# 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.

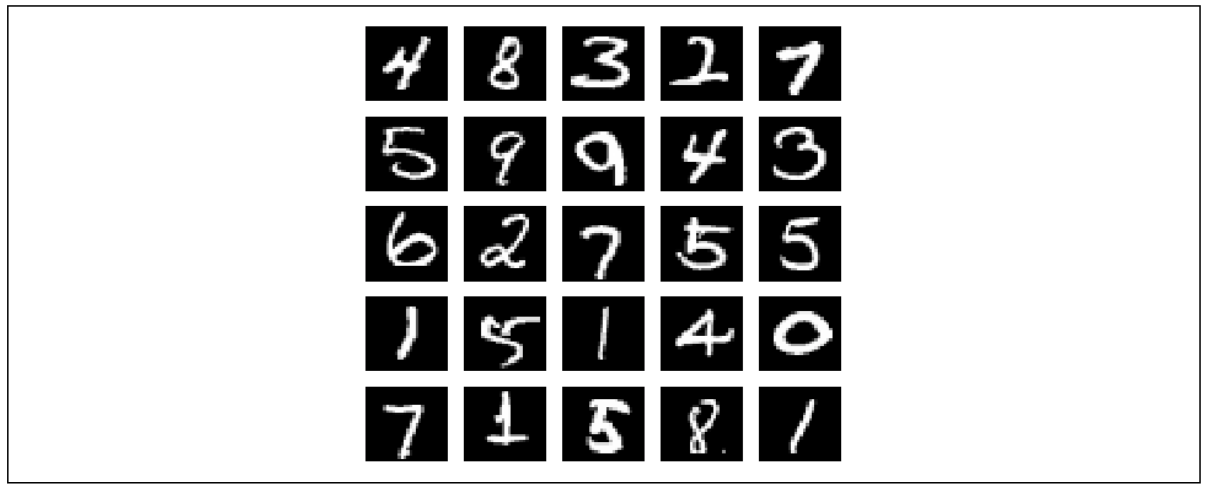
## Multilayer Perceptron (MLP)

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

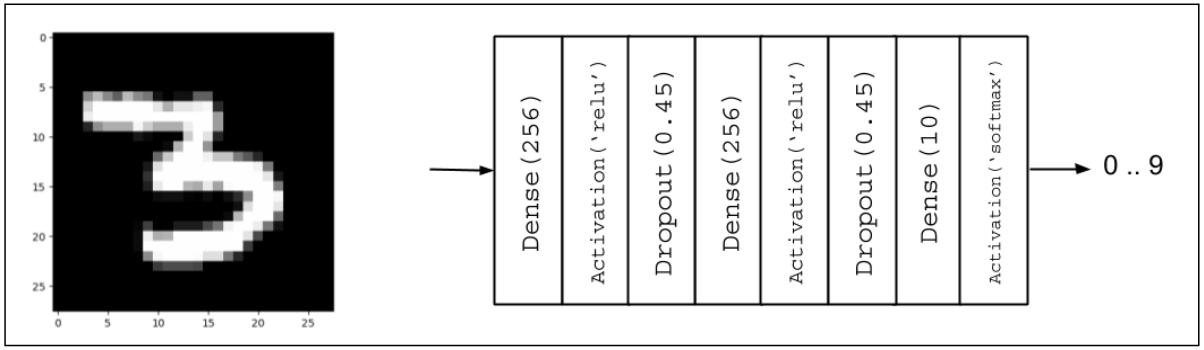
En la figura anterior, se pueden ver imágenes de muestra de los dígitos del MNIST, cada una con un tamaño de 28 x 28 píxeles, en escala de grises.

