

Diabetes Health Classifier (DHC)

Una rete neurale per la predizione del rischio di diabete

Author

Corrado Vaccaro



Contents

1	Introduzione	4
2	Architettura e Componenti della rete neurale DHC	5
2.1	Architettura della rete neurale	5
2.2	Schema dell'architettura della rete	5
2.3	Analisi delle principali funzioni implementate	6
3	Dati di input.....	10
3.1	Fonte dei dati	10
3.2	Esempio di dati	10
3.3	Preprocessing dei dati.....	10
3.4	Perché è necessaria la normalizzazione dei dati.....	12
4	Processo di Addestramento	14
4.1	Processo di Forward Propagation.....	14
4.2	Processo di Backward Propagation	20
5	Risultati e loro analisi	25
5.1	Curva di apprendimento.....	25
5.2	Valutazione finale	26
5.3	Discussione dei risultati.....	28
6	Fase di testing.....	30
6.1	Caricamento del modello salvato	30
6.2	Testing su dataset da 2000 esempi	30
6.3	Testing su dataset da 5000 esempi	32
6.4	Analisi generale dei test.....	34
7	Predizione su nuovi pazienti	35
7.1	Introduzione	35
7.2	Procedura di predizione	35
7.3	Dataset dei nuovi pazienti.....	36
7.4	Risultati ottenuti	36
7.5	Analisi dei risultati.....	39

7.6	Considerazioni finali	39
8	Conclusioni	40
8.1	Riepilogo dei principali risultati.....	40
8.2	Direzioni future per miglioramenti e ricerca.....	40
8.3	Considerazione finale.....	41
9	Appendici	42
9.1	Lo script di training: Diabets_training.....	42
9.2	Lo script di testing: Diabets_testing	45
9.3	Lo script di produzione: Diabets_predict	47
9.4	Significato dei campi del dataset	49
9.5	Moduli per il funzionamento della rete neurale	51
10	Lista delle figure	52
11	Revisioni	53

1 Introduzione

La presente relazione documenta la progettazione, implementazione e analisi di **Diabetes Health Classifier (DHC)**, una rete neurale sviluppata per affrontare il problema della classificazione della salute rispetto al rischio di diabete.

L'obiettivo principale di DHC è prevedere la presenza di diabete in un individuo sulla base di una serie di indicatori sanitari raccolti attraverso il dataset pubblico "**Diabetes Health Indicators Dataset**", disponibile su Kaggle ([link al dataset](#)).

Il dataset è composto da oltre 70.693 record, ciascuno descritto tramite 21 variabili, tra cui fattori demografici (età, sesso), comportamentali (abitudine al fumo, attività fisica), condizioni di salute (pressione alta, colesterolo alto) e indicatori medici (indice di massa corporea, depressione).

Il compito affidato alla rete DHC è un **problema di classificazione binaria**: determinare se un individuo sia affetto da diabete o meno, a partire dagli indicatori forniti.

Tale compito ha una forte rilevanza pratica, considerando che il diabete è una malattia cronica in crescita a livello globale e una diagnosi precoce può migliorare significativamente gli esiti clinici.

DHC è stato realizzato con l'obiettivo di:

- Costruire un modello predittivo accurato e generalizzabile.
- Analizzare l'importanza dei diversi fattori sanitari nella previsione.
- Esplorare il comportamento della rete durante il processo di addestramento attraverso grafici e metriche di performance.

La rete neurale è stata implementata utilizzando il linguaggio Python e librerie di machine learning come NumPy e Pandas.

Verranno forniti dettagli tecnici riguardanti l'architettura della rete, il processo di training, la valutazione delle prestazioni e l'interpretazione dei risultati.

2 Architettura e Componenti della rete neurale DHC

2.1 Architettura della rete neurale

La rete neurale progettata per il progetto Diabetes Health Classifier (DHC) è una rete feedforward semplice ma efficace per un compito di classificazione binaria (presenza o assenza di diabete).

Struttura della rete:

Strato	Numero di Neuroni	Funzione di Attivazione
Input Layer	21	Nessuna
Hidden Layer	64	Leaky ReLU
Output Layer	1	Sigmoid

Descrizione degli strati

1. Input Layer:

- Descrizione: Riceve 21 indicatori di salute provenienti dal dataset.
- Funzione di attivazione: Nessuna funzione di attivazione applicata.

2. Hidden Layer:

- Numero di neuroni: 64.
- Funzione di attivazione: ReLU Leaky, che introduce non linearità e risolve il problema dei neuroni "morti" tipico della ReLU classica, permettendo una piccola pendenza anche per valori negativi:

$$f(x) = \begin{cases} x & \text{se } x > 0 \\ 0.01x & \text{se } x \leq 0 \end{cases}$$

3. Output Layer:

- Numero di neuroni: 1.
- Funzione di attivazione: Sigmoid, che converte l'output in una probabilità compresa tra 0 e 1, adatta alla classificazione binaria (malato / non malato):

$$f(x) = \frac{1}{1 + e^{-x}}$$

2.2 Schema dell'architettura della rete

L'architettura della rete DHC è rappresentata dal seguente schema:

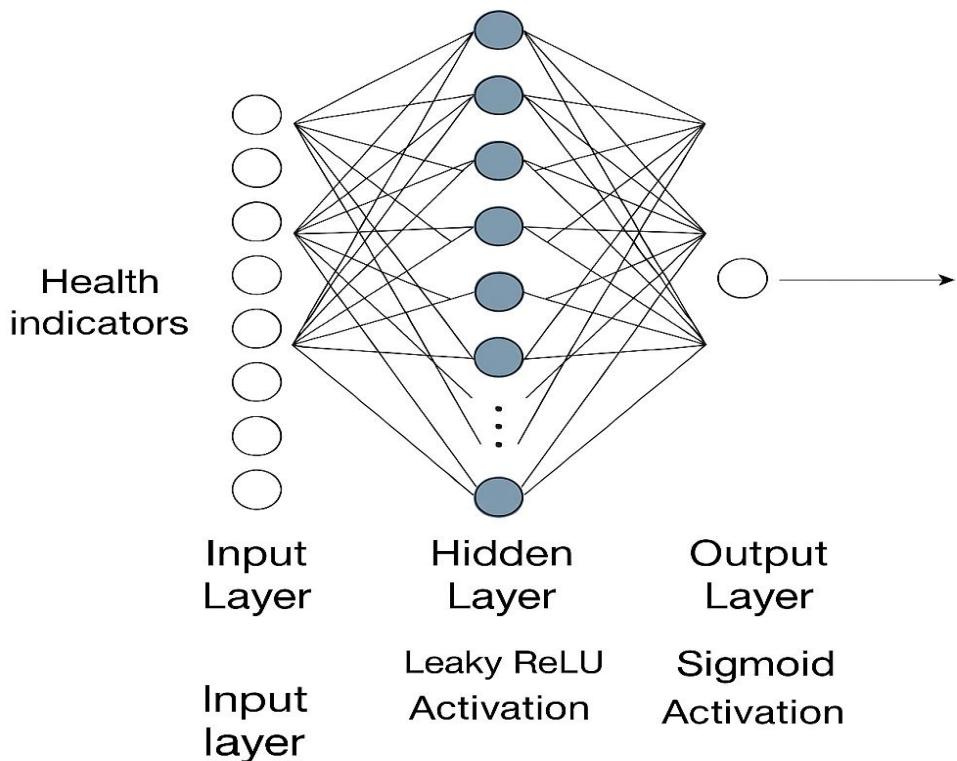


Figura 1 - Rappresentazione grafica della DHC

mentre di seguito abbiamo la sua rappresentazione logica:

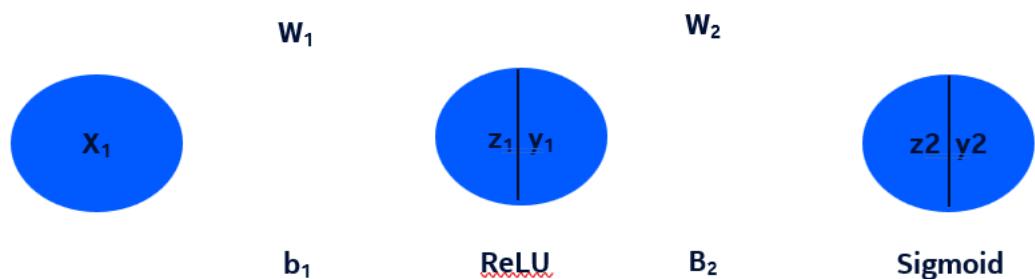


Figura 2 - Rappresentazione logica della DHC

2.3 Analisi delle principali funzioni implementate

Di seguito vengono riportate e descritte le principali funzioni implementate nello script `Diabets_training.py`, all'interno della classe `NeuralNetwork`.

Funzione `_init_`

```
python
class NeuralNetwork:
    def __init__(self, hidden_layer_size=100):
        self.hidden_layer_size = hidden_layer_size
```

Scopo:

Costruisce l'oggetto "rete neurale", inizializzando il parametro che definisce il numero di neuroni nello strato nascosto.

Parametri di input:

- hidden_layer_size (default = 100): numero di neuroni del layer nascosto.

Output:

- Nessun valore restituito. Viene creato un attributo interno self.hidden_layer_size.

Funzionamento interno:

- Semplice assegnazione del valore passato alla variabile d'istanza.

Funzione `_init_weights`

```
python
def _init_weights(self, input_size, hidden_size):
    rng = np.random.default_rng()
    limit1 = np.sqrt(6 / (input_size + hidden_size))
    self._W1 = rng.uniform(-limit1, limit1, (input_size, hidden_size))
    self._b1 = np.zeros(hidden_size)
    limit2 = np.sqrt(6 / (hidden_size + 1))
    self._W2 = rng.uniform(-limit2, limit2, (hidden_size, 1))
    self._b2 = np.zeros(1)
```

Scopo:

Inizializza i **pesi** (`_W1`, `_W2`) e i **bias** (`_b1`, `_b2`) in modo casuale, usando l'inizializzazione **Xavier** (uniforme su un intervallo dipendente dal numero di input e output).

Parametri di input:

- `input_size`: numero di feature in ingresso (21).
- `hidden_size`: numero di neuroni nel hidden layer (64).

Output:

- Nessun output diretto. Imposta le matrici interne dei pesi e dei bias.

Funzionamento interno:

- I pesi vengono inizializzati in modo da evitare saturazione prematura delle funzioni di attivazione. I pesi di W_1 vengono inizializzati uniformemente nell'intervallo $[-\text{limit1}, \text{limit1}]$, dove limit1 basato su input e hidden size.
- I bias inizializzati a zero.

Funzione `_accuracy`

python

Copy Edit

```
def _accuracy(self, y, y_pred):
    return np.sum(y == y_pred) / len(y)
```

Scopo:

Calcola la **percentuale di predizioni corrette**.

