UNIVERSIDAD DE GRANADA

Práctica 1: MNIST

Carlos Morales Aguilera

December 1, 2020

MNIST

December 1, 2020

1 Práctica 1: MNIST

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales. Deberá evaluar el uso de varios tipos de redes neuronales para resolver un problema de OCR: el reconocimiento de dígitos manuscritos de la base de datos MNIST (http://yann.lecun.com/exdb/mnist/).

2 Introducción a MNIST

La base de datos MNIST (Instituto Nacional de Estándares y Tecnología) es una gran base de datos de dígitos escritos a mano que se usa comúnmente para entrenar varios sistemas de procesamiento de imágenes. La base de datos también se usa ampliamente para capacitación y pruebas en el campo del aprendizaje automático. Fue creado "re-mezclando" las muestras de los conjuntos de datos originales del NIST. Los creadores sintieron que dado que el conjunto de datos de capacitación del NIST se tomó de los empleados de la Oficina del Censo Estadounidense, mientras que el conjunto de datos de prueba se tomó de estudiantes estadounidenses de secundaria, no era adecuado para experimentos de aprendizaje automático. Además, las imágenes en blanco y negro del NIST se normalizaron para encajar en un cuadro delimitador de 28x28 píxeles y se suavizaron, lo que introdujo niveles de escala de grises.

3 Redes Neuronales

Las redes neuronales son sistemas computacionales, bioinspirados en el sistema neuronal humano, tomando como ejemplo las neuronas que constituyen el cerebro, aportando a los ordenadores de una inteligencia artificial. Estas redes se encuentran formadas por una serie de unidades básicas denominadas neuronas que se comunican entre sí formando una red que adquiere conocimiento. Debido a su similitud con el cerebro humano, la principal característica de estas redes neuronales es su capacidad de *aprendizaje*. Este modelo se puede aplicar tanto en problemas de aprendizaje *supervisado*, *no supervisado* como *reforzado*.

El problema **MNIST** es uno de los principales a la hora de empezar a trabajar con redes neuronales en problemas de aprendizaje *supervisado*, donde se debe entrenar una red neuronal de forma que esta sea capaz de clasificar un número manuscrito en su clase correctamente, haciendo uso para ello de una red que se ajuste correctamente a las características del problema.

4 Herramientas

Para la realización de esta práctica se han escogido las siguientes herramientas:

4.0.1 Tensorflow

Tensorflow es una biblioteca de código abierto para el aprendizaje automático multitarea. Ha sido desarrollado por Google para realizar las funciones de *backend* y entrenar redes neuronales para detectar, analizar y descifrar patrones y correlaciones. Emplea para ello unas técnicas de aprendizaje y razonamiento humanos. Hoy en día, se utiliza en la investigación y los productos de Google. *Tensorflow* ofrece varios niveles de abstracción para crear y entrenar modelos y brinda flexibilidad y control con características como la API funcional de *Keras*.

Instalación: Tras instalar *Python3* con *Pip3*, ejecutar: `pip3 install tensorflow`.

4.0.2 Keras

Keras, por otro lado, es una biblioteca de redes neuronales de alto nivel que se ejecuta en la parte superior de *TensorFlow*, *CNTK* o *Theano*. El uso de Keras en el aprendizaje profundo permite la creación de prototipos fácil y rápida, así como la ejecución sin problemas en la CPU y la GPU. Este marco está escrito en código *Python*, que es fácil de depurar y permite una fácil extensibilidad.

Instalación: Tras instalar *Python3* con *Pip3*, ejecutar: `pip3 install Keras`.

4.0.3 Numpy

Numpy es el paquete de software básico para la computación científica en *Python*. Es una biblioteca de *Python* que proporciona objetos de matriz multidimensionales, varios objetos derivados (como matrices y matrices enmascaradas) y varias rutinas para operaciones rápidas en matrices, incluidas matemáticas, lógica, clasificación, selección, etc.

Instalación: Tras instalar *Python3* con *Pip3*, ejecutar: `pip3 install numpy`.

4.0.4 ANN visualizer

ANN visualizer es una biblioteca de Python de visualización destinada a trabajar con *Keras*. Utiliza por debajo una implementación que hace uso de la biblioteca Graphviz de *Python* para crear un gráfico presentable de la red neuronal que está construyendo.

Instalación:

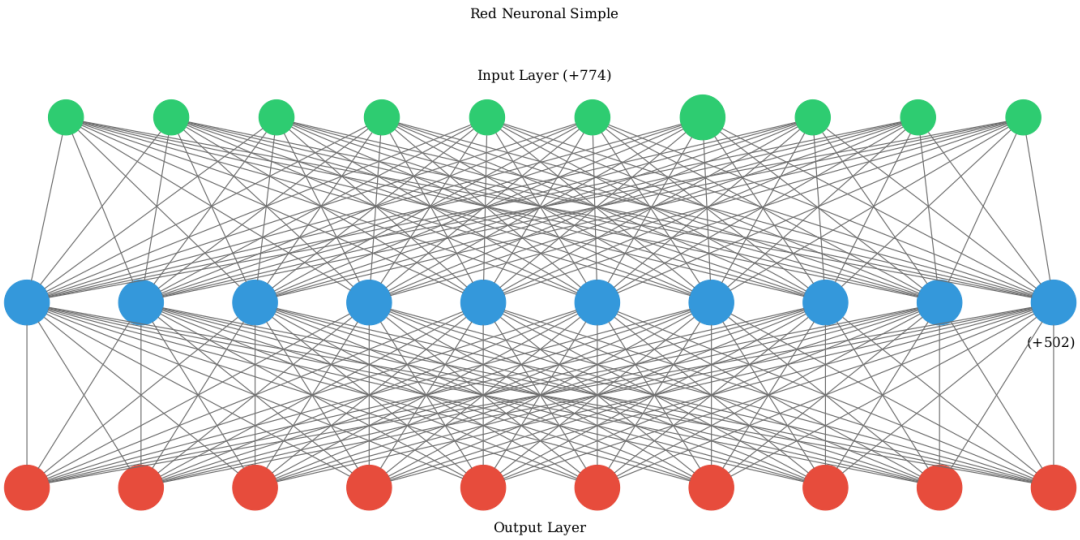
1. Instalar *Graphviz*: `sudo apt-get install graphviz && pip3 install graphviz`.
2. Instalar *ANN visualizer*: `pip3 install ann_visualizer`.

