PROGETTO
MACHINE
LEARNING
2022

Neural Networks and Deep Learning

Alessandro Quirile N97/402

Università degli Studi di Napoli Federico II
Laurea Magistrale in Informatica LM-18

Indice

Tabella 1: Notazione adottata in questa relazione

Simbolo	**Significato**
$\mathbb{R}$	L'insieme dei numeri reali
$\mathbb{R}^d$	L'insieme dei vettori d-dimensionali su $\mathbb{R}$
$\mathbb{1}_{[expr]}$	La funzione indicatrice
$\mathbf{v}$	Un vettore
v_i	L'i-esima componente del vettore $\mathbf{v}$
$\langle \mathbf{v}, \mathbf{w} \rangle$	Il prodotto scalare dei vettori $\mathbf{v}$ e $\mathbf{w}$
$\mathcal{M}_{m \times n}(\mathbb{R})$	L'insieme delle matrici di dimensione $m \times n$ a coefficienti in $\mathbb{R}$
A	Una matrice
a_{ij}	L'elemento alla cella (i, j) di A
$f'(x)$	La derivata di una funzione $f : \mathbb{R} \to \mathbb{R}$ nel punto x
$\frac{\partial f(\mathbf{w})}{\partial w_i}$	La derivata parziale di una funzione $f : \mathbb{R}^d \to \mathbb{R}$ nel punto $\mathbf{w}$ rispetto a w_i
$\nabla f(\mathbf{w})$	Il gradiente di una funzione $f : \mathbb{R}^d \to \mathbb{R}$ nel punto $\mathbf{w}$
log	Il logaritmo naturale
a_i^h	L'input del neurone i-esimo nello strato h
z_i^h	L'output del neurone i-esimo nello strato h
w_{ij}^h	Il peso della connessione dal nodo j al nodo i nello strato h
b_i^h	Il bias del neurone i-esimo appartenente allo strato h
$\mathbb{P}(C_k \vert \mathbf{x})$	Probabilità condizionata C_k dato $\mathbf{x}$

Capitolo 1

Introduzione

Questa relazione fa riferimento alla traccia numero 2 del progetto di *Neural Networks and Deep Learning* che consiste nella progettazione ed implementazione di funzioni per la forward propagation di una rete neurale multistrato e di funzioni per la backpropagation, per qualsiasi scelta della funzione di attivazione, con la possibilità di usare almeno la somma di quadrati o la cross-entropy (con e senza softmax) come funzione di errore.

Si consideri come input le immagini raw del dataset `MNIST`. Si ha, allora, un problema di classificazione a 10 output. Si estragga opportunamente un dataset globale e lo si divida opportunamente in training, validation e test set. Si fissi la discesa del gradiente con momento come algoritmo di aggiornamento dei pesi e si studi l'apprendimento di una rete neurale (ad esempio epoche necessarie per l'apprendimento, andamento dell'errore sul training e validation set, accuratezza sul test set) con un solo strato di neuroni interni al variare del learning rate e del momento per almeno cinque diverse dimensioni dello strato interno. Scegliere e rimanere invariati tutte le altre scelte, come ad esempio le funzioni di output.

Capitolo 2

Analisi del problema

Le principali entità coinvolte nel problema sono le seguenti:

1. Rete neurale - un modello matematico caratterizzato da alcune proprietà come, ad esempio, un insieme di uno o più strati detti *layer*
2. Layer - un insieme di uno o più neuroni, caratterizzato da un tipo (input, hidden o output) e da una funzione di attivazione che viene assunta unica dalla traccia per ogni neurone appartenente a quel layer
3. Immagine - un oggetto che deve essere "scansionato" dalla rete e classificato in una delle 10 classi disponibili (una per ogni cifra da 0 a 9)

2.0.1 L'algebra dietro le scelte progettuali

Dato per noto il comportamento della rete, ed il fatto che essa realizzi un mapping funzionale tra un vettore in input $\mathbf{x} \in \mathbb{R}^d$ ed un vettore di output $\mathbf{y} \in \mathbb{R}^c$, bisogna cercare un modo efficiente ed elegante per codificare pesi, bias, uscite di ciascun nodo e somme pesate. Per fare ciò si può utilizzare la notazione matriciale: supponiamo di avere una rete shallow con uno strato di input con d neuroni, uno strato interno con m neuroni ed uno strato di output con c neuroni, allora possiamo individuare due matrici di pesi: la prima, che chiameremo W^1, codifica i pesi che vanno dai nodi di input a quelli interni, mentre la seconda, che chiameremo W^2, i pesi che vanno dai nodi interni a quelli di uscita.

Limitiamoci a definire formalmente cosa accade per i pesi dallo strato di input a quello interno: ricordando che w_{ij} codifica il peso della connessione dal neurone j al

neurone i allo strato h, allora:

$$W^1 = \begin{pmatrix} w^1_{11} & w^1_{12} & w^1_{13} & \dots & w^1_{1d} \\ w^1_{21} & w^1_{22} & w^1_{23} & \dots & w^1_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^1_{m1} & w^1_{m2} & w^1_{m3} & \dots & w^1_{md} \end{pmatrix} \in \mathcal{M}_{m\times d}(\mathbb{R})$$

Analogamente per i bias, sotto forma di vettore colonna:

$$\mathbf{b}^1 = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \in \mathcal{M}_{m\times 1}(\mathbb{R})$$

E quindi gli input che ciascun neurone nello strato interno riceve potrà essere codificato analogamente, trasponendo opportunamente il vettore riga $\mathbf{x} \in \mathbb{R}^d$ in vettore colonna per garantire che la matrice W^1 ed il vettore $\mathbf{x}$ siano *conformabili*[1]:

$$\mathbf{a}^1 = W^1\mathbf{x}^t + \mathbf{b}^1 = \begin{pmatrix} \sum_{j=1}^d w^1_{1j}x_j + b^1_1 \\ \sum_{j=1}^d w^1_{2j}x_j + b^1_2 \\ \vdots \\ \sum_{j=1}^d w^1_{mj}x_j + b^1_m \end{pmatrix} \in \mathcal{M}_{m\times 1}(\mathbb{R})$$

A questo punto è immediato calcolare gli output dei neuroni dello strato interno applicando al vettore $\mathbf{a}$ un'opportuna funzione di attivazione scelta f:

$$\mathbf{z}^1 = f(\mathbf{a}^1) = \begin{pmatrix} f(\sum_{j=1}^d w^1_{1j}x_j + b^1_1) \\ f(\sum_{j=1}^d w^1_{2j}x_j + b^1_2) \\ \vdots \\ f(\sum_{j=1}^d w^1_{mj}x_j + b^1_m) \end{pmatrix} \in \mathcal{M}_{m\times 1}(\mathbb{R})$$

In questo modo abbiamo codificato numerosi dati in matrici relativamente semplici da manipolare; più precisamente è possibile definire le seguenti operazioni binarie interne:

$$+ : (A, B) \in \mathcal{M}_{m\times n}(\mathbb{R}) \times \mathcal{M}_{m\times n}(\mathbb{R}) \mapsto A + B \in \mathcal{M}_{m\times n}(\mathbb{R})$$
$$\cdot : (\alpha, A) \in \mathbb{R} \times \mathcal{M}_{m\times n}(\mathbb{R}) \mapsto \alpha \cdot A \in \mathcal{M}_{m\times n}(\mathbb{R})$$

[1]Due matrici A e B sono conformabili quando il numero di colonne di A è uguale al numero di righe di B

suggerendo il fatto che una matrice possa essere ottenuta come *combinazione lineare* di matrici definite sullo stesso campo, $\mathbb{R}$ nel nostro caso.

Un approccio simile può essere seguito anche per gli elementi del dataset $\mathcal{X} \times \mathcal{Y}$ e per le relative predizioni $\hat{Y}$, le quali verranno codificate come matrici dove l'i-esima colonna corrisponde alla predizione sull'elemento alla colonna i-esima di X. Più precisamente, denotando con X la matrice dei campioni in input alla rete, con d il numero di feature di ciascun campione, con s il numero di campioni, con $\mathbf{y}$ il vettore delle vere etichette e con $\hat{\mathbf{y}}$ il vettore delle predizioni, allora:

$$X = \begin{pmatrix} x_1 & x_2 & ... & x_d \\ x_1 & x_2 & ... & x_d \\ \vdots & \vdots & \ddots & \vdots \\ x_1 & x_2 & ... & x_d \end{pmatrix} \in \mathcal{M}_{s\times d}(\mathbb{R}) \quad \xrightarrow{\text{transpose}} \quad X^t = \begin{pmatrix} x_1 & x_1 & ... & x_1 \\ x_2 & x_2 & ... & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_d & x_d & ... & x_d \end{pmatrix} \in \mathcal{M}_{d\times s}(\mathbb{R})$$

$$\mathbf{y} = (5, 0, 4, ..., 8) \in \mathbb{R}^s \quad \xrightarrow{\text{one-hot}} \quad Y = \begin{pmatrix} 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ 1 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 1 \\ 0 & 0 & 0 & ... & 0 \end{pmatrix} \in \mathcal{M}_{c\times s}(\mathbb{R})$$

$$\hat{Y} = \begin{pmatrix} \hat{\mathbf{y}}_\mathbf{1} & \hat{\mathbf{y}}_\mathbf{2} & ... & \hat{\mathbf{y}}_\mathbf{s} \end{pmatrix} \in \mathcal{M}_{1\times s}(\mathbb{R})$$

dove $\hat{\mathbf{y}}_\mathbf{i} \in \mathcal{M}_{c\times 1}(\mathbb{R})$ è il vettore colonna relativo alla predizione dell'item nella colonna i-esima di X^t, per ogni $1 \leq i \leq s$. Si può notare che la codifica one-hot implica che $\sum_{k=1}^{c} t_k^n = 1$.

Per quanto riguarda i dati del training set di MNIST, $d = 28 \times 28 = 784$, $s = 60,000$ e $c = 10$.

Capitolo 3

Implementazione

Per l'implementazione si è utilizzato Python 3.10 e un approccio Object-Oriented misto procedurale. Gli attori "Rete Neurale" e "Layer" che sono stati individuati al Capitolo 2 verranno implementati come *classi*, mentre "Immagine" verrà modellata come un vettore d-dimensionale, con $d = 28 \times 28$, da dare in input alla rete opportunamente normalizzato, come vedremo.

In questo capitolo verranno illustrate dapprima ogni funzione atomica che è stata implementata, per poi approdare al codice principale vero e proprio che richiamerà tali funzioni. Nel progetto è stato fatto uso principalmente della libreria `numpy` per il calcolo scientifico ottimizzato.

3.1 `activation_functions.py`

Questo file definisce le funzioni di attivazione che potranno essere associate a ciascun layer della rete.

Le funzioni di attivazione definite sono la sigmoide, la funzione identità e la ReLU:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \quad id(a) = a \quad ReLU(a) = \max(0, a)$$

Occorre specificare un vettore `a` di input per ogni funzione di attivazione e, opzionalmente, un valore booleano `derivative` che specifica se calcolare la derivata della funzione in quel punto. Il tipo dell'input `a` è `numpy.ndarray` che rappresenta un vettore omogeneo *multidimensionale* di taglia prefissata: gli algoritmi funzionano quindi per tensori, per matrici, per vettori e anche per semplici scalari. Non è un

caso che `numpy.matrix` sia una sottoclasse di `numpy.ndarray`. Si ricordi, inoltre, che Python è un linguaggio debolmente e dinamicamente tipato:

```
import numpy as np

def sigmoid(a, derivative=False):
  f_a = 1 / (1 + np.exp(-a))
  if derivative:
    return np.multiply(f_a, (1 - f_a))
  return f_a


def identity(a, derivative=False):
  f_a = a
  if derivative:
    return np.ones(np.shape(a))
  return f_a


def relu(a, derivative=False):
  f_a = np.maximum(0, a)
  if derivative:
    return (a > 0) * 1
  return f_a
```

Un esempio di funzionamento è dato in Figura:

```
>>> a = np.array([-5, 0, 5])
>>> sigmoid(a)
array([0.00669285, 0.5       , 0.99330715])
>>> identity(a)
array([-5,  0,  5])
>>> relu(a)
array([0, 0, 5])
```

3.2 post_processing_functions.py

Questo file contiene la definizione della funzione di post processing softmax che consente di interpretare il generico output della rete y_k come probabilità condizionata $\mathbb{P}(C_k|\mathbf{x})$ per ogni $1 \leq k \leq c$. Se tramite un processo di learning riesco a interpretare le risposte della rete y_k sotto forma di probabilità condizionate, allora posso costruire un classificatore la cui regola di decisione è la seguente: $\mathbb{P}(C_i|\mathbf{x}) > \mathbb{P}(C_j|\mathbf{x}) \implies \mathbf{x} \in C_i$ per ogni $1 \leq i \neq j \leq c$, con l'obiettivo di minimizzare la probabilità di *misclassification*. Occorre quindi trasformare ciascun output della rete applicando la seguente regola:

$$z_k^n = \frac{e^{y_k^n}}{\sum_{k=1}^{c} e^{y_k^n}}$$

Implementata in questo modo:

```
import numpy as np

def softmax(y):
  epsilon = 10 ** -308
  e_y = np.exp(y - np.max(y, axis=0))
  sm = e_y / np.sum(e_y, axis=0)
  return np.clip(sm, epsilon, 1 - epsilon)
```

Viene utilizzata una variabile `epsilon` ed il metodo `np.clip` al solo fine di rendere l'algoritmo più stabile numericamente. Con `axis=0` viene considerato il massimo elemento per ciascuna *colonna*, poiché essa rappresenta – come visto – la predizione sul *singolo* codificato come vettore *colonna*. Un esempio è dato in Figura per una matrice `y` di dimensione 3×4 riempita con 12 valori interi consecutivi, da 0 a 11:

```
>>> y = np.asmatrix(np.arange(12)).reshape(3,4)
>>> softmaxed_y = softmax(y)
>>> softmaxed_y
matrix([[3.29320439e-04, 3.29320439e-04, 3.29320439e-04, 3.29320439e-04],
        [1.79802867e-02, 1.79802867e-02, 1.79802867e-02, 1.79802867e-02],
        [9.81690393e-01, 9.81690393e-01, 9.81690393e-01, 9.81690393e-01]])
>>> np.sum(softmaxed_y, axis=0)
matrix([[1., 1., 1., 1.]])
```

Si noti che la somma degli elementi di ciascuna colonna è 1, com'è giusto che sia, siccome ogni elemento di ciascuna colonna rappresenta una probabilità.

