



**Università
degli Studi
di Palermo**

Applicazione della decomposizione in valori singolari alle reti neurali

Giacomo Lombardo

Gianvito Cognata

15 Gennaio 2025

Indice

1	Introduzione	3
2	Cenni teorici	4
2.1	Singular Value Decomposition	4
2.2	Reti neurali	12
2.3	Tipi di Reti Neurali	18
3	Caso di applicazione: dati multidimensionali	22
3.1	Introduzione al dataset e alla rete utilizzata	22
3.2	Addestramento e risultati	25
3.3	Applicazione di SVD al dataset	26
3.4	Addestramento con dati a dimensionalità ridotta	28
4	Caso di applicazione: computer vision	31
4.1	Introduzione al dataset e alla rete utilizzata	31
4.2	Preprocessing del dataset	33
4.3	Addestramento di ResNet18	39
4.4	Valutazione del modello	40
4.5	Applicazione dell'SVD al dataset	43
4.6	Addestramento della ResNet18 con immagini compresse	48
5	Conclusioni e considerazioni	52
6	Fonti	53

1 Introduzione

Negli ultimi anni, l’Intelligenza Artificiale (IA) ha avuto un impatto significativo in molteplici settori, ma il suo rapido sviluppo solleva preoccupazioni legate all’ambiente e alla sostenibilità. I modelli di IA come ChatGPT richiedono ingenti risorse naturali, come acqua ed energia elettrica, per l’addestramento e l’uso. Ad esempio, scrivere un’email di 100 parole con ChatGPT consuma circa 519 ml di acqua e 0,14 kWh di elettricità. Questo impatto ecologico solleva questioni sulla riduzione delle emissioni di CO₂ e l’uso più efficiente delle risorse nei data center. Le aziende, i governi e i consumatori hanno un ruolo cruciale nel promuovere un’IA più sostenibile. Durante l’addestramento di modelli di IA di grandi dimensioni, come GPT-3, si stima che vengano prodotti circa 552 tonnellate di CO₂, con un notevole impatto sull’ambiente. L’addestramento di questi modelli richiede un’enorme potenza computazionale e una spaventosa quantità di dati, che implica un uso intensivo di data center e sistemi di raffreddamento, consumando acqua per mantenere i server operativi. Inoltre, l’energia elettrica necessaria per il funzionamento dei data center grava ulteriormente sull’impronta ecologica complessiva. Questo solleva preoccupazioni sulla sostenibilità a lungo termine dell’IA, spingendo per l’adozione di tecnologie più efficienti e pratiche ecocompatibili. L’obiettivo di questo progetto è dimostrare come, riducendo la dimensionalità dei dati e, di conseguenza, la quantità di risorse per addestrare, si riesca comunque a raggiungere degli ottimi risultati, più che validi in numerosi contesti di applicazione.

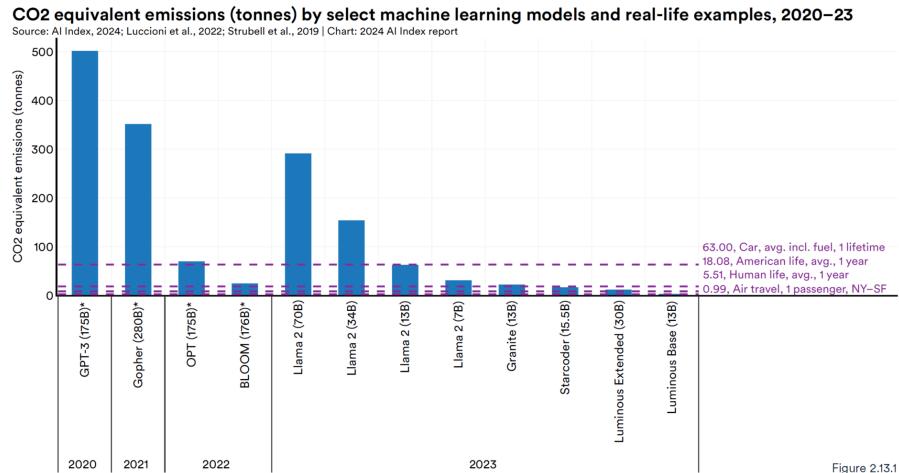


Figure 2.13.1

Figura 1: Emissioni di CO₂

2 Cenni teorici

2.1 Singular Value Decomposition

La tecnica della SVD permette di scomporre una matrice A (sia quadrata che rettangolare) nel prodotto di tre matrici:

$$A = U\Sigma V^T$$

Questo potente strumento matematico fu introdotto inizialmente da Eugenio Beltrami nel 1873 per matrici quadrate non singolari. La sua generalizzazione alle matrici rettangolari è attribuita a Carl Eckart e Gale Young, che ne svilupparono la teoria tra il 1936 e il 1939.

Applicazioni della SVD

La decomposizione ai valori singolari trova applicazione in numerosi campi, tra cui:

- **Compressione delle immagini:** la SVD consente di comprimere un'immagine, mantenendo gran parte delle informazioni rilevanti, risultando uno strumento efficace per la riduzione dello spazio di archiviazione (ad esempio per memorizzare i grandi dataset di addestramento delle reti neurali convoluzionali).
- **Riconoscimento facciale:** la SVD viene utilizzata in algoritmi basati sulle *eigenfaces*, che permettono di identificare e classificare i volti attraverso una rappresentazione compatta dei dati visivi.
- **Analisi delle Componenti Principali (PCA):** la SVD è alla base del calcolo delle componenti principali di un insieme di dati, consente di individuarne le caratteristiche più significative e di esprimere i dati in uno spazio di rappresentazione a dimensionalità ridotta, semplificandone così l'analisi.

Grazie alla sua eleganza matematica e alla sua efficacia computazionale, la tecnica della SVD rappresenta un pilastro fondamentale nell'algebra lineare numerica. Le sue applicazioni spaziano dall'elaborazione dei segnali all'apprendimento automatico, rendendola uno strumento indispensabile nell'analisi dei dati.

Definizione Matematica

Data una qualsiasi matrice $A \in R^{m \times n}$, questa può essere fattorizzata come

$$A = U\Sigma V^T,$$

dove $U \in R^{m \times m}$ e $V \in R^{n \times n}$ sono ortogonali, $\Sigma \in R^{m \times n}$ è una matrice pseudo-diagonale se $m \neq n$, altrimenti è diagonale. Lungo la diagonale principale della

matrice Σ si troveranno i cosiddetti valori singolari, disposti in ordine decrescente e tutti strettamente non negativi. Questi valori singolari sono le radici degli autovalori ($\sigma_i = \sqrt{\lambda_i}$) della matrice $A^T A$:

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m,n)} \geq 0.$$

Le colonne delle matrici U e V sono dette **vettori singolari**. In particolare, la matrice U contiene i cosiddetti **vettori singolari sinistri** disposti per colonne, mentre la matrice V contiene, sempre nelle colonne, i cosiddetti **vettori singolari destri**. I vettori singolari sinistri sono gli autovettori della matrice AA^T , invece i vettori singolari destri sono gli autovettori della matrice $A^T A$. La tecnica di decomposizione SVD permette di scrivere la matrice di partenza A come una somma di sottomatrici, date dal prodotto dei vettori singolari e pesate per i rispettivi valori singolari:

$$A = \sum_{i=1}^{\min(m,n)} \sigma_i u_i v_i^T$$

Questa è comunemente chiamata *forma del prodotto esterno* ed è derivata nel seguente modo, considerata A quadrata di dimensione $n \times n$:

$$A = U\Sigma V^T = (u_1 \ u_2 \ \dots \ u_n) \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{pmatrix} \begin{pmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{pmatrix}$$

Teorema della decomposizione spettrale

Data una matrice $A \in R^{m \times n}$ e dato $p = \min(n, m)$, applicando la decomposizione in valori singolari $A = U\Sigma V^T$, se risultano:

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0 \quad r < p$$

cioè r è il numero di valori singolari positivi, mentre i successivi $\sigma_{r+1} \dots \sigma_p$ sono tutti nulli, allora:

- r è il rango di $A \in R^{m \times n}$
- $A = \sum_{i=1}^r \sigma_i u_i v_i^T = \sigma_1 u_1 v_1 + \sigma_2 u_2 v_2 + \dots + \sigma_r u_r v_r$ è detta **decomposizione spettrale di A**

dove u_i e v_i sono gli i -esimi vettori colonna delle matrici U e V rispettivamente. Notare che la decomposizione spettrale della matrice A non comporta alcuna perdita di informazioni sulla stessa, in quanto vengono considerati tutti i valori singolari non nulli.

Approssimazione Lower Rank

Data una matrice $A = U\Sigma V^T$ con $A \in R^{n \times m}$, e dato r il rango di A :

Fissiamo un intero $k < r$

$$A_k = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_k u_k v_k^T = U_k \Sigma_k V_k^T$$

A_k è una **approssimazione lower rank** della matrice A , con $U_k = (u_1, \dots, u_k)$, $V_k = (v_1, \dots, v_k)$, $\Sigma_k = \text{diag}(\sigma_1, \dots, \sigma_k)$. Questa approssimazione comporta certamente un errore che è possibile quantificare come segue:

$$\|A - A_k\|_2 = \sigma_{k+1}$$

Dato che i valori singolari σ_i sono ordinati in maniera decrescente, al tendere di k ad r , l'approssimazione migliora. In particolare, possiamo valutare la qualità dell'approssimazione considerando la cosiddetta **energia dei dati**. Per energia dei dati si intende il contenuto informativo degli stessi che, tramite l'applicazione dell'SVD e la successiva approssimazione lower rank, diminuisce. Per conoscere la percentuale di energia dei dati rimasta, è possibile considerare gli autovalori della matrice $A^T A$, cioè i quadrati dei valori singolari:

$$\text{Quality} = \frac{\sum_{i=1}^k \lambda_i}{\sum_{j=1}^n \lambda_j} = \frac{\sum_{i=1}^k \sigma_i^2}{\sum_{j=1}^n \sigma_j^2}$$

Risulta quindi evidente che i valori singolari rappresentino il contenuto informativo lungo una precisa direzione dello spazio di rappresentazione dei dati.

Algoritmi per la decomposizione in valori singolari

Gli algoritmi moderni per l'SVD mirano a bilanciare l'efficienza computazionale con la stabilità numerica, evitando la propagazione di errori dovuti alla limitata precisione delle operazioni in virgola mobile. Una delle tecniche più diffuse per calcolare l'SVD è la riduzione della matrice originale a una **forma bidiagonale** tramite una successione di trasformazioni ortogonali, come le rotazioni di Givens o le riflessioni di Householder. Una volta ottenuta la forma bidiagonale, si applicano algoritmi iterativi per calcolare i valori singolari e i corrispondenti vettori singolari. Tra questi algoritmi, il **metodo QR con shift** è particolarmente noto per la sua robustezza e stabilità numerica. Alternativamente, il **metodo divide-et-impera** è spesso utilizzato per matrici di grandi dimensioni, grazie alla sua capacità di parallelizzare i calcoli e sfruttare al meglio l'architettura hardware moderna. Un altro approccio importante è l'algoritmo di Jacobi modificato, che si distingue per la sua elevata precisione ma è meno competitivo in termini di velocità rispetto agli altri metodi. La fase di bidiagonalizzazione è di fondamentale importanza, poiché non solo riduce il problema dal punto di vista dimensionale e computazionale, ma preserva anche le proprietà numeriche della matrice originale, minimizzando la sensibilità agli errori di arrotondamento. Questo approccio rende gli algoritmi basati sulla bidiagonalizzazione particolarmente adatti per applicazioni che richiedono alta precisione, come la risoluzione di sistemi mal condizionati o l'analisi di dati di grandi dimensioni.

