Universidad de Granada

Master Profesional en Ingeniería Informática

Práctica 1

MNIST

Autor:
Manuel Jesús García Manday
(nickter@correo.ugr.es)

Master en Ingeniería Informática

Índice

1. Introducción.

Para la resolución de esta práctica se han empleado tres tipos de modelos diferentes de redes neuronales artificiales para intentar resolver el problema de reconocimiento de dígitos manuscritos que proporciona la base de datos MNIST. Para cada modelo de red se han realizado varias pruebas, variando los parámetros de cada uno de ellos en función de los resultados obtenidos.

En siguientes apartados se detallará todo el proceso realizado así como los diferentes resultados conseguidos con cada prueba.

2. Entorno de trabajo.

Se ha definido un entorno de trabajo donde se desarrollarán las implementaciones de los diferentes modelos redes así como sus pruebas. Dicho entorno está basado en **Python** [1] en su versión **3.5**. Para la implementación de los modelos de redes se va emplear la librería **Keras** [2] en su versión **2.0.2**, la cual a su vez se apoya en la versión **1.0.1** de la librería **Tensorflow** [3]. Mediante esta API definiremos los tres modelos de redes neuronales que serán entrenados y testeados con sus respectivos conjunto de datos.

Keras es una API de alto nivel para la creación de redes neuronales escrita en Python que se ejecuta por encima de otras librerías como la mencionada Tensorflow, **CNTK** [4] o **Theano** [5]. Soporta ambos tipos de redes neuronales, convolutivas y recurrentes además de su combinación. Se puede ejecutar tanto en CPU como en GPU.

Tensorflow es una librería de código abierto usada para computación numérica usando gráficos de flujos de datos. Posee una arquitectura flexible que permite realizar aplicaciones de cómputo en una o mas CPUs o GPUs en un ordenador personal, servidor o dispositivo móvil.

CNTK (Computational Network Toolkit) actualmente conocido como **The Microsoft Cognitive Toolkit** es el motor de intelegencia artificial de Microsoft a través del cual se proporcionan librerías para trabajar con redes neuronales.

Theano es una librería escrita en Python que permite definir, optimizar y evaluar expresiones matemáticas envueltas en arrays multidimensionales de manera eficiente. Entre sus propiedades se puede destacar la fuerte integración que tiene con la librería **NumPy** [6], el uso transparente de la GPU, etc.

3. Modelos de redes neuronales

Como se ha mencionado anteriormente, son tres los diferentes tipos de modelo de red neuronal los que se han implementado. Una red neuronal simple con una capa de entrada y una capa de salida, una red neuroal multicapa con una capa oculta, y una red neuronal convolutiva con diferentes niveles de capas. A estos tipos de modelo de red neuronal se le han ido aplicando variaciones en sus parámetros para ajustar los resultados que se han ido obteniendo como se demostrará en el actual apartado.

Se ha definido un script base en python en el que se apoyarán todas las implementaciones y pruebas de los diferentes modelos de red. Este script obtiene los conjuntos de imágenes y etiquetas de MNIST, tanto de entrenamiento como de prueba. Las imágenes obtenidas, que tienen sus valores de escala de grises representados en estructuras de array de dos dimensiones, son convertidos a un array de una sola dimensión. Estos valores a su vez son normalizados de 0 a 1. Del mismo modo las etiquetas son también obtenidas y tratadas para el

[1] Disponible en el sitio web de Python (https://www.python.org)
[2] Disponible en el sitiio web de Keras (https://keras.io)
[3] Disponible en el sitio web de Tensorflow (https://www.tensorflow.org)
[4] Disponible en el girhub de CNTK (https://github.com/Microsoft/cntk)
[5] Disponible en el sitio web de Theano (http://www.deeplearning.net/software/theano/)
[6] Disponible en el sitio web de NumPy (http://www.numpy.org)

entrenamiento y testeo de la red neuronal.

```
# load (downloaded if needed) the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# convert 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')

# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255

# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Figura 1: Descarga de imágenes de MNIST y paso a array de una dimensión.

El primer modelo de red neuronal implementado es un modelo muy simple que consta de una capa de entrada y una capa de salida. La capa de entrada esta formada por 784 neuronas, el equivalente al número de pixeles de una imagen (28x28), las neuronas de una capa de entrada no son de procesamiento a diferencia del resto. En la capa de salida se definen 10 neuronas, el equivalente al número de clases que una imagen puede tomar, las cuales tienen una función de activación *softmax*. En la siguiente figura se puede ver la implementación de este modelo de red neuronal.

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 2: Modelo de red neuronal con una capa de entrada y una capa de salida.

Una vez definido el modelo de red pasamos a entrenarlo y posteriormente a testearlo. Para el entrenamiento de los modelos de red se utilizará el conjunto de datos de MNIST destinado a ello, el cual se compone de 60.000 imágenes de dígitos manuscritos. Se establecerán 10 épocas para el entrenamiento y un tamaño de batch de 200. El conjunto de imágenes de test para la red lo forman 10.000 imágenes.

