Lez13_

November 21, 2023

1 Lezione 13 - Reti derivanti da LeNet

1.1 Recap

Riprendiamo la **LeNet-5**. La rete è composta da una parte convoluzionale, con due livelli convoluzionali, ciascun livello è seguito da un livello di *pooling*. Le immagini hanno un solo canale, essendo in bianco e nero. In input immagini 28x28. I due livelli di pooling riducono le dimensioni dell'immagine, alla fine sarà di dimensione 5x5. Il primo livello convoluzionale usa 6 filtri, il secondo 16. Tali filtri sono matrici di parametri che descrivono esattamente le operazioni che eseguiamo su ciascuna immagine presa da input. Alcune operazioni possono essere media dei colori o bordi, a seconda del Kernel. Non scegliamo noi l'operazione, ma i parametri del kernel sono frutto dell'addestramento. Io dico "voglio 6 filtri", poi per risolvere la classificazione, il processo di training cercherà i 6 filtri migliori, i quali estraggono dall'immagine le caratteristiche rilevanti. Questa è la prima parte.

Nella seconda parte, abbiamo 3 livelli *Dense*. In input si avrà un vettore *appiattito*, che passerà per questi tre livelli. Alla fine si applica funzione *softmax*, che associerà la probabilità di essere un certo numero da 0 a 9.

1.2 LeNet e Fashion MNIST

Prendiamo il notebook `tf_tenet_fashion.ipynb`. Quando abbiamo addestrato il modello, non ci siamo mai preoccupati del *tempo di processamento*, adesso avrò un fattore più rilevante.

```
mlp_model = keras.models.Sequential()
mlp_model.add(keras.layers.Flatten(input_shape=[28, 28]))
mlp_model.add(keras.layers.Dense(300, activation="relu"))
mlp_model.add(keras.layers.Dense(100, activation="relu"))
mlp_model.add(keras.layers.Dense(10, activation="softmax"))
.
.
.
history = mlp_model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                   callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

Qui abbiamo usato *Early Stopping*, accuratezza si aggira sul 90%. Ora vogliamo provare con *LeNet*; importante è fare il **reshaping**. Ciò vuol dire aggiungere un' informazione sul numero dei canali. La prima dimensione è quella del minibatch, altezza, lunghezza, numero dei canali. *Non vedere l'immagine come matrici, bensì come "oggetti tridimensionali"*. Vediamo nuovamente i parametri:

- `1` = se istanze del training sono 60k, non modificarle.
- `28,28` = la loro dimensione è importante, se cambiassi le immagini in `30x30`, la rete funzionerebbe in modo diverso/non funzionerebbe. Se kernel = 5, come vediamo nel modello `Sequential`, questo si sposta nella matrice `28x28`, le dimensioni sono le stesse dell'input *se usiamo il padding*, quindi l'output avrebbe dimensione `28x28`. Senza sarebbero *minori*.
- `1` = aggiungo il numero dei canali, pari ad 1.

```
X_train2 = X_train.reshape(-1, 28, 28, 1) # single channel
X_valid2 = X_valid.reshape(-1, 28, 28, 1)
```

Il primo livello convoluzionale, qualora partisse da `30x30`, porterebbe ad un pooling `15x15`, e nel secondo pooling `7x7`. Col `Flatten`, la disporrei come un vettore, di dimensione totale $7 \cdot 7 \cdot 16$ elementi in input al primo livello connesso. E' un valore diverso dal partire con immagini `28x28`. Se ad un livello completamente connesso do in input più elementi di quelli previsti, il livello non funziona con un numero variabili di neuroni in input, poichè la sua matrice W è definita da parametri *fissi*. Tutto ciò per dire che la parte convoluzionale funziona **con qualsiasi input**, ma il problema è la parte **connessa**, la quale lavora con parte fissa. In avanti, potremo lavorare solo con la parte *convoluzionale*.

