Máster Universitario en Ingeniería Informática

Inteligencia Computacional

PRÁCTICA 1: Redes Neuronales

Mnist

UNIVERSIDAD DE GRANADA

Carlos Santiago Sánchez Muñoz

Grupo de prácticas 1 - Lunes

Email: carlossamu7@correo.ugr.es

4 de noviembre de 2020

Índice

1. Introducción

El problema a abordar es un problema de clasificación. La base de datos es a usar es *Optical Recognition of Handwritten Digits* que contiene imágenes de dígitos del sistema de numeración arábigo. El objetivo consiste en aprender de esta base de datos para poder clasificar otras imágenes con dígitos manuscritos.

1.1. Conjunto de datos

El MNIST es un conjunto de datos desarrollado por Yann LeCun, Corinna Cortes y Christopher Burges para la evaluación de modelos de aprendizaje automático sobre el problema de la clasificación de los dígitos escritos a mano. La base de datos se construyó a partir de varios conjuntos de datos de documentos escaneados disponibles en la carpeta Instituto Nacional de Estándares y Tecnología (NIST).

Las imágenes de los dígitos fueron tomadas de una variedad de documentos escaneados, normalizados en tamaño y centrados. Esto hace de los datos un conjunto excelente para evaluar modelos. La base de datos dispone de 60,000 instancias para el conjunto de entrenamiento y 10,000 instancias para el conjunto de test. Dichas imágenes tienen 28×28 píxeles.

El objetivo es encontrar mediante el aprendizaje una función de manera que dada una entrada de tamaño 28×28 nos dé como salida la clase a la que corresponde. Hay diez clases para predecir, una por cada dígito.

1.2. Redes neuronales

Los algoritmos de aprendizaje para este conjunto de datos son los basados en redes neuronales artificiales. En 1998 LeCun construyó un clasificador lineal de una capa obteniendo un 12 % de error sobre el conjunto de test. Posteriormente se introdujo una capa oculta facilitando el aprendizaje y reduciendo dicho error al 3 %.

Actualmente el usando *Deep Learning* con varias redes neuronales convolucionales con múltiples capas ocultas dicho error se ha reducido al $0,23$ %. En este trabajo se va a comenzar construyendo una red neuronal más sencilla y posteriormente incrementarla realizando un ajuste de los diferentes parámetros de la red con el fin de reducir el error sobre el conjunto de prueba. Consultar evolución aquí.

Dicha red neuronal va a intentar a aprender una función $f : \mathcal{X} \to \mathcal{Y}$ de modo que para cada instancia del $(x_n, y_n) \in (\mathcal{X}, \mathcal{Y})$ se tenga $f(x_n) = y_n$. Claramente los elementos de $\mathcal{X}$ son imágenes (matrices) de píxeles de tamaño 28×28 y el conjunto $\mathcal{Y} = \{0, 1, ..., 9\}$.

2. Implementación

Esta sección está dedicada a cubrir todos los aspectos relacionados con la implementación de la red. Comenzaremos con la lectura de datos, luego con el preprocesamiento, la construcción del modelo, la evaluación y finalmente realizaremos algunas predicciones.

He ido apuntando en un archivo `Makefile` los paquetes que he ido necesitando de manera que con la orden `make install` se instalarán todos. La orden `make all` ejecutará el fichero, básicamente lanzará en el shell la orden `python3 mnist.py`.

2.1. Lectura de datos

El lenguaje de programación que he elegido es `Python`. En él se dispone de la biblioteca Keras. A través de ella voy a leer el dataset. He implementado la siguiente función de lectura:

```
""" Lectura de datos
"""
def read_data():
    # divide los datos en train y test
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    return (x_train, y_train), (x_test, y_test)
```

Tal y como se observa la función devuelve los datos correctamente separados en conjuntos de entrenamiento y test separando las entradas de las etiquetas. Nos gustaría obtener información genérica del conjunto de datos por lo que he implementado una función `summarize_dataset` que se encarga de ello.

```
""" Preprocesa los datos de entrada
- x_train: entrada del conjunto de entrenamiento.
- y_train: etiquetas del conjunto de entrenamiento.
- x_test: entrada del conjunto de test.
- y_test: etiquetas del conjunto de test.
"""
def summarize_dataset(x_train, y_train, x_test, y_test):
    print('Dimension de x_train:', x_train.shape, 'y_train:', y_train.shape)
    print('Dimension de x_test:', x_test.shape, 'y_test:', y_test.shape)
    print(x_train.shape[0], 'ejemplos de entrenamiento')
    print(x_test.shape[0], 'ejemplos de test\n')
```

