Documentazione del software

Progetto di Neural Networks and Deep Learning

Università degli Studi di Napoli Federico II
Corso di Laurea Magistrale in Informatica
Docente: Prevete Roberto

Ivano Matrisciano
N97/392

Simone Piccinelli
N97/389

10 agosto 2022

1. Introduzione

Il presente documento ha l'intento di descrivere il funzionamento e alcuni aspetti implementativi di una libreria per la realizzazione di reti neurali scritta in python. Verranno presentate le motivazioni dietro alcune scelte progettuali e la descrizione degli algoritmi principali.
Infine verranno testate le performance della libreria utilizzando il dataset mnist, presentando e commentando i risultati.

1.1 Traccia

Di seguito il virgolettato della traccia seguita per la realizzazione del progetto.

PARTE A (Comune a tutti)

"Progettazione ed implementazione di funzioni per simulare la propagazione in avanti di una rete neurale multi-strato. Dare la possibilità di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di attivazione per ciascun strato.

Progettazione ed implementazione di 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."

PARTE B (Da scegliere)

"7. (Difficoltà medio-alta) Si consideri come input le immagine raw del dataset mnist. Si ha, allora, un problema di classificazione a C classi, con C=10. Si estragga opportunamente un dataset globale di N coppie, e lo si divida opportunamente in training, validation e test set (ad esempio, 5000 per il training set, 2500 per il validation set, 2500 per il test set). Si fissi la resilient backpropagation (RProp) come algoritmo di aggiornamento dei pesi (aggiornamento batch). Si studi l'apprendimento di una rete neurale (ad esempio epoche necessarie per l'apprendimento, andamento dell'errore su training e validation set, accuratezza sul test) con un solo strato di nodi interni e con la sigmoide come funzione di attivazione dei nodi interni al variare del criterio di early-stopping. Facendo riferimento all'articolo "Early Stopping | but when?Lutz Prechelt, 1999", si considerino i due algoritmi di early-stopping GL e PQ con differenti valori del parametro alfa. Considerare reti con un diverso numero di nodi interni (almeno 3 diverse dimensioni). Se è necessario, per questioni di tempi computazionali e spazio in memoria, si possono ridurre (ad esempio dimezzarle) le dimensioni delle immagini raw del dataset mnist (ad esempio utilizzando in matlab la funzione imresize)."

1.2 Cenni teorici

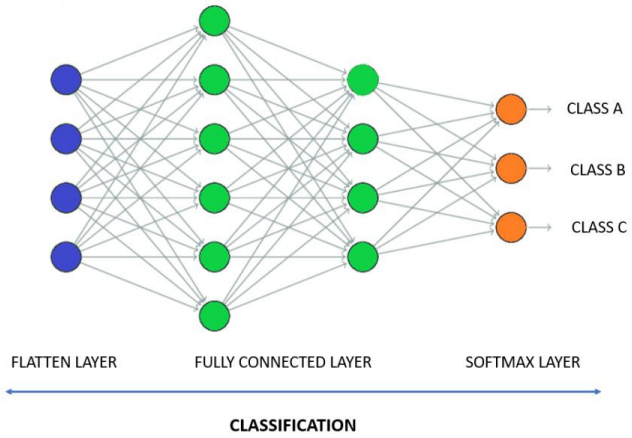


figura 1.2.1, esempio di layer fully connected

La **propagazione in avanti** di uno strato pienamente connesso può essere descritta come moltiplicazione matriciale tra la matrice dei pesi dello strato e il vettore in input allo strato stesso.

$$W \in \Re^{m \times d}$$
$$x \in \Re^{d \times 1}$$
$$b \in \Re^{m \times 1}$$

$$a = Wx + b$$
$$z = f(a)$$

La libreria è stata realizzata sfruttando questa proprietà, in quanto eseguire un'unica moltiplicazione matriciale è più efficiente, in termini prestazionali e di uso della memoria, rispetto a iterare sui singoli neuroni.

La libreria utilizza l'algoritmo della **back-propagation** come metodo per il calcolo del gradiente della funzione di errore rispetto ai pesi. Questo metodo gode della proprietà di avere un tempo di calcolo **lineare** rispetto al numero di pesi, $O\left(N_w\right)$.

Ad ogni peso della rete si associa un valore δ

- δ_i^n: delta associato al nodo i quando in input alla rete c'è il punto n del training set
- z_j^n : l'output del nodo j quando in input alla rete c'è l'elemento n del training set

Evitiamo di scrivere l'apice n, ma lo consideriamo come sottinteso.
La derivata si calcola secondo la seguente regola locale:

$$\frac{\partial E^{(n)}}{\partial w_{ij}} = \delta_i z_j$$

Data la località della regola è possibile calcolare le derivate in modo parallelo. Il calcolo procede a ritroso partendo dallo strato di uscita della rete.

- Per i nodi di output, che identificheremo col pedice k invece di i, $\delta_k = g'_k(a_k) \frac{\partial E^{(n)}}{\partial y_k}$
- Per tutti gli altri nodi $\delta_i = g'_i(a_i) \sum_h \left(w_{hi} \delta_h \right)$

La libreria implementa due diverse strategie di aggiornamento dei pesi, che utilizzano le informazioni ricavate dalla back propagation per modificare i valori dei pesi nel modo più opportuno: il Gradient Descent e la Resilient Back Propagation.

La tecnica del **gradient descent** prevede di aggiornare i pesi nel seguente modo:

$$w_{ij} = w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

Dove η è un iper parametro fisso, detto *learning rate*.

Nella **resilient back propagation** i pesi della rete vengono aggiornati usando un "*learning rate*" Δ che non è costante, ma è personalizzato per ogni peso e viene modificato ad ogni aggiornamento dei pesi:

- Se dopo l'aggiornamento dei pesi, la derivata dell'errore ha mantenuto lo stesso segno rispetto al valore precedente, allora si incrementa il valore Δ moltiplicandolo con una quantità η^+ (maggiore di 1), fino a raggiungere un valore massimo.
- Se invece la derivata ha cambiato segno, il Δ viene decrementato moltiplicandolo per una quantità $\eta^- \in (0, 1)$, fino a raggiungere un valore minimo, e si aggiorna il peso della rete.

Il **Generalization Loss**, all'epoca t-esima, è l'incremento relativo del validation error sul minimo errore registrato.

$$GL(t) = \left(\frac{E_{val}(t)}{E_{min}(t)} - 1 \right)$$

Viene definito un parametro α e il processo di training viene terminato non appena GL(t) supera il valore scelto. Dunque il criterio ferma l'apprendimento quando la rete inizia a overfittare e si perde di generalità.

Tuttavia è possibile preferire o un'interruzione più tempestiva quando i progressi ottenuti in ogni epoch iniziano ad essere sufficientemente bassi, o un'interruzione meno tempestiva nei casi in cui la perdita di generalità consiste in un massimo locale delle funzione di errore, ma l'errore di validazione si sta comunque generalmente abbassando.

Il criterio **Progress Quotient** prevede di analizzare le ultime *k* epoche in modo da costruire una funzione di training progress. Ad ogni epoch viene calcolato il valore $P_k(t)$ nel seguente modo:

$$P_k(t) = 1000\left(\frac{E_{tr-sum}}{k\,E_{tr-min}} - 1\right)$$

Dove $E_{tr\text{-}sum}$ è la somma di tutti gli errori sul training set presenti nelle ultime k-epoch, mentre $E_{tr\text{-}min}$ è il minimo tra questi valori.

