Introducción a Keras para clasificación de imágenes

Visión por Computador

Anabel Gómez Ríos
anabelgrios@decsai.ugr.es

Universidad de Granada

25 de octubre de 2019

Índice

Índice

Keras

- Keras es una API de alto nivel para *Deep Learning* escrita en python.
- Puede correr sobre TensorFlow, Theano o CNTK, aunque van a dejar de desarrollar sobre Theano y CNTK. Nosotros vamos a usar como *back-end* TensorFlow.
- Actualmente está en la versión 2.3.0.
- Documentación disponible en su web: `https://keras.io/`.
- Código disponible en GitHub: `https://github.com/keras-team/keras`.

Índice

Lectura de imágenes

- Dado que el *back-end* es TensorFlow, el vector con las imágenes tendrá dimensión (x, y, z, w), donde x es el número de imágenes, y es la altura de las imágenes, z es la anchura de las imágenes y w es el número de canales de las imágenes (1 si están en escala de grises y 3 si están a color).
- Este vector de imágenes podremos tenerlo en memoria si todas las imágenes entran en ella (como es el caso de esta práctica). Si no entrasen, Keras tiene la función `flow_from_directory()` que va leyendo las imágenes poco a poco de un directorio y las descarta después de usarlas y antes de coger el siguiente *batch* de imágenes.
- En las funciones que se os proporcionan para hacer la práctica, hay funciones para leer las imágenes junto con sus etiquetas y transformarlas al formato necesario para Keras.

Índice

Fases principales

Las fases principales para crear, entrenar y usar un modelo para predicción son, por orden, las siguientes:

1. Definición del modelo.
2. Declaración del optimizador a usar.
3. Compilación del modelo.
4. Entrenamiento.
5. Predicción.

Índice

Definición del modelo

- En Keras hay dos clases para definir modelos de redes neuronales: `Sequential` y `Model`.
- `Sequential` fuerza a que todas las capas de la red vayan una detrás de otra de forma secuencial, sin permitir ciclos ni saltos entre las capas.
- `Model` permite cualquier tipo de red neuronal, incluyendo ciclos y saltos entre capas.
- La forma en la que se va construyendo la red en ambas clases es distinta.

Construcción del modelo

- Con `Sequential` podemos usar el método `add` directamente sobre el modelo, porque la nueva capa se añadirá directamente después de la última capa añadida.

```
my_model = Sequential()
my_model.add(Dense(50, input_shape = (32, 32, 3))
```

- Con `Model` tenemos que especificar sobre qué capa estamos añadiendo la nueva capa.

```
a = Input(shape(32,))
b = Dense(32)(a)
model = Model(inputs = a, outputs = b)
```

Construcción del modelo

- Es necesario definir siempre las dimensiones de entrada en la primera capa del modelo.
- En nuestro caso, vamos a hacer clasificación multiclase y definiremos como última capa una capa *fully connected* (`Dense` en Keras) con tantas neuronas como clases tenga el problema y una activación *softmax* para transformar las salidas de las neuronas en la probabilidad de pertenecer a cada clase.

$$\text{softmax}(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{N} \exp(z_j)} \quad , \tag{1}$$

donde $\mathbf{z}$ es el vector de salida de la capa `Dense` y $\text{softmax}(\mathbf{z})$ es el vector que contiene en la componente j la probabilidad de que la imagen pertenezca a la clase j, para $j = 1, ..., N$, con N el total de clases.

Construcción del modelo

- En Keras, las funciones de activación pueden introducirse en cualquier punto de la red con la capa `Activation`. Por ejemplo, las funciones tangente hiperbólica o ReLU:

```
Activation('tanh')
Activation('relu')
```

- Sin embargo, lo más usual es introducirlas detrás de cualquier capa usando el argumento `activation` de esa capa. Lo siguiente introduciría una activación ReLU después de la capa `Dense`:

```
Dense(50, activation = 'relu')
```

Capas disponibles en Keras

En las prácticas vamos a usar algunas de las siguientes capas:

- *Fully connected*: `Dense(units, activation = None, ...)`
- *Dropout*: `Dropout(rate, noise_shape = None, seed = None)`
- *Flatten*: `Flatten()`
- Convolución 2D: `Conv2D(filters, kernel_size, strides = (1,1), padding = 'valid', activation = None, ...)`
- *Pooling* 2D: `MaxPooling2D(pool_size = (2,2), strides = None, ...)`. Equivalentemente, `AveragePooling2D()`, `GlobalMaxPooling()`, `GlobalAveragePooling()`,...
- *Batch Normalization*: `BatchNormalization()`.

