Universidad de Granada

Inteligencia Computacional
INGENIERÍA INFORMÁTICA

Práctica de redes neuronales

Reconocimiento óptico de caracteres

MNIST

Autor: Jonathan Martín Valera

Curso 2018-2019

Máster en ingeniería informática

Departamento de Ciencias de la Computación de Inteligencia Artificial

Tabla de contenidos

1. Introducción

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales. Se empleará el uso de varias 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. Hardware y software

Para el entrenamiento y ejecución de la red neuronal se ha utilizado el siguiente hardware:

- **Procesador:** Intel(R) Core(TM) i5-8250U CPU @1.60GHz 1.80GHz
- **Memoria RAM:** 8,00 GB
- **Tarjeta gráfica:** NVIDIA GeForce MX150.

Para desarrollar el software necesario se ha utilizado lo siguiente:

- **Lenguaje programación:** Python 3.6
- **Sistema operativo:** Ubuntu 18.04 x64
- **IDE:** Eclipse Neon
- **Frameworks y librerías:**
    - Tensorflow: https://www.tensorflow.org/?hl=es
    - Keras: https://keras.io/
    - Matplotlib: https://matplotlib.org/
    - MNIST: https://github.com/sorki/python-mnist
    - Numpy: http://www.numpy.org/

3. Descripción de los frameworks

En esta sección voy a describir el motivo del por qué he elegido dichos frameworks y para qué utilizado cada uno.

3.1 Tensorflow

Tensorflow es una librería de código libre para computación numérica usando grafos de flujo de datos.
Según [1], se dice que Tensorflow es muy bueno, pero hay que saber que utilizar. Muchas de las cosas que vas a querer hacer, no hace falta que las programes a mano y reinventes la rueda, puedes usar otras librerías más simples de usar (Keras).

En este caso, yo he utilizado esta librería para poder **cargar un modelo** previamente guardado, utilizando el import load_model de tensorflow.python.keras.models. También se ha utilizado para **normalizar los datos** de las imágenes de entrenamiento y test, utilizando tf.keras.utils.normalize.

Figura 1: Logo de Tensorflow

3.2 Keras

Keras es una librería de muy alto nivel que funciona encima de Theano o Tensorflow [1]. Además, Keras enfatiza el minimalismo, puedes hacer una red neuronal con muy poquitas líneas de código. Consta de una amplia documentación y una gran sencillez, adecuada para una persona principiante en este mundo, es decir, viene perfecta para mí.

En este proyecto, keras se ha utilizado como base del software para **construir, entrenar y evaluar a la red neuronal**.

Figura 2: Logo de Keras

3.3 Matplotlib

Matplotlib es una biblioteca para representar gráficos 2D en una variedad de formatos [2].

En este caso he utilizado matplotlib para poder **representar gráficamente un gráfico que muestra el porcentaje de acierto en función del número de épocas**, observar la evolución que tiene dicha función y determinar cuál puede ser el número máximo de épocas para realizar early stopping y evitar que la red sobreaprenda.

Figura 3: Logo de matplotlib

3.4 MNIST

La base de datos MNIST es una gran base de datos de dígitos escritos a mano que se usa comúnmente para entrenar varios sistemas de procesamiento de imágenes [3].

Para poder cargar los datos de dicha base de datos adjuntados a esta práctica, he utilizado una librería de Python llamada MNIST [4].

Estos datos serán los utilizados por la red neuronal, que se encargará de realizar un modelo entrenando con un conjunto de datos y validando dicho modelo con otro conjunto test.

Figura 4: Ejemplos de números usando en MNIST

3.5 NumPy

Numpy [5] es un paquete fundamental para la computación científica en Python.

En este caso vamos a utilizar este paquete para convertir los datos MNIST, ya que vienen en forma de lista y es necesario pasarlo a un array para posteriormente poder proporcionarle una estructura (shape) determinada para cargar dichos datos en la capa de entrada de nuestra red neuronal.

También se ha utilizado para cargar los resultados de la evaluación de la red neuronal que se han almacenado en un fichero txt para posteriormente representar dichos datos con matplotlib.

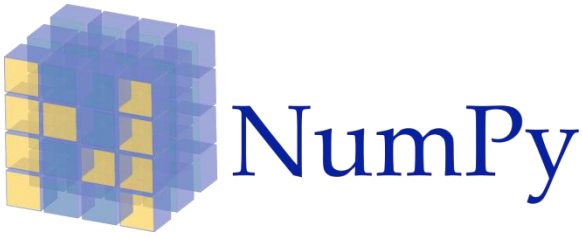

Figura 5: Logo de Numpy

4. Tipos de redes neuronales

En esta sección se explicará brevemente los tipos de redes neuronales más comunes en la actualidad y que se podrían usar para resolver el problema planteado en esta práctica. El principal objetivo de esta sección es conocer para qué sirve cada una, ventajas y desventajas para poder usar la más acertada en este proyecto.