Infine, dato un valore α, il processo di training viene interrotto quando $\frac{GL(t)}{P_k(t)} > \alpha$

2. Implementazione

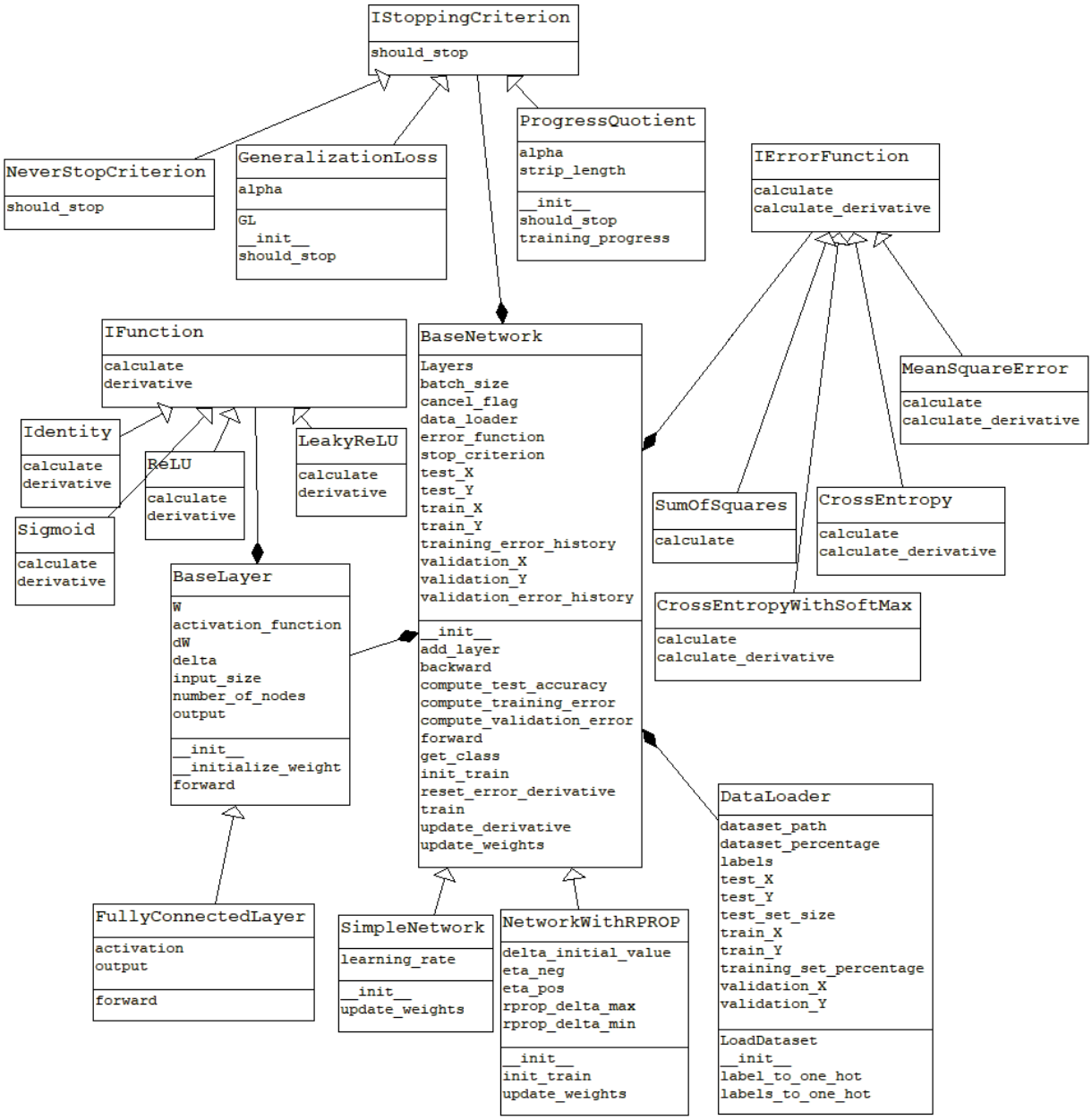
2.1 Panoramica sul progetto

L'utente dovrà avere la possibilità di utilizzare la libreria creando un oggetto rete a cui aggiungere uno o più strati. Per ogni strato dovrà essere specificato il numero di nodi e la funzione di attivazione. L'utente dovrà oltremodo specificare la funzione d'errore della rete e tutti gli iperparametri necessari.

L'utente dovrà inoltre poter specificare un criterio di early stop, il massimo numero di epoche e la dimensione del batch. L'utente dovrà poi poter avviare la fase di training della rete ricevendo a schermo statistiche sull'errore e sul tempo impiegato.
A termine della fase di training la rete dovrà stampare l'accuratezza sul test set.

Una volta concluso il processo di training l'utente dovrà poter utilizzare la rete per fare inferenza.

La libreria realizzata ha la seguente struttura:



2.2 Linguaggio e ambiente di sviluppo

Per l'implementazione degli algoritmi si è scelto di utilizzare il linguaggio python nella sua versione 3.10.4, usando Visual Studio Code come IDE.

2.2.1 Dipendenze

Il software fa uso dei seguenti pacchetti, che è necessario installare per poter eseguire il codice:

- **numpy**, utilizzato per eseguire moltiplicazioni matriciali in modo più performante rispetto alle strutture e procedure messe a disposizione da python
- **matplotlib**, utilizzato per produrre e mostrare a schermo grafici
- **python-mnist**, utilizzato per caricare il dataset in memoria

Per installare i pacchetti menzionati, eseguire il seguente comando:

```
pip install numpy matplotlib python-mnist
```

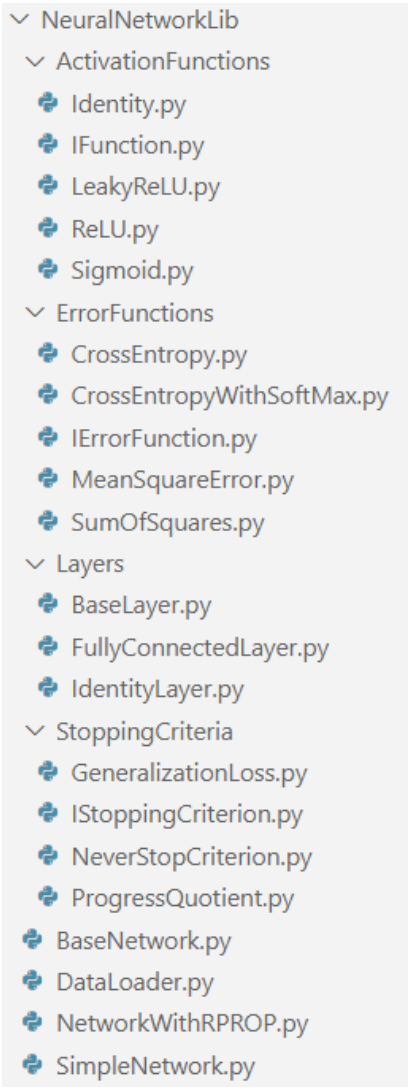
o, analogamente:

```
python3 -m pip install numpy matplotlib python-mnist
```

2.2.2 Struttura dei file

Di seguito è mostrata la struttura dei file che compongono la libreria:

- ***NeuralNetworkLib/***
  - ***ActivationFunctions/*** contiene tutte le funzioni di attivazione realizzate, quali Sigmoide, ReLU e LeakyReLU
  - ***ErrorFunctions/*** contiene tutte le funzioni di errore realizzate, quali CrossEntropy o CrossEntropyWithSoftMax
  - ***Layers/*** contiene i tipi di strati realizzati, come il FullyConnectedLayer
  - ***StoppingCriteria/*** contiene i criteri di stop realizzati, come il GL o PQ
  - *BaseNetwork.py* classe base da cui derivano tutte le reti implementabili
  - *SimpleNetwork.py* rete che implementa lo SGD come algoritmo di aggiornamento dei pesi
  - *NetworkWithRPROP.py* rete che implementa la RPROP come algoritmo di aggiornamento dei pesi
  - *DataLoader.py* per il caricamento del dataset: lettura da file, divisione in training, validation e test set, trasformazione delle label in one-hot e normalizzazione dei valori.



2.3 Implementazione parte A

Per la parte A della traccia è stato necessario realizzare:

- un data loader per caricare in memoria gli elementi del dataset
- la funzione di forward della rete
- il meccanismo di backpropagation della rete
- il meccanismo di calcolo dell'errore e della derivata dei pesi
- il meccanismo di aggiornamento dei pesi

2.3.1 Strutture dati utilizzate

Una rete è composta da una python-list di strati. Ogni strato fully connected contiene a sua volta le seguenti strutture, realizzate con array di numpy:

- una matrice di pesi *W* di dimensioni appropriata
- un vettore di attivazioni (un'attivazione per ogni neurone)
- un vettore di output (un output per ogni neurone)
- un vettore di delta, per implementare la backpropagation (un delta per ogni neurone)
- una matrice *dW* di dimensioni pari alla matrice *W*, che contiene l'errore accumulato in un batch

In più la rete mantiene una lista di loss sul training set e una lista di loss sul validation set, in modo da poter valutare i criteri di stop e di plottare l'andamento degli errori alla fine del processo di training.

