Elaborazione di Segnali Multimediali

Classificazione di immagini mediante deep learning

L.Verdoliva, D.Cozzolino

In questa esercitazione sviluppiamo un approccio basato sulle Reti Neurali Convoluzionali per la classificazione di immagini. A tal fine useremo la libreria *Keras* che offre un insieme di funzioni che facilitano varie operazioni come: la preparazione dei dati, la definizione dell'architettura della rete neurale, l'addestramento e l'analisi delle prestazioni. Inoltre, le principali funzioni di Keras girano su GPU Nvidia riducendo enormemente i tempi di esecuzione. Per questa esercitazione utilizzeremo il servizio Colaboraty (o Colab per brevità) di Google che permette di eseguire codice Python per un tempo limitato sui server di Google che sono muniti di GPU. Se avete una GPU Nvidia sul vostro computer, potete anche installare la libreria Keras nell'environment del corso scrivendo il seguente comando nel Prompt di Anaconda:

```
conda install -n corso tensorflow-gpu keras=2.3.1
```

Colab permette di eseguire codice Python in un qualsiasi browser web, inoltre non richiede una configurazione in quanto già risultano installate le principali librerie Python incluse le librerie per il deep learning come Keras. Per utilizzare Colab, avete bisogno di un account Google e di attivare il servizio alla pagina: https://gsuite.google.com/u/1/marketplace/search/Colaboratory . Una volta attivato il servizio, potete creare un nuovo Notebook Colab all'interno di una cartella di Google Drive. Al tal fine, accedete tramite browser a Google Drive (https://drive.google.com/drive/my-drive), create una cartella per il corso in cui salverete i vari Notebook, e selezionate `"Google Colaboratory"` dal menu `"New"` per creare il vostro primo Notebook Colab. Un Notebook Python rispetto a uno script è organizzato in celle. Ogni cella può essere eseguita singolarmente e il relativo output apparirà al di sotto della cella nella stessa pagina web. Per poter eseguire codice sulla GPU dovete cambiare le impostazioni del Notebook accedendo a `"Notebook settings"` presente nel menù a tendina `"Edit"`.

1 Classificazione usando LeNet

Come primo esempio di classificazione affrontiamo il problema del riconoscimento delle cifre da 0 a 9 scritte a mano e usiamo il dataset MNIST composto da un totale di 70.000 immagini su scala di grigi di 28×28 pixel. Alcune immagini di esempio sono riporte in figura 1. Per comodità dividiamo il codice da scrivere in quattro sezioni:

1. caricamento e preparazione dei dati;
2. definizione dell'architettura;
3. addestramento;
4. analisi delle prestazioni.

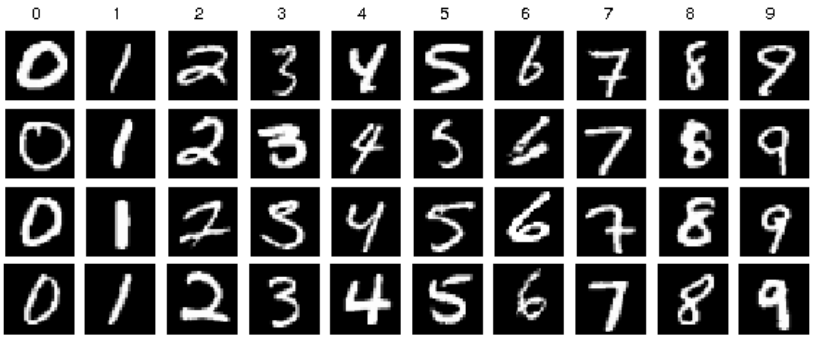

Figura 1: Esempi di immagini del dataset MNIST.

.

1.1 Caricamento e preparazione dei dati

Per caricare il dataset MNIST usiamo la seguente funzione di Keras:

```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Il dataset risulta già suddiviso in due parti: un training set di 60.000 immagini e un test set di 10.000 immagini. Per ogni insieme sono state caricate due variabili. Ad esempio, per il training set si ha la variabile `y_train` che è il vettore di 60.000 etichette e la variabile `x_train` che è un array contentente le 60.000 immagini. Separiamo le 60.000 immagini in due insiemi: uno composto da 5.000 immagini che servono per la validazione (validation set) e uno formato dalle restanti 55.000 immagini, che sono utilizzate per il training.