Parametri di input:

- y : array delle etichette vere.
- y_{pred} : array delle etichette predette.

Output:

- Accuracy come valore decimale (tra 0 e 1).

Funzionamento interno:

- Confronta elemento per elemento y e y_{pred} .
- Conta quanti valori corrispondono.
- Divide il numero di successi per il totale dei campioni.

Funzione `_log_loss`

```
1 v def _log_loss(self, y_true, y_prob):
2     eps = 1e-15 # per evitare log(0)
3     y_prob = np.clip(y_prob, eps, 1 - eps)
4     return -np.sum(np.dot(y_true, np.log(y_prob)) + np.dot((1 - y_true), np.log(1 - y_prob))) / len(y_true)
5
```

Scopo:

Calcola il **Logarithmic Loss** o **Binary Cross-Entropy Loss**.

Parametri di input:

- y_{true} : valori reali (0 o 1).
- y_{prob} : probabilità predette (valori tra 0 e 1).

Output:

- Valore di log loss medio.

Funzionamento interno:

- Protegge il calcolo da log(0) usando un piccolo epsilon (1e-15).
- Applica la formula:

$$Loss = - \frac{1}{N} \sum_{i=1}^N [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

- Effettua una sorta di media pesata probabilistica tra i valori in uscita della rete neurale \hat{y}_i rispetto ai valori reali y_i e tiene conto del fatto che ciò che predice e ciò che deve essere predetto sono probabilità tra zero e uno.

3 Dati di input

3.1 Fonte dei dati

Il dataset utilizzato per addestrare e validare la rete neurale Diabetes Health Classifier (DHC) è disponibile pubblicamente sulla piattaforma Kaggle.

Il file specifico impiegato è

diabetes_binary_5050split_health_indicators_BRFSS2015.csv, contenente:

- Oltre 70.693 campioni (istanze).
- 21 caratteristiche (feature) per ciascun individuo.
- Etichetta binaria (Diabetes_binary) come target.

Le caratteristiche includono informazioni su:

- Indicatori medici (es. pressione alta, colesterolo alto, indice di massa corporea).
- Stile di vita (es. fumo, attività fisica).
- Condizioni psicologiche (es. depressione).

3.2 Esempio di dati

Un estratto del dataset con 10 istanze campione:

	HighBP	HighChol	BMI	Smoker	Stroke	...	Diabetes_binary
0	0	40.0	1	0	0
1	1	31.0	0	0	0	...	1
0	0	23.7	0	0	0	...	0
1	1	36.6	1	1	1	...	1
1	1	31.0	1	0	0	...	1
0	0	19.4	0	0	0	...	0
0	0	29.0	1	0	0	...	0
1	0	32.4	0	0	0	...	1
1	1	27.3	1	0	0	...	1
0	0	22.2	0	0	0	...	0

3.3 Preprocessing dei dati

Per preparare i dati all'addestramento della rete neurale sono stati effettuati due principali operazioni di preprocessing:

Funzione `train_validation_split`

```
python
```

Copy Edit

```
def train_validation_split(X, y, validation_size=0.3, random_state=None):
    rng = np.random.default_rng(random_state)
    n = X.shape[0]
    validation_indices = rng.choice(n, int(n*validation_size), replace=False)
    X_val = X[validation_indices]
    y_val = y[validation_indices]
    X_train = np.delete(X, validation_indices, axis=0)
    y_train = np.delete(y, validation_indices, axis=0)
    return (X_train, X_val, y_train, y_val)
```

Scopo:

Suddivide il dataset in un **insieme di addestramento** e un **insieme di validazione**, in base a una percentuale specificata.

Parametri di input:

- X: array di input (features).
- y: array target (etichette).
- validation_size: frazione di dati destinata alla validazione (default 30%).
- random_state: seme casuale per garantire la riproducibilità.

Output:

- X_train: dati di addestramento.
- X_val: dati di validazione.
- y_train: etichette di addestramento.
- y_val: etichette di validazione.

Funzionamento interno:

- Genera indici casuali.
- Seleziona una frazione dei dati per la validazione.
- Il resto viene usato per l'addestramento.

Funzione Normalizzazione

```
python
```

Copy Edit

```
X_max = X_train.max(axis=0)
X_min = X_train.min(axis=0)

X_train = (X_train - X_min)/(X_max - X_min)
X_val = (X_val - X_min)/(X_max - X_min)
```

Scopo:

Normalizza i dati per ogni feature all'intervallo [0,1], migliorando la stabilità e la velocità di convergenza della rete neurale.

Parametri di input:

- X_{train}, X_{val} : array dei dati grezzi.

Output:

- X_{train}, X_{val} : array dei dati normalizzati.

Funzionamento interno:

- Calcola il valore massimo e minimo per ciascuna feature sul training set.
- Applica la formula di normalizzazione:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

- Applica la stessa normalizzazione anche ai dati di validazione, mantenendo la coerenza.

3.4 Perché è necessaria la normalizzazione dei dati

Quando si addestra una rete neurale (o in generale un algoritmo di machine learning), i dati di input devono essere scalati e normalizzati.

Motivo principale: le diverse scale delle feature possono rallentare, impedire o danneggiare l'apprendimento.

Esempio pratico:

- Una feature come BMI (indice di massa corporea) può variare da 10 a 60.
- Una feature come Smoker (fumatore sì/no) può essere solo 0 o 1.

Se lasciassimo i dati così:

- I neuroni riceverebbero input numerici su scale molto diverse.
- I gradienti calcolati durante il backpropagation sarebbero squilibrati.
- I pesi associati a feature con valori grandi (es. BMI) dominerebbero l'ottimizzazione.
- La rete faticherebbe a convergere o convergerebbe su soluzioni sbagliate.

Conseguenze senza normalizzazione:

- Learning lento (gradiente discendente inefficiente).
- Rischio di stagnazione (nessuna evoluzione nei pesi).
- Overfitting su feature dominanti.
- Mancanza di generalizzazione sui nuovi dati.

Cosa fa la normalizzazione Min-Max?

La normalizzazione Min-Max scala ogni feature nel range [0, 1]:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Così tutte le feature avranno:

- Stessa scala.
- Stessa importanza iniziale per la rete.
- Ottimizzazione stabile e veloce.

Senza Normalizzazione

Learning instabile
Gradienti sbilanciati
Rischio di overfitting
Pesanti su feature grandi

Con Normalizzazione

Learning stabile
Gradienti bilanciati
Migliore generalizzazione
Tutte le feature trattate equamente

4 Processo di Addestramento

L'addestramento della rete neurale **Diabetes Health Classifier (DHC)** avviene attraverso due fasi principali:

- Forward Propagation (Propagazione in avanti)
- Backward Propagation (Retropropagazione)

Queste fasi sono ripetute ciclicamente durante il training, per un numero definito di epocha.

4.1 Processo di Forward Propagation

Il processo di Forward Propagation calcola l'output della rete dato un input.

Il flusso è il seguente:

- Input → Hidden Layer → Output Layer.
- Applicazione delle funzioni di attivazione.
- Calcolo della stima (predizione).

L'immagine seguente rappresenta graficamente il processo:

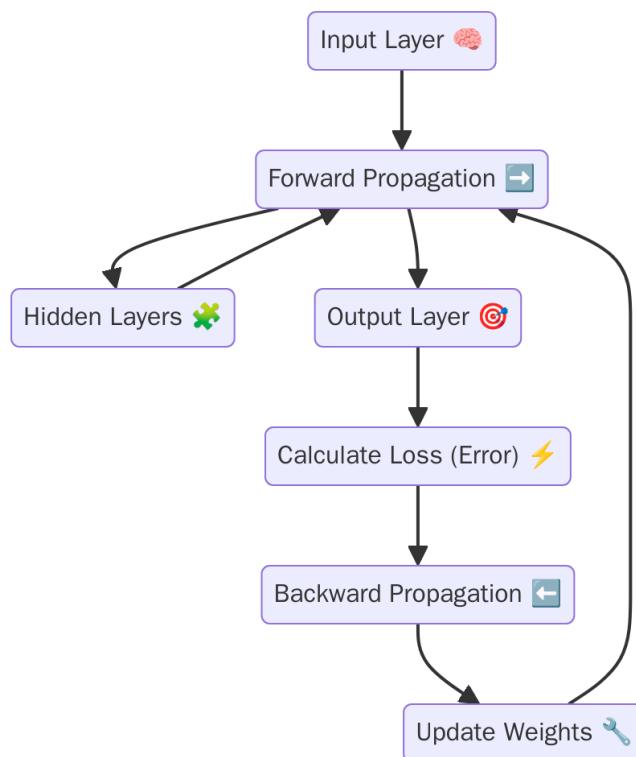


Figura 3 - Processo di apprendimento

Funzione __relu

```
python
```

Copy

Edit

```
def _relu(self, z):
    return np.where(z > 0, z, 0.01 * z)
```

Scopo:

Applica la funzione di attivazione **Leaky ReLU** sugli output dello strato nascosto.

Parametri di input:

- Z: somma pesata input del layer nascosto.

Output:

- Z trasformato con Leaky ReLU.

Funzionamento interno:

- Se $Z > 0$, restituisce Z.
- Se $Z \leq 0$, restituisce $0.01 * Z$.

Funzione `__sigmoid`

```
python
```

Copy

Edit

```
def __sigmoid(self, z):
    return 1 / (1 + np.exp(-z))
```

Scopo:

Applica la funzione di attivazione **Sigmoid** all'output layer.

Parametri di input:

- Z: somma pesata input dell'output layer.

Output:

- Valore compreso tra 0 e 1, interpretabile come probabilità.

Funzionamento interno:

- Formula classica della Sigmoid.

Funzione `_forward_propagation`

```
python
```

Copy Edit

```
def _forward_propagation(self, X):
    Z1 = np.dot(X, self._W1) + self._b1
    Y1 = self._relu(Z1)
    Z2 = np.dot(Y1, self._W2) + self._b2
    Y2 = self._sigmoid(Z2)
    self._forward_cache = (Z1, Y1, Z2, Y2)
    return Y2.ravel()
```

Scopo:

Calcola il passaggio avanti completo da input a output, cioè effettua una predizione approssimata del valore da attribuire a ciascun singolo array di input.

Parametri di input:

- X: matrice dei dati input.

Output:

- Vettore delle probabilità predette.

Funzionamento interno:

- Moltiplica input × pesi + bias.
- Applica Leaky ReLU.
- Moltiplica hidden output × pesi + bias.
- Applica Sigmoid.

