Lez12_ReteiNeuraliConvoluzionali2

November 17, 2023

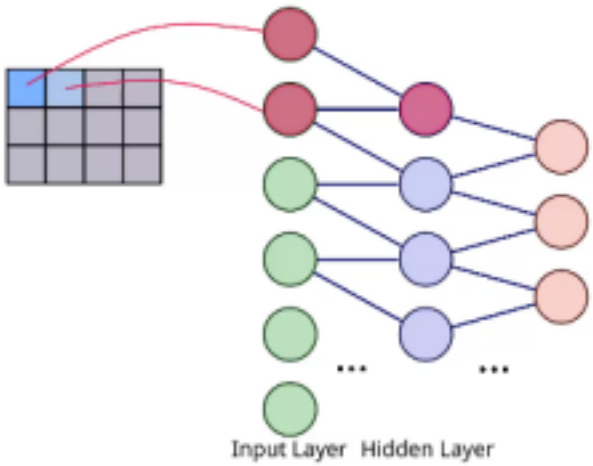
1 Lezione 12 - Reti convoluzionali 2

2 Recap

Siamo partiti da un esempio per capire il perchè c'è necessità di queste reti. Nella classificazione delle immagini, abbiamo tantissimi pixel da gestire. Un livello nascosto che deve gestire tale input, seppur con poche unità, porterà ad un livello completamente connesso avente un miliardo di parametri. Tale rete sarebbe difficile da addestrare. Un altro problema, legato a queste immagini, è che ignoriamo alcune loro caratteristiche, come la posizione dei pixel. Due pixel vicini rappresenteranno quasi sicuramente la stessa parte del soggetto. Noi vorremmo sfruttare questa conoscenza! A questo servono le *reti convoluzionali*, le quali lavorano su sequenze di dati basate su *serie temporali*, con ordinamento, o su immagini (griglia di pixel).

2.1 Interazioni sparse

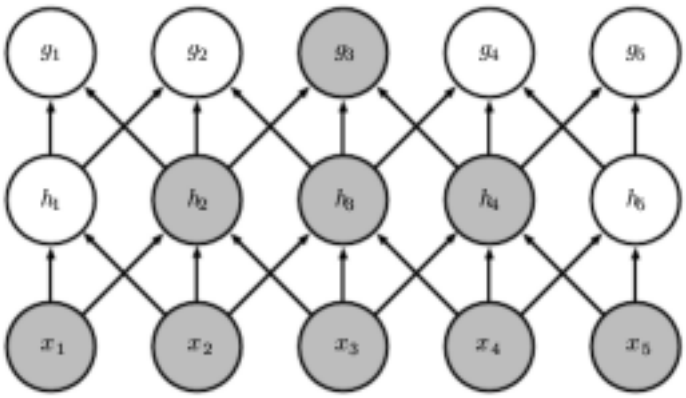
E' una delle caratteristiche analizzate, e si basa sull'idea di non avere una interazione totalmente connessa tra livelli, bensì con una porzione ridotta. Una unità del livello nascosto interagisce con una porzione delle features in input. Cosi il numero di parametri da memorizzare e allenare è minore.



Così facendo, non perdiamo interazioni nell'input? Magari, il pixel in blu potrebbe essere legato all'ultimo pixel in basso a destra.

2.2 Receptive Field

Indica lo unit set, cioè quali sono le unità che interagiscono con una certa unità. Abbiamo visto che, tramite interazioni *indirette*, come un albero che si propaga, possiamo toccare porzioni distanti dell'input.



2.3 Condivisione dei parametri

Sappiamo che ogni coppia ha un peso $w_{i,j}$ associato all'arco, che va memorizzato e addestrato.

Nelle reti convoluzionali, è come se avessimo una piccola matrice w che riusiamo per ogni unità, cioè è *condivisa*, abbiamo tolto archi e parametri da gestire. Riferendoci all' immagine *Sparce Connections*, noi useremo sempre la coppia w_1, w_2, non userò altre coppie. Il tipo di calcolo non cambia a seconda della posizione in cui mi trovo nell'input. Ciò introduce un nuovo concetto.

