Problema affrontato: classificazione dei fiori di Iris

David Moonsmee e Francesco Paladino

Introduzione

Per affrontare il problema della classificazione dei fiori iris, dobbiamo riuscire a identificare correttamente la specie di un fiore a partire dalle sue caratteristiche.

Questo compito implica la capacità di riconoscere schemi complessi nei dati. Per risolvere questo problema, utilizziamo una rete **neurale a più strati (Multi-Layer Perceptron)**, un modello che, attraverso un processo di addestramento, è in grado di apprendere le relazioni tra le caratteristiche dei fiori e le tipologie a cui appartengono.

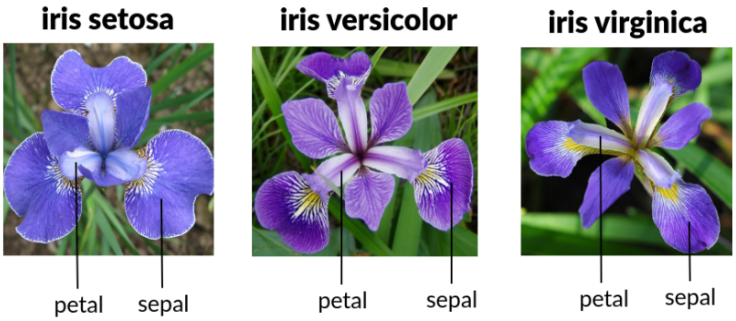
L'obiettivo del progetto è esplorare come questo modello, una volta addestrato, riesca a generalizzare su nuovi dati e a classificare correttamente i fiori, anche quelli mai visti prima.

Informazioni rilevanti

I fiori presi in considerazione nel caso di studio fanno parte della specie di iris. Essi sono caratterizzati da 4 valori: altezza del sepalo, grandezza del sepalo, altezza del petalo, grandezza del petalo.

Le tipologie dei fiori presi in considerazione e studiati nel caso sono solamente tre:

- **Iris Setosa**: la specie più piccola, con 30-50 cm di altezza.
- **Iris Versicolor**: specie intermedia, con 50-80 cm di altezza.
- **Iris Virginica**: la specie più alta, con 60-100 cm di altezza.



Progettazione

Il dataset utilizzato è composto da 150 righe nel seguente formato:

