RECONOCIMIENTO ÓPTICO DE CARACTERES MNIST

UNIVERSIDAD DE GRANADA

INTELIGENCIA COMPUTACIONAL

PRÁCTICA DE REDES NEURONALES

CURSO 2018/2019

SERGIO SAMANIEGO MARTÍNEZ

MÁSTER EN INGENIERÍA INFORMÁTICA

Contenido

1. INTRODUCCIÓN

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales.

En este caso vamos a usar uno de los problemas más abordados para las redes neuronales como es el reconocimiento de dígitos manuscritos de la base de datos MNIST.

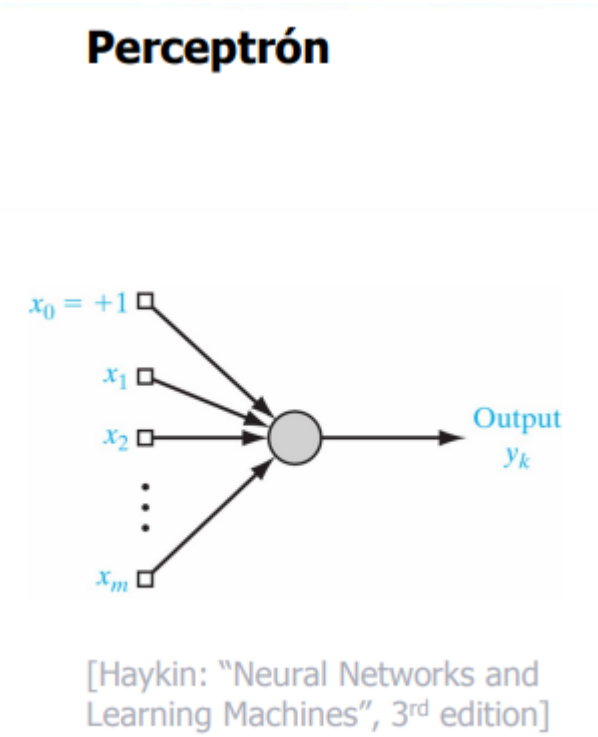
Las imágenes de los dígitos manuscritos, son imágenes de 28x28 píxeles en los que cada píxel contendrá los valores de color correspondientes, en este caso de 0 a 255 (ya que únicamente tenemos blanco y negro).

La principal tarea es entrenar una red neuronal que haya sido diseñada por nosotros mismos y que sea capaz de tener una tasa de acierto relativamente buena.

2. IMPLEMENTACIÓN

2.1 PRIMERA APROXIMACIÓN

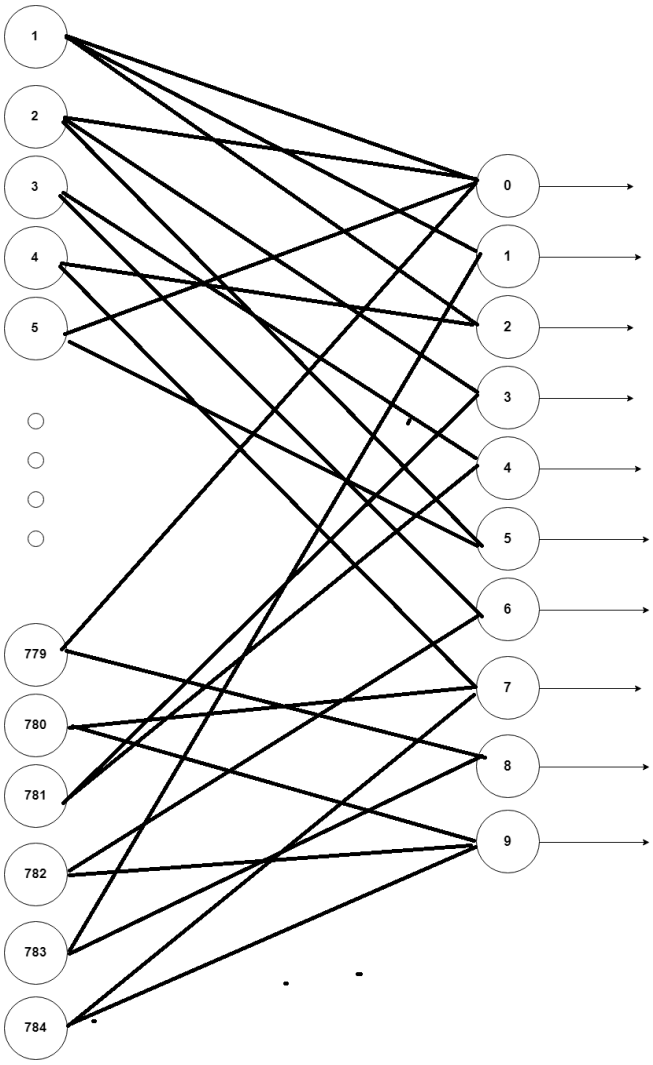
La primera aproximación que se abordó fue comenzar implementando un algoritmo básico como era el uso de perceptrones para el reconocimiento de los dígitos.



