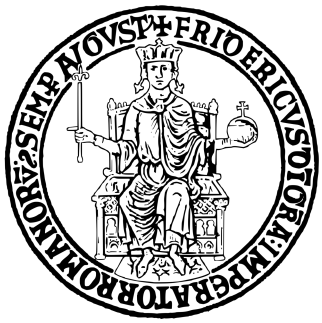
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE



CORSO DI LAUREA MAGISTRALE IN INFORMATICA

NEURAL NETWORKS AND DEEP LEARNING

ANNO ACCADEMICO 2023/2024

Progettazione e implementazione di una libreria per il supporto alla costruzione e all'uso di reti neurali feed forward multistrato full connected

Docenti

Prof. Roberto Prevete

Studenti

Francesco Jr. Iaccarino – N97000440

Fabiola Salomone – N97000457

Questa pagina è stata lasciata intenzionalmente bianca.

INDICE

1. INTRODUZIONE

1.1 Analisi del problema

L'obiettivo di tale progetto consiste nella classificazione di immagini monocromatiche (in bianco e nero) che raffigurano numeri scritti a mano. Per raggiungere tale scopo si adotta l'implementazione di una metodologia basata su una rete neurale *feed-forward multistrato full connected*.

Precisamente il progetto è diviso in due parti :

1. Parte A :

   - Progettazione ed implementazione di una libreria di funzioni per simulare la propagazione in avanti di una rete neurale multistrato *full connected*. Permettendo l'implementazione di 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 multistrato, per qualunque scelta della funzione di attivazione dei nodi della rete con la possibilità di usare almeno una *sum of square* o la *cross-entropy* con e senza *soft-max* come funzione di errore.

2. Parte B :

   - Si considera come input le immagini del dataset MNIST[1]. 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* e *test set* (considerando almeno 10000

[1] La base di dati MNIST è una vasta base di dati di cifre scritte a mano che è comunemente impiegata come insieme di addestramento in vari sistemi per l'elaborazione delle immagini: http://yann.lecun.com/exdb/mnist/

elementi per il *training set* e 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 uno solo strato di nodi interni **al variare del numero di nodi interni** (scegliendo almeno 5 dimensioni diverse) e con funzione di errore *cross-entropy* più *soft-max*. Scegliere e mantenere invariati tutti gli altri "iper-parametri" come, ad esempio, le funzioni di attivazione ed i parametri della RProp.

1.2 Struttura della documentazione

La documentazione sarà composta nel seguente modo :

- Nel capitolo 1 "**Introduzione**" , forniremo una breve introduzione che definirà il problema trattato nel contesto dell'implementazione di una rete neurale per la classificazione delle immagini del dataset *MNIST*.

- Il capitolo 2 "**Parte A**", sarà dedicato alla descrizione delle funzionalità implementate nella Parte A del progetto. Analizzeremo le soluzioni adottate per raggiungere gli obiettivi prefissati, concentrandoci sull'architettura e sulle operazioni fondamentali della rete neurale.

- Nel capitolo 3 "**Parte B**", esploreremo le funzionalità aggiuntive implementate nella Parte B del progetto, con particolare attenzione all'algoritmo di aggiornamento dei pesi RProp (Resilient BackPropagation).

- Nel capitolo 4 "**Test e Analisi del risultati**", presenteremo i risultati ottenuti attraverso test e analisi.

1.3 Struttura della libreria

La libreria è composta da tre differenti moduli che sono :

- ***main.py*** : contiene l'implementazione di una rete neurale *feed forward* multistrato *full-connected* , con un solo strato nascosto, per la classificazione del *dataset MNIST* utilizzando l'algoritmo di aggiornamento dei pesi RProp (Resilient Backpropagation).

- ***functions.py*** : contiene tutte le funzioni che occorrono per implementare da zero una rete neurale *feed forwa*rd multistrato *full-connected*.

- ***loader.py*** : contiene il codice per effettuare il caricamento di un qualunque *dataset* (in tal caso è implementato il codice per il caricamento del *dataset MNIST*).

Ed un pacchetto, chiamato "*mnist*" che rappresenta il *dataset MNIST*.

2 PARTE A

In questo capitolo, viene illustrata la realizzazione della rete neurale e delle funzionalità offerte dalla libreria realizzata. La libreria è stata implementata in linguaggio python 3 e offre la possibilità di creare una rete neurale multistrato con qualsiasi numero di nodi per ogni strato. Durante la creazione della rete è data la possibilità di scegliere una qualsiasi funzione di attivazione per ciascun strato, ed è data anche la possibilità di scegliere una funzione di errore arbitraria. Inoltre, la libreria in oggetto, offre funzioni che permettono di effettuare la fase di apprendimento e di valutazione della rete neurale.

2.1 Descrizione dei metodi : "functions.py"

2.1.1 Costruttore — __init__

Il costruttore consente di creare e inizializzare un oggetto che rappresenta una rete neurale.

```
1.  def __init__(self, neurons_per_layer):
2.      self.layers = self.inizialize_numebr_of_neurons_per_layers(np.array(neurons_per_layer))
3.      self.dict_activation_functions = self.set_dict_activation_functions()
4.      self.dict_error_functions = self.set_dict_error_functions()
```

In cui abbiamo le chiamate :

```
1. def  inizialize_numebr_of_neurons_per_layers(self, sizes):
2.       #Inizializza il numero di neuroni in ciascun strato
3.       layers = {}
4.       for i in range(sizes.shape[0]):
5.           layers["layer" + str(i)] = sizes[i]
6.       return layers
```

```
1. def get_number_of_layers(self):
2.       #Restituisce il numero di strati nella rete neurale
3.       return len(self.layers)-1
```

```
1. def  set_dict_activation_functions(self):
2.      #Definisce un dizionario contenente tutte le funzioni di attivazione e le loro derivate
3.      return {
4.          "tanh" : [self.tanh, self.derivative_tanh],
5.          "relu" : [self.relu, self.derivative_relu],
6.          "leaky_relu": [self.leaky_relu, self.derivative_leaky_relu],
7.          "sigmoid": [self.sigmoid, self.derivative_sigmoid],
8.          "identity": [self.identity, self.derivative_identity]
9.       }
```

```
1. def  set_dict_error_functions(self):
2.      #Definisce un dizionario contenente le funzioni di errore e le loro derivate
```

```
3.      return{
4.        "cross_entropy": [self.cross_entropy, self.derivative_cross_entropy],
5.        "sum_of_squares": [self.sum_of_squares, self.derivative_sum_of_squares]
6.      }
```

Il metodo "*__init__*" prende un solo parametro in input :

- ***neurons_per_layer*** : una lista/vettore contenente come primo valore la dimensione dell'input e come restanti valori il numero di neuroni per ogni *layer* della rete neurale.

Una volta ricevuto in input tale parametro il metodo provvede alla creazione di diversi dizionari :

- ***dict_layer*** : dizionario che presenta come chiave una stringa rappresentante il livello della rete neurale e come valore, un intero, che indica il numero di nodi. Ad esempio, supponiamo che la lista "*neurons_per_layer*" sia "[10,20,5] dove 10 è la dimensione dell'input, 20 è il numero di neuroni nel primo layer e 5 è il numero di neuroni nel secondo *layer*. La creazione del dizionario "*dict_layers*" potrebbe assomigliare a qualcosa del genere :

  dict_layers : {

  'input_size' : 10,

  'layers1' : 20,

  'layers2' : 5 }

- ***dict_activation_functions*** : dizionario che presenta come chiave una stringa indicante il nome della funzione di attivazione e come valore una lista contenente in posizione zero il puntatore alla corretta funzione di attivazione e in posizione uno il puntatore alla corretta funzione di attivazione derivata. Ad esempio, supponiamo di aver un *layer* di input, uno o più layer nascosti e un layer di *output*. Ogni *layer* ha associata una funzione di attivazione come "*identity*", "*relu*", "*sigmoide*", "*softmax*". Ad esempio, la creazione del dizionario "*dict_activation_functions*" potrebbe assomigliare a qualcosa del genere :

dict_activation_functions = {

‘input_layer’ : ‘identity’,

‘hidden_layers’ : [‘relu’, sigmoid’],

‘output_layer’ : ‘softmax’ }

- ***dict_error_functions*** : dizionario che presenta come chiave una stringa indicante il nome della funzione di errore e come valore una lista contenente in posizione zero il puntatore alla corretta funzione di errore e in posizione uno il puntatore alla corretta funzione di errore derivata. Ad esempio supponiamo di voler considerare due tipi di funzioni di errore che sono la “*cross_entropy*” con la sua derivata e la “*sum_of_squares*” con la sua derivata. La creazione del dizionario “*dict_error_functions*” potrebbe assomigliare a qualcosa del genere :

dict_error_functions = {

‘cross_entropy’ : [‘cross_entropy’, ‘derivate_cross_entropy’].

‘sum_of_squares’ : [‘sum_of_squares’, ‘derivate_ sum_of_squares’]

}

2.1.2 Metodo — set_activation_functions

Il metodo “*set_activation_functions*” permette a chi utilizza la libreria in questione di specificare la funzione di attivazione che si vuole utilizzare per ogni *layer* della rete neurale.

```
1. def   set_activation_functions(self, activation_function):
2.      #Imposta le funzioni di attivazione
3.       self.activation_functions = activation_function
```

Va notato che in tal caso è necessario che la funzione sia chiamata prima di addestrare la rete, pena un errore che termina l’esecuzione.

Il metodo “*set_activation_functions*” prende un solo parametro in input :

- ***activation_function*** : una lista/vettore contenente i nomi, espressi come stringhe, delle funzioni di attivazione scelte per ogni *layer*. All'interno della lista l'ordine delle stringhe è cruciale in quanto riflette la corrispondenza con i vari *layer*. In altre parole, il primo elemento della lista rappresenta la funzione di attivazione associata al primo *layer*, il secondo elemento al secondo *layer*, e così via. Questa convenzione dell'ordine è di fondamentale importanza per garantire una corretta configurazione delle funzioni di attivazione nei diversi strati della rete neurale.

In sintesi, tale metodo si occupa di memorizzare il parametro passato in un attributo di classe, utilizzando lo stesso nome, così da poterlo utilizzare in futuro all'interno della classe.

2.1.3 Metodo — set_error_functions

Il metodo "*set_error_functions*" permette a chi utilizza la libreria in questione di specificare la funzione di errore che si vuole utilizzare e se si vuole impiegare la funzione *softmax* come *post-processing*.

```
1. def set_error_function(self, error_function, softmax_post_processing = True):
2.         self.error_function = error_function
3.         self.softmax_post = softmax_post_processing
```

Va notato che in tal caso è necessario che tale funzione sia chiamata prima di addestrare la rete, pena un errore che termina l'esecuzione.

Il metodo "*set_error_functions*" prende due parametri in input :

- ***error_function*** : una stringa che rappresenta la funzione di errore che si è scelti di utilizzare;

- ***softmax_post_processing*** : un booleano con cui si indica se si vuole impiegare o meno la funzione *softmax* come *post processing*. Tuttavia, di *default* tale parametro è impostato a "*True*"

Tecnicamente tale metodo si occupa di memorizzare il parametro passato in un attributo di classe, utilizzando lo stesso nome, così da poterlo utilizzare in futuro all'interno della classe.

Si sottolinea che l'opzione "*softmax_post_ processing*" avrà effetto solo nel caso in cui si scelga di utilizzare la *cross-entropy* come funzione di errore. In altre parole, la decisione di attivare o disattivare la funzione *softmax* come passaggio successivo (*post-processing*) influenzerà il comportamento solo quando la funzione di errore selezionata è la *cross-entropy*. Altrimenti, questa opzione potrebbe non avere rilevanza o non essere considerata.

2.1.4 Metodo — eprint

Il metodo "*eprint*" è progettato e implementato per stampare messaggi d'errore sullo "*standard error*" e terminare l'esecuzione del programma.

```
1. def eprint(self, *args, **kwargs):
2.     # Funzione di stampa di errore e uscita dal programma
3.     print(*args, file=sys.stderr, **kwargs)
4.     sys.exit()
```

Il metodo "*eprint*" prende due parametri in input :

- ***args*** : un parametro che raccoglie tutti gli argomenti posizionali passati alla funzione sotto forma di tupla. Può essere utilizzato per accettare un numero variabile di argomenti posizionali;

- ***kwargs*** : un parametro che raccoglie tutti gli argomenti con nome (chiave-valore) passati alla funzione sottoforma di dizionario. Può essere utilizzato per accettare un numero variabile di argomenti con nome.