5 Realización de la práctica

5.1 Modelo de Red Neuronal Simple

5.1.1 Modelo de Red

El primer modelo que se plantea es un modelo sencillo donde existe una capa de entrada con un total de 512 neuronas intercomunicadas, y otra capa de salida que clasifica en los distintos 10 números que se contemplan. Para ello se ha realizado el siguiente modelo de capas:



5.1.2 Lectura de datos

Para la realización del modelo, lo primero sería leer los datos, para ello *keras* aporta un conjunto de datasets, entre los cuales se incluye *MNIST*, por lo que basta con cargar el conjunto de datos.

Nota: Al tratarse de un modelo *no determinístico* se establece al inicio una semilla aleatoria para reproducir siempre el mismo resultado.

[1]:
```
from keras.datasets import mnist
import numpy as np
import tensorflow as tf

# Set random seed to assert deterministic solution
np.random.seed(1234)
tf.random.set_seed(1234)

# Load the train and test dataset with their classes
(train, train_class), (test, test_class) = mnist.load_data()
```

5.1.3 Preprocesamiento de los datos

A continuación, se realiza un preprocesamiento consistente en los siguientes pasos:

- La información viene almacenada en arrays de 60000 muestras de entrenamiento de 28x28, y 10000 muestras de prueba, por lo que hay que transformar los arrays de forma que adquieran el formato (60000, 28*28) en lugar de (60000, 28, 28), y (10000, 28*28) en lugar de (10000, 28, 28).
- La información viene con valores comprendidos en el intervalo [0,255], que refleja la intensidad de la celda, por lo que se deben normalizar los valores para trabajar en el rango [0,1].
- Las etiquetas se deben tratar como valores categóricos, por lo que se realiza una transformación mediante la función to_categorical perteneciente a *Keras*.

[2]:
```
from keras.utils import to_categorical

# Preprocessing data
# Reshape the data
train = train.reshape((60000, 28*28))
test = test.reshape((10000, 28 * 28))

# Normalize data
train = train.astype('float32') / 255
test = test.astype('float32') / 225

# Encode the labels categorically
train_class = to_categorical(train_class,10)
test_class  = to_categorical(test_class,10)
```

5.1.4 Diseño del modelo

El modelo propuesto es el de una **Red Neuronal Simple**, con una capa de entrada y otra de salida, por lo que se necesitará crear un modelo *secuencial* con dos capas,una de entrada y otra de salida.

Para la definición del modelo secuencial se utiliza la clase Sequential perteneciente a los modelos que ofrece *Keras*. A continuación se añadirán las diferentes capas:

- Una capa de entrada, que recibe la información con las dimensiones que hemos procesado previamente, es decir, (28*28,). Para ello se utiliza una capa Dense. Esta capa sigue la estructura *Dense(units, activation)* realizando la operación *output=activation(dot(input,kernel) + bias)*, donde la función de activación indica si se activa la neurona y units representa el número de unidades de salida de la capa. La función de activación empleada es RELU conocida como *Rectified Linear Unit*, consistente en que si la salida que realiza la operación es positiva, se activa y emite el valor positivo, si es negativa devuelve un 0 (no se activa).
- Una capa de salida, que recibe la información de la capa previa y realiza una operación de softmax que convierte un vector real en un vector de probabilidades categóricas. Para ello se utiliza una capa *Dense* que utiliza como función de activación la función *softmax* y como número de unidades de salida 10 unidades (que representan los 10 números diferentes).

[3]:
```
from keras import models
from keras import layers

# Define the model
neural_network = models.Sequential()

# Add different layers
neural_network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,␣
↪)))
neural_network.add(layers.Dense(10, activation='softmax'))
```

5.1.5 Entrenamiento del modelo

Para poder entrenar el modelo, primero es necesario configurar las métricas que este va a emplear para evaluar tanto el *accuracy* como el *loss*, e indicar el método de aprendizaje, también llamado *optimizador*.

Para el cálculo de la pérdida (*loss*) se ha utilizado la entropía cruzada mediante la función categorical_crossentropy.

Por otro lado, se han utilizado distintos algoritmos de aprendizajes, pero finalmente se ha decidido mantener el algoritmo de Root Mean Square Propagation, conocido como **RMSPROP**. Este método, en lugar de mantener un acumulado de los gradientes, utiliza el concepto de ventana para únicamente considerar en el aprendizaje los gradientes más recientes. Para ello se ha utilizado la función rmsprop de *Keras*.

Todo este procedimiento se realiza utilizando la función compile de los modelos de *Keras*.

[4]:
```
# Compile the model
neural_network.compile(optimizer='rmsprop', loss='categorical_crossentropy',␣
↪metrics = ['accuracy'])
```

A continuación, ya se procede a entrenar o ajustar el modelo. Para ello se ha utilizado la función fit de los modelos de *Keras*. Para ello además hay que decidir dos valores importantes en el ajuste de la red: **épocas** y **tamaño del bloque**.

- **Épocas**: Representa el número de iteraciones que debe realizar el algoritmo de entrenamiento sobre el conjunto de datos. Con este valor se puede separar el entrenamiento en varias fases y evaluarlas independientemente. Un valor pequeño no permitirá a la red lo suficiente, mientras que un valor excesivo provocará un sobreaprendizaje.
- **Tamaño del bloque**: Denominado *batch size*, representa la división que se realiza de los datos para ser procesados en paralelo, un conjunto grande ajustará mejor la red, pero puede llevar un sobreesfuerzo computacional, por lo que se recomienda distribuir correctamente los datos en conjuntos adecuados.