El resultado es el siguiente:

```
Dimensión de x_train: (60000, 28, 28) y_train: (60000,)
Dimensión de x_test: (10000, 28, 28) y_test: (10000,)
60000 ejemplos de entrenamiento
10000 ejemplos de test
```

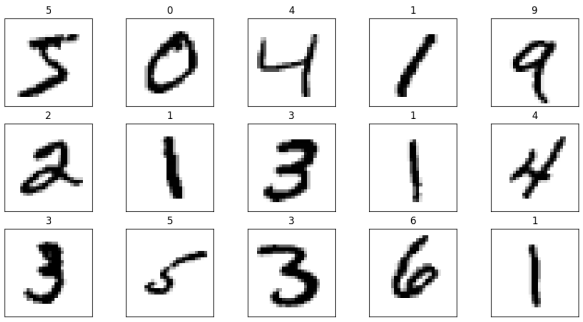
Efectivamente hay 60000 instancias de entrenamiento y 10000 instancias de test. Las imágenes son de 28×28 píxeles como ya se sabía.

2.2. Preprocesamiento

Como primera etapa del preprocesamiento quiero implementar una función `show_data` que imprima algunas imágenes de MNIST con su etiqueta correspondiente:

```
""" Muestra 15 imagenes y su etiqueta
- x: imagen.
- y: etiqueta.
- set_init: primera imagen a mostrar.
"""
def show_data(x, y, set_init=0):
    fig = plt.figure(figsize=(15, 10))
    for id in np.arange(15):     # sólo imprimo 15
        ax = fig.add_subplot(3, 5, id+1, xticks=[], yticks=[])
        ax.imshow(x[id + set_init], cmap=plt.cm.binary)
        ax.set_title(str(y[id + set_init]))
    plt.gcf().canvas.set_window_title('IC - Practica 1')
    plt.show()   # pintamos la imagen
```

Las imágenes son las siguientes:



Ya se anunció que las imágenes están centradas y son un conjunto decente con el que entrenar. Las imágenes están en escala de grises, por lo que en primer lugar vamos a remodelar las matrices. Las variables `IMG_ROWS` e `IMG_COLS` valen 28 ambas.

```
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, IMG_ROWS, IMG_COLS)
    x_test = x_test.reshape(x_test.shape[0], 1, IMG_ROWS, IMG_COLS)
    input_shape = (1, IMG_ROWS, IMG_COLS)
else:
    x_train = x_train.reshape(x_train.shape[0], IMG_ROWS, IMG_COLS, 1)
    x_test = x_test.reshape(x_test.shape[0], IMG_ROWS, IMG_COLS, 1)
    input_shape = (IMG_ROWS, IMG_COLS, 1)
```

Los valores de píxeles son enteros en el rango $[0, 255]$ (blanco y negro). En teoría hemos estudiado que el escalado resulta muy útil a la hora de mejorar el entrenamiento y que el gradiente sea más ortogonal. En este caso podríamos normalizar ese rango al $[0, 1]$. Para ello convertiremos las imágenes a flotantes y posteriormente realizaremos la normalización.

```
""" Preprocesa los datos de entrada
- x_train: entrada del conjunto de entrenamiento.
- x_test: entrada del conjunto de test.
"""
def preprocess_data(x_train, x_test):
    x_train = x_train.astype('float32') # conversión a float32
    x_test = x_test.astype('float32')   # conversión a float32
    x_train /= 255                      # normalización a [0,1]
    x_test /= 255                       # normalización a [0,1]
```

Por otro lado, sabemos que hay 10 clases y que las clases se representan como enteros del 0 al 9. Vamos a convertir el vector de etiquetas a una matriz binaria. Para ello cada elemento de la clase se va a transformar en un vector binario de tamaño 10, que contendrá ceros en todas las posiciones a excepción de aquella clase a la cual pertenece. La función `to_categorical()` realiza esta transformación fácilmente. La variable `N_CLASSES` vale 10.

```
y_train_categorical = keras.utils.to_categorical(y_train, N_CLASSES)
y_test_categorical = keras.utils.to_categorical(y_test, N_CLASSES)
```

Esto daría por finalizada la fase de preprocesamiento.

