Elaborazione di Segnali Multimediali

Classificazione di immagini tramite fine-tuning

L.Verdoliva, D.Cozzolino

In questa esercitazione realizzeremo il *fine-tuning*, che permette di classificare immagini anche quando il dataset disponibile non è sufficientemente grande. L'idea è quella di sfruttare reti già addestrate su dataset molto grandi per estrarre le feature di basso livello, e apprendere solo i pesi degli strati che sono relativi alle feature di alto livello. E' una forma molto semplice di *transfer learning* che risulta comunque molto efficace. Applicheremo questa strategia alla classificazione di immagini. Per questa esercitazione useremo anche le funzione delle librerie `keras_cv` e `easy_cv_dataset` che vanno installate con la seguente istruzione:

```
!pip install --upgrade git+https://github.com/davin11/easy-cv-dataset keras-cv
```

1 Preparazione dei dati

Utilizzeremo il dataset TrafficSigns, quindi l'obiettivo è identificare se nell'immagine sia una segnale stradale di tipo: obbligo, divieto o pericolo. Su Colab potete direttamente eseguire le seguenti istruzioni per scaricare e decomprimere il dataset:

```
!wget -q -c https://www.grip.unina.it/download/guide_TF/TrafficSigns.zip
!unzip -q -n TrafficSigns.zip
```

A questo punto vi troverete una cartella denominata "TrafficSigns" che contiene due sotto-cartelle "train" e "test". Entrambe le cartelle "train" e "test" contengono le cartelle "mandatory", "prohibition", e "warning". Visualizzate un'immagine per ogni tipo di segnale stradale con il seguente codice:

```
img0 = io.imread('TrafficSigns/train/mandatory/img0065_00.png')
img1 = io.imread('TrafficSigns/train/prohibition/img2929_00.png')
img2 = io.imread('TrafficSigns/train/warning/img2470_00.png')

plt.figure(figsize=(15,5))
plt.subplot(1,3,1); plt.imshow(img0); plt.title('mandatory')
plt.subplot(1,3,2); plt.imshow(img1); plt.title('prohibition')
plt.subplot(1,3,3); plt.imshow(img2); plt.title('warning')
plt.show()
```

Procediamo a preparare i dati tramite le funzioni di `easy_cv_dataset.image_dataframe_from_directory` che semplificano tale operazione. Avendo le immagini organizzate in cartelle ognuna delle quali è associata ad una classe, possiamo ottenere un elenco dell'immagini con le relative classi specificando solo la cartella padre:

```
import easy_cv_dataset as ds
train_table = ds.image_dataframe_from_directory('TrafficSigns/train')
```

L'istruzione precedente crea una tabella, `train_table`, che ha due colonne, la colonna *image* contenente i filepath delle immagini di training e la colonna *class* contiene l'informazione della classe per ogni immagine. In Colab, possiamo avere una visualizzazione grafica della tabella eseguendo la seguente istruzione:

```
display(train_table)
```

Dividiamo la tabella di training in due tabelle rispettivamente per effettivo addestramento e per la validazione tramite la funzione `train_test_split`:

```
from sklearn.model_selection import train_test_split
train_table, valid_table = train_test_split(train_table, test_size=0.2,
                                    random_state=34,
                                    stratify=train_table['class'])
```

Alla funzione dobbiamo indicare il parametro `test_size` per indicare la percentuale dei dati di validazione pari al 20%, mentre il parametro `stratify=train_table['class']` serve per garantire un divisione equilibrata tra le classi. Adesso, utilizzando la funzione `image_classification_dataset_from_dataframe`, possiamo indicare come preparare le immagini del training-set.

```
batch_size = 8
img_height, img_width = 150, 150

from keras_cv.layers import Resizing
train_dataset = ds.image_classification_dataset_from_dataframe(
                        train_table, batch_size=batch_size, shuffle=True,
                        pre_batching_processing=Resizing(img_height, img_width),
                        post_batching_processing=None,
                        do_normalization=True,
                        class_mode='categorical')
```

La funzione `image_classification_dataset_from_dataframe` richiede come primo parametro la tabella precedentemente creata. Il secondo parametro `batch_size` indica la dimensione del batch che verrà utilizzata durante il training e deve essere specificato per indicare quante immagini devono essere elaborate insieme. Il parametro `shuffle` indica se il dataset deve essere rimescolato all'inizio di ogni epoca. I parametri `pre_batching_processing` e `post_batching_processing` permettono di applicare una operazione a tutte le immagini del dataset, rispettivamente prima e dopo la costruzione del batch. Quando si vogliono elaborare immagini con diversa risoluzione, possiamo indicare a `pre_batching_processing` un operazione di ridimensionamento delle immagini ad una dimensione data. Il parametro `do_normalization`, se è settato a `True`

riporta tutte l'immagine nel dinamica [0,1]. L'ultimo parametro `class_mode` serve per indicare il formato delle etichette: con la stringa `'categorical'` si utilizzerà il formato OneHot.

