



# Reconocimiento óptico de números escritos a mano en la base de datos MNIST con redes neuronales

Inteligencia Computacional

Pedro Manuel Gómez-Portillo López

gomezportillo@correo.ugr.es

26 de Noviembre de 2018

# Índice general

# Índice de gráficos

# 1. Resumen

Las redes neuronales son una herramienta muy potente que pueden ser entrenadas para resolver todo tipo de problemas.

En esta práctica se hará uso de ellas para reconocer caracteres escritos a mano. Haciendo uso de la base de datos MNIST de números manuscritos y del framework de desarrollo Keras, se entrenará una red neuronal del modo más óptimo posible para que, al evaluarla, reconozca el mayor número de dígitos posibles.

## 1.1. Palabras clave

*MNIST, reconocimiento óptimo de caracteres, deep learning, keras*

# 2. Introducción

Las redes neuronales son un modelo computacional basado en un gran conjunto de neuronas individuales de forma aproximadamente análoga al comportamiento observado en los axones de las neuronas en los cerebros biológicos[1].

Actualmente existe una gran cantidad de programas que hacen uso de las redes neuronales para toda clase de aplicaciones; desde traducción de texto a clústering de datos

Una de estas aplicaciones es el reconocimiento óptico de elementos. Concretamente, la base de datos MNIST[2] es un conjunto de imágenes con números escritos a mano. Esta base de dato tiene un conjunto de 60,000 imágenes, con un tamaño estándar de 28*28 píxeles cada una, y es la que se utilizará en esta práctica.

Concretamente, se utilizará esta base de datos para entrenar una red neuronal y aplicarla al reconocimiento de estos caracteres, intentado acertar en el mayor número posible.

En esta práctica se podría optar o bien por desarrollar una red neuronal desde 0 o bien por utilizar un *framework* o conjunto de librerías ya desarrollado. Esta decisión se basará y se justificará en los capítulos #3.Estado del arte y #4.Implementación.

1 https://www.frontiersin.org/research-topics/4817/artificial-neural-networks-as-models-of-neural-information-processing

2 http://yann.lecun.com/exdb/mnist/

## 2.1. Características de mi ordenador

Mi ordenador personal, en el cual se ha realizado esta práctica, cuenta con las siguientes características.

- **Sistema operativo**: Ubuntu 18.04 LTS
- **Procesador**: Intel Core™ i7-6700HQ Quad-Core  2.6GHz
- **Cache**: 6M
- **Memoria RAM**: 8GB DDR4 2133MHz
- **Tarjeta gráfica**: Sí, pero no recodida por el sistema operativo
- **Disco duro**: SanDisk SSD Plus 256GB

# 3. Estado del arte

Antes de elegir lenguaje de programación a usar se realizó una búsqueda de los frameworks para desarrollo de aplicaciones basadas en *deep learning* disponibles, y de este modo tener razones para poder justificar esta elección.

A continuación se presentan los frameworks más importantes que hay en el mercado.

## 3.1. Keras[3]

Keras es una librería de código abierto escrita en Python, diseñada específicamente para hacer experimentos con redes neuronales.

Lo interesante de este framework es que funciona como **front-end** de la librería que utilice por debajo, es decir, que redirige las llamadas a sus funciones a las de la librería con la que le digamos que trabaje. Keras puede ejecutarse sobre MXNet, DL4J, TensorFlow, o Theano, algunas de las que hablaremos más adelante.

3 https://keras.io/

## 3.2. Pytorch[4]

Pytorch es un framework para Python que permite el prototipado y desarrollo rápido de aplicaciones *deep learning*, especialmente centrado en la aceleración por GPU.

La característica principal de Pytorch es que utiliza grafos computacionales dinámicos por cuestiones de eficiencia y optimización, ya que según su API estos grafos se paralelizan especialmente bien en una tarjeta gráfica.

## 3.3. TensorFlow[5]

TensorFlow es es un framework de código abierto desarrollado por Google que, según su página web, *se utiliza para realizar cálculos numéricos mediante diagramas de flujo de datos; los nodos de los diagramas representan operaciones matemáticas y las aristas reflejan las matrices de datos multidimensionales (tensores) comunicadas entre ellas*.

Este framework trabaja a un nivel más bajo que Keras o Pytorch y, aunque es uno de los más populares y usados por empresas como Dropbox, su uso parece ser recomendado para proyectos más grandes y complejos que éste.

## 3.4. Scikit-learn[6]

Scikit-learn es un framework de código abierto escrita en Python utilizada por empresas como Spotify que, según su página web, es *simple, eficiente y accesible, perfecta para para técnicas de análisis y minería de datos*. Está escrita sobre otras librerías de Python como *NumPy*, *SciPy*, y *matplotlib*.

4 https://pytorch.org/
5 https://www.tensorflow.org/
6 https://scikit-learn.org/

## 3.5. Theano[7]

Theano es un frameworks de bajo nivel, como TensorFlow, y es otro de los soportados por Keras. Según su página web permite *definir, optimizar y evaluar expresiones matemáticas, especialmente las que trabajan con matrices multi-dimensionales*.

