01 entonces el nuevo valor de x = -0.48. GD se realiza de forma iterativa. En cada paso, y se acercará a su valor mínimo. En x = 0,5, dy = 0,0. GD ha encontrado el valor mínimo absoluto de y = -1.25. El gradiente no recomienda ningún cambio adicional en x.

La elección de la tasa de aprendizaje es crucial. Un valor grande de 𝜖 puede no encontrar el valor mínimo, ya que la búsqueda simplemente oscilará hacia adelante y hacia atrás alrededor del valor mínimo.

Por un lado, un valor grande de 𝜖 puede requerir un número significativo de iteraciones antes de encontrar el mínimo. En el caso de múltiples mínimos, la búsqueda puede quedarse atascada en un mínimo local.

En la figura se puede ver un ejemplo de mínimos múltiples. Si por alguna razón la búsqueda comenzó en el lado izquierdo del gráfico y la tasa de aprendizaje es muy pequeña, existe una alta probabilidad de que GD encuentre x = -1.51 como el valor mínimo de y. GD no encontrará el mínimo global en x = 1,66. Una tasa de aprendizaje suficientemente valorada permitirá al GD superar la colina en x = 0.0.

En las prácticas de Deep Learning normalmente se recomienda comenzar con una tasa de aprendizaje mayor (por ejemplo, de 0,1 a 0,001) y disminuirla gradualmente a medida que la pérdida se acerca al mínimo.



GD no se usa normalmente en redes neuronales profundas, ya que es común encontrar millones de parámetros para entrenar. Es computacionalmente ineficiente realizar un GD completo. En su lugar, se utiliza SGD. En SGD, se elige un mini lote de muestras para calcular un valor aproximado del descenso. Los parámetros (por ejemplo, pesos y sesgos) se ajustan mediante la siguiente ecuación:

Dónde**,**  y **g =** son los parámetros y el tensor de gradiente de la función de pérdida, respectivamente. El **g** se calcula a partir de derivadas parciales de la función de pérdida. Se recomienda que el tamaño del mini-lote sea una potencia de 2 para fines de optimización de la GPU. En la red propuesta, *batch\_size = 128*

La ecuación anterior calcula las últimas actualizaciones de parámetros de capa. Entonces, ¿cómo ajustamos los parámetros de las capas anteriores? En este caso, se aplica la regla de diferenciación de la cadena para propagar las derivadas a las capas inferiores y calcular los gradientes en consecuencia. Este algoritmo se conoce como **retropropagación** en el Deep Learning.

Dado que la optimización se basa en la diferenciación, se deduce que un criterio importante de la función de pérdida es que debe ser suave o diferenciable. Esta es una importante restricción a tener en cuenta al introducir una nueva función de pérdida.

Dado el conjunto de datos de entrenamiento, la elección de la función de pérdida, el optimizador y el regularizador, el modelo ahora se puede entrenar llamando a la función fit()

*# loss function for one-hot vector*

*# use of adam optimizer*

*# accuracy is a good metric for classification tasks model.*

*compile(loss='categorical\_crossentropy',*

*optimizer='adam', metrics=['accuracy'])*

*# train the network*

*model.fit(x\_train, y\_train, epochs=20, batch\_size=batch\_size)*

Esta es otra característica útil de Keras. Con solo proporcionar los datos **x** y **y**, el número de epochs para entrenar y el tamaño del lote,fit() hace el resto. En otros marcos de deep learning, esto se traduce en múltiples tareas, como preparar los datos de entrada y salida en el formato adecuado, cargarlos, monitorearlos, etc. Si bien todo esto debe hacerse dentro de un bucle for, en Keras, todo se hace en una sola línea.

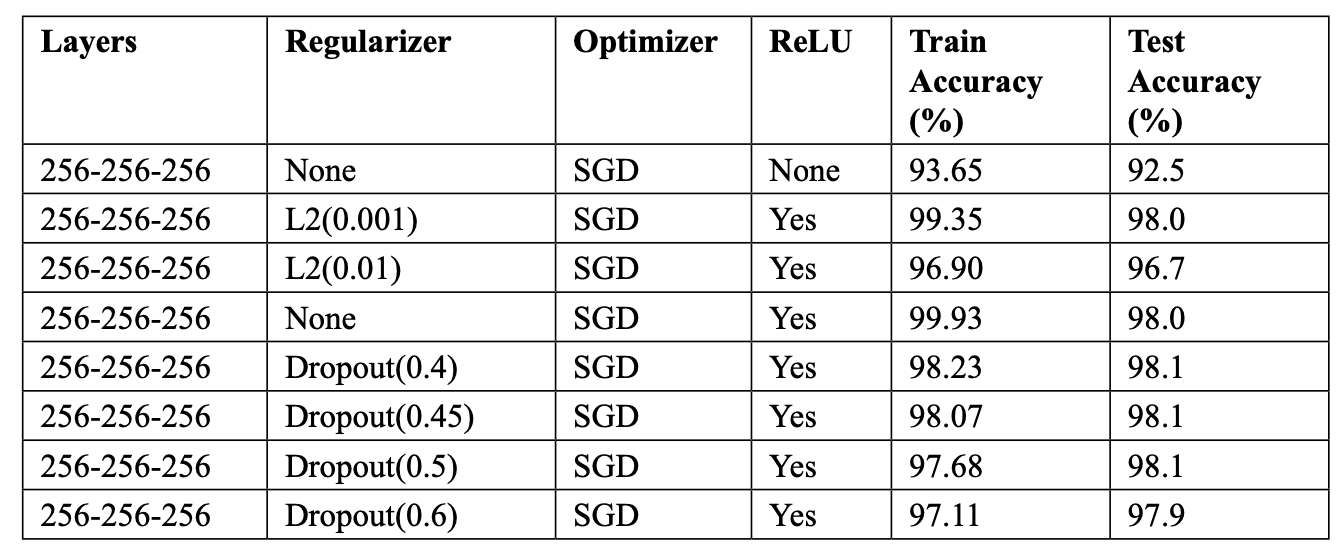
En la función fit(), un epoch es el muestreo completo de todos los datos de entrenamiento. El parámetro batch\_sizees el tamaño de muestra del número de entradas a procesar en cada paso de entrenamiento. Para completar una época, fit() procesará el número de pasos igual al tamaño del conjunto de datos del entrenamiento dividido por el tamaño del lote más 1 para compensar cualquier parte fraccional.

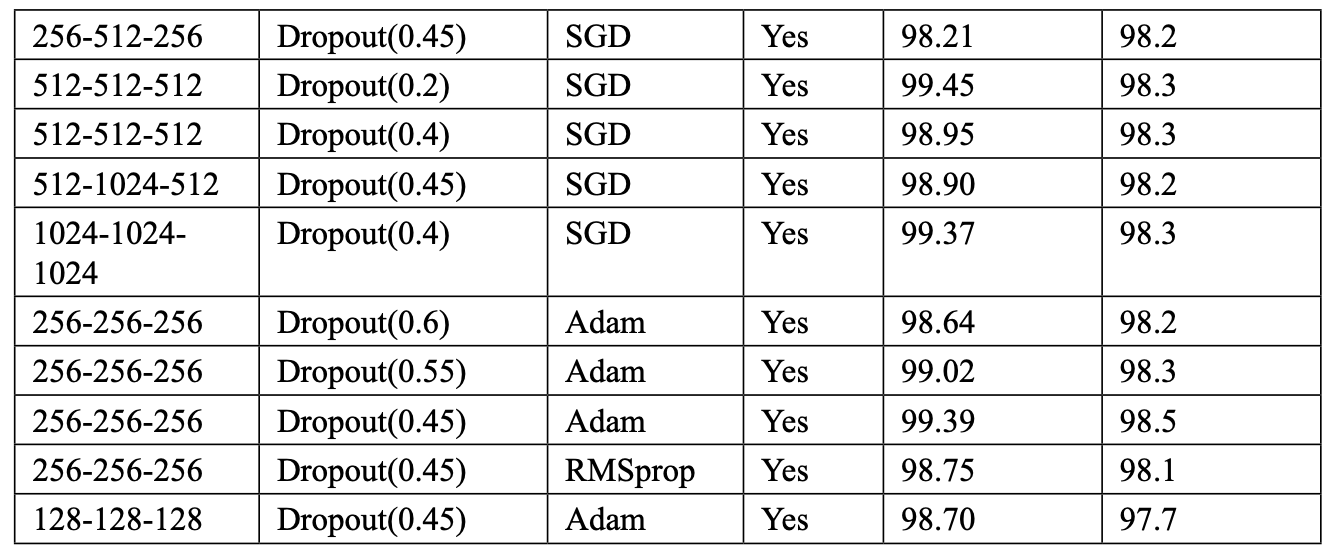
Después de entrenar el modelo, ahora podemos evaluar su rendimiento.

**Evaluación del desempeño**

En este punto, el modelo para el clasificador de dígitos MNIST ahora está completo. La evaluación del desempeño será el siguiente paso crucial para determinar si el modelo entrenado propuesto ha llegado a una solución satisfactoria. Entrenar el modelo durante 20 epochs será suficiente para obtener métricas de rendimiento comparables.

La siguiente tabla, muestra las diferentes configuraciones de red y las medidas de desempeño correspondientes. En Capas (Layers), se muestra el número de unidades para las capas 1 a 3. Para cada optimizador, se utilizan los parámetros predeterminados en tf.keras. Se pueden observar los efectos de variar el regularizador, el optimizador y el número de unidades por capa. Otra observación importante en la tabla es que las redes más grandes no necesariamente se traducen en un mejor desempeño.





