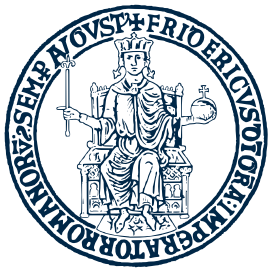
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Neural Networks: Progettazione ed implementazione di una rete neurale per il dataset MNIST

Traccia 1

Studente
Ivan Capasso

N97000381

Anno Accademico 2021–2022

Indice

1 Introduzione

1.1 Traccia

E' stato chiesto di:

- Progettare ed implementare funzioni per simulare la propagazione in avanti di una rete neurale multi-strato. Dando la possibilità di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione per ciascun strato.
- Progettare ed implementare funzioni per la realizzazione della back-propagation per reti neurali multi-strato, per qualunque scelta della funzione di attivazione dei nodi della rete e la possibilità di usare almeno la somma dei quadrati o la cross-entropy con e senza soft-max come funzione di errore.
- Partendo dal dataset mnist, si costruisca un problema di classificazione a C (10) classi. Si fissi la resilient backpropagation (RProp) come algoritmo di aggiornamento dei pesi. Si studi l'apprendimento di una rete neurale con uno strato di nodi interni al variare del numero di nodi e con funzione di errore cross-entropy più softmax.

1.2 Cenni teorici

Ci poniamo in un contesto di apprendimento supervisionato, ovvero un processo di learning guidato da dati etichettati.

Esistono principalmente due tipologie di problemi supervisionati: classificazione e regressione.

Noi consideriamo problemi di classificazione in cui, ad esempio, si vuole categorizzare delle immagini di dimensione $m \times n$, le quali appartengono ad un numero fissato e conosciuto di categorie o classi (supervisione), tramite un algoritmo che mi dica a quale classe è associata l'immagine.

Tuttavia la difficoltà è nel voler generalizzare tale algoritmo in modo da poter predire accuratamente nuove osservazioni.
Si parla dunque di un approccio di machine learning (*ML*) che ci permette di adottare sempre lo stesso algoritmo per risolvere qualsiasi problema di classificazione.

La prima cosa da fare in un processo di learning con generalizzazione è avere un grande numero di dati a disposizione e definire una rappresentazione standard delle features.
Definito il dataset, definisco:

- Il numero di variabili d'ingresso d
- Il numero di nodi di output c (Numero di classi)
- Il numero di nodi interni m
- Numero di strati dei nodi interni n

Fissati m ed n, definiamo un modello di rete con questi parametri, il nostro obbiettivo è trovare i parametri $w1$ $b1$ $w2$ $b2$ (Caso di una rete Shallow) per un problema di apprendimento dato.

Per fare ciò è necessaria una qualche forma di valutazione di questa approssimazione che ci permetta di individuare i parametri che la migliorano.
A tal fine viene introdotta una funzione di errore tale per cui è possibile ridefinire il processo di learning.

Il compito del processo di learning è individuare i parametri O per i quali si minimizza la funzione di errore E, ovvero per i quali si approssimano al meglio le probabilità condizionate.

Back Propagation: Il gradiente di una funzione (E,nel nostro caso) è il vettore delle derivate parziali della funzione, misura quanto l'output di una funzione cambi variando l'input di poco. Siamo quindi interessati a calcolare il gradiente della funzione d'errore E.

Il calcolo del gradiente si basa sull'algoritmo di back-propagation introdotto negli anni '80, molto efficiente in termini di complessità computazionale ($O(p)$ con p numero dei parametri della rete).

Esso si limita al calcolo del gradiente, ovvero il calcolo delle derivate della funzione di errore. Queste possono essere poi adoperate da tecniche come la discesa del gradiente per individuare i pesi e bias che minimizzano la funzione di errore stessa.

Modalità di learning: Il calcolo delle derivate eseguito dall'algoritmo di back-propagation viene affiancato da una modalità di learning che definisce in quale momento e per quanto tempo applicare la regola di aggiornamento:

- Online learning: per ogni coppia (x^n, t^n) calcolate le derivate di (E^n) e aggiorna i pesi
- Batch learning: calcolata la derivata di E rispetto a w_{ij} e aggiorno i pesi

L'analisi dei costi in termini di spazio e tempo tiene conto di un aggravante, ovvero l'iterazione, detta epoca, di questo aggiornamento fino al soddisfacimento di una condizione di uscita.

Da un punto di vista temporale (operazioni da svolgere), la modalità online è più pesante perché i pesi si aggiornano per ogni coppia del dataset, rispetto alla modalità batch che aggiorna ad ogni derivata dell'errore totale.

Tuttavia l'online learning potrebbe velocizzare la convergenza dell'apprendimento perché si verifica la condizione di uscita ad ogni coppia presa in considerazione per il calcolo di $E(n)$, mentre il batch learning causerebbe un'importante occupazione di memoria dovendo mantenere le derivate delle singole funzioni di errore E(n) per poi calcolare quella complessiva.

2 Architettura del sistema

Dobbiamo costruire una rete neurale multistrato full-connected, ovvero una rete in cui per ogni neurone, questo riceve una connessione da tutti i neuroni dello strato precedente.

Le rete può essere anche di tipo shallow, ovvero reti neurali con un solo strato nascosto.