Aún así, Theano no está especialmente centrado en el *deep learning*, por lo que en lugar de usar esta librería tiene más sentido usar frameworks como Keras que se apoyen en el para realizar tareas de *deep learning.*

## 3.6. Lasagne[8]

Lasagne es una librería escrita en Python que nació exclusivamente para permitir usar Theano en tareas de *deep* learning, y proporcionando una interfaz más amigable. Es una de las principales competidores de Keras, aunque parece no ser tan preferida como esta

## 3.7. DSSTNE[9]

DSSTINE (pronunciado *destiny*), es un framework de código abierto desarrollado por Amazon especialmente pensado para entrenar y desplegar modelos de recomendación.

Además, únicamente permite ejecutarse sobre GPU, aunque permite hacerlo en paralelo usando varias. Por otro lado, su documentación es muy pobre y a veces es necesario bucear en su código fuente para entender qué hace.

7 http://www.deeplearning.net/software/theano/
8 https://lasagne.readthedocs.io/
9 https://github.com/amzn/amazon-dsstne

## 3.8. MXNet[10]

MXNet es una librería de uso general desarrollada por Apache. A pesar de asegurarse bastante potente parece estar en una fase muy verde.

## 3.9. DL4J[11]

DeepLearning4J es una librería distribuida de código abierto escrita en Java y disponible para Java, Python y C++.

Según su página web, se especializa en redes neuronales profundas, y su documentación parece muy completa y está muy bien escrita.

## 3.10. Microsoft Cognitive Toolkit[12]

CNTK es una librería de código abierto desarrollada por Microsoft Research, la división de investigación de Microsoft que, según su página web, *entrena algoritmos de deep learning para pensar como personas.*

A pesar de lo prometedor que parecía, no parece ser muy popular, ya que en comparación con el resto de frameworks no hay muchos ejemplos por la web, ni siquiera en páginas especializadas como Kaggle[13].

10 https://mxnet.incubator.apache.org/
11 https://deeplearning4j.org/
12 https://www.microsoft.com/en-us/cognitive-toolkit/
13 https://www.kaggle.com/

# 4. Implementación

Tras realizar el estudio acerca del estado del arte, y basándonos que necesitamos un framework centrado en *deep learning* que no necesite GPU para ejecutarse, se ha decidido utilizar el lenguaje de programación Python3 con el framework Keras con TensorFlow como back-end.

Cabe destacar que, aunque me hubiera parecido mucho más interesante programar la práctica desde 0, como se proponía en el guión, en lugar de utilizar una librería ya implementada, las restricciones de tiempo y esfuerzo a las que ha tenido que enmarcase esta práctica debido al resto de asignaturas han hecho que finalmente opte por utilizar una framework de desarrollo de aplicaciones *deep learning*.

## 4.1. Framework y librerías utilizadas

A continuación se presentan los programas más importantes utilizados en esta práctica, así como su versión específica. Se han obviado las dependencias.

- **Python 3.6.6**. Será el lenguaje de programación de la práctica.
- **Keras 2.2.4.** Se utilizará como front-end para trabajar con TensorFlow, ya que actúa como interfaz facilitando su uso.
- **TensorFlow 1.13.0**. Será el corazón de la aplicación

- **Matplotlib 3.0.0**. Se utilizará para visualizar el contenido de la base de datos MNIST.

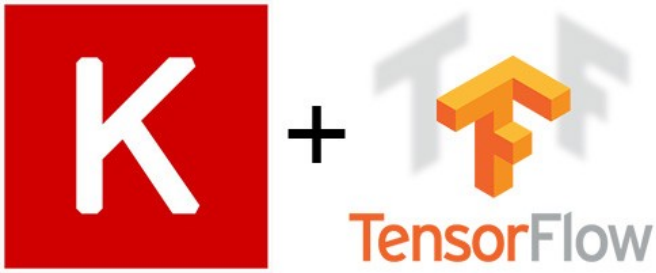


*Gráfico 1: Logos de Keras y TensorFlow*

Para instalar estas librerías usando el proyecto solo es necesario situarse en el directorio raíz y ejecutar `make install`.

## 4.2. Detalles comunes de la implementación

A continuación se presentan la información de la implementación del proyecto. Primero se presentará la información común a todas las versiones del proyecto y a continuación se pasará a describir cada versión por separado. Los resultados detallados de cada versión pueden verse en la sección #5.Resultados mientras que aquí solo se presentará la configuración de cada una.

En total, este proyecto ha pasado por tres versiones, pero es necesario destacar que las versiones no son totalmente sucesivas, sino que entre ellas se realizaron varias subversiones probando diferentes configuraciones de épocas y capas, y no se decidía definir una nueva versión hasta obtener una mejoría suficiente.

Además, se ha utilizado un repositorio en GitHub para realizar el control de versiones de la práctica, cuya dirección es la siguiente,

https://github.com/gomezportillo/mnist.

Al final de cada versión se ha publicado una *release* del proyecto, por lo que las tres versiones pueden verse en la correspondiente sección del repositorio.

Todas las versiones utilizan las mismas librerías, por lo que para probar cualquiera de ellas, y habiendo clonado previamente el repositorio e instalado las dependencias, bastaría con situarse en el directorio raiz y ejecutar `git checkout v{x}.0`, donde `{x}` es la versión del proyecto, entre 1 y 3, a la que nos queremos mover, y escribir `make` para ejecutar la práctica en dicha versión.

## 4.3. Primera versión

La primera versión fue la más larga y complicada, ya que hubo que configurar todo el entorno de desarrollo para poder trabajar con las librerías elegidas.