El aumento de la profundidad de esta red no muestra beneficios adicionales en términos de precisión para los conjuntos de datos de entrenamiento y prueba. Por otro lado, un número menor de unidades, como 128, también podría reducir la precisión tanto de la prueba como del entrenamiento. La mejor precisión del entrenamiento al 99,93% se obtiene cuando se elimina el regularizador y se utilizan 256 unidades por capa. La precisión de la prueba, sin embargo, es mucho menor, 98.0%, como resultado del sobreajuste de la red.

La precisión más alta de la prueba es con el optimizador Adam y Dropout (0.45) al 98.5%. Técnicamente, todavía hay cierto grado de sobreajuste dado que su precisión de entrenamiento es del 99,39%. Tanto el entrenamiento como la precisión de la prueba son iguales al *98,2%* para 256-512-256, Dropout(0,45) y SGD. La eliminación de las capas Regularizer y ReLU da como resultado que tenga el peor rendimiento. En general, encontraremos que la capa Dropout tiene un mejor rendimiento que *l2.*

El ejemplo indica que es necesario mejorar la arquitectura de la red.

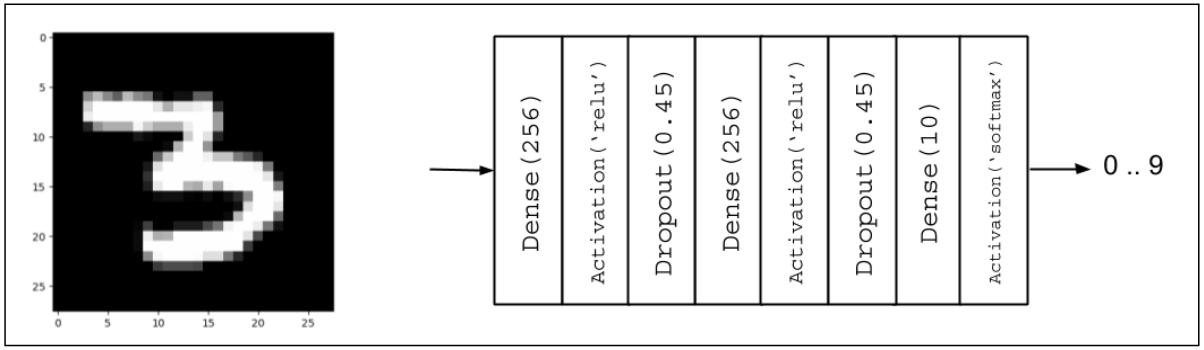
**Resumen del modelo**

El uso de la biblioteca de Keras nos proporciona un mecanismo rápido para verificar la descripción del modelo llamando

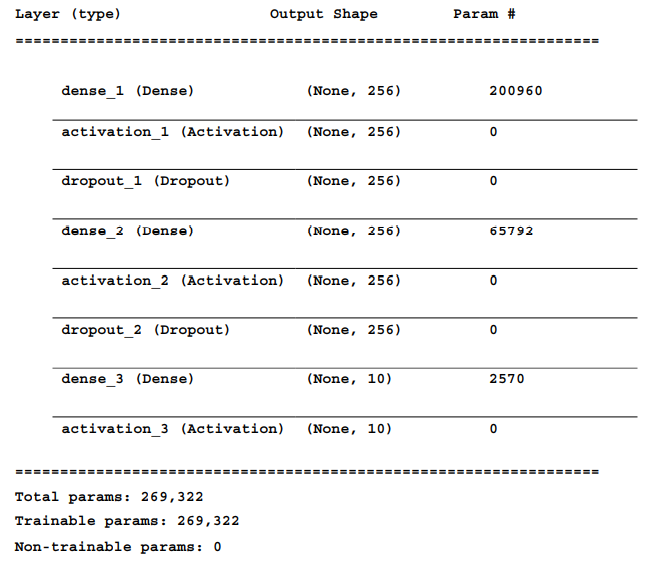
model.summary()

A continuación, se muestra el resumen del modelo de la red propuesta. Requiere un total de 269,322 parámetros. Esto es sustancial considerando que tenemos una tarea simple de clasificar los dígitos MNIST. Los MLP no son eficaces en función de los parámetros. El número de parámetros se puede calcular a partir de la figura vista al inicio, centrándose en cómo se calcula la salida del perceptrón. De la entrada a la capa Densa: 784 × 256 + 256 = 200,960. Desde la primera capa densa a la segunda capa densa: 256 × 256 + 256 = 65,792. Desde la segunda capa densa a la capa de salida: 10 × 256 + 10 = 2,570.

El total es 269,322



## Resumen de un modelo de clasificador de dígitos MLP MNIST:



# Red neuronal convolucional (CNN)

Ahora vamos a pasar a la segunda red neuronal artificial, CNN.

Las CNN son un tipo de red neuronal artificial (ANN); están vagamente inspirados en el concepto de que la corteza visual humana procesa imágenes y permite que nuestro cerebro reconozca objetos en el mundo e interactúe con ellos, lo que nos permite hacer una serie de cosas, como conducir, practicar deportes, leer, ver películas. , y así.

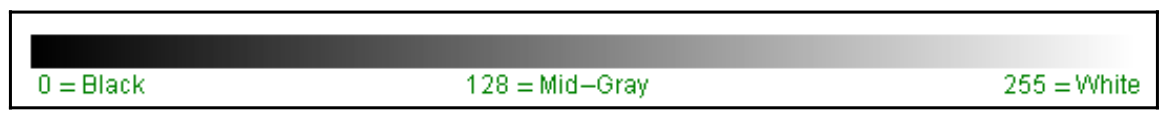
Se ha descubierto que en nuestro cerebro tienen lugar cálculos que se asemejan un poco a las convoluciones. Además, nuestros cerebros poseen células simples y complejas. Las celdas simples recogen características básicas, como bordes y curvas, mientras que las celdas complejas muestran invariancia espacial, mientras que también responden a las mismas señales que las celdas simples.

## Tipos de datos utilizados en ConvNets

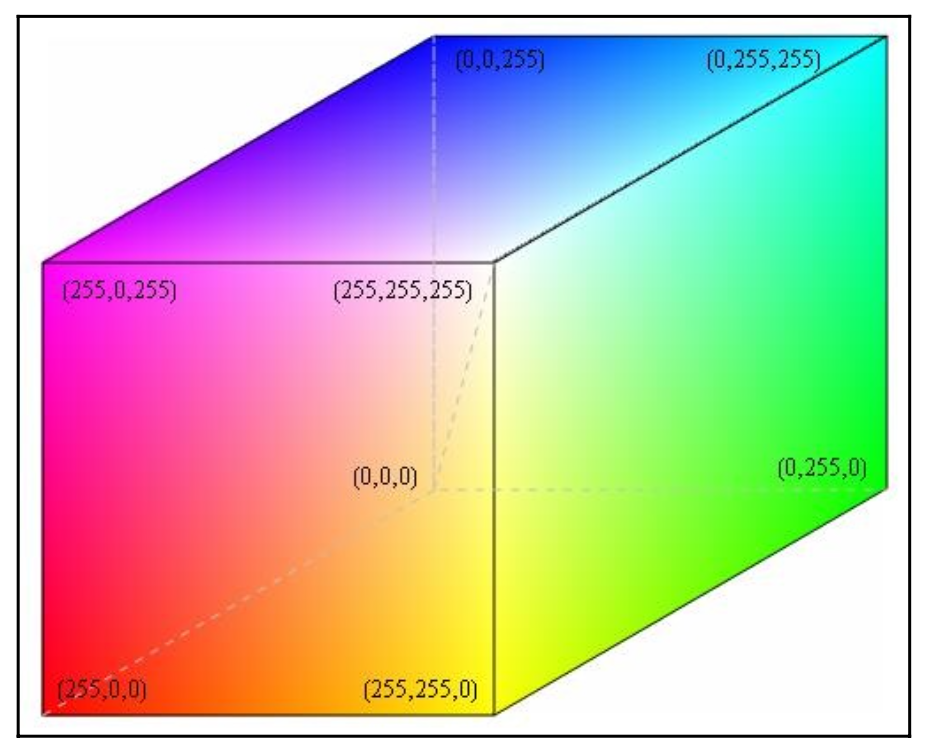
Las CNN funcionan excepcionalmente bien en tareas visuales, como la clasificación de objetos y el reconocimiento de objetos en imágenes y videos y el reconocimiento de patrones en música, clips de sonido, etc. Trabajan eficazmente en estas áreas porque son capaces de explotar la estructura de los datos para aprender sobre ellos. Esto significa que no podemos alterar las propiedades de los datos. Por ejemplo, las imágenes tienen una estructura fija y si modificáramos esto, la imagen dejaría de tener sentido. Esto difiere de las RNA, donde el orden de los vectores de características no importa. Por lo tanto, los datos de las CNN se almacenan en matrices multidimensionales.

En las computadoras, las imágenes están en escala de grises (blanco y negro) o en colores (RGB), y los videos (RGB-D) están formados por píxeles. Un píxel es la unidad más pequeña de una imagen digitalizada que se puede mostrar en una computadora y contiene valores en forma de [0, 255]. El valor de píxel representa su intensidad.

