Assignment 3

Design of a CNN architecture for MNIST digit dataset with max 7,5k parameters

Introduzione

Il progetto è stato realizzato su google **colab** in **python**, servendosi delle librerie Keras, **Keras_metrics** (è già presente il comando pip per installarla) e **Pandas**.

Analisi del Dataset e Reshape dei dati

```
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

Il dataset, appena **caricato** tramite le funzioni dedicate di keras, presenta la forma riportata a **sinistra**.

Per la rete convoluzionale ho bisogno di una **dimensione** diversa da quella fornita, che rappresenta il canale colore della mia immagine. Trattandosi di immagini in **scala di grigi**, tale dimensione è solamente 1.

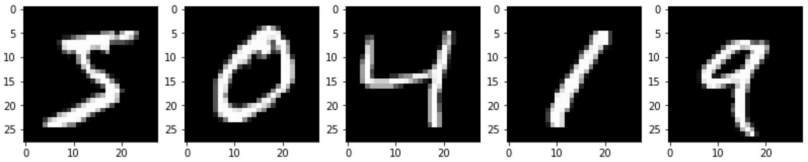
Eseguo quindi il **reshape dei dati** come segue:

```
x_train = np.expand_dims(x_train, axis=3)
x_test = np.expand_dims(x_test, axis=3)

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28, 1)
(60000,)
(10000, 28, 28, 1)
(10000,)
```

Le immagini appaiono come seguono:



Definizione del modello e numero di parametri per layer

```
model = Sequential()
model.add(Conv2D(filters=9, kernel_size=6, input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(filters=7, kernel_size=6))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(47, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))


model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'],)
model.summary()
```

La definizione del modello, per questa consegna, **impone** un vincolo sul numero di **parametri** da utilizzare di massimo **7500**.

Per avere performance migliori ho cercato di avere un modello che più si avvicinasse al numero limite di parametri, ottenendo un modello da **7482 parametri**.

In particolare ci sono due livelli **CONV2D** di keras, aventi rispettivamente le seguenti caratteristiche:

- **9** filtri con kernel size **6**
- **7** filtri con kernel size **6**

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_89 (Conv2D)           (None, 23, 23, 9)         333
_________________________________________________________________
max_pooling2d_82 (MaxPooling (None, 11, 11, 9)         0
_________________________________________________________________
conv2d_90 (Conv2D)           (None, 6, 6, 7)           2275
_________________________________________________________________
max_pooling2d_83 (MaxPooling (None, 3, 3, 7)           0
_________________________________________________________________
flatten_42 (Flatten)         (None, 63)                0
_________________________________________________________________
dense_106 (Dense)            (None, 47)                3008
_________________________________________________________________
dropout_53 (Dropout)         (None, 47)                0
_________________________________________________________________
dense_107 (Dense)            (None, 32)                1536
_________________________________________________________________
dropout_54 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_108 (Dense)            (None, 10)                330
=================================================================
Total params: 7,482
Trainable params: 7,482
Non-trainable params: 0
_________________________________________________________________
```

Il **MaxPooling2D** è stato impostato per dimezzare la dimensione: ce ne sono presenti due, quindi il dimezzamento è doppio. Ho provato ad usare **AveragePooling2D** ma le performance sembravano essere leggermente **peggiori**.

Successivamente, dopo il layer che si occupa del flattering dei dati, ci sono due livelli **DENSE** rispettivamente da **47** e **32** neuroni con funzione di attivazione **RELU,** che generalmente è quella che si comporta meglio.
Entrambi i suddetti sono seguiti da un livello che si occupa di fare il **dropout** dei dati per aiutare a prevenire il fenomeno **dell'overfitting** (settati a 0,2)

La **loss** che ho scelto, trattandosi di dati categorizzati, è la **categorical_crossentropy**.
L'ottimizzatore che mi sembra aver dato performance più stabili anche in base all'esperienza pregressa con i precedenti laboratori e assignment è **ADAM** con learning rate non specificato, ovvero lasciato come di default da keras a 0,001.

Training del modello

Per la fase di training ho utilizzato un **early_stop** sulla *validation loss* con un valore di patience pari a **3** per evitare overfitting.

Come limite **massimo** ho scelto di mettere **50 epoche.**

```
early_stop = EarlyStopping(monitor='val_loss', patience=3, verbose=1) #Imposto che le epoche si fermino quando val_loss smett

history=model.fit(x_train, to_categorical(y_train), epochs=50, validation_split=0.20, callbacks=[early_stop], batch_size=200)
```

Ho impostato uno **split** del **20%** per rappresentare il **validation set** a partire dai dati di training. Il **batch size** è stato impostato a 200.

