UNIVERSITAT POLITÈCNICA DE VALÈNCIA

MIARFID
Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital
Universitat Politècnica de València

Práctica 2. Reconocimiento de dígitos y clasificación de imágenes mediante redes neuronales artificiales

Redes Neuronales Artificiales

Autor: Juan Antonio López Ramírez

Curso 2019-2020

Índice general

CAPÍTULO 1
Introducción

El objetivo de la práctica consiste en resolver dos problemas mediante el uso de redes neuronales artificiales. Estos problemas son:

- Reconocimiento de dígitos del 0 al 9, para el que emplearemos un perceptrón multicapa y la base de datos de MNIST para recoger 60000 muestras de entrenamiento y 10000 de prueba. Las imágenes son de 28x28 píxeles, remodeladas a vectores de 784 valores.
- Clasificación de imágenes de 10 tipos distintos, utilizando redes convolucionales y la base de datos CIFAR, de la que hemos recogido 50000 muestras de entrenamiento y 10000 de prueba. Las imágenes son de 32x32 píxeles, remodeladas a vectores de 1024 valores.

Para implementar las redes neuronales, se han empleado los Jupyter Notebooks que proporciona Google Colaboratory. El entorno de ejecución por defecto que se ha usado ha sido el acelerador por GPU debido a un mejor rendimiento de tiempo.

Figura 1.1: Ejemplo de muestras de la base de datos MNIST

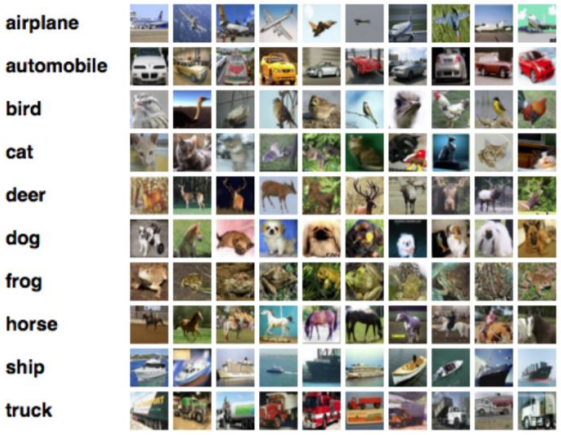

Figura 1.2: Ejemplo de muestras de la base de datos CIFAR con los distintos tipos de imágenes en los que hay que clasificar.

CAPÍTULO 2
Reconocimiento de dígitos con MNIST

Para el problema de reconocimiento de dígitos, se ha partido de la configuración por defecto del fichero *5_mlp_BN_GN_LRA_DA.py*, que se encuentra en https://github.com/RParedesPalacios/DeepLearningLab/blob/master/Examples/Keras/MNIST/. Es decir, la primera configuración de nuestro perceptrón multicapa contaba con cuatro ajustes de mejora:

- **La normalización del *Batch***, mediante las instrucciones:

```
from keras.layers.normalization import BatchNormalization as BN
...
model.add(BN()) #Para cada capa de la red neuronal
...
```

- La introducción de **ruido gaussiano**, que en este caso ha sido del 30 %, representado en el código por:

```
from keras.layers import GaussianNoise as GN
...
model.add(GN(0.3)) #Para cada capa de la red neuronal
...
```

- El ***learning rate annealing***, mediante la implementación de una retrollamada y un planificador que tiene en cuenta el *epoch* en el que nos encontramos para modificar el *learning rate*. En nuestro caso, su valor será de 0.1 hasta el *epoch* 25, que pasará a valer 0.01, y a partir del *epoch* 50 valdrá 0.001. Implementado en código esto es:

```
from keras.callbacks import LearningRateScheduler as LRS
...
def scheduler(epoch):
    if epoch < 25:
        return .1
    elif epoch < 50:
        return 0.01
    else:
        return 0.001

set_lr = LRS(scheduler)
...
history = model.fit(x_train, y_train,..., callbacks=[set_lr])
```

- ***Data Augmentation***, moviendo las muestras en altura y en anchura en un 10 %:

```
datagen = ImageDataGenerator(
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=False)
...
history=model.fit_generator(datagen.flow(x_train, y_train,
    batch_size=batch_size), ...)
...
```

Una vez realizado esto, junto con un *batch* de tamaño 100, 75 *epochs* y una topología de 1024 neuronas por capa, con 3 capas ocultas con función de activación *relu* y la capa de salida con una función *softmax*, el resultado fue un acierto en test del 99.03 %, como se puede apreciar en la figura 2.1.