Si el valor de píxel es 0, entonces es negro, si es 128, entonces es gris, y si es 255, entonces es blanco. Podemos ver esto en la siguiente captura de pantalla:



Como podemos ver, las imágenes en escala de grises solo requieren 1 byte de datos, pero las imágenes en color, por otro lado, se componen de tres valores diferentes: rojo, azul y verde, ya que cualquier color se puede mostrar usando una combinación de estos tres. colores. Podemos ver el espacio de color en el siguiente diagrama:



Dependiendo de en qué parte del cubo estemos, obtenemos claramente un color diferente.

En lugar de mirarlo como un cubo o con diferentes intensidades de color, podemos verlo como si tuviera tres canales separados: rojo, azul y verde. Luego, cada píxel requiere 3 bytes de almacenamiento.

Normalmente, no podemos ver los píxeles individuales en las imágenes y videos que vemos en nuestros monitores porque tienen una resolución muy alta. Esto puede variar mucho, pero los píxeles suelen estar entre varios cientos y varios miles de **puntos** (píxeles) **por pulgada** (ppp).

Supongamos que tenemos una imagen en escala de grises con un tamaño de 512 × 512 × 1 (alto × ancho × canal).

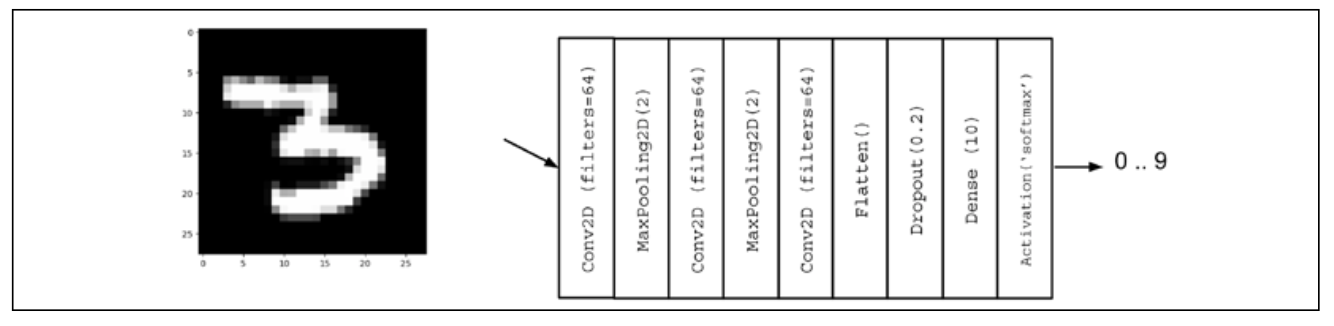
Podemos almacenarlo en un tensor bidimensional (matriz), donde cada valor de *i* y *j* es un píxel con cierta intensidad. Para almacenar esta imagen en nuestro disco, necesitamos 512 × 512 = 262,144 bytes.

Ahora, supongamos que tenemos una imagen en color con un tamaño de 512 × 512 × 3 (alto × ancho × canal). Podemos almacenarlo en un tensor tridimensional, donde cada valor de *i*, *j* y *k* es un píxel de color con cierta intensidad. Para almacenar esta imagen en nuestro disco, necesitaríamos 512 × 512 × 3 = 786,432 bytes, lo que nos dice que almacenar una imagen en color requiere mucho más espacio y, por lo tanto, más tiempo para procesar.

Un video en color se puede representar como una secuencia de cuadros (imágenes). Comenzamos haciendo que el tiempo sea discreto para que cada cuadro sea un paso de tiempo fijo aparte del otro. Podemos almacenar un video normal (escala de grises) en una matriz tridimensional, donde un eje representa la altura del marco, otro representa el ancho y el tercero representa la duración del tiempo.

Vamos a resolver el mismo problema de clasificación de dígitos MNIST, pero ahora usando una CNN.

La siguiente figura muestra el modelo CNN que usaremos para la clasificación de dígitos MNIST, mientras que su implementación se ilustra en el listado. Se necesitarán algunos cambios en el modelo anterior para implementar el modelo CNN. En lugar de tener un vector de entrada, el tensor de entrada ahora tiene nuevas dimensiones (height, width, channels) o (image\_size, image\_size, 1) = (28, 28, 1) para las imágenes MNIST en escala de grises. Será necesario cambiar el tamaño del tren y las imágenes de prueba para cumplir con este requisito de forma de entrada.



import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Activation, Dense, Dropout

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten

from tensorflow.keras.utils import to\_categorical, plot\_model

from tensorflow.keras.datasets import mnist

# load mnist dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# compute the number of labels

num\_labels = len(np.unique(y\_train))

# convert to one-hot vector

y\_train = to\_categorical(y\_train)

y\_test = to\_categorical(y\_test)

# input image dimensions

image\_size = x\_train.shape[1]

# resize and normalize

x\_train = np.reshape(x\_train,[-1, image\_size, image\_size, 1])

x\_test = np.reshape(x\_test,[-1, image\_size, image\_size, 1])

x\_train = x\_train.astype('float32') / 255

x\_test = x\_test.astype('float32') / 255

# network parameters

# image is processed as is (square grayscale)

input\_shape = (image\_size, image\_size, 1)

batch\_size = 128

kernel\_size = 3

pool\_size = 2

filters = 64

dropout = 0.2

# model is a stack of CNN-ReLU-MaxPooling

model = Sequential()

model.add(Conv2D(filters=filters,

kernel\_size=kernel\_size,

activation='relu',

input\_shape=input\_shape))

model.add(MaxPooling2D(pool\_size))

model.add(Conv2D(filters=filters,

kernel\_size=kernel\_size,

activation='relu'))

model.add(MaxPooling2D(pool\_size))

model.add(Conv2D(filters=filters,

kernel\_size=kernel\_size,

activation='relu'))

model.add(Flatten())

# dropout added as regularizer

model.add(Dropout(dropout))

# output layer is 10-dim one-hot vector

model.add(Dense(num\_labels))

model.add(Activation('softmax'))

model.summary()

plot\_model(model, to\_file='cnn-mnist.png', show\_shapes=True)

# loss function for one-hot vector

# use of adam optimizer

# accuracy is good metric for classification tasks

model.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

# train the network

model.fit(x\_train, y\_train, epochs=10, batch\_size=batch\_size)

\_, acc = model.evaluate(x\_test,

y\_test,

batch\_size=batch\_size,

verbose=0)

print("\nTest accuracy: %.1f%%" % (100.0 \* acc))

El principal cambio aquí es el uso de las capas Conv2D. La función de activación de ReLU ya es un argumento de Conv2D. La función ReLU se puede mostrar como una capa de activación (Activation) cuando la capa de normalización por lotes (Batch normalization) se incluye en el modelo.

Batch normalization se utiliza en CNN profundas para que se puedan utilizar grandes tasas de aprendizaje sin causar inestabilidad durante el entrenamiento.

## Convolución

Las capas en las CNN están conectadas a través de una operación lineal conocida como convolución, que es de donde proviene su nombre y lo que la convierte en una arquitectura tan poderosa para las imágenes.

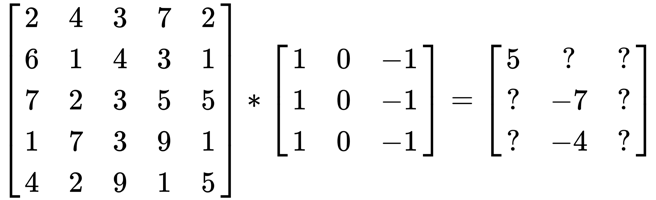
## Convoluciones bidimensionales

Matemáticamente, escribimos convoluciones como:

Lo que esto significa es que tenemos una función, ***f***, que es nuestra entrada y una función, ***g***, que es nuestro kernel (núcleo). Al **convolucionar**, recibimos un resultado (a veces denominado mapa de características).

Sin embargo, en las CNN, generalmente usamos convoluciones discretas, que se escriben de la siguiente manera:

Supongamos que tenemos una matriz bidimensional con una altura de 5 y un ancho de 5, y un kernel bidimensional con una altura de 3 y un ancho de 3. Entonces, la convolución y su salida se verán de la siguiente manera:

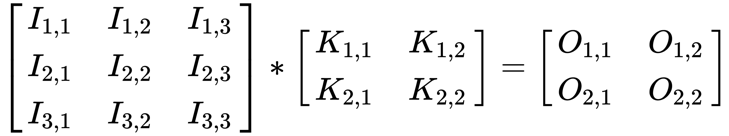


¿Qué valores irían en los campos faltantes?

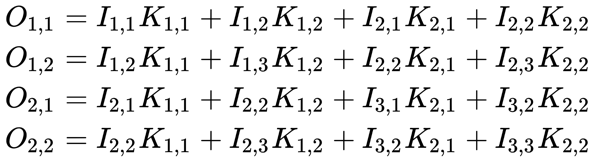
Como puede ver, el kernel se desliza sobre la entrada y produce un mapa de características con una altura de 3 y un ancho de 3. Este mapa de características nos dice el grado en que las funciones **f** y **g** se superponen cuando uno pasa sobre el otro. Podemos pensar en esto como escanear la entrada en busca de un patrón determinado; en otras palabras, el mapa de características busca el mismo patrón en diferentes lugares de la entrada.

