UNIVERSIDAD DE GRANADA

Inteligencia de Negocio

grado en ingeniería informática

Práctica Final

Fashion MNIST

Autor
José María Sánchez Guerrero

Rama
Sistemas de Información

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2020-2021

Índice

1. Descripción y análisis del problema

En este trabajo, nos vamos a enfrentar al conjunto de datos *Fashion MNIST*, que contiene una serie de imágenes de artículos de Zalando que tendremos que clasificar dependiendo de cual represente el ejemplo. El dataset consta de un conjunto de 60.000 ejemplos de entrenamiento y otro de test con 10.000 ejemplos más. Cada dato es una imagen en escala de grises de 28x28, y cada una, a su vez, asociada con una etiqueta de 10 clases para identificar la prenda.

Las clases de las etiquetas son las siguientes:

Label	**Descripción**
0	T-shirt/top (Camiseta/top)
1	Trouser (Pantalón)
2	Pullover (Sudadera)
3	Dress (Vestido)
4	Coat (Abrigo)
5	Sandal (Sandalias)
6	Shirt (Camisa)
7	Sneaker (Zapatilla)
8	Bag (Bolso)
9	Ankle boot (Botín)

Este conjunto surje como un conjunto de datos similar al *MNIST*, con la misma estructura y dimensión de imágenes, pero con un poco más de complejidad. Esto se debe a las siguientes razones:

- **MNIST es demasiado fácil**. Las redes convolucionales ya alcanzan prácticamente un 100 % de acierto (convolucionales pueden alcanzar el 99.7 % y las técnicas más clásicas fácilmente llegan al 97 %.
- **MNIST está sobreutilizado**, por lo que ya no supone un reto.
- **MNIST no puede representar** tareas de visión por computador modernas, como bien explica François Chollet en el siguiente enlace [1]

Para obtener los datos tenemos dos opciones. La primera es descargarlo directamente desde la web y guardarlo en nuestro directorio local, y la segunda es ejecutando la línea de código que dejare señalada (y comentada), que te lo descarga automáticamente desde la web.

2. Descripción de los algoritmos

En mi caso, voy a seguir las recomendaciones del guión de ir de menor a mayor dificultad. Empezaremos por los clasificadores clásicos de *Machine Learning*, ya que son una buena forma de comenzar con el problema; pasaremos a una aproximación basada en *Deep Learning*, creando nosotros nuestra propia red convolucional utilizando *Keras*; y por último, vamos a utilizar una red neuronal pre-entrenada y adaptarla a nuestro problema.

2.1. Clasificadores clásicos

Tenemos muchos algoritmos diferentes con los que poder probar, pero voy a hacerlo con los que ya implementé en la última práctica de la asignatura, ya que estoy familiarizado con ellos. Posteriormente, probaremos con otros algoritmos y cambiando parámetros para ver distintos resultados; y también veremos los que mejores resultados obtuvieron en la página web y si hemos conseguido igualar o superar los resultados.

El primer algoritmo que veremos es el ***RandomForest***, un conjunto (ensemble) de árboles de decisión entrenados con una muestra extraída aleatoriamente del conjunto de entrenamiento. Utiliza promedios, agregando a cada nueva observación las predicciones de todos los árboles individuales, para mejorar la precisión y controlar el sobreajuste.

La implementación es muy sencilla:

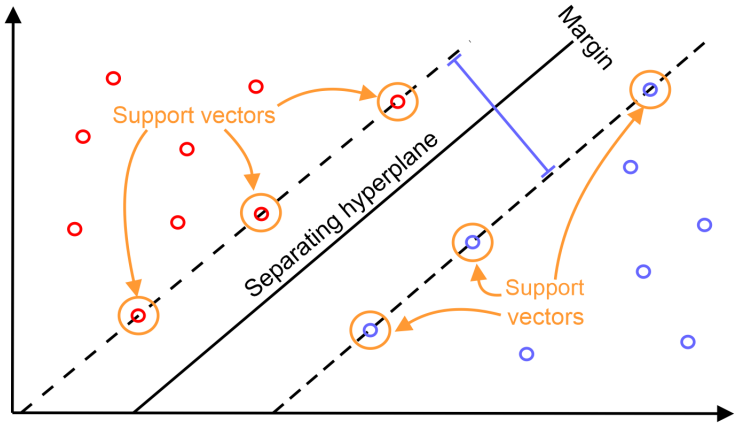
```
1    rf = RandomForestClassifier(n_estimators=100, criterion='gini')
```