2.1 Funzionalità richieste

Si potranno selezionare, modificando codice nel main:

- Funzione di attivazione (per nodi nascosti e di output)
- Funzione di errore (Cross-Entropy+SoftMax , Sum of Squares)
- Funzione di aggiornamento dei pesi (Gradiente standard,Rprop)

E' anche possibile modificare gli iperparametri della rete,ovvero:

- *eta* (learning rate)
- K (numero di strati della rete)
- M (numero di nodi dello strato interno)
- *MAX_EPOCHES* (numero di epoche da attuare nella fase di learning)
- *dim_train e dim_val* (Dimensione rispettivamente del training e del validation set)

Al termine della fase di learning, verrà mostrato un plot riguardante l'andamento dell'errore, all'aumentare delle epoche.

2.2 Strutture dati utilizzate

La rete neurale è stata rappresentata come un oggetto composto dai seguenti attributi:

- W : Rappresenta i pesi della rete
- b: Rappresenta i bias della rete
- d: numero di variabili d'ingresso
- m : numero di nodi interni di uno strato
- c : numero di nodi d'uscita di uno strato
- k : numero di strati della rete
- f : funzioni di attivazione per gli strati interni; una per ogni layer, potenzialmente differente le une dalle altre.
- g : funzione di attivazione per lo strato di output

I nodi della rete sono rappresentati dal vettore W, un'array di matrici che, come numero di righe ha il numero degli archi entranti, come colonna, il numero di archi uscenti.

Esempio: Nel nostro caso d'uso (dataset mnsit) nel primo livello della rete, supponendo che m=50, l'array W(1) avrà dimensione 784x50.
I bias invece sono rappresentati come array contenenti vettori colonna (esempio per i nodi interni b(1)=50x1).

2.3 Algoritmi utilizzati

L'algoritmo principale è quello della learning phase, nel nostro caso, parliamo di un algoritmo di learning per una rete multistrato full-connected con il metodo d'aggiornamento batch, utilizzante come metodo d'aggiornamento la discesa del gradiente standard (Nella parte B vedremo il caso specifico con Rprop).

Algorithm 1 Batch learning (SGD)

Require: $net, xTrain, tTrain, xVal, tVal, eta, n\_epoches$
  for $epoch = 1$ to $n\_epoches$ **do**
    for $n = 1$ to $dim\_training$ **do**
      //Estraggo un punto dal training set
      $derivate\_pesi, derivate\_bias \leftarrow backProp(miaNet, x\_sin, t\_sin)$
5: **end for**
    for $i = 1$ to k **do**
      $tot\_deriv\_pesi(i) \leftarrow tot\_deriv\_pesi(i) + derivate\_pesi(i)$
      $tot\_deriv\_bias(i) \leftarrow tot\_deriv\_bias(i) + derivate\_buas(i)$
    end for
10: $miaNet \leftarrow discesaDelGradienteStandard(miaNet, tot\_deriv\_pesi(i), tot\_deriv\_bias(i)$
    //Simulo la rete usando come input l'intero training e validation set.
    //Applico la cross-entropy all'output della rete
    if $err\_validation(epoca) < min\_err$ **then**
      $min\_err \leftarrow err\_val(epoca)$
15:   $newNet \leftarrow miaNet$
    end if
  end for
  return $newNet$

A riga 4, dalla back propagation, ricavo le derivate dei pesi/bias per un singolo punto del training set, dopodiché a riga 8/9 sommo questa derivate con tutte le altre derivate dei punti precedenti.

Iterato su tutti i punti del training applico la regola d'aggiornamento, nel nostro caso la discesa del gradiente standard, per ottenere una nuova rete con i pesi/bias aggiornati.

Simulo la rete su tutti i punti del training/validation set, e tramite la funzione di errore (nel nostro caso cross-entropy) mi ricavo l'errore sul training/validation set.
Mi chiedo se questo nuovo errore sul validation è il più piccolo trovato fino ad ora, se si, mi salvo l'errore e la rete ricavata da quest'epoca.

Diamo una rapida occhiata alla funzione di back propagation

Algorithm 2 Back propagation

Require: net,x,t
  // Forward Propagation
  $layers\_input, layers\_output \leftarrow forwardStep(net, x)$
  //Calcolo delta
  $delta \leftarrow get\_delta(net, t, layers\_input, layers\_output)$
5: //Calcolo derivate
  $weights\_deriv, bias\_deriv \leftarrow get\_weights\_bias\_deriv(net, x, delta, layers\_output)$

Nella forward step l'input passa attraverso la rete e ogni strato nascosto accetta i dati in ingresso, li elabora secondo la funzione di attivazione (f) e passa al livello successivo.

Si continua finché non si arriva allo strato di output, dove verrà utilizzata una funzione di attivazione apposta per lo strato d'output (g).

gli output di queste funzione di attivazione serviranno per il calcolo dei delta, e dà li calcolare la derivata dei pesi/bias.

Terminata la fase d'addestramento, si procede ad aggiornare i nodi della rete utilizzando la discesa del gradiente standard.