3.3 loss_functions.py

Questo file contiene la definizione delle funzioni di loss quali la sum of squares

$$E^n = \frac{1}{2}\sum_{k=1}^{c}(y_k^n - t_k^n)^2$$

e la cross entropy multiclasse *categorical* (non *sparse*, infatti richiede che i target siano codificati one-hot)

$$E^n = -\sum_{k=1}^{c} t_k^n \log y_k^n$$

dove y denota la matrice di predizioni e t la matrice dei target.

```
from post_processing_functions import *

def sum_of_squares(y, t, derivative=False):
  if derivative:
    return y - t
  return 0.5 * np.sum(np.sum(np.square((y - t))))


def cross_entropy(y, t, derivative=False, post_process=True):
  if post_process:
    if derivative:
      return y - t
    sm = softmax(y)
    losses = -np.sum(np.multiply(t, np.log(sm)), axis=0)
    return np.mean(losses)
  else:
    if derivative:
      return -np.sum(np.divide(t, y), axis=1)
    losses = -np.sum(np.multiply(t, y), axis=0)
    return np.mean(losses)
```

Nel calcolo della cross-entropy viene utilizzato il parametro `post_process` che stabilisce se applicare la funzione `softmax` agli output della rete. L'errore di ciascuna predizione viene calcolato, come già visto precedentemente, in base a ciascuna colonna (`axis=0`). Questa implementazione consente di confrontare gli errori anche qualora due dataset avessero cardinalità molto differenti, rendendo quindi di fatto il valore restituito dalla cross-entropy "indipendente" rispetto alla taglia del dataset –

come spesso viene fatto in letteratura. A valle della backpropagation questo comporterà un decremento della norma del gradiente, che diventerà N volte più piccolo dove N è la taglia del dataset (ovvero il numero di colonne della matrice dei target – o delle predizioni), lasciando però invariate proprietà qualitative come direzione e verso. Questo consentirà inoltre di scegliere il learning rate, rispetto ad una delle tipologie di learning (online, batch o mini-batch) in maniera meno sensibile e più rilassata.

Un esempio di funzionamento per le due funzioni di errore quando ricevono in ingresso dei *vettori* è dato in Figura:

```
>>> y = np.array([-1.2, 3, 0.7])
>>> t = np.array([0,1,0])
>>> sum_of_squares(y, t)
2.965
>>> cross_entropy(y, t)
0.1090825587424627
```

Un esempio di funzionamento per la `cross_entropy` quando riceve in ingresso *matrici* `y` e `t` è in Figura:

```
y
matrix([[-0.02,  1.2 ],
        [ 0.3 , -3.2 ]])
t
matrix([[0, 0],
        [1, 1]])
sm = softmax(y)
sm
matrix([[0.42067575, 0.98787157],
        [0.57932425, 0.01212843]])
losses = -np.sum(np.multiply(t, np.log(sm)), axis=0)
losses
matrix([[0.54589294, 4.41220258]])
np.mean(losses)
2.4790477608938857
cross_entropy(y,t)
2.4790477608938857
```

Come si nota la funzione `cross_entropy` dapprima applica la `softmax` alla matrice di predizioni `y` (se specificato dal flag `post_process`), poi calcola un vettore di errori

`losses` dove l'i-esimo elemento rappresenta l'errore relativo alla predizione i-esima, ed infine calcola la media degli errori.

3.4 metrics.py

Il file `metrics.py` contiene le definizioni di alcune metriche, come l'accuratezza, che possono essere usate per valutare, sui dati di test, il modello che minimizza l'errore sul validation set.

La formula usata è:

$$acc(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{y_i = \hat{y}_i}$$

La cui implementazione è la seguente:

```
import numpy as np

from utils import one_hot_to_label

def accuracy_score(targets, predictions):
  targets = one_hot_to_label(targets)
  predictions = one_hot_to_label(predictions)
  return np.mean(predictions == targets)
```

Vengono trasformati i target e le predizioni dalla codifica one-hot (categorica) a quella numerica (sparsa) e viene contato il numero di corrispondenze, per poi dividere il risultato ottenendo la media. Un esempio di funzionamento è dato in Figura:

```
>>> t = np.array([0,1,0])
>>> y = np.array([0.1, 0.8, 0.1])
>>> accuracy_score(t, y)
1.0
```

3.5 utils.py

Questo file contiene la definizione di funzioni di generica utilità, come ad esempio funzioni per disegnare il grafico degli errori oppure per trasformare la codifica categorica in quella sparsa:

```
import math
import numpy as np
from collections import Counter
from matplotlib import pyplot as plt

# Trasforma un vettore dalla codifica one-hot a quella numerica
def one_hot_to_label(targets):
  return np.asarray(np.argmax(targets, axis=0))


# Disegna i grafici degli errori sul training e validation
# Se show_best_net e' True, disegna un pallino in corrispondenza
# Del piu' piccolo errore sul validation set
def plot_losses(epochs, train_losses, val_losses, show_best_net=
                                    False):
  plt.plot(epochs, train_losses)
  plt.plot(epochs, val_losses, color="orange")
  legends = ["Training Loss", "Validation Loss"]
  if show_best_net:
    xmin = epochs[np.argmin(val_losses)]
    ymin = np.min(val_losses)
    plt.scatter(xmin, ymin, marker=".", c="orange")
    legends.append("Best model")
  plt.legend(legends)
  plt.xlabel("Epochs")
  plt.ylabel("Loss")
  plt.grid(True)
  plt.show()


# Disegna il grafico delle accuratezze su training e sul validation
def plot_accuracies(epochs, train_accuracies, val_accuracies):
  plt.plot(epochs, train_accuracies)
  plt.plot(epochs, val_accuracies, color="orange")
  plt.legend(["Training Accuracy", "Validation Accuracy"])
  plt.xlabel("Epochs")
  plt.ylabel("Accuracy")
  plt.grid(True)
  plt.show()
```

```
# Disegna una distribuzione di probabilita' uniforme U[a,b]
def plot_distribution(a, b, samples):
  data = np.random.uniform(a, b, samples)
  plt.hist(data, facecolor='blue')
  plt.xlabel(f"X~U[{a},{b}]")
  plt.ylabel('Count')
  plt.grid(True)
  plt.show()


# Verifica se le classi sono tutte bilanciate
def balanced(targets_train, targets_val, targets_test, tolerance=3):
  rules = [is_balanced(targets_train, tolerance),
           is_balanced(targets_val, tolerance),
           is_balanced(targets_test, tolerance)]
  return all(rules)


# Verifica se targets contiene un numero bilanciato di classi
# All'interno di un singolo set
def is_balanced(targets, tolerance):
  c = Counter(targets)
  percentages = [v / c.total() * 100 for v in c.values()]
  return math.isclose(min(percentages), max(percentages), abs_tol=
                                    tolerance)
```

Come si vedrà nel codice `main`, la verifica di bilanciamento viene fatta per controllare che, a seguito dello split dei dati, ciascuna delle tre *partizioni* contenga campioni statisticamente significativi. Il bilanciamento delle classi, inoltre, consente di limitare il testing del modello "migliore" alla sola accuratezza, senza utilizzare altre metriche come *precision*, *recall* e *media armonica* $F1$. Il controllo viene fatto calcolando le percentuali di ricorrenza di ciascuna classe nell'insieme `targets` e valutando che il minimo ed il massimo siano distanti entro una `tolerance`.

Ad esempio se `targets` contiene le etichette `["Cane", "Gatto", "Cane", "Rana", "Cane"]`, allora `Counter('Cane': 3, 'Gatto': 1, 'Rana': 1)`, le percentuali sono, rispettivamente, 60%, 20% e 20% e quindi `is_balanced(targets, 2)` dà `False` mentre se `targets` contiene le etichette `["Cane", "Gatto", "Rana"]`, allora `is_balanced(targets, 2)` dà `True`.

3.6 data_processing.py

Questo file definisce le funzioni per processare i dati e renderli idonei ad una rete neurale fully-connected:

```
import numpy as np
import sklearn

# Trasforma le feature in numeri tra [0;1]
# Facoltativamente mescola gli item del dataset
def normalize(X, targets, shuffle):
  if shuffle:
    X, targets = sklearn.utils.shuffle(X, targets.T)
  X = (X - np.min(X)) / (np.max(X) - np.min(X))
  return X, targets


# Partizionamento del training set
# Genera il validation set
# Il training set conterra', alla fine, val_size colonne in meno
def split(X_train, targets_train, val_size):
  X_val, X_train = np.hsplit(X_train, [val_size])
  targets_val, targets_train = np.hsplit(targets_train, [val_size])
  return X_val, targets_val, X_train, targets_train


# Codifica one-hot delle etichette
def one_hot(targets):
  return np.asmatrix(np.eye(targets.max() + 1)[targets]).T
```

• Il metodo `normalize` trasforma le feature `X` tra 0 e 1 e, opzionalmente, mescola gli item (utile ad esempio in modalità online o minibatch, oppure per uno split sempre casuale dei dati – come vedremo).

• Il metodo `split` serve per ricavare, a partire dal training set in input `X_train`, un validation set di taglia `val_size`, aggiornando opportunamente la cardinalità del training set che passerà da N a $N - val\_size$. Costruisce quindi due *partizioni*: `X_train` e `X_val` devono essere disgiunti. Un esempio di funzionamento è dato in Figura:

```
>>> X_train, targets_train
(matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]]), array([0, 1, 2, 3]))
>>> X_val, targets_val, X_train, targets_train = split(X_train, targets_train, val_size=2)
>>> X_val, targets_val
(matrix([[ 0,  1],
        [ 4,  5],
        [ 8,  9],
        [12, 13]]), array([0, 1]))
>>> X_train, targets_train
(matrix([[ 2,  3],
        [ 6,  7],
        [10, 11],
        [14, 15]]), array([2, 3]))
```

3.7 models.py

Il file `models.py` contiene le definizioni dei modelli di machine learning, nel nostro caso una rete neurale di tipo fully connected. Illustriamo dapprima la classe `Layer` ed in seguito la classe `NeuralNetwork`.

```
from activation_functions import relu, identity
from loss_functions import cross_entropy
from metrics import accuracy_score
from utils import *

class Layer:

  def __init__(self, neurons, ty=None, activation=None):
    self.neurons = neurons  # Numero di neuroni
    self.ty = ty  # Tipo (input, hidden, output)
    self.activation = activation  # Funzione di attivazione
    self.out = None  # Matrice degli output
    self.weights = None  # Matrice dei pesi
    self.bias = None  # Vettore colonna dei bias
    self.w_sum = None  # Matrice delle somme pesate
    self.dact = None  # Derivata della funz. di attivazione
    self.dE_dW = None  # Derivata dell'errore risp. a W
    self.dE_db = None  # Derivata dell'errore risp. al bias
    self.deltas = None  # Per la backprop
    self.diff_w = None  # Per il momentum (MGD)
    self.diff_b = None  # Per il momentum (MGD)


  # Costruisce il layer impostando la funzione di attivazione
  # Ed inizializzando pesi, bias e altri dati che servono per il MGD
  def build(self, prev_layer_neurons):
    # Imposta la funzione di attivazione
    self.set_activation()

    # Matrice di pesi campionati da una distribuzione uniforme
    self.weights = np.asmatrix(
          np.random.uniform(-0.02, 0.02,
          size=(self.neurons, prev_layer_neurons))
    )

    # Vettore bias campionato da una distribuzione uniforme
    self.bias = np.asmatrix(
          np.random.uniform(-0.02, 0.02, self.neurons)
    ).T
```

```
    # Per la MGD - inizializzata a zero
    self.diff_w = np.asmatrix(
          np.zeros(shape=np.shape(self.weights))
    )

    # Per la MGD - inizializzata a zero
    self.diff_b = np.asmatrix(
          np.zeros(shape=np.shape(self.bias))
    )


# Imposta, se non specificata, la funzione di attivazione
# Tiene in considerazione il teorema di approx. universale
def set_activation(self):
    if self.activation is None:
        if self.ty == "hidden":
            self.activation = relu  # Funzione non polinomiale ma der.
        elif self.ty == "output":
            self.activation = identity  # Una trasformazione lineare


# Singolo passo della propagazione in avanti
# Se il layer corrente e' di input, fa il mirroring dell'input
# Altrimenti calcola la somma pesata e applica la f.att.
def forward_prop_step(self, z):
    if self.ty == "input":
        self.out = z
    else:
        self.w_sum = np.dot(self.weights, z) + self.bias
        self.out = self.activation(self.w_sum)
    return self.out


# Singolo passo della backprop
# Calcola le derivate della funzione di costo risp. ai pesi/bias
def back_prop_step(self, next_layer, prev_layer, target, loss):
    if self.ty == "output":
        # BP1 - calcolo i delta dei nodi di output
        self.dact = self.activation(self.w_sum, derivative=True)
        self.deltas = np.multiply(self.dact, loss(self.out, target,
                                  derivative=True))
    else:
        # BP2 - calcolo i delta dei nodi interni
        self.dact = self.activation(self.w_sum, derivative=True)
        self.deltas = np.multiply(self.dact, np.dot(next_layer.weights
                                  .T, next_layer.deltas))
```

```
    # Calcolo delle derivate risp ai pesi
    self.dE_dW = self.deltas * prev_layer.out.T

    # Calcolo delle derivate risp ai bias
    # dE/db = dE/da * da/db = dE/da * 1 = dE/da = delta
    self.dE_db = np.sum(self.deltas, axis=1)


  # MGD - Momentum Gradient Descent
  # Applica la learning rule ai pesi
  # Se momentum = 0, coincide con Gradient Descent
  def update_weights(self, l_rate, momentum):
    self.diff_w = l_rate * self.dE_dW + momentum * self.diff_w
    self.weights = self.weights - self.diff_w


  # MGD - Momentum Gradient Descent
  # Applica la learning rule ai bias
  # Se momentum = 0, coincide con Gradient Descent
  def update_bias(self, l_rate, momentum):
    self.diff_b = l_rate * self.dE_db + momentum * self.diff_b
    self.bias = self.bias - self.diff_b
```

