# UNIVERSITÀ DEGLI STUDI DI MESSINA

## DIPARTIMENTO MIFT


**CORSO DI LAUREA TRIENNALE IN INFORMATICA**

**RELAZIONE**
**DI**
**LABORATORIO DI INTELLIGENZA ARTIFICIALE**

# *Progettazione e implementazione di un percettrone multistrato*

**Docente:**
***Prof. Giorgio Mario Grasso***

**Studente:**
***Antonio Mastrolembo Ventura***

## ANNO ACCADEMICO - 2024/2025

# Indice

# Caso di studio

Il seguente lavoro ha come obiettivo la progettazione e la successiva implementazione di un **percettrone multistrato** che verrà utilizzato per la classificazione delle specie di fiori del dataset Iris. L'obiettivo è quello di addestrare un modello in grado di predire la specie corretta di un fiore basandosi su misure di sepali e petali.
In generale sono previste due fasi (secondo l'approccio di un apprendimento supervisionato):

1. **Addestramento della rete**: in questa fase vengono presentati alla rete neurale i vettori di input con la relativa classe di appartenenza. In questo modo viene confrontato l'output del percettrone con l'output atteso per permettere alla rete di aggiornare i pesi, e di conseguenza, avere un risultato successivo più preciso.

2. **Test della rete**: in questa seconda fase vengono dati alla rete degli input differenti utilizzati rispetto alla fase di addestramento al fine di verificare se ha imparato correttamente.

## Dataset e features

Sia per la fase di addestramento che per quella di test è stato utilizzato il **dataset iris** (ampiamente utilizzato in machine learning) contenente 150 campioni di fiori appartenenti a tre specie: **Iris-setosa**, **Iris-versicolor** e **Iris-virginica**.

Ogni campione è descritto da 4 features continue: **lunghezza del sepalo**, **larghezza del sepalo**, **lunghezza del petalo** e **larghezza del petalo**.

L'obiettivo è quello di classificare ogni fiore in una delle tre specie basandosi appunto su queste quattro caratteristiche.

# Cenni Teorici

Volendo fare un'analogia con i neuroni del cervello umano possiamo definire meglio alcuni aspetti fondamentali del percettrone: un **neurone**, i **pesi** e i **bias**.

Nel cervello umano, i neuroni sono **unità fondamentali di elaborazione delle informazioni**. Si connettono tra loro attraverso **sinapsi**, che regolano il passaggio del segnale elettrico tra neuroni. Un **neurone artificiale** nel percettrone multistrato si comporta in modo simile: riceve segnali in ingresso, li elabora e produce un output. Se il segnale in ingresso supera una certa **soglia** (bias), il neurone **si attiva** e trasmette un segnale agli altri neuroni.

Nel cervello, le **sinapsi** determinano la forza della connessione tra due neuroni: più è forte la connessione, più il segnale viene trasmesso con intensità. Nel percettrone multistrato, i **pesi** (weights) funzionano allo stesso modo: ogni connessione tra neuroni ha un peso che determina **l'importanza** di quel segnale. Se un peso è **alto**, il segnale viene amplificato; se un peso è **basso**, il segnale viene attenuato.

Infine il **bias**: nel cervello, alcuni neuroni hanno una **soglia di attivazione** più alta di altri, nel senso che serve una certa quantità di stimolo per farli attivare. Nel percettrone multistrato, il bias è un valore aggiuntivo che aiuta a **regolare la soglia di attivazione del neurone**. Un bias più alto o più basso può rendere il neurone più o meno propenso ad attivarsi, indipendentemente dagli input ricevuti. Lo immaginiamo per semplicità come se fosse un rubinetto che può regolare la quantità di acqua da far passare.

# Progettazione

L’architettura del Percettrone Multistrato implementato è composta da tre livelli:

- Input Layer: 4 neuroni, corrispondenti alle 4 features del dataset;
- Hidden Layer: un solo livello nascosto con 8 neuroni;
- Output Layer: 3 neuroni, uno per ciascuna classe da predire (Setosa, Versicolor e Virginica).

La **funzione di attivazione** utilizzata è la **Sigmoide**, poiché permette di ottenere valori di output nel range [0,1], facilitando l’interpretazione probabilistica della classificazione. Chiaramente in questo modo ‘superiamo’ il limite del singolo percettrone che è in grado di trattare soltanto sistemi linearmente separabili.