Algorithm 3 Discesa del gradiente standard

Require: $net, deriv\_pesi, deriv\_bias, eta$
  for $i = 1$ to k **do**
    $net.W(i) \leftarrow net.W(i) - (eta * deriv\_pesi(i))$
    $net.b(i) \leftarrow net.b(i) - (eta * deriv\_bias(i))$
  end for
5: **return** net

Ad ogni epoca si ricalcola il gradiente con i nuovi pesi/bias e si itera fino ad arrivare a convergenza (se esiste).

Supponendo ci sia, a meno di minimi locali, questa regola di aggiornamento porterebbe lentamente a convergenza a causa di un'alta sensibilità delle zone piatte.
Queste portano la derivata ad essere prossima a zero, ottenendo un aggiornamento lento.

3 Specifiche di sistema e analisi dei tempi

E' stata sviluppata una rete neurale multistrato in linguaggio python, il codice è trovabile sul mio profilo github.

Il dataset utilizzato è quello di MNIST, un database di cifre (da 0 a 9) scritte a mano con oltre 60.000 sample; le cifre sono state normalizzate in base alle dimensioni e centrate in un'immagine a dimensione fissa ed è considerato "*l'hello world delle reti neurali*".

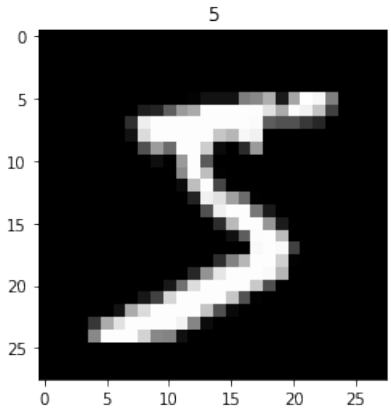


Figura 1: Una rappresentazione grafica di un' elemento del database mnist: Si tratta del numero 5 scritto a mano, l'immagine è composta da pixel bianchi e neri e formano una matrice binaria di dimensioni 28x28

Quello che vogliamo costruire è una rete neurale che, dopo una fase di learning ci consente di stabilire, dato un elemento di questo database, banalmente, quale delle 9 cifre quest'elemento rappresenta. Stiamo parlando quindi di un problema di classificazione a 10 classi, con un numero di nodi d'ingresso pari a 784, ovvero un neurone di ingresso per ogni pixel.

Il progetto è composto da 5 file:

- **main.py:**

```
from learning import batch_learning
import utility as utl
from mnist import MNIST
from net import MultilayerNet

# parametri di default
n_hidden_layers = 2
n_hidden_nodes_per_layer = [35,35]
act_fun_codes = [0,0,1]
error_fun_code = 1

# caricamento dataset
mndata = MNIST('./data/')
X, t = mndata.load_training()
X = utl.get_mnist_data(X)
t = utl.get_mnist_labels(t)
```

```
17
18      X, t = utl.get_random_dataset(X, t, n_samples = 10000)
19      X = utl.get_scaled_data(X)
20
21      X_train, X_test, t_train, t_test = utl.train_test_split(X, t, test_size = 0.25)
22      X_train, X_val, t_train, t_val = utl.train_test_split(X_train, t_train, test_size = 0.3334)
23
24      net = MultilayerNet(n_hidden_layers= n_hidden_layers, n_hidden_nodes_per_layer= n_hidden_nodes_per_layer,
25                          act_fun_codes= act_fun_codes, error_fun_code= error_fun_code)
26
27      net = batch_learning(net, X_train, t_train, X_val, t_val)
28      y_test = net.sim(X_test)
29      utl.print_result(y_test,t_test)
```

Da qui è possibile modificare il numero di layer della rete, nonché i nodi interni per ogni strato. Si può anche cambiare la funzione di attivazione/errore. Si occupa anche di prelevare opportunamente il dataset, dividerlo tra training, validation e test set, istanziare un oggetto di tipo MultilayerNet, avviare la fase di learning su quest'oggetto e infine simulare il test set sulla nuova rete ottenuta per ricavarne accuratezza e precisione.

- net.py:

```
1   class MultilayerNet:
2       def __init__(self, n_hidden_layers, n_hidden_nodes_per_layer, act_fun_codes, error_fun_code):
3           self.n_input_nodes = 784 # dipende dal dataset: mnist_in = 784
4           self.n_output_nodes = 10 # dipende dal dataset: mnist_out = 10
5           self.n_layers = n_hidden_layers + 1
6
7           self.error_fun_code = error_fun_code
8           self.act_fun_code_per_layer = act_fun_codes.copy()
9
10          self.nodes_per_layer = n_hidden_nodes_per_layer.copy()
11          self.nodes_per_layer.append(self.n_output_nodes)
12
13          self.weights = list()
14          self.bias = list()
15          self.deltaW = list()
16          self.deltaB = list()
17
18          self.__initialize_weights_and_bias()
19
20      def __initialize_weights_and_bias(self):
21          mu, sigma, lr = 0, 0.1,0.01
22
23          for i in range(self.n_layers):
24              if i == 0:
25                  self.weights.append(np.random.normal(mu, sigma,
26                  size=(self.nodes_per_layer[i], self.n_input_nodes)))
27                  self.deltaW.append(np.full((self.nodes_per_layer[i], self.n_input_nodes), lr))
28              else:
29                  self.weights.append(np.random.normal(mu, sigma,
```

```
                size=(self.nodes_per_layer[i], self.nodes_per_layer[i-1])))
                self.deltaW.append(
                np.full((self.nodes_per_layer[i], self.nodes_per_layer[i-1]), lr))

                self.bias.append(np.random.normal(mu, sigma, size=(self.nodes_per_layer[i], 1)))
                self.deltaB.append(np.full((self.nodes_per_layer[i], 1), lr))

    def forward_step(self, x):
        def forward_step(self, x):
        layers_input = list()
        layers_output = list()

        for i in range(self.n_layers):
            if i == 0:
                input = np.dot(self.weights[i], x) + self.bias[i]
                layers_input.append(input)

            else:
                input = np.dot(self.weights[i], layers_output[i-1]) + self.bias[i]
                layers_input.append(input)

            act_fun = fun.activation_functions[self.act_fun_code_per_layer[i]]
            output = act_fun(input)
            layers_output.append(output)

        return layers_input, layers_output
    def sim(self, x):
        //Implementazione simulazione rete.
```

