Reconocimiento óptico de caracteres MNIST

Introducción

En este documento se recoge el todo el trabajo que he desarrollado para la realización de la práctica *"Reconocimiento óptico de caracteres MNIST"* para la asignatura **Inteligencia Computacional** del **Máster profesional en Ingeniería Informática** (2020-2021).

En primer lugar aclarar que nunca antes había programado una red neuronal, es un ámbito de la informática que no me llama tanto como otras áreas y con el que nunca había tenido contacto (salvo algunas asignaturas muy básicas en el grado). Aún así he intentado crear una red neuronal desde cero, basándome en una gran cantidad de documentación que se detallará más adelante. Además, debido a los pobres resultados que he obtenido en esta primera red neuronal, he creado una segunda con ayuda de algunas librerías que ya incluyen muchos de los algoritmos que se suelen usar y que facilitan mucho la creación de redes neuronales.

Hardware y software utilizado

El hardware es el siguiente:

- **CPU**: Intel Core i5-4690k @ 4,1GHz
- **RAM**: Kingston HyperX 16GB 1600MHZ CL10 DDR3
- **GPU**: Nvidia GeForce GTX 970 4GB

El lenguaje de programación y las librerías que he utilizado son:

- **Lenguaje**: `Python 3.8.5`
- **Librerías**:
  - `numpy`
  - `keras`
  - `tensorflow`
  - `mnist`
  - `csv`
  - `datetime`
  - `sklearn`
  - `math`
  - `time`

Organización del código

Junto a este documento se adjunta el código fuente que se ha generado, el cual tiene la siguiente estructura que se muestra a continuación. Además se encuentra alojado en GitHub.

```
mnist_nn
├── LICENSE
├── requirements.txt
└── src
    ├── basic                          # Red neuronal "básica"
    |   ├── __init__.py
```

```
    |   ├── activation_functions.py   # Funciones de activación.
    |   ├── datasets                  # Contiene los datasets MNIST.
    |   ├── neural_network.py         # Abstracción de una red neuronal.
    |   ├── output                    # Contiene archivos .csv con los
resultados.
    |   ├── train.py                  # Script para entrenar una red neuronal.
    |   └── utils.py                  # Algunas funciones auxiliares.
    └── keras
        ├── __init__.py
        ├── train.py                  # Red neuronal hecha con Keras.
        └── utils.py                  # Algunas funciones auxiliares.
```

Primera red neuronal

> El código fuente de esta red neuronal se encuentra en el directorio `src/basic`.

Esta primera red neuronal se ha creado en `Python 3.8.5` nativo sin el uso de bibliotecas de terceros para crear redes neuronales pero sí basándome en diferentes guías y documentación. En este caso he intentado abstraer el concepto de una red neuronal para poder crear diferentes redes neuronales de diferente tamaño.

He implementado además diferentes funciones de activación para comprobar su efectividad al entrenar la red neuronal. Se pueden encontrar en el archivo `src/basic/activation_functions.py` y son: `sigmoid`, `sigmoid_derivative`, `relu`, `elu`, `lrelu`, `softmax` y `gelu`.

La estructura del fichero que contiene la red neuronal es la siguiente:

```python
class NeuralNetwork:
    def __init__(self, layers, epochs, rate, activation='sigmoid'):
      # Inicialización de los datos de la red neuronal.

    def generate_weights(self):
      # Generación de los pesos usados en las diferentes capas.

    def back_propagation(self, values, output):
      # Implementación de la propagación de valores "hacia atrás".

    def forward_propagation(self, values):
      # Implementación de la propagación de valores "hacia delante".

    def update(self, changes):
      # Actualización de los valores de las capas.

    def accuracy(self, x, y):
      # Calcula la precisión de la red.

    def train(self, x_train, y_train, x, y):
      # Entrena la red.
```

He realizado varios tests con dos redes neuronales, cuya configuración es la siguiente:

- Primera red
    - Capa de entrada de 784 neuronas (28 x 28, una por cada pixel de la imagen de entrada).
    - Dos capas ocultas, de 128 y 64 neuronas respectivamente.

    - Capa de salida, con 10 neuronas (los 10 posibles valores numéricos de la predicción).
- Segunda red
    - Capa de entrada de 784 neuronas (28 x 28, una por cada pixel de la imagen de entrada).
    - Capa oculta de 28 neuronas.
    - Capa de salida, con 10 neuronas (los 10 posibles valores numéricos de la predicción).

