Práctica 1
Introducción al Deep Learning con Keras

TÉCNICAS ESTADÍSTICAS PARA EL APRENDIZAJE II

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

UNIVERSITAS
Miguel Hernández

Introducción al Deep Learning con Keras

Tensorflow es un framework de *machine learnig* que facilita el acceso a una serie de algoritmos y modelos para tratar problemas modernos de clasificación y regresión como los de reconocimiento y clasificación de imágenes, procesamiento de lenguaje natural…

Por otra parte, **Keras** es un interfaz de *deep learning* para Python que permite acceder a las herramientas de Tensorflow de una forma sencilla y reduciendo el riesgo de cometer errores.

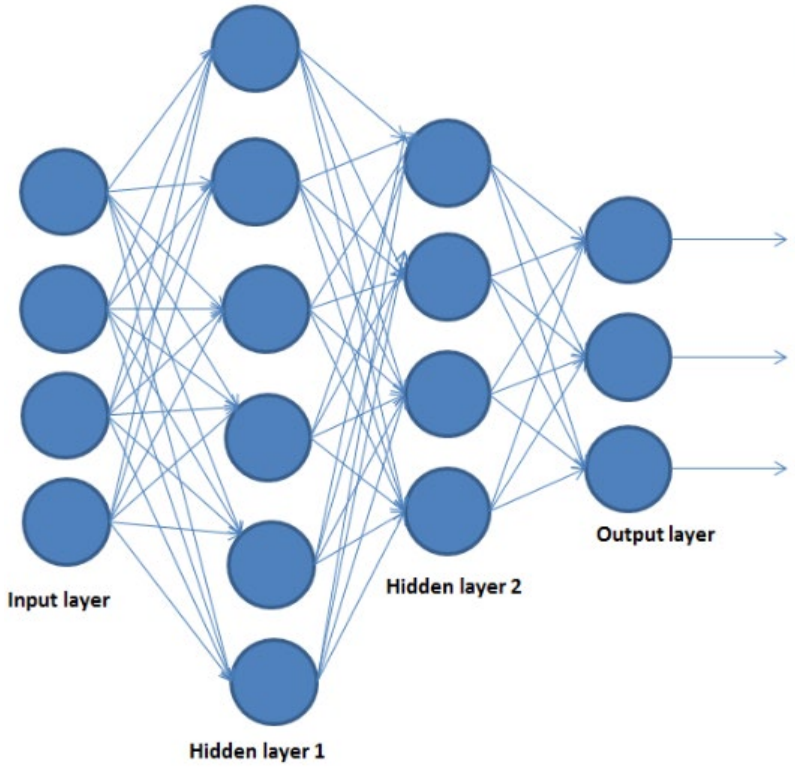
Para utilizar Keras solo necesitamos instalar en nuestro sistema una versión de Python y la librería **tensorflow**.


```
from tensorflow import keras
print(keras.__version__)
```

En esta práctica crearemos nuestra primera red neuronal profunda haciendo uso de Keras.

Como sabemos, una red neuronal profunda consta de varias capas ocultas que pueden **aprender a modelar datos** de formas complejas y mapeos complejos entre entradas y salidas.

En concreto, veremos cómo implementar una red neuronal profunda para un problema de **clasificación de varias clases**.



Usaremos el **conjunto de datos de MNIST**, un conjunto de datos clásico en la comunidad de aprendizaje automático, que ha existido casi tanto tiempo como el campo mismo y se ha estudiado intensamente. Se podría decir que "resolver" MNIST es como el "Hola mundo" del aprendizaje profundo.

La base de datos está formada por **dígitos escritos a mano** (imágenes en escala de grises de 28 píxeles por 28 píxeles) en 10 categorías (0 a 9) y consta de 60000 imágenes de entrenamiento, junto con un conjunto de prueba de 10000 imágenes, que usaremos para evaluar qué tan bien ha aprendido a reconocer dígitos nuestra red neuronal.

Construiremos un **modelo que reconocerá los dígitos escritos a mano** de este conjunto de datos.

El conjunto de datos MNIST viene precargado en Keras, en forma de train/test, cada uno de los cuales incluye un conjunto de **imágenes (*x*)** y **etiquetas asociadas (*y*)**.