Finalmente se ha decidido utilizar un valor de **15** epócas y un tamaño de bloque de **128** (aunque se han evaluado otros, como se puede observar en la comparativa final del modelo).

[5]:
```
# Train the model
neural_network.fit(train, train_class, epochs=15, batch_size=128)
```

```
Epoch 1/15
469/469 [==============================] - 2s 3ms/step - loss: 0.2538 -
accuracy: 0.9257
Epoch 2/15
469/469 [==============================] - 2s 3ms/step - loss: 0.1028 -
accuracy: 0.9694
Epoch 3/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0677 -
accuracy: 0.9800
Epoch 4/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0499 -
```

```
accuracy: 0.9850
Epoch 5/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0377 -
accuracy: 0.9884
Epoch 6/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0288 -
accuracy: 0.9909
Epoch 7/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0221 -
accuracy: 0.9936
Epoch 8/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0168 -
accuracy: 0.9951
Epoch 9/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0126 -
accuracy: 0.9964
Epoch 10/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0104 -
accuracy: 0.9968
Epoch 11/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0080 -
accuracy: 0.9978
Epoch 12/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0065 -
accuracy: 0.9982
Epoch 13/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0050 -
accuracy: 0.9987
Epoch 14/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0037 -
accuracy: 0.9990
Epoch 15/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0029 -
accuracy: 0.9992
```

[5]: `<tensorflow.python.keras.callbacks.History at 0x7eff1445bd00>`

5.1.6 Evaluación del modelo

Tras construir el modelo, se procede a evaluar el mismo, tanto para el conjunto de datos de entrenamiento, como el conjunto de datos de prueba. Para ello de nuevo se ha utilizado la función evaluate, la cual nos permite predecir con la red creada el conjunto de datos que le indiquemos, esta además devuelve los valores *loss* y *accuracy* de la forma que se indicó en el proceso de compilación o configuración, realizando para el *accuracy* una validación cruzada.

[6]:
```
# Evaluate the model
train_loss, train_accuracy = neural_network.evaluate(train, train_class,␣
 ↪verbose=0) # Train data
```

```
test_loss, test_accuracy = neural_network.evaluate(test, test_class, verbose=0)␣
↪# Test data
```

Una vez se ha evaluado el modelo, se visualizan los resultados obtenidos:

[7]:
```
# Print train results
print(f"Train Loss: {train_loss}")
print(f"Train Accuracy: {train_accuracy} %")

# Print test results
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy} %")
```

```
Train Loss: 0.0015359376557171345
Train Accuracy: 0.999750018119812 %
Test Loss: 0.09498418867588043
Test Accuracy: 0.9825000166893005 %
```

Por último, se ha decidido hacer una tabla comparativa viendo diferentes configuraciones del modelo y ver los resultados obtenidos:

Épocas	Tamaño bloque	Algoritmo aprendizaje	Accuracy Entrenamiento	Accuracy Prueba
2	128	rmsprop	0.9796000123023987 %	0.9717000126838684 %
15	128	rmsprop	**0.999750018119812 %**	**0.9825000166893005 %**
2	128	sgd	0.8850499987602234 %	0.8925999999046326 %
15	128	sgd	0.9356833100318909 %	0.9354000091552734 %
2	128	adam	0.9785166382789612 %	0.9714999794960022 %
15	128	adam	0.9996833205223083 %	0.9824000000953674 %

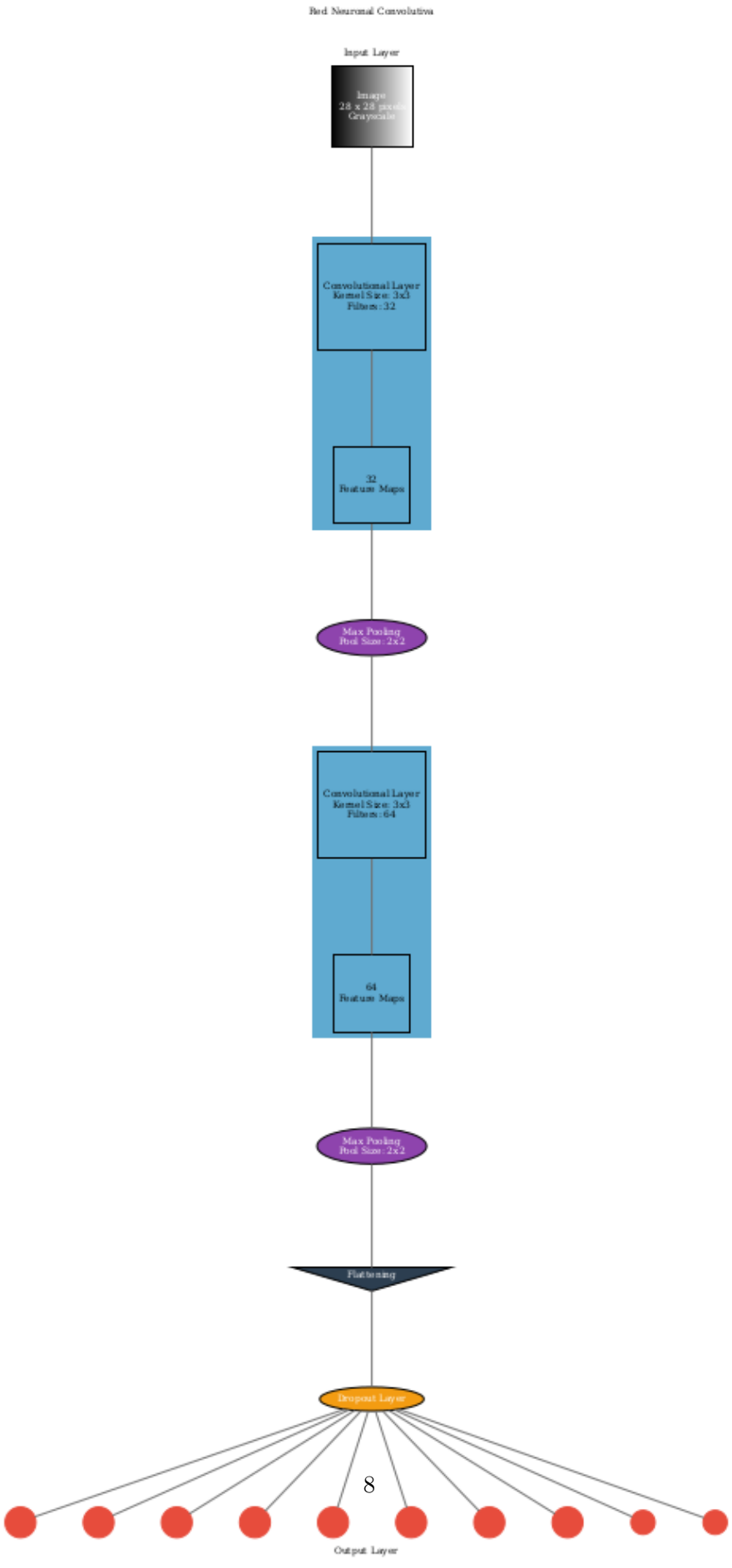
Tras observar los resultados obtenidos, nos plantearemos en el siguiente modelo comprobar si es más interesante plantearnos el algoritmo **ADAM** para el aprendizaje.

