UNIVERSIDAD DE GRANADA

MNIST:
Práctica de redes neuronales

Autores
Juan Manuel Castillo Nievas

Máster Profesional en Ingeniería Informática

—

Granada, 7 de diciembre de 2020

Índice

1. Introducción

En este documento se va a presentar el entrenamiento de distintas redes neuronales sobre un conjunto de entrenamiento para el reconocimiento óptico de imágenes y se van a evaluar los resultados utilizando un conjunto de prueba. La base de datos **MNIST** [1] está formada por imágenes de dígitos escritos a mano. Cada dígito tiene un tamaño de 28x28 y está centrado. **MNIST** contiene 60.000 ejemplos de entrenamiento y 10.000 ejemplos de prueba.

En la Figura 1 se puede visualizar un ejemplo de un dígito en la base de datos **MNIST**. Cada píxel contiene un número de tipo **uint8** que está en el rango [0-255] y que representa el color en una escala de grises.

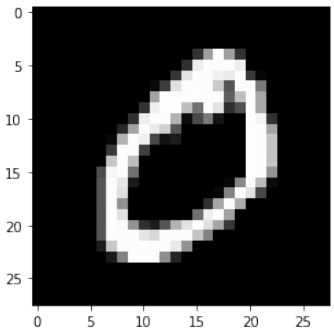

Figura 1: Ejemplo de la base de datos **MNIST**

El objetivo de esta práctica es entrenar distintas redes neuronales ajustando los hiperparámetros para conseguir el mayor número de aciertos posible en el reconocimiento de dígitos.

2. Entorno de trabajo

Para el desarrollo de esta práctica se ha elegido **Python** como lenguaje de programación. En **Python** se puede encontrar la biblioteca de código abierto **Keras** [2] que es una API para desarrollar y experimentar distintas redes neuronales de una forma muy sencilla y con una documentación muy clara.

Realmente **Keras** es una API de alto nivel de **Tensorflow** [3]. De acuerdo a la definición en su propia web, **Tensorflow** es "la principal biblioteca de código abierto para enseñarte a desarrollar y entrenar modelos de aprendizaje automático." Sin embargo, esta biblioteca está diseñada para construir modelos de aprendizaje automático y resulta más compleja y menos *user-friendly*, mientras

que **Keras** está diseñada para redes neuronales específicamente [4]. De acuerdo a las ventajas de **Keras** proporcionadas en la misma web de **Tensorflow** [5]: "Keras tiene una interfaz simple y consistente optimizada para casos de uso comun. Proporciona informacion clara y procesable sobre los errores del usuario."

Para el desarrollo de código se ha usado **Google Colab** [6]. Realmente **Google Colab** está basado en **Jupyter** [7], pero ofrece la ventaja de poder usar y compartir *jupyter notebooks* sin la necesidad de descargarlo o instalarlo. Además, ofrece acceso gratuito a recursos informáticos incluyendo GPUs [8].

Resumiendo en una lista el entorno de trabajo:

- **Google Colab** para el desarrollo de código
- **Python 3.6.9** como lenguaje de programación
- **Keras 2.4.0** como biblioteca para implementar las redes neuronales
- **Tensorflow 2.3.0** como biblioteca para construir modelos de aprendizaje automático (y que usa Keras)
- **Matplotlib** como biblioteca usada en Python para construir y visualizar imágenes y gráficas

3. Antes del entrenamiento

En esta sección se van a explicas los detalles de código que se hacen previamente a la implementación de la red neuronal.

3.1. Obtención del conjunto MNIST

Keras cuenta con una función que carga directamente los conjuntos de entrenamiento y prueba que se encuentran en **MNIST**. En el Código 1 se encuentra la función que devuelve el conjunto **MNIST**. La salida de esta función se divide en lo siguiente:

- **Training_images:** el conjunto de imágenes de entrenamiento
- **Training_labels:** las etiquetas (clases) de cada ejemplo en el conjunto de entrenamiento
- **Test_images:** el conjunto de imágenes de prueba

- **Test_labels:** las etiquetas (clases) de cada ejemplo en el conjunto de prueba

```
(training_images, training_labels), (test_images, test_labels) =
.keras.datasets.mnist.load_data(path="mnist.npz")
```

Listing 1: Obtención del conjunto **MNIST** en **Keras**

3.2. Preparación de los datos

Sea cual sea la implementación que se use, primero hay que hacer un preprocesamiento de los datos del dataset **MNIST** para poder manejar los datos de forma más fácil. Este dataset ya tiene todas las imágenes centralizadas, pero recordemos que cada imagen es una matriz de 28x28 en el que cada elemento es de tipo **uint8** y con un número en el rango [0-255]. El preprocesamiento realizado es el siguiente:

1. **Normalización de los datos:** se normalizan todos los píxeles de la imagen dividiendo entre 255. De esta forma, cada imagen estará formada por píxeles cuyo valor está en el rango [0,1]
2. **Cambiar la estructura de cada imagen de matriz a vector:** cada imagen es una matriz de 28x28. El conjunto **MNIST** tiene un formato de Mx28x28. Se utiliza la función **.reshape()** para convertir cada imagen en un vector de 784 elementos.
3. **Pasar de uint8 a float32:** cada elemento (píxel) de la imagen es un número de tipo **uint8**. Toda la matemática usada en redes neuronales es de tipo continuo y no discreto, y los mejores resultados de este tipo se obtienen con **float32** [9].

3.3. Visualización de los ejemplos

Se ha creado una función que recibe como entrada un número y visualiza en una gráfica la imagen correspondiente del conjunto de entrenamiento. Para ello se hace uso de la librería **Matplotlib**. En el Código 2 se encuentra el código de esta función.