Seguiremos el script **Practica1.py**:

x_train e *y_train* forman el conjunto de entrenamiento.

x_test e *y_test* forman el conjunto de test.

En primer lugar, alimentaremos la red neuronal con los datos de entrenamiento *x_train* e *y_train*.

Después, la red aprenderá a asociar imágenes y etiquetas.

Finalmente, le pediremos a la red que produzca predicciones para *x_test* y verificaremos si estas predicciones coinciden con las etiquetas de *y_test*.

Preprocesamiento de los datos

Las imágenes están codificadas como matrices 3D y las etiquetas son una matriz 1D de dígitos, que van de 0 a 9.

Las imágenes de entrenamiento se almacenan en una matriz de forma (images, width, height), en este caso (60000, 28, 28), de tipo entero con valores en el intervalo [0, 255].

Las transformaremos en una matriz doble de forma (60000, 28*28), es decir, **las imágenes de 28x28 se aplanan en vectores de 28x28 =784 de longitud**, donde cada entrada en el vector representa el valor de gris para un solo píxel en la imagen.

Luego, convertimos los **valores de la escala de grises** de números enteros que oscilan entre 0 y 255 en valores que oscilan **entre 0 y 1**.

También necesitamos **codificar categóricamente las etiquetas** que son un vector entero con valores de 0 a 9. Las convertiremos en un formato de matriz binaria (**codificación *one-hot***), con la función *to_categorical()* de Keras.

Salida de la red en problemas de clasificación categórica

Ejemplo: supongamos que nuestras etiquetas a clasificar son: "perro", "gato", "oso" y "pez".

Al modelar **problemas de clasificación multiclase** utilizando redes neuronales, es conveniente transformar las etiquetas al **formato *one-hot*,** donde **cada etiqueta se representa mediante un vector binario**, donde todas las posiciones valen 0 excepto la de la clase a representar, que tendrá valor 1.

Por ejemplo, para las clases del ejemplo con animales tendríamos:

"perro" --> [1, 0, 0, 0]
"gato" --> [0, 1, 0, 0]
"oso" --> [0, 0, 1, 0]
"pez" --> [0, 0, 0, 1]

Salida de la red en problemas de clasificación categórica

Cuando tenemos un problema de **clasificación binaria**, nos basta con entrenar una red neuronal con una sola neurona sigmoidal en la capa de salida. La salida de dicha neurona la podemos interpretar como la probabilidad de que un ejemplo dado pertenezca a la clase positiva.

Cuando tenemos **K clases diferentes (K > 2)** se podría añadir una neurona de salida con función logística para cada clase, aunque en estos casos se recomienda utilizar la **función de activación *softmax*** para la salida de la red:

$$y_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

donde y_j corresponde al nivel de activación de la j-ésima neurona de salida y z_j es su entrada neta.

Salida de la red en problemas de clasificación categórica

La función *softmax* asigna **probabilidades en el rango [0,1] a cada clase** y las probabilidades suman 1, es decir, fuerza que las salidas de la red representen una distribución de probabilidad.

Esta función permite que la **salida *i*-ésima** de la red pueda interpretarse como una **estimación de la probabilidad de que la clase *i* sea la clase correcta** para el ejemplo correspondiente a la entrada de la red. El valor de salida con el valor más grande se tomará como la clase predicha por el modelo.

En problemas de **clasificación binaria**, podemos utilizar tanto una única unidad sigmoidal de tipo logístico como una capa *softmax* con dos neuronas de salida. Ambas alternativas funcionarán igual de bien, si bien lo más habitual es quedarnos con una sola neurona, más simple.

Salida de la red en problemas de clasificación categórica

La **función de coste** más adecuada para resolver problemas de clasificación utilizando redes neuronales artificiales es la basada en la **entropía cruzada [*cross-entropy*]**. Para un lote de n ejemplos:

$$E_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K} t_{ij}\log y_{ij}$$

donde t_{ij} es la salida deseada para la j-ésima neurona de la capa *softmax* dado el i-ésimo ejemplo del conjunto de datos e y_{ij} es la salida observada en la j-ésima neurona para el i-ésimo ejemplo.

