13 Quarta Lezione A

13.1 Reti neurali

Essendo il computer molto stupido a qualcuno è venuta la brillante idea di cercare di educarlo, tanto per divertirsi un po'. Cominciamo quindi questa *descensio ad inferos* di cui al più faremo i primi gradini. Vogliamo infatti costruire un semplice esempio di rete neurale, che serva a classificare dei dati. Quello che vogliamo fare è costruire una scatola nera che prenda in input un certo set di dati a cui corrisponde un certo label, e la nostra scatola nera deve imparare a capire qual è il label a seconda di varie caratteristiche presenti nei dati in input. Metaforicamente potremmo dire che "allenare" una rete neurale è come fare esercizi guardando le soluzioni finché non diventi bravo e impari a farli da solo. Si tratta quindi di apprendimento supervisionato. Vediamo uno schema di una rete neurale e cerchiamo di capire cosa succede.

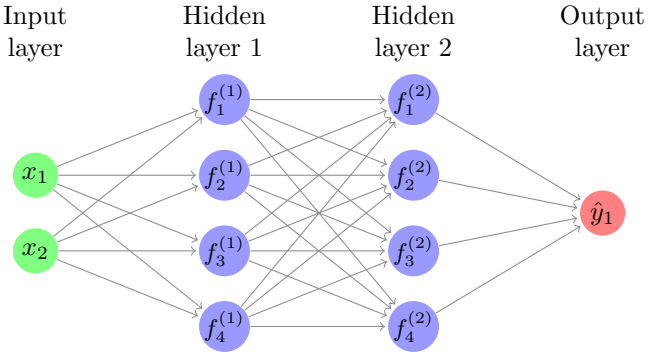

Figura 7: Schema di una rete neurale per una classificazione binaria, eventualmente il layer di output può avere due neuroni in caso di one-hot-encoding

Questo è un piccolo esempio della struttura di una rete neurale. Essa è suddivisa in layer e ogni layer contiene un certo numero di neuroni (i pallini). Il layer di input è quello a cui noi passiamo i dati, mentre il layer di output è quello che ci restituisce il risultato; è negli hidden layer che avviene la magia. Supponiamo di avere dei dati del tipo:

x_1	x_2	y
0.34	0.56	1
0.5	0.89	1
0.2	0.7	0
0.52	0.1	1
0.9	0.83	0
⋮	⋮	⋮

Tabella 1: Tabella di dati da classificare: ogni dato ha due features ovvero le due coordinate x_1 e x_2 ed un label o target ovvero y, cioè abbiamo assegnato ad ogni punto sul piano un valore binario 0 o 1.

Quindi abbiamo dei punti sul piano, senza perdere di generalità supponiamoli fra 0 ed 1, e li abbiamo targhettati con uno zero o con un uno. Vediamo ora come addestrare la rete. Come prima cosa passiamo i dati in input alla rete e facciamo il primo passo, cioè dobbiamo andare verso il primo layer nascosto, quello che in genere si fa è la seguente cosa:

$$\mathbf{z} = W\mathbf{x} + \mathbf{b} \quad . \tag{81}$$

Non molto magico eh? vediamo gli elementi di questa formula $\mathbf{x}$ è il vettore in input quindi $\mathbf{x} = (x_1, x_2)$; W è una matrice e gli elementi vengono chiamati pesi; infine $\mathbf{b}$ è un vettore ed è chiamato bias. Le dimensioni di questa matrice e questo vettore sono dati dalla dimensione del layer: nella fattispecie per il primo layer W sarà una matrice 4×2 e $\mathbf{b}$ un vettore lungo 4. Per cui anche il nostro output $\mathbf{z}$ sarà un vettore lungo 4. Più in generale possiamo dire, essendo questa struttura la stessa per ogni layer, che la dimensione di W sarà sempre della forma: (numero di neuroni nel layer attuale)$\times$(numero di neuroni layer precedente), mentre $\mathbf{b}$ sarà sempre un vettore lungo (numero di neuroni nel layer attuale). La domanda sorge spontanea: che valori hanno le entrate di W e di $\mathbf{b}$? Inizialmente saranno scelte in maniera casuale e poi vedremo che durante le iterazioni di allenamento, dette epoche, esse verranno aggiornate secondo una certa regola. Prima di passare al secondo layer però c'è un'altra operazione da fare. Infatti solitamente non è $\mathbf{z}$ l'output ma $f(\mathbf{z})$ dove f è una certa funzione $f : \mathbb{R} \to \mathbb{R}$. Questa f è chiamata funzione di attivazione del neurone; essa restituirà nel nostro caso un vettore lungo 4 che sarà l'input per il layer successivo. Possiamo vedere tutto ciò come delle regole per far