In caso di rete con rprop, ogni layer contiene, oltre alle strutture già indicate, una matrice *rprop_deltas* delle stesse dimensioni di W, e una matrice *dW_old*, contenente l'errore al batch precedente.

2.3.2 Descrizione del codice

DataLoader

Il primo step per poter costruire e allenare la rete neurale è disporre di un dataset. La classe DataLoader si occupa di leggere da disco il dataset e di preprocessarlo.
Al momento della costruzione del DataLoader vengono specificati alcuni parametri. Di seguito viene riportata la firma del costruttore della classe:

```
def __init__(self, dataset_path, dataset_percentage = 0.3,
training_set_percentage = 0.75, test_set_size = -1):
```

- dataset_path indica il percorso in cui cercare i file del dataset
- dataset_percentage indica quanti dati caricare in memoria. È un valore compreso tra 0 e 1, dove 1 significa caricare tutti i dati. Per motivi prestazionali si può pensare di abbassare questo valore per allenare la rete usando un dataset più piccolo.
- training_set_percentage indica quanti degli elementi caricati devono essere utilizzati per il training set. Il suo complementare indica quanti dati sono utilizzati per il validation set
- test_set_size indica, in numero assoluto, la dimensione del test set. Se -1 si utilizza il test set nella sua interezza

Il metodo `LoadDataset` fa partire la procedura di caricamento dei dati:

1. Il dataset intero viene letto dal disco.
2. Parte del dataset viene scartata in base al parametro dataset_percentage.
3. Il set rimanente viene diviso in training e validation nelle proporzioni specificate dall'utente tramite il parametro training_set_percentage.
4. Il sistema popola la variabile membro *labels*, contenente i nomi delle classi.
5. Il sistema carica i dati del test set, ignorando parte del dataset come specificato nel parametro test_set_size.
6. Il sistema normalizza i vettori di feature, dividendo tutti i valori per 255
7. Il sistema trasforma tutte le label in vettori one-hot
8. Il sistema va a popolare le seguenti variabili membro: train_X, train_Y, validation_X, validation_Y, test_X, test_Y.
9. La procedura di caricamento termina con la stampa a schermo del numero di elementi caricati per ogni set.

Ogni set X è una matrice numpy di dimensioni *Nxd* in cui le righe rappresentano i punti del dataset e le colonne rappresentano le feature normalizzate.

Ogni set Y è una matrice numpy in cui ad ogni riga, che rappresenta un punto del dataset, viene associato un vettore colonna rappresentante la label del punto sottoforma di vettore one-hot.

La trasformazione da classe a vettore di one-hot è implementata nella funzione `labels_to_one_hot`:

```
def labels_to_one_hot(self, Y) -> np.array:
    one_hot = np.empty( (len(Y), len(self.labels), 1) )
    for i in range(0, len(Y)):
        a = (np.eye(len(self.labels))[Y[i]])
        one_hot[i] = a.reshape(len(a), 1)
    return one_hot
```

Questa funzione accetta in ingresso una matrice Y di dimensioni Nx1, dove ad ognuno degli N elementi del training set è associata la classe. Restituisce un tensore di dimensioni $N \times C \times 1$. Ad ognuno dei N elementi del set si associa un vettore colonna di dimensioni Cx1 contenente la rappresentazione one-hot della classe associata all'elemento stesso.

BaseLayer

Ogni tipo di strato nella libreria eredita da BaseLayer e ne implementa una logica di forward personalizzata.
La classe BaseLayer si occupa unicamente di generare:

- la matrice dei pesi W con una distribuzione di probabilità uniforme centrata in 0
- la matrice dW contenente gli errori dei pesi, di dimensioni pari a W e inizializzata a 0
- il vettore di delta, di dimensioni pari al numero dei nodi e inizializzato a 0

La matrice W viene generata in modo da avere tante righe quanti sono i neuroni nello strato e tante colonne quanti sono gli input di ciascun neurone. Il numero di input di ciascun neurone corrisponde al numero di output del layer precedente (o al numero di feature del dataset, nel caso del primo layer) incrementato di 1, per includere il bias.

```
def __initialize_weight(self, random_min=-0.1, random_max=0.1) -> None:
    W_size = (self.number_of_nodes, self.input_size + 1)
    self.W = np.random.uniform(random_min, random_max, W_size)
    self.dW = np.zeros_like(self.W)
    self.delta = np.zeros(self.number_of_nodes)
```

FullyConnectedLayer

La classe FullyConnectedLayer eredita da BaseLayer e implementa il metodo *forward* nel seguente modo:

```
def forward(self, X: np.array) -> np.array:
    input = np.array(X, copy=True)
    input = np.append(input, 1.0)
    input = input.reshape(len(input), 1)

    self.activation = self.W @ input
    self.output = self.activation_function.calculate(self.activation)
    return self.output
```

Il sistema crea una copia del vettore in input, a questa copia viene aggiunto in coda il valore 1 per includere il bias e ne viene poi fatto un reshape per trasformarlo in un vettore colonna.
Le attivazioni dei neuroni vengono calcolate tramite moltiplicazione matriciale tra i pesi e l'input e infine si calcolano le uscite dei neuroni facendo passare le attivazioni attraverso la funzione di attivazione, specificata nel costruttore.
Sia le attivazioni che gli output vengono salvate come variabili membro del layer.

BaseNetwork

BaseNetwork è la classe base che implementa tutte le funzionalità necessarie al funzionamento della rete, ad eccezione del meccanismo di aggiornamento dei pesi, che sarà specificato nelle classi derivate.

Al momento della costruzione dell'oggetto viene specificato:

- il dataloader,
- la funzione di errore
- il criterio di stop

Una volta aggiunti i layer è possibile far partire il processo di training, di cui viene riportata una versione leggermente semplificata:

Per avviare il **train** della rete è necessario specificare:

- la dimensione del batch
- il numero massimo di epoche

```
def train(self, batch_size, MAX_EPOCH=100):
   self.batch_size = batch_size
   training_access_order = np.arange(len(self.train_X))
   self.init_train()

   for epoch in range(0, MAX_EPOCH):
       np.random.shuffle(training_access_order) #shuffling the training set

       n = 0
       while n < len(self.train_X): # do more batches until we analyze the
whole training set

           self.reset_error_derivative() #reset error between batches
           for b in range(0, self.batch_size):
               #process a batch

               x = self.train_X[training_access_order[n]]
               y = self.forward(x) #network's output
               t = self.train_Y[training_access_order[n]] #golden label
               self.backward(y, t)
               self.update_derivative(x)

               n += 1
           self.update_weights()
       #we now compute training and validation errors and we store them for
later use
       training_error = self.compute_training_error()
       validation_error = self.compute_validation_error()

       self.training_error_history.append(training_error)
       self.validation_error_history.append(validation_error)

       if epoch > 0 and
self.stop_criterion.should_stop(self.training_error_history,
self.validation_error_history): #only check the stopping criterion after the
first epoch
           print("Stopping criterion met.")
           break
```

L’accesso agli elementi del training set viene randomizzato in ogni epoch, in modo da migliorare le performance della rete, soprattutto con batch size ridotti.

La procedura di training parte salvando in una variabile membro la dimensione di batch fornita dall'utente e generando un vettore contenente gli indici per l'accesso agli elementi del training set. Tale vettore sarà mescolato all'inizio di ogni epoch. Viene poi chiamata la procedura init_train, che consente alle classi derivate di specificare codice aggiuntivo prima dell'inizio del training.

Successivamente si itera fino al numero massimo di epoche specificate e, dopo aver mescolato gli indici di accesso agli elementi del training set, si eseguono i batch fino all'esaurimento dei punti del training set.

All'inizio di ogni batch si resettano gli errori sui pesi accumulati, dopodiché si itera su un numero di elementi di training pari al batch size e per ognuno di questi elementi viene calcolato l'output della rete. Viene poi eseguita la backpropagation e si accumula la derivata dell'errore.
Alla fine di un batch si aggiornando i pesi con il meccanismo implementato in update_weights()

Alla fine di ogni epoch viene calcolato l'errore sul training set e sul validation set. Questi errori vengono storicizzati in appositi array e vengono stampati a schermo insieme al tempo impiegato per completare l'epoch.
Dopo la prima epoch viene anche interrogato il criterio di stop e, eventualmente, viene fermato il processo di training stampando a schermo la motivazione.

