UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Laboratorio di Intelligenza Artificiale

Progettazione e implementazione di un percettrone multistrato

Docente:
Prof. Giorgio Mario Grasso

Studente:
Andrea Amante

ANNO ACCADEMICO 2024/25

Indice

1 Introduzione

1.1 Caso di studio

Il caso di studio presentato riguarda la progettazione e l'implementazione di un **percettrone multistrato** (MLP) utilizzato per la classificazione di tre specie differenti di iris basata sulle loro caratteristiche fisiche.
L'MLP progettato utilizza l'**apprendimento supervisionato**, generalmente suddiviso in due fasi:

1. **Fase di addestramento**, nella quale si presentano alla rete neurale i vettori di input e la classe di appartenenza associata. In questa fase, i dati ì vengono processati e l'output ottenuto viene confrontato con il risultato atteso, in modo da aggiornare i valori dei *pesi* e dei *bias* al fine di aumentarne l'accuratezza.
2. **Fase di test**, dove la rete neurale analizza dati diversi da quelli presentati nella fase precedente e fornisce i risultati in base ai parametri impostati in fase di allenamento.

1.2 Dataset utilizzato

Per l'apprendimento e la fase di test è stato impiegato l'**iris dataset**, uno dei primi dataset utilizzati per problemi di classificazione. Il dataset contiene un totale di 150 campioni, ciascuno dei quali rappresenta un fiore appartenente a una delle tre specie di iris.

1.2.1 Le features

Per ogni campione si hanno 4 *features*, ossia le caratteristiche misurate per ogni fiore, e 3 *classi*, ovvero la specie di appartenenza.
In questo caso, le features esaminate sono:

1. **Lunghezza del sepalo** (*sepal length*)
2. **Larghezza del sepalo** (*sepal width*)
3. **Lunghezza del petalo** (*petal length*)
4. **Larghezza del petalo** (*petal width*)

1.2.2 Le classi

Per quanto riguarda le classi, i campioni analizzati sono:

1. **Iris-setosa**
2. **Iris-versicolor**
3. **Iris-virginica**

I dati sono rappresentati in formato **CSV** (comma-separated values), dove i primi 4 valori per ogni riga rappresentano le caratteristiche fisiche del fiore e l'ultima la relativa classe di appartenenza. Infine, si può affermare che il dataset è bilanciato in quanto per ciascun fiore sono presenti 50 campioni.

2 Progettazione

La rete è stata progettata con un'architettura composta da tre livelli principali: *input layer*, *hidden layer* e *output layer*. Ogni livello svolge un ruolo specifico nel processo di apprendimento e classificazione.

2.1 Architettura della rete neurale

L'MLP sviluppato è una rete **feedforward**, in cui il segnale si propaga in una sola direzione, dall'input all'output, senza cicli o loop. L'informazione fluisce attraverso una serie di strati di neuroni. Parte dallo strato di input, per poi passare per più strati nascosti fino allo strato di output. La rete neurale è formata da:

- Un **input layer** con 4 neuroni, uno per ciascuna caratteristica del fiore da analizzare.
- Un **hidden layer** con un numero variabile di neuroni.
- Un **output layer** con 3 neuroni, ciascuno corrispondente ad una delle classi di iris.

La rete utilizza la funzione di attivazione **sigmoidea**, che determina il valore del segnale in uscita da ciascun neurone presente nell'hidden layer. Il processo di apprendimento è basato sull'**algoritmo di backpropagation**, con l'aggiunta del **learning rate** variabile e del **momentum**, parametri utilizzati per migliorare la stabilità e la velocità di convergenza.

2.1.1 I neuroni

Il neurone rappresenta l'unità fondamentale di una rete neurale. Riceve uno o più segnali di input e, dopo averli elaborati, produce un segnale di output. I neuroni di uno strato sono connessi ai neuroni dello strato successivo tramite dei collegamenti, ognuno dei quali ha un peso associato. L'output di un neurone è dato da:

$$y = \sigma(\sum_{i=1}^{n} x_i w_i + b)$$

Dove:

1. σ è la funzione di attivazione, in questo caso la funzione sigmoidea.
2. x_i sono gli input del neurone.
3. w_i è il peso associato a ciascun input.
4. b è il bias.

