Reti neurali

Il cervello artificiale (ma molto semplificato)

Le reti neurali sono ispirate al cervello umano, ma attenzione: sono una **semplificazione estrema**. Il cervello ha 86 miliardi di neuroni. Le reti neurali che vedremo ne hanno... qualche decina. Ma funzionano comunque sorprendentemente bene!

Il neurone biologico vs artificiale

Biologico	Artificiale
Dendriti ricevono segnali	Input (i dati)
Sinapsi hanno diversa "forza"	Pesi (weights)
Soma elabora i segnali	Somma pesata + bias
Assone trasmette il risultato	Funzione di attivazione → output

Il neurone artificiale fa una cosa semplice:

1. Prende gli input
2. Li moltiplica per i pesi
3. Somma tutto (+ un bias)
4. Passa il risultato attraverso una funzione di attivazione

Formula: `output = attivazione(w1*x1 + w2*x2 + ... + bias)`

Il Perceptrone

Il **perceptrone** è il neurone artificiale più semplice: un singolo neurone che prende decisioni binarie (sì/no, 0/1). Proviamo a farlo funzionare come una porta logica AND:

```
import numpy as np

# Porta AND: entrambi gli input devono essere 1
dati_and = [
    ([0, 0], 0),
    ([0, 1], 0),
    ([1, 0], 0),
    ([1, 1], 1),
]

# Perceptrone
np.random.seed(42)
pesi = np.random.randn(2) * 0.5
bias = 0.0
lr = 0.1

def step(x):
    """Funzione di attivazione: 1 se x >= 0, altrimenti 0"""
    return 1 if x >= 0 else 0

# Addestramento
print("Addestramento del perceptrone (AND):")
for epoca in range(20):
    errori = 0
    for inputs, target in dati_and:
        x = np.array(inputs)
        # Forward: calcola l'output
        somma = np.dot(x, pesi) + bias
        output = step(somma)

        # Calcola l'errore
        errore = target - output
        if errore != 0:
            errori += 1
            # Aggiorna pesi e bias
            pesi += lr * errore * x
            bias += lr * errore

    if epoca % 5 == 0:
```

```
        print(f"  Epoca {epoca:2}: errori={errori}, pesi={pesi.round(2)}
    if errori == 0:
        print(f"  Epoca {epoca:2}: CONVERGENZA! Zero errori!")
        break

# Test
print(f"
Risultati finali:")
print(f"Pesi: {pesi.round(3)}, Bias: {bias:.3f}
")
for inputs, target in dati_and:
    x = np.array(inputs)
    output = step(np.dot(x, pesi) + bias)
    ok = "OK" if output == target else "ERRORE"
    print(f"  {inputs[0]} AND {inputs[1]} = {output}  (atteso: {target})
```

!!! tip "Prova anche OR"

```
Cambia `dati_and` con i dati della porta OR e vedrai che il perceptrone
```

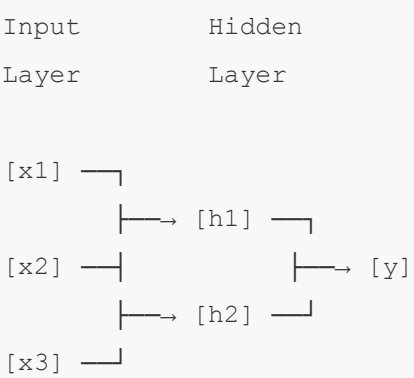
Funzioni di attivazione

La funzione di attivazione decide se il neurone "si attiva" o no. Ce ne sono diverse, ognuna con le sue caratteristiche:

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

x = np.linspace(-5, 5, 100)

# Sigmoid: schiaccia tutto tra 0 e 1
sigmoid = 1 / (1 + np.exp(-x))

# ReLU: semplice ma potente
relu = np.maximum(0, x)

# Tanh: come sigmoid ma tra -1 e 1
tanh = np.tanh(x)

fig, axes = plt.subplots(1, 3, figsize=(14, 4))

axes[0].plot(x, sigmoid, 'b-', linewidth=2)
axes[0].set_title('Sigmoid')
axes[0].grid(True, alpha=0.3)
axes[0].axhline(y=0.5, color='gray', linestyle='--', alpha=0.5)

axes[1].plot(x, relu, 'r-', linewidth=2)
axes[1].set_title('ReLU')
axes[1].grid(True, alpha=0.3)