Es una función de error con una pendiente muy pronunciada que facilita un cambio razonable en los pesos de la neurona durante su entrenamiento, con lo que se observan mejoras notables en la velocidad del aprendizaje de una red neuronal.

Definición del modelo

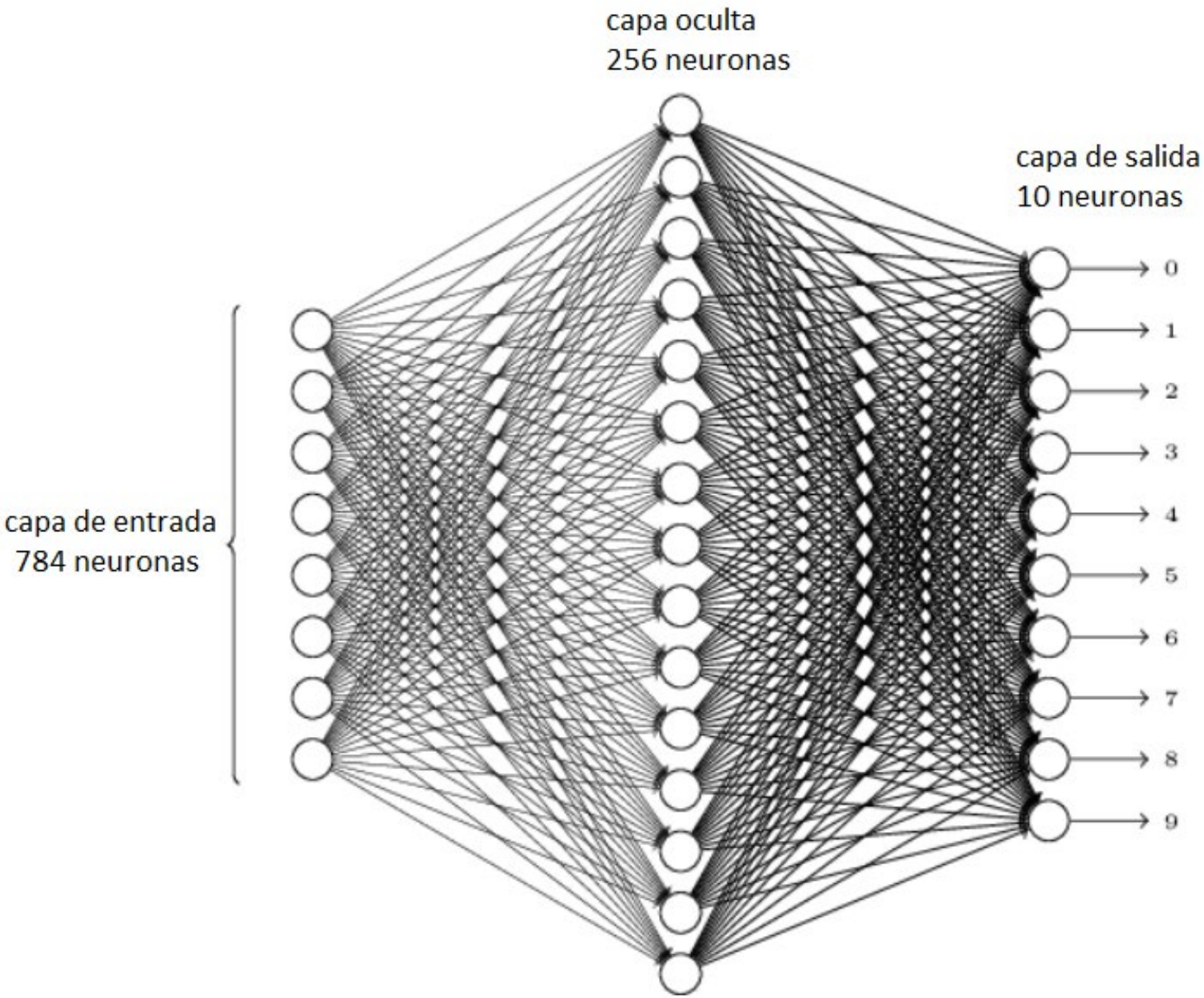
Los modelos en Keras se definen como una secuencia de capas.