2.4 Reti equivarianti

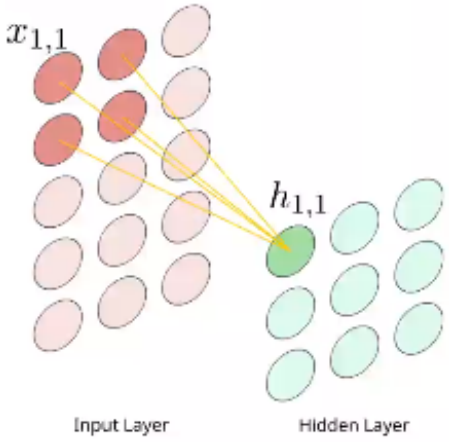
Spostare un input comporta uno spostamento nell'output. Cioè, se lo applichiamo ad una immagine, sia essa centrale o traslata, non ho problemi. Cambiare posizione pixel in input, porterà ad uno stesso risultato in output, traslato. Ciò non era vero nelle reti viste finora. Non ho certezza di efficienza per ogni *movimento*, quindi ad esempio per una rotazione non è detto che basti questa proprietà.

2.5 Ragionare con griglie

Se lavoriamo con input a griglia, ha senso immaginare anche le unità nascoste rappresentate in una griglia. E' conveniente perchè ogni unità è connessa con una piccola porzione dell'input. Nell'immagine, $h_{1,1}$, la più in alta a sinistra, gestirà una porzione dell'input situata sempre in *alto a sinistra*. Gli archi avranno un peso uguale per tutti gli altri parametri. Moltiplichiamo $x_{1,1}$ per un parametro $k_{1,1}$, che riuseremo anche per altri $h_{i,j}$.

$$h_{1,1} = k_{1,1}x_{1,1} + k_{1,2}x_{1,2} + k_{2,1}x_{2,1} + k_{2,2}x_{2,2}$$

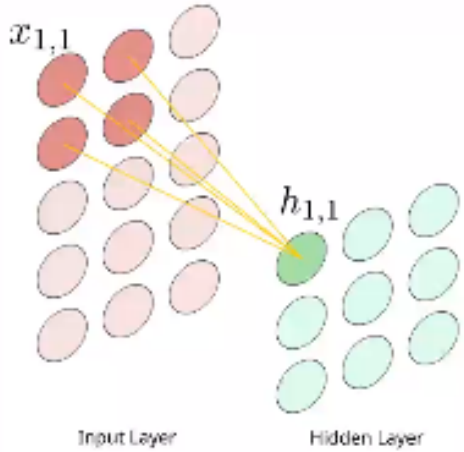
$$h_{i,j} = k_{1,1}x_{i,j} + k_{1,2}x_{i,j+1} + k_{2,1}x_{i+1,j} + k_{2,2}x_{j+1,i+1}$$



Otteniamo una matrice K, simile a W, ma con parametri ridotti. Tale matrice si chiama **Kernel** o *Filtro*. E' una matrice di parametri, set di valori che calcolo durante il training, non sono valori dati.

2.5.1 Esempio

$$K = \begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$$



- What are we computing?

Cosa stiamo calcolando?

Con tutti i pesi pari a 0.25 stiamo facendo una media dei quattro pixel che andiamo a selezionare, come se fosse un'immagine sfocata, avente una media dei colori.

Ricordiamo che, considerando immagini bidimensionali, con X input, e K kernel, ciò che stiamo facendo è:

$(X * K)(i, j) = \sum_a \sum_b \sum_c K_{a,b,c} X_{i+a,j+b,c}$

(NB: non è la vera convoluzione, perchè la formula è diversa, nelle librerie si usa *Cross-Correlazione*, non abbiamo la proprietà *commutativa*, ma già abbiamo visto che non è un problema).