accendere o spegnere un neurone. Fatto questo il procedimento si ripete in maniera uguale per ogni layer fino ad arrivare al layer di output. Ora che il passaggio attraverso la rete è stato fatto dobbiamo capire se l'output che genera la rete è simile a quello che noi vogliamo. É dunque arrivato il momento di calcolare una funzione che misuri la distanza tra risultato esatto e risultato della rete. Questa funzione è chiamata *loss* e nel nostro caso di classificazione binaria useremo quella che è chiamata binary cross entropy:

$$\mathcal{L} = \frac{1}{N}\sum_{i}^{N}(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)) \quad . \tag{82}$$

Non vi sarà sfuggito il fatto che questa è la classica espressione per l'entropia di un sistema di fermioni. Precisiamo che N è il numero di dati che passiamo per farlo allenare (volendo il numero di righe della tabella di sopra). Fatto questo non ci resta che decidere il modo il cui aggiornare i pesi e bias della rete. Abbiamo detto che la loss misura la distanza di quanto la rete sbaglia, quindi basterà minimizzarla. Trovando il punto di minimo avremo i parametri (pesi e bias) ottimali della nostra rete. Come se stessimo eseguendo un fit, circa. Per farlo usiamo il più semplice e classico algoritmo: il gradiente discendente (di cui è presente una discussione in una delle appendici). Abbiamo quindi che:

$$W^{i+1} = W^i - \alpha\frac{\partial\mathcal{L}}{\partial W^i}, \tag{83}$$

$$\mathbf{b}^{i+1} = \mathbf{b}^i - \alpha\frac{\partial\mathcal{L}}{\partial \mathbf{b}^i}, \tag{84}$$

dove α è chiamato "learning rate" ed è un parametro che scegliamo noi. Immagino non vi sorprenderà sapere che le derivate copra citate si calcolano con la chain rule:

$$\frac{\partial\mathcal{L}}{\partial W} = \frac{\partial\mathcal{L}}{\partial\hat{y}}\frac{\partial f}{\partial z}\frac{\partial(W\mathbf{x}+\mathbf{b})}{\partial W}, \tag{85}$$

$$\frac{\partial\mathcal{L}}{\partial \mathbf{b}} = \frac{\partial\mathcal{L}}{\partial\hat{y}}\frac{\partial f}{\partial z}\frac{\partial(W\mathbf{x}+\mathbf{b})}{\partial \mathbf{b}}. \tag{86}$$

Fatto questo abbiamo dei nuovi pesi e bias con cui ripartire prendendo nuovamente i dati in input e propagarli lungo la rete. Questo passaggio si chiama "feed forward" mentre l'updating è chiamato "backpropagation". Fatti questi, è finita un'iterazione, ovvero un epoca, e parte la successiva. Il numero di epoche è anch'esso un parametro che scegliamo noi. Altra cosa che sta al nostro giudizio è la scelta delle funzioni di attivazione, per le quali, eccezion fatta per il layer di output, non ci sono chissà che criteri per selezionarle. In letteratura si possono trovare le scelte più disparate. Noi useremo per tutti i layer nascosti delle tangenti iperboliche e per il layer di output una sigmoide. Aggiungiamo altre due cose: avendo usato come loss la binary cross entropy può essere interessante misurare l'accuratezza del risultato della rete anche in un altro modo, calcoliamo:

$$a = 1 - \left|\sum_{i}^{N}\frac{\hat{y}_i - y_i}{N}\right|, \tag{87}$$

detta accuratezza più siamo vicini ad uno meglio è però bisogna stare attenti a non overfittare. La rete potrebbe infatti star "imparando a memoria" e questo non va bene. Per controllare che questo non succeda usiamo quella è chiamata validation loss. Si tratta sempre di calcolare la loss, ma non sui dati che usiamo come dati di allenamento, ma su un altro set che usiamo esclusivamente per questo calcolo. Quindi noi passiamo alla rete dei dati e li dividiamo in due, su un set allena e sull'altro valida. Quello che facciamo è plottare insieme la loss calcolata sui dati di allenamento e quella calcolata sui dati di validation (in funzione delle epoche). Se le due scendono insieme la rete si sta comportando bene. Se invece ad un certo punto la loss scende ma la loss di validation sale vuol dire che la rete sta overfittando, cioè sta imparando a memoria le caratteristiche dei dati di allenamento. Quindi per evitare questo bisogna scegliere alcuni parametri con un po' di senno, ad esempio learning rate, numero di neuroni e numero di layer o anche numero di epoche. Detto questo passiamo ora a vedere il codice. Implementeremo una rete neurale con un solo layer nascosto con numero di neuroni variabile, due neuroni in input e uno solo in output. Lo scopo sarà proprio di riconoscere dati dei punti nel quadrato $[0,1]\times[0,1]$ se essi hanno un valore associato 1 oppure 0. L'inizializzazione verrà fatta estraendo dei numeri distribuiti gaussianamente. Omettiamo per brevità, come sempre, le funzioni che fanno i plot e quello che è il main del codice.

```
"""
Code that implement a shallow neural network for a binary classifications.
The code is witten imposed 2 imput, 1 output e one hidden layer.
Is possible to choose the dimesions of hidden layer.
It is also possible to save plots during the run to see how the network is learning.
```

```
"""
import numpy as np
import matplotlib.pyplot as plt

#============================================================
# Loss function binary classification
#============================================================

def Loss(Yp, Y):
    '''
    loss function, binary crosss entropy

    Parameters
    ----------
    Yp : 1darray
        actual prediction
    Y : 1darray
        Target

    Returns
    -------
    float, binary crosss entropy
    '''
    m = len(Y)
    return -np.sum(Y*np.log(Yp) + (1 - Y)*np.log(1 - Yp))/m

#============================================================
# Activation function
#============================================================

# Hidden layer
def g1(x):
    return np.tanh(x)
# Output layer
def g2(x):
    return 1 / (1 + np.exp(-x))

#============================================================
# Initialization
#============================================================

def init(n):
    '''
    Random initialization of parameters weights and biases

    Parameters
    ----------
    n : int
        number of neurons in the hidden layer

    Returns
    -------
    W1, b1 : 2darray
        weights and bias for hidden layer
    W2, b2 : 2darray
        weights and bias for output layer
    '''
    # Hidden layer
    # nx2 because 2 featurs and n neurons
    W1 = np.random.randn(n, 2)
    b1 = np.random.rand(n, 1)
    # Output layer
    # 1xn because 1 output and n neurons
    W2 = np.random.randn(1, n)
    b2 = np.random.rand(1, 1)
    return W1, b1, W2, b2

#============================================================
# Network prediction function
#============================================================

def predict(X, W1, b1, W2, b2):
    """
    Function that returns the prediction of the network

    Parameters
```

```
    ----------
    X : 2darray
        data, featurs
    W1, b1, W2, b2 : 2darray
        parameter of the network

