Práctica de redes neuronales

Reconocimiento óptico de caracteres MNIST

Irene Béjar Maldonado

8/12/2019

Índice

1. Introducción

En esta memoria se describirán los pasos seguidos para la construcción de una red neuronal para resolver el problema del reconocimiento de caracteres manuscritos de la base de datos MNIST (“MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges” n.d.).

1.1. Problema

Como ya se ha dicho el problema a resolver es el reconocimiento de caracteres manuscritos en la base de datos MNIST. Esta base de datos proporciona imágenes de números manuscritos todos ellos etiquetados. Contiene dos *datasets:* uno de entrenamiento con 60000 imágenes y otro de test con 10000. A continuación se muestra una pequeña muestra de algunas imágenes (Figura 1).

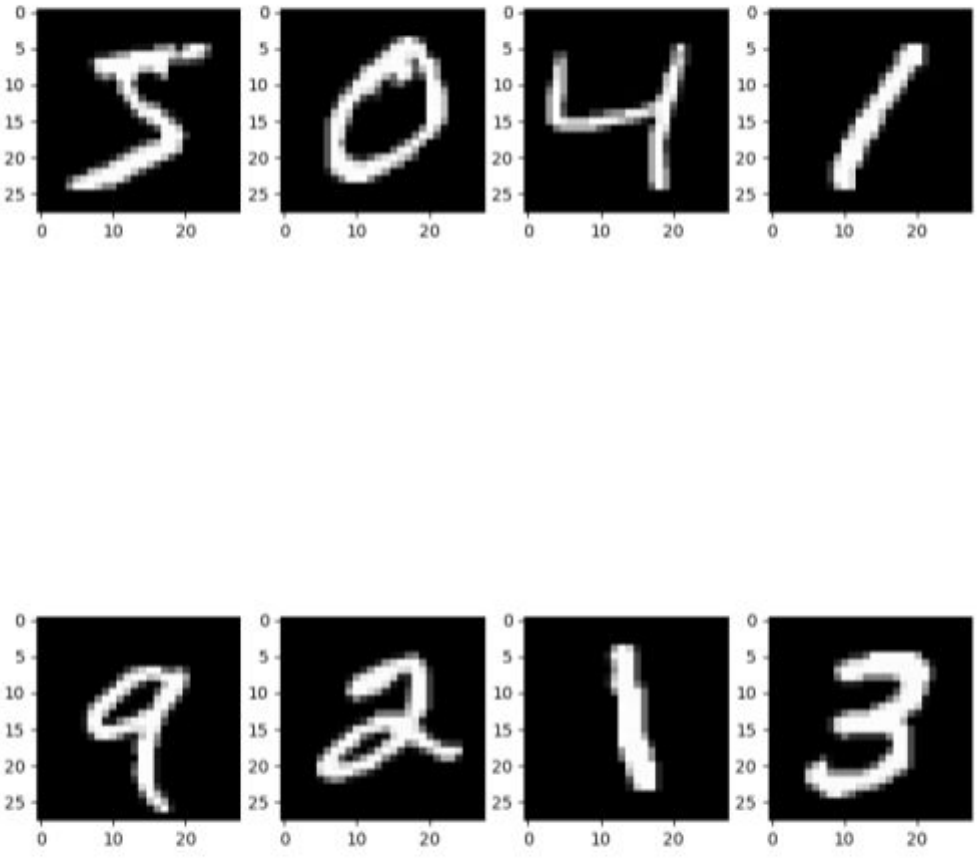


Figura 1: Muestra de las imágenes de MNIST

Todas las imágenes de la base de datos tienen un tamaño de 28x28 píxeles, están centradas y etiquetadas. Imágenes y etiquetas están separadas en dos vectores distintos.

El objetivo es clasificar correctamente las imágenes, es decir, que coincida la etiqueta de la imagen y la predicción hecha por la red neuronal. A lo largo de la memoria se explicarán los distintos enfoques seguidos para lograr esta tarea.

1.2. Entorno de trabajo

El desarrollo, entrenamiento y validación de todas las redes neuronales se ha realizado en mi ordenador personal, el cual tiene las siguientes especificaciones:

Procesador	Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8
Memoria RAM	1x DRR3 8GB
Gráficos	GeForce GTX 1050/PCIe/SSE2 con 2GB dedicados

Para la implementación de las redes se han usado los siguientes lenguajes y bibliotecas:

- ***Python*** como lenguaje de programación.
- ***Keras*** como API para la implementación de redes neuronales multicapa.
- ***TensorFlow***, en su versión para GPU, como backend para la API de Keras.
- ***Scikit-Learn***, biblioteca de *machine learning*, para la automatización de algunos procesos. (“Scikit-Learn Documentation” n.d.)

2. Explicación de las capas de *Keras* usadas

Con el objetivo de explicarme mejor a la hora de explicar los modelos usados y aclarar la funcionalidad de cada capa se expondrán brevemente las capas usadas durante el desarrollo y los parámetros más importantes a utilizar (“About Keras Layers - Keras Documentation” n.d.)

Pero primero unas aclaraciones (“Cheat Sheet: Keras & Deep Learning Layers — Brendan Herger” n.d.)

> ***¿Qué es una capa?***
> Es una unidad atómica, con una arquitectura de aprendizaje profundo. Las redes se construyen añadiendo capas sucesivamente.
>
> ***¿Qué propiedades tienen?***
> Todas las capas tienen:
> - Pesos que crean una combinación linear con las salidas de la capa anterior.
> - Una función de activación, que puede ser no lineal.
> - Un nodo *bias*, que es el equivalente de a tener una variable de entrada con una valor de 1 constante.
>
> ***Versiones de las capas***
> Algunas capas tienen distintas versiones como 1D y 2D, por lo general, las primeras se usan para series temporales y texto, las segundas para imágenes (nuestro caso).

2.1. Dense

Es una capa completamente conectada, es decir, las salidas de la capa anterior están conectadas a todas las neuronas de esta capa (Figura 2):

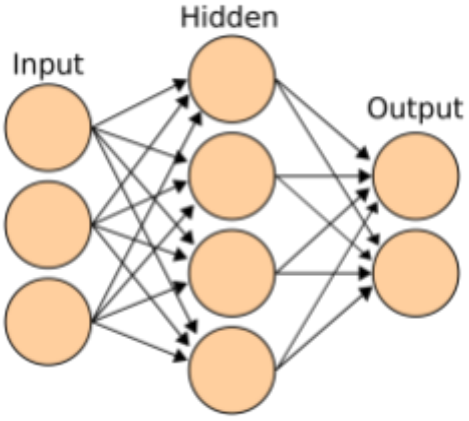


Figura 2: Ejemplo de capas completamente conectadas[1]

[1] Fuente: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Artificial_neural_network.svg/1920px-Artificial_neural_network.svg.png

Los parámetros que tenemos que tener en cuenta son:

- **units**: Número de neuronas que queremos que tenga la capa.
- **activation**: Función de activación
- **Input shape:** Las dimensiones que tiene nuestra entrada.

Generalmente se usa para la clasificación.

2.2. Dropout

El *dropout* consiste en establecer aleatoriamente una parte de las unidades de entrada a 0 en cada actualización durante el tiempo de entrenamiento, lo que ayuda a evitar el sobreajuste.

Los parámetros a tener en cuenta son:

- **Rate**: float entre 0 and 1. Ratio de neuronas que vas a sufrir el *dropout*.

2.3. Flatten

Aplana la entrada, es decir, la coloca en un único vector. No afecta el tamaño del lote.

Figura 3: Ejemplo de cómo se aplana una matriz unidimensional[2]

[2] https://stackoverflow.com/questions/43237124/what-is-the-role-of-flatten-in-keras

2.4. Conv2D

Crea convoluciones 2D, por ejemplo, sobre imágenes. Las convoluciones es un algoritmo de *deep learning* que puede incorporar una imagen de entrada, asignar importancia (pesos y sesgos aprendibles) a varios aspectos/objetos en la imagen y poder diferenciar uno de otro. Básicamente, lo que hace es aplicar un filtro a la imagen para extraer características. (Saha 2018).

Los parámetros a tener en cuenta son (Rosebrock 2018)

- **filters:** El número de filtros que queremos que aprende la capa. Una buena estrategia para saber qué número de filtros colocar es tener en cuenta que las primeras capas (cercanas a la entrada) aprenden menos filtros que las capas más profundas en la red (cercanas a la salida). Sabiendo esto, se recomienda empezar con filtros en el rango de [32, 64, 128] en las primeras capas e ir aumentándolo en potencia de 2 para las capas más profundas.
- **kernel_size:** Una tupla de dos números que especifican la altura y la anchura de la matriz de convolución. Los valores típicos son del (1,1) hasta el (7,7). Para saber qué tamaño de filtro usar se recomienda empezar por filtros de (5,5) y (7,7) para imágenes superiores a 128x128 píxeles, e ir reduciendo el tamaño. Si son más pequeñas empezar por uno de (3,3).
- **strides:** Una tupla de dos números indicando el “salto” que se da a lo largo del eje x y el eje y al aplicar la convolución. Este parámetro sólo es importante si queremos sustituir las capas de MaxPooling (sección 2.5), para ello se aumenta el salto de (1,1), que es el salto por defecto, a (2,2).
- **padding**: Le pone un marco a la imagen, así se reduce menos. Por defecto es “valid”, sin marco. Para tener el marco hay que ponerlo a “same”.
- **activation:** Función de activación.
- **kernel_reguralirizer:** Regularización para la capa, importante si se están detectando signos de overfitting. (“Regularizers - Keras Documentation” n.d.)
- **kernel_initializer:** Controla el método que se usa para inicializar los valores de la clase Conv2D antes de que empiece el entrenamiento. Por defecto de usa “glorot_uniform”. Para redes convolutivas también se recomienda usar el inicializador “he_normal”.

2.5. MaxPooling2D / AveragePooling2D

Las capas de *Polling* son las responsables de reducir el tamaño de la salida de las capas convolutivas. Esto se hace para disminuir la potencia necesaria para procesar los datos. El método que se usa es mediante la extracción de características dominantes, escogiendo una porción de imagen y quedándose con los valores que más nos interesen. (Saha 2018)

Hay dos formas de conseguir extraer estas características:

- MaxPooling, consiste en quedarse con el máximo valor de la porción de imagen que estamos mirando.
- AveragePooling, nos quedamos con la media de todo los valores.

El primer método también ayuda a eliminar el ruido en la imagen, el segundo solo reduce el tamaño de la entrada.
Los parámetros que tenemos que tener en cuenta son los siguientes:

- **pool_size:** Una tupla de dos números que especifican la altura y la anchura de la matriz que se usará para reducir la entrada.
- **strides:** Una tupla de dos números indicando el “salto” que se da a lo largo del eje x y el eje y. Si no se especifica por defecto será el **pool_size**.
- **padding**: Le pone un marco a la imagen, así se reduce menos. Por defecto es “valid”, sin marco. Para tener el marco hay que ponerlo a “same”. Cuando estamos intentado reducir el tamaño, no se suele usar marco.

2.6. BatchNormalization

Es una capa se se usa para normalizar las salidas de la capa anterior. Con estos ganamos algunos beneficios: (“One Simple Trick to Train Keras Model Faster with Batch Normalization | DLology” n.d.) (“Normalization Layers - Keras Documentation” n.d.)

- Las redes entrenan más rápido y convergen más fácilmente.
- Permite usar tasas de aprendizaje más altas.
- Hace que los pesos sean más fáciles de inicializar.

En general, usar esta capa hace que la red tenga un mejor rendimiento en general.

3. Implementaciones

En esta sección se describirán las distintas implementaciones que se han hecho para la resolución del problema descrito. Se irá explicando desde la propuesta más simple hasta la más compleja, exponiendo la estructura de la red, las mejoras hechas, los problemas encontrados y los datos sobre el rendimiento del modelo.

3.1. Ejemplos de *Keras,* modelo simple de Red Neuronal Profunda

En una primera aproximación hacia la API de *Keras* encontré este ejemplo que ellos mismos proponen para explicar la herramienta. La estructura es bastante simple, consta solamente de 5 capas, sin ninguna capa convolutiva. (“Mnist Mlp - Keras Documentation” n.d.)

3.1.1. Preprocesamiento de los datos

Es simple, se dejan 60000 imágenes como conjunto de entrenamiento y 10000 imágenes como conjunto de validación y conjunto de test. Después se han realizado las siguientes transformaciones:

- Las imágenes se han colocado en un vector de dimensión 784.
- Las etiquetas se han pasado a un matriz categórica (un las clases se representan con vectores binarios con un 1 en la posición que indica la clase)
- Se han normaliza los valores de las imágenes.

3.1.2. Modelo

El modelo consta de 5 capas, que se muestran a continuación (Figura 4):

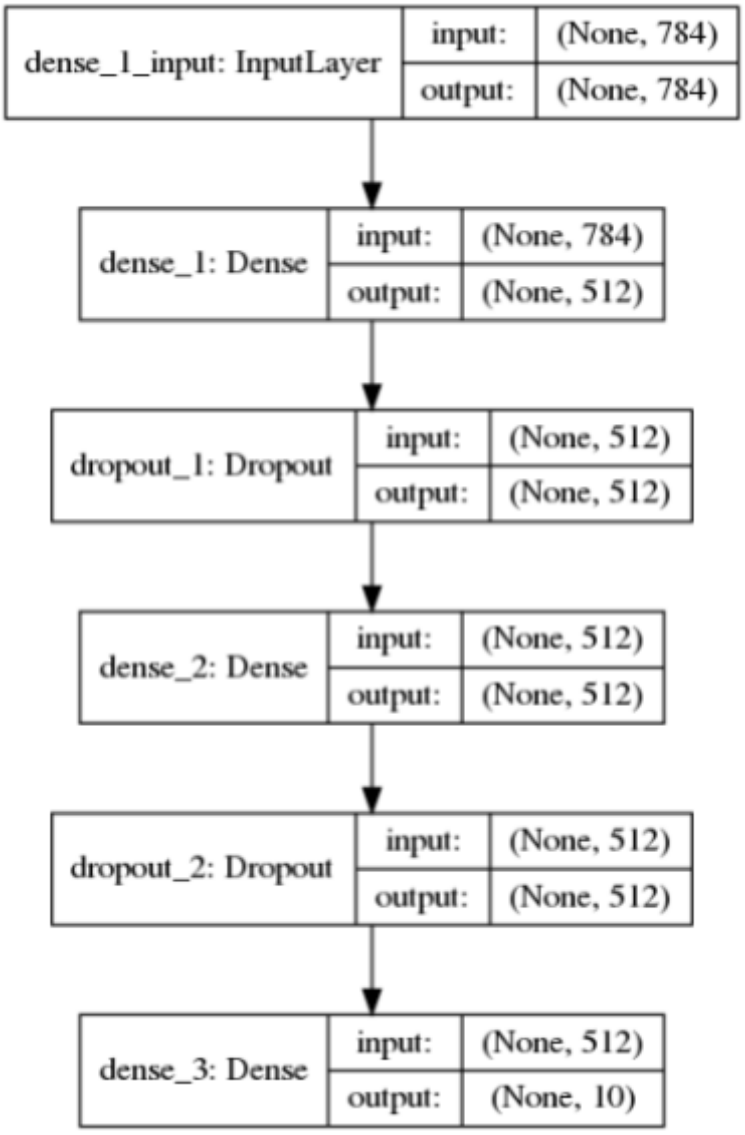


Figura 4: Esquema de la red

Se puede observar que la red recibe 784 entradas, que se corresponde a cada pixel de la imagen. La primera capa *Dense* tiene 512 neuronas que reciben la imagen y reciben un *Dropout* del 20%, es decir, el 20% de las neuronas se van a apagar. A continuación tenemos una segunda capa *Dense* con las mismas características. Ambas capas usan como función de activación *ReLU*. Por último, la capa de salida que clasifica las 10 clases, que usa como función de activación *SoftMax*.

Para entrenar la red se ha usado el mismo conjunto para validación y para prueba. Como **función de coste** se ha usado ***Categorical Crossentropy*** y como optimizador *RMSprop*. Se ha usado el entrenamiento por lotes con un tamaño de 128 imágenes por lote y se ha entrenado durante 20 épocas.

3.1.3. Resultados

Los resultados obtenidos con esta red son los siguientes:

Tiempo de entrenamiento (segundos):	33.19
Segundos por época:	2
% Acierto entrenamiento	99.884
% Fallo entrenamiento	0.382
% Acierto Prueba	98.199
% Fallo Prueba	13.876

Tabla 1: Resultados de la red

El **porcentaje de fallo** lo estamos midiendo **en base** a nuestra **función de coste**. De estos resultados podemos observar que la red se ajusta demasiado bien al conjunto de entrenamiento, ya que, el porcentaje de fallo es muchísimo más bajo que en el conjunto de prueba y el tiene también más tasa de acierto. Por lo tanto, podemos concluir que este modelo sufre sobreaprendizaje.

Es importe señalar que pese a ser una red tan simple tiene un buen porcentaje de acierto, además. de todas las redes, es la más rápida. Por último, esta red tiene aún mucho margen de mejora, tal y como indica el tutorial de Keras.

3.2. Ejemplo de *Keras*, modelo sencillo de red neuronal convolutiva

Este ejemplo no es mucho más complejo que el anterior, solo cambiamos la primera capa *Dense* por dos capas convolutiva, el resto es muy similar. (“Mnist Cnn - Keras Documentation” n.d.)

3.2.1. Preprocesamiento de los datos

Al igual que en el anterior se dejan 60000 imágenes como conjunto de entrenamiento y 10000 imágenes como conjunto de validación y conjunto de test. Después se han realizado las siguientes transformaciones:

- Las etiquetas se han pasado a un matriz categórica (un las clases se representan con vectores binarios con un 1 en la posición que indica la clase)
- Se han normaliza los valores de las imágenes.

3.2.2. Modelo

La estructura del modelo se muestra a continuación (Figura):

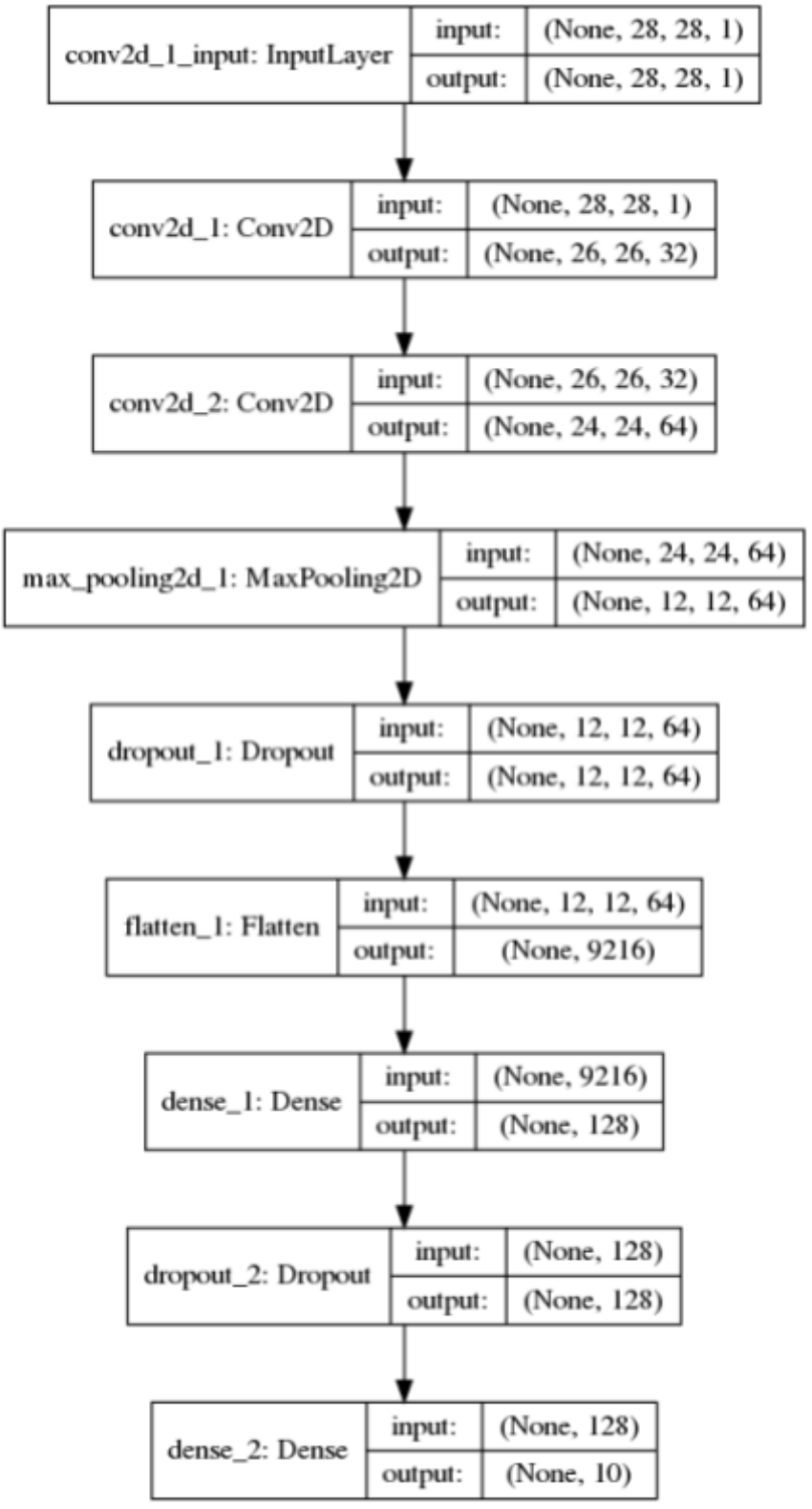


Figura 5: Esquema de la red

La red recibe como entrada una matriz de 28x28, es decir, una imagen del conjunto. Después esta imagen pasa por dos capas convolutivas, la primera con 32 filtros y una matriz de 3x3, la segunda con 64 filtros y una matriz también de 3x3. Tras esto nuestra imagen queda reducida a una matriz de 24x24. A continuación tenemos una capa de *MaxPooling* con una matriz 3x3, que nos reduce la salida de la capa anterior a la mitad, 12x12 y añadimos *Dropout* del 25% al conectarla con la siguiente capa. Con la capa *Flatten,* aplanamos la entrada para la siguiente capa *Dense* que recibe 9216 entradas y tiene 128 neuronas, también se conecta con *Dropout* del 50% a la siguiente capa. Por último, tenemos la capa de salida que clasifica.

Todas las capas convolutivas y la capa *Dense* de 128 tiene una activación *ReLU,* la capa de salida tiene una activación *SoftMax*.

Para entrenar la red se ha usado el mismo conjunto para validación y para prueba. Como **función de coste** se ha usado ***Categorical Crossentropy*** y como **optimizador *Adadelta***. Se ha usado el entrenamiento por lotes con un tamaño de 128 imágenes por lote y se ha entrenado durante 5 épocas.

3.2.3. Resultados

Los resultados obtenidos son los siguientes:

Tiempo de entrenamiento (segundos):	50.45
Segundos por época:	3.049
% Acierto entrenamiento	99.381
% Fallo entrenamiento	1.973
% Acierto Prueba	98.849
% Fallo Prueba	3.049

Tabla 2: Resultados de la red

Al igual que en el caso anterior, el **porcentaje de fallo** es en relación a la **función de coste**. En este ejemplo también se observa un poco de sobreajuste (los porcentajes de fallo y acierto son ligeramente más altos en el conjunto de entrenamiento que en el de prueba).

Si lo comparamos con el modelo anterior, vemos que hemos mejorado los resultados, el fallo es mucho menor y hemos aumentado el acierto en 1%. Esto se debe a que las capas convolutivas ofrecen mejores resultados a la hora de clasificar imágenes que las capas completamente conectadas. También, al igual que la red anterior, aún hay mucho margen de mejora. Simplemente añadiendo deformaciones a las imágenes (rotándolas un poco y haciendo zoom de forma aleatoria) y aumentando el número de épocas a 20 conseguimos eliminar bastante el sobreajuste. A continuación se muestra el resultado:

Tiempo de entrenamiento (segundos):	344.929
Segundos por época:	16
% Acierto entrenamiento	99.295
% Fallo entrenamiento	2.456
% Acierto Prueba	99.269
% Fallo Prueba	2.401

Tabla 3: Resultados de la red

La red tiene ya prácticamente los mismos resultados en el conjunto de prueba y el conjunto de entrenamiento.

3.3. Primer modelo de red neuronal convolutiva

Este primer modelo se puede considerar simplemente una prueba que se hizo para entender mejor el funcionamiento de las redes y las capas, qué entradas deben recibir y que salidas proporcionan. Aún no se está intentado aumentar el porcentaje de acierto sobre el conjunto de entrenamiento.

Hay que indicar que en todos los modelos, a partir de ahora, la tasa de error se medirá en base a la función de coste.

3.3.1. Preprocesamiento

Se usa el mismo que en la red anterior (Apartado 3.2.1). Se utilizan 60000 imágenes como conjunto de entrenamiento y 10000 imágenes como conjunto de validación y conjunto de test. Después se transforman las etiquetas a matrices categóricas y se normalizan las imágenes.

3.3.2. Modelo

El modelo es incluso más simple que el anterior, solo dispone de una capa convolutiva y una *Dense.* Se muestra a continuación:

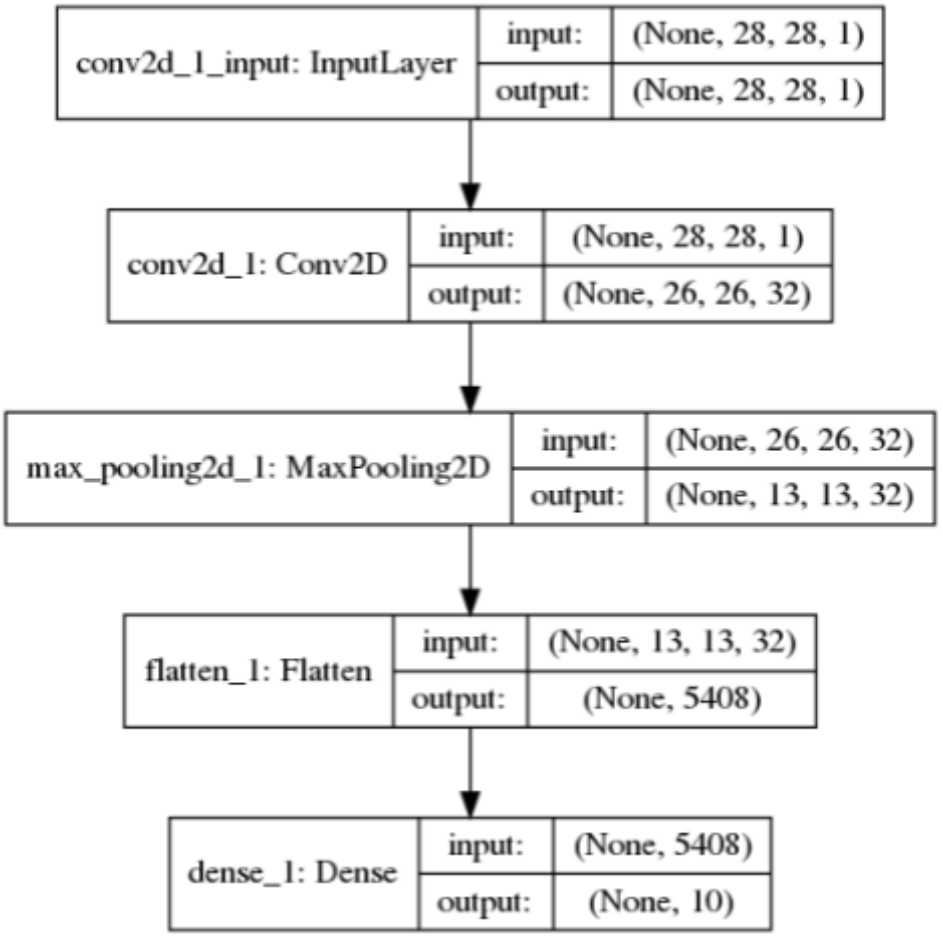


Figura 6: Esquema del modelo

La red recibe como entrada los píxeles de la imagen (matriz de 28x28) y los pasa a una capa convolutiva de 32 filtros y una ventana de 3x3, así la imagen queda reducida a una matriz de 26x26. Después pasa a una capa de *MaxPooling* con una matriz de 2x2, quedando la imagen reducida a la mitad, 13x13. Por último, se aplana la salida de esta última capa y el vector se le da directamente a la capa de salida.

La capa convolutiva tiene una activación *ReLU* mientras que la capa de salida tiene una activación *SoftMax*.

Para entrenar la red se ha usado el mismo conjunto para validación y para prueba. Como **función de coste** se ha usado ***Categorical Crossentropy*** y como **optimizador *SGD*** (Gradiente Descendente Estocástico), con una tasa de aprendizaje del 0.1 y usando un *momentum* de 0.9. Se ha usado el entrenamiento por lotes con un tamaño de 32 imágenes por lote y se ha entrenado durante 20 épocas.

3.3.3. Resultados

Los resultados obtenidos son los siguientes:

Tiempo de entrenamiento (segundos):	118.226
Segundos por época:	7
% Acierto entrenamiento	99.676
% Fallo entrenamiento	1.224
% Acierto Prueba	98.420
% Fallo Prueba	5.292

Tabla 4: Resultados de la red

En comparación con la red convolutiva anterior, esta empeora significativamente los resultados. Volvemos a tener un sobreajuste el conjunto de entrenamiento y hemos empeorado el porcentaje de aciertos. Aunque es notablemente más rápida.

De estos resultados podemos concluir varias cosas. La primera es que la estructura de capa convolutiva seguida de *MaxPooling* suele ajustarse demasiado bien al conjunto de prueba, por lo que conviene introducir ruido con un *Dropout* para evitarlo. Lo segundo es que dos capas convolutivas seguidas proporcionan mejores resultados que una sola, aunque esto hace que aumente el tiempo de entrenamiento.

También se ha probado si una capa de tipo *Dense* con muchas neuronas después de todas las capas convolutivas es mejor que colocar directamente la salida de la red, y efectivamente, es mejor. Algunos autores sugieren que el tamaño ideal de esta última capa se encuentra en 128 neuronas (cdeotte 2018).

3.4. Segundo modelo de red neuronal convolutiva

En esta segundo aproximación se buscaba encontrar una estructura que tuviese buenos resultados y que entrenase en un tiempo relativamente pequeño. Se planteó un primer modelo el cual tenía muy “malos” resultados, y a base de retocar la red se consiguió un modelo más aceptable.

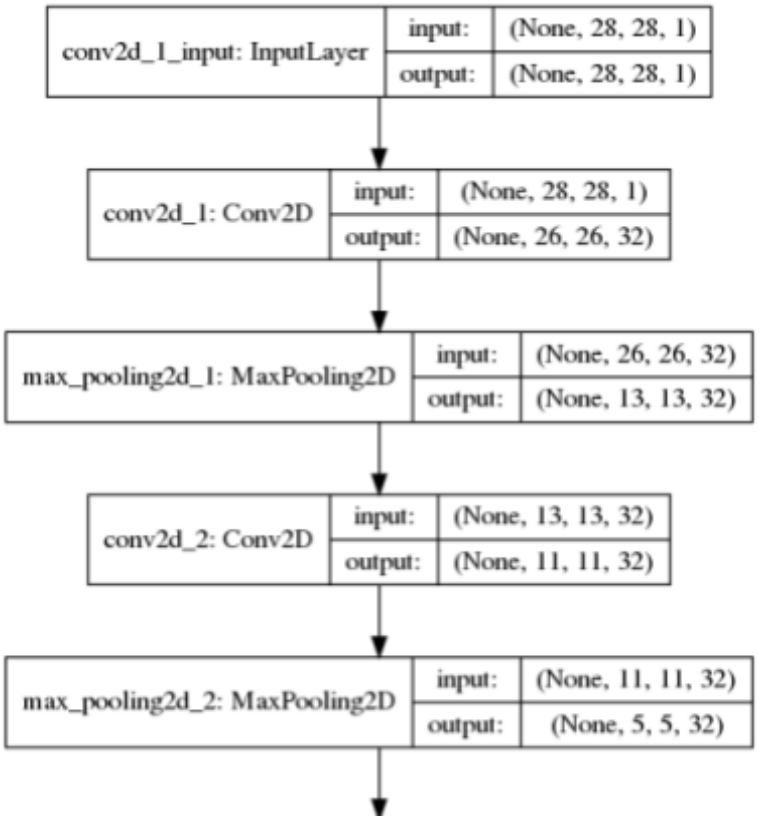
Para hacer este modelo me he ayudado de los experimentos hechos en esta página (cdeotte 2018).

3.4.1. Preprocesamiento

El preprocesamiento de los datos es exactamente el mismo que en el Apartado 3.2.1.

3.4.2. Modelo primera versión

Ahora vamos a ver el primer modelo que se propuso. El esquema es el siguiente:



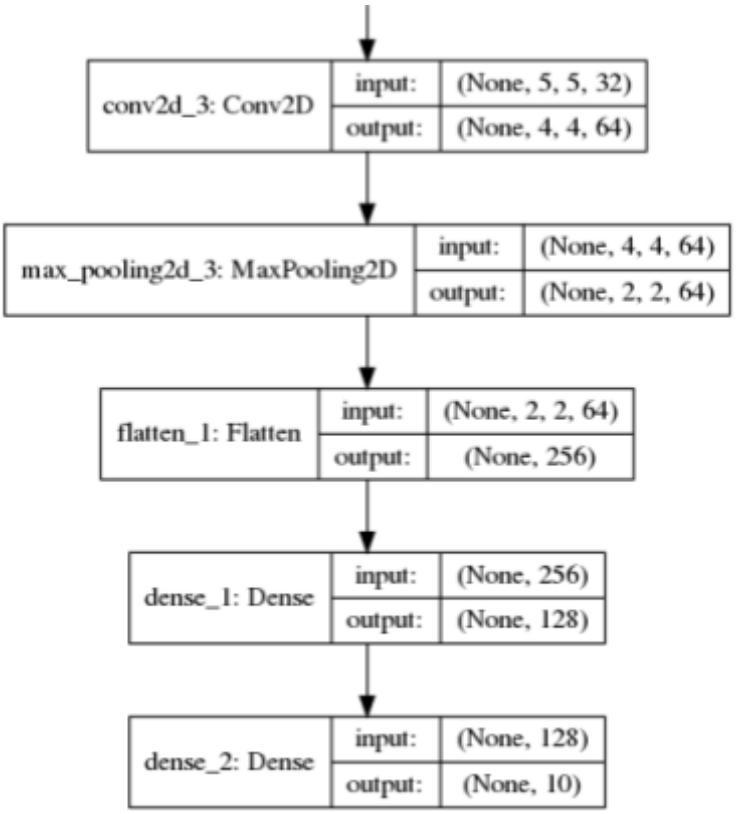


Figura 7: Esquema de la red

Consta de tres capas convolutivas seguidas de *MaxPooling*. Las capas convolutivas tienen filtros de 32, 32 y 64 respectivamente, y todas tienen una matriz de 3x3. Por otro lado, las capas de *Max Pooling* tienen una matriz de 2x2. Al final, al igual que en la red anterior, se aplana la salida de la última capa convolutiva y esta sirve de entrada a una capa *Dense* con 128 neuronas y, por último, la capa de salida.

Todas las capas convolutivas y la última capa densa tienen activación *ReLU* y la capa de salida tiene una activación *Softmax*.

Para entrenar la red se ha usado el mismo conjunto para validación y para prueba. Como **función de coste** se ha usado ***Categorical Crossentropy*** y como optimizador *RMSprop*. Se ha usado el entrenamiento por lotes con un tamaño de 32 imágenes por lote y se ha entrenado durante 20 épocas.

Lo más llamativo de este modelo es que, al no hacer una buena distribución de los filtros aplicados a la imágen, cuando llegamos a las últimas capas de la red dicha imagen es muy pequeña, de tan solo 2x2 pixeles. Este problema es el que se intenta solucionar, principalmente, para que lleguen más datos a las últimas capas con el fin de clasificar y generalizar mejor.

3.4.3. Resultados del modelo

El rendimiento del modelo es el siguiente:

Tiempo de entrenamiento (segundos):	168.45
Segundos por época:	8
% Acierto entrenamiento	99.859
% Fallo entrenamiento	0.496
% Acierto Prueba	98.89
% Fallo Prueba	8.07

Tabla 5: Resultados de la red

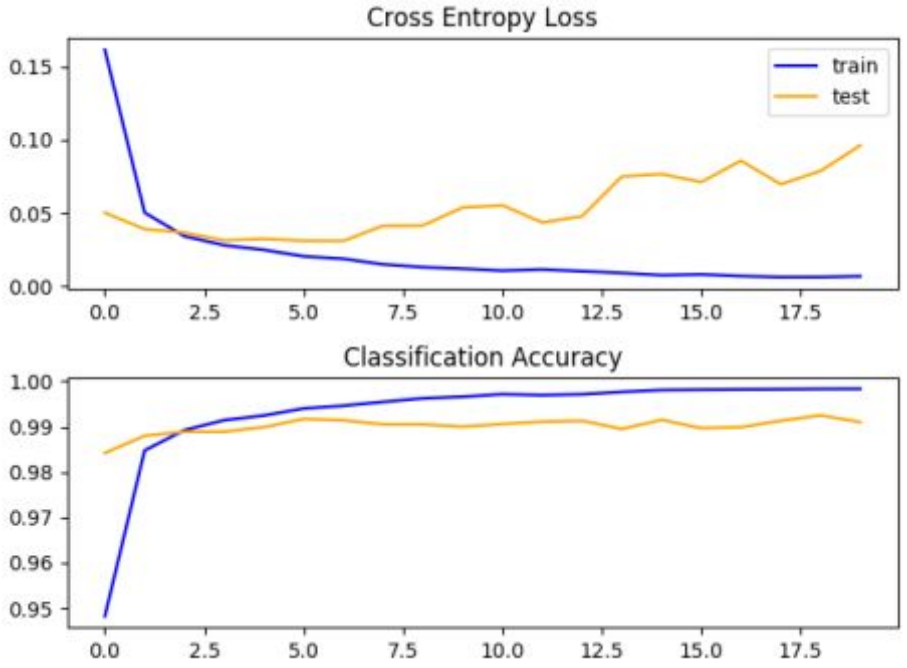


Figura 8: Historial del Accuracy y el Loss a lo largo del entrenamiento

De estos resultados podemos observar que la red se ajusta demasiado bien al conjunto de entrenamiento, casi llega a alcanzar el 100% de aciertos. También se observa en la gráfica como a partir de la época 6 el error de la función de coste se dispara.

Hasta ahora, en todas las redes, se ha estado usando el mismo conjunto para pruebas y para validación, por los que no podemos estar seguros de si la red generaliza bien, ya que ha estado viendo el conjunto sobre el que se va a comprobar los resultados durante el entrenamiento. Por esto, a partir de ahora, se usará una parte del conjunto de test para validación.

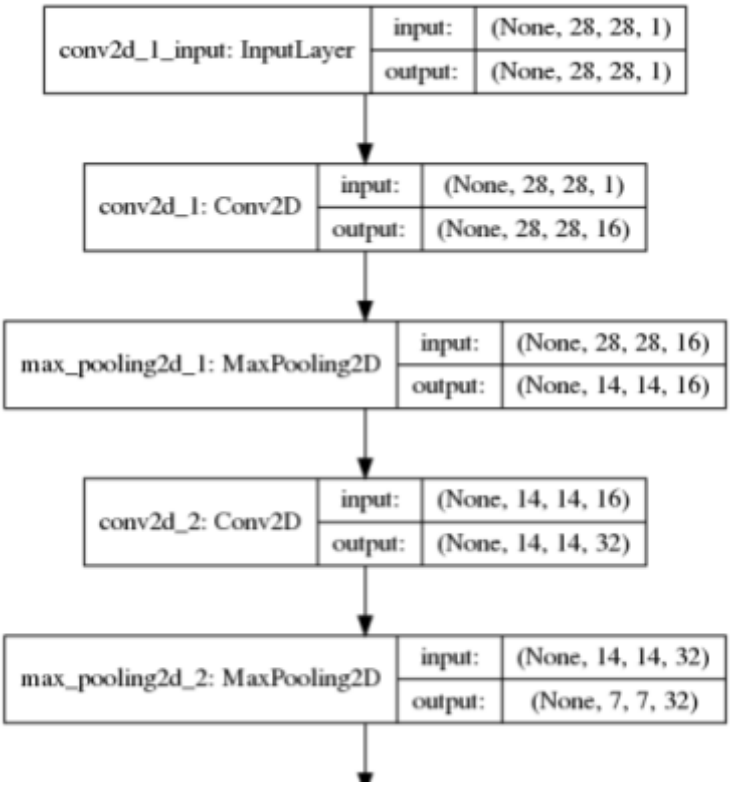
Como se ha dicho antes, la distribución de capas hace que la imagen se reduzca demasiado cuando llega a la última capa, por ello se han probado distintos filtros y tamaños de matrices para las capas convolutivas. También, con el objetivo de añadir ruido se ha colocado un *Dropout* entre la capa *Dense* y la capa de salida.

3.4.4. Preprocesamiento

Como hemos indicado anteriormente, ahora tendremos tres conjunto de datos, uno para entrenamiento con 54000 imágenes, uno de validación con 6000 imágenes y uno de prueba con 10000 imágenes.

3.4.5. Modelo segunda versión

El esquema de la versión mejorada es el siguiente:



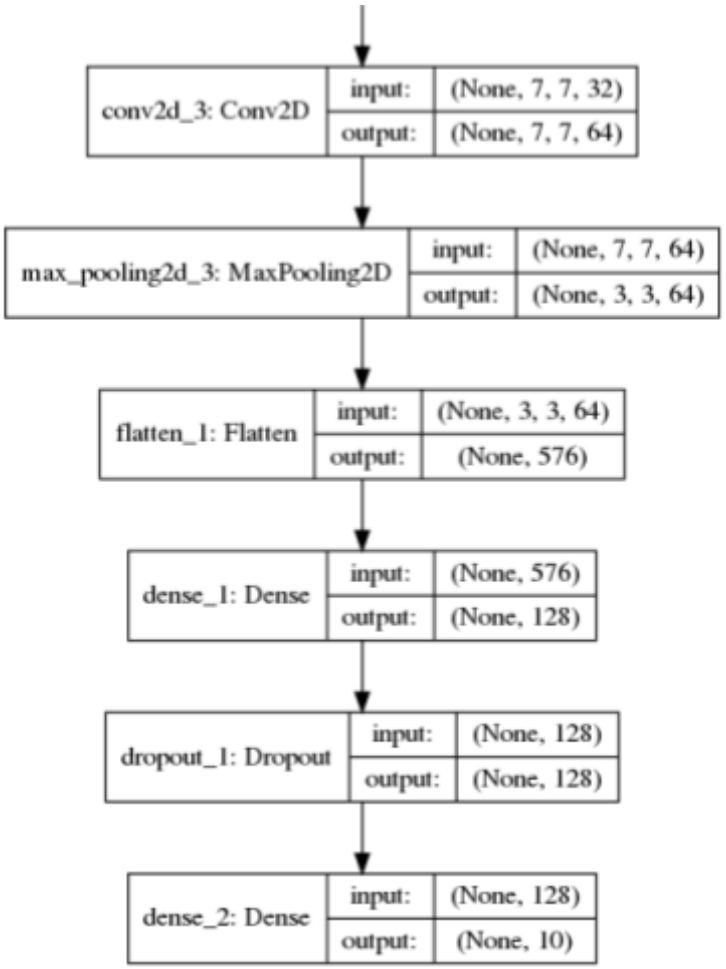


Figura 9: Esquema de la red

Respecto del modelo anterior, se ha añadido un *Dropout* para la capa *Dense* antes de la salida y se han modificado los filtros y los tamaños de las matrices para las capas convolutivas. Ahora la primera capa tiene un filtro de de 16 y una matriz de 3x3, la segunda un filtro de 32 y una matriz de 2x2 y la tercera un filtro de 64 y una matriz de 1x1. Además de les ha puesto un marco a todas las imágenes. Con esto hemos conseguido hemos conseguido aumentar la imagen a 3x3, solo ha sido un pixel más. Las funciones de activación son las mismas que en el modelo anterior.

Para entrenar se han seguido los mismos parámetros que en el modelo anterior.

3.4.6. Resultados del modelo

Los resultados que consigue este modelo son los siguientes:

Tiempo de entrenamiento (segundos):	191.87
Segundos por época:	10
% Acierto entrenamiento	99.024
% Fallo entrenamiento	0.325
% Acierto Prueba	98.549
% Fallo Prueba	5.31

Tabla 6: Resultados de la red

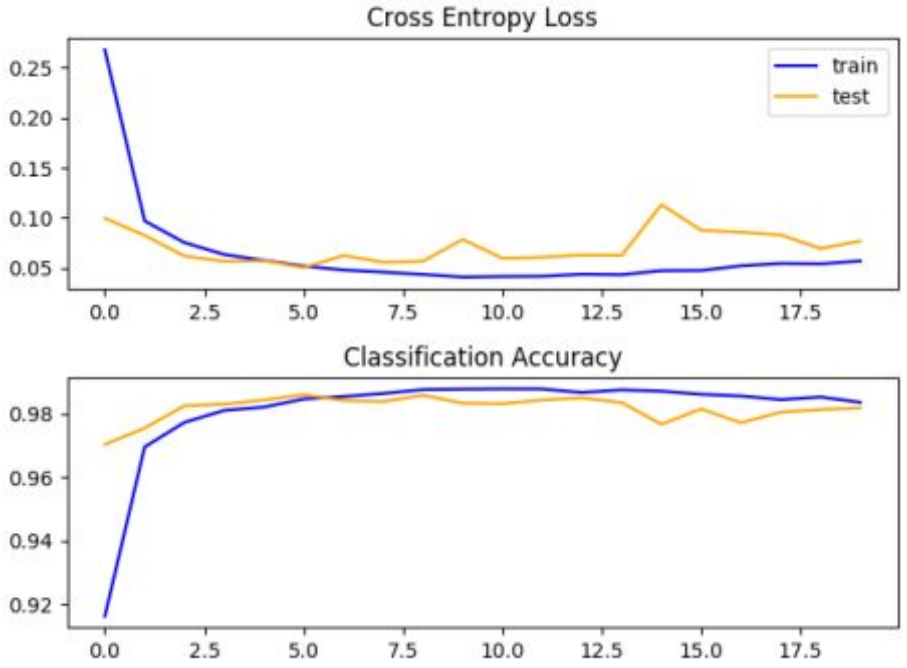


Figura 10: Historial del Accuracy y el Loss a lo largo del entrenamiento

Al hacer el cambio hemos conseguido disminuir el error y solucionar un poco el sobre ajuste, pero no ha desaparecido del todo y no se ha mejorado la tasa de acierto sobre el conjunto de prueba. Con el objetivo de solucionar esto, se creará un tercer modelo.

3.5. Tercer modelo de red neuronal convolutiva

Con este tercer modelo se quiere conseguir que no haya sobre aprendizaje y aumentar de manera significativa la tasa de acierto sobre el conjunto de prueba. El objetivo es conseguir un compromiso entre el tiempo de entrenamiento y la precisión que consigue la red.

3.5.1. Preprocesamiento

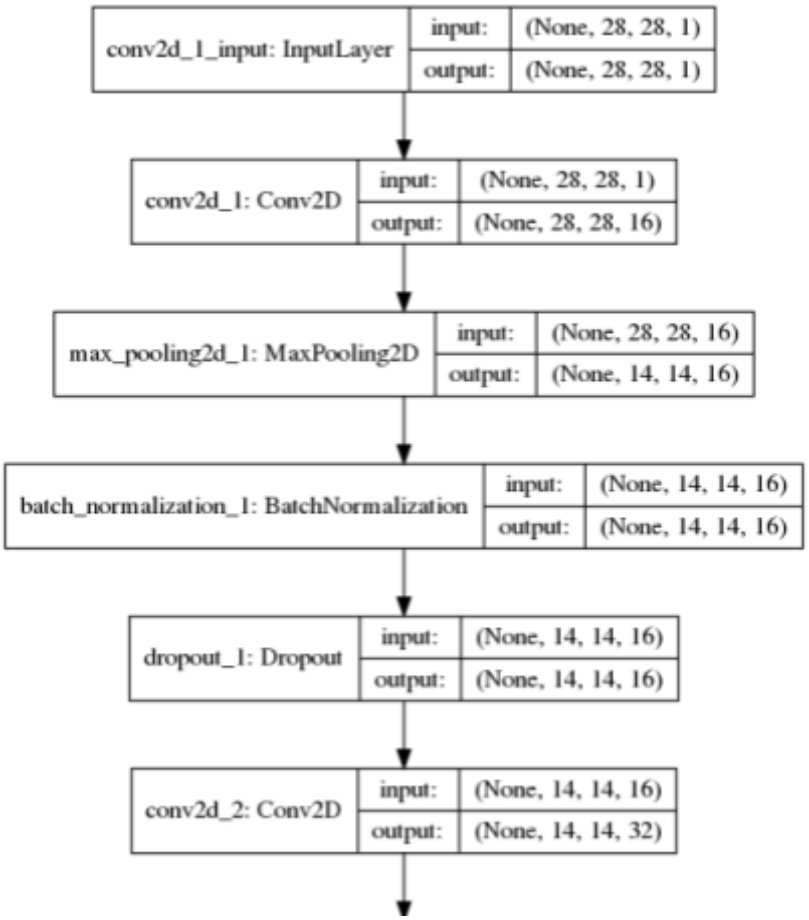
Se han dejado 54000 imágenes como conjunto de entrenamiento, 6000 imágenes como conjunto de validación y 10000 como conjunto de test. Después se han realizado las siguientes transformaciones:

- Las etiquetas se han pasado a un matriz categórica (un las clases se representan con vectores binarios con un 1 en la posición que indica la clase)
- Se han normalizado los valores de las imágenes.
- Para el conjunto de entrenamiento se ha añadido ruido a todas las imágenes. La implementación la he cogido de aquí (adityaecdrid 2019)

3.5.2. Modelo

Para realizar este modelo se ha tenido en mente el problema del sobre aprendizaje que se lleva arrastrando, para ello se ha usado más la técnica de *Dropout*. Para las capas convolutivas se ha usado regularización a nivel de capa (*kernel_regularizer)* y se ha inicializado (*kernel_initializer)*, esto se explica en la sección 2.4, pero básicamente sirve para disminuir el sobre ajuste.

La estructura de la red es la siguiente:


conv2d_1_input: InputLayer
input: (None, 28, 28, 1)
output: (None, 28, 28, 1)
conv2d_1: Conv2D
input: (None, 28, 28, 1)
output: (None, 28, 28, 16)
max_pooling2d_1: MaxPooling2D
input: (None, 28, 28, 16)
output: (None, 14, 14, 16)
batch_normalization_1: BatchNormalization
input: (None, 14, 14, 16)
output: (None, 14, 14, 16)
dropout_1: Dropout
input: (None, 14, 14, 16)
output: (None, 14, 14, 16)
conv2d_2: Conv2D
input: (None, 14, 14, 16)
output: (None, 14, 14, 32)

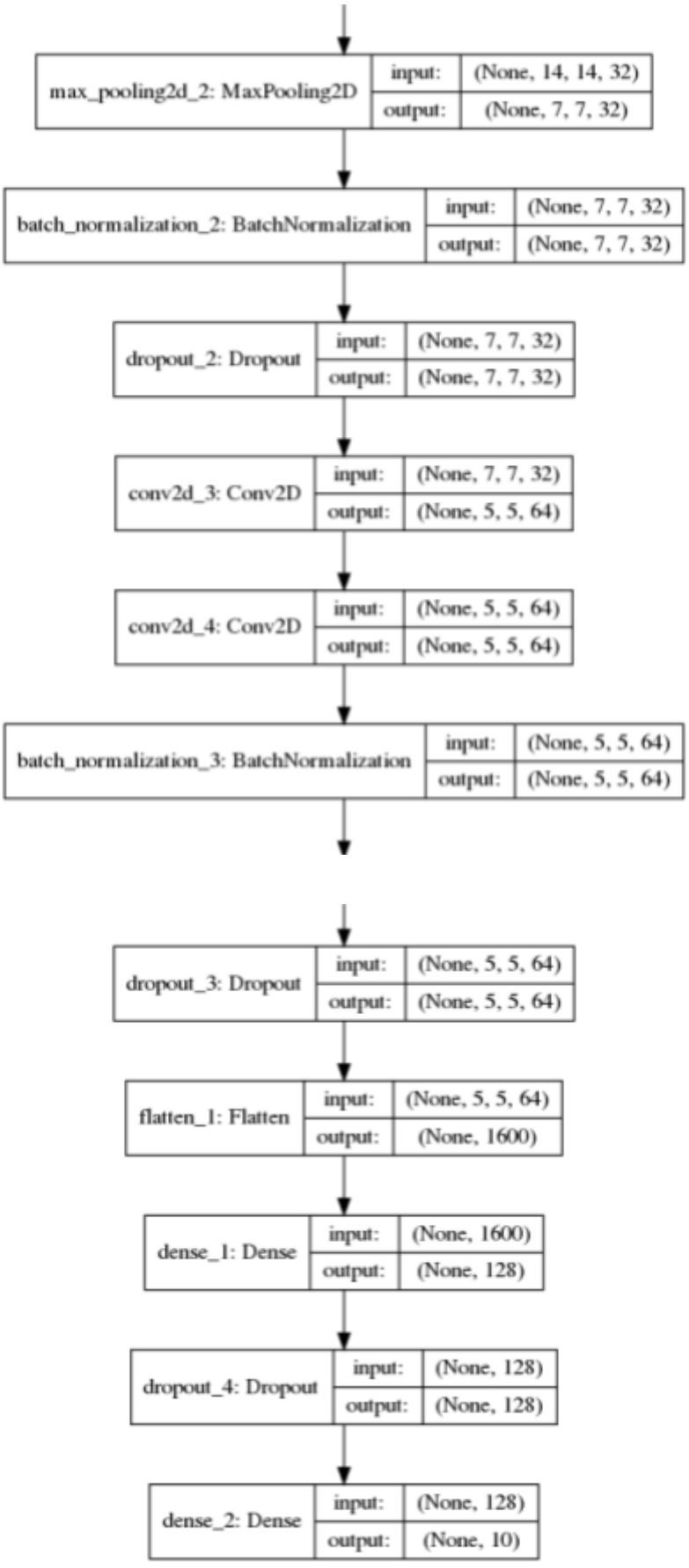


Figura 11: Esquema de la red

La red consta de 4 capas convolutivas con los siguientes parámetros (Tabla 7):

	Filtros	Tam. Matriz
Primera	16	3x3
Segunda	32	3x3
Tercera	64	3x3
Cuarta	64	1x1

Tabla 7: Distribución de filtros y tamaño de la matriz

Con esta configuración al final obtenemos una imagen de 5x5. Lo cual aumenta el tamaño de la red anterior.

Las capas de *Dropout* van aumentando el porcentaje con el que se desactivan las neuronas, siendo en la primera donde menos hay: 0.20 → 0.30 → 0.35 → 0.40.

Las capas de *MaxPooling* todas tienen un tamaño de ventana de 2x2.

También se han añadido capas de *BatchNormalization* para mejorar la velocidad de la red y hacer que generalice algo mejor.

En las últimas capas tenemos la misma configuración al final, una *Dense* con 128 neuronas y la capa de salida. Se ha probado a aumentar y disminuir el número de neuronas, pero 128 es la que ofrece mejor resultado.

Por último, para la activación se ha usado lo mismo, *ReLU* en las convolutivas y la *Dense* y *SoftMax* en la capa de salida.

Para entrenar la red se ha usado como **función de coste *Categorical Crossentropy*** y como optimizador *RMSprop*. También se ha usado reducción de la tasa de aprendizaje que se activa cuando la red lleva 3 épocas sin mejorar y se aplica una reducción de 0.5 hasta un mínimo de 0.0001.

Se ha usado el entrenamiento por lotes con un tamaño de 64 imágenes por lote y se ha entrenado durante 30 épocas. Tras probar varias combinaciones, está ha resultado ser la que tiene mejor rendimiento.

3.5.3. Resultados

Los resultados obtenidos con esta red son:

Tiempo de entrenamiento (segundos):	356.04
Segundos por época:	12
% Acierto entrenamiento	99.5185
% Fallo entrenamiento	4.84
% Acierto Prueba	99.5100
% Fallo Prueba	4.44

Tabla 8: Resultados de la red

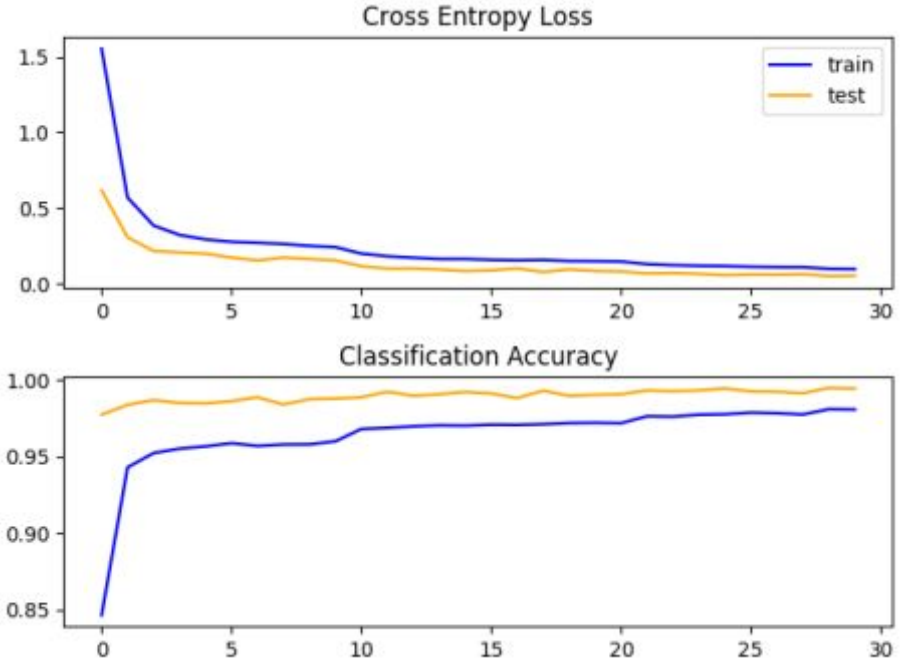


Figura 12: Historial del Accuracy y el Loss a lo largo del entrenamiento

Con esta red ya se ha conseguido solucionar el sobre aprendizaje y conseguir una tasa de acierto muy buena.

Aún se puede intentar mejorar más si aplicamos la técnica del *Early Stop,* ya que es la única técnica de optimización que queda por aplicar. La idea es usar la parada temprana para mantener la red entrenando hasta que deje de mejorar. Esto se hace usandolo junto con la reducción de aprendizaje. Esta idea se le ocurrió a mi compañero ***Antonio Morales de Haro***.

Para el *Early Stop* se ha activado que guarde la mejor configuración de pesos que haya obtenido la red durante el entrenamiento.
A continuación se muestran los resultados tras usar el *Early Stop:*

1. Con 6 épocas de paciencia, considerando una mejora del 0.0001 y monitorizando el 'val_loss'. La red ha estado entrenando un total de 69 épocas.

Tiempo de entrenamiento (segundos):	955.065
Segundos por época:	12
% Acierto entrenamiento	99.5537
% Fallo entrenamiento	3.391
% Acierto Prueba	99.5299
% Fallo Prueba	3.258

Tabla 10: Resultados de la red

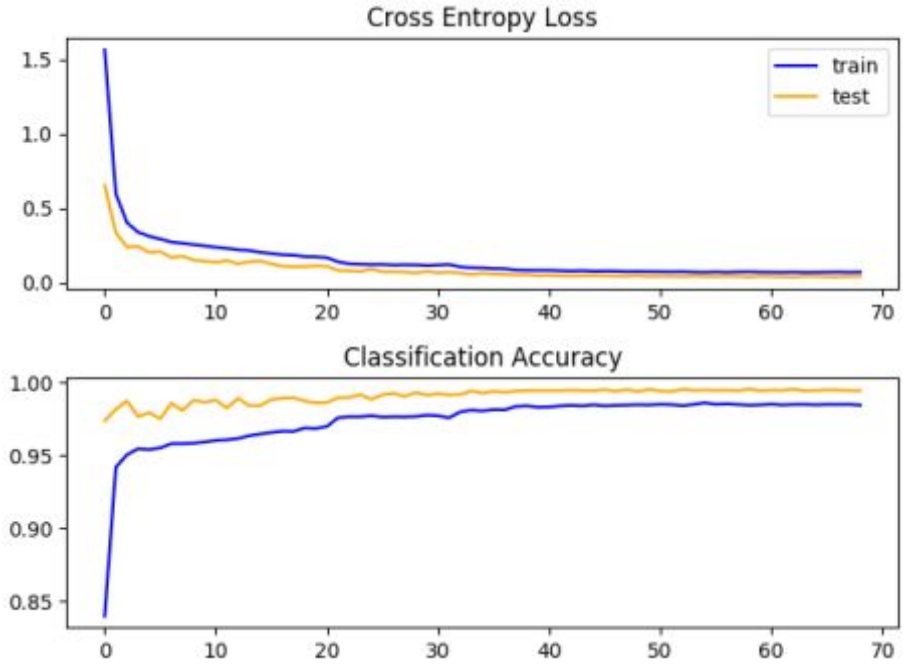


Figura 13: Historial del Accuracy y el Loss a lo largo del entrenamiento

Llegados a este punto, podemos decir, que esta es la capacidad máxima de la red, a partir de aquí es poco probable que la red mejore más.

2. Con una paciencia de 6 épocas, considerando una mejora del 0.001 y monitorizando el ‘val_loss’. La red ha entrenado un total de 15 épocas.

Tiempo de entrenamiento (segundos):	224.190
Segundos por época:	14
% Acierto entrenamiento	99.055
% Fallo entrenamiento	11.79
% Acierto Prueba	99.059
% Fallo Prueba	11.63

Tabla 11: Resultados de la red

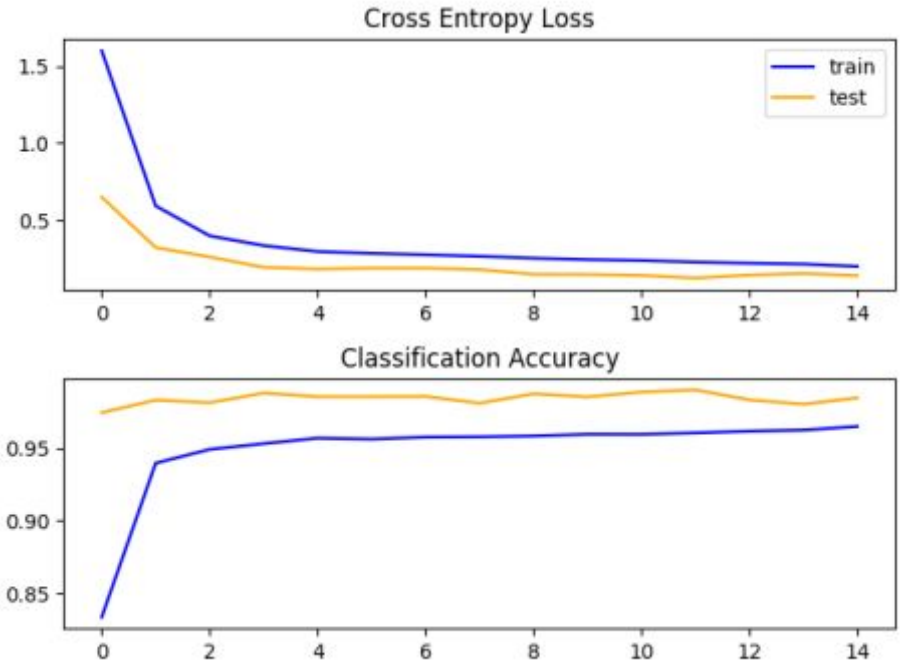


Figura 14: Historial del Accuracy y el Loss a lo largo del entrenamiento

Con esta configuración obtenemos un resultado aceptable en un tiempo muy bueno. Hay que decir que a veces hace menos épocas y la tasa de acierto cae al 98%. La pega es que la tasa de fallo (que proporciona la función de coste) aumenta muchísimo en comparación con las ejecuciones anteriores. En definitiva, se consigue menor tiempo de entrenamiento a costa de menor precisión.

3. Con una paciencia de 4, considerando una mejora del 0.0001 y monitorizando el ‘val_loss’. Con esta configuración la red entrena entre 25 y 35 épocas más o menos.

   Los resultados que se muestran a continuación la red ha entrenado durante 27:

Tiempo de entrenamiento (segundos):	330.12
Segundos por época:	6.64
% Acierto entrenamiento	99.275
% Fallo entrenamiento	11.79
% Acierto Prueba	99.320
% Fallo Prueba	6.11

Tabla 12: Resultados de la red

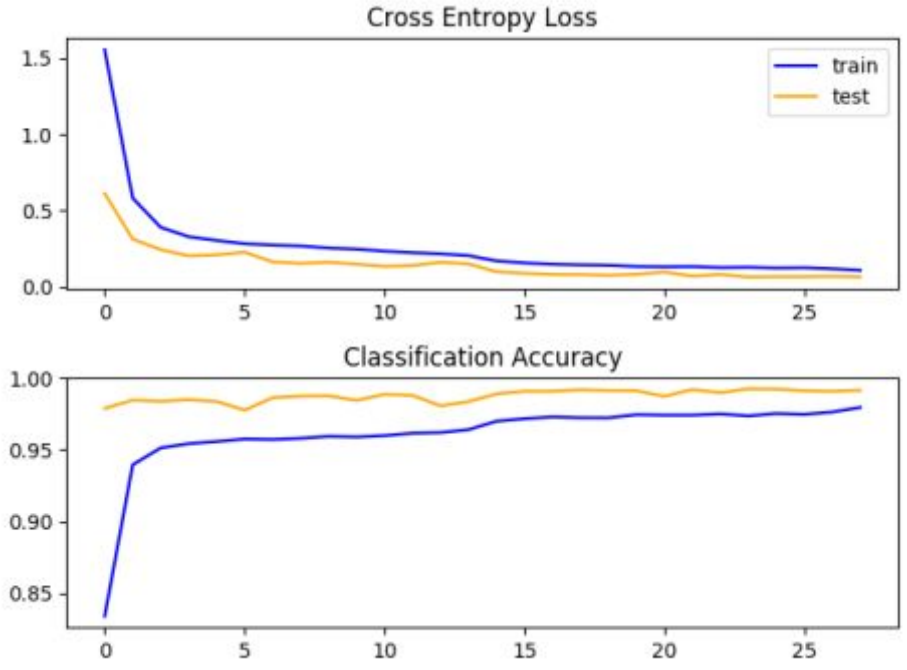


Figura 15: Historial del Accuracy y el Loss a lo largo del entrenamiento

Con esta configuración conseguimos más o menos los mismos resultados que entrenando durante 30 épocas, que es el mejor resultado que ofrece la red en términos de tiempo/precisión. Por esto vamos a dejar esta configuración.

Esta es la última red que se programó. El objetivo, como se dice al principio, era conseguir un compromiso entre tiempo que tarda la red en entrenar y la precisión que se obtiene. Con el *Early Stop* esto se consigue de manera bastante eficiente.

4. Conclusiones

En cuanto a las redes realizadas durante la práctica, se ha intentado buscar un compromiso entre tiempo de aprendizaje y precisión de la red. Añadiendo más capas convolutivas a la red se consiguen mejores resultados, sobre todo en cuestión al error proporcionado por la función de coste. La última red programada ofrece unos buenos resultados en precisión sin tener un tiempo de entrenamiento demasiado elevado, en comparación a otras redes que tienen más capas convolutivas.

La realización de esta práctica me ha servido para comprobar la utilidad de las redes neuronales en el reconocimiento de imágenes, en particular, y en el aprendizaje automático, en general. Aunque se ha realizado la implementación con una biblioteca externa, he podido comprobar la utilización práctica de lo explicado durante las clases de teoría.

Finalmente quiero añadir que gracias a todo esto, los conocimientos adquiridos durante el curso y durante la realización de la práctica, he podido profundizar más en un tema que era totalmente desconocido para mi y que ahora me resulta muy interesante.

5. Bibliografía

“About Keras Layers - Keras Documentation.” n.d. Accessed November 24, 2019. https://keras.io/layers/about-keras-layers/.

adityaecdrid. 2019. “MNIST with Keras for Beginners(.99457).” Kaggle. August 22, 2019. https://kaggle.com/adityaecdrid/mnist-with-keras-for-beginners-99457.

cdeotte. 2018. “How to Choose CNN Architecture MNIST.” Kaggle. September 24, 2018. https://kaggle.com/cdeotte/how-to-choose-cnn-architecture-mnist.

“Cheat Sheet: Keras & Deep Learning Layers — Brendan Herger.” n.d. Brendan Herger. Accessed November 24, 2019. https://www.hergertarian.com/keras-layers-intro.

“Mnist Cnn - Keras Documentation.” n.d. Accessed December 4, 2019. https://keras.io/examples/mnist_cnn/.

“MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges.” n.d. Accessed November 24, 2019. http://yann.lecun.com/exdb/mnist/.

“Mnist Mlp - Keras Documentation.” n.d. Accessed December 4, 2019. https://keras.io/examples/mnist_mlp/.

“Normalization Layers - Keras Documentation.” n.d. Accessed December 7, 2019. https://keras.io/layers/normalization/.

“One Simple Trick to Train Keras Model Faster with Batch Normalization | DLology.” n.d. Accessed December 7, 2019. https://www.dlology.com/blog/one-simple-trick-to-train-keras-model-faster-with-batch-normalization/.

“Regularizers - Keras Documentation.” n.d. Accessed December 7, 2019. https://keras.io/regularizers/.

Rosebrock, Adrian. 2018. “Keras Conv2D and Convolutional Layers - PyImageSearch.” PyImageSearch. December 31, 2018. https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/.

Saha, Sumit. 2018. “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 Way.” Medium. Towards Data Science. December 15, 2018. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

“Scikit-Learn Documentation.” n.d. Scikit-Learn: Machine Learning in Python — Scikit-Learn 0.21.3 Documentation. Accessed November 24, 2019. https://scikit-learn.org/stable/.