```
lenet_alt = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(filters=6, kernel_size=5, activation='relu', padding='same'
            tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
            tf.keras.layers.Conv2D(filters=16, kernel_size=5,activation='relu'),
            tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(120, activation='sigmoid'),
            tf.keras.layers.Dense(84, activation='sigmoid'),
            tf.keras.layers.Dense(10, activation="softmax")])


lenet_alt.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])
```

In questa seconda applicazione, si hanno meno parametri, maggiormenti presenti nella parte *completamente connessa*. Addrestriamo per 30 epoche:

```
history2 = lenet.fit(X_train2, y_train, epochs=30,
                     validation_data=(X_valid, y_valid),
                    callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

Poco più lenta, ma comunque ancora gestibile mediante il nostro pc. L'accuracy finale è pari all'89.6%, più o meno come la prima simulazione. L'**early stopping** non è intervenuto, quindi non c'è overfitting, e probabilmente aumentando le epoche i risultati potrebbero migliorare.

Se usassimo funzioni `relu`, l'accuracy cresce fino al 91%. Se passassi da `adam` ad `SGD` ho un cambiamento importante nelle prestazioni. (in negativo, credo).

1.3 Reti convoluzionali più profonde

Sviluppo fermo fino al 2010, mancavano dati ed hardware. Bisogna usare le GPU. Si facevano competizioni, vediamo alcune delle migliori proposte:

1.3.1 AlexNet

Simile a *LeNet*, con queste differenze:

- 8 livelli convoluzionali (LeNet 2)
- Livelli più grandi.
- Lavora su immagini più grandi e multicanale.
- Relu invece che sigmoid.
- Uso del Dropout.

E' fornito il notebook `tf_alexnet_fashion.ipynb`, consigliabile usare **Google Colab**, per usare la GPU. In alto a destra, dove vediamo `ram` e `disco`, possiamo settare `T4 GPU`.

L'input è di tre canali e dimensioni `224x224`, noi vorremmo applicarlo a `FASHION_MNIST`che ha dimensioni `28x28`. Introduciamo un modello di Resizing (allarghiamo le immagini). Notiamo anche il *dropout*.

```
alex = tf.keras.models.Sequential([
            tf.keras.layers.Resizing(227,227), # example, do not upscale input in practice!
            tf.keras.layers.Conv2D(filters=96, kernel_size=11, strides=4,
                                   activation='relu'),
            keras.layers.BatchNormalization(),
            tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
            tf.keras.layers.Conv2D(filters=256, kernel_size=5, padding='same',
                                   activation='relu'),
            keras.layers.BatchNormalization(),
            tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
            tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding='same',
                                   activation='relu'),
            keras.layers.BatchNormalization(),
            tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding='same',
                                   activation='relu'),
            keras.layers.BatchNormalization(),
            tf.keras.layers.Conv2D(filters=256, kernel_size=3, padding='same',
                                   activation='relu'),
            keras.layers.BatchNormalization(),
            tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(4096, activation='relu'),
            tf.keras.layers.Dropout(0.5),
            tf.keras.layers.Dense(4096, activation='relu'),
            tf.keras.layers.Dropout(0.5),
            tf.keras.layers.Dense(10, activation="softmax")])
```

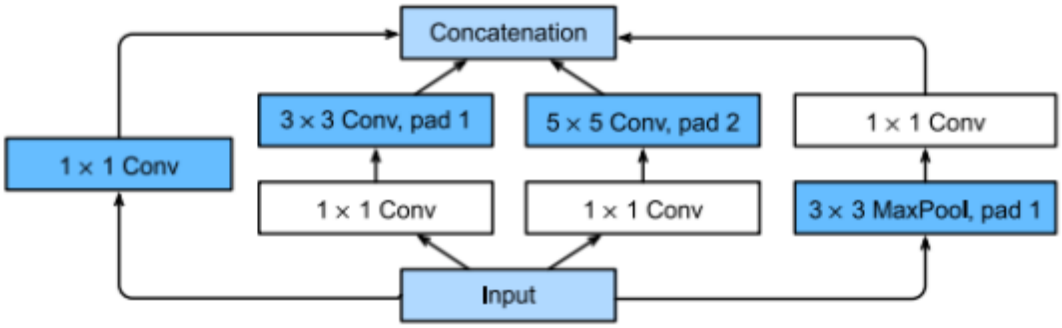
```
alex.compile(loss="sparse_categorical_crossentropy",
             optimizer="adam",
             metrics=["accuracy"])
```