Come si può leggere dai commenti, questo brano di codice implementa la classe `Layer`. Tutti i metodi relativi ad un'istanza ricevono tacitamente da Python il riferimento corrente (`self`)

• Il metodo `__init__` corrisponde al costruttore della classe e riceve in ingresso:

1. `neurons` - il numero di neuroni nel layer
2. `ty` - il tipo tra `input`, `hidden` o `output`. Se non specificato (`None`), viene stabilito automaticamente dall'algoritmo
3. `activation` - la funzione di attivazione comune a tutti i neuroni di quel layer. Se non specificata (`None`) viene scelta dall'algoritmo tenendo in considerazione il teorema di approssimazione universale

• Il metodo `build` costruisce l'istanza di layer scegliendo una funzione di attivazione ed inizializzando, tramite un campionamento uniforme $U[-0.02, 0.02]$, la matrice dei pesi e il vettore colonna bias, mentre inizializza a zero la matrice delle differenze di pesi e quelle dei bias, che servono per implementare il MGD. Riceve in ingresso:

1. `prev_layer_neurons` - numero di neuroni nello strato precedente

• Il metodo `set_activation` imposta le funzioni di attivazione basandosi sui criteri del teorema di approssimazione universale, scegliendo per i neuroni interni una funzione non polinomiale ma derivabile (es. sigmoide, ReLU...) e per i neuroni di uscita una trasformazione lineare (es. identità).

• Il metodo `forward_prop_step` serve per implementare il singolo passo di propagazione in avanti facendo il mirroring del dato in ingresso `x` se il layer è di input, altrimenti calcolando le somme pesate e applicandovi la funzione di attivazione. Riceve in ingresso:

1. `z` - dato in input in base al quale calcolare gli output

• Il metodo `back_prop_step` serve per implementare il singolo passo della propagazione all'indietro e calcola le derivate della funzione di errore rispetto ai pesi/bias. In base al fatto che il layer sia di output oppure interno (quello di input non partecipa alla backprop e viene scartato da un metodo della classe `NeuralNetwork`), vengono calcolati i `deltas` di ciascun nodo e modellati, al solito, sotto forma di matrici. In particolare la backpropagation si basa su un calcolo *locale* tra due nodi $i \neq j$ facendo in modo tale che la derivata possa calcolarsi in tempo lineare sul numero di parametri come $\frac{\partial E^n}{\partial w_{ij}} = \delta_i^n \cdot z_j^n$ dove:

$$\delta_i^n = \begin{cases} g_k'(a_k^n) \cdot \frac{\partial E^n}{\partial y_k^n} & \text{se } i = k \text{ è un nodo di output} \\ f'(a_i^n) \cdot \sum_j (w_{ji} \cdot \delta_j^n) & \text{se } i \text{ è un nodo interno} \end{cases}$$

se f denota la funzione di attivazione per i nodi interni e g quella per i nodi di output. Si noti che l'algoritmo *non aggiorna i pesi della rete* ma si limita *al calcolo del gradiente*, questo perché l'aggiornamento dei pesi verrà fatto da un'altra funzione (nel nostro caso dalla discesa del gradiente con momento). Riceve in ingresso:

1. `next_layer` - riferimento al layer "sulla destra" di quello corrente
2. `prev_layer` - riferimento al layer "sulla sinistra" di quello corrente
3. `targets` - etichette true
4. `loss` - funzione di costo (nel caso di MNIST sarà la cross-entropy, essendo un task di classificazione)

• Il metodo `update_weights` implementa la learning rule del modello ovvero la discesa del gradiente con momento data da $\Delta w_{ij}^t = -\eta \frac{\partial E^t}{\partial w_{ij}} + \mu \cdot \Delta w_{ij}^{t-1}$. Calcola dapprima un tasso di variazione per i pesi, chiamato `diff_w`, che è una variabile d'istanza, e poi aggiorna i pesi. Riceve in ingresso:

1. `l_rate` - tasso di apprendimento $\eta \in (0; 1)$
2. `momentum` - coefficiente del momento $\mu \in [0; 1)$. Rappresenta un fattore "di inerzia" in grado di accelerare l'apprendimento per zone piatte della funzione di costo. Se è 0, MGD si comporta come GD standard.

• Il metodo `update_bias` è l'analogo al metodo `update_weights` ma per i bias.

Nello stesso file è stata definita anche la classe `NeuralNetwork`:

```
class NeuralNetwork:

  def __init__(self):
    self.layers = []


  # Aggiunge un layer alla rete
  def add(self, layer):
    self.layers.append(layer)


  # Costruisce la rete impostando il tipo di ciascuno strato
  def compile(self):
    for i, layer in enumerate(self.layers):
      if i == 0:
        layer.ty = "input"
      else:
        layer.ty = "output" if i == (len(self.layers) - 1)
                            else "hidden"
        layer.build(self.layers[i - 1].neurons)


  # Stampa la rete in stdout
  def summary(self):
    for i, layer in enumerate(self.layers):
      print("Layer", i)
      print("neurons:", layer.neurons)
      print("type:", layer.ty)
      print("act:", layer.activation)
      print("weights:", np.shape(layer.weights))
      print("bias:", np.shape(layer.bias))


  # Fase di learning della rete
```

```
def fit(self, X_train, targets_train, X_val, targets_val,
                                  max_epochs, l_rate, momentum):

  # Liste per conservare gli errori su train e val ad ogni epoca
  train_losses, val_losses = [], []

  # Liste per conservare l'acc su train e val ad ogni epoca
  train_accuracies, val_accuracies = [], []

  predictions_val = self.predict(X_val)
  min_val_loss = cross_entropy(predictions_val, targets_val)
  best_model = self  # Rete che minimizza l'errore sul VS
  best_epoch = 0  # Epoca a cui corrisponde l'errore sul VS minore

  # Batch mode
  for epoch in range(max_epochs):
    # Forward prop
    predictions_train = self.predict(X_train)

    # Calcolo l'accuratezza delle predizioni sul training set
    train_acc = accuracy_score(targets_train, predictions_train)
    train_accuracies.append(train_acc)

    # Calcolo il gradiente della funzione di costo risp. pesi/bias
    self.back_prop(targets_train, cross_entropy)

    # Aggiorno i pesi
    self.learning_rule(l_rate, momentum)

    # Calcolo l'errore sul training set all'epoca e
    train_loss = cross_entropy(predictions_train, targets_train)
    train_losses.append(train_loss)

    # Fase di model selection
    predictions_val = self.predict(X_val)

    # Calcolo l'accuratezza delle predizioni sul validation set
    val_acc = accuracy_score(targets_val, predictions_val)
    val_accuracies.append(val_acc)

    # Calcolo l'errore sul validation set all'epoca e
    val_loss = cross_entropy(predictions_val, targets_val)
    val_losses.append(val_loss)

    # Scelgo la rete che minimizza l'errore sul VS
    # E aggiorno opportunamente le variabili
```

```
      if val_loss < min_val_loss:
        min_val_loss = val_loss
        best_epoch = epoch
        best_model = self

      # Stampo in stdout i risultati relativi all'epoca e
      print(f"E({epoch}): "
      f"train_loss: {train_loss :.14f} "
      f"val_loss: {val_loss :.14f} "
      f"train_acc: {train_acc * 100 :.2f} % "
      f"val_acc: {val_acc * 100 :.2f} %")

    # Alla fine:
    # Stampo l'epoca in cui il validation set e' minimo
    print(f"Validation loss is minimum at epoch: {best_epoch}")

    # Disegno le funzioni di errore sul training e sul validation
    plot_losses(np.arange(max_epochs), train_losses, val_losses)

    # Disegno le accuratezze sul training e sul validation
    plot_accuracies(np.arange(max_epochs), train_accuracies,
                                    val_accuracies)

    # Restituisco la rete che minimizza l'errore sul validation set
    return best_model


# Effettua le predizioni delle classi dei campioni in X
def predict(self, X):
    for layer in self.layers:
      X = layer.forward_prop_step(X)
    return X


# Calcola il gradiente della funzione di costo risp. pesi/bias
# Il layer di input non viene coinvolto
def back_prop(self, target, loss):
    for i, layer in enumerate(self.layers[:0:-1]):
      next_layer = self.layers[-i]
      prev_layer = self.layers[-i - 2]
      layer.back_prop_step(next_layer, prev_layer, target, loss)


# Applica la regola di learning aggiornando pesi/bias
# Il layer di input non viene coinvolto
def learning_rule(self, l_rate, momentum):
```

```
    for layer in [lyr for lyr in self.layers if lyr.ty != "input"]:
      layer.update_weights(l_rate, momentum)
      layer.update_bias(l_rate, momentum)
```

• Il metodo `__init__` corrisponde al costruttore della classe.

• Il metodo `add` aggiungere un layer alla lista di layer della rete. Riceve come parametro:

1. `layer` - riferimento ad un'istanza di layer da aggiungere

• Il metodo `summary` stampa in `stdout` i parametri della rete.

• Il metodo `fit` corrisponde alla fase di learning e ha il compito di individuare i parametri θ della rete che minimizzano l'errore sul validation set, ovvero i parametri θ per i quali si approssimano al meglio le probabilità condizionate $\mathbb{P}(C_k|\mathbf{x})$ per ogni $1 \leq k \leq c = 10$. È stata scelta come modalità di learning la batch-mode in cui l'aggiornamento dei pesi/bias avviene dopo che tutti i dati sono stati processati: i parametri non vengono aggiornati ad ogni item ma ad ogni `batch_size` che, nel caso dell'offline learning, corrisponde alla cardinalità dell'intero dataset. Lo pseudocodice è il seguente:

Algorithm 1: Batch learning

Result: $\theta^* = \text{argmin}_{\theta \in \Theta} E_{VS}(\theta)$
foreach *e in epochs* **do**
  foreach *sample in training set* **do**
    forward propagation
    backpropagation
  end
  $\theta \leftarrow$ aggiorno i pesi/bias ; `// tramite qualche learning rule`
  E_TrS(e) $\leftarrow$ calcola l'errore sul training set all'epoca e
  E_VS(e) $\leftarrow$ calcola l'errore sul validation set all'epoca e
end
return θ *tale che E_VS(*θ*) è il minimo*

Di norma la convergenza nella modalità batch è più lenta che nella modalità online, in cui i pesi/bias vengono aggiornati per ciascun elemento. Un ottimo compromesso è la modalità mini-batch. Tuttavia, poiché nel calcolo del gradiente verrà restituito un vettore la cui norma è N volte più piccola rispetto a quella vera, la scelta del

`l_rate` è meno sensibile rispetto al fatto che si scelga l'una o l'altra modalità di learning. Quindi, dopo aver calcolato gli errori su training e validation set all'epoca `epoch` segue una fase di model selection in cui si verifica se l'errore *sul validation set* all'epoca corrente è minore rispetto al più piccolo errore *sul validation set* trovato fino a quell'istante. In tal caso, la variabile viene aggiornata e scelgo come `best_model` l'istanza corrente della rete. Questo processo è chiaramente iterativo e viene ripetuto `max_epochs` volte. Vengono inoltre stampati in `stdout` gli errori e l'accuratezza all'epoca corrente. Le sintassi `:.14f` e `:.2f` servono rispettivamente per stampare le prime 14 cifre decimali e le prime 2. Alla fine dei ciclo più esterno – ovvero quello che itera `max_epochs` volte – stampo in `stdout` l'epoca a cui corrisponde il validation loss più piccolo, ovvero la variabile `best_epoch`. Inoltre stampo i grafici degli errori passandogli come parametri il numero di epoche sotto forma di intervallo, `np.arange(max_epochs)`, la lista di errori sul training set `train_losses` e quella sul validation set `val_losses`. Opzionamente è possibile passare al metodo `utils.plot_accuracies`, come visto nel Paragrafo 3.5, un parametro booleano `show_best_net` che, se `True`, disegna un pallino arancione in corrispondenza del minimo della funzione di errore sul validation set, rispetto all'epoca.

