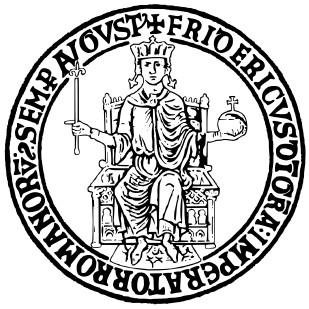
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Progetto di Neural Networks and Deep Learning

Progettazione e Sviluppo di una Libreria per l'addestramento di Reti Neurali dedicate al confronto tra Regole di Aggiornamento dei Pesi

Studenti
Giuseppe Piccolo
Francesco Mazza

Professore
Roberto Prevete

Anno Accademico 2022–2023

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Questa documentazione descrive il lavoro svolto per la realizzazione di una libreria python dedicata alla creazione e l'apprendimento di reti neurali e l'implementazione di vari algoritmi di apprendimento. In questo lavoro, è stato fatto un confronto delle prestazioni delle reti al variare delle configurazioni: in primis le regole di aggiornamento dei pesi, che sono la parte principale della traccia, ma anche le funzioni di attivazione e il *pre-processing* sui dati. Dopo una breve descrizione della traccia, sarà riassunto il lavoro svolto. Nella seconda parte della documentazione, verrà descritta la libreria in tutti i suoi dettagli. Infine nella terza ed ultima parte saranno analizzati i risultati ottenuti.

1.1 Parte A

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.

1.2 Parte B (Traccia 5)

Dato il dataset MNIST di immagini raw. 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 (considerare almeno 10000 elementi per il training set e 2500 per il test set). Seguendo l'articolo "Empirical evaluation of the improved Rprop learning algorithms, Christian Igel, Michael Husken, neu-

rocomputing, 2003" si confronti la classica resilient backpropagation (RProp) con almeno 2 varianti proposte nell'articolo, come algoritmo di aggiornamento dei pesi (aggiornamento batch). Si fissi la funzione di attivazione ed il numero di nodi interni (almeno tre diverse dimensioni) e si confrontino i risultati ottenuti con i diversi algoritmi di apprendimento. Se è necessario, per questioni di tempi computazionali e spazio in memoria, si possono ridurre le dimensioni delle immagini raw del dataset mnist (ad esempio utilizzando in matlab la funzione imresize)

1.3 Lavoro svolto

L'intera libreria è stata scritta utilizzando il linguaggio di programmazione **Python**, accompagnato assieme alcune librerie che hanno facilitato alcuni task. La libreria sviluppata permette di creare reti neurali con architettura **Feed-Forward**, con un qualsiasi numero di layer interni, ciascuno dotato di un arbitrario numero di nodi interni. É possibile quindi creare sia reti di tipo **Shallow** che reti di tipo **Deep**. Sono state inoltre implementate diverse funzioni per l'inizializzazione dei pesi di un layer, che utilizzano metodologie differenti. Diverse sono anche le funzioni di attivazione implementate, utilizzabili per i diversi layer. Tramite la libreria, è possibile valutare le prestazioni di una rete neurale, usando le funzioni d'errore, a seconda se si sta risolvendo un problema di classificazione, oppure un problema di regressione. Come richiesto dalla traccia, sono state implementate le diverse varianti della *Resilient Back Propagation* utilizzabili come regole di aggiornamento dei pesi. In più è stato aggiunto lo *Stachastic Gradient Descent*, così da compararlo con le varianti dell'RProp. Gli esperimenti sono stati eseguiti su reti neurali di tipo shallow, con un solo strato interno, facendo variare il numero di nodi interni, il preprocessing sui dati, la funzione di attivazione e l'inizializzazione dei pesi della rete.

Capitolo 2

Implementazione libreria

In questo capitolo verranno descritte tutte le funzionalità implementate nella libreria per lo svolgimento della traccia del progetto. In particolare verranno descritte le tecnologie impiegate per lo sviluppo del progetto, l'architettura usata per l'implementazione della libreria e i metodi utilizzati per verificare i parametri di valutazione di una rete neurale, misurare le prestazioni di un determinato algoritmo di apprendimento e confrontarli tra loro.

2.1 Funzionalità libreria

Come richiesto dalla Parte A della traccia, la libreria offre la possibilità di creare reti neurali multi-strato con la possibilità di scegliere il numero di neuroni di ogni strato. L'implementazione di questi strati è stata effettuata tramite la programmazione a oggetti. Quindi ogni strato di una rete sarà rappresentata da un oggetto. Ogni layer avrà dei metodi che permettono di gestire tutti i calcoli della propagazione in avanti e la propagazione all'indietro. La libreria offre anche la possibilità di scegliere diverse funzioni di attivazione per gli strati interni. Anche in questo caso, una specifica funzione di attivazione di un determinato layer, sarà rappresentata tramite un oggetto. Quindi, per ogni funzione di attivazione implementata è stata definita una classe in cui sono state inserite le funzioni per i calcoli necessari alla forward-propagation e alla back-propagation. Sono anche state implementate le classi che permettono il calcolo del parametro loss tramite Cross-Entropy, Cross-Entropy più Softmax e Sum of Squares per lo strato finale di una rete neurale. Gli algoritmi implementati, per effettuare l'aggiornamento dei parametri sono: Stochastic Gradient Descent, `Rprop`$^-$, `Rprop`$^+$, `iRprop`$^-$ e `iRprop`$^+$. Anche in questo caso, sono state dichiarate delle classi che si occupano di eseguire tutti i vari passi che riguardano l'aggiornamento dei parametri. Infine, sono state aggiunte delle funzioni che per-

mettono di ottenere le immagini dal dataset MNIST e delle funzioni che permettono di graficare i parametri di valutazione loss e accuratezza.

2.2 Tecnologie

Il linguaggio di programmazione usato per implementare la libreria è python nella sua versione 3. Sono state utilizzate anche alcune librerie per facilitare lo sviluppo. Per eseguire i vari prodotti tra matrici e vettori e per facilitare la manipolazione di questi elementi è stata usata la libreria NumPy. Per ottenere l'intero dataset MNIST, è stata usata la libreria Keras. Per mischiare i dati di questo dataset è stata usata la funzione shuffle proveniente dal pacchetto sklearn. Infine, per mostrare i vari grafici, come pacchetto è stato utilizzato matplotlib.

2.3 Classi della libreria

Come già accennato, si è cercato di usare molto le classi per lo sviluppo della libreria, così da rendere quanto più semplice possibile la creazione e l'utilizzo delle reti neurali. Ogni elemento come: layer, funzione di attivazione, algoritmo di aggiornamento è stato rappresentato come classe. Ovviamente, ognuna di queste classi offre dei metodi che vengono usati per la forward-propagation e per la back-propagation. In questa sezione, saranno descritte tutte le classi con i propri metodi.

2.3.1 Classe Layer_Dense

Questa classe permette di definire oggetti che rappresentano un layer della rete neurale **Feed-Forward Full Connected**, tramite costruttore, è possibile impostare il numero di input da passare al layer e il numero di neuroni dello strato. Tramite i metodi **forward** e **backward** è possibile eseguire la forward-propagation e la back-propagation. Di seguito, sono elencati i metodi della classe:

- `__init__(self, n_inputs, n_neurons, initialization="xavier")`

    1. `n_inputs`: numero di valori in input da dare in ingresso al layer
    2. `n_neurons`: numero di neuroni del layer
    3. `initialization`: tipo di inizializzazione da usare per generare i pesi. I valori possibili sono: "xavier", "xavier_variant" e "random". Di default il valore è "xavier"

- `forward(self, inputs)`

    1. `inputs`: input da dare in ingresso alla rete per eseguire la forward-propagation

- `backward(self, dvalues)`

    1. `dvalues`: derivate dei valori generati dal layer precedente durante la back-propagation

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `weights`: matrice dei pesi del layer
- `bias`: vettore dei biase del layer
- `dweights_cache`: matrice delle derivate dei pesi dell'iterazione t-1
- `dbiases_cache`: vettore delle derivate dei bias dell'iterazione t-1
- `inputs`: dati di ingresso del layer
- `output`: output del layer
- `dweights`: derivate pesi dell'iterazione corrente
- `dbiases`: derivate bias dell'iterazione corrente
- `dinputs`: derivate dei valori dati in input

2.3.2 Classe Activation_ReLu

Questa classe permette di utilizzare la funzione di attivazione ReLu in uno specifico layer. Tramite il metodo **forward** è possibile attivare la funzione per la forward-propagation, mentre con **backward** è possibile eseguire la derivata della funzione per la backward-propagation. Di seguito, sono elencati i metodi della classe:

- `forward(self, inputs)`

    1. `inputs`: input su cui attivare la funzione durante la forward-propagation

- `backward(self, dvalues)`

    1. `dvalues`: derivate dei valori generati dal layer precedente durante la back-propagation

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `inputs`: dati su cui attivare la funzione di attivazione

- `output`: output della funzione di attivazione
- `dinputs`: derivate generate durante la back-propagation

2.3.3 Classe Activation_LReLu

Questa classe permette di utilizzare la funzione di attivazione Leaky ReLu in uno specifico layer. Tramite il metodo **forward** è possibile attivare la funzione per la forward-propagation, mentre con **backward** è possibile eseguire la derivata della funzione per la backward-propagation. Di seguito, sono elencati i metodi della classe:

- `__init__(self, alpha=0.01)`
    1. `alpha`: iperparametro da utilizzare nella LReLu durante la fase di forward-propagation e durante la fase di back-propagation, di default è impostato a 0.01
- `forward(self, inputs)`
    1. `inputs`: input su cui attivare la funzione durante la forward-propagation
- `backward(self, dvalues)`
    1. `dvalues`: derivate dei valori generati dal layer precedente durante la back-propagation

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `alpha`: iperparametro della LReLu
- `inputs`: dati su cui attivare la funzione di attivazione
- `output`: output della funzione di attivazione
- `dinputs`: derivate generate durante la back-propagation

2.3.4 Classe Activation_Sigmoid

Questa classe permette di utilizzare la funzione di attivazione Sigmoide in uno specifico layer. Tramite il metodo **forward** è possibile attivare la funzione per la forward-propagation, mentre con **backward** è possibile eseguire la derivata della funzione per la backward-propagation. Di seguito, sono elencati i metodi della classe:

- `forward(self, inputs)`
    1. `inputs`: input su cui attivare la funzione durante la forward-propagation

- `backward(self, dvalues)`

    1. `dvalues`: derivate dei valori generati dal layer precedente durante la back-propagation

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `inputs`: dati su cui attivare la funzione di attivazione
- `output`: output della funzione di attivazione
- `dinputs`: derivate generate durante la back-propagation

2.3.5 Classe Activation_Linear

Questa classe permette di utilizzare la funzione di attivazione Identità in uno specifico layer. Tramite il metodo **forward** è possibile attivare la funzione per la forward-propagation, mentre con **backward** è possibile eseguire la derivata della funzione per la backward-propagation. Di seguito, sono elencati i metodi della classe:

- `forward(self, inputs)`

    1. `inputs`: input su cui attivare la funzione durante la forward-propagation

- `backward(self, dvalues)`

    1. `dvalues`: derivate dei valori generati dal layer precedente durante la back-propagation

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `inputs`: dati su cui attivare la funzione di attivazione
- `output`: output della funzione di attivazione
- `dinputs`: derivate generate durante la back-propagation

2.3.6 Classe Activation_Softmax

Questa classe permette di utilizzare la funzione di attivazione Softmax nell'output layer. Tramite il metodo **forward** è possibile attivare la funzione per la forward-propagation. Di seguito, sono elencati i metodi della classe:

- `forward(self, inputs)`

    1. `inputs`: input su cui attivare la funzione durante la forward-propagation

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `inputs`: dati su cui attivare la funzione di attivazione
- `output`: output della funzione di attivazione

2.3.7 Classe Loss

Tutte le classi che permettono di calcolare l'errore di una rete neurale, ereditano da questa classe. Questa classe offre il metodo **calculate**. Questo metodo, richiama il metodo forward, definito nella classe figlia, eseguire il calcolo degli errori, successivamente esegue la media di quest'ultimi e restituisce il risultato. Di seguito, sono elencati i metodi della classe:

- `calcultate(self, output, y)`
    1. `output`: output prodotto dalla rete neurale
    2. `y`: etichetta reale di uno specifico valore dato in input

2.3.8 Classe Loss_CategoricalCrossentropy

Questa classe eredita dalla classe **Loss**. Viene utilizzata per calcolare l'errore dell'output di una rete neurale tramite la funzione matematica **Cross-Entropy**. I due metodi che possono essere utilizzati sono: **forward**, tramite questo metodo è possibile eseguire il calcolo dell'errore dell'output della rete neurale, **backward**, tramite questo metodo è possibile calcolare la derivata della Cross-Entropy. Di seguito, sono elencati i metodi della classe:

- `forward(self, y_pred, y_true)`
    1. `y_pred`: output predetto dalla rete neurale
    2. `y_true`: etichette dei valori dati in input
- `backward(self, dvalues, y_true)`
    1. `dvalues`: valori su cui applicare la derivata della Cross-Entropy
    2. `y_true`: etichette dei valori dati in input

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `dinputs`: derivate generate durante la back-propagation

2.3.9 Classe Loss_MeanSquaredError

Questa classe eredita dalla classe **Loss**. Viene utilizzata per calcolare l'errore dell'output di una rete neurale tramite la funzione matematica **Sum of Squares**. I due metodi che possono essere utilizzati sono: **forward**, tramite questo metodo è possibile eseguire il calcolo dell'errore dell'output della rete neurale, **backward**, tramite questo metodo è possibile calcolare la derivata della Sum of Squares. Di seguito, sono elencati i metodi della classe:

- `forward(self, y_pred, y_true)`
    1. `y_pred`: output predetto dalla rete neurale
    2. `y_true`: etichette dei valori dati in input
- `backward(self, dvalues, y_true)`
    1. `dvalues`: valori su cui applicare la derivata della Sum of Squares
    2. `y_true`: etichette dei valori dati in input

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `dinputs`: derivate generate durante la back-propagation

2.3.10 Classe Activation_Softmax_Loss_CategoricalCrossentropy

Questa classe eredita dalla classe **Loss**. Viene utilizzata per calcolare l'errore dell'output di una rete neurale tramite la funzione matematica **Softmax + Cross-Entropy**. I tre metodi che possono essere utilizzati sono: **forward**, tramite questo metodo è possibile eseguire il calcolo dell'errore dell'output della rete neurale, **forward_without_loss** permette di eseguire la propagazione in avanti senza eseguire il calcolo dell'errore, **backward**, tramite questo metodo è possibile calcolare la derivata della Softmax + Cross-Entropy. Di seguito, sono elencati i metodi della classe:

- `forward(self, inputs, y_true)`
    1. `inputs`: valori su cui attivare la funzione Softmax+Cross-Entropy
    2. `y_true`: etichette dei valori dati in input
- `forward_without_loss(self, inputs)`
    1. `inputs`: valori su cui attivare la funzione Softmax+Cross-Entropy
- `backward(self, dvalues, y_true)`

1. `dvalues`: valori su cui applicare la derivata della Softmax+Cross-Entropy
2. `y_true`: etichette dei valori dati in input

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `activation`: oggetto che rappresenta la funzione Softmax
- `loss`: oggetto che rappresenta la funzione d'errore
- `output`: output generato dalla funzione di attivazione
- `dinputs`: derivate generate durante la back-propagation

2.3.11 Classe Optimizer_SGD

Questa classe permette di utilizzare la regola di aggiornamento della **Discesa del gradiente**, per aggiornare i parametri di una rete neurale, durante il suo apprendimento. Alla creazione di un oggetto di questa classe, bisogna specificare il **learning rate**. Di deafault, questo valore è impostato a **0.0001**, ma è possibile specificare al costruttore un altro valore. Un altro metodo che offre la classe è **update_params**, che permette di applicare l'algoritmo della Discesa del gradiente sui parametri di uno strato, specificato come parametro quando si richiama il metodo. Di seguito, sono elencati i metodi della classe:

- `__init__(self, learning_rate=0.0001)`
    1. `inputs`: rappresenta l'iperparametro learning rate
- `update_params(self, layer)`
    1. `layer`: layer di una rete neurale su cui applicare la regola di aggiornamento

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `learning_rate`: learning rate impostato per l'esecuzione dell'algoritmo

2.3.12 Classe Optimizer_RProp_Minus

Questa classe permette di utilizzare la regola di aggiornamento dell'$RProp^-$, per aggiornare i parametri di una rete neurale, durante il suo apprendimento. Alla creazione di un oggetto di questa classe, bisogna specificare il passo incrementale η^+, il passo decrementale η^-, la soglia massima Δ_{max} e la soglia minima Δ_{min}. Di deafault, questi valori sono rispettivamente impostati a **1,2**, **0,5**, **50**, e **0** ma è

possibile specificare al costruttore degli altri valori per questi iperparametri. Un altro metodo che offre la classe è **update_params**, che permette di applicare l'algoritmo dell'$RProp^-$ sui parametri di uno strato, specificato come parametro quando si richiama il metodo. Di seguito, sono elencati i metodi della classe:

- `__init__(self, eta_pos=1.2, eta_neg=0.5, delta_max=50, delta_min=0)`
    1. `eta_pos`: rappresenta il passo incrementale
    2. `eta_neg`: rappresenta il passo decrementale
    3. `delta_max`: rappresenta la soglia massima
    4. `delta_min`: rappresenta la soglia minima
- `update_params(self, layer)`
    1. `layer`: layer di una rete neurale su cui applicare la regola di aggiornamento

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `eta_positive`: passo incrementale impostato per l'esecuzione dell'algoritmo
- `eta_negative`: passo decrementale impostato per l'esecuzione dell'algoritmo
- `delta_max`: soglia massima impostata per l'esecuzione dell'algoritmo
- `delta_min`: soglia minima impostata per l'esecuzione dell'algoritmo

2.3.13 Classe Optimizer_RProp_Plus

Questa classe permette di utilizzare la regola di aggiornamento dell'$RProp^+$, per aggiornare i parametri di una rete neurale, durante il suo apprendimento. Alla creazione di un oggetto di questa classe, bisogna specificare il passo incrementale η^+, il passo decrementale η^-, la soglia massima Δ_{max} e la soglia minima Δ_{min}. Di deafault, questi valori sono rispettivamente impostati a **1,2**, **0,5**, **50**, e **0** ma è possibile specificare al costruttore degli altri valori per questi iperparametri. Un altro metodo che offre la classe è **update_params**, che permette di applicare l'algoritmo dell'$RProp^+$ sui parametri di uno strato, specificato come parametro quando si richiama il metodo. Di seguito, sono elencati i metodi della classe:

- `__init__(self, eta_pos=1.2, eta_neg=0.5, delta_max=50, delta_min=0)`
    1. `eta_pos`: rappresenta il passo incrementale
    2. `eta_neg`: rappresenta il passo decrementale

3. `delta_max`: rappresenta la soglia massima
4. `delta_min`: rappresenta la soglia minima

- `update_params(self, layer)`
    1. `layer`: layer di una rete neurale su cui applicare la regola di aggiornamento

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `eta_positive`: passo incrementale impostato per l'esecuzione dell'algoritmo
- `eta_negative`: passo decrementale impostato per l'esecuzione dell'algoritmo
- `delta_max`: soglia massima impostata per l'esecuzione dell'algoritmo
- `delta_min`: soglia minima impostata per l'esecuzione dell'algoritmo

2.3.14 Classe Optimizer_iRProp_Plus

Questa classe permette di utilizzare la regola di aggiornamento dell'$iRProp^+$, per aggiornare i parametri di una rete neurale, durante il suo apprendimento. Alla creazione di un oggetto di questa classe, bisogna specificare il passo incrementale η^+, il passo decrementale η^-, la soglia massima Δ_{max} e la soglia minima Δ_{min}. Di deafault, questi valori sono rispettivamente impostati a **1,2**, **0,5**, **50**, e **0** ma è possibile specificare al costruttore degli altri valori per questi iperparametri. Un altro metodo che offre la classe è **update_params**, che permette di applicare l'algoritmo dell'$iRProp^+$ sui parametri di uno strato, specificato come parametro quando si richiama il metodo. Di seguito, sono elencati i metodi della classe:

- `__init__(self, eta_pos=1.2, eta_neg=0.5, delta_max=50, delta_min=0)`
    1. `eta_pos`: rappresenta il passo incrementale
    2. `eta_neg`: rappresenta il passo decrementale
    3. `delta_max`: rappresenta la soglia massima
    4. `delta_min`: rappresenta la soglia minima

- `update_params(self, layer, loss)`
    1. `layer`: layer di una rete neurale su cui applicare la regola di aggiornamento
    2. `loss`: loss valore di perdita calcolato durante l'epoca corrente

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `eta_positive`: passo incrementale impostato per l'esecuzione dell'algoritmo
- `eta_negative`: passo decrementale impostato per l'esecuzione dell'algoritmo
- `delta_max`: soglia massima impostata per l'esecuzione dell'algoritmo
- `delta_min`: soglia minima impostata per l'esecuzione dell'algoritmo

2.3.15 Classe Optimizer_iRProp_Minus

Questa classe permette di utilizzare la regola di aggiornamento dell'`iRProp`$^-$, per aggiornare i parametri di una rete neurale, durante il suo apprendimento. Alla creazione di un oggetto di questa classe, bisogna specificare il passo incrementale η^+, il passo decrementale η^-, la soglia massima Δ_{max} e la soglia minima Δ_{min}. Di deafault, questi valori sono rispettivamente impostati a **1,2**, **0,5**, **50**, e **0** ma è possibile specificare al costruttore degli altri valori per questi iperparametri. Un altro metodo che offre la classe è **update_params**, che permette di applicare l'algoritmo dell'`iRProp`$^-$ sui parametri di uno strato, specificato come parametro quando si richiama il metodo. Di seguito, sono elencati i metodi della classe:

- `__init__(self, eta_pos=1.2, eta_neg=0.5, delta_max=50, delta_min=0)`
  1. `eta_pos`: rappresenta il passo incrementale
  2. `eta_neg`: rappresenta il passo decrementale
  3. `delta_max`: rappresenta la soglia massima
  4. `delta_min`: rappresenta la soglia minima
- `update_params(self, layer)`
  1. `layer`: layer di una rete neurale su cui applicare la regola di aggiornamento

Oltre ai metodi precedenti, questa classe definisce anche i seguenti campi:

- `eta_positive`: passo incrementale impostato per l'esecuzione dell'algoritmo
- `eta_negative`: passo decrementale impostato per l'esecuzione dell'algoritmo
- `delta_max`: soglia massima impostata per l'esecuzione dell'algoritmo
- `delta_min`: soglia minima impostata per l'esecuzione dell'algoritmo