2.1.5 Metodo — check_iper_parameters

Il metodo "*check_iper_parameters*" permette di verificare la validità degli iper-parametri della rete neurale fornendo eventualmente messaggi di errore e terminando l'esecuzione del programma se viene rilevato un problema.

```
1. def check_iper_parameters(self):
2.         #Controllo degli iper-parametri
3.         if len(self.layers) < 3:
4.             self.eprint("Number of layers have to be at least 3")
5.         for nodes in self.layers.values():
6.             if nodes < 1:
7.                 self.eprint("The minimum number of neurons for each layer is 1")
8.         if not hasattr(self, "activation_functions"):
9.             self.eprint("You have to pass activation functions")
10.        if not hasattr(self, "error_function"):
11.            self.eprint("You have to pass error function")
12.        if self.error_function not in self.dict_error_functions.keys():
13.            self.eprint("Error function can be only cross-entropy or sum-of-squares")
14.        if len(self.activation_functions) != self.get_number_of_layers():
15.            self.eprint("Number of activation function different from number of layer")
16.        for af in self.activation_functions:
17.            if(af not in self.dict_activation_functions.keys()):
18.                self.eprint("Activation function not available")
19.        if self.error_function == "cross_entropy":
20.            if not hasattr(self, "softmax_post"):
21.                self.eprint("You have to set post-processing true or false")
22.            if self.softmax_post != True and self.softmax_post != False:
23.                self.eprint("Softmax post-processing can to be only True or False")
24.            if self.softmax_post == True and self.layers["layer" + str(self.get_number_of_layers())] ==
1:
25.                self.eprint("When have 1 node on output layer you can't use softmax")
26.            if self.softmax_post == True and self.activation_functions[self.get_number_of_layers()-1]
!= "identity":
27.                self.eprint("When softmax post-processing is True the activation function of the last
layer have to be identity")
28.        elif self.error_function == "sum-of-squares":
29.            if self.neurons_per_layer[self.get_number_of_layers()] != 1:
30.                self.eprint("When use sum-of-squares you have to be only 1 node as output layer")
```

Tecnicamente, il metodo "*check_iper_parameters*" accetta *input* impliciti attraverso '*self*', che rappresenta l'istanza della classe ed esegue controlli specifici sugli iper-parametri. Restituisce messaggi di errore e termina l'esecuzione del programma se uno qualsiasi dei controlli fallisce.

Tale metodo aiuta a garantire che la rete neurale sia configurata correttamente prima dell'addestramento.

2.1.6 Metodo — initialize_weights_and_biases

Il metodo "*initialize_weights_and_biases*" si occupa dell'inizializzazione dei pesi e dei *bias*.

```
1. def initialize_weights_and_biases(self):
2.     # Inizializza i pesi e i bias della rete neurale
3.     parameters = {}
4.
5.     for i in range(len(self.layers)-1):
6.         np.random.seed(15)
7.         # Inizializza i pesi con una distribuzione casuale
8.         parameters["W"+str(i+1)] = np.random.randn(self.layers.get("layer"+str(i+1)),
self.layers.get("layer"+str(i))) * np.sqrt(1./self.layers.get("layer"+str(i)))
9.         # Inizializza i bias a zero
10.         parameters["b"+str(i+1)] = np.zeros((self.layers.get("layer"+str(i+1)), 1))
11.
12.     return parameters
```

Tale metodo non prende nulla in input ma restituisce in output il dizionario contenente i pesi e i *bias* inizializzati per tutti gli strati della rete neurale (eccetto l'ultimo).

2.1.7 Metodo — one_hot

Il metodo "*one_hot*" permette di convertire le etichette di classe in una rappresentazione *one-hot*.

```
1. def one_hot(self, Y):
2.       one_hot_Y = np.zeros((Y.size, Y.max() + 1))
3.       one_hot_Y[np.arange(Y.size), Y] = 1
4.       one_hot_Y = one_hot_Y.T
5.       return one_hot_Y
```

Il metodo "*one_hot*" prende un solo parametro in input :

- ***Y*** : rappresenta l'array delle etichette di classe. Si presume che le etichette siano numeriche e che ogni numero rappresenti una classe diversa.

Tuttavia, il metodo "*one_hot*" restituisce in output un solo parametro:

- ***One_hot_Y*** : che è la rappresentazione one-hot delle etichette di classe. È in particolare si presenta come una matrice dove ogni riga rappresenta un campione e ogni colonna corrisponde a una classe. Se un campione appartiene a una classe, il valore corrispondente nella colonna della classe sarà 1, altrimenti sarà 0.

2.1.8 Metodo — train

Il metodo "*train*" permette a colui che utilizza la libreria in questione di eseguire l'addestramento della rete.

```
1. def  train(self, X_train, Y_train, X_val, Y_val, type_update_parameter, learning_rate=0.02, etaPlus=1.2, etaMinus=0.5, epochs=50):
2.
3.        parameters={}
4.        self.check_iper_parameters()
5.
6.        y_one_hot = self.one_hot(Y_train)
7.        parameters = self.initialize_weights_and_biases()
8.        error_function = self.dict_error_functions.get(self.error_function)[0]
9.
10.        self.train_errors = []
11.        self.validation_errors = []
12.
13.        if(type_update_parameter == "rprop"):
14.            best_parameters = self.train_rprop(X_train, Y_train, X_val, Y_val, epochs, etaPlus, etaMinus, parameters, y_one_hot, error_function)
15.        elif(type_update_parameter == "standard"):
16.            best_parameters = self.train_standard(X_train, Y_train, epochs, learning_rate, X_val, Y_val, parameters, y_one_hot, error_function)
17.        else:
18.            self.eprint("The type of update parameter have to be standard or rprop")
19.
20.        self.plot_error_on_epochs()
21.
22.        return best_parameters
```

Il metodo "*train*" prende nove parametri in input :

- **X_train** : rappresenta il dataset di valori utilizzati per addestrare la rete;
- **Y_train** : rappresenta il dataset di target utilizzati per addestrare la rete;
- **X_val** : rappresenta il dataset di valori utilizzati per valutare la rete;
- **Y_val** : rappresenta il dataset di target utilizzati per valutare la rete;
- **epochs** : intero che rappresenta il numero massimo di iterazioni utilizzate per addestrare la rete;

- **learning_rate** : double che rappresenta la velocità di addestramento della rete;
- **etaPlus** : double che rappresenta il valore di etaPlus per la RPROP;
- **etaMinus** : double che rappresenta il valore di etaMinus per la RPROP;
- **type_update_parameter** : stringa che può assumere solo due valori " *standard*" e "*rprop*" specificando quindi se di vuole utilizzare un aggiornamento dei pesi standard oppure tramite RProp.

Il metodo in questione effettua verifiche tramite la funzione "*check_iper_parametrs*" assicurandosi che tutti i parametri passati in *input* rispettino i vincoli impostati e che l'oggetto di tipo "*NeuralNetwork*" che chiama il metodo "*train*" abbia effettuato le dovute chiamate ai metodi "*set_activation_functions*", "*set_error_function*".

Una volta che sono stati effettuati tali controlli viene chiamato il metodo "*inizialize_weights_and_biases*", il quale restituisce un dizionario chiamato "*parameters*" contenente come chiavi :

- "W_i" : stringa che rappresenta la matrice di pesi tra i livelli i-1 ed il livello i-esimo ;
- "b_i" : stringa che rappresenta il vettore del bias del livello i-esimo.

Da notare che i valori del dizionario "*parameters*" sono matrici quando si considera W_i mentre sono vettori quando si considera b_i. Ad esempio :

parameters : {
"W_i" : [[0, 0, …., 0],
[0, 1, …., 0],
…….,
[1, 0, ..…., 0]],
"b_i" : [[0, 8, 5, …., 1],
}
con $1 \leq i \leq$ *numero layer*

Invocate le suddette funzioni, il metodo "*train*" provvede a chiamare *"train_standard"* oppure *"train_rprop"* in base al valore del parametro "*type_update_parameter*", passando a tali metodi tutti i parametri che esso stesso riceve come parametri di input. In particolare, il parametro "*learning_rate*" viene utilizzato solo per la funzione "train_standard", mentre i parametri "*etaPlus*" ed "*etaMinus*" vengono utilizzati solo per la funzione "*train_rprop*".

2.1.8.1 Metodo - train_standard

Il metodo "*train_standard*" viene utilizzato per addestrare la rete neurale utilizzando l'algoritmo della discesa del gradiente standard.

```
1. def train_standard(self, X_train, Y_train, epochs, learning_rate, X_val, Y_val, parameters,
Y_train_one_hot, error_function):
2.
3.        #Allenamento della rete neurale utilizzando il gradiente discendente standard
4.        for i in range(epochs):
5.
6.            forward_cache = self.forward_propagation(X_train, parameters)
7.
8.            gradients = self.backward_prop(X_train, Y_train_one_hot, parameters, forward_cache)
9.
10.           parameters = self.update_parameters(parameters, gradients, learning_rate)
11.
12.
13.           #Validation check
14.           forward_cache_val_set = self.forward_propagation(X_val, parameters)
15.           val_err = error_function(forward_cache_val_set["output"], Y_val.T)
16.
17.           self.train_errors.append(error_function(forward_cache["output"], Y_train.T))
18.           self.validation_errors.append(val_err)
19.
20.           if i == 0:
21.               min_err = val_err
22.               best_parameters = parameters
23.               self.best_epoch = i
24.
25.           if val_err < min_err:
26.               min_err = val_err
27.               best_parameters = parameters
28.               self.best_epoch = i
29.
30.           #End validation check
31.
32.           if(i%(epochs/10) == 0):
33.               print("Accuracy of Train Dataset after", i, "iterations: ", self.accuracy(X_train,
Y_train_one_hot, parameters), "%")
34.               print("Loss of Train Dataset after", i, "iterations: ",
error_function(forward_cache["output"], Y_train.T))
35.
36.       return best_parameters
```

Tecnicamente il metodo "*train_standard*" guida l'addestramento della rete neurale attraverso iterazioni facendo uso di "*forward propagation*" e "*backword propagation*", inoltre calcola il gradiente, aggiorna i parametri come pesi e i *bias* tramite il metodo *"update_parameters"* , monitora le metriche di addestramento e si occupa della validazione. In fine restituisce i parametri che minimizzano l'errore sulla validazione.

2.1.9 Metodo — forward_propagation

Nella *forward propagation*, l'output di un neurone viene trasmetto come *input* ad un neurone dello strato successivo. Dal momento che la rete neurale è *full-connected*, *l'output* di ogni neurone dello strato i viene trasmesso a tutti i neuroni dello strato i+1.

```
1. def forward_propagation(self, x, parameters):
2.         #Propagazione in avanti attraverso la rete neurale
3.         forward_value = {}
4.
5.         for i in range(self.get_number_of_layers()):
6.             activation_function = self.dict_activation_functions.get(self.activation_functions[i])[0]
7.             if i == 0:
8.                 #Calcolo dell'output del primo strato nascosto
9.                 forward_value["a"+str(i+1)] = np.dot(parameters.get("W"+str(i+1)), x) +
parameters.get("b"+str(i+1))
10.                 forward_value["z"+str(i+1)] = activation_function(forward_value.get("a"+str(i+1)))
11.             elif i == self.get_number_of_layers()-1:
12.                 #Calcolo dell'output del layer di output
13.                 forward_value["a"+str(i+1)] = np.dot(parameters.get("W"+str(i+1)),
forward_value.get("z"+str(i))) + parameters.get("b"+str(i+1))
14.                 forward_value["z"+str(i+1)] = activation_function(forward_value.get("a"+str(i+1)))
15.                 if self.softmax_post == True:
16.                     #Applica la post-elaborazione softmax se necessario
17.                     forward_value["output"] = self.post_processing(forward_value.get("z"+str(i+1)))
18.                 else:
19.                     forward_value["output"] = forward_value["z"+str(i+1)]
20.             else:
21.                 #Calcolo dell'output degli strati nascosti intermedi
22.                 forward_value["a"+str(i+1)] = np.dot(parameters.get("W"+str(i+1)),
forward_value.get("z"+str(i))) + parameters.get("b"+str(i+1))
23.                 forward_value["z"+str(i+1)] = activation_function(forward_value.get("a"+str(i+1)))
24.
25.         return forward_value
26.
```

Il metodo *forward propagation* prende in *input* due parametri :

- ***x*** : insieme di dati da cui partire per la propagazione in avanti ;

- ***parameters*** : dizionario contenente matrici dei pesi e vettori del *bias.*

L'implementazione, tecnicamente, prevede un ciclo su tutti gli strati, in cui per ogni strato vengono calcolati la funzione d'attivazione e *l'output* attraverso la notazione matriciale.