2.3. Construcción del modelo

Para comenzar esta subsección es fundamental entender los tipos de capas disponibles en Keras, qué hacen y por qué son útiles. Se puede consultar la documentación aquí. Expongo una breve explicación de las que he usado:

- `Sequential`: Indica que la topología de la red es lineal y que todas las capas reciben múltiples entradas o *inputs* y devuelven múltiples salidas o *outputs*.

- `Conv2D`: Es una capa de convolución en $2D$, esta capa crea un núcleo (que actúa como filtro) de convolución que se pasa por todas las imágenes generando la salida. Dicho núcleo es una matriz de convolución que se puede usar para difuminar, afilar, grabar, detectar bordes. El tamaño del mismo suele ser 3×3, 5×5 o 7×7. En nuestro caso el último quizás sea muy grande teniendo en cuenta las dimensiones de nuestro dataset.

  El parámetro `activation` especifica el nombre de la función de activación que desea aplicar después de realizar la convolución.

- `MaxPooling2D`: hay que elegir el tamaño de esa piscina, en nuestro caso va a ser 2×2. Se van a recorrer las imágenes de características de izquierda-derecha y de arriba-abajo tomando *piscinas* de tamaño 2×2 (4 pixeles) y preservando el valor más alto de ellos.

  Es un proceso de *subsampling* que permite reducir el tamaño de las muestras haciendo que el número de neuronas de la red no se dispare. Dicho *subsampling* se hace de manera inteligente ya que prevalecen las características más importantes de cada filtro.

- `Dropout`: Es una capa de regularización que desactiva neuronas en función de una probabilidad pasada como argumento. El *overfitting* ocurre cuando nuestra neurona le asigna mucha importancia a un conjunto pequeño de los valores de la salida

de la capa anterior como característica predictiva. Esta capa ayuda a prevenir el textitoverfitting.

La capa `Dropout`, lleva a 0 de manera aleatoria distintos valores del vector de entrada en cada una de las iteraciones de nuestro entrenamiento. Poner esta capa entre la última capa oculta y la de salida podría ser una mala decisión.

- `Flatten`: Convierte los elementos de la matriz de imágenes de entrada en un array plano. Esto hace que nos acerquemos a nuestro objetivo pues la salida es un array binario de tamaño 10.
- `Dense`: Es una capa completamente conectada o *fully-connected*. Devuelve una salida del tamaño indicado y usa la activación indicada, en mi caso `relu` o `softmax`.

Una vez añadidas las capas que se desean hay que compilar el modelo usando `compile` indicando cuál es la métrica a optimizar y el optimizador. En mi caso la métrica elegida es el `accuracy`.

He comenzado con un modelo muy sencillo para ver qué resultados proporciona. Lo llamaremos `model1`. A partir de este integrando nuevas capas y ajustando los parámetros adecuadamente iremos construyendo el modelo final.

```
""" Construccion del modelo. Devuelve el modelo.
- input_shape: tamanio del input.
"""
def construc_model1(input_shape):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=input_shape))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(N_CLASSES, activation='softmax'))
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.SGD(lr=0.01),
                  metrics=['accuracy'])
    return model
```

El modelo `model2` incluye capas `Conv2D` y `MaxPooling2D` así como un mayor número de capas `Dense` y otra capa `Dropout`.

```
""" Construccion del modelo. Devuelve el modelo.
- input_shape: tamanio del input.
"""
def construc_model2(input_shape):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
```

```
    model.add(Dense(N_CLASSES, activation='softmax'))
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adam(),
                  metrics=['accuracy'])
    return model
```

El modelo `model3` ajusta algunos parámetros del modelo anterior como por ejemplo usar un filtro más grande en la convolución o una probabilidad distinta para la tasa de abandono.

```
""" Construccion del modelo. Devuelve el modelo.
- input_shape: tamanio del input.
"""
def construc_model3(input_shape):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(5, 5),
                     activation='relu',
                     input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.3))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(N_CLASSES, activation='softmax'))
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adam(),
                  metrics=['accuracy'])
    return model
```