2.4 Dataset

Il dataset utilizzato per questo progetto è *mnist*, una vasta base di dati che contiene cifre scritte a mano, comunemente impiegata per addestrare sistemi di elaborazione delle immagini. In totale il dataset contiene 60000 immagini per l'addestramento e 10000 per il test. Le cifre sono state centrate in immagini di 28x28 pixel in scala di grigio. Il dato risultante è quindi una matrice di dimensione 60000x28x28 in cui ogni elemento è un valore da 0 a 255. Per gestire l'ottenimento delle immagini, è stata implementata la funzione **get_dataset**. Tramite questa funzione è possibile specificare quante immagini ottenere dal training set e dal test set. Per non ottenere ogni volta lo stesso sottoinsieme dal dataset, la funzione, prima di restituire il sottoinsieme, effettua uno *shuffle* degli elementi dell'intero dataset MNIST, tramite l'apposita funzione della libreria **sklearn** e poi restituisce i primi **n** elementi del training set e i primi **m** elementi del test set, specificandoli nei parametri della funzione. Questa funzione si occupa anche di convertire la matrice di dimensione 60000x28x28 del training set in un vettore di dimensione 60000x784 e la matrice del test set di dimensione 10000x28x28, in un vettore di dimensione 10000x784. Un'altra che funzionalità che offre è quello di eseguire lo scaling delle immagini, scalandole da un intervallo [0, 255] a [0, 1] tramite il parametro **scaled**. Di seguito, è mostrata la firma della funzione:

- `get_dataset(dimension_training_set, dimensione_test_set)`
    1. `dimension_training_set`: dimensione del training set che si vuole ottenere
    2. `dimensione_test_set`: dimensione del test set che si vuole ottenere
    3. `scaled`: valore booleano che indica l'esecuzione dello scaling delle immagini

Capitolo 3

Setup Sperimentale

In questo capitolo vengono descritte le impostazioni utilizzate per settare la rete e le modalità di apprendimento.
Tutte le reti utilizzate sono caratterizzate da:

- Uno strato di input, con dimensione 784 (che dipende dai dati del dataset)
- Uno strato interno formato da un numero di neuroni che varia a seconda del test.
- Uno strato di output formato da 10 neuroni, 1 per ogni possibile classe.

Tutti gli strati delle reti sono *full-connected* e sono progettati per effettuare un apprendimento di tipo *batch*.

Per lo strato interno, sono state utilizzate come funzioni di attivazione la **Sigmoide**, la **ReLu** e la **Leaky ReLu** mentre per lo strato di output è stata utilizzata la **Softmax**. I test effettuati hanno l'obiettivo di confrontare le prestazioni delle reti a seconda della regola di aggiornamento dei pesi utilizzata. Le prestazioni delle varie regole di aggiornamento, sono state misurate al variare di:

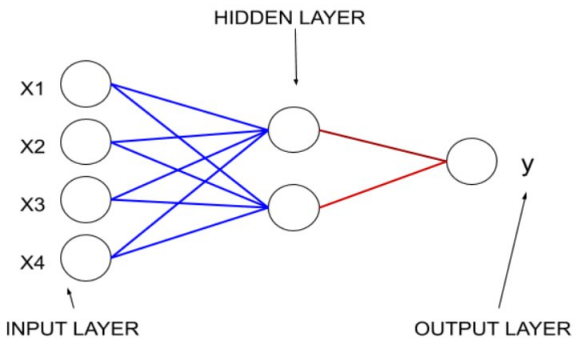


Figura 3.1: Schema delle reti utilizzate

- Numero di neuroni dello strato interno
- Funzione di attivazione utilizzata per lo strato interno
- Preprocessing sui dati
- Regola utilizzata per l'inizializzazione dei pesi degli strati della rete

La classica regola di aggiornamento dei pesi della *back-propagation* è stata messa a confronto con quelle descritte all'interno del paper fornito con la traccia. In particolare, sono state messe a confronto le seguenti regole di aggiornamento:

- Stochastic Gradient Descent
- Resilient Back Propagation (che chiameremo `RProp`$^-$)
- `RProp`$^+$
- `iRProp`$^-$
- `iRProp`$^+$

Le differenti versioni della *resilient back propagation* richiedono come iperparametri il passo incrementale (η^+), il passo decrementale (η^-) e i valori di soglia massima e minima. I valori di questi iperparametri sono stati presi dal paper fornito insieme alla traccia e settati a:

- $\eta^+ = 1.2$
- $\eta^- = 0.5$
- $\Delta_{max} = 50$
- $\Delta_{min} = 0$

Per quanto riguarda il dataset, sono stati prelevati 20000 campioni per il *training set* e 5000 per il *test set*. Per l'apprendimento, l'intero dataset viene immesso nella rete per ogni epoca caratterizzando un apprendimento di tipo batch. Per ciascuno dei metodi di apprendimento sono previste 1000 epoche.

3.1 Esecuzione esperimenti

Per l'esecuzione dei nostri esperimenti, sono state dichiarate delle funzioni che si occupano di impostare il setup degli esperimenti, nello specifico le funzioni usate sono:

- `execute_SGD`: per creare una rete shallow e addestrarla tramite l'algoritmo **Stochastic Gradient Descent**;
- `execute_RProp_minus`: per creare una rete shallow e addestrarla tramite l'algoritmo **RProp$^-$**;
- `execute_RProp_plus`: per creare una rete shallow e addestrarla tramite l'algoritmo **RProp$^+$**;
- `execute_iRProp_plus`: per creare una rete shallow e addestrarla tramite l'algoritmo **iRProp$^+$**;
- `execute_iRProp_minus`: per creare una rete shallow e addestrarla tramite l'algoritmo **iRProp$^-$**;

Queste funzioni, hanno lo scopo di creare una rete neurale *shallow* ed eseguire la procedura di training, sulla rete creata. Un altro scopo di queste funzioni, è quello di calcolare ad ogni epoca l'accuratezza e l'errore della rete neurale e testare il tutto tramite il test set del dataset MNIST. Queste funzioni prendono come parametri:

- `train_X`: è il dataset da utilizzare durante la fase di training;
- `train_Y`: sono le etichette del dataset utilizzato per la fase di training;
- `test_X`: è il dataset da utilizzare per testare le prestazioni della rete;
- `test_Y`: sono le etichette del dataset utilizzato per testare le prestazioni della rete;
- `epochs`: è il numero di epoche da eseguire durante la fase di training;
- `n_neurons`: è il numero di neuroni dello strato interno che la rete deve avere;
- `activation1`: è la funzione di attivazione da utilizzare nello strato interno della rete;
- `weight_init_rule="random"`: modalità da usare per inizializzare i pesi della rete neurale, di default il valore è impostato a "random";

3.2 Esecuzione dei test in parallelo

L'esecuzione degli esperimenti descritti ha richiesto, per ogni test, l'addestramento di 5 reti neurali (una per ognuna delle regole di aggiornamento dei pesi) su un training set di 20k elementi per 1000 epoche ciascuna. Per questo motivo ogni test richiedeva

una discreta quantità di tempo per il completamento. Per ovviare al problema è stato implementato uno script che permette l'esecuzione in parallelo dei test. Utilizzando 5 thread per eseguire i test, è stato possibile ridurre i tempi computazionali di circa il 65%. Passando da un tempo medio per un test con 150 neuroni di 804 secondi ad un tempo medio di 281 secondi. Per l'esecuzione in parallelo dei test è stato fatto uso della libreria `threading` di python.

Capitolo 4

Analisi dei Risultati

In questo capitolo saranno mostrati e analizzati i risultati ottenuti durante i test eseguiti.

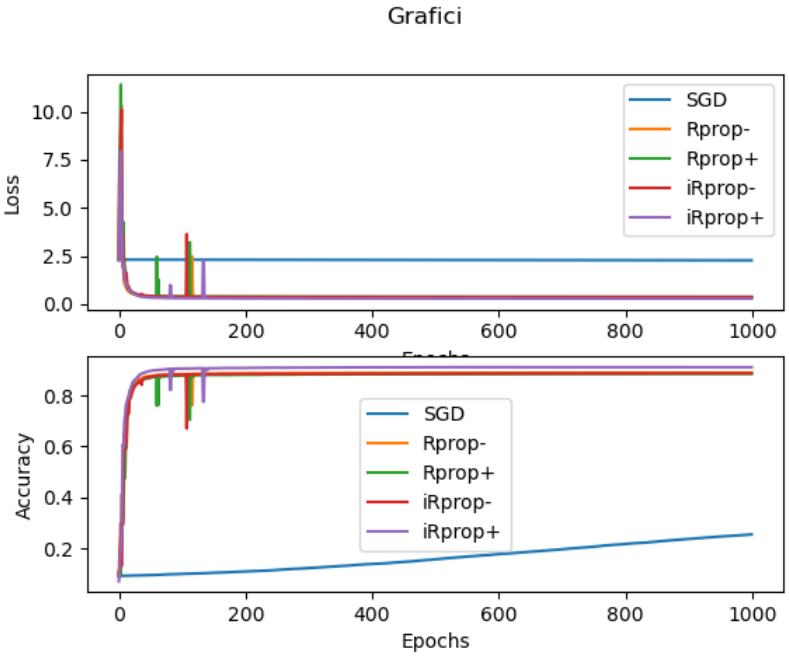


Figura 4.1: Risultati sul training set, rete da 64 neuroni con Sigmoide e inizializzazione random

4.1 Funzione di attivazione Sigmoide

Nei primi test è stata utilizzata la funzione di attivazione *sigmoide* così come descritto all'interno del paper. Inizialmente, i test sono stati eseguiti con un numero di neuroni dello strato interno pari a 64. La figura 4.1 mostra quanto la capacità di apprendimento del gruppo di regole della *resilient back-propagation* sia nettamente

TEST SET - 5k - 64 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.257	2.274
Rprop$^-$	0.865	0.453
Rprop$^+$	0.872	0.436
iRprop$^+$	0.867	0.431
iRprop$^-$	0.885	0.380

Tabella 4.1: Risultati sul test set, rete da 64 neuroni con Sigmoide e inizializzazione random.

superiore rispetto alla più classica discesa del gradiente, raggiungendo prestazioni eccellenti sul training set e convergendo dopo poco meno di 100 epoche. Per quanto riguarda la classica *back-propagation* con SGD, si evince un miglioramento costante delle prestazioni sul *training set* all'aumentare delle epoche che risulta essere però estremamente più lento rispetto alle regole alternative presentate. Nella tabella 4.1 sono mostrati i risultati che la rete ha ottenuto sul test set. I dati evidenziano, come atteso, prestazioni disastrose per la classica back-propagation che avrebbe richiesto un numero di epoche molto maggiore rispetto alle rivali per ottenere prestazioni comparabili. Per quanto riguarda le regole del gruppo *resilient back-propagation*, esse assumono un comportamento simile tra loro con una leggera superiorità della regola iRprop$^-$.

Nei test successivi è stato provato l'incremento dei neuroni dello strato interno da 64 a 150. La figura 4.2 mostra le prestazioni della rete al susseguirsi delle epoche. Ancora una volta le regole del gruppo *resilient back-propagation* mostrano una capacità di convergenza a prestazioni di accuratezza superiori al 90% entro le prime 50 epoche. Non sono tuttavia apprezzabili miglioramenti significativi delle prestazioni sul training set rispetto alla rete precedente che aveva meno della metà dei neuroni. Viceversa la SGD presenta un miglioramento nella velocità di crescita delle prestazioni all'aumentare delle epoche non riuscendo però ad ottenere ancora risultati soddisfacenti nel target di 1000 epoche. Per quanto riguarda le prestazioni sul test set, la tabella 4.2 mostra come con 150 neuroni le prestazioni siano migliorate leggermente raggiungendo con la regola iRprop$^-$ un'accuratezza del 90%.

Avendo raggiunto risultati soddisfacenti con 150 unità neurali, l'idea è stata quella di trovare il numero minimo di neuroni dopo il quale le prestazioni iniziano a degradare. Nella figura 4.3 sono mostrati i dati di training della rete con 32 neuroni. Anche se le regole del gruppo *resilient back-propagation* hanno mostrato una certa instabilità fino all'epoca 200, superato tale intervallo le prestazioni sul training set sono rimaste ottime. Per quanto riguarda l'SGD, come previsto, le prestazioni sono calate notevolmente. Il rendimento della rete sul test set è indicato nella tabella 4.3

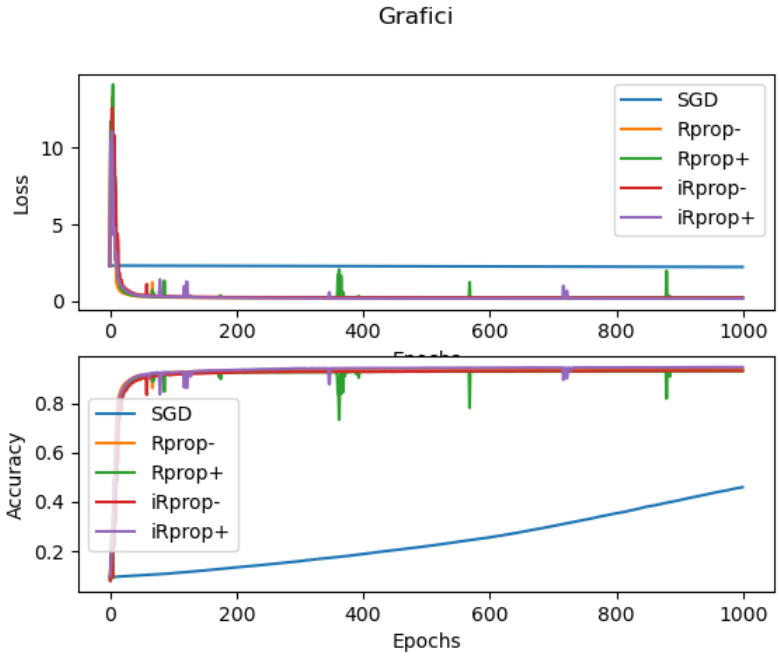


Figura 4.2: Risultati sul training set, rete da 150 neuroni con Sigmoide e inizializzazione random

TEST SET - 5k - 150 neuroni		
Regola	**Accuracy**	**Loss**
`SGD`	0.470	2.223
`Rprop`$^-$	0.887	0.456
`Rprop`$^+$	0.887	0.396
`iRprop`$^+$	0.884	0.432
`iRprop`$^-$	0.904	0.403

Tabella 4.2: Risultati sul test set, rete da 150 neuroni con Sigmoide e inizializzazione random.

che evidenzia una discesa nelle prestazioni anche per le regole del gruppo *resilient back-propagation*.

Diminuendo ancora il numero di neuroni le performance della rete calano ulteriormente portando a differenze nelle regole del gruppo *resilient back-propagation*. Come mostrato dalla figura 4.4, con 16 neuroni l'accuratezza sul training set è al di sotto del 80% per tutte le regole. In particolare la versione con *weight-backtracking* della *resilient back-propagation* converge fin da subito ad un'accuratezza di poco superiore al 60% e un valore di funzione d'errore superiore ad 1. Mentre la controparte senza *weight-backtracking* ottiene le prestazioni migliori superando leggermente le due varianti *improved* che invece si equivalgono.

L'ultimo test è stato effettuato con soli 8 neuroni. Il test, i cui dati sono mostrati in figura 4.5, mostra una differenza ancora maggiore tra le regole del gruppo *resilient*

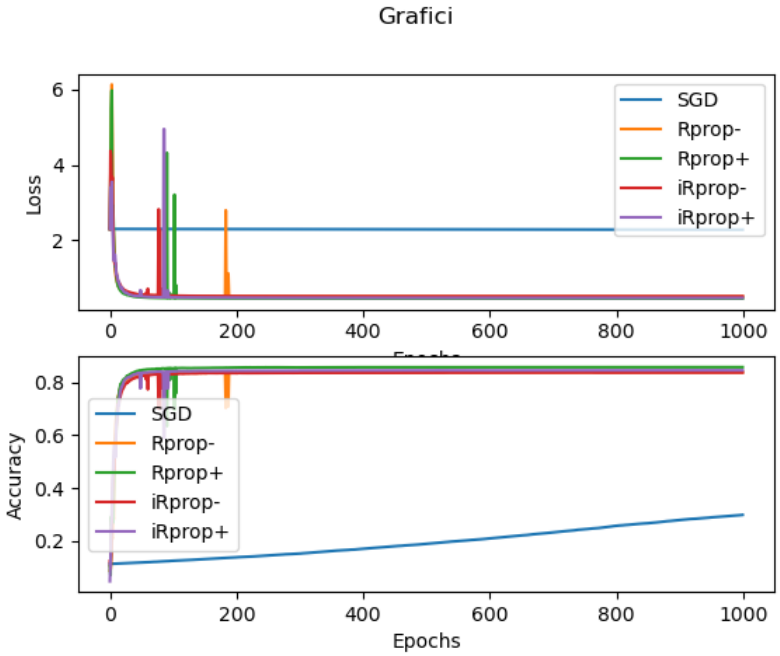


Figura 4.3: Risultati sul training set, rete da 32 neuroni con Sigmoide e inizializzazione random

TEST SET - 5k - 32 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.310	2.282
Rprop$^-$	0.820	0.589
Rprop$^+$	0.843	0.523
iRprop$^+$	0.826	0.573
iRprop$^-$	0.833	0.547

Tabella 4.3: Risultati sul test set, rete da 32 neuroni con Sigmoide e inizializzazione random.

back-propagation. Questa volta, la regola che perde meno performance è iRprop$^+$, mentre le regole non *improved* degradano con andamento simile. La peggiore delle quattro è quindi iRprop$^-$ che supera di poco un'accuratezza del 50% sul training set. Nella tabella 4.4 sono mostrati, solo per completezza, i dati relativi alle performance delle reti da 16 e 8 neuroni sul test set.

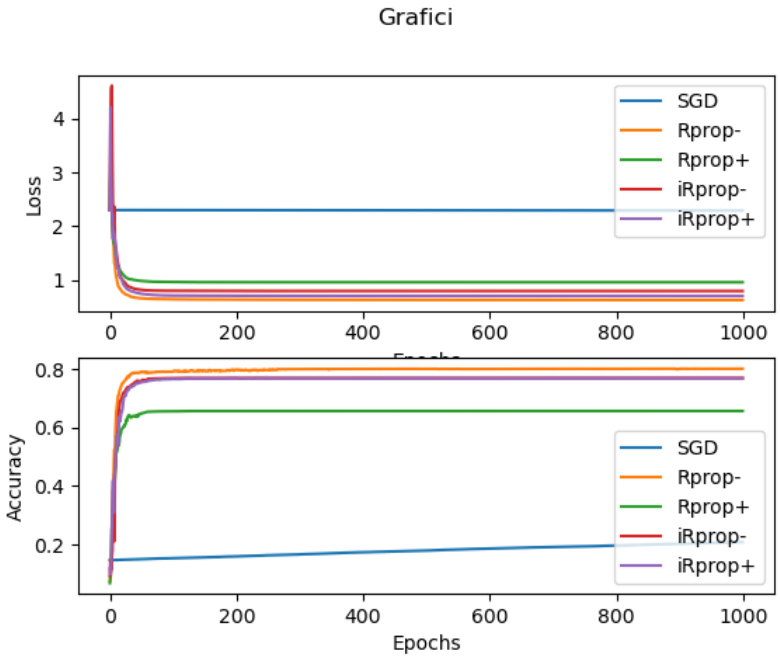


Figura 4.4: Risultati sul training set, rete da 16 neuroni con Sigmoide e inizializzazione random

TEST SET - 5k - 8 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.110	2.298
Rprop$^-$	0.596	1.164
Rprop$^+$	0.572	1.149
iRprop$^+$	0.503	1.309
iRprop$^-$	0.712	0.860

TEST SET - 5k - 16 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.199	2.293
Rprop$^-$	0.775	0.707
Rprop$^+$	0.637	1.020
iRprop$^+$	0.765	0.838
iRprop$^-$	0.759	0.759

Tabella 4.4: Risultati sul test set, reti rispettivamente da 8 e 16 neuroni con Sigmoide e inizializzazione random.

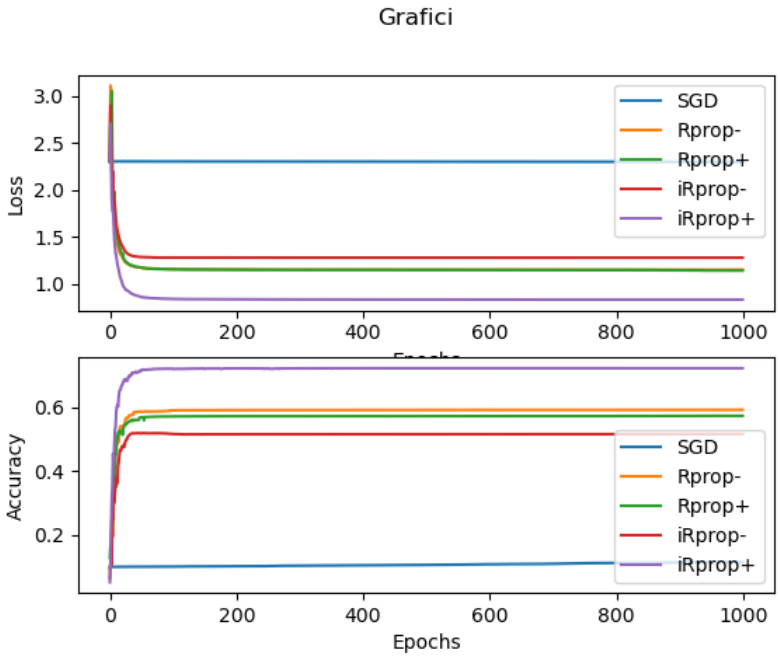


Figura 4.5: Risultati sul training set, rete da 8 neuroni con Sigmoide e inizializzazione random

4.2 Funzione di attivazione ReLu

In questa sezione saranno mostrati e analizzati i risultati e le performance delle differenti regole di aggiornamento dei pesi, al variare del numero di neuroni dello strato interno, utilizzando come funzione di attivazione la ReLu. Anche in questo caso il primo test è stato effettuato con 64 neuroni nello strato interno. Fin da subito i test hanno evidenziato una forte instabilità nelle performance delle regole del gruppo *resilient back-propagation*. Questa instabilità la si può notare nei grafici in figura 4.6 che mostra le performance sul training set di due reti identiche addestrate sugli stessi 20k elementi utilizzando la ReLu come funzione di attivazione dello strato interno. Nella prima rete la versione senza *weight-backtracking* della *resilient back-propagation* ottiene performance eccellenti sul training set con un'accuratezza superiore al 95%, mentre le altre versioni ottengono risultati estremamente deludenti. La stessa tendenza si riflette sulle performance relative al test set. Nella seconda rete, che ricordiamo ha le stesse caratteristiche della prima, le performance sono totalmente diverse. L'`Rprop`$^-$, che nella prima rete ha surclassato le altre, adesso ha le performance peggiori di tutte con un'accuratezza inferiore al 20%. Mentre la versione *improved* della *resilient back-propagation* con *weight-backtracking*, che nella prima rete aveva un'accuratezza inferiore al 40%, raggiunge adesso un'accuratezza superiore al 95%. Un fattore da notare è che il metodo SGD non risente di questa instabilità anzi, l'utilizzo della ReLu rispetto alla Sigmoide ha notevolmente migliorato le sue performance che adesso sono paragonabili a quelle del gruppo *resilient back-propagation*. Da notare (figura 4.7) che l'utilizzo di un maggior numero di neuroni non risolve il problema di questa instabilità portando ad ogni test risultati discordanti.

Le cause di questa instabilità sono dunque da ricercarsi nella struttura di base della rete e nel modo in cui viene fatta la *back-propagation* della ReLu. La prima cosa da notare è che ogni rete, quando istanziata, inizializza con valori casuali i pesi delle connessioni. Considerando poi che durante la fase di *back-propagation*, il calcolo della derivata nella ReLu porta a zero tutti i valori minori di 0, si ottiene la causa che porta a questa forte instabilità. Per ovviare al problema, i test sono stati eseguiti utilizzando la Leaky ReLu che sfrutta un iperparametro α per rendere i valori negativi molto vicini allo zero ma non nulli.

4.3 Funzione di attivazione Leaky ReLu

In questa sezione vedremo i risultati ottenuti utilizzando questa particolare versione della ReLu, confrontando le differenti regole di aggiornamento dei pesi della rete al variare del numero di neuroni dello strato interno.

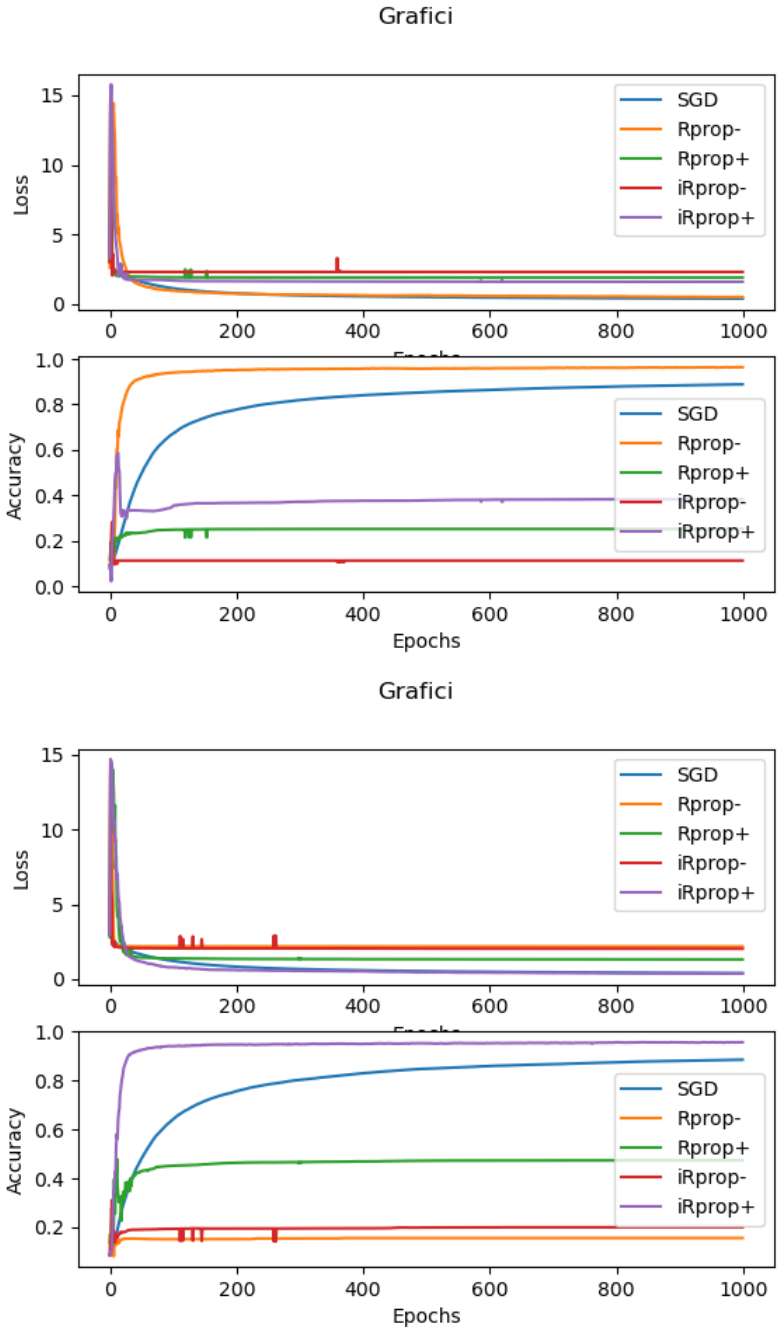


Figura 4.6: Risultati di due reti identiche addestrate sugli stessi 20k elementi.

Il primo test è stato realizzato utilizzando 64 neuroni per lo strato interno. La figura 4.8 mostra come con un numero veramente ridotto di epoche, tutte le regole del gruppo *resilient back-propagation* ottengono performance elevatissime con accuratezza sul training set vicina al 100%. Per quanto riguarda la SGD, essa mantiene le performance migliori ottenute con il passaggio da Sigmoide a ReLu non riuscendo però ad eguagliare le prestazioni delle regole del gruppo *resilient back-propagation*. Le prestazioni sul test set sono mostrate nella tabella 4.5. I dati mostrano che

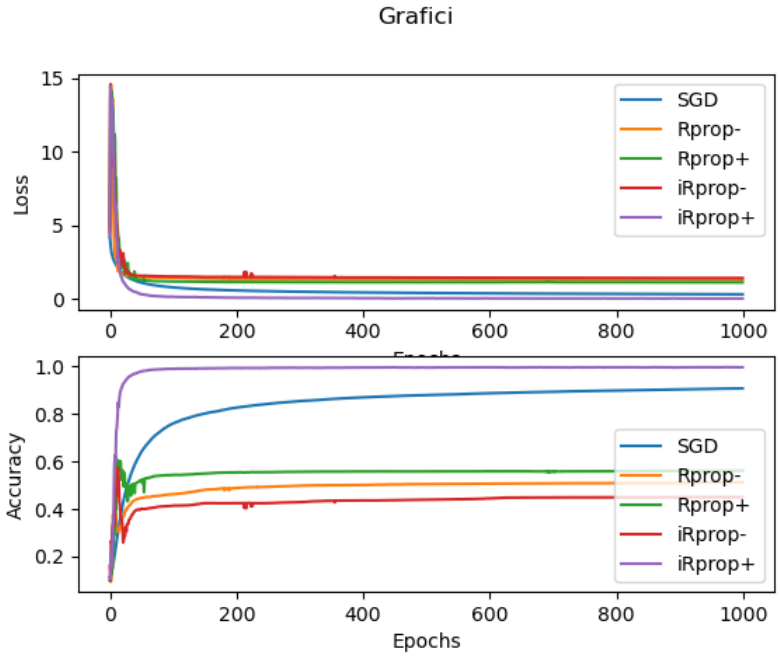


Figura 4.7: Risultati sul training set, rete da 200 neuroni con ReLu e inizializzazione random

TEST SET - 5k - 64 neuroni		
Regola	**Accuracy**	**Loss**
`SGD`	0.894	0.358
`Rprop`$^-$	0.912	1.413
`Rprop`$^+$	0.918	1.320
`iRprop`$^+$	0.910	1.455
`iRprop`$^-$	0.905	1.530

Tabella 4.5: Risultati sul test set, rete da 64 neuroni con LReLu e inizializzazione random.

tutte le versioni della *resilient back-propagation* offrono le stesse prestazioni, con differenze minime tra loro. Mentre l'SGD migliora nettamente le proprie prestazioni arrivando, per la prima volta, ad essere paragonabile alle sue rivali.

Aumentando il numero di neuroni a 150 si ottiene un leggero incremento nelle prestazioni sul training set, come mostrato dalla figura 4.9. Anche le prestazioni sul test set, mostrate nella tabella 4.6, migliorano ma non in modo tale da giustificare l'aumento di neuroni. In questa configurazione, le regole del gruppo *resilient back-propagation* offrono prestazioni effettivamente di poco superiori alla SGD. La versione `Rprop`$^+$ si distingue con un'accuratezza sul test set di poco superiore al 93%, superando le altre regole di poco più di un punto percentuale. La SGD si mantiene su un'accuratezza di poco superiore al 90% mostrando un valore della funzione loss nettamente più basso delle rivali.

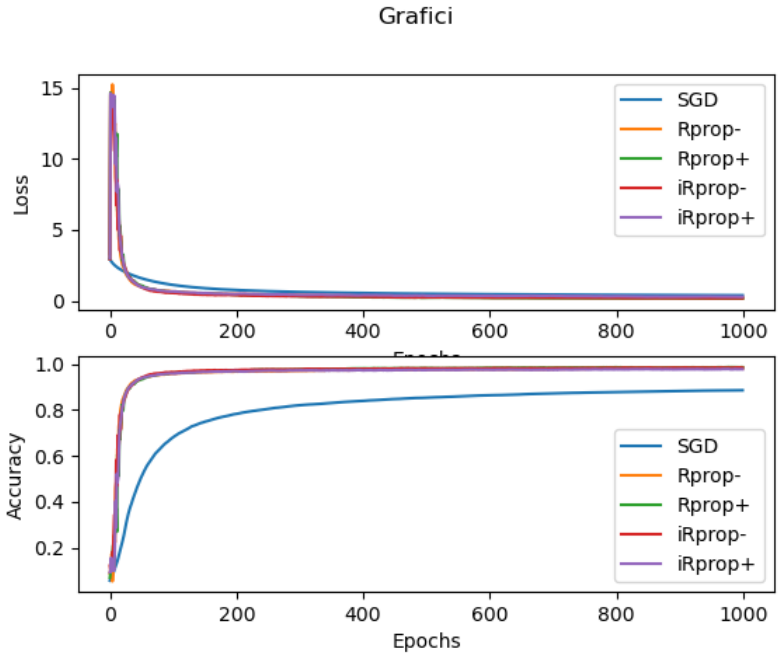


Figura 4.8: Risultati sul training set, rete da 64 neuroni con LReLu e inizializzazione random

TEST SET - 5k - 150 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.907	0.316
Rprop$^-$	0.914	1.388
Rprop$^+$	0.931	1.109
iRprop$^+$	0.924	1.227
iRprop$^-$	0.928	1.164

Tabella 4.6: Risultati sul test set, rete da 150 neuroni con LReLu e inizializzazione random.

Anche in questo caso sono stati fatti dei test diminuendo drasticamente il numero di neuroni per verificare il numero minimo dopo il quale le prestazioni si deteriorano. La figura 4.10 mostra i risultati sul training set ottenuti da una rete con 32 neuroni addestrata sul training set con 20k elementi utilizzando, come funzione di attivazione dello strato interno, la *Leaky ReLu*. I dati mostrano come passando da 150 a 32 neuroni, le regole del gruppo *resilient back-propagation* mantengono le loro prestazioni quasi del tutto inalterate. L'unica a risentirne, specialmente nelle prime 500 epoche, è la SGD. Sul test set si verifica una situazione analoga con una perdita di accuratezza di pochi punti percentuale per tutte le regole.

Come mostrato dalla figura 4.11, sono stati effettuati anche test con 16 e 8 neuroni che hanno evidenziato un calo generale delle prestazioni pur non drammatico per quanto riguarda le regole del gruppo *resilient back-propagation*. Sul test set invece,

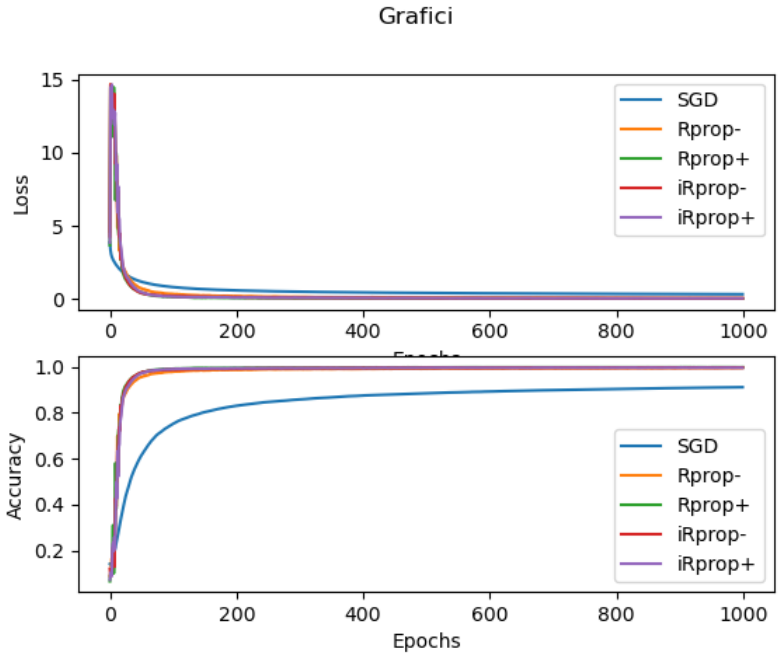


Figura 4.9: Risultati sul training set, rete da 150 neuroni con LReLu e inizializzazione random.

TEST SET - 5k - 32 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.882	0.400
Rprop$^-$	0.895	1.666
Rprop$^+$	0.891	1.730
iRprop$^+$	0.885	1.807
iRprop$^-$	0.896	1.665

Tabella 4.7: Risultati sul test set, rete da 32 neuroni con LReLu e inizializzazione random.

le prestazioni rimangono simili per tutte le regole con un degrado nell'accuratezza che si aggira intorno al 83% per la rete con soli 8 neuroni.

4.4 Problema della ReLu

Come descritto nella sezione 4.2, le regole del gruppo *resilient back-propagation*, utilizzando come funzione di attivazione dello strato interno la ReLu, hanno un comportamento aleatorio con prestazioni estremamente variabili in base all'inizializzazione casuale dei pesi della rete. Per risolvere il problema è stato scelto di utilizzare un algoritmo di inizializzazione dei pesi differente, detto *Xavier's initialization*. In questa particolare regola di inizializzazione, tutti i *bias* sono settati a

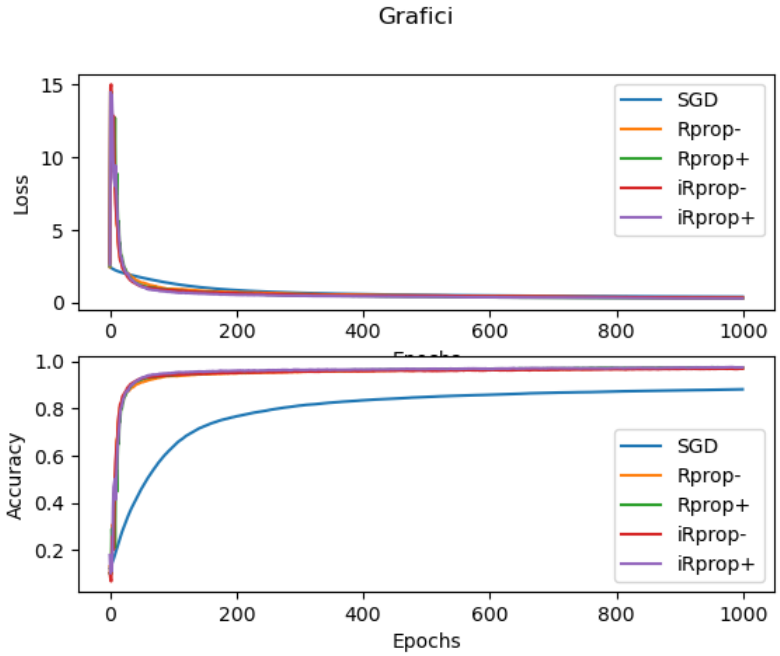


Figura 4.10: Risultati sul training set, rete da 32 neuroni con LReLu e inizializzazione random.

TEST SET - 5k - 8 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.832	0.546
$\texttt{Rprop}^-$	0.852	1.121
$\texttt{Rprop}^+$	0.842	1.766
$\texttt{iRprop}^+$	0.856	1.633
$\texttt{iRprop}^-$	0.830	1.938

TEST SET - 5k - 16 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.862	0.464
$\texttt{Rprop}^-$	0.878	1.486
$\texttt{Rprop}^+$	0.885	1.755
$\texttt{iRprop}^+$	0.875	1.960
$\texttt{iRprop}^-$	0.882	1.717

Tabella 4.8: Risultati sul test set, reti di rispettivamente 8 e 16 neuroni con LReLu e inizializzazione random.

zero mentre i pesi sono inizializzati come:

$$W_{ij} \sim U\left[-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\right]$$

Dove U indica la distribuzione uniforme, fan_{in} è la dimensione del layer precedente e fan_{out} indica la dimensione del layer attuale. Sono stati quindi effettuati diversi test inizializzando i pesi con questa regola ed utilizzando come funzione di attivazione del layer interno la ReLu. Il test hanno mostrato risultati estremamente deludenti anche con un notevole numero di neuroni. Nella figura 4.12 è possibile vedere i dati delle prestazioni sul training set che mostrano il problema dell'aleatorietà dei risultati già visto in precedenza.

TEST SET - 5k - 64 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.774	3.632
Rprop$^-$	0.576	1.361
Rprop$^+$	0.696	1.320
iRprop$^+$	0.395	1.597
iRprop$^-$	0.551	1.404

Tabella 4.9: Risultati sul test set, rete da 64 neuroni con ReLu e *Xavier's Initialization* 2.

4.4.1 Adattamento della Xavier's Initialization

Considerando che la *Xavier's Initialization* classica non risolveva il problema della ReLu, sono stati effettuati ulteriori test con una versione di questa regola implementata all'interno della libreria di *tensor flow*. Per poterla utilizzare abbiamo adattato l'algoritmo in modo da funzionare tramite *numpy*. In questo modo si ottiene un' inizializzazione che setta tutti i *bias* a 0 e tutti i pesi come segue:

$$W_{ij} \sim U\left[-\sqrt{\frac{3}{\max\{1, \frac{fan_{in}+fan_{out}}{fan_{in}}\}}}, \sqrt{\frac{3}{\max\{1, \frac{fan_{in}+fan_{out}}{fan_{in}}\}}}\right]$$

I test effettuati con la versione adattata della *Xavier's Initialization*, hanno mostrato risultati interessanti. Il primo test è stato effettuato come di consueto utilizzando una rete con 64 neuroni nello strato interno con funzione di attivazione ReLu. Nella figura 4.13 sono mostrati i dati sul training set che risultano essere nuovamente deludenti. Ancora una volta le regole del gruppo *resilient back-propagation* assumono un atteggiamento aleatorio nei test. Come prevedibile, sul test set le performance sono altrettanto mediocri come mostrato dalla tabella 4.9. Tuttavia un fenomeno interessante si verifica all'aumentare del numero di neuroni. Infatti i test hanno mostrato che portando il numero di neuroni della rete fino a 100 le prestazioni rimangono deludenti. Eppure a partire da 110 neuroni, la rete con la funzione di attivazione ReLu, inizializzata con la *Xavier's Initialization* revisionata, comincia ad ottenere performance elevate.

La figura 4.14 mostra le prestazioni sul training set ottenute con 110 neuroni che sono decisamente più elevate ma mostrano ancora un minimo di aleatorietà. Problema che è stato totalmente risolto portando il numero di neuroni dello strato interno a 150. Infatti in quest'ultima configurazione la rete ottiene prestazioni eccellenti con tutte le regole del gruppo *resilient back-propagation*. Anche la SGD migliora all'aumentare del numero di neuroni pur non essendo essa affetta dal problema dell'inizializzazione dei pesi.

TEST SET - 5k - 110 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.801	3.211
Rprop$^-$	0.903	1.541
Rprop$^+$	0.901	1.597
iRprop$^+$	0.889	1.682
iRprop$^-$	0.711	1.304

TEST SET - 5k - 150 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.831	2.718
Rprop$^-$	0.911	1.432
Rprop$^+$	0.906	1.518
iRprop$^+$	0.915	1.637
iRprop$^-$	0.906	1.511

Tabella 4.10: Risultati sul test set, reti di rispettivamente 110 e 150 neuroni con ReLu e *Xavier's Initialization* 2.

TEST SET - 5k - 150 neuroni		
Regola	**Accuracy**	**Loss**
SGD	0.924	0.264
Rprop$^-$	0.915	1.248
Rprop$^+$	0.909	1.331
iRprop$^+$	0.903	1.484
iRprop$^-$	0.913	1.335

Tabella 4.11: Risultati sul test set normalizzato, rete da 150 neuroni con ReLu e inizializzazione random.

4.4.2 Normalizzazione del Dataset

I test eseguiti, fin'ora descritti, erano fatti utilizzando il dataset così com'è. Quindi ogni immagine del dataset risultava in un vettore di 784 elementi appartenenti all'intervallo $[0, 255]$. Nonostante la grande differenza nei valori del dataset, le reti hanno sempre ottenuto buoni risultati a parte per quelle, come detto precedentemente, che utilizzavano la funzione di attivazione ReLu. É stato dunque deciso di effettuare alcuni test applicando una normalizzazione al dataset. Per i test che seguono ogni elemento del dataset è stato normalizzato in modo che i 784 elementi di ogni vettore appartenessero all'intervallo $[0, 1]$. Per farlo è stata applicata una divisione per 255 ad ognuno degli elementi. Per mantenere la consistenza, nei test che seguono, il *learning rate* della SGD è stato incrementato a 0.1. Dai test eseguiti è emerso che la normalizzazione riesce da sola a risolvere il problema della ReLu che torna a performare in maniera ottimale anche con l'inizializzazione dei pesi random, come mostrato dalla figura 4.15. I dati sul test set (tabella 4.11) mostrano che in questa configurazione, nonostante le performance sul training set siano peggiori, la SGD performa meglio delle regole del gruppo *resilient back-propagation*.

Sono stati effettuati dei test con il dataset normalizzato anche per le funzioni di attivazione LReLu e Sigmoide. I test (tabella 4.12) hanno mostrato un andamento della LReLu leggermente migliore rispetto a quello della ReLu, mentre le prestazioni sono state inferiori per quanto riguarda la Sigmoide.

TEST SET - 5k - LReLu		
Regola	**Accuracy**	**Loss**
SGD	0.923	0.274
Rprop$^-$	0.929	1.040
Rprop$^+$	0.927	1.065
iRprop$^+$	0.923	1.176
iRprop$^-$	0.931	1.065

TEST SET - 5k - Sigmoide		
Regola	**Accuracy**	**Loss**
SGD	0.881	0.463
Rprop$^-$	0.874	1.681
Rprop$^+$	0.865	1.880
iRprop$^+$	0.881	1.787
iRprop$^-$	0.871	1.888

Tabella 4.12: Risultati sul test set normalizzato, rete da 150 neuroni con LReLu e Sigmoide ripettivamente.

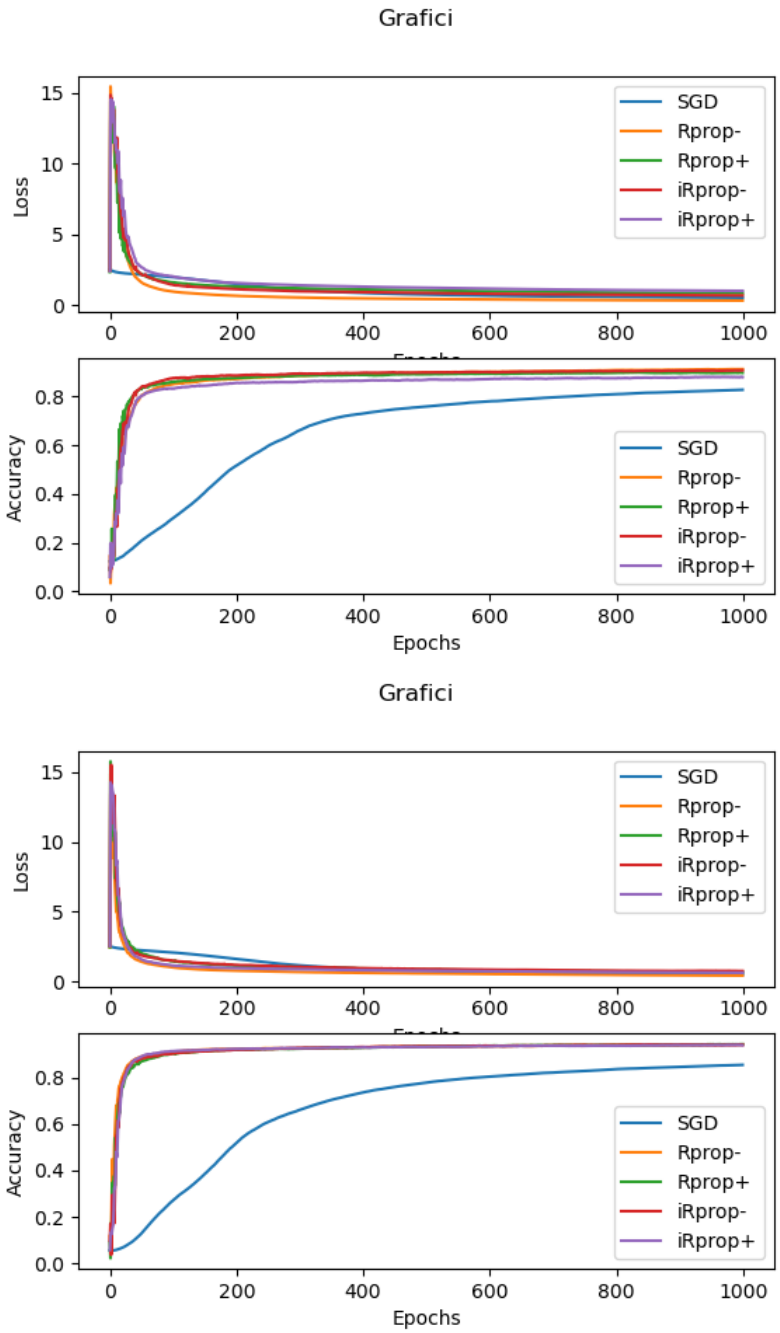


Figura 4.11: Risultati sul training set, reti da rispettivamente 8 e 16 neuroni con LReLu e inizializzazione random.

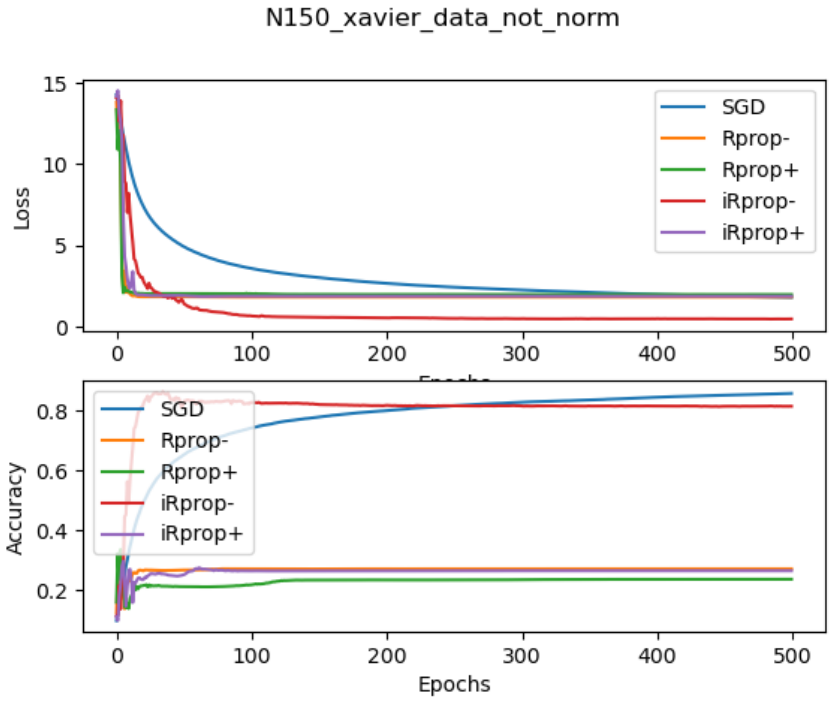


Figura 4.12: Risultati sul training set, rete da 150 neuroni con ReLu e *Xavier's Init* 1.

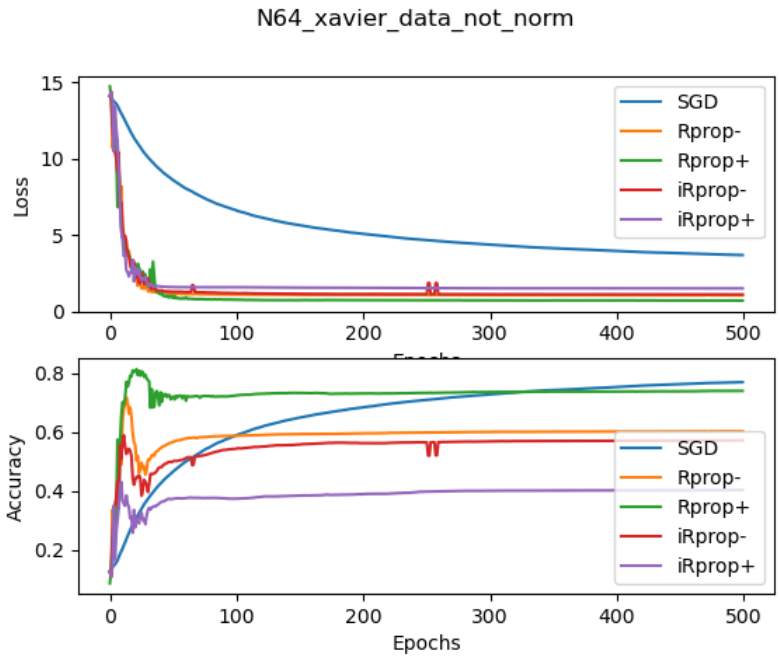


Figura 4.13: Risultati sul training set, rete da 150 neuroni con ReLu e *Xavier's Init* 2.

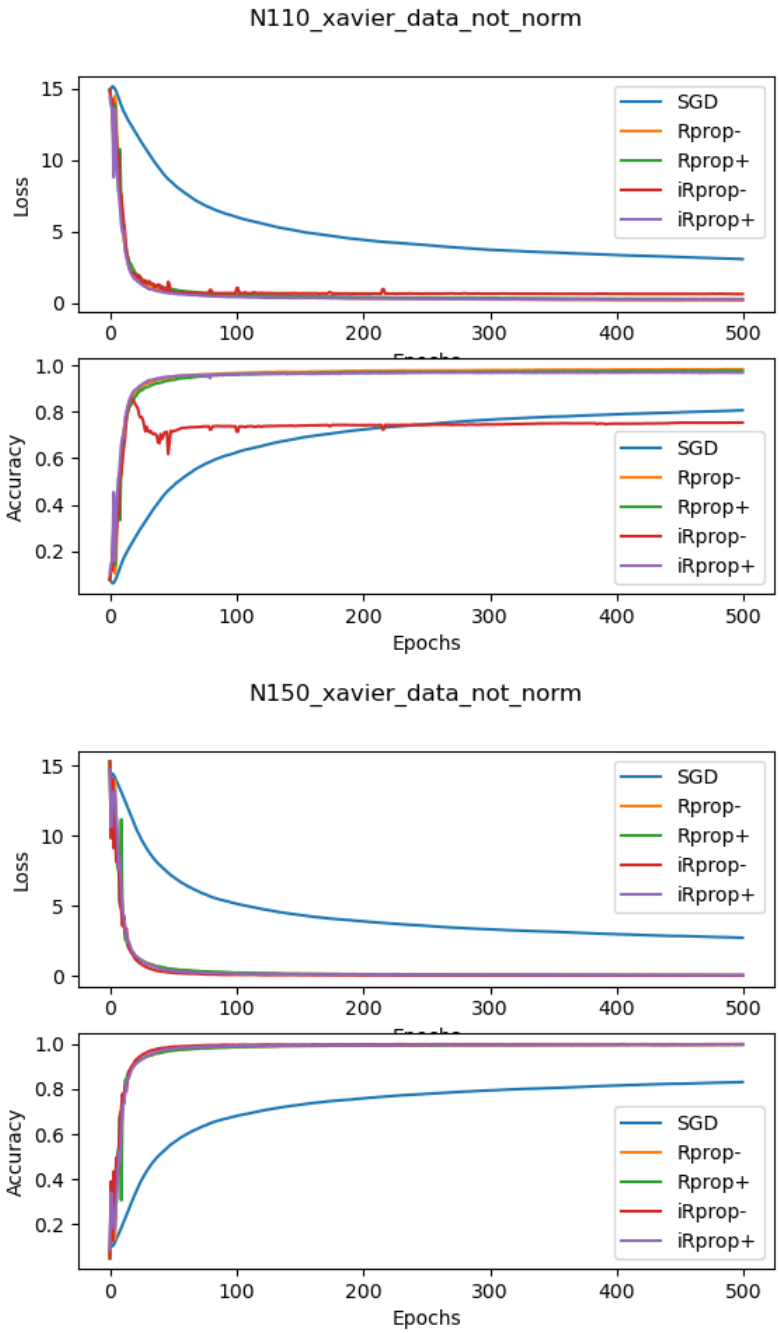


Figura 4.14: Risultati sul training set, redi di respettivamente 110 e 150 neuroni con ReLu e *Xavier's Init* 2.