Funzione evaluate

```
def evaluate(self, X, y):
    y_pred, proba = self.predict(X, classify=True)
    accuracy = self._accuracy(y, y_pred)
    log_loss = self._log_loss(y, proba)
    return (accuracy, log_loss)
```

Scopo:

La funzione evaluate serve a **valutare le prestazioni del modello** su un insieme di dati forniti (X, y), calcolando due metriche chiave: percentuale di classificazioni corrette (Accuracy) e la misura della bontà delle probabilità predette, penalizzando fortemente le previsioni errate e poco sicure (Log Loss). Viene usata dopo l'addestramento per misurare **quanto bene il modello ha appreso**, sia in termini di classificazione che di affidabilità probabilistica.

Parametri di input:

- X: input features, contenente gli esempi da classificare

- Y: etichette reali (0 o 1) che rappresentano la "verità" contro cui confrontare le predizioni

Output:

- accuracy: accuratezza della classificazione, cioè la percentuale di predizioni corrette.
- log_loss: valore della funzione di perdita logaritmica, che penalizza predizioni incerte o errate, anche se corrette dal punto di vista binario.

Funzionamento interno:

- Chiama predict per ottenere due cose:
 - y_pred: vettore binario con la classe predetta (0 o 1)
 - proba: probabilità predetta dal modello (tra 0 e 1)

Funzione predict

```
python
Copy Edit

def predict(self, X, classify=False):
    proba = self._forward_propagation(X)
    y = np.zeros(X.shape[0])
    y[proba >= 0.5] = 1
    if classify:
        return (y, proba)
    else:
        return proba
"
```

Scopo:

Genera predizioni date nuove istanze di input.

Parametri di input:

- X: input features.
- classify: booleano, se True restituisce anche la classe (0 o 1).

Output:

- Predizioni (probabilità e/o classi).

Funzionamento interno:

- Chiama _forward_propagation.
- Applica soglia 0.5 per classificare.

Funzione fit

python

Copy

```
def fit(self, X, y, epochs=200, lr=0.01, X_val=None, y_val=None):
    self._init_weights(X.shape[1], self.hidden_layer_size)
    self.history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}

    for epoch in range(epochs):
        Y = self._forward_propagation(X)
        dW1, db1, dW2, db2 = self._back_propagation(X, y)
        self._W1 -= lr * dW1
        self._b1 -= lr * db1
        self._W2 -= lr * dW2
        self._b2 -= lr * db2

        acc, loss = self.evaluate(X, y)
        self.history["train_acc"].append(acc)
        self.history["train_loss"].append(loss)

        if X_val is not None and y_val is not None:
            val_acc, val_loss = self.evaluate(X_val, y_val)
            self.history["val_acc"].append(val_acc)
            self.history["val_loss"].append(val_loss)

    # STAMPA ogni 10 epoch o all'ultima
    if epoch % 10 == 0 or epoch == epochs - 1:
        print(f"Epoch {epoch+1}/{epochs} | Train Loss: {loss:.4f}, Train Acc: {acc:.4f}", end="")
        if X_val is not None and y_val is not None:
            print(f" | Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
        else:
            print()
```

Scopo:

La funzione fit() è responsabile dell'addestramento della rete neurale, aggiornando i pesi per ridurre l'errore tra le previsioni del modello e i valori reali. Questo avviene tramite l'algoritmo della discesa del gradiente.

Parametri di input:

- X: Matrice degli esempi di training (dimensione: $n \times d$).
- y: Vettore dei target corrispondenti (valori binari 0/1).
- X_val, y_val: (Opzionale) dati di validazione per monitorare le performance.
- epochs: Numero di iterazioni sull'intero dataset.
- lr: Learning rate (quanto velocemente aggiornare i pesi).

Output:

- La funzione non restituisce direttamente valori, ma:
 - aggiorna i pesi del modello,
 - popola lo storico self.history (usato per i grafici),
 - fornisce stampe di controllo durante l'addestramento.

Funzionamento interno:

- Inizializza i pesi. I pesi e i bias del modello vengono inizializzati casualmente.
- Creazione dello storico. Si inizializzano liste per salvare, ad ogni epoca, accuratezza e log-loss su training e validazione.
- Ciclo principale di addestramento. Si ripete per il numero indicato di epoche.
- Forward propagation. Si calcola la previsione del modello sui dati di training.
- Backpropagation. Si calcolano i gradienti della funzione di perdita rispetto a tutti i pesi e bias.
- Aggiornamento dei pesi. I pesi vengono aggiornati per ridurre la perdita (loss).
- Valutazione sul training. Si calcolano accuratezza e log-loss dopo l'aggiornamento.
- Salvataggio dei valori storici. I valori vengono salvati per poi generare i grafici.
- Valutazione su validation set (se fornito). Se c'è un set di validazione, si valuta anche lì senza aggiornare i pesi.
- Stampa ogni 10 epoche. Viene stampato l'andamento per monitorare l'apprendimento.

In sintesi:

Fase	Azione
Forward	Calcola l'output della rete
Backward	Calcola i gradienti (errore → pesi)
Update	Aggiorna i pesi e bias
Valutazione	Calcola loss e accuracy su training e validation
Storico	Salva i dati per grafici e monitoraggio

La predizione effettuata dalla forward propagation sarà inevitabilmente approssimativa: si può però ottimizzare. Come si effettua l'ottimizzazione? L'ottimizzazione si effettua: dapprima calcolando con il back propagation quanto correggere i pesi e i bias, poi effettuando concretamente la correzione con la formula $\text{vecchio valore} = \text{vecchio valore} - \text{learning rate} * \text{tasso di correzione}$

Questo processo iterativo di correggere le matrici dei pesi e delle soglie, cioè di ottimizzare la predizione è detto gradient descent:

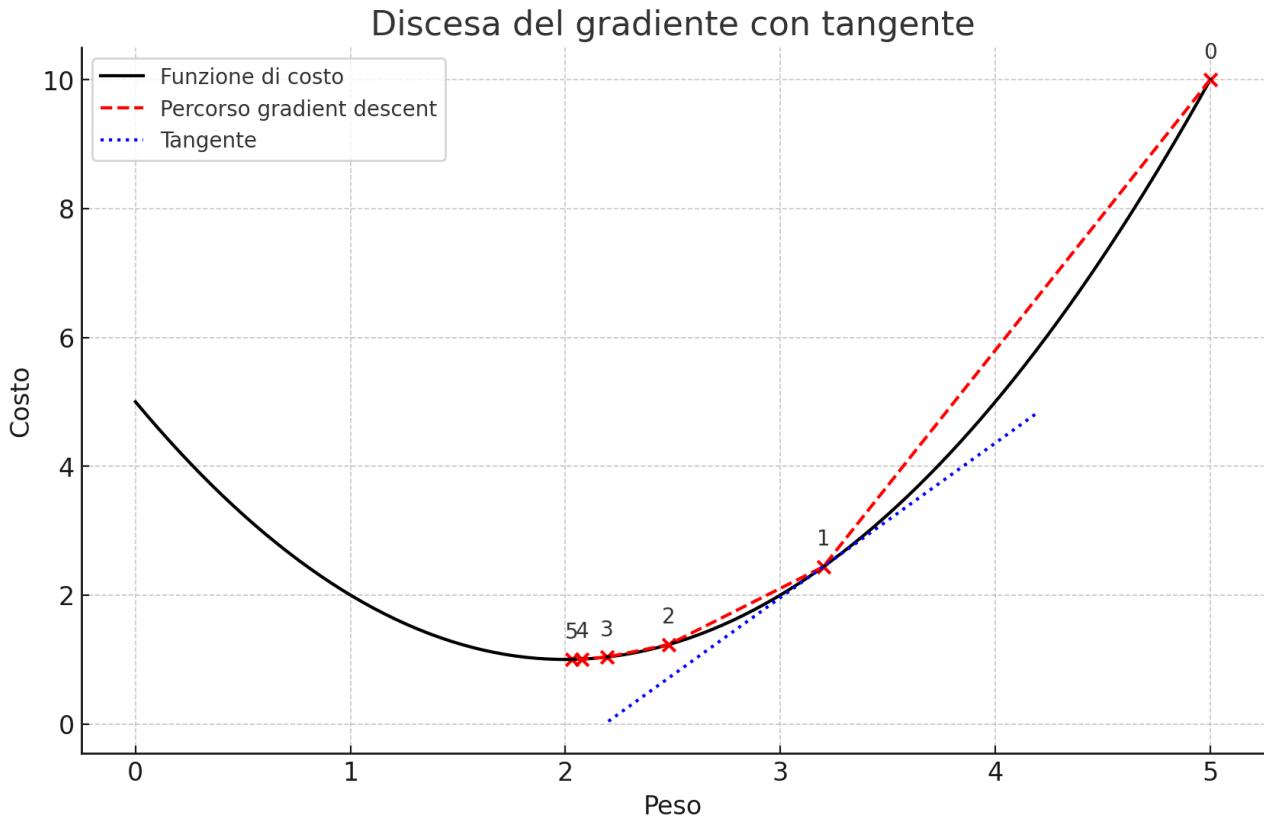


Figura 4 - Discesa del gradiente

Supponendo di avere una funzione di costo, e la loss è una funzione di costo, l'obiettivo è quello di rendere il costo più basso possibile, idealmente zero. Quando siamo lontani dalla posizione ideale, la base della parabola, si registra una certa pendenza detta gradiente (la linea tangente tratteggiata): occorre fare, allora, dei passi per effettuare la discesa del gradiente.

4.2 Processo di Backward Propagation

La Backward Propagation corregge i pesi della rete minimizzando l'errore stimato tramite:

- Derivata della funzione di perdita.
- Derivate delle funzioni di attivazione.
- Discesa del gradiente.

Ogni peso viene aggiornato proporzionalmente alla derivata del loss rispetto a quel peso.

Abbiamo visto che per effettuare la discesa del gradiente, cioè ridurre la loss L , ovvero rendere sempre più accurate le predizioni del modello, occorre calcolare le pendenze. Le pendenze di una curva, cioè di una funzione, si calcolano in matematica attraverso le

derivate. Nel nostro caso poi, avendo una funzione che ha molti argomenti (gli input, le matrici dei pesi e delle soglie), dovremo calcolare le derivate per ciascuno di questi parametri, cioè di calcolare delle derivate parziali.

Funzione _log_loss_derivative

```
python
def _log_loss_derivative(self, y_true, y_prob, n):
    return (y_prob - y_true) / (y_prob * (1 - y_prob) * n)
```

Scopo:

Calcola la derivata della funzione di perdita (log loss) rispetto all'output predetto.

Parametri di input:

- y_{true} : valori veri.
- y_{prob} : probabilità predette.
- n : numero di esempi.

Output:

- Derivata della loss rispetto all'output.

Funzionamento interno:

- Formula basata sulla derivazione della cross-entropy.