He realizado algunas pruebas cambiando el optimizador, por ejemplo a `SGD` o `Adadelta`, y los resultados son peores. En mis pruebas el mejor ha sido el de `Adam`. Finalmente el modelo `model` es el elegido.

```
""" Construccion del modelo. Devuelve el modelo.
- input_shape: tamanio del input.
"""
def construc_model(input_shape):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(5, 5),
                     activation='relu',
                     input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(N_CLASSES, activation='softmax'))
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adam(),
                  metrics=['accuracy'])
    return model
```

El modelo final es un modelo secuencial con dos capas de convolución seguidas de dos capas `MaxPoolin2D`. La primera tiene un filtro de tamaño 5×5 mientras que en la segunda

es 3×3. A continuación se desactivan neuronas con un 20 % de opciones y se transforman las entradas a vectores. Comenzamos con varias capas densas reduciendo la dimensionalidad y volvemos a prevenir del sobreajuste con otra desactivación de neuronas, este caso del 50 %. Por último, se usa es una capa densa de tamaño el número de clases esta vez con una activación `softmax` y así obtenemos la salida deseada.

Los optimizadores se pueden consultar en aquí. Tal y como se ha comentado se usa el optimizador de `Adam`. El algoritmo de optimización de Adam es una extensión del descenso de gradiente estocástico que recientemente ha tenido una adopción más amplia para aplicaciones de aprendizaje profundo en visión por computador.

A este modelo final se le ha hecho una variante (`model_LeakyReLu`) para probar una función de activación estudiada en teoría que se llama `LeakyReLu`. Está disponible en el fichero `.py` pero como no ha resultado satisfactoria aquí no se va a comentar.

La construcción del modelo final queda así:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
dropout (Dropout)            (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 512)               819712
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_2 (Dense)              (None, 50)                6450
_________________________________________________________________
dropout_1 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                510
=================================================================
Total params: 911,664
Trainable params: 911,664
Non-trainable params: 0
```

2.4. Entrenamiento

El entrenamiento del modelo se va a realizar usando la función `fit` sobre el modelo. Hay que indicar algunos parámetros:

- `batch_size`: tamaño del batch.
- `epoch`: número de épocas.
- `verbose`: indica si se debe imprimir por pantalla el avance del entrenamiento.

En mi caso tengo dos variables globales `BATCH_SIZE = 128` y `EPOCHS=15` que le pasaré como argumento.

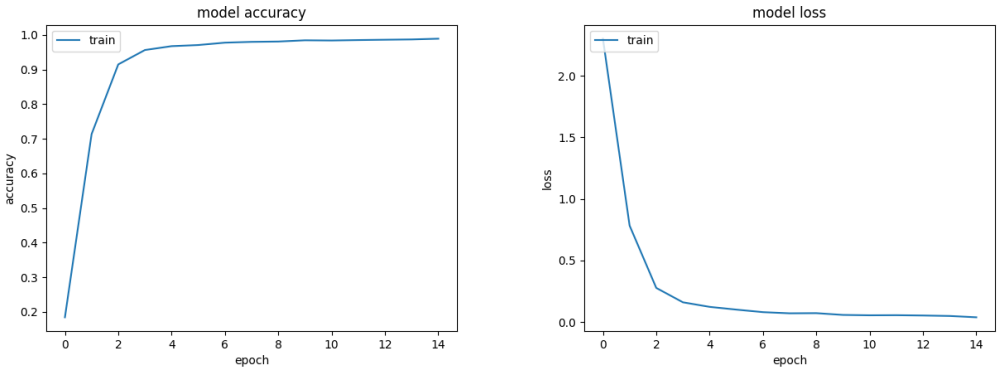
Asimismo voy a sacar un 10 % del conjunto de entrenamiento para usarlo como conjunto de validación. El hecho de que tenga un número de instancias suficientemente grande me permite poder dejar 6000 instancias para validar. No es necesario usar validación cruzada dividiendo el dataset en cinco partes, entrenando con cuatro partes y validando con la quinta parte. Las instancias son elegidas aleatoriamente y no es necesario un muestreo estratificado ya que como el conjunto inicial está balanceado y contiene el mismo número de instancias de cada clase la muestra aleatoria también estará balanceada.