I tempi di esecuzione sono più dilatati, perchè usiamo una GPU low-quality condivisa con migliaia di utenti. Dopo aver completato le epoche, iniziamo a vedere il fenomeno dell'overfitting, perchè usiamo rete complicata per un problema non troppo complicato!

1.3.2 GoogLeNet

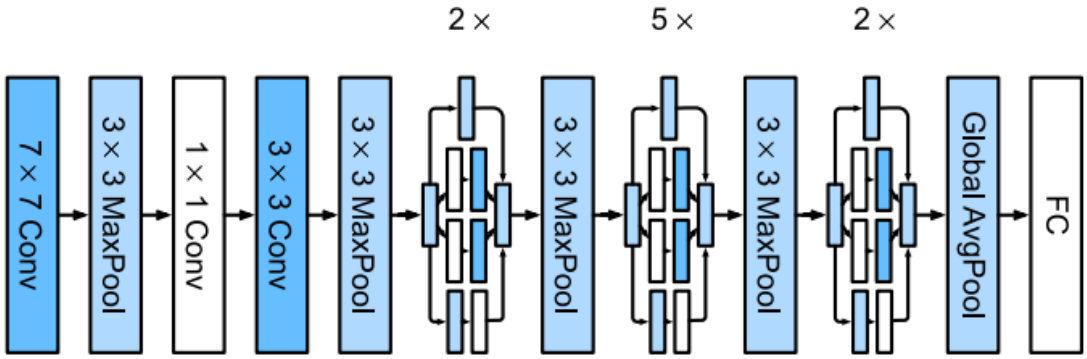
Richiama LeNet, fatta dai dipendenti di Google. Composta da: - Stem : 2/3 livelli di convoluzioni, per prelevare i dati.

- Body: Altri blocchi convoluzionali, per processare i dati.
- Head: per la predizione.



La novità principale risiede nell'**Inception Block**, per evitare di scegliere le dimensioni del Kernel. L'input passa per 4 rami convoluzionali. Alcuni solo convoluzione, altri convoluzioni + Pooling, altri altro ancora. Alla fine li *concateno*. Usando kernel convoluzionali di dimensioni diverse, così non scelgo.