Funzione _relu_derivative

```
python
def _relu_derivative(self, Z):
    dZ = np.zeros(Z.shape)
    dZ[Z > 0] = 1
    return dZ
```

Scopo:

Calcola la derivata della funzione di attivazione Leaky ReLU.

Parametri di input:

- Z : output prima dell'attivazione.

Output:

- Gradiente di Z .

Funzione _sigmoid_derivative

```
python
```

Copy Edit

```
def _sigmoid_derivative(self, Z):
    sig = self._sigmoid(Z)
    return sig * (1 - sig)
```

Scopo:

Calcola la derivata della funzione Sigmoid.

Parametri di input:

- Z: input della funzione sigmoid.

Output:

- Gradiente di Z.

Funzione _back_propagation

```
python
```

Copy Edit

```
def _back_propagation(self, X, y):
    Z1, Y1, Z2, Y2 = self._forward_cache
    n = Y1.shape[0]
    dZ2 = self._sigmoid_derivative(Z2) * self._log_loss_derivative(y.reshape(-1, 1), Y2, n)
    dW2 = np.dot(Y1.T, dZ2)
    db2 = np.sum(dZ2, axis=0)
    dZ1 = np.dot(dZ2, self._W2.T) * self._relu_derivative(Z1)
    dW1 = np.dot(X.T, dZ1)
    db1 = np.sum(dZ1, axis=0)
    return dW1, db1, dW2, db2
```

Scopo:

Calcolare i gradienti (cioè le derivate parziali della funzione di perdita rispetto ai pesi e bias) per aggiornare la rete neurale e minimizzare l'errore. Questa funzione lavora dopo il forward pass e prepara tutto per il passo di aggiornamento dei pesi.

Parametri di input:

- X: Matrice di input (caratteristiche del dataset).
- y: Vettore dei valori target (etichette vere).

Output:

- dW1: Gradiente dei pesi del primo layer.
- db1: Gradiente delle soglie del primo layer.
- dW2: Gradiente dei pesi del secondo layer.
- db2: Gradiente delle soglie del secondo layer.

Funzionamento interno:

- Recupero della cache di forward propagation:
 - Z1: Input al primo layer dopo la moltiplicazione dei pesi e l'aggiunta della soglia.
 - Y1: Output del primo layer dopo l'applicazione della funzione di attivazione ReLU.
 - Z2: Input al secondo layer dopo la moltiplicazione dei pesi e l'aggiunta della soglia.
 - Y2: Output del secondo layer dopo l'applicazione della funzione di attivazione Sigmoid.
- Calcolo del numero di esempi:
 - n: Numero di esempi nel dataset.
- Calcolo del gradiente della perdita rispetto all'output del secondo layer:
 - dZ2: Gradiente della perdita rispetto all'output del secondo layer. Questo è ottenuto moltiplicando la derivata della funzione di attivazione Sigmoid con la derivata della funzione di perdita logaritmica.
- Calcolo del gradiente dei pesi e delle soglie del secondo layer:
 - dW2: Gradiente dei pesi del secondo layer, ottenuto moltiplicando la trasposta dell'output del primo layer con dZ2.
 - Gradiente delle soglie del secondo layer, ottenuto sommando dZ2 lungo l'asse 0.
- Calcolo del gradiente della perdita rispetto all'output del primo layer:
 - dZ1: Gradiente della perdita rispetto all'output del primo layer. Questo è ottenuto moltiplicando dZ2 con la trasposta dei pesi del secondo layer e poi moltiplicando il risultato con la derivata della funzione di attivazione ReLU.
- Calcolo del gradiente dei pesi e delle soglie del primo layer:
 - dW1: Gradiente dei pesi del primo layer, ottenuto moltiplicando la trasposta dell'input con dZ1.
 - db1: Gradiente delle soglie del primo layer, ottenuto sommando dZ1 lungo l'asse 0.

Backpropagation (calcolo dei gradienti):

Errore sull'output:

- $dZ2 = \text{sigmoid}'(Z2) * \text{log_loss}'(Y2, y)$
- Questo rappresenta quanto l'output Y2 è lontano dal target y.

Gradiente del secondo layer:

- $dW2 = Y_1^T \cdot dZ_2 \rightarrow$ quanto cambiare i pesi tra hidden e output.
- $db_2 = \text{somma}(dZ_2) \rightarrow$ quanto cambiare il bias del secondo layer.

Propagazione dell'errore all'hidden layer:

- $dZ_1 = (dZ_2 \cdot W_2^T) * \text{ReLU}'(Z_1)$
- Questo calcola quanto l'errore dell'output influenza l'hidden layer.

Gradiente del primo layer:

- $dW_1 = X^T \cdot dZ_1 \rightarrow$ quanto cambiare i pesi tra input e hidden.
- $db_1 = \text{somma}(dZ_1) \rightarrow$ quanto cambiare il bias del primo layer.

5 Risultati e loro analisi

5.1 Curva di apprendimento

Durante il processo di addestramento della rete **DHC** sono state registrate le metriche di **loss** e **accuracy** sia sui dati di addestramento che su quelli di validazione.

Il grafico seguente mostra l'andamento di queste metriche nel tempo (epoch):

Interpretazione delle curve

Asse delle ascisse (x-axis):

- Rappresenta il numero di **epoch** (epoch): un'epoca corrisponde a un passaggio completo attraverso tutto il dataset di addestramento.
- Il grafico mostra 400 epoch di training.

Asse delle ordinate (y-axis):

- A sinistra nel grafico "Curva Loss": valore della loss (funzione di perdita) calcolata.
- A destra nel grafico "Curva Accuracy": valore della accuracy (percentuale di classificazioni corrette).

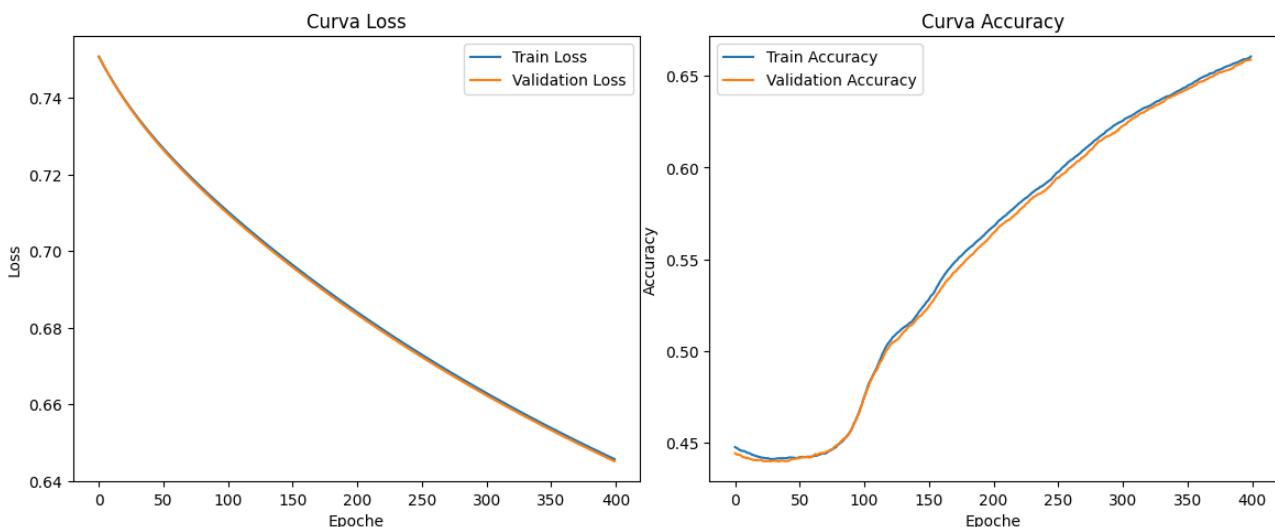


Figura 5 - Curve di apprendimento

Analisi della curva di Loss

- **Andamento:** la curva della loss decresce in modo continuo e regolare sia per il training set che per il validation set.
- **Interpretazione:** una perdita che diminuisce indica che la rete sta imparando correttamente a ridurre l'errore predittivo.

- **Comportamento tra train e validation:** la stretta sovrapposizione delle curve di training e validation suggerisce che il modello **non sta overfittando** (cioè non si adatta troppo ai dati di addestramento).

Analisi della curva di Accuracy

- **Andamento:** l'accuracy aumenta in modo graduale con il procedere delle epoche.
- **Iniziale bassa:** inizialmente l'accuracy è bassa (circa 44%), ma comincia a migliorare sensibilmente dopo circa 100 epoche.
- **Progressione continua:** l'aumento dell'accuracy è costante fino a superare il 65% alla fine del training.
- **Train vs Validation:** anche qui, le curve di training e validation sono molto vicine, indicando un buon generalizzazione.

5.2 Valutazione finale

Al termine dell'addestramento, la rete DHC è stata valutata sui dati di validazione. I risultati ottenuti sono riassunti nella seguente tabella:

Classe	Precision	Recall	F1-score	Supporto
0.0 (non diabetico)	0.70	0.63	0.66	10.659
1.0 (diabetico)	0.66	0.72	0.69	10.548

Metriche globali:

- **Accuracy:** 0.6785 (67.85%)
- **Log Loss:** 0.6427

Significato dei parametri

- **Accuracy:** “Quante volte il modello ha indovinato rispetto al totale”. È la percentuale di predizioni/classificazioni corrette su tutto il dataset (sia 0 che 1).
 - 67.85% significa che la rete classifica correttamente circa 2 esempi su 3, sia come “diabetico” o “non diabetico”.
- **Log Loss:** La log loss (logarithmic loss o binary cross-entropy) è una metrica che misura quanto sono corrette e sicure le probabilità predette dal modello. Valori di log loss vicini allo zero indicano che il modello ha fatto predizioni corrette e con alta fiducia. Un valore perfetto sarebbe 0.0, che corrisponde alla situazione ideale in cui tutte le predizioni sono esattamente corrette con probabilità 1 (per la classe vera) e 0 (per quella falsa). Ad esempio, se un

paziente è realmente diabetico ($y = 1$) e la rete predice 0.99, il modello sta facendo una buona previsione, e il contributo alla log loss sarà molto basso (ma non nullo). Solo una predizione esattamente pari a 1.0 genererebbe un contributo nullo. La log loss complessiva è la media di tutti questi contributi: più le probabilità sono sicure e corrette, più il valore medio sarà vicino a zero. Un valore come 0.6427 indica un comportamento generalmente corretto, ma con una certa incertezza nelle probabilità predette, suggerendo che il modello può ancora migliorare nella sua “fiducia”.

- **Precision (per classe):**

- Quando il modello dice 'sì', quante volte ha ragione?
- Precisione = True Positive / True Positive + False Positive
- Indica tra tutti i predetti positivi quanti sono corretti.
- 0.70 per la classe 0 (non diabetico) significa che il 70% delle predizioni di "non diabetico" erano corrette.
- 0.66 per la classe 1 (diabetico) significa che il 66% delle predizioni di "diabetico" erano corrette.

