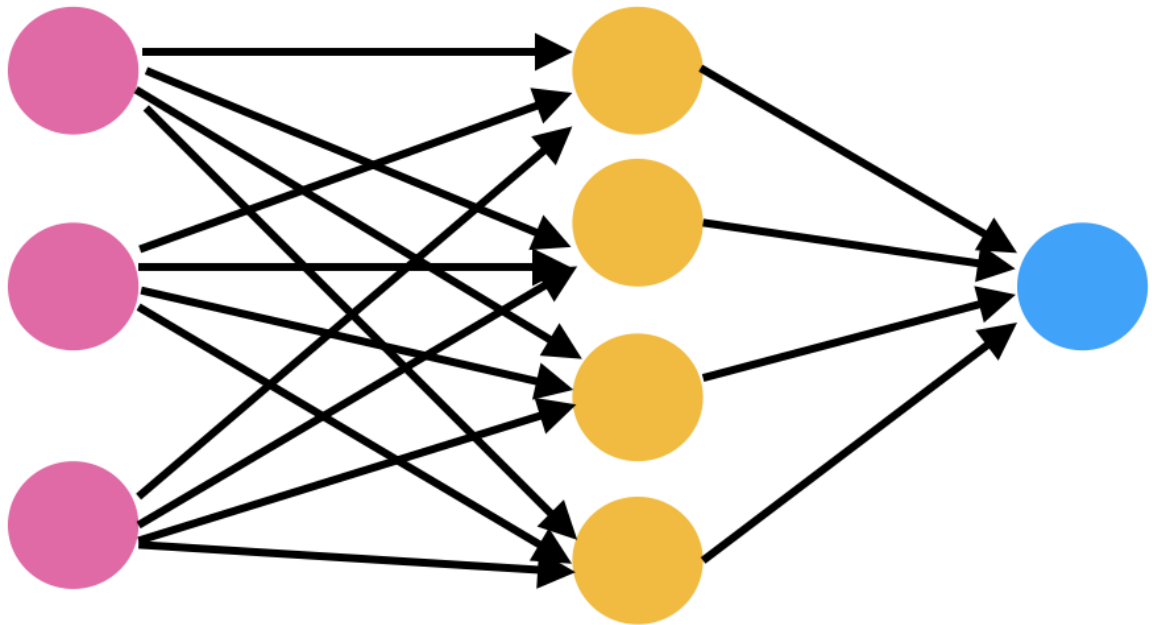


MEMORIA DE INTELIGENCIA COMPUTACIONAL:

REDES NEURONALES



ÍNDICE

1. Introducción.....	3
1.1 Base de datos.....	3
1.2 Preprocesamiento de datos.....	4
1.3 Redes neuronales.....	4
1.4 Entrenamiento.....	6
1.5 Test.....	6
2. Primera versión (Perceptrón).....	6
2.1 Entrenamiento.....	7
2.2 Test.....	7
2.3 Resultados.....	8
3. Segunda versión.....	8
3.1 Preprocesamiento.....	9
3.2 Entrenamiento.....	9
3.3 Test.....	11
3.4 Resultados.....	11
4. Conclusión.....	14
5. Bibliografía.....	14

1. Introducción

En esta memoria se va a explicar los pasos seguidos para desarrollar una red neuronal y cómo se ha entrenado para resolver un problema. El objetivo es entrenar una red neuronal de varias capas para que pueda reconocer dígitos escritos a mano. La base de datos usada para su entrenamiento y test se trata de “*The MNIST database of handwritten digits*” de Yan LeCunn [1].

1.1 Base de datos

Para el entrenamiento de la red cuenta con dos archivos (archivos de entrenamiento). El primero de 60.000 imágenes de 28x28 píxeles en las cuales aparece un dígito numérico entre 0 y 9. También tiene otro archivo de 60.000 etiquetas de números entre 0 y 9 que corresponden por orden de posición a las imágenes anteriormente explicadas. Para la comprobación del entrenamiento realizado en la red neuronal existen otros dos archivos de las mismas características (archivos test) pero esta vez con 10.000 elementos.