• Il metodo `predict` restituisce una matrice di predizioni per gli elementi nel dataset in input. Riceve in ingresso:

1. `X` - un dataset di cui predire le etichette

• Il metodo `back_prop` calcola il gradiente della funzione di costo rispetto ai pesi/-bias. Coinvolge esclusivamente gli strati nascosti e quello di output sfruttando la *list slicing* di Python per bypassare lo strato di input. Partendo dallo strato di output, memorizza un riferimento al layer sulla sua "destra" e quello che si trova sulla sua "sinistra". Riceve in ingresso:

1. `targets` - i true target
2. `loss` - una funzione di costo (cross-entropy per classificazione)

• Il metodo `learning_rule` aggiorna i pesi/bias della rete in base ad un learning rate e in base al coefficiente del momento, applicando quindi la discesa del gradiente con momento come regola di aggiornamento. Riceve in ingresso:

1. `l_rate` - tasso di apprendimento $\eta \in (0; 1)$
2. `momentum` - coefficiente del momento, $\mu \in [0; 1)$. Se $\mu = 0$, il momento non incide sulla learning rule che si comporterà come il gradient descent standard (anche detto *vanilla*)

3.8 main.py

Il file `main.py` contiene la logica principale del progetto.

```
import numpy as np
from mnist.loader import MNIST
from data_processing import normalize, split, one_hot
from metrics import accuracy_score
from models import NeuralNetwork, Layer
from utils import balanced

if __name__ == '__main__':
  # Recupero il dataset di MNIST
  mndata = MNIST(path="data", return_type="numpy")

  # 60.000 immagini, 28x28 feature
  X_train, targets_train = mndata.load_training()

  # 10.000 immagini, 28x28 feature
  X_test, targets_test = mndata.load_testing()

  # Trasformazione delle feature
  X_train, targets_train = normalize(X_train, targets_train,
                                     shuffle=True)
  # Trasformazione delle feature
  X_test, targets_test = normalize(X_test, targets_test,
                                     shuffle=False)
  # Trasposizione
  X_train = X_train.T
  X_test = X_test.T

  # Partizionamento del training set per estrarre il validaiton set
  # Dopo lo split il training set conterra' 50.000 coppie
  # Mentre il validation set conterra' le rimanenti 10.000
  X_val, targets_val, X_train, targets_train = split(X_train,
                                    targets_train, val_size=10000)

  # Verifico che i tre insiemi contengano classi in modo bilanciato
  # Se il bilanciamento e' assicurato, allora lo split e' idoneo
  # E potra' essere usata l'accuracy come unica metrica
  # Per valutare il modello sul test set
  # Altrimenti occorre impiegare altre metriche aggiuntive
  # Come precision, recall e media armonica F1
  if not balanced(targets_train, targets_val, targets_test):
    raise Exception("Classes are not balanced")
```

```
# Codifica one-hot dei target
targets_train = one_hot(targets_train)
targets_val = one_hot(targets_val)
targets_test = one_hot(targets_test)

# Creazione della rete
net = NeuralNetwork()
d = np.shape(X_train)[0]  # numero di feature, 28x28
c = np.shape(targets_train)[0]  # numero di classi, 10

# Rete shallow
# Il primo layer e' di input e contiene d=784 neuroni
# Il secondo e' hidden e contiene 100 neuroni
# Il terzo e' di output e contiene c=10 neuroni
for neurons in (d, 100, c):
  net.add(Layer(neurons))

# Costruzione della rete
net.compile()

# Fase di learning della rete
# Al termine ottengo il modello che minimizza l'errore sul VS
best_net = net.fit(X_train, targets_train, X_val, targets_val,
                   max_epochs=50, l_rate=0.000005, momentum=0.9)

# Testing del modello ottenuto
# Se garantito il bilanciamento,
# Allora ci si puo' limitare a considerare l'accuratezza
# Come unica metrica sui dati del test set
predictions_test = best_net.predict(X_test)
test_acc = accuracy_score(targets_test, predictions_test)
print(f"Accuracy score on test set is: {test_acc * 100 :.2f} %")
```

Innanzitutto occorre scaricare i quattro file di `MNIST` ed inserirli in una cartella all'interno del progetto, io l'ho chiamata `data`. Poi viene usata la classe `mnist.loader.MNIST` per ricavare i dati del training e del test set. Il primo contiene, come da documentazione, 60.000 coppie mentre il secondo ne contiene 10.000. Segue poi una fase di normalizzazione dei dati per contenerli nell'intervallo $[0; 1]$ e renderli idonei alla computazione della rete neurale ed una fase di trasposizione per i motivi già discussi nel Capitolo 2. Opzionalmente è possibile effettuare uno `shuffle` per mescolare i campioni all'interno di ciascun dataset. Non è strettamente necessario per la modalità batch perché la rete riceve in ingresso l'intero dataset e non computa il gradiente sulla singola coppia $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$. È possibile comunque fare uno shuffle dei dati

di training per avere una maggiore aleatorietà durante lo split per generare i dati del validation. Lo `split`, infatti, viene usato per ricavare *a partire dal training set* un insieme *casuale*[1] di `val_size` coppie da inserire nel validation set. Dopo lo `split`, infatti, il training conterrà `60,000 - val_size = 50,000` coppie mentre il validation set ne conterrà `val_size = 10,000`. Come già detto, è fondamentale che i due insiemi siano disgiunti. Con il metodo `balanced` controllo se, a seguito dello `split`, i tre insiemi contengono in modo bilanciato ciascuna delle 10 classi. Questo serve per garantire che lo split è stato idoneo, nel senso che ogni dataset contenga campioni significativi al problema, e per limitare il testing alla sola accuratezza[2]. Se le classi sono bilanciate, allora proseguo codificando i target in one-hot e istanziando una rete neurale `net` costituita da $d = 28 \times 28 = 784$ neuroni in ingresso, uno per ogni feature, uno strato interno[3] con un numero di neuroni arbitrario ed infine uno strato di output con $c = 10$ classi di output, una per ogni possibile cifra (0-9). Viene poi compilata ed inizializzata la rete tramite il metodo `net.compile` già presentato nel paragrafo precedente. Segue la fase di learning dei dati, in cui alla fine del processo iterativo già illustrato verrà restituita la rete `best_net` che, come visto, minimizza l'errore sul *validation set* e che meglio approssima le probabilità condizionate. Infine vengono testate le prestazioni di `best_net` in termini di accuratezza sui dati di testing.

[1] Se `shuffle=True`

[2] Se le classi non fossero bilanciate, bisognerebbe procedere anche con il calcolo di precision, recall e media armonica $F1$

[3] La traccia due richiede di studiare l'apprendimento di una rete *shallow*, ma il progetto deve essere sufficientemente generale da consentire la costruzione di reti con più di uno strato interno e per qualsiasi funzione di attivazione (parte A)

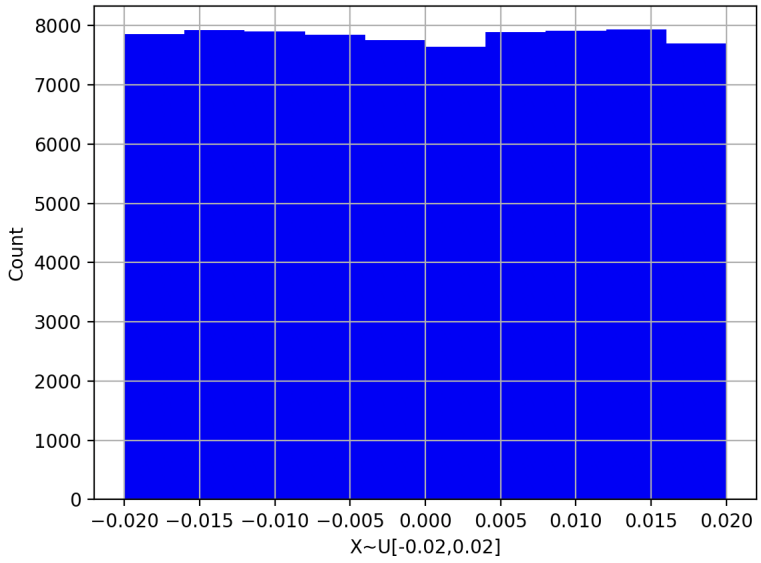
Capitolo 4

Un esempio di costruzione di una rete multistrato

Questo capitolo dimostra che il codice funziona correttamente anche per reti *multistrato*, come richiesto dalla parte A della traccia: supponiamo di voler costruire una rete con tre strati interni dove il primo contiene 100 neuroni ed è attivato dalla ReLU, il secondo ne contiene 200 ed è attivato dalla ReLU ed il terzo ne contiene 300 ed è attivato dalla sigmoide. Stampiamo in `stdout` la struttura della rete invocando il metodo `models.NeuralNetwork.summary`:

```
Layer 0
neurons: 784
type: input
act: None
weights: ()
bias: ()

Layer 1
neurons: 100
type: hidden
act: <function relu at 0x15e7c8540>
weights: (100, 784)
bias: (100, 1)

Layer 2
neurons: 200
type: hidden
act: <function relu at 0x15e7c8540>
weights: (200, 100)
bias: (200, 1)

Layer 3
neurons: 300
type: hidden
act: <function sigmoid at 0x15e7c8400>
weights: (300, 200)
bias: (300, 1)

Layer 4
neurons: 10
type: output
act: <function identity at 0x15e7c84a0>
weights: (10, 300)
bias: (10, 1)
```

I pesi/bias sono variabili aleatorie inizializzati tramite un campionamento da una distribuzione di probabilità uniforme. La seguente Figura mostra un istogramma relativo alla distribuzione di probabilità dei campioni della matrice dei pesi del primo livello W^1 ovvero quelli che vanno dai neuroni di input a quelli del primo strato interno:



Scriveremo che $W^L \sim U[a, b]$ per ogni strato L, dove $a = -0.02$ e $b = 0.02$. Lo stesso per i bias. Il valore atteso è $\mathbb{E}(W^L) = \frac{b+a}{2} = 0$ mentre la varianza è $\sigma^2(W^L) = \frac{(b-a)^2}{12} \simeq 1.3 \times 10^{-4}$. L'inizializzazione dei pesi/bias è piuttosto delicata e tipicamente segue un approccio euristico: nell'ambito delle reti neurali si è notato che la distribuzione uniforme e quella normale funzionano particolarmente bene:

"The normal vs uniform init seem to be rather unclear in fact.

If we refer solely on the Glorot's and He's initializations papers, they both use a similar theoritical analysis: they find a good variance for the distribution from which the initial parameters are drawn. This variance is adapted to the activation function used and is derived without explicitly considering the type of the distribution. As such, their theorical conclusions hold for any type of distribution of the determined variance. In fact, in the Glorot paper, a uniform distribution is used whereas in the He paper it is a gaussian one that is chosen. The only "explaination" given for this choice in the He paper is:

Recent deep CNNs are mostly initialized by random weights drawn from Gaussian distributions with a reference to AlexNet paper. It was indeed released a little later than Glorot's initialization but however there is no justificaion in it of the use of a normal distribution.

In fact, in a discussion on Keras issues tracker, they also seem to be a little confused and basically it could only be a matter of preference... (i.e. hypotetically es would prefer uniform distribution whereas Hinton would prefer normal ones...) One the discussion, there is a small benchmark comparing Glorot initialization using a uniform and a gaussian distribution. In the end, it seems that the uniform wins but it is not really clear."

Le predizioni che questa rete effettua sono codificate nella seguente matrice:

```
targets shape: (10, 50000)
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]
 ...
 [0. 1. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 1.]]
```

```
predictions shape: (10, 50000)
[[ 0.00434948  0.00438473  0.0045575  ...  0.00464914  0.00451593
   0.00440221]
 [ 0.0059173   0.00606518  0.00606225 ...  0.00620783  0.00595139
   0.00583473]
 [-0.00331938 -0.00316514 -0.00327216 ... -0.00345239 -0.00330043
  -0.00326819]
 ...
 [-0.01178243 -0.01184652 -0.01185587 ... -0.01182317 -0.01178048
  -0.01154375]
 [-0.01613273 -0.01606235 -0.0160239  ... -0.01618442 -0.01583515
  -0.01601781]
 [ 0.01460437  0.01459565  0.01459277 ...  0.0145757   0.01447486
   0.01459174]]
```

A questo punto è possibile applicare coerentemente la funzione softmax per trattare gli output come probabilità condizionate, ovvero $y_k^n = \mathbb{P}(C_k|\mathbf{x})$ per ogni $1 \leq k \leq c = 10$.