Capas disponibles en Keras

La documentación de las capas anteriores, y más, puede encontrarse en:

- https://keras.io/layers/core/
- https://keras.io/layers/convolutional/
- https://keras.io/layers/pooling/
- https://keras.io/layers/normalization/

Construcción del modelo

Una vez el modelo está construido, podemos ver una descripción del mismo usando `summary` sobre el objeto creado:

```
my_model.summary()
```

Índice

Declaración del optimizador

- Para poder modificar los parámetros del optimizador a usar, es necesario declararlo previamente y crear un objeto. Por ejemplo, para usar el gradiente descendente estocástico deberíamos declararlo y así podríamos cambiar alguno de los parámetros. Si no, usaría los que tiene por defecto:

```
from keras.optimizers import SGD
opt = SGD(lr = 0.01, decay = 1e-6, momentum = 0.9,
          nesterov = True)
```

- La documentación de los optimizadores está en https://keras.io/optimizers/.

Índice

Compilación del modelo

- Para compilar un modelo, es necesario definir la **función de pérdida** o **función objetivo** que se va a usar (la que se va a minimizar). Esta función depende del problema que se esté resolviendo. En el caso de clasificación binaria, se suele usar `binary_crossentropy` y en el caso de clasificación multiclase se suele usar `categorical_crossentropy`.
- La documentación completa con todas las funciones de pérdida disponibles está en https://keras.io/losses/.

Compilación del modelo

- En la compilación también se puede especificar con el argumento `metrics` las métricas que se quieren calcular a lo largo de todas las épocas de entrenamiento. Por ejemplo, para problemas de clasificación multiclase, es común usar la métrica *accuracy*, definida como el porcentaje de imágenes bien clasificadas.
- Para compilar, usamos el método `compile()`, disponible en las clases `Sequential` y `Model`:

```
my_model.compile(optimizer,
                 loss = 'categorical_crossentropy',
                 metrics = ['accuracy'], ...)
```

Índice

Entrenamiento

- Una vez el modelo está compilado, podemos pasar a entrenarlo. Para ello, se puede usar el método `fit()` o el método `fit_generator()`.
- El primero recibe las imágenes de entrenamiento directamente y el segundo un generador que será el que se encargará de ir generando las imágenes.
- El uso de uno u otro dependerá de si todas las imágenes están cargadas en memoria y no vamos a usar la clase `ImageDataGenerator` (`fit()`) o de si las imágenes no están todas en memoria o se va a usar `ImageDataGenerator` (bien para *data augmentation*, para usar alguna función de preprocesamiento o para separar un conjunto de validación durante el entrenamiento) (`fit_generator()`).

Entrenamiento

- Cuando se entrena un modelo con `fit()` o `fit_generator()`, Keras guarda el estado del modelo por donde se ha quedado entrenando. Esto quiere decir que si volvemos a usar una de las funciones anteriores, el entrenamiento seguirá por donde se ha quedado, y no empezará desde el principio.
- Por esta razón, si vamos a usar varias veces `fit()` o `fit_generator()` sobre el mismo modelo definido previamente, con distintos argumentos en `ImageDataGenerator` para, por ejemplo, probar distintos tipos de data augmentation, tenemos que reestablecer los pesos de la red a como estaban antes del entrenamiento.