Notiamo che la struttura dell'oggetto MultilayerNet è quella descritta in 2.2.

Notare anche l'attributo *"nodes_per_layer"* un'array di interi che semplicemente in base all'intero passato a quello strato della rete, capisce quale funzione di attivazione usare (le associazioni tra numeri e funzioni è presente nel file functions.py).

Inoltre, MultilayerNet, definisce i metodi per la *forward step* e la *simulazione* della rete. Nella forward step, per ogni layer della rete, effettuo il prodotto scalare, o con x (primo strato) oppure con l'output dello strato precedente.

il risultato di quest'operazione viene dato alla funzione di attivazione di quello strato, e quello che verrà restituito è proprio l'output di questo strato di errore (per completezza, si restituisce anche la lista contenenti i prodotti scalari).

- learning.py: Racchiude la implementazione del batch learning, con le relative implementazione degli algoritmi di back-propagation, SGD ed rProp. Vediamo l'implementazione della back propagation:

```
def __back_propagation(net, x, t):
    # x: singola istanza
    layers_input, layers_output = net.forward_step(x)
```

```
    delta = __get_delta(net, t, layers_input, layers_output)
    weights_deriv, bias_deriv = __get_weights_bias_deriv(net, x, delta, layers_output)

return weights_deriv, bias_deriv

def __get_delta(net, t, layers_input, layers_output):
    delta = list()
    for i in range(net.n_layers):
        delta.append(np.zeros(net.nodes_per_layer[i]))

    for i in range(net.n_layers - 1, -1, -1):
        act_fun_deriv = fun.activation_functions_deriv[net.act_fun_code_per_layer[i]]

        if i == net.n_layers - 1:
            # calcolo delta nodi di output
            error_fun_deriv = fun.error_functions_deriv[net.error_fun_code]
            delta[i] = act_fun_deriv(layers_input[i]) * error_fun_deriv(layers_output[i], t)

        else:
            # calcolo delta nodi interni
            delta[i] = act_fun_deriv(layers_input[i]) * np.dot(np.transpose(net.weights[i + 1]), delta[i + 1])

    return delta

def __get_weights_bias_deriv(net, x, delta, layers_output):
    weights_deriv = []
    bias_deriv = []

    for i in range(net.n_layers):
        if i == 0:
            weights_deriv.append(np.dot(delta[i], np.transpose(x)))
        else:
            weights_deriv.append(np.dot(delta[i], np.transpose(layers_output[i - 1])))
        bias_deriv.append(delta[i])

    return weights_deriv, bias_deriv
```

- utility.py: Contiene vari metodi utili per compattare le azioni eseguite del main (retrieve del dataset, retrieve di accuratezza e precisione). Contiene anche un metodo che mostra il plot delle funzioni d'errore del training/validation set.
- functions.py: Implementazione delle funzioni di attivazione/errore (e delle loro derivate).

```
# funzioni di attivazione
def identity(x):
    return x

def sigmoid(x):
```

```
6       return 1 / (1 + np.exp(-x))
7
8   # derivate funzioni di attivazione
9   def identity_deriv(x):
10      return np.ones(x.shape)
11
12  def sigmoid_deriv(x):
13      z = sigmoid(x)
14      return z * (1 - z)
15
16  # funzioni di errore
17  def sum_of_squares(y, t):
18      return 0.5 * np.sum(np.power(y - t, 2))
19
20  def cross_entropy(y, t, epsilon=1e-15):
21      y = np.clip(y, epsilon, 1. - epsilon)
22      return - np.sum(t * np.log(y))
23
24  def cross_entropy_softmax(y, t):
25      softmax_y = softmax(y, axis=0)
26      return cross_entropy(softmax_y, t)
27
28  # derivate funzioni di errore
29  def sum_of_squares_deriv(y, t):
30      return y - t
31
32  # da verificare
33  def cross_entropy_deriv(y, t):
34      return - t / y
35
36  def cross_entropy_softmax_deriv(y, t):
37      softmax_y = softmax(y, axis=0)
38      return softmax_y - t
39
```

E' stato utilizzato un data parser in python (python-mnist) per importare correttamente il dataset ed effettuare operazioni di formattazione per l'ottimizzazione.