Tali quantità vengono calcolate nel seguente modo :

$$a_i^h = \sum_{j=1}^{m_{h-1}} w_{ij}^h \cdot z_j^{h-1} + b_i^h$$

$$z_i^h = f_h(a_i^h)$$

Dove :

- **i** : indica l'i-esimo nodo;
- f : indica la funzione d'attivazione ;
- **h** : indica il *layer* corrente.

Al termine dell'esecuzione, la funzione "*fordword_propagation*" ritorna un dizionario "*forward_value*" che ha come chiavi :

- a_i : stringa che rappresenta la matrice dell'attivazione del livello i-esimo;
- z_i : stringa che rappresenta la matrice dell'output del livello i-esimo.

I valori del dizionario "*forward_propagation*" sono matrici sia quando si tratta di a_i sia quando si tratta di z_i.

2.1.10 Metodo — backward_propagation

Il metodo "*backward_propagation***"** è una tecnica per il calcolo della derivata della funzione di errore rispetto ai pesi.

Tale metodo prende in input quattro parametri :

- ***X*** : rappresenta il dataset utilizzato per calcolare la derivata della funzione d'errore all'ultima iterazione;

- ***Y*** : rappresenta il dataset contenente i target utilizzato per calcolare la derivata della funzione d'errore alla prima iterazione ;

- ***Parameters*** : dizionario contenente matrici dei pesi e vettori dei *bias* ;

- ***Forward_value*** : dizionario contenente il risultato della forward propagation.

Per la back-propagation si ha che :

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_{ij}}} = \sum_{\mathrm{n=1}}^{\mathrm{N}} \frac{\partial \mathrm{E^{(n)}}}{\partial \mathrm{w_{ij}}} \qquad \frac{\partial \mathrm{E^{(n)}}}{\partial \mathrm{w_{ij}}} = \frac{\partial \mathrm{E^{(n)}}}{\partial \mathrm{a_i^n}} \cdot \frac{\partial \mathrm{a_i^n}}{\partial \mathrm{w_{ij}}}$$

Dove :

$$\frac{\partial a_i^n}{\partial \mathrm{w_{ij}}} = \mathrm{z_j^n} \qquad \delta_\mathrm{i}^\mathrm{n} \equiv \frac{\partial \mathrm{E}}{\partial a_i^n}$$

Quindi :

$$\frac{\partial \mathrm{E^{(n)}}}{\partial \mathrm{w_{ij}}} = \delta_\mathrm{i}^\mathrm{n} \cdot \mathrm{z_j^n}$$

Per il calcolo di $\delta_\mathrm{i}^\mathrm{n}$, si ha che :

- Neuroni di output :

$$\delta_\mathrm{k}^\mathrm{n} = \frac{\partial \mathrm{E^{(n)}}}{\partial a_k^n} = g'(a_k^n) \cdot \frac{\partial \mathrm{E^{(n)}}}{\partial y_k^n}$$

- Neuroni interni :

$$\delta_{\mathrm{i}}^{\mathrm{n}} = \frac{\partial \mathrm{E}^{(\mathrm{n})}}{\partial a_i^n} = f'(a_i^n) \cdot \sum_k w_{ki}\ \delta_{\mathrm{k}}^{\mathrm{n}}$$

Si noti la presenza di una formula ricorsiva tramite la quale a partire dai $\delta_{\mathrm{i}}^{\mathrm{n}}$ dei nodi "più esterni" (quelli di *output*) possiamo "risalire" ai $\delta_{\mathrm{i}}^{\mathrm{n}}$ di tutti i nodi interni.

La funzione "**backward_prop**" restituirà come output un dizionario "delta" avente come chiavi le stringhe "dzi", "dwi" e "dbi" ($1 \leq i \leq numero\ layer$) e avente come valori :

- **dzi** : Matrice dei delta calcolati per lo strato i-esimo;
- **dwi** : Matrice dei gradienti dei neuroni per lo strato i-esimo;
- **dbi** : Vettore dei gradienti del bias per lo strato i-esimo.

2.1.11 Metodo — update_parameters

Tale metodo è utilizzato per effettuare l'aggiornamento delle matrici di pesi e vettori di *bias* in seguito ad una forward propagation e una backward propagation.

Il metodo "*update_parameters*" prende in input tre parametri :

- **parameters** : dizionario contenente le matrici dei pesi e i vettori di bias da aggiornare;

- **gradients** : dizionario contenente le matrici in cui sono presenti le derivate della funzione d'errore rispetto ai pesi;

- **learning_rate** : velocità di aggiornamento dei pesi.

L'aggiornamento viene effettuata nel seguente modo :

$$w_{ij} \leftarrow w_{ij} - \eta\left(\frac{dE^{(n)}}{dw_{ij}}\right)$$

Ciò che ritorna tale funzione è il dizionario "*parameters*" preso in ingresso, con le medesime chiavi ma con i valori aggiornati.

2.1.12 Funzioni di attivazione e loro derivate

Di seguito sono riportate le funzioni di attivazione utilizzate nella libreria e le relative derivate.

2.1.12.1 Identità

La funzione identità, denominata identity, è una funzione in cui per ogni valore di un input x, la funzione identità restituisce lo stesso valore x. La rappresentazione matematica della funzione d'identità è spesso indicata con :

$$f(x) = x$$

In Python, in particolare nella libreria in esame, l'implementazione della funzione d'identità è la seguente :

```
1. def  identity(self, x):
2.     return x
```

La funzione identità è dotata anche di derivata. La rappresentazione matematica della derivata della funzione d'identità è indicata con :

$$f'(x) = \frac{d}{dx}(x) = 1$$

In Python, in particolare nella libreria in esame, l'implementazione della derivata della funzione d'identità è indicata come segue :

```
1. def  derivative_identity(self,x):
2.         return 1
```

2.1.12.2 Sigmoide

La funzione sigmoide è una funzione che mappa qualsiasi valore reale in un intervallo compreso tra 0 e 1. La rappresentazione matematica della funzione sigmoide è spesso indicata con :

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

In Python, in particolare nella libreria in esame, l'implementazione della funzione sigmoide è la seguente :

```
1. def  sigmoid(self,x):
2.         return np.where(x<0, np.exp(x)/(1.0 + np.exp(x)), 1.0/(1.0 + np.exp(-x)))
```

La funzione sigmoide è dotata anche di derivata. La rappresentazione matematica della derivata della funzione sigmoide è indicata con :

$$\frac{d\sigma(x)}{dx} = \sigma(x) \cdot \big(1 - \sigma(x)\big)$$

In Python, in particolare nella libreria in esame, l'implementazione della derivata della funzione sigmoide è indicata con :

```
1.  def  derivative_sigmoid(self,x):
2.          return self.sigmoid(x) * (1- self.sigmoid(x))
```

2.1.12.3 Tangente iperbolica

La funzione tangente iperbolica (tanh) è una funzione che mappa ogni valore reale in un intervallo compreso tra -1 e 1. La rappresentazione matematica della funzione della tangente iperbolica è spesso indicata con :

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

In Python, in particolare nella libreria in esame, l'implementazione della funzione tangente iperbolica è la seguente :

```
1.def  tanh(self, x):
2.    return np.tanh(x)
```

La funzione tangente iperbolica è dotata anche di derivata. La rappresentazione matematica della derivata della funzione tangente iperbolica è indicata con :

$$tanh'(x) \ = \ 1 - \ tanh^2(x)$$

In Python, in particolare nella libreria in esame, l'implementazione della derivata della funzione della tangente iperbolica è indicata con :

```
1. def  derivative_tanh(self, x):
2.      return (1 - np.power(np.tanh(x), 2))
```

2.1.12.4 ReLU

La funzione ReLU (Rectified Linear Units) ha la seguente rappresentazione matematica :

$$ReLU(x) = \begin{cases} x & se\ x > 0 \\ 0 & se\ altrimenti \end{cases} \quad \Leftrightarrow \quad ReLU(x) = max(x, 0)$$

In altre parole, la ReLU restituisce x se x è maggiore di zero e zero altrimenti. La funzione ha un comportamento lineare per valori positivi e si annulla per valori negativi.

In Python, in particolare nella libreria in esame, l'implementazione della funzione ReLU è la seguente :

```
1. def relu(self, x):
2.      return  np.maximum(x, 0)
```

La funzione ReLU è dotata anche di derivata. La rappresentazione matematica della derivata della funzione ReLU è indicata con :

$$ReLU'(x) = \begin{cases} 1 & se\ x > 0 \\ 0 & se\ altrimenti \end{cases}$$

In Python, in particolare nella libreria in esame, l'implementazione della derivata della funzione ReLU è indicata con :

```
1. def  derivative_relu(self, x):
2.       return np.where(x>0, 1, 0)
```

2.1.12.5 Leaky-ReLU

La funzione Leaky ReLU è una variante della funzione ReLU, ha la seguente rappresentazione matematica :

$$L - ReLU(x) = \begin{cases} x & se\ x > 0 \\ \alpha \cdot x & se\ altrimenti \end{cases}$$

La differenza principale rispetto alla funzione di attivazione ReLU standard è la gestione dei valori negativi. Nella funzione di attivazione ReLU standard, ogni valore di input inferiore o uguale a zero produce un'uscita lineare e zero. Questo può causare il problema noto come "*dying ReLU*", in cui alcuni neuroni diventano inattivi durante l'addestramento.

La Leaky ReLU risolve questo problema permettendo una pendenza piccola ma non nulla per i valori negativi.

Notiamo che nella formula matematica α è un piccolo coefficiente di pendenza positiva, spesso impostato a un valore come 0.01 ed inoltre non viene appreso, a differenza di altre funzioni di attivazione, durante l'allenamento della rete neurale ma è impostato prima dell'allenamento e rimane costante durante tutto il processo di addestramento.

Con ciò, ossia con l'introduzione di questa pendenza si evita il problema dei neuroni inattivi durante l'allenamento, specialmente con valori di input negativi.

In Python, in particolare nella libreria in esame, l'implementazione della funzione Leaky ReLU è la seguente :

```
1.def  leaky_relu(self, x):
2.   return np.where(x>0, x, 0.01*x)
```

La funzione Leaky ReLU è dotata anche di derivata. La rappresentazione matematica della derivata della funzione ReLU è indicata con :

$$L - ReLU'(x) = \begin{cases} 1 \; se \; x > 0 \\ \alpha \; altrimenti \end{cases}$$

In Python, in particolare nella libreria in esame, l'implementazione della derivata della funzione Leaky ReLU è indicata con :

```
1. def   derivative_leaky_relu(self, x):
2.     return np.where(x>0, 1, 0.01)
```

2.1.13 Funzioni di errore

Di seguito sono riportate le funzioni di errore utilizzate nella libreria in esame e le relative derivate.

2.1.13.1 Cross-Entropy

La funzione Cross-Entropy viene utilizzata in problemi di classificazione ed è definita matematicamente dalla seguente formula :

$$cross\_entropy(y,t) \; = \; - \sum_{n=1}^{N} \sum_{k=1}^{c} (t_k^n \cdot \log(\, y_k^n))$$

In cui “y” rappresenta le probabilità predette dalla rete neurale per ciascuna delle “c” classi e “t” rappresenta le etichette reali. La somma è effettuata su N esempi.

In Python, in particolare nella libreria in esame, l’implementazione della funzione d’errore “cross-entropy” è la seguente :

```
1. def   cross_entropy(self, y_pred, y_true):
2.     y_true_one_hot_vec = (y_true[:,np.newaxis] == np.arange(10))
3.     loss_sample = (np.log(y_pred.T, where=y_pred.T!=0) * y_true_one_hot_vec).sum(axis=1)
4.     return -loss_sample.sum(axis=0)
```

La funzione d’errore “cross_entropy” è dotata anche di derivata. In Python, in particolare nella libreria in esame, l’implementazione della derivata della funzione d’errore “cross_entropy” è indicata con :

```
1. def   derivative_cross_entropy(self, out, t):
2.     if(self.softmax_post == False):
3.         return (out - t)/(out*(1 - out))
4.     else:
5.         return (out - t)
```

2.1.13.2 Cross-Entropy con softmax

La funzione *cross-entropy* con *softmax* consiste nell’applicare uno step di normalizzazione agli output della rete, tramite la *softmax*, in modo da interpretare i valori come probabilità essendo compresi tra 0 ed 1.