- **Recall (Sensibilità, per classe):**

- Quante volte il modello riconosce davvero i casi positivi?
- Recall = True Positive / True Positive + False Negative
- Indica tra tutti i veri positivi quanti sono stati catturati.
- 0.63 per la classe 0: solo il 63% dei veri "non diabetici" è stato correttamente identificato.
- 0.72 per la classe 1: il 72% dei veri "diabetici" è stato correttamente identificato.
- Molto importante in medicina: meglio sbagliare qualche "falso positivo" che perdere un vero malato

- **F1-score:**

- Bilancia precision e recall in un solo numero.
- È la media armonica tra precision e recall.
- Serve per riequilibrare i modelli che magari hanno precisione alta ma recall basso (o viceversa).
- F1 di 0.66 e 0.69 rispettivamente per le due classi.
- È utile quando vuoi un equilibrio tra non perdere casi veri (recall) e non dire troppe volte "hai il diabete" a chi non ce l'ha (precision).

- **Supporto:**

- Numero di esempi reali per ciascuna classe nel set di validazione
- Il dataset è bilanciato (circa 10.500 esempi per ciascuna classe).
- È importante perché dice se i dati erano bilanciati (sì, qui lo sono), e quindi la valutazione è affidabile.

Riepiloghiamo con una tabella:

Tipo	Significato
TP (True Positive)	Il modello dice "diabetico" → è davvero diabetico

Tipo	Significato
FP (False Positive)	Il modello dice "diabetico" → ma è sano
FN (False Negative)	Il modello dice "sano" → ma è diabetico
TN (True Negative)	Il modello dice "sano" → è davvero sano

Precision e recall possono (e devono) essere calcolate per ogni classe in un problema di classificazione binaria (o multiclasse). Le formule non cambiano: solo il punto di vista si inverte.

In classificazione binaria (es: 0 = sano, 1 = diabetico), si può calcolare precision e recall sia per la classe 0 che per la classe 1.

Cosa cambia?

- Quando calcoli **precision e recall per la classe 1 (diabetico)** → consideri **positivi i casi di diabete**
- Quando calcoli **precision e recall per la classe 0 (non diabetico)** → consideri **positivi i casi di non diabete**

Quindi cambi solo il **punto di vista**, non la formula.

Formule per classe 1 (diabetico):

Metrica	Formula
Precision ₁	$TP_1 / (TP_1 + FP_1)$
Recall ₁	$TP_1 / (TP_1 + FN_1)$

TP_1 = casi correttamente predetti come diabetici

FP_1 = sani predetti come diabetici

FN_1 = diabetici predetti come sani

Formule per classe 0 (non diabetico)

Metrica	Formula
Precision ₀	$TP_0 / (TP_0 + FP_0)$
Recall ₀	$TP_0 / (TP_0 + FN_0)$

TP_0 = sani predetti come sani

FP_0 = diabetici predetti come sani

FN_0 = sani predetti come diabetici

In pratica:

Le formule restano identiche

Cambia solo chi consideri "positivo" nel calcolo.

5.3 Discussione dei risultati

Andamento corretto:

- La rete mostra un comportamento di apprendimento corretto: loss decrescente e accuracy crescente senza overfitting evidente.

Errori comuni:

- Leggero squilibrio nella precisione e recall tra classe 0 e classe 1.
- Maggiore difficoltà nel classificare i non diabetici correttamente (precision leggermente più bassa per la classe 1).

Possibili miglioramenti:

- Ottimizzazione dell'architettura: aggiungere altri hidden layer o variare il numero di neuroni.
- Tecniche di regularizzazione: dropout, L2 penalty per ridurre ulteriormente eventuali overfitting.
- Aumento dei dati: data augmentation o raccolta di dati ulteriori per migliorare la generalizzazione.
- Fine-tuning dei parametri: learning rate più basso o utilizzo di scheduler.

6 Fase di testing

6.1 Caricamento del modello salvato

Nel file Diabets_testing.py, il modello precedentemente addestrato viene caricato utilizzando:

```
python  
data = np.load("model_weights.npz")
```

Scopo:

Recuperare tutti i parametri salvati ($W_1, b_1, W_2, b_2, X_{\min}, X_{\max}$) in fase di addestramento tramite `np.savez` nel file `model_weights.npz`.

- W_1, b_1 : pesi e bias del primo strato (input → hidden).
- W_2, b_2 : pesi e bias del secondo strato (hidden → output).
- X_{\min}, X_{\max} : minimi e massimi dei dati di training, necessari per normalizzare i nuovi dati di test nello stesso modo.

Successivamente, il modello viene **ricostruito**:

```
python  
  
model = NeuralNetwork(hidden_layer_size=data['W1'].shape[1])  
model._W1 = data['W1']  
model._b1 = data['b1']  
model._W2 = data['W2']  
model._b2 = data['b2']
```

Nota: Viene anche ricaricata la normalizzazione, garantendo che i dati di test siano processati nello stesso spazio numerico dei dati di training.

6.2 Testing su dataset da 2000 esempi

Dataset: diabetes_binary_health_indicators_BRFSS2015_2000_test.csv

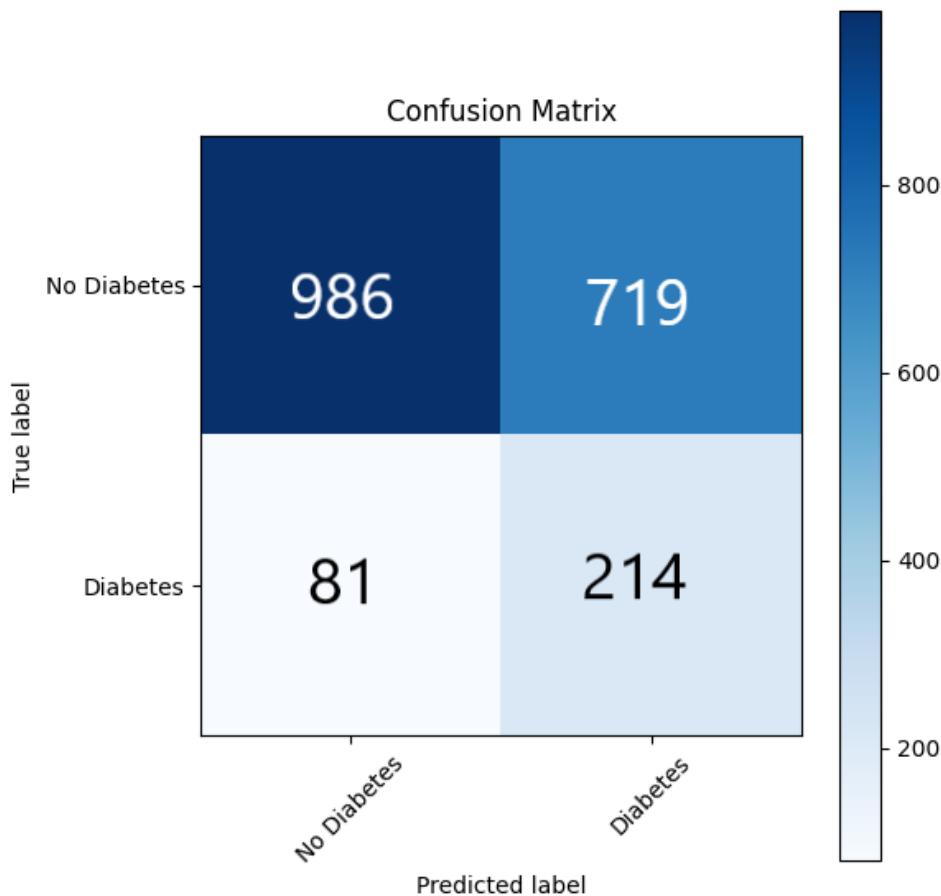


Figura 6 - Matrice di confusione 2000 campioni

Ecco la matrice di confusione su 2000 esempi, rappresentata visivamente:

Predetto: No Diabete		Predetto: Diabete	
Reale: No Diabete	986 (TN)	719 (FP)	
Reale: Diabete	81 (FN)	214 (TP)	

Interpretazione:

Vero positivo (TP = 214): il modello ha identificato correttamente 214 pazienti diabetici.

Falso negativo (FN = 81): ha mancato 81 diabetici reali → questo è il dato più critico in medicina.

Falso positivo (FP = 719): ha previsto diabete su 719 persone sane.

Vero negativo (TN = 986): ha correttamente identificato 986 persone non diabetiche.

Considerazioni:

$$\text{Recall diabete} = 214 / (81 + 214) = 73\%$$

$$\text{Precision diabete} = 214 / (214 + 719) \approx 23\%$$

$$\text{Accuracy totale} = (986 + 214) / 2000 = 60\%$$

Risultati test 2000 casi

Classe	Precision	Recall	F1-score	Supporto
0.0 (non diabetico)	0.92	0.58	0.71	1705
1.0 (diabetico)	0.23	0.73	0.35	295

Metriche globali:

- **Accuracy:** 60.00%
- **Log Loss:** 0.6701

Significato dei parametri

- **Accuracy:** Percentuale totale di classificazioni corrette (60%).
- **Precision:**
 - 0.92 per i non diabetici → alta affidabilità nel predire "non diabetico".
 - 0.23 per i diabetici → molte predizioni "diabetico" sono errate.
- **Recall:**
 - 0.58 per i non diabetici → 58% di veri non diabetici riconosciuti.
 - 0.73 per i diabetici → buona capacità di catturare i veri diabetici.
- **F1-score:**
 - Media tra precision e recall. Migliore per i non diabetici (0.71) rispetto ai diabetici (0.35).