Nótese que en la línea 4 se usa la función **.reshape(28,28)** ya que en el preprocesamiento de datos se ha convertido cada imagen de 28x28 en un vector de 784. En la Figura 2 se puede ver el uso de esta función y su respectiva salida.

```python
def visualize(sample):
  image = training_images[sample]
  fig = plt.figure
  plt.imshow(image.reshape(28,28), cmap='gray')
  plt.show()
```

Listing 2: Visualización de un dígito

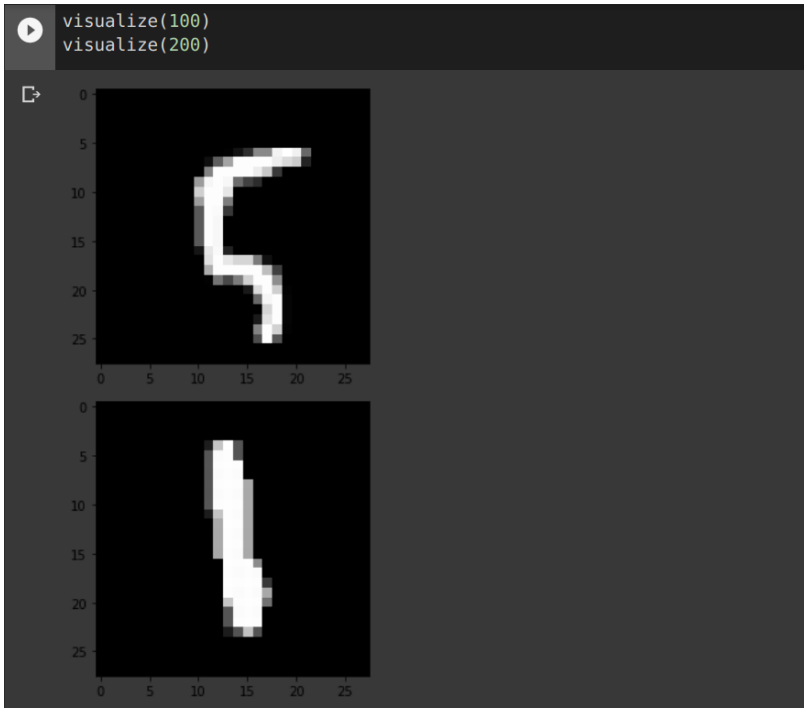

Figura 2: Ejemplo de uso de la función **visualize()**

4. Implementación

A lo largo de esta sección se van a presentar las distintas implementaciones de redes neuronales que se han realizado para el reconocimiento óptico de dígitos con sus respectivos análisis de resultados. Todas las implementaciones de cada versión están documentadas y subidas en mi **Github**:

`https://github.com/Jumacasni/IC/tree/main/MNIST`.

Cada versión tiene un archivo **.py** y un archivo **.ipynb**. Este último archivo es el *jupyer notebook* que se ha desarrollado en **Google Colab** y en el que, además del codigo, se muestran las salidas en consola de la ejecucion de cada versión (los logs de cada época, gráficas generadas, visualización de la red, etc.)

En cada versión propuesta se va aumentando el número de capas utilizadas hasta llegar a un esquema parecido al de la Figura 3. Este esquema es un buen diseño de la red neuronal que consigue muy buenos resultados, y es esto lo que se pretende en esta práctica.

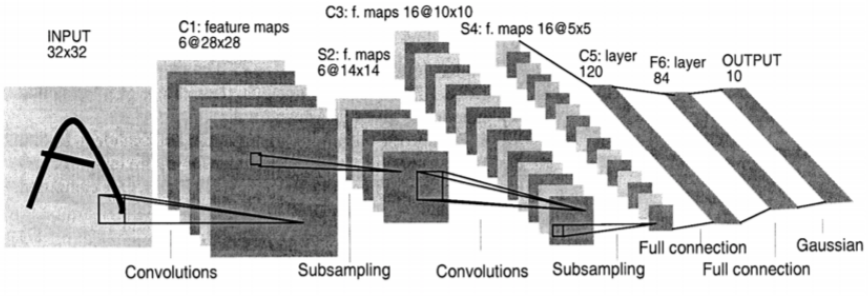

Figura 3: Esquema de capas seguido

4.1. Versión 1 - Perceptrón: 12.32 % de error

En esta primera versión, y con motivo de iniciarse en el mundo de las redes neuronales, se ha implementado de forma propia el algoritmo del perceptrón. Obviamente, a pesar de que este algoritmo no va a dar los mejores resultados y es por ello que se va a usar **Keras** en las próximas versiones para implementar redes neuronales más complejas y con mejores resultados, sí que me ha servido para entender el concepto de red neuronal y su funcionamiento.

He subido a mi respositorio de **Github** el código **.py** y **.ipynb**. El código **.ipynb** contiene también las salidas en consola de la ejecución:

- **.py:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version1_perceptron.py`
- **.ipynb:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version1_perceptron.ipynb`

El proceso de la implementación de este algoritmo es el siguiente:

- Se crea la clase **Neuron** que representa una neurona. En total habrá diez neuronas (una para cada dígito). Cada neurona tiene los siguientes atributos:
  - **weights:** un vector de pesos
  - **number:** el dígito de la neurona (un número del 0 al 9)
- La clase **Neuron** tiene las siguientes funciones:
  - **init_weights():** función que inicializa los pesos de la neurona a 0
  - **predict(image_number):** se calcula la suma de los pesos con la imagen de entrada. Devuelve **true** si la suma es mayor que 0.
  - **update_weights(training_image,update_politic):** actualiza los pesos:
    - Si **update_politic** es **True**, indica que la imagen **training_image** ha sido clasificada como correcta en la neurona pero en realidad no pertenece a esa clase, con lo cual se resta el vector de entrada (la imagen) del vector de pesos.
    - Si **update_politic** es **False**, indica que la imagen **training_image** ha sido clasificada como incorrecta en la neurona pero en realidad sí pertenece a esa clase, con lo cual se añade el vector de entrada (la imagen) al vector de pesos.
- Se crean las diez neuronas (dígitos del 0 al 9) y se entrena cada neurona con el conjunto de entrenamiento, actualizando los pesos dependiendo de si se clasifica correctamente o no.
- Una vez finalizado el entrenamiento, se pasa el conjunto de prueba para comprobar el número de errores que se comete y ver cómo de bien se clasifican los dígitos. Adicionalmente, se ha calculado el error para cada clase concreta.

4.1.1. Resultados

Los resultados obtenidos usando el algoritmo del perceptrón se muestran en la Figura 4. El porcentaje de error total es de **12.32 %**. Se puede observar que los dígitos en los que más errores se cometen son el 3, 8 y 9, respectivamente.

El tiempo que ha tardado en ejecutarse el entrenamiento ha sido de **15.38 segundos**

RESULTADOS VERSIÓN 1: PERCEPTRÓN	
Clase	**Porcentaje de error (%)**
0	0.19
1	0.42
2	0.15
3	4.12
4	0.43
5	0.15
6	1.5
7	0.3
8	2.85
9	2.21
Total	**12.32**

Figura 4: Porcentajes de error usando la versión 1

4.2. Versión 2: 7.46 % de error sobre el conjunto de prueba

A partir de esta versión ya se ha empezado a utilizar la librería **Keras** para crear una red neuronal más compleja que la de un simple perceptrón. Como se trata de ajustar los hiperparámetros de la red para ir consiguiendo mejores resultados, a lo largo de estas versiones se va a trabajar con 3 conjuntos:

1. **Conjunto de entrenamiento:** formado por 48.000 ejemplos (el 80 % del conjunto de entrenamiento de **MNIST**). Se va a usar para entrenar la red neuronal.
2. **Conjunto de validación:** formado por 12.000 ejemplos (el 20 % del conjunto de entrenamiento de **MNIST**). Se va a usar para validar el entrenamiento.
3. **Conjunto de prueba:** formado por los 10.000 ejemplos del conjunto de prueba de **MNIST**. Se usa para testear la calidad de la red neuronal.

La idea es que se va a entrenar la red con el **conjunto de entrenamiento** y luego se van ajustar los hiperparámetros de la red de acuerdo a los resultados obtenidos sobre el **conjunto de validación**. El **conjunto de prueba** sirve para obtener una estimación no sesgada de lo buena que es la red.

Siguiendo las orientaciones del guión de la práctica, en esta versión se ha creado una red neuronal simple que tiene una capa de entrada y una capa de salida con una función de activación **softmax**. He subido a mi respositorio de **Github** el código **.py** y **.ipynb** de esta versión. El código **.ipynb** contiene también las salidas en consola de la ejecución:

- **.py:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version2.py`
- **.ipynb:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version2.ipynb`

La función de activación **softmax** es muy habitual usarla en la última capa en problemas de clasificación debido a que devuelve una probabilidad que, entre todas las neuronas en dicha capa, suman 1. En el problema del reconocimiento de dígitos, usando **softmax** en la capa de salida obtenemos la probabilidad de que el dígito pertenezca a esa clase [10].

Se ha jugado con los parámetros **batch_size** y **epochs** [11]:

- **batch_size:** se refiere al tamaño del mini-lote que se envía a la red neuronal. Esto indica la frecuencia con la que se ajustan los pesos. Se conoce como una forma de **gradiente descendente estocaśtico**.
- **epochs:** una época se produce cuando el conjunto de entrenamiento al completo pasa la red neuronal. Una época produce un subajuste, y muchas épocas producen un sobreajuste.

En la Figura 5 se puede ver lo que pasa al aumentar el número de épocas y mini-lotes. Con 20 épocas y mini-lotes del tamaño de 128, se obtiene un 7.31 % de error sobre el conjunto de prueba y un 6.99 % de error sobre el conjunto de entrenamiento. Si incrementamos el número de épocas a 100, obtenemos un 7.25 % de error sobre el conjunto de prueba y un 6.31 % sobre el conjunto de entrenamiento. Esto se traduce en un **sobreajuste**, pues estamos consiguiendo mejores resultados en el conjunto de entrenamiento pero aumenta el error en el conjunto de prueba.

VERSIÓN 2				
Epochs	Batch_size	% error conjunto de prueba	% error conjunto de entrenamiento	Tiempo empleado (seg)
20	128	7.31	6.99	13.29
20	64	7.13	6.92	19.5
20	32	7.19	6.94	33.7
100	128	7.25	6.31	66.4
100	64	7.57	6.36	99.53
100	32	7.7	6.56	168.68

Figura 5: Comparación de resultados cambiando **epochs** y **batch_size**

Para evitar esto, se ha utilizado la técnica de regularización **Early Stopping**. Se utiliza para parar el entrenamiento cuando la red ha parado de mejorar de acuerdo a una medida. En este caso, interesa parar el entrenamiento cuando la función de error sobre el conjunto de validacion deja de mejorar.