axes[2].plot(x, tanh, 'g-', linewidth=2)
axes[2].set_title('Tanh')
axes[2].grid(True, alpha=0.3)
axes[2].axhline(y=0, color='gray', linestyle='--', alpha=0.5)

plt.suptitle('Funzioni di attivazione', fontsize=14)
plt.tight_layout()
plt.show()

print("Sigmoid: output tra 0 e 1 (perfetta per probabilita')")
print("ReLU: la piu' usata nelle reti moderne (semplice e veloce)")
print("Tanh: output tra -1 e 1 (a volte meglio della sigmoid)")
```

Una rete neurale: il forward pass

Una rete neurale ha più **strati** (layers) di neuroni:



Il **forward pass** è il calcolo che va dagli input all'output: moltiplica per i pesi, somma, attiva, ripeti per ogni strato.

```
import numpy as np

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

# Rete semplice: 2 input → 3 neuroni nascosti → 1 output
np.random.seed(42)

# Pesi casuali
W1 = np.random.randn(2, 3)  # 2 input → 3 neuroni nascosti
b1 = np.zeros(3)
W2 = np.random.randn(3, 1)  # 3 neuroni nascosti → 1 output
b2 = np.zeros(1)

# Input di esempio
x = np.array([0.5, 0.8])

# Forward pass (passo per passo!)
print("FORWARD PASS")
print(f"Input: {x}")

# Strato nascosto
z1 = np.dot(x, W1) + b1
print(f"
Strato nascosto (prima di attivazione): {z1.round(3)}")
a1 = sigmoid(z1)
print(f"Strato nascosto (dopo sigmoid): {a1.round(3)}")

# Strato output
z2 = np.dot(a1, W2) + b2
print(f"
Strato output (prima di attivazione): {z2.round(3)}")
a2 = sigmoid(z2)
print(f"Strato output (dopo sigmoid): {a2.round(3)}")

print(f"
Output finale: {a2[0]:.4f}")
print("(Con pesi casuali l'output e' casuale. Ora dobbiamo ADDESTRARE!)"
```

Backpropagation (l'idea)

Il forward pass calcola l'output. Ma come si aggiornano i pesi per migliorare? Con la **backpropagation** (retropropagazione dell'errore).

L'idea: tipo accordare una chitarra.

1. Suoni una nota → senti che è stonata (calcoli l'errore)
2. Giri la chiavetta un pochino (aggiusti i pesi)
3. Suoni di nuovo → meno stonata!
4. Ripeti finché non è intonata

In termini tecnici: si calcola quanto ogni peso ha contribuito all'errore, e si aggiusta proporzionalmente. È calcolo differenziale applicato (ma non preoccuparti, lo fa il computer!).

Rete neurale completa: il problema XOR

Ecco il grande momento: una rete neurale addestrata da zero con numpy! Risolviamo XOR, il problema che un singolo perceptrone non poteva risolvere:

```
import numpy as np

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

def sigmoid_derivata(x):
    return x * (1 - x)

# Dataset XOR
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Inizializza pesi casuali
np.random.seed(42)
W1 = np.random.randn(2, 4)    # 2 input → 4 neuroni nascosti
b1 = np.zeros((1, 4))
W2 = np.random.randn(4, 1)    # 4 neuroni nascosti → 1 output
b2 = np.zeros((1, 1))

lr = 1.0
epoche = 5000

print("Addestramento rete neurale su XOR:")
print(f"Struttura: 2 → 4 → 1")
print(f"{'Epoca':>6} {'Loss':>10}")
print("-" * 18)

for epoca in range(epoche):
    # Forward pass
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)

    # Calcola l'errore (loss)
    loss = np.mean((y - a2) ** 2)

    # Backpropagation
    d2 = (a2 - y) * sigmoid_derivata(a2)
```

```
    d1 = np.dot(d2, W2.T) * sigmoid_derivata(a1)

    # Aggiorna pesi
    W2 -= lr * np.dot(a1.T, d2)
    b2 -= lr * np.sum(d2, axis=0, keepdims=True)
    W1 -= lr * np.dot(X.T, d1)
    b1 -= lr * np.sum(d1, axis=0, keepdims=True)

    if epoca % 1000 == 0:
        print(f"{epoca:>6} {loss:>10.6f}")

print(f"{epoche:>6} {loss:>10.6f}")