Sólo necesitamos declararlo y probar con distintos argumentos. Los más relevantes son el '*n_ estimators*' que indica el número de árboles, '*criterion*' que es la función utilizada para el cálculo de los promedios, y después tenemos más parámetros para controlar la profundidad máxima de los árboles, número mínimo de hojas,de características, etc. Probaremos con diferentes opciones para ver cuál nos va mejor.

El siguiente que vamos a utilizar es el ***Support Vector Machine (SVM)***, ya que es el que mejores resutados tiene sobre el papel, es decir, en las pruebas *benchmark* que realizaron en la web. Este algoritmo busca un hiperplano que separe de forma óptima a los puntos de una clase de la de otra. Al decir de forma óptima nos referimos a que busca el hiperplano que tenga la máxima distancia (margen) con los puntos que estén más cerca de él mismo.

Para tener más claro lo que hace este algoritmo, mostraremos una imagen de

cómo separa los datos dejando el pasillo más grande posible:



La implementación es muy sencilla:

```
svc = SVC(C=10, kernel='poly')
```

Tiene un parámetro 'C' que será el error permitido para este pasillo, es decir, contra más pequeño sea el valor, mayor error va a permitir y más pequeño será el pasillo que divide los datos; y el '*kernel*' que es el algoritmo utilizado para el cálculo de ésta. También probaremos distintos parámetros para ver con cuál nos ha ido mejor.

Por último, vamos a probar con ***StackingClassifier***, que fue con el que mejores resultados obtuve en la práctica 3. Este clasificador consiste en 'apilar' varios clasificadores en una salida perteneciente a un estimador final. Es decir, le pasamos varios clasificadores para calcular la predicción final, y dependiendo de la fuerza que tenga cada clasificador individualmente, se utilizará su salida como entrada de un estimador final (este será en mi caso LogisticRegression(), ya que es el más utilizado). La implementación es la siguiente:

```
# Podemos definir antes los clasificadores
RndForestClf = RandomForestClassifier()

# Clasificadores a apilar en el modelo
estimators = [
    ('lr', LogisticRegression()),
    ('RndForestClf', RndForestClf),
    ('svr', LinearSVC() )
]

# Declaracion del modelo
StackingClf = StackingClassifier(estimators=estimators,
final_estimator=LogisticRegression())
```

La idea es juntar los mejores resultados de los modelos probados anteriormente e intentar quedarnos con lo mejor de cada uno de ellos. Posteriormente, veremos si la idea nos ha funcionado.

2.2. Aproximaciones basadas en *Deep Learning*

Como la mayoría de los problemas de clasificación de imágenes obtienen mejores resultados con modelos de *Deep Learning*, basados en redes convolucionales, vamos a implementar uno de ellos. Para ello, vamos a utilizar ***Keras***, una herramienta especialmente diseñada para facilitar la creación de éstas.

En mi caso he implementado una que ya utilicé para el MNIST y que funcionó bastante bien (un 98.7% de precisión). La estructura de esta red se define en la siguiente tabla:

Layer No.	Layer Type	Kernel size (for conv layers)	Output dimension
1	Conv2D	3	32
2	Relu	-	32
3	Conv2D	3	64
4	MaxPooling2D	2	32
5	Dropout	-	32
6	Flatten	-	1024
7	Dense	1	128
8	Relu	-	128
9	Dropout	-	128
10	Dense	1	10
11	Softmax	-	10

Primero se añade una capa de convolución 2D, la cual crea un núcleo de convolución (de tamaño 3x3) que hay que juntar con otra *inputlayer* para producir un tensor de salidas. Al ser la primera capa, tenemos que añadir el parámetro *input_shape* que hace referencia a las dimensiones de las imágenes leídas anteriormente, cuyo valor era de 28x28 (*input dimension*).