En un esfuerzo por hacer los resultados de este proyecto reproducibles, se ha configurado un archivo `Makefile` con las instrucciones de instalación necesarias para volver a preparar el mismo entorno de desarrollo. Para instalarlo, simplemente es necesario situarse en el directorio raíz del proyecto y ejecutar `make install`, lo que básicamente instalará Python3, luego pip3 y lo usará para instalar las librerías indicadas en el archivo `requirements.txt`.

Tras instalar Keras y TensorFlow y sus dependencias fue posible empezar a trabajar en la práctica. Como nunca había trabajado con estas librerías acudí al sitio web de Keras y seguí su tutorial de iniciación[14]. En este tutorial se presenta el lenguaje y los principales pasos que hay que seguir para definir y configurar una red neuronal desde cero, así que aunque en el tutorial se utiliza la base de datos *Fashion MNIST*[15] de prendas de vestir, fue fácil adaptarla para utilizar la base de datos *MNIST*.

Tras haber configurado el entorno de desarrollo, haber entendido cómo trabajar con el framework Keras y haber obtenido mis primeros resultados, la configuración final de capas fue la siguiente, usando un modelo secuencial con 15 épocas de entrenamiento.

14 https://www.tensorflow.org/tutorials/keras/basic_classification
15 https://github.com/zalandoresearch/fashion-mnist

- La primera capa, llamada *Flatten*, transforma las imágenes de la base de datos de una matriz bidimensional de 28x28 píxeles a un vector unidimensional de 784 píxeles (28*28), lo que permite que la salida sea procesada por capas totalmente conectadas.

- La segunda capa, llamada *Dense*, es una capa totalmente conectada con 128 neuronas y una función de activación de rectificación lineal.

- La tercera capa, llamada *Dense,* es la capa de salida, con 10 neuronas (una por cada clase) y una función de activación *softmax* para poder realizar predicciones probabilísticas más tarde.

La siguiente imagen, generada con la función de Keras `plot_model` a partir del modelo, refleja la configuración de capas utilizada.

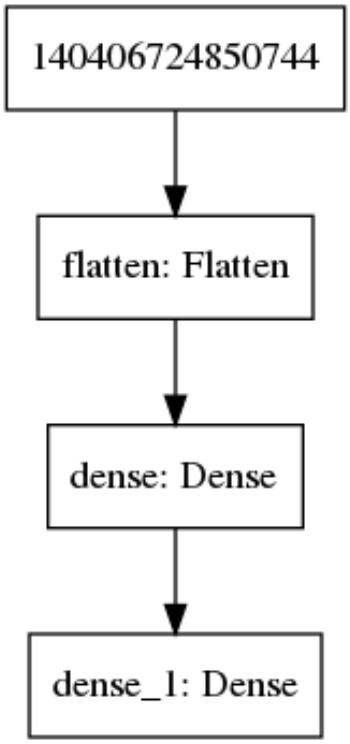


*Gráfico 2: Configuración de capas de la primera versión*

Con esta configuración tan simple pude obtener un porcentaje de acierto sobre el conjunto de entrenamiento del **99.8%** y un **97.62%** sobre el conjunto de prueba, que me pareció un resultado más que aceptable para una primera versión. Los resultados de esta versión pueden verse detallados en profundidad en el siguiente capítulo.

La ejecución de esta versión duró un poco más que la primera (~300 segundos frente a los solamente ~20 segundos de la primera), así que aunque al principio me planteé usar algún servicio PaaS[16] como *Amazon Web Services* para ejecutar la práctica, finalmente decidí que debido a su coste y a que los tiempos no eran excesivamente altos podía seguir ejecutándola en mi ordenador personal.

El enlace de esta versión en el repositorio es el siguiente,

https://github.com/gomezportillo/mnist/tree/v1.0

## 4.4. Segunda versión

Como ya tenía configurado el entorno de desarrollo y sabía cómo trabajar con el framework elegido, en la segunda versión pude navegar tranquilamente por la API de Keras[17] para conocer y entender todas las funciones y parámetros que tiene.

Al final realicé una configuración de 5 capas, usando un modelo secuencial con 15 épocas de entrenamiento.

- La primera capa, llamada *Conv2D*, es una capa convolutiva con la función de activación de rectificación lineal y una ventana convolutiva de 3x3.

- La segunda capa, llamada *MaxPooling2D*, es una capa de operación de agrupación máxima con un tamaño de *pool* de 2x2.

- La tercera capa, llamada *Dropout*, descarta el 25% de las neuronas aleatoriamente para evitar problemas de sobreapendizaje en la fase de entrenamiento.

16 Platform as a Service
17 https://keras.io/

- La cuarta capa, llamada *Flatten*, convierte los datos de entrada de una matriz bidimensional de 28x28 a un vector unidimensional para que puedan ser procesados por capas totalmente conectadas.

- La quinta capa, llamada *Dropout*, tiene un número de nodos igual al número de clases del modelo, del 0 al 9, con la función de activación *softmax* para poder realizar predicciones más adelante.

La siguiente imagen refleja la configuración de capas utilizada.