```
""" Entrena el modelo. Devuelve el history.
- model: modelo.
- x_train: entrada del conjunto de entrenamiento.
- y_train: etiquetas del conjunto de entrenamiento.
- x_test: entrada del conjunto de test.
- y_test: etiquetas del conjunto de test.
"""
def train_model(model, x_train, y_train, x_test, y_test):
    x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
    test_size=0.1, random_state=1)
    return model.fit(x_train, y_train,
              batch_size=BATCH_SIZE,
              epochs=EPOCHS,
              verbose=1,
              validation_data=(x_test, y_test))
```

Vamos a tomar una medida del tiempo antes y después de realizar este proceso de cara a tener el tiempo de ejecución. El entrenamiento deja una historia de cómo se ha ido mejorando la tasa de acierto a lo largo de las épocas.

```
""" Muestra la historia del entrenamiento (acc y loss).
- history: historia.
"""
def show_history(history):
    # Accuracy
    plt.plot(history.history['accuracy'])
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train'], loc='upper left')
    plt.title('model accuracy')
    plt.gcf().canvas.set_window_title('IC - Practica 1')
    plt.show()
    # Loss
    plt.plot(history.history['loss'])
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train'], loc='upper left')
    plt.title('model loss')
    plt.gcf().canvas.set_window_title('IC - Practica 1')
    plt.show()
```

Para nuestro problema devuelve los siguientes resultados:



(a) Tasa de acierto (b) Pérdida

Imagen 1: Historia del entrenamiento

Lógicamente conforme avanza el número de épocas el porcentaje de acierto asciende. En 4 épocas ya es muy alto y a partir de ahí mejora lentamente el aprendizaje de pesos. A la pérdida le ocurre justo lo contrario.

2.5. Evaluación

La función `evaluate` ejecutada sobre el modelo evalúa en el conjunto indicado. Usaremos tanto el de entrenamiento como el de test. Normalmente el porcentaje de acierto en el conjunto de entrenamiento es mejor que sobre el de prueba.

```
""" Evalua el modelo. Devuelve los scores.
- model: modelo.
- x_train: entrada del conjunto de entrenamiento.
- y_train: etiquetas del conjunto de entrenamiento.
- x_test: entrada del conjunto de test.
- y_test: etiquetas del conjunto de test.
"""
def evaluate(model, x_train, y_train, x_test, y_test):
    return model.evaluate(x_train, y_train, verbose=0),
           model.evaluate(x_test, y_test, verbose=0)
```

Más adelante se comentarán los resultados obtenidos.

2.6. Predicción y matriz de confusión

Una vez entrenado el modelo podemos predecir las etiquetas de nuevos dígitos manuscritos. Usando el conjunto de test y la función de Keras `predict` obtendríamos las etiquetas.

```
y_pred_categorical = model.predict(x_test)
y_pred = inverse_categorical(y_pred_categorical)
print("Etiquetas predecidas: ", np.array(y_pred))
```

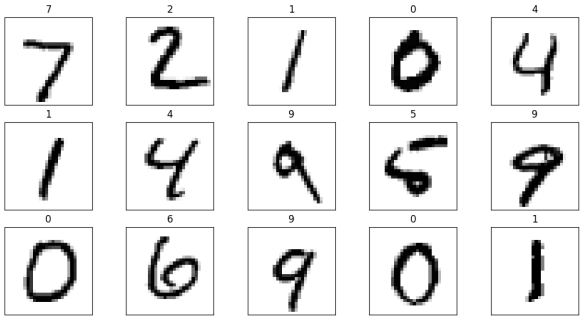
Obsérvese que la predicción es un vector binario de tamaño 10, para recuperar la clase tengo que revertir al valor entero en el rango $[0, 9]$ por lo que he implementado la función:

```
""" Invierte la funcion to_categorical
- y_categorical: matriz binaria a revertir.
"""
def inverse_categorical(y_categorical):
    y = []
    for i in range(len(y_categorical)):
        y.append(np.argmax(np.round(y_categorical[i])))
    return y
```

Las predicciones obtenidas son:

```
Etiquetas predecidas:  [7 2 1 ... 4 5 6]
```

Voy a ejecutar la función `show_img` que ya tengo implementada sobre el conjunto de test. Las primeras 15 imágenes son las siguientes:



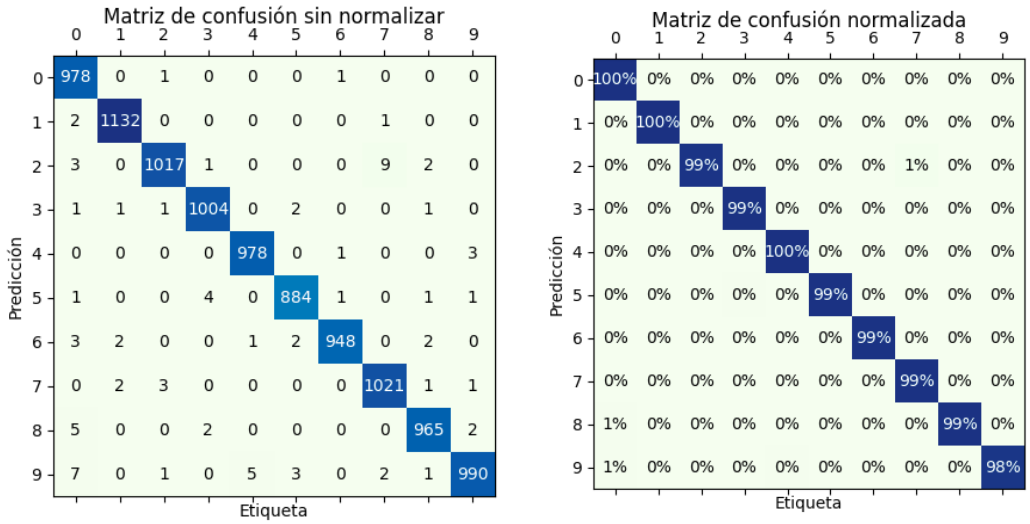
En la asignatura de Aprendizaje Automático implementé una función para mostrar la matriz de confusión de un problema de regresión así que voy a reutilizar dicho código para este problema.

```
""" Muestra matriz de confusion.
- y_real: etiquetas reales.
- y_pred: etiquetas predichas.
- message: mensaje que complementa la matriz de confusion.
- norm (op): indica si normalizar (dar en %) la matriz de confusion.
"""
def show_confussion_matrix(y_real, y_pred, message="", norm=True):
    mat = confusion_matrix(y_real, y_pred)
    if(norm):
        mat = 100*mat.astype("float64")/mat.sum(axis=1)[:, np.newaxis]
    fig, ax = plt.subplots()
    ax.matshow(mat, cmap="GnBu")
    ax.set(title="Matriz de óconfusin {}".format(message),
           xticks=np.arange(10), yticks=np.arange(10),
           xlabel="Etiqueta", ylabel="óPrediccin")