El perceptrón es una neurona donde recibe las entradas, utilizan su función de activación y proporcionan una salida.

Lo que haremos será utilizar un perceptrón para cada clase, en nuestro caso tendremos 10 perceptrones que corresponderán con cada uno de los números, de 0 a 9.

Por lo tanto, nuestra red neuronal quedaría de una forma parecida a la siguiente.



Donde cada píxel de la imagen estaría conectado a todos los perceptrones y cada perceptrón estaría encargado de clasificar cada uno de los números.

Con esta red neuronal, la función de activación era binaria, de forma que si la salida era mayor que cero la neurona estaba activada y en caso contrario no.

Además la salida de cada neurona venía dada por la multiplicación de los pesos por la entrada.

$$z = \sum_i x_i w_i$$

$$y = \begin{cases} 1 & si\ z \geq 0 \\ 0 & en\ otro\ caso \end{cases}$$

Una vez realizada la función de activación, comprobábamos si estaba clasificado correctamente el número comparándolo con las etiquetas que se nos ofrecen para cada imagen. Cuando realizamos la comprobación ajustamos los pesos en función de la predicción hecha por nuestro perceptrón.

La actualización de los pesos se realizaba siguiendo las siguientes reglas:

- Si el perceptrón se activaba y no debería haberlo hecho: Le restábamos el valor de la entrada a esos pesos.
- Si el perceptrón no se activaba y debería haberlo hecho: Le sumamos el valor de la entrada a los pesos.

Nos quedaría algo como el siguiente código:

```
if(c.activada) {
    if(target != c.nombre ) {
        for(int i=0; i<(28*28); i++) {
            c.weight[i] = c.weight[i] + (c.input[i]*(-1));
        }
    }
}
else {
    if(c.nombre == target) {
        for(int i=0; i<(28*28); i++) {
            c.weight[i] = c.weight[i] + (c.input[i]*(1));
        }
    }
}
```

2.2 MODELO SECUENCIAL

En un principio, la idea era realizar las redes neuronales por mi propia cuenta, sin ayuda de librerías externas, para de esta forma aprender cómo funcionan las redes neuronales y entenderlas lo mejor posible.

Pero debido a las limitaciones de tiempo, no me ha sido posible llegar a una implementación funcional de las mismas, así que tuve que recurrir finalmente al uso de estas librerías.

En esta parte se hace un cambio de lenguaje y se comienza a usar Python, debido a que ya había realizado prácticas anteriormente de redes neuronales con este mismo lenguaje y podían servirme de ayuda para la realización de esta práctica.

Para el uso de las redes, se hace uso de la librería de Redes Neuronales que tiene Python como es **Keras**.