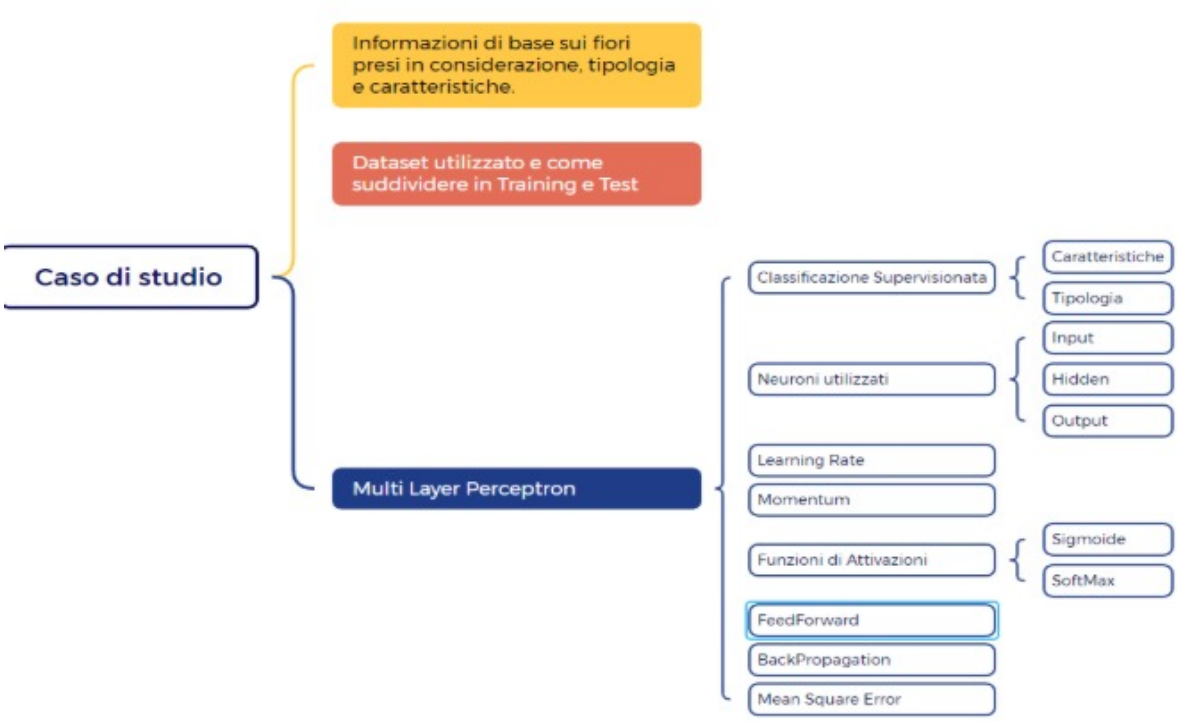
`5.1,3.5,1.4,0.2,Iris-setosa`

Il dataset è stato diviso in:

- **Training Set**: 120 righe (40 Setosa, 40 Versicolor e 40 Virginica).
- **Test Set**: 30 righe (10 per ogni specie).

La divisione è stata effettuata per utilizzare l'apprendimento supervisionato.

Diagramma concettuale del progetto



Analisi dei dati

Per analizzare/studiare i dati, abbiamo utilizzato un **Perceptrone Multistrato** (Multi layer Perceptron) che è un tipo di rete neurale artificiale ispirato al funzionamento dei neuroni biologici (dendrite, soma , etc..).

Un perceptrone è il modello più semplice di una rete neurale. Crea degli output attraverso un input elaborato.

Il **Perceptrone multistrato** è un'estensione del perceptrone semplice e permette di affrontare problemi più complessi, poiché possiamo affrontare problemi non linearmente separabili. Il perceptrone multistrato utilizza un concetto avanzato degli strati.

Struttura del Perceptrone Multistrato

- **Strato di Input**: rappresenta le informazioni dei dati che andiamo a tenere in considerazione. Nel nostro caso noi abbiamo 4 attributi del fiore e il numero di neuroni da utilizzare in input è 4.
- **Strato Nascosto (Hidden)**: questo è uno strato di elaborazione delle informazioni attraverso pesi e funzioni di attivazione. Si può usare un numero variabile, noi abbiamo messo 10 neuroni hidden.
- **Strato di Output**: restituisce il risultato finale in forma probabilistica che equivale al numero di output possibili dei dati. Nel nostro caso stiamo cercando di prevedere solo 3 fiori, quindi utilizziamo solamente 3 neuroni di output.

Modelli matematici

Perceptrone Semplice

$$y(X) = g\left(\sum_{i=1}^{d} W_i X_i + W_0\right) = W^T X$$

dove:

- $y(X)$: è la previsione del perceptrone per un input x (vettore) di d variabili.
- W_i: sono i pesi associati a ciascun input x.
- W_0: è il bias, che aiuta nella valutazione della funzione di attivazione.
- g: funzione di attivazione.
- $W^T X$: prodotto scalare tra il vettore dei pesi W e il vettore degli input X.

Perceptrone Multistrato

$$y_k(X) = g\left(\sum_{j=0}^{M} W_{kj}^{(2)} \cdot g\left(\sum_{i=1}^{C} W_{ji}^{(1)} X_i\right)\right)$$

dove:

- M: numero di neuroni nascosti (hidden).
- C: numero di neuroni di output.
- W_{kj} e W_{ji}: sono i pesi associati.
- g: funzione di attivazione.
- X_i: input.