Entrenamiento

- Esto se puede hacer guardando los pesos de la red antes del primer entrenamiento (y después de la compilación) usando

```
weights = my_model.get_weights()
```

y después reestablecerlos antes del siguiente entrenamiento usando

```
my_model.set_weights(weights)
```

La clase `ImageDataGenerator`

- Esta clase nos permite normalizar los datos (bien con media y varianza, o usando una función de preprocesamiento), usar *data augmentation* y separar del conjunto de entrenamiento una parte para validación (entre otras cosas).
- Para usarla, tenemos que crear un objeto de esta clase y usarlo como generador de imágenes a la hora de entrenar y/o testear el modelo.
- *Data augmentation* sólo debe usarse en el conjunto de entrenamiento. La normalización de las imágenes debe hacerse en ambos conjuntos, pero la normalización del conjunto de test debe hacerse con los parámetros de las imágenes de entrenamiento.
- La documentación de esta clase se encuentra en `https://keras.io/preprocessing/image/`.

La clase ImageDataGenerator

- Si se usan los argumentos `featurewise_center` y/o `featurewise_std_normalization` para normalizar con media 0 y varianza 1 los conjuntos de datos, es necesario usar la función `fit()` sobre el generador creado:

```
datagen = ImageDataGenerator(featurewise_center=True,
            featurewise_std_normalization = True)
datagen.fit(imagenes_train)
```

La clase `ImageDataGenerator`: Separar un conjunto de validación

- Para separar un 10 % del conjunto de entrenamiento para validación, usaremos la clase `ImageDataGenerator`. Con ella, definiremos un generador `datagen` que se encargará de generar las imágenes de entrenamiento:

```
datagen = ImageDataGenerator(validation_split = 0.1)
```

- Si hemos especificado el argumento `validation_split`, cuando estemos generando imágenes con la función `flow()` sobre el generador, podremos especificar si queremos generar el conjunto de entrenamiento o el de validación:

```
datagen.flow(imagenes_train, etiquetas_train,
             batch_size = 32,
             subset = 'training')
```

La clase ImageDataGenerator: Usar fit_generator con el conjunto de validación

- Este datagen.flow() será el primer argumento de la función fit_generator.
- Tendremos que usar también el argumento validation_data para especificar el conjunto de validación y hacerlo usando el mismo generador pero especificando en subset que queremos generar el conjunto de validación:

```
my_model.fit_generator(datagen.flow(imagenes_train,
                                    etiquetas_train,
                                    batch_size = 32,
                                    subset = 'training'),
                       validation_data = datagen.flow(
                                    imagenes_train,
                                    etiquetas_train,
                                    batch_size = 32,
                                    subset = 'validation'))
```

La clase `ImageDataGenerator`: Usar `fit_generator`

- `fit_generator` tiene otros tres parámetros a tener en cuenta: `steps_per_epoch`, `epochs` y `validation_steps`.
- `steps_per_epoch` fija el número de *batches* de imágenes que se usan antes de terminar una época del entrenamiento y pasar a la siguiente. Se suele fijar al número de imágenes en entrenamiento entre el tamaño de cada *batch*.
- `epochs` fija el número de épocas durante las que se entrena la red.
- `validation_steps` fija el número de batches de imágenes de validación que se generan al final de cada época. Se suele fijar al número de imágenes en validación entre el tamaño de cada `batch`.

Usar `fit_generator`

Finalmente, una llamada a `fit_generator` para entrenar un modelo debe quedar parecido a lo siguiente:

```
my_model.fit_generator(datagen.flow(imagenes_train,
                                    etiquetas_train,
                                    batch_size = 32,
                                    subset = 'training'),
                       steps_per_epoch = len(imagenes_train)*0.9/32,
                       epochs = 30,
                       validation_data = datagen.flow(
                                    imagenes_train,
                                    etiquetas_train,
                                    batch_size = 32,
                                    subset = 'validation'),
                       validation_steps = len(imagenes_train)*0.1/32)
```

La clase `ImageDataGenerator`

- En el modelo final, la llamada a la clase `ImageDataGenerator` para la creación de los objetos `datagen` (uno para entrenamiento y otro para test) tendrá varios parámetros:
  - El generador de train tendrá los parámetros correspondientes a la normalización de los datos de entrada que se use, el *data augmentation* que se use y el porcentaje que se guarde para validación.
  - El generador de test sólo tendrá la normalización (que se hará con los parámetros obtenidos de las imágenes de train).