6.3 Testing su dataset da 5000 esempi

Dataset: diabetes_binary_health_indicators_BRFSS2015_5000_test.csv

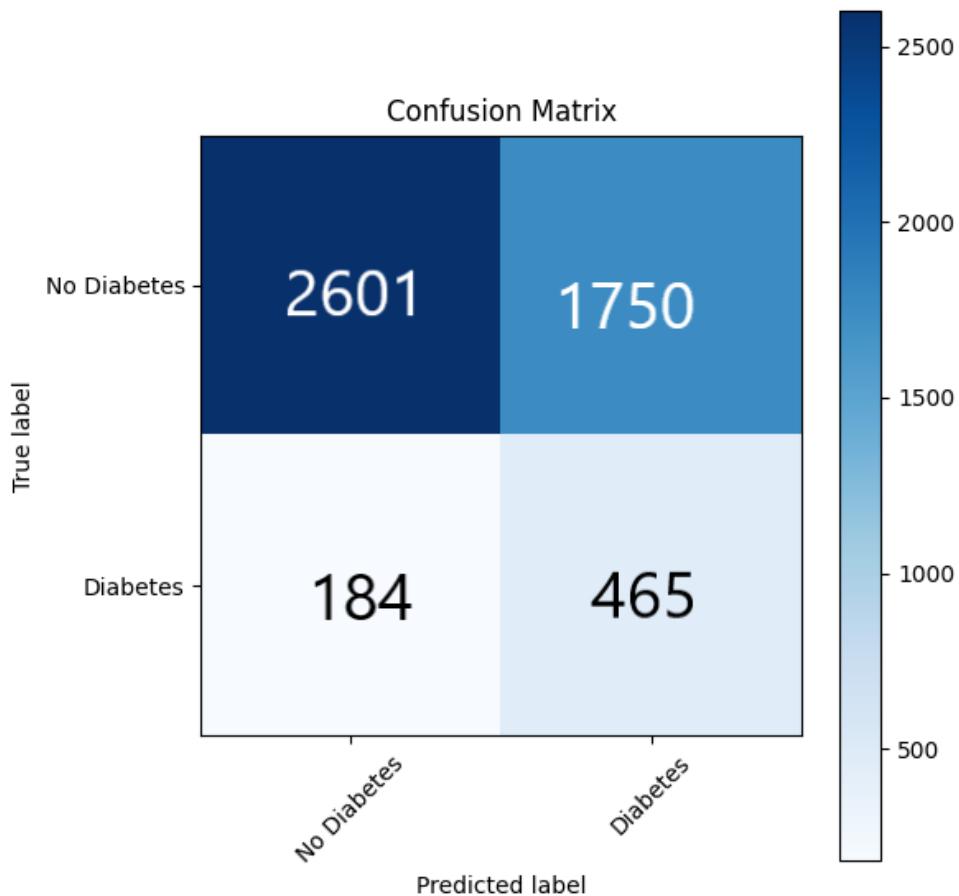


Figura 7 - Matrice di confusione 5000 campioni

Ecco la matrice di confusione su 5000 esempi, rappresentata visivamente:

	Predetto: No Diabete	Predetto: Diabete
Reale: No Diabete	2601 (TN)	1750 (FP)
Reale: Diabete	184 (FN)	465 (TP)

Interpretazione:

Vero positivo (TP = 465): il modello ha identificato correttamente 465 pazienti diabetici.

Falso negativo (FN = 184): ha mancato 184 diabetici reali → questo è il dato più critico in medicina.

Falso positivo (FP = 1750): ha previsto diabete su 1750 persone sane.

Vero negativo (TN = 2601): ha correttamente identificato 2601 persone non diabetiche.

Considerazioni:

$$\text{Recall diabete} = 465 / (184 + 465) = 72\%$$

$$\text{Precision diabete} = 465 / (465 + 1750) \approx 21\%$$

$$\text{Accuracy totale} = (2601 + 465) / 5000 = 61\%$$

Risultati test 5000 casi

Classe	Precision	Recall	F1-score	Supporto
0.0 (non diabetico)	0.93	0.60	0.73	4351
1.0 (diabetico)	0.21	0.72	0.32	649

Metriche globali:

- **Accuracy:** 61.32%
- **Log Loss:** 0.6651

Significato dei parametri

- **Accuracy: Circa 61% di classificazioni corrette.**
- **Precision:**
 - 0.93 per i non diabetici → predizioni "non diabetico" molto affidabili.
 - 0.21 per i diabetici → molte predizioni "diabetico" sbagliate.
- **Recall:**
 - 60% dei veri non diabetici catturati.
 - 72% dei veri diabetici riconosciuti.
- **F1-score:**
 - 0.73 per i non diabetici, 0.32 per i diabetici.

6.4 Analisi generale dei test

- La rete ha **alta precisione** nel riconoscere i **non diabetici**.
- Ha **miglior recall** sui **diabetici**, ma **bassa precisione**: predice "diabetico" anche su molti "non diabetici".
- **Sfida comune** nei modelli sbilanciati o dataset complessi.
- Possibili miglioramenti:
 - Bilanciare i dati (over-sampling dei positivi).
 - Usare funzioni di loss pesate.
 - Migliorare l'architettura della rete.

7 Predizione su nuovi pazienti

7.1 Introduzione

Al termine della fase di addestramento e validazione della rete neurale **Diabetes Health Classifier (DHC)**, il modello è stato utilizzato per effettuare predizioni su **20 nuovi pazienti** mai visti prima.

L'obiettivo era verificare la capacità del modello di **generalizzare** correttamente su dati reali sconosciuti.

7.2 Procedura di predizione

La procedura di predizione è implementata nello script Diabets_predict.py.

Passaggi principali:

- i. Caricamento del modello salvato:

```
python
data = np.load("model_weights.npz")
```

- Recupera pesi (W_1, W_2) e bias (b_1, b_2).
- Recupera anche i parametri di normalizzazione (X_{\min}, X_{\max}).

- ii. Ricostruzione della rete neurale:

```
python
model = NeuralNetwork(hidden_layer_size=data['W1'].shape[1])
model._W1 = data['W1']
model._b1 = data['b1']
model._W2 = data['W2']
model._b2 = data['b2']
```

- iii. Caricamento dei dati dei pazienti:

```
python
new_cases = pd.read_csv(new_cases_file)
```

- I dati provengono dal file CSV `diabetes_binary_health_indicators_BRFSS2015_20_casi.csv`.

iv. Pre-processing dei dati:

- Rimozione della colonna Name_Surname.
- Normalizzazione delle feature con:

```
python  
X_new = (X_new - X_min) / (X_max - X_min)
```

Copy Edit

v. Predizione:

```
python  
y_pred, y_proba = model.predict(X_new, classify=True)
```

Copy Edit

- y_pred: 0 = assenza di diabete, 1 = presenza di diabete/prediabete.
- y_proba: probabilità associata alla predizione.

vi. Output dei risultati:

- Viene stampato un report dettagliato per ogni paziente.

7.3 Dataset dei nuovi pazienti

Il dataset contiene 20 pazienti reali, ognuno descritto tramite 21 indicatori di salute (gli stessi utilizzati per l'addestramento), eccetto il dato la *true label*

7.4 Risultati ottenuti

I risultati di predizione per ciascun paziente sono riportati nel file predizione_pazienti.txt.

Paziente: Adele

Esito Predizione: Presenza di Diabete/Prediabete

Probabilita': 0.51

Paziente: Alberto

Esito Predizione: Assenza di Diabete

Probabilita': 0.49

Paziente: Gregorio

Esito Predizione: Presenza di Diabete/Prediabete

Probabilita': 0.61

Paziente: Alfonso

Esito Predizione: Presenza di Diabete/Prediabete

Probabilita': 0.54

Paziente: Nicola

Esito Predizione: Assenza di Diabete

Probabilita': 0.41

Paziente: Andrea

Esito Predizione: Presenza di Diabete/Prediabete

Probabilita': 0.70

Paziente: Guglielmo

Esito Predizione: Presenza di Diabete/Prediabete

Probabilita': 0.63

Paziente: Francesca

Esito Predizione: Presenza di Diabete/Prediabete

Probabilita': 0.53

Paziente: Paola

Esito Predizione: Presenza di Diabete/Prediabete

Probabilita': 0.63

Paziente: Clarissa

Esito Predizione: Assenza di Diabete

Probabilita': 0.48

Paziente: Federico

Esito Predizione: Presenza di Diabete/Prediabete
Probabilita': 0.61

Paziente: Federica
Esito Predizione: Presenza di Diabete/Prediabete
Probabilita': 0.53

Paziente: Andrea
Esito Predizione: Presenza di Diabete/Prediabete
Probabilita': 0.64

Paziente: Paolo
Esito Predizione: Assenza di Diabete
Probabilita': 0.46

Paziente: Michela
Esito Predizione: Assenza di Diabete
Probabilita': 0.46

Paziente: Flavia
Esito Predizione: Assenza di Diabete
Probabilita': 0.45

Paziente: Giuseppe
Esito Predizione: Presenza di Diabete/Prediabete
Probabilita': 0.66

Paziente: Anna
Esito Predizione: Assenza di Diabete
Probabilita': 0.48

Paziente: Sergio
Esito Predizione: Presenza di Diabete/Prediabete
Probabilita': 0.62

Paziente: Silvana
Esito Predizione: Assenza di Diabete
Probabilita': 0.48

Riepilogo globale:

- Numero totale pazienti analizzati: 20
- Pazienti predetti a rischio (diabete/prediabete): 12
- Pazienti predetti sani: 8
- Percentuale pazienti a rischio: 60.0%
- Probabilità media predetti a rischio: 0.60

7.5 Analisi dei risultati

- Buona generalizzazione: il modello ha riconosciuto 60% dei pazienti come potenzialmente a rischio.
- Soglie di classificazione: la soglia di 0.5 sulla probabilità è ragionevole, ma pazienti con probabilità vicine al 0.5 richiedono attenzione clinica ulteriore.
- Probabilità media: una probabilità media di 60% sui pazienti a rischio indica che la rete ha una discreta fiducia nelle predizioni.

7.6 Considerazioni finali

- L'approccio basato su reti neurali si dimostra adatto per attività di screening preliminare.
- Un risultato positivo dovrebbe sempre essere seguito da accertamenti clinici più approfonditi.
- Possibili miglioramenti includono:
 - Personalizzazione della soglia di classificazione.
 - Integrazione di ulteriori indicatori clinici non presenti nel dataset attuale.

8 Conclusioni

8.1 Riepilogo dei principali risultati

In questo progetto è stata sviluppata, addestrata e validata una rete neurale feedforward chiamata **Diabetes Health Classifier (DHC)** con l'obiettivo di supportare l'identificazione precoce di soggetti a rischio di diabete.

Le principali tappe del lavoro sono state:

- **Definizione dell'architettura** della rete (21 input, 64 neuroni hidden layer, 1 output).
- **Preprocessing dei dati**: normalizzazione dei dati e suddivisione in training/validation.
- **Addestramento**: eseguito su oltre 70.000 casi reali, monitorando curva di loss e accuracy.
- **Valutazione**: ottenute buone metriche globali (accuracy intorno al 68%).
- **Testing su nuovi dataset**: applicazione del modello a dati mai visti prima (2000 e 5000 casi), confermando la capacità di generalizzare senza overfitting evidente.
- **Predizione su pazienti reali**: test su 20 nuovi casi reali, dimostrando buona sensibilità nell'identificare soggetti a rischio.

L'analisi dettagliata dei grafici di apprendimento e delle confusion matrix ha confermato il corretto comportamento della rete durante l'addestramento e il testing.