    Returns
    -------
    A1 : 1d array
        intermediate prediction
    A2 : 1d array
        final prediction
    """
    # Hidden layer
    Z1 = W1 @ X + b1
    A1 = g1(Z1)
    # Output layer
    Z2 = W2 @ A1 + b2
    A2 = g2(Z2)

    return A1, A2

#=============================================================
# Backpropagation function
#=============================================================

def backpropagation(X, Y, step, A1, A2, W1, b1, W2, b2):
    '''
    Backpropagation function.
    Update weights and biases with gradient descendent
    all the quantities came from taking the derivative of the Loss

    Y : 1darray
        Target
    step : float
        learning rate
    A1, A2 : 1darray
        predictions of the network
    W1, b1, W2, b2 : 2darray
        parameter of the network
    '''
    m = len(Y)
    # Output layer
    dLdZ2 = (A2 - Y)
    dLdW2 = dLdZ2 @ A1.T / m
    dLdb2 = np.sum(dLdZ2, axis=1)[:, None] / m
    # Hidden layer
    dLdZ1 = W2.T @ dLdZ2 * (1 - A1**2)
    dLdW1 = dLdZ1 @ X.T / m
    dLdb1 = np.sum(dLdZ1, axis=1)[:, None] / m

    # Update of parameters
    W1 -= step * dLdW1
    b1 -= step * dLdb1
    W2 -= step * dLdW2
    b2 -= step * dLdb2

    return W1, b1, W2, b2

#=============================================================
# Accuracy mesuraments
#=============================================================

def accuracy(Yp, Y):
    '''
    accuracy of prediction. We use:
    accuracy = 1 - | sum ( prediction - target )/target_size |

    Parameters
    ----------
    Yp : 1darray
        actual prediction
    Y : 1darray
        Target

    Returns
```

```
    -------
    a : float
        accuracy
    '''
    m = len(Y)
    a = 1 - abs(np.sum(Yp.ravel() - Y)/m)
    return a

#============================================================
# Train of the network
#============================================================

def train(X, Y, n_epoch, neuro, step, sp=False, verbose=True):
    '''
    function for the training of the network

    Parameters
    ----------
    X : 2darray
        data, featurs
    Y : 1darray
        Target
    n_epoch : int
        number of epoch
    neuro : int
        number of neurons in the hidden layer
    step : float
        learning rate
    sp : boolean, optional, default False
        if True a plot of boundary is saved each 100 epoch
        usefull for animations
    verbose : boolean, optional, default True
        if True print loss and accuracy each 100 epoch

    Returns
    -------
    result : dict
        params -> W1, b1, W2, b2 weights and bias of network
        train_Loss -> loss on train data
        valid_Loss -> loss on validation data
    '''

    W1, b1, W2, b2 = init(neuro)
    L_t = np.zeros(n_epoch) # training loss
    L_v = np.zeros(n_epoch) # validation loss
    N = X.shape[1]          # total number of data
    M = N//4                # nuber of data for validation

    # split dataset in validation and train
    X_train, Y_train = X[:, :N-M ], Y[:N-M ]
    X_valid, Y_valid = X[:,  N-M:], Y[ N-M:]

    for i in range(n_epoch):
        # train
        A1, A2 = predict(X_train, W1, b1, W2, b2)
        L_t[i] = Loss(A2, Y_train)
        # validation
        _, Yp = predict(X_valid, W1, b1, W2, b2)
        L_v[i] = Loss(Yp, Y_valid)
        # update
        W1, b1, W2, b2 = backpropagation(X_train, Y_train, step, A1, A2, W1, b1, W2, b2)

        if not i % 100:
            #if sp : plot(X_train, Y_train, (W1, b1, W2, b2), i)

            if verbose:
                acc = accuracy(A2, Y_train)
                print(f'Loss = {L_t[i]:.5f}, accuracy = {acc:.5f}, epoch = {i} \r', end='')

    if verbose: print()

    result = {'params'     : (W1, b1, W2, b2),
              'train_Loss' : L_t,
              'valid_Loss' : L_v,
             }