Índice

Predicción

- Una vez la red ha terminado de entrenar, podemos usarla para predecir la clase de nuevas imágenes. Para ello, se usan las funciones `predict_generator()` o `predict()` de las clases `Model` y `Sequential`, en función de si se está usando un generador para el conjunto de test o no, de forma análoga a `fit()`.
- Si se usa `predict_generator()` es necesario usar los argumentos `shuffle = False` y `batch_size = 1` de `flow` para que las predicciones de las nuevas imágenes estén en el mismo orden que las imágenes de test, y se pueda comparar con las etiquetas reales.

```
predicciones = my_model.predict_generator(
                    datagen_test.flow(imagenes_test,
                    batch_size = 1,
                    shuffle = False),
                    steps = len(imagenes_test))
```

Predicción: Cálculo de accuracy

- Una vez tenemos las predicciones hechas, podemos calcular el porcentaje de imágenes de test que el modelo ha clasificado bien (la métrica *accuracy*).
- Para ello, en las funciones dadas para la práctica, se os proporciona la función `calcularAccuracy(labels, preds)`, donde `labels` es el vector de las etiquetas de test reales y `preds` es el vector de predicciones devuelto por `fit_generator`.

Índice

Redes preentrenadas

- Keras tiene algunas de las redes más famosas y usadas ya creadas, por lo que no es necesario construirlas desde cero cada vez.
- Además, también están preentrenadas en ImageNet, de forma que si se quiere, se puede partir el entrenamiento desde ahí.
- Estos modelos están en Keras Applications: `https://keras.io/applications/`.
- Se dispone, por ejemplo, de VGG19, ResNet50 o InceptionV3.
- Cada red dispone de una función de preprocesado `preprocess_input()` distinta, que habrá que usar con cada modelo. Se le puede pasar como argumento al generador de la clase `ImageDataGenerator`.
- Son instancias de la clase `Model` porque no son, por lo general, redes secuenciales.

Redes preentrenadas: Ejemplo

- Supongamos que queremos cargar ResNet50 ya preentrenada en ImageNet. Tenemos que quitarle la última capa de 1000 neuronas (usando el argumento `include_top` al cargar la red):

```
from keras.applications.resnet import ResNet50,
      preprocess_input
resnet50 = ResNet50(include_top = False,
      weights = 'imagenet', pooling = 'avg')
```

- El argumento `pooling = 'avg'` introduce un `GlobalAveragePooling` después de lo que ahora sería la última capa, que es una convolución 2D, por lo que su salida son dos dimensiones. Esto lo convierte a una dimensión añadiendo además un *pooling*. Otra opción sería usar `Flatten()`.

Índice

Usar una red preentrenada como un extractor de características

- Si usamos el modelo `resnet50` que acabamos de crear, preentrenado en ImageNet y habiéndole quitado la última capa de 1000 neuronas que clasificaba las imágenes de entrada en las clases de ImageNet, lo podemos usar como un extractor de características (usando `predict_generator()`. En concreto, la que ahora es la última de nuestro modelo, es una capa con 2048 neuronas. Por tanto, podemos considerar que estamos transformando cada imagen en un vector de 2048 características.
- Con este vector de características podríamos entrenar otro modelo, como un SVM clásico o una red con varias capas *fully connected* (un perceptrón multicapa).

Índice

Fine-tuning

- Otra opción sería reentrenar la red entera para que clasifique nuestro problema.
- Para ello, como mínimo, es necesario añadir al final del modelo una capa *fully connected* con tantas neuronas como clases y activación *softmax*.
- Normalmente, se suelen añadir más capas *fully connected* antes de la última con activación softmax.
- Para esto es necesario usar la clase `Model` porque ResNet50 es una instancia de esta clase.

```
x = resnet50.output
x = Dense(32, activation = 'relu')(x)
last = Dense(10, activation = 'softmax')(x)
new_model = Model(inputs = resnet50.input,
                  outputs = last)
```