Prepariamo anche i dati del dataset di validazione:

```
valid_dataset = ds.image_classification_dataset_from_dataframe(
                    valid_table, batch_size=batch_size, shuffle=False,
                    pre_batching_processing=Resizing(img_height, img_width),
                    do_normalization=True,
                    class_mode='categorical')
```

2 Fine-tuning

Il fine-tuning consiste nell'addestrare una rete non partendo da pesi random, ma utilizzando i pesi già ottenuti da un pre-addestramento su un altro dataset (molto più grande di quello che si ha a disposizione). In particolare, si usano i pesi relativi ai primi strati che estraggono feature comuni a molte immagini (feature di basso livello) e si riaddestrano gli strati successivi, che estraggono feature specifiche dell'applicazione in esame.

Utilizziamo allora la rete ResNet50 pre-addestrata su ImageNet:

```
# ResNet50 pre-addestrata
base_model = keras.applications.ResNet50(weights='imagenet', include_top=False,
                                  input_shape=(img_width, img_height, 3))
```

Il parametro `include_top=False` indica che gli ultimi livelli non vengono istanziati e vanno definiti in base al problema da affrontare. Definiamo allora un modello che include gli ultimi livelli costituiti da un layer di AveragePooling e un FullyConnected.

```
from tensorflow.keras import layers

model = keras.models.Sequential()
model.add(base_model)
model.add(layers.GlobalAveragePooling2D())
model.add(layers.Dense(3, activation='softmax'))
```

Il fatto di non addestrare i primi strati riduce i parametri da apprendere e anche il rischio di over-fitting. Per bloccare i parametri dei primi 25 strati di ResNet50 utilizzate il seguente codice:

```
train_after_layer = 25
for layer in base_model.layers[:train_after_layer]:
    layer.trainable = False
```

Definiamo l'ottimizzatore a la funzione di loss:

```
model.compile(loss='categorical_crossentropy',
         optimizer=keras.optimizers.SGD(learning_rate=1e-4, momentum=0.9),
         metrics=['accuracy', ])
```

Possiamo utilizzare un learning rate anche molto basso poiché non partiamo da pesi random. Utilizzare il metodo `fit` per il training:

```
model.fit(train_dataset, epochs=1, validation_data=valid_dataset, verbose=True)
```

Notate che con una singola epoca riusciamo ad ottenere un'accuratezza sui dati di validazione superiore al 90%.

3 Analisi delle prestazioni

Utilizzando le funzioni `image_dataframe_from_directorye` `image_classification_dataset_from_dataframe` per preparare le immagini del test-set.

```
test_table = ds.image_dataframe_from_directory('TrafficSigns/test')
test_dataset = ds.image_classification_dataset_from_dataframe(
                     test_table, batch_size=batch_size, shuffle=False,
                     pre_batching_processing=Resizing(img_height, img_width),
                     do_normalization=True,
                     class_mode='categorical')
```

Per valutare le prestazioni sul test set eseguite il codice:

```
test_loss, test_accuracy = model.evaluate(test_dataset, verbose=True)
print('Test loss:', test_loss)
print('Test accuracy:', test_accuracy)
```

4 Data Augmentation

Possiamo prevedere per l'immagini di training varie operazioni random che permettono di aumentare l'insieme di dati di training in modo artificiale (*data augmentation*). A tal fine possiamo utilizzare le operazione random definita nella libraria `keras_cv.layers` ed eventualmente combinarle insieme tramite `keras.Sequential`:

```
from keras_cv.layers import RandomBrightness, RandomRoom

augmenter = keras.Sequential(layers=[
   RandomBrightness(factor=(-0.1, 0.1), value_range=(0, 255)),
   RandomRoom((-0.2,0.2)),
])
```

Ad esempio nel codice precedete, abbiamo definito `augmenter` che è la sequenza di un cambiamento di luminosità con un fattore additivo random tra -25.5 e 25.5, un ridimensionamento un fattore di scale random tra 0.8 e 1.2. La sequenza di operazioni, `augmenter`, va passata al parametro `post_batching_processing` della funzione `image_classification_dataset_from_dataframe`.

```
train_dataset = ds.image_classification_dataset_from_dataframe(
train_table, batch_size=batch_size, shuffle=True,
pre_batching_processing=Resizing(img_height, img_width),
post_batching_processing=augmenter,
do_normalization=True,
class_mode='categorical')
```

5 Esercizi proposti

1. *Architecture* Provate il *fine-tuning* utilizzando altre reti pre-addestrate, come Xception, InceptionV3 e EfficientNetB4. Nel primo caso usate la funzione `keras.applications.Xception` bloccando i primi 25 strati, nel secondo caso invece usate la funzione `keras.applications.InceptionV3` bloccando i primi 50 strati enn l'ultimo usate la funzione `keras.applications.EfficientNetB4` bloccando i primi 200 strati. Per maggiori informazioni consulta la pagina web: https://keras.io/api/applications/

2. *Augmentation* Provate il *fine-tuning* aggiungendo anche ulteriori operazioni per l'augmentation allo scopo di migliorare la robustezza. Considerate rotazioni e traslazioni random utilizzando i layer descritti alla pagina web: https://keras.io/api/keras_cv/layers/

3. *Evitare l'over-fitting*. Addestrare per molte epoche può portare ad una riduzione delle prestazioni sulla validazione a cause dell'over-fitting. Per evitare questo problema, Keras prevede la possibilità di conservarsi i pesi delle rete relativa all'epoca migliore cioè quella con migliore accuratezza valutata sul set di validazione. A tal fine seguire la documentazione ufficiale di Keras alla pagina web: https://keras.io/api/callbacks/model_checkpoint/