Vamos a crear una red neuronal de **dos capas neuronales densamente conectadas** (también llamadas totalmente conectadas).

En primer lugar, podemos pasar a la red la **forma de los datos de entrada**. En nuestro caso, son vectores numéricos de longitud 28x28=784 (cada vector representa una imagen en escala de grises).

La **primera capa oculta** tiene 256 neuronas y **función de activación** ***relu.***

La **capa final** genera un vector numérico de longitud 10 (probabilidades para cada dígito de 0 a 9, sumando 1) usando una **función de activación** ***softmax***. Cada puntuación será la probabilidad de que la imagen del dígito actual pertenezca a una de nuestras clases de 10 dígitos.


capa oculta
256 neuronas
capa de salida
10 neuronas
capa de entrada
784 neuronas
0
1
2
3
4
5
6
7
8
9

Compilación

Lo que queremos es **encontrar un conjunto de pesos y sesgos para** que la salida de la red se aproxime a *y_train* para todas las entradas de entrenamiento *x_train* (**minimizar la función de pérdida**).

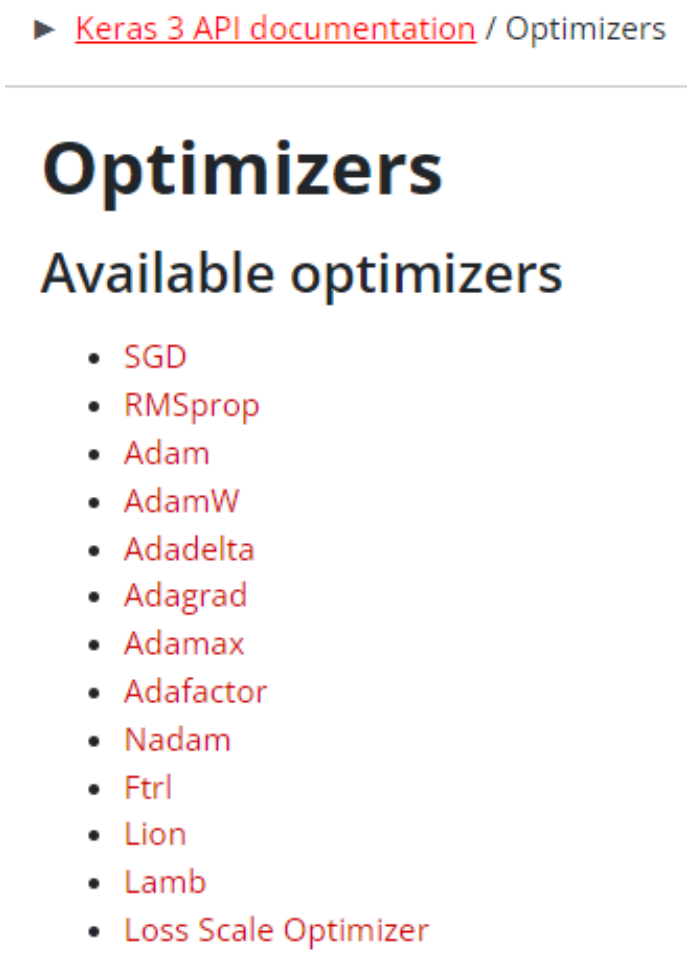
Para ello, debemos decidir tres cosas, como parte del paso de **compilación**, que realizaremos con la función *compile():*
loss, *optimizer* y *metrics*.

Función de pérdida (*loss*): cómo la red podrá medir qué tan bien está haciendo el trabajo con los datos de entrenamiento y, por lo tanto, cómo podrá dirigirse en la dirección correcta.

Para el argumento *loss* hay muchas opciones. Dos de esas opciones son *binary_crossentropy* usada para clasificación binaria y *categorical_crossentropy* usada para clasificación en más de dos grupos.

Compilación

Un **optimizador (*optimizer*)**: mecanismo a través del cual la red se actualizará en función de los datos que ve y su función de pérdida.