Capitolo 5

Analisi dei risultati

Stabilito l'approccio *hold-out* come schema di valutazione, l'obiettivo di questo capitolo è studiare l'apprendimento di una rete neurale (in termini di epoche necessarie per il learning, andamento dell'errore su training e validation set, accuratezza sul test set) con un solo strato di neuroni interni al variare del learning rate η e del momento μ per almeno cinque diverse dimensioni dello strato interno. È richiesto di lasciare invariate tutte le altre scelte come, ad esempio, le funzioni di output. A tal proposito ho fissato:

- La `ReLU` come funzione di attivazione dei nodi interni - andava bene anche la `sigmoid`, una qualsiasi funzione non polinomiale ma derivabile
- La `identity` come funzione di attivazione dei nodi di output, andava bene una qualsiasi trasformazione lineare
- `max_epochs = 100`, il numero massimo di epoche per il learning
- `learning_mode = batch`, la modalità di learning (online, batch o mini-batch)

e farò variare il numero di nodi interni `m` $\in \{50, 100, 200, 300, 400\}$, il `l_rate` $\eta \in \{10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}\}$ ed il `momentum` $\mu \in \{0, 0.4, 0.9\}$.

5.1 Modello Alfa - 50 nodi interni

Costruiamo la seguente rete con 50 nodi interni:

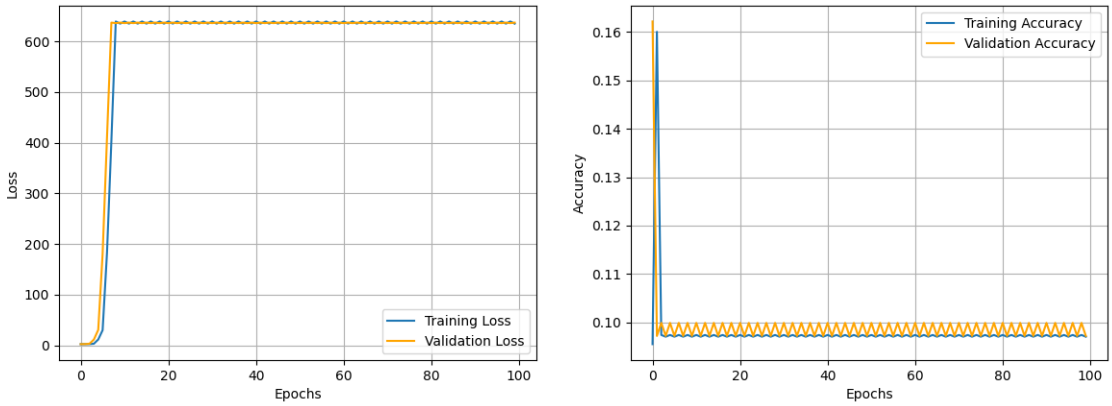
```
Layer 0
neurons: 784
type: input
act: None
weights: ()
bias: ()

Layer 1
neurons: 50
type: hidden
act: <function relu at 0x1500c8540>
weights: (50, 784)
bias: (50, 1)

Layer 2
neurons: 10
type: output
act: <function identity at 0x1500c84a0>
weights: (10, 50)
bias: (10, 1)
```

5.1.1 Versione 1

Vediamo cosa succede quando $\eta = 10^{-4}$ e $\mu = 0$.

```
E(0): train_loss: 2.30296444694919 val_loss: 2.28254619428754 train_acc: 9.55 % val_acc: 16.22 %
E(1): train_loss: 2.28292299402735 val_loss: 2.37813204963768 train_acc: 16.01 % val_acc: 9.72 %
E(2): train_loss: 2.37663302323823 val_loss: 3.24765955234342 train_acc: 9.74 % val_acc: 9.99 %
E(3): train_loss: 3.25364186414259 val_loss: 12.32686281666716 train_acc: 9.70 % val_acc: 9.72 %
E(4): train_loss: 12.30289517376864 val_loss: 40.48433675718626 train_acc: 9.74 % val_acc: 9.99 %
E(5): train_loss: 40.58016006113891 val_loss: 197.25357737410803 train_acc: 9.70 % val_acc: 9.72 %
E(6): train_loss: 196.87023713514523 val_loss: 533.74963862249570 train_acc: 9.74 % val_acc: 9.99 %
E(7): train_loss: 535.19156231559066 val_loss: 640.26233716214756 train_acc: 9.70 % val_acc: 9.72 %
E(8): train_loss: 640.12049792041921 val_loss: 638.34750739881383 train_acc: 9.74 % val_acc: 9.99 %
E(9): train_loss: 640.37580855553040 val_loss: 640.26233716214756 train_acc: 9.70 % val_acc: 9.72 %
E(10): train_loss: 640.12049792041921 val_loss: 638.34750739881383 train_acc: 9.74 % val_acc: 9.99 %
E(11): train_loss: 640.37580855553040 val_loss: 640.26233716214756 train_acc: 9.70 % val_acc: 9.72 %
E(12): train_loss: 640.12049792041921 val_loss: 638.34750739881383 train_acc: 9.74 % val_acc: 9.99 %
E(13): train_loss: 640.37580855553040 val_loss: 640.26233716214756 train_acc: 9.70 % val_acc: 9.72 %
E(14): train_loss: 640.12049792041921 val_loss: 638.34750739881383 train_acc: 9.74 % val_acc: 9.99 %
E(15): train_loss: 640.37580855553040 val_loss: 640.26233716214756 train_acc: 9.70 % val_acc: 9.72 %
```

Figura 5.1: $\eta = 10^{-4}$, $\mu = 0$ (immagine tagliata)




```
Accuracy score on test set is:  10.09 %.
```

Commento

L'andamento a "zig-zag" dell'errore, ed il fatto che esso aumenta anziché diminuire, fanno pensare che `l_rate` sia troppo alto perché ci allontana dalla soluzione ottima, e quindi dal minimo globale della funzione di costo. Anche se aumentassimo

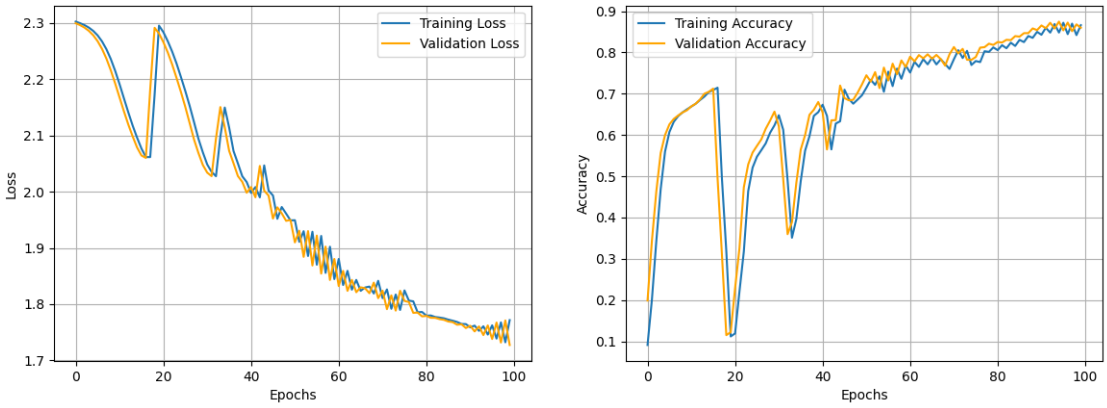
il coefficiente del momento μ, il risultato sarebbe pressocché identico: *oscillazioni divergenti*. Non vale la pena, quindi, aumentare il valore del `momentum` se prima non diminuiamo opportunamente il valore del `l_rate`.

5.1.2 Versione 2

Abbassiamo il `l_rate`, scegliamo quindi $\eta = 10^{-5}$ e lasciamo il `momentum` ancora nullo:

```
E(0): train_loss: 2.30218961431930 val_loss: 2.29926937143342 train_acc: 9.11 % val_acc: 19.89 %
E(1): train_loss: 2.29938553668344 val_loss: 2.29592759731225 train_acc: 20.08 % val_acc: 34.36 %
E(2): train_loss: 2.29600060182561 val_loss: 2.29155601048951 train_acc: 34.21 % val_acc: 46.27 %
E(3): train_loss: 2.29155088245057 val_loss: 2.28575430763033 train_acc: 46.82 % val_acc: 55.60 %
E(4): train_loss: 2.28566065486300 val_loss: 2.27791281091673 train_acc: 55.98 % val_acc: 59.98 %
E(5): train_loss: 2.27771590464797 val_loss: 2.26742445918341 train_acc: 60.71 % val_acc: 62.57 %
E(6): train_loss: 2.26709589768085 val_loss: 2.25373135677277 train_acc: 63.20 % val_acc: 63.89 %
E(7): train_loss: 2.25322803355362 val_loss: 2.23649919888753 train_acc: 64.56 % val_acc: 64.72 %
E(8): train_loss: 2.23576940688237 val_loss: 2.21581037463382 train_acc: 65.47 % val_acc: 65.45 %
E(9): train_loss: 2.21480736008472 val_loss: 2.19238335142095 train_acc: 66.21 % val_acc: 65.98 %
E(10): train_loss: 2.19107708501513 val_loss: 2.16749981914020 train_acc: 66.88 % val_acc: 66.92 %
E(11): train_loss: 2.16590061417702 val_loss: 2.14273865986594 train_acc: 67.56 % val_acc: 67.57 %
E(12): train_loss: 2.14088205123982 val_loss: 2.11930708757306 train_acc: 68.47 % val_acc: 68.61 %
E(13): train_loss: 2.11729562136476 val_loss: 2.09810867101786 train_acc: 69.30 % val_acc: 70.00 %
E(14): train_loss: 2.09594626630102 val_loss: 2.07851724519090 train_acc: 70.40 % val_acc: 70.44 %
E(15): train_loss: 2.07659996358552 val_loss: 2.06428797055009 train_acc: 70.88 % val_acc: 71.22 %
```

Figura 5.2: $\eta = 10^{-5}$, $\mu = 0$ (immagine tagliata)




```
Accuracy score on test set is:  85.73 %.
```

Commento

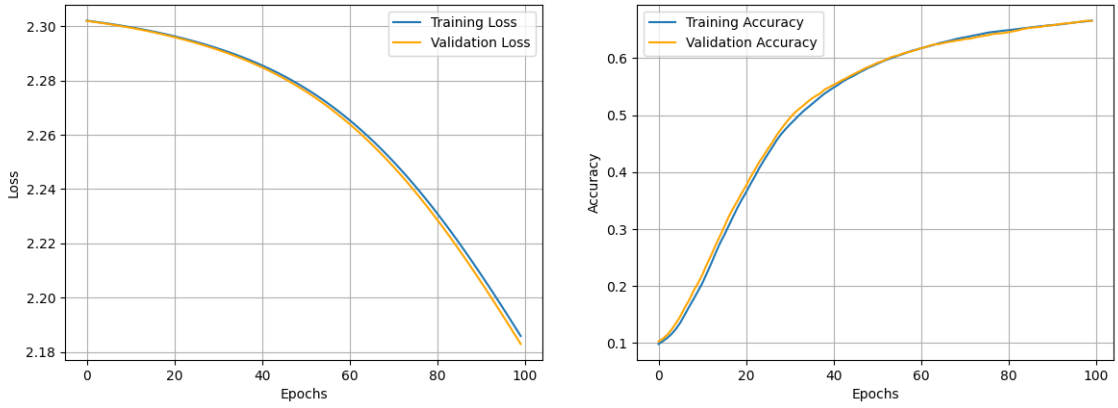
La nostra intuizione è stata corretta: diminuendo di un decimo il `l_rate` ottengo delle prestazioni migliori rispetto a prima, anche se le curve sono ancora piuttosto "zig-zaganti": il learning sembrava procedere bene fino all'epoca 18-esima, prima di avere un salto e ritornare ad un fenomeno a "zig-zag". Dopo 15 epoche l'accuratezza è aumentata di circa 7 volte rispetto alla soluzione precedente ma si può fare di meglio: diminuiamo ancora il `l_rate`.