Il meccanismo di **backpropagation** è così implementato:

```
def backward(self, y, t):
   #calculate delta for the output layer
   output_layer = self.Layers[-1]
   output_layer.delta =
np.multiply(output_layer.activation_function.derivative(output_layer.activatio
n), self.error_function.calculate_derivative(t, y))

   #calculate the delta for every non-output layer, starting from the last
hidden layer all the way to the first layer
   for i in reversed(range(len(self.Layers) - 1)):
       layer = self.Layers[i]
       next_layer = self.Layers[i+1]
       g_prime_in_a = layer.activation_function.derivative(layer.activation)

       for j in range(layer.number_of_nodes): # for each neuron in the given
(non-output) layer
           error = np.dot(next_layer.W[:, j], next_layer.delta)
           layer.delta[j] = error * g_prime_in_a[j]
```

Per la backpropagation si parte col calcolare i delta dello strato di output.
Per ogni neurone nello strato di output, viene calcolato il delta secondo la seguente formula:

$$\delta_k = g'_k(a_k) \frac{\partial E^{(n)}}{\partial y_k}$$

Per tutti gli altri nodi si calcola il delta secondo la formula:

$$\delta_i = g'_i(a_i) \sum_h \left(w_{hi} \delta_h \right)$$

Ovvero, il delta del neurone i-esimo è dato dal valore della derivata della funzione di attivazione g, calcolata nell'attivazione del neurone stesso, moltiplicata per la somma, su tutti i neuroni h che ricevono in ingresso il valore del neurone i, del peso della connessione da i ad h per il delta del neurone h (precedentemente calcolato)

Ogni volta che si esegue la backpropagation, l'errore ottenuto viene accumulato nella matrice dW usando la seguente funzione:

```
def update_derivative(self, x):
    for l in range(len(self.Layers)):
        if l == 0:
            input = x
        else:
            input = self.Layers[l-1].output

        layer = self.Layers[l]

        input = np.append(input.copy(), 1.0)
        layer.dW += layer.delta.reshape(len(layer.delta), 1) @ input.reshape(1,
len(input))
```

Per ogni layer nella rete recuperiamo l'input del layer stesso come l'output del layer precedente, tranne per il primo layer della rete per cui l'input è l'elemento x del training set.
Una volta recuperato l'input del layer, si aggiunge 1 per includere il bias e si incrementa l'errore accumulato di una quantità data dalla moltiplicazione matriciale tra i delta (presi come vettore colonna) e gli input (presi come vettore riga).

Altre funzioni, come il calcolo dell'errore o l'implementazione delle funzioni di attivazione e di errore non verranno analizzate in questo documento.

SimpleNetwork

La classe SimpleNetwork implementa BaseNetwork.
Richiede che nel costruttore sia impostato il learning rate, oltre ai parametri richiesti da BaseNetwork.

Il meccanismo di aggiornamento dei pesi viene implementato nel seguente modo:

```
def update_weights(self):
    for layer in self.Layers:
        layer.W -= self.learning_rate * layer.dW
```

2.4 Implementazione parte B

Per la realizzazione della parte B della traccia si è estesa la libreria, includendo una rete che implementi la Resilient Backpropagation e realizzando i criteri di stop Generalization Loss e Progress Quotient.

L'implementazione della RPROP è stata eseguita seguendo il paper: *"Rprop - Description and Implementation Details, Martin Riedmiller, 1994"*

I criteri di stop sono stati realizzati seguendo il paper *"Early Stopping | but when? Lutz Prechelt, 1999"*, così come indicato nella traccia.

2.4.1 Descrizione algoritmi

La resilient backpropagation prevede di ignorare il modulo dell'errore di un peso, considerandone invece il solo segno. L'rprop prevede inoltre di avere learning rate personalizzati per ogni peso, capaci di crescere e decrescere dinamicamente in base all'andamento dell'errore.
Se l'errore mantiene lo stesso segno per più iterazioni, significa che aggiornando il peso lo si sta muovendo nella direzione giusta, è così possibile pensare di aumentare il learning rate associato al peso in modo da ridurre il numero di epoche necessarie.
Alternativamente, se l'errore sul peso cambia segno, significa che l'ultimo aggiornamento è stato troppo grande e si diminuisce il learning rate. Per evitare di punire più volte un cambio di segno, il delta viene azzerato e il peso verrà modificato al successivo passo di aggiornamento.

Per quanto riguarda i criteri di stop, Generalization Loss prevede di fermare il training appena l'errore sul validation risulta peggiorare rispetto al miglior errore sul validation registrato. Il peggioramento deve superare una soglia definita con il parametro alpha.
Progress Quotient invece tiene conto anche della *velocità* del processo di learning, per cui anche se l'errore sul validation aumenta, si decide di non interrompere subito il training se questo sta generalmente migliorando abbastanza velocemente. Questo lo si ottiene andando a definire una finestra di dimensioni fisse sugli ultimi elementi dello storico del training error. Si calcola poi la somma degli errori in questa finestra e la si rapporta all'errore minimo nella finestra stessa. Infine si calcola il rapporto tra GL e la quantità appena ricavata. Se questo rapporto supera una certa soglia, il processo di training viene interrotto.

2.4.2 Descrizione del codice

NetworkWithRPROP

La classe eredita da BaseNetwork e il costruttore accetta i seguenti parametri aggiuntivi:

- eta_pos
- eta_neg

Il metodo init_train() viene chiamato all'inizio del processo di training e ha il compito di allocare le strutture dati necessarie.

```
def init_train(self):
    super().init_train()
    for layer in self.Layers:
        layer.dW_old = np.zeros_like(layer.W)
        layer.rprop_deltas = np.empty_like(layer.W)
        layer.rprop_deltas.fill(self.delta_initial_value)
```

Per ogni layer viene dunque allocata una matrice dW_old, in cui memorizzare gli errori ottenuto al passo precedente; viene allocata la matrice di delta, uno per ogni peso della rete. Questa matrice viene infine inizializzata a delta_initial_value, impostato di default a 0.01

Di seguito viene riportato il codice che implementa l'aggiornamento dei pesi:

```
def update_weights(self):
    for layer in self.Layers:
        #we now compute the element-wise multiplication between dW and dW_old
        #this gives us a matrix with the same shape as dW
        #each cell will be > 0 if dW and dW_old have the same sign, < 0 if they
have opposite sign or 0 if one of them is 0
        dw_sign = np.multiply(layer.dW, layer.dW_old) #np.sign is not necessary

        #we now iterate over every element of that matrix
        for i in range(layer.number_of_nodes):
            for j in range(layer.input_size + 1):
                if (dw_sign[i][j] > 0): #if the error kept the same sign
                    layer.rprop_deltas[i][j] = min(layer.rprop_deltas[i][j] *
self.eta_pos, self.rprop_delta_max) #update the rprop_delta, increasing it by
eta_pos until we hit rprop_delta_max
                    layer.W[i][j] -= np.sign(layer.dW[i][j]) *
layer.rprop_deltas[i][j] #update the weight
```

```
                    layer.dW_old[i][j] = layer.dW[i][j] #update the old error
                elif dw_sign[i][j] < 0 : #if the error changed sign
                    layer.rprop_deltas[i][j] = max(layer.rprop_deltas[i][j] *
self.eta_neg, self.rprop_delta_min) #decrease rprop_delta until we hit
rprop_delta_min
                    layer.dW_old[i][j] = 0 #zero-out the old error
                else: #if the error was 0
                    layer.W[i][j] -= np.sign(layer.dW[i][j]) *
layer.rprop_deltas[i][j] #update the weight
                    layer.dW_old[i][j] = layer.dW[i][j] #update the old error
```

Per ogni layer si esegue la moltiplicazione elemento per elemento delle matrici dW e dW_old e si salva il risultato in dw_sign. Si itera poi su ogni elemento di dw_sign:

- se l'errore del peso ha mantenuto lo stesso segno, si aumenta il delta del peso fino ad un valore massimo, si aggiorna il peso considerando il segno dell'errore e il delta appena aggironato e infine si aggiorna dW_old
- se l'errore del peso ha cambiato segno, si diminuisce il delta fino ad un cetro minimo, si aggiorna dW_old impostando il relativo valore a 0, ma non si aggiorna ancora il peso
- se l'errore è 0 si diminuisce il peso e si aggiorna dW_old

```
class GeneralizationLoss(IStoppingCriterion):
    def __init__(self, alpha):
        self.alpha = alpha

    def should_stop(self, training_error_history, validation_error_history) ->
bool:
        should_stop = GeneralizationLoss.GL(validation_error_history) >
self.alpha
        return should_stop

    def GL(validation_error_history):
        best_validation_error = np.min(validation_error_history) #the lowest
validation error so far, aka E_opt
        last_validation_error = validation_error_history[-1] #aka E_va

        GL = 100 * (last_validation_error / best_validation_error - 1)
        return GL
```

```
class ProgressQuotient(IStoppingCriterion):
    def __init__(self, alpha, strip_length):
        self.alpha = alpha
        self.strip_length = strip_length # aka k

    def should_stop(self, training_error_history, validation_error_history) ->
bool:
        GL = GeneralizationLoss.GL(validation_error_history)
        P_k = ProgressQuotient.training_progress(self.strip_length,
training_error_history)

        PQ = GL/P_k

        should_stop = PQ > self.alpha
        return should_stop

    def training_progress(strip_length, training_error_history):
        """how much was the average training error during the strip larger
than the minimum training error during the strip?"""

        strip_start = max(len(training_error_history) - strip_length, 0) # if
there are not enough epochs, the strip_length is the biggest possible
        strip_end = len(training_error_history)

        strip = training_error_history[strip_start:strip_end]
        strip_error_sum = np.sum(strip)
        strip_min_error = np.min(strip)

        P_k = 1000 * (strip_error_sum/(len(strip) * strip_min_error) - 1)

        return P_k
```

2.5 Esempio d'uso

Di seguito è riportato un esempio di utilizzo della libreria:

```
import random
import signal
import os

import matplotlib.pyplot as plt
import numpy as np

from NeuralNetworkLib.ErrorFunctions.CrossEntropyWithSoftMax import
CrossEntropyWithSoftMax
from NeuralNetworkLib.SimpleNetwork import SimpleNetwork
from NeuralNetworkLib.NetworkWithRPROP import NetworkWithRPROP
from NeuralNetworkLib.ActivationFunctions.Sigmoid import Sigmoid
from NeuralNetworkLib.DataLoader import DataLoader
from NeuralNetworkLib.Layers.FullyConnectedLayer import
FullyConnectedLayer

from NeuralNetworkLib.StoppingCriteria.GeneralizationLoss import
GeneralizationLoss
from NeuralNetworkLib.StoppingCriteria.ProgressQuotient import
ProgressQuotient

#Loading the dataset
dataset_path = os.path.normpath(os.path.join(os.getcwd(),
"../dataset/mnist/"))
data_loader = DataLoader(dataset_path, dataset_percentage=1,
training_set_percentage=0.75) #loads mnist, splitting it into 75%
training and 25% validation
data_loader.LoadDataset()


#setting up network parameters

#stop_criterion = GeneralizationLoss(alpha=0.1)
```

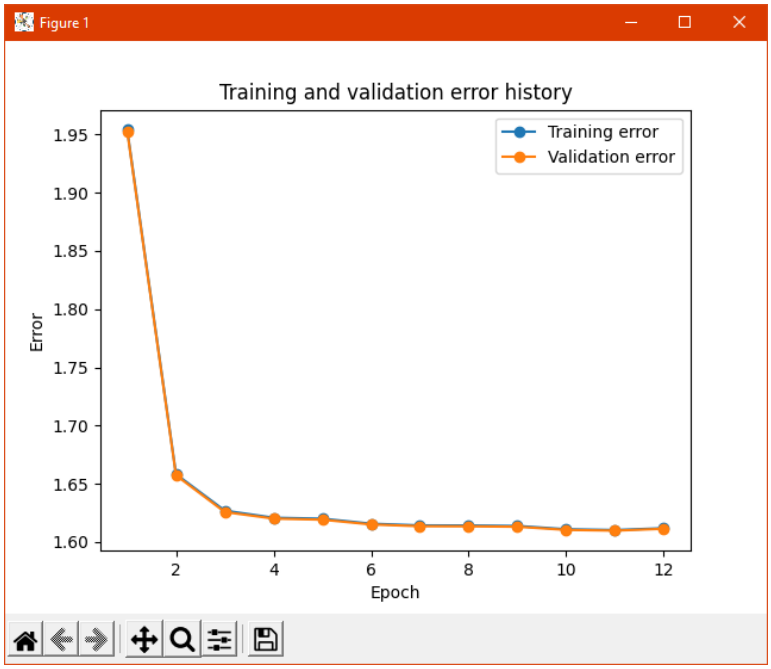
```
stop_criterion = ProgressQuotient(alpha=0.01, strip_length=5)

#net = SimpleNetwork(data_loader, CrossEntropyWithSoftMax,
learning_rate=0.15, stop_criterion=stop_criterion)
net = NetworkWithRPROP(data_loader, CrossEntropyWithSoftMax,
eta_pos=1.2, eta_neg=0.5, stop_criterion=stop_criterion)

number_of_nodes = 15
number_of_output_nodes = 10
input_size = len(data_loader.train_X[0])

net.add_layer(FullyConnectedLayer(input_size, number_of_nodes,
activation_function=Sigmoid)) #hidden layer
net.add_layer(FullyConnectedLayer(net.Layers[-1].number_of_nodes,
number_of_output_nodes, activation_function=Sigmoid)) #output layer

#HANDLE SIGINT
def ctrl_c_handler(signum, frame):
    """Hanldes SIGINT (CTRL+C). If detected the training process is
interrupted"""
    net.cancel_flag = True
signal.signal(signal.SIGINT, ctrl_c_handler)

#Start the training process
net.train(batch_size=1500, MAX_EPOCH=40)

#compute and print test accuracy
test_accuracy = net.compute_test_accuracy()
print(f"Test accuracy: {test_accuracy}")


#Plot training and validation error
time_axis = range(1, len(net.training_error_history) + 1)
plt.plot(time_axis, net.training_error_history, marker="o")
plt.plot(time_axis, net.validation_error_history, marker="o")

plt.title("Training and validation error history")
plt.xlabel("Epoch")
```

```
plt.ylabel("Error")
plt.legend(["Training error","Validation error"])
plt.show()
```

Lanciando questo esempio si ottiene in output:

```
Loading dataset...
Training set loaded: 45000 elements
Validation set loaded: 15000 elements
Test set loaded: 10000 elements
Epoch #1: training error: [1.95401765], validation error: [1.95283479]. Took 0:00:11.000700
Epoch #2: training error: [1.65823068], validation error: [1.65724721]. Took 0:00:10.544779
Epoch #3: training error: [1.62692494], validation error: [1.62575413]. Took 0:00:10.396886
Epoch #4: training error: [1.62084011], validation error: [1.62007746]. Took 0:00:10.504923
Epoch #5: training error: [1.6200069], validation error: [1.61916013]. Took 0:00:10.508748
Epoch #6: training error: [1.61571912], validation error: [1.61505272]. Took 0:00:10.488609
Epoch #7: training error: [1.61424592], validation error: [1.61355408]. Took 0:00:10.480631
Epoch #8: training error: [1.61416968], validation error: [1.61347266]. Took 0:00:10.853278
Epoch #9: training error: [1.61375374], validation error: [1.61308719]. Took 0:00:10.747869
Epoch #10: training error: [1.6109712], validation error: [1.61038483]. Took 0:00:10.710708
Epoch #11: training error: [1.61028962], validation error: [1.60973697]. Took 0:00:10.466748
Epoch #12: training error: [1.61184877], validation error: [1.61136304]. Took 0:00:10.545767
Stopping criterion met.
Training completed in 0:02:07.256652
Test accuracy: 0.891
```

A schermo viene mostrata la seguente finestra:



3. Esperimenti

3.1 Setup dell'ambiente

I risultati sono stati raccolti usando python versione `3.9.13` fatto girare su una macchina Windows con processore AMD Ryzen 3700X @4.225GHz e 16GB di RAM DDR4 3600MT/s CL16.