Estos optimizadores son variantes del algoritmo de descenso de gradiente. En la actualidad uno de los algoritmos preferidos es *adam* ya que suele proporcionar buenos resultados de forma general.

En este ejemplo vamos a usar el optimizador "*rmsprop*", similar al optimizador de descenso de gradiente, excepto que puede variar la tasa de aprendizaje para que el algoritmo converja más rápido.

Compilación

Métricas para evaluar el train/test (*metrics*): en este ejemplo nos centraremos en la precisión (*accuracy*), es decir, la fracción de las imágenes que se clasificaron correctamente.

Importante: las métricas solo brindan información, mientras que la función de pérdida (*loss*) juega un papel fundamental en el funcionamiento del algoritmo.

Entrenamiento y evaluación del modelo

Durante el **entrenamiento** se van suministrando muestras a la red, se calcula el error cometido entre la predicción de la red y las etiquetas reales, y se ajustan los pesos para reducir este error utilizando el **algoritmo** ***backpropagation***.

Si este proceso lo realizamos **muestra a muestra**, una época de entrenamiento se refiere a cuando hemos suministrado todas las muestras de entrenamiento. Por lo tanto, realizar más épocas se refiere a volver a entrenar suministrando las mismas muestras una y otra vez.

En vez de realizar este proceso muestra a muestra, lo podemos realizar **por lotes o batchs, es decir, suministrando un grupo de muestras** a la red y ajustando el error de forma conjunta para ese grupo (método que utilizan la mayoría de optimizadores).

Entrenamiento y evaluación del modelo

Ajustaremos los datos de entrenamiento al modelo configurado del siguiente modo:

- El **número de épocas** en 10 (número de veces que se itera sobre el conjunto de datos de entrenamiento completo durante el proceso de entrenamiento).
- El **tamaño del lote** en 128 (número de muestras que se propagarán a través de la red en cada iteración del entrenamiento).
- El **porcentaje de validación** en 20 (fracción de los datos que se utilizarán como datos de validación).

Entrenamiento y evaluación del modelo

Para entrenar o ajustar nuestro modelo a los datos de entrenamiento utilizaremos la función *fit()*.

Esta función tiene el **parámetro** ***verbose***, que indica el nivel de detalle de la información que se muestra en pantalla durante el entrenamiento. Si es 0 no se muestra nada, 1 se muestra una barra de progreso y 2 se muestra una línea por época. El valor por defecto es "auto", que equivale a 1 en la mayoría de los casos.

Las cantidades que se muestran durante el entrenamiento son:

- La pérdida de la red.
- La precisión de la red.

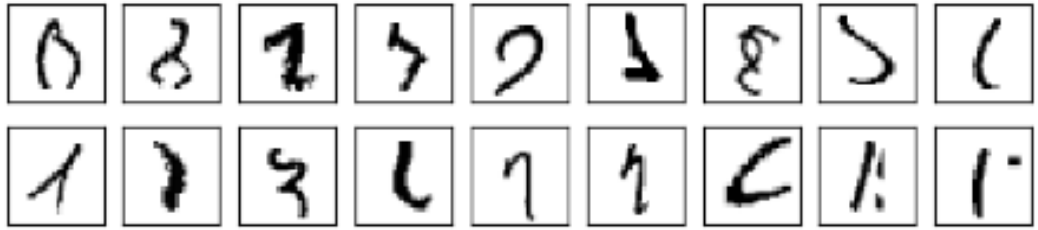
Entrenamiento y evaluación del modelo

A grandes rasgos, en cada iteración se toman los datos de entrenamiento de *x_train*, se pasan a través de la red neuronal (con los valores que en ese momento tengan sus parámetros), se compara el resultado obtenido con el esperado (indicado en *y_train*) y se calcula la función de pérdida (*loss*) para guiar el proceso de ajuste de los parámetros del modelo.

Intuitivamente consiste en aplicar el optimizador especificado en método *compile()* para calcular un nuevo valor de cada uno de los parámetros (pesos y sesgos) del modelo en cada iteración, de tal forma que se reduzca el valor de la función de pérdida en siguientes iteraciones.

Además, después del entrenamiento debemos verificar que el modelo también funciona bien en el **conjunto de prueba**.