Por otro lado, tenemos la función de activación, que en nuestro caso hemos elegido *'relu'*. La función ReLU o Rectified Linear Unit transforma los valores introducidos anulando los negativos y dejando los positivos igual.

Lo siguiente que hacemos es reducir el muestreo utilizando *MaxPooling2D()*. Lo que hace esto es encontrar el valor máximo en una ventana de muestra (en nuestro

caso de 2x2) y pasa este valor como característica más importante sobre ese área.

También añadimos una nueva capa **Dropout()**. Ésta lo que hace es poner aleatoriamente a 0 distintos valores del vector de entrada en cada una de las iteraciones de nuestro entrenamiento. Esto se hace para obligar a la neurona a dar respuesta para cuando una o varia características importantes desparecen en esa iteración; y así reducir un poco más el *overfitting*. El parámetro que le pasamos a esta función es la probabilidad de que una neurona sea puesta a 0, y lo iremos aumentando a medida que aumentemos la dimensionalidad del vector de entrada.

Por último, tenemos las capas lineales totalmente conectadas. En la primera de ellas volveremos a utilizar una activación ReLU, mientras que en la última tendremos que hacer una activación *'softmax'* para pasar a probabilidad (entre 0 y 1) a las neuronas de salida. Antes de añadir estas capas lineales, tenemos que utilizar la función *Flatten()*, que transforma las k dimensiones actuales a una sola.

La implementación del modelo es la siguiente:

```
model = Sequential()

# Primera capa convolucional
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))

# Segunda capa convolucional
model.add(Conv2D(64, (3, 3), activation='relu'))
# Reduccion de la salida de la primera capa
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Pasamos a una dimension
model.add(Flatten())
# Activacion de las capas lineales totalmente conectadas
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Una vez tenemos definido el modelo, tendremos que definir también un optimizador y, posteriormente, compilarlo con el modelo. El optimizador que vamos a utilizar va a ser **Adam**, que es un método del gradiente descendente estocástico *SGD* basado en una estimación adaptativa de los pesos en vectores de primer y segundo orden. Es computacionalmente eficiente, tiene pocos requisitos de memoria y suele ser bastante adecuado para problemas con grandes cantidades de datos. También tendremos que elegir una función de pérdida, que va a ser *categorical_crossentropy* ya que nuestro problema es de clasificación de varias clases.

La implementción del compilador es la siguiente:

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Por último, ya solo nos queda entrenar nuestro modelo. Para ello utilizaremos la función *fit*, en la que también tendremos que fijar varios parámetros antes de la ejecutarla. Tendremos que tener en cuenta las épocas (*epochs*), las cuales, en un principio, las he puesto en 40 y ya veremos cómo funciona. También hay que fijar tanto los pasos por época, como los pasos de validación (*steps_per_epoch* y *validation_steps*, respectivamente), y suelen tener un valor equivalente al número de imágenes en entrenamiento/validación entre el tamaño de cada *batch*.

Esta es su implementación:

```
model.fit(x_train, y_train,
      steps_per_epoch=int(np.ceil(len(x_train)/batch_size)),
      epochs=epochs,
      verbose=1,
      validation_data=(x_test, y_test),
      validation_steps=int(np.ceil(len(x_test)/batch_size))
      )
```

2.3. Redes preentrenadas - ResNet50

Vamos a adaptar ***ResNet50*** como red preentrenadas de clasificación de imágenes, ya que es una de las más modernas y más utilizadas para el tratamiento de imágenes de la vida real por su buen funcionamiento. Para ello, realizamos los '*imports*' correspondientes que nos proporciona *Keras* con esta clase ya implementada.

Como esta red pre-entrenada ya tienen un *input_shape* predeterminado de 32x32, tendremos que expandir nuestras imágenes para que tengan esta dimensión. Para ello, simplemente ampliamos nuestra imagen con dos columnas de 0s tanto arriba como abajo, y posteriormente, expandimos la imagen. También le hemos metido 3 canales de color, que es lo que admite ResNet50. Este es el código, no tiene mucho misterio:

```
# Aniadimos columnas de 0s para que sean de 32x32,
# que es lo que admite resnet50
x_train_prep = x_train.reshape(-1,28,28)
x_test_prep = x_test.reshape(-1,28,28)

new_Cols = np.zeros([28,2])
new_Rows = np.zeros([2,32])
expand_X_train = np.zeros([len(x_train_prep),32,32])
```

```
    expand_X_test = np.zeros([len(x_test_prep),32,32])

    # Expandimos la imagen
    for i in range(len(x_train)):
        aux = np.c_[new_Cols, x_train_prep[i]]
        aux = np.c_[aux, new_Cols]
        aux = np.r_[new_Rows, aux]
        expand_X_train[i] = np.r_[aux, new_Rows]

    for i in range(len(x_test)):
        aux = np.c_[new_Cols, x_test_prep[i]]
        aux = np.c_[aux, new_Cols]
        aux = np.r_[new_Rows, aux]
        expand_X_test[i] = np.r_[aux, new_Rows]


    # Aniadimos el 3 canal
    new_X_train = np.zeros([len(x_train),32,32,3])
    new_X_test = np.zeros([len(x_test),32,32,3])

    new_X_train[:,:,:,] = expand_X_train.reshape(-1,32,32,1)
    new_X_test[:,:,:,] = expand_X_test.reshape(-1,32,32,1)
```

La implementación de la red es muy similar a la del apartado anterior, pero en este caso, la "primera capa" es la red completa pre-entrenada. Este modelo estará preentrenado gracias el parámetro $weights =' imagenet'$ y también le quitaremos la última capa con $include\_top = False$. Realizamos *GlobalAveragePooling* después de la que ahora sería la última capa, que lo que hace es una convolución 2D y posteriormente un pooling para convertirlo en una dimensión. El parámetro que lo hace es $pooling =' avg'$.

La implementación es la siguiente:

```
    # Definir el modelo ResNet50 (preentrenado en ImageNet).
    model = Sequential()
    model.add(ResNet50(include_top=False,
                       weights='imagenet',
                       input_tensor=None,
                       input_shape=(32,32,3),
                       pooling='avg',classes=10))
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(BatchNormalization())
    model.add(Dense(10, activation='softmax'))
```

Al igual que antes, tenemos que definir un optimizador y compilarlo con el modelo. Vamos a utilizar el mismo de antes, pero con unas recomendaciones para la propia red *ResNet50*. Las más destacadas son: el establecer un *learning_rate* del optimizador y la otra el la introducción de *ReduceLROnPlateau*, una función

que va reduciendo el valor anterior en un factor de 2-10 cuando el aprendizaje se estanca. Para entrenar nuestro modelo lo haremos de formar similiar al anterior.

Esta es la implementación de todo:

```
# Compilar el modelo
model.compile(optimizer=keras.optimizers.Adam(lr=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

red_lr= ReduceLROnPlateau(monitor='val_accuracy',
                          patience=3,
                          verbose=1,
                          factor=0.7)