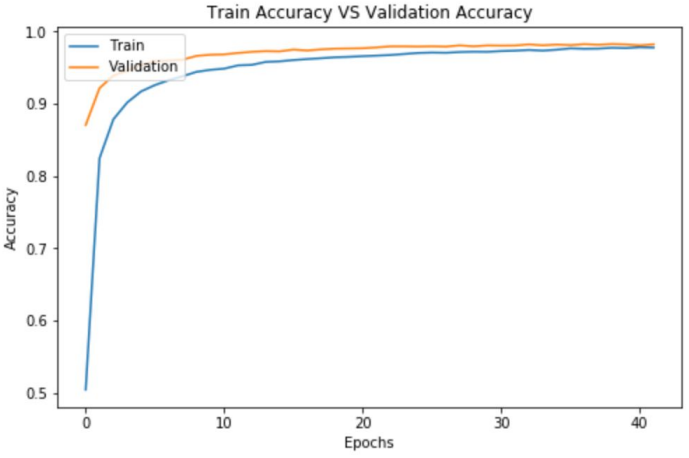
A seconda dell'esecuzione, la fase di fit **si arresta automaticamente intorno all'epoca 40** come mostrato della figura che segue

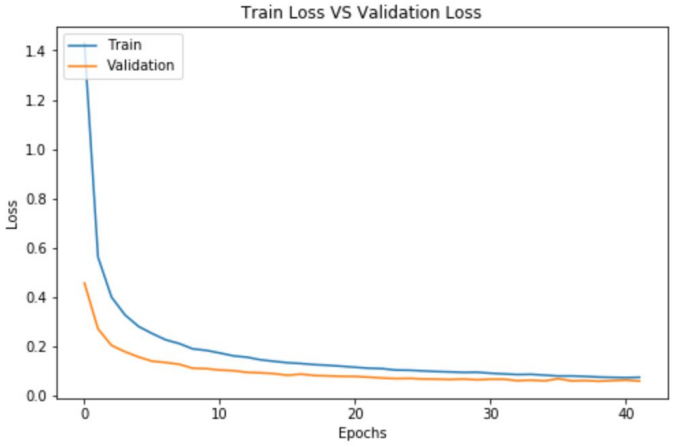
```
Epoch 39/50
48000/48000 [==============================] - 2s 32us/step - loss: 0.0760 - acc: 0.9777 - val_loss: 0.0592 - val_acc: 0.9828
Epoch 40/50
48000/48000 [==============================] - 1s 31us/step - loss: 0.0748 - acc: 0.9772 - val_loss: 0.0615 - val_acc: 0.9823
Epoch 41/50
48000/48000 [==============================] - 2s 32us/step - loss: 0.0737 - acc: 0.9784 - val_loss: 0.0639 - val_acc: 0.9810
Epoch 42/50
48000/48000 [==============================] - 2s 32us/step - loss: 0.0752 - acc: 0.9780 - val_loss: 0.0597 - val_acc: 0.9824
Epoch 00042: early stopping
```

Performance sui dati di training e di validation

L'accuracy ottenuta nella fase di training sul **validation set** è del **98,24%** (a seconda dell'esecuzione sono arrivato anche a picchi di circa *98,50%*).
Sul **training set** invece **l'accuracy** è intorno al **97,8%.**

Si riportano come richiesto l'andamento dei valori di di **accuracy** e di **loss** sui dati di train e di validation

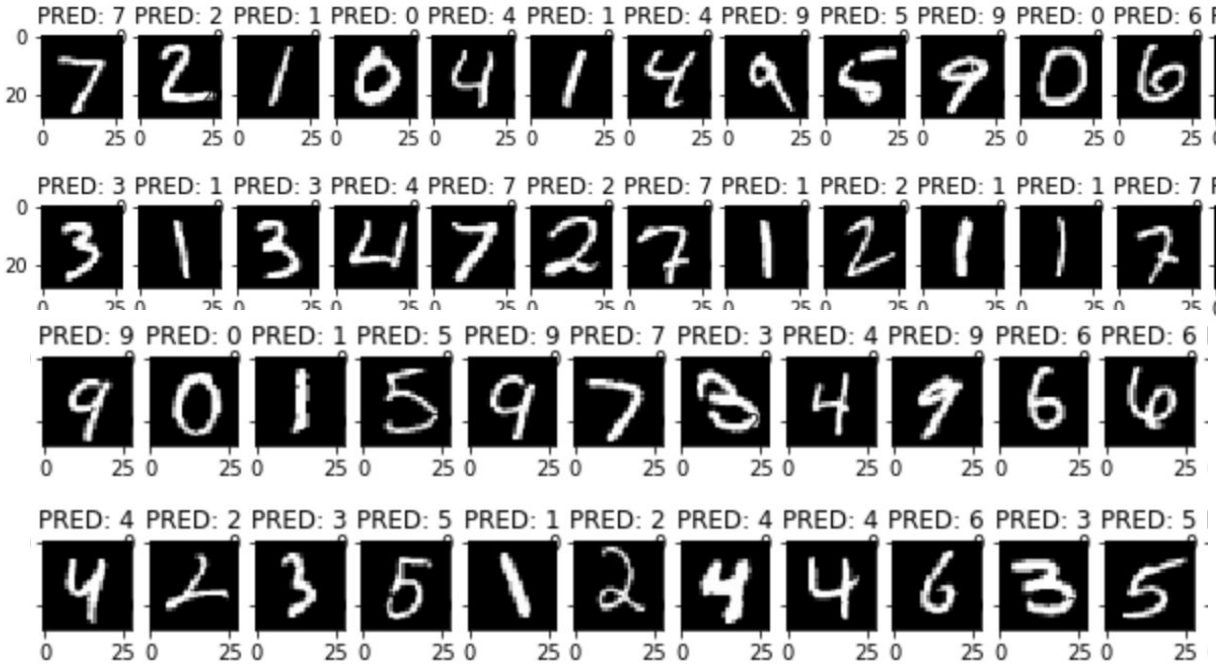




Predizione sul test set

```
[200] predictions = model.predict(x_test)
```

Applico il modello a **dati che il modello stesso non ha mai visto** tramite l'istruzione sopra.