2.6 Convoluzione basica - come si usa

Addestrare livello convulazionale, vuol dire addestrare i parametri del kernel. Non sono io che scelgo l'operazione sui pixel, è il training che definisce i valori di K, cambiando operazione da applicare a ciascun livello. Abbiamo già visto che un kernel $[-1, 1]$ applicato ad una immagine, vuol dire fare: $pixel_1 \cdot (-1) + pixel_2 \cdot (1)$

(lo abbiamo visto sull'esempio del cane, di cui evidenziamo i bordi).

2.7 Zero Padding

Abbiamo visto che il kernel riduce la dimensione dell'output. Soluzione a ciò è lo *zero padding*. Tra i vari vantaggi, la proiezione della finestra sul mio input, sarà centrata su questo elemento dell'input.

2.8 Convoluzione come prodotto tra matrici

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \qquad K = \begin{bmatrix} k_{1,1} & k_{1,2} \\ k_{2,1} & k_{2,2} \end{bmatrix}$$

We construct a matrix M from K and a vector v from A

$$M = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} \end{bmatrix}$$

$$v = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{2,1} & a_{2,2} & a_{2,3} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

$$Mv^T = \begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{1,3} \cdot 0 + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} + a_{2,3} \cdot 0 + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,1} \cdot 0 + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{2,3} \cdot 0 + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} + a_{2,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1} \cdot 0 + a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,1} \cdot 0 + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

Reshaping into a 2x2 matrix we get the result:

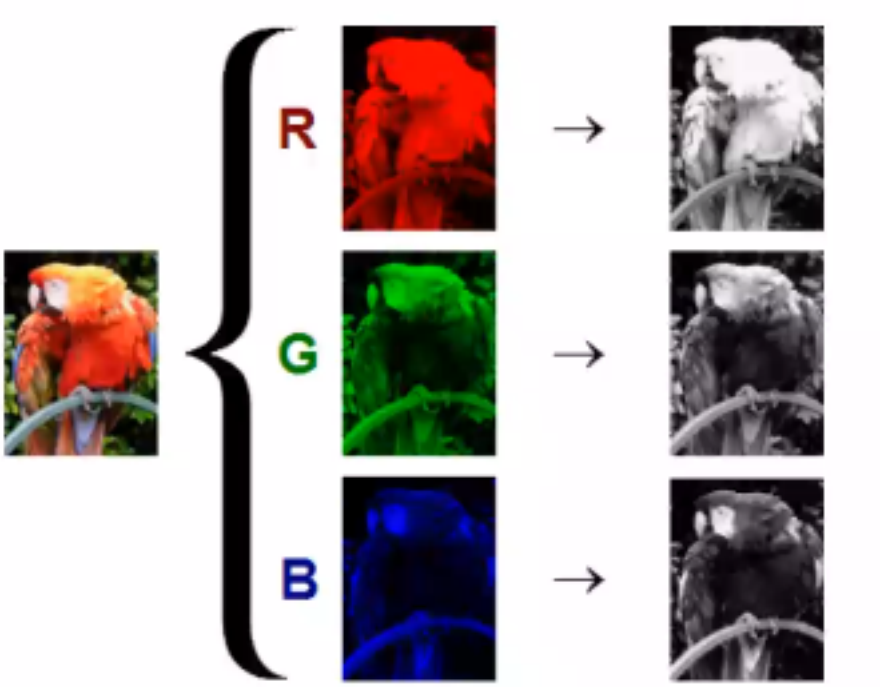
$$\begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} & a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} \\ a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} & a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

(*stai sereno e tranquillo che lo leggerò sicuramente*). In realtà non ci serve comprendere quelle formule, perchè in realtà non stiamo facendo nulla di diverso dalle operazioni tra matrici e tensori viste dall'inizio del corso.

Anche con operazioni interne diverse, possiamo usare ciò che abbiamo visto fino ad ora.