4.2.1. Resultados

Red neuronal creada:

- Una capa de salida con función de activación **softmax**
- **Algoritmo de optimización: RMSprop**

- **Función de error:** cross entropy
- **Tamaño de mini-lote:** 64
- **Épocas:** 100
- **Early Stopping** cuando **val_loss** deja de mejorar

La estructura de esta red neuronal se muestra en la Figura 6. Está compuesta por la capa de entrada y una capa de salida de 10 neuronas con una función de activación **softmax**. Como optimizador se ha usado **RMSprop**.

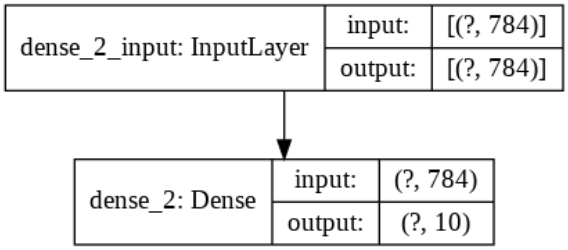

Figura 6: Estructura de la red neuronal de la versión 2

Se obtiene un **7.41 % de error sobre el conjunto de entrenamiento** y un **7.46 % sobre el conjunto de prueba** y ha tardado un total de **8.84 segundos**. El **Early Stopping** se ha producido en la **época 8**.

En las Figuras 7 y 8 se puede ver gráficamente la función de pérdida y la precisión a lo largo de las épocas.

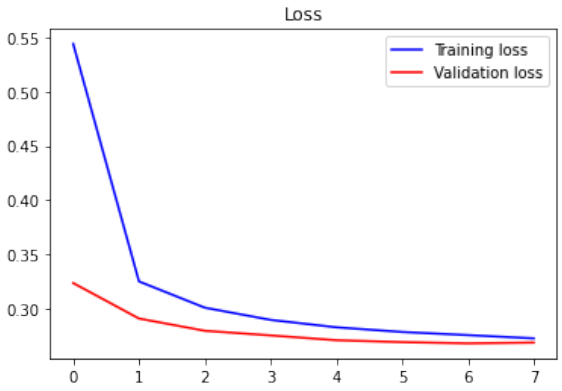

Figura 7: Función de pérdida versión 2

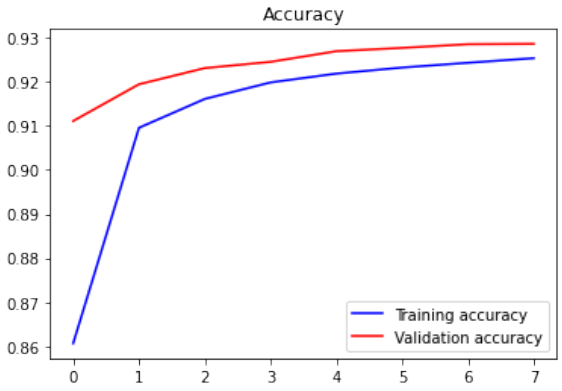

Figura 8: Precisión versión 2

4.3. Versión 3: 1.9 % de error sobre el conjunto de prueba

En esta versión se ha añadido una capa oculta con una función de activación **ReLU**. En la Figura 9 se puede ver una comparación cuando se modifica el número de neuronas en la capa oculta y poniendo **loss** o **val_loss** en el criterio de parada del Early Stopping.

VERSIÓN 3						
Número de neuronas	Early Stopping	Épocas realizadas (max 30)	Función de activación	% error conjunto de prueba	% error conjunto de entrenamiento	Tiempo empleado (seg)
512	loss	23	relu	1.96	0.01	122.27
512	val_loss	5	relu	2.1	1.01	26.72
256	loss	23	relu	1.9	0.05	72.08
256	val_loss	5	relu	2.24	1.07	20.12
128	loss	30	relu	2.32	0.02	66.07
128	val_loss	8	relu	2.35	1.18	17.54

Figura 9: Comparación de los resultados versión 3

La que mejor resultado da es la red neuronal que tiene una capa oculta con 256 neuronas y utilizando **loss** como criterio de parada.

4.3.1. Resultados

Red neuronal creada:

- Una capa oculta de 256 neuronas con función de activación **ReLU**
- Una capa de salida de 10 neuronas con función de activación **softmax**
- **Algoritmo de optimización: RMSprop**
- **Función de error:** cross entropy
- **Tamaño de mini-lote:** 64
- **Épocas:** 30
- **Early Stopping** cuando **loss** deja de mejorar

La estructura de esta red neuronal se muestra en la Figura 10. Está compuesta por la capa de entrada, una capa oculta de 256 neuronas con función de activación **ReLU** y una capa de salida de 10 neuronas con una función de activación **softmax**. Como optimizador se ha usado **RMSprop**.

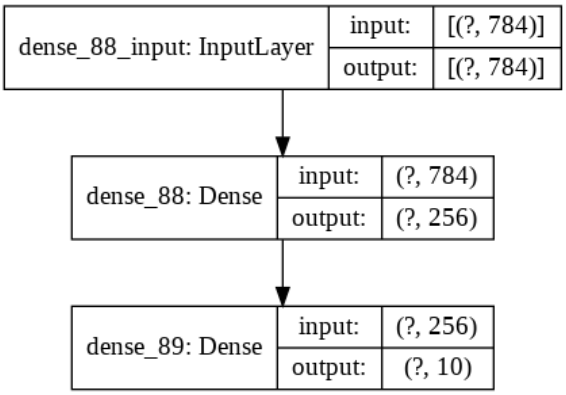

Figura 10: Estructura de la red neuronal de la versión 3

Se obtiene un **0.05 % de error sobre el conjunto de entrenamiento** y un **1.9 % sobre el conjunto de prueba** y ha tardado un total de **72.08 segundos**. El **Early Stopping** se ha producido en la **época 22**.