3.1 Algoritmi utilizzati

La differenza principale dal set-up generico, è l'utilizzo dell'Rprop come regola di aggiornamento. L'obiettivo principale di questa regola, oltre che un miglioramento del-

l'efficacia, è la mancata dipendenza dagli iperparametri (eta, nel caso della discesa standard), ciò rende la model selection molto più semplice.

Vediamo come cambia la fase di learning utilizzando l'algoritmo Rprop.
Innanzitutto nella rete vanno inizializzati, della stessa dimensioni dei pesi/bias, i corrispettivi delta (net.deltaWRP,net.deltabRP),questi array vengono inizializzati a 0,001.

Algorithm 4 Batch learning(Rprop

Require: $net, xTrain, tTrain, xVal, tVal, eta, n\_epoches$
  for $epoch = 1$ to $n\_epoches$ **do**
    for $n = 1$ to $dim\_training$ **do**
      //Estraggo un punto dal training set
      $derivate\_pesi, derivate\_bias \leftarrow backProp(miaNet, x\_sin, t\_sin)$
    end for
    for $i = 1$ to k **do**
      $tot\_deriv\_pesi(i) \leftarrow tot\_deriv\_pesi(i) + derivate\_pesi(i)$
      $tot\_deriv\_bias(i) \leftarrow tot\_deriv\_bias(i) + derivate\_buas(i)$
    end for
    if $epoch == 1$ **then**
      $miaNet \leftarrow discesaDelGradienteStandard(miaNet, tot\_deriv\_pesi(i), tot\_deriv\_bias(i)$
    else
      $miaNet \leftarrow RProp(miaNet, deriv\_pesi, prec\_deriv\_pesi, deriv\_bias, prec\_deriv\_bias)$
    end if
    $prec\_deriv\_pesi \leftarrow deriv\_pesi$
    $prec\_deriv\_bias \leftarrow deriv\_bias$
    //Simulo la rete usando come input l'intero training e validation set.
    //Applico la cross-entropy all'output della rete
    if $err\_validation(epoca) < min\_err$ **then**
      $min\_err \leftarrow err\_val(epoca)$
      $newNet \leftarrow miaNet$
    end if
  end for
  return $newNet$

Osserviamo che alla prima epoca si utilizza la discesa del gradiente standard, dalla 2 epoca in poi l'RPROP.

Questo perché l'RPROP per poter funzionare ha bisogna di conoscere la derivate dei pesi/bias della epoca precedente.

Come annunciato, si vuole cercare di rendere la discesa del gradiente indipendente dagli iperparametri. L'idea di base è:

- Associare a ciascun peso/bias uno step di aggiornamento, chiamato delta
- Modificare opportunamente questi delta durante il learning

Dunque è possibile riscrivere la regola di aggiornamento da (caso gradiente standard):

$$w_{ij}^{(t)} = w_{ij}^{(t-1)} - n * g_{ij}^{(t)}$$

a(caso Rprop):

$$w_{ij}^{(t)} = w_{ij}^{(t-1)} - delta_{ij} * sign(g_{ij}^{(t)})$$

In questo modo, se la derivata risulta essere positiva si decrementa il peso di un singolo delta, mentre se è negativo, si incrementa.

Con questa modifica della regola di aggiornamento, i delta saranno i nuovi parametri che bisognerà aggiornare durante il processo di learning.

Tale aggiornamento avviene considerando le derivate dell'epoca corrente e quella dell'epoca precedente.

Algorithm 5 Rprop

Require: $net, deriv\_pesi, prec\_deriv\_pesi, deriv\_bias, prec\_deriv\_bias$
  for $i = 1$ to k **do**
    if $deriv\_pesi(i) * prec\_deriv\_pesi(i) > 0$ **then**
      //1 Caso: incremento delta
      $net.deltaWRP(i) \leftarrow min(net.deltaWRP(i) * etaPlus, stepSizesPlus)$
    else if $deriv\_pesi(i) * prec\_deriv\_pesi(i) < 0$ **then**
      //2 Caso: Aggiornamenti con pesi di segno discorde
      $net.deltaWRP(i) \leftarrow max(net.deltaWRP(i) * etaMinus, stepSizesMinus)$
      $deriv\_pesi(i) \leftarrow 0$
    end if
    $net.W(i) \leftarrow net.W(i) - (net.deltaWRP(i) * sign(deriv\_pesi(i))$
    //Aggiorno similmente i bias
  end for
  return net

Come vediamo dall'algoritmo di rProp, le derivate hanno un ruolo fondamentale per la modifica dei singoli delta.

- Se nell'epoca t, la derivata di un peso è ¡ 0 si procedere verso il minimo ed è possibile fare salti più grandi.
- Se nell'epoca t, la derivata di un peso è ¿ 0 è stato effettuato un salto che ha superato il minimo globale.

Dunque guardando le due derivate delle due epoche è possibile capire intuitivamente il salto effettuato. In generale vale che:

- Il prodotto delle due derivate è ¡0 il salto è stato troppo grande (in torno al minimo), quindi bisogna ridurre il delta.
- Il prodotto delle due derivare è ¿0 il salto è "corretto", quindi ci si può permettere di incrementare il delta.