L’addestramento del modello è stato eseguito mediante **Backpropagation**, che consente di aggiornare i pesi, insieme ad alcune tecniche di ottimizzazione quali il **learning rate variabile** e il **momentum**: un learning rate variabile mi permette di controllare la velocità con cui i pesi della rete vengono aggiornati durante l’addestramento, all’inizio è un valore alto, ma man mano che l’errore si riduce il learning rate diminuisce consentendo aggiornamenti più piccoli e precisi. Il momentum invece stabilizza l’addestramento mitigando le oscillazioni e supera il problema dei minimi locali, ma occhio al suo valore:

- Se il momentum è **alto**, l’aggiornamento dei pesi avviene in modo **più veloce** e aiuta a raggiungere rapidamente il minimo dell’errore. Tuttavia, se il gradiente cambia direzione in modo improvviso, la rete potrebbe diventare **instabile** e iniziare a oscillare senza trovare il minimo ottimale.
- Se il momentum è **basso**, la discesa è **più controllata e stabile**, riducendo il rischio di oscillazioni. Tuttavia, l’apprendimento potrebbe essere **più lento** e impiegare più tempo per raggiungere il minimo dell’errore.

# Implementazione

Il Percettrone Multistrato viene implementato attraverso il linguaggio Python. In particolare abbiamo la classe ***MultiLayerPerceptron*** che definisce nel **costruttore** i seguenti parametri:

```
class MultiLayerPerceptron:
    def __init__(self, input_size, hidden_size, output_size, learning_rate, momentum):
        # Architettura della rete
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.momentum = momentum

        # Inizializzo i neuroni
        self.input_neurons = np.zeros(self.input_size)
        self.hidden_neurons = np.zeros(self.hidden_size)
        self.output_neurons = np.zeros(self.output_size)

        # Inizializzo i pesi: input -> hidden e hidden -> output
        self.weights_IH = np.random.rand(self.input_size, self.hidden_size) - 0.5
        self.weights_HO = np.random.rand(self.hidden_size, self.output_size) - 0.5

        # Inizializzo i bias: hidden e output
        self.bias_H = np.zeros(self.hidden_size)
        self.bias_O = np.zeros(self.output_size)

        # Termini di velocità per il momentum (per l'aggiornamento dei pesi)
        self.velocity_IH = np.zeros_like(self.weights_IH)
        self.velocity_HO = np.zeros_like(self.weights_HO)
        self.velocity_H = np.zeros_like(self.bias_H)
        self.velocity_O = np.zeros_like(self.bias_O)
```

Successivamente definisco la funzione di attivazione sigmoide e la sua derivata (utilizzata per l'aggiornamento dei pesi nella backpropagation):

```
    # Funzione di attivazione - Sigmoide
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    # Derivata della sigmoide per la modifica dei pesi
    def sigmoid_derivative(self, x):
        return x * (1 - x)
```

A questo punto posso definire la funzione di **feedforward**, grazie alla quale il mio percettrone multistrato computa l'output finale a partire dagli input iniziali (e passando per il layer nascosto):

```
    def feed_forward(self, sample: np.ndarray):
        # Salvo l'input che mi arriva (i quattro dati del fiore)
        self.input_neurons = sample

        # INPUT -> HIDDEN
        # Somma pesata degli input per i pesi e il bias per la soglia del layer nascosto
        self.hidden_input = np.dot(self.input_neurons, self.weights_IH) + self.bias_H

        # Applico la funzione di attivazione per ottenere l'output del layer nascosto
        self.hidden_neurons = self.sigmoid(self.hidden_input)

        # HIDDEN -> OUTPUT
        self.output_input = np.dot(self.hidden_neurons, self.weights_HO) + self.bias_O

        # Applico la funzione di attivazione per ottenere l'output finale
        self.output_neurons = self.sigmoid(self.output_input)

        # Restituisco l'output che contiene un array
        # di 3 valori (probabilità di appartenenza a ciascuna classe)
        return self.output_neurons
```