```
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200, verbose=2)
```

Figura 3: Entrenamiento de red neuronal simple.

```
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Figura 4: Prueba de red neuronal simple.

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_nn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-28 20:52:58.500703: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 20:52:58.500729: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 20:52:58.500737: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 20:52:58.500743: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
6s - loss: 0.2781 - acc: 0.9212 - val_loss: 0.1412 - val_acc: 0.9576
Epoch 2/10
5s - loss: 0.1115 - acc: 0.9677 - val_loss: 0.0923 - val_acc: 0.9704
Epoch 3/10
6s - loss: 0.0719 - acc: 0.9796 - val_loss: 0.0785 - val_acc: 0.9774
Epoch 4/10
6s - loss: 0.0506 - acc: 0.9857 - val_loss: 0.0743 - val_acc: 0.9770
Epoch 5/10
6s - loss: 0.0373 - acc: 0.9893 - val_loss: 0.0679 - val_acc: 0.9789
Epoch 6/10
6s - loss: 0.0269 - acc: 0.9927 - val_loss: 0.0635 - val_acc: 0.9802
Epoch 7/10
6s - loss: 0.0213 - acc: 0.9945 - val_loss: 0.0623 - val_acc: 0.9807
Epoch 8/10
6s - loss: 0.0143 - acc: 0.9970 - val_loss: 0.0628 - val_acc: 0.9802
Epoch 9/10
6s - loss: 0.0111 - acc: 0.9977 - val_loss: 0.0602 - val_acc: 0.9802
Epoch 10/10
6s - loss: 0.0085 - acc: 0.9983 - val_loss: 0.0602 - val_acc: 0.9818
 9664/10000 [============================>..] - ETA: 0sBaseline Error: 1.82%
```

Figura 5: Resultados red neuronal simple.

Se puede ver en la **Figura 4** como se ha obtenido un resultado del **1.83 %** de error sobre el conjunto de prueba en un tiempo total de 59 segundos. No es una buena solución, por lo que vamos a probar a definir otro modelo de red neuronal para mejorar esos resultados.

Siguiendo la línea del modelo anterior definido, lo que se propone en esta solución es añadir nuevas capas ocultas para mejorar el proceso de aprendizaje ya que con un modelo de red neuronal simple los perceptrones no son capaces de extraer tanto nivel de características. A este tipo de redes se les conoce como redes neuronales multicapas. El modelo de red definido tiene la estructura que muestra la siguiente figura.

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/2), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 6: Red neuronal multicapa.

Como se puede apreciar en la imagen, el modelo de red mantiene el mismo número de neuronas en las capas de entrada y de salida que el modelo anterior. La capa oculta de este modelo se compone una única capa con la mitad de neuronas que la capa de entrada. Tras entrenar la red con los datos pertinentes, el testeo de la misma devolvió los siguientes resultados.

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_nn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 11:47:18.733083: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:47:18.733110: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:47:18.733116: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:47:18.733148: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
11s - loss: 0.2349 - acc: 0.9309 - val_loss: 0.1094 - val_acc: 0.9663
Epoch 2/10
11s - loss: 0.0808 - acc: 0.9757 - val_loss: 0.0881 - val_acc: 0.9734
Epoch 3/10
11s - loss: 0.0493 - acc: 0.9847 - val_loss: 0.0606 - val_acc: 0.9810
Epoch 4/10
11s - loss: 0.0322 - acc: 0.9900 - val_loss: 0.0678 - val_acc: 0.9777
Epoch 5/10
12s - loss: 0.0239 - acc: 0.9922 - val_loss: 0.0712 - val_acc: 0.9800
Epoch 6/10
10s - loss: 0.0168 - acc: 0.9949 - val_loss: 0.0742 - val_acc: 0.9792
Epoch 7/10
10s - loss: 0.0147 - acc: 0.9955 - val_loss: 0.0799 - val_acc: 0.9788
Epoch 8/10
10s - loss: 0.0138 - acc: 0.9956 - val_loss: 0.0710 - val_acc: 0.9819
Epoch 9/10
11s - loss: 0.0133 - acc: 0.9954 - val_loss: 0.0821 - val_acc: 0.9814
Epoch 10/10
11s - loss: 0.0103 - acc: 0.9965 - val_loss: 0.0695 - val_acc: 0.9842
 9632/10000 [===========================>..] - ETA: 0sBaseline Error: 1.58%
```

Figura 7: Resultado prueba red neuronal multicapa.

Podemos observar como el resultado en la precisión de acierto ha mejorado con respecto al modelo de red anterior de un **1.82 %** a un **1.58 %**. Una mejora evidente debida a la inclusión de la capa oculta en el modelo.

Aunque el porcentaje de error ha disminuido con la propuesta de este nuevo modelo, con algunas variaciones podemos seguir intentando reducirlo más. Vamos a probar a modificar el número de neuronas de la única capa que hay dentro de la capa oculta para observar si los resultados son favorables. Para ellos definimos este nuevo modelo de red neuronal.

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 8: Red neuronal multicapa (ii).