Per poter fare una veloce **analisi visiva** dei risultati ho stampato alcune delle immagini contenute nel test set con sopra la dicitura "PRED: **x**" dove **x** indica la classificazione dell'immagine corrispettiva.
Da questa analisi, seppur molto approssimativa, le performance del **modello sembrano essere molto buone** anche su numeri rappresentati in modo po' **strano** come i seguenti:

Vediamo ora **tramite le label del test set** quanto effettivamente il nostro modello è performante su dati che non ha mai visto

Performance sul test set

```
results = model.evaluate(x_test, to_categorical(y_test), batch_size=128)
print('test loss:', results[0])
print('test accuracy:', results[1])
```

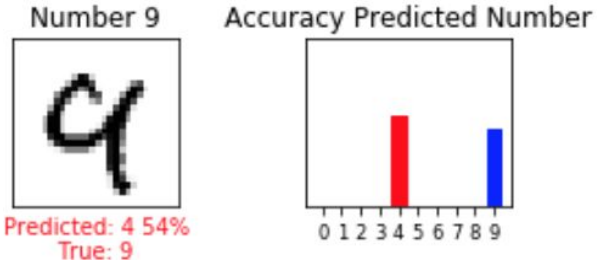
```
10000/10000 [==============================] - 0s 29us/step
test loss: 0.04822861978558358
test accuracy: 0.9857
```

L'accuracy sui dati di test è pari al **98,57%** quindi leggermente superiore a quella ottenuta sul validation test precedentemente.

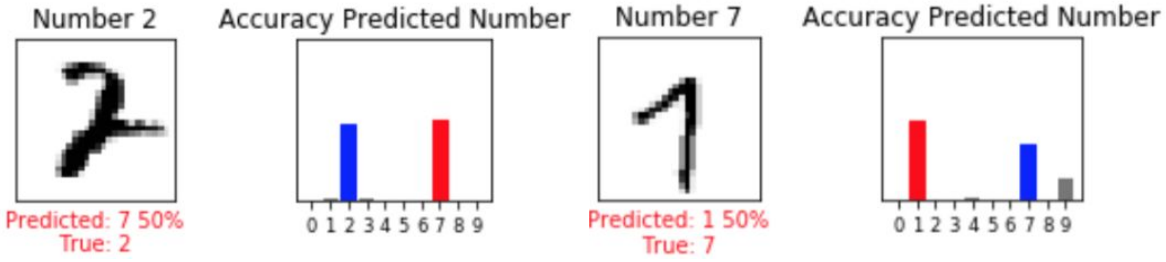
Errori sul test set

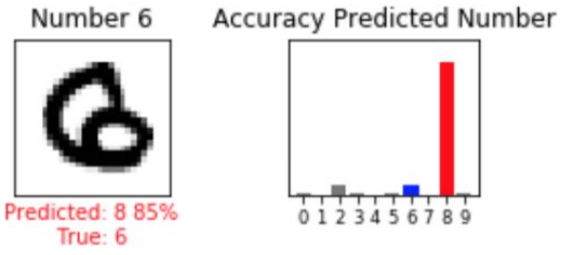
Riporto qui **alcuni errori** di predizione, a dimostrazione che alcuni numeri del dataset mnist sono scritti in modo abbastanza approssimativo.

Anche un "umano" avrebbe potuto predire la seguente immagine come un 4 scritto male.



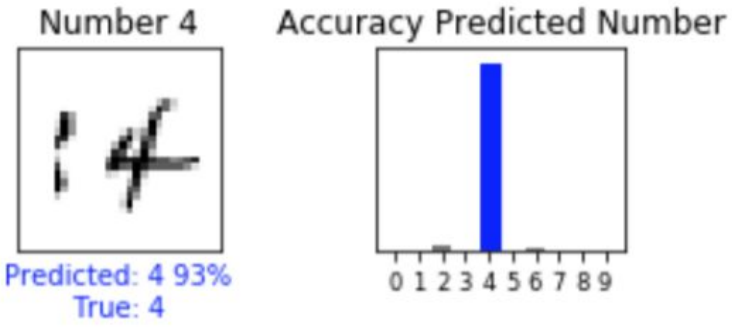
Riporto altri tre esempi significativi in tal senso:





Nonostante gli errori è possibile notare come il modello sbagli comunque in modo "ragionevole"

Avviene però anche che con **dati sporchi** (vedi sulla sinistra del seguente esempio) il modello predica il **risultato corretto** con per altro **poca incertezza**:



Riporto altri esempi generali direttamente dalla funzione presente anche nel source code.