Las imágenes vienen almacenadas en un array de tamaño 60.000 donde cada elemento de ese array es una matriz de enteros de 28x28. En total se tienen 47040000 elementos para el archivo de imágenes de entrenamiento. Cada elemento de la matriz o píxel es un número entero entre 0 y 255 que representa un valor en la escala de grises, siendo 0 totalmente blanco (no hay nada escrito) y 255 totalmente negro (se ha escrito en ese píxel). A continuación se muestran algunas imágenes del conjunto de imágenes de entrenamiento [2]:

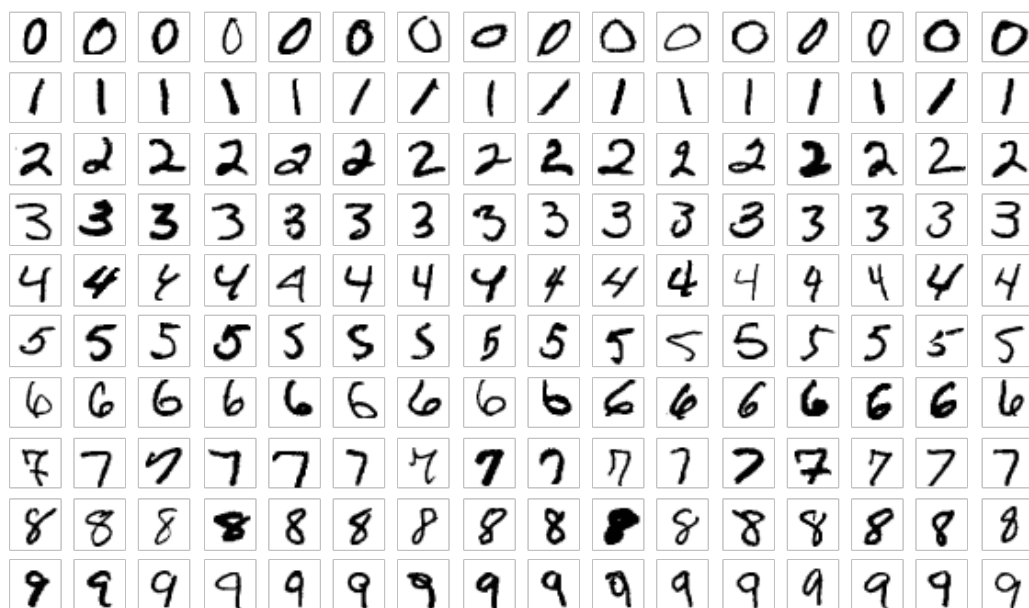


Ilustración 1

Las etiquetas tanto para el conjunto de entrenamiento como para el conjunto de test vienen almacenadas en un array donde cada elemento de ese array es un número entero entre 0 y 9.

1.2 Preprocesamiento de datos

Dependiendo del caso se realizarán modificaciones al conjunto de datos, en concreto a las imágenes del conjunto de entrenamiento para que la red neuronal sea capaz de interpretarlas o bien pueda aprender mejor de ellas.

Los tipos de preprocesamiento de datos que se han hecho a lo largo de la práctica son los siguientes:

- Normalización de los datos: Se han transformado los valores de la imagen de $[0,255]$ a $[0,1]$ simplemente al dividir cada valor entre 255. Esto se ha hecho para que a la hora de aprender no se trabajen con valores tan altos que pueden dar lugar a desbordamiento de variables y porque es conveniente trabajar con valores menos altos para el aprendizaje de la red.
- Cambio de formato: Se han transformado las matrices de las imágenes de 28×28 a vectores de 784 elementos con el objetivo de mejorar ligeramente su acceso.
- Transformaciones en las imágenes: Se ha usado un generador de transformaciones aleatorias en las imágenes el cual las desplaza sobre algún eje, les aplica zoom, las rota ligeramente, etc. Con esto se pretende que la red generalice mejor al usar un conjunto de entrenamiento más variado.

1.3 Redes neuronales

Una red neuronal es un modelo de computación basado originalmente en el funcionamiento del cerebro. Las redes neuronales están formadas por neuronas simples que constituyen una unidad de cálculo en la que reciben una serie de entradas y obtienen una salida (ilustración 2).

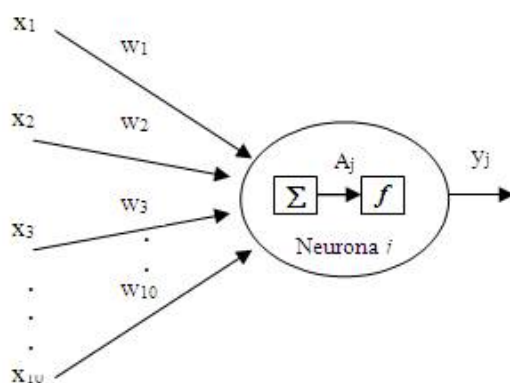


Ilustración 2

No voy a entrar en detalles técnicos del funcionamiento de una neurona artificial porque la memoria no trata de eso. Sí se va a explicar, sin embargo, que se han usado redes neuronales de una o más capas para el reconocimiento de los dígitos explicados en la sección anterior. Una capa no es más que un conjunto de neuronas la cual obtiene un conjunto de entradas de la anterior capa y genera un conjunto de salidas que envía a la siguiente capa.