Este nuevo modelo mantiene el mismo número de capas que el anterior, con la modificación del número de neuronas de la capa oculta que pasa a tener el mismo que la capa de entrada. Con la nueva red neuronal definida

pasamos a entrenarla y testearla para observar su comportamiento.

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_nn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 13:11:32.994534: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 13:11:32.994564: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 13:11:32.994570: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 13:11:32.994574: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
13s - loss: 0.2100 - acc: 0.9370 - val_loss: 0.1103 - val_acc: 0.9667
Epoch 2/10
13s - loss: 0.0744 - acc: 0.9775 - val_loss: 0.0836 - val_acc: 0.9741
Epoch 3/10
13s - loss: 0.0451 - acc: 0.9855 - val_loss: 0.0681 - val_acc: 0.9781
Epoch 4/10
12s - loss: 0.0296 - acc: 0.9904 - val_loss: 0.0700 - val_acc: 0.9791
Epoch 5/10
13s - loss: 0.0228 - acc: 0.9925 - val_loss: 0.0829 - val_acc: 0.9776
Epoch 6/10
12s - loss: 0.0186 - acc: 0.9939 - val_loss: 0.0749 - val_acc: 0.9794
Epoch 7/10
13s - loss: 0.0145 - acc: 0.9953 - val_loss: 0.0697 - val_acc: 0.9802
Epoch 8/10
12s - loss: 0.0130 - acc: 0.9957 - val_loss: 0.0772 - val_acc: 0.9817
Epoch 9/10
12s - loss: 0.0109 - acc: 0.9964 - val_loss: 0.0960 - val_acc: 0.9766
Epoch 10/10
12s - loss: 0.0138 - acc: 0.9955 - val_loss: 0.0747 - val_acc: 0.9826
 9856/10000 [============================>.] - ETA: 0sBaseline Error: 1.74%
```

Figura 9: Resultados red neuronal multicapa (ii).

La imagen de la **Figura 9** nos muestra en el resultado como ha aumentado el porcentaje de error en el acierto, lo que nos marca que el modelo de red definido no mejora al anterior. Probaremos esta vez a reducir en cuatro veces el número de neuronas de la capa oculta y ver que resultados arroja una vez que esta haya sido entrenada.

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/4), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 10: Red neuronal multicapa (iii).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_nn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 13:31:59.073880: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 13:31:59.074403: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 13:31:59.074419: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 13:31:59.074424: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
8s - loss: 0.2635 - acc: 0.9244 - val_loss: 0.1195 - val_acc: 0.9642
Epoch 2/10
8s - loss: 0.0917 - acc: 0.9728 - val_loss: 0.0911 - val_acc: 0.9717
Epoch 3/10
8s - loss: 0.0562 - acc: 0.9826 - val_loss: 0.0642 - val_acc: 0.9788
Epoch 4/10
8s - loss: 0.0357 - acc: 0.9890 - val_loss: 0.0625 - val_acc: 0.9806
Epoch 5/10
7s - loss: 0.0257 - acc: 0.9922 - val_loss: 0.0719 - val_acc: 0.9782
Epoch 6/10
8s - loss: 0.0194 - acc: 0.9942 - val_loss: 0.0599 - val_acc: 0.9821
Epoch 7/10
8s - loss: 0.0151 - acc: 0.9955 - val_loss: 0.0635 - val_acc: 0.9825
Epoch 8/10
8s - loss: 0.0131 - acc: 0.9958 - val_loss: 0.0735 - val_acc: 0.9796
Epoch 9/10
7s - loss: 0.0090 - acc: 0.9973 - val_loss: 0.0817 - val_acc: 0.9817
Epoch 10/10
8s - loss: 0.0100 - acc: 0.9968 - val_loss: 0.0695 - val_acc: 0.9815
 9952/10000 [============================>.] - ETA: 0sBaseline Error: 1.85%
```

Figura 11: Resultados red neuronal multicapa (iii).

Vemos que el resultado lanzado por el nuevo modelo de red no mejora tampoco el hasta ahora mejor resultado obtenido con el primer modelo de red neuronal multicapa. El porcentaje de error en el acierto empeora al obtenido con el anterior modelo de red, por lo que el decrementar tanto el número de neuronas en la capa oculta no aporta buenos resultados.

Siguiendo con este tipo de red neuronal, vamos ahora a implementar un nuevo modelo en el que se va a introducir una nueva capa en la capa oculta. De este modo la capa oculta estará compuesta por dos capas en lugar de una sola como en el ejemplo anterior. Estas dos capas en principio van a tener el mismo número de neuronas, el correspondiente a la mitad de las que posee la capa de entrada, es decir, tanto la primera capa como la segunda de la capa oculta tendrán la mitad de neuronas que la capa de entrada.