4.1 Red neuronal profunda (DNN)

Esta es una red realmente versátil, ya que con ella se puede procesar texto, imágenes pequeñas… La estructura es la siguiente [:

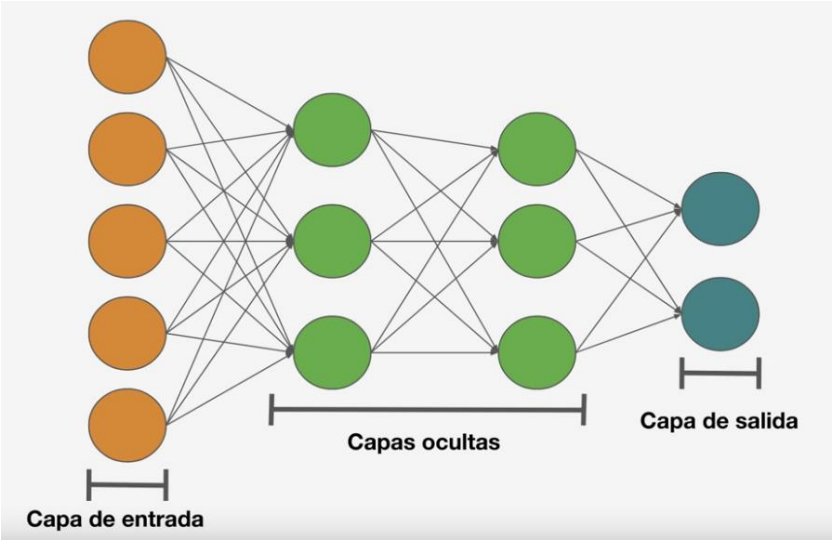

Figura 6: Arquitectura de una DNN

Dentro de las capas ocultas vamos a tener dos o más capas. El funcionamiento es el siguiente:

Hay una capa de entrada que se comunica con cada una de las neuronas de la primera capa oculta, cada capa oculta se comunica con todas las neuronas de la siguiente capa y así sucesivamente hasta llegar a la capa de salida que se puede encargar de realizar una regresión, clasificación…

El principal inconveniente de esta red es que existen tantas conexiones entre las capas, que cuando tenemos un tipo de dato especialmente grande se vuelve muy pesado computacionalmente.

4.2 Red neuronal convolucional (CNN)

Tiene un gran potencial en el uso de procesamiento de imágenes. Esto es debido a que se tiene la misma estructura que en la red neuronal profunda (DNN), pero en las capas intermedias se van a tener unas capas que se van a encargar de realizar convoluciones y maxpooling [6][7][8].

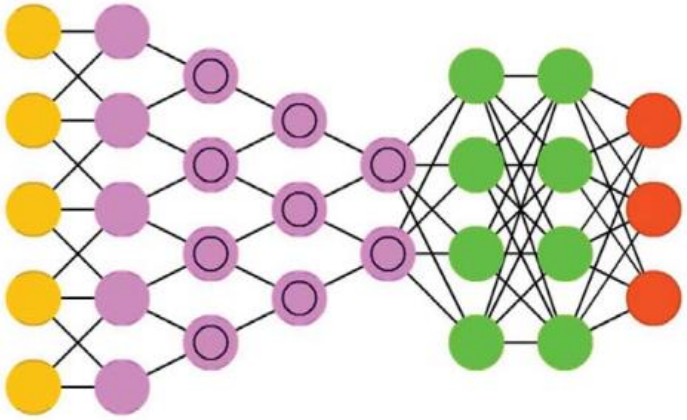

Figura 7: Arquitectura de una CNN

Cada capa va a ir identificando los elementos más importantes de la imagen, y va aumentando el nivel de abstracción de la imagen, desde una capa para buscar líneas o figuras hasta otra que busque elementos elaborados como objetos.

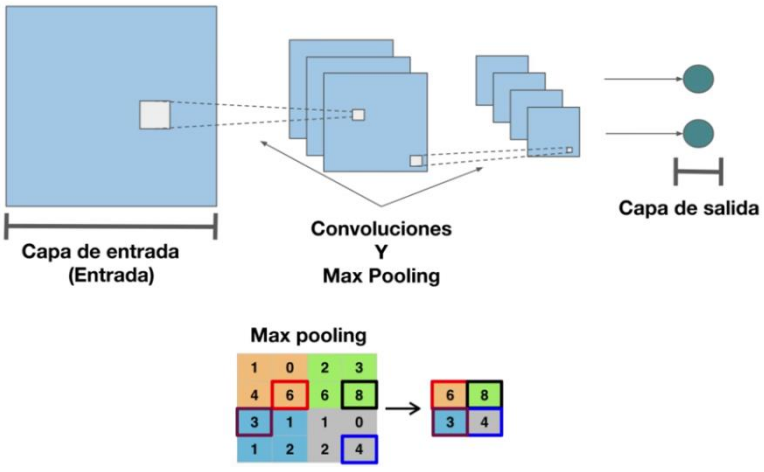

Figura 8: Ejemplo de convolución y maxpooling

4.3 Red neuronal recurrente (RNN)

La red neuronal recurrente [6] se utiliza para tipos de datos secuenciales. Por ejemplo el precio de una acción de una empresa. También se utiliza mucho para el texto, ya que la estructura del texto es secuencial.