Pesi: i pesi sono valori numerici associati a ciascun input della rete neurale che determinano quanta importanza ha ogni input nell'elaborazione dei dati e nel processo decisionale. Durante la fase di training, i pesi vengono aggiornati per minimizzare l'errore generato tra l'output previsto e l'output reale, migliorando la precisione.

Bias: il bias è una costante aggiuntiva che aiuta a spostare la funzione di attivazione a destra o sinistra, aumentando la flessibilità del modello anche quando gli input sono tutti a zero.

Funzioni di attivazione

E' una regola matematica che decide quando e con quale intensità attivare il neurone (è come una soglia che guarda la differenza di potenziale elettrico). Nel nostro caso stiamo utilizzando due funzioni di attivazione.

Sigmoide (Hidden Layer)

La funzione sigmoide trasforma il valore in input in un numero compreso tra 0 e 1. Questa proprietà è particolarmente utile nei problemi di classificazione binaria, dove l'output può essere interpretato come una probabilità.

$$g(A) = \frac{1}{1 + e^{-A}}$$

dove:

- $-A$: rappresenta il valore in ingresso che viene applicato alla base del logaritmo naturale.

La sigmoide nei livelli nascosti aiuta a modellare la non linearità tra le caratteristiche degli input. Quindi dopo permette al modello di apprendere relazioni più complesse dove le classi non sono separabili tra una semplice retta nella fase di elaborazione delle informazioni.

Softmax (Output Layer)

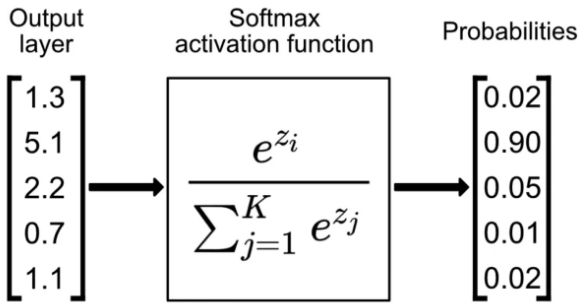
La funzione Softmax viene utilizzata per la classificazione multi-classe perché trasforma un insieme di valori in una distribuzione di probabilità dove la somma di tutti gli output (probabilità) è sempre 1, dove però poi andiamo a prendere l'output con la percentuale più alta.

$$g(a_i) = \frac{e^{a_i}}{\sum_{i=1}^{n} e^{a_i}}$$

dove:

- e^{a_i}: rappresenta l'esponenziale del valore a_i, rendendo tutti i numeri positivi.
- $\sum_{i=1}^{n} e^{a_i}$: rappresenta la somma esponenziale di tutti gli a_i.

Utilizzare Softmax per l'output è l'ideale in quanto abbiamo 3 classi distinte (Setosa, Versicolor, Virginica) dato che andiamo a trasformare tutte e 3 le classi in percentuali, andando successivamente a selezionare quella con la percentuale più alta.



Inizializzazione della Rete Neurale

```
class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate,
        momentum):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.momentum = momentum

        # Inizializzazione dei 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)
```

Learning Rate: è un parametro che serve per regolare l'apprendimento del modello. Determina la grandezza degli aggiornamenti del modello (i pesi della rete) e serve per minimizzare la perdita.

Se il valore di learning è troppo alto stiamo accelerando la convergenza, ma ciò causa instabilità se non è ben gestito, mettendo al rischio il modello dal non convergere mai.

Se il valore di learning invece è troppo basso stiamo rallentando il processo di apprendimento con il rischio di rimanere bloccati nei minimi locali.

- *Nota*: i minimi locali sono dei punti generalmente più bassi rispetto ai punti circostanti, non sono i punti più bassi possibili.

Momentum: il momentum è una tecnica di ottimizzazione che aumenta la velocità di convergenza verso il minimo assoluto, questa tecnica smorza le oscillazioni aggiungendo alla formula di variazione del peso un altro parametro ovvero il momentum.

Inizializzazione dei pesi, bias e momentum

```
# Inizializzazione dei 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)

# Pesi
self.Input_Hidden_weights = starting_weights(self.input_size, self.hidden_size)
self.Hidden_Output_weights = starting_weights(self.hidden_size, self.
    output_size)

# Bias
self.Hidden_Layer_bias = starting_bias(self.hidden_size)
self.Output_Layer_bias = starting_bias(self.output_size)

# Velocit  Input - Hidden - Output
self.Input_Hidden_velocity = np.zeros_like(self.Input_Hidden_weights)
self.Hidden_Output_velocity = np.zeros_like(self.Hidden_Output_weights)

# Velocit  Layer Hidden - Output
self.Hidden_Layer_velocity = np.zeros_like(self.Hidden_Layer_bias)
self.Output_Layer_velocity = np.zeros_like(self.Output_Layer_bias)
```

Funzione di attivazione: Sigmoide e Softmax

```
# Attivazione Sigmoidale per l'input
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

# Attivazione per l'output
def softmax(self, x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True)) # Axis serve per
        indicare le operazioni lungo le righe
return exp_x / np.sum(exp_x, axis=1, keepdims=True) # Keepdims = True per
    restituire array in forma (n, 1)
```

Implementazione FeedForward

E' il processo di propagazione dei dati attraverso una rete neurale in cui l'input viene trasformato strato dopo strato per generare un output, ovvero crea un flusso unidirezionale dell'informazione dall'input fino all'output.

```
# Propagazione della rete neurale, viene usato per indicare che
# la rete deve non pu  andare indietro e pu  andare solo avanti, tramite
# il formato Input - Hidden - Output dove ogni output    l'input per lo strato
    successivo.

def feedforward(self, X):
    self.input_neurons = X # I valori
    self.hidden_input = matrix_mul(self.input_neurons, self.
        Input_Hidden_weights) + self.Hidden_Layer_bias # Calcolo input hidden
    self.hidden_neurons = self.sigmoid(self.hidden_input) # Attivazione

    self.output_input = matrix_mul(self.hidden_neurons, self.
        Hidden_Output_weights) + self.Output_Layer_bias
    self.output_neurons = self.softmax(self.output_input)

    return self.output_neurons
```

Derivazione delle funzioni di attivazione per la back-propagation

Durante la back-propagation, si usa la chain rule per calcolare come l'errore influisce sui parametri (pesi, bias e momentum) di ogni strato. La derivata della funzione di attivazione serve perché senza non saremmo in grado di capire come l'errore si propaga attraverso ogni strato della rete e di conseguenza non potremmo aggiornare i pesi.

```
# Serve per la derivazione per la backpropagation poich
# Esso ci dice quanto scostamento c'   tra l'output del neurone rispetto al suo input
def sigmoide_derivate(self, x):
    s = self.sigmoid(x)
    return s * (1 - s)

# Discorso analogo del sigmoide
def softmax_derivative(self, x):
    s = self.softmax(x)
    return s * (1 - s)
```

Back-propagation

La back-propagation serve per addestrare una rete neurale:

- **Calcola l'errore:** confronta l'output della rete con il risultato desiderato.
- **Propaga l'errore all'indietro:** calcola come l'errore si propaga attraverso ogni strato, usando le derivata delle funzioni di attivazione.
- **Aggiorna i parametri:** modifica i pesi e i bias della rete per ridurre l'errore, usando i gradienti calcolati e il momentum.

Regola matematica

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}}$$

dove:

- Il primo termine misura come l'errore varia rispetto all'attivazione a_j del neurone j.
- Il secondo termine misura come l'attivazione a_j varia rispetto al peso W_{ji}.

Codice back-propagation 1/2

```
# Propagazaione all'indietro, serve per allenare il modello
# propaga l'errore all'indietro aggiustando i pesi, per minimizzare l'errore.
def backpropagation(self, X, Y):

    output_error = self.output_neurons - Y # Differenza tra Previsione e Valore vero
    output_delta = output_error * self.softmax_derivative(self.output_input) # Errore tra Input - Output

    # Stesso discorso per Hidden (Per  mentre torna indietro aggiungiamo i pesi del livello)
    hidden_error = matrix_mul(output_delta, self.Hidden_Output_weights.T) # Trasposizione, poich  torna all'indietro l'errore.
    hidden_delta = hidden_error * self.sigmoide_derivate(self.hidden_input)
```

Codice back-propagation 2/2

```
    # Aggiornamento della velocit  e dei pesi
    # Utilizziamo la velocit  per tenere traccia dei pesi passati
    self.Hidden_Output_velocity = self.momentum * self.Hidden_Output_velocity -
         self.learning_rate * matrix_mul(self.hidden_neurons.T, output_delta)
    self.Output_Layer_velocity = self.momentum * self.Output_Layer_velocity -
        self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)
    self.Input_Hidden_velocity = self.momentum * self.Input_Hidden_velocity -
        self.learning_rate * matrix_mul(X.T, hidden_delta)
    self.Hidden_Layer_velocity = self.momentum * self.Hidden_Layer_velocity -
        self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
    # Di conseguenza aggiorniamo i pesi.
    self.Output_Layer_bias += self.Output_Layer_velocity
    self.Input_Hidden_weights += self.Input_Hidden_velocity
    self.Hidden_Layer_bias += self.Hidden_Layer_velocity
    self.Hidden_Output_weights += self.Hidden_Output_velocity
```

Nota: il gradiente è un vettore che indica la salita più ripida. Conoscendo il punto e il verso possiamo utilizzare la tecnica della discesa del gradiente, ovvero possiamo andare nel lato opposto del gradiente calcolato tramite componenti vettoriali.

Funzione di errore (MSE)

La funzione di errore calcola l'errore quadratico medio tra il risultato generato e il risultato reale. Infatti si chiama **Mean Square Error**.

$$E(w) = \sum_{i=1}^{c} (y_i - y_i')^2$$

```
def Mean_Squared_Error(self, Y_train, Y_test):
        return np.square(np.subtract(Y_train, Y_test)).mean()
```

Codici esterni

Qui sono presenti tutti i codici principali per l'esecuzione del programma, tralasciando le altre funzioni che sono tutte per il processamento e per l'elaborazione dei dati.

Costruzione modello

```
# Generalizzazzione del Modello
def Model(X, Y):
        input_size = len(X[0]) # Nel nostro caso sono 4, poich  1 Neurone di
            Input = 1 feature
        hidden_size = 10
        output_size = len(Y[0]) # Qui invece 3, poich  1 Neurone di Output = 1
             classe
        learning_rate = 0.01
        momentum = 0.9
        mlp = MLP(input_size, hidden_size, output_size, learning_rate, momentum
            )
        return mlp
```

Addestramento del modello

```
def ModelTraining(mlp, iterazione, Feature_train, Target_train, Feature_test,
    Target_test):
        MSE_train = np.zeros(iterazione) # Perdita derivante dal trainng
            dataset
        MSE_test = np.zeros(iterazione) # Perdita derivante dal test dataset

        for i in range(iterazione):
            train_output = Output(mlp, Feature_train)
            train_errorprop = mlp.backpropagation(Feature_train, Target_train)
            MSE_train[i] = mlp.Mean_Squared_Error(Target_train, train_output)

            test_predictions = Output(mlp, Feature_test)
            MSE_test[i] = mlp.Mean_Squared_Error(Target_test, test_predictions)

        return MSE_train, MSE_test
```

Esportazione del modello in formato JSON

Utilizzando il formato JSON (Javascript Object Notation) è possibile esportare il modello in un formato leggibile sia dalle macchine che dagli esseri umani. Grazie alla sua struttura gerarchica e stratificata, è possibile analizzare in modo rapido e intuitivo il contenuto del modello, inclusi i parametri, i pesi e la sua architettura.

```
def export_model(mlp, file_path):
    IrisDataModel = {
        "input_size": mlp.input_size,
        "hidden_size": mlp.hidden_size,
        "output_size": mlp.output_size,
        "learning_rate": mlp.learning_rate,
        "momentum": mlp.momentum,

        "Weights": {
            "Input_Hidden_weights": mlp.Input_Hidden_weights.tolist(),
            "Hidden_Output_weights": mlp.Hidden_Output_weights.tolist(),
        },
        "Bias": {
            "Hidden_Layer_bias": mlp.Hidden_Layer_bias.tolist(),
            "Output_Layer_bias": mlp.Output_Layer_bias.tolist(),
        },
        "Momentum": {
            "Input_Hidden_velocity": mlp.Input_Hidden_velocity.tolist(),
            "Hidden_Output_velocity": mlp.Hidden_Output_velocity.tolist(),
            "Hidden_Layer_velocity": mlp.Hidden_Layer_velocity.tolist(),
            "Output_Layer_velocity": mlp.Output_Layer_velocity.tolist(),
        }
    }

    with open(file_path, 'w') as file:
        json.dump(IrisDataModel, file, indent=2)
    print(f"Modello Creato in {file_path}")
```

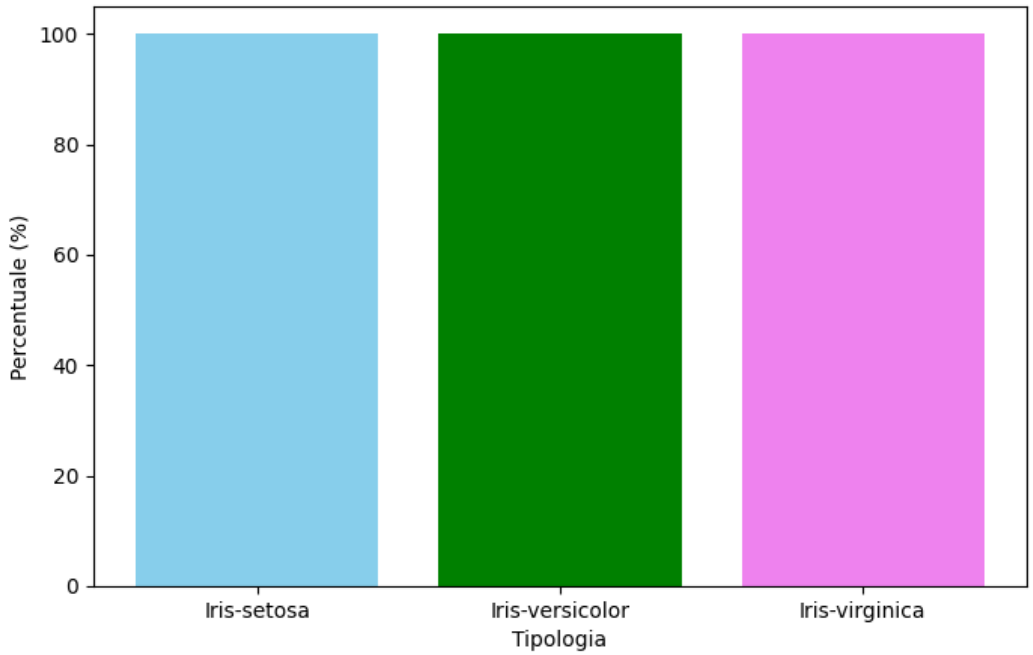
Nota: è presente anche un codice simile per l'importazione.

Fase di testing del modello

```
def TestingModel(mlp, X, Y, Types):

    Feature_predict = Output(mlp, X) # Tutte le probabilit  generate dei feature
    #print(Feature_predict)

    Flower_predict = np.argmax(Feature_predict, axis=1) # Prende il max indice della riga [0.1, 0.7, 0.5]
    #print(Flower_predict)

    Flower_list = np.argmax(Y, axis=1) # I veri valori presenti in formato [0, 1, 2]
    #print(Flower_list)

    for element in range(len(Feature_predict)):

        Confidence = (np.max(Feature_predict[element]) * 100).round(2) # Percentuale arrotondato tramite probabilit  [0.1, 0.7, 0.5]
        #print(Confidence)

        Predicted_Flower = Types[Flower_predict[element]] # Estrazione fiore in formato Stringa
        #print(Predicted_Flower)

        Current_Flower = Types[Flower_list[element]]

        if (Predicted_Flower == Current_Flower): # Comparazione tra le stringhe
            Result = "Corretto"
        else:
            Result = "Errato"

        print(f"Iterazione: [{element+1}] / Fiore Corrente: {Current_Flower} / Percentuale predetta: {Confidence}% / Esito: {Result}")

    Correct_Flowers = np.sum(Flower_predict == Flower_list) # Somma dei fiori corretti
    Accuracy_tot = (Correct_Flowers / len(Y)) * 100 # Percentuale totale
```

Risultati

Grafico dell'accuracy



La funzione serve a verificare l'efficacia del modello importato, che è già stato addestrato, nel prevedere correttamente il fiore del dataset Iris. Il grafico non mostra l'intero processo di addestramento, ma si concentra esclusivamente sul **tasso di successo del modello nel classificare correttamente i fiori**.

Il grafico è composto da tre colonne, ognuna delle quali rappresenta il tasso di successo del modello per ciascuna delle tre classi di fiori nel dataset. Ogni pilastro indica la percentuale di correttezza con cui il modello è riuscito a prevedere un fiore, con l'obiettivo finale di raggiungere il 100% di accuratezza per ogni classe.

Grafico delle perdite in funzione del training e test set

In parallelo, possiamo visualizzare il **grafico delle perdite di errore (MSE)** che mostra come varia l'errore sia nel set di addestramento che in quello di test. Questo grafico ci permette di confermare se il nostro modello ha generalizzato correttamente, evitando situazioni di **Overfitting** (dove il modello memorizza troppo i dati, perdendo capacità di generalizzazione) o **Underfitting** (quando il modello non riesce a imparare abbastanza dai dati, portando a errori significativi).

Se l'errore diminuisce su entrambi i set allora significa che il modello sta migliorando e si comporta bene sia sui dati su cui è stato allenato, sia su quelli nuovi che non ha mai visto prima.

Conclusioni

Abbiamo affrontato il problema della classificazione dei fiori iris tramite l'utilizzo di un perceptrone multistrato, che ci è servito per classificare i fiori tramite le loro caratteristiche. Il modello ha mostrato un buon apprendimento con una riduzione dell'errore sia nei dati di addestramento che di test, mostrando di essere capace di generalizzare.

L'accuratezza ottenuta conferma l'efficacia del modello nel distinguere le diverse tipologie di fiori. Le funzioni di attivazione Sigmoide e Softmax hanno aiutato a gestire la complessità dei dati, mentre l'uso del momentum e di un learning rate adeguato ha consentito ad un addestramento stabile ed efficiente. Abbiamo suddiviso con cura il dataset (80%/20%) per valutare in modo affidabile le prestazioni del modello e abbiamo reso il modello facilmente riutilizzabile, poiché abbiamo creato un'esportazione in formato JSON.

In conclusione, il nostro studio dimostra come le reti neurali possano essere strumenti efficaci per la classificazione di dati complessi, in quanto i programmi tradizionali non riuscirebbero mai a raggiungere questi risultati con così tanta efficienza.