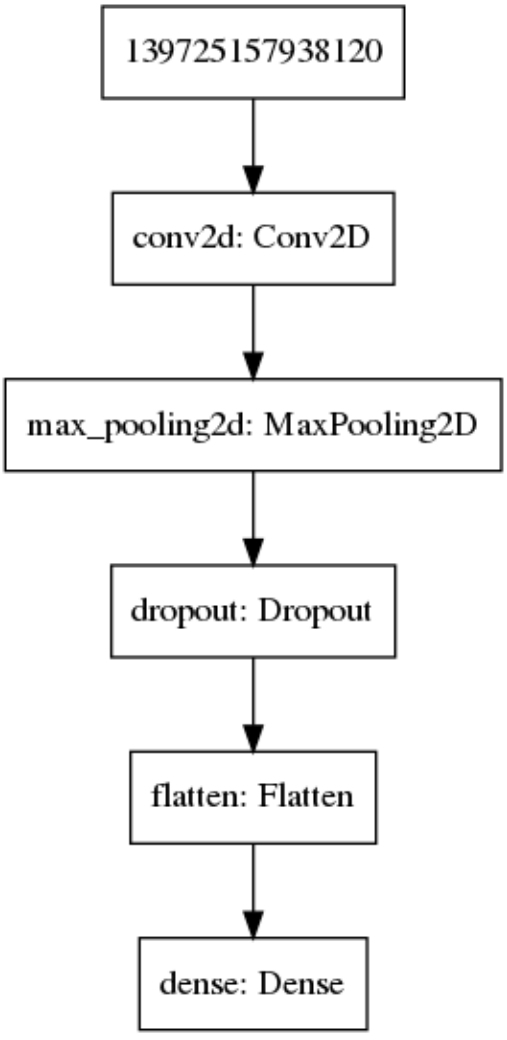


*Gráfico 3: Configuración de capas de la segunda versión*

Con esta configuración, algo más compleja, ya que usa capas convolutivas e intenta evitar el sobreaprendizaje de la red usando funciones *dropout*, fue posible obtener un porcentaje de acierto sobre el conjunto de entrenamiento de un **99.27%** y un **98.22%** sobre el conjunto de prueba.

Me pareció interesante que, comparado con la versión anterior, normalmente obtenía resultados algo peores en el conjunto de entrenamiento pero casi un 1% mejores en el conjunto de prueba, lo que achaqué a que había solucionado los problemas de sobreaprendizaje.

El enlace de esta versión en el repositorio es el siguiente,

https://github.com/gomezportillo/mnist/tree/v2.0

## 4.5. Tercera versión

Por último, en la tercera versión del proyecto necesitaba un porcentaje de aciertos lo más cercano al 100% posible tanto en el conjunto de entrenamiento como en el prueba.

El resultado final fue una configuración de 9 capas, usando un modelo secuencial con 20 épocas de entrenamiento. Como ya había resuelto el problema del sobreaprendizaje pude usar más épocas de entrenamiento que en las versiones anteriores.

- La primera capa es una capa convolutiva llamada *Conv2D* con una función de activación de rectificación lineal, 32 mapas característicos y una ventana convolutiva de 4x4.

- La segunda capa es una capa convolutiva llamada *Conv2D* con la función de activación de rectificación lineal y una ventana convolutiva de 3x3.

- La tercera capa, llamada *MaxPooling2D, es una capa* de operación de agrupación máxima con un tamaño de *pool* de 2x2.

- La cuarta capa, llamada *Dropout*, descarta el 20% de las neuronas aleatoriamente para evitar problemas de sobreapendizaje en la fase de entrenamiento.

- La quinta capa, llamada *Flatten*, convierte los datos de la matriz bidimensional en un vector lineal para que la sexta capa pueda ser una capa completamente conectada.

- La sexta capa, llamada *Dense*, es una capa completamente conectada con 248 neuronas y función de activación de rectificación lineal.

- La séptima capa, llamada *Dense*, es una capa completamente conectada con 124 neuronas (la mitad que la anterior) y función de activación de rectificación lineal.

- La octava capa, llamada *Dropout*, vuelve a descartar el 40% de las neuronas aleatoriamente.

- La novena capa, llamada *Dense*, es la capa de salida, con 10 neuronas (una por cada clase) y una función de activación *softmax* para poder realizar predicciones probabilísticas más adelante.

La imagen inferior refleja la configuración final de capas utilizada.

Con esta configuración pude obtener un porcentaje de aciertos sobre el conjunto de prueba de un **99.91%** y un **99.15%** sobre el conjunto de prueba.