Funzione **backpropagation** per l'aggiornamento dei pesi e dei bias:

```
# Backpropagation
    def backpropagation(self, sample: np.ndarray, target: np.ndarray):
        # Calcolo l'errore dell'output (differenza tra output e target)
        output_error = self.output_neurons - target

        # Calcolo del delta del livello di output per vedere di quanto modificare i pesi
        output_delta = output_error * self.sigmoid_derivative(self.output_neurons)

        # Calcolo dell'errore dell'hidden layer
        hidden_error = np.dot(output_delta, self.weights_HO.T)

        # Calcolo del delta del livello nascosto per vedere di quanto modificare i pesi
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_neurons)


        # Aggiorno i pesi - Hidden -> Output
        self.velocity_HO = self.momentum * self.velocity_HO - self.learning_rate *
        np.outer(self.hidden_neurons, output_delta)
        self.weights_HO += self.velocity_HO

        # Aggiorno il bias del livello di output
        self.velocity_O = self.momentum * self.velocity_O - self.learning_rate *
        output_delta
        self.bias_O += self.velocity_O

        # Aggiorno i pesi - Input -> Hidden
        self.velocity_IH = self.momentum * self.velocity_IH - self.learning_rate *
        np.outer(sample, hidden_delta)
        self.weights_IH += self.velocity_IH

        # Aggiorno il bias del livello nascosto
        self.velocity_H = self.momentum * self.velocity_H - self.learning_rate *
        hidden_delta
        self.bias_H += self.velocity_H
```

Funzione **get_loss** per ottenere la perdita:

```
    def get_loss(self, target: np.ndarray, prediction: np.ndarray):
        return np.mean((target - prediction) ** 2)
```

Funzione **train_test** per addestrare e testare la rete neurale con il dataset iris:

```
def train_test(self, X_train, y_train, X_test, y_test, epochs, debug=False):
        # Array per salvare le loss
        train_loss = []
        test_loss = []

        # Addestramento
        for epoch in range(epochs):

            # Riduco il learning rate (in questo caso del 5%) ad ogni epoca
            # per addestrare la rete in modo più preciso
            self.learning_rate *= 0.95

            for i in range(len(X_train)):
                self.feed_forward(X_train[i])
                self.backpropagation(X_train[i], y_train[i])

            # Calcolo la loss per ogni epoca
            train_loss.append(self.get_loss(y_train, np.array([self.feed_forward(x) for x
            in X_train])))

            # Calcolo la loss sul test set
            test_predictions = np.array([self.feed_forward(x) for x in X_test])
            test_loss.append(self.get_loss(y_test, test_predictions))

            print(f"Epoch {epoch+1}/{epochs}: Training Loss = {train_loss[-1]:.4f}, Test
            Loss = {test_loss[-1]:.4f}, Learning Rate = {self.learning_rate:.5f}")
```

```
        print("\n--- Risultati del Test ---")
        correct_predictions = 0
        for i in range(len(X_test)):
            # Calcolo il valore predetto dalla rete
            pred = self.feed_forward(X_test[i])

            # Estraggo il valore più alto (quindi la classe predetta)
            predicted_class = np.argmax(pred)

            # Estraggo la classe reale
            actual_class = np.argmax(y_test[i])

            # Se la classe predetta è uguale a quella reale allora la predizione
            # è corretta
            is_correct = predicted_class == actual_class
            correct_predictions += int(is_correct)

            print(f"Test {i+1}/{len(X_test)} - Predetto: {predicted_class},
            Reale: {actual_class} - {'✔ CORRETTO' if is_correct else '✘
            ERRATO'}")

        # Calcolo l'accuratezza in percentuale
        accuracy = (correct_predictions / len(X_test)) * 100
        print(f"\nAccuracy Finale sul Test Set: {accuracy:.2f}%")
```

## Test della Rete Neurale

Come già detto in precedenza, il test della rete neurale viene fatto sui dati (del dataset iris) non utilizzati in fase di addestramento. Questo per mettere effettivamente alla prova la rete su dei dati che non ha mai visto e per verificare se è in grado di generalizzare. Di seguito il codice che preleva le informazioni dal dataset *iris.data* e separa in due liste differenti i dati (80% per l'addestramento e 20% per il test):

```
# Estraggo i dati dal dataset
data = pd.read_csv("MLP/iris.data", header=None)

# Estraggo i dati relativi ai petali e ai sepali
X = data.iloc[:, :-1].values

# Estraggo i dati relativi alle classi in one-hot encoding
y = pd.get_dummies(data.iloc[:, -1]).values

# Suddivido i dati in training e test set (80% training, 20% test)
X_train = np.concatenate((X[:40], X[50:90], X[100:140]))
y_train = np.concatenate((y[:40], y[50:90], y[100:140]))

X_test = np.concatenate((X[40:50], X[90:100], X[140:150]))
y_test = np.concatenate((y[40:50], y[90:100], y[140:150]))
```

A questo punto creo un'istanza 'nn' della rete neurale ed imposto i vari parametri di configurazione, nello specifico abbiamo:

- 4 Neuroni di Input;
- 1 Layer Nascosto con 8 Neuroni;
- 3 Neuroni di Output;