2.1.2 I pesi

I pesi (w) determinano l'importanza di ciascun input per ogni neurone. Il peso di ciascun collegamento viene modificato durante il processo di addestramento per minimizzare l'errore del modello, permettendo alla rete di apprendere. In particolare:

- Un peso alto amplifica il contributo di un input.
- Un peso basso riduce il contributo di un input.

2.1.3 I bias

Il bias (b) è un parametro che sposta la funzione di attivazione di un neurone, influenzando la sua propensione ad attivarsi. Può essere positivo (*eccitatorio*), contribuendo ad aumentare la somma pesata degli input, o negativo (*inibitorio*), rendendo l'attivazione più difficile.

2.1.4 Il learning rate

La funzione di perdita misura l'errore tra l'output previsto e quello effettivo di un modello di rete neurale. L'obiettivo è quello di minimizzare il valore di questa funzione in modo da ottenere risultati più accurati. Il **learning rate** influenza la stabilità del processo di ottimizzazione. In particolare, valori molto alti possono portare ad una convergenza più rapida con il rischio di saltare il minimo globale e ottenere una soluzione imprecisa. Al contrario, valori molto bassi garantiscono una maggior stabilità nel processo di apprendimento ma a scapito della velocità, richiedendo più tempo per raggiungere la convergenza.

2.1.5 Il momentum

Il **momentum** viene utilizzato per accelerare la convergenza e mitigare le oscillazioni nel processo di apprendimento. Questo fattore tiene conto delle variazioni precedenti nei pesi, accumulando una "*velocità*" che aiuta a superare i minimi locali e a mantenere la direzione dell'ottimizzazione. Valori di momentum molto alti possono portare a un'importante accelerazione verso il minimo globale, ma con il rischio di instabilità e oscillazioni se il gradiente cambia improvvisamente direzione. Al contrario, valori molto bassi assicurano un processo di apprendimento più stabile e controllato, ma possono rallentare la discesa verso il minimo.

3 Implementazione

La rete neurale è implementata nella classe `NeuralNetwork`.

3.1 Librerie utilizzate

Le librerie utilizzate per la realizzazione di un MLP sono:

- `numpy`, necessaria per effettuare calcoli numerici e manipolare matrici e vettori.
- `matplotlib.pyplot`, che permette la visualizzazione di grafici.
- `json`, utilizzata per esportare ed importare i parametri della rete in formato JSON.

3.2 Funzioni principali

3.2.1 Funzione costruttore

Costruttore (`__init__`), che inizializza i parametri principali della rete, come:

- Numero di neuroni per input, hidden e output layer.
- Learning rate e momentum.
- Pesi iniziali generati casualmente e bias.
- Velocità iniziale per l'aggiornamento dei pesi con il momentum.

```
1  def __init__(self, input_size: int, hidden_size: int, output_size: int
       , eta: float=.01, momentum: float=.9):
2      self.input_size = input_size
3      self.hidden_size = hidden_size
4      self.output_size = output_size
5      self.eta = eta
6      self.momentum = momentum
7
8      self.input_neurons = np.zeros((1, self.input_size))
9      self.hidden_neurons = np.zeros((1, self.hidden_size))
10     self.output_neurons = np.zeros((1, self.output_size))
11
12     self.IH_weights = np.random.uniform(-1, 1, (self.input_size, self.
           hidden_size))
13     self.HO_weights = np.random.uniform(-1, 1, (self.hidden_size, self
           .output_size))
14
15     self.HL_bias = np.zeros((1, self.hidden_size))
16     self.OL_bias = np.zeros((1, self.output_size))
17
18     self.IH_velocity = np.zeros_like(self.IH_weights)
19     self.HO_velocity = np.zeros_like(self.HO_weights)
20     self.HL_velocity = np.zeros_like(self.HL_bias)
21     self.OL_velocity = np.zeros_like(self.OL_bias)
```