Ad ogni esecuzione è stato impostato il seed del random ad un valore fisso per ridurre la variabilità dei risultati:

```
random.seed(1)
```

I risultati sono stati ottenuti configurando una rete con le seguenti caratteristiche:

- Dimensioni dataset: 100% (diviso in 75% training e 25% validation)
    - Training set: 45000 elementi
    - Validation set: 15000 elementi
    - Test set: 10000 elementi
- Tipo di rete: NetworkWithRPROP
    - η^+ : 1.2
    - η^- : 0.5
- Funzione di errore: CrossEntropyWithSoftMax
- Criterio di stop: *variabile in base al test*
- Struttura della rete:
    - 1 hidden layer
        - Tipologia: fully connected
        - Dimensioni: *variabili in base al test*
        - Funzione di attivazione: Sigmoide
    - Output layer:
        - Tipologia: fully connected
        - Dimensioni: 10 neuroni (uno per ogni classe)
        - Funzione di attivazione: Sigmoide
- Parametri di training:
    - Batch size: 1500
    - Numero massimo di epoche: 40

Il software è stato lanciato senza debugger, per migliorare le prestazioni.

3.2 Presentazione risultati

Di seguito viene riportata la tabella riassuntiva dei risultati ottenuti, facendo variare la dimensione dell'hidden layer, il criterio di stop e il relativo parametro α.

Criterio di stop	Strip	Dimensione hidden layer	Alpha	# epoche	Tempo totale training	Accuratezza sul test	Rapporto performance/tempo
GL	N/A	10	0,01	6	0.00.44	0,8548	1,94
			0,04	20	0:02:28	0,835	0,56
			0,10	8	0:00:58	0,8686	1,50
		20	0,01	6	0:01:10	0,8987	1,28
			0,04	40	0:08:16	0,9105	0,18
		30	0,01	40	0:09:48	0,9276	0,16
		100	0,01	6	0:07:32	0,9448	0,21
PQ	5	10	0,01	10	0:01:13	0,8688	1,19
			0,04	18	0:02:11	0,8606	0,66
			0,10	33	0.04.04	0,8839	0,36
		20	0,01	8	0.01.34	0,9014	0,96
			0,04	8	0.01.32	0,9112	0,99
			0,10	16	0.03.05	0,9106	0,49
		30	0,01	8	0.01.58	0,9246	0,78
			0,04	14	0.03.27	0,9124	0,44
			0,10	16	0.03.52	0,915	0,39
		40	0,01	10	0.02.53	0,9209	0,53
			0,04	12	0.03.26	0,9237	0,45
			0,10	14	0.04.00	0,9226	0,38
		100	0,01	12	0.06.27	0,9387	0,24

Dove il rapporto performance su tempo (ptr) è così calcolato:

$$ptr = 100\frac{acc}{\Delta t}$$

Dove acc è l'accuratezza sul test set e Δt è il tempo totale di training.