Por otro lado, obtenemos un buen resultado, pero el objetivo real es mejorar la precisión en el conjunto de prueba frente a los resultados obtenidos.

5.2 Modelo de Red Neuronal Convolutiva

5.2.1 Modelo de Red

Tras los resultados obtenidos con el primer modelo, se va a evaluar un segundo modelo, más complejo en su diseño, para comprobar si se pueden mejorar los resultados obtenidos previamente. Para ello se plantea el siguiente modelo de **Red Neuronal Convolutiva**:


Red Neuronal Convolutiva
Input Layer
Image
28 x 28 pixels
Grayscale
Convolutional Layer
Kernel Size: 3x3
Filters: 32
32
Feature Maps
Max Pooling
Pool Size: 2x2
Convolutional Layer
Kernel Size: 3x3
Filters: 64
64
Feature Maps
Max Pooling
Pool Size: 2x2
Flattening
Dropout Layer
Output Layer

5.2.2 Lectura de datos

Para leer los datos del modelo, se realiza la misma lectura empleada en el primer modelo haciendo uso de los datasets de *keras*.

Nota: Al tratarse de un modelo *no determinístico* se establece al inicio una semilla aleatoria para reproducir siempre el mismo resultado.

[8]:
```
from tensorflow.keras.datasets import mnist
import numpy as np
import tensorflow as tf

# Set random seed to assert deterministic solution
np.random.seed(1234)
tf.random.set_seed(1234)
# Load the train and test dataset with their classes
(train, train_class), (test, test_class) = mnist.load_data()
```

5.2.3 Preprocesamiento de los datos

A continuación, se realiza un preprocesamiento consistente en los siguientes pasos:

- Al igual que en el primer modelo, se realiza una normalización de los datos ya que la información viene con valores comprendidos en el intervalo [0,255], que refleja la intensidad de la celda, por lo que se deben normalizar los valores para trabajar en el rango [0,1].
- Una de las mejoras que se presentan es el empleo de la biblioteca numpy para trabajar con arrays, haciendo uso de la función expand_dims, en este caso para reducir las dimensiones de (60000, 28, 28) a un array de 60000 elementos con la dimensión (28, 28, 1) y de (10000, 28, 28) a un array de 10000 elementos con la dimensión (28, 28, 1).
- Las etiquetas se deben tratar como valores categóricos, por lo que se realiza una transformación mediante la función to_categorical perteneciente a *Keras*.

[9]:
```
import numpy as np
from tensorflow.keras.utils import to_categorical

# Preprocessing data
# Normalize data
train = train.astype("float32") / 255
test = test.astype("float32") / 255

# Reshape the data
train = np.expand_dims(train, -1)
test = np.expand_dims(test, -1)

# Encode the labels categorically
train_class = to_categorical(train_class,10)
test_class  = to_categorical(test_class,10)
```

5.2.4 Diseño del modelo

El modelo propuesto es el de una **Red Neuronal Convolutiva**, que posee una seria de capas de convolución y otras capas de *pooling*, por lo que se necesitará crear un modelo *secuencial*.

Para la definición del modelo secuencial se utiliza la clase Sequential perteneciente a los modelos que ofrece *Keras*. A continuación se añadirán las diferentes capas:

- Una capa de entrada, que recibe la información con las dimensiones que hemos procesado previamente, es decir, (28, 28, 1). Para ello se utiliza la función Input de *Keras*, al cual se le indican las dimensiones del conjunto de datos de entrada.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con una función de activación RELU conocida como *Rectified Linear Unit*, consistente en que si la salida que realiza la operación es positiva, se activa y emite el valor positivo, si es negativa devuelve un 0 (no se activa). Para ello se utiliza la capa Conv2D de las *layers* de *Keras*.
- Una capa de *pooling* de 2 dimensiones, que reduce el tamaño de la muestra tomando el valor máximo de una ventana definida como valor de representación de la ventana, en nuestro caso de un tamaño de 2x2. Para ello se utiliza la capa MaxPooling2D de las *layers* de *Keras*.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con una función de activación **RELU**.
- Una capa de *pooling* de 2 dimensiones, que utiliza una ventana de 2x2.
- Una capa de *flatten* o *aplanamiento*, que reduce las dimensiones, ya que poseemos unas dimensiones de (None, 5, 5, 64) y pasamos a unas de (None, 1600) al aplanar. Para ello se utiliza la capa Flatten de las *layers* de *Keras*.
- Una capa de *dropout* o de regularización, donde se le indica un porcentaje de exclusión de neuronas para evitar el *overfitting*. Para ello se utiliza la capa Dropout de las *layers* de *Keras*.
- Una capa de salida, que recibe la información de la capa previa y realiza una operación de softmax que convierte un vector real en un vector de probabilidades categóricas. Para ello se utiliza una capa *Dense* que utiliza como función de activación la función *softmax* y como número de unidades de salida 10 unidades (que representan los 10 números diferentes).