    for i in range(10):
        for j in range(10):
            if(norm):
```

```
                ax.text(j, i, "{:.0f}%".format(mat[i, j]), ha="center", va="center",
                    color="black" if mat[i, j] < 50 else "white")
            else:
                ax.text(j, i, "{:.0f}".format(mat[i, j]), ha="center", va="center",
                    color="black" if mat[i, j] < 50 else "white")
    plt.gcf().canvas.set_window_title("áPrctica 3 - óClasificacin")
    plt.show()
```

Las matrices de confusión (sin normalizar y normalizada) que se obtienen son:



(a) Sin normalizar

(b) Normalizada

Imagen 2: Matriz de confusión

3. Resultados y conclusiones

Esta sección está dedicada a analizar los resultados obtenidos en el aprendizaje y dar las conclusiones oportunas.

3.1. Resultados

En esta sección se van a exponer los resultados de los diferentes modelos construidos.

Modelo `model1`.

Conjunto	**Accuracy**	**Loss**
Train	$99,6583\,\%$	$1,6087\,\%$
Test	$97,9600\,\%$	$8,6809\,\%$

Modelo `model2`.

Conjunto	**Accuracy**	**Loss**
Train	$99,6633\,\%$	$1,2887\,\%$
Test	$98,9700\,\%$	$4,9930\,\%$

Modelo `model3`.

Conjunto	**Accuracy**	**Loss**
Train	$99,5917\,\%$	$1,6329\,\%$
Test	$99,1000\,\%$	$3,9794\,\%$

Modelo `model_LeakyReLu`.

Conjunto	**Accuracy**	**Loss**
Train	$99,4967\,\%$	$1,8730\,\%$
Test	$98,8000\,\%$	$5,1479\,\%$

Modelo `model`.

Conjunto	**Accuracy**	**Loss**
Train	$99,6683\,\%$	$1,4626\,\%$
Test	$99,2400\,\%$	$4,4805\,\%$

En general los resultados son muy buenos. Ya sabíamos que sobre este conjunto de datos se iba a obtener algo similar. Simplemente se han ido refinando los modelos hasta conseguir una tasa de acierto muy buena sobre el conjunto de test. Llama la atención que el primero modelo, el cual era bastante sencillo, es capaz de tener sobre el conjunto de entrenamiento una tasa de acierto del $99,66\,\%$.

3.2. Conclusiones

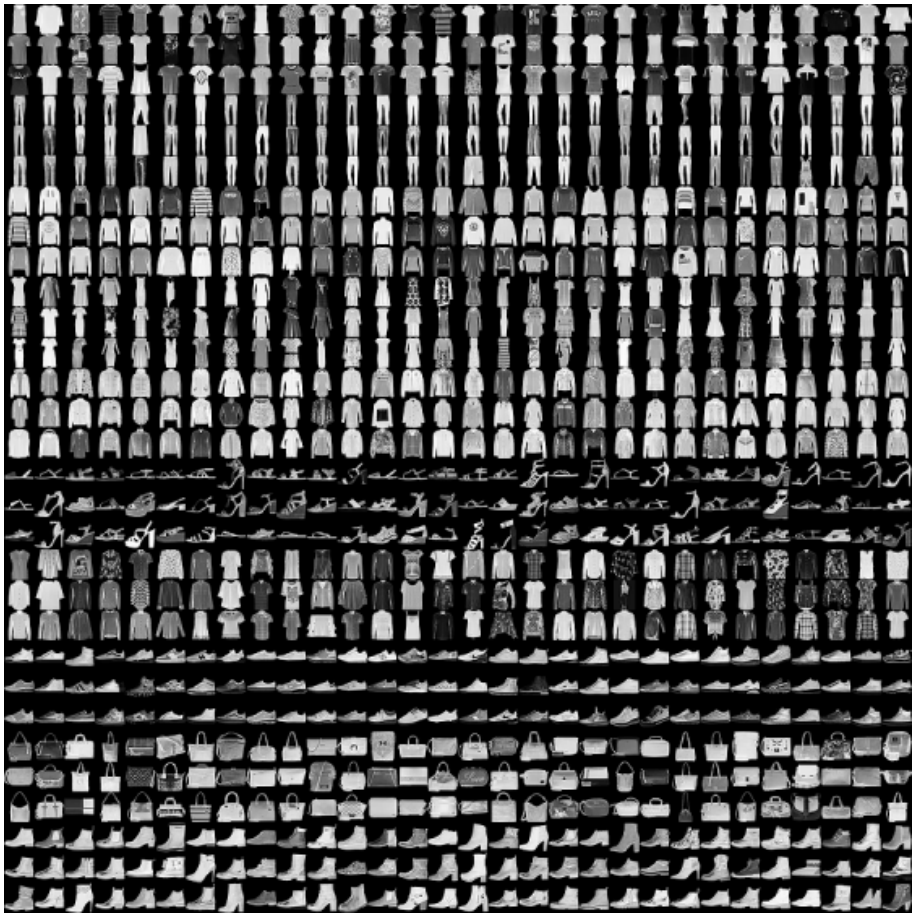
La primera conclusión de este trabajo es que construir una red neuronal que funcione de manera precisa sobre el conjunto de prueba en un tiempo razonable no es una tarea fácil. Hay que hacer un buen diseño de la topología de la red y un posterior ajuste de los parámetros de las capas. Para este proceso de refinado y sabiendo que el volumen de datos es grande disponer de una GPU o poder usar recursos *cloud* puede ser de gran utilidad.