Para esta práctica se han usado dos versiones de redes neuronales:

- Versión del perceptrón: Esta versión consiste en una red neuronal de una capa formada por perceptrones de activación lineal.

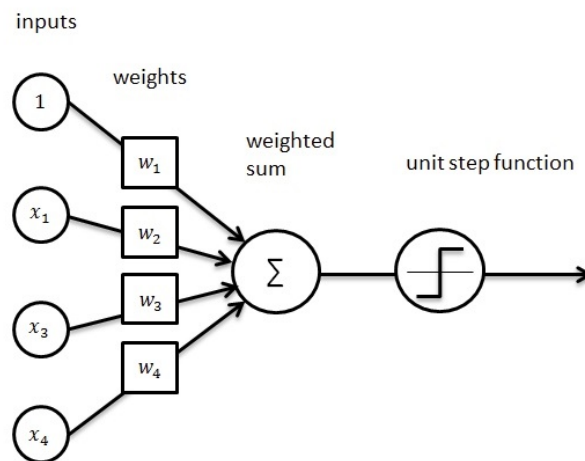


Ilustración 3

- Versión multicapa: Red neuronal con una capa oculta y otra externa de activación no lineal y entrenamiento por *back propagation*.

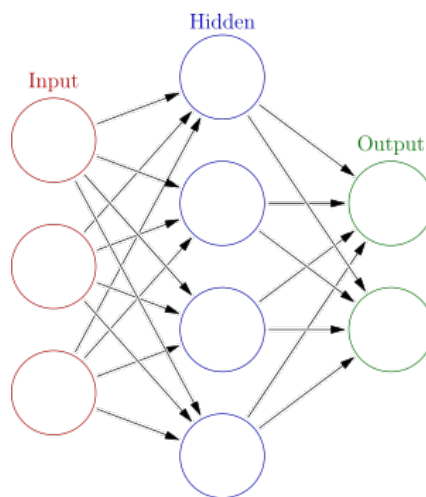


Ilustración 4

1.4 Entrenamiento

El entrenamiento de la red neuronal consiste en variar los parámetros (pesos) de las neuronas en base al error que han generado para intentar disminuirlo. Para la primera versión del perceptrón se ha usado un entrenamiento donde se ajustaban los pesos linealmente. Para la versión multicapa se ha usado back propagation y drop out.

El entrenamiento se ha realizado con el conjunto de datos de entrenamiento (60000 elementos). El objetivo es que la red neuronal aprenda a reconocer dígitos distintos a los del conjunto de entrenamiento sin que se equivoque.

1.5 Test

Una vez se ha entrenado la red neuronal el siguiente paso es comprobar cuán bien reconoce dígitos ajenos al conjunto de entrenamiento. Para ello se ha usado el conjunto de datos de test (10000 elementos) y la red ha intentado predecir qué dígito representa cada imagen. Se han contado el número de aciertos y errores realizados por la red y se ha obtenido un porcentaje de acierto o precisión. El objetivo es crear una red cuyo porcentaje de acierto sea al rededor del 99%.

En las siguientes secciones se va a explicar con detalle el desarrollo e implementación de los algoritmos de las redes neuronales y los resultados obtenidos.

2. Primera versión (Perceptrón)

Esta versión ha sido íntegramente desarrollada por mí salvo las funciones para leer datos de la base de datos de MNIST. El lenguaje de programación usado ha sido java y el entorno de programación usado ha sido Eclipse para Windows 10.

Esta red neuronal consiste en una clase llamada “Entrenador” la cual contiene 10 objetos de la clase “Neurona” y se encarga de leer los datos de MNIST, aportarlos a las neuronas y, en base a las salidas que estas proporcionan, alterar los pesos de las neuronas.

2.1 Entrenamiento

El entrenamiento de la red neuronal es muy simple, consiste en la actualización lineal de los pesos de los perceptrones. El proceso es el siguiente:

- Se realiza la sumatoria de pesos por entradas.
- Si la sumatoria es mayor que el umbral, quiere decir que la neurona se ha activado y viceversa.
- Si el perceptrón ha acertado en su predicción no se hace nada.
- Si el perceptrón se ha activado y no debería, se resta la entrada a los pesos usando una tasa de aprendizaje
- Si el perceptrón no se ha activado y debería, se suma la entrada a los pesos usando una tasa de aprendizaje.