En resumen, se obtiene una capa de entrada, y se realizan dos veces seguidas el procedimiento de convolución en 2D y *pooling* de 2D, con ventanas de 3x3 y 3x2 respectivamente. A continuación se realiza una capa de *flatten* que "aplana" los datos y se realiza un *dropout* del 50% de las neuronas. Por último se realiza una operación de *softmax* para clasificar con probabilidades categóricas los resultados en las 10 clases existentes, conformando un total de 8 capas.

[10]:
```
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras import layers

# Define the model
neural_network = models.Sequential()

# Add different layers
neural_network.add(keras.Input(shape=(28, 28, 1)))
neural_network.add(layers.Conv2D(32, kernel_size=(3, 3), activation="relu"))
neural_network.add(layers.MaxPooling2D(pool_size=(2, 2)))
neural_network.add(layers.Conv2D(64, kernel_size=(3, 3), activation="relu"))
```

```
neural_network.add(layers.MaxPooling2D(pool_size=(2, 2)))
neural_network.add(layers.Flatten())
neural_network.add(layers.Dropout(0.5))
neural_network.add(layers.Dense(10, activation="softmax"))
```

5.2.5 Entrenamiento del modelo

Para poder entrenar el modelo, primero es necesario configurar las métricas que este va a emplear para evaluar tanto el *accuracy* como el *loss*, e indicar el método de aprendizaje, también llamado *optimizador*.

Para el cálculo de la pérdida (*loss*) se ha utilizado la entropía cruzada mediante la función categorical_crossentropy.

Por otro lado, se han utilizado distintos algoritmos de aprendizajes, pero finalmente se ha decidido mantener el algoritmo de Adam. Este método, supone la combinación del algoritmo de rmsprop junto a la aplicación del gradiente descendente estocástico con Momentum. Para ello se ha utilizado la función adam de *Keras*.

Todo este procedimiento se realiza utilizando la función compile de los modelos de *Keras*.

[11]:
```
# Compile the model
neural_network.compile(optimizer="adam", loss="categorical_crossentropy",␣
↪metrics=["accuracy"])
```

A continuación, ya se procede a entrenar o ajustar el modelo. Para ello se ha utilizado la función fit de los modelos de *Keras*. Para ello además hay que decidir dos valores importantes en el ajuste de la red: **épocas** y **tamaño del bloque**.

- **Épocas**: Representa el número de iteraciones que debe realizar el algoritmo de entrenamiento sobre el conjunto de datos. Con este valor se puede separar el entrenamiento en varias fases y evaluarlas independientemente. Un valor pequeño no permitirá a la red lo suficiente, mientras que un valor excesivo provocará un sobreaprendizaje.
- **Tamaño del bloque**: Denominado *batch size*, representa la división que se realiza de los datos para ser procesados en paralelo, un conjunto grande ajustará mejor la red, pero puede llevar un sobreesfuerzo computacional, por lo que se recomienda distribuir correctamente los datos en conjuntos adecuados.

Finalmente se ha decidido utilizar un valor de **15** epócas y un tamaño de bloque de **128** (aunque se han evaluado otros, como se puede observar en la comparativa final del modelo).

[12]:
```
# Train the model
neural_network.fit(train, train_class, epochs=15, batch_size=128)
```

```
Epoch 1/15
469/469 [==============================] - 19s 41ms/step - loss: 0.3415 -
accuracy: 0.8968
Epoch 2/15
469/469 [==============================] - 19s 40ms/step - loss: 0.1045 -
accuracy: 0.9679
```

```
Epoch 3/15
469/469 [==============================] - 18s 38ms/step - loss: 0.0779 -
accuracy: 0.9764
Epoch 4/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0671 -
accuracy: 0.9794
Epoch 5/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0583 -
accuracy: 0.9819
Epoch 6/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0532 -
accuracy: 0.9840
Epoch 7/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0486 -
accuracy: 0.9849
Epoch 8/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0447 -
accuracy: 0.9861
Epoch 9/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0424 -
accuracy: 0.9865
Epoch 10/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0393 -
accuracy: 0.9877
Epoch 11/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0390 -
accuracy: 0.9881
Epoch 12/15
469/469 [==============================] - 18s 38ms/step - loss: 0.0366 -
accuracy: 0.9885
Epoch 13/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0336 -
accuracy: 0.9898
Epoch 14/15
469/469 [==============================] - 18s 37ms/step - loss: 0.0326 -
accuracy: 0.9895
Epoch 15/15
469/469 [==============================] - 18s 38ms/step - loss: 0.0304 -
accuracy: 0.9905
```

[12]: `<tensorflow.python.keras.callbacks.History at 0x7eff172ed070>`

5.2.6 Evaluación del modelo

Tras construir el modelo, se procede a evaluar el mismo, tanto para el conjunto de datos de entrenamiento, como el conjunto de datos de prueba. Para ello de nuevo se ha utilizado la función evaluate, la cual nos permite predecir con la red creada el conjunto de datos que le indiquemos, esta además devuelve los valores *loss* y *accuracy* de la forma que se indicó en el proceso de compilación

o configuración, realizando para el *accuracy* una validación cruzada.