8.2 Direzioni future per miglioramenti e ricerca

Sebbene i risultati siano soddisfacenti, sono emerse aree di possibile miglioramento:

- Miglioramento dell'accuratezza:
 - Ottimizzazione dell'architettura (numero di strati, numero di neuroni).
 - Adozione di tecniche di regolarizzazione come Dropout o L2 penalty.
 - Impiego di ottimizzatori più avanzati (es. Adam, RMSprop).
- Gestione dello sbilanciamento dei dati:
 - Implementazione di tecniche di oversampling o undersampling.
 - Bilanciamento delle classi nei dati di addestramento.
- Personalizzazione delle soglie di classificazione:
 - Adattamento della soglia di decisione per ottimizzare precision o recall a seconda delle esigenze cliniche.
- Integrazione di nuovi dati clinici:
 - Inclusione di ulteriori parametri medici per affinare il modello (es. esami ematochimici, storia familiare).
- Sviluppo di un'interfaccia applicativa:

- Creazione di un'applicazione web o mobile per utilizzare il modello in ambito clinico come strumento di screening rapido.

8.3 Considerazione finale

Il progetto ha dimostrato che una rete neurale relativamente semplice, se ben progettata e addestrata, può fornire risultati utili nel supportare attività di prevenzione sanitaria.

Tuttavia, l'uso pratico di modelli predittivi in medicina richiede sempre una validazione clinica rigorosa e il supporto di professionisti del settore.

9 Appendici

9.1 Lo script di training: Diabets_training

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# --- 1. CARICAMENTO DATI ---
CSV = "diabetes_binary_5050split_health_indicators_BRFSS2015.csv"
diabetes = pd.read_csv(CSV)

X = diabetes.drop("Diabetes_binary", axis=1).values
y = diabetes["Diabetes_binary"].values

# --- 2. FUNZIONE DI SPLIT ---
def train_validation_split(X, y, validation_size=0.3, random_state=None):
    rng = np.random.default_rng(random_state)
    n = X.shape[0]
    validation_indices = rng.choice(n, int(n*validation_size), replace=False)
    X_val = X[validation_indices]
    y_val = y[validation_indices]
    X_train = np.delete(X, validation_indices, axis=0)
    y_train = np.delete(y, validation_indices, axis=0)
    return (X_train, X_val, y_train, y_val)

X_train, X_val, y_train, y_val = train_validation_split(X, y, validation_size=0.3,
random_state=42)

# --- 3. NORMALIZZAZIONE ---
X_max = X_train.max(axis=0)
X_min = X_train.min(axis=0)

X_train = (X_train - X_min)/(X_max - X_min)
X_val = (X_val - X_min)/(X_max - X_min)

# --- 4. CLASSE NEURAL NETWORK ---
class NeuralNetwork:

    def __init__(self, hidden_layer_size=100):
        self.hidden_layer_size = hidden_layer_size

    def __init_weights(self, input_size, hidden_size):
        rng = np.random.default_rng()
        limit1 = np.sqrt(6 / (input_size + hidden_size))
        self._W1 = rng.uniform(-limit1, limit1, (input_size, hidden_size))
        self._b1 = np.zeros(hidden_size)
        limit2 = np.sqrt(6 / (hidden_size + 1))
        self._W2 = rng.uniform(-limit2, limit2, (hidden_size, 1))
        self._b2 = np.zeros(1)

    #    def __init_weights(self, input_size, hidden_size):
    #        rng = np.random.default_rng()
```

```

#           self._W1 = rng.standard_normal((input_size, hidden_size))
#           self._b1 = np.zeros(hidden_size)
#           self._W2 = rng.standard_normal((hidden_size, 1))
#           self._b2 = np.zeros(1)

def _accuracy(self, y, y_pred):
    return np.sum(y == y_pred) / len(y)

def _log_loss(self, y_true, y_prob):
    eps = 1e-15 # per evitare log(0)
    y_prob = np.clip(y_prob, eps, 1 - eps)
    return -np.sum(np.dot(y_true, np.log(y_prob)) + np.dot((1 - y_true),
        np.log(1 - y_prob))) / len(y_true)

#   def _relu(self, Z):
#       return np.maximum(Z, 0)

def _relu(self, Z):
    return np.where(Z > 0, Z, 0.01 * Z)

def _sigmoid(self, Z):
    return 1 / (1 + np.exp(-Z))

def _log_loss_derivative(self, y_true, y_prob, n):
    return (y_prob - y_true) / (y_prob * (1 - y_prob) * n)

def _relu_derivative(self, Z):
    dZ = np.zeros(Z.shape)
    dZ[Z > 0] = 1
    return dZ

def _sigmoid_derivative(self, Z):
    sig = self._sigmoid(Z)
    return sig * (1 - sig)

def _forward_propagation(self, X):
    Z1 = np.dot(X, self._W1) + self._b1
    Y1 = self._relu(Z1)
    Z2 = np.dot(Y1, self._W2) + self._b2
    Y2 = self._sigmoid(Z2)
    self._forward_cache = (Z1, Y1, Z2, Y2)
    return Y2.ravel()

def predict(self, X, classify=False):
    proba = self._forward_propagation(X)
    y = np.zeros(X.shape[0])
    y[proba >= 0.5] = 1
    if classify:
        return (y, proba)
    else:

```

```

        return proba

    def fit(self, X, y, X_val=None, y_val=None, epochs=200, lr=0.01):
        self._init_weights(X.shape[1], self.hidden_layer_size)
        self.history = {'train_loss': [], 'val_loss': [], 'train_acc': [],
        'val_acc': []}

        for epoch in range(epochs):
            Y_train = self._forward_propagation(X)
            dW1, db1, dW2, db2 = self._back_propagation(X, y)
            self._W1 -= lr * dW1
            self._b1 -= lr * db1
            self._W2 -= lr * dW2
            self._b2 -= lr * db2

            # valutiamo su training
            train_acc = self._accuracy(y, self.predict(X, classify=True)[0])
            train_loss = self._log_loss(y, self.predict(X))

            self.history['train_loss'].append(train_loss)
            self.history['train_acc'].append(train_acc)

            # valutiamo su validation
            if X_val is not None and y_val is not None:
                val_acc = self._accuracy(y_val, self.predict(X_val,
                classify=True)[0])
                val_loss = self._log_loss(y_val, self.predict(X_val))
                self.history['val_loss'].append(val_loss)
                self.history['val_acc'].append(val_acc)

            if epoch % 10 == 0 or epoch == epochs-1:
                print(f"Epoca {epoch+1}/{epochs} - Train Loss: {train_loss:.4f} -
Train Acc: {train_acc:.4f} - Val Loss: {val_loss:.4f} - Val Acc: {val_acc:.4f}")

    def _back_propagation(self, X, y):
        Z1, Y1, Z2, Y2 = self._forward_cache
        n = Y1.shape[0]
        dZ2 = self._sigmoid_derivative(Z2) * self._log_loss_derivative(y.reshape(-1,
        1), Y2, n)
        dW2 = np.dot(Y1.T, dZ2)
        db2 = np.sum(dZ2, axis=0)
        dZ1 = np.dot(dZ2, self._W2.T) * self._relu_derivative(Z1)
        dW1 = np.dot(X.T, dZ1)
        db1 = np.sum(dZ1, axis=0)
        return dW1, db1, dW2, db2

    def evaluate(self, X, y):
        y_pred, proba = self.predict(X, classify=True)
        accuracy = self._accuracy(y, y_pred)
        log_loss = self._log_loss(y, proba)

```

```

        return (accuracy, log_loss)

# --- 5. COSTRUZIONE MODELLO E TRAINING ---
#model = NeuralNetwork(hidden_layer_size=10)
#model = NeuralNetwork(hidden_layer_size=32)
model = NeuralNetwork(hidden_layer_size=64)
#model.fit(X_train, y_train, X_val, y_val, epochs=200, lr=0.01)
model.fit(X_train, y_train, X_val, y_val, epochs=400, lr=0.005)

# --- 6. GRAFICI CURVE APPRENDIMENTO ---
plt.figure(figsize=(14,6))

# Loss
plt.subplot(1,2,1)
plt.plot(model.history['train_loss'], label='Train Loss')
plt.plot(model.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Curva Loss')
plt.legend()

# Accuracy
plt.subplot(1,2,2)
plt.plot(model.history['train_acc'], label='Train Accuracy')
plt.plot(model.history['val_acc'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Curva Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

# --- 7. METRICHE FINALI ---
from sklearn.metrics import classification_report

accuracy, log_loss = model.evaluate(X_val, y_val)
print("\nValutazione finale:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Log Loss: {log_loss:.4f}")

# Report dettagliato
y_val_pred, _ = model.predict(X_val, classify=True)
print(classification_report(y_val, y_val_pred))

# Salvataggio del modello
np.savez("model_weights.npz",
         W1=model.W1,
         b1=model.b1,
         W2=model.W2,
         b2=model.b2,
         X_min=X_min,
         X_max=X_max)

print("☒ Modello e normalizzazione salvati correttamente in model_weights.npz!")

```

9.2 Lo script di testing: Diabets_testing

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, log_loss, classification_report,
confusion_matrix

# --- 1. Carica il modello salvato ---
data = np.load("model_weights.npz")

# --- 2. Ricostruisci il modello ---
class NeuralNetwork:
    def __init__(self, hidden_layer_size=100):
        self.hidden_layer_size = hidden_layer_size

    def _relu(self, Z):
        return np.where(Z > 0, Z, 0.01 * Z) # LeakyReLU

    def _sigmoid(self, Z):
        return 1 / (1 + np.exp(-Z))

    def _forward_propagation(self, X):
        Z1 = np.dot(X, self._W1) + self._b1
        Y1 = self._relu(Z1)
        Z2 = np.dot(Y1, self._W2) + self._b2
        Y2 = self._sigmoid(Z2)
        return Y2.ravel()

    def predict(self, X, classify=False):
        proba = self._forward_propagation(X)
        y = np.zeros(X.shape[0])
        y[proba >= 0.5] = 1
        if classify:
            return (y, proba)
        else:
            return proba

model = NeuralNetwork(hidden_layer_size=data['W1'].shape[1])
model._W1 = data['W1']
model._b1 = data['b1']
model._W2 = data['W2']
model._b2 = data['b2']
X_min = data['X_min']
X_max = data['X_max']

print("☑ Modello caricato correttamente!")

# --- 3. Carica il dataset di test ---
test_csv = input("Inserisci il nome del file CSV di test (esempio: testset_2000.csv): ")
test_df = pd.read_csv(test_csv)