Metodi di Bidiagonalizzazione

La bidiagonalizzazione è una procedura fondamentale per ridurre una matrice $A \in R^{m \times n}$ a una forma più semplice, detta forma bidiagonale, attraverso l'uso di trasformazioni ortogonali. Questo processo permette di esprimere A come:

$$A = UBV^T$$

dove $U \in R^{m \times m}$ e $V \in R^{n \times n}$ sono matrici ortogonali, mentre $B \in R^{m \times n}$ è una matrice bidiagonale, ovvero una matrice che ha elementi non nulli solo sulla diagonale principale e sulla sovradiagonale (o sottodiagonale nel caso di matrici trasposte). La bidiagonalizzazione è un passaggio cruciale nella maggior parte degli algoritmi per il calcolo della decomposizione ai valori singolari (SVD), poiché riduce significativamente la complessità computazionale del problema successivo.

Riflessioni di Householder Uno dei metodi più comuni per ottenere la bidiagonalizzazione è l'uso delle riflessioni di Householder. Queste trasformazioni ortogonali consentono di annullare, passo dopo passo, gli elementi al di sotto della diagonale principale e della sovradiagonale, utilizzando delle particolari matrici, dette matrici elementari di Householder. In generale, una matrice elementare si può definire come $E(\alpha, u, v) = I - \alpha uv^T$ e, nel caso di una matrice elementare di Householder u e v si definiscono come un vettore w tale che $\|w\|^2 = 1$ e α lo scalare 2.

$$H = I - 2ww^T$$

Le matrici elementari di Householder sono involutorie ($H = H^{-1}$), e vengono tipicamente chiamate **riflettori di Householder** perché, premoltiplicate per un qualsiasi vettore $x \neq 0$, lo riflettono rispetto alla direzione ortogonale a w .

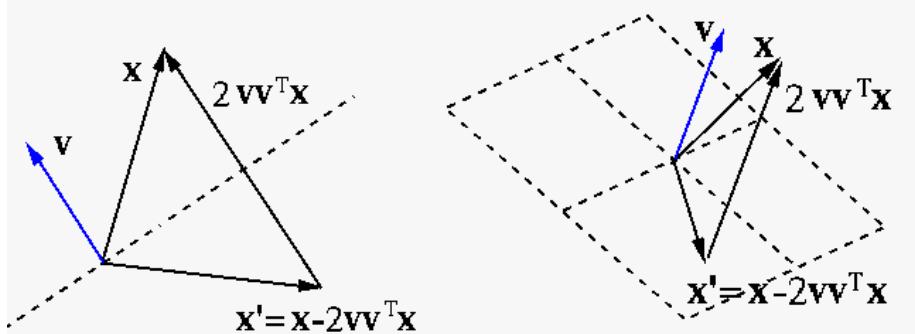


Figura 2: v rappresenta il vettore w

In particolare, dato un vettore colonna $x \in R^n$, è possibile utilizzare una matrice di Householder specifica per trasformarlo in un ulteriore vettore con la prima componente diversa da zero, cioè $Hx = -\sigma e_1$, con e_1 il primo vettore

della base canonica di R^n e $\sigma^2 = \|x\|_2^2$. La matrice di Householder che permette questa trasformazione è:

$$H = I - \frac{vv^T}{\sigma(\sigma + x_1)}$$

Dove $v = x + \sigma e_1$. Applicando queste riflessioni in sequenza sia alle righe che alle colonne di A , facendo attenzione a lasciare la sovradiagonale quando si trasformano le righe, si ottiene la riduzione bidiagonale. Questo metodo è noto per la sua stabilità numerica, grazie al fatto che le trasformazioni di Householder preservano la norma della matrice.