Sulla base di questo ragionamento si introducono degli iperparametri che permetto di incrementare e decrementare i delta (etaPlus, etaMinus).

Inoltre, servono altri 2 iperparametri per limitare superiormente e inferiormente gli eta (stepSizesPlus,stepSizesMinus).

Anche se all'apparenza il numero di iperparametri è aumentato in letteratura si conoscono valori che funzionano tipicamente bene.

3.2 Set-up sperimentale

Nei test fatti molti iperparametri sono rimasti fissati, tra cui:

- d=784: la dimensione di un' elemento di input (x)
- c=10 : numero di nodi d'uscita (rappresentano la probabilità di essere un numero da 0 a 9)
- K=2 : due strato nascosto
- f=Sigmoide : funzione di attivazione per gli strati interni
- g=Identità : funzione di attivazione per lo strato di output
- eta=0.0001 : eta per il gradiente standard
- etaPlus=0.5, etaMinus=1.2 : eta per l'rProp
- stepSizesPlus=1e-06 , stepSizesMinus=50 : limiti superiori e inferiori di etaPlus e etaMinus

Il numero di elementi tra training, validation e test set è stato diviso:

- 5.000 Elementi per il training set
- 2.500 Elementi per il validation set
- 2.500 Elementi per il test set

Per un totale di 10.000 elementi complessivi (N_Samples). Ad ogni test, cambiava la

dimensione del numero di nodi interni e le epoche, rispettivamente:

- Test 1) m=5, epoch=500
- Test 2) m=10, epoch=500
- Test 3) m=20, epoch=100
- Test 4) m=35, epoch=100
- Test 5) m=50, epoch=100

Per ogni caso di test si riporta un grafico con i valori della funzione di errore per il training e validation set, si è cercato di evidenziare la zona della funzione dov'era presente il minimo di queste funzioni (escludendo la parte iniziale delle funzioni, che partiva da valori come 11.000).

Il caso di test si conclude con il termine della fase di learning e una simulazione della rete a partire dal test set. Per la misura delle performance si è tenuto conto dell'accuratezza e della precisione della rete.

- Accuratezza: è la misura delle prestazioni più intuitiva ed è semplicemente un rapporto tra l'osservazione prevista correttamente e le osservazioni totali.
- La precisione è il rapporto tra le osservazioni positive corrette e il totale delle osservazioni positive.

Ci aspettiamo che, al crescere del numero di nodi interni, la rete apprenda più velocemente, e che la funzione di errore raggiunga un minimo più velocemente (in un numero di epoche inferiori).

Inoltre, più il minimo è piccolo, più ci aspettiamo che accuratezza e precisione aumentino (Dato che il dataset mnist è bilanciato).

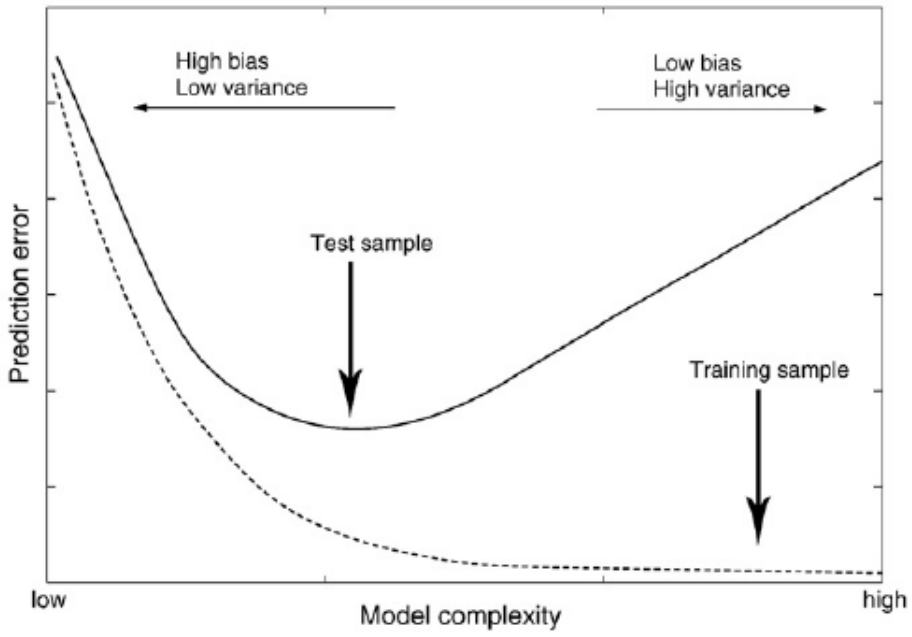


Figura 2: La forma che ci aspettiamo dalle funzioni d'errore sul training/validation set

3.3 Risultati ottenuti

Test 1 Il primo test prevede solo 5 nodi per i 2 layer interni.