model.fit(new_X_train,y_train,
          batch_size = batch_size,
          epochs = epochs,
          validation_data = (new_X_test,y_test),
          verbose = 1,
          callbacks=[red_lr])
```

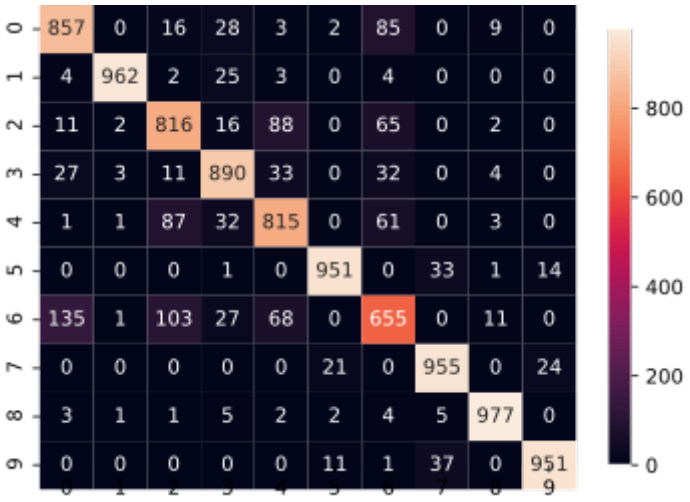
3. Estudio experimental

A continuación, voy a mostrar una tabla con todos los resultados que he obtenido, para posteriormente comentarlos:

Algorithm	**Params**	**Accuracy Mean**
RandomForestClassifier	default: {n_estimators=100, criterion='gini'}	0.8255
RandomForestClassifier	{n_estimators=50, criterion='entropy', max_depth=50}	0.8308
RandomForestClassifier	{n_estimators=75, criterion='entropy', max_depth=50}	0.8343
RandomForestClassifier	{n_estimators=100, criterion='entropy', max_depth=50}	0.8326
SVC	default: {C=1, kernel='rbf'}	0.8829
SVC	{C=10, kernel='poly'}	0.8808
SVC	{C=100, kernel='poly'}	0.8805
StackingClassifier	estimators = [LogisticRegression(), RndForestClf, LinearSVC()]	0.8679
Neural Network	Conv2D = [32-64-128], optimizer=Adam()	**0.9329**
ResNet50	weights='imagenet', optimizer=Adam(lr=0.0001)	0.9290

Como podemos observar, los resultados han sido más o menos los esperados. Los resultados de *benchmark* han estado un poco por debajo de lo que sale en la web, sobre todo los *RandomForest*. Estos últimos han obtenido un 4 % de porcentaje de acierto menos de lo esperado, y la razón puede ser por los parámetros no configurados (la aleatoriedad del algoritmo puede afectar, pero no tanto ni en todas las ejecuciones).

Los *SVC*, sin embargo, si han obtenido unos resultados a la altura. Si es cierto que también están por debajo de los valores máximos de la web, pero la diferencia apenas es de un 1 % de precisión. Lo más destacable de esto es que los parámetros no han afectado demasiado al resultado final entre unas ejecucione del mismo algoritmo y otras.



Si observamos la matriz confusión de uno de estos, podemos ver que la mayor parte de los fallos se cometen en las camisetas y camisas, donde es normal que se puedan confundir la una con la otra debido a su similitud. A veces, también encontramos confusión con algunos vestidos, los cuales supongo que si son cortos o no tienen una forma muy definida, se confunden también con alguna de estas dos.

Por otra parte, el *StackingClassifier* no ha obtenido un mal resultado. De hecho a sido mejor que el *RandomForest* y, puede ser que, con un poco de optimización de los algoritmos que lo componen, se llegue a valores más competitivos y que incluso puedan superar al *SVC*. El mayor inconveniente de este algoritmo es que es el que más tardará en ejecutarse.

Si pasamos a las redes neuronales, podemos confirmar que son el mejor algoritmo de clasificación de imágenes, como ya preveíamos en un principio. La que mejores resultados (con un **93 % de acierto**) nos ha dado ha sido la que he hecho a mano, y la cual me sirvió para clasificar el *MNIST* con un porcentaje de acierto muy alto. Junto a esto, tenemos la red pre-entrenada o *ResNet50*, que también ha oobtenido un porcentaje bastante alto. No me ha dado tiempo a depurar mucho ninguno de los parámetros, pero igual podíamos haber obtenido resultados mejores, aunque las pruebas realizadas no han llevado mucho tiempo de entrenamiento (bastante menos que, por ejemplo, *StackingClassifier*) y ha dado unos resultados bastante competentes.

4. Planteamiento de futuro