```

```
234     return result
```

Dopo queste belle tre paginate di codice vediamo un po' di risultati. Questo codice era scritto per far vedere come rete impara creando un gif, che trovate disponibile sulla cartella. Il risultato finale è il seguente:

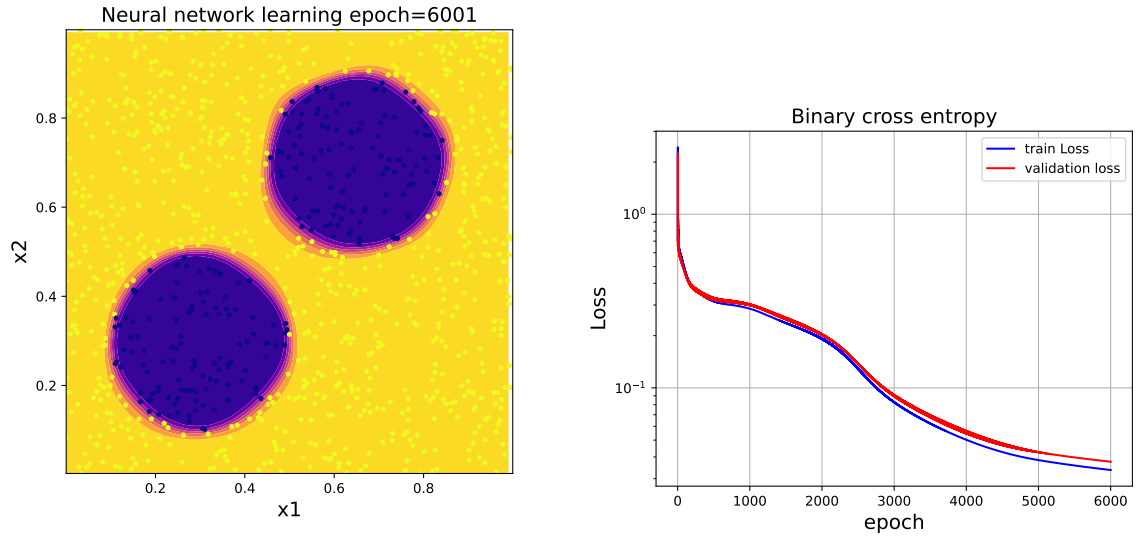

Figura 8: Risultati della rete neurale, predizione e andamento della loss. I parametri qui sono: 20 neuroni nel layer nascosto, un learning rate di 1.5, 6001 epoche, 3000 dati di train e 1000 di validation.

Il plot a sinistra è fatto sui dati di test, quindi un set di dati che la rete non ha usato per allenarsi e il risultato sembra soddisfacente, vediamo poi che le due loss scendono insieme il che è segno che la rete si comporta bene. Volendo ora farvi vedere andamenti diversi della loss vi mostro due grafici provenienti da un altro codice, in cui ho implementato una rete neurale in maniera leggermente diversa e un po' più generale. Anche perché, se vedete la loss nella figura sopra, noterete che la linea è un po' spessa, dovuto al fatto che sta oscillando (immagino sia una concausa tra problema da affrontare e algoritmo di minimizzazione). Il problema è questa volta il MNIST, ovvero il riconoscimento di cifre scritte a mano in bassa risoluzione, ogni immagine è 28×28 pixel. Vediamo un caso in cui la rete overfitta e un caso in cui si comporta bene:

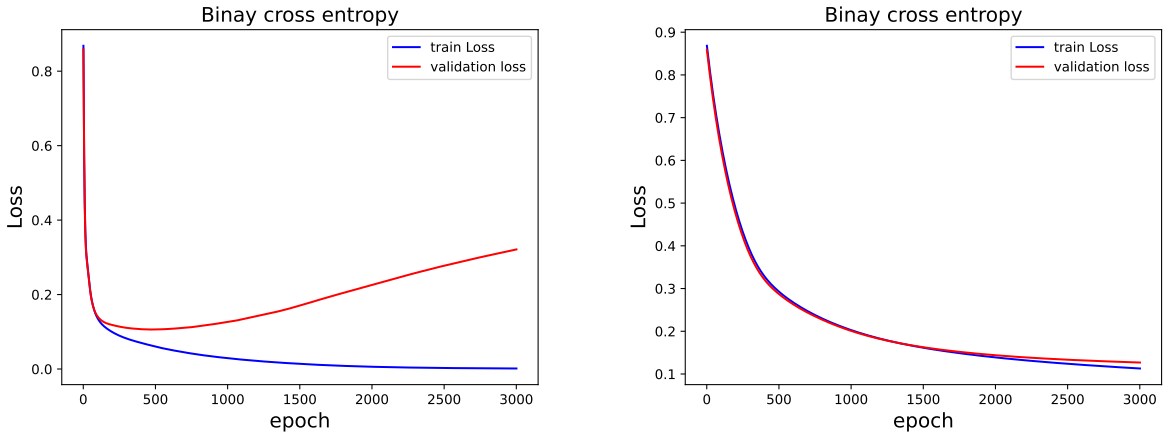

Figura 9: Andamento della loss in funzione delle epoche. Nel grafico a sinistra la rete sta overfittando, mentre a destra l'allenamento procede bene. I parametri qui sono: 2 layer nascosti ognuno da 50 neuroni; l'ottimizzatore usato è ADAM e il learning rate iniziale è di 0.05 per il caso di overfit e di 0.01 per quello di buon fit; le funzioni di attivazione sono: per i leayer nascosti le "relu", mentre una sigmoide per il layer finale. Le epoche sono 3001, 2250 i dati di train e 750 quelli di validation.

Vediamo quindi che anche solo il cambiamento del learning rate iniziale (dove si specifica iniziale in quanto ADAM è un algoritmo adattivo) provoca un comportamento che non vogliamo, facendo overfittare la rete. Questo lo possiamo vedere anche, trattandosi di un classificatore, grazie a quelle che sono le matrici di confusione. Mostriamo di seguito le matrici nel caso di overfit e di fit facendo anche il confronto vedendo i dati train

e quelli di test. Spieghiamo velocemente come si legge questa matrice: sull'asse delle ordinate è presente la predizione della rete mentre su quello delle ascisse ci sta il risultato esatto. Le entrate di questa matrice ci dicono fondamentalmente quante volte la rete ha detto "x" e doveva dire "y"; quindi gli elementi sulla diagonale sono le risposte esatte mentre i termini fuori sono risposte sbagliate. Per chiarezza precisiamo che questa matrice viene calcolata sempre a fine allenamento.

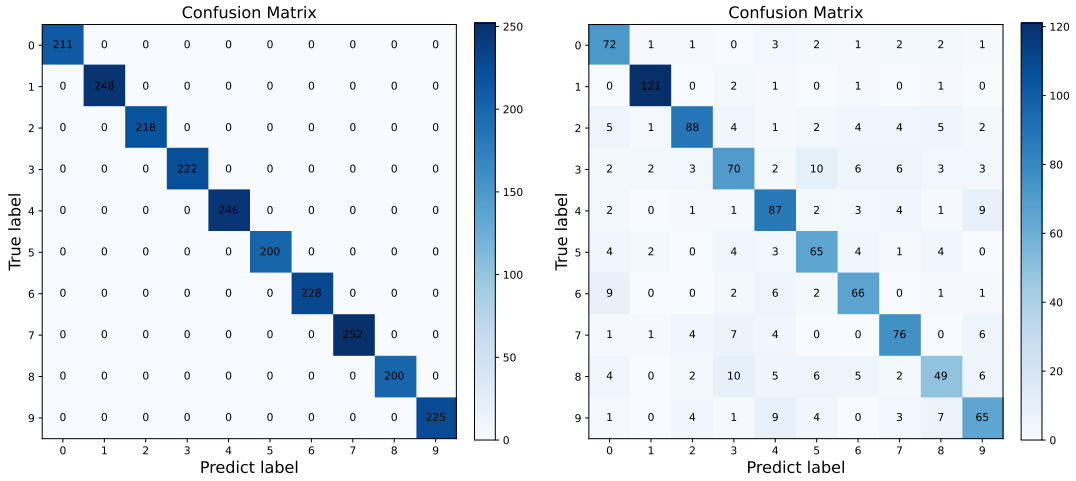

Figura 10: Vediamo qui le matrici nel caso di overfit calcolate sui dati di train, a sinistra, e su quelli di test, a destra. Avendo imparato a memoria le caratteristiche dei dati di train vediamo effettivamente che la rete non sbaglia una singola predizione. Mentre a destra, su dati che per la rete sono nuovi, ci sono molti più errori.