5.1.3 Versione 3

Fissiamo il `l_rate` $\eta = 10^{-6}$ e lasciamo il `momentum` ancora a zero.

```
E(0): train_loss: 2.30210910716052 val_loss: 2.30199722673386 train_acc: 9.84 % val_acc: 10.31 %
E(1): train_loss: 2.30189408578534 val_loss: 2.30177731131307 train_acc: 10.34 % val_acc: 10.77 %
E(2): train_loss: 2.30167612425863 val_loss: 2.30155198571195 train_acc: 10.93 % val_acc: 11.47 %
E(3): train_loss: 2.30145315068661 val_loss: 2.30131996090633 train_acc: 11.62 % val_acc: 12.39 %
E(4): train_loss: 2.30122379494198 val_loss: 2.30108082986972 train_acc: 12.51 % val_acc: 13.52 %
E(5): train_loss: 2.30098703557662 val_loss: 2.30083398479268 train_acc: 13.62 % val_acc: 14.74 %
E(6): train_loss: 2.30074241477514 val_loss: 2.30057881909554 train_acc: 14.99 % val_acc: 16.25 %
E(7): train_loss: 2.30048966769671 val_loss: 2.30031581533556 train_acc: 16.39 % val_acc: 17.55 %
E(8): train_loss: 2.30022873261372 val_loss: 2.30004424697084 train_acc: 17.72 % val_acc: 19.17 %
E(9): train_loss: 2.29995919031645 val_loss: 2.29976342128320 train_acc: 19.10 % val_acc: 20.48 %
E(10): train_loss: 2.29968088213922 val_loss: 2.29947349167681 train_acc: 20.51 % val_acc: 21.98 %
E(11): train_loss: 2.29939354698312 val_loss: 2.29917401187726 train_acc: 22.19 % val_acc: 23.77 %
E(12): train_loss: 2.29909677780774 val_loss: 2.29886458940660 train_acc: 23.90 % val_acc: 25.36 %
E(13): train_loss: 2.29879006959674 val_loss: 2.29854513749494 train_acc: 25.73 % val_acc: 27.15 %
E(14): train_loss: 2.29847319927916 val_loss: 2.29821532600215 train_acc: 27.46 % val_acc: 28.73 %
E(15): train_loss: 2.29814599165427 val_loss: 2.29787502993583 train_acc: 28.93 % val_acc: 30.38 %
```

Figura 5.3: $\eta = 10^{-6}$, $\mu = 0$ (immagine tagliata)




```
Accuracy score on test set is:  67.73 %.
```

Commento

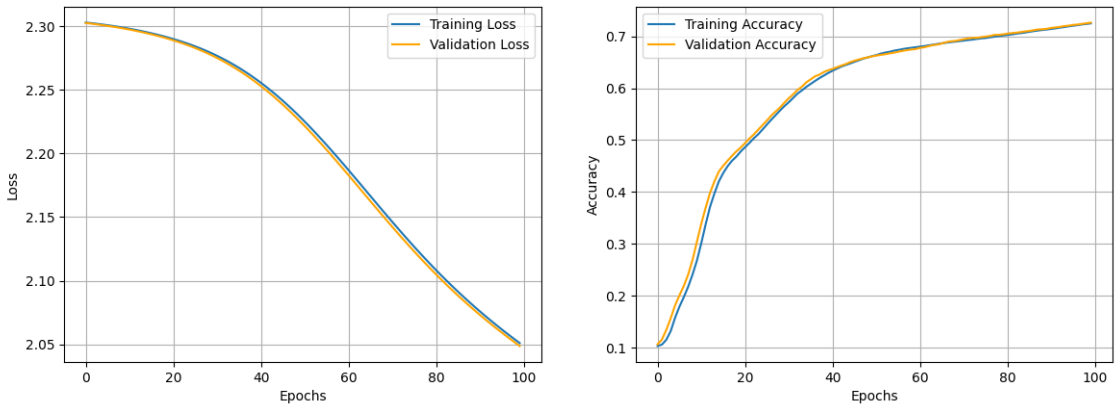
Le curve adesso hanno un andamento più liscio ma, di contro, la convergenza è rallentata: diminuire il `l_rate` per sbarazzarci dei salti è stata una buona idea, ma ha reso più lento il training. Inoltre, avendo scelto $\mu = 0$, la learning rule applicata è essenzialmente la discesa del gradiente *standard* che risulta essere molto sensibile alle zone piatte della funzione di errore, laddove la derivata è approssimativamente nulla rendendo di fatto difficoltoso il learning in termini di aggiornamento dei parametri. Per aumentare l'*inerzia* del movimento lungo lo spazio dei parametri Θ possiamo scegliere un valore del `momentum` $\mu \in (0; 1)$ che sia idoneo ad aumentare il `l_rate` effettivo da η a $\frac{\eta}{1-\mu}$. Per questo motivo il coefficiente del momento può portare ad una convergenza più veloce verso il minimo della funzione di costo.

5.1.4 Versione 4

Lasciamo il `l_rate` η invariato ma scegliamo il `momentum` $\mu = 0.4$.

```
E(0): train_loss: 2.30275306029204 val_loss: 2.30246542111106 train_acc: 10.36 % val_acc: 10.68 %
E(1): train_loss: 2.30247023632307 val_loss: 2.30207085976933 train_acc: 10.63 % val_acc: 11.60 %
E(2): train_loss: 2.30207874348171 val_loss: 2.30163363324704 train_acc: 11.56 % val_acc: 13.46 %
E(3): train_loss: 2.30164439213043 val_loss: 2.30117461149438 train_acc: 13.22 % val_acc: 15.69 %
E(4): train_loss: 2.30118753067330 val_loss: 2.30069621307325 train_acc: 15.78 % val_acc: 18.18 %
E(5): train_loss: 2.30071066614756 val_loss: 2.30019601160249 train_acc: 17.93 % val_acc: 20.11 %
E(6): train_loss: 2.30021133024369 val_loss: 2.29967083339104 train_acc: 19.77 % val_acc: 21.92 %
E(7): train_loss: 2.29968635147111 val_loss: 2.29911858341168 train_acc: 21.78 % val_acc: 24.13 %
E(8): train_loss: 2.29913297746241 val_loss: 2.29853695859988 train_acc: 24.12 % val_acc: 26.95 %
E(9): train_loss: 2.29854965172243 val_loss: 2.29792471909138 train_acc: 26.82 % val_acc: 30.45 %
E(10): train_loss: 2.29793554372568 val_loss: 2.29728062012794 train_acc: 30.17 % val_acc: 33.92 %
E(11): train_loss: 2.29728999763104 val_loss: 2.29660417412638 train_acc: 33.80 % val_acc: 37.01 %
E(12): train_loss: 2.29661201973545 val_loss: 2.29589440837625 train_acc: 37.17 % val_acc: 39.86 %
E(13): train_loss: 2.29590079413956 val_loss: 2.29515048669171 train_acc: 39.74 % val_acc: 42.06 %
E(14): train_loss: 2.29515528481831 val_loss: 2.29437080135550 train_acc: 42.03 % val_acc: 43.94 %
E(15): train_loss: 2.29437395251033 val_loss: 2.29355359033914 train_acc: 43.65 % val_acc: 45.05 %
```

Figura 5.4: $\eta = 10^{-6}$, $\mu = 0.4$ (immagine tagliata)




```
Accuracy score on test set is:  73.87 %.
```

Commento

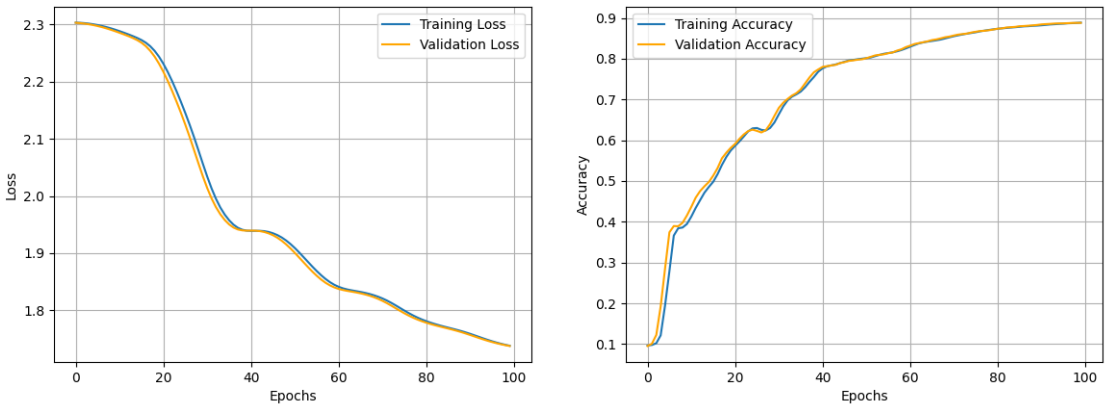
Come si può notare questa versione ha velocizzato, seppur di poco, la convergenza al minimo della funzione di costo. Questo guadagno è apprezzabile soprattutto a partire dalla 40-esima epoca dove le funzioni di errore per la versione precedente risultavano più gonfie e meno ripide rispetto a quelle attuali.

5.1.5 Versione 5

Lasciamo di nuovo il `l_rate` invariato e aumentiamo il `momentum` $\mu = 0.9$

```
E(0): train_loss: 2.30338762029288 val_loss: 2.30313387321518 train_acc: 9.66 % val_acc: 9.54 %
E(1): train_loss: 2.30307393218907 val_loss: 2.30252164129237 train_acc: 9.79 % val_acc: 10.10 %
E(2): train_loss: 2.30247576671174 val_loss: 2.30164072004698 train_acc: 10.23 % val_acc: 12.25 %
E(3): train_loss: 2.30161348287172 val_loss: 2.30049746407515 train_acc: 12.15 % val_acc: 19.26 %
E(4): train_loss: 2.30049542127100 val_loss: 2.29909584773866 train_acc: 19.48 % val_acc: 28.52 %
E(5): train_loss: 2.29912004504485 val_loss: 2.29743037168197 train_acc: 27.91 % val_acc: 37.43 %
E(6): train_loss: 2.29748131651958 val_loss: 2.29549983607326 train_acc: 36.62 % val_acc: 38.96 %
E(7): train_loss: 2.29557995212619 val_loss: 2.29331933411061 train_acc: 38.42 % val_acc: 38.91 %
E(8): train_loss: 2.29342539682218 val_loss: 2.29091556968905 train_acc: 38.60 % val_acc: 39.75 %
E(9): train_loss: 2.29104190302710 val_loss: 2.28833002857390 train_acc: 39.47 % val_acc: 41.55 %
E(10): train_loss: 2.28846543422661 val_loss: 2.28560673130336 train_acc: 41.27 % val_acc: 43.66 %
E(11): train_loss: 2.28573774628652 val_loss: 2.28277288996120 train_acc: 43.40 % val_acc: 45.92 %
E(12): train_loss: 2.28288577704501 val_loss: 2.27980609613269 train_acc: 45.28 % val_acc: 47.52 %
E(13): train_loss: 2.27988911727728 val_loss: 2.27660610908563 train_acc: 47.14 % val_acc: 48.70 %
E(14): train_loss: 2.27664692397352 val_loss: 2.27290461787335 train_acc: 48.54 % val_acc: 49.75 %
E(15): train_loss: 2.27290867362999 val_loss: 2.26826692111684 train_acc: 49.81 % val_acc: 51.31 %
```

Figura 5.5: $\eta = 10^{-6}$, $\mu = 0.9$ (immagine tagliata)




```
Accuracy score on test set is:  89.55 %.
```

Commento

Con questo tuning degli iperparametri abbiamo ottenuto un buon risultato: a parità di epoche, ad esempio alla 15-esima, si ottiene in questo modo un'accuratezza già

del 45%, le funzioni di errori sono ancora più liscie e ripide e l'accuratezza sui tre dataset è buona. Intorno alla 40-esima epoca c'è un piccolo *plateau* in cui l'apprendimento, intuitivamente, "si interrompe". Il modello ha richiesto circa 30 secondi per l'apprendimento, anche se questo dipende molto dalla macchina su cui si fanno gli esperimenti.

5.2 Modello Bravo - 100 nodi interni

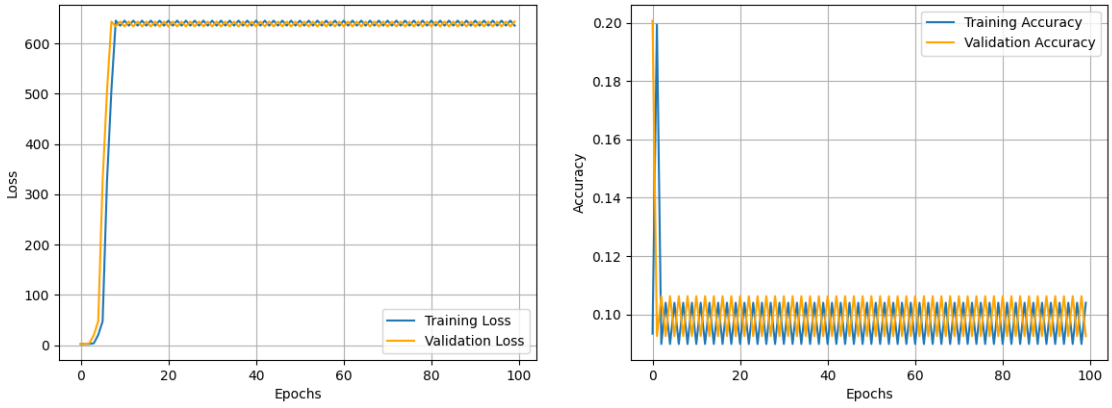
Vediamo cosa succede quando aumentiamo il numero di neuroni interni fino a 100. Facciamo nuovamente variare il `l_rate` η ed il `momentum` μ.