En los tests que he realizado la configuración ha sido la siguiente:

- Funciones de activación: **sigmoidal** y **sigmoidal derivada**.
- Epochs: **10**, **20**, **30**, **40** y **50**.
- Tasa de aprendizaje: **0.001**, **0.01**, **0.1**, **0.25**, **0.5** y **1**.

Resultados

Los resultados se encuentran en el directorio `src/basic/output` y en general son bastante pésimos. Las siguientes tablas resumen los resultados, mostrando para cada función de activación la mejor precisión obtenida (según epochs y tasa de aprendizaje). Algunos de estos resultados se han obviado, pues la precisión era realmente baja y los tiempos de entrenamiento excesivamente altos (llegando hasta los 15-20 minutos de entrenamiento).

Sigmoid derivative {.tabset}

En este caso, las redes neuronales han usado *sigmoid_derivative* como función de activación y han ofrecido un rendimiento pésimo, encontrándose la precisión máxima (**35.69%**) en la red de tres capas con una tasa de aprendizaje de 0.01. Se puede observar que a mayor número de epochs las redes comienzan a divergir, bajando mucho la precisión.

[784, 28, 10]

Epochs	L. Rate	Acc. (%)
10	0.001	18.4
10	0.01	27
10	0.1	28.57
10	0.25	20.16
20	0.001	24.75
20	0.01	**35.69**
20	0.1	22.03
20	0.25	17.41
30	0.001	28.34
30	0.01	44.01
30	0.1	19.20
30	0.25	16.42

[784, 128, 64, 10]

Epochs	L. Rate	Acc. (%)
10	0.001	11.47
10	0.01	10.35
10	0.1	7.34
10	0.25	9.63
20	0.001	11.47
20	0.01	9.19
20	0.1	7.32
20	0.25	**11.77**
30	0.001	8.97
30	0.01	9.21
30	0.1	0.4
30	0.25	7.44

Sigmoid {.tabset}

En este otro caso, las redes neuronales han usado *sigmoid* como función de activación y ofrecen un rendimiento bastante mejor que el observado anteriormente, encontrándose la precisión máxima (**90.95%**) en la red de cuatro capas con una tasa de aprendizaje de 0.25. También ocurre que a partir de 20 epochs las redes neuronales comienzan a divergir.

[784, 28, 10]

Epochs	L. Rate	Acc. (%)
10	0.001	35.6
10	0.01	68.76
10	0.1	72.06
10	0.25	69.65
20	0.001	44.44
20	0.01	72.21
20	0.1	68.73
20	0.25	70.76
30	0.001	56.52
30	0.01	71.73
30	0.1	68.64
30	0.25	**72.88**

[784, 128, 64, 10]

Epochs	L. Rate	Acc. (%)
10	0.001	30.81
10	0.01	21.84
10	0.1	80.45
10	0.25	88.51
20	0.001	29.40
20	0.01	27.62
20	0.1	84.28
20	0.25	**90.95**
30	0.001	12.53
30	0.01	27.88
30	0.1	85.62
30	0.25	89.97

Conclusión

Aún siendo la primera vez que creo una red neuronal desde 0, y aunque los resultados no son los mejores, creo que un **90.95%** es un valor relativamente *aceptable*.

Tras realizar diferentes pruebas con otras funciones de activación los resultados obtenidos eran similares (bastante malos) por lo que no se han tenido en cuenta.

No he seguido tomando muestras ni mejorando la implementación de la red neuronal pues se acercaba la fecha de entrega de esta práctica y los tiempos de entrenamiento eran excesivamente grandes. De media el entrenamiento de estas redes ha tomado entre 15 y 20 minutos.

Tras analizar estos primeros datos decidí crear una segunda red neuronal con librerías especializadas para ello, lo cual seguramente funcionaría mejor que mi implementación.

Segunda red neuronal (Keras)

> El código fuente de esta red neuronal se encuentra en el directorio `src/keras`.

Esta segunda red neuronal mejora significativamente los resultados que se han obtenido en la red neuronal anterior, llegando a un máximo de **99.34%** de precisión.

Para la implementación de la red he decidido utilizar Keras, una librería muy completa que permite crear redes tanto simples como complejas de una manera sencilla.

Estructura de la red

La estructura elegida ha sido una red secuencial, un modelo en el que una capa va tras otra, siendo la entrada de una la salida de la anterior. Para esto se ha hecho uso del método `Sequential()` de Keras.