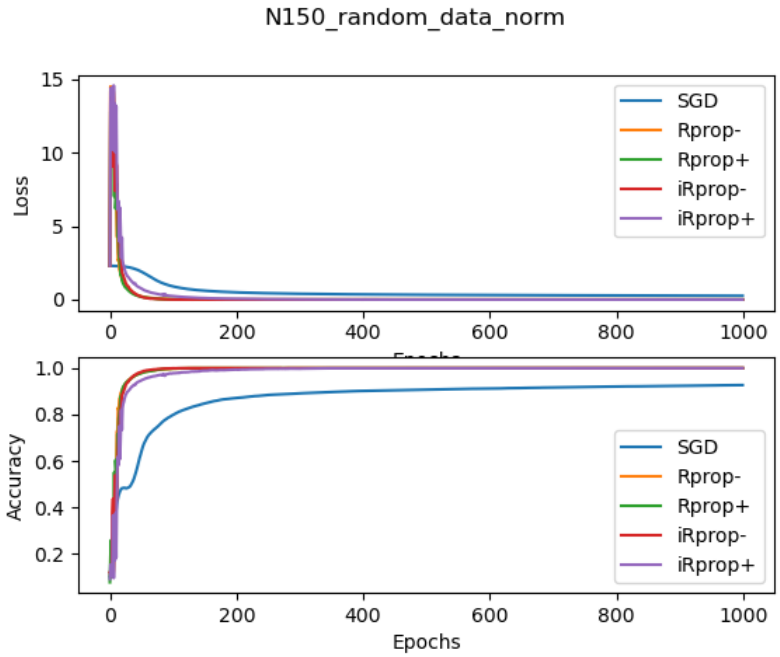


Figura 4.15: Risultati sul training set normalizzato, rete da 150 neuroni con ReLu e inizializzazione random.

4.5 Le regole a confronto

In questa sezione saranno messe a confronto tra loro le regole di inizializzazione dei pesi utilizzando i dati raccolti dai test eseguiti. Per ottenere i dati mostrati, alle reti neurali che erano state addestrate su dataset di 20k elementi, sono stati sottoposti test set di 5k elementi. Complessivamente i test sono divisi in 12 configurazioni di reti differenti, ognuna delle quali comprende una rete neurale per ogni regola di aggiornamento dei pesi. Di seguito saranno descritte le 12 configurazioni di reti in modo da facilitare la lettura dei grafici in figura 4.16.

- `lrelu_xavier_norm`: Reti con pesi inizializzati secondo la *Xavier's Init* che utilizzano come funzione di attivazione dello strato interno la LReLu e alle quali è stato sottoposto il dataset normalizzato.

- `lrelu_xavier_not_norm`: Reti con pesi inizializzati secondo la *Xavier's Init* che utilizzano come funzione di attivazione dello strato interno la LReLu e alle quali è stato sottoposto il dataset così com'è.

- `lrelu_random_norm`: Reti con pesi inizializzati con distribuzione uniforme che utilizzano come funzione di attivazione dello strato interno la LReLu e alle quali è stato sottoposto il dataset normalizzato.

- `lrelu_random_not_norm`: Reti con pesi inizializzati con distribuzione uniforme che utilizzano come funzione di attivazione dello strato interno la LReLu e alle quali è stato sottoposto il dataset così com'è.

- `relu_xavier_norm`: Reti con pesi inizializzati secondo la *Xavier's Init* che utilizzano come funzione di attivazione dello strato interno la ReLu e alle quali è stato sottoposto il dataset normalizzato.

- `relu_xavier_not_norm`: Reti con pesi inizializzati secondo la *Xavier's Init* che utilizzano come funzione di attivazione dello strato interno la ReLu e alle quali è stato sottoposto il dataset così com'è.

- `relu_random_norm`: Reti con pesi inizializzati con distribuzione uniforme che utilizzano come funzione di attivazione dello strato interno la ReLu e alle quali è stato sottoposto il dataset normalizzato.

- `relu_random_not_norm`: Reti con pesi inizializzati con distribuzione uniforme che utilizzano come funzione di attivazione dello strato interno la ReLu e alle quali è stato sottoposto il dataset così com'è.

- `sigm_xavier_norm`: Reti con pesi inizializzati secondo la *Xavier's Init* che utilizzano come funzione di attivazione dello strato interno la Sigmoide e alle quali è stato sottoposto il dataset normalizzato.
- `sigm_xavier_not_norm`: Reti con pesi inizializzati secondo la *Xavier's Init* che utilizzano come funzione di attivazione dello strato interno la Sigmoide e alle quali è stato sottoposto il dataset così com'è.
- `sigm_random_norm`: Reti con pesi inizializzati con distribuzione uniforme che utilizzano come funzione di attivazione dello strato interno la Sigmoide e alle quali è stato sottoposto il dataset normalizzato.
- `sigm_random_not_norm`: Reti con pesi inizializzati con distribuzione uniforme che utilizzano come funzione di attivazione dello strato interno la Sigmoide e alle quali è stato sottoposto il dataset così com'è.

Come si può notare dai grafici, i test non hanno evidenziato particolari differenze tra le regole di aggiornamento nemmeno al variare di inizializzazione dei pesi o preprocessing sui dati, a meno di alcune eccezioni che hanno mostrato prestazioni estremamente scadenti. L'accuratezza maggiore sul test set è stata raggiunta dalla rete `lrelu_random_not_norm` con la regola di aggiornamento *resilient back-propagation* con *weight back-tracking*. Mentre le prestazioni generali leggermente migliori le ha ottenute la rete `lrelu_random_norm` con la regola di aggiornamento SGD che ha ottenuto un'accuratezza di 0.923 e un valore della loss di appena 0.274. Per quanto riguarda l'andamento generale, la Sigmoide ha mostrato in tutte le varianti prestazioni leggermente inferiori.

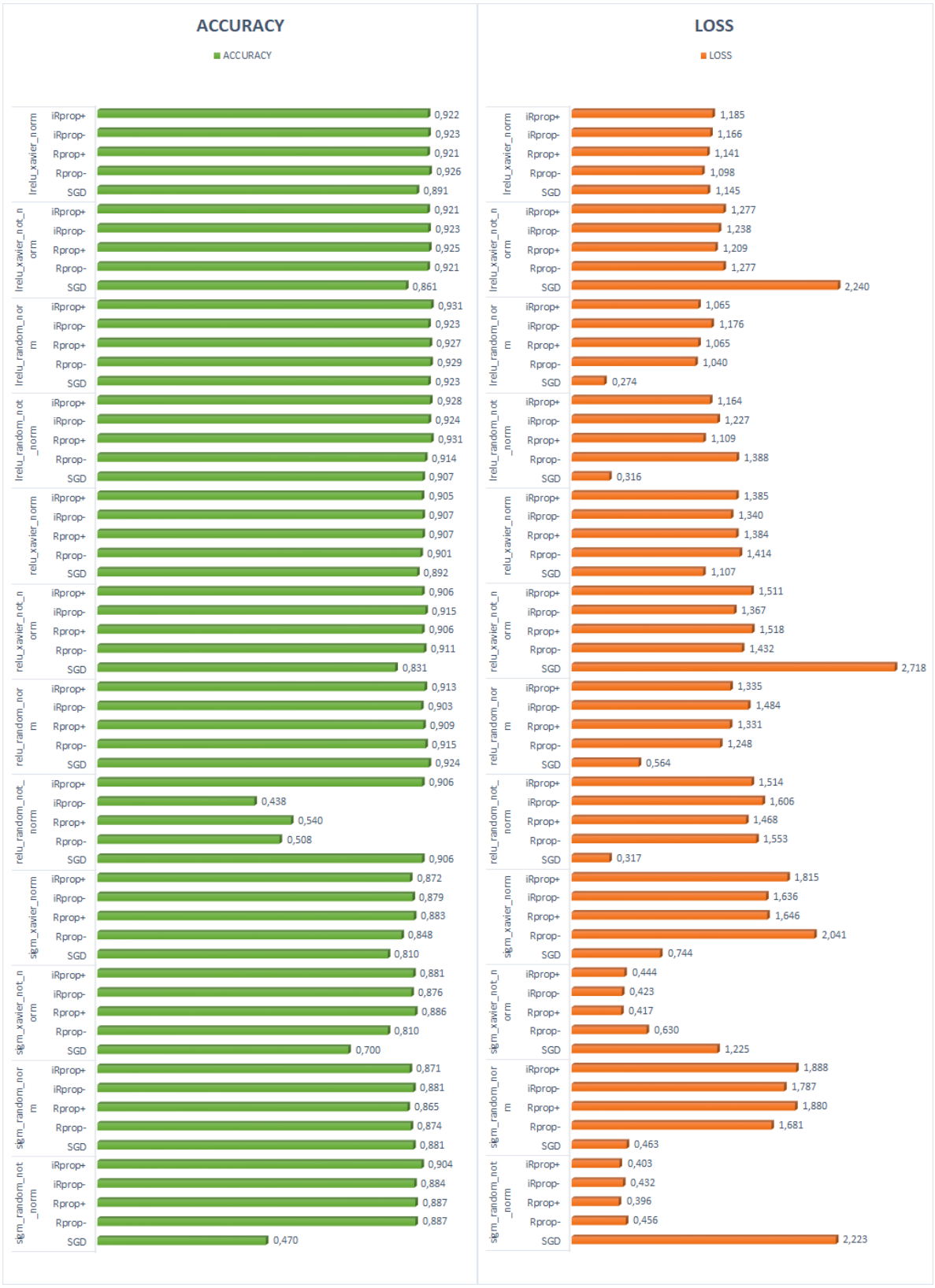


Figura 4.16: Risultati generali di tutte le reti testate.

4.6 Test con immagini esterne

Sono stati condotti alcuni test su determinate reti addestrate per verificare la capacità di riconoscere cifre di immagini non appartenenti al dataset. In particolare sono state realizzate 10 immagini 28x28 pixel a 24 bit rappresentanti ognuna delle 10 cifre numeriche. Utilizzando la libreria *open cv* di python, le immagini sono state portate ad array di 784 elementi compresi tra 0 e 255 proprio come quelli del dataset. É stato poi realizzato uno script che effettua il training di una rete utilizzando i dati del dataset MNIST e, al termine, oltre a verificare le prestazioni sul test set, ritorna un oggetto `NeuralNetwork` che contiene tutti gli elementi della rete addestrata e permette di effettuare il *forward* di ulteriori elementi e vederne i risultati. Questo oggetto è stato utilizzato per somministrare alla rete gli array relativi alle cifre create a mano. Il test ha mostrato risultati interessanti. Infatti, nonostante le immagini fossero realizzate pixel per pixel manualmente e non godessero delle stesse pre-elaborazioni fatte sul dataset MNIST, la rete si è dimostrata particolarmente capace a riconoscere le cifre 0, 1, 2, 3, 5 e 9. Mentre in nessuno dei test è stato possibile fargli riconoscere le cifre 4 e 8.

Capitolo 5

Conclusioni e sviluppi futuri

Lo sviluppo di questo progetto ha l'obiettivo di porre le basi riguardo alla progettazione e all'implementazioni di reti neurali. Nel nostro caso specifico, la progettazione ha riguardato la categoria delle reti neurali **Shallow** di tipo **Feed-Forward** per risolvere un problema di classificazione. Nei nostri esperimenti è stato fatto variare il numero di neuroni dello strato interno, è stato fatto variare il tipo di inizializzazione dei pesi ed è stata fatta variare la funzione dello strato interno. In particolar modo, si è posta maggior attenzione alle regole di aggiornamento. Sono state confrontate le prestazioni dell'**SGD** e dell'**RProp** con tutte le varianti di quest'ultimo elencate nel paper: "Empirical evaluation of the improved Rprop learning algorithms, Christian Igel, Michael Husken, neurocomputing, 2003". I test sono stati effettuati combinando tutti questi elementi, ottenendo diverse combinazioni di implementazioni, così da avere un confronto completo di questi algoritmi. La libreria è stata sviluppata in modo tale da poter essere estendibile, per essere utilizzata in impieghi futuri ed è possibile applicare facilmente, a una rete neurale di questa libreria, altre regole di aggiornamento una volta implementate, se si volessero condurre altri esperimenti. Utilizzando questo libreria sarebbe anche possibile condurre esperimenti su reti neurali **Deep** di tipo **Feed-Forward**. Non sono stati condotti esperimenti con reti **Deep**, perché non richiesto nella traccia scelta.