5.2.1 Versione 1

Vediamo cosa succede quando $\eta = 10^{-4}$ e $\mu = 0$.

```
E(0): train_loss: 2.30253858242997 val_loss: 2.26123466744311 train_acc: 9.34 % val_acc: 20.07 %
E(1): train_loss: 2.26062271427970 val_loss: 2.50997542089419 train_acc: 19.93 % val_acc: 9.26 %
E(2): train_loss: 2.51146989539055 val_loss: 3.80074632050282 train_acc: 8.99 % val_acc: 10.63 %
E(3): train_loss: 3.79418236272990 val_loss: 20.36848104649554 train_acc: 10.40 % val_acc: 9.26 %
E(4): train_loss: 20.39418903434424 val_loss: 47.46070514912486 train_acc: 8.99 % val_acc: 10.63 %
E(5): train_loss: 47.35732587522237 val_loss: 326.04774674651992 train_acc: 10.40 % val_acc: 9.26 %
E(6): train_loss: 326.46071752356221 val_loss: 505.67343533908314 train_acc: 8.99 % val_acc: 10.63 %
E(7): train_loss: 506.29183096422463 val_loss: 643.52463972190162 train_acc: 10.40 % val_acc: 9.26 %
E(8): train_loss: 645.43946948523546 val_loss: 633.80865166350395 train_acc: 8.99 % val_acc: 10.63 %
E(9): train_loss: 635.41143509503513 val_loss: 643.52463972190162 train_acc: 10.40 % val_acc: 9.26 %
E(10): train_loss: 645.43946948523546 val_loss: 633.80865166350395 train_acc: 8.99 % val_acc: 10.63 %
E(11): train_loss: 635.41143509503513 val_loss: 643.52463972190162 train_acc: 10.40 % val_acc: 9.26 %
E(12): train_loss: 645.43946948523546 val_loss: 633.80865166350395 train_acc: 8.99 % val_acc: 10.63 %
E(13): train_loss: 635.41143509503513 val_loss: 643.52463972190162 train_acc: 10.40 % val_acc: 9.26 %
E(14): train_loss: 645.43946948523546 val_loss: 633.80865166350395 train_acc: 8.99 % val_acc: 10.63 %
E(15): train_loss: 635.41143509503513 val_loss: 643.52463972190162 train_acc: 10.40 % val_acc: 9.26 %
```

Figura 5.6: $\eta = 10^{-4}$, $\mu = 0$ (immagine tagliata)




```
Accuracy score on test set is:  8.92 %.
```

Commento

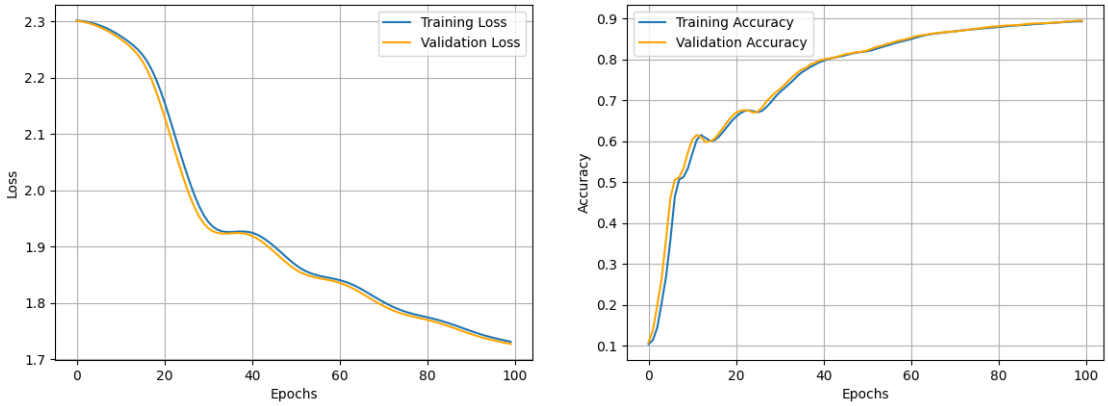
Il risultato che abbiamo ottenuto è molto simile a quando, nella rete composta da 50 neuroni interni, avevamo scelto la stessa configurazione di `l_rate` η: questo esperimento suggerisce che, indipendentemente dal numero di neuroni interni, un `l_rate` così alto non potrà mai portare a prestazioni accettabili perché l'aggiornamento porterà da un punto all'altro della funzione di costo senza mai convergere al minimo globale. Per questo motivo nei prossimi esperimenti non sceglieremo più un `l_rate` così alto.

5.2.2 Versione 2

Nell'esperimento con 50 neuroni avevamo ottenuto prestazioni molto buone quando il `l_rate` $\eta = 10^{-6}$ ed il `momentum` $\mu = 0.9$. Vediamo quali risultati osserviamo quando scegliamo questa specifica configurazione di iperparametri, ma con 100 neuroni interni.

```
E(0): train_loss: 2.30176729476456 val_loss: 2.30140232056951 train_acc: 10.46 % val_acc: 10.86 %
E(1): train_loss: 2.30118439583176 val_loss: 2.30028511370277 train_acc: 11.43 % val_acc: 13.84 %
E(2): train_loss: 2.30008297032769 val_loss: 2.29869212314318 train_acc: 14.50 % val_acc: 19.91 %
E(3): train_loss: 2.29851186569371 val_loss: 2.29664574713207 train_acc: 20.53 % val_acc: 26.55 %
E(4): train_loss: 2.29649133773134 val_loss: 2.29413456823630 train_acc: 26.91 % val_acc: 36.06 %
E(5): train_loss: 2.29401227754584 val_loss: 2.29113284789468 train_acc: 35.96 % val_acc: 46.15 %
E(6): train_loss: 2.29104993615761 val_loss: 2.28762014695486 train_acc: 46.44 % val_acc: 50.59 %
E(7): train_loss: 2.28758754069897 val_loss: 2.28360123150212 train_acc: 50.67 % val_acc: 51.20 %
E(8): train_loss: 2.28362983387786 val_loss: 2.27910536766409 train_acc: 51.28 % val_acc: 53.39 %
E(9): train_loss: 2.27920417442881 val_loss: 2.27416904858392 train_acc: 53.37 % val_acc: 57.29 %
E(10): train_loss: 2.27434780112173 val_loss: 2.26882284731126 train_acc: 56.98 % val_acc: 60.40 %
E(11): train_loss: 2.26908858151248 val_loss: 2.26307050818615 train_acc: 60.31 % val_acc: 61.50 %
E(12): train_loss: 2.26342646343584 val_loss: 2.25679214528955 train_acc: 61.50 % val_acc: 61.03 %
E(13): train_loss: 2.25723562824155 val_loss: 2.24952306798493 train_acc: 60.80 % val_acc: 59.78 %
E(14): train_loss: 2.25005336249648 val_loss: 2.24059764578981 train_acc: 60.11 % val_acc: 59.98 %
E(15): train_loss: 2.24122239952681 val_loss: 2.22937570332258 train_acc: 60.19 % val_acc: 60.47 %
```

Figura 5.7: $\eta = 10^{-6}$, $\mu = 0.9$ (immagine tagliata)




```
Accuracy score on test set is:  90.23 %.
```

Commento

Abbiamo ottenuto delle ottime prestazioni: dalla 60-esima epoca in poi gli errori sul training set e sul validation set sono decrescenti (questo suggerisce che, aumentando ulteriormente `max_epochs`, si potrebbero ottenere risultati ancora migliori) e l'accuratezza sul test set è maggiore del 90%. Di conseguenza il tasso di errore è veramente basso rendendo questo modello un ottimo candidato ottimale. Il modello ha richiesto circa 35 secondi per l'apprendimento.

5.3 Modello Charlie - 200 nodi interni

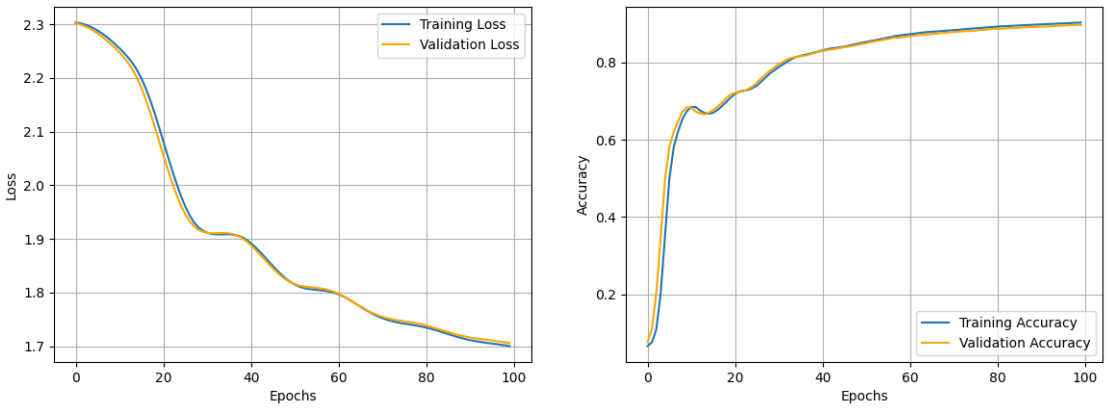
Aumentiamo la complessità del modello e introduciamo 200 nodi interni.

5.3.1 Versione 1

Valutiamo cosa succede, ancora, quando il `l_rate` $\eta = 10^{-6}$ ed il `momentum` $\mu = 0.9$

```
E(0): train_loss: 2.30288974891621 val_loss: 2.30172661748162 train_acc: 6.68 % val_acc: 7.65 %
E(1): train_loss: 2.30175672391826 val_loss: 2.29959791765241 train_acc: 7.61 % val_acc: 11.20 %
E(2): train_loss: 2.29961041981980 val_loss: 2.29657045379114 train_acc: 10.96 % val_acc: 20.27 %
E(3): train_loss: 2.29656264096734 val_loss: 2.29273295313085 train_acc: 20.24 % val_acc: 34.80 %
E(4): train_loss: 2.29270269488327 val_loss: 2.28813604157404 train_acc: 35.27 % val_acc: 49.80 %
E(5): train_loss: 2.28808326942956 val_loss: 2.28280435776044 train_acc: 50.11 % val_acc: 58.27 %
E(6): train_loss: 2.28272633248960 val_loss: 2.27678067420353 train_acc: 58.13 % val_acc: 62.07 %
E(7): train_loss: 2.27665908605223 val_loss: 2.27013506605136 train_acc: 62.25 % val_acc: 64.82 %
E(8): train_loss: 2.26994512580989 val_loss: 2.26294462269579 train_acc: 65.39 % val_acc: 67.30 %
E(9): train_loss: 2.26266310078808 val_loss: 2.25527379619448 train_acc: 67.39 % val_acc: 68.38 %
E(10): train_loss: 2.25488224557939 val_loss: 2.24710546185691 train_acc: 68.47 % val_acc: 68.29 %
E(11): train_loss: 2.24659552491571 val_loss: 2.23813112851646 train_acc: 68.41 % val_acc: 67.26 %
E(12): train_loss: 2.23751414294477 val_loss: 2.22770407967745 train_acc: 67.55 % val_acc: 66.83 %
E(13): train_loss: 2.22698719340891 val_loss: 2.21507468510054 train_acc: 66.94 % val_acc: 66.52 %
E(14): train_loss: 2.21426487739579 val_loss: 2.19963307275703 train_acc: 66.69 % val_acc: 66.99 %
E(15): train_loss: 2.19874394597910 val_loss: 2.18107702712392 train_acc: 66.97 % val_acc: 67.65 %
```

Figura 5.8: $\eta = 10^{-6}$, $\mu = 0.9$ (immagine tagliata)




```
Accuracy score on test set is:  91.09 %.
```

Commento

Abbiamo ottenuto un modello ancora più espressivo dei precedenti, seppur di poco. Il modello ha richiesto circa 55 secondi per l'apprendimento.

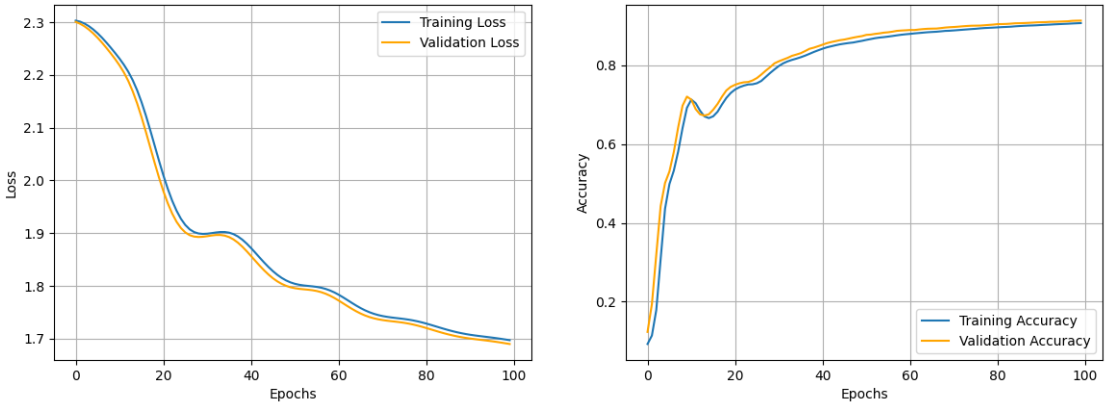
5.4 Modello Delta - 350 nodi interni

Aumentiamo il numero di neuroni interni a 350 per gli stessi valori di `l_rate` e `momentum`, che sembrano essere quelli ottimali: un giusto compresso tra velocità di apprendimento e prestazioni.