Para comprender mejor cómo se mueve el kernel sobre la entrada, piense en una máquina de escribir. La convolución comienza en la parte superior izquierda, aplica la multiplicación y la suma por elementos, luego se mueve un paso hacia la derecha y se repite hasta que alcanza la posición más a la derecha sin ir más allá de los límites de la entrada. Luego se mueve hacia abajo una fila y repite este proceso hasta que alcanza la posición inferior derecha.

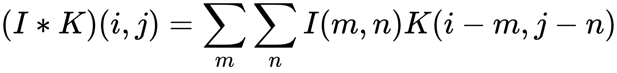
Suponga que ahora tenemos un tensor bidimensional de 3 × 3 como entrada y le aplicamos un núcleo de 2 × 2. Se verá de la siguiente manera:



Podemos escribir matemáticamente las salidas individuales en el mapa de características de la siguiente manera:



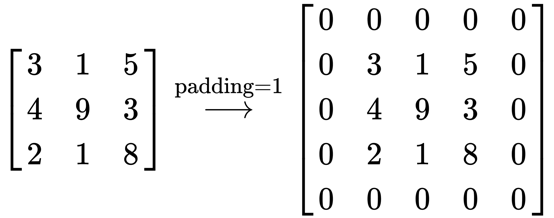
Ahora, podemos reescribir la ecuación de convolución discreta anterior de la siguiente manera:



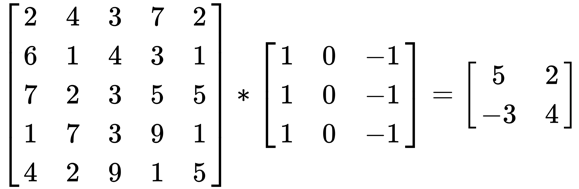
De la operación anterior, podemos decir que si seguimos aplicando convoluciones a los mapas de características, la altura y el ancho de cada capa disminuirán posteriormente. Entonces, a veces, es posible que deseemos preservar el tamaño de I después de la operación de convolución (especialmente si estamos construyendo una CNN muy profunda), en cuyo caso, rellenamos el exterior de la matriz con ceros. Lo que hace es aumentar el tamaño de la matriz antes de aplicar la operación de convolución.

Entonces, si I es una matriz n × n y nuestro kernel es una matriz k × k y queremos que nuestro mapa de características también sea n × n, entonces rellenamos I una vez, convirtiéndolo en una (n + 2) × (n +2) matriz. Ahora, después de convolucionar los dos, el mapa de características resultante tendrá un tamaño n × n.

La operación de relleno tiene el siguiente aspecto:



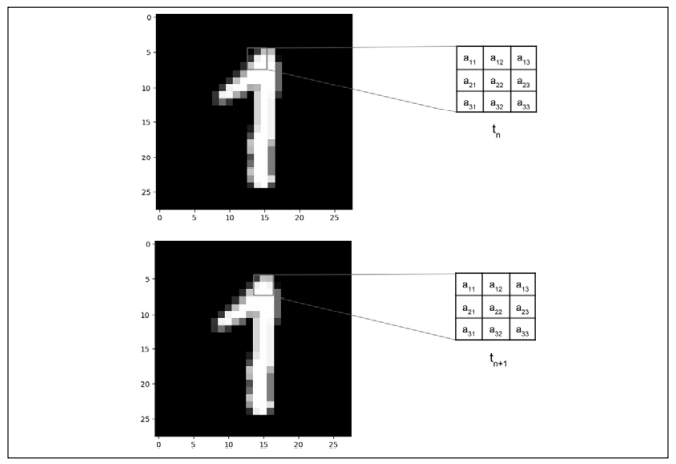
Si queremos reducir el tamaño del mapa de características, podemos usar un kernel más grande o podemos aumentar el tamaño de la zancada; cada uno dará un resultado diferente. Cuando la zancada es 1, deslizamos nuestro núcleo como de costumbre, uno a la vez. Sin embargo, cuando aumentamos el paso a 2, el núcleo salta dos posiciones cada vez. En este caso la matriz anterior se verá como:



Además, podemos repetir este proceso tantas veces como queramos, usando diferentes núcleos y produciendo múltiples mapas de características. Luego apilamos estas salidas juntas y formamos una matriz tridimensional de mapas de características, que llamamos capa.

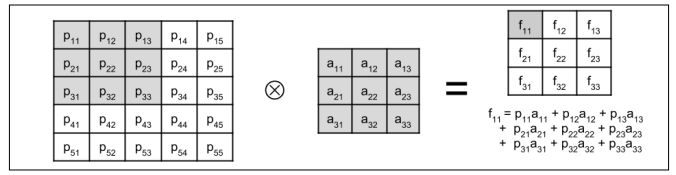
Por ejemplo, digamos que tenemos una imagen con un tamaño de 52 × 52 y un núcleo con un tamaño de 12 × 12 y un paso de 2. Aplicamos esto a nuestra entrada 15 veces y apilamos las salidas. Obtenemos un tensor tridimensional con un tamaño de .

Si, en el modelo MLP, el número de unidades caracteriza las capas densas (Dense Layers), el ***kernel*** (núcleo) caracteriza las operaciones de la CNN. Como se muestra en la figura, el kernel se puede visualizar como un parche o ventana rectangular que se desliza a través de toda la imagen de izquierda a derecha y de arriba a abajo. Esta operación se llama convolución. ***Transforma la imagen de entrada en un mapa de características, que es una representación de lo que el núcleo ha aprendido de la imagen de entrada.*** Luego, el mapa de características se transforma en otro mapa de características en la capa siguiente y así sucesivamente. El número de mapas de características generados por Conv2D está controlado por el argumento de filtros (filters).



La convolución se muestra en los pasos y donde el núcleo se movió un paso de 1 píxel hacia la derecha.

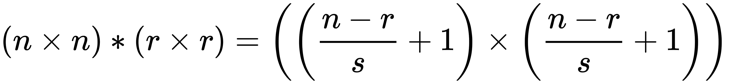
El cálculo involucrado en la convolución se muestra en la siguiente figura.



Para simplificar, se ilustra una imagen de entrada de 5 × 5 (o mapa de características de entrada) donde se aplica un kernel de 3 × 3. El mapa de características resultante se muestra después de la convolución. El valor de un elemento del mapa de características está sombreado. Notará que el mapa de características resultante es más pequeño que la imagen de entrada original, esto se debe a que la convolución solo se realiza en elementos válidos. El kernel no puede ir más allá de los bordes de la imagen. Si las dimensiones de la entrada deben ser las mismas que las de los mapas de características de salida, Conv2D acepta la opción padding = 'same'. La entrada se rellena con ceros alrededor de sus bordes para mantener las dimensiones sin cambios después de la convolución.

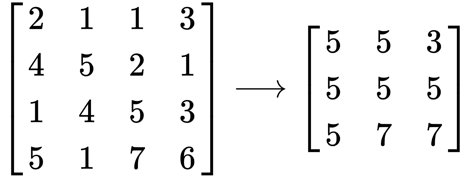
## Pooling (Agrupación)

Otra operación de uso frecuente en las CNN se conoce como **agrupación** (**submuestreo** o **reducción de muestreo**). Esto funciona de manera similar a la operación de convolución, **excepto que reduce el tamaño del mapa de características deslizando una ventana a través del mapa de características y promedia todos los valores dentro de cada ventana en cada paso o genera el valor máximo.** La operación de agrupación se diferencia de la convolución en que no tiene ningún parámetro y, por lo tanto, **no se puede aprender ni ajustar**. Podemos calcular el tamaño del mapa de características después de la agrupación, de la siguiente manera:

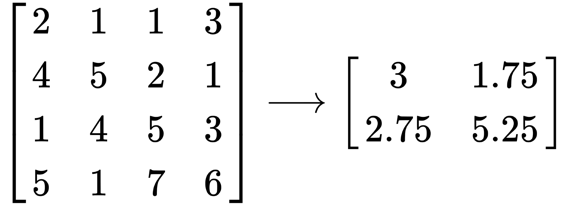


Aquí, I es un tensor bidimensional en forma de n × n, la operación de agrupación es un tensor bidimensional en forma de r × r, y ***s*** es la zancada/paso.

A continuación, se muestra un ejemplo de ***agrupación máxima*** con un paso de 1:



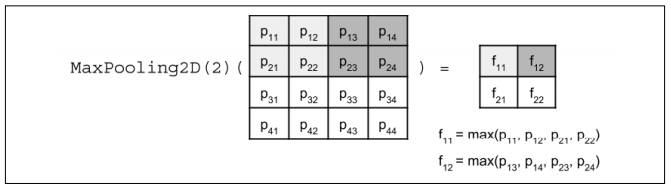
A continuación, se muestra un ejemplo de ***agrupación promedio*** con un paso de 2:



Como regla general, se ha descubierto que la operación de agrupación máxima funciona mejor.

## Operaciones de agrupación

El último cambio es la adición de una capa MaxPooling2D con el argumento pool\_size = 2. MaxPooling2D comprime cada mapa de características. Cada parche de tamaño pool\_size × pool\_size se reduce a 1 punto del mapa de características. El valor es igual al máximo valor de puntos de característica dentro del parche. MaxPooling2D se muestra en la siguiente figura para dos parches:

****

La importancia de MaxPooling2D es la reducción del tamaño del mapa de características, lo que se traduce en un aumento del tamaño del campo receptivo. Por ejemplo, después de MaxPooling2D(2), el kernel de 2 × 2 ahora se convoluciona aproximadamente con un parche de 4 × 4. La CNN ha aprendido un nuevo conjunto de mapas de características para un tamaño de campo receptivo diferente.