Con este método simple y para la red neuronal descrita anteriormente la fase de entrenamiento dura muy poco y obtiene resultados de error entre 25 y 30 por ciento.

```
Comienza el entrenamiento....
dic 07, 2018 1:17:44 PM practical1.MNISTDatabase readImages
INFORMACIÓN: Reading MNIST data...
dic 07, 2018 1:17:44 PM practical1.MNISTDatabase readImages
INFORMACIÓN: Reading 60000 28x28 images...
dic 07, 2018 1:17:57 PM practical1.MNISTDatabase readImages
INFORMACIÓN: MNIST images read from C:\Users\atr_9\eclipse-workspace\MNIST\data\train-images-idx3-ubyte.gz
dic 07, 2018 1:17:57 PM practical1.MNISTDatabase readLabels
INFORMACIÓN: Reading MNIST labels...
dic 07, 2018 1:17:57 PM practical1.MNISTDatabase readLabels
INFORMACIÓN: MNIST labels read from C:\Users\atr_9\eclipse-workspace\MNIST\data\train-labels-idx1-ubyte.gz
NUMERO DE ERRORES: 15496
PORCENTAJE DE ERROR: 25.826666666666664
El entrenamiento ha tardado 0 segundos
```

Ilustración 5: Resultado del entrenamiento de los perceptrones

En la ilustración anterior se muestra el resultado de una ejecución de la red neuronal donde ha obtenido un porcentaje de error del 25,8% y ha tardado tan poco que muestra que ha tardado 0 segundos.

2.2 Test

Este modelo de red neuronal, como es lógico, obtiene peores resultados para el conjunto de test. Para el caso de la ilustración anterior, en la fase de test se han obtenido los siguientes resultados:

```
Comienza el test....
dic 07, 2018 1:17:57 PM practical1.MNISTDatabase readImages
INFORMACIÓN: Reading MNIST data...
dic 07, 2018 1:17:57 PM practical1.MNISTDatabase readImages
INFORMACIÓN: Reading 10000 28x28 images...
dic 07, 2018 1:17:59 PM practical1.MNISTDatabase readImages
INFORMACIÓN: MNIST images read from C:\Users\atr_9\eclipse-workspace\MNIST\data\t10k-images-idx3-ubyte.gz
dic 07, 2018 1:17:59 PM practical1.MNISTDatabase readLabels
INFORMACIÓN: Reading MNIST labels...
dic 07, 2018 1:17:59 PM practical1.MNISTDatabase readLabels
INFORMACIÓN: MNIST labels read from C:\Users\atr_9\eclipse-workspace\MNIST\data\t10k-labels-idx1-ubyte.gz
NUMERO DE ERRORES: 2801
PORCENTAJE DE ERROR: 28.01
```

Ilustración 6: Resultado del test de los perceptrones

2.3 Resultados

Con este modelo se obtienen unos resultados que, aunque aciertan más del 60% de las veces, son bastante malos. Difieren bastante de los resultados esperados para la asignatura por lo que el siguiente paso va a ser desarrollar una red neuronal multicapa con distintas técnicas de aprendizaje.

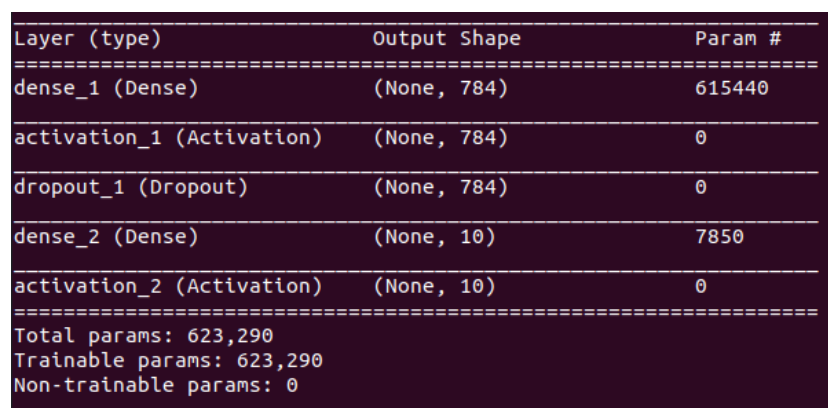
3. Segunda versión

Al principio se intentó desarrollar una versión en java de una red neuronal multicapa pero se quedó en la etapa de back propagation debido al tiempo necesario para su implementación y a que era muy posible que no consiguiera los mismos resultados que usando librerías profesionales.