Tale funzione è utilizzata come funzione di perdita durante l’addestramento della rete neurale per problemi di classificazione. L’obiettivo di tale funzione è quello di minimizzare questa perdita per migliorare le prestazioni della rete neurale.

La formula matematica per la *cross-entropy* con *softmax* è :

$$cross\_entropy\_con\_softmax(y,t) = -\sum_{n=1}^{N}\sum_{k=1}^{c}(t_k^n \cdot \log(softmax(y_k^n)))$$

In cui :

- $\boldsymbol{N}$ *:* è il numero totale di esempi nel dataset ;
- $\boldsymbol{c}$ *:* è il numero totale di classi ;
- $\boldsymbol{y}$ **:** rappresenta gli output predetti dalla rete neurale ;
- $\boldsymbol{t}$: rappresenta le etichette one-hot-encoded delle classi reali ;
- $\boldsymbol{y_k^n}$: rappresenta l'output predetto per la classe k dell'esempio n;
- $\boldsymbol{softmax}(\boldsymbol{y_k^n})$ applica la funzione *softmax* all'output per ottenere probabilità normalizzate per ogni classe.

2.1.13.**3** Sum-of-squares

La funzione *sum-of-squares* è utilizzata come funzione di errore in problemi di regressione. Si occupa di calcolare la distanza tra i valori y predetti dalla rete e i valori t "corretti".

La formula matematica per la *sum-of-squares* è :

$$sum\_of\_squares(y,t) = -\sum_{n=1}^{N}\sum_{k=1}^{c}(y_k^n - t_k^n)^2$$

In Python, in particolare nella libreria in esame, l'implementazione della funzione d'errore "*sum-of-squares*" è la seguente :

```
1. def sum_of_squares(self, out, t):
2.   return (np.sum((out - t)**2))/2
```

La funzione d'errore "*sum_of_squares*" è dotata anche di derivata. In Python, in particolare nella libreria in esame, l'implementazione della derivata della funzione d'errore "*sum_of_squares*" è indicata con :

```
1. def derivative_sum_of_squares(self, out, t):
2.     return (out - t)
```

2.1.14 Post-processing

2.1.14.**1** Soft-max

La funzione *softmax* è una funzione che trasforma le componenti di un vettore in probabilità. Essa, è calcolata nel seguente modo:

$$soft - max(a_i) = \frac{e^{a_i}}{\sum_{h=1}^{m_l} e^{a_h}}$$

Dove :

- **e** è la costante di Nepero (approssimativamente 2.71828)
- $\boldsymbol{a_i}$ è l'elemento i-esimo del vettore a
- $\boldsymbol{m_l}$ è la lunghezza del vettore a

Tuttavia, va osservato che quando si lavora con valori numerici elevati, l'applicazione della *softmax standard* può causare *overflow* numerico, portando a risultati indesiderati e instabilità numerica.

Per risolvere tale problema, è stata implementata, nella libreria in esame, una variante della *softmax*. Tale variante della *softmax* è ottenuta sottraendo il massimo valore dall'intero vettore prima di applicare la *softmax*. In questo modo, manteniamo la relativa probabilità tra gli elementi, ma garantiamo che gli esponenti rimandano entro limiti gestibili, evitando così l'overflow numerico.

L'implementazione di questa variante della softmax è stata realizzata attraverso le seguenti funzioni :

```
1. def  softmax(self, x):
2.       for i in range(x.shape[1]):
3.           x[:, i] = x[:, i] - np.max(x[:, i])
4.
5.       expX = np.exp(x)
6.       return expX/np.sum(expX, axis = 0)
```

```
1. def   post_processing(self, z):
```

```
2.    return self.softmax(z)
```

Questa versione della softmax è stata incorporata nella libreria per garantire stabilità numerica durante gli esperimenti, specialmente in situazioni in cui potrebbero verificarsi valori numerici elevati. La sottrazione del massimo valore assicura che gli esponenti siano compresi nell'intervallo $(-\infty,0)$, riducendo il rischio di problemi numerici.

2.2 Descrizione dei metodi : "loader.py"

Il codice è organizzato in metodi e funzioni presenti nella classe "MNISTLoader" descritta nella seguente sottosezione.

2.2.1 Costruttore — __init__

Il costruttore consente di creare e inizializzare un'istanza del caricatore MNIST con il percorso dei dati

```
1. def   __init__(self, data_path):
2.     # Inizializza l'istanza del caricatore MNIST con il percorso dei dati
3.     self._mnist_data = mnist.MNIST(data_path)
4.     # Carica i dati MNIST
5.     self._load_data()
```

Il metodo "__init__" prende un solo parametro in input :

- **data_path** : rappresenta il percorso dei dati MNIST

2.2.2 Metodo — __load__data

Il metodo *"__**load__data**"* permette di caricare i dati e le etichette dal set di addestramento *MNIST*, trasponendo i dati per avere un formato più comodo (feature come colonne) e codificando le etichette in formato *one-hot*.

```
1. def   _load_data(self):
2.       # Carica dati e etichette dal set di addestramento MNIST
3.       data, labels = self._mnist_data.load_training()
```

```
4.      # Trasponi i dati per avere un formato più comodo (feature come colonne)
5.      data = np.array(data).T
6.      # Codifica le etichette in formato one-hot
7.      labels = self.encode(np.array(labels))
8.      # Assegna i dati e le etichette all'istanza della classe
9.      self.data = data
10.      self.labels = labels
```

2.2.3 Metodo — encode

Il metodo "*encode*" effettua la codifica delle etichette in un formato variabile a seconda delle esigenze.

```
1. @staticmethod
2.     def encode(labels: np.ndarray) -> np.ndarray:
3.         # Codifica le etichette in formato one-hot
4.         encoded_labels = np.zeros(shape=(10, labels.shape[0]))
5.         for n in range(labels.shape[0]):
6.             encoded_labels[labels[n]][n] = 1
7.         return encoded_labels
```

Il metodo "*encode*" prende un solo parametro in input :

- **labels** : rappresenta l'insieme delle etichette da codificare

Ciò che restituisce è l'insieme di etichette codificate.

2.2.4 Metodo — decode

Il metodo "*decode*" effettua la decodifica delle etichette in formato originale.

```
1. @staticmethod
2.    def decode(labels: np.ndarray) -> np.ndarray:
3.       # Decodifica le etichette one-hot restituendo gli indici degli elementi massimi
4.       return np.array([
5.          np.argmax(labels[:, n]) for n in range(labels.shape[1])
6.       ])
```

Il metodo "*decode*" prende un solo parametro in input :

- **labels** : rappresenta l'insieme delle etichette da decodificare

Ciò che restituisce è l'insieme di etichette decodificate.

2.3 Pacchetto “mnist”

La classe MNIST, la quale estende la classe *Loader*, rappresenta il dataset MNIST che è una vasta base di dati di cifre scritte a mano che è comunemente impiegata come set di addestramento in vari sistemi per l’elaborazione delle immagini. I metodi implementati rispettano l’interfaccia definita dalla classe *Loader*.

3 PARTE B

In questo capitolo, vengono discusse ulteriori funzionalità della libreria. Nello specifico viene presentata la resilient backpropagation. Viene inoltre discussa la selezione di un modello di rete neurale a singolo strato, sulla base degli iper-parametri dell'RPROP e del numero di nodi interni tramite la descrizione del modulo "main.py".

3.1 Descrizione dei metodi : "functions.py"

3.1.1 Metodo – rprop

La *Resilient Back-Propagation* (RPROP) è una regola di aggiornamento che rallenta l'apprendimento se sta andando troppo veloce e viceversa, tale regola può essere utilizzata solamente in modalità batch.

L'implementazione di RPROP avviene mediante la medesima funzione "rprop" :

```
 1. def rprop(self,etaP,etaM,dict_matrix_prec_deltas,dict_matrix_prec_grad,
dict_array_prec_bias_deltas, dict_array_prec_bias_grad,weights,gradients):
 2.         #Aggiorna i pesi e i bias utilizzando l'algoritmo RPROP
 3.         etaPlus = etaP
 4.         etaMinus = etaM
 5.         delta = 0.0
 6.         deltaMax = 50.0
 7.         deltaMin = 1e-6
 8.
 9.         for l in range(self.get_number_of_layers()):
10.
11.             #Aggiornamento di W
12.             matrix_prec_grad = dict_matrix_prec_grad[l+1]
13.             matrix_prec_deltas = dict_matrix_prec_deltas[l+1]
14.             matrix_act_grad = gradients["dw"+str(l+1)]
15.             for i in range(matrix_act_grad.shape[0]):
16.                 for j in range(matrix_act_grad.shape[1]):
17.                     if(matrix_act_grad[i][j] * matrix_prec_grad[i][j] >= 0):
18.                         delta = matrix_prec_deltas[i][j] * etaPlus
19.                         if(delta > deltaMax): delta = deltaMax
20.                     elif(matrix_act_grad[i][j] * matrix_prec_grad[i][j] < 0):
21.                         delta = matrix_prec_deltas[i][j] * etaMinus
22.                         if(delta < deltaMin):
23.                             delta = deltaMin
24.
25.                     weights["W"+str(l+1)][i][j] = weights["W"+str(l+1)][i][j] -
(np.sign(matrix_act_grad[i][j])*delta)
```

```
26.                 matrix_prec_grad[i][j] = matrix_act_grad[i][j]
27.                 matrix_prec_deltas[i][j] = delta
28.
29.         #Aggiornamento di b
30.         matrix_prec_bias_grad = dict_array_prec_bias_grad[l+1]
31.         matrix_prec_bias_deltas = dict_array_prec_bias_deltas[l+1]
32.         matrix_act_bias_grad = gradients["db"+str(l+1)]
33.         for i in range(matrix_act_bias_grad.shape[0]):
34.             if(matrix_act_bias_grad[i][0] * matrix_prec_bias_grad[i][0] >= 0):
35.                 delta = matrix_prec_bias_deltas[i][0] * etaPlus
36.                 if(delta > deltaMax): delta = deltaMax
37.             elif(matrix_act_bias_grad[i][0] * matrix_prec_bias_grad[i][0] < 0):
38.                 delta = matrix_prec_bias_deltas[i][0] * etaMinus
39.                 if(delta < deltaMin):
40.                     delta = deltaMin
41.             weights["b"+str(l+1)][i][0] = weights["b"+str(l+1)][i][0] -
(np.sign(matrix_act_bias_grad[i][0])*delta)
42.             matrix_prec_bias_grad[i][0] = matrix_act_bias_grad[i][0]
43.             matrix_prec_bias_deltas[i][0] = delta
44.
45.     return weights
```

Tale funzione prende in input sei parametri :

- **dict_matrix_prec_deltas** : dizionario contenente come chiave una stringa indicante il layer i-esimo e come valore la matrice dei delta calcolati all'iterazione precedente (escluso il *bias*) ;

- **dict_matrix_prec_grad** : dizionario contenente come chiave una stringa indicante il layer i-esimo e come valore la matrice dei gradienti calcolati all'iterazione precedente (escluso il *bias*) ;

- **dict_array_prec_biad_deltas** : dizionario contenente come chiave una stringa indicante il *layer* i-esimo e come valore il vettore dei delta del *bias* calcolati all'iterazione precedente ;

- **dict_array_prec_bias_grad** : dizionario contenente come chiave una stringa indicante il *layer* i-esimo e come valore il vettore dei gradienti del *bias* calcolati all'iterazione precedente ;

- **weights** : dizionario contenente come chiave una stringa "W_i" o "b_i" e come valore la matrice dei pesi o il vettore del *bias* ;

- **gradients** : dizionario contenente come chiave una stringa "dzi", "dwi", "dbi" e come valore la matrice del delta, la matrice dei gradienti rispetto ai pesi e il vettore dei gradienti rispetto al *bias*.