Hay otros medios de agrupación y compresión. Por ejemplo, para lograr una reducción de tamaño del 50% como MaxPooling2D(2), AveragePooling2D(2) toma el promedio de un parche en lugar de encontrar el máximo. La convolución escalonada, Conv2D (strides = 2,…), saltará cada dos píxeles durante la convolución y seguirá teniendo el mismo efecto de reducción de tamaño del 50%. Existen diferencias sutiles en la efectividad de cada técnica de reducción.

En Conv2D y MaxPooling2D, tanto pool\_size como kernel pueden ser no cuadrados. En estos casos, se deben indicar tanto el tamaño de las filas como de las columnas. Por ejemplo, pool\_ size = (1, 2) y kernel = (3, 5).

El resultado de la última operación MaxPooling2D es una pila de mapas de características. La función de Flatten es convertir la pila de mapas de características en un formato vectorial que sea adecuado para las capas Dropout o Dense, similar a la capa de salida del modelo MLP.

## Tamaño de convolución y agrupación

Ahora que conocemos los distintos tipos de convolución y agrupación, es hora de hablar sobre un tema muy importante asociado con ellos: su tamaño. Como ha visto, cuando aplicamos una convolución a una imagen, la salida era de un tamaño menor que la entrada. El tamaño de salida está determinado por el tamaño del kernel, el paso y si tenemos o no relleno. Estas son cosas muy importantes a tener en cuenta al diseñar las CNN.

Hay varios tamaños de circunvoluciones que se utilizan en la práctica, siendo los más utilizados 7 × 7, 5 × 5 y 3 × 3. Sin embargo, también podemos utilizar otros tamaños, incluidos, entre otros, 11 × 11, 13 × 13, 9 × 9, 17 × 17, etc.

En la práctica, generalmente usamos convoluciones más grandes con un paso más grande para generar un mapa de características de un tamaño más pequeño para reducir la restricción computacional y, por defecto, usar los núcleos 3 × 3 y 5 × 5 más. Esto se debe a que son computacionalmente más factibles. Generalmente, tener un kernel más grande nos permitirá ver un espacio más grande en la imagen y capturar más relaciones, pero tener múltiples kernels de 3 × 3 ha demostrado tener un rendimiento similar mientras que es menos intensivo en computación, lo que preferimos.

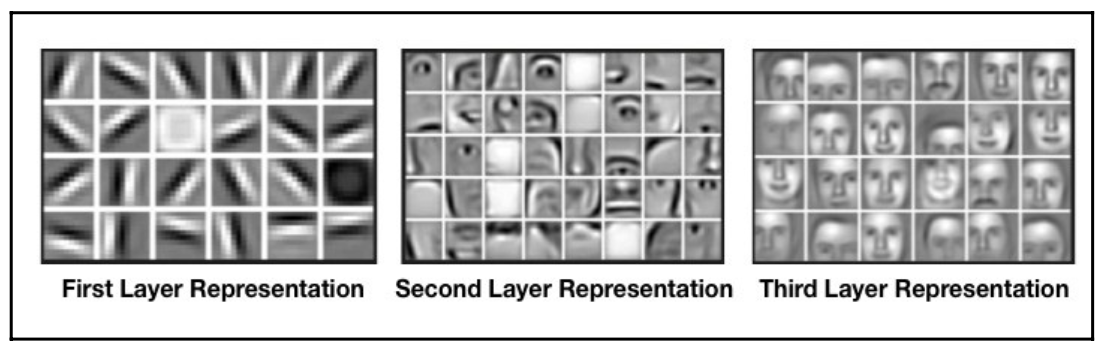
## Trabajar con la arquitectura ConvNet

Ahora que conocemos todos los diferentes componentes que componen una ConvNet, podemos unirlo todo y ver cómo construir una CNN profunda. En esta sección, construiremos una arquitectura completa y observaremos cómo funciona la propagación hacia adelante y cómo decidimos la profundidad de la red, el número de núcleos a aplicar, cuándo y por qué usar la agrupación, etc.

En general, lo que aprende la arquitectura se puede demostrar con el siguiente flujo:

Imagen de entrada→degradados+bordes→texturas→patrones→características básicas del objeto→objeto→salida

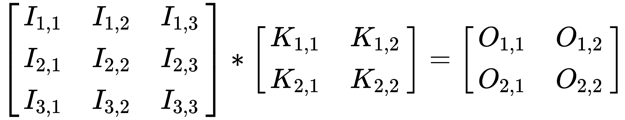
Como puede ver en el flujo anterior, las características aumentan en complejidad en las últimas capas. Lo que esto significa es que las primeras capas (las más cercanas a la capa de entrada) aprenden características muy básicas, como bordes y líneas, texturas o cómo se diferencian ciertos colores. Las últimas capas toman el mapa de características de la capa anterior como entrada y aprenden patrones más complejos de él. Por ejemplo, si creamos un modelo de reconocimiento facial, la primera capa aprendería las líneas, curvas y degradados más simples posibles. La siguiente capa tomaría los mapas de características de la capa anterior y los usaría para aprender características más complejas, como el cabello y las cejas. La capa posterior aprendería características aún más complejas, como ojos, narices, oídos, etc.



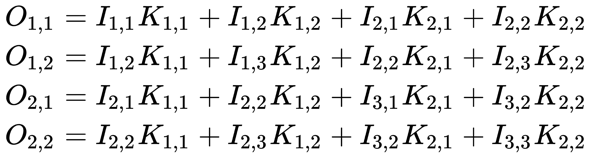
## Entrenamiento y optimización

¿Necesitamos un algoritmo completamente nuevo para facilitar nuestro entrenamiento y optimización? ¡No! Todavía podemos usar la retropropagación y el descenso de gradiente para calcular el error, diferenciarlo de las capas anteriores y actualizar los pesos para acercarnos lo más posible a los óptimos globales.

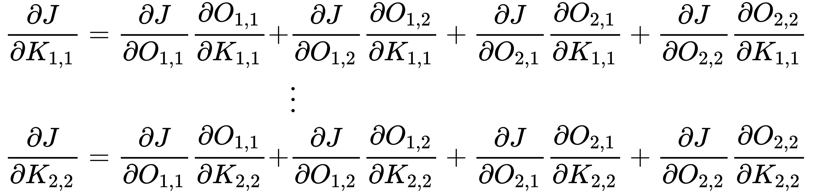
Pero antes de continuar, veamos cómo funciona la retropropagación en las CNN, particularmente con los núcleos. Repasemos el ejemplo que usamos anteriormente en este capítulo, donde convolucionamos una entrada de 3 × 3 con un kernel de 2 × 2, que se veía de la siguiente manera:



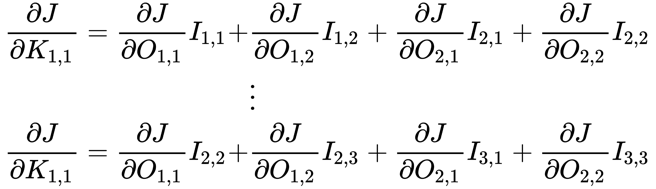
Expresamos cada elemento de la matriz de salida como:



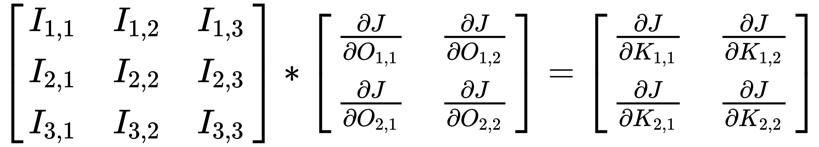
En las CNN, encontramos el gradiente del error con respecto al kernel. Dado que nuestro núcleo tiene cuatro elementos, las derivadas tienen el siguiente aspecto:



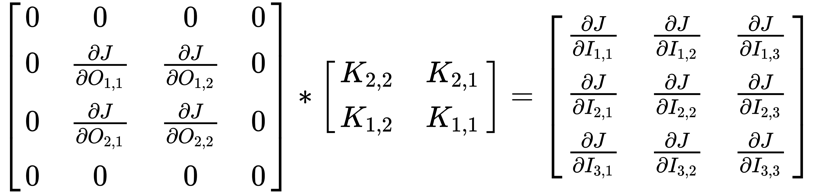
Si observamos estas ecuaciones con atención, que representan la salida del cálculo feedforward, podemos ver que al tomar la derivada parcial con respecto a cada elemento del núcleo, obtenemos el elemento de entrada respectivo, , del que depende. Si sustituimos este valor nuevamente en las derivadas, podemos simplificarlas para obtener lo siguiente:



Podemos simplificar esto aún más reescribiéndolo como una operación de convolución. Esto se ve como sigue:



Pero, ¿y si quisiéramos encontrar la derivada con respecto a la entrada? Bueno, nuestra matriz jacobiana ciertamente se vería un poco diferente. Tendríamos una matriz de 3 × 3 ya que hay nueve elementos en la matriz de entrada:

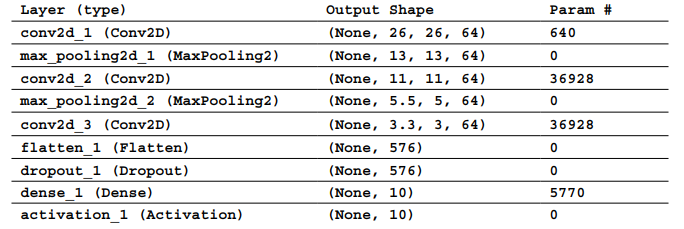


Podemos verificar esto si lo derivamos nosotros mismos a mano a través de las ecuaciones anteriores. Sin embargo, ahora prestemos especial atención al kernel que usamos. Si lo miramos con atención, casi parece el determinante, pero eso no es lo que es. Simplemente giramos (es decir, transpusimos) el núcleo 180 ° para poder calcular los gradientes.