Tipos de capas

Se han desarrollado diferentes redes, cada una con su configuración de capas distinta, pero en general los tipos de capas que he usado han sido:

- Convolucional (`Conv2D(...)`)
- Polling (`MaxPolling2D(...)`)
- Normalización (`BatchNormalization(...)`)
- Dropout (`Dropout(...)`)
- Flatten (`Flatten(...)`)
- Dense (`Dense(...)`)

Resultados

Algunos primeros tests y pruebas arrojaron bastante buenos resultados comparados con la red neuronal creada desde 0. La siguiente tabla resume algunos de estos:

Epochs	Time (s)	Acc. (%)	Error (%)	Loss
10	401	98.97	1.03	0.037
15	686	99.04	0.96	0.087
10	364	99.08	0.92	0.034
10	604	99.08	0.92	0.03
15	2543	99.08	0.92	0.03
15	269	99.11	0.89	0.037

De estas redes no tengo el código pues fueron las primeras pruebas que hice. Tras sobrepasar el 99% de precisión comencé a guardar los archivos con las redes neuronales que iban mejorando los resultados. Se pueden encontrar junto a los archivos de resultados, en el directorio `src/keras/output/<training>/train.py`.

En las siguientes secciones se encuentran las diferentes mejoras que he ido haciendo a la red neuronal, ordenadas de menor a mayor precisión y teniendo en cuenta siempre el training que ha tomado menor tiempo.

1ª mejora

Epochs	Time (s)	Acc. (%)	Error (%)	Loss
15	271	99.22	0.78	0.022

```
nn_model.fit(images_train, labels_train, batch_size=64, epochs=epochs,
verbose=1)
```

Para entrenar la red se han tomado *batches* de tamaño 64, lo que acelera el proceso de entrenado un poco frente a utilizar *batches* de tamaño menor.

Topología de la red

```
nn_model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28,28,1)))
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Conv2D(32, (3, 3), activation='relu'))
nn_model.add(BatchNormalization())
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Dropout(0.25))
nn_model.add(Flatten())
nn_model.add(Dense(128, activation='relu'))
nn_model.add(Dropout(0.5))
nn_model.add(Dense(10, activation='softmax'))
```

Se observa que:

- Capa 1: Es convolucional con 32 filtros de convolución y 3 filas y tres columnas para cada uno de estos filtros, la función de activación es `relu` y la entrada de estos filtros es una matriz de 28x28.
- Capa 2: Capa de *polling*, que ayuda a reducir los parámetros entre capas tomando el máximo de los 4 valores (2x2) que tom en cada iteración.

- Capa 3: Convolucional con 32 filtros de convolución y 3 filas y tres columnas para cada uno de estos filtros.
- Capa 4: Capa de normalización, toma los valores y los acota entre 0 y 1 (para simplificar las operaciones).
- Capa 5: Segunda capa de *polling*.
- Capa 6: Capa de *Dropout*, que evita el *overfitting*.
- Capa 7: Capa *Flatten*, para hacer unidimensionales los valores devueltos por las capas convolucionales.
- Capa 8: Primera capa densa con activación `relu`.
- Capa 9: Segunda capa de *Dropout*.
- Capa 10: Segunda capa densa con activación `softmax`. Capa final con tamaño 10, los 10 valores posibles.

2ª mejora

Epochs	Time (s)	Acc. (%)	Error (%)	Loss
15	285	99.23	0.77	0.032

```
nn_model.fit(images_train, labels_train, batch_size=64, epochs=epochs,
verbose=1)
```

Para entrenar la red se han tomado *batches* de tamaño 64, lo que acelera el proceso de entrenado un poco frente a utilizar *batches* de tamaño menor.

Topología de la red

```
nn_model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28,28,1)))
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Conv2D(32, (3, 3), activation='relu'))
nn_model.add(BatchNormalization())
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Dropout(0.1))
nn_model.add(Flatten())
nn_model.add(Dense(128, activation='relu'))
nn_model.add(Dropout(0.25))
nn_model.add(Dense(10, activation='softmax'))
```

Se observa que:

- Las capas 6 y 9 cambian el ratio de acción de la función *Dropout*. De 0.25 y 0.5 a 0.1 y 0.25 respectivamente.
- El resto de capas permanecen igual

3ª mejora

Epochs	Time (s)	Acc. (%)	Error (%)	Loss
20	271	99.29	0.71	0.034

```
nn_model.fit(images_train, labels_train, batch_size=128, epochs=epochs,
verbose=1)
```

Para entrenar la red se han tomado *batches* de tamaño 128, al contrario que en las redes anteriores. Esto ha agilizado el proceso de entrenamiento.