La nuova regola di aggiornamento è :

$$w_{ij} = w_{ij} - sign\left(\frac{\partial \mathrm{E}}{\partial w_{ij}}\right)\Delta_{ij}$$

Per adattare Δ_{ij}, si introducono due iper-parametri $\eta^+ > 1$ e $0 < \eta^- > 1$. Pertanto, si ha che :

$$\text{incremento} \Longrightarrow \Delta_{ij} = \eta^+ \cdot \Delta_{ij} \qquad \text{se } \left(\frac{\partial E^{(e)}}{\partial w_{ij}}\right) \cdot \left(\frac{\partial E^{(e+1)}}{\partial w_{ij}}\right) > 0$$

$$\text{decremento} \Longrightarrow \Delta_{ij} = \eta^- \cdot \Delta_{ij} \qquad \text{se } \left(\frac{\partial E^{(e)}}{\partial w_{ij}}\right) \cdot \left(\frac{\partial E^{(e+1)}}{\partial w_{ij}}\right) < 0$$

Alla prima epoca, i Δ_{ij} sono inizializzati tutti ad un valore di 0.01, talvolta, vengono forniti come utleriori iper-parametri Δ_{max} e Δ_{min} che indicano i valori limite che può assumere Δ_{ij} , Δ_{max} e Δ_{min} sono fissati rispettivamente a 50.0 ed e^{-6}.

3.1.2 Metodo – train_rprop

Il metodo "*train_standard*" viene utilizzato per addestrare la rete neurale utilizzando l'algoritmo RPROP.

```
1. def train_rprop(self, X_train, Y_train, X_val, Y_val, epochs, etaPlus, etaMinus, parameters,
Y_train_one_hot, error_function):
2.       #Addrestramento della rete neurale utilizzando l'algoritmo RPROP
3.
```

```
    dict_matrix_prec_deltas, dict_matrix_prec_grad, dict_array_prec_bias_deltas, dict_array_prec_bias_grad = self.initialize_rprop_matrix()
    best_parameters = {}

    for i in range(epochs):
        forward_cache = self.forward_propagation(X_train, parameters)

        gradients = self.backward_prop(X_train, Y_train_one_hot, parameters, forward_cache)

        parameters = self.rprop(etaPlus,etaMinus,dict_matrix_prec_deltas,dict_matrix_prec_grad, dict_array_prec_bias_deltas,dict_array_prec_bias_grad,parameters,gradients)

        #Validation check
        forward_cache_val_set = self.forward_propagation(X_val, parameters)
        val_err = error_function(forward_cache_val_set["output"], Y_val.T)

        if val_err > 0.0 and error_function(forward_cache["output"], Y_train.T) > 0.0:
            self.train_errors.append(error_function(forward_cache["output"], Y_train.T))
            self.validation_errors.append(val_err)

        if i == 0:
            min_err = val_err
            for j in parameters.keys():
                best_parameters[j] = np.copy(parameters.get(j))
                best_parameters[j] = np.copy(parameters.get(j))
            self.best_epoch = i

        if val_err < min_err and val_err > 0.0:
            min_err = val_err
            self.best_epoch = i
            for j in parameters.keys():
                best_parameters[j] = np.copy(parameters.get(j))
                best_parameters[j] = np.copy(parameters.get(j))

        #End validation check

        if(i%(epochs/10) == 0):
            print("Accuracy of Train Dataset after", i, "iterations: ", self.accuracy(X_train, Y_train_one_hot, parameters), "%")
            print("Loss of Train Dataset after", i, "iterations: ", error_function(forward_cache["output"], Y_train.T))

    return best_parameters
```

Tecnicamente il metodo "*train_rprop*" guida l'addestramento della rete neurale attraverso iterazioni facendo uso di "*forward propagation*" e "*backward propagation*", inoltre calcola gradiente, aggiorna i parametri come pesi e del *bias* tramite il metodo *"rprop"* e monitora le metriche di addestramento e validazione. In fine restituisce i parametri che minimizzano l'errore sulla validazione.

3.1.3 Metodo – accuracy

Il metodo "*accuracy*" permette di calcolare l'accuratezza della rete neurale rispetto ai dati di input e alle etichette fornire.

```
1.  def accuracy(self, inp, labels, parameters):
2.      forward_cache = self.forward_propagation(inp, parameters)
3.      a_out = forward_cache['output']
4.      a_out = np.argmax(a_out, 0)
5.      labels = np.argmax(labels, 0)
6.      acc = np.mean(a_out == labels)*100
7.
8.      return acc
```

Il metodo "*accuracy*" prende in input tre parametri :

- **inp** : dati di input per i quali si vuole calcolare l'accuratezza ;
- **labels** : etichette di ground truth corrispondenti ai dati di input ;
- **parameters** : i parametri allineati della rete neurale, ottenuti durante il processo di addestramento.

Mentre restituisce in output :

- **accuracy** : l'accuratezza della rete neurale calcolata come percentuale di predizione corrette rispetto alle etichette reali.

Tecnicamente l'accuratezza è calcolata confrontando le etichette predette con le etichette reali e calcolando la percentuale di predizione corrette rispetto al numero totale di campioni.

3.1.4 Metodo – plot_error_on_epochs

Il metodo "*plot_error_on_epochs*" è implementata e progettata per visualizzare un grafico dell'andamento degli errori durante l'addestramento di una rete neurale

```
1.  def   plot_error_on_epochs(self):
2.    pyplot.plot(self.train_errors, label="Training set", color = "red")
3.    pyplot.plot(self.validation_errors, label="Validation set", color = "blue")
4.    ax = pl.gca()
5.     ylim = ax.get_ylim()
6.     pyplot.vlines(self.best_epoch, ylim[0], ylim[1], label="Minimo",
```

```
color="green")
7.    pyplot.xlabel("Epoche")
8.    pyplot.ylabel("Errore")
9.    pyplot.legend()
10.   pyplot.show()
```

Dove :

- Linea rossa : rappresenta la curva degli errori sul set di allenamento ;
- Linea blu : rappresenta la curva degli errori sul set di validazione;
- Linea verde : rappresenta l'epoca in cui si è ottenuto il minimo errore sul set di validazione.

3.2 Descrizione del modulo : "main.py"

Il modulo "*main.py*" va a implementare una rete neurale *feed-forward* multistrato *full-connected*, con un solo strato interno, per l'addestramento su un dataset MNIST.

```
1. import numpy as np
2. from loader import MNISTLoader
3. from functions import NeuralNetwork
4.
5. # Caricamento il dataset MNIST
6. data_path = "/Users/Bia/Desktop/mnist"  #cambiare a seconda del percorso in cui si trova la cartella mnist
7. mnist_loader = MNISTLoader(data_path)
8.
9. # Caricamento il dataset
10. X, y_onehot = mnist_loader.data.T, mnist_loader.labels.T
11.
12. # Normalizzazione delle immagini
13. X = X / 255.0
14.
15. # Impostazione le dimensioni del training set e del test set  ( va distribuito bene e deve essere minore di 60000 la somma train + validation)
16. train_size = 40000
17. test_size = 8000
18. validation_size = 12000
19.
20. # Dividisione del dataset in training e test set
21. X_train, X_valid, X_test = X[:train_size], X[train_size:train_size+validation_size], X[train_size+validation_size:]
22. y_train, y_valid, y_test = y_onehot[:train_size], y_onehot[train_size:train_size+validation_size], y_onehot[train_size+validation_size:]
23.
24.
25. y_train = np.array(y_train, dtype=np.int64)
26. y_valid = np.array(y_valid, dtype=np.int64)
27. y_test = np.array(y_test, dtype=np.int64)
28.
```

```
29. # Lista dei numeri di nodi interni da testare
30. hidden_sizes = [128]
31.
32. # Impostazione degli iperparametri
33. epochs = 40
34.
37.
38. # Creazione di un'istanza della classe NeuralNetwork
39. neural_net = NeuralNetwork(neurons_per_layer=[X_train.shape[1]] + hidden_sizes +
[y_train.shape[1]])
40.
41. # Impostazione delle funzioni di attivazione e di errore
42. activation_functions = ["relu", "identity"]
43. neural_net.set_activation_functions(activation_functions)
44. neural_net.set_error_function("cross_entropy", softmax_post_processing=True)
45.
46. # Addestramento della rete con RProp
47. best_parameters = neural_net.train(X_train.T, np.argmax(y_train, axis=1), X_valid.T,
np.argmax(y_valid, axis=1), type_update_parameter="rprop",
48.                                   epochs = epochs)
49.
50. # Propagazione in avanti sulla rete per il test set
51. forward_test = neural_net.forward_propagation(X_test.T, best_parameters)
52.
53. print("Dimensione del set di test:", y_test.shape[0])
54. print("Dimensione del validation set:", y_valid.shape[0])
55. print("Dimensione del trainging set:", y_train.shape[0])
56.
57.
58. # Calcolo dell'accuratezza sul test set
59. y_one_hot = neural_net.one_hot(np.argmax(y_train, axis=1))
60. test_accuracy = neural_net.accuracy(X_test.T, neural_net.one_hot(np.argmax(y_test, axis=1)),
best_parameters)
61. print("Test Accuracy: {:.2f}%".format(test_accuracy))
62.
63.
64. # Calcolo della loss sul test set
65. test_loss = neural_net.cross_entropy(forward_test["output"], np.argmax(y_test, axis=1))
66. print("Loss on Validation Dataset:", test_loss)
```

Ecco una panoramica del codice e del suo flusso il cui obiettivo è quello di ottenere un modello che possa classificare correttamente le cifre scritte a mano nel *dataset MNIST* :

Iniziamo importando le librerie necessarie che sono "*NumPy*" per le operazioni matematiche efficienti e due classi personalizzate, "*MNISTLoader*" per il caricamento del *dataset MNIST* e "*NeuralNetwork*" fornita dalla libreria "*functions*"

Successivamente, si fa uso di una classe chiamata "MNISTLoader" per importare il dataset *MNIST*, una collezione di immagini di cifre scritte a mano. Queste immagini

vengono quindi normalizzate per essere pronte all'elaborazione da parte della rete neurale.

Il dataset viene diviso in tre insiemi distinti : training, validation e test. Questo è un passo fondamentale nell'addestramento della rete neurale, poiché ci consente di allenare il modello su un insieme di dati, ottimizzarlo su un insieme di validazione e infine valutarne le prestazioni su un insieme di test indipendente.

La rete neurale stessa viene quindi definita e configurata. Si specificano il numero di nodi in ciascun layer, le funzioni di attivazione e la funzione di errore. Questo passo è cruciale per la corretta convergenza del modello durante l'addestramento.

Si fa uso dell'algoritmo di ottimizzazione Rprop per allenare la rete. Infine, i risultati dell'addestramento vengono presentati attraverso la valutazione della rete sul set di test. Vengono calcolate l'accuratezza del modello e la perdita ottenuta durante il processo di valutazione.

4 TEST E ANALISI DEI RISULTATI

In questo capitolo saranno mostrati i test per studiare l'apprendimento di una rete neurale con le seguenti caratteristiche:

- Una matrice di caratteristiche data in input di dimensione 784x50000 (per il training set) ;
- Un solo strato interno ;
- Settecentottantaquattro neuroni per lo strato di input ;
- Dieci neuroni per lo strato di output ;
- Funzione di attivazione per gli strati interni: ReLU ;
- Funzione di attivazione per lo strato di output: Identità ;
- Funzione di errore utilizzata: Cross-entropy con post-processing sull'output Softmax ;
- Euristica di apprendimento: Resilient back-propagation.

Per tutti i casi di test considerati, si sono utilizzate le seguenti caratteristiche:

- Dimensione del dataset: 60000 ;
- Dimensione del training set: 40000 ;
- Dimensione del validation set: 12000 ;
- Dimensione del test set: 8000 ;
- Approccio di valutazione: Hold-out ;
- Numero di epoche utilizzate: 35, 50, 80, 100 e 150.

L'obiettivo di questo studio consiste nell'effettuare un'analisi delle prestazioni della rete neurale al variare delle seguenti caratteristiche :

- Numero di epoche;
- Numero di neuroni.

I parametri della RPROP sono stati fissati nel seguente modo:

- EtaPlus: 1.2 ;
- EtaMinus: 0.5 ;
- DeltaMax: 50.0 ;
- DeltaMin: e^{-6} ;

- I valori iniziali dei delta sono stati inizializzati a 0.01.