Esta es una vista muy simplificada de cómo funciona la propagación hacia atrás en las CNN; lo hemos dejado simple porque el resto funciona exactamente como lo hizo en las FNN.

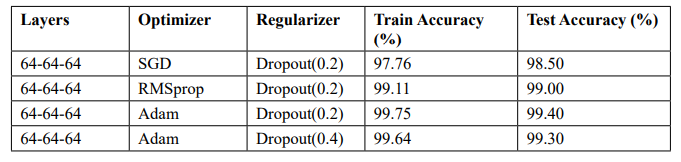
## Performance evaluation and model summary

Como se muestra en el listado, el modelo CNN anterior requiere un número menor de parámetros en 80,226 en comparación con 269,322 cuando se utilizan capas MLP. La capa conv2d\_1 tiene 640 parámetros porque cada kernel tiene 3 × 3 = 9 parámetros, y cada uno de los 64 mapas de características tiene un kernel y un parámetro de sesgo. El número de los parámetros para otras capas de convolución se pueden calcular de manera similar.





La siguiente tabla muestra una precisión de prueba máxima del 99,4%, que se puede lograr para una red de 3 capas con 64 mapas de características por capa utilizando el optimizador Adam con dropout = 0,2. Las CNN son más eficientes en cuanto a parámetros y tienen una mayor precisión que las MLP. Asimismo, las CNN también son adecuadas para aprender representaciones a partir de datos secuenciales, imágenes y videos.



# Red Neuronal Recurrente (RNN)

Las redes neuronales que estudiaremos en este capítulo son muy efectivas para datos secuenciales y se utilizan en aplicaciones como comercio algorítmico, subtítulos de imágenes, clasificación de sentimientos, traducción de idiomas, clasificación de videos, etc.

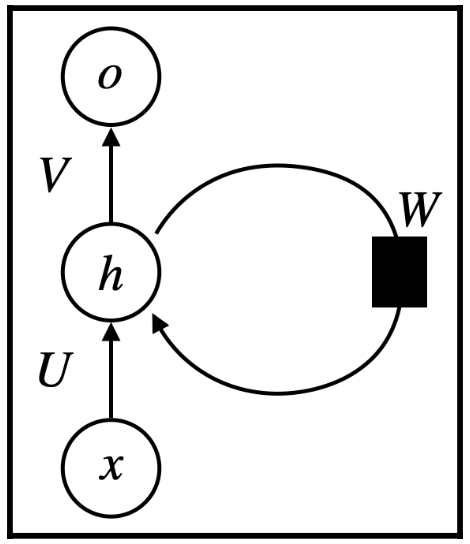
En las redes neuronales regulares, se asume que todas las entradas y salidas son independientes, pero en las RNN, **cada salida depende de la anterior, lo que les permite capturar dependencias en secuencias, como en el lenguaje, donde la siguiente palabra depende de la palabra anterior y la anterior**.

Los RNN al igual que CNN están inspirados biológicamente. La necesidad de esta forma de red neuronal surge del hecho de que las redes neuronales difusas (FNN) no pueden capturar dependencias basadas en el tiempo en los datos.

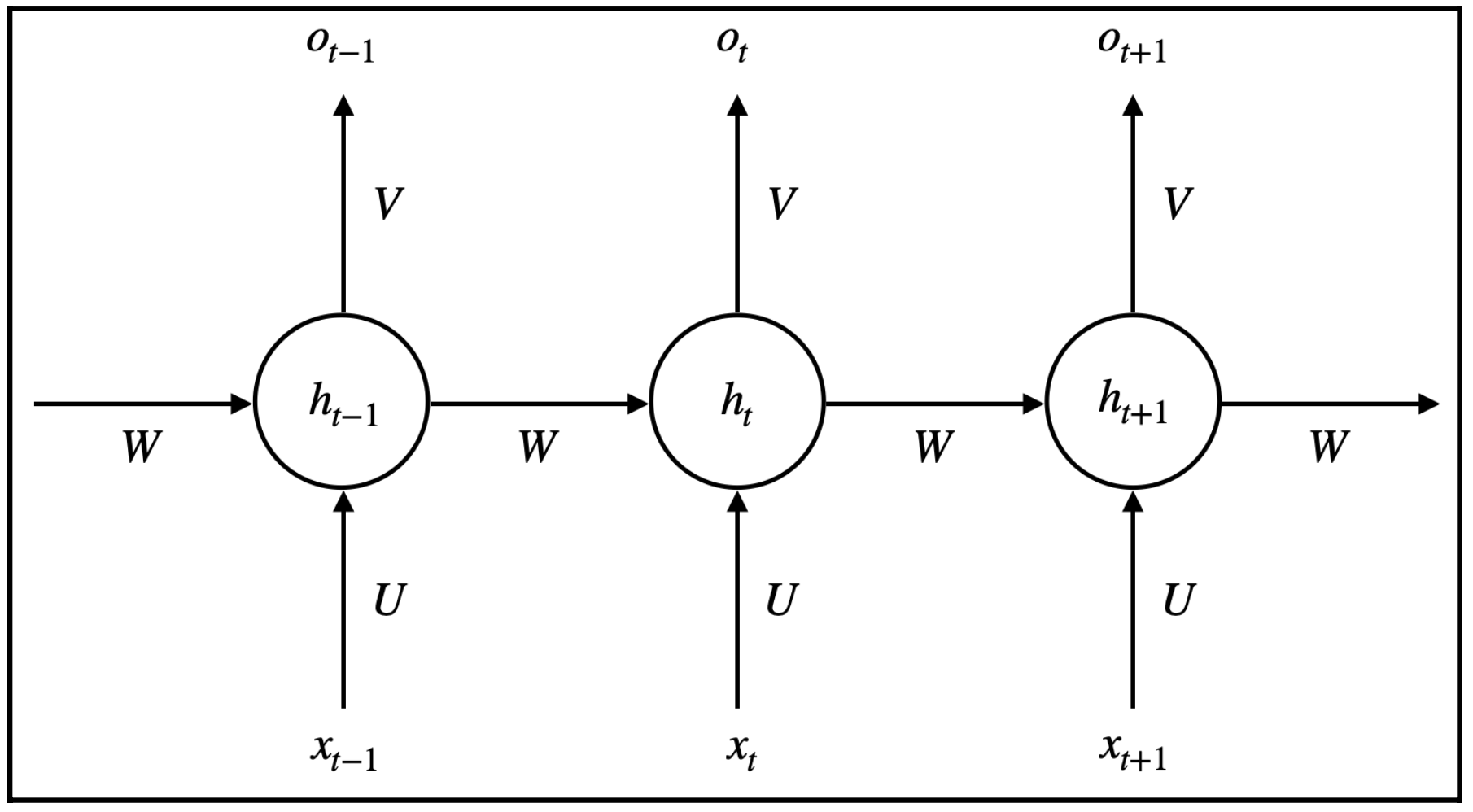
La palabra recurrente en el nombre de esta red neuronal proviene del hecho de que tiene conexiones cíclicas y se realiza el mismo cálculo en cada elemento de la secuencia. Esto le permite aprender (o memorizar) partes de los datos para hacer predicciones sobre el futuro. La ventaja de un RNN es que puede escalar a secuencias mucho más largas que los modelos que no se basan en secuencias.

## Vanilla RNNs

Esta es la versión más básica de una RNN, tiene el siguiente aspecto:



Esto nos resulta familiar, pues si removiéramos el ciclo (W) tendría la misma apariencia de un red neuronal tradicional con una capa oculta. Ahora si “desenrolláramos” este ciclo y viéramos toda la red neuronal tendría este aspecto:



Donde los parámetros es la entrada en el paso del tiempo t, es el estado oculto en el paso de tiempo t y es la salida en el paso del tiempo t.

En el diagrama anterior, podemos observar que se realiza el mismo cálculo en la entrada en cada paso de tiempo. Los parámetros (U, V y W) permanecen iguales en cada paso de tiempo. Debido a esto, los RNN consumen más memoria y deben entrenarse durante más tiempo en comparación con los CNN. También es importante tener en cuenta que en los RNN, el intervalo de tiempo no se corresponde necesariamente con el tiempo del mundo real; simplemente significa que la secuencia de entrada tiene una longitud t.

Pero, ¿por qué estos pesos siguen siendo los mismos en todos los pasos de tiempo? ¿Por qué no podemos tener parámetros separados que deben aprenderse en diferentes pasos de tiempo? La razón de esto es que los parámetros separados no pueden generalizarse a longitudes de secuencia que no se encuentran durante el proceso de entrenamiento. Tener los mismos tres pesos compartidos en la secuencia y en diferentes pasos de tiempo permite que la red maneje información que puede ocurrir en múltiples posiciones, como suele ocurrir en el lenguaje. Por ejemplo, *el* puede aparecer en varias posiciones en una oración dada y el RNN debe poder reconocerlo y extraerlo independientemente de la posición o posiciones en que se encuentre.