3.2.1 Rete con 10 neuroni interni

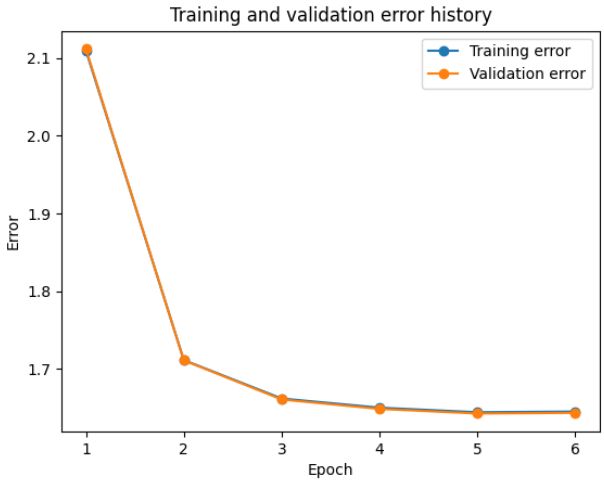


fig 3.1, GL α=0.01, 10 neuroni

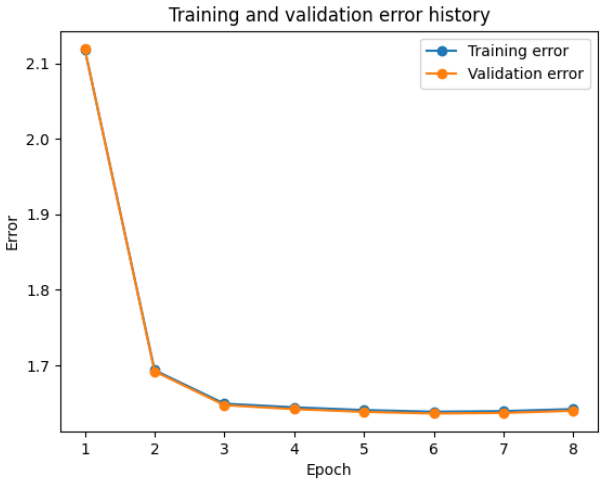


fig 3.2, GL α=0.04, 10 neuroni

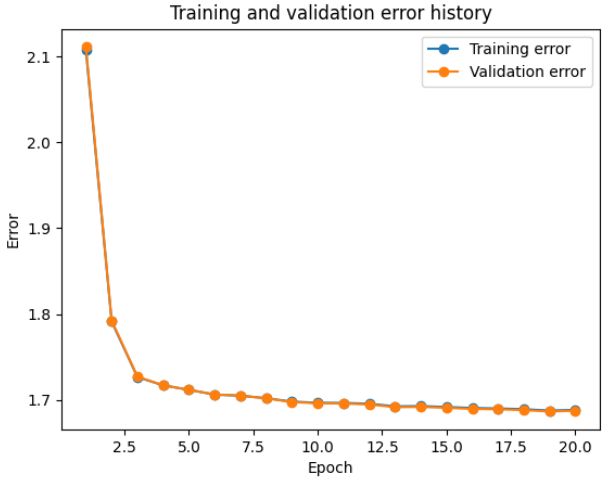


fig 3.2, GL α=0.1, 10 neuroni

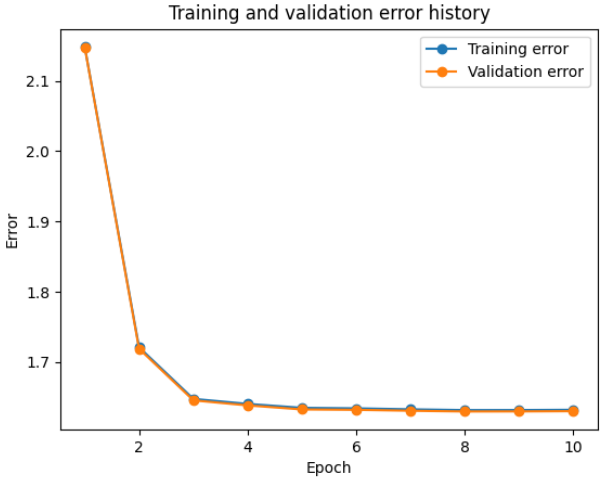


fig 3.3, PQ α=0.01, 10 neuroni

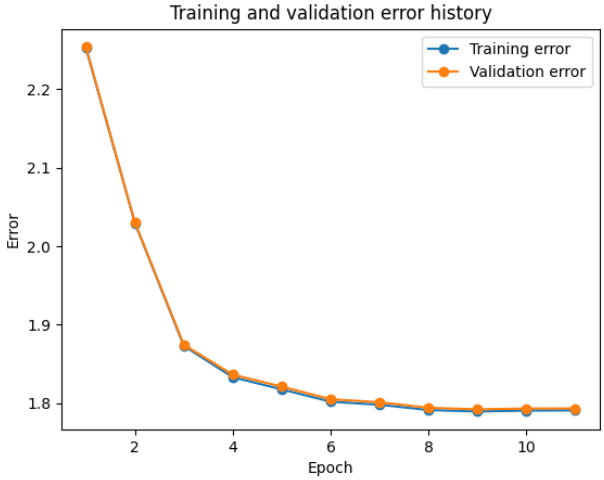


fig 3.3, PQ α=0.04, 10 neuroni

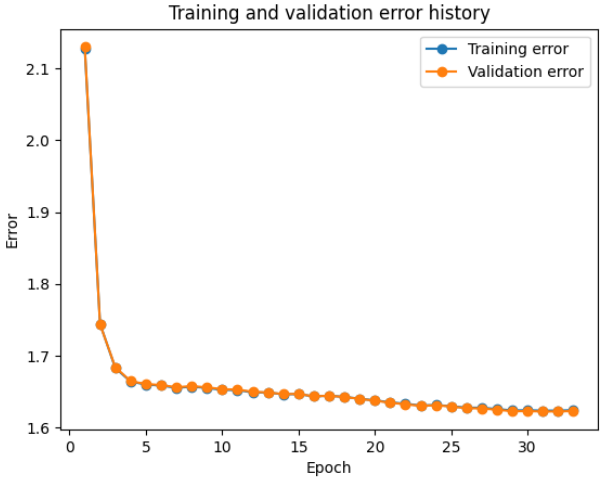


fig 3.3, PQ α=0.1, 10 neuroni

3.2.2 Rete con 20 neuroni interni

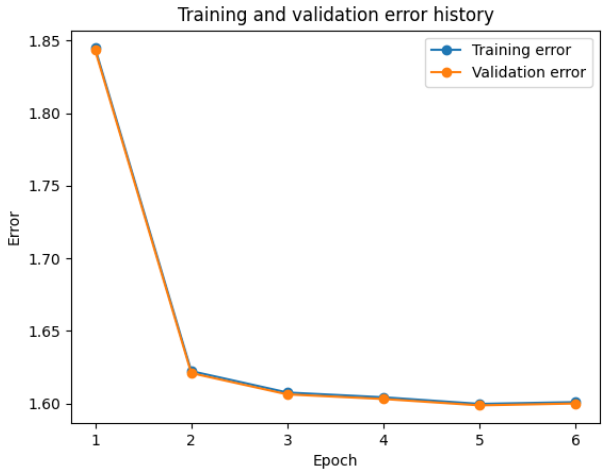


fig 3.4, GL α=0.01, 20 neuroni

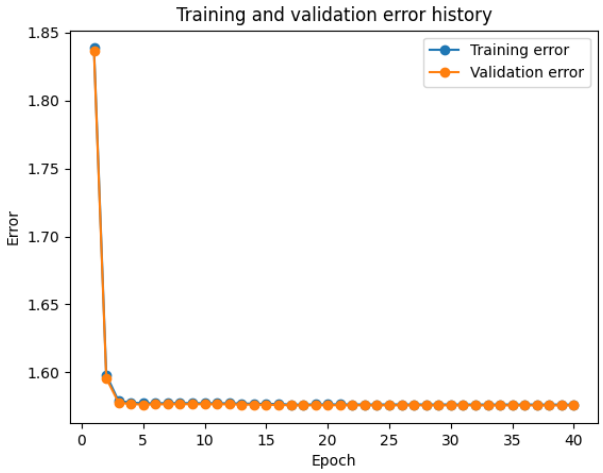


fig 3.5, GL α=0.04, 20 neuroni

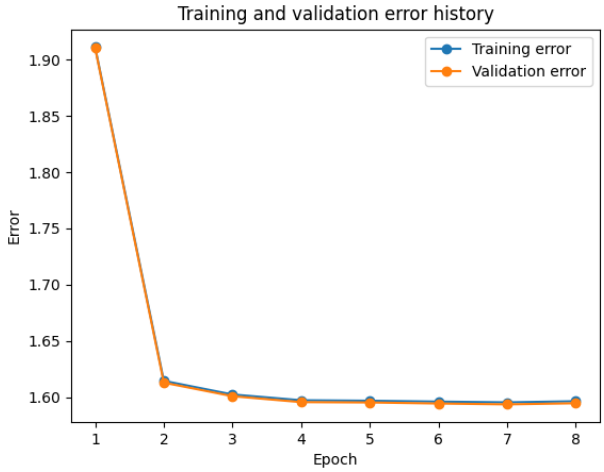


fig 3.6, PQ α=0.01, 20 neuroni

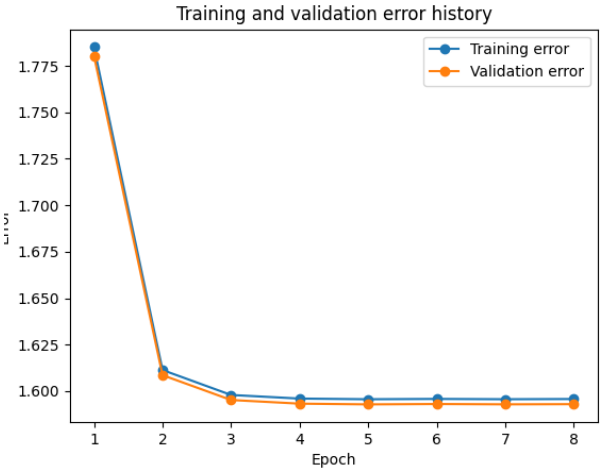


fig 3.7, PQ α=0.04, 20 neuroni

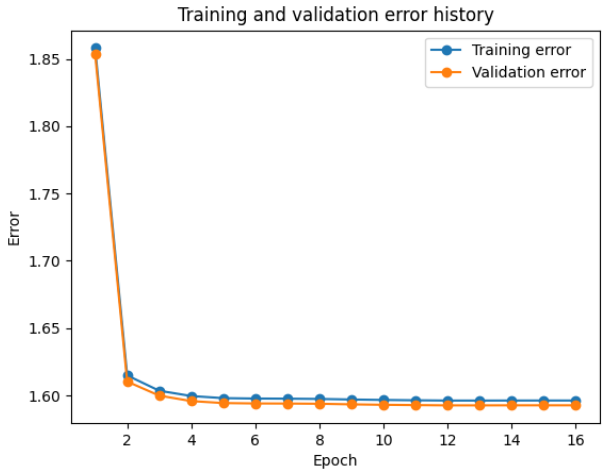


fig 3.8, PQ α=0.1, 20 neuroni

3.2.3 Rete con 30 neuroni interni

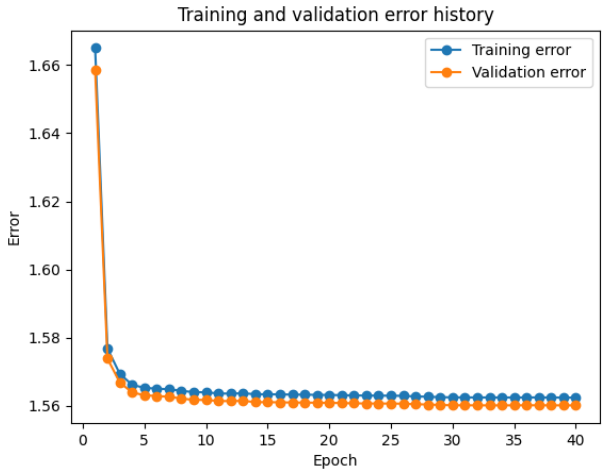


fig 3.9, GL α=0.01, 30 neuroni

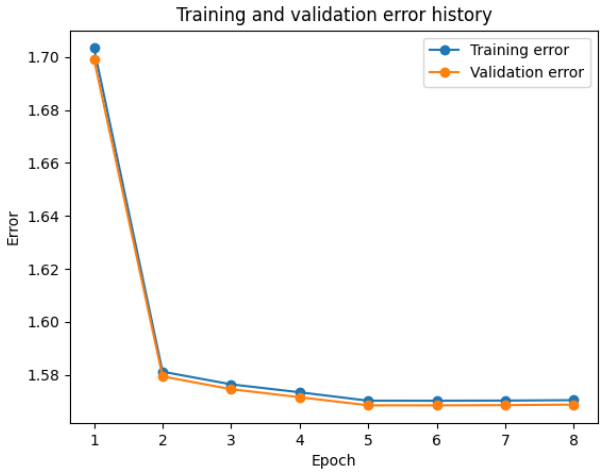


fig 3.10, PQ α=0.01, 30 neuroni

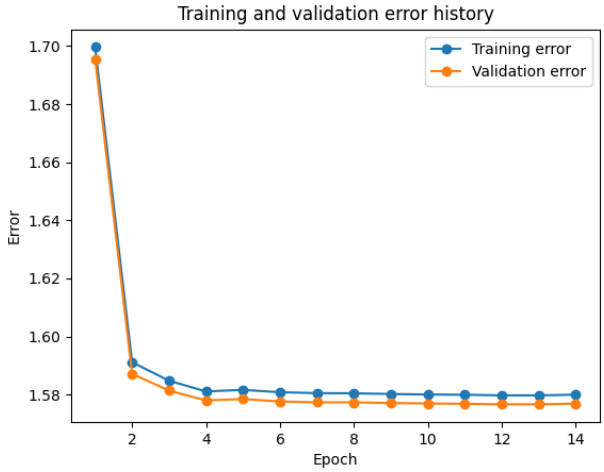


fig 3.11, PQ α=0.04, 30 neuroni

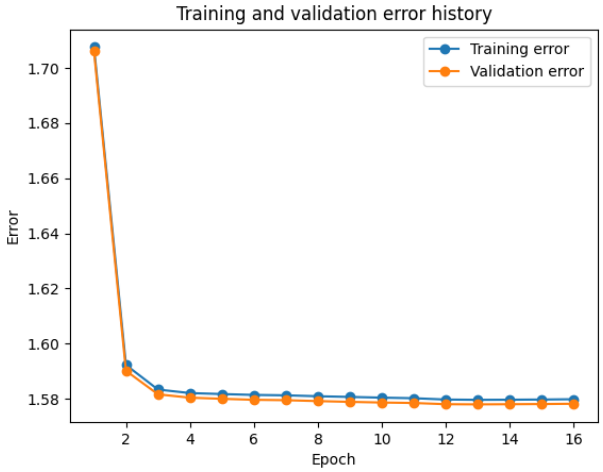


fig 3.12, PQ α=0.1, 30 neuroni

3.2.4 Rete con 40 neuroni interni

Per questioni di tempi computazionali la rete è stata testata unicamente con PQ

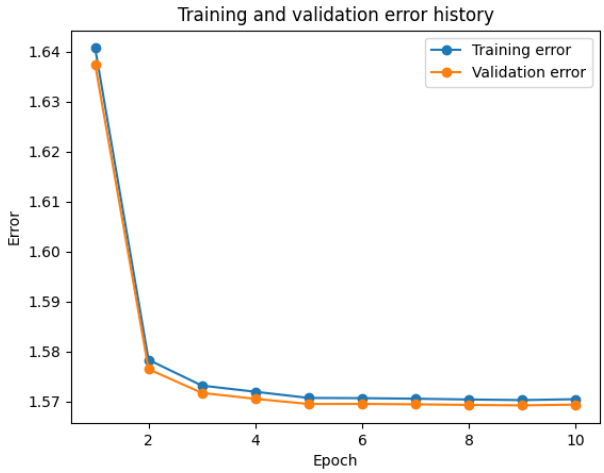


fig 3.13, PQ α=0.01, 40 neuroni

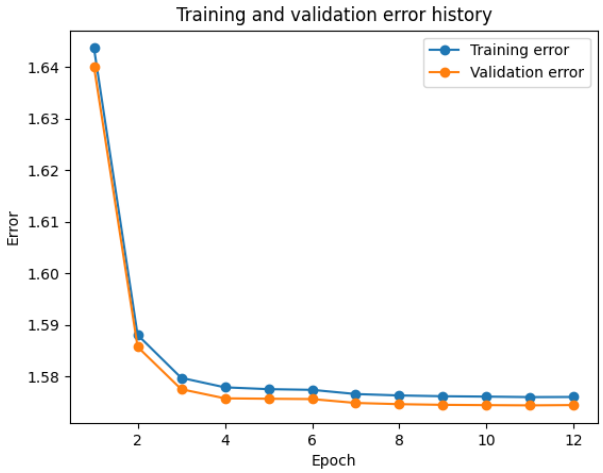


fig 3.14, PQ α=0.04, 40 neuroni

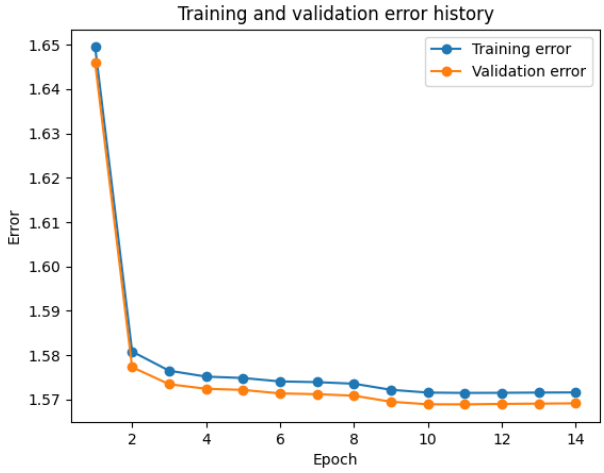


fig 3.15, PQ α=0.1, 40 neuroni

3.2.5 Rete con 100 neuroni interni

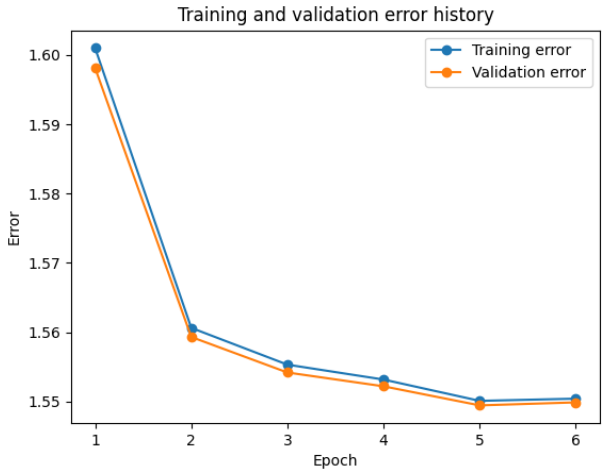


fig 3.16, GL α=0.01, 100 neuroni

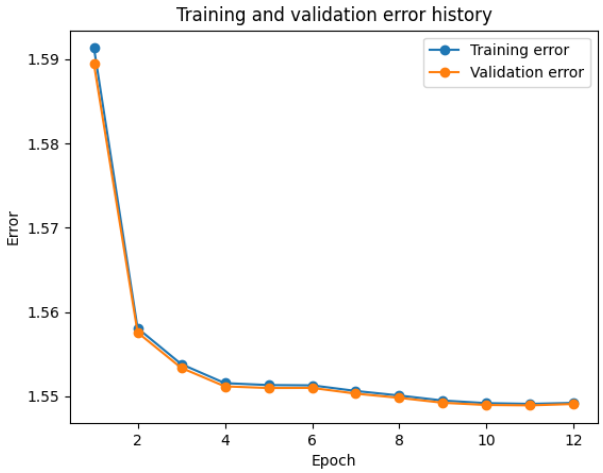


fig 3.16, PQ α=0.01, 100 neuroni

3.3 Commento sui risultati

La rete con soli 10 neuroni interni, allenata utilizzando il criterio GL con α=0.01 è stata quella con il miglior rapporto performance/tempo. Il processo di training, durato soli 44 secondi ha portato ad un'accuratezza sul test set dell'85%. Questa configurazione rappresenta dunque il miglior compromesso tra costi e risultati.