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/2), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/2), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 12: Red neuronal multicapa (iv).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_nn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 14:03:26.768844: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:03:26.768873: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:03:26.768880: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:03:26.768886: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
12s - loss: 0.2271 - acc: 0.9337 - val_loss: 0.1124 - val_acc: 0.9645
Epoch 2/10
11s - loss: 0.0774 - acc: 0.9761 - val_loss: 0.0751 - val_acc: 0.9764
Epoch 3/10
11s - loss: 0.0485 - acc: 0.9849 - val_loss: 0.0675 - val_acc: 0.9774
Epoch 4/10
13s - loss: 0.0332 - acc: 0.9894 - val_loss: 0.0653 - val_acc: 0.9798
Epoch 5/10
13s - loss: 0.0285 - acc: 0.9908 - val_loss: 0.1066 - val_acc: 0.9712
Epoch 6/10
11s - loss: 0.0245 - acc: 0.9918 - val_loss: 0.0804 - val_acc: 0.9808
Epoch 7/10
11s - loss: 0.0190 - acc: 0.9937 - val_loss: 0.0835 - val_acc: 0.9809
Epoch 8/10
11s - loss: 0.0170 - acc: 0.9947 - val_loss: 0.0727 - val_acc: 0.9811
Epoch 9/10
11s - loss: 0.0130 - acc: 0.9960 - val_loss: 0.0791 - val_acc: 0.9810
Epoch 10/10
11s - loss: 0.0147 - acc: 0.9953 - val_loss: 0.0821 - val_acc: 0.9825
 9952/10000 [============================>.] - ETA: 0sBaseline Error: 1.75%
```

Figura 13: Resultados red neuronal multicapa (iv).

Analizando el resultado obtenido con este nuevo modelo de red se puede observar como seguimos sin mejorar el mejor porcentaje de error conseguido hasta ahora de **1.58 %**. Vamos a variar los parámetros del número de neuronas de las capas dentro de la capa oculta para analizar el comportamiento de los resultados que se obtengan.

Se ha definido un primer modelo de red en base al anterior en el que esta vez la primera capa oculta mantiene el mismo número de neuronas que la capa de entrada, mientras que la segunda capa oculta continua manteniendo la mitad.

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/2), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 14: Red neuronal multicapa (v).

```
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 14:16:34.825005: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:16:34.825032: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:16:34.825039: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:16:34.825045: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
17s - loss: 0.2073 - acc: 0.9379 - val_loss: 0.1048 - val_acc: 0.9674
Epoch 2/10
16s - loss: 0.0757 - acc: 0.9766 - val_loss: 0.0809 - val_acc: 0.9749
Epoch 3/10
16s - loss: 0.0471 - acc: 0.9854 - val_loss: 0.0726 - val_acc: 0.9775
Epoch 4/10
18s - loss: 0.0347 - acc: 0.9888 - val_loss: 0.0711 - val_acc: 0.9795
Epoch 5/10
17s - loss: 0.0270 - acc: 0.9916 - val_loss: 0.0915 - val_acc: 0.9764
Epoch 6/10
16s - loss: 0.0253 - acc: 0.9916 - val_loss: 0.0704 - val_acc: 0.9811
Epoch 7/10
16s - loss: 0.0188 - acc: 0.9937 - val_loss: 0.0786 - val_acc: 0.9795
Epoch 8/10
16s - loss: 0.0195 - acc: 0.9933 - val_loss: 0.0731 - val_acc: 0.9818
Epoch 9/10
16s - loss: 0.0135 - acc: 0.9957 - val_loss: 0.0745 - val_acc: 0.9811
Epoch 10/10
16s - loss: 0.0139 - acc: 0.9952 - val_loss: 0.0804 - val_acc: 0.9822
 9664/10000 [============================>..] - ETA: 0sBaseline Error: 1.78%
```

Figura 15: Resultados red neuronal multicapa (v).

No se han mejorado los resultados con este nuevo modelo viendo su **1.78 %** de error en el acierto, por lo que vamos a modificar los parámetros de este variando el número de neuronas de las capas ocultas. En esta ocasión la primera capa oculta tendrá la mitad de neuronas que la capa de entrada, mientras que la segunda capa oculta se compondrá del mismo número de neuronas que las que tiene que la capa de entrada.

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/2), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 16: Red neuronal multicapa (vi).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_nn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 14:25:16.994864: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:25:16.994892: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:25:16.994899: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:25:16.994905: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
13s - loss: 0.2093 - acc: 0.9378 - val_loss: 0.1070 - val_acc: 0.9654
Epoch 2/10
12s - loss: 0.0761 - acc: 0.9764 - val_loss: 0.0773 - val_acc: 0.9764
Epoch 3/10
16s - loss: 0.0470 - acc: 0.9851 - val_loss: 0.0828 - val_acc: 0.9741
Epoch 4/10
14s - loss: 0.0357 - acc: 0.9884 - val_loss: 0.0728 - val_acc: 0.9785
Epoch 5/10
13s - loss: 0.0295 - acc: 0.9905 - val_loss: 0.0755 - val_acc: 0.9777
Epoch 6/10
13s - loss: 0.0238 - acc: 0.9923 - val_loss: 0.0625 - val_acc: 0.9843
Epoch 7/10
13s - loss: 0.0181 - acc: 0.9942 - val_loss: 0.0752 - val_acc: 0.9796
Epoch 8/10
13s - loss: 0.0186 - acc: 0.9936 - val_loss: 0.0972 - val_acc: 0.9758
Epoch 9/10
13s - loss: 0.0179 - acc: 0.9944 - val_loss: 0.0773 - val_acc: 0.9798
Epoch 10/10
13s - loss: 0.0131 - acc: 0.9958 - val_loss: 0.0772 - val_acc: 0.9820
 9504/10000 [===========================>..] - ETA: 0sBaseline Error: 1.80%
```

Figura 17: Resultados red neuronal multicapa (vi).