Algunas de las imágenes MNIST son difíciles de reconocer incluso para los humanos:



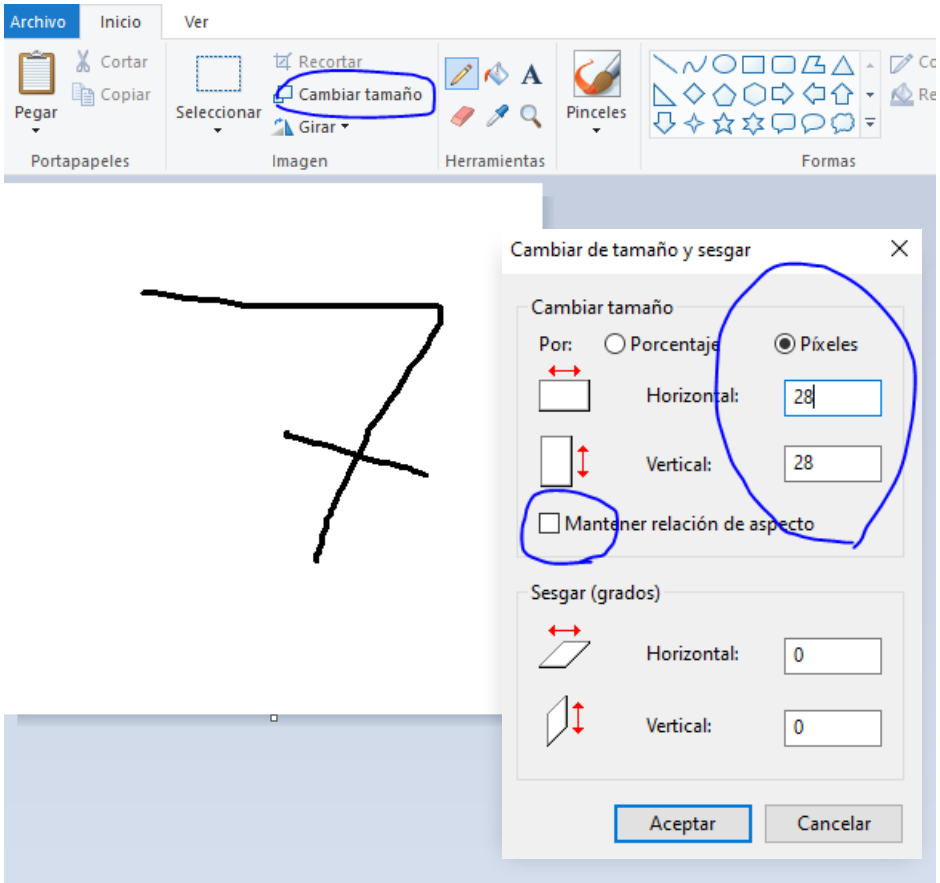
¿Cuántas veces consigue clasificar correctamente tu red neuronal las muestras del conjunto de prueba?

¿Podemos entender los principios por los cuales nuestra red clasifica los dígitos escritos a mano?

En los comienzos de la investigación de la Inteligencia Artificial, se esperaba que dicha investigación también ayudara a comprender los principios detrás de la inteligencia y, tal vez, el funcionamiento del cerebro humano. ¡Quizás terminemos sin entender ni el cerebro ni cómo funciona la Inteligencia Artificial!

Ejercicio 1.

Probemos con otros números escritos a mano.



Archivo → Propiedades → Blanco y Negro

Ejercicio 2.

Hemos observado que con un desarrollo muy simple de la red neuronal hemos obtenido un buen resultado.

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

Por ejemplo, puedes probar a cambiar el número de neuronas de la red, añadir otra capa oculta, cambiar el optimizador, probar con un número de épocas diferente…

Ejercicio 3.

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

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

Construye una red neuronal que reconozca las prendas de ropa de este conjunto de datos.

Ejercicio 3.


```
# Para importar los datos
from keras.datasets import fashion_mnist
# Nombre de las clases (numeradas de 0 a 9 en el conjunto de datos)
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

¿Cuántas veces consigue clasificar correctamente tu red neuronal las muestras del conjunto de prueba?