Como ya se comentó en la introducción, se va a abordar el problema del reconocimiento de dígitos manuscritos, el cual es un problema muy famoso y disponemos de los datos en el mismo Keras.

```
from keras.datasets import mnist

mnist.load_data()
```

Lo primero que vamos a hacer es cargar estos datos en en variables para poder utilizarlas con nuestro modelo.

```
X_train tamaño, dimensiones (60000, 28, 28)
y_train tamaño, dimensiones (60000,)
X_test tamaño, dimensiones (10000, 28, 28)
y_test tamaño, dimensiones (10000,)
```

Nuestras variables referentes a la X tendrán las imágenes del train o test y las referentes a las Y las etiquetas con los valores correctos de cada imagen.

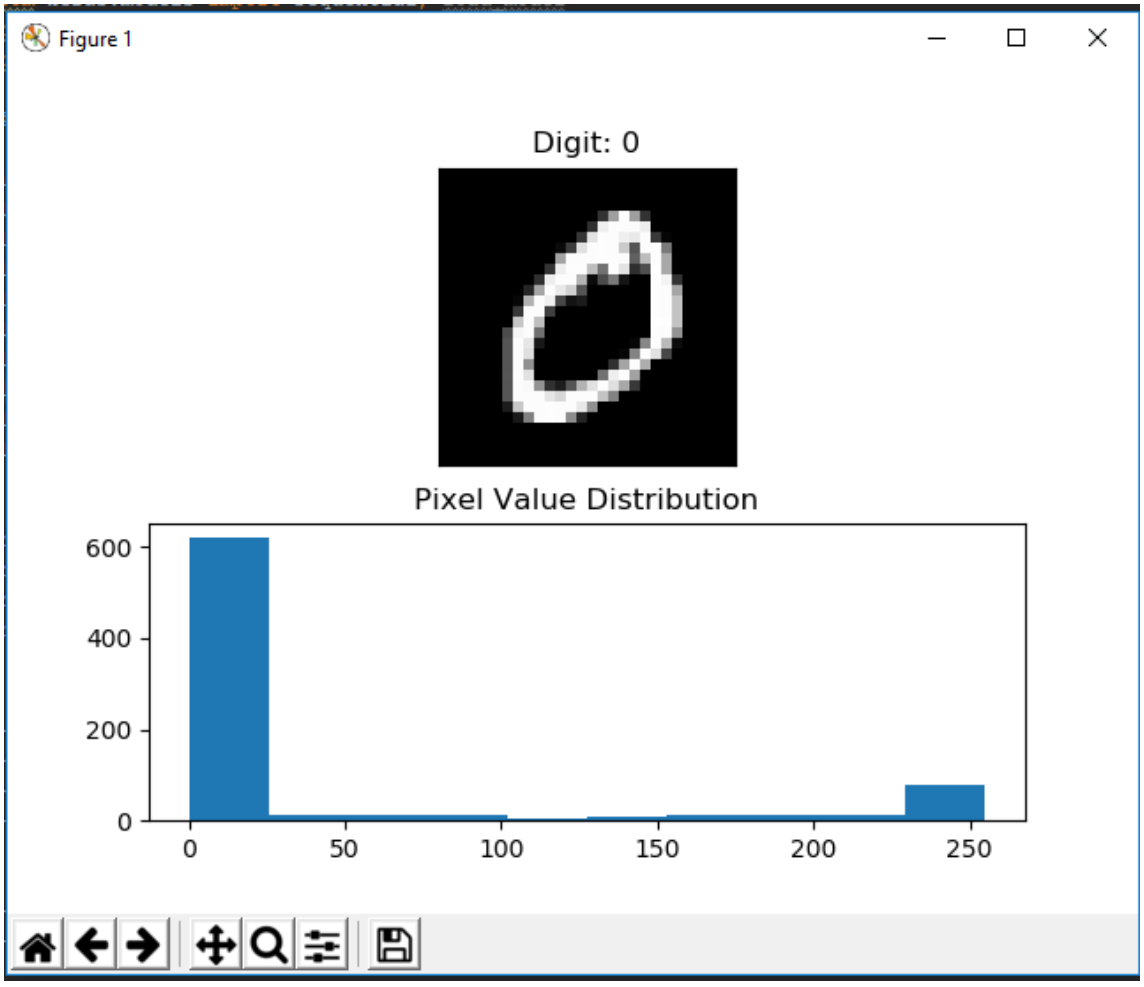
Por lo tanto, en la imagen anterior podemos observar que tenemos 60000 datos para el entrenamiento, donde cada imagen viene representada por una matriz de 28x28 y para el test tendremos 10000 datos.