Lo que hacen las capas recurrentes es recibir un dato, generar una predicción y la salida de la capa oculta realimenta de nuevo a la capa oculta. Su estructura [7] es la siguiente:

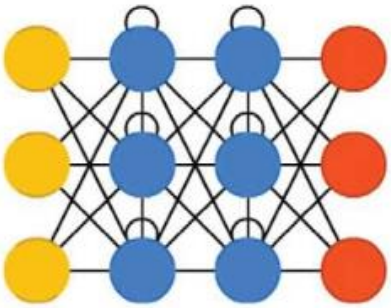

Figura 9: Arquitectura de una RNN

5. Diseño de la aplicación

Tras haber conocido los fundamentos de las principales redes neuronales que se utilizan en la actualidad, se ha realizado un diseño de la aplicación para construir dos tipos de redes neuronales, que son las redes neuronales profundas y redes neuronales convolucionales.

El principal motivo del descarte de la red neuronal recurrente es que en este problema no tenemos datos secuenciales ni texto a poder procesar, sino que tenemos imágenes que son especialidad de las redes convolucionales y profundas.

Para construir esta aplicación he realizado un **modelo basado en orientación a objetos**. He considerado implementarlo de esta forma para poder construir simultáneamente las dos redes neuronales con el fin de observar el comportamiento de ambas.

También se ha realizado un script de automatización, que construye un tipo de red neuronal específica pasando unos parámetros como épocas… y almacena el resultado en un fichero de texto tras evaluar dicho modelo con el de test.

Este script se ha utilizado para poder obtener los datos y observar el comportamiento de la red en función del número de épocas.

Por último, se ha creado una clase para poder tratar y procesar los datos para que la red neuronal los pueda utilizar correctamente.

El diseño propuesto es el siguiente:

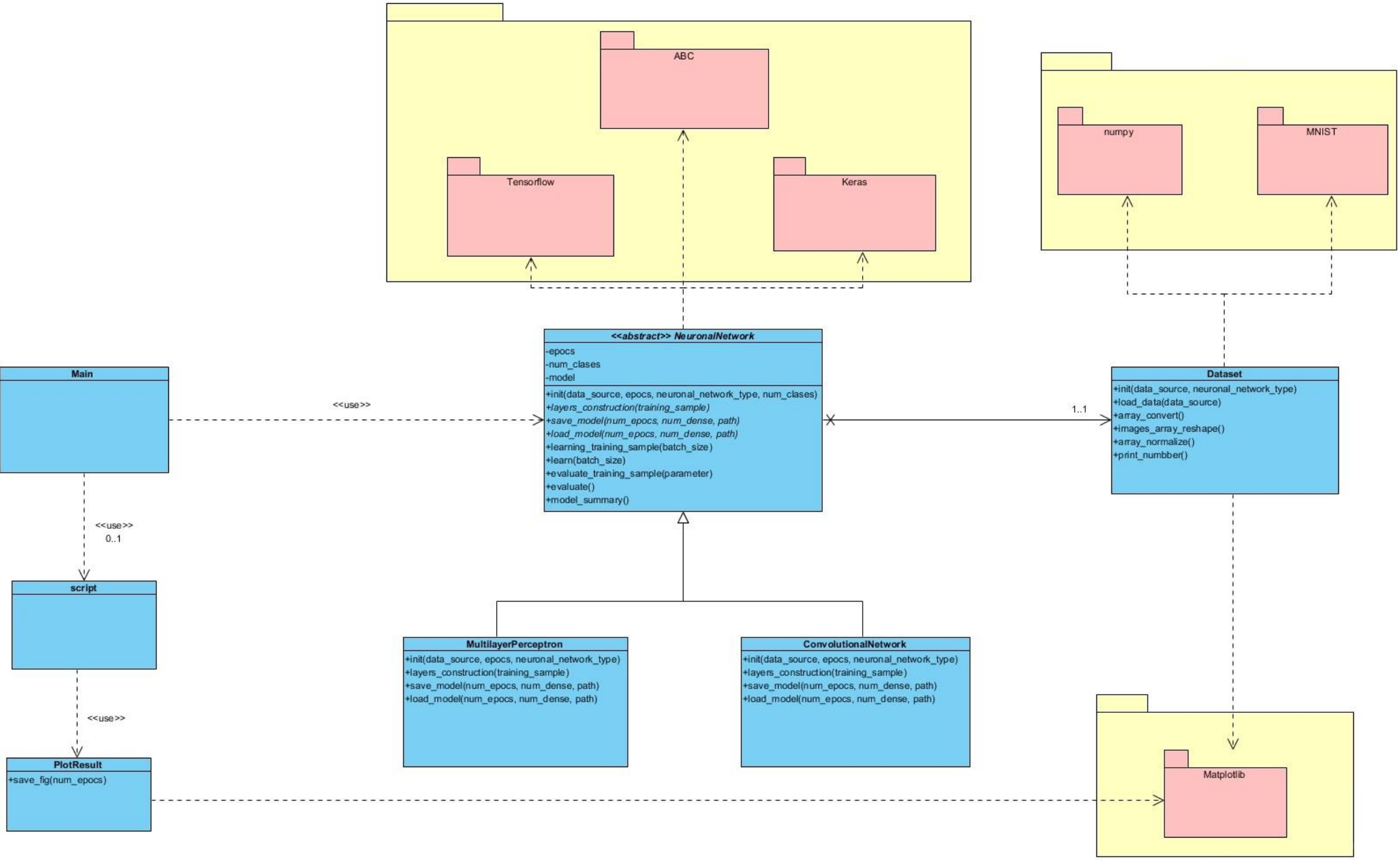

Figura 10: Diseño de la aplicación

6. Descripción del modelo

Para la inicialización de los datos, construcción de las capas, entrenamiento y evaluación se han seguido etapas diferentes que se han modularizado e implementado en métodos diferentes. Dichos métodos realizan lo siguiente:

- **Init**

Este método es el encargado de inicializar las variables de la clase, como son:

  - ***Número de épocas.***
  - ***Número de clases*** (en este caso son 10 números diferentes).
  - ***Datos*****:** Conjunto de datos preprocesados (convertidos a array, reestructurado el shape y normalizados).
  - ***Model*****:** Modelo Sequential() de keras sobre el que añadiremos una serie de capas para construir nuestra red neuronal.

- **Layers construction**

En este método vamos a añadir el conjunto de capas a nuestra red junto con una serie de parámetros y luego compilaremos dicho modelo mediante un optimizador, una función de pérdidas y una métrica de exactitud.

Para construir todas estas capas y compilador se han utilizado las funciones relacionadas de keras.

- **Learn**

En este método vamos a hacer que nuestro modelo previamente construido realice el proceso de aprendizaje de los datos proporcionados en un número de épocas y con un tamaño de lote especificado.

- **Learn training sample**

Este método es el mismo que el anterior solo que para aprender utiliza un subconjunto de datos de entrenamiento. En este caso, he empleado 45.000 de las 60.000 imágenes para entrenar y las 15.000 restantes para realizar el test del modelo.

- **Evaluate**

  Este método Permite verificar el modelo que hemos entrenado con el conjunto de prueba. Nos devuelve los datos de loss y de acc (pérdida y acierto).

- **Evaluate training simple**

  Este método es igual que el anterior, solo que no utiliza el conjunto de prueba para verificar, sino que utiliza el subconjunto restante de entrenamiento (en este caso las 15.000 imágenes y etiquetas restantes).

- **Model summary**

  Este método nos devuelve información sobre nuestro modelo, como la información sobre las capas, el número total de parámetros utilizados y el número de parámetros entrenados. Este método utiliza por debajo el método summary de keras.

- **Save model**

  Este método nos permite guardar nuestro modelo en una ruta ya predeterminada con un nombre de archivo determinado por el tipo de red neuronal, un número de capas dense y las épocas empleadas para entrenar dicho modelo. También se puede especificar una ruta alternativa.

- **Load model**
  Este método nos permite cargar nuestro modelo previamente almacenado. Para ello utiliza por debajo la función load_model de tensorflow.

7. Implementación

En esta sección se detallará cómo se ha implementado cada una de las redes neuronales, mostrando cuál de ellas ha dado mejor resultado y que configuración de modelado se ha utilizado.

7.1 Red neuronal profunda (DNN)

La red neuronal profunda se corresponde con la clase MultilayerPerceptron en el diseño propuesto. La construcción del modelo se ha basado en el uso de la librería Keras, empleando distintas documentaciones y tutoriales, [9] [10] y utilizando como base modelos ya implementados por otros usuarios [11] [12].

7.1.1 Búsqueda y experimentos de parámetros

Con el fin de obtener una red neuronal que pueda generalizar al máximo y poder predecir el mayor número de imágenes, se ha estado realizando pruebas y experimentos con diferentes parámetros.

En primer lugar, he construido una red profunda, y he automatizado su ejecución (gracias al script adjunto a esta práctica) para una serie de épocas, seleccionando un subconjunto de datos del training como conjunto de entrenamiento, y el subconjunto restante de training como conjunto de prueba. Tras ejecutarlo para un total de 30 épocas, se ha obtenido los siguientes resultados:

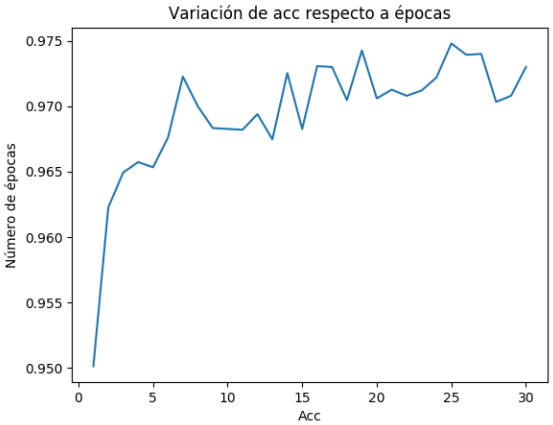

Figura 11: Variación de acc respecto al número de épocas

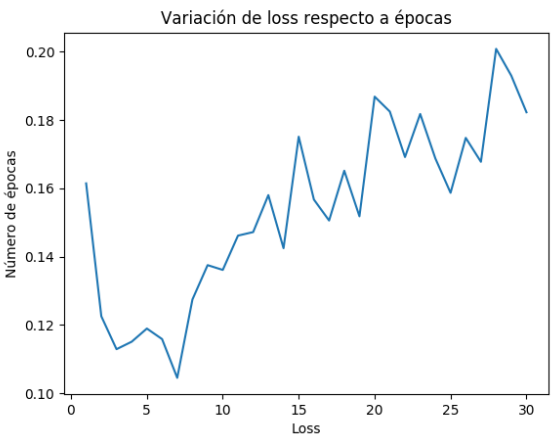

Figura 12: Variación de loss respecto al número de épocas

Como se puede observar, a partir de la 6º época empezamos a tener un buen resultado de acc y donde loss es más bajo. Por lo tanto, a la vista de estos resultados, iré probando la red en intervalos comprendidos entre 15 y 30 épocas, ya que considero que es el intervalo donde el error puede ser más bajo y con un número de épocas computacionalmente viable (por eso he realizado la prueba para el intervalo [1,30])

A continuación se ha estado probando diferentes modelos añadiendo nuevas capas con distintos números de neuronas, probando diferentes funciones de activación y modificando el optimizador del compilador para comprobar el comportamiento de la red ante estos cambios.

Todas estas pruebas se han realizado utilizando como test un subconjunto no entrenado del conjunto de entrenamiento (45.000 training, 15.000 test).

A continuación se va a mostrar una gráfica en la que se puede observar cómo se comporta la red utilizando las siguientes funciones de activación.

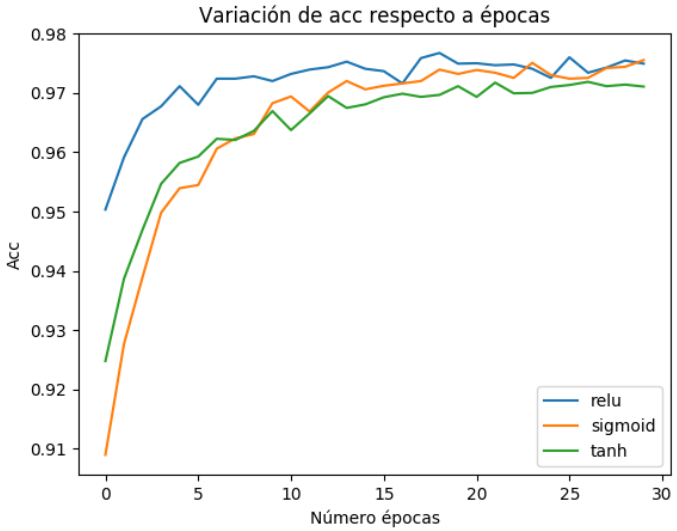

Figura 13: Variación de acc respecto al número de épocas y función de activación

Observando la figura 13, podemos comprobar que el comportamiento inicial en las primeras épocas es un poco diferente. Por ejemplo la función de activación relu tiene mejor porcentaje de acierto en las primeras épocas, pero que a medida de que se van avanzando, tanto la función de activación sigmoid como tanh consiguen casi igualarse.

Como el rango de épocas que se va a utilizar es [15,30] se ha escogido como funciones de activación la función **relu** y **sigmoid,** dado que tienen mejor resultado aunque no difiere en gran cantidad con tanh.

Una vez escogida la función de activación, se ha probado a añadir dropout a la red para observar el comportamiento de la red. En la siguiente figura podemos observar el comportamiento de la red utilizando la función de activación sigmoid al aplicarle dropout.

**Nota: Todos los datos con los que se han representado estas gráficas están adjuntos a esta práctica en results/multilayer_perceptron/training*

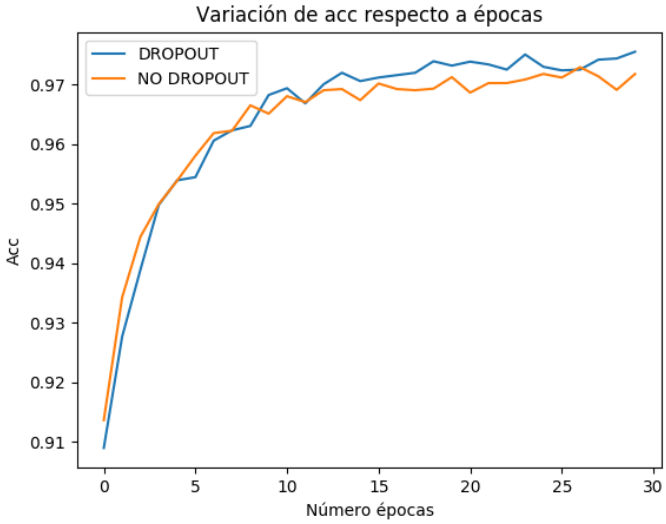

Figura 14: Comparativa usando DROPOUT

Podemos observar que el comportamiento es ligeramente superior al usar DROPOUT. Posteriormente se han realizado más comparativas y verdaderamente se ha comprobado que usando dropout se consigue que la red aprenda mejor y se obtenga un mejor modelo más genérico.