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:

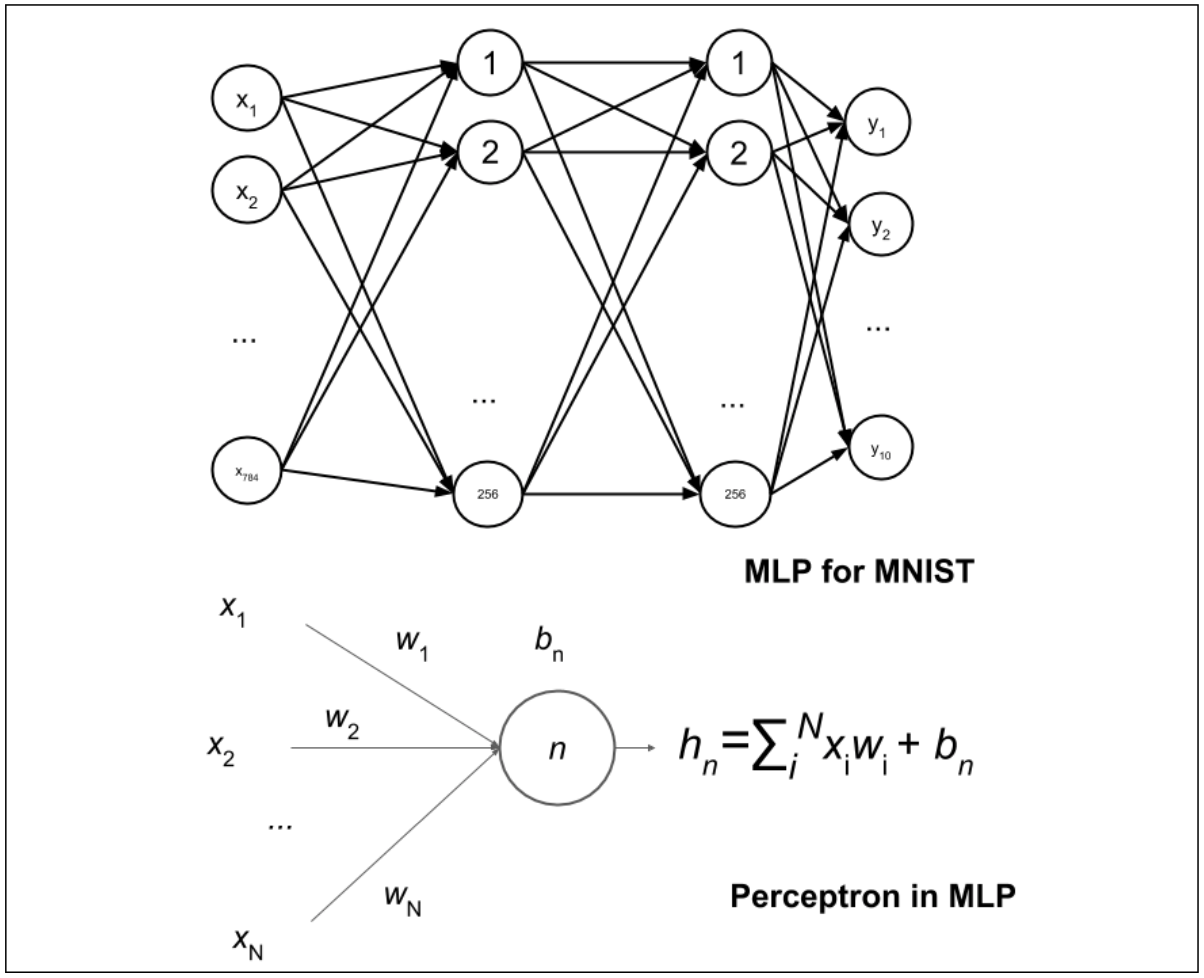
### Imagen que contiene Gráfico Descripción generada automáticamente

## El modelo de clasificación de dígitos MNIST

Para la clasificación de dígitos MNIST 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))