2.2.1 PREPROCESAMENTO

En este apartado vamos a ver las transformaciones que haremos sobre los datos para mejorar el entrenamiento y el aprendizaje de nuestra red neuronal.

Lo primero que haré será transformar las variables que contienen las imágenes, que como vimos están almacenadas como matrices de 28x28, en vectores de una única dimensión. Esto facilitará el entrenamiento al modelo y reducirá el tiempo de ejecución del mismo. Teniendo los datos como matrices, cada "epoch" nos llegaba a tardar unos 100 segundos, pero al transformar los datos a vectores de una dimensión, cada "epoch" se nos llega a reducir a unos 10 segundos, más o menos.

Una vez se transforman los datos a vectores de una dimensión, se van a normalizar los datos.



En esta imagen se ven los valores que tienen los píxeles de la imagen. Como es lógico, ya que estamos hablando de una imagen tendrán un rango de 0 a 255 y únicamente una dimensión, ya que trabajamos con imágenes blanco y negro y no RGB.

Por lo tanto, para normalizar los datos, lo que haremos será dividir por el máximo y de esta forma mantendremos los valores de los píxeles entre 0-1 y no entre 0-255.

```
# Normalizamos datos
X_train /= 255
X_test /= 255
```

Por último, se transformarán los valores de las etiquetas que nos indican los valores de cada imagen.

Ahora mismo, esta variable es un vector donde cada elemento tendrá un valor comprendido entre 0 y 9.

Como esto es lo que queremos clasificar realmente, podemos diferenciar 10 clases diferentes, donde cada clase se corresponderá a un número diferente.

Por lo tanto, lo que haremos será transforma cada etiqueta en un vector binario, donde habrá únicamente un número 1, en la posición del vector que se corresponda con la etiqueta.

```
Etiqueta de un número antes de transformar:  5
Etiqueta de un número antes de transformar:  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Con esta transformación se nos facilitará la clasificación de cada clase teniendo una clase para cada dígito correspondiente.

2.2.2 CONSTRUCCIÓN DEL MODELO

Vamos a comenzar a construir nuestro modelo.

Para ello usaremos el modelo Secuencial de Keras, ya que es sencillo de utilizar, nos permite agregar capas fácilmente y lo entiendo mucho mejor que si por ejemplo utilizara redes convolutivas, que no conseguía configurar correctamente.

Para construir el modelo, tenemos el código que se ve a continuación:

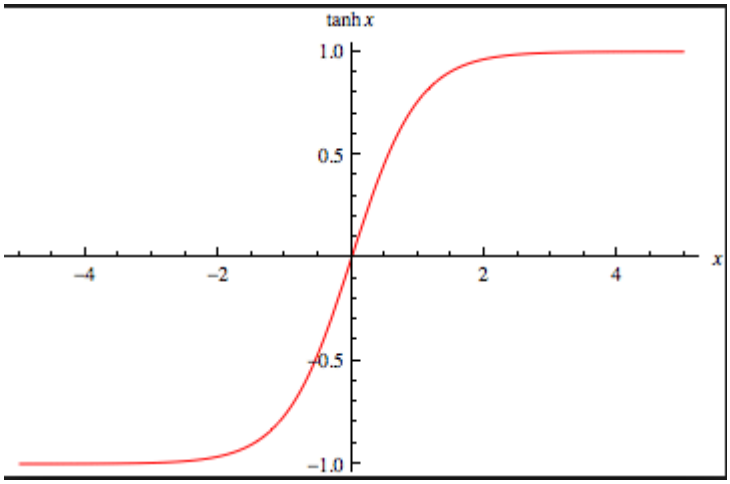
```
# Construimos el modelo
model = Sequential()
model.add(Dense(784, input_shape=(784,)))
model.add(Activation('tanh'))
model.add(Dropout(0.2))