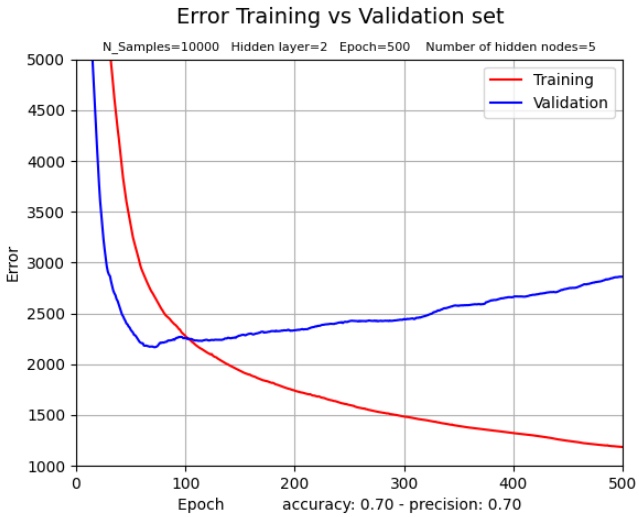


Figura 3: Funzione di errore di training e validation del test 1

Si nota un' apprendimento molto lento, con un'errore sul training che non arriva neanche a 1.000 dopo 500 iterazioni, mentre l'errore minimo sul validation si aggira attorno ad i 2.200.

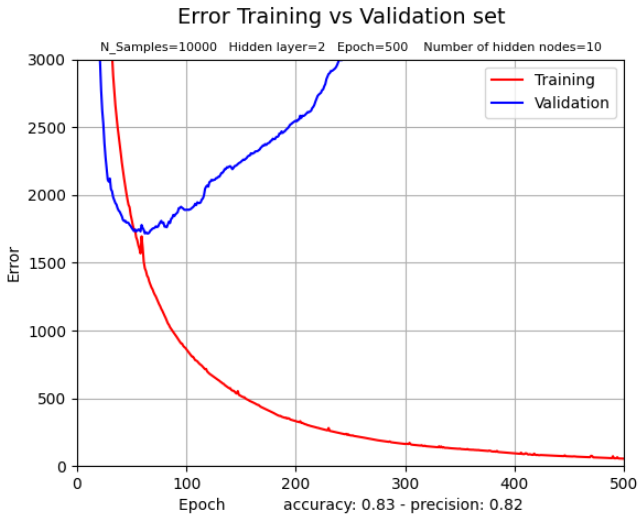


Figura 4: Funzione di errore di training e validation del test 2

Test 2 Già dal secondo caso di test, passando semplicemente da 5 a 10 nodi interni, notiamo come si arrivi ad un errore sul validation che si aggira sui 1.600, e un errore sul training set che si avvicina allo 0.

Test 3 Il terzo caso di test prevede 20 nodi per i 2 layer interni.

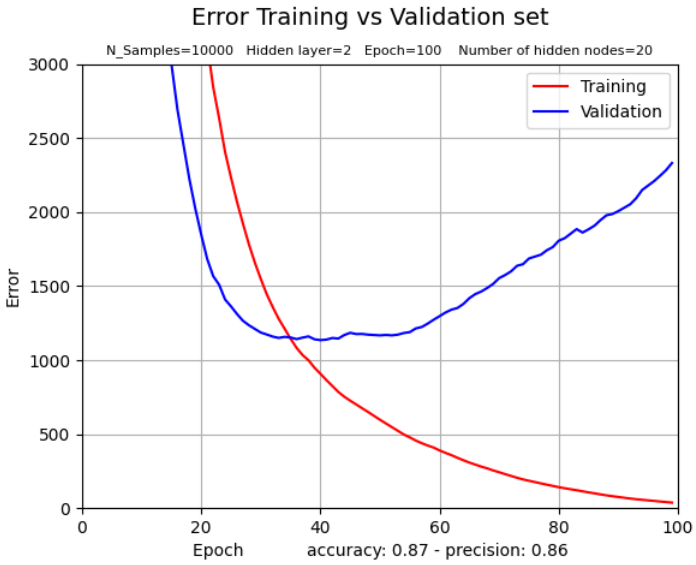


Figura 5: Funzione di errore di training e validation del test 3

Si è deciso di porre le epoche=100 da qui in poi poiché, come si evince dal grafico, il punto di minimo del validation viene raggiunto intorno alla 30° epoca.
Inoltre,l'errore sul training già dall'ottantesima epoca tende allo zero.

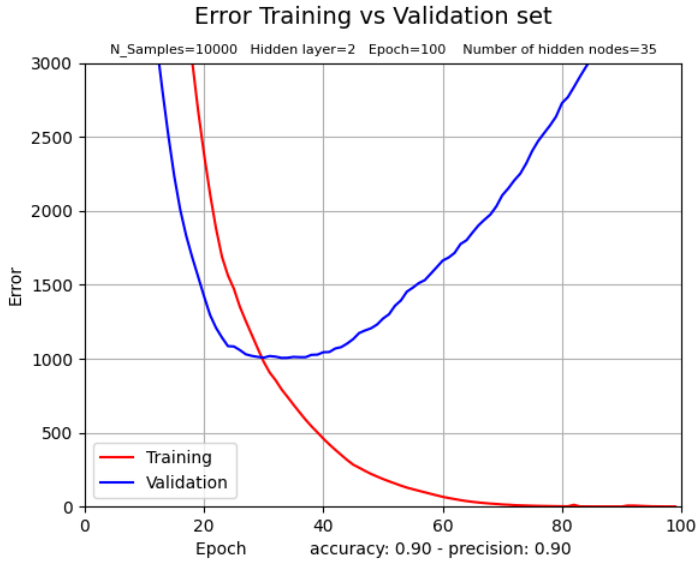


Figura 6: Funzione di errore di training e validation del test 4

Test 4 Nel quarto caso di test osserviamo un calo drastico del minimo della funzione d'errore sul validation, che si aggira sui 1.000.