# Test
print(f"
Risultati finali:")
for i in range(4):
    output = a2[i][0]
    print(f"  {X[i][0]} XOR {X[i][1]} = {output:.4f}  (arrotondato: {rou

print("
Funziona! Il perceptrone singolo non poteva, la rete a 2 strati SI!")
```

MLPClassifier di scikit-learn

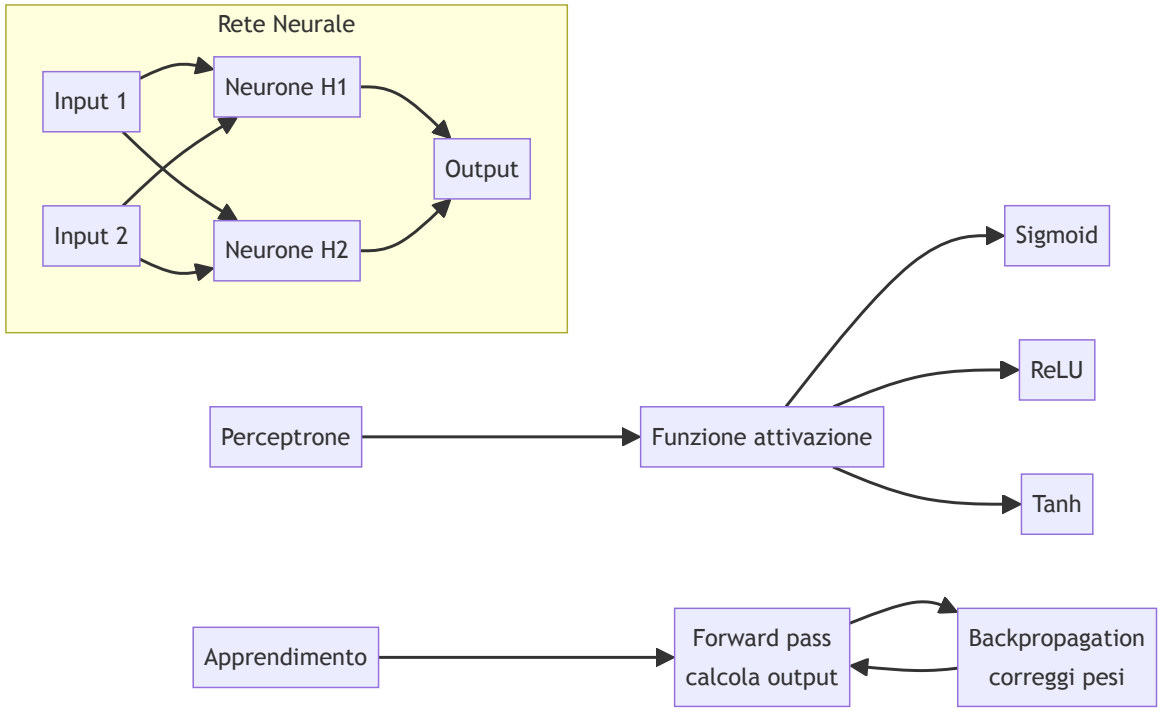
Scrivere la rete a mano è educativo, ma per progetti veri usiamo `MLPClassifier` di sklearn:

```
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

np.random.seed(42)

# Dataset piu' complesso
n = 200
ore = np.concatenate([np.random.normal(7, 1.5, n//2), np.random.normal(3
voti_prec = np.concatenate([np.random.normal(7.5, 1, n//2), np.random.no
etichette = np.array([1] * (n//2) + [0] * (n//2))
X = np.column_stack([ore, voti_prec])
y = etichette

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,

# Rete neurale con sklearn (molto piu' semplice!)
mlp = MLPClassifier(
    hidden_layer_sizes=(10, 5),  # 2 strati nascosti: 10 e 5 neuroni
    max_iter=1000,
    random_state=42
)
mlp.fit(X_train, y_train)

acc = accuracy_score(y_test, mlp.predict(X_test))
print(f"Accuracy: {acc:.2%}")
print(f"Struttura: {X.shape[1]} → 10 → 5 → 1")
print(f"Epoche usate: {mlp.n_iter_}")
```

!!! warning "Perche' non usiamo solo numpy?"

```
Scrivere reti neurali a mano e' perfetto per capire COME funzionano. Ma

- **Auto-differenziazione**: calcolano i gradienti automaticamente
- **GPU**: sfruttano la scheda video per velocizzare tutto
- **Reti pre-addestrate**: puoi partire da modelli gia' addestrati su mi
```

Mappa concettuale



Esercizi

Esercizio 1: Porte logiche

Modifica il perceptrone per imparare la porta OR e la porta NAND. Funziona per tutte?