El enlace de esta versión en el repositorio es el siguiente,

https://github.com/gomezportillo/mnist/tree/v3.0

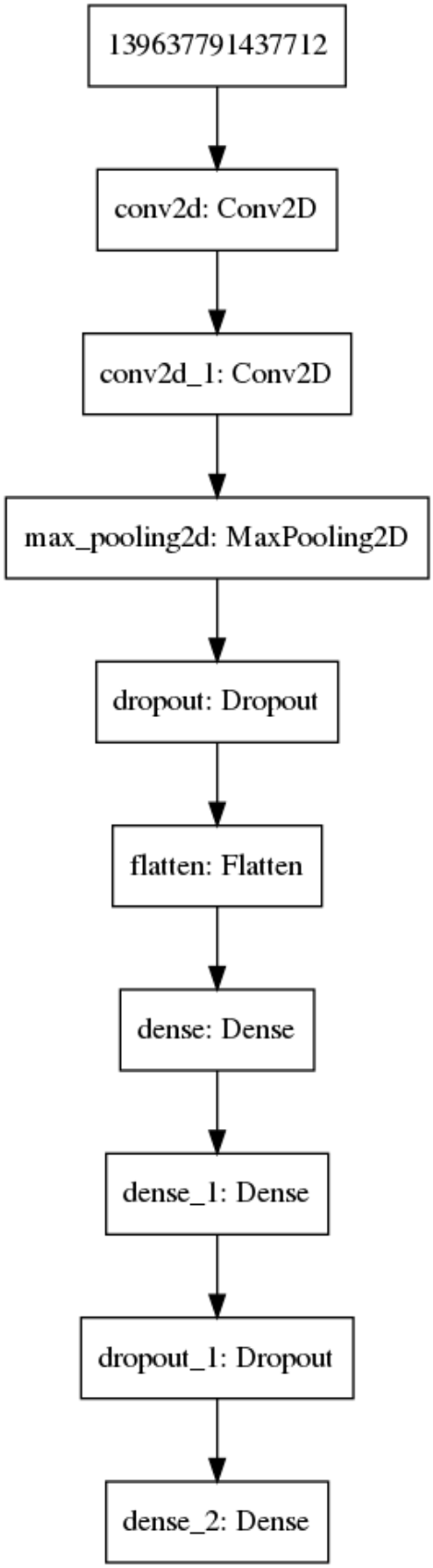


*Gráfico 4: Configuración de capas de la tercera versión*

# 5. Resultados

A continuación se presentan los resultados detallados de las diferentes versiones del proyecto.

## 5.1. Primera versión

Tras ejecutar la primera versión, cuya fase de entrenamiento duró **22.2 segundos**, obtuve los siguientes resultados, formateados en una tabla. La columna *Época* indica a qué época del entrenamiento hacen referencia los resultados, la columna *Loss* es un valor escalar que intentamos reducir a 0 y la columna *Acierto* indica el porcentaje de aciertos que se han obtenido.

| Época | Loss | Acierto | Época | Loss | Acierto |
|---:|---:|---:|---:|---:|---:|
| 1 | 0.3603 | 0.9007 | 11 | 0.0271 | 0.9927 |
| 2 | 0.1674 | 0.9526 | 12 | 0.0240 | 0.9939 |
| 3 | 0.1205 | 0.9656 | 13 | 0.0209 | 0.9946 |
| 4 | 0.0929 | 0.9736 | 14 | 0.0174 | 0.9958 |
| 5 | 0.0750 | 0.9784 | 15 | 0.0149 | 0.9965 |
| 6 | 0.0622 | 0.9819 | | | |
| 7 | 0.0524 | 0.9853 | | | |
| 8 | 0.0439 | 0.9877 | | | |
| 9 | 0.0375 | 0.9895 | | | |
| 10 | 0.0326 | 0.9911 | | | |

El siguiente gráfico refleja los datos de la tabla anterior.

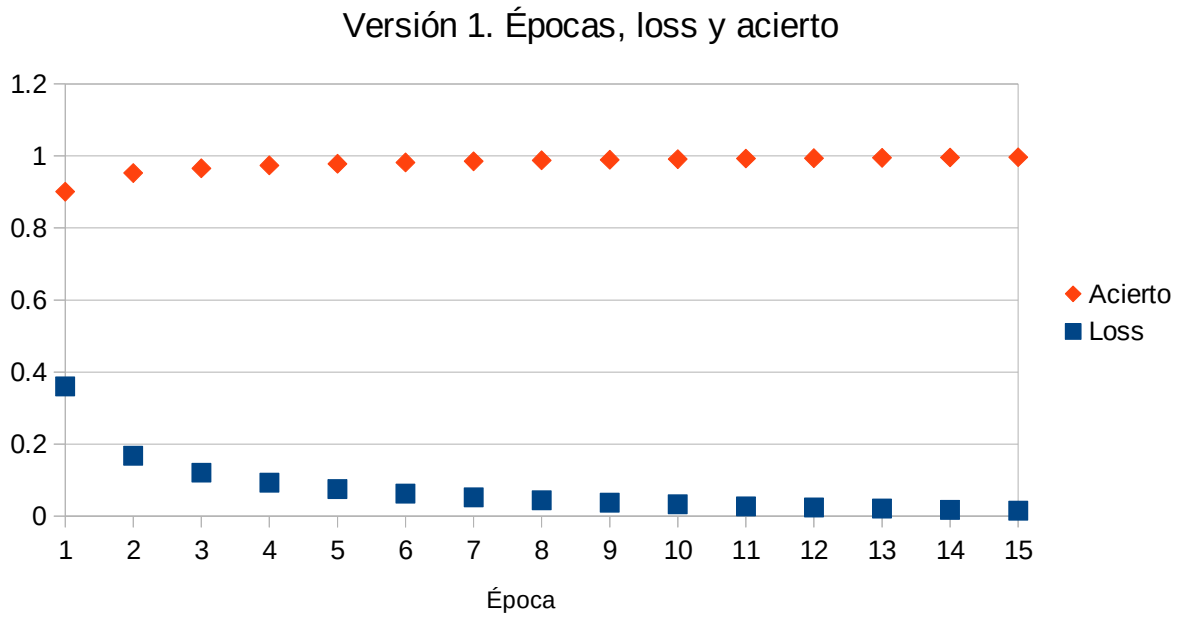


*Gráfico 5: Resultados de la fase de entrenamiento de la primera versión*

Tras la fase de entrenamiento evalué el modelo, primero con el conjunto de entrenamiento y luego con el conjunto de prueba, y obtuve los siguientes resultados.

| **Conjunto** | **Loss** | **Acierto** |
|---|---|---|
| Entrenamiento | 0.01141 | 99.80500% |
| Prueba | 0.07754 | 97.62000% |

A modo de ejemplo, se ha incluido la salida de la consola tras ejecutar esta versión, que puede comprobarse en el #8.1.Anexo 1. Salida de la consola tras ejecutar la primera versión.