```
x_val   = x_train[::12]
y_val   = y_train[::12]
x_train = np.delete(x_train, np.arange(0, x_train.shape[0], 12), 0)
y_train = np.delete(y_train, np.arange(0, y_train.shape[0], 12), 0)
```

A questo punto bisogna preparare i dati affinché siano compatibili con il formato dell'input alla rete. In particolare, Keras richiede che le immagini siano `float` a 32 bit e adotta di default la convenzione Channels-Last. Nella convenzione Channels-Last le immagini sono organizzate in un array di 4 dimensioni: la prima dimensione serve ad accedere ad un'immagine del dataset, poi riga e colonna ed infine la banda (o canale) dell'immagine.

```
img_rows = 28;  img_cols = 28; img_channels = 1;

# reshape nel formato Channels-Last
x_train = np.reshape(x_train, (55000,img_rows,img_cols,img_channels))
x_val   = np.reshape(x_val  , ( 5000,img_rows,img_cols,img_channels))
x_test  = np.reshape(x_test , (10000,img_rows,img_cols,img_channels))
```

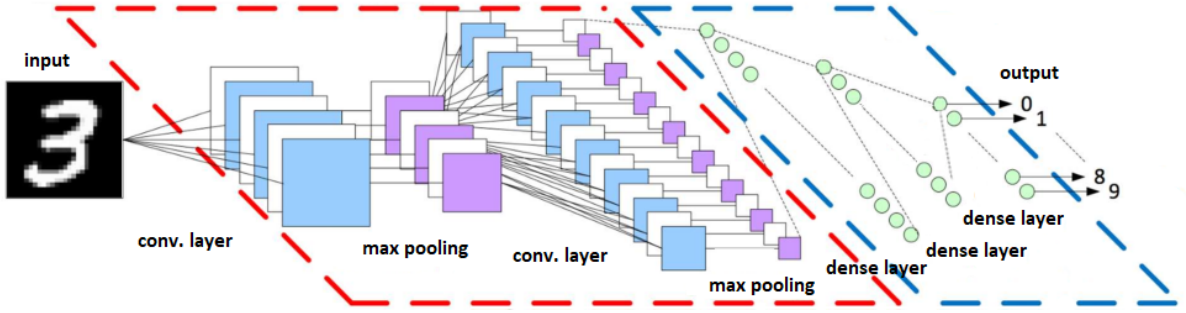

Figura 2: Architettura LeNet.

.

```
# conversione a float e normalizzazione
x_train = np.float32(x_train) / 255
x_val   = np.float32(x_val)   / 255
x_test  = np.float32(x_test)  / 255
```

Notate che per il dataset MNIST il numero di canali è pari ad uno perché le immagini sono su scala di grigi. Anche le etichette vanno preparate opportunamente, per questa esercitazione useremo il formato One-Hot. Nel formato One-Hot ogni etichetta viene rappresentata come un vettore la cui lunghezza è uguale al numero di classi presenti: il vettore conterrà tutti valori nulli eccetto un singolo 1 nella posizione relativa all'etichetta (per esempio, se il numero di classi è 10, allora l'etichetta 8 sarà convertita nella sequenza 0000000010, l'etichetta 0 in 1000000000). Usiamo la funzione `keras.utils.to_categorical` per convertire il vettore di etichette nel formato One-Hot.

```
num_classes = 10

# conversione delle etichette nel formato one-hot
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train, num_classes)
y_val   = to_categorical(y_val  , num_classes)
y_test  = to_categorical(y_test , num_classes)
```

1.2 Definizione dell'architettura della rete neurale

Vediamo adesso come si definisce l'architettura di rete e implementiamo LeNet rappresentata in figura 2. Notate che l'architettura da definire è formata da una sequenza di strati (o layer) di diverso tipo. Per definire ogni tipo di layer la libreria Keras mette a disposizione una funzione del modulo `keras.layers`. Iniziamo ad istanziare l'oggetto `keras.models.Sequential` che rappresenta l'intera architettura ed a cui aggiungeremo man mano i vari strati tramite il metodo `add` (L'elenco degli strati è presenti in Tab. 1).