Rotazioni di Givens Un altro approccio consiste nell'utilizzo delle rotazioni di Givens, che sono particolarmente utili quando è necessario annullare singoli elementi specifici di una matrice. Una rotazione di Givens agisce su una coppia di righe o colonne, ed è definita da una matrice $G \in R^{n \times n}$ che modifica solo due righe i e j :

$$G = \begin{bmatrix} I & 0 & 0 \\ 0 & \begin{bmatrix} c & -s \\ s & c \end{bmatrix} & 0 \\ 0 & 0 & I \end{bmatrix},$$

dove $c = \cos(\theta)$ e $s = \sin(\theta)$ sono calcolati in modo da annullare l'elemento desiderato. Le rotazioni di Givens sono particolarmente efficaci in situazioni in cui la matrice è già quasi bidiagonale, ma richiedono un numero maggiore di operazioni rispetto alle riflessioni di Householder in matrici dense.

Confronto tra i Metodi Le riflessioni di Householder sono generalmente preferite per matrici dense, grazie alla loro efficienza computazionale complessiva e alla capacità di annullare più elementi in una singola operazione. Le rotazioni di Givens, invece, sono utilizzate principalmente in contesti in cui la matrice è già sparsa o ha una struttura quasi bidiagonale, poiché possono preservare questa struttura con modifiche localizzate.

Riduzione a Forma Bidiagonale

Si applicano trasformazioni ortogonali da entrambi i lati della matrice A per ridurla a una matrice bidiagonale B :

$$A = U_1 B V_1^T, \quad (1)$$

dove B ha elementi diversi da zero solo sulla diagonale principale e sulla sovradiagonale (o sottodiagonale), U e V invece derivano dalle trasformazioni eseguite con matrici di Householder o matrici di Givens.

SVD dalla Matrice Bidiagonale

La matrice B è ulteriormente decomposta come $B = U_2 \Sigma V_2^T$, utilizzando metodi iterativi come:

- **QR Iteration:** Iterazioni basate sulla fattorizzazione QR per ridurre B a una forma diagonale.
- **Divide and Conquer (D&C):** Suddivide il problema in sottoproblemi più piccoli per migliorare l'efficienza computazionale. Il metodo *Divide and Conquer* per il calcolo della SVD segue i seguenti passaggi:
 - Divisione:** La matrice A di dimensione $m \times n$ viene suddivisa in due sotto-matrici più piccole, continuando la suddivisione fino alla dimensione minima specificata.
 - SVD parziali:** Si calcolano le SVD parziali per ciascuna sottomatrice.
 - Combinazione:** I risultati delle SVD parziali vengono combinati, unendo le matrici ortogonali U e V e i valori singolari, attraverso la risoluzione di equazioni secolari.
 - Ricostruzione:** Si ottiene la SVD completa unendo i risultati parziali per ottenere la decomposizione finale.

Questo approccio riduce la complessità computazionale e permette l'utilizzo del calcolo parallelo, ovvero il grande punto di forza dell'hardware moderno(in particolare le GPU).

Calcolo della SVD con metodo di Golub-Kahan

L'algoritmo di Golub-Kahan-Reinsch per il calcolo della SVD di una matrice $A \in R^{m \times n}$ consiste di due fasi, una diretta e una iterativa. Nella prima fase A è trasformata in una matrice bidiagonale superiore della forma:

$$A = UBV^T$$

dove $U \in R^{m \times m}$ e $V \in R^{n \times n}$ sono matrici ortogonali ottenute dall'applicazione di riflettori di Householder alla matrice A iniziale. In particolare U è ottenuta a partire da riflessioni di Householder applicate su colonne, per eliminare tutti gli elementi sotto la diagonale, V invece è ottenuta attraverso le riflessioni fatte su righe che permettono di eliminare tutto ciò che si trova a destra della sovradiagonale. $B \in R^{m \times n}$ è una matrice bidiagonale superiore ottenuta a partire dalla A attraverso le applicazioni dei riflettori su righe e colonne. Nel secondo passaggio del processo, viene applicato l'algoritmo di iterazione QR alla matrice bidiagonale B. Questa seconda fase ha l'obiettivo di ottenere i valori singolari della matrice A, riducendo la matrice B alla sua forma diagonale, dove i valori singolari compaiono sulla diagonale principale. Durante l'iterazione QR, la matrice B viene successivamente scomposta in due matrici ortogonali Q e R mediante apposita fattorizzazione. La matrice Q è ortogonale, mentre la matrice R è triangolare superiore. Ogni volta che viene eseguita una fattorizzazione, le matrici Q e R vengono usate per aggiornare la matrice B, così come le matrici ortogonali U e V, che vengono ottenute attraverso i prodotti successivi con