[13]:
```
# Evaluate the model
train_loss, train_accuracy = neural_network.evaluate(train, train_class,␣
 ↪verbose=0) # Train data
test_loss, test_accuracy = neural_network.evaluate(test, test_class, verbose=0)␣
 ↪# Test data
```

Una vez se ha evaluado el modelo, se visualizan los resultados obtenidos:

[14]:
```
# Print train results
print(f"Train Loss: {train_loss}")
print(f"Train Accuracy: {train_accuracy} %")

# Print test results
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy} %")
```

```
Train Loss: 0.01396924164146185
Train Accuracy: 0.9957500100135803 %
Test Loss: 0.023504970595240593
Test Accuracy: 0.9918000102043152 %
```

Por último, se ha decidido hacer una tabla comparativa viendo diferentes configuraciones del modelo y ver los resultados obtenidos:

Épocas	Tamaño bloque	Algoritmo aprendizaje	Accuracy Entrenamiento	Accuracy Prueba
2	128	rmsprop	0.9811166524887085 %	0.9832000136375427 %
15	128	rmsprop	0.9943333268165588 %	0.9908999800682068 %
2	128	sgd	0.9122999906539917 %	0.9230999946594238 %
15	128	sgd	0.9747833609580994 %	0.9750000238418579 %
2	128	adam	0.9817000031471252 %	0.9818999767303467 %
15	128	adam	**0.9957500100135803 %**	**0.9918000102043152 %**

Se puede observar que, aunque se empeora frente a los resultados en el conjunto de entrenamiento del modelo de Red Neuronal Simple, claramente se mejora respecto a los resultados en el conjunto de prueba, por lo que se deduce que el modelo se ajusta mejor al problema y menos al conjunto de entrenamiento.

6 Bibliografía y Referencias

En la realización de la práctica se han utilizado diferentes bibliografías para comprender los conceptos relativos a ella, muchos de los conceptos han sido enlazados directamente en los términos cuando se empleaban.

6.1 Bibliografía

- The MNIST Database of Handwritten Digit Images for Machine Learning Research
- Convolutional Neural Network Committees for Handwritten Character Classification
- Hands-on deep learning for images with TensorFlow : build intelligent computer vision applications using TensorFlow and Keras
- Optimizers Explained - Adam, Momentum and Stochastic Gradient Descent

6.2 Tutoriales

- Confusion Matrix with Keras
- Solve the MNIST Image Classification Problem
- MNIST with Keras for Beginners
- Image Processing for MNIST using Keras
- Handwritten Digit Recognition Using PyTorch — Intro To Neural Networks

7 Conclusiones

7.1 Realización de la práctica

Durante la realización de la práctica se han podido observar, asimilar y comprender los conceptos explicados en clase, para ello primero he leído algunos documentos y libros relacionados con el tema, intentando que estuvieran enfocados en el problema en particular para poder comprender perfectamente las partes del mismo, y ya posteriormente extrapolarlo. Al valorar las opciones de implementación sobre el problema en cuestión, pude observar que existían múltiples formar de resolverlo e implementarlo. Hasta el momento solo había trabajado ligeramente con redes neuronales con *R*, por lo que busqué inicialmente en dicho lenguaje, y posteriormente decidí ver cual era la herramienta más utilizada para ello, observando que era (a simple vista) *Keras*. Posteriormente me decanté por *Python3* ya que observé que muchas de las documentaciones, tutoriales e incluso artículos hacían referencia a este lenguaje, por lo que deducí que era actualmente el más utilizado para dicho problema, y además supondría un reto aprender a manejar redes neuronales fuera de *R*.

Durante la realización de la práctica mi primer enfoque fue realizar un modelo sencillo, consistente en una **Red Neuronal Simple**, donde tras ver diferentes artículos, tutoriales e implementaciones pude observar que de forma sencilla se obtenían buenos resultados. Tras finalizar dicha implementación me plantee diferentes opciones y decidí, por cuestión de tratar de obtener mejores resultados, realizar un modelo más complejo como una **Red Neuronal Convolutiva**. Este modelo resultó más difícil de realizar, y realicé más pruebas hasta que finalmente conseguí ajustarlo de forma que obtuviera mejores resultados que el modelo inicial.

En paralelo a la realización de los modelos surgieron duda sobre los tipos de neurona, las funciones de activación, el algoritmo de aprendizaje que pude ir comprendiendo leyendo documentación y contrastando con los apuntes vistos en clase. Recomiendo leer el artículo Optimizers Explained -

Adam, Momentum and Stochastic Gradient Descent, donde pude comprender realmente la diferencia entre los principales algoritmos de aprendizaje empleados en la actualidad.

Finalmente, tras realizar una serie de pruebas, he podido evaluar las diferencias entre los parámetros de los modelos, los algoritmos de entrenamiento, como evitar el *overfitting*, y como realizar un modelo simple pero competitivo para la práctica.