```
import numpy as np

# Dati OR
dati_or = [([0, 0], 0), ([0, 1], 1), ([1, 0], 1), ([1, 1], 1)]

# Dati NAND
dati_nand = [([0, 0], 1), ([0, 1], 1), ([1, 0], 1), ([1, 1], 0)]

# Addestra un perceptrone per ogni porta
```

??? success "Soluzione"

```pyodide
import numpy as np

def perceptrone(dati, nome, lr=0.1, epoche=100):
    w = np.random.randn(2) * 0.5
    b = 0.0
    for _ in range(epoche):
        for inputs, target in dati:
            x = np.array(inputs)
            output = 1 if np.dot(w, x) + b > 0.5 else 0
            errore = target - output
            w += lr * errore * x
            b += lr * errore
    print(f"
```

Porta {nome}:") for inputs, target in dati: x = np.array(inputs) pred = 1 if np.dot(w, x) + b > 0.5 else 0 ok = "OK" if pred == target else "ERRORE" print(f" {inputs} -> atteso: {target}, predetto: {pred} {ok}")

```
dati_or = [([0, 0], 0), ([0, 1], 1), ([1, 0], 1), ([1, 1], 1)]
dati_nand = [([0, 0], 1), ([0, 1], 1), ([1, 0], 1), ([1, 1], 0)]
perceptrone(dati_or, "OR")
perceptrone(dati_nand, "NAND")
print("
```

Entrambe funzionano! OR e NAND sono linearmente separabili.") ```

Esercizio 2: Rete neurale personalizzata

Modifica la rete neurale numpy per avere 8 neuroni nello strato nascosto invece di 4. Converge piu' velocemente o piu' lentamente?

```
import numpy as np

# Modifica la rete XOR con 8 neuroni nascosti

# Confronta la velocita' di convergenza
```

??? success "Soluzione"

```pyodide
import numpy as np
np.random.seed(42)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
hidden_size = 8
w1 = np.random.randn(2, hidden_size) * 0.5
b1 = np.zeros((1, hidden_size))
w2 = np.random.randn(hidden_size, 1) * 0.5
b2 = np.zeros((1, 1))
lr = 1.0
for epoca in range(5000):
    h = sigmoid(X @ w1 + b1)
    out = sigmoid(h @ w2 + b2)
    errore = y - out
    d_out = errore * out * (1 - out)
    d_h = (d_out @ w2.T) * h * (1 - h)
    w2 += lr * h.T @ d_out
    b2 += lr * d_out.sum(axis=0, keepdims=True)
    w1 += lr * X.T @ d_h
    b1 += lr * d_h.sum(axis=0, keepdims=True)
    if epoca % 1000 == 0:
        mse = (errore**2).mean()
        print(f"Epoca {epoca}: MSE = {mse:.4f}")
print(f"
```

Risultati con {hidden_size} neuroni nascosti:") for i in range(4): print(f" {X[i]} -> {out[i][0]:.3f} (atteso: {y[i][0]})") ```

Esercizio 3: MLPClassifier tuning

Prova diverse configurazioni di MLPClassifier (numero di strati, neuroni, learning rate) e trova quella con l'accuracy migliore usando cross-validation.

```
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score

np.random.seed(42)

# Crea il dataset

# Prova diverse configurazioni

# Stampa i risultati e trova la migliore
```

??? success "Soluzione"

```pyodide
install="scikit-learn,numpy"
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score
np.random.seed(42)
n = 200
X = np.random.rand(n, 4) * 10
y = ((X[:, 0] * X[:, 1]) > 25).astype(int)
configs = [
    ("(10,)", (10,)),
    ("(50,)", (50,)),
    ("(10, 10)", (10, 10)),
    ("(50, 25)", (50, 25)),
    ("(100, 50, 25)", (100, 50, 25)),
]
print("Configurazione     | Media  | Std")
print("-------------------|--------|-----")
migliore_score = 0
migliore_config = ""
for nome, layers in configs:
    mlp = MLPClassifier(hidden_layer_sizes=layers, max_iter=500, random_
    scores = cross_val_score(mlp, X, y, cv=5)
    print(f"{nome:19s}| {scores.mean():.3f}  | {scores.std():.3f}")
    if scores.mean() > migliore_score:
        migliore_score = scores.mean()
        migliore_config = nome
print(f"
```

Migliore: {migliore_config} con score={migliore_score:.3f}") ```