5.4.1 Versione 1

```
E(0): train_loss: 2.30326635221349 val_loss: 2.30085210696308 train_acc: 9.24 % val_acc: 12.31 %
E(1): train_loss: 2.30132561773192 val_loss: 2.29732960697530 train_acc: 11.39 % val_acc: 19.38 %
E(2): train_loss: 2.29772674474419 val_loss: 2.29242187719690 train_acc: 17.87 % val_acc: 31.97 %
E(3): train_loss: 2.29272971747222 val_loss: 2.28628233080632 train_acc: 30.87 % val_acc: 44.27 %
E(4): train_loss: 2.28650199221341 val_loss: 2.27900313787801 train_acc: 43.58 % val_acc: 50.11 %
E(5): train_loss: 2.27915453389223 val_loss: 2.27068145713672 train_acc: 49.83 % val_acc: 53.03 %
E(6): train_loss: 2.27081444507666 val_loss: 2.26146876486909 train_acc: 53.23 % val_acc: 57.96 %
E(7): train_loss: 2.26166825420412 val_loss: 2.25157253003942 train_acc: 58.10 % val_acc: 64.24 %
E(8): train_loss: 2.25193097146267 val_loss: 2.24115895041642 train_acc: 64.06 % val_acc: 69.74 %
E(9): train_loss: 2.24174867350854 val_loss: 2.23020096399562 train_acc: 69.18 % val_acc: 72.04 %
E(10): train_loss: 2.23105780021641 val_loss: 2.21832726902361 train_acc: 71.28 % val_acc: 71.28 %
E(11): train_loss: 2.21946522587590 val_loss: 2.20477433821768 train_acc: 70.37 % val_acc: 68.86 %
E(12): train_loss: 2.20620803452017 val_loss: 2.18864099661297 train_acc: 68.46 % val_acc: 67.54 %
E(13): train_loss: 2.19039406029524 val_loss: 2.16929315734750 train_acc: 67.02 % val_acc: 67.23 %
E(14): train_loss: 2.17138802707911 val_loss: 2.14654124707275 train_acc: 66.58 % val_acc: 67.55 %
E(15): train_loss: 2.14899740347184 val_loss: 2.12068079045255 train_acc: 67.04 % val_acc: 68.71 %
```

Figura 5.9: $\eta = 10^{-6}$, $\mu = 0.9$ (immagine tagliata)




```
Accuracy score on test set is:  91.54 %.
```

Commento

Le prestazioni sono ulteriormente migliorate, seppur di poco. Il modello ha richiesto circa 1 minuto e 30 secondi per l'apprendimento.

5.5 Modello Echo - 500 nodi interni

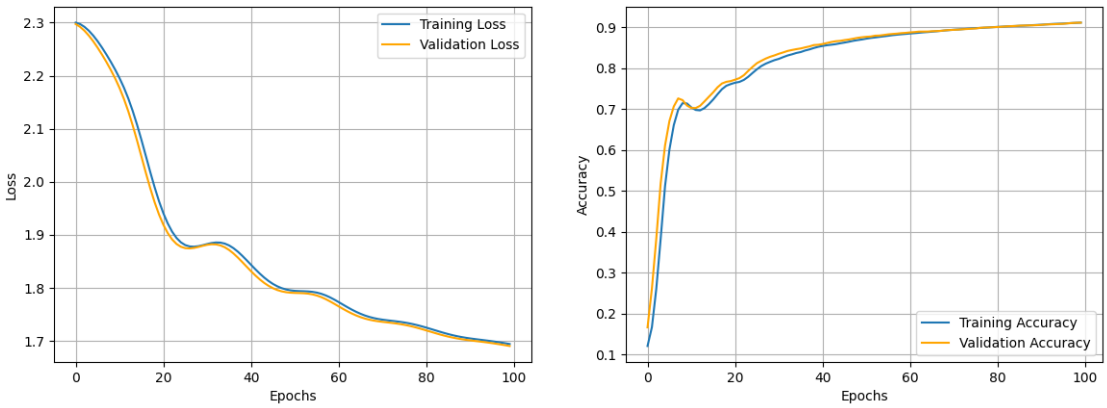
Aumentiamo il numero di neuroni interni a 500.

5.5.1 Versione 1

Riproponiamo l'esperimento quando il numero di neuroni interni è 500.

```
E(0): train_loss: 2.30011498795693 val_loss: 2.29733972037681 train_acc: 12.03 % val_acc: 16.59 %
E(1): train_loss: 2.29733136549993 val_loss: 2.29210472291242 train_acc: 16.63 % val_acc: 26.10 %
E(2): train_loss: 2.29211528456108 val_loss: 2.28475619057225 train_acc: 26.04 % val_acc: 38.31 %
E(3): train_loss: 2.28480105425654 val_loss: 2.27558156481022 train_acc: 38.27 % val_acc: 51.83 %
E(4): train_loss: 2.27567601446934 val_loss: 2.26485279305889 train_acc: 51.18 % val_acc: 61.13 %
E(5): train_loss: 2.26500579622165 val_loss: 2.25284544406206 train_acc: 60.11 % val_acc: 67.11 %
E(6): train_loss: 2.25306444028599 val_loss: 2.23980988595252 train_acc: 66.11 % val_acc: 70.68 %
E(7): train_loss: 2.24010345910525 val_loss: 2.22589799822420 train_acc: 69.85 % val_acc: 72.64 %
E(8): train_loss: 2.22626608144921 val_loss: 2.21104798690906 train_acc: 71.47 % val_acc: 72.22 %
E(9): train_loss: 2.21149357217875 val_loss: 2.19490688065498 train_acc: 71.41 % val_acc: 70.90 %
E(10): train_loss: 2.19544320826771 val_loss: 2.17687091572691 train_acc: 70.44 % val_acc: 70.30 %
E(11): train_loss: 2.17750928416368 val_loss: 2.15630186164233 train_acc: 69.81 % val_acc: 70.28 %
E(12): train_loss: 2.15705807488806 val_loss: 2.13280536593556 train_acc: 69.71 % val_acc: 70.82 %
E(13): train_loss: 2.13369856352554 val_loss: 2.10640134199620 train_acc: 70.27 % val_acc: 71.90 %
E(14): train_loss: 2.10745470759971 val_loss: 2.07759622367713 train_acc: 71.23 % val_acc: 73.03 %
E(15): train_loss: 2.07882938513512 val_loss: 2.04735793362139 train_acc: 72.37 % val_acc: 74.13 %
```

Figura 5.10: $\eta = 10^{-6}$, $\mu = 0.9$ (immagine tagliata)




```
Accuracy score on test set is:  92.00 %.
```

Commento

Risulta evidente che questo è il miglior modello poiché è in grado di imparare, in un numero molto ridotto di epoche – come si evince dai grafici – le cifre di MNIST, con un'accuratezza sul test set del 92%. L'apprendimento tra l'epoca iniziale e la 20-esima epoca risulta essere estremamente veloce, poi tra la 20-esima e la 40-esima c'è un piccolo peggiormaneto del learning, siccome i valori delle funzioni di errore

salgono (di poco) rispetto a prima, poi dalla 60-esima epoca in poi il learning prosegue bene. Ovviamente questo risultato si riflette anche sui grafici dell'accuratezza, dove tra l'epoca iniziale e la 20-esima ha un picco di salita molto ripido. Tuttavia il modello ha impiegato ben 2 minuti per il learning: di fatto è possibile ottenere prestazioni molto simili ma dimuendo anche di parecchio i tempi di attesa. Quest'ultima considerazione, ovvero quella legata ai tempi di attesa è un fattore estremamente dipendente dall'hardware e non particolarmente oggettivo: non è un caso che molti framework di machine learning sfruttino approcci ottimizzati impiegando, ad esempio, l'accelerazione della GPU.

5.6 Il peggiore ed il migliore

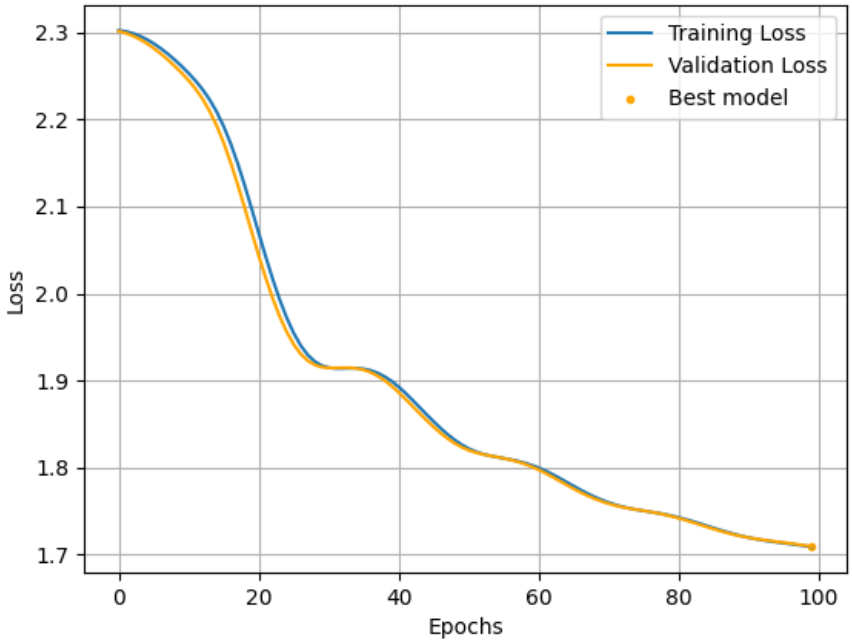
Bisogna fare una premessa importante: in generale un modello troppo complesso *potrebbe* facilmente portare al problema dell'*overfitting* soprattutto per problemi di machine learning particolarmente difficili, peccando di generalizzazione: un modello complesso *non* implica necessariamente un modello migliore. Parallelamente il teorema di approssimazione universale ci assicura l'*esistenza* di una rete shallow sufficientemente espressiva da poter approssimare in modo più o meno accurato una qualsiasi funzione continua definita su un intervallo chiuso e limitato $D \subset \mathbb{R}^d$ quando il numero di neuroni interni è sufficientemente alto e quando le funzioni di attivazione per lo strato interno e quello di output sono, rispettivamente, non polinomiali ma derivabili e trasformazioni lineari (motivo per cui è stata scelta la ReLU/sigmoide per lo strato interno e la funzione identità per quello di uscita). Questo teorema, tuttavia, risponde ad un problema *decisionale*: assicura soltanto l'esistenza di un modello così espressivo, ma non suggerisce in che modo è possibile trovarlo perché esso sarà compito del learning che, tipicamente, fa uso di strategie euristiche. Per questo motivo bisogna fare varie prove ed esperimenti.

Il modello *peggiore* è sicuramente la rete Alfa, con 50 nodi interni, un `l_rate` $\eta = 10^{-4}$ e `momentum` $\mu = 0$: non si è raggiunta assolutamente la capacità di generalizzare il problema e persino sui dati di training il modello ha prestazioni disastrose: inoltre un'accuratezza del 10% sul test set corrisponde ad una predizione *a caso*: se ci venisse fornita un'immagine MNIST e lanciassimo un dado a 10 facce, avremmo una prestazione equivalente a quella ottenuta da questo modello. Il tasso di errore (error rate) sul test set è del $100 - 10 = 90\%$

Per quanto riguarda il modello *migliore*, le versioni "ottimali" per ciascuno dei cinque modelli, ovvero quelle con `l_rate` $\eta = 10^{-6}$ e `momentum` $\mu = 0.9$, hanno prestazioni

più o meno equivalenti e ciascuna dimostra un'accuratezza sul test set del $90 \pm 2\%$. Il modello Echo, con 500 neuroni interni, è quello oggettivamente più accurato sui dati di test (92%) e richiede meno epoche per l'apprendimento, a dispetto tuttavia di un lungo periodo di training in termini di tempo (circa due minuti) anche se questo risultato è poco significativo, da un punto di vista teorico-computazionale, perché estremamente dipendente dall'hardware su cui si lancia la sperimentazione; il modello Charlie, quello con 200 neuroni interni, potrebbe essere un ottimo compromesso perché riesce ad apprende piuttosto velocemente, specialmente tra la 15-esima e la 20-esima epoca. Sceglieresti un modello con un'accuratezza del 98% sui dati di test, ma che costa $\$100,000$ per essere addestrato, oppure un modello accurato al 96% ma che costa $\$10,000$? Spesso, in contesti reali, non è fattibile attendere troppo tempo per il learning e bisogna trovare un compromesso.

In Figura viene illustrato un grafico più dettagliato sull'apprendimento di questo modello; viene evidenziata anche l'epoca in cui l'errore sul validation set è il minimo.



Si noti, infine, che la curva dell'errore, a partire dalla 80-esima epoca, risulta *strettamente decrescente* ed il `best_model` si trova in corrispondenza di una delle ultime

iterazioni, come evidenziato dal pallino arancione: aumentando il numero di epoche è plausibile che il modello migliori ancora di più le sue prestazioni, minimizzando ulteriormente la funzione di costo.

Note finali

Tutto il progetto è stato versionato tramite un VCS, il codice sorgente è disponibile qui: `https://github.com/alessandroquirile/Neural-Networks-Project.git`

Tutti gli esperimenti sono stati eseguiti su questa macchina:

Specifiche di sistema	
Sistema Operativo	MacOS Ventura 13.0.1
CPU	Apple M1
Memoria	16GB LPDDR4
Disco	Macintosh Apple SSD 500GB

L'intero documento è stato scritto e compilato con un editor LaTeX.