Se invece si limita la visione alle reti che hanno ottenuto un'accuratezza sul test set di almeno il 90%, tra queste la rete più efficiente, ovvero quella col miglior rapporto performance/tempo, è stata la rete composta da 20 neuroni interni, trainata utilizzando PQ con α = 0.04

Le prestazioni migliori in assoluto sono state registrate utilizzando una rete con 100 neuroni interni, il cui tempo di training è stato efficacemente limitato dal criterio di early stopping impostato su Generalization Loss con α = 0.01 . In questo casi si è misurata un'accuratezza sul test set pari al 94,48%.

Per via dell'inizializzazione stocastica dei pesi i risultati non sono del tutto deterministici.

In alcuni casi GL non è riuscita a interrompere prematuramente il learning, come nel caso con N= 20 neuroni, nel quale il training si è protratto fino al limite delle 40 epoche. Questo accade perché, finché il numero di neuroni non aumenta a dismisura, anche il rischio di overfitting rimane contenuto.

4. Conclusione e sviluppi futuri

Cosa abbiamo visto di bello, che problemi ci sono stati durante l'implementazione

Abbiamo confrontato le prestazioni di diversi criteri di early stopping in un problema di classificazione con un numero C di classi, con C = 10. È emerso come il criterio basato sulla generalization loss non sempre risulti efficace nel fermare il processo di learning al primo momento utile, protraendolo fino al numero massimo di epoche possibili.

Come possibili sviluppi futuri troviamo:

- salvataggio dei parametri
- scelta dei parametri col minimo errore sul validation set
- introduzione di altri tipi di layer (come gli strati convolutivi)
- training multithreaded
- esportazione delle informazioni di training come csv

Tramite il seguente link è possibile accedere al sorgente della libreria per visualizzarlo, copiarlo o proporre migliorie: https://github.com/imatrisciano/NeuralNetworkLib