Se puede apreciar como los resultados que estamos obteniendo no tienden a mejorar el que hasta ahora es el más óptimo. En esta ocasión con el modelo anterior se ha conseguido un **1.80 %** en segundos, lo que hace seguir en la misma línea que el anterior.

Mateniendo esta misma línea de modelo de red, vamos a optar por introducir otra capa oculta más para analizar el resultado que podemos obtener con esta nueva opción. Para ello, partiendo del modelo de red en el que hasta ahora se han estado basando todos los demás, añadiremos una nueva capa oculta. Se mantendrá el número de neuronas para las dos primeras capas ocultas, es decir, el mismo que la capa de entrada para la primera capa oculta y la mitad de las neuronas para la segunda capa oculta. Esta tercera capa estará compuesta por el mismo número de neuronas que la segunda. En la siguiente imagen se puede ver el nuevo modelo de red definido.

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/2), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(round(num_pixels/2), input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 18: Red neuronal multicapa (vii).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_nn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 14:49:54.445870: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:49:54.445901: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:49:54.445909: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 14:49:54.445917: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
12s - loss: 0.2254 - acc: 0.9335 - val_loss: 0.1041 - val_acc: 0.9680
Epoch 2/10
11s - loss: 0.0775 - acc: 0.9760 - val_loss: 0.0791 - val_acc: 0.9752
Epoch 3/10
11s - loss: 0.0483 - acc: 0.9849 - val_loss: 0.0715 - val_acc: 0.9765
Epoch 4/10
11s - loss: 0.0347 - acc: 0.9892 - val_loss: 0.0648 - val_acc: 0.9819
Epoch 5/10
11s - loss: 0.0287 - acc: 0.9906 - val_loss: 0.0976 - val_acc: 0.9723
Epoch 6/10
11s - loss: 0.0218 - acc: 0.9926 - val_loss: 0.0738 - val_acc: 0.9791
Epoch 7/10
11s - loss: 0.0174 - acc: 0.9945 - val_loss: 0.0816 - val_acc: 0.9795
Epoch 8/10
11s - loss: 0.0185 - acc: 0.9941 - val_loss: 0.0725 - val_acc: 0.9813
Epoch 9/10
11s - loss: 0.0163 - acc: 0.9947 - val_loss: 0.0894 - val_acc: 0.9776
Epoch 10/10
11s - loss: 0.0180 - acc: 0.9943 - val_loss: 0.0819 - val_acc: 0.9798
 9984/10000 [============================>.] - ETA: 0sBaseline Error: 2.02%
```

Figura 19: Resultados red neuronal multicapa (vii).

Una vez entrenada y testeada esta nueva red se puede observar como sigue sin disminuir el error en el acierto con un **2.02 %** obtenido en esta ocasión, que ha padecido un aumento debido posiblemente al sobre-entremiento de la misma. Este último experimento hace que nos planteemos una nueva alternativa de modelo de red neuronal dando paso a los modelos de redes neuronales convolutivas.

Este tipo de modelo de red es característico por las capas de convolución y agrupación que sufren los datos de entrada. La capa de convolución es el primer proceso por el que pasan los datos de entrada. El cometido de este capa es aplicar un filtro de tamaño que oscila entre 3x3 y 5x5 a la matriz de datos de entrada que forma la imagen, y obtener un conjunto de mapas de características de la misma. Con este proceso se obtienen mapas de características que esencialmente es la misma imagen que la de entrada, pero donde cada píxel es mucho mas rico en información ya que contiene información sobre la región en la que se encuentra y no sólo la del propio pixel. Con la aplicación de esta capa se consigue también reducir el número de conexiones y de parámetros a entrenar en comparación con una red neuronal multicapa.

La otra capa que se va aplicar en el nuevo modelo de red neuronal es una capa de agrupación o de *pooling*. Esta capa toma como datos de entrada los mapas de características generados por la capa anterior y agrupa dichas características de varias coordenadas contiguas con alguna función de agrupación como la media o el máximo. La salida de esta capa será los mismos mapas de características que tomó de entrada, pero con el tamaño reducido en función del tipo de agrupamiento que se realice (2 ó 4 suele ser lo normal ya que un factor de agrupamiento mayor puede hacer perder información de características) .

En esta tesitura, se ha implementado un modelo de red neuronal convolutiva como el que se muestra en la siguiente imagen, el cual se ha entrenado y testeado siguiendo el mismo proceso que los anteriores para comprobar su nivel de acierto en la clasificación de las imágenes de dígitos manuscritos. La capa de convolución que se aplica en este modelo de red generará 32 mapas de características empleando para ello unos filtros de tamaño 5x5. La segunda capa, de agrupamiento, utilizará un factor (2,2), lo que hará que el tamaño de los array de entrada a esta capa se reduzcan a la mitad a su salida.

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 20: Red neuronal convolutiva (i).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python simple_cnn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-30 16:19:40.877546: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 16:19:40.877709: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 16:19:40.877720: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 16:19:40.877726: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
150s - loss: 0.2310 - acc: 0.9345 - val_loss: 0.0825 - val_acc: 0.9739
Epoch 2/10
146s - loss: 0.0737 - acc: 0.9780 - val_loss: 0.0467 - val_acc: 0.9843
Epoch 3/10
141s - loss: 0.0534 - acc: 0.9839 - val_loss: 0.0433 - val_acc: 0.9857
Epoch 4/10
137s - loss: 0.0403 - acc: 0.9877 - val_loss: 0.0406 - val_acc: 0.9861
Epoch 5/10
137s - loss: 0.0339 - acc: 0.9893 - val_loss: 0.0346 - val_acc: 0.9884
Epoch 6/10
137s - loss: 0.0278 - acc: 0.9913 - val_loss: 0.0310 - val_acc: 0.9896
Epoch 7/10
137s - loss: 0.0235 - acc: 0.9927 - val_loss: 0.0358 - val_acc: 0.9878
Epoch 8/10
137s - loss: 0.0207 - acc: 0.9937 - val_loss: 0.0322 - val_acc: 0.9887
Epoch 9/10
137s - loss: 0.0169 - acc: 0.9943 - val_loss: 0.0299 - val_acc: 0.9900
Epoch 10/10
147s - loss: 0.0144 - acc: 0.9958 - val_loss: 0.0313 - val_acc: 0.9904
Baseline Error: 0.96%
```