Dunque, per ogni caso di test le informazioni da confrontare sono:

- Epoche necessarie per l'apprendimento ;
- Andamento dell'errore sul training e sul validation ;
- Accuratezza sul test ottenuta dalla rete migliore che ha raggiunto l'errore minimo sul validation set rispetto alle altre durante il processo di learning.

4.1 Risultati ottenuti

4.1.1 Test numero 1

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	10	35	ReLU	28	90.62%

Graficamente :

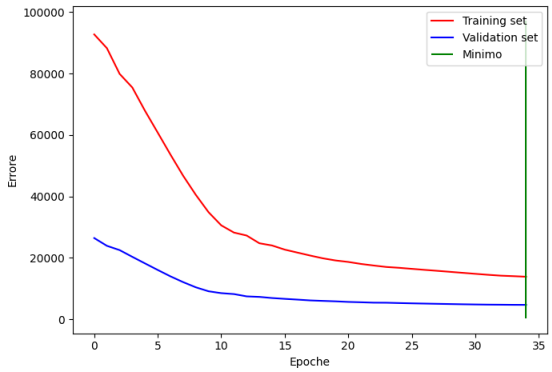


Figura 1 : un layer, 10 nodi, 35 epoche, ReLU

4.1.2 Test numero 2

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	30	35	ReLU	28	92.85%

Graficamente :

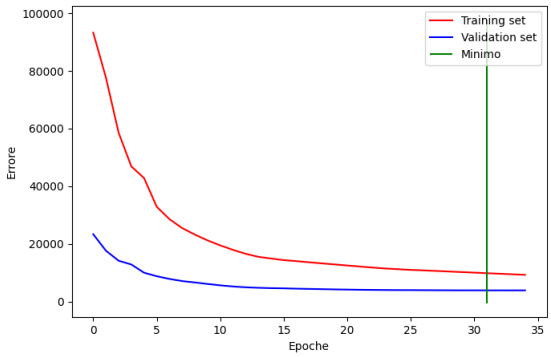


Figura 2 : un layer, 30 nodi, 35 epoche, ReLU

4.1.3 Test numero 3

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	50	35	ReLU	28	93.80%

Graficamente :

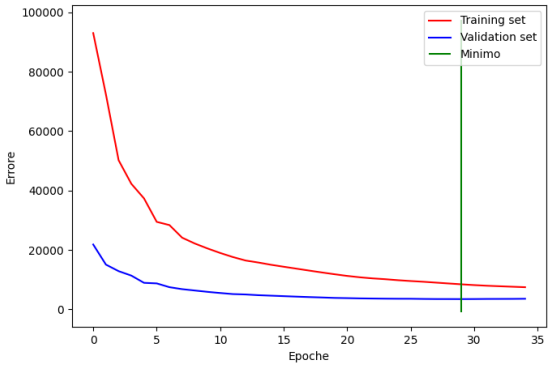


Figura 3 : un layer, 50 nodi, 35 epoche, ReLU

4.1.4 Test numero 4

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	64	35	ReLU	28	93.50%

Graficamente :

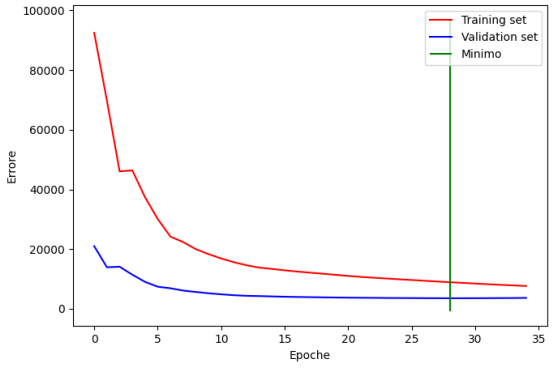


Figura 4 : un layer, 64 nodi, 35 epoche, ReLU

4.1.5 Test numero 5

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	128	35	ReLU	28	94.64%

Graficamente :

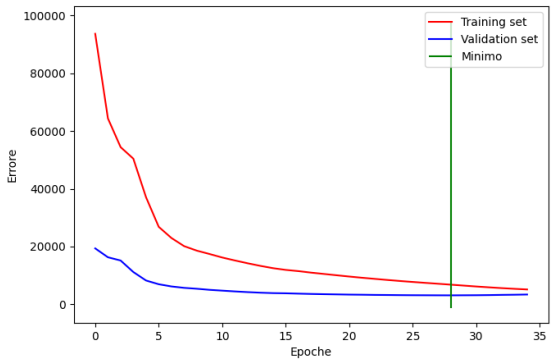


Figura 5 : un layer, 128 nodi, 35 epoche, ReLU

4.1.6 Test numero 6

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	256	35	ReLU	28	95.11%

Graficamente :

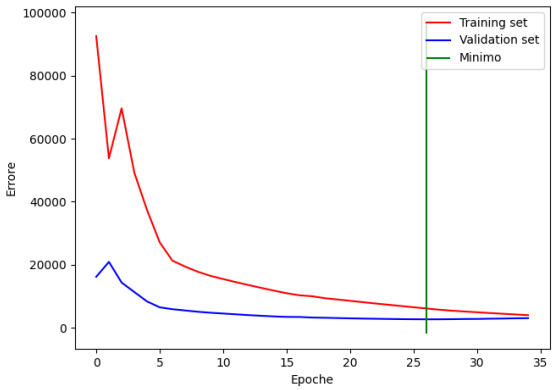


Figura 6 : un layer, 256 nodi, 35 epoche, ReLU

4.1.7 Test numero 7

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	512	35	ReLU	28	95.79%

Graficamente :

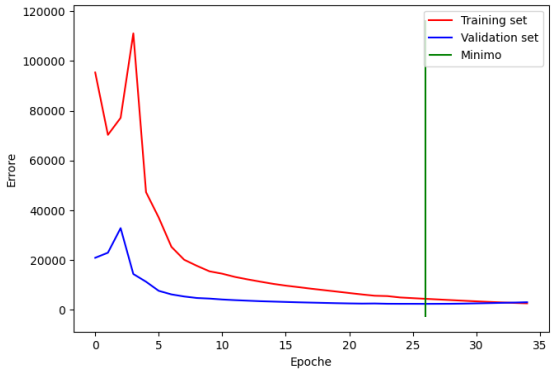


Figura 7 : un layer, 512 nodi, 35 epoche, ReLU

4.1.8 Test numero 8

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	1024	35	ReLU	28	96.29%

Graficamente :

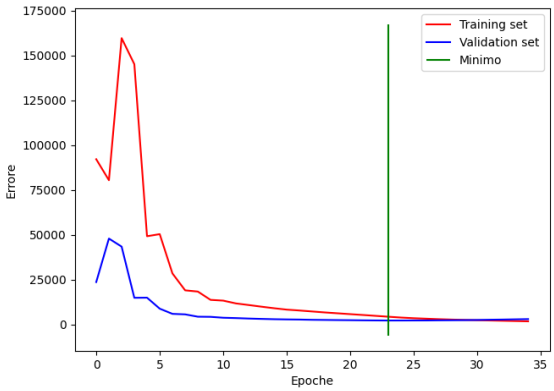


Figura 8 : un layer, 1024 nodi, 35 epoche, ReLU

4.1.9 Test numero 9

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	10	50	ReLU	45	90.80%

Graficamente :

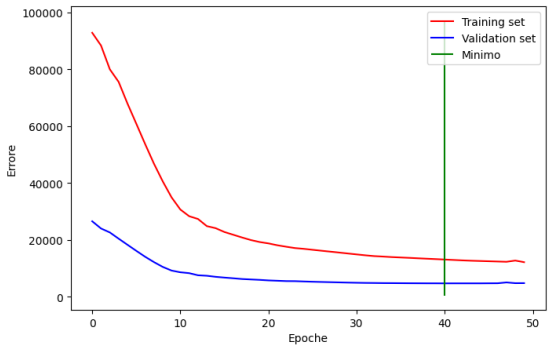


Figura 9 : un layer, 10 nodi, 50 epoche, ReLU

4.1.10 Test numero 10

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	30	50	ReLU	45	92.85%

Graficamente :

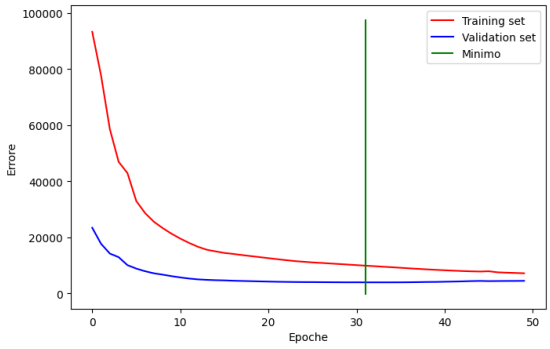


Figura 10 : un layer, 30 nodi, 50 epoche, ReLU

4.1.11 Test numero 11

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	50	50	ReLU	45	93.80%

Graficamente :

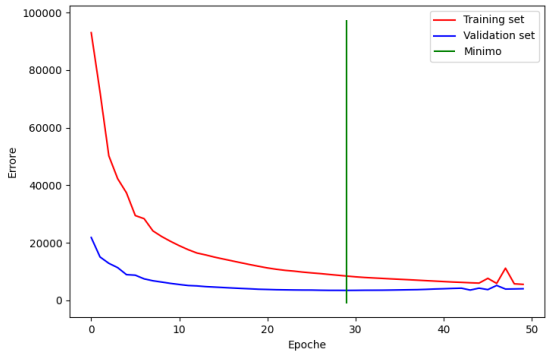


Figura 11 : un layer, 50 nodi, 50 epoche, ReLU

4.1.12 Test numero 12

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	64	50	ReLU	45	93.50%

Graficamente :

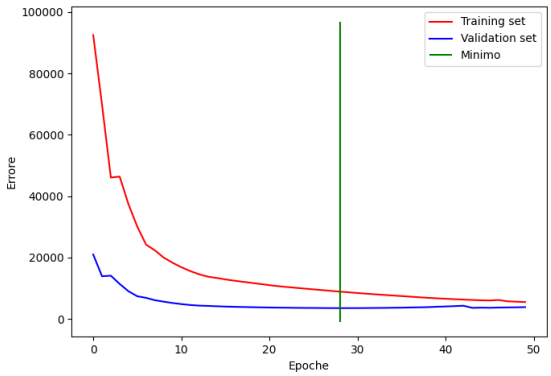


Figura 12 : un layer, 64 nodi, 50 epoche, ReLU

4.1.13 Test numero 13

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	128	50	ReLU	45	94.64%

Graficamente :

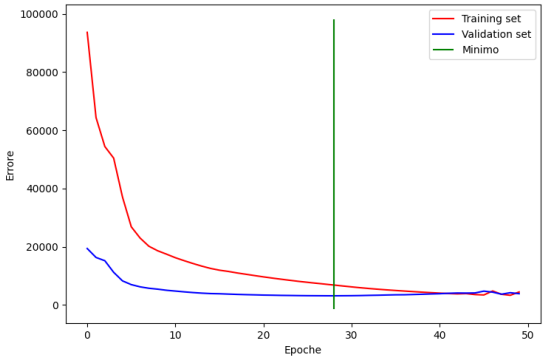


Figura 13 : un layer, 128 nodi, 50 epoche, ReLU

4.1.14 Test numero 14

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	256	50	ReLU	45	95.11%

Graficamente :

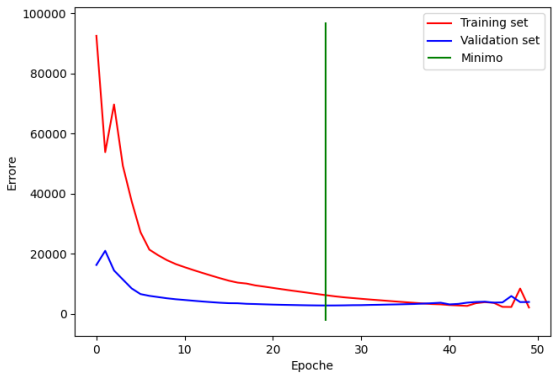


Figura 14 : un layer, 256 nodi, 50 epoche, ReLU

4.1.15 Test numero 15

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	512	50	ReLU	40	95.79%

Graficamente :