model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))


model.add(Dense(10))
model.add(Activation('softmax'))
```

La primera capa tendrá un tamaño de 784 neuronas, y al ser la primera debemos indicarle también el tamaño de entrada con el parámetro “input_shape” que también será de 784.

Se ha utilizado como activación la **tangente hiperbólica** ya que conseguía muy buenos resultados comparándola con otros métodos de activación.

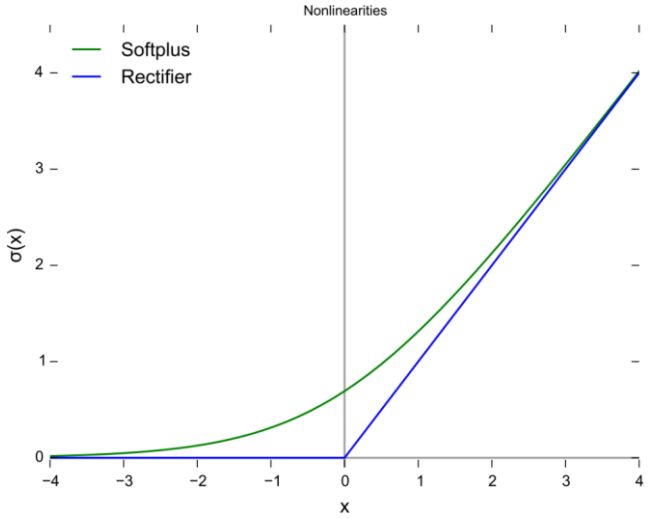


Además, a la capa le añadiremos un “dropout” que nos ayudará a prevenir lo que conocemos como sobreaprendizaje.

En la siguiente capa, ya no es necesario indicar el tamaño de entrada y únicamente indicaremos el tamaño de la capa, que en este caso será de 512.

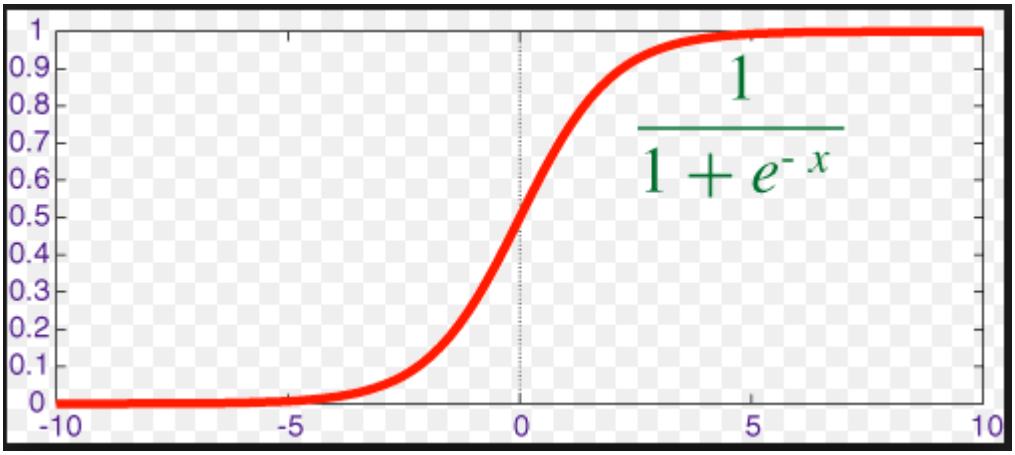
Para la activación se ha utilizado otra función de activación para no utilizar la misma y de esta forma ayudar a que únicamente se activen las neuronas cuando correspondan.

En esta capa se ha usado la función de activación “Rectifier Linear Unit” que es una función lineal que comienza a crecer a partir de los valores positivos, como vemos en esta imagen.



Además, también se le incluirá también un “dropout”.

Finalmente, tendremos la última capa de salida que contará con 10 neuronas, una correspondiente a cada clase, es decir, a cada dígito y como función de activación en estas neuronas se usará “softmax”, ya que es una función “estándar” para los algoritmos de clasificación multi-etiqueta, como es este problema.



Construido el modelo, pasaremos indicar el proceso de aprendizaje que tendrá nuestra red.

Indicaremos:

- Función objetivo: Usaremos la función “categorical_crossentropy”. Essta función medirá el rendimiento del modelo dando un valor entre 0 y 1. El valor irá aumentando a medida que el modelo se equivoca de la etiqueta predicha a la etiqueta real.
- Metricas: Que nos permitirá juzgar la actuación de nuestro modelo.
- Optimizador: En este caso usaremos el optimizador Adam, ya que era el que mejores resultados nos daba.