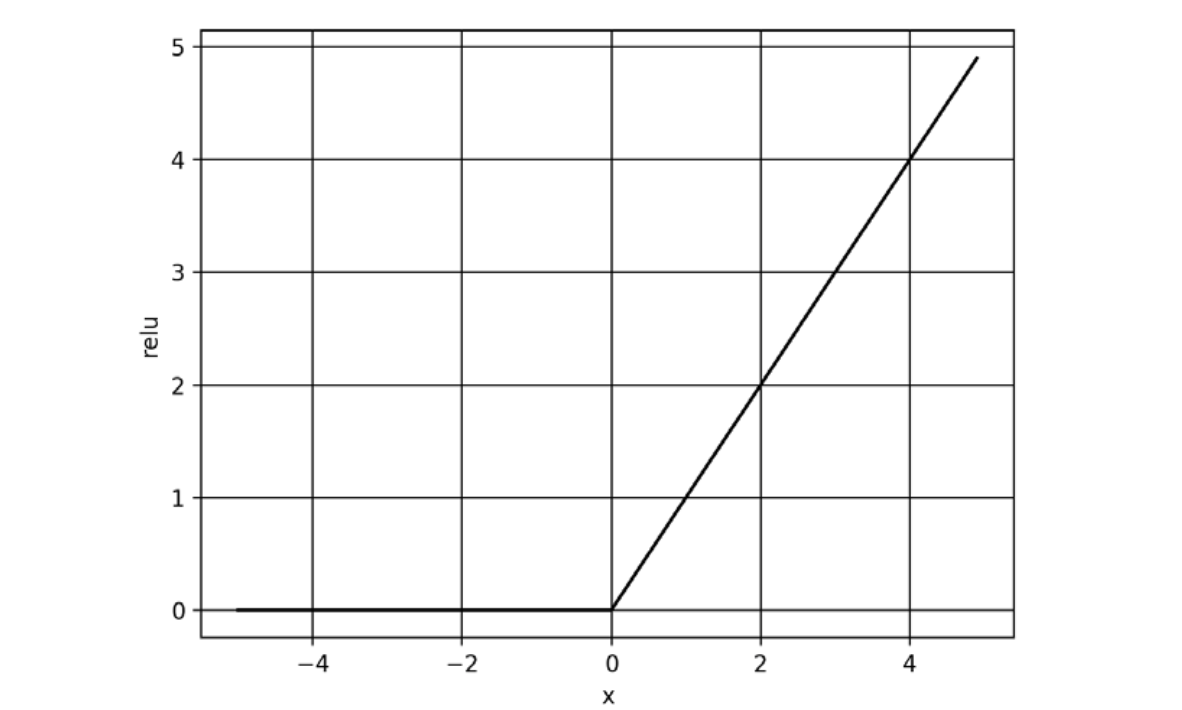
model.add(Dense(num\_labels))

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

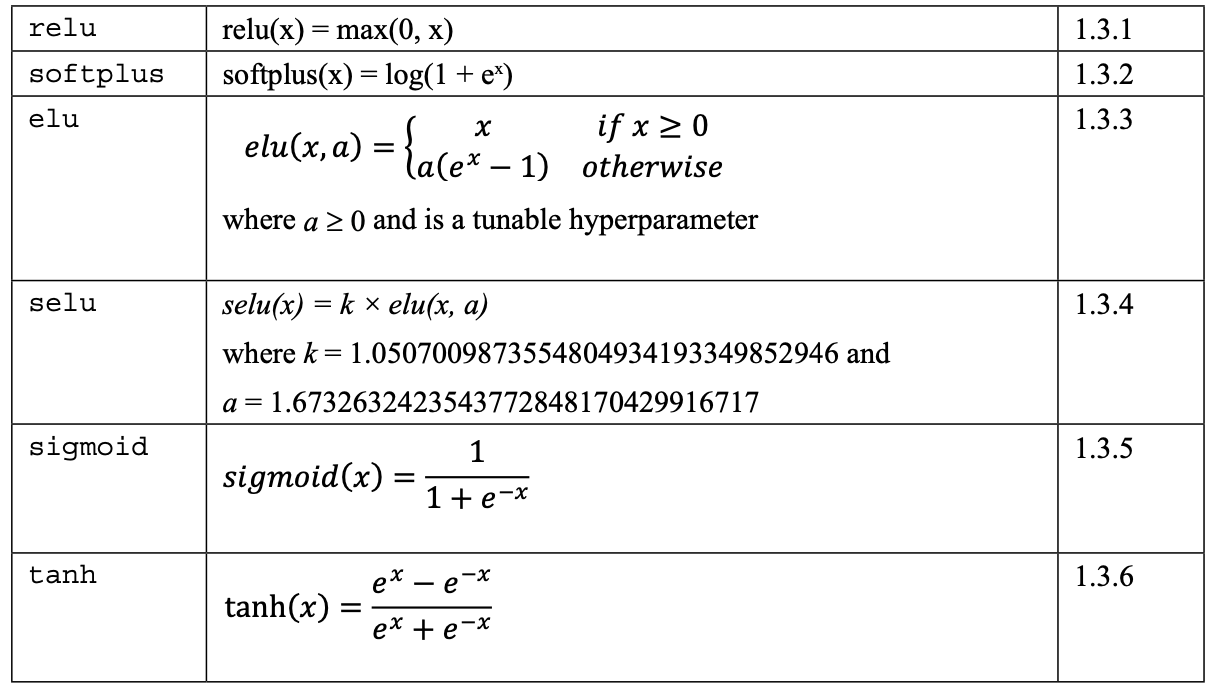
add(Activation('softmax'))