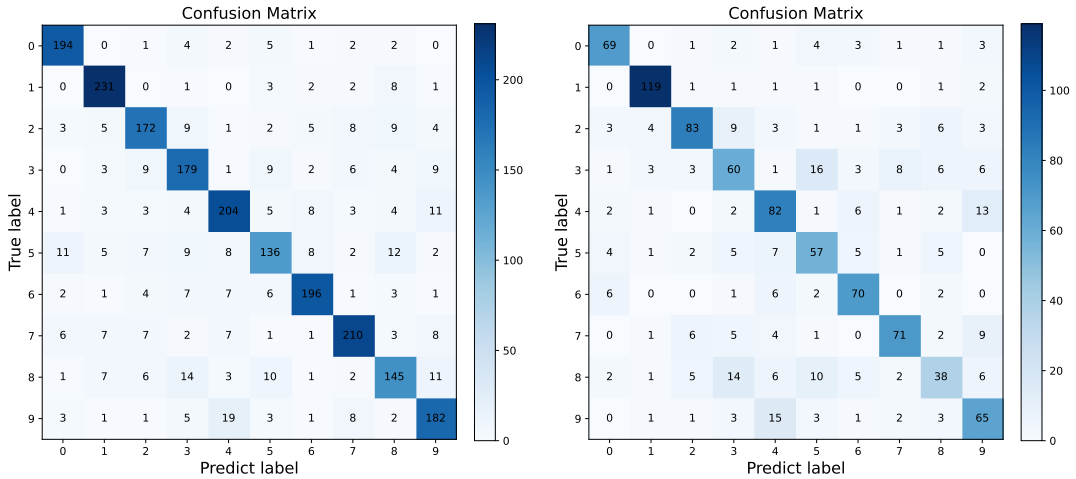

Figura 11: Vediamo qui le matrici nel caso di buon fit calcolate sui dati di train, a sinistra, e su quelli di test, a destra. Questa volta la rete non ha imparato a memoria, vediamo infatti che ci sono risposte sbagliate un po' ovunque. Le due matrici sembrano parenti.

Per concludere questa discussione in caso fosse di vostro interesse vi mostro il codice per calcolare queste matrici di confusione:

```
def confmat(true_target, pred_target, plot=True, k=0):
    '''
    Function for creation and plot of confusion matrix

    Parameters
    ----------
    true_target : 1darray
        vaules that must be predict
    pred_target : 1darray
        values that the network has predict
```

```
    plot : bool, optional, default True
        if True the matix is plotted.
    k : int, optional, default 0
        number of figure, necessary in order not to overlap figures

    Return
    ------
    mat : 2darray
        confusion matrix
    '''

    dat = np.unique(true_target)      # classes
    N   = len(dat)                    # Number of classes
    mat = np.zeros((N, N), dtype=int) # confusion matrix

    # creation of confusion matrxi
    for i in range(len(true_target)):
        mat[true_target[i]][pred_target[i]] += 1

    if plot :
        fig = plt.figure(0, figsize=(7, 7))
        ax = fig.add_subplot()

        c = ax.imshow(mat, cmap=plt.cm.Blues) # plot matrix
        b = fig.colorbar(c, fraction=0.046, pad=0.04)
        # write on plot the value of predictions
        for i in range(mat.shape[0]):
            for j in range(mat.shape[1]):
                ax.text(x=j, y=i, s=mat[i, j],
                    va='center', ha='center')

        # Label
        ax.set_xticks(dat, dat)
        ax.set_yticks(dat, dat)
        ax.tick_params(top=False, bottom=True, labeltop=False, labelbottom=True)

        plt.xlabel('Predict label', fontsize=15)
        plt.ylabel('True label', fontsize=15)
        plt.title('Confusion Matrix', fontsize=15)
        plt.tight_layout()

    return mat
```