2.9 Immagini come griglie di Pixel

Solo le immagini in bianco e nero sono a due dimensioni, se sono colorate, esse sono composte da tre *canali*: red, green, blue. Ogni canale è una matrice bidimensionale che ci dice tali intensità. L'immagine è la somma dei tre colori.

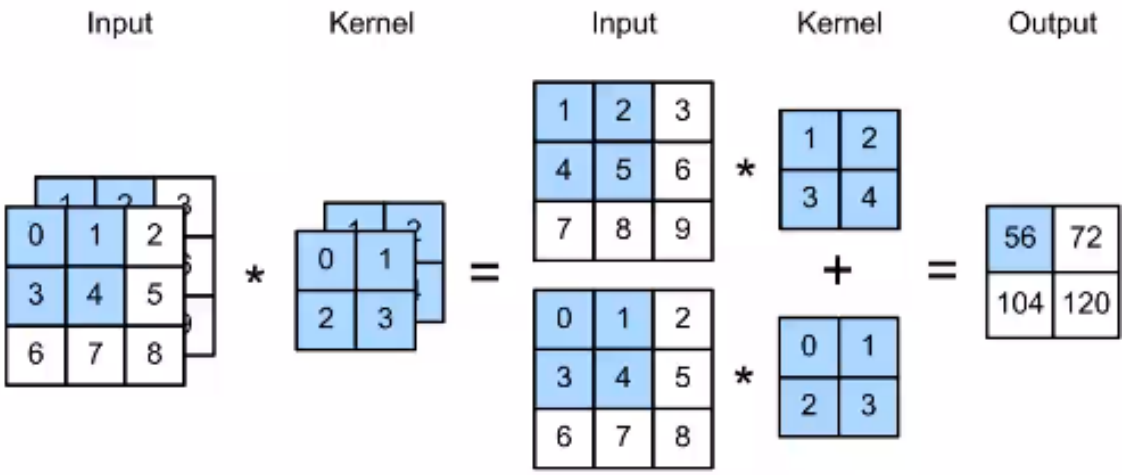


Si indicano prima i canali, poi le righe, e poi le colonne, cioè l'input è del tipo: " *channel* x r x c "

Con più *channels*, ci serve un kernel 3D:

$(X * K)(i, j) = \sum_a \sum_b \sum_c K_{a,b,c} X_{i+a,j+b,c}$

graficamente:



Prendiamo i valori $0, 1, 3, 4$ e moltiplichiamo per $0, 1, 2, 3$ del kernel. Lo stesso faremo per il secondo canale. Alla fine l'output è bidimensionale, perchè non abbiamo fatto un calcolo separato per i canali. Quindi, con canali in entrata, il risultato è condensato per tutti i canali.

NB: Non sono due kernel in figura, bensì è un unico kernel tridimensionale.

Esempio: 1x1 kernel Sembra avere poco senso, ma se ne usassi uno per ogni canale? E' utile per calcolare la media per ogni canale. Avremmo kernel: "1 x 1 x c_i"

2.9.1 Multiple Output Channels

Ragionando su immagini a colori, cioè oggetti tridimensionali, cioè tre canali, e per ogni canale una matrice di 3 righe e 3 colonne che definisco una porzione di pixel.

Ora, con più canali, se prendo porzione di pixel, dovrò vederla su più canali (intensità rosso, verde, blu). Avrei tre matrici *2x2* in input, allora anche il kernel deve avere terza dimensione. Posso avere parametri diversi per intensità rosso, verde, blu.

Un livello convoluzionale *applica più kernel* allo stesso tempo, kernel tridimensionali che prodocuno un oggetto in uscita. Ogni kernel è come un parametro diverso che mi permette di trovare caratteristiche diverse, e in uscita avrò diversi *canali*.

