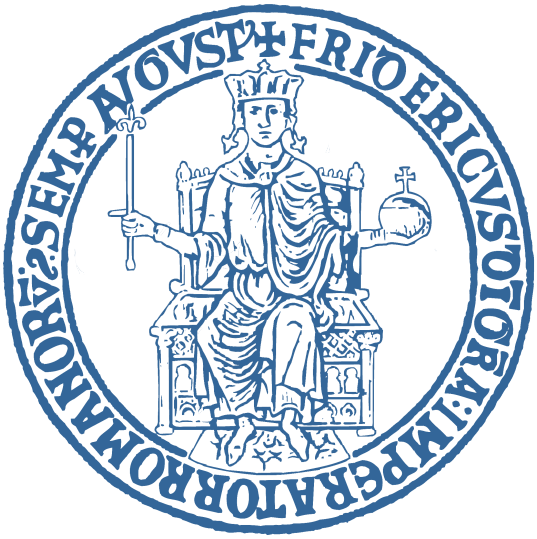


Università degli Studi di Napoli Federico II

Relazione progetto Neural Networks and Deep Learning

2022/2023

Professore	*Studenti*
Roberto Prevete	Raffaele Di Maso Vincenzo Tramo

Indice

Introduzione

Il presente progetto si focalizza sulla progettazione e implementazione di una neural network utilizzando Python, seguendo principalmente un approccio orientato agli oggetti. L'obiettivo principale è sviluppare un set di strumenti per implementare e addestrare fully connected neural networks, con l'obiettivo successivo di confrontare diverse varianti dell'algoritmo di ottimizzazione RProp (Resilient Backpropagation) presentate e discusse nell'articolo "Empirical evaluation of the improved Rprop learning algorithms" di Christian Igel e Michael Hüsken.

La relazione è suddivisa in tre parti:

- Contestualizzazione e analisi delle componenti principali di una rete neurale: definizione di pseudo-codici che descrivono una prima versione del comportamento della propagazione in avanti e all'indietro.
- Progettazione e implementazione di una libreria per la realizzazione e l'addestramento di una rete neurale completamente connessa.
- Implementazione e analisi della regola di aggiornamento RProp e varianti, con commento sui confronti e risultati ottenuti.

Il progetto è ospitato su una repository pubblica su GitHub, accessibile tramite il seguente link: `https://github.com/vtramo/neural-networks-experiments-from-scratch`

Nella repository, è possibile trovare tutte le implementazioni delle reti neurali sviluppate da zero utilizzando il linguaggio di programmazione Python. Le implementazioni includono fully-connected neural networks, l'algoritmo di back-propagation e le varie varianti dell'algoritmo RProp.

Pseudocodice algoritmi di forward propagation e back-propagation

1 Algoritmi di forward propagation e back-propagation

Questo capitolo presenta i due algoritmi fondamentali del progetto: forward propagation e back-propagation. Viene fornito lo pseudocodice di entrambi gli algoritmi per illustrarne il funzionamento in modo astratto. Nella Sezione 2, invece, saranno presentati i dettagli implementativi in Python di questi algoritmi, fornendo un'implementazione concreta delle operazioni descritte nei pseudocodici.

1.1 Forward propagation

I due pseudocodici sotto riportati descrivono gli algoritmi di forward propagation utilizzati nella rete neurale.

L'algoritmo di forward propagation è utilizzato per calcolare l'output di una rete neurale dato un'input $\mathbf{x}$ (fornito come input all'algoritmo). La rete neurale ha una profondità l e il vettore $\mathbf{W}^{(i)}$ rappresenta la matrice dei pesi del layer i, mentre $\mathbf{b}^{(i)}$ rappresenta i bias del layer i. Il processo di forward propagation inizia inizializzando $\mathbf{z}^{(0)}$ con l'input $\mathbf{x}$. Successivamente, per ogni layer k da 1 a l, vengono eseguite le seguenti operazioni:

1. Calcolo dell'input pesato $\mathbf{a}^{(k)}$ del layer k come la somma tra il bias $\mathbf{b}^{(k)}$ del layer k e il prodotto tra la matrice dei pesi $\mathbf{W}^{(k)}$ del layer k e l'output $\mathbf{z}^{(k-1)}$ del layer precedente $k-1$;
2. Applicazione della funzione di attivazione $g(\cdot)$ all'input pesato $\mathbf{a}^{(k)}$ per ottenere l'output $\mathbf{z}^{(k)}$ del layer k.

Il processo viene ripetuto per tutti i layer della rete, fino a raggiungere l'ultimo layer l. L'output finale della forward propagation è $\mathbf{z}^{(l)}$, che rappresenta l'output della rete neurale per l'input $\mathbf{x}$.

Algoritmo 1 Algoritmo di Forward Propagation

Input: Profondità rete l
Input: $\mathbf{W}^{(i)}, i \in \{1, \ldots, l\}$, la matrice dei pesi del modello
Input: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$, i bias del modello
Input: $\mathbf{x}$, l'input da processare
  $\mathbf{z}^{(0)} = \mathbf{x}$
  for $k = 1$ to l **do**
    $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \times \mathbf{z}^{(k-1)}$
    $\mathbf{z}^{(k)} = g(\mathbf{a}^{(k)})$
  end for
  return $\mathbf{z}^{(l)}$

L'algoritmo di training forward propagation, riportato nell'Algoritmo 2, è una variante dell'algoritmo di forward propagation presentato in precedenza. Rispetto all'algoritmo di forward propagation standard, il training forward propagation estende le operazioni includendo il calcolo delle derivate delle funzioni di attivazione per ciascun layer. Questi valori sono essenziali per la fase successiva di back-propagation. L'algoritmo di training forward propagation restituisce sia gli output che le derivate delle funzioni di attivazione di ciascun layer.

Algoritmo 2 Algoritmo di Training Forward Propagation

Input: Profondità rete l
Input: $\mathbf{W}^{(i)}, i \in \{1, \ldots, l\}$, la matrice dei pesi del modello
Input: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$, i bias del modello
Input: $\mathbf{x}$, l'input da processare
  $\mathbf{z}^{(0)} = \mathbf{x}$
  for $k = 1$ to l **do**
    $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \times \mathbf{z}^{(k-1)}$
    $\mathbf{g}'^{(k)} = g'(\mathbf{a}^{(k)})$
    $\mathbf{z}^{(k)} = g(\mathbf{a}^{(k)})$
  end for
  return $(\mathbf{g}', \mathbf{z})$

1.1.1 Back-propagation

L'algoritmo di back-propagation, riportato nell'Algoritmo 3, è un algoritmo chiave nell'addestramento delle reti neurali. Esso calcola i gradienti della funzione di errore rispetto ai parametri della rete (cioè i pesi e i bias)

utilizzando il metodo di retropropagazione dell'errore. Questi gradienti vengono successivamente utilizzati per aggiornare i pesi della rete attraverso un algoritmo di ottimizzazione durante la fase di apprendimento.

Algoritmo 3 Algoritmo di back-propagation

Input: Profondità rete l
Input: $E(\theta)$ error function in funzione dei parametri della rete θ
Input: $\mathbf{W}^{(i)}, i \in \{1, \ldots, l\}$, la matrice dei pesi del modello
Input: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$, i bias del modello
Input: $\mathbf{x}$, l'input da processare
Input: $\mathbf{t}$, corrispondente valore atteso
  $(\mathbf{g}', \mathbf{z})$ = chiama *Training Forward Propagation* {Algoritmo 2}
  $\boldsymbol{\delta}^{(l)} = \frac{dE}{d\mathbf{z}^{(l)}} \odot \mathbf{g}'^{(l)}$
  for $k = l$ **downto** 1 **by** -1 **do**
    for $\delta_i^{(k)}$ **in** $\boldsymbol{\delta}^{(k)}$ **do**
      $\nabla_{\mathbf{W}_i^{(k)}} E = \mathbf{z}^{(k-1)} \cdot \delta_i^{(k)}$ **unless** $k = 1$ **then** $\mathbf{x} \cdot \delta_i^{(k)}$
    end for
    $\nabla_{\mathbf{b}^{(k)}} E = \boldsymbol{\delta}^{(k)}$
    $\boldsymbol{\delta}^{(k-1)} = \mathbf{g}'^{(k-1)} \odot \left[\left(\mathbf{W}^{(k)} \right)^T \times \boldsymbol{\delta}^{(k)} \right]$ **unless** $k = 1$ **then pass**
  end for
  return $(\nabla_{\mathbf{W}} E, \nabla_{\mathbf{b}} E)$

Di seguito viene fornita una spiegazione dettagliata del funzionamento dell'algoritmo di back-propagation:

1. L'algoritmo richiede come input la profondità della rete l, la funzione di errore $E(\theta)$ rispetto ai parametri della rete θ, le matrici dei pesi $\mathbf{W}^{(i)}$ e i bias $\mathbf{b}^{(i)}$ per ogni layer i, l'input $\mathbf{x}$ da processare e il corrispondente valore atteso $\mathbf{t}$;

2. Prima di tutto, viene eseguita la forward propagation chiamando l'algoritmo *Training Forward Propagation* (Algoritmo 2), che calcola gli output $\mathbf{z}$ e le derivate delle funzioni di attivazione $\mathbf{g}'$ di ogni layer;

3. Vengono calcolati i delta dei neuroni del layer finale l utilizzando la regola della catena. Il vettore $\boldsymbol{\delta}^{(l)}$ è ottenuto moltiplicando la derivata della funzione di errore rispetto all'output $\frac{dE}{d\mathbf{z}^{(l)}}$ dell'ultimo layer l e successivamente moltiplicando il vettore delle derivate delle funzioni di attivazione $\mathbf{g}'^{(l)}$ del medesimo layer l. Questo prodotto avviene elemento per elemento;

4. Viene eseguito un ciclo **for** all'indietro partendo dal layer finale l e risalendo a 1. All'interno del ciclo, vengono calcolati i gradienti dei pesi $\nabla_{\mathbf{W}^{(k)}} E$ e dei bias $\nabla_{\mathbf{b}^{(k)}} E$ del layer k utilizzando $\boldsymbol{\delta}^{(k)}$ calcolato in precedenza. La formula per calcolare $\nabla_{\mathbf{W}_i^{(k)}} E$ (le derivate parziali della funzione di errore rispetto ai pesi associati agli archi che partono dal layer precedente $k-1$ ed entrano nel neurone i nel layer k) è basata sulla moltiplicazione tra $\delta_i^{(k)}$ (il delta associato al neurone i nel layer k) e l'output del layer precedente $\mathbf{z}^{(k-1)}$ ($\mathbf{x}$ nel caso dell'input layer);

5. Ad ogni iterazione è possibile calcolare il vettore dei delta del layer precedente $\boldsymbol{\delta}^{(k-1)}$ (eccetto per l'input layer) moltiplicando la derivata della funzione di attivazione $\mathbf{g}'^{(k-1)}$ del layer precedente con il prodotto tra la matrice dei pesi trasposta $\left(\mathbf{W}^{(k)} \right)^T$ e il vettore $\boldsymbol{\delta}^{(k)}$ del layer corrente. Il calcolo dei delta del layer precedente permette alla prossima iterazione di usare questi delta per calcolare le prossime derivate parziali;

6. L'algoritmo restituisce i gradienti $\nabla_{\mathbf{W}} E$ e $\nabla_{\mathbf{b}} E$. Questi gradienti verranno poi utilizzati per aggiornare i pesi durante la fase di apprendimento tramite un algoritmo di ottimizzazione (ad esempio, discesa del gradiente).

2 Progettazione rete neurale

La rete neurale è stata implementata utilizzando il paradigma ad oggetti con il linguaggio di programmazione Python. Le diverse componenti sono organizzate all'interno di moduli basati sul dominio di utilizzo e insieme costituiscono il package `nnkit`.

2.1 Panoramica OOP

Il diagramma in Figura 1 offre una panoramica visiva delle classi e delle loro interazioni, fornendo una comprensione chiara dell'architettura complessiva del sistema.

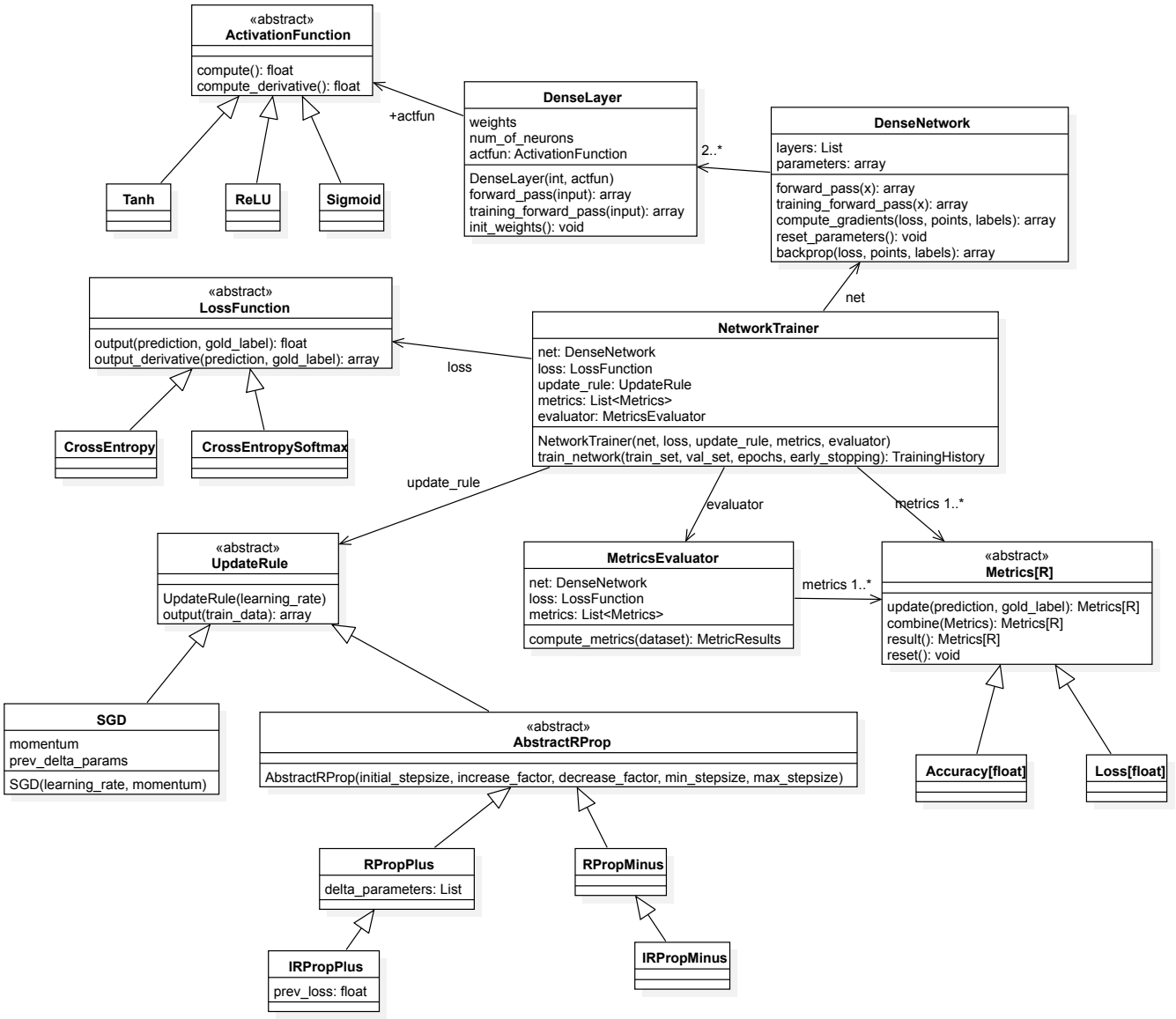


Figura 1: Diagramma delle classi UML non dettagliato della libreria `nnkit`

2.1.1 Moduli

Come accennato precedentemente, le componenti del sistema sono organizzate all'interno dei moduli della libreria `nnkit`, i quali saranno presentati di seguito:

```
nnkit
├── core
│   ├── neuronet.py
│   ├── losses.py
│   └── activations.py
└── training
    ├── metrics.py
    ├── model_selection.py
    ├── neurotrain.py
    ├── stopping.py
    └── update_rules.py
```

3 Implementazione rete neurale

Il package `nnkit.core` contiene le implementazioni delle classi che costituiscono una rete neurale.

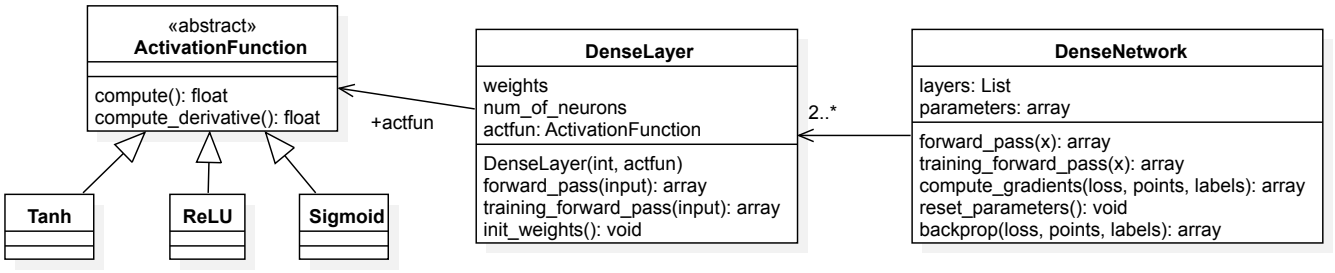


Figura 2: Diagramma delle classi UML più dettagliato del modulo `neuronet.py`

3.1 Dense Neural Network

La classe `DenseNetwork` (modulo `core.neuronet`) rappresenta un'implementazione di una rete neurale fully-connected. All'interno di questa classe, sono inclusi i metodi di forward propagation e back-propagation.

3.1.1 Implementazione algoritmo di forward propagation

La funzionalità di forward propagation è stata implementata tramite due metodi distinti, a seconda che la rete si trovi in fase di training o meno. Rifacendosi allo pseudocodice già precedentemente definito, di seguito è mostrata l'implementazione della propagazione in avanti senza training.

```
class DenseNetwork:
...
    #  Forward-pass di default (metodo di DenseNetwork)
    def __output(self, x: np.ndarray) -> np.ndarray:
        output = x

        for layer in self.__layers:
            output = layer(output)

        return output
```

La funzione `__output(self, x: np.ndarray)` itera sui layer propagando l'output aggiornato all'interno della rete. Inizialmente, l'input `x` viene assegnato alla variabile `output`, poiché rappresenta il punto di partenza del processo di propagazione. Successivamente, questo `output` viene passato come input al primo hidden layer. Il primo hidden layer elabora i dati e restituisce un nuovo `output` aggiornato. Questo nuovo `output` viene quindi utilizzato come input per il prossimo layer nel ciclo successivo dell'iterazione. Il processo continua iterando

attraverso tutti i layer nascosti fino a raggiungere il layer di output della rete.

D'altra parte, la funzione di forward propagation in fase di addestramento calcola e memorizza gli output e le derivate della funzione di attivazione di ogni layer. Ogni layer (di tipo `DenseLayer`), a sua volta, espone un metodo chiamato `training_forward_pass`, il quale, dato l'output del precedente layer (o l'input), restituisce un dizionario contenente due elementi indicizzati rispettivamente da `'z'` e `'d'`. Il primo elemento del dizionario contiene l'output del layer, mentre il secondo elemento contiene le derivate della funzione di attivazione. Ogni dizionario restituito da ogni layer viene memorizzato in una lista e restituito al chiamante sottoforma di tupla.

```
class DenseNetwork:
...
    # Forward-pass in fase di training (metodo di DenseNetwork)
    def training_forward_pass(self, x: np.ndarray) -> tuple[dict[str, np.ndarray]]:
        training_output = []

        prev_layer_output = x
        for layer in self.__layers:
            layer_training_output = layer.training_forward_pass(prev_layer_output)
            training_output.append(layer_training_output)
            prev_layer_output = layer_training_output['z']

        return tuple(training_output)
```

Da come si evince dal codice di sopra, il metodo `training_forward_pass` di `DenseNetwork` chiama a sua volta (nel ciclo for) il metodo `training_forward_pass` di `DenseLayer`. Di seguito vengono riportati i metodi di forward propagation della classe `DenseLayer`.

```
class DenseLayer:
...
    # Forward-pass di default (metodo di DenseLayer)
    def __forward_pass(self, inputs: np.ndarray) -> np.ndarray:
        inputs = np.concatenate(([1], inputs)) # Aggiungi output bias neuron

        a = np.matmul(self.weights, inputs)
        z = self.activation_function(a)

        return self.__post_processing(z)

    # Forward-pass in fase di training (metodo di DenseLayer)
    def training_forward_pass(self, inputs: np.ndarray) -> dict[str, np.ndarray]:
        inputs = np.concatenate(([1], inputs)) # Aggiungi output bias neuron

        a = np.matmul(self.weights, inputs)
        z = self.activation_function(a)
        d = self.activation_function.derivative(a)

        return {'z': self.__post_processing(z), 'd': d}
```

Entrambi i metodi di forward propagation di `DenseLayer` prima di processare l'input, concatenano all'input stesso un vettore contenente un uno, il quale rappresenta l'output del bias neuron. Questo è necessario perché il bias è integrato nella matrice dei pesi del corrispondente layer e viene posizionato sempre in posizione 0 in ogni riga della matrice. I due metodi sfruttano `numpy.matmul` per effettuare il prodotto tra matrici per ottenere `a`. Inoltre, durante il forward pass per l'addestramento, vengono mantenute anche le derivate della funzione di attivazione `d`. Si noti che in entrambi i metodi viene applicata una operazione di `post_processing` sull'output `z` prima di restituirlo. Al momento, le uniche operazioni di post-processing utilizzate nel progetto sono l'applicazione della funzione softmax all'output del layer di output (se è specificata `Softmax()` come funzione di attivazione) o la funzione d'identità che lascia invariato l'output (che è l'operazione di default).

3.1.2 Implementazione algoritmo di back-propagation

L'Algoritmo 3 è invece implementato nel metodo `backprop` della classe `DenseNetwork`, come mostrato di seguito:

```
# Back-propagation (metodo di DenseNetwork)
def backprop(self, loss_function: LossFunction, point: np.ndarray, label: np.ndarray)
    -> np.ndarray:

    # Training Forward Pass
    net_output = self.training_forward_pass(point)

    # Calcolo Delta Output layer
    net_output_last_layer = net_output[-1]
    output_last_layer = net_output_last_layer['z']
    der_actfun_last_layer = net_output_last_layer['d']
    der_lossfun = loss_function.output_derivative(output_last_layer, label)
    delta_last_layer = der_lossfun * der_actfun_last_layer

    # Calcolo Delta Hidden Layers
    delta_layers = np.zeros(self.depth, dtype=object)
    delta_layers[self.depth - 1] = delta_last_layer
    for index_layer in reversed(range(0, self.depth - 1)):
        parameters_next_layer = self.parameters[index_layer + 1]


        # remove bias
        weights_next_layer = np.array([
            parameters_next_layer[i][1:]
            for i in range(0, len(parameters_next_layer))
        ])

        net_output_curr_layer = net_output[index_layer]
        der_actfun_curr_layer = net_output_curr_layer['d']
        delta_next_layer = delta_layers[index_layer + 1]
        delta_curr_layer = der_actfun_curr_layer
            * np.matmul(weights_next_layer.transpose(), delta_next_layer)
        delta_layers[index_layer] = delta_curr_layer

    # Calcolo gradienti
    gradients = np.zeros(self.parameters.shape, dtype=object)
    for index_layer in reversed(range(0, self.depth)):
        delta_curr_layer = delta_layers[index_layer]
        output_prev_layer = net_output[index_layer - 1]['z'] if index_layer != 0 else point
        gradients_curr_layer = np.array([
            np.concatenate(([delta], delta * output_prev_layer))
            for delta in delta_curr_layer
        ])
        gradients[index_layer] = gradients_curr_layer

    return gradients
```

I gradienti calcolati vengono successivamente utilizzati per aggiornare i parametri della rete, seguendo una regola di aggiornamento specifica. Si noti comunque che mentre l'Algoritmo 3 presentato in Sezione 1 mostra un unico ciclo per il calcolo dei gradienti e dei delta degli hidden layers, la nostra implementazione di `backprop` suddivide il calcolo dei delta e dei gradienti in due cicli distinti. Per migliorare l'efficienza (a discapito della leggibilità del codice) abbiamo creato una versione alternativa di back-propagation chiamata `backprop_onecycle` (anch'essa presente nella classe `DenseNetwork`).

3.2 Funzioni di attivazione

Il modulo `nnkit.core.activations` contiene le implementazioni delle funzioni di attivazione. Tutte le funzioni di attivazione estendono la classe astratta `ActivationFunction`:

```
class ActivationFunction(object, metaclass=ABCMeta):

    @abstractmethod
```

```
    def __call__(self, a: np.ndarray) -> np.ndarray:
        pass

    @abstractmethod
    def derivative(self, a: np.ndarray) -> np.ndarray:
        pass
```

Il servizio che i due metodi astratti dovrebbero offrire sono abbastanza intuitivi. Prendiamo come esempio la funzione di attivazione della tangente iperbolica:

```
class Tanh(ActivationFunction):

    def __call__(self, a: float) -> float:
        return np.tanh(a)

    def derivative(self, a: float) -> float:
        tanh_a = self(a)
        return 1 - tanh_a ** 2
```

Il metodo `__call__` calcola e restituisce la tangente iperbolica dell'input `a`. Questo metodo viene utilizzato durante la fase di forward propagation.

Il metodo `derivative`, invece, calcola la derivata della funzione di attivazione rispetto ad `a`. Questa derivata è utilizzata nella fase di addestramento durante la back-propagation, poiché consente di calcolare i delta dei neuroni.

Altre funzioni di attivazione implementate sono: `ReLU`, `Identity`, `Sigmoid` e `Softmax` (considerata come un'operazione di post-processing).

3.3 Addestramento

Approfondiamo adesso le componenti che rendono possibile l'addestramento di una rete neurale, distribuite nel package `nnkit.training`.

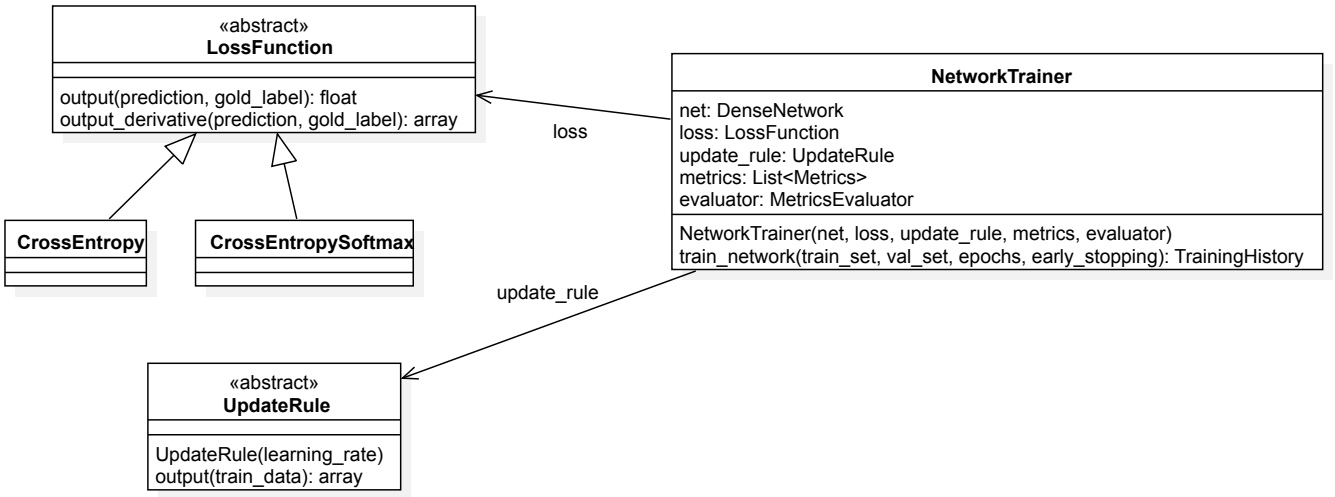


Figura 3: Diagramma delle classi UML del modulo `training`

3.3.1 Data Label Set

La classe `DataLabelSet`, definita nel modulo `nnkit.datasets.utils`, è stata omessa nel diagramma per evitare di renderlo troppo ingombrante e di difficile lettura. Tuttavia, è un componente importante del progetto, progettato per rappresentare un dataset generico composto da punti e rispettive etichette. Essa offre un'interfaccia comoda per iterare sulle coppie di punti ed etichette, consentendo di specificare la modalità di iterazione desiderata (online, mini-batch o batch) specificando un intero `batch_size` quando si istanzia l'oggetto. Oltre alla funzionalità di iterazione, la classe `DataLabelSet` fornisce altre utili operazioni, come la suddivisione del dataset in più parti, che risulta particolarmente utile per ottenere un validation set da utilizzare durante il processo di addestramento e validazione del modello. Nonostante la sua esclusione dal diagramma, la classe

`DataLabelSet` svolge un ruolo essenziale nel gestire i dati di input e le etichette associandole tra loro, fornendo un'astrazione utile e riutilizzabile per il trattamento di diversi tipi di dataset nel contesto del progetto. Di seguito riportiamo un esempio di utilizzo della classe, in cui il training set viene diviso per ottenere il validation set:

```
# Training data / Validation data
training_set = DataLabelSet(train_images, train_labels, batch_size=1, name='training')
training_set, validation_set = training_set.split(
    split_factor=0.2,
    split_set_batch_size=len(train_images),
    split_set_name='validation'
)
```

3.3.2 Update Rule

L'update rule determina come i parametri della rete vengono aggiornati alla fine di ogni epoca. Ogni regola di aggiornamento estende la classe astratta `UpdateRule`, mostrata di seguito (presente nel modulo `nnkit.training.update_rules`):

```
class UpdateRule(object, metaclass=ABCMeta):

    def __init__(self, learning_rate: float = 0.1, name: str = ""):
        if not name:
            name = self.__class__.__name__.lower()

        self.name = name
        self._learning_rate = learning_rate

    @abstractmethod
    def __call__(self, train_data: TrainingData) -> np.ndarray:
        pass
```

Le sottoclassi ereditano la proprietà relativa al tasso di apprendimento (`learning_rate`) e implementano ulteriori proprietà e parametri specifici per quella regola di aggiornamento. Ad esempio, la sottoclasse `SGD` (*Stochastic Gradient Descent*) può richiedere, oltre al `learning_rate`, il `momentum`.

```
class SGD(UpdateRule):

    def __init__(self, learning_rate: float, momentum: float = 0.0, name: str = ""):
        super().__init__(learning_rate, name)
        self.__momentum = momentum
        self.__prev_delta_parameters = 0.0

    def __call__(self, train_data: TrainingData) -> np.ndarray:
        parameters = train_data.parameters
        gradients = train_data.gradients

        delta_parameters = -(self._learning_rate * gradients)
            - (self.__momentum * self.__prev_delta_parameters)
        if self.__momentum != 0:
            self.__prev_delta_parameters = delta_parameters

        return parameters + delta_parameters
```

Le regole di aggiornamento implementate sono: `SGD`, `RPropPlus`, `IRPropPlus`, `RPropMinus` e `IRPropMinus`.

3.3.3 Loss Function

La classe astratta `LossFunction` è un componente fondamentale per il calcolo dell'errore durante l'addestramento della rete. Essa fornisce un'interfaccia comune per diverse funzioni di perdita utilizzate per valutare la discrepanza tra le predizioni del modello e le etichette di riferimento. Tutte le funzioni di errore estendono la classe `LossFunction` che è definita nel seguente modo:

```
class LossFunction(object, metaclass=ABCMeta):

    def __init__(self, name: str = "loss"):
        self.name = name

    @abstractmethod
    def __call__(self, prediction: np.ndarray, gold_label: np.ndarray) -> np.ndarray:
        pass

    @abstractmethod
    def output_derivative(self, prediction: np.ndarray, gold_label: np.ndarray) -> np.ndarray:
        pass
```

Molto similmente alla classe `ActivationFunction`, la classe `LossFunction` fornisce un metodo `__call__` che è responsabile di calcolare l'errore tra le predizioni del modello e le etichette di riferimento utilizzando la specifica funzione di perdita, mentre la funzione `output_derivative` calcola la derivata della funzione di errore rispetto all'output della rete neurale (fondamentale per l'algoritmo di back-propagation). La classe `LossFunction` è definita come una classe astratta, il che significa che non viene istanziata direttamente, ma viene utilizzata come base per la creazione di sottoclassi concrete (esattamente come `ActivationFunction` e `UpdateRule`). Un esempio di classe concreta è `CrossEntropy`:

```
    class CrossEntropy(LossFunction):

    def __init__(self, name: str = "cross_entropy_loss"):
        super().__init__(name)

    def __call__(self, predictions: np.ndarray, gold_labels: np.ndarray) -> np.ndarray:
        axis = int((predictions.ndim == 2))
        return -np.sum(gold_labels * np.log(predictions), axis)

    def output_derivative(self, prediction: np.ndarray, gold_label: np.ndarray) -> np.ndarray:
        return (prediction - gold_label) / (prediction * (1 - prediction))
```

3.3.4 NetworkTrainer

La classe `NetworkTrainer` (modulo `nnkit.training.neurotrain`) è il componente responsabile dell'addestramento di una rete neurale. La classe viene inizializzata con una `DenseNetwork`, una `LossFunction`, una `UpdateRule`, una lista di `Metrics` e un `MetricsEvaluator`. L'addestramento comincia con la chiamata al metodo `train_network`, passando come parametri due `DataLabelSet` (training set e validation set), il numero di epoche e uno `StoppingCriterion` opzionale (ossia un criterio di *early-stopping*, vedere Sezione 3.6). Riportiamo di seguito l'implementazione del metodo `train_network`:

```
class NetworkTrainer:
...

    def train_network(
        self,
        training_set: DataLabelSet,
        validation_set: DataLabelSet,
        epochs: int,
        early_stopping: function[TrainingData, bool] | StoppingCriterion = lambda _: False
    ) -> TrainingHistory:

        # Inizializza TrainingHistory
        self.__train_history = TrainingHistory(
            epochs,
            steps=training_set.steps,
            update_rule=self.__update_rule.name
        )
```

```
        # Inizia addestramento
        for epoch in range(epochs):
            self.__current_epoch = epoch
            for points, labels in training_set:
                self.__compute_gradients(points, labels)
                self.__update_parameters()
                self.__reset_gradients()

            # Alla fine di ogni epoca valuta il modello sul validation set e sul training set
            val_metric_results, train_metric_results = self.__evaluate_net(
                validation_set,
                training_set
            )

            # Tieni traccia dei migliori parametri in base alle
            # metriche calcolate sul validation set (validation loss)
            self.__update_best_parameters(val_metric_results)
            self.__print_epoch_info()

            # Controllo early-stopping (break ciclo)
            train_data = TrainingData(
                val_loss=self.__last_val_loss,
                train_loss=self.__last_train_loss
            )
            if early_stopping(train_data):
                print("Stopped early!")
                break

        # Setta i parametri migliori alla rete
        self.__net.parameters = self.__best_parameters.parameters

        # Ritorna TrainingHistory al chiamante
        history = self.__train_history
        self.__reset_trainer()
        return history
...
```

Il ciclo `for` esterno del processo di addestramento itera sul numero di epoche specificato in input. Ad ogni epoca, il ciclo interno itera su tutte le coppie di punti-etichetta presenti nel training set. Per ciascuna coppia, vengono calcolati i gradienti utilizzando il metodo `__compute_gradients`, che rappresenta un passo fondamentale della fase di back-propagation. Successivamente, i gradienti calcolati vengono utilizzati per aggiornare i parametri della rete neurale tramite il metodo `__update_parameters`. Questo passo di aggiornamento dei parametri è essenziale per il processo di apprendimento, poiché permette alla rete di adattarsi ai dati di training e migliorare le sue prestazioni.

Dopo aver completato un'epoca di addestramento, vengono calcolate le metriche di valutazione sia sul training set che sul validation set. Questo è importante per monitorare le prestazioni del modello durante il processo di addestramento e per evitare problemi di overfitting. Inoltre, in questa fase, si verifica se i parametri ottenuti hanno una migliore capacità di generalizzazione rispetto a quelli ottenuti in precedenza. In altre parole, si controlla se l'errore sul validation set è diminuito, e in caso positivo, i parametri vengono salvati come i migliori parametri finora ottenuti.

La chiamata al metodo `early_stopping` è responsabile di valutare se è necessario interrompere l'apprendimento in anticipo. Essenzialmente è un predicato che restituisce `True` o `False` (se non specificato nessun criterio di early-stopping restituisce di default `False`).

```
class NetworkTrainer:
...
    def __compute_gradients(self, points: np.ndarray, labels: np.ndarray) -> None:
        if self.__is_multiprocessing_allowed(num_samples_dataset=len(points)):
            self.__compute_gradients_in_parallel(points, labels)
        else:
            self.__gradients += self.__net.compute_gradients(self.__loss_function,
```

```
                points, labels)
...
```

Il frammento di codice soprastante esplicita la possibilità di effettuare in parallelo il calcolo dei gradienti a seconda della disponibilità di processori nella macchina su cui si addestra. Nel caso `multiprocessing` i gradienti sono calcolati da ogni processore e successivamente sommati per ottenere il gradiente totale. Invece, l'esecuzione non parallela salva in maniera diretta i gradienti restituiti dalla rete.

Al termine dell'addestramento, il metodo `train_network` della classe `NetworkTrainer` restituisce un oggetto `TrainingHistory` come output, contenente la storia dell'addestramento, comprese le metriche calcolate ad ogni epoca e i migliori parametri con maggiore capacità di generalizzazione. Questo oggetto è prezioso per generare grafici, analizzare le prestazioni del modello e salvare i risultati ottenuti.

Di seguito riportiamo un esempio per addestrare una rete:

```python
# Build Network
net = DenseNetwork(
    DenseLayer(num_inputs=784, num_neurons=256, activation_function=Tanh()),
    DenseLayer(num_neurons=10, activation_function=Softmax())
)

# Load data / Data pre-processing
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = (train_images.astype('float32') / 255)
train_labels = one_hot(train_labels)
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
test_labels = one_hot(test_labels)

# Training data / Validation data
training_set = DataLabelSet(train_images, train_labels, batch_size=len(train_images), name='training')
training_set, validation_set = training_set.split(
    split_factor=0.2,
    split_set_batch_size=len(train_images),
    split_set_name='validation'
)

# Train the network
trainer = NetworkTrainer(
    net=net,
    update_rule=IRPropPlus(),
    loss_function=CrossEntropySoftmax(),
    metrics=[Accuracy()]
)
history = trainer.train_network(training_set, validation_set, epochs=30)

# Test the network
test_set = DataLabelSet(test_images, test_labels, batch_size=len(test_images), name='test')
evaluator = MetricsEvaluator(net, metrics=[Accuracy()], loss_function=CrossEntropySoftmax())
metrics = evaluator.compute_metrics(test_set)
print(metrics)
```

3.4 Valutazione del modello

Il modulo `nnkit.training.metrics` include le classi e i metodi necessari per valutare le prestazioni dei modelli. La classe `NetworkTrainer`, a sua volta, utilizza di base il modulo `nnkit.training.metrics` per calcolare le prestazioni della rete neurale durante il processo di apprendimento. Questo permette di confrontare il valore di loss delle predizioni epoca dopo epoca e mantenere traccia del miglior risultato ottenuto.

3.4.1 MetricsEvaluator

La classe `MetricsEvaluator`, situata nel modulo `training.metrics`, ha il compito di calcolare le metriche di valutazione di un modello. Oltre a una lista opzionale di oggetti `Metrics`, richiede anche una funzione di

perdita (`loss_function`) che viene incapsulata all'interno di un oggetto di classe `Loss<float>`. Quest'ultimo rappresenta la metrica relativa alla funzione costo.

```
class MetricsEvaluator:
...
    def compute_metrics(self, dataset: DataLabelSet) -> MetricResults:
        if self.__is_multiprocessing_allowed(num_samples_dataset=len(dataset)):
            self.__compute_metrics_in_parallel(dataset)
        else:
            (points, labels) = dataset.get()
            predictions = self.__net(points)
            self.__metrics = self.__update_metrics(predictions, labels)

        metrics = copy.copy(self.__metrics)
        self.__reset_metrics()
        return MetricResults(metrics)
...
```

Anche il metodo `compute_metrics` può essere eventualmente parallelizzato. Al termine del metodo, tutte le metriche specificate all'inizializzazione sono calcolate sul dataset e incapsulate in un oggetto `MetricResults` successivamente restituito.

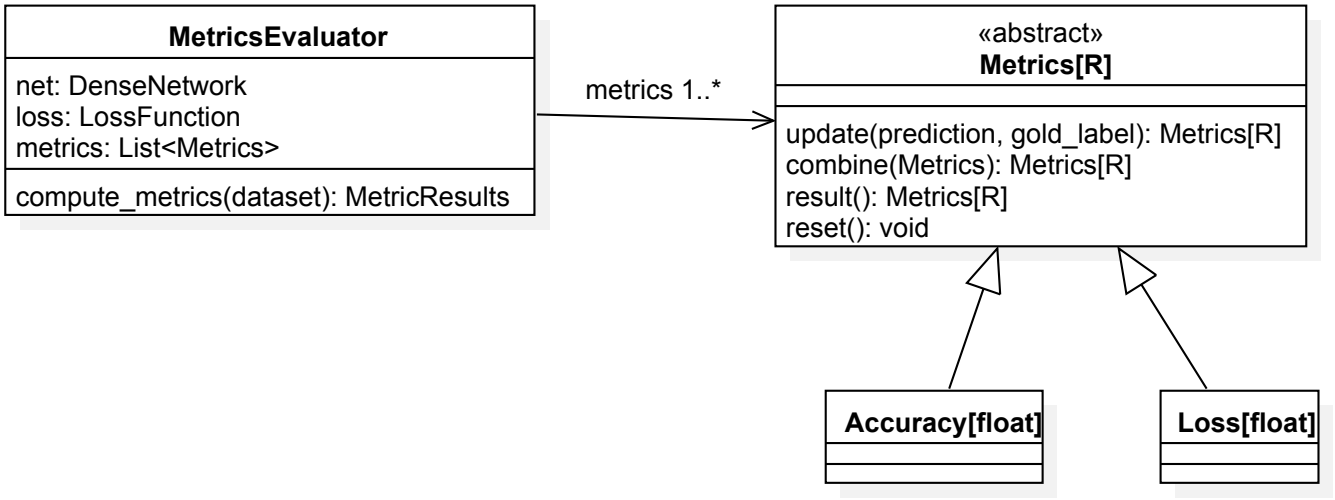


Figura 4: Diagramma delle classi UML non dettagliato del modulo `nnkit.training.metrics`

3.4.2 Metrics

La classe astratta `Metrics` definisce una tipologia di metrica utilizzata per la valutazione di modelli. Questa classe è dichiarata come generica (*Generics*), il cui tipo può variare a seconda della rappresentazione delle metriche specifiche. Ad esempio, per le metriche di accuratezza e costo (*Accuracy* e *Loss*), il tipo associato è `float`.

Inoltre, la classe `Metrics` definisce una serie di operazioni tra cui l'aggiornamento, il reset e la combinazione di metriche. Quest'ultima operazione risulta utile, in particolare, in scenari in cui il calcolo delle metriche viene parallelizzato, ad esempio con l'utilizzo di processori diversi, e i risultati parziali devono essere combinati per ottenere il valore totale della metrica.

3.5 Selezione del modello

In questa sezione, verrà presentata l'implementazione del modulo `nnkit.training.model_selection`. Questo modulo fornisce le funzionalità necessarie per selezionare il modello più adatto ai dati e agli obiettivi dell'applicazione.

3.5.1 KFold

Nella versione attuale della libreria, il modulo `nnkit.training.model_selection` include l'implementazione della classe `KFold`. Questa classe accetta un intero k come input e restituisce un generatore chiamato

`KFoldGenerator`. Il generatore permette di iterare su k coppie di set di addestramento e validazione. Le coppie vengono ottenute suddividendo il dataset di input in k parti, dove in ogni iterazione una parte diversa viene selezionata come set di validazione mentre le restanti vengono unite per formare il set di addestramento.

```
class KFold:

    def __init__(self, k: int = 5, shuffle: bool = False):
        self.k = k
        self.shuffle = shuffle

    class KFoldGenerator:

        def __init__(
            self, k: int,
            points_folds: list[np.ndarray],
            labels_folds: list[np.ndarray],
            train_batch_size: int
        ):
            self.points_folds = points_folds
            self.labels_folds = labels_folds
            self.k = k
            self.train_batch_size = train_batch_size
            self.__index = self.k-1

        def __iter__(self) -> self.KFoldGenerator:
            return self

        def __next__(self) -> tuple[DataLabelSet, DataLabelSet]:
            if self.__index < 0:
                raise StopIteration

            test_points = self.points_folds[self.__index]
            test_labels = self.labels_folds[self.__index]

            left_points_folds = self.points_folds[:self.__index]
            right_points_folds = self.points_folds[self.__index+1:]
            left_labels_folds = self.labels_folds[:self.__index]
            right_labels_folds = self.labels_folds[self.__index+1:]

            train_points = np.concatenate(left_points_folds + right_points_folds)
            train_labels = np.concatenate(left_labels_folds + right_labels_folds)

            self.__index -= 1
            train_set = DataLabelSet(train_points, train_labels, name='training',
                batch_size=self.train_batch_size)
            test_set = DataLabelSet(test_points, test_labels, name='test',
                batch_size=len(test_points))
            return train_set, test_set

    def __call__(self, dataset: DataLabelSet) -> KFoldGenerator:
        points_folds, labels_folds = dataset.fair_divide(self.k)
        generator = self.KFoldGenerator(self.k, points_folds, labels_folds,
         dataset.batch_size)
        return generator
```

3.6 Early Stopping

Nonostante non sia direttamente conforme alla traccia del progetto scelta, abbiamo comunque deciso di approfondire e implementare le tecniche di early stopping descritte nell'articolo "Early Stopping - but when?" di Lutz Prechelt.

L'early stopping rappresenta un criterio per interrompere l'addestramento in anticipo, mirando a ottenere un

equilibrio ottimale tra la capacità di generalizzare e il tempo di calcolo impiegato. Nell'articolo, questi criteri vengono definiti come *stopping criteria* e consistono fondamentalmente in predicati che indicano quando fermare l'addestramento. Tra tutti i criteri disponibili, ci focalizziamo su quelli che garantiscono il minor errore di generalizzazione e quelli che offrono il migliore "price-performance ratio". In altre parole, ci interessiamo a quelli che richiedono meno tempo di addestramento per ottenere un determinato livello di errore di generalizzazione o che, in media, portano al minimo errore di generalizzazione per un dato tempo di allenamento.

Abbiamo realizzato tre classi di stopping criteria, le stesse descritte nell'articolo, e le elenchiamo qui brevemente:

1. **Prima classe**: indichiamo con $E_{va}(t)$ l'errore sul validation set misurato dopo l'epoca t. Indichiamo con $E_{opt}(t)$ il più piccolo errore sul validation set ottenuto fino all'epoca t. Definiamo *generalization loss* ottenuto all'epoca t come l'aumento relativo del validation error rispetto al validation error visto fino ad adesso (in percentuale):
$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$
Un alto generalization loss è un motivo per fermare l'addestramento, perché indica overfitting. La prima classe di stopping criteria, indicata con GL_α, si ferma quando il generalization loss supera una specifica soglia α:
$$GL_\alpha : \ GL(t) > \alpha$$

2. **Seconda classe**: questa classe considera che gli errori di generalizzazione abbiano una probabilità elevata di essere "riparati" se il generalization loss è alto, ma l'errore sul training continua a decrescere rapidamente. L'articolo assume che l'overfitting inizi solo quando l'errore diminuisce lentamente. Per formalizzare questa nozione, definiamo un *training strip di lunghezza k* come una sequenza di k epoche numerate da $n+1$ a $n+k$, dove n è divisibile per k. Il *training progress* misurato dopo un tale training strip è definito come:
$$P_k(t) := 1000 \cdot \left(\frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \cdot \min_{t'=t-k+1}^{t} E_{tr}(t)} \right)$$
dove $E_{tr}(t)$ rappresenta l'errore sul training set misurato dopo l'epoca t. La quantità $P_k(t)$ si traduce in "di quanto l'errore medio sul training durante lo strip era maggiore dell'errore sul training durante lo strip?". La seconda classe di stopping criteria, indicata con PQ_α, si ferma quando il quoziente del generalization loss e del training progress supera un certa soglia α:
$$PQ_\alpha : \ \frac{GL(t)}{P_k(t)} > \alpha$$

3. **Terza classe**: l'ultima classe, indicata con UP_s, si basa esclusivamente sul segno delle variazioni dell'errore di validazione. La definizione è la seguente:
$$UP_s : \text{ fermati dopo l'epoca } t$$
$$\Longleftrightarrow$$
$$UP_{s-1} \text{ si ferma dopo l'epoca } t-k \text{ e } E_{va}(t) > E_{va}(t-k)$$
$$UP_1 : \text{ fermati dopo la prima "end-of-strip epoch } t\text{" con } E_{va}(t) > E_{va}(t-k)$$
L'idea fondamentale di questa definizione è che quando l'errore di validazione aumenta non solo una volta, ma per s training strip consecutivi, si presume che tali aumenti indichino l'inizio dell'overfitting finale, indipendentemente dall'entità di tali incrementi.

La figura 5 rappresenta il diagramma delle classi utilizzate per realizzare e implementare gli stopping criteria appena elencati.

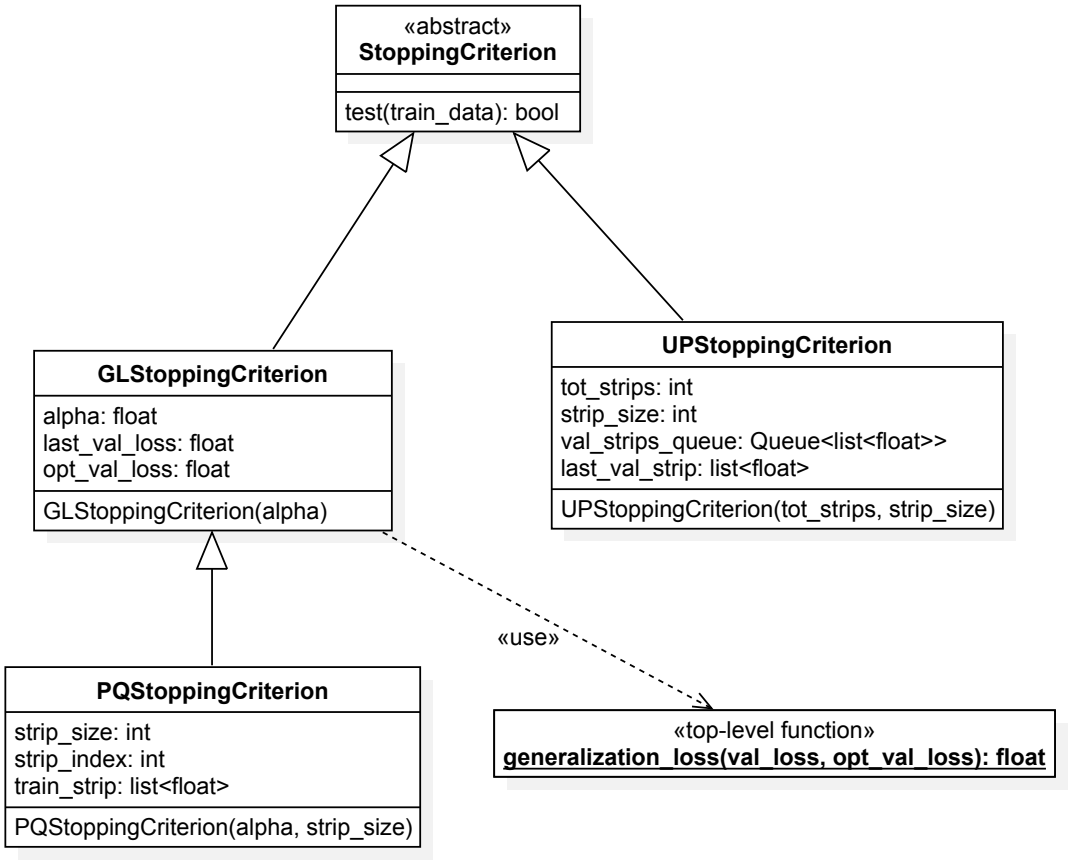


Figura 5: Diagramma delle classi UML stopping criteria

Di seguito riportiamo una breve spiegazione delle classi rappresentate in figura 5:

- `StoppingCriterion`: classe astratta che rappresenta un predicato. Essa espone un unico metodo astratto, `test(train_data: TrainingData): bool`, il quale accetta un oggetto `train_data` di tipo `TrainingData` contenente informazioni sintetiche riguardanti l'ultima epoca, come ad esempio l'errore sul training, l'errore sulla validazione, e così via. Il metodo restituisce un booleano che indica se l'addestramento deve essere interrotto o meno.
- `GLStoppingCriterion`: implementazione della prima classe di stopping criteria. È necessario memorizzare l'ultimo errore sul validation e il miglior errore sul validation finora osservato, aggiornandolo ad ogni nuova invocazione del predicato. Inoltre, fa uso di una funzione di livello superiore `generalization_loss(val_loss, opt_val_loss)` per calcolare il generalization loss.
- `PQStoppingCriterion`: implementazione della seconda classe di stopping criteria. Eredita dalla classe `GLStoppingCriterion`. È necessario specificare uno `strip_size` quando si istanzia l'oggetto che rappresenta la lunghezza k del training strip.
- `UPStoppingCriterion`: implementazione della terza classe di stopping criteria. È necessario specificare il numero totale di strips s e la dimensione di tali strips primaa di istanziare l'oggetto.

Resilient backpropagation

4 RProp (o RProp$^+$)

Una singola iterazione dell'algoritmo RProp può essere suddivisa in due fasi. La prima fase riguarda la regolazione dei valori degli step-size, che è essenzialmente la stessa per tutte le varianti considerate in questo progetto. Per ogni peso w_{ij} viene assegnato un valore individuale di step-size Δ_{ij} secondo la seguente regola:

$$\Delta_{ij}^{(t)} := \begin{cases} \min(\eta^+ \Delta_{ij}^{(t-1)}, \Delta_{\max}) & \textbf{se } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \max(\eta^- \Delta_{ij}^{(t-1)}, \Delta_{\min}) & \textbf{se } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & \text{altrimenti} \end{cases}$$

dove $0 < \eta^- < 1 < \eta^+$ e gli step-sizes sono confinati dai parametri $\Delta_{\min}$ e $\Delta_{\max}$.

La seconda parte riguarda l'aggiornamento dei pesi ed è differente per ogni variante di RProp.

L'algoritmo RProp originale, noto come RProp$^+$ nell'articolo di Christian Igel e Michael Hüsken, si basa su una regola generale denominata *weight-backtracking* per migliorare l'addestramento della rete. Questa regola prevede di tornare indietro all'aggiornamento precedente per alcuni o tutti i pesi e la decisione viene presa in base a delle euristiche. L'euristica adottata da RProp$^+$ è molto semplice. Se il segno della derivata parziale non è cambiato, allora viene eseguito un aggiornamento regolare dei pesi:

$$\textbf{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} \geq 0 \textbf{ then } \Delta w_{ij}^{(t)} := -\operatorname{sign}\left(\frac{\partial E}{\partial w_{ij}}\right) \Delta_{ij}^{(t)}$$

Nel caso in cui si è verificato un cambiamento del segno della derivata parziale, l'aggiornamento del peso diventa l'aggiornamento del peso precedente cambiato di segno:

$$\textbf{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \textbf{ then } \Delta w_{ij}^{(t)} := -\Delta w_{ij}^{(t-1)};\ \frac{\partial E}{\partial w_{ij}}^{(t)} := 0$$

Settare la derivata parziale uguale a zero permette di evitare un aggiornamento del learning rate alla prossima iterazione.

4.1 Varianti di RProp

4.1.1 RProp$^-$

RProp$^-$ è la prima variante di RProp. In questa variante non è più necessario memorizzare gli aggiornamenti dei pesi precedenti perché la nozione di weight-backtracking viene eliminata. L'aggiornamento dei pesi è in ogni caso il seguente:

$$\Delta w_{ij}^{(t)} := -\operatorname{sign}\left(\frac{\partial E}{\partial w_{ij}}\right) \Delta_{ij}^{(t)}$$

4.1.2 iRProp$^+$

La versione iRProp$^+$ (improved RProp con weight-backtracking) si basa sulla considerazione che un cambio di segno nella derivata parziale indica che l'algoritmo ha saltato un minimo locale, ma non fornisce informazioni sull'effetto dell'aggiornamento del peso sull'errore, se sia aumentato o diminuito. Per questo motivo, tale variante propone di considerare anche l'errore come criterio di valutazione (oltre al cambiamento di segno della derivata parziale) per decidere se fare backtracking o meno. I pesi vengono retrocessi per gli aggiornamenti che hanno determinato un cambiamento di segno nelle derivate parziali corrispondenti, ma solo se l'errore è aumentato come conseguenza di tali aggiornamenti. La regola di backtracking diventa dunque la seguente:

$$\begin{aligned} &\textbf{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \textbf{ then} \\ &\quad \Delta_{ij}^{(t)} = \min(\eta^+ \Delta_{ij}^{(t-1)}, \Delta_{\max}) \\ &\quad \Delta w_{ij}^{(t)} = \begin{cases} -\Delta w_{ij}^{(t-1)} & \textbf{se } E^{(t)} > E^{(t-1)} \\ 0 & \text{altrimenti} \end{cases} \\ &\quad \frac{\partial E}{\partial w_{ij}}^{(t)} = 0 \\ &\textbf{end if} \end{aligned}$$

Algoritmo 4 RProp$^+$

Input: $\mathbf{W}_{(t)}^{(i)}, i \in \{1, \ldots, l\}$, la matrice dei parametri del modello (incluso i biases) epoca t
Input: $\nabla_{\mathbf{W}}^{(t)} E$, il gradiente dei parametri epoca t
Input: η^+, increase factor
Input: η^-, decrease factor
Input: $\Delta_{\max}$, max step-size
Input: $\Delta_{\min}$, min step-size
Input: Δ_0, initial step-size di Δ_{ij} da usare quando $t = 0$
for $k = 1$ to l **do**
  for w_{ij} **in** $\mathbf{W}^{(k)}$ **do**
    if $\frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} > 0$ **then**
      $\Delta_{ij}^{(t)} = \min(\eta^+ \Delta_{ij}^{(t-1)}, \Delta_{\max})$
      $\Delta w_{ij}^{(t)} = -\operatorname{sign}\left(\frac{\partial E}{\partial w_{ij}}^{(t)}\right) \Delta_{ij}^{(t)}$
    else if $\frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0$ **then**
      $\Delta_{ij}^{(t)} = \max(\eta^- \Delta_{ij}^{(t-1)}, \Delta_{\min})$
      $\Delta w_{ij}^{(t)} = -\Delta w_{ij}^{(t-1)}$
      $\frac{\partial E}{\partial w_{ij}}^{(t)} = 0$
    else
      $\Delta_{ij}^{(t)} = \Delta_{ij}^{(t-1)}$
      $\Delta w_{ij}^{(t)} = -\operatorname{sign}\left(\frac{\partial E}{\partial w_{ij}}^{(t)}\right) \Delta_{ij}^{(t)}$
    end if
    $w_{i,j}^{(t+1)} = w_{i,j}^{(t)} + \Delta w_{i,j}^{(t)}$
  end for
end for

4.1.3 iRProp$^-$

La variante iRProp$^-$ (improved RProp senza weight-backtracking) differisce dalla variante RProp$^-$ solo in un aspetto. Quando il segno della derivata parziale cambia, iRProp$^-$ riduce lo step-size corrispondente senza modificare il peso corrispondente, impostando la derivata parziale stessa a zero. Questa modifica garantisce che nel prossimo step il peso venga aggiornato utilizzando lo step-size ridotto, poiché il prodotto tra la derivata parziale precedente e quella attuale sarà uguale a zero. La modifica è la seguente:

if $\frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0$ **then**
  $\Delta_{ij}^{(t)} = \max(\eta^- \Delta_{ij}^{(t-1)}, \Delta_{\min})$
  $\frac{\partial E}{\partial w_{ij}}^{(t)} = 0$
end if
$\Delta w_{ij}^{(t)} = -\operatorname{sign}\left(\frac{\partial E}{\partial w_{ij}}^{(t)}\right) \Delta_{ij}^{(t)}$ {In ogni caso}

4.2 Implementazione RProp e varianti

Le versioni con *weight-backtracking* richiedono la memorizzazione dei parametri, a differenza delle versioni senza, le quali hanno solo variazioni relative al comportamento. Di conseguenza, la classe astratta `AbstractRProp` implementa tutti i metodi necessari per la resilient propagation, le cui variazioni sono implementate tramite override dei metodi nelle sottoclassi.

```
class AbstractRProp(UpdateRule, metaclass=ABCMeta):

    def __init__(
        self,
        initial_step_size: float = 0.01,
        increase_factor: float = 1.2,
        decrease_factor: float = 0.5,
        min_step_size: float = 1e-6,
        max_step_size: float = 50,
```

```
    name: str = ""
):
    super().__init__(1.0, name)
    self._initial_stepsize = initial_step_size
    self._increase_factor = increase_factor
    self._decrease_factor = decrease_factor
    self._min_stepsize = min_step_size
    self._max_stepsize = max_step_size
    self._stepsizes = None
    self._prev_gradients = None
    self._gradients_change = None
    self._gradients_sign = None

def __call__(self, train_data: TrainingData) -> np.ndarray:
    parameters = train_data.parameters
    gradients = train_data.gradients
    self._gradients_sign = numpy_deep_apply_func(np.sign, gradients)

    if self._stepsizes is None:
        self._stepsizes = numpy_deep_full_like(parameters, self._initial_stepsize)

    if self._prev_gradients is not None:
        self._gradients_change = self._compute_gradients_change()
        self._stepsizes = self._compute_stepsizes()

    self._prev_gradients = gradients

    delta_parameters = self._compute_delta_parameters()
    return parameters + delta_parameters

def _compute_gradients_change(self) -> np.ndarray:
    prev_gradients_sign = numpy_deep_apply_func(np.sign, self._prev_gradients)
    return prev_gradients_sign * self._gradients_sign

def _compute_stepsizes(self) -> np.ndarray:
    stepsizes = numpy_deep_zeros_like(self._stepsizes)

    for i, layer_gradients_change in enumerate(self._gradients_change):
        for (j, k), gradient_change in np.ndenumerate(layer_gradients_change):
            if gradient_change == 1:
                stepsizes[i][j][k] = min(self._stepsizes[i][j][k]
                    * self._increase_factor, self._max_stepsize)
            elif gradient_change == -1:
                stepsizes[i][j][k] = max(self._stepsizes[i][j][k]
                    * self._decrease_factor, self._min_stepsize)

    return stepsizes

def _compute_delta_parameters(self) -> np.ndarray:
    delta_parameters = numpy_deep_zeros_like(self._stepsizes)

    for i, layer_delta_parameters in enumerate(delta_parameters):
        for (j, k), _ in np.ndenumerate(layer_delta_parameters):
            delta_parameters[i][j][k] = self._compute_delta_parameter((i, j, k))

    return delta_parameters

@abstractmethod
def _compute_delta_parameter(self, indexes: tuple[int, int, int]) -> float:
    pass
```

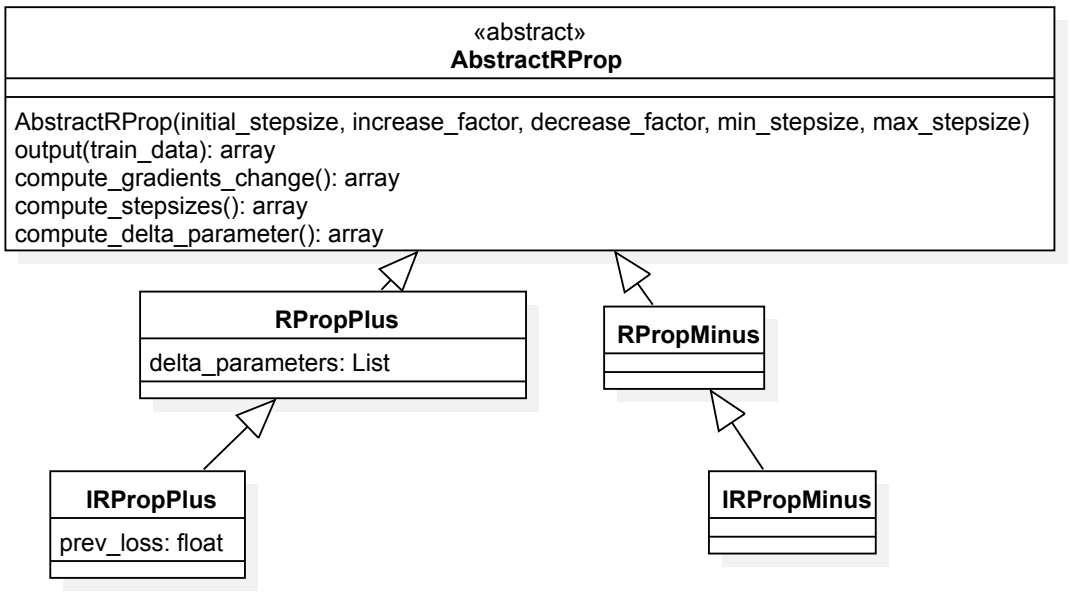


Figura 6: Diagramma delle classi UML non dettagliato per RProp e varianti

Il diagramma in Figura 6 rappresenta la gerarchia delle classi RProp, la quale è strutturata in questo modo perché:

- Le varianti con *weight-backtracking* (`IRPropPlus` e `RPropPlus`) salvano entrambe i `delta_parameters`. Inoltre `IRPropPlus` salva anche l'errore precedente nella proprietà `prev_loss`.

```python
class RPropPlus(AbstractRProp):

    def __init__(
        self,
        initial_step_size: float = 0.01,
        increase_factor: float = 1.2,
        decrease_factor: float = 0.5,
        min_step_size: float = 1e-6,
        max_step_size: float = 50,
        name: str = ""
    ):
        super().__init__(initial_step_size, increase_factor, decrease_factor,
    min_step_size, max_step_size, name)
        self._delta_parameters = None

    def _compute_delta_parameters(self) -> np.ndarray:
        self._delta_parameters = super()._compute_delta_parameters()
        return self._delta_parameters

    def _compute_delta_parameter(self, indexes: tuple[int, int, int]) -> float:
        (i, j, k) = indexes

        if self._gradients_change is None or self._gradients_change[i][j][k] >= 0:
            return -self._gradients_sign[i][j][k] * self._stepsizes[i][j][k]

        self._prev_gradients[i][j][k] = 0
        return -self._delta_parameters[i][j][k]


class IRPropPlus(RPropPlus):

    def __init__(
        self,
        initial_step_size: float = 0.01,
```

```
        increase_factor: float = 1.2,
        decrease_factor: float = 0.5,
        min_step_size: float = 1e-6,
        max_step_size: float = 50,
        name: str = ""
    ):
        super().__init__(initial_step_size, increase_factor, decrease_factor,
    min_step_size, max_step_size, name)
        self._loss = 0.0
        self._prev_loss = 0.0

    def __call__(self, train_data: TrainingData):
        self._loss = train_data.val_loss
        updated_parameters = super().__call__(train_data)
        self._prev_loss = self._loss
        return updated_parameters

    def _compute_delta_parameter(self, indexes: tuple[int, int, int]) -> float:
        (i, j, k) = indexes

        if self._gradients_change is not None and self._gradients_change[i][j][k] < 0:
            if self._loss > self._prev_loss:
                delta_parameter = -self._delta_parameters[i][j][k]
            else:
                delta_parameter = 0
            self._prev_gradients[i][j][k] = 0
        else:
            delta_parameter = -self._gradients_sign[i][j][k] * self._stepsizes[i][j][k]

        return delta_parameter
```

- Le varianti senza *weight-backtracking* (`RPropMinus` e `IRPropMinus`) non hanno requisiti di memorizzazione.

```
class RPropMinus(AbstractRProp):

    def _compute_delta_parameter(self, indexes: tuple[int, int, int]) -> float:
        (i, j, k) = indexes
        return -self._gradients_sign[i][j][k] * self._stepsizes[i][j][k]


class IRPropMinus(RPropMinus):

    def _compute_delta_parameter(self, indexes: tuple[int, int, int]) -> float:
        (i, j, k) = indexes

        if self._gradients_change is not None and self._gradients_change[i][j][k] < 0:
            self._prev_gradients[i][j][k] = 0

        return -self._gradients_sign[i][j][k] * self._stepsizes[i][j][k]
```

4.3 Confronto

In questa sezione, effettueremo un confronto tra l'algoritmo RProp (o RProp$^+$) e le sue varianti ovvero RProp$^-$, IRProp$^+$ e IRProp$^-$ utilizzando la libreria `nnkit` proposta in precedenza. Confronteremo anche la classica discesa del gradiente con momentum insieme a RProp e le sue varianti. Abbiamo deciso di farlo perché entrambe le tipologie di algoritmi (RProp e la discesa del gradiente) sostanzialmente si prefiggono lo stesso obiettivo, cioè quello di accelerare nelle zone in cui la funzione di errore risulta essere piatta. L'obiettivo principale sarà valutare le prestazioni di ciascuna regola di aggiornamento utilizzando il noto dataset MNIST come punto di riferimento.

4.3.1 Definizione dell'architettura

La rete neurale utilizzata per il confronto è una rete neurale fully-connected (`DenseNetwork` in `nnkit`) composta da un layer di input con 784 nodi, che rappresentano i pixel di immagini MNIST 28x28 appiattite, seguito da un hidden layer con 256 nodi interni e un output layer con 10 nodi. Nei nodi interni viene utilizzata la funzione di attivazione tangente iperbolica (`tanh`). Nei nodi di output, viene applicata la funzione di attivazione `softmax`, che converte i valori in una distribuzione di probabilità per l'assegnazione delle classi. Di seguito viene riportato il codice per costruire la rete appena citata con la libreria `nnkit`:

```
net = DenseNetwork(

    DenseLayer(num_inputs=784, num_neurons=256, activation_function=Tanh()),
    DenseLayer(num_neurons=10, activation_function=Softmax())

)
```

4.3.2 Definizione degli iperparametri

L'addestramento della rete neurale è stato eseguito utilizzando la modalità *batch* di processamento del dataset per quanto riguarda l'algoritmo RProp e le sue varianti, mentre per la discesa del gradiente con momentum la modalità *online*. Il numero di epoche è stato fissato a 40.

I parametri scelti per RProp (e varianti) sono i parametri di default, ad eccezione del parametro Δ_0 (*initial stepsize*) che è stato impostato a 0.001:

- $\Delta_0 = 0.001$;
- $\Delta_{min} = 1e-6$;
- $\Delta_{max} = 50.0$;
- $\eta^- = 0.5$;
- $\eta^+ = 1.2$.

Per quanto riguarda i parametri della discesa del gradiente, abbiamo scelto i più classici:

- $\eta = 0.1$;
- $\mu = 0.9$.

La funzione di errore utilizzata per l'addestramento è la *Cross Entropy Softmax*, che risulta essere particolarmente adatta per problemi di classificazione multiclasse. Inoltre, come metrica aggiuntiva per valutare le prestazioni della rete, è stata utilizzata l'accuratezza.

4.3.3 Definizione del dataset

Per l'addestramento e la valutazione della rete neurale, è stato utilizzato il dataset MNIST. Per preparare le feature delle immagini MNIST 28x28, è stata effettuato un pre-processing delle immagini portando i valori da un intervallo di 0-255 a un intervallo di 0-1. Tuttavia, per limitare la dimensione del training set e ridurre i tempi di addestramento, si è scelto di impiegare un sottoinsieme del dataset MNIST contenente 12.500 elementi. Il training set è ottenuto selezionando i primi 12.500 campioni dal dataset MNIST (essendo già ben bilanciato) e utilizzandoli come set di addestramento per la rete neurale. Il validation set è ottenuto suddividendo il training set originale utilizzando un fattore di divisione (split factor) del 0.2. Ciò ha portato ad un validation set composto da 2.500 campioni.

Il test set, utilizzato per valutare le prestazioni finali della rete, è costituito da 10.000 elementi del test set del dataset MNIST originale.

4.3.4 Inizializzazione dei parametri

I parametri sono stati inizializzati in ciascun layer utilizzando la seguente regola:

```
class DenseLayer:
...
    def initialize_weights(self, num_inputs: int) -> np.ndarray:
        self.__weights = np.random.normal(size=(self.num_neurons, num_inputs + 1)) * 0.001
        return self.__weights
```

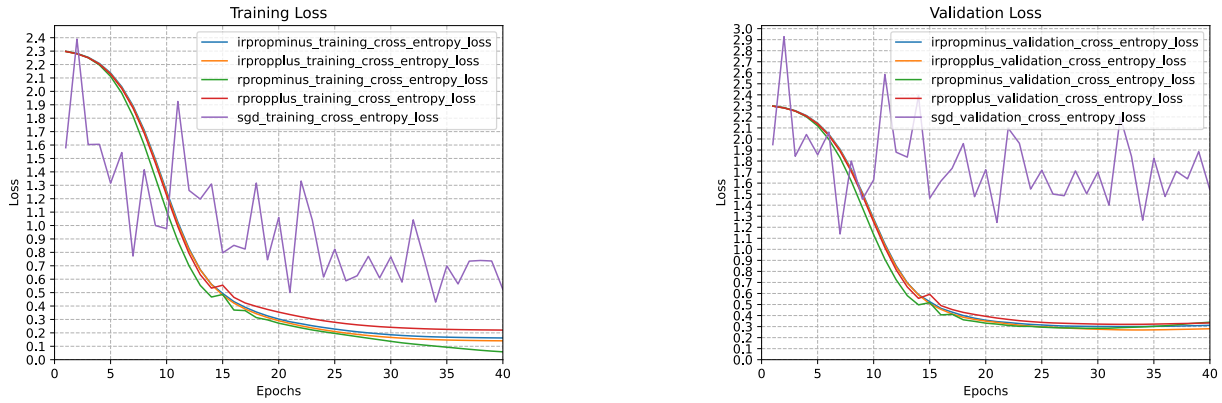
I pesi vengono inizializzati generandoli da una distribuzione Gaussiana normale. Successivamente vengono moltiplicati per 0.001 riducendo l'ampiezza dei pesi iniziali per evitare che diventino troppo grandi (per evitare situazioni di overflow).

4.3.5 Addestramento e risultati

Dopo aver definito tutti i parametri e le suddivisioni del dataset, è stato implementato un ciclo per ottenere i dati di addestramento per ciascuna variante dell'algoritmo RProp.

```
for update_rule in [IRPropPlus(initial_step_size=0.001), IRPropMinus(initial_step_size=0.001),
    RPropPlus(initial_step_size=0.001), RPropMinus(initial_step_size=0.001)]:
        trainer = NetworkTrainer(
            net=net,
            update_rule=update_rule,
            loss_function=CrossEntropySoftmax(),
            metrics=[Accuracy()]
        )
        history = trainer.train_network(training_set, validation_set, epochs=epochs)
        histories.append(history)
        trainer.net.reset_parameters()
```

Nelle Figure 7(a) e 7(b) è possibile notare come, per tutte le varianti di RProp, i valori di perdita per il training e la validazione rimangono pressoché simili tra di loro durante la discesa. In genere, questo è segno di un buon addestramento, poiché il modello generalizza bene su dati non visti (ossia il validation set).



(a) Costo sul training set.

(b) Costo sul validation set.

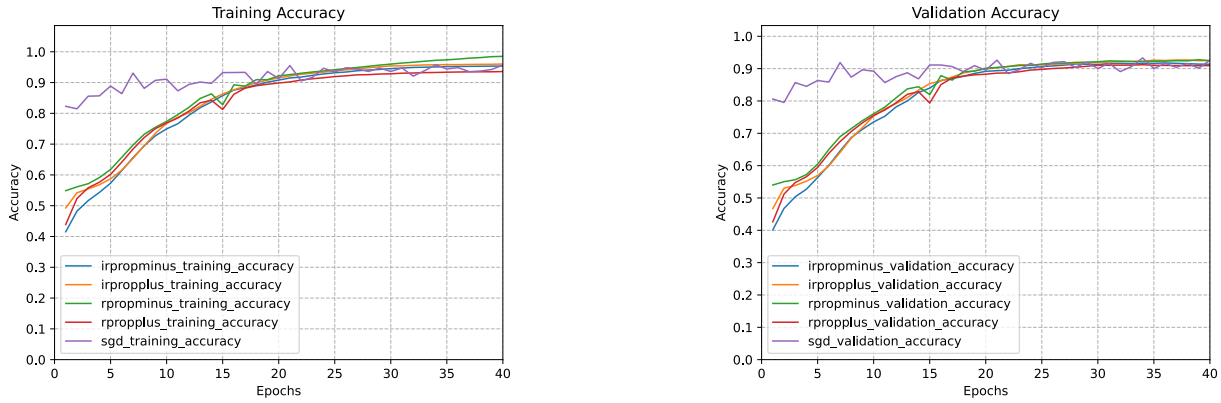
Figura 7: Costo durante l'addestramento.

Tutte le varianti di RProp hanno approssimativamente lo stesso andamento in entrambi i grafici. Nella fase iniziale dell'addestramento, tra l'epoca 1 e l'epoca 15, RProp$^-$ ottiene il minor errore sia sul training che sul validation. Tuttavia, nel caso di RProp$^-$, si osserva una tendenza alla convergenza verso lo zero del training loss intorno all'epoca 40, come mostrato nel grafico 7(a). Allo stesso tempo, l'errore sul validation di RProp$^-$ sembra invertire la sua traiettoria e cominciare ad aumentare, come evidenziato nel grafico 7(b). Questo tipo di comportamento suggerisce la possibilità di un *overfitting*, cioè una situazione in cui il modello si adatta eccessivamente ai dati di training, ma perde la capacità di generalizzare su dati nuovi. A partire dall'epoca 15 in poi, RProp$^+$ dimostra di avere il più alto training loss e il più alto validation loss, mentre IRProp$^-$ e IRProp$^+$ sono pressoché identici. Tuttavia, un'attenta analisi rivela che IRProp$^+$ è in realtà il migliore, sia in termini di stabilità sia in termini di minimo errore sul training e sul validation.

Mentre RProp e le sue varianti mostrano una convergenza stabile verso un minimo della funzione di errore, la discesa del gradiente con momentum presenta un comportamento opposto, come evidenziato dai grafici. Nelle iterazioni, l'inerzia del momento può portare il modello a oscillare lungo la superficie della funzione di errore, rendendo il processo di addestramento meno stabile e meno predittivo rispetto a RProp. Questo comportamento oscillante può essere problematico, specialmente quando il modello si avvicina a un minimo locale o quando la superficie della funzione di errore è caratterizzata da zone piatte o picchi pronunciati. In confronto, RProp e le sue varianti si dimostrano più adattabili e performanti nell'effettuare aggiornamenti dei pesi in modo consistente verso la direzione di miglioramento dell'errore, consentendo una convergenza più efficace. C'è anche da dire che, mentre l'agoritmo RProp e varianti hanno tutti uno step-size iniziale di $\Delta_0 = 0.001$, l'algoritmo stocastich

gradient descent con momentum ha un learning rate fisso per tutti i parametri e per tutte le epoche uguale a $\eta = 0.1$ e un momentum di $\mu = 0.9$. Tuttavia, da come si evince dai grafici, le ampiezze di queste oscillazioni sembrano diminuire man mano che l'addestramento va avanti. Inoltre, se si osserva bene il grafico 7(b), la discesa del gradiente con momentum ottiene il suo minimo validation loss tra l'epoca 5 e l'epoca 10 (precisamente epoca 6). Alla fine dell'addestramento verranno tenuti i parametri che hanno ottenuto un validation loss più piccolo, questo vuol dire che tutti gli sforzi compiuti dopo l'epoca 6 sono stati invani perché non hanno prodotto nessun miglioramento. Un opportuno criterio di *early stopping*, come quelli presentati nella Sezione 3.6, in questo caso avrebbe potuto rilevare questa situazione e fermare prima l'addestramento.

Come metrica di valutazione aggiuntiva è stata utilizzata l'accuratezza. Le Figure 8(a) e 8(b) di seguito mostrano il comportamento dell'accuratezza durante la fase di addestramento.



(a) Accuracy sul training set.

(b) Accuracy sul validation set.

Figura 8: Accuracy durante l'addestramento.

Per quanto riguarda RProp e le sue varianti, possiamo osservare che, nei grafici 8(a) e 8(b), nella fase iniziale dell'addestramento RProp$^-$ ottiene subito un'accuratezza più alta rispetto alle altre varianti. Tuttavia, verso l'epoca 40, RProp$^-$ mostra un'accuratezza sul training pari a uno e, come già rilevato nei grafici precedenti sulla funzione di errore, questo indica molto probabilmente overfitting. RProp$^+$ sembra essere la variante che ha un'accuratezza più bassa rispetto alle altre. IRProp$^-$ e IRProp$^+$ mostrano approssimativamente una accuratezza uguale sul training, mentre sul validation l'accuratezza sembra essere più alta per IRProp$^+$.

L'accuratezza ottenuta con l'algoritmo della discesa del gradiente con momentum risulta essere notevolmente elevata fin dalle prime epoche, dimostrando un rapido apprendimento iniziale. Successivamente, l'accuratezza si stabilizza e converge verso un valore stabile.

Alla fine dell'addestramento abbiamo calcolato l'accuratezza su un insieme di test fornito dal dataset MNIST per valutare le performance finali di ogni modello.

Tabella 1: Prestazioni di SGD con momentum e RProp (e varianti)

Variante RProp	Numero campioni di test	Accuracy sul validation	Accuracy sul test
SGD con momentum	10,000	0.9332	0.9126
RProp$^+$	10,000	0.9120	0.9071
RProp$^-$	10,000	0.9276	0.9198
IRProp$^+$	10,000	0.9268	0.9229
IRProp$^-$	10,000	0.9176	0.9192

La Tabella 1 riporta le accuratezze ottenute sul test e sul validation per ciascuna regola di aggiornamento. Notiamo che, sebbene la discesa del gradiente con momentum abbia ottenuto un'accuratezza sul validation leggermente migliore rispetto a RProp e le sue varianti, i parametri associati non generalizzano altrettanto bene quanto quelli ottenuti dalle varianti di RProp. Infatti, l'accuratezza sul test per SGD con momentum risulta inferiore rispetto a tutte le altre varianti di RProp. Un dato particolarmente interessante è che la rete addestrata con IRProp$^+$ ha ottenuto l'accuratezza sul test più elevata rispetto a tutte le altre reti addestrate con gli altri algoritmi, confermando l'efficacia di IRProp$^+$ nella capacità di generalizzazione.

5 Conclusioni

L'obiettivo principale di questo progetto era fornire un ambiente di apprendimento e sperimentazione mirato a una migliore comprensione dei dettagli e dei concetti fondamentali delle reti neurali. Per raggiungere questo scopo, abbiamo creato reti neurali completamente connesse utilizzando il linguaggio di programmazione Python. Il focus principale è stato rivolto allo studio e all'implementazione di tecniche cruciali come l'algoritmo di back-propagation e l'algoritmo RProp, oltre alle sue varianti. Questa esperienza ci ha consentito di acquisire familiarità con l'architettura e il funzionamento delle reti neurali, migliorando la nostra capacità di affrontare problemi di apprendimento automatico in modo più efficace e competente.

Il progetto è ospitato su una repository pubblica su GitHub, accessibile tramite il seguente link: `https://github.com/vtramo/neural-networks-experiments-from-scratch`