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]** |  |  |

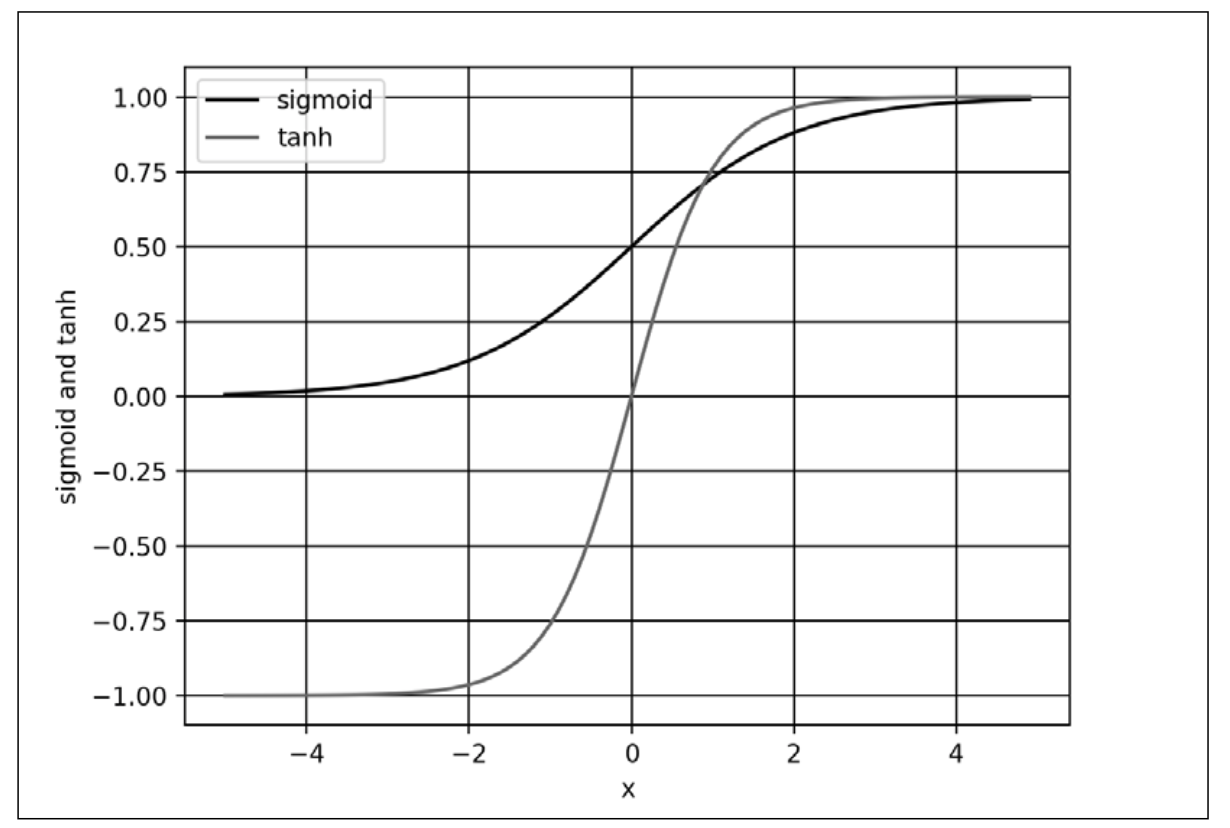
El tensor de salida de la predicción sugiere que la imagen de entrada será 7 dado que su índice tiene la probabilidad más alta (**9.99995947e-01**). El método numpy.argmax() se puede utilizar para determinar el índice del elemento con el valor más alto.