Topología de la red

```
nn_model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28,28,1)))
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Conv2D(32, (3, 3), activation='relu'))
nn_model.add(BatchNormalization())
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Dropout(0.1))
nn_model.add(Flatten())
nn_model.add(Dense(128, activation='relu'))
nn_model.add(Dropout(0.25))
nn_model.add(Dense(10, activation='softmax'))
```

Se observa que:

- Las capas son exactamente iguales.
- Se han incrementado los *epochs* de 15 a 20, lo que debería aumentar el tiempo de entrenamiento.
- Se ha modificado el número de batches de 64 a 128, lo que reduce el tiempo de entrenamiento.
- El tiempo de ejecución es el mismo que en el caso anterior, pero la precisión ha aumentado un 0.06%.

4ª mejora

Epochs	Time (s)	Acc. (%)	Error (%)	Loss
30	514	99.34	0.66	0.038

```
nn_model.fit(images_train, labels_train, batch_size=64, epochs=epochs,
verbose=1)
```

Para entrenar la red se han tomado *batches* de tamaño 64.

Topología de la red

```
nn_model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28,28,1)))
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Conv2D(32, (3, 3), activation='relu'))
nn_model.add(BatchNormalization())
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Dropout(0.1))
nn_model.add(Flatten())
nn_model.add(Dense(128, activation='relu'))
nn_model.add(Dropout(0.25))
nn_model.add(Dense(10, activation='softmax'))
```

Se observa que:

- Las capas son exactamente iguales.
- Se han incrementado los *epochs* de 20 a 30.
- Se ha modificado el número de batches de 128 a 64.

- En este último caso la diferencia más significativa ha sido el número de epochs, dedicando mas tiempo al training.

5ª mejora

Epochs	Time (s)	Acc. (%)	Error (%)	Loss
30	415	99.30	0.7	0.027

```
nn_model.fit(images_train, labels_train, batch_size=64, epochs=epochs,
verbose=1)
```

Para entrenar la red se han tomado *batches* de tamaño 64.

Topología de la red

```
nn_model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28,28,1)))
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Conv2D(32, (3, 3), activation='relu'))
nn_model.add(BatchNormalization())
nn_model.add(MaxPooling2D(pool_size=(2,2)))
nn_model.add(Dropout(0.1))
nn_model.add(Flatten())
nn_model.add(Dense(128, activation='sigmoid'))
nn_model.add(Dropout(0.25))
nn_model.add(Dense(10, activation='softmax'))
```

Se observa que:

- Las capas son exactamente iguales.
- La función de activación de la primera capa densa es `sigmoid`.

Otras mejoras y pruebas

Se ha probado lo siguiente:

- Agregar más capas de convolución con diferente tamaño (64 y 128), pero no aportaban una mejora significativa.
- Agregar capas de *polling*, sin mucho éxito.
- Modificar el tamaño de capa densa (tamaño 128), empeorando la precisión de la red.
- Utilizar otras funciones de activación, como `gelu` o `sigmoidal`, pero sin mejoras significativas en la precisión y en ocasiones se ha producido algo de divergencia.
- Se ha compilado la red con la función de optimización `adam`.
- Se ha compilado la red con la función de pérdida `categorical_crossentropy`, que calcula la entropía cruzada entre el valor de los valores y las predicciones.
- La métrica de compilado se ha asignado a `accuracy`.

Valoración personal

Realizar esta práctica me ha parecido realmente interesante. Ha sido la primera vez que he trabajado con redes neuronales de este tipo y de esta manera. En la primera red neuronal *from scratch* he comprobado de primera mano que puede ser muy complejo el desarrollo de esos algoritmos, mientras que por otro lado, en la segunda red neuronal (con *Keras*), es bastante sencillo crear "pequeñas" redes con buenos resultados.

La inteligencia artificial es un campo de la informática que nunca me ha llamado mucho, quizá porque no había trabajado con este tipo de cosas, ya que ahora me parece algo realmente interesante.

Referencias

- Convolutional networks
- Neural Network From Scratch with NumPy and MNIST
- Build a neural network from scratch
- ReLU and Softmax Activation Functions
- The Sequential model
- Probabilistic losses
- Dense layer
- Model trining APIs