Finalmente, se ha verificado el porcentaje de acierto que tiene el modelo utilizando diferentes optimizadores con las dos funciones de activación relu y sigmoid (que previamente hemos escogido por tener un comportamiento ligeramente superior).

RA: Función activación **relu** con optimizador **adam**
SA: Función activación **sigmoid** con optimizador **adam**
RR: Función activación **relu** con optimizador **RMSprop**
SR: Función activación **sigmoid** con optimizador **RMSprop**
RS: Función activación **relu** con optimizador **SGD**
SS: Función activación **sigmoid** con optimizador **SGD**

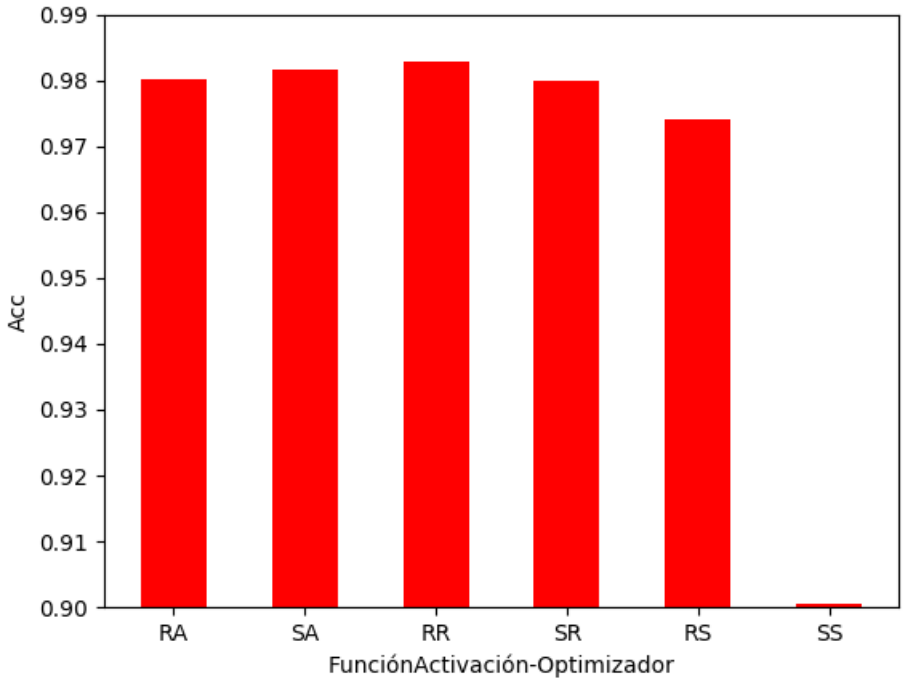

Figura 15: Comparativa usando diferentes funciones de activación y optimizadores

Observando la figura 15, podemos ver que los porcentajes de acierto son muy similares entre todos menos al utilizar la función de activación sigmoid con el optimizador sgd.

Al no haber una gran diferencia significativa, he elegido como optimizadores Adam y RMSprop, ya que dan mejores tasas de acierto.

Las conclusiones finales que se han obtenido son las siguientes:

- Número de épocas: 15-30
- Función de activación: relu o sigmoid
- Dropout : Si
- Optimizador: Adam o RMSprop

7.1.2 Descripción del algoritmo

Tras haber investigado y experimentado con una serie de parámetros, se ha procedido a construir diferentes modelos teniendo en cuenta la información del punto anterior.

En primer lugar se han construido diferentes modelos usando un número variable de capas, de batch size, de dropout… y finalmente se va a describir el que mejor resultado ha dado.

Una vez preprocesados nuestros datos, lo que se ha realizado es añadir en primer lugar la capa de entrada de la red neuronal, utilizando la función Flatten de keras, con el objetivo de aplanar la matriz de datos y construir esa primera capa con 60.000 entradas. En este caso se ha distinguido un posible caso que es utilizar el conjunto de 60.000 imágenes o el conjunto de 45.000 llamado training_sample.

A continuación, se ha añadido una capa con 512 neuronas y con la función de activación sigmoid seguido de una capa dropout con un valor de 0.2.

Posteriormente se ha vuelto añadir otra capa con 512 neuronas con la función de activación sigmoid seguida de otra capa dropout con valor de 0.2

Finalmente se ha añadido una capa de salida con 10 neuronas (una para cada número) con la función de activación softmax [13] para que cada neurona tenga un porcentaje de activación y se active la más alta (esta función es muy útil para clasificación).