```
# Compilamos el modelo
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='Adam')
```

Por último, entrenaremos el modelo.

En este caso le indicaremos el “batch_size” que tras probar con varios valores, finalmente me decidí por este, ya que no es demasiado pequeño que haría que el modelo tardara demasiado, ni tampoco demasiado grande que haría que diera muchos saltos erróneos que no convergieran reduciendo el error.

El número de “epoch” será 20, donde en cada “epoch” se hará el backpropagation para modificar los valores de pesos de las neuronas y los datos de validación.

```
# Entrenamos el modelo
history = model.fit(X_train, Y_train,
          batch_size=128, epochs=20,
          verbose=2,
          validation_data=(X_test, Y_test))
```

Hecho esto ya tendremos nuestro modelo entrenado y listo para predecir las clases del conjunto de test.

3. RESULTADOS

3.1 PRIMERA APROXIMACIÓN (PERCEPTRÓN)

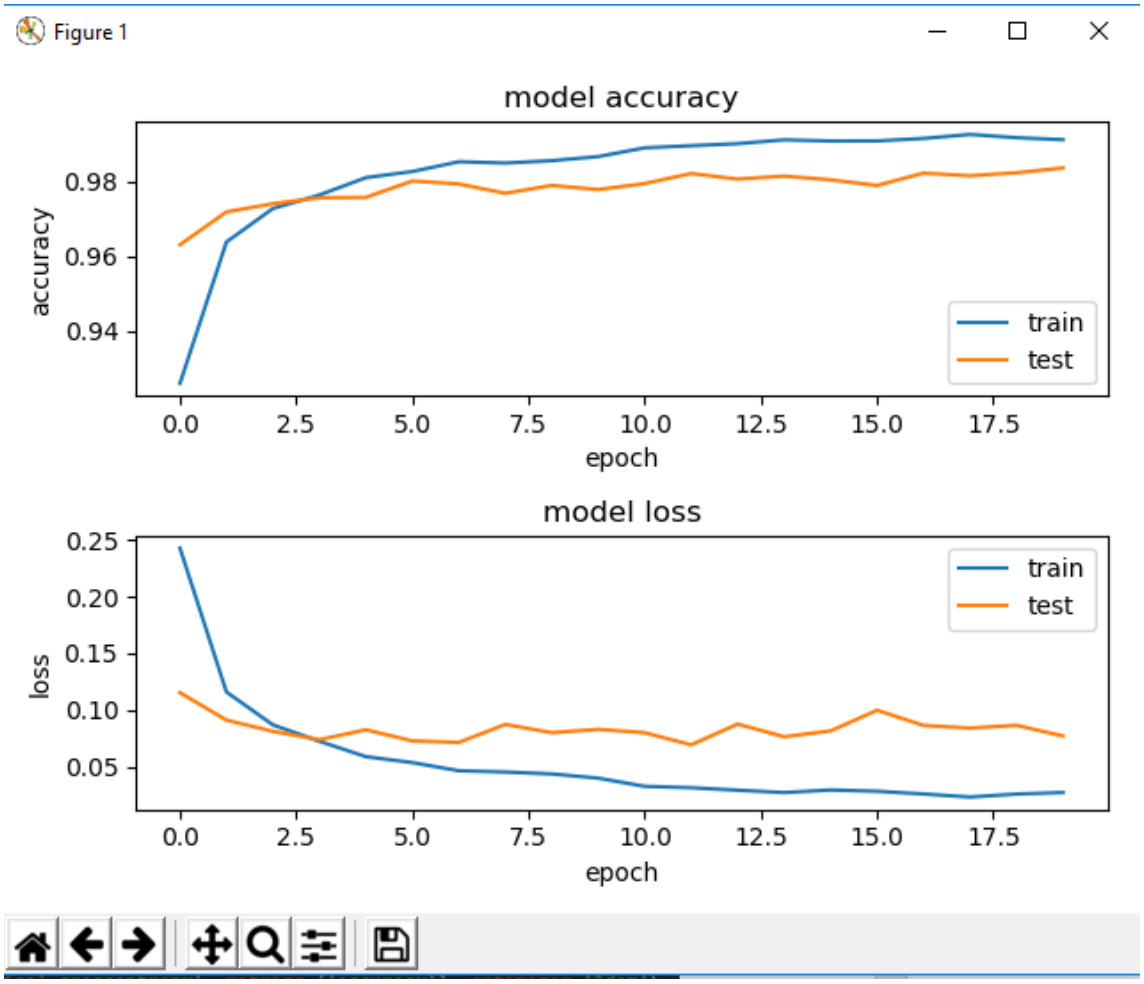
Los resultados obtenidos con la red neuronal realizada con perceptrones, nos daba un error de alrededor del 20-25%, que es algo normal tratándose de una red neuronal tan sencilla como la que se ha realizado.