Figura 21: Resultados red neuronal convolutiva (i).

Se puede apreciar como el resultado de **0.96 %** de error en el acierto obtenido ha mejorado notablemente el que hasta ahora era el mejor conseguido de **1.58 %** ya que incluso ha bajado de 1, lo que nos manifiesta que de las 10000 imágenes de prueba sólo se ha equivocado en 96. En el nuevo modelo de red neuronal convolutiva, ademas de las capas de convolución y agrupación que hacen que las neuronas extraigan mejor información de las caracterísitcas, también se aplican dos procesos que ayudan a agilizar la información de entrada que se presenta a las neuronas de procesamiento. Se emplea un método *dropout* para aleatoriamente establecer a 0 el valor de algunos datos de entrada y evitar de este modo el sobre-entrenamiento. También se aplica un método de *flatten* para representar los datos de entrada en un array de una dimensión en lugar de la representación de dos dimensiones que es como se obtienen. Por último, se agrega una capa de neuronas en la capa oculta que contiene 128 neuronas de procesamiento con una función de activación *relu*. La capa de salida se mantiene siendo la misma que en el resto de experimentaciones anteriores, es decir, con 10 neuronas correspondientes a los diferentes tipos de clasificación, y una función de activación *softmax*.

Viendo las notables mejoras que este modelo de red presenta, se va a definir otro nuevo modelo de red neuronal convolutiva siguiendo la misma línea que el anterior. Tomándo este base, la siguiente propuesta de modelo de red va contener una capa mas de convolución y otra de agrupamiento. Los parámetros de las capas

de convolución serán diferentes, generando la primera capa 30 mapas de características usando unos filtros de 5x5, mientras que la segunda capa generará 15 mapas utilizando filtros de tamaño 3x3. Ambas capas emplearán la misma función de activación *relu*.

Las dos capas de agrupación tomarán el mismo parámetro (2,2), reduciendo a la mitad el tamaño de los mapas de características que reciben como entrada. También se añadirá una capa más de neuronas con un tamaño de 50 a la que ya existe de 128, ambas con la función *relu* como función de activación.