Veamos qué sucede en cada paso de tiempo de las RNN, desde la entrada hasta la salida a través de todas las capas ocultas.

Matemáticamente, podemos calcular el estado oculto en cada paso de tiempo usando la siguiente ecuación:

page256image4419776

Aquí, es una no linealidad, como ReLU, tanh o sigmoide. La salida se calcula de la siguiente manera:

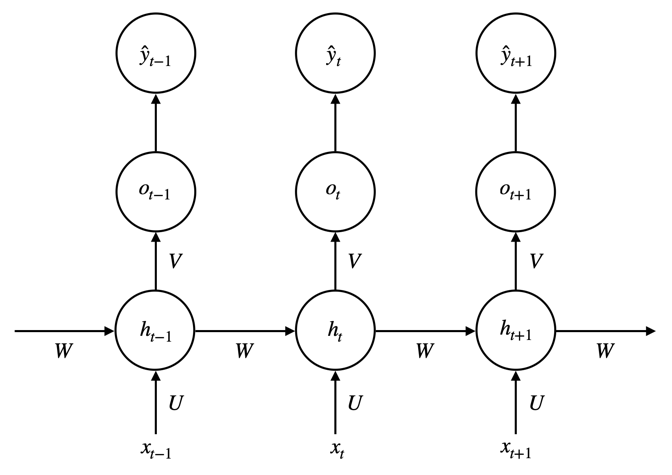
page256image4421856

Podemos calcular el vector de probabilidad de la salida usando una función no lineal, f2, (como softmax), de la siguiente manera:

page256image19511936

Al usar estas ecuaciones y aplicarlas repetidamente, podemos calcular los estados ocultos y las salidas en cada paso de tiempo.

Entonces, la RNN se ve así:

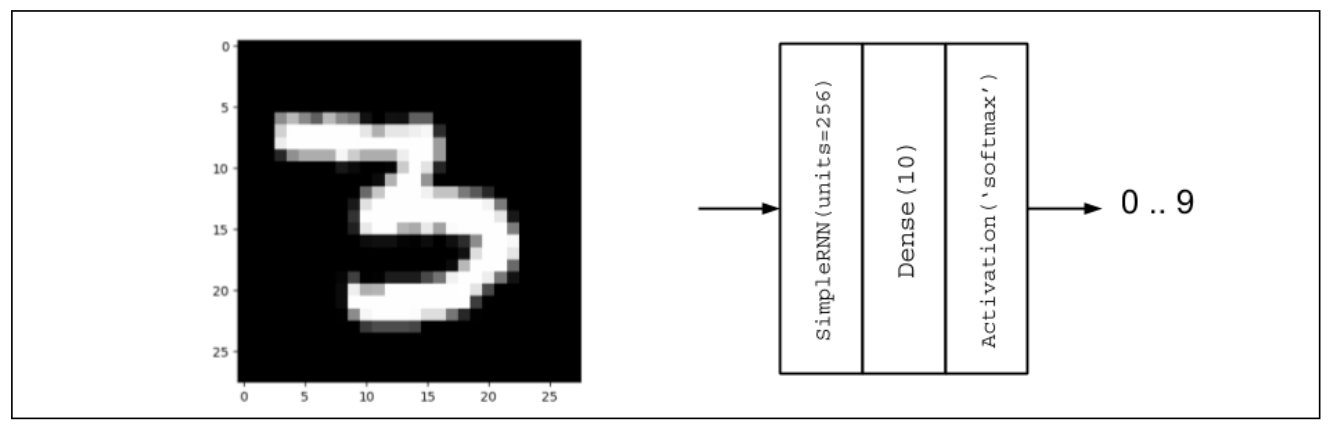


## El tipo de datos utilizados en RNNs

RNN se utilizan con frecuencia para tareas como el procesamiento del lenguaje natural, la traducción automática y el comercio algorítmico, y han dado resultados tremendos en ellas. Para estas tareas, necesitamos datos **secuenciales o de series de tiempo**, es decir, los datos tienen un orden fijo. Por ejemplo, los idiomas y la música tienen un orden fijo. Cuando hablamos o escribimos oraciones, siguen un marco, que es lo que nos permite comprenderlas. Si rompemos las reglas y mezclamos palabras que no se correlacionan, entonces la oración ya no tiene sentido.

Estructuraremos este tipo de datos de manera similar a como lo haríamos para convoluciones unidimensionales. Sin embargo, en lugar de tener un kernel que convoluciona sobre los datos, el RNN (con el que nos familiarizaremos en breve) tomará la misma entrada, donde el nodo corresponde al paso de tiempo de los datos (esto se aclarará momentáneamente).

Las RNN son una familia de redes que son adecuadas para aprender representaciones de datos secuenciales como texto en procesamiento de lenguaje natural (NLP) o un flujo de datos de sensores en instrumentación. Si bien cada muestra de datos de MNIST no es de naturaleza secuencial, no es difícil imaginar que cada imagen pueda interpretarse como una secuencia de filas o columnas de píxeles. Por lo tanto, un modelo basado en RNN puede procesar cada imagen MNIST como una secuencia de vectores de entrada de 28 elementos con pasos de tiempo iguales a 28. La siguiente lista muestra el código para el modelo RNN de la figura:



import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Activation, SimpleRNN

from tensorflow.keras.utils import to\_categorical, plot\_model

from tensorflow.keras.datasets import mnist

# load mnist dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# compute the number of labels

num\_labels = len(np.unique(y\_train))

# convert to one-hot vector

y\_train = to\_categorical(y\_train)

y\_test = to\_categorical(y\_test)

# resize and normalize

image\_size = x\_train.shape[1]

x\_train = np.reshape(x\_train,[-1, image\_size, image\_size])

x\_test = np.reshape(x\_test,[-1, image\_size, image\_size])

x\_train = x\_train.astype('float32') / 255

x\_test = x\_test.astype('float32') / 255

# network parameters

input\_shape = (image\_size, image\_size)

batch\_size = 128

units = 256

dropout = 0.2

# model is RNN with 256 units, input is 28-dim vector 28 timesteps

model = Sequential()

model.add(SimpleRNN(units=units,

dropout=dropout,

input\_shape=input\_shape))

model.add(Dense(num\_labels))

model.add(Activation('softmax'))

model.summary()

plot\_model(model, to\_file='rnn-mnist.png', show\_shapes=True)

# loss function for one-hot vector

# use of sgd optimizer

# accuracy is good metric for classification tasks

model.compile(loss='categorical\_crossentropy',

optimizer='sgd',

metrics=['accuracy'])

# train the network

model.fit(x\_train, y\_train, epochs=20, batch\_size=batch\_size)

\_, acc = model.evaluate(x\_test,

y\_test,

batch\_size=batch\_size,

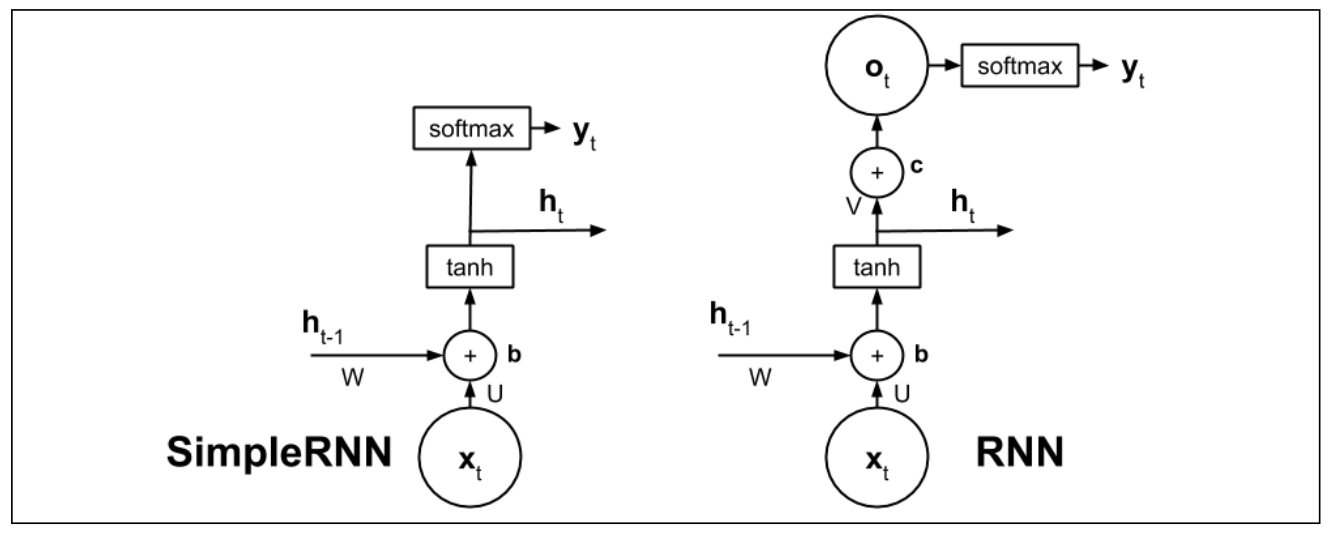
verbose=0)

print("\nTest accuracy: %.1f%%" % (100.0 \* acc))

Hay dos diferencias principales entre el clasificador RNN y los dos modelos anteriores. Primero está input\_shape = (image\_size, image\_size), que en realidad es input\_shape = (timesteps, input\_dim) o una secuencia de vectores input\_dim-dimension de longitud de pasos de tiempo. El segundo es el uso de una capa SimpleRNN para representar una celda RNN con units = 256. La variable de unidades representa el número de unidades de salida. Si la CNN se caracteriza por la convolución de los núcleos a través del mapa de características de entrada, la salida de la RNN es una función no solo de la entrada actual, sino también de la salida anterior o del estado oculto. Dado que la salida anterior también es una función de la entrada anterior, la salida actual también es una función de la salida y la entrada anteriores y así sucesivamente. La capa SimpleRNN en Keras es una versión simplificada del verdadero RNN. La siguiente ecuación describe la salida de SimpleRNN:

En esta ecuación, **b** es el sesgo, mientras que **W** y **U** se denominan kernel recurrente (pesos para la salida anterior) y kernel (pesos para la entrada actual), respectivamente. El subíndice t se utiliza para indicar la posición en la secuencia. Para una capa SimpleRNN con units = 256, el número total de parámetros es 256 + 256 × 256 + 256 × 28 = 72,960, correspondientes a las contribuciones **b**, **W** y **U**.