Test 5 Infine, in quest'ultimo caso di test, abbiamo testato la rete con 50 nodi negli strati nascosti.

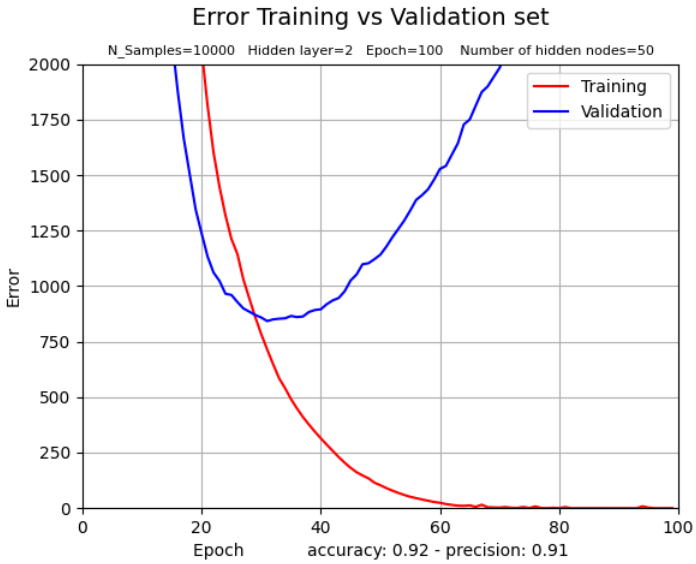


Figura 7: Funzione di errore di training e validation del test 5

L'errore minimo sul validation è ulteriormente diminuito (intorno agli 850) e la funzione d'errore sul training ormai raggiunge lo 0 in poche epoche (attorno alle 60).

3.4 Conclusioni sui risultati ottenuti

E' chiaro come, al crescere di numero di nodi interni, la funzione di learning migliori sempre di più nel giro di poche epoche, e che, almeno fino al numero di neuroni che ho testato, l'andamento tra accuratezza/precisione e il minimo della funzione d'errore sul validation set sia lineare:

Aumentando il numero di neuroni negli strati interni diminuisce l'errore e viceversa. Non sembra essere lineare invece, il numero di epoche necessarie per l'apprendimento invece con il numero di nodi interni:

Infatti, si può osservare, in particolare dal Test 2 al Test 3, che è bastato passare da 10 a 20 nodi interni per raggiungere il minimo errore sul validation rispettivamente dall'epoca 85 all'epoca 35.

Mentre, dal test 4 al 5, il numero di epoche necessarie all'apprendimento, non è cambiato (rimane sulla 30° epoca).

Sebbene nei primi 3 casi di test, i risultati non sono considerabili accettabili (errori alti, accuratezza relativamente bassa), negli ultimi casi di test si raggiunge un'accuratezza/precisione dello 0.92, che riteniamo tutto sommato accettabile.

Una differenza rispetto alle iniziative sono state le prime iterazioni di ogni test, dove l'errore sul training era molto superiore rispetto a quello del validation (anche il doppio, nel primo caso).

Si è notato che questo fenomeno era correlato al numero di elementi totali che si inserivano nei rispettivi set.

Infatti, sono stati fatti anche altri test con il numero di elementi tra training e validation set cambiati (2500 training, 7500 validation), e a prescindere dalle 5 configurazioni la funzione d'errore sul validation assumeva valori superiori rispetto a quella del training.

Osserviamo quindi anche una correlazione tra l'aumento degli elementi dati in input alla rete con l'aumento dell'errore di funzione.

3.5 Sviluppi futuri

Notiamo che il punto minimo dell'errore sul validation che vanno dai 2.200 fino ad 850.

All'inizio è sembrato un valore molto alto, il che ha fatto subito pensare a un bug del programma, anche perché dalla letteratura trovata online si trovano errori molto più piccoli.

Però dopo una lunga fase di debugging sulle 5 reti ricavate ed osservando i vari parametri di model valutation (accuracy,precision etc..) si è arrivati alla conclusione che un'errore alto non implica necessariamente un' apprendimento poco efficiente e che quest'ultimo và rapportarlo, tra tutti gli iperparametri , sopratutto con la dimensione degli elementi di training/validation set.

Detto questo, sicuramente tra gli sviluppi futuri c'è la voglia di migliorare il risultato ottenuto, abbassando i valori delle funzioni d'errore.

Avendo osservato che si tratta di un problema di classificazione con errore lineare, per evitare di fare overtraining, invece che eseguire sempre lo stesso numero di epoche ad ogni test, si potrebbe pensare di implementare un criterio di early-stopping.

Ovvero, volta trovato un nuovo minimo della funzione di errore sul validation set, osservo il risultato delle successive X epoche, se il valore aumenta, posso considerare la mia fase di learning conclusa, poiché saprò di aver trovato il minimo globale della funzione.

In questo modo, usiamo i vantaggi di "precisione" che ci offre il batch learning, e magari andiamo ad alleggerire l'allocazione in memoria.