Definida la red, pasamos a entrenarla y probarla para ver su comportamiento.

```
# define the model
def cnn_model():
    # create model
    model = Sequential()
    model.add(Conv2D(30, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(15, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 22: Red neuronal convolutiva (ii).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python complex_cnn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-28 11:21:34.943839: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 11:21:34.943866: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 11:21:34.943873: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 11:21:34.943879: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
60000/60000 [==============================] - 160s - loss: 0.3918 - acc: 0.8799 - val_loss: 0.0963 - val_acc: 0.9686
Epoch 2/10
60000/60000 [==============================] - 156s - loss: 0.0953 - acc: 0.9706 - val_loss: 0.0562 - val_acc: 0.9816
Epoch 3/10
60000/60000 [==============================] - 164s - loss: 0.0690 - acc: 0.9787 - val_loss: 0.0385 - val_acc: 0.9886
Epoch 4/10
60000/60000 [==============================] - 163s - loss: 0.0563 - acc: 0.9824 - val_loss: 0.0330 - val_acc: 0.9885
Epoch 5/10
60000/60000 [==============================] - 163s - loss: 0.0478 - acc: 0.9853 - val_loss: 0.0309 - val_acc: 0.9896
Epoch 6/10
60000/60000 [==============================] - 168s - loss: 0.0436 - acc: 0.9861 - val_loss: 0.0282 - val_acc: 0.9906
Epoch 7/10
60000/60000 [==============================] - 164s - loss: 0.0387 - acc: 0.9876 - val_loss: 0.0277 - val_acc: 0.9905
Epoch 8/10
60000/60000 [==============================] - 154s - loss: 0.0345 - acc: 0.9889 - val_loss: 0.0226 - val_acc: 0.9922
Epoch 9/10
60000/60000 [==============================] - 162s - loss: 0.0325 - acc: 0.9902 - val_loss: 0.0217 - val_acc: 0.9932
Epoch 10/10
60000/60000 [==============================] - 160s - loss: 0.0282 - acc: 0.9907 - val_loss: 0.0219 - val_acc: 0.9927
10000/10000 [==============================] - 14s
Large CNN Error: 0.73%
```

Figura 23: Resultados red neuronal convolutiva (ii).

Nuevamente el resultado ha vuelto a mejorar bajando el porcentaje de error en el acierto hasta un *0.73 %* con respecto al anterior, lo que confirma que el aumentar las capas de convolución y agrupamiento así como la de neuronas, hace que estas últimas obtengan unas caraterísticas de mejor calidad.

Con el objetivo de seguir mejorando la precisión de acierto de la red neuronal vamos a volver a definir un nuevo modelo de red neuronal convolutiva tomando el anterior y modificando una serie de parámetros. Observando que las mejoras han venido producidas por las capas de convolución y de agrupamiento, se van a modificar los parámetros del primer tipo. A las dos capas de convolución existentes se les va a aumentar el número de mapas de características a generar, siendo 40 en la primera y 20 en la segunda. En ambas se mantendrá tanto el tamaño del filtro como la función de activación como se puede visualizar en las imágenes.

```python
# define the model
def cnn_model():
    # create model
    model = Sequential()
    model.add(Conv2D(40, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(20, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 24: Red neuronal convolutiva (iii).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python complex_cnn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-28 12:47:10.311558: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 12:47:10.312004: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 12:47:10.312017: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 12:47:10.312023: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
60000/60000 [==============================] - 218s - loss: 0.3461 - acc: 0.8937 - val_loss: 0.0813 - val_acc: 0.9752
Epoch 2/10
60000/60000 [==============================] - 257s - loss: 0.0875 - acc: 0.9734 - val_loss: 0.0485 - val_acc: 0.9846
Epoch 3/10
60000/60000 [==============================] - 395s - loss: 0.0625 - acc: 0.9801 - val_loss: 0.0347 - val_acc: 0.9882
Epoch 4/10
60000/60000 [==============================] - 238s - loss: 0.0515 - acc: 0.9841 - val_loss: 0.0306 - val_acc: 0.9893
Epoch 5/10
60000/60000 [==============================] - 204s - loss: 0.0418 - acc: 0.9866 - val_loss: 0.0315 - val_acc: 0.9895
Epoch 6/10
60000/60000 [==============================] - 202s - loss: 0.0370 - acc: 0.9885 - val_loss: 0.0258 - val_acc: 0.9913
Epoch 7/10
60000/60000 [==============================] - 203s - loss: 0.0329 - acc: 0.9893 - val_loss: 0.0262 - val_acc: 0.9915
Epoch 8/10
60000/60000 [==============================] - 202s - loss: 0.0284 - acc: 0.9909 - val_loss: 0.0231 - val_acc: 0.9918
Epoch 9/10
60000/60000 [==============================] - 202s - loss: 0.0266 - acc: 0.9913 - val_loss: 0.0227 - val_acc: 0.9929
Epoch 10/10
60000/60000 [==============================] - 207s - loss: 0.0253 - acc: 0.9916 - val_loss: 0.0204 - val_acc: 0.9933
 9984/10000 [============================>.] - ETA: 0sLarge CNN Error: 0.67%
```

Figura 25: Resultados red neuronal convolutiva (iii).

Se puede apreciar en las imagenes de arriba como de nuevo el porcentaje de error en el acierto vuelve a disminuir hasta el **0.67 %** con este nuevo modelo de red neuronal convolutiva como fruto del aumento de los mapas de características a generar en las capas de convolución.

Con este nuevo resultado obtenido que mejora a todos los anteriores, vamos a definir un nuevo modelo de red para intentar decrementar algo más el porcentaje de error actual. Tomando el mismo modelo de red neuronal convolutiva con el que hemos obtenido el mejor resultado, es decir, el que se visualiza en la **Figura 24** vamos a realizar el mismo proceso que antes aumentando el número de mapas producir por las capas de convolución. En esta ocasión, a la primera capa se le asignará 50 mapas de características y 25 a la segunda. El resto de parámetros se mantendrán igual.