## 5.2. Segunda versión

Tas ejecutar la primera versión, cuya fase de entrenamiento duró **309.8 segundos**, obtuve los siguientes resultados.

| Época | Loss | Acierto | Época | Loss | Acierto |
|---|---|---|---|---|---|
| 1 | 0.3648 | 0.8989 | 11 | 0.0497 | 0.9845 |
| 2 | 0.1361 | 0.9618 | 12 | 0.0478 | 0.9854 |
| 3 | 0.1361 | 0.9712 | 13 | 0.0455 | 0.9860 |
| 4 | 0.0847 | 0.9757 | 14 | 0.0410 | 0.9872 |
| 5 | 0.0759 | 0.9773 | 15 | 0.0404 | 0.9874 |
| 6 | 0.0691 | 0.9791 | | | |
| 7 | 0.0628 | 0.9815 | | | |
| 8 | 0.0599 | 0.9821 | | | |
| 9 | 0.0567 | 0.9830 | | | |
| 10 | 0.0522 | 0.9835 | | | |

El gráfico inferior refleja los datos de la esta tabla.

Tras la fase de entrenamiento evalué el modelo, primero con el conjunto de entrenamiento y luego con el conjunto de prueba, y obtuve los siguientes resultados.

| Conjunto | Loss | Acierto |
|---|---|---|
| Entrenamiento | 0.02590 | 99.27000% |
| Prueba | 0.05184 | 98.22000% |

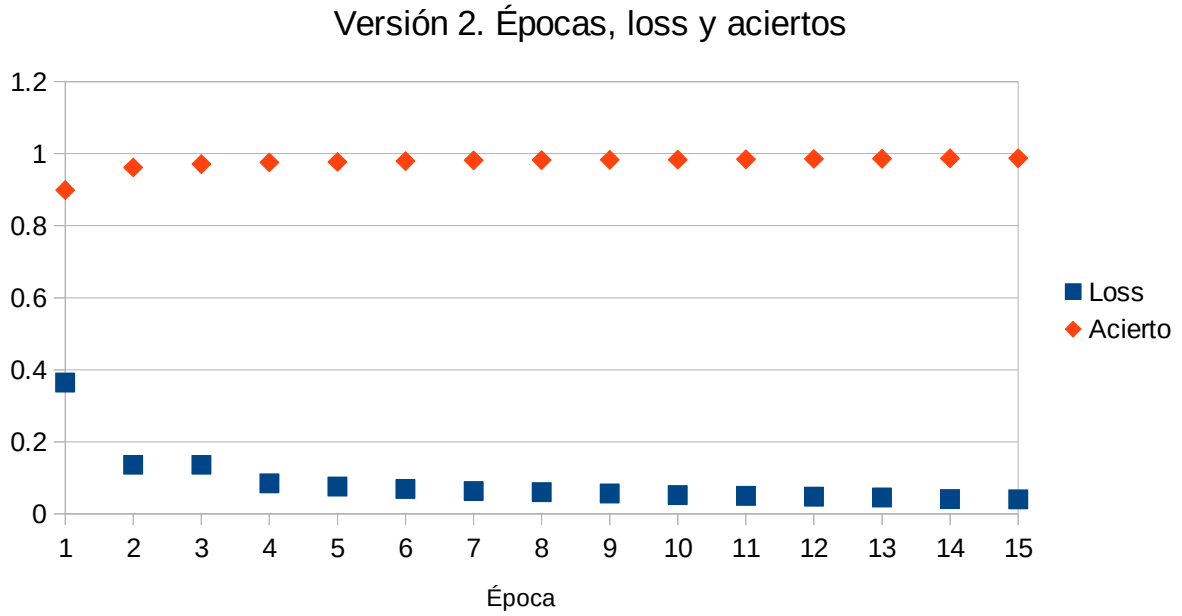


*Gráfico 6: Resultados de la fase de entrenamiento de la segunda versión*

## 5.3. Tercera versión

Tas ejecutar la primera versión, cuya fase de entrenamiento duró **309.8 segundos**, obtuve los siguientes resultados.

| Época | Loss | Acierto | Época | Loss | Acierto |
|---|---|---|---|---|---|
| 1 | 0.1970 | 0.9389 | 11 | 0.0093 | 0.9971 |
| 2 | 0.0560 | 0.9831 | 12 | 0.0088 | 0.9973 |
| 3 | 0.0385 | 0.9882 | 13 | 0.0074 | 0.9978 |
| 4 | 0.0288 | 0.9909 | 14 | 0.0078 | 0.9975 |
| 5 | 0.0227 | 0.9934 | 15 | 0.0055 | 0.9983 |
| 6 | 0.0198 | 0.9940 | 16 | 0.0076 | 0.9977 |
| 7 | 0.0160 | 0.9952 | 17 | 0.0060 | 0.9983 |
| 8 | 0.0130 | 0.9957 | 18 | 0.0064 | 0.9980 |
| 9 | 0.0114 | 0.9964 | 19 | 0.0039 | 0.9988 |
| 10 | 0.0086 | 0.9972 | 20 | 0.0054 | 0.9985 |

El siguiente gráfico refleja los datos de la esta tabla.