Esta versión se ha desarrollado en Python usando la librería de keras con tensorflow como backend [3]. La creación del modelo, incluyendo las capas elegidas y el entrenamiento han sido elegidos por mí con ayuda de distintos ejemplos encontrados en internet. Sin embargo todas las funciones usadas son las de la librería de keras y por lo tanto han sido implementadas por terceros. El proyecto se ha desarrollado en Ubuntu 18.04 y no hace uso de la GPU por lo que los tiempo de ejecución son más elevados que si se le diera uso.

El proyecto consiste en un script de python que usa el modelo *Sequential* de keras el cual permite añadirle distintos tipos de capas ordenadamente. Se han añadido dos capas al modelo:

- Capa oculta: Capa de 784 neuronas las cuales tienen 784 entradas y una salida por neurona con activación “relu”.
- Capa externa: Capa de 10 neuronas con 784 entradas cada una y una salida de activación “softmax”.



Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 784)	615440
activation_1 (Activation)	(None, 784)	0
dropout_1 (Dropout)	(None, 784)	0
dense_2 (Dense)	(None, 10)	7850
activation_2 (Activation)	(None, 10)	0
Total params: 623,290		
Trainable params: 623,290		
Non-trainable params: 0		

Ilustración 7: Estructura de la red neuronal

Al principio se usó un modelo mucho mas complejo que incluía varias capas convolutivas y max pooling pero el tiempo de cómputo necesario era tan alto que se descartó.

3.1 Preprocesamiento

En todos los modelos desarrollados en esta versión se ha usado la normalización de las imágenes, es decir se han pasado de valores de [0,255] a [0,1].

Dependiendo de si se querían dibujar gráficas o números o si se querían procesar los datos se han usado distintos formatos del conjunto de datos:

- (60000,28,28,1)
- (60000,784)

Para la versión mas compleja se había usado un generador de imágenes que realizaba variaciones aleatorias a estas. El siguiente código ilustra las variaciones que realizaba este generador.

```
datagen = ImageDataGenerator(  
    featurewise_center=False, # set input mean to 0 over the dataset  
    samplewise_center=False, # set each sample mean to 0  
    featurewise_std_normalization=False, # divide inputs by std of the dataset  
    samplewise_std_normalization=False, # divide each input by its std  
    zca_whitening=False, # apply ZCA whitening  
    rotation_range=15, # randomly rotate images in the range (degrees, 0 to 180)  
    zoom_range = 0.1, # Randomly zoom image  
    width_shift_range=0.1, # randomly shift images horizontally (fraction of total width)  
    height_shift_range=0.1, # randomly shift images vertically (fraction of total height)  
    horizontal_flip=False, # randomly flip images  
    vertical_flip=False) # randomly flip images
```

El objetivo de usar este generador era que el conjunto de entrenamiento fuera más variado y que la red neuronal aprendiera a generalizar más y no hubiera sobre aprendizaje en este conjunto.

3.2 Entrenamiento

El entrenamiento usado por la red es *back propagation*, este entrenamiento varía los pesos de las neuronas usando el error cometido por la capa externa y su propagación hacia capas internas [4]. En la siguiente ilustración se puede ver la fórmula usada para calcular las variaciones de pesos para cada capa.

$$\Delta w_{ij}^k = \begin{cases} -\eta x_i^k f'(z_j^k) (y_j - t_j) & \text{para las neuronas de salida,} \\ -\eta x_i^k f'(z_j^k) \left(\sum_p \delta_p^{k+1} w_{pj}^{k+1} \right) & \text{para las neuronas ocultas.} \end{cases}$$

Ilustración 8

También se ha usado la técnica de *drop out* que consiste en ignorar ciertas neuronas aleatoriamente en alguna etapa de forward or back propagation con el objetivo de evitar que la red neuronal sobre aprenda, es decir, que aprenda errores.