Se ho n canali in input, necessito di un parametro per ogni elemento dell'input, quindi come se mi servissero n kernel. Nel calcolo della convoluzione, tra i vari canali viene fatta la somma. Un solo kernel ci fa passare da n canali in input ad uno solo kernel in output. Se usassi n kernel, otterrò n canali in output partendo da n canali in input.

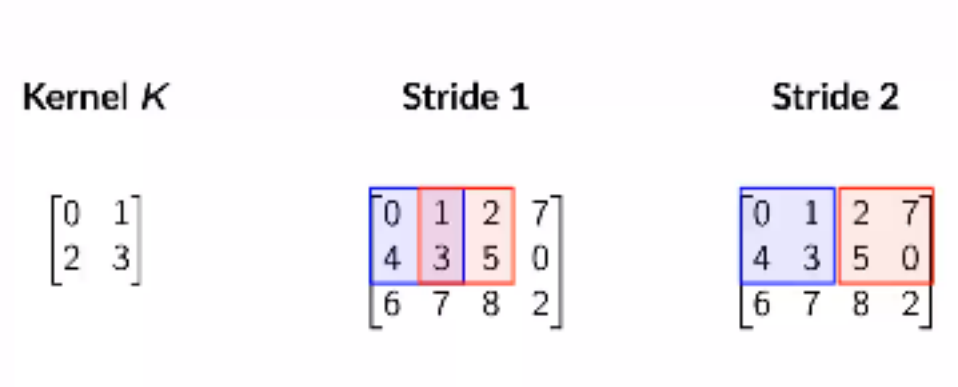
Sia X il tensore in input, con n_h x n_w x c_i e Kernel K con k_h x k_w x c_i x c_0, più il bias. In output avrò tensore H di dimensione n_h x n_w x c_0, assumendo *zero padding*. La computazione di H calcolando la convoluzione con K per ogni posizione dell'input, potendo anche aggiungere una funzione ϕ (es RELU) ciò che facciamo è calcolare:

$H_{i,j,k} = \phi \sum_{u,v,c} K_{u,v,c,k} X_{i+u,j+v,c} + b_k$

con b_k bias per il k-esimo canale in output.

2.10 Stride

Possiamo far saltare la finestra di convoluzione di varie posizione, non più per ogni posizione, ma ad esempio spostandoci di due in due, quindi metà delle convoluzioni, e riduciamo la risoluzione dell'immagine prodotta. Ciò è definito dal parametro *Stride*.

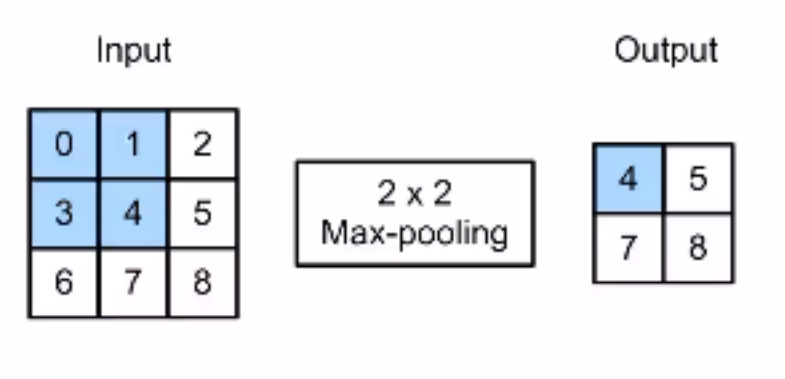


Con *Stride 1*, mi sposto di una posizione, prima con il quadrato blu, poi quello rosso. Con *Stride 2*, mi sposto di due dimensioni.

Questo conclude i concetti del **livello convoluzionale**. Tuttavia, non è l'unico livello introdotto in queste reti.

2.11 Pooling

Finestra di dimensione fissa, che viene fatta scorrere su tutte le regioni dell'input, può usare lo *stride*. Mi calcola un singolo valore in output per ogni singolo valore attraversato. Simile alla convoluzione, ma senza parametri. Servono ad aggregare i valori dei pixel. Utile per *massimo* o *average*.