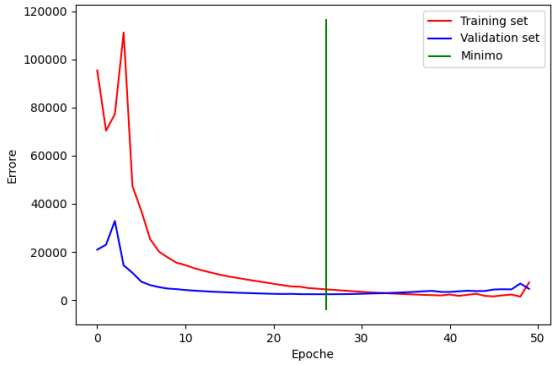


Figura 15 : un layer, 512 nodi, 50 epoche, ReLU

4.1.16 Test numero 16

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	1024	50	ReLU	40	96.29%

Graficamente :

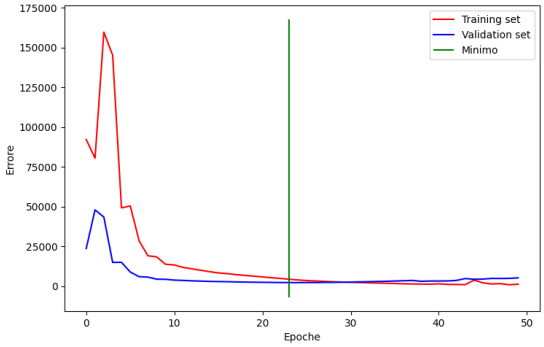


Figura 16 : un layer, 1024 nodi, 50 epoche, ReLU

4.1.17 Test numero 17

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	10	80	ReLU	72	90,80%

Graficamente :

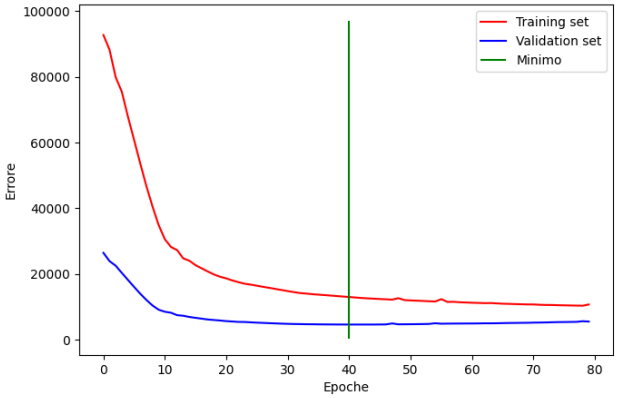


Figura 17 : un layer, 10 nodi, 80 epoche, ReLU

4.1.18 Test numero 18

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	30	80	ReLU	72	92,85%

Graficamente :

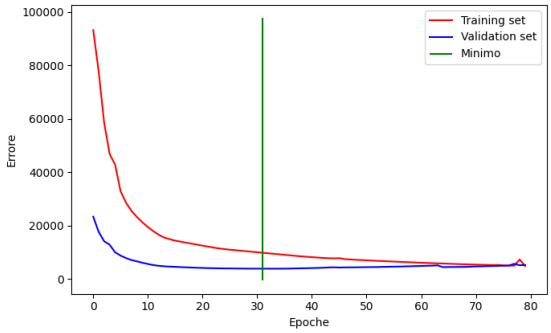


Figura 18 : un layer, 30 nodi, 80 epoche, ReLU

4.1.19 Test numero 19

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	50	80	ReLU	64	93,80%

Graficamente :

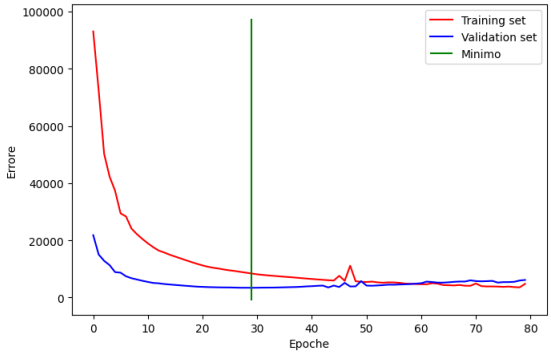


Figura 19 : un layer, 50 nodi, 80 epoche, ReLU

4.1.20 Test numero 20

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	64	80	ReLU	72	93,50%

Graficamente :

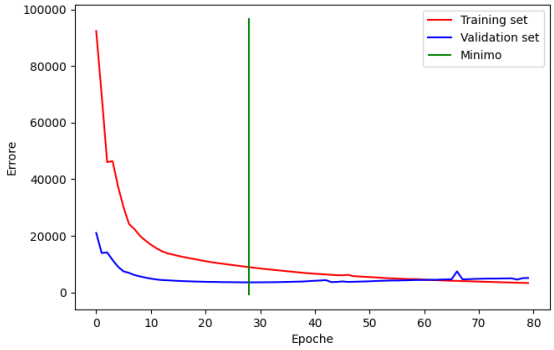


Figura 1: un layer, 64 nodi, 80 epoche, ReLU

4.1.21 Test numero 21

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	128	80	ReLU	72	94.64%

Graficamente :

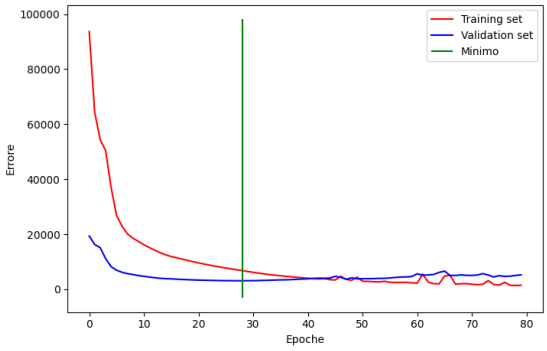


Figura 21 : un layer, 128 nodi, 80 epoche, ReLU

4.1.22 Test numero 22

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	256	80	ReLU	72	95,11%

Graficamente :

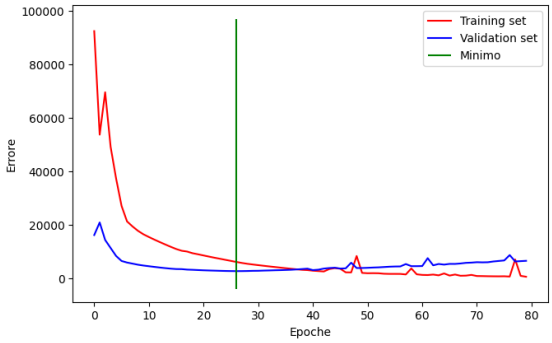


Figura 22 : un layer, 256 nodi, 80 epoche, ReLU

4.1.23 Test numero 23

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	512	80	ReLU	72	95.79%

Graficamente :

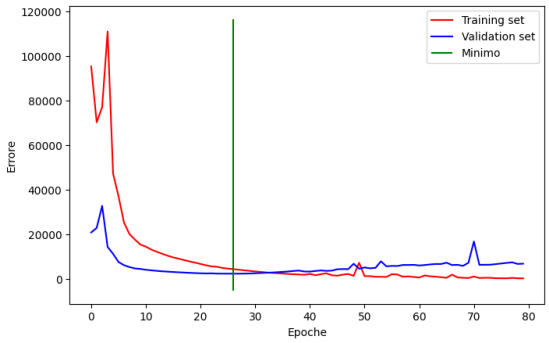


Figura 2: un layer, 512 nodi, 80 epoche, ReLU

4.1.24 Test numero 24

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	1024	80	ReLU	72	95,79%

Graficamente :

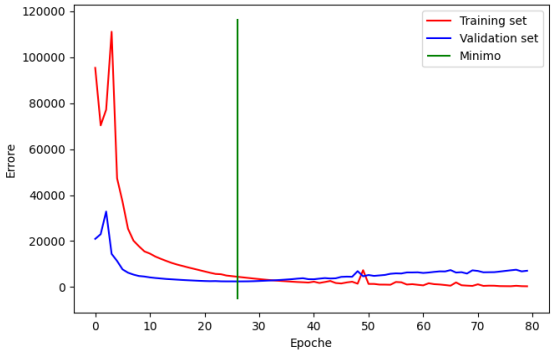


Figura 24 : un layer, 1024 nodi, 80 epoche, ReLU

4.1.25 Test numero 25

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	10	100	ReLU	90	90,80%

Graficamente :

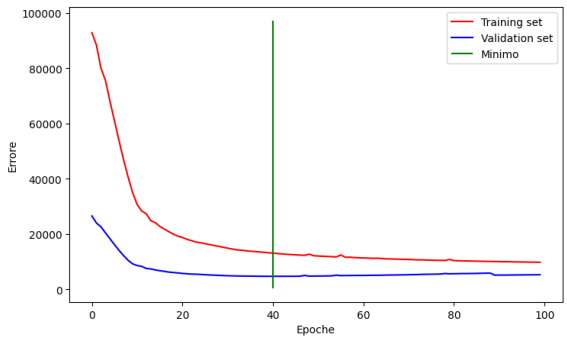


Figura 25 : un layer, 10 nodi, 100 epoche, ReLU

4.1.26 Test numero 26

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	30	100	ReLU	90	92,85%

Graficamente :

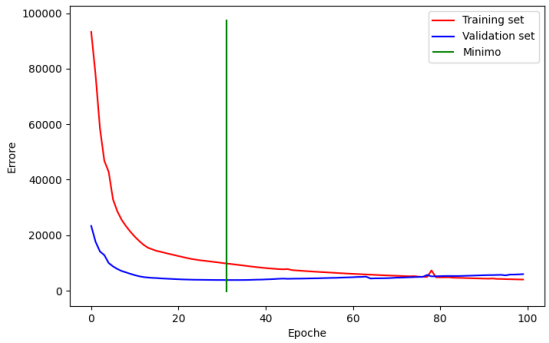


Figura 26 : un layer, 30 nodi, 100 epoche, ReLU

4.1.27 Test numero 27

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	50	100	ReLU	90	93,80%

Graficamente :

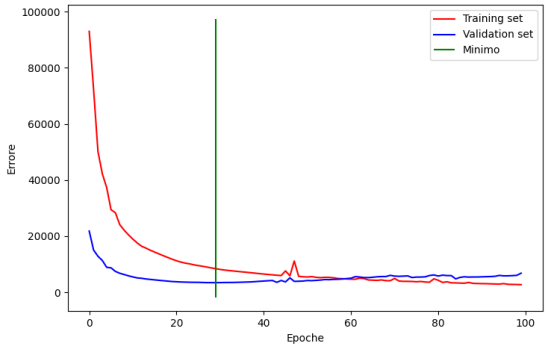


Figura 27 : un layer, 50 nodi, 100 epoche, ReLU

4.1.28 Test numero 28

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	64	100	ReLU	90	93,50%

Graficamente :

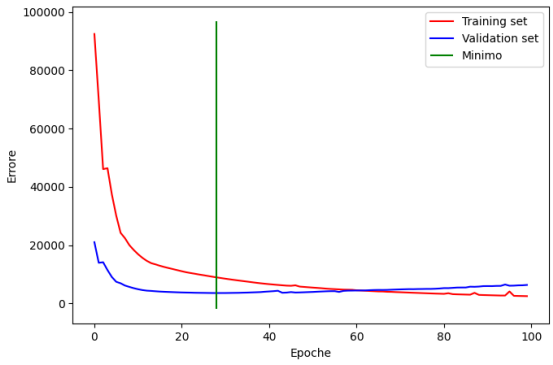


Figura 28 : un layer, 64 nodi, 100 epoche, ReLU

4.1.29 Test numero 29

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	128	100	ReLU	90	94,64%

Graficamente :

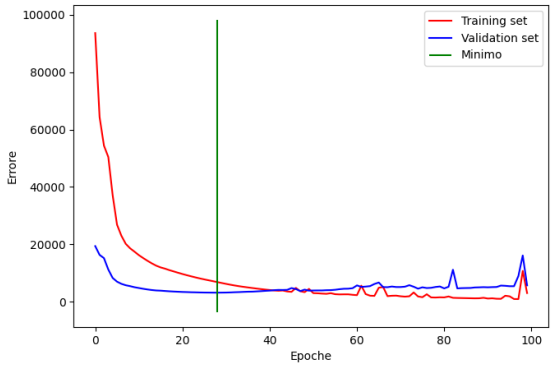


Figura 29 : un layer, 128 nodi, 100 epoche, ReLU

4.1.30 Test numero 30

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	256	100	ReLU	90	95,11%

Graficamente :