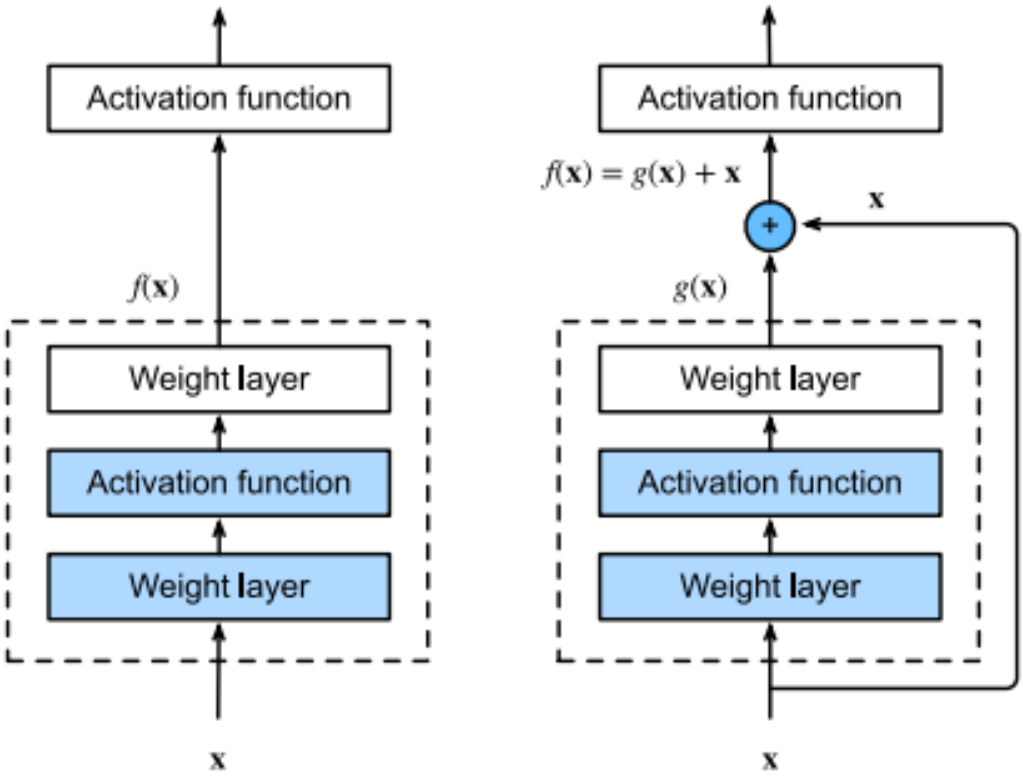
A parte questo blocco, la struttura è:



Global Pooling lavora con più canali, e ne ritorna *uno solo*. **Pooling classico** lavora *canale per canale*.

1.3.3 ResNet

Può avere anche più di 100 livelli, e con *meno* parametri. Questo modello ha introdotto un'architettura più profonda ed efficiente sfruttando le skip connection residue o shortcut connections. Lega un'unità di un certo livello con un altro *diverso* dal successivo, ad esempio lego primo livello e terzo livello. Alla base del **residual block**. Vogliamo imparare $f(x)$:



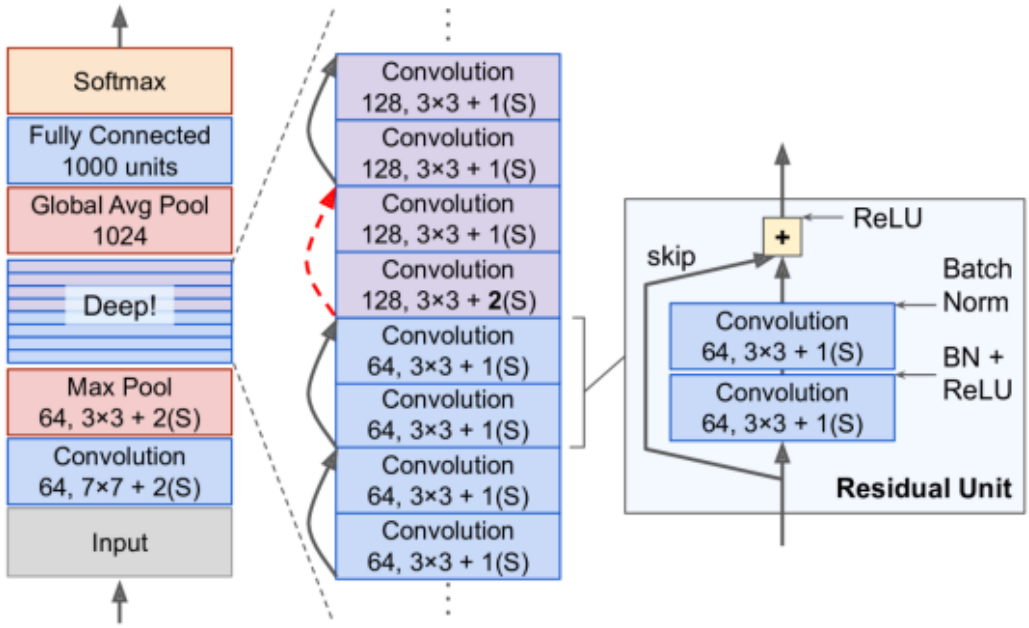
Con *Residual block* i livelli apprendono $g(x)$, e calcolano $f(x) = g(x) + x$. E quindi? Il vantaggio quale è?