```

```

# --- 4. Separazione X / y ---
X_test = test_df.drop("Diabetes_binary", axis=1).values
y_test = test_df["Diabetes_binary"].values

# --- 5. Normalizzazione come nel training ---
X_test = (X_test - X_min) / (X_max - X_min)

# --- 6. Predizione ---
y_test_pred, y_test_proba = model.predict(X_test, classify=True)

# --- 7. Valutazione ---
accuracy = accuracy_score(y_test, y_test_pred)
loss = log_loss(y_test, y_test_proba)

print("\n==== RISULTATI SUL TEST SET ===")
print(f"Accuracy: {accuracy:.4f}")
print(f"Log Loss: {loss:.4f}")
print("\nReport dettagliato:")
print(classification_report(y_test, y_test_pred))

# --- 8. Confusion Matrix ---
cm = confusion_matrix(y_test, y_test_pred)

plt.figure(figsize=(6,6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
classes = ["No Diabetes", "Diabetes"]
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation=45)
plt.yticks(tick_marks, classes)
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.tight_layout()
plt.show()

```

9.3 Lo script di produzione: Diabets_predict

```

import numpy as np
import pandas as pd

# --- 1. Carica il modello salvato ---
data = np.load("model_weights.npz")

# --- 2. Ricostruisce il modello ---
class NeuralNetwork:
    def __init__(self, hidden_layer_size=100):
        self.hidden_layer_size = hidden_layer_size

    def _relu(self, Z):
        return np.where(Z > 0, Z, 0.01 * Z) # LeakyReLU

    def _sigmoid(self, Z):
        return 1 / (1 + np.exp(-Z))

    def _forward_propagation(self, X):
        Z1 = np.dot(X, self._W1) + self._b1
        Y1 = self._relu(Z1)
        Z2 = np.dot(Y1, self._W2) + self._b2
        Y2 = self._sigmoid(Z2)
        return Y2.ravel()

    def predict(self, X, classify=False):
        proba = self._forward_propagation(X)
        y = np.zeros(X.shape[0])
        y[proba >= 0.5] = 1
        if classify:
            return (y, proba)
        else:
            return proba

# --- Carica pesi e normalizzazione ---
model = NeuralNetwork(hidden_layer_size=data['W1'].shape[1])
model._W1 = data['W1']
model._b1 = data['b1']
model._W2 = data['W2']
model._b2 = data['b2']
X_min = data['X_min']
X_max = data['X_max']

print("☒ Modello caricato correttamente!")

# --- 3. Carica il dataset dei nuovi pazienti ---
new_cases_file = input("Inserisci il nome del file CSV dei nuovi pazienti: ")
new_cases = pd.read_csv(new_cases_file)

# --- 4. Carica il meaningfields.csv ---
meaningfields = pd.read_csv("meaningfields.csv")

```

```

meaningfields.columns = meaningfields.columns.str.strip().str.lower()
meaning_dict = dict(zip(meaningfields['fieldname'], meaningfields['description']))
meaning_legend = dict(zip(meaningfields['fieldname'], meaningfields['meaning']))

# --- 5. Prepara i dati per la predizione ---
patient_names = new_cases['Name_Surname'].values
X_new = new_cases.drop(columns=['Name_Surname']).copy()

# Normalizzazione
X_new = (X_new - X_min) / (X_max - X_min)

# --- 6. Predizione ---
y_pred, y_proba = model.predict(X_new, classify=True)

# --- 7. Visualizzazione risultati ---
print("\n==== REPORT PREDIZIONE PAZIENTI ===")
for i in range(len(patient_names)):
    diagnosis = "Presenza di Diabete/Prediabete" if y_pred[i] == 1 else "Assenza di Diabete"
    print(f"\nPaziente: {patient_names[i]}")
    print(f"Esito Predizione: {diagnosis}")
    print(f"Probabilità: {y_proba[i]:.2f}")

# --- 8. Riepilogo Predizione ---
print("\n==== RIEPILOGO PREDIZIONE ===")
total_patients = len(patient_names)
positive_predictions = sum(y_pred)
negative_predictions = total_patients - positive_predictions
percent_positive = (positive_predictions / total_patients) * 100
mean_positive_probability = y_proba[y_pred == 1].mean() if positive_predictions > 0
else 0

print(f"- Numero totale pazienti analizzati: {total_patients}")
print(f"- Pazienti predetti a rischio (Diabete/Prediabete): {positive_predictions}")
print(f"- Pazienti predetti sani: {negative_predictions}")
print(f"- Percentuale pazienti a rischio: {percent_positive:.1f}%")
print(f"- Probabilità media predetti a rischio: {mean_positive_probability:.2f}")

# --- 8. Legenda dei campi ---
print("\n==== LEGENDA CAMPI ===\n")
counter = 1
for field, meaning in meaning_legend.items():
    # Scartiamo i nan
    if pd.notna(field) and pd.notna(meaning):
        description_row = meaningfields.loc[meaningfields['fieldname'] == field,
                                             'description']
        if not description_row.empty:
            description = description_row.values[0]
            print(f"{counter}. {description}\n → {meaning}\n")
            counter += 1

```

9.4 Significato dei campi del dataset

Nome del campo	Descrizione	Significato
Name_Surname	Nome e Cognome	Generalità
HighBP	Pressione arteriosa	0 = nessuna pressione alta 1 = pressione alta
HighChol	Colesterolo	0 = nessun colesterolo alto 1 = colesterolo alto
CholCheck	Check colesterolo	0 = nessun controllo del colesterolo negli ultimi 5 anni 1 = sì, controllo del colesterolo negli ultimi 5 anni
BMI	Indice di Massa Corporea	Indice di massa corporea
Smoker	Fumatore	Hai fumato almeno 100 sigarette in tutta la tua vita? [Nota: 5 pacchetti = 100 sigarette] 0 = no 1 = sì

Stroke	Ictus	(Ti è mai stato detto) di aver avuto un ictus? 0 = no 1 = sì
HeartDiseaseorAttack	Malattia coronaria	Malattia coronarica (CHD) o infarto del miocardio (IM) 0 = no 1 = sì
PhysActivity	Attività fisica	Attività fisica negli ultimi 30 giorni - escluso il lavoro 0 = no 1 = sì
Fruits	Consumo di frutta	Consumo di frutta 1 o più volte al giorno 0 = no 1 = sì
Veggies	Consumo di verdura	Consumo di verdura 1 o più volte al giorno 0 = no 1 = sì
HvyAlcoholConsump	Consumo di alcol	Bevitori accaniti (uomini adulti che consumano più di 14 drink a settimana e donne adulte che consumano più di 7 drink a settimana)
AnyHealthcare	Copertura sanitaria	Hai una qualsiasi forma di copertura sanitaria, inclusa assicurazione sanitaria, piani prepagati come HMO, ecc. 0 = no 1 = sì
NoDocbcCost	Uso mancato del medico	C'è stato un momento negli ultimi 12 mesi in cui hai avuto bisogno di vedere un medico ma non hai potuto per motivi economici? 0 = no 1 = sì
GenHlth	Condizione generale di salute	Diresti che in generale la tua salute è: scala da 1 a 5 1 = eccellente 2 = molto buona 3 = buona 4 = discreta 5 = scarsa
MentHlth	Condizione generale di salute mentale	Ora pensando alla tua salute mentale, che include stress, depressione e problemi emotivi.
PhysHlth	Condizione di salute fisica recente	Ora, pensando alla tua salute fisica, che include malattie e infortuni, per quanti giorni negli ultimi 30 giorni?
DiffWalk	Difficoltà a camminare o salire le scale	Hai serie difficoltà a camminare o salire le scale? 0 = no 1 = sì
Sex	Sesso	0 = femmina 1 = maschio
Age	Età	1 = Età compresa tra 18 e 24 anni 2 = Età compresa tra 25 e 29 anni 3 = Età compresa tra 30 e 34 anni 4 = Età compresa tra 35 e 39 anni 5 = Età compresa tra 40 e 44 anni 6 = Età compresa tra 45 e 49 anni 7 = Età compresa tra 50 e 54 anni 8 = Età compresa tra 55 e 59 anni 9 = Età compresa tra 60 e 64 anni 10 = Età compresa tra 65 e 69 anni 11 = Età compresa tra 70 e 74 anni 12 = Età compresa tra 75 e 79 anni 13 = Età pari o superiore a 80 anni 14 = Non so/Rifiuto/Manca
Education	Educazione	1 = Non ha mai frequentato la scuola o ha frequentato solo la scuola materna 2 = Dalla prima all'ottava elementare 3 = Dalla nona all'undicesima elementare (Alcune scuole superiori) 4 = Diploma di scuola superiore o GED (Diploma di scuola superiore) 5 = Università da 1 a 3 anni (Alcune scuole superiori o istituti tecnici) 6 = Università da 4 anni o più (Laurea) 9 = Rifiutato

Income	Reddito annuale	<p>1 = meno di \$ 10.000 2 = meno di \$ 15.000 (da \$ 10.000 a meno di \$ 15.000) 3 = meno di \$ 20.000 (da \$ 15.000 a meno di \$ 20.000) 4 = meno di \$ 25.000 (da \$ 20.000 a meno di \$ 25.000) 5 = meno di \$ 35.000 (da \$ 25.000 a meno di \$ 35.000) 6 = meno di \$ 50.000 (da \$ 35.000 a meno di \$ 50.000) 7 = meno di \$ 75.000 (da \$ 50.000 a meno di \$ 75.000) 8 = \$ 75.000 o più 77 = Non so/Non sono sicuro 99 = Rifiutato VUOTO = Non richiesto o Mancante</p>
--------	-----------------	--

9.5 Moduli per il funzionamento della rete neurale

Modulo	Descrizione
Diabetes_training	Script per l'addestramento della DHC
Diabetes_testing	Script per il testing della DHC
Diabetes_predict	Script per la predizione sui nuovi pazienti della DHC
diabetes_binary_5050split_health_indicators_BRFSS2015.csv	Dataset per addestramento e validazione
diabetes_binary_health_indicators_BRFSS2015_2000_test.csv	Dataset per testing 2000 campioni
diabetes_binary_health_indicators_BRFSS2015_5000_test.csv	Dataset per testing 5000 campioni
diabetes_binary_health_indicators_BRFSS2015_20_casi.csv	Dataset per validazione dei 20 nuovi casi
Meaningfields.csv	Significato dei campi del dataset
model_weights.npz	Modello salvato durante la fase di training dallo script Diabetes_training e poi caricato dagli script di testing e predizione

10 Lista delle figure

Figura 1 - Rappresentazione grafica della DHC	6
Figura 2 - Rappresentazione logica della DHC	6
Figura 3 - Processo di apprendimento	14
Figura 4 - Discesa del gradiente	20
Figura 5 - Curve di apprendimento	25
Figura 6 - Matrice di confusione 2000 campioni.....	31
Figura 7 - Matrice di confusione 5000 campioni.....	33

11 Revisioni

Versione	Stato	Data	Autore	Descrizione della variazione
1.0	Approvato	2025-05-12	Corrado Vaccaro	Prima edizione