Se pasará a entrenar y probar el nuevo modelo de red para analizar el resultado.

```
# define the model
def cnn_model():
    # create model
    model = Sequential()
    model.add(Conv2D(50, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(25, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 26: Red neuronal convolutiva (iv).

```
(mnist) MacBook-Pro-de-Jesus:scripts jesusgarciamanday$ python complex_cnn.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2017-08-28 19:15:46.480341: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 19:15:46.480365: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 19:15:46.480371: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-28 19:15:46.480375: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
use FMA instructions, but these are available on your machine and could speed up CPU computations.
60000/60000 [==============================] - 230s - loss: 0.3263 - acc: 0.8992 - val_loss: 0.0748 - val_acc: 0.9754
Epoch 2/10
60000/60000 [==============================] - 223s - loss: 0.0808 - acc: 0.9755 - val_loss: 0.0488 - val_acc: 0.9832
Epoch 3/10
60000/60000 [==============================] - 233s - loss: 0.0576 - acc: 0.9822 - val_loss: 0.0335 - val_acc: 0.9889
Epoch 4/10
60000/60000 [==============================] - 218s - loss: 0.0476 - acc: 0.9851 - val_loss: 0.0325 - val_acc: 0.9895
Epoch 5/10
60000/60000 [==============================] - 212s - loss: 0.0390 - acc: 0.9878 - val_loss: 0.0303 - val_acc: 0.9905
Epoch 6/10
60000/60000 [==============================] - 232s - loss: 0.0355 - acc: 0.9887 - val_loss: 0.0280 - val_acc: 0.9903
Epoch 7/10
60000/60000 [==============================] - 240s - loss: 0.0292 - acc: 0.9903 - val_loss: 0.0292 - val_acc: 0.9902
Epoch 8/10
60000/60000 [==============================] - 225s - loss: 0.0268 - acc: 0.9915 - val_loss: 0.0250 - val_acc: 0.9917
Epoch 9/10
60000/60000 [==============================] - 216s - loss: 0.0238 - acc: 0.9926 - val_loss: 0.0253 - val_acc: 0.9912
Epoch 10/10
60000/60000 [==============================] - 215s - loss: 0.0224 - acc: 0.9928 - val_loss: 0.0214 - val_acc: 0.9926
 9984/10000 [============================>.] - ETA: 0sLarge CNN Error: 0.74%
```

Figura 27: Resultados red neuronal convolutiva (iv).

En esta ocasión el porcentaje de error no ha disminuido sino que ha aumentando hasta un **0.74 %**. Esto puede ser debido a que al aumentar el número de mapas de características generados por la capa de convolución se haya repetido mucho información de los datos dando lugar a que haya algo de sobre-aprendizaje.

Como se ha podido comprobar con las pertinentes experimentaciones realizadas, el modelo de red neuronal convolucional es el que mejor resultado ha obtenido llegando al **0.67 %** lo que hace que sólo se equivoque en 67 imágenes de las 10000 de prueba. Son varias las conclusiones que se pueden extraer con los resutaldos de la batería de pruebas que se ha realizado con todos los modelos de redes neuronales, pero sin duda el que se realice previamente un proceso de convolución y agrupamiento parece ser clave para que las capas de neuronas reciban con mayor claridad los datos de entradas. También es importante reseñar que en función de los parámetros que se configuren en estas capas se podrá ajustar en mayor o menor medida el resultado del porcentaje de error en el acierto de las imágenes.

Es por eso por lo que hay que probar diversas configuraciones hasta encontrar los parámetros óptimos para la correcta clasificación de las imágenes de dígitos manuscritos.

4. Listado de soluciones.

En la siguiente tabla se muestran las diferentes soluciones obtenidas:

Tipo de red	Capas ocultas (*)	Capas convolución (**)	Capas pooling	% Error	Tiempo (segundos)
Simple	0	-	-	1.83	59
Multicapa	1(392)	-	-	1.58	108
Multicapa	1(784)	-	-	1.74	125
Multicapa	1 (196)	-	-	1.85	78
Multicapa	2 (392, 392)	-	-	1.75	115
Multicapa	2 (784, 392)	-	-	1.78	162
Multicapa	2 (392, 784)	-	-	1.80	133
Multicapa	2 (392, 392)	-	-	2.02	111
Convolutiva	1 (128)	1(32)	1	0.96	1406
Convolutiva	2 (128, 50)	2(30, 15)	2	0.73	1628
Convolutiva	2 (128, 50)	2(40, 20)	2	0.67	2328
Convolutiva	2 (128, 50)	2(50, 25)	2	0.74	2244

(*) número de neuronas
(**) número de mapas

Cuadro 1: Lista de soluciones.

Se puede apreciar como los resultados obtenidos con las redes neuronales convolutivas son los que menor porcentaje de error en el acierto tienen, incrementando considerablemente el tiempo de cómputo.

5. Bibliografía.

https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/

https://elitedatascience.com/keras-tutorial-deep-learning-in-python