7.2 Conclusiones extraídas de la realización de la práctica

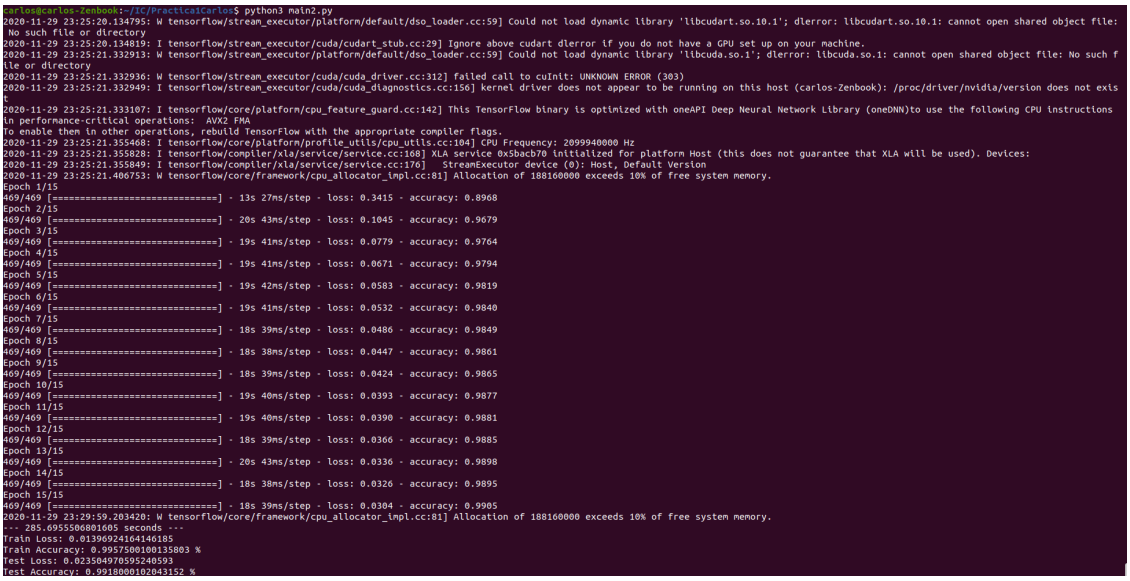
- Tal y como se enseña en esta asignatura y en otras directamente relacionadas como *Tratamiento Inteligente de Datos*, para poder resolver un problema de clasificación (o reconocimiento de dígitos en este caso, que es un problema de clasificación), es necesaria una planificación previa. Para ello son esenciales las tareas de estudiar y entender el conjunto de datos, realizar un preprocesamiento adecuado y comprender el enfoque a realizar para el diseño del modelo.
- La elección de cada uno de los componentes del modelo no es una tarea trivial, no consiste en simplemente escoger resultados previos que consideremos buenos, sino comprender cada uno de los elementos, que función realizan, por qué son relevantes o no, comparar y escoger la opción que mejor se adapte.
- La comprensión de ciertos elementos como el algoritmo de aprendizaje puede ser una tarea compleja, en mi caso llegar a comprender como funciona el algoritmo de aprendizaje *Adam* ha sido realmente complejo, teniendo que informarme buscando referencias y explicaciones varias. La comprensión de qué se hace y por qué es una de las tareas que más tiempo conlleva, ya que la implementación luego es realmente trivial conociendo las herramientas.
- Durante la realización de la práctica, he podido observar que tal y como explico en los apartados previos, me ha llevado más tiempo comprender que realizar o implementar.
- Existen problemas dentro de la realización de un problema de este estilo, como puede ser la capacidad de cómputo, la potencia de nuestro sistema o los requisitos que requieren los modelos de los problemas. En mi caso poseo un equipo prácticamente nuevo y no me ha supuesto problema, pero he tratado de realizar una comparativa de tiempo ejecutando la misma práctica en un ordenador con menor capacidad computacional y en una máquina *AWS*, donde he podido observar una clara diferencia de tiempo de ejecución del modelo.
- Una de las tareas que han conllevado más tiempo ha sido observar el comportamiento de la red diseñada, y tratar de evitar el *overfitting*, ya que un número excesivo de épocas podía empeorar los resultados obtenidos, o simplemente realizar un cómputo y gasto de tiempo innecesario.
- Se realizo un pequeño modelo con *PyTorch*, pero se descartó porque la complejidad del modelo era elevada y los resultados no eran considerables frente a los ya obtenidos.
- Los modelos de *Machine Learning* son modelos *no determinísticos* tal y como se explica en este enlace, ya que me presenté al problema de encontrarme con diferentes resultados en diferentes ejecuciones.

8 Anexo: Capacidad Computacional

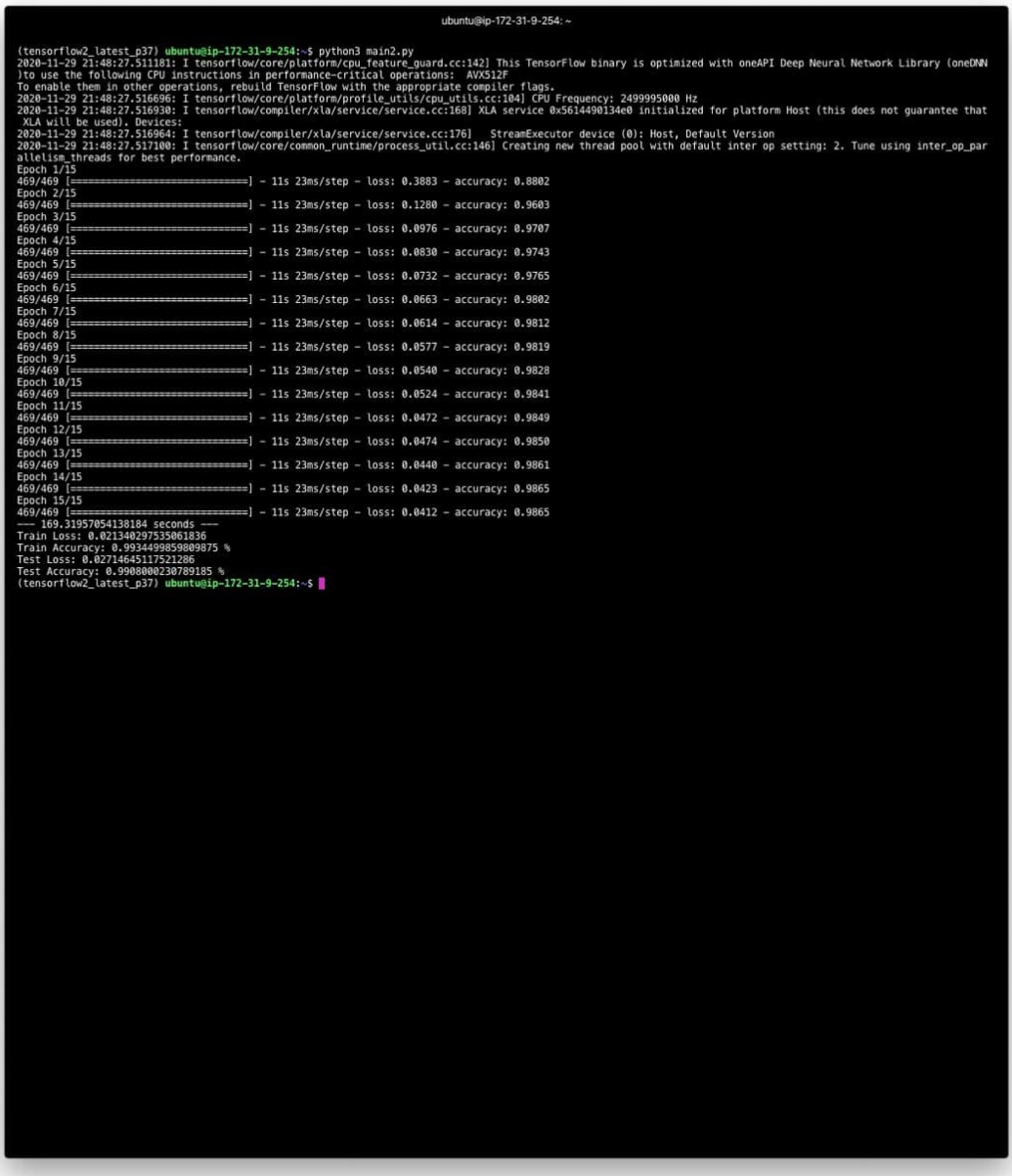
En este anexo se pretende demostrar la importancia de la capacidad computacional a la hora de entrenar un determinado modelo en distintas máquinas, ya que además de tratarse de un modelo *no determinístico*, se puede observar que la capacidad afecta al rendimiento, tiempo de ejecución y resultados obtenidos.