He subido a mi respositorio de **Github** el código **.py** y **.ipynb** de esta versión. El código **.ipynb** contiene también las salidas en consola de la ejecución:

- **.py:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version3.py`
- **.ipynb:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version3.ipynb`

En las Figuras 11 y 12 se puede ver gráficamente la función de pérdida y la precisión a lo largo de las épocas.

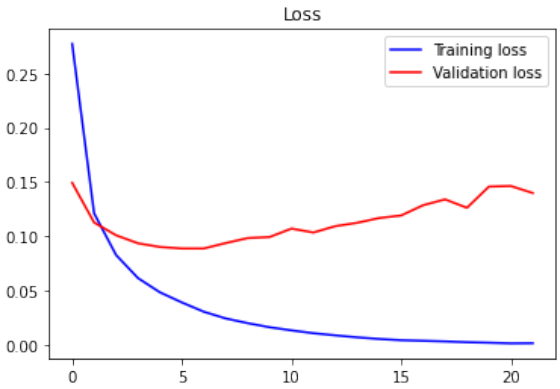

Figura 11: Función de pérdida versión 3

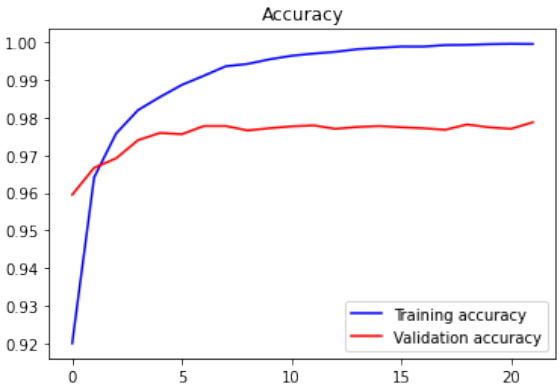

Figura 12: Precisión versión 3

4.4. Versión 4: 1.26 % de error sobre el conjunto de prueba

En esta versión se han añadido capas convolucionales. Cuando se hizo el preprocesamiento de datos en la Sección 3.2, las imágenes pasan de ser 28x28 a ser 784x1. Para poder añadir capas convolucionales, en **Keras** se requiere una tercera dimensión. La estructura es **(Número_imágenes, Ancho_imagen, Alto_imagen, Dimensión_color)**.

En concreto, se han añadido **dos capas convolucionales** y una capa adicional **Flatten** que convierte su entrada en un vector para las capas **Dense**, que son las capas que están totalmente conectadas.

El algoritmo de optimización se ha cambiado a **adam** porque **RMSprop** estaba dando subidas muy bruscas e irregulares en **validation_loss** tal y como muestra la Figura 13.

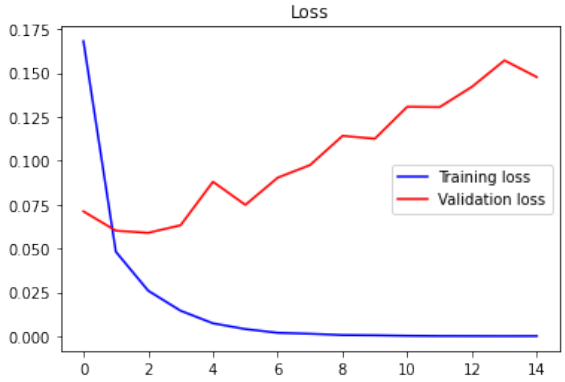

Figura 13: **Validation_loss** utilizando **RMSprop**

4.4.1. Resultados

Red neuronal creada:

- Una capa convolucional con 64 filtros y un kernel de 3x3, función de activación **ReLU**
- Una capa convolucional con 32 filtros y un kernel de 3x3, función de activación **ReLU**
- Una capa flatten
- Una capa oculta de 256 neuronas con función de activación **ReLU**

- Una capa de salida de 10 neuronas con función de activación **softmax**
- **Algoritmo de optimización: Adam**
- **Función de error:** cross entropy
- **Tamaño de mini-lote:** 64
- **Épocas:** 30
- **Early Stopping** cuando **loss** deja de mejorar

La estructura de esta red neuronal se muestra en la Figura 14.

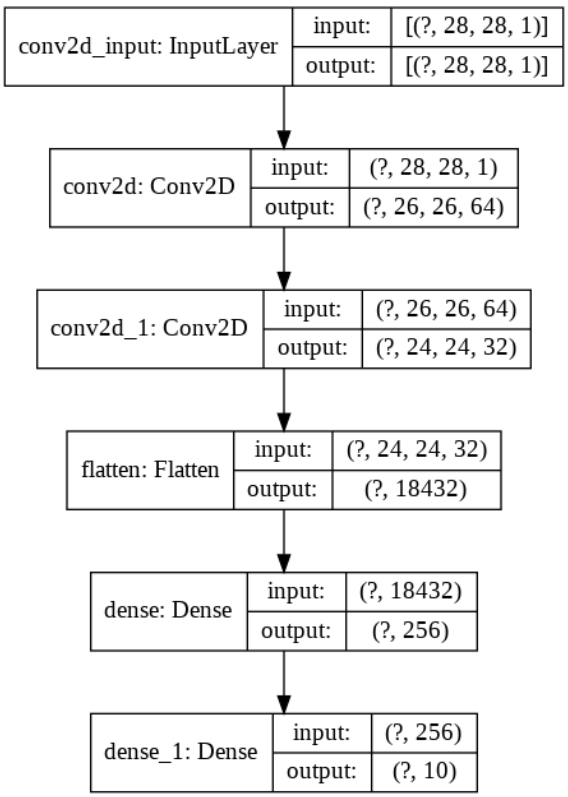

Figura 14: Estructura de la red neuronal de la versión 4

Se obtiene un **0.18 % de error sobre el conjunto de entrenamiento** y un **1.26 % sobre el conjunto de prueba** y ha tardado un total de **1313.58 segundos**. El **Early Stopping** se ha producido en la **época 8**.