Posteriormente, se aumentaron los *epochs*, primero a 100 y luego a 150, pero los resultados eran muy similares al original. Cuando se disminuyó el ruido gaussiano hasta el 10 %, se alcanzó un acierto del 99.20 % (véase la figura 2.1). De forma paralela, se redujo el número de neuronas de 1024 a 512, obteniendo un acierto del 99.22 %.

También se probó a introducir ***Dropout***, primero del 30 % y posteriormente del 50 %, y siempre el mismo en todas las capas ocultas. Pero, debido a que los resultados obtenidos eran siempre peores que sus configuraciones equivalentes sin *Dropout*, se han descartado.

Finalmente, reduciendo el número de capas a 2 y cambiando el learning rate annealing de la forma en que se explica en la figura 2.1, obtuvimos un **99.34 %** de acierto con 1024 neuronas por capa y un 99.28 % con 512 neuronas. Por tanto, la configuración final de nuestra red neuronal, debido a que el error es menor que el 0.8 % que se pide en el enunciado del ejercicio y es la mayor tasa de acierto que hemos obtenido, es:

- La normalización del *batch*, el ruido gaussiano, el *learning rate annealing* y el *Data Augmentation* explicados anteriormente.
- Una topología consistente en 2 capas ocultas de 512 neuronas cada una, todas con función de activación *relu*, mientras que la capa de salida cuenta con una función de activación *softmax*.
- Un tamaño de *batch* de 100 y 150 *epochs*.

Fecha prueba	Tamaño del batch	Epochs	Capas ocultas			Learning Rate Annealing	Acierto en test
			Nº de capas	Nº Neuronas/Capa	Ruido Gaussiano		
09/12/2019	100	75	3	1024	0.3	Si	99.03%
10/12/2019	100	100	3	1024	0.3	Si	99.01%
15/12/2019	100	150	3	1024	0.3	Si	98.97%
16/12/2019	100	150	3	1024	0.5	Si	98.11%
17/12/2019	100	150	3	1024	0.1	Si	99.20%
22/12/2019	100	150	3	512	0.3	Si	99.03%
27/12/2019	100	150	3	512	0.1	Si	99.22%
03/01/2020	100	150	2	1024	0.1	Si (modificado*)	99.34%
03/01/2020	100	150	2	512	0.1	Si (modificado*)	99.28%
	* El factor de aprendizajo cambia en la epoch 75 y 125, en lugar de la 25 y 50						

Figura 2.1: Configuraciones del MLP utilizadas y resultados obtenidos.

CAPÍTULO 3
Clasificación de imágenes con CIFAR

Para el problema de clasificación de imágenes, hemos partido de la configuración que venía por defecto en el archivo *1_cifar_conv.py*, ubicado en la página web https://github.com/RParedesPalacios/DeepLearningLab/blob/master/Examples/Keras/CIFAR/. Esta consiste, básicamente, en una topología de 6 capas ocultas más la de entrada y la de salida. La capa de entrada y las 5 primeras capas ocultas consisten en bloques convolucionales con un tamaño de parámetros de 1024 (imágenes de entrada), 32, 64, 128, 256 y 512, respectivamente. La última capa oculta es una capa densa de 512 parámetros. Todas las capas ocultas tienen un ruido gaussiano del 30 % (salvo la capa oculta densa) y función de activación *relu*. La capa de salida tiene una función de activación *softmax*. El código que representa dicha topología es:

```
model = Sequential()

model=CBGN(model,32,x_train.shape[1:])
model=CBGN(model,64)
model=CBGN(model,128)
model=CBGN(model,256)
model=CBGN(model,512)

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))

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

Donde la función *CBGN* define un bloque convolucional con *Batch Normalization*, ruido gaussiano y un maxpooling de 2x2. Esto queda representado en código de esta forma:

```
def CBGN(model,filters,ishape=0):
  if (ishape!=0):
    model.add(Conv2D(filters, (3, 3), padding='same',
                     input_shape=ishape))
  else:
    model.add(Conv2D(filters, (3, 3), padding='same'))

  model.add(BN())
  model.add(GN(0.3))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  return model
```

Además, la red convolucional viene configurada con un tamaño de batch de 100 y 75 epochs.

El resultado obtenido fue de 81.26 %. Las primeras modificaciones consistieron en aumentar los epochs de 75 a 100 e ir disminuyendo el ruido gaussiano, pero el resultado no variaba de forma significativa (véase la figura 3.1).

Posteriormente, se añadió el *learning rate annealing* de la misma forma que se había implementado en el anterior problema, es decir, que el factor de aprendizaje varía después del epoch 25 y del 50. En este caso, el porcentaje de acierto aumentó significativamente a un 83.05.

Luego, se aumentó el número de epochs a 150 y se añadió *Data Augmentation* en un 10 % de altura y de anchura y volteando las muestras en horizontal, consiguiendo llegar al 85.87 % de acierto.

Después, se probó a cambiar el *learning rate annealing* para que la modificación del factor de aprendizaje ocurriese más tarde. Así, en lugar de variar en el epoch 25 y 50, cambiaría en el 75 y en el 125 (la misma modificación que en MNIST). Este experimento resultó favorable, pues se pasó a un resultado de 87.68 %.

A continuación, modificamos el *Data Augmentation* para un 20 % de altura y de anchura, junto con un rango de rotación de 20 grados y un zoom también del 20 %. Por último, establecemos la media de entrada a 0 sobre el conjunto de datos y dividimos las entradas por el conjunto de datos estándar, ambas medidas en función de las características de las muestras. Estas medidas implementadas se ven reflejadas aquí:

```
datagen = ImageDataGenerator(
            featurewise_center=True,
            featurewise_std_normalization=True,
            width_shift_range=0.2,
            height_shift_range=0.2,
            rotation_range=20,
            zoom_range=[1.0,1.2],
            horizontal_flip=True)
```

Para hacer la normalización de funciones, es necesario también entrenar el *ImageDataGenerator* con las muestras de entrenamiento, crear otro *ImageDataGenerator* para prueba, también con normalización de funciones, y por último entrenar este nuevo *ImageDataGenerator*. Esto se ha implementado así:

```
datagen.fit(x_train)

testdatagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
)

testdatagen.fit(x_train)
```

Y en las últimas líneas de código:

```
model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
          epochs=epochs,
          validation_data=testdatagen.flow(x_test, y_test),
          callbacks=[set_lr],
          shuffle=True)
```

Además, hemos cambiado la función *CBGN*, que define los bloques convolucionales, para añadir una nueva capa densa por bloque con *Batch Normalization*, 30 % de ruido gaussiano y función de activación *relu*. De esta forma, el código resultante es:

```
def CBGN(model,filters,ishape=0):
  if (ishape!=0):
    model.add(Conv2D(filters, (3, 3), padding='same',
                     input_shape=ishape))
  else:
    model.add(Conv2D(filters, (3, 3), padding='same'))

  model.add(BN())
  model.add(GN(0.3))
  model.add(Activation('relu'))

  model.add(Conv2D(filters, (3, 3), padding='same'))
  model.add(BN())
  model.add(GN(0.3))
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  return model
```

Con estas últimas modificaciones en la configuración de la red convolucional, el resultado obtenido ha sido de **90.81 %**, superando el umbral del 10 % de error que se nos pedía en el enunciado del ejercicio, como se puede apreciar en la figura 3.1.

Fecha prueba	Tamaño del batch	Epochs	Ruido Gaussiano	Learning rate annealing	Data Augmentation	Resultados
09/12/2019	100	75	0.3	No	No	81.26%
10/12/2019	100	100	0.3	No	No	81.87%
21/12/2019	100	100	0.2	No	No	80.78%
22/12/2019	100	100	0.1	No	No	81.21%
23/12/2019	100	100	0.3	Si (Default)	No	83.05%
24/12/2019	100	100	0.2	Si (Default)	No	82.32%
25/12/2019	100	100	0.1	Si (Default)	No	81.35%
01/01/2020	100	100	0.3	Si (Default)	Si	85.83%
02/01/2020	100	150	0.3	Si (Default)	Si	85.87%
02/01/2020	100	150	0.3	Si (Modified*)	Si	87.68%
05/01/2020	100	150	0.3	Si (Modified*)	Si (Modified^)	90.81%

* La configuración modificada implica que lr cambia en la epoch 75 y en la 125, en lugar de en la 25 y 50

^ Altura y anchura originalmente movidas un 10%, en la nueva configuración, un 20%. Además, se ha añadido rotation_range de 20 y horizontal_flip

Figura 3.1: Configuraciones de la CNN utilizadas y resultados obtenidos.

CAPÍTULO 4
Conclusiones

Gracias a estos experimentos prácticos se han visto los efectos que pueden tener las técnicas vistas en las sesiones de teoría.

La utilización de *learning rate annealing* y, sobre todo, su modificación para que el cambio en el factor de aprendizaje ocurra en un instante del entrenamiento más tardío, ha repercutido mucho en ambos problemas, siendo determinante a la hora de superar los umbrales de error que se proponían en el enunciado.

Otra técnica importante ha sido *Data Augmentation*, especialmente en las redes convolucionales, donde el simple hecho de añadirla ya nos hizo obtener una mejora nada despreciable del 4 %, pero fue al introducir cambios en las imágenes, como pueden ser voltearlas sobre el eje horizontal o rotarlas, cuando conseguimos el resultado esperado.