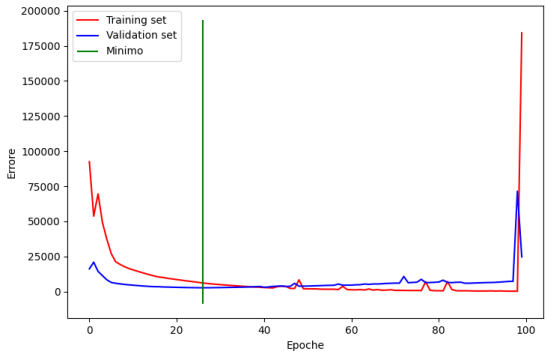


Figura 30 : un layer, 256 nodi, 100 epoche, ReLU

4.1.31 Test numero 31

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	512	100	ReLU	80	95,79%

Graficamente :

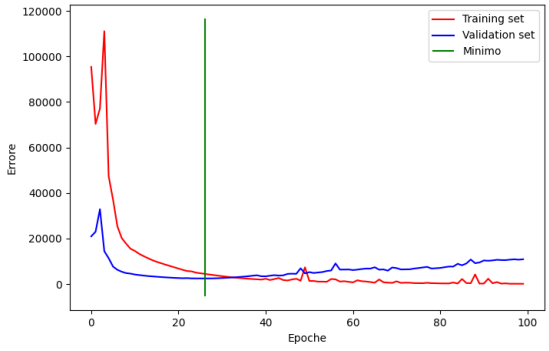


Figura 31 : un layer, 512 nodi, 100 epoche, ReLU

4.1.32 Test numero 32

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	1024	100	ReLU	90	96,29%

Graficamente :

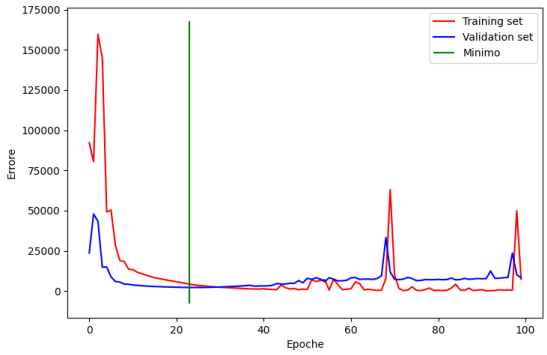


Figura 32 : un layer, 1024 nodi, 100 epoche, ReLU

4.1.33 Test numero 33

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	10	150	ReLU	135	90,80%

Graficamente :

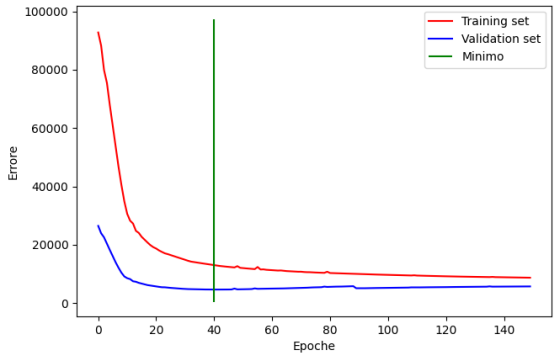


Figura 33 : un layer, 10 nodi, 150 epoche, ReLU

4.1.34 Test numero 34

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	30	150	ReLU	135	92.85%

Graficamente :

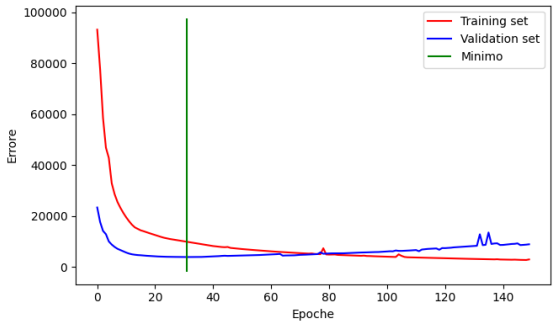


Figura 34 : un layer, 30 nodi, 150 epoche, ReLU

4.1.35 Test numero 35

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	50	150	ReLU	135	93,80%

Graficamente :

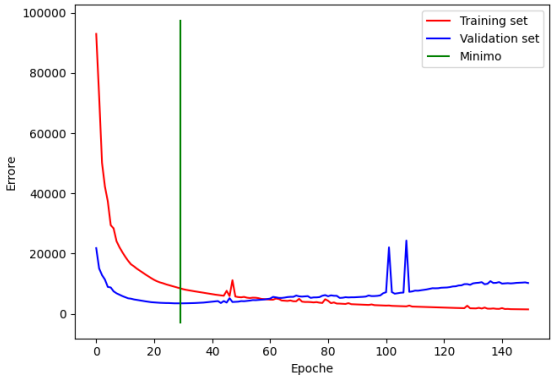


Figura 35 : un layer, 30 nodi, 150 epoche, ReLU

4.1.36 Test numero 36

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	64	150	ReLU	135	93,50%

Graficamente :

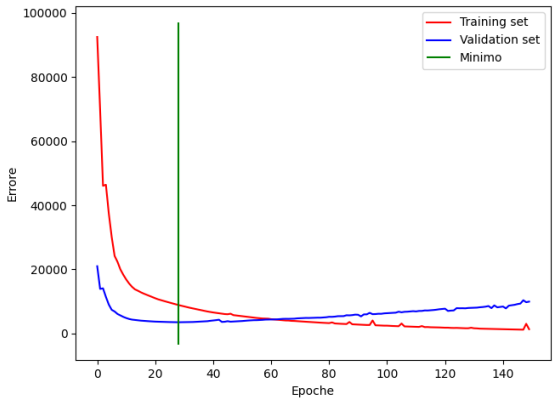


Figura 36 : un layer, 64 nodi, 150 epoche, ReLU

4.1.37 Test numero 37

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	128	150	ReLU	135	94.64%

Graficamente :

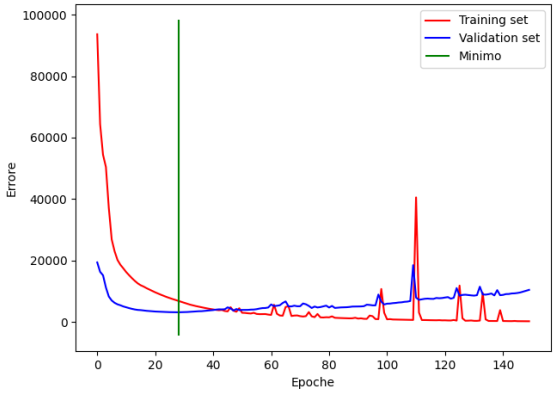


Figura 37 : un layer, 128 nodi, 150 epoche, ReLU

4.1.38 Test numero 38

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	256	150	ReLU	135	95,11%

Graficamente :

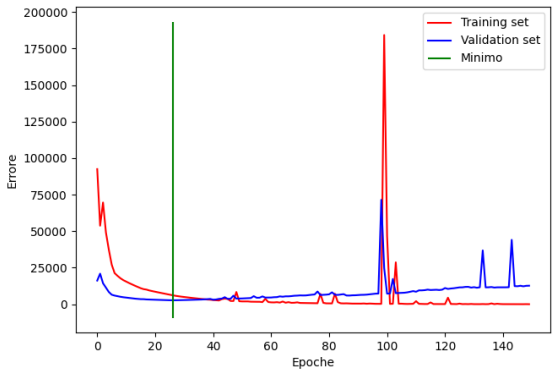


Figura 3: un layer, 256 nodi, 150 epoche, ReLU

4.1.39 Test numero 39

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	512	150	ReLU	135	95.79%

Graficamente :

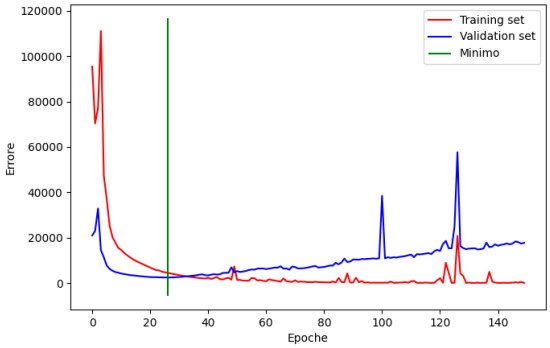


Figura 39 : un layer, 512 nodi, 150 epoche, ReLU

4.1.40 Test numero 40

Hidden layers	Neuroni per ogni layer	Numero epoche	Funzione di attivazione usata	Migliore epoca	Accuracy sul test set
1	1024	150	ReLU	135	96.29%

Graficamente :

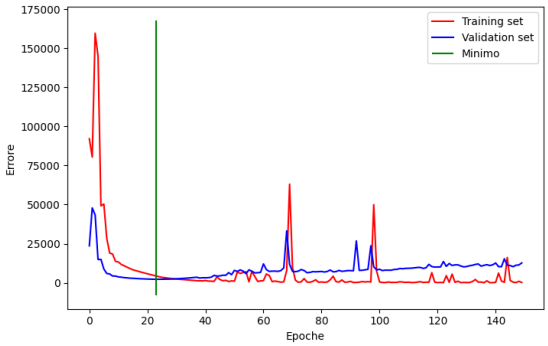


Figura 40 : un layer, 1024 nodi, 150 epoche, ReLU

4.2 Analisi dei test

I test effettuati mirano a comprendere l'influenza di diverse configurazioni sui modelli di reti neurali. Ognuno dei 40 test eseguiti ha esplorato variazioni nei parametri chiave, quali il numero di neuroni e il numero di epoche per ogni singolo strato nascosto.

Iniziamo con l'osservare l'andamento generale. Con l'aumentare del numero di neuroni, si evidenzia un miglioramento significativo delle prestazioni della rete, come testimoniato dall'incremento graduale dell'accuratezza sui set di test. È tuttavia cruciale sottolineare che tale miglioramento segue un'andatura verso un punto di saturazione, suggerendo che al di là di un certo punto, l'aggiunta di neuroni potrebbe non tradursi in guadagni sostanziali.

Un aspetto notevole è che, all'aumentare delle epoche, emerge una tendenza positiva nelle prestazioni della rete neurale. Tuttavia, è altrettanto evidente che si raggiunge una fase di stabilizzazione, suggerendo che ulteriori incrementi nel numero di epoche potrebbero generare benefici limitati o addirittura comportare un deterioramento delle performance. Questa situazione può essere attribuita a possibili problematiche legate all'overfitting, in cui la rete tende a adattarsi eccessivamente ai dati di addestramento, perdendo la sua capacità di generalizzare su nuovi dati. Pertanto, è cruciale bilanciare attentamente il numero di epoche per ottenere prestazioni ottimali senza cadere in un punto di saturazione o incorrere in fenomeni dannosi come l'overfitting.

Guardando alle funzioni di attivazione, la ReLU è stata particolarmente efficace in tutti i casi esaminati. Ciò nonostante, potrebbe risultare interessante esplorare altre funzioni di attivazione, come la sigmoide o la tangente iperbolica, per valutare se possono portare a miglioramenti in scenari specifici.

Inoltre, è interessante notare che le prestazioni della rete sembrano beneficiare da una corretta combinazione di numero di neuroni ed epoche. Alcune configurazioni si dimostrano più efficienti rispetto ad altre, indicando che la scelta accurata di entrambi questi fattori, numero di neuroni e numero di epoche, è cruciale per ottenere risultati ottimali. In quanto l'ottimizzazione della rete attraverso l'adeguata regolazione di neuroni ed epoche può contribuire a massimizzare l'accuratezza sui set di test.

In conclusione, dai test effettuati è emerso che i risultati ottenuti evidenziano l'importanza della selezione accurata delle configurazioni per le reti neurali. Questa scelta dovrebbe essere guidata dalla comprensione approfondita delle caratteristiche dei dati e degli obiettivi di apprendimento specifici. Il bilanciamento tra il numero di neuroni e il numero di epoche gioca un ruolo critico nell'ottenere prestazioni ottimali, e il raggiungimento di un punto di saturazione indica che un eccessivo aumento di tali parametri potrebbe non tradursi in miglioramenti significativi o addirittura condurre a problemi di overfitting. Pertanto, un approccio attentamente ponderato nella configurazione della rete può massimizzare l'efficienza dell'apprendimento automatico in base al contesto specifico in esame.