3.2.2 Funzione `sigmoid` e `sigmoid_derivate`

Necessarie per ottenere il calcolo della funzione di attivazione e della sua derivata, utilizzata nel processo di backpropagation.

```
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(self, x):
    s = self.sigmoid(x)
    return s * (1 - s)
```

3.2.3 Funzione `feedforward` e `backpropagation`

La funzione di feedforward propaga i dati dagli input agli output, mentre la funzione di backpropagation esegue l'aggiornamento dei pesi e dei bias.

```
def feedforward(self, X: vector) -> vector:
    self.input_neurons = X
    self.hidden_input = np.dot(self.input_neurons, self.IH_weights) +
        self.HL_bias
    self.hidden_neurons = self.sigmoid(self.hidden_input)

    self.output_input = np.dot(self.hidden_neurons, self.HO_weights) +
         self.OL_bias
    self.output_neurons = self.sigmoid(self.output_input)

    return self.output_neurons


def backpropagation(self, X: vector, y: vector) -> None:
    output_error = self.output_neurons - y
    output_delta = output_error * self.sigmoid_derivative(self.
        output_input)

    hidden_error = np.dot(output_delta, self.HO_weights.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(self.
        hidden_input)

    self.HO_velocity = self.momentum * self.HO_velocity - self.eta *
        np.dot(self.hidden_neurons.T, output_delta)
    self.HO_weights += self.HO_velocity

    self.OL_velocity = self.momentum * self.OL_velocity - self.eta *
        np.sum(output_delta, axis=0, keepdims=True)
    self.OL_bias += self.OL_velocity

    self.IH_velocity = self.momentum * self.IH_velocity - self.eta *
        np.dot(X.T, hidden_delta)
    self.IH_weights += self.IH_velocity

    self.HL_velocity = self.momentum * self.HL_velocity - self.eta *
        np.sum(hidden_delta, axis=0, keepdims=True)
    self.HL_bias += self.HL_velocity
```

3.2.4 Funzione `get_loss`

Calcola la funzione di perdita, utilizzando l'errore quadratico medio.

```
def get_loss(self, y_true: vector, y_pred: vector) -> float:
    return np.mean((y_true - y_pred) ** 2)
```

3.2.5 Funzione `train` e `test`

Utilizzate per allenare e testare la rete neurale con i dati forniti.

```
def train(self, X: vector, y: vector, epochs: int, debug: bool=False)
    -> None:
    loss = np.zeros(epochs)

    for i in range(epochs):
        self.feedforward(X)
        self.backpropagation(X, y)

        loss[i] = self.get_loss(y, self.output_neurons)

    if debug:
        for i in range(epochs):
            print(f"Epoch {i+1}/{epochs}, Loss: {loss[i]:.2f}")

        plt.figure(figsize=(8, 5))
        epochs = np.arange(len(loss))

        plt.plot(epochs, loss, label='Loss', marker='o', markersize=4)
        plt.title('Loss in function of epochs')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.grid(True)
        plt.legend()
        plt.show()


def test(self, X: vector, y: vector, debug: bool=False) -> float:
    predictions = self.feedforward(X)

    predicted_classes = np.argmax(predictions, axis=1)
    actual_classes = np.argmax(y, axis=1)

    correct_predictions = np.sum(predicted_classes == actual_classes)
    accuracy = (correct_predictions / y.shape[0]) * 100

    if debug:
        species = ["Iris-setosa", "Iris-versicolor", "Iris-virginica"]
        for i, prediction in enumerate(predictions):
            percentage = (np.max(prediction) * 100).round(2)
            predicted = species[predicted_classes[i]]
            actual = species[actual_classes[i]]
            print(f"{percentage}% that it is {predicted}. Actual: {
                actual}. Result: {predicted == actual}")
        print(f"Test Accuracy: {accuracy:.2f}%")

    return accuracy
```