Se i pesi sono inizializzati a 0, normalmente la predizione iniziale sarà vicina a 0 nel caso normale, mentre coi *residual block* sarà $0 + x$, quindi come se apprendesse una *funzione identità*, rendendola più rapida nell'apprendere. La seconda rete sarà più rapida nell'apprendere $f(x)$.

Altro vantaggio è: Concateniamo i blocchi in figura.

- Nel primo caso, il primo blocco sarà vicino allo 0, il secondo, terzo saranno addestrati su input non significativi, cioè *rumore*.
- Con *Skip Connection*, quantomeno ci avviciniamo all'input x, quindi c'è un progresso in più.

Generalmente è simile a *GoogLeNet*, cambia giusto la parte centrale.



ResNet è facile da modificare, per questo ne esistono varie versioni. Noi difficilmente partiamo da 0, ma sfrutto ciò che esiste per ciò che mi serve! Non si sviluppano più *nuovi modelli*, bensì **meta-modelli** adattabili, come **RegNet(x)**.

1.3.4 Usare Modelli pre-allenati

Non serve implementarli da 0, `Keras` fornisce il loro utilizzo mediante:

```
m = keras.applications.ResNet50(weights='imagenet')
```

Modello già addestrato coi pesi, sono pronto per la *predizione*.

1.3.5 Esempio tf_pretrained_cnn.ipynb

Vogliamo classificare dinousari, quindi facciamo resize delle immagini nella fase di *pre-processamento*. Otteniamo vettore y di 1000 numeri, cioè le 1000 classi di `imagenet`.

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
Y_proba = model.predict(inputs)
print(Y_proba.shape)
```

Per trasformare le probabilità in classi, usiamo:

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images_resized)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print("  {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

L'output è il seguente:

```
Image #0
  n01704323 - triceratops  99.99%
  n02417914 - ibex         0.01%
  n02454379 - armadillo    0.00%
```

1.4 Classificare nuove classi - Transfer Learning

Se volessi classificare eventualmente nuove classi, non devo rifare tutto da 0. Possiamo sfruttare cose già fatte, perchè ad esempio informazioni come bordi, sfumati etc sono significative. Dobbiamo cambiare gli ultimi livelli, quelli *completamente connessi*. Tale operazione è alla base del **Transfer Learning**.

Se il mio dataset è limitato, non riuscirei ad addestrare GoogLeNet o AlexNet.

Come ? Nel deep learning, ciò che facciamo è:

1. Prendere i layer da un modello precedentemente addestrato
2. Congelare i layer inferiori, in modo da evitare di distruggere le informazioni che contengono durante i futuri cicli di addestramento
3. Aggiungere nuovi layer addestrabili *sopra* agli strati congelati
4. Addestrare i nuovi layer sul proprio dataset
5. Opzionale - *Fine-tuning*: scongelare l'intero modello e riaddestrarlo sul nuovo dataset con un tasso di apprendimento molto basso, altrimenti nelle poche iterazioni che faccio, potrei cambiare significativamente i parametri, allora mi limito a fare piccole modifiche.

Usiamo il *nostro dataset interamente*, non è detto che io abbia l'originale, inoltre classificare *cani* e *gatti* usando i *dinosauri* non sembra un'ottima idea!

Nota: il termine "fine-tuning" è spesso utilizzato per indicare l'intero flusso di lavoro, non solo l'ultimo passaggio.

1.4.1 Esempio - tf_transfer.ipynb

Vogliamo classificazione binaria, tra *cani* e *gatti*. Usiamo **Xception CNN**, variante del *GoogLeNet*. In origine classificava 1000 tipologie, noi ci limitamo alle 2 appena citate.