Se ha hecho un entrenamiento por lotes de tamaño 128 y se han hecho 20 epochs, es decir, se ha recorrido el conjunto de entrenamiento 20 veces. Con el modelo explicado el tiempo de entrenamiento de la red neuronal ronda los 120 segundos.

```
Epoch 3/20
- 6s - loss: 0.0742 - acc: 0.9781 - val_loss: 0.0734 - val_acc: 0.9781
Epoch 4/20
- 6s - loss: 0.0548 - acc: 0.9834 - val_loss: 0.0683 - val_acc: 0.9786
Epoch 5/20
- 6s - loss: 0.0431 - acc: 0.9866 - val_loss: 0.0738 - val_acc: 0.9774
Epoch 6/20
- 6s - loss: 0.0340 - acc: 0.9899 - val_loss: 0.0616 - val_acc: 0.9816
Epoch 7/20
- 6s - loss: 0.0277 - acc: 0.9915 - val_loss: 0.0561 - val_acc: 0.9833
Epoch 8/20
- 6s - loss: 0.0214 - acc: 0.9936 - val_loss: 0.0627 - val_acc: 0.9810
Epoch 9/20
- 6s - loss: 0.0205 - acc: 0.9936 - val_loss: 0.0659 - val_acc: 0.9806
Epoch 10/20
- 6s - loss: 0.0166 - acc: 0.9949 - val_loss: 0.0587 - val_acc: 0.9831
Epoch 11/20
- 6s - loss: 0.0152 - acc: 0.9954 - val_loss: 0.0639 - val_acc: 0.9811
Epoch 12/20
- 6s - loss: 0.0144 - acc: 0.9951 - val_loss: 0.0582 - val_acc: 0.9830
Epoch 13/20
- 6s - loss: 0.0135 - acc: 0.9956 - val_loss: 0.0668 - val_acc: 0.9817
Epoch 14/20
- 6s - loss: 0.0121 - acc: 0.9960 - val_loss: 0.0675 - val_acc: 0.9816
Epoch 15/20
- 6s - loss: 0.0091 - acc: 0.9972 - val_loss: 0.0734 - val_acc: 0.9808
Epoch 16/20
- 6s - loss: 0.0096 - acc: 0.9971 - val_loss: 0.0698 - val_acc: 0.9834
Epoch 17/20
- 6s - loss: 0.0091 - acc: 0.9970 - val_loss: 0.0770 - val_acc: 0.9813
Epoch 18/20
- 6s - loss: 0.0100 - acc: 0.9966 - val_loss: 0.0778 - val_acc: 0.9805
Epoch 19/20
- 6s - loss: 0.0089 - acc: 0.9972 - val_loss: 0.0653 - val_acc: 0.9838
Epoch 20/20
- 6s - loss: 0.0066 - acc: 0.9977 - val_loss: 0.0647 - val_acc: 0.9844
Tiempo de entrenamiento: 117
```

Ilustración 9: Resultado del entrenamiento de la red neuronal

Como se puede ver en la ilustración anterior, se han tardado 117 segundos en entrenar la red neuronal y se ha obtenido un porcentaje de acierto del 99,77%.

Durante la implementación del proyecto se incluyó la funcionalidad para almacenar y leer los datos del modelo como:

- Estructura: Capas de la red neuronal, activación de neuronas y técnicas de entrenamiento.
- Pesos: Los pesos de las neuronas.
- Historia: Valores obtenidos durante la fase de entrenamiento como porcentaje de errores.

Así, si se desea una ejecución rápida del proyecto basta con seleccionar en el código que no se quiere que aprenda la red y esta cargará los datos de la última ejecución.

3.3 Test

Este es un resultado de la comprobación del entrenamiento de la red neuronal sobre el conjunto de test:

Numero de errores: 156. Porcentaje de error = 1.56%.

Ilustración 10

Se puede ver que el resultado es menor que el 1,8 % que es el resultado que se indica en el guión de esta práctica para la red neuronal más compleja.

3.4 Resultados

El proyecto, al igual que se le puede indicar si se desea que aprenda o no, se le puede indicar que muestre gráficas e imágenes para mostrar los resultados. Así, cuando se le indica, la primera imagen mostrada corresponde a los 40 primeros dígitos del conjunto de datos de entrenamiento, simplemente con el fin de ilustrar los elementos con los que va a trabajar la red.