```
Comenzando el test .....
El número de errores encontrados es de: 2416
El número de aciertos es de: 7584
La tasa de aciertos de esta red neuronal es del: 75.84%
La tasa de errores de esta red neuronal es del: 24.16%
```

Realmente, para ser una red tan sencilla, los resultados obtenidos no son del todo malos. De 10000 resultados se equivoca en unos 2000 resultados, lo cual, no nos sirve para hacer una clasificación fiable, pero para una primera aproximación nos sirve.

3.2 MODELO SECUENCIAL

Una vez que se ha entrenado el modelo, vamos a ver su actuación, evaluando el modelo y a prediciendo las clases.



En estas imágenes podemos ver cómo el modelo está aprendiendo correctamente gracias al algoritmo implementado ya que vemos que el acierto va aumentando conforme avanzan las iteraciones del modelo y los valores incorrectos van disminuyendo.

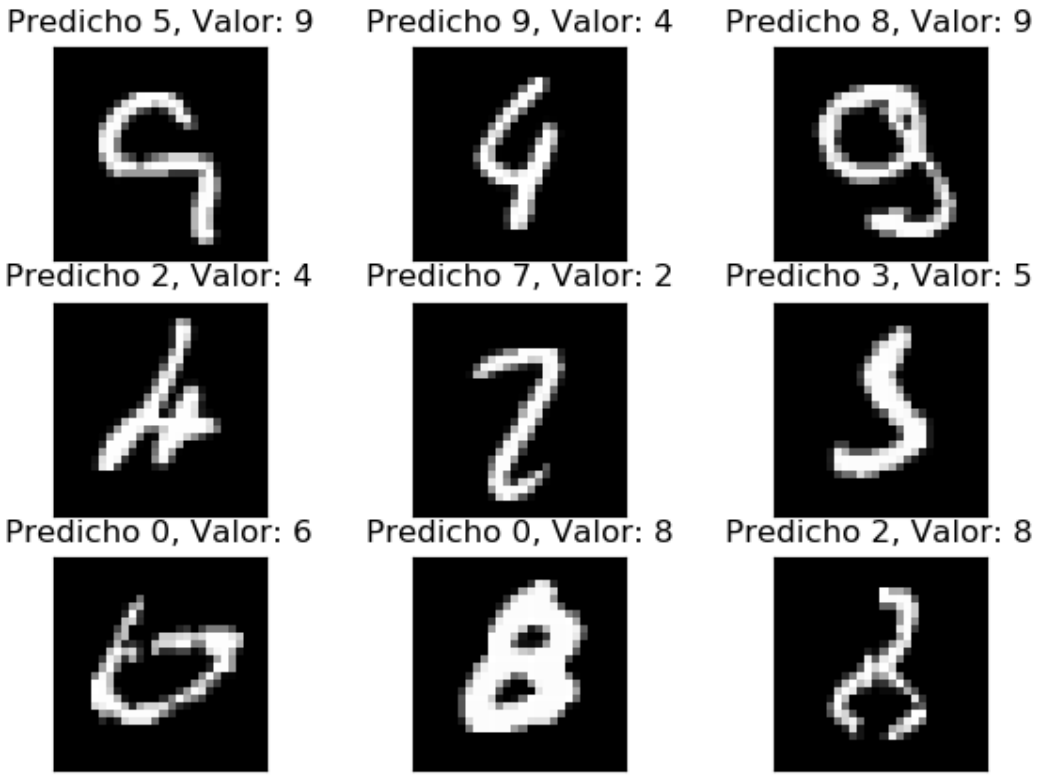
Esta imagen nos ilustra de forma muy sencilla y visual cómo el modelo actúa correctamente.