A largo plazo, tenemos varios planteamientos para enfocar este problema. Como siempre he dicho, lo mejor para mejorar un modelo, pienso que no es ni mejorar tus algoritmos lo máximo que puedas, ni el preprocesamiento de datos, etc., sino que es aumentar los datos que tenemos. Cuantas más muestras y más ejemplos del dataset tengamos, mejor va a funcionar nuestro algoritmo y más va a aprender de ellos. Esta opción no siempre es viable, y menos en nuestro caso, en el que, como estudiante, simplemente disponemos del dataset proporcionado y no tenemos ni la capacidad ni el tiempo de aumentarlo masivamente.

Dejando esto a un lado, las opciones más reales de trabajar con este dataset serían las siguientes. Lo primero que haría es utilizar la herramienta *ImageDataGenerator* para ampliar nuestro conjunto de datos. Ésta herramienta lo que hace es: utiliza nuestros datos de entrenamiento para generar unos nuevos a partir de ellos. ¿Cómo lo hace? Pues mediante la transformación de imágenes, ya sea cogiendo una parte de ella, rotándola, desplazándola, distorsionándola o deformado su contenido.

Otra opción es la de mejorar nuestros algoritmos. Por ejemplo, para los *ensemble* que hemos utilizado, haría una búsqueda exhaustiva de parámetros. Para el *RandomForest* igual utilizaría un *GridSearchCV*, función la cual le metes una serie de parámetros, los evalúa y te devuelve los más óptimos para nuestro conjunto de entrenamiento. También podemos establecerlos a mano, haciendo una serie de pruebas masivas con distintos valores y quedándonos con los mejores. Para las redes neuronales, no tenemos la opción del cambio de parámetros, sin embargo, tenemos más flexibilidad a la hora de crearlas, ya que somos nosotros los que establecemos las capas (en el caso de que la red no sea pre-entrenada). Aqui ya entra la investigación y la información que podemos recoger de sobre las redes neuronales. Se miran estudios, papers, publicaciones o ejemplos tanto de *FashionMNIST* como de *MNIST* (por su similitud) y se recogen ideas de qué experimentos han resultado más beneficiosos; y posteriormente se intentan adaptar a nuestros modelos o hasta modificándolos para ver si conseguimos mejorar nuestras redes.

Referencias

[1] Twitter *François Chollet*
https://twitter.com/fchollet/status/852594987527045120

[2] Keras *Sequential Model*
https://keras.io/guides/sequential_model/

[3] Keras *ResNet50*
https://keras.io/api/applications/resnet/#resnet50-function/

[4] Keras *optimizers*
https://keras.io/api/optimizers/

[5] Keras *ReduceLROnPlateau*
https://keras.io/api/callbacks/reduce_lr_on_plateau/

[6] Scikit-Learn. *SVC*
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

[7] Scikit-Learn. *RandomForestClassifier*
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier

[8] Scikit-Learn. *recall_ score*
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html

[9] Scikit-Learn. *precision_ score*
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

[10] *Should I normalize/standardize/rescale the data?*
http://www.faqs.org/faqs/ai-faq/neural-nets/part2/

[11] Wikipedia. *Sensitivity and specifity*
https://en.wikipedia.org/wiki/Sensitivity_and_specificity

[12] DataFlair. *Kernel Functions-Introduction to SVM Kernel & Examples*
https://data-flair.training/blogs/svm-kernel-functions/

[13] Isaac Changhau. *Loss Functions in Neural Networks*
https://isaacchanghau.github.io/post/loss_functions/

[14] MathWorks*Support Vector Machine*
https://es.mathworks.com/discovery/support-vector-machine.html

[15] Scikit-Learn. *cross_ val_ score*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

[16] Scikit-Learn. *train_ test_ split*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[17] Scikit-Learn. *Confusion Matrix*
https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html