3.3 Funzioni secondarie

Le funzioni `export_model`, `import_model`, e `train_and_test` non sono necessarie al funzionamento della rete, ma hanno l'obiettivo di aggiungere funzionalità al progetto. In particolare:

- `import_model` e `export_model` permettono di importare ed esportare i parametri in formato JSON, ripristinando lo stato della rete neurale.
- `train_and_test` combina l'addestramento e la valutazione del modello in una singola funzione, calcolando la perdita per il training set e il test set durante ogni epoca e visualizzando graficamente le perdite.

4 Test della rete neurale

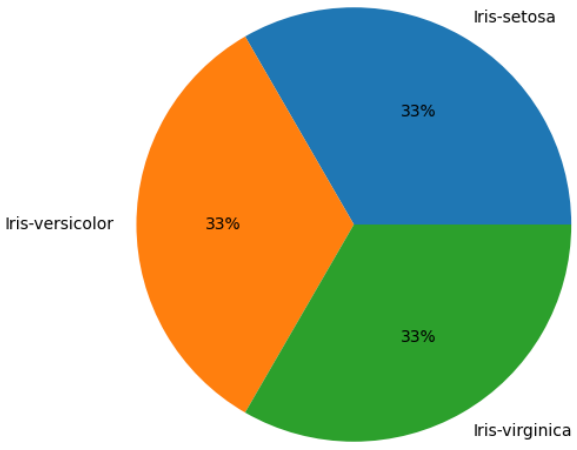
4.1 Librerie utilizzate

Le librerie necessarie vengono importate per gestire i dati e la rete neurale.

- `pandas`, per leggere e manipolare il dataset in formato CSV.
- `numpy`, utilizzata in questo caso per operazioni matriciali.
- `NeuralNetwork`, la classe implementata nel file `NeuralNetwork.py`, che contiene la definizione della rete neurale.

```
1  import pandas as pd
2  import numpy as np
3  from NeuralNetwork import NeuralNetwork
4
5  df = pd.read_csv("iris.data", header=None)
6  df.columns = ["Sepal length", "Sepal width", "Petal length", "Petal
       width", "Class"]
7
8  X = df[["Sepal length", "Sepal width", "Petal length", "Petal width"
       ]].values
9  y = pd.get_dummies(df["Class"]).values
10
11 X_train = np.concatenate((X[:40], X[50:90], X[100:140]))
12 y_train = np.concatenate((y[:40], y[50:90], y[100:140]))
13 X_test = np.concatenate((X[40:50], X[90:100], X[140:150]))
14 y_test = np.concatenate((y[40:50], y[90:100], y[140:150]))
```

Il dataset IRIS viene caricato da un file CSV chiamato `iris.data` utilizzando `pandas`. Successivamente, i dati vengono preparati per l'addestramento della rete. I dati sono suddivisi in modo bilanciato per garantire la presenza di ogni classe sia nel training set che nel test set.



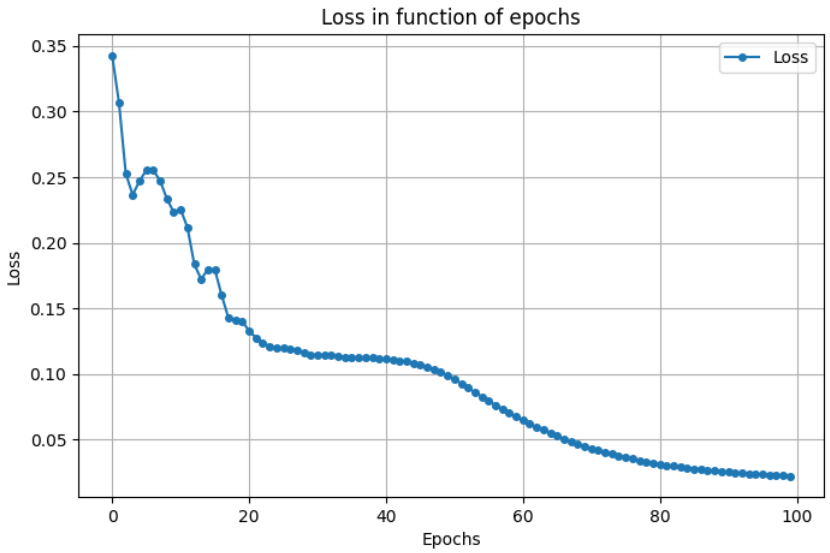
4.1.1 Fase di addestramento

Per la fase di addestramento, viene inizializzata una rete neurale con:

- 4 neuroni nell'input layer (uno per ciascuna feature).
- 8 neuroni nel hidden layer.
- 3 neuroni nell'output layer (uno per ciascuna classe).
- Learning rate impostato a 0.01.
- Momentum impostato a 0.9.

```
nn = NeuralNetwork(4, 8, 3, eta=.01, momentum=0.9)
nn.train(X_train, y_train, epochs=100, debug=True)
```

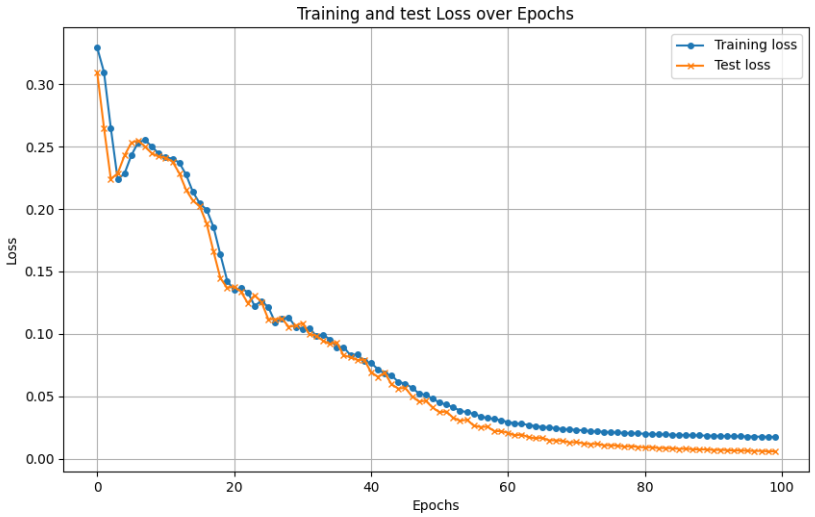
La rete viene addestrata per 100 epoche, e se `debug=True`, vengono visualizzate la perdita per epoca e il grafico dell'errore.



Il grafico mostra come il valore di `loss` viene dimezzato in appena 10 epoche e raggiunge valori trascurabili nelle ultime 20 epoche.

4.1.2 Perdita per il training set e test set

La funzione `train_and_test` genera grafici delle perdite sul training e sul test set durante l'addestramento. Se la perdita sul training set e sul test set diminuiscono entrambe in modo simile durante l'addestramento, si può affermare che il modello dimostra un'ottima capacità di generalizzazione.



In caso contrario, si potrebbero verificare due condizioni:

- **Overfitting**,quando il modello si adatta eccessivamente ai dati di addestramento, mostrando una perdita molto bassa su di essi ma alta sul test set.
- **Underfitting**, si manifesta quando il modello non riesce a catturare adeguatamente i pattern nei dati, con perdite elevate sia sul training set che sul test set.

5 Conclusioni

In conclusione si può affermare come il percettrone multistrato sviluppato abbia presentato ottimi risultati in fase di test, classificando correttamente i campioni in modo molto accurato. Tramite l'implementazione di tecniche di ottimizzazione quali learning rate variabile e momentum, si è raggiunta rapidamente la convergenza evitando sia l'overfitting che l'underfitting.