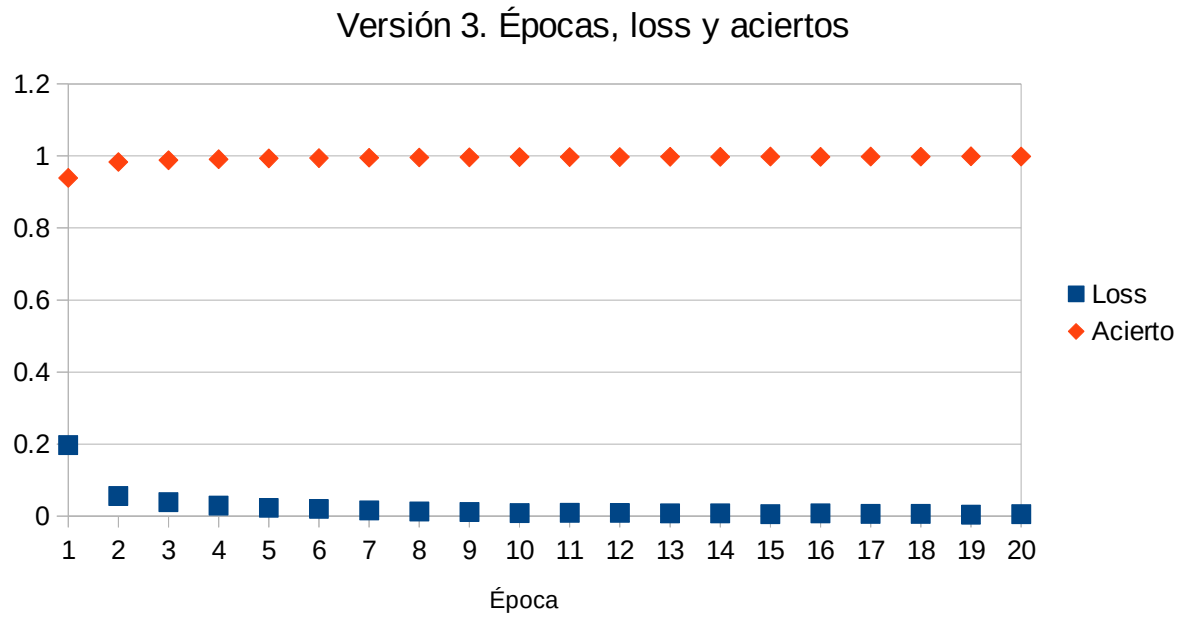


*Gráfico 7: Resultados de la fase de entrenamiento de la tercera versión*

Tras la fase de entrenamiento evalué el modelo, primero con el conjunto de entrenamiento y luego con el conjunto de prueba, y obtuve los siguientes resultados.

| **Conjunto** | **Loss** | **Acierto** |
|---|---|---|
| Entrenamiento | 0.00237 | 99.90167% |
| Prueba | 0.04271 | 99.15000% |

Por lo tanto, el **resultado final** del proyecto sobre el conjunto de prueba es de un 99.15%, lo que supone tan solo un **0.85% de fallos sobre los 10,000 números manuscritos**, o lo que es lo mismo, 85 errores.

# 6. Conclusiones

Las redes neuronales son un campo muy profundo y, aunque esta práctica solo me ha servido como introducción a él, me alegro mucho de haberla realizado. Sé que, tarde o temprano, en mi futura vida laboral tendré que realizar un proyecto basado en redes neuronales y esta práctica me servirá para recordar conceptos y tener un punto de partida.

Creo que ha sido un modo muy bueno de aplicar los términos y conceptos estudiados en teoría, viendo cómo la práctica avanzaba a través de sus diferentes versiones, aunque al haberlo ejecutado en local los tiempos de espera hayan sido, sobre todo hacia el final de la práctica, bastante elevados.

Personalmente estoy muy contento con los resultados obtenidos. He pasado de conocer las redes neuronales de un modo teórico a configurar y usar un framework de desarrollo de *deep learning* y obtener unos resultados muy cercanos a un 100% de acierto.

# 7. Referencias


- https://www.tensorflow.org/tutorials/keras/basic_classification
- https://www.tensorflow.org/guide/keras
- https://github.com/erseco/ugr_inteligencia_computacional
- https://nextjournal.com/gkoehler/digit-recognition-with-keras
- https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py
- http://www.machinelearningtutorial.net/2016/12/24/python-keras-mnist/
- https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/
- https://pythonprogramming.net/mnist-python-playing-neural-network-tensorflow/
- https://mxnet.incubator.apache.org/tutorials/python/mnist.html
- https://platzi.com/blog/librerias-de-machine-learning-tensorflow-scikit-learn-pythorch-y-keras/
- https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py
- https://www.kaggle.com/moghazy/guide-to-cnns-with-data-augmentation-keras