La siguiente figura muestra los diagramas de SimpleRNN y RNN cuando se utilizan para tareas de clasificación. Lo que hace SimpleRNN más simple que un RNN es la ausencia de los valores de salida **o\_t = Vh\_t + c** antes de que se calcule la función softmax:



Las RNN pueden ser inicialmente más difíciles de entender en comparación con las MLP o las CNN. En un MLP, el perceptrón es la unidad fundamental. Una vez que se comprende el concepto de perceptrón, un MLP es solo una red de perceptrones. En una CNN, el kernel es un parche o ventana que se desliza a través del mapa de características para generar otro mapa de características. En un RNN, lo más importante es el concepto de bucle automático. De hecho, solo hay una celda.

La ilusión de múltiples celdas aparece porque una celda existe por paso de tiempo, pero en realidad es la misma celda que se reutiliza repetidamente a menos que se desenrolle la red. Las redes neuronales subyacentes de los RNN se comparten entre las células.

El siguiente resumen indica que usar SimpleRNN requiere un menor número de parámetros.

**Layer (type) Output Shape Param #**

**=================================================================**

**simple\_rnn\_1 (SimpleRNN) (None, 256) 72960**

**dense\_1 (Dense) (None, 10) 2570**

**activation\_1 (Activation) (None, 10) 0**

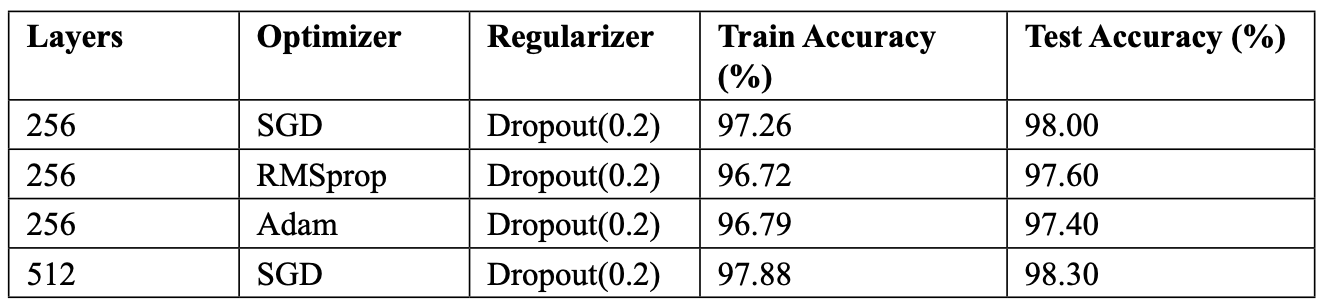
**=================================================================**

**Total params: 75,530**

**Trainable params: 75,530**

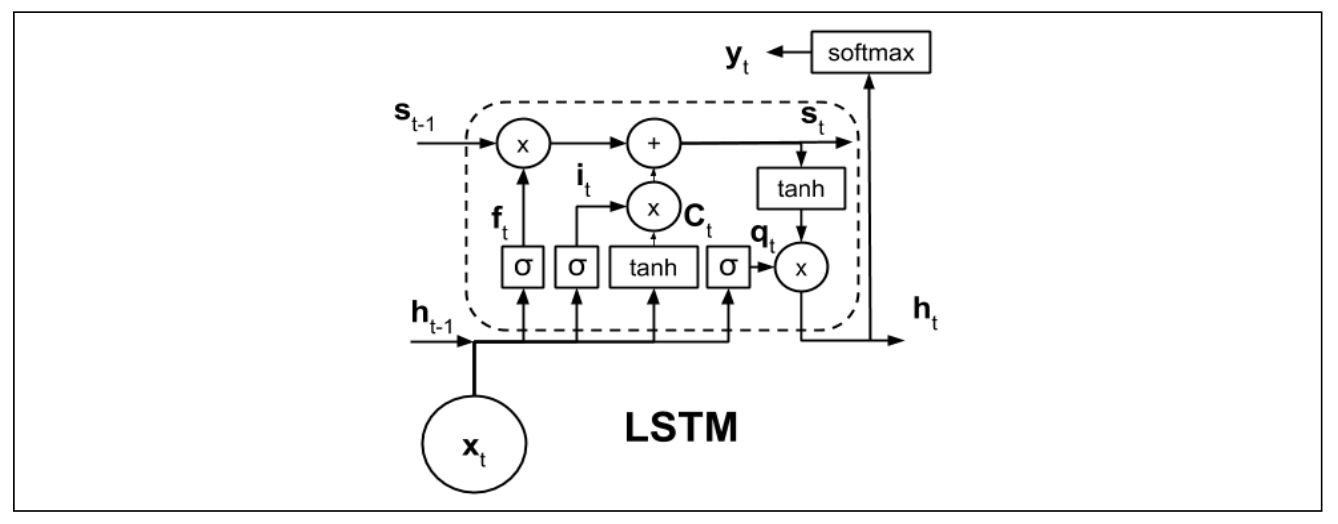
**Non-trainable params: 0**

La siguiente tabla muestra que SimpleRNN tiene la precisión más baja entre las redes presentadas:



En muchas redes neuronales profundas, se utilizan con más frecuencia otros miembros de la familia RNN. Por ejemplo, **Long Short-Term Memory (LSTM)** se ha utilizado tanto en la traducción automática como en los problemas de respuesta a preguntas. LSTM aborda el problema de la dependencia a largo plazo o el recuerdo de información pasada relevante para la salida actual.

A diferencia de un RNN o un SimpleRNN, la estructura interna de la celda LSTM es más compleja. La siguiente figura muestra un diagrama de LSTM. LSTM utiliza no solo la entrada actual y las salidas pasadas o estados ocultos, sino que introduce un estado de celda, s\_t, que transporta información de una celda a otra. El flujo de información entre los estados de la celda está controlado por tres puertas, f\_t, i\_t y q\_t. Las tres puertas tienen el efecto de determinar qué información debe retenerse o reemplazarse y la cantidad de información en la entrada pasada y actual que debe contribuir al estado o salida actual de la celda. No discutiremos los detalles de la estructura interna de la celda LSTM aquí. Sin embargo, se puede encontrar una guía intuitiva de LSTM en http://colah.github.io/ posts / 2015-08-Understanding-LSTMs.



Hay muchas otras formas de configurar RNN. Una forma es hacer un modelo RNN que sea bidireccional. Por defecto, los RNN son unidireccionales en el sentido de que la salida actual solo está influenciada por los estados pasados ​​y la entrada actual.

En los RNN bidireccionales, los estados futuros también pueden influir en los estados presentes y pasados ​​al permitir que la información fluya hacia atrás. Los productos anteriores se actualizan según sea necesario en función de la nueva información recibida. Los RNN se pueden convertir en bidireccionales llamando a una función contenedora. Por ejemplo, la implementación de LSTM bidireccional es Bidireccional (LSTM ()).

Para todos los tipos de RNN, aumentar el número de unidades también aumentará la capacidad. Sin embargo, otra forma de aumentar la capacidad es apilando las capas RNN. Sin embargo, debe tenerse en cuenta que, como regla general, la capacidad del modelo solo debe aumentarse si es necesario. El exceso de capacidad puede contribuir al sobreajuste y, como resultado, puede llevar tanto a un tiempo de entrenamiento más largo como a un rendimiento más lento durante la predicción.

**Long-Short-Memory**

Como vimos anteriormente, el RNN estándar tiene algunas limitaciones; en particular, sufren el problema del gradiente de desaparición. La arquitectura LSTM fue propuesta por Jürgen Schmidhuber (ftp: / / ftp. Idsia. Ch / pub / juergen / lstm. Pdf) como una solución al problema de dependencia a largo plazo que enfrentan los RNN.

Las células LSTM se diferencian de las células RNN de vainilla en algunos aspectos. En primer lugar, contienen lo que llamamos un bloque de memoria, que es básicamente un conjunto de subredes conectadas de forma recurrente. En segundo lugar, cada uno de los bloques de memoria contiene no solo celdas de memoria autoconectadas, sino también tres unidades multiplicativas que representan las puertas de entrada, salida y olvido.

Echemos un vistazo a cómo se ve una sola celda LSTM, luego nos sumergiremos en el meollo de la misma para obtener una mejor comprensión. En el siguiente diagrama, puede ver cómo se ve un bloque LSTM y las operaciones que tienen lugar dentro de él: