Universidad de Granada

Inteligencia Computacional

Práctica de redes neuronales: Reconocimiento óptico de caracteres MNIST

Realizado por:
Francisco Solano López Rodríguez
Email: fransol0728@correo.ugr.es

MÁSTER EN INGENIERÍA INFORMÁTICA

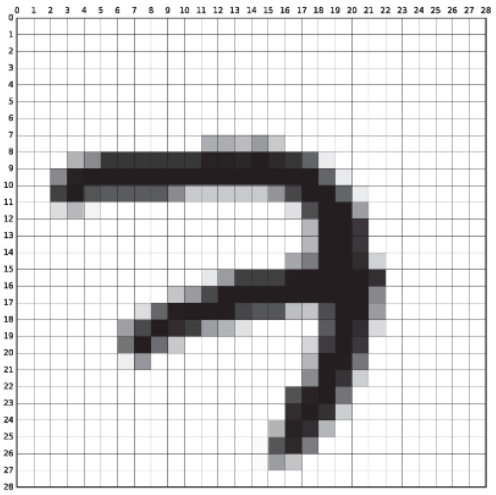
ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Índice

1. Introducción

1.1. Descripción del problema

La base de datos MNIST, se trata de una gran base de datos de dígitos escritos a mano. Esta base de datos contiene un conjunto de 60.000 imágenes para entrenamiento y un conjunto de 10.000 imágenes para validación. Cada una de las imágenes de las que se dispone, está en escala de grises, en donde el valor de cada píxel se encuentra entre 0 y 255. Los dígitos de la imagen se encuentran centrados en una imagen de 28x28 píxeles. Además de los datos con las imágenes, también se dispone de otros dos ficheros con las etiquetas de cada imagen, uno de ellos con las etiquetas del conjunto de entrenamiento y el otro con las etiquetas del conjunto de validación.



El problema que se propone es el de obtener un modelo de clasificación de dígitos, que mediante un entrenamiento previo sobre el conjunto train, sea capaz de clasificar de forma correcta las imágenes del conjunto test. Para ello se va a recurrir al uso de redes neuronales, se comenzará utilizando el modelo más sencillo, el perceptrón simple, posteriormente se recurrirá al uso del perceptrón multicapa y por último se hará uso de una red convolucional, la cual extrae características de las imagenes, mediante la convolución de dichas imágenes con una máscara. Con esta última red se conseguirá una puntuación de acierto bastante considerable sobre el conjunto de validación.

1.2. Implementación y entorno de ejecución

Todas las redes utilizadas han sido implementadas en el lenguaje de programación C++. Prácticamente toda la implementación ha sido propia, incluso las operaciones matriciales para las cuales he recurrido a la sobrecarga de operadores. En primer lugar utilicé una librería llamada Eigen, la cual permite realizar cálculos matriciales de forma eficiente haciendo uso de la GPU, pero debido a las características de mi ordenador portátil, no he podido beneficiarme de las ventajas de dicha librería, resultando ser más eficiente mi propia implementación de cálculos matriciales en mi ordenador.

Las ejecuciones han sido realizadas sobre el sistema operativo Ubuntu 18.04.3, un procesador Intel Core i3 y una memoria RAM de 16GB y un disco duro SSD.

2. Preprocesamiento

Previamente al entrenamiento de los modelos que se describirán en las siguientes secciones, los datos han sido preprocesados. Como ya se ha dicho antes las imágenes están en escala de grises, y el valor de un píxel puede variar en un rango de 0 a 255. Debido a ello hemos transformado el dominio de cada pixel al intervalo [0,1], simplemente dividiendo el valor de cada píxel por 255.

También se ha realizado una transformación sobre las etiquetas de los datos. Como bien sabemos nuestro problema trata de determinar a que dígito corresponde una imagen, es decir a que valor corresponde del 0 al 9. Por ello en todos los modelos de red neuronal que se van a utilizar en este trabajo, la capa de salida va a tener 10 nodos, uno por cada dígito posible. Es por este motivo que se han transformado las etiquetas, utilizando las técnica conocida en inglés como one-hot, en la cual cada etiqueta va a ser un vector de 10 posiciones, donde todas van a ser 0, excepto la posición correspondiente al valor del dígito, que tendrá el valor 1. Ejemplo:

$$0 \mapsto (1,0,0,0,0,0,0,0,0,0)$$
$$4 \mapsto (0,0,0,0,1,0,0,0,0,0)$$

3. Perceptrón simple

3.1. Descripción

El primer modelo utilizado ha sido el perceptrón simple. Para ello se ha implementado un perceptrón simple de una sola neurona y posteriormente una red neuronal compuesta de 10 perceptrones simple, uno por cada dígito. La implementación del perceptrón simple ha sido muy sencilla, el entrenamiento del perceptrón consiste simplemente en actualizar los pesos de la forma que se muestra en el siguiente código, en el que podemos apreciar claramente que los pesos solo se actualizan cuando la salida deseada difiere de la salida del perceptrón.

```
bias += learning_rate*(label-output);
for(int j = 0; j < input_size; j++)
    weights[j] += learning_rate*(label-output)*data[j];
```

3.2. Resultados

En la tabla siguiente se muestran los resultados de algunos experimentos realizados. En las siguientes secciones veremos como mejoran estos resultados con otros modelos más avanzados.

Épocas	Learning Rate	Train acc	Test acc	Tiempo (seg)
10	0.1	90.4433 %	90.3 %	6
20	0.1	90.2717 %	89.63 %	6
10	0.01	91.0783 %	91.43 %	6
20	0.01	91.48 %	91.36 %	6
10	0.001	87.6817 %	88.3 %	6
30	0.001	89.62 %	90.3 %	6

4. Perceptrón multicapa

4.1. Descripción

Para la implementación del perceptrón multicapa se ha creado una clase, en la cual se almacenan los pesos de la red, los bias y el número de nodos en cada capa oculta. Se ha implementado una red genérica, de forma que no solo sirve para resolver el problema que se trata en esta práctica, sino cualquier otro problema de clasificación, en el que los datos de entrada se puedan expresar en forma de vector. Para el caso que nos ocupa, el tamaño de la capa de entrada es de 28x28, un nodo de entrada por cada píxel de la imagen, y la capa de salida tendrá 10 nodos, uno por cada dígito posible. Lo que variaremos en cada experimento será el numero de capas ocultas y los nodos de cada capa oculta, además de probar distintos parámetros como por ejemplo distintos valores para la tasa de aprendizaje.

Inicialización de los pesos y bias

Los bias han sido inicializados a 0. Los pesos serán inicializados según una variable aleatoria que sigue una distribución normal de media 0 y varianza igual a 1, dicha variable aleatoria se dividirá por la raíz cuadrada del número de nodos en la capa anterior, es decir:

$$W^l \sim N\left(0, \frac{1}{\sqrt{n_{l-1}}}\right)$$

Funciones de activación

Como función de activación en las capas ocultas se ha utilizado la sigmoide y para la capa de salida una softmax. También utilicé como función de activación para las capas ocultas la función relu, que como ventaja frente a la sigmoide tiene que es muy sencilla de calcular y más aún su derivada, lo que hace que los cálculos vayan más rápidos, pero finalmente opté por quitarla debido a que daba unos resultados algo inferiores a los obtenidos por la sigmoide.

Comentar que para la softmax apliqué un pequeño truco a la hora de implementarla. El problema que tiene la softmax es que los valores pueden ser muy grandes al utilizar una exponencial y más aun en el denominador al ser una sumatoria de exponenciales y dividir por números muy grandes puede ser numéricamente inestable, lo cual puede dar como resultado valores incorrectos. Para solucionar esto se ha multiplicado numerador y denominador por una constante, que al pasarla dentro de la exponencial queda como el logaritmo de dicha constante, de esta forma podemos elegir una constante C que reduzca el exponente y haga que el resultado de calcular la exponencial no de un valor excesivamente alto. El valor que se ha escogido para $\log C$ es $-\max(z_1, \ldots, z_n)$.

$$\frac{e^{z_i}}{\sum_j e^{z_j}} = \frac{Ce^{z_i}}{C\sum_j e^{z_j}} = \frac{e^{z_i+\log C}}{\sum_j e^{z_j+\log C}}$$

Función de pérdida

Primeramente cuando implemente la red neuronal, lo hice solamente con la función de activación sigmoide, para todas las capas, la de salida inclusive, y comencé usando como función de pérdida el error cuadrático. Después cambié la implementación y sustituí la función de activación de la capa de salida por la softmax, por lo que cambié la función de pérdida a la función de entropía cruzada. Por tanto para calcular el gradiente de los pesos y los bias durante el backpropagation se ha tenido en cuenta que la función que se pretende minimizar es la función de entropía cruzada, la cual tiene la siguiente expresión matemática:

$$L(\mathrm{y}, \hat{\mathrm{y}}) = -\sum \mathrm{y}_i \log(\hat{\mathrm{y}}_i)$$

Algoritmo de optimización

Como algoritmo de optimización se ha usado el gradiente descendente estocástico, en el cual el conjunto de entrenamiento se particiona en mini lotes y se ejecuta el algoritmo de backpropagation sobre dichos mini lotes, se van sumando los gradientes obtenidos por los elementos del mini lote y finalmente se actualizan los pesos y los bias, restando los grandientes multiplicados por el factor de aprendizaje.

Se han probado diferentes tamaños del mini lote y el que mejor resultado ha dado ha sido el tamaño igual a 1, es decir entrenar elemento por elemento, que se conoce también como entrenamiento online.

Regularización

Como técnica de regularización he probado los métodos de regularización L1 y L2, los cuales consisten en añadir una penalización en la función de coste de manera que se pondere también el valor de los pesos de la red, con el objetivo de que los pesos no sean demasiado grandes y produzca un modelo más simple. Sin embargo finalmente opté por no usar dichos métodos de regularización, debido a que obtenía resultados levemente inferiores.

4.2. Resultados

En este apartado vamos a mostrar los resultados de algunos de los experimentos realizados. En todos ellos como ya se ha dicho la función de activación para las capas ocultas es la sigmoide y para la capa de salida la softmax. Además el entrenamiento es online, es decir el tamaño de los minibatches es de un solo elemento. El learning rate es de 0.01 y el entrenamiento de 30 épocas.

Experimento 1

En primer lugar vamos a mostrar los resultados de una red sencilla con una sola capa oculta de 32 nodos, veremos como esta sencilla red es capaz de mejorar considerablemente los resultados del perceptrón simple. Los resultados han sido los siguientes:

```
train_loss: 0.0822066 - train_acc: 0.9846
test_loss: 0.166355 - test_acc: 0.9638
tiempo: 293 segundos
```

Como podemos ver se ha obtenido un porcentaje de acierto de un 98,46 % en el conjunto train y un 96,38 % en el conjunto test. A continuación se van a mostrar unas gráficas en las que vamos a ver la evolución de accuracy y la función de pérdida durante el transcurso de cada época. Como puede observarse llega un momento en el que el conjunto train sigue mejorando pero el de test se estanca.

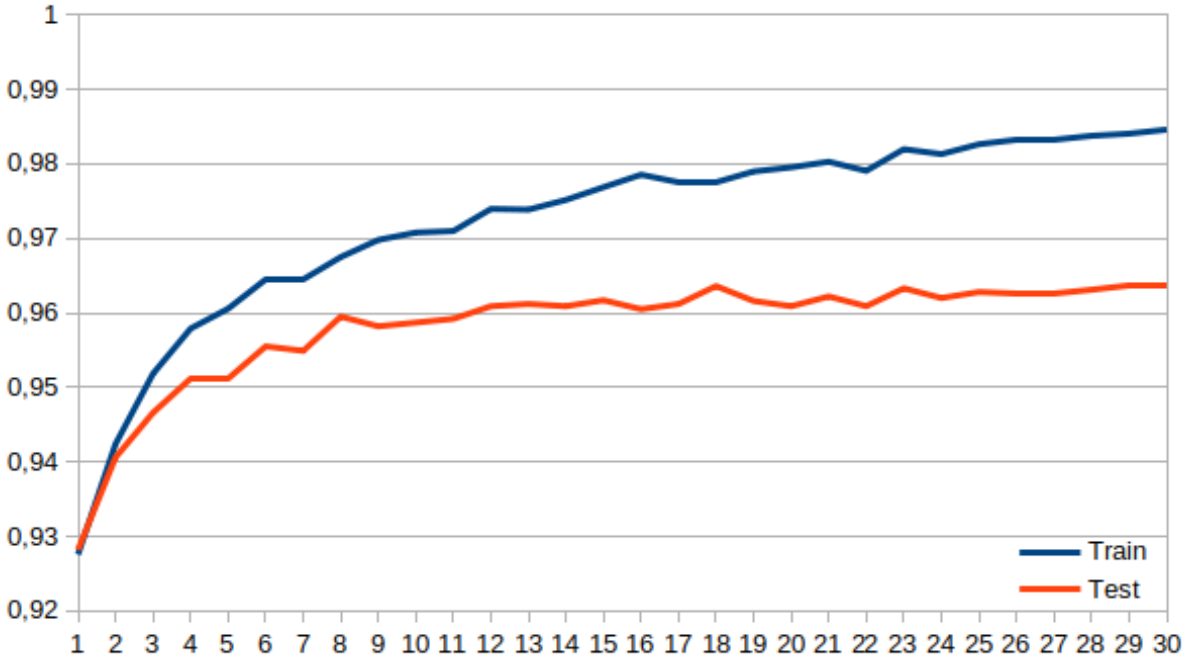


Figura 1: Precisión red multicapa 1 capa oculta 32 nodos

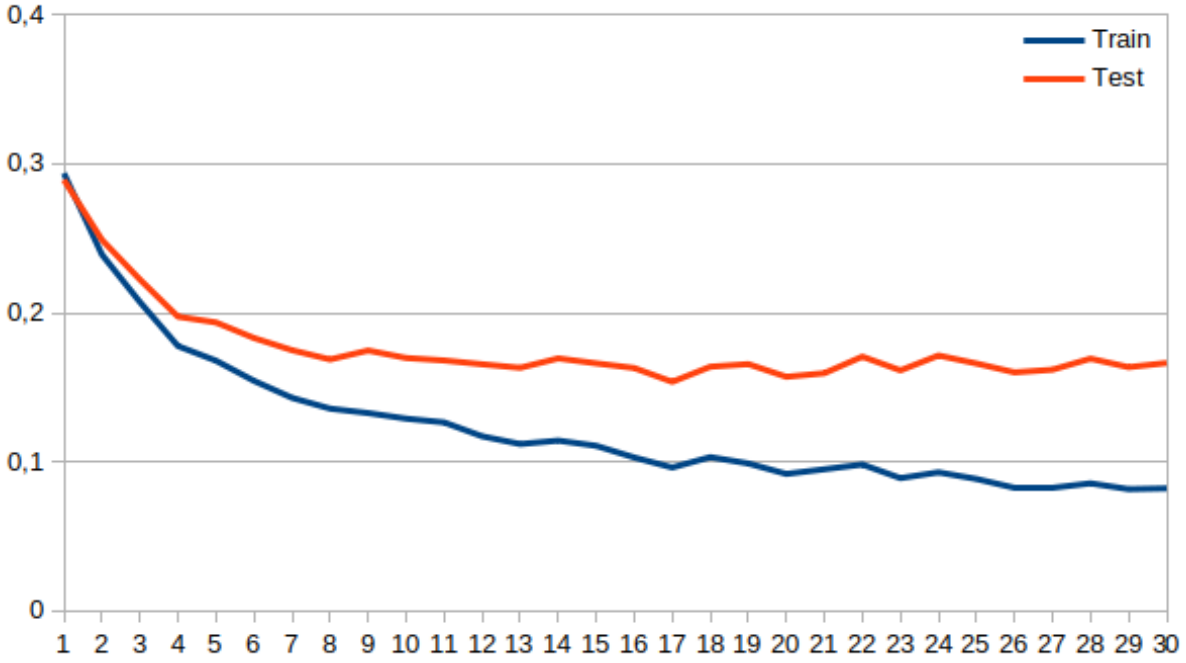


Figura 2: Función de perdida red multicapa 1 capa oculta 32 nodos

Experimento 2

En este segundo experimento voy a mostrar los resultados de la red multicapa que me ha dado mejores resultados, esta tan solo tiene una capa oculta con 256 nodos.

```
train_loss: 0.0214857 - train_acc: 0.997467
test_loss: 0.0812664 - test_acc: 0.9815
tiempo: 3142 segundos
```

Como podemos apreciar en este segundo experimento en el conjunto de entrenamiento, la función de pérdida es muy cercana al 0 y el accuracy es prácticamente igual a 1. En el conjunto de validación también ha bajado el valor de la función de pérdida en comparación con el experimento anterior, además ya se ha superado el 98 % de acierto sobre el test.

Respecto al tiempo podemos ver que ya es bastante elevado siendo de unos 150 segundos por cada época, lo que en total ha dado un tiempo de ejecución total de ás de 3000 segundos.

A continuación se muestra la gráfica del accuracy, en ella se puede apreciar como va subiendo hasta llegar prácticamente a 1 en el conjunto de entrenamiento, en cambio en el conjunto test una vez que llega a 0.98 se queda un poco estancado y ya no se aprecian mejoras.

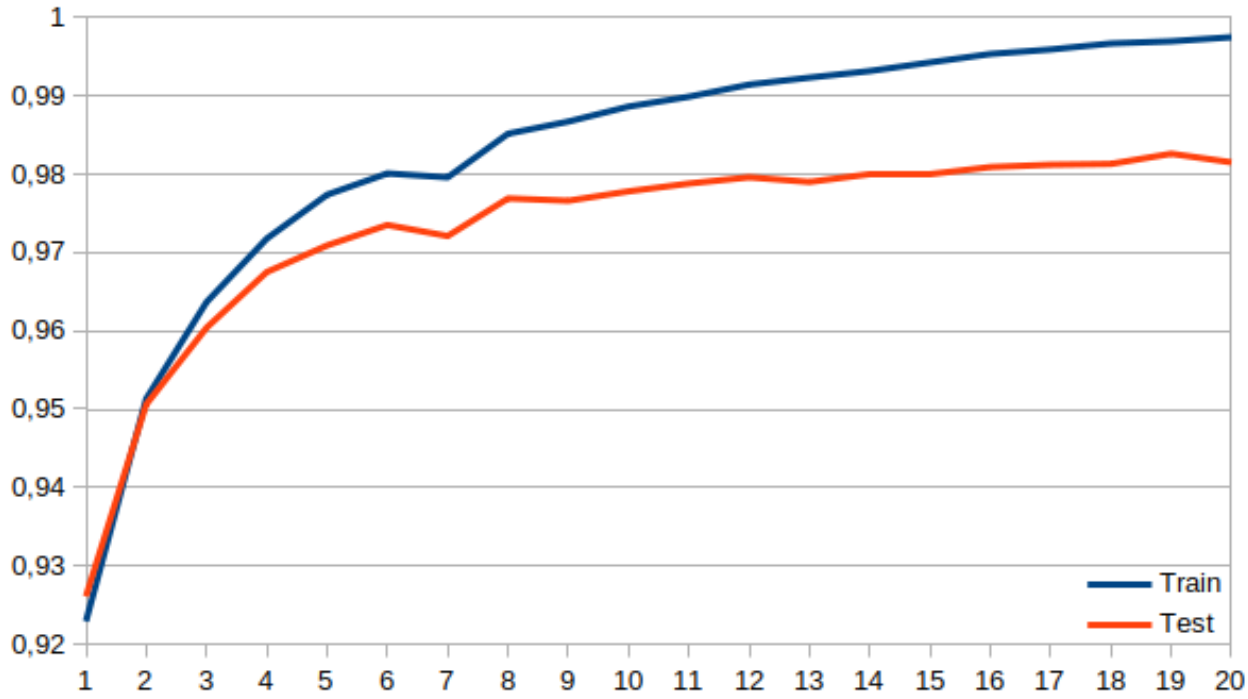


Figura 3: Precisión red multicapa 1 capa oculta 256 nodos

En la siguiente gráfica se muestra la evolución de la función de pérdida. De nuevo podemos apreciar como en el train sigue bajando, pero en el test se vuelve a estancar.

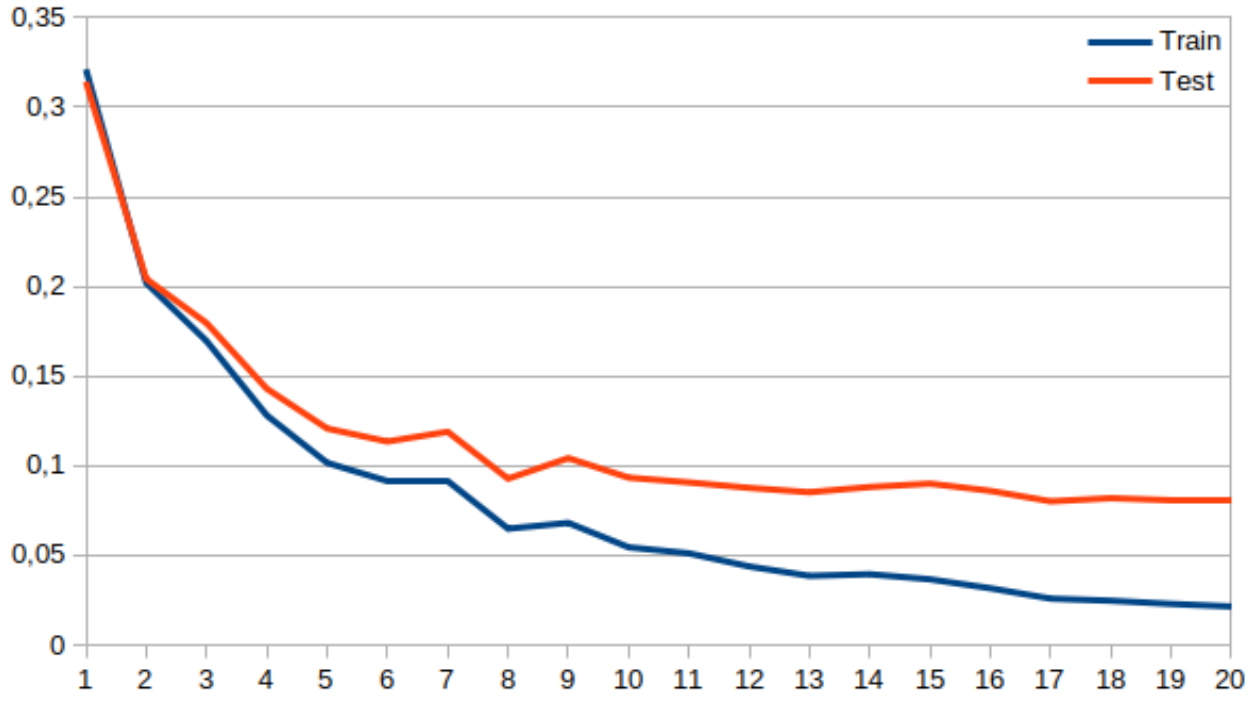


Figura 4: Función de perdida red multicapa 1 capa oculta 256 nodos

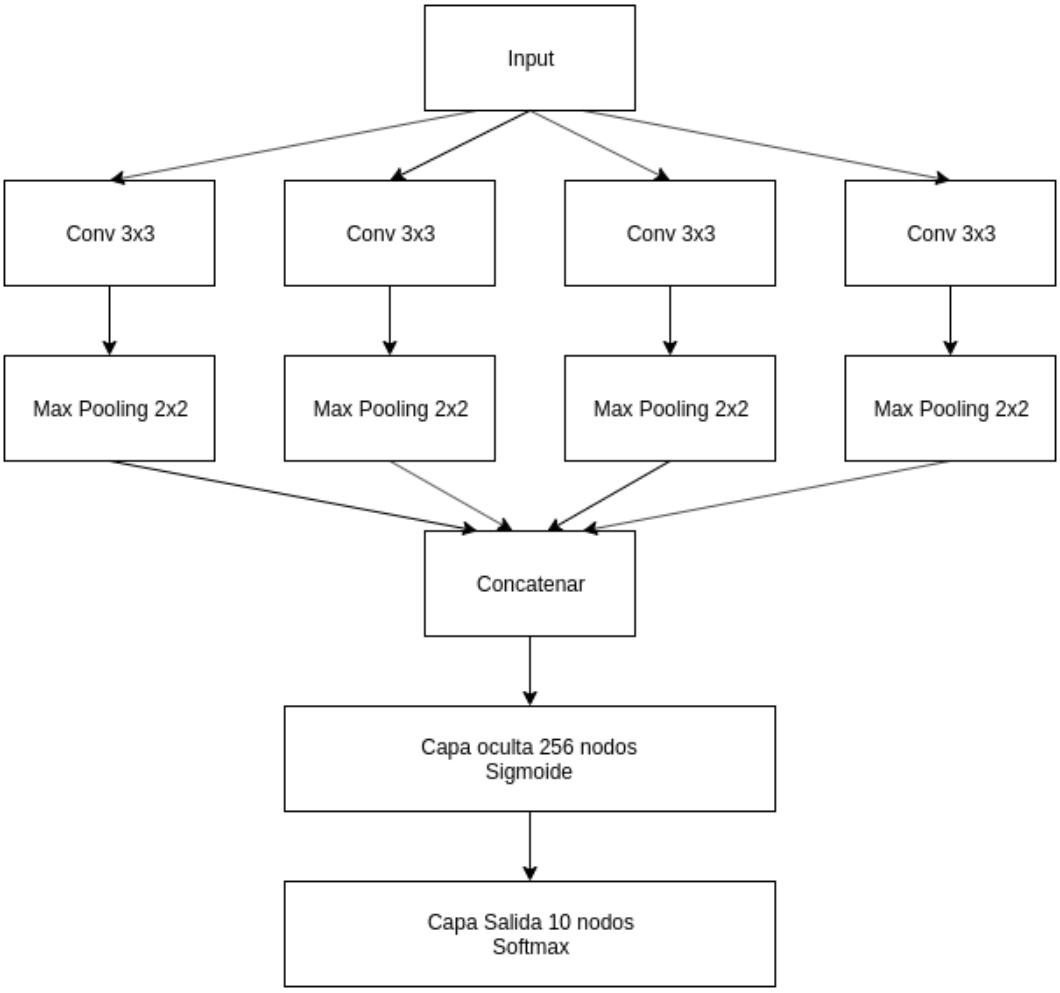
5. Red neuronal convolucional

5.1. Descripción

Para esta red se ha implementado una capa de convolución y una de pooling. Las máscaras utilizadas para la convolución han sido todas de 3x3 y el pooling se ha hecho sobre bloques de 2x2. Para la propagación hacia atrás para obtener el gradiente del pooling no se han tenido que hacer operaciones adicionales, simplemente se han puesto los valores del gradiente que recibe en las posiciones que tenían el valor máximo en los bloques 2x2 del pooling y un cero en las demás. Obtener los gradientes de la capa convolucional también ha sido sencillo, solamente se tenían que realizar una convolución de la entrada de la capa convolucional con el gradiente del pooling.

En un primer lugar solo implemente la capa de convolución, pero el proceso de entrenamiento era demasiado lento y eso que solamente incluí una capa de convolución. Debido a ello acabé implementando la capa de pooling. Respecto a la parte de perceptrón multicapa que hay tras las capas de convolución y pooling, se ha utilizado el mismo modelo que el que se describió en el experimento 2 de perceptrón multicapa.

La red neuronal convolucional que he implementado finalmente y con la que he obtenido los mejores resultados tiene la arquitectura que se ve en la siguiente imagen (No se han añadido más capas de convolución debido al largo tiempo de ejecución que suponía).



5.2. Resultados

Los resultados obtenidos por la red neuronal descrita son los siguientes:

```
train_loss: 0,0012713 - train_acc: 1
test_loss: 0,0464722 - test_acc: 0,9860
tiempo: 6747 segundos
```

Como podemos ver el valor de la función de pérdida llega ya prácticamente a cero en el conjunto de entrenamiento y el accuracy a llegado a 1, es decir un acierto en el 100 % de los casos. En el conjunto test el valor de la función de pérdida se ha reducido en la mitad, con respecto al experimento 2 de la red multicapa y el accuracy ha sido de 0.986, siendo el valor más alto alcanzado en los experimentos.

En el diseño de esta red como podemos ver tan solo se usa una capa de convolución con 4 máscaras, esto es debido al enorme tiempo de ejecución que suponía añadir más, probablemente si se hubieran añadido más máscaras y más capas, se habrían alcanzado mejores valores.

A continuación se muestran las gráficas de la evolución del accuracy y de la función de pérdida a lo largo de las épocas.

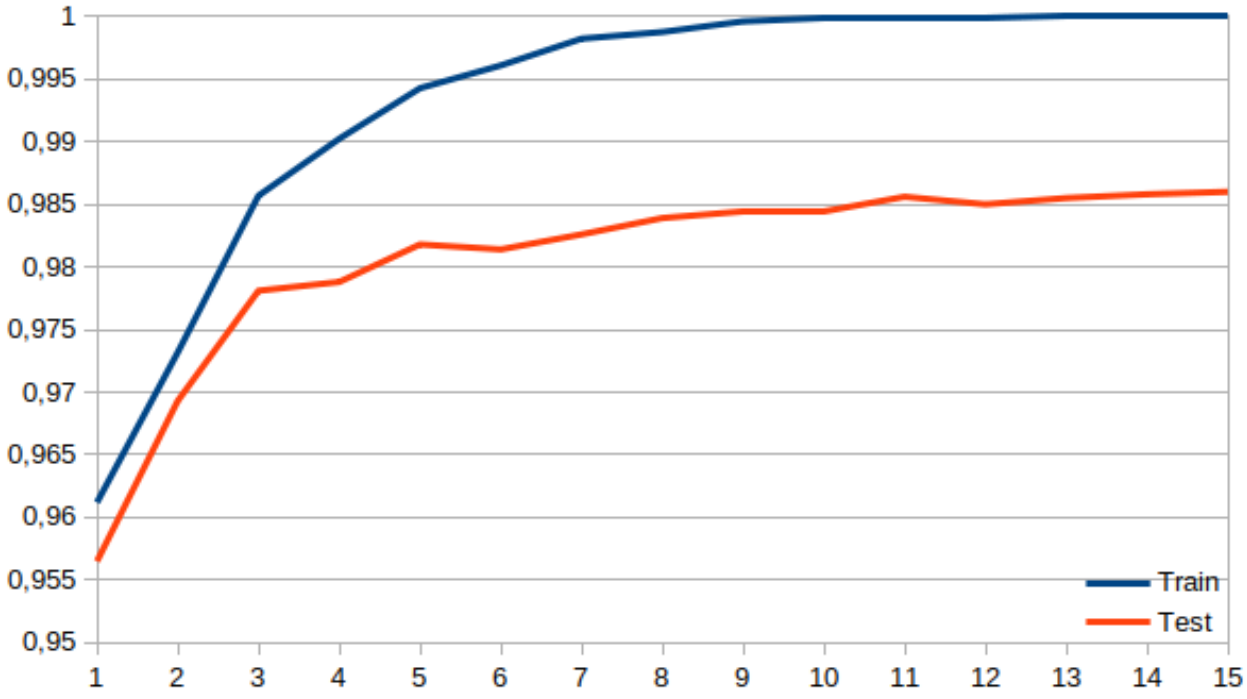


Figura 5: Precisión red convolucional

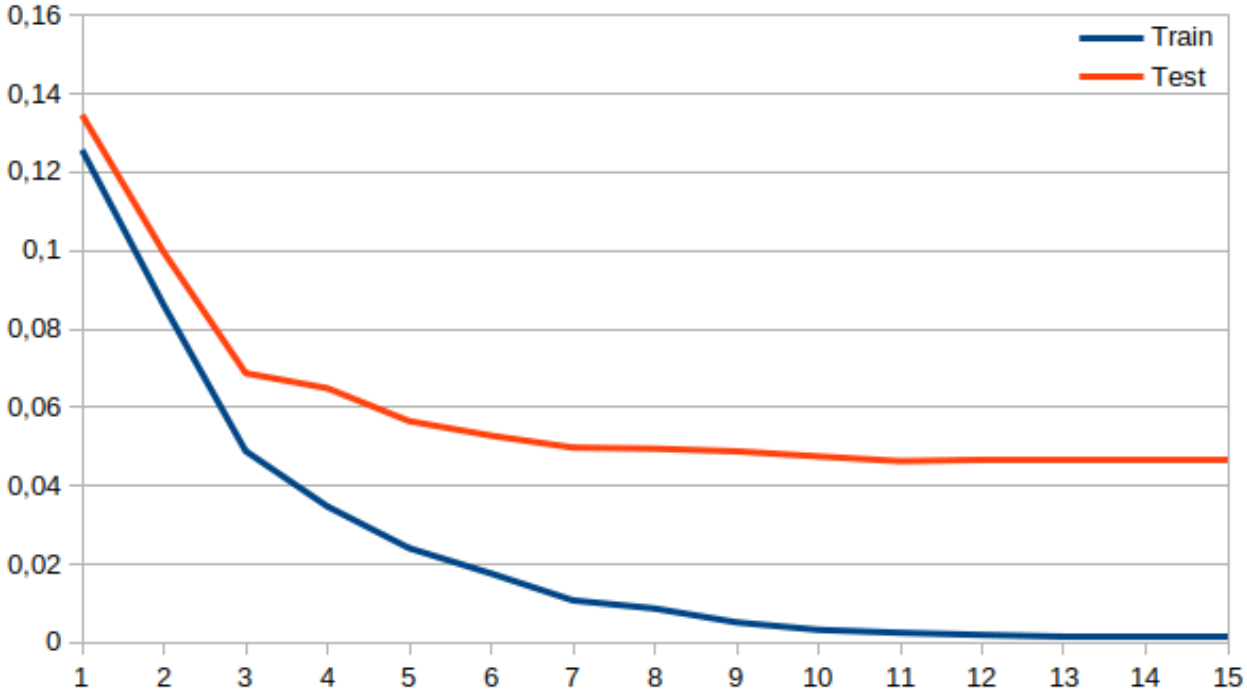


Figura 6: Función de perdida red convolucional

6. Bibliografía

Además del material de clase se han consultado las siguientes referencias:

- http://neuralnetworksanddeeplearning.com/chap1.html
- http://cs231n.github.io/neural-networks-1/
- https://mc.ai/convolutional-backpropagation/