Este proceso ha sido el recorrido que he realizado por los diferentes modelos explicados. En función de los resultados, de la presencia de *overfitting* u otros factores he ajustado la red a las necesidades.

Finalmente, tras la implementación la tasa de acierto conseguida sobre el conjunto de prueba es del $99,24\,\%$. Es un buen resultado pero el tiempo de entrenamiento para llegar a él es considerable.

4. Extra novedoso

Durante el desarrollo de este proyecto con redes neuronales y el dataset MNIST descubrí un dataset análogo denominado MNIST-fashion. De una manera breve me gustaría entrenar la red implementada sobre este conjunto de datos y añadir algún comentario como final novedoso al trabajo.



Esta base de datos nació porque MNIST estaba muy usada y ya se han conseguido resultado resultados excelentes. Tiene las mismas dimensiones (60000 para entrenamiento y 10000 para test) con imágenes de 28×28 y el mismo número de etiquetas.

Etiqueta	**Prenda**	**Etiqueta**	**Prenda**
0	Camiseta/top	5	Sandalia
1	Pantalón	6	Camisa
2	Suéter	7	Zapatilla
3	Vestido	8	Bolsa
4	Abrigo	9	Bota

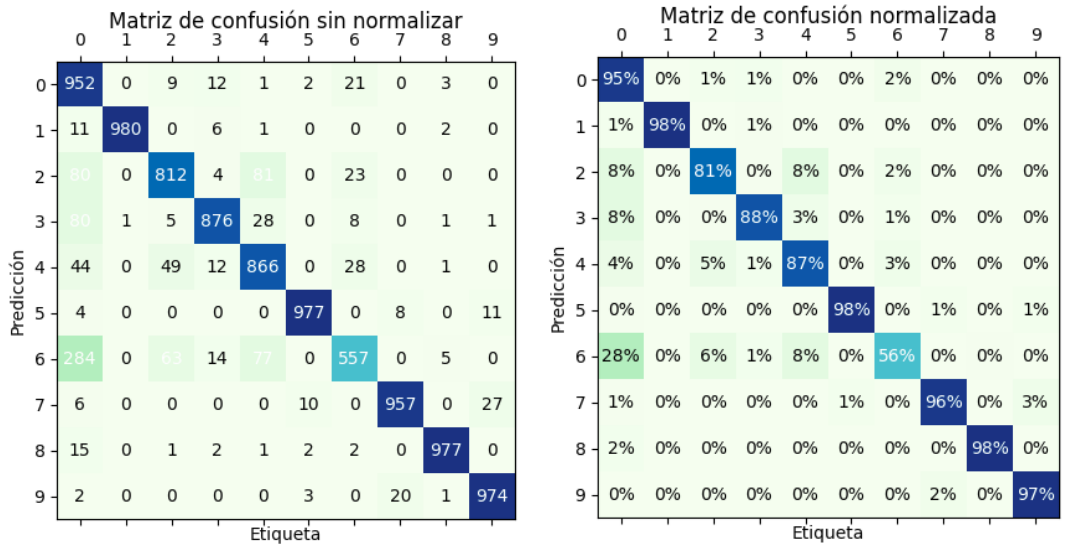
Se ha modificado la función de lectura de datos como sigue para poder leer también este conjunto.

```
""" Lectura de datos
- fashion: indica si se debe leer mnist (False) o fashion_mnist (True).
"""
def read_data(fashion=False):
    # divide los datos en train y test
    if(fashion):
        (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
    else:
        (x_train, y_train), (x_test, y_test) = mnist.load_data()
    return (x_train, y_train), (x_test, y_test)
```

El resto sería igual que lo ya desarrollado y usando el modelo final de MNIST los resultados son los siguientes:

Conjunto	**Accuracy**	**Loss**
Train	$93,3367\,\%$	$17,8289\,\%$
Test	$90,4300\,\%$	$30,8050\,\%$

En este conjunto de datos la tasa de acierto es menor que sobre el propio MNIST. No obstante es satisfactoria también. Voy a mostrar las matrices de confusión de este ejemplo.



(a) Sin normalizar (b) Normalizada

Imagen 3: Matriz de confusión de fashion MNIST

Hay un porcentaje muy alto de confusión entre las clases 0 y 6. Resulta entendible pues el primero representa a la clase camiseta y el segundo a la clase camisa.