El modelo resultante es el siguiente:

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_1 (Dense)	(None, 512)	401920
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262656
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 10)	5130

Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0

Figura 16: Modelo de capas de la red neuronal profunda

Tras haber creado dicho modelo, a continuación se ha compilado utilizando el compilador Adam y entrenado usando el batch size por defecto que es 32 y utilizando 30 épocas.

Los resultados obtenidos son los siguientes:

Entrenamiento	**Test**
- loss: 0.0072	- loss: 0.0969
- acc: 0.9976	- acc: 0.9817
- error: 0.24%	- error: 1.83%

Tiempo total de entrenamiento = 480s

Como podemos observar, el error mínimo que se ha conseguido obtener es de **1.83%**. Para ver más información sobre esto, se puede consultar el directorio adjunto en results\multilayer_perceptron\best.

También se adjunta otros modelos empleados con los que se ha obtenido errores de 1.99% ,2%… empleando otros parámetros.

7.2 Red neuronal convolucional (CNN)

La red neuronal profunda se corresponde con la clase ConvolutionalNetwork en el diseño propuesto. La construcción del modelo se ha basado en el uso de la librería Keras, empleando distintas documentaciones y tutoriales, [8] [14] y utilizando como base modelos ya implementados por otros usuarios [15]

Según [6], este tipo de redes tiene un mejor rendimiento en problemas de clasificación de imágenes, por lo que se va a tratar de mejorar el 1,83% de error que se obtuvo con la red neuronal profunda.

7.2.1 Búsqueda y experimentos de parámetros

En primer lugar, he utilizado la capa conv2D de keras, que permite añadir una capa convolucional especificando unos parámetros. El principal parámetro que he estado modificando es el **kernel size** [16], que define el ancho y alto de la máscara que se va a utilizar para analizar la imagen. He estado observando que se suele aplicar un tamaño de 3x3, 5x5, y en este caso he analizado qué kernel size daba mejor resultado y en mi caso he obtenido mejores resultados con un kernel size de 3x3.

Posteriormente se ha cambiado el valor de **filter** [17] para modificar el número de canales (o filtros con tamaño igual al kernel size) que actúan dentro de la capa conv2D con diferentes valores como 32, 64, 128… y he obtenido mejor resultado con un tamaño de 32.

También se ha probado con diferentes funciones de activación (como en la red DNN) y finalmente he obtenido mejor resultado con sigmoid, descartando la posibilidad de utilizar relu y tanh.

Algo muy utilizado en las redes convolutivas es aplicar maxPooling [18] para reducir aun más el tamaño de la imagen, y en este caso he probado con tamaño 2x2 y 3x3, obteniendo mejor resultado utilizando 2x2.

Respecto al parámetro llamado stride [8] que nos indica cuantos píxeles nos movemos en cada convolución. En este caso lo he dejado siempre por defecto que es 1.

Otra experimento que se ha realizado ha sido el de incluir capas de dropout antes y después de utilizar una capa dense y se ha obtenido un mejor resultado utilizando dropout con valor 0.2.

Finalmente, se ha estado probando con un número diverso de capas convolutivas y profundas. Tras realizar varias pruebas, se ha obtenido una configuración que se mostrará en el siguiente apartado cuando se explique el algoritmo que mejor resultado ha dado.

En esta documentación se aporta un enlace en el que se puede testear una serie de modelos que corresponden a estos experimentos realizados [ENLACE].

7.2.2 Descripción del algoritmo

Tras haber investigado y experimentado utilizando diferentes configuraciones, se va a proceder a describir el algoritmo que mejor resultado ha dado.

En primer lugar se ha añadido una capa convolutiva 2D, en el que se ha establecido un valor para el filtro de 32 y un kernel size de 3x3, dándole como forma el conjunto de imágenes de entrenamiento.

A continuación se añade otra capa convolutiva 2D con los mismos parámetros (filtro de 32 y kernel size de 3x3) pero esta vez añadiendo la función de activación sigmoid.

También se ha añadido un maxPooling [18] para reducir la altura y anchura de la imagen y que de esta forma no se haga tan pesada el procesamiento de dicha imagen en la convolución.

Posteriormente se ha vuelto a duplicar con los mismos parámetros la capa convolutiva y otro maxPooling.

Al resultado de esta última capa, se ha aplicado un dropout con valor 0.2 y ha unido a una red profunda, aplanando los datos de salida con flatten y utilizando una capa profunda de 512 neuronas y la función de activación sigmoid.

Finalmente se ha vuelto a aplicar otro dropout y se ha añadido una capa de salida con 10 neuronas (una por cada número) empleando la función de activación softmax.

El modelo de capas resultante es el siguiente:

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_1 (Dense)	(None, 512)	401920
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262656
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 10)	5130

Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0

Figura 17: Modelo de capas de la red neuronal convolucional

Tras haber creado dicho modelo, a continuación se ha compilado utilizando el compilador Adam y entrenado usando el batch size por defecto que es 32 y utilizando 30 épocas.