```
from tensorflow import keras
lenet = keras.models.Sequential()
```

	Tipo	Supporto Spaziale	Numero di Filtri	Attivazione
1	Convoluzionale	5×5	6	ReLU
2	Max Pooling	2×2	-	-
3	Convoluzionale	5×5	16	ReLU
4	Max Pooling	2×2	-	-
5	Fully Connected	-	120	ReLU
6	Fully Connected	-	84	ReLU
7	Fully Connected	-	10	Softmax

Tabella 1: Elenco degli strati dell'architettura LeNet.

Utilizziamo il metodo `Conv2D` per creare il primo layer che è uno strato convoluzionale:

```
from tensorflow.keras import layers
lenet.add(layers.Conv2D(6, (5, 5), padding='same', activation='relu',
                        input_shape=(img_rows,img_cols,img_channels)))
```

Il primo parametro del metodo `Conv2D` indica il numero di filtri, mentre il secondo indica la dimensione dei filtri. L'opzione `padding` può essere impostata a `'same'` per effettuare lo zero-padding o `'valid'` per non effetuare padding. Il parametro `activate` indica la funzione di attivazione (non lineare) da applicare dopo l'operazione di convoluzione. Infine il parametro `input_shape` serve per indicare le dimensioni dell'immagine di ingresso ed è obbligatorio solo per il primo layer della rete. Il successivo layer, che effettua il max-pooling, dobbiamo invece crearlo con il metodo `MaxPooling2D` fornendogli la dimensione della finestra di pooling:

```
lenet.add(layers.MaxPooling2D((2, 2) ))
```

Procediamo in maniera analoga per il successivo strato convoluzionale e quello di pooling:

```
lenet.add(layers.Conv2D(16, (5, 5), padding="valid", activation='relu'))
lenet.add(layers.MaxPooling2D((2, 2) ))
```

A questo punto bisogna convertire l'immagine in un vettore: si passa da un array di dimensioni $5 \times 5 \times 16$ ad un vettore di 400 elementi. A tal fine utilizziamo il layer `Flatten`:

```
lenet.add(layers.Flatten())
```

Creiamo gli ultimi tre strati della rete, di tipo Fully-Connected, con il metodo `Dense`:

```
lenet.add(layers.Dense(120, activation='relu'))
lenet.add(layers.Dense(84, activation='relu'))
lenet.add(layers.Dense(num_classes, activation='softmax'))
```

Il primo parametro del metodo `Dense` indica il numero di feature all'uscita del layer, mentre il parametro `activation` indica la funzione di attivazione da applicare. Keras permette di creare molti tipi di strati diversi, per una documentazione completa consultate il sito ufficiale https://keras.io/api/layers . Notate che la rete ha in uscita un numero di valori pari al numero di classi, inoltre la funzione di attivazione *softmax* normalizza l'uscita in modo che ogni valore sia non negativo e che la loro somma sia pari ad 1. In sostanza, le uscite della rete rappresentano la probabilità che viene assegnata ad ogni classe. Per esempio, se la rete analizza l'immagine MNIST raffigurante la cifra 3, la rete dovrebbe assegnare alla classe 3 una probabilità alta ed un probabilità bassa alle altre classi. Per visualizzare una tabella riassuntiva dell'architettura, eseguite la seguente istruzione:

```
lenet.summary()
```

1.3 Addestramento

Durante l'addestramento, le immagini del training set sono divise in gruppi chiamati *batch*. Un'iterazione di addestramento prevede di analizzare un singolo *batch* e di modificare i parametri della rete secondo un predefinito algoritmo di ottimizzazione. L'obiettivo dell'ottimizzazione è minimizzare un indice di distanza tra l'uscita della rete e l'uscita desiderata. Pertanto, prima di cominciare l'addestramento bisogna definire la distanza adottata, chiamata *loss function*, e l'algoritmo di ottimizzazione utilizzato. In questa esercitazione useremo la cross-entropy loss come *loss function* e lo Stochastic Gradient Descent (SGD) come algoritmo di ottimizzazione. Utilizziamo il metodo `compile` per definirli:

```
lenet.compile(loss = keras.losses.categorical_crossentropy,
    optimizer = keras.optimizers.SGD(lr=0.01),
    metrics = ['accuracy'])
```

Il metodo `compile` permette anche di definire con il parametro `metrics` una lista di indici prestazionali da visualizzare durante l'addestramento. Notate che per lo Stochastic Gradient Descent abbiamo dovuto indicare anche il learning rate. Adesso possiamo eseguire il training tramite il metodo `fit`

```
lenet.fit(x_train, y_train, batch_size=200, epochs=10,
     validation_data=(x_val, y_val), verbose=True)
```

I primi due parametri del metodo `fit` sono le immagini di training e le relative etichette secondo il formato visto precedentemente. Il parametro `batch_size` serve per indicare il numero di immagini presenti in ogni *batch*. Mentre, il parametro `epochs` serve per fissare il numero di epoche di addestramento. Dopo un'epoca di addestramento tutte le immagini del training set sono state analizzate. Alla fine di ogni epoca vengono calcolate le prestazioni sui dati di validazione forniti con il parametro `validation_data`. L'opzione verbose=True serve solo a mostrare una barra progressiva durante l'addestramento.

1.4 Analisi delle prestazioni

Prima di eseguire un'analisi delle prestazioni, testiamo la rete su una singola immagine.

```
ID = 1000 # indice dell'immagine da analizzare
img   = x_test[ID]
label = y_test[ID]
img   = np.reshape(img, (-1,img_rows,img_cols,img_channels))
pred  = lenet.predict(img)

print('vettore delle probabilità predette:', pred)

plt.figure()
plt.imshow(img[0,:,:,0], cmap='gray', clim=[0,1])
plt.title('Etichetta vera: %d; Etichetta predetta: %d' %
           (np.argmax(label), np.argmax(pred)))
plt.show()
```

Per valutare le prestazioni sul test set eseguite il codice:

```
test_loss, test_accuracy = lenet.evaluate(x_test, y_test, verbose=True)
print('Test loss:', test_loss)
print('Test accuracy:', test_accuracy)
```

1.4.1 Esercizi proposti

1. *Learning-Rate.* Ripetete l'esercitazione cambiando il learning rate. Vedete cosa succede sia con un learning rate più alto (lr=1.0) che con un learning rate più basso (lr=0.0001). Fate attenzione che il metodo `compile` può essere eseguito una singola volta per ogni architettura definita, quindi per cambiare il learning rate dovete anche ri-eseguire il codice di definizione dell'architettura.

2. *Dataset CIFAR-10.* Utilizzate LeNet per la classificazione di oggetti adoperando il dataset CIFAR-10 composto da 60.000 immagini 32×32 a colori di 10 classi (Aereo, Automobile, Uccello, Gatto, Cervo, Cane, Rana, Cavallo, Barca, Camion). Potete caricare il dataset utilizzando la funzione `keras.datasets.cifar10.load_data`.

2 ImageNet

In Keras sono disponibili molte reti per il riconoscimento di oggetti già addestrate sul dataset ImageNet (http://www.image-net.org/). ImageNet è un ampio dateset che consiste in più di 14 milioni di immagini annotate manualmente in differenti categorie. Proviamo ad utilizzare la rete Xception sull'immagine 'test.jpg', eseguendo il seguente codice:

```
# Caricamento dell'immagine
import skimage.io as io
from skimage.transform import resize
img = np.float32(io.imread('test.jpg'))/255
img = resize(img, (299, 299), order=1)
```

```
# Caricamento della rete
from keras.applications.xception import Xception
from keras.applications.xception import preprocess_input, decode_predictions
model = Xception(weights='imagenet')

# Preparazione dei dati
x = np.reshape(img, (1,img.shape[0],img.shape[1],img.shape[2]))
x = preprocess_input(np.copy(x))

# Predizione
preds = model.predict(x)

# Visualizzazione del risultato
plt.figure(); plt.imshow(img/255); plt.show();
preds_d = decode_predictions(preds, top=5)[0]
for p in preds_d:
    print(p)
```

Provate ad eseguire lo stesso codice anche sulle immagini 'elefante.jpg', 'computer.jpg', 'baba.jpg'. Provate anche la rete InceptionV3, sostituendo il codice di caricamento della rete con il seguente:

```
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input,decode_predictions
model = InceptionV3(weights='imagenet')
```

Per la lista di reti già addestrare in Keras consultate la pagina web: https://keras.io/api/applications/.