```
Test Loss 0.07723034290659 78
Test Accuracy 0.9836
```

En esta imagen se puede observar el acierto de nuestro modelo, así como la función objetivo que vemos que realmente no es nada mala.

Por lo tanto, vemos que se llega a obtener un 95,75% de acierto sobre el conjunto de test, un resultado que no es nada malo, teniendo en cuenta que se han usado redes lineales en lugar de redes convolutivas.

Además podemos observar algunos de los ejemplos en los que hemos fallado.



4.CONCLUSIONES

El resultado global puede ser aún mejorado ya que nos hemos quedado en un 95,75% de acierto. Esto puede ser aún disminuido haciendo uso de otro tipo de redes, como es el caso de las redes convolutivas.

Realmente, para estar utilizando un modelo de redes lineales, bastante sencillo de construir se obtiene un resultado muy aceptable ya que, de 10000 imágenes testeadas, se ha fallado únicamente en 425.

```
9836  clasificados correctamente
164  erróneos
```

Haciendo uso de esta misma topología de red neuronal, se puede también seguir investigando con distintos optimizadores, distintas funciones de activación y seguramente se podría mejorar un poco más el resultado, pero obviamente, nunca podríamos llegar al porcentaje de acierto que se puede llegar a conseguir haciendo uso de redes convolutivas.

Además, no he podido configurar correctamente tensorflow para que haga uso de la GPU en lugar de la CPU y por lo tanto si complico el modelo más, el tiempo de ejecución sería mucho mayor, es decir, con la GPU se podría haber investigado un poco modelos algo más complejos sin que aumentara demasiado el tiempo de ejecución del algoritmo.

5. BIBLIOGRAFÍA

[Perceptrones – Fernando Berzal]

https://elvex.ugr.es/decsai/deep-learning/slides/NN2%20Perceptron.pdf

[Redes convolutivas – Fernando Berzal]

https://elvex.ugr.es/decsai/deep-learning/slides/NN7%20CNN.pdf

[Keras]

https://keras.io/getting-started/sequential-model-guide/

[Plot con Python]

https://matplotlib.org/users/pyplot_tutorial.html