Los resultados obtenidos son los siguientes:

Entrenamiento	**Test**
- loss: 0.0072	- loss: 0.02333
- acc: 0.9976	- acc: 0.9935
- error: 0.24%	- error: 0.65%

Tiempo total de entrenamiento = 480s

Como se puede observar, se ha obtenido un error del 0.65% que es bastante más aceptable que el que se obtuvo con la red profunda (1,83%)

Para ver más información sobre esto, se puede consultar el directorio adjunto en results\convolutional_network\best.

También se adjunta otros modelos empleados con los que se ha obtenido errores 0.78%, 0.75%, 0.66%… empleando diferentes parámetros.

8. Ejecución de la aplicación

Para poder ejecutar esta aplicación, es necesario tener instalado lo siguiente:

- **Lenguaje programación:** Python 3.6
- **Frameworks y librerías:**
  - Tensorflow: https://www.tensorflow.org/?hl=es
  - Keras: https://keras.io/
  - Matplotlib: https://matplotlib.org/
  - MNIST: https://github.com/sorki/python-mnist
  - Numpy: http://www.numpy.org/

Tras dicha instalación, basta con modificar los parámetros del main ,crear el tipo de red neuronal que se quiera (ConvolutionalNetwork o MultilayerPerceptron) y ejecutar dicho main. Por defecto guardará el modelo en ./models, pero se puede modificar especificando un path en el método save_model. También se pueden cargar los modelos, activando el parámetro del main asociado a true y especificando la ruta como tercer parámetro de la función load_model.

Para cualquier otra pregunta o duda acerca de cómo he realizado esta práctica, se puede poner en contacto conmigo para quedar en una tutoría e informarle mejor.

Nota*: Adjunto a esta documentación se añade un directorio src donde se ubican los fuentes (sin los datos por motivos de espacio). Para ver todo el material que se ha generado en esta práctica (código fuente, imágenes, modelos, resultados…) se puede acceder al siguiente enlace de google drive donde he compartido todo este material.

https://drive.google.com/drive/folders/10wx8bWffJTm7dFQGK215xxXaYe27mv94?usp=sharing

9. Bibliografía

[1] Ricardo Guerrero Gomez-O, Frameworks de Deep Learning, disponible en https://medium.com/@ricardo.guerrero/frameworks-de-deep-learning-un-repaso-antes-de-acabar-el-2016-5b9bf5b9f9af

[2] matplotlib, disponible en https://matplotlib.org/

[3] THE MNIST DATABASE of handwritten digits, disponible en http://yann.lecun.com/exdb/mnist/

[4] Simple MNIST data parser written in Python, disponible en https://github.com/sorki/python-mnist

[5] scipy.org, Numpy, disponible en http://www.numpy.org/

[6] AMP Tech, Tipos de redes neuronales, disponible en https://www.youtube.com/watch?v=V5BYRPJThjE

[7] Víctor Bonilla Pardo, Machine Learning con TensorFlow, disponible en http://oa.upm.es/49683/1/TFG_VICTOR_BONILLA_PARDO.pdf

[8] AMP Tech, Redes neuronales convolucionales CNN (Clasificación de imagenes), disponible en https://www.youtube.com/watch?v=ns2L2T6wvAY

[9] Karlijn Willems, Keras Tutorial: Deep Learning in Python https://www.datacamp.com/community/tutorials/deep-learning-python?utm_source=adwords_ppc&utm_campaignid=898687156&utm_adgroupid=48947256715&utm_device=c&utm_keyword=&utm_matchtype=b&utm_network=g&utm_adpostion=1t1&utm_creative=255798340456&utm_targetid=dsa-473406573755&utm_loc_interest_ms=&utm_loc_physical_ms=1005414&gclid=CjwKCAiA8rnfBRB3EiwAhrhBGhgSbTAJ06C9SGFE2FDaJ5e5p-JnzGjfBylw6S2vQX7p9R83_ltwzhoCSKQQAvD_BwE

[10] keras.io, Keras: The Python Deep Learning library, disponible en https://keras.io/

[11] sentdex, Deep Learning with Python, TensorFlow, and Keras tutorial, disponible en https://www.youtube.com/watch?v=wQ8BIBpya2k

[12] Jordi Torres.AI, Primeros pasos en Keras, disponible en https://torres.ai/primeros-pasos-en-keras/

[13] AMP Tech, ¿Cómo funciona softmax?, disponible en https://www.youtube.com/watch?v=ma-F0RsMAjQ

[14] keras.io, Convolutional Layers, disponible en https://keras.io/layers/convolutional/

[15] yashk2810, MNIST_keras_CNN-checkpoint.ipynb, disponible en https://github.com/yashk2810/MNIST-Keras/blob/master/.ipynb_checkpoints/MNIST_keras_CNN-checkpoint.ipynb

[16] stats.stackexchange.com, What does kernel size mean?, disponible en https://stats.stackexchange.com/questions/296679/what-does-kernel-size-mean

[17] stackoverflow.com, Keras Conv2D: filters vs kernel_size, disponible en https://stackoverflow.com/questions/51180234/keras-conv2d-filters-vs-kernel-size

[18] Jonathan Hui blog, Convolutional neural networks (CNN) tutorial, disponible en https://jhui.github.io/2017/03/16/CNN-Convolutional-neural-network/