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).

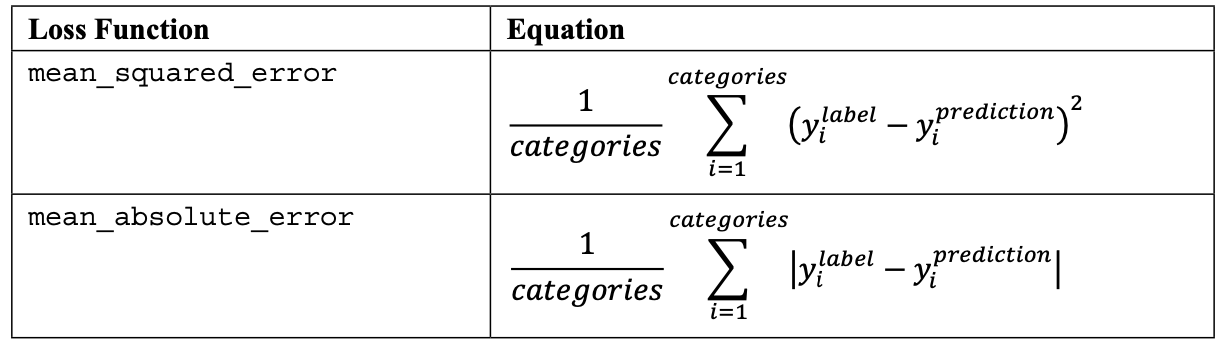
La función tanh mapea su entrada en el rango de -1.0 a 1.0. Esto es importante si la salida puede oscilar tanto en valores positivos como negativos. La función tanh se utiliza más popularmente en la capa interna de redes neuronales recurrentes, pero también se ha utilizado como activación de la capa 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 .

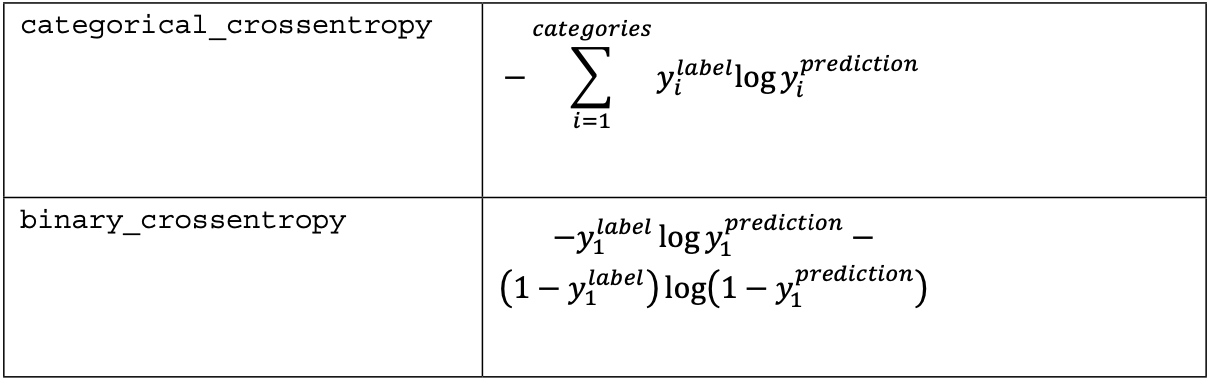
En el siguiente gráfico se muestra las funciones sigmoid y tanh.

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 el aprendizaje profundo, hay muchas otras métricas de rendimiento. Sin embargo, depende de la aplicación de destino del modelo. En la literatura, las métricas de rendimiento del modelo entrenado en el conjunto de datos de prueba se informa para su comparación con otros modelos de aprendizaje profundo.