He subido a mi respositorio de **Github** el código **.py** y **.ipynb** de esta versión. El código **.ipynb** contiene también las salidas en consola de la ejecución:

- **.py:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version4.py`
- **.ipynb:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version4.ipynb`

En las Figuras 15 y 16 se puede ver gráficamente la función de pérdida y la precisión a lo largo de las épocas.

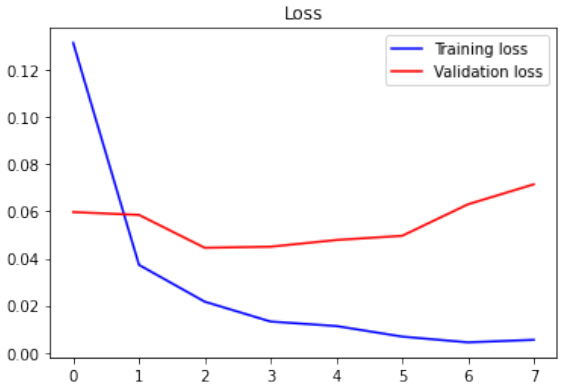

Figura 15: Función de pérdida versión 4

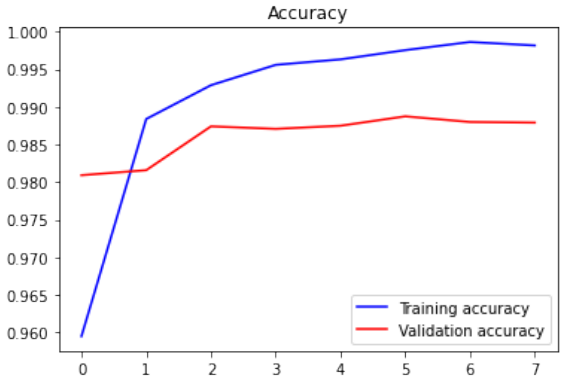

Figura 16: Precisión versión 4

4.5. Versión 5: 0.81 % de error sobre el conjunto de prueba

Se ha aumentado (disminuido) el conjunto de entrenamiento (de validación) en 2.000 ejemplos. El conjunto de entrenamiento ahora tiene 50.000 ejemplos y el de validación tiene 10.000 ejemplos.

En esta versión se ha añadido una capa convolutiva **pooling** después de las dos primeras capas convolutivas para reducir la dimensionalidad. A continuación, se han añadido dos nuevas capas convolutivas y una capa convolutiva **pooling**.

4.5.1. Resultados

Red neuronal creada:

- Una capa convolucional con 32 filtros y un kernel de 3x3, función de activación **ReLU**
- Una capa convolucional con 32 filtros y un kernel de 1x1, función de activación **ReLU**
- Una capa convolucion pooling de 2x2
- Una capa convolucional con 64 filtros y un kernel de 3x3, función de activación **ReLU**
- Una capa convolucional con 64 filtros y un kernel de 1x1, función de activación **ReLU**
- Una capa convolucion pooling de 2x2
- Una capa flatten
- Una capa oculta de 256 neuronas con función de activación **ReLU**
- Una capa de salida de 10 neuronas con función de activación **softmax**
- **Algoritmo de optimización: Adam**
- **Función de error:** cross entropy
- **Tamaño de mini-lote:** 64
- **Épocas:** 30
- **Early Stopping** cuando **loss** deja de mejorar

La estructura de esta red neuronal se muestra en la Figura 17.

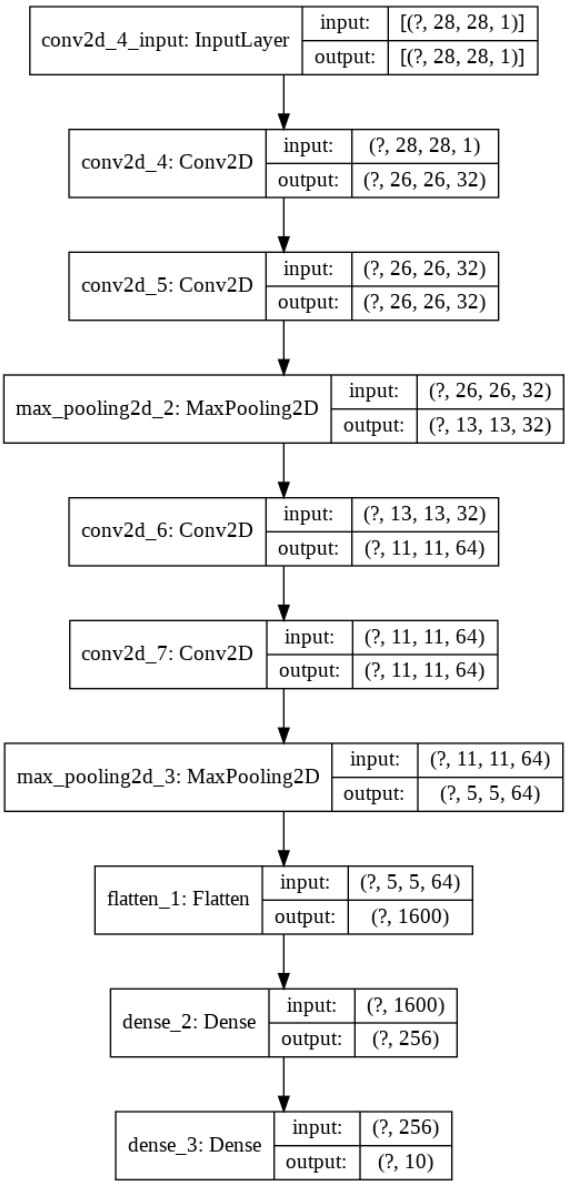

Figura 17: Estructura de la red neuronal de la versión 5

Se obtiene un **0.29 % de error sobre el conjunto de entrenamiento** y un **0.81 % sobre el conjunto de prueba** y ha tardado un total de **599.75 segundos**. El **Early Stopping** se ha producido en la **época 10**. El tiempo de entrenamiento ha bajado con respecto a la versión 4 y el porcentaje de error ha bajado del 1 %.