2x2 è dimensione finestra, in quella finestra prendo il valore massimo, cioè 4. Se scorro verso destra, ho a che fare con $1, 2, 4, 5$, di cui il massimo è 5. Viene usato per fare *downscale dell'input*, riducendo la risoluzione ma mantenendo comunque l'informazione.

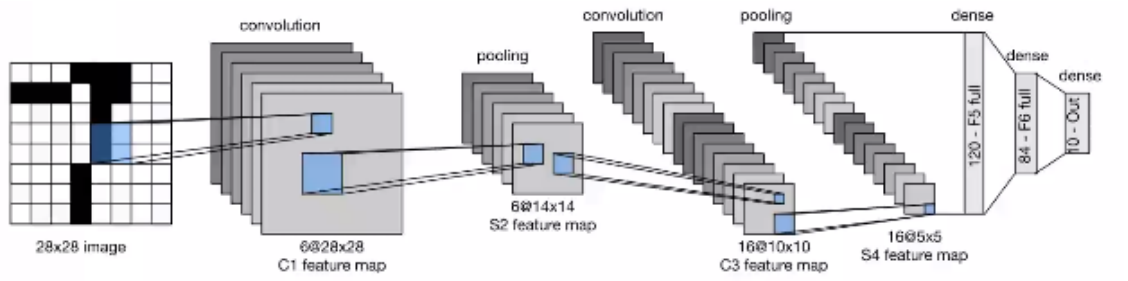
Con più canali, lo applico ad ogni canale, quindi il numero dei canali non cambia. Con una finestra di dimensione 3x3, stride viene posto a 3, riducendo di un fattore 9 l'immagine.

2.12 LeNet-5

Prima rete convoluzionale presentata. Usata per riconoscere cifre scritte a mano. L'approccio iniziale era *supervised*, ma con *reti neurali convoluzionali* l'errore era minore dell'1%.

2.12.1 Architettura

Convolutional Encoder Codifica input in feature di più alto livello, usabili per riconoscimento. Presenta due livelli convoluzionali. Si passa per 6 kernel, e le dimensioni restano sempre 28x28. Con *Pooling*, la dimensione passa a 14x14. Poi 16 filtri convoluzionali, passando a 10x10. Con pooling, di nuovo, si passa ad immagini 5x5.



Nella seconda parte, abbiamo livelli densi completamenti connesso, la prima con 120 neuroni. Come gliele passo? Tutto l'output sul livello convoluzionale viene messo in un *unico grande vettore* e lo fornisco al livello completamente connesso. L'ultimo livello è softmax a 10 livelli, per predire quale delle 10 cifre è predetta.

2.13 Esempio LeNet5

2.13.1 Parte di preparazione

[20]:
```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)
tf.random.set_seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

"Iniziamo caricando il dataset MNIST. Keras dispone di diverse funzioni per caricare dataset popolari in `keras.datasets`. Il dataset è già diviso in un set di addestramento e uno di test, ma può essere utile suddividere ulteriormente il set di addestramento per ottenere un set di validazione."

[21]:
```
mnist = keras.datasets.mnist
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
```

Il set di addestramento contiene 60.000 immagini in scala di grigi, ognuna di dimensioni 28x28 pixel.

[22]:
```
X_train_full.shape
```

[22]:
```
(60000, 28, 28)
```

Dividiamo il set di addestramento completo in un set di validazione e un set di addestramento più piccolo. Inoltre, scaliamo le intensità dei pixel nell'intervallo da 0 a 1 e le convertiamo in numeri floating-point, dividendo per 255.

[23]:
```
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

Puoi visualizzare un'immagine utilizzando la funzione `imshow()` di Matplotlib, con la mappa di colori `'binary'`.

plt.imshow(X_train[1], cmap="binary") plt.axis('off') plt.show()

[24]:
```
y_train
```

[24]:
```
array([7, 3, 4, …, 5, 6, 8], dtype=uint8)
```

2.14 LeNet 5

Partiamo con il metodo `reshape`.