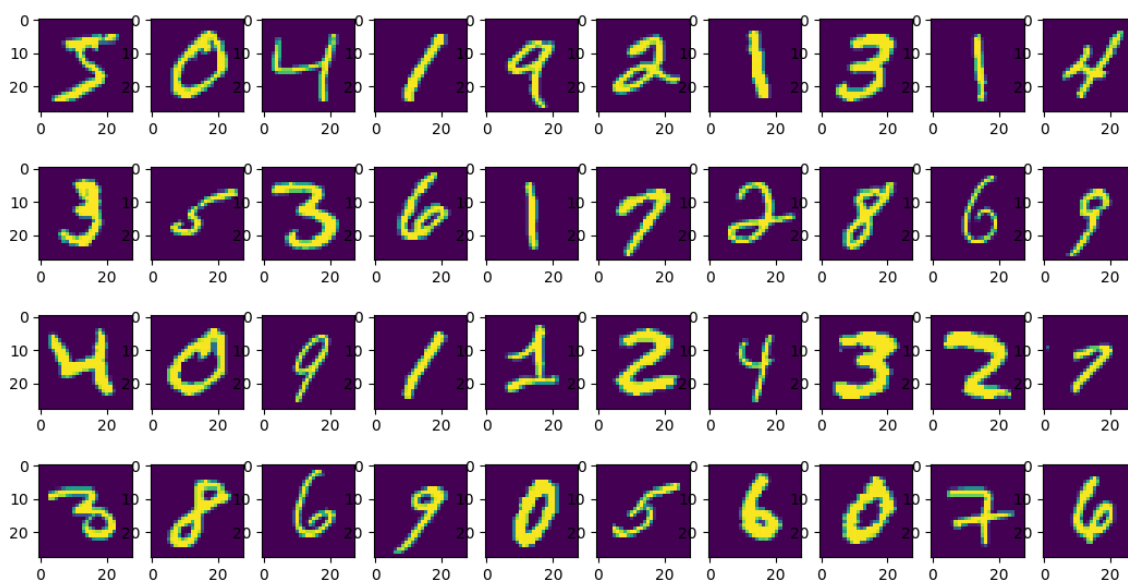


Ilustración 11: Los cuarenta primeros dígitos del conjunto de entrenamiento

Una vez realizado el entrenamiento, el primer resultado es la matriz de confusión de las predicciones. Esta matriz se imprime por consola, el eje y (vertical) representa las clases reales y el eje x (horizontal) representa las clases predichas.

[[968	0	1	0	1	2	5	1	1	1]
[0	1126	3	1	0	0	2	0	3	0]	
[0	2	1019	1	1	0	2	4	2	1]	
[0	0	1	993	0	5	0	3	1	7]	
[0	1	3	0	964	0	4	1	0	9]	
[1	0	0	5	1	878	4	0	1	2]	
[2	2	2	1	2	3	946	0	0	0]	
[0	4	9	1	0	0	0	1008	2	4]	
[1	0	5	3	3	1	1	3	951	6]	
[1	2	0	3	5	4	0	2	1	991]]	

Ilustración 12: Matriz de confusión

Una matriz de confusión de un buen algoritmo de clasificación debe mostrar en su diagonal números elevados y en el resto de elementos deben aparecer ceros o valores muy bajos. De la anterior matriz se interpreta que el algoritmo ha obtenido buenos resultado debido a los valores altos de su diagonal principal. Por otro lado, se puede ver que por ejemplo para el dígito 0 (primer elemento del eje x) se han acertado 968 veces y que se ha dicho 5 veces (séptima columna) que se trataba de un 6.

A continuación se muestra la precisión del algoritmo durante su etapa de aprendizaje en comparación con la precisión en la etapa de test. Se dibujan dos gráficas, la primera muestra la precisión y la segunda la pérdida de la red neuronal:

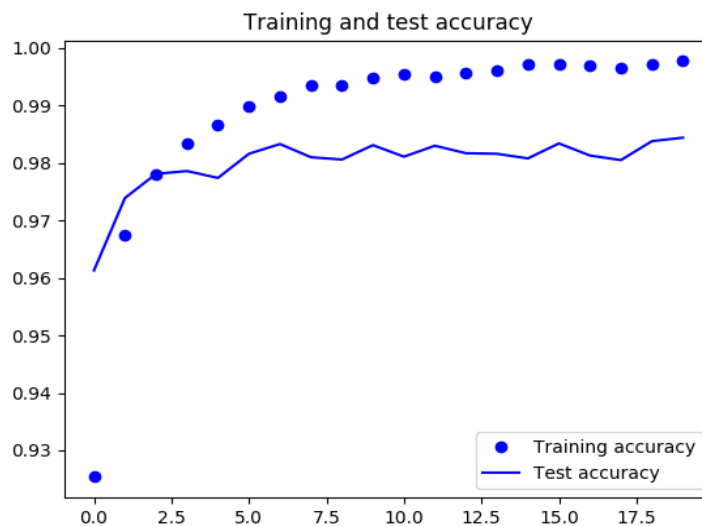


Ilustración 13: Precisión de la red neuronal

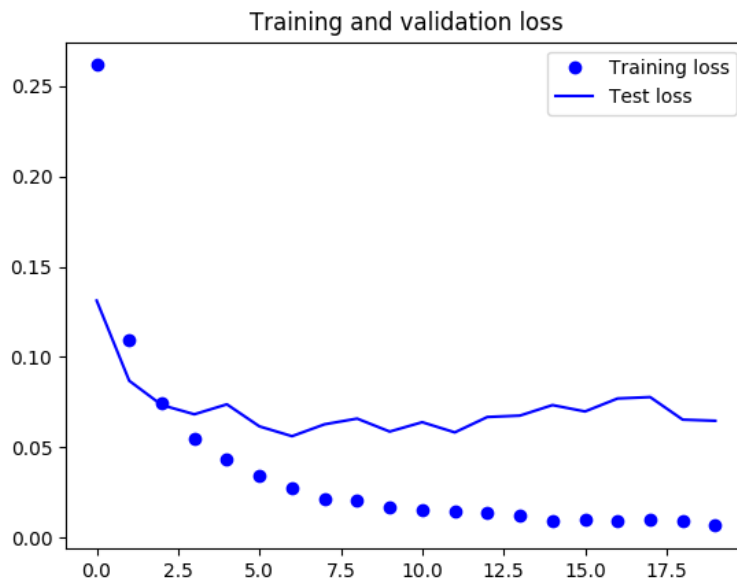


Ilustración 14: Pérdida o errores de la red neuronal

En las anteriores gráficas se puede ver cómo el algoritmo mejora a la precisión a medida que realiza la etapa de aprendizaje. La precisión en el conjunto de entrenamiento llega a ser casi del 100% lo que es preferible ya que un 100% de acierto implicaría sobreaprendizaje.

Por último, el programa hace una recopilación del top 6 errores donde ha cometido la mayor equivocación. Es decir, donde las mayores probabilidades de que un dígito sea de una clase y sin embargo no lo sea.

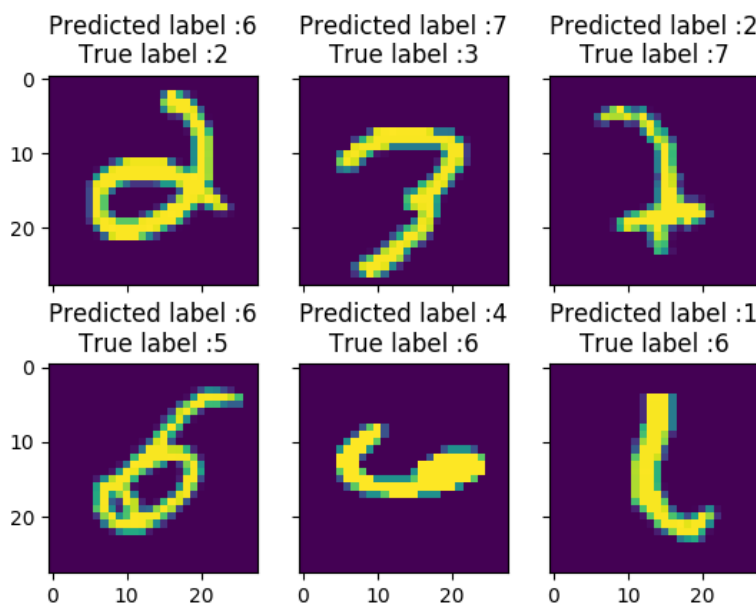


Ilustración 15: Top 6 errores

4. Conclusión

En esta práctica se han realizado dos modelos de redes neuronales. Si bien el modelo que ha obtenido mejores resultados ha sido gracias al uso de librerías externas, hay que tener en cuenta la implementación de los otros modelos ya que ha sido muy útil para comprender el funcionamiento de las redes neuronales.

Finalmente el mejor resultado obtenido ha sido un porcentaje de error del 1,56%. Este resultado es mejor que el mínimo esperado (1,8%) que se indicaba en el guión de la práctica.

5. Bibliografía

[1] LeCunn, Y., Cortes, C. and J.C. Burges, C. (n.d.). MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges. [online] Yann.lecun.com. Available at: <http://yann.lecun.com/exdb/mnist/> [Accessed 7 Dec. 2018].

[2] Wikipedia contributors. (2018, November 21). MNIST database. In Wikipedia, The Free Encyclopedia. Retrieved 11:04, December 7, 2018, from https://en.wikipedia.org/w/index.php?title=MNIST_database&oldid=869917056

[3] Keras.io. (2018). Home - Keras Documentation. [online] Available at: <https://keras.io/> [Accessed 7 Dec. 2018].

[4] Berzal, F. (2018). Redes Neuronales & Deep Learning. 1st ed. Granada.