# 8. Anexos

## 8.1. Anexo 1. Salida de la consola tras ejecutar la primera versión

```
/usr/bin/python3 src/mnist.py

Using TensorFlow backend.

Using TensorFlow v1.12.0

The number of occurrence of each number in the train set is {0:
5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7:
6265, 8: 5851, 9: 5949}

The number of occurrence of each number in the test set is {0:
980, 1: 1135, 2: 1032, 3: 1010, 4: 982, 5: 892, 6: 958, 7: 1028,
8: 974, 9: 1009}

Train on 60000 samples, validate on 60000 samples

Epoch 1/15

2018-11-22 23:14:01.439601: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU
supports instructions that this TensorFlow binary was not
compiled to use: AVX2 FMA
```

```
60000/60000 [==============================] - 2s 27us/step -
loss: 0.3603 - acc: 0.9007 - val_loss: 0.1898 - val_acc: 0.9472


Epoch 2/15


60000/60000 [==============================] - 1s 24us/step -
loss: 0.1674 - acc: 0.9526 - val_loss: 0.1290 - val_acc: 0.9639


Epoch 3/15


60000/60000 [==============================] - 1s 25us/step -
loss: 0.1205 - acc: 0.9656 - val_loss: 0.0996 - val_acc: 0.9711


Epoch 4/15


60000/60000 [==============================] - 1s 25us/step -
loss: 0.0929 - acc: 0.9736 - val_loss: 0.0774 - val_acc: 0.9777


Epoch 5/15


60000/60000 [==============================] - 1s 24us/step -
loss: 0.0750 - acc: 0.9784 - val_loss: 0.0606 - val_acc: 0.9826


Epoch 6/15


60000/60000 [==============================] - 1s 24us/step -
loss: 0.0622 - acc: 0.9819 - val_loss: 0.0499 - val_acc: 0.9866


Epoch 7/15


60000/60000 [==============================] - 1s 25us/step -
loss: 0.0524 - acc: 0.9853 - val_loss: 0.0417 - val_acc: 0.9884


Epoch 8/15


60000/60000 [==============================] - 2s 25us/step -
loss: 0.0439 - acc: 0.9877 - val_loss: 0.0369 - val_acc: 0.9902


Epoch 9/15
```

```
60000/60000 [==============================] - 1s 25us/step -
loss: 0.0375 - acc: 0.9895 - val_loss: 0.0288 - val_acc: 0.9930


Epoch 10/15


60000/60000 [==============================] - 1s 24us/step -
loss: 0.0326 - acc: 0.9911 - val_loss: 0.0243 - val_acc: 0.9942


Epoch 11/15


60000/60000 [==============================] - 1s 24us/step -
loss: 0.0271 - acc: 0.9927 - val_loss: 0.0218 - val_acc: 0.9952


Epoch 12/15


60000/60000 [==============================] - 1s 24us/step -
loss: 0.0240 - acc: 0.9939 - val_loss: 0.0168 - val_acc: 0.9963


Epoch 13/15


60000/60000 [==============================] - 2s 26us/step -
loss: 0.0209 - acc: 0.9946 - val_loss: 0.0161 - val_acc: 0.9966


Epoch 14/15


60000/60000 [==============================] - 2s 26us/step -
loss: 0.0174 - acc: 0.9958 - val_loss: 0.0128 - val_acc: 0.9973


Epoch 15/15


60000/60000 [==============================] - 2s 25us/step -
loss: 0.0149 - acc: 0.9965 - val_loss: 0.0114 - val_acc: 0.9980


Training time: 22.613s


60000/60000 [==============================] - 1s 22us/step


Train loss: 0.01141


Train accuracy: 99.80500%
```

```
10000/10000 [==============================] - 0s 23us/step

Test loss: 0.07754

Test accuracy: 97.62000%
```

## 8.2. Anexo 2. Estructura de directorios de la práctica

Salida del comando `tree` ejecutado en el directorio raíz del proyecto.

```
[...]/mnist/$ tree

.
├── deliverables
│   ├── ver_1
│   │    ├── assigned_labels_test.txt
│   │    ├── assigned_labels_train.txt
│   │    ├── console.output
│   │    ├── model.png
│   │    ├── README.md
│   │    └── result.txt
│   ├── ver_2
│   │   ├── assigned_labels_test.txt
│   │   ├── assigned_labels_train.txt
│   │   ├── console.output
│   │   ├── model.png
│   │   ├── README.md
│   │   └── result.txt
│   └── ver_3
```

```
│      ├── assigned_labels_test.txt
│      ├── assigned_labels_train.txt
│      ├── console.output
│      ├── model.png
│      ├── README.md
│      └── result.txt
├── doc
│    ├── documentacion.odt
│    ├── documentacion.pdf
│    ├── guion_practica.pdf
│    └── sheets
│        └── epoch_vs_loss.ods
├── LICENSE
├── Makefile
├── README.md
├── requirements.txt
└── src
    ├── aux.py
    ├── mnist_functions.py
    └── mnist.py
```

- El direcotrio **deliverables/** contiene los resultados de cada versión del proyecto.

- La carpeta **doc/** contiene la documentación final y el guión de la práctica.

- Los archivo **LICENSE** y **README.txt** son archivos de configuración del repositorio.

- El archivo **Makefile** sirve para instalar y ejecutar el proyecto.

- El archivo **requirements.txt** contiene los nombres y las versiones de las librerías utilizadas en el proyecto, y es usado por la herramienta `pip` para instalarlas.

- El directorio **src/** contiene los archivos de código fuente de la práctica.

    - **aux.py** contiene funciones auxiliares utilizados para renderizar las imágenes de la base de datos.

    - **mist_functions.py** contiene las llamadas a la API de Keras envueltas en funciones personalizadas ya que, en un intento por aumentar la claridad y la mantenibilidad del proyecto, el archivo principal llama a estas funciones.

    - **mnist.py** es el archivo principal y contiene la inicialización de las librerías, la configuración de las capas de la red neuronal y las llamadas a las funciones de *mnist_functions.py*.