He subido a mi respositorio de **Github** el código **.py** y **.ipynb** de esta versión. El código **.ipynb** contiene también las salidas en consola de la ejecución:

- **.py:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version5.py`
- **.ipynb:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version5.ipynb`

En las Figuras 18 y 19 se puede ver gráficamente la función de pérdida y la precisión a lo largo de las épocas.

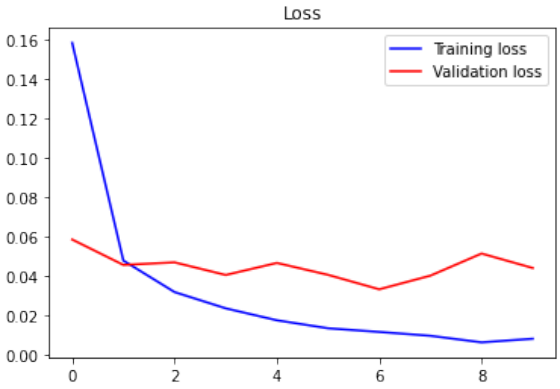

Figura 18: Función de pérdida versión 5

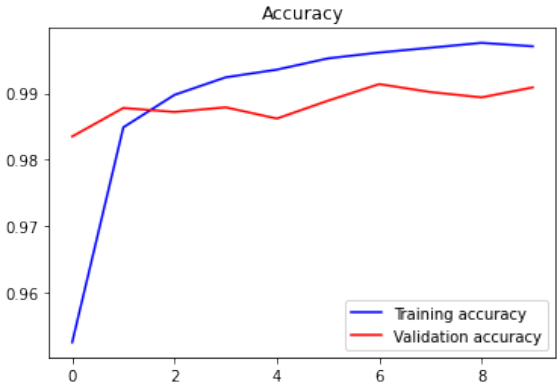

Figura 19: Precisión versión 5

4.6. Versión 6: 0.47 % de error sobre el conjunto de prueba

En esta última versión se ha añadido una capa totalmente conectada de 128 neuronas justo antes de la capa de salida. También se ha aplicado la técnica de regularización **Dropout** añadiendo dos capas que reducen a la mitad el número de neuronas. Esto ayuda a reducir el sobreaprendizaje y obtener mejores resultados.

También se ha eliminado el **Early Stopping** ya que producía paradas muy tempranas (solo hacía entre 10-15 épocas aproximadamente). He decidido probar diferentes épocas (25, 35 y 50) y he ido obteniendo mejores resultados. Con 25 épocas, se obtenía un error del **0.49 % sobre el conjunto de prueba**. Con 35 épocas, se obtenía un error del **0.47 % sobre el conjunto de prueba**. Con 50 épocas, se obtenía un error del **0.56 % sobre el conjunto de prueba.**

Se ha probado a disminuir e incrementar el tamaño de los mini-lotes y también a cambiar el porcentaje de neuronas eliminadas en las capas Dropout, pero el porcentaje de error sobre el conjunto de prueba empeoraba alrededor de un 0.1 %.

4.6.1. Resultados

Red neuronal creada:

- Una capa convolucional con 32 filtros y un kernel de 3x3, función de activación **ReLU**
- Una capa convolucional con 32 filtros y un kernel de 1x1, función de activación **ReLU**
- Una capa convolucion pooling de 2x2
- Una capa convolucional con 64 filtros y un kernel de 3x3, función de activación **ReLU**
- Una capa convolucional con 64 filtros y un kernel de 1x1, función de activación **ReLU**
- Una capa convolucion pooling de 2x2
- Una capa flatten
- Una capa dropout que elimina el 50 % de neuronas
- Una capa oculta de 256 neuronas con función de activación **ReLU**
- Una capa dropout que elimina el 50 % de neuronas
- Una capa oculta de 128 neuronas con función de activación **ReLU**
- Una capa de salida de 10 neuronas con función de activación **softmax**

- **Algoritmo de optimización: Adam**
- **Función de error:** cross entropy
- **Tamaño de mini-lote:** 64
- **Épocas:** 35

La estructura de esta red neuronal se muestra en la Figura 20.

Se obtiene un **0.61 % de error sobre el conjunto de entrenamiento** y un **0.47 % sobre el conjunto de prueba**. El error obtenido sobre el conjunto de prueba es mejor que el del conjunto de entrenamiento. Lo malo es que el tiempo de ejecución ha pasado a ser de **2151.15 segundos**.

He subido a mi respositorio de **Github** el código **.py** y **.ipynb** de esta versión. El código **.ipynb** contiene también las salidas en consola de la ejecución:

- **.py:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version6.py`
- **.ipynb:** `https://github.com/Jumacasni/IC/blob/main/MNIST/version6.ipynb`

En las Figuras 21 y 22 se puede ver gráficamente la función de pérdida y la precisión a lo largo de las épocas.