Para ello se han realizado diferentes ejecuciones en distintas máquinas, pero se realizará la comparativa comparando una ejecución en mi máquina propia y en una máquina virtual *AWS* básica:

Ejecución máquina propia:



Ejecución máquina *AWS* básica:



9 Anexo: Implementaciones

En este anexo se pretenden visualizar las diferentes implementaciones que se han realizado de los modelos.

9.1 Primer modelo: Red Neuronal Simple

Fichero: **redNeuronalSimple.py**

```
from keras.datasets import mnist
from keras import models
from keras import layers
```

```
from keras.utils import to_categorical
import time
import numpy as np
import tensorflow as tf

# Set random seed to assert deterministic solution
np.random.seed(1234)
tf.random.set_seed(1234)

# Load the train and test dataset with their classes
(train, train_class), (test, test_class) = mnist.load_data()

# Preprocessing data
# Reshape the data
train = train.reshape((60000, 28*28))
test = test.reshape((10000, 28 * 28))

# Normalize data
train = train.astype('float32') / 255
test = test.astype('float32') / 225

# Encode the labels categorically
train_class = to_categorical(train_class,10)
test_class  = to_categorical(test_class,10)

# Define the model
neural_network = models.Sequential()

# Add different layers
neural_network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28, )))
neural_network.add(layers.Dense(10, activation='softmax'))

# Start to count time
start_time = time.time()

# Compile the model
neural_network.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics = ['accura

# Train the model
neural_network.fit(train, train_class, epochs=15, batch_size=128)

# Evaluate the model
train_loss, train_accuracy = neural_network.evaluate(train, train_class, verbose=0) # Train da
test_loss, test_accuracy = neural_network.evaluate(test, test_class, verbose=0) # Test data

# Print time
print("--- %s seconds ---" % (time.time() - start_time))
```

```
# Print train results
print(f"Train Loss: {train_loss}")
print(f"Train Accuracy: {train_accuracy} %")

# Print test results
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy} %")
```

9.2 Segundo modelo: Red Neuronal Convolutiva

Fichero: **redNeuronalConvolutiva.py**

```
import numpy as np
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras.utils import to_categorical
import time
import numpy as np
import tensorflow as tf

# Set random seed to assert deterministic solution
np.random.seed(1234)
tf.random.set_seed(1234)

# Load the train and test dataset with their classes
(train, train_class), (test, test_class) = mnist.load_data()

# Preprocessing data
# Normalize data
train = train.astype("float32") / 255
test = test.astype("float32") / 255

# Reshape the data
train = np.expand_dims(train, -1)
test = np.expand_dims(test, -1)

# Encode the labels categorically
train_class = to_categorical(train_class,10)
test_class  = to_categorical(test_class,10)

# Define the model
neural_network = models.Sequential()

# Add different layers
neural_network.add(keras.Input(shape=(28, 28, 1)))
neural_network.add(layers.Conv2D(32, kernel_size=(3, 3), activation="relu"))
```

```
neural_network.add(layers.MaxPooling2D(pool_size=(2, 2)))
neural_network.add(layers.Conv2D(64, kernel_size=(3, 3), activation="relu"))
neural_network.add(layers.MaxPooling2D(pool_size=(2, 2)))
neural_network.add(layers.Flatten())
neural_network.add(layers.Dropout(0.5))
neural_network.add(layers.Dense(10, activation="softmax"))

# Start to count time
start_time = time.time()

# Compile the model
neural_network.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])

# Train the model
neural_network.fit(train, train_class, epochs=15, batch_size=128)

# Evaluate the model
train_loss, train_accuracy = neural_network.evaluate(train, train_class, verbose=0) # Train da
test_loss, test_accuracy = neural_network.evaluate(test, test_class, verbose=0) # Test data

# Print time
print("--- %s seconds ---" % (time.time() - start_time))

# Print train results
print(f"Train Loss: {train_loss}")
print(f"Train Accuracy: {train_accuracy} %")

# Print test results
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy} %")
```