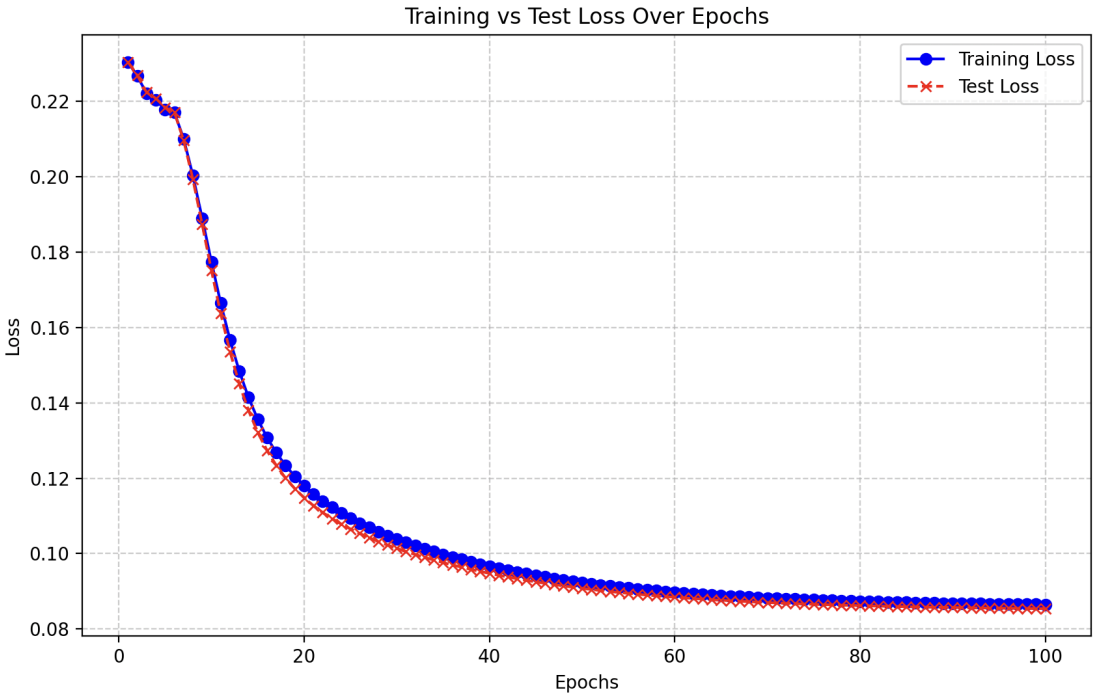
Gli ultimi due parametri sono il learning rate, inizialmente impostato a 0.01 (e che diminuirà del 5% ogni epoca) e il momentum impostato a 0.9:

```
# Creo la rete neurale
nn = MultiLayerPerceptron(4, 8, 3, learning_rate=0.01, momentum=0.9)
```

Infine avvio la funzione per addestrare la rete ed effettuare il test. La rete viene addestrata per 100 epoche:

```
# Addestramento e Test
nn.train_test(X_train, y_train, X_test, y_test, epochs=100, debug=True)
```

Il grafico generato è il seguente:




Il test effettuato con i campioni rimanenti dimostra come la rete abbia effettivamente compreso al meglio in fase di addestramento:

```
--- Risultati del Test ---
Test 1/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 2/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 3/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 4/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 5/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 6/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 7/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 8/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 9/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 10/30 - Predetto: 0, Reale: 0 - ✔ CORRETTO
Test 11/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 12/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 13/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 14/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 15/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 16/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 17/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 18/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 19/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 20/30 - Predetto: 1, Reale: 1 - ✔ CORRETTO
Test 21/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 22/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 23/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 24/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 25/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 26/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 27/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 28/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 29/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO
Test 30/30 - Predetto: 2, Reale: 2 - ✔ CORRETTO

Accuracy Finale sul Test Set: 100.00%
```

# Conclusioni

In conclusione possiamo affermare che il modello presenta **un'ottima capacità di generalizzazione**, perché sia la perdita sul training set che quella sul test set seguono più o meno lo stesso andamento; i campioni vengono classificati in modo corretto e non si sono presentati problemi comuni quali l'**overfitting** o l'**underfitting**.

L'overfitting si sarebbe verificato nel caso in cui la loss del training sarebbe risultata molto bassa, mentre quella del test avrebbe smesso di diminuire o addirittura sarebbe aumentata. Questo avrebbe significato che il modello si adatta troppo ai dati di training e non riesce a gestire nuovi dati

Se invece entrambe le curve (training e test) fossero rimaste molto alte e la loss non fosse scesa abbastanza si sarebbe parlato di underfitting.