- -1 indica a NumPy di calcolare automaticamente la dimensione dell'asse in base alle altre dimensioni e alla lunghezza totale del tensore. In pratica, se X_train ha un totale di 60.000 campioni e ogni campione è un'immagine di 28x28 pixel, l'uso di -1 calcolerà automaticamente la dimensione appropriata per adattarsi al numero totale di elementi.
- 28 e 28 sono le dimensioni dell'immagine in pixel, quindi si sta riformattando ogni campione per avere una dimensione di 28x28.
- 1 indica che ogni pixel dell'immagine è rappresentato da un singolo canale. Nell'MNIST, le immagini sono in scala di grigi, quindi hanno un solo canale.

[25]:
```
X_train2 = X_train.reshape(-1, 28, 28, 1) # single channel, -1 = non alterarla,␣
↪Oggetto con 4 dimensioni, l'ultima dimensione che ho aggiunto è 1 canale.
X_valid2 = X_valid.reshape(-1, 28, 28, 1)
display(X_train2.shape)
```

```
(55000, 28, 28, 1)
```

Passiamo al `modello sequenziale`, in particolare vediamo lo strato `Conv2D`, cioè layer **convoluzionale** di una rete neurale. I suoi parametri sono: - `filters=6`: Specifica il numero di kernel/filtri nel livello. I Kernel sono delle "finestre" che scorrono sull'immagine di input per estrarre caratteristiche. - `kernel_size=5`: Indica la dimensione del kernel che scorre sull'immagine durante la convoluzione. In questo caso, è una finestra 5x5 (tupla). Può essere anche un intero (5, se ragioniamo con matrici quadrate allora è una abbreviazione di 5x5). - `activation='sigmoid'`: Specifica la funzione di attivazione applicata dopo la convoluzione. Nel caso specifico, si utilizza la funzione di attivazione sigmoid. La funzione sigmoid mappa i valori a una scala compresa tra 0 e 1. - `padding='same'`: Questo parametro determina il tipo di padding da applicare all'immagine di input durante la convoluzione. In questo caso, il padding 'same' fa in modo che l'immagine di output abbia la stessa altezza e larghezza dell'immagine di input utilizzando zero-padding, se necessario. - `input_shape=(28, 28, 1)`: Specifica la forma dei dati di input. In questo caso, le immagini sono di dimensioni 28x28 pixel con un singolo canale (scala di grigi).

Quindi, questo layer convoluzionale prende in input immagini di 28x28 pixel con un canale e applica 6 filtri 5x5 utilizzando la funzione di attivazione sigmoide.

Poi abbiamo `AvgPool2D`, con pool size e strides. Abbiamo un altro `Conv2D` con 16 filtri, null'altro.

`Flatten` mette tutto ciò in un unico livello, da dare in pasto ai livelli `dense` totalmente connessi. Il livello di uscita usa `softmax`.

[26]:
```
lenet = tf.keras.models.Sequential([ #classico modello sequenziale.
            tf.keras.layers.Conv2D(filters=6, kernel_size=5,␣
 ↪activation='sigmoid', padding='same', input_shape=(28, 28, 1)),
            tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
            tf.keras.layers.Conv2D(filters=16,␣
 ↪kernel_size=5,activation='sigmoid'),
            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.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])

lenet.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 28, 28, 6)         156

 average_pooling2d_6 (Avera  (None, 14, 14, 6)         0
 gePooling2D)

 conv2d_7 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_7 (Avera  (None, 5, 5, 16)          0
 gePooling2D)

 flatten_3 (Flatten)         (None, 400)               0

 dense_9 (Dense)             (None, 120)               48120

 dense_10 (Dense)            (None, 84)                10164

 dense_11 (Dense)            (None, 10)                850
```

```
=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```