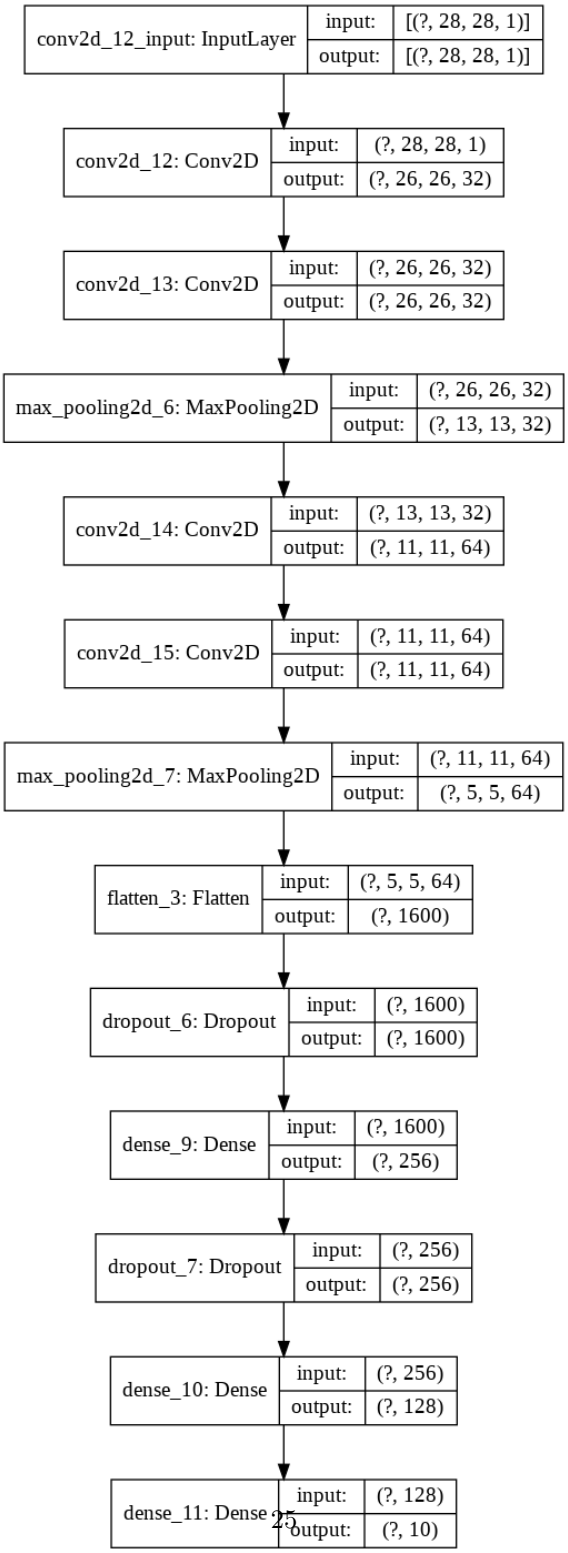

Figura 20: Estructura de la red neuronal de la versión 6

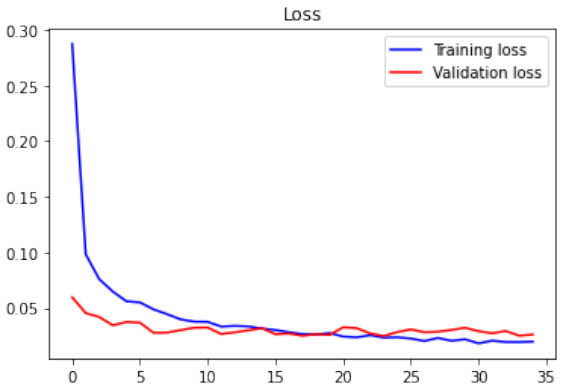

Figura 21: Función de pérdida versión 6

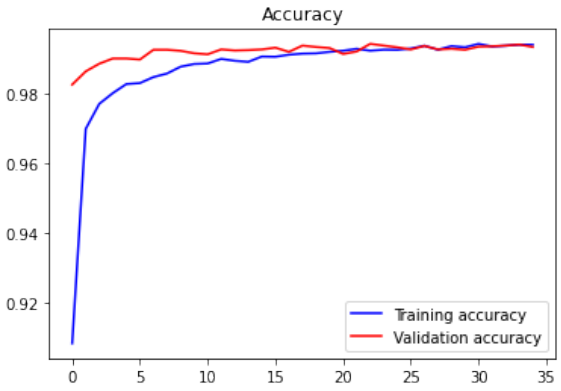

Figura 22: Precisión versión 6

Referencias

[1] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[2] Keras. https://keras.io/.

[3] Tensorflow. https://www.tensorflow.org/.

[4] Ambika Choudhury. Tensorflow vs keras: Which one should you choose. https://analyticsindiamag.com/tensorflow-vs-keras-which-one-should-you-choose/, 2019.

[5] Tensorflow guide: Keras. https://www.tensorflow.org/guide/keras?hl=es.

[6] Google colab. https://colab.research.google.com/notebooks/intro.ipynb#recent=true.

[7] Jupyter. https://jupyter.org/.

[8] Google colab faqs. https://research.google.com/colaboratory/faq.html.

[9] StackOverflow. Why do i have to convert uint8 into float32? https://stackoverflow.com/questions/59986353/why-do-i-have-to-convert-uint8-into-float32, 2020.

[10] Chengwei. How to choose last-layer activation and loss function. https://www.dlology.com/blog/how-to-choose-last-layer-activation-and-loss-function/, 2017.

[11] Sagar Sharma. Epoch vs batch size vs iterations. https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9, 2017.