la matrice Q durante il processo di iterazione. Il ciclo continua fino a quando la matrice B non diventa diagonale, il che è indicato dal valore dell'errore (il massimo valore della triangolare inferiore esclusa la diagonale) che scende sotto una certa soglia definita dalla tolleranza. Al termine, la matrice B contiene i valori singolari lungo la diagonale, mentre le matrici U e V sono aggiornate per rappresentare gli autovettori sinistri e destri.

```

1 function [U, S, V] = svd_algorithm(X)
2 % Applica la decomposizione di Householder per ottenere U, B e
3 % V
4 [UU, B, VV] = householder_bidiagonalization(X);
5 % Ottiene la matrice bidiagonale B e le matrici ortogonali U, V
6
7 [n, m] = size(B); % Ottieni la dimensione di B
8 B = B(1:n, 1:m); % Assicura che B sia della dimensione
9 % corretta
10
11 % Applica l'iterazione QR sulla matrice bidiagonale B
12 % per ottenere la matrice diagonale S
13 [W, S, Z] = qr_iteration_bidiag(B, 10^-8); % Esegui
14 % l'iterazione QR
15
16 % Calcola la matrice finale U e V
17 U = UU * W; % Moltiplica U di Householder per W
18 V = VV * Z; % Moltiplica V di Householder per Z
19
20 end

```

Listing 1: Algoritmo per il calcolo dell'SVD attraverso il metodo di Golub-Kahan

```

1 function [U, B, V] = householder_bidiagonalization(A)
2 % Ottieni la dimensione di A
3 [n, m] = size(A);
4
5 % Inizializzo U e V a matrici identit
6 U=eye(n);
7 V=eye(m);
8
9 % Applica le trasformazioni di Householder per ottenere la
10 % matrice bidiagonale
11 for k = 1:min(n, m-1) % Itera fino alla dimensione minima tra
12 % n e m-1
13 % Prendi la k-esima colonna a partire dalla riga k
14 x = A(k:n, k);
15
16 % Calcola il vettore di Householder v per annullare le
17 % componenti sotto la diagonale
18 e1 = zeros(length(x), 1);
19 e1(1) = 1;
20 alpha = norm(x);
21 v = x + sign(x(1)) * alpha * e1;
22 v = v / norm(v); % Normalizza il vettore v
23
24 % Calcola la matrice di Householder Hk per righe
25 Hk = eye(n-k+1) - 2 * (v * v');

```

```

24 % Applicare Hk a sinistra (aggiornare le righe di A)
25 A(k:n, k:m) = Hk * A(k:n, k:m);
26
27 % Aggiorna la matrice U
28 z = zeros(k-1, n-k+1);
29 Hk = [eye(k-1) z; z.' Hk]; % Costruisce Hk includendo la
29 % parte superiore a sinistra
30 U = U * Hk;
31
32 % Ora riflettiamo le colonne a partire dalla colonna k+1
33 x = A(k, k+1:m)';
34
35 % Calcola il vettore di Householder v per annullare le
35 % componenti sotto la diagonale
36 e1 = zeros(length(x), 1);
37 e1(1) = 1;
38 alpha = norm(x);
39 v = x + sign(x(1)) * alpha * e1; % Calcola il vettore v
39 % per le colonne
40 v = v / norm(v); % Normalizza v
41
42 % Calcola la matrice di Householder Hk2 per colonne
43 Hk2 = eye(m-k) - 2 * (v * v');
44
45 % Applicare Hk2 a destra (aggiornare le colonne di A)
46 A(k:n, k+1:m) = A(k:n, k+1:m) * Hk2;
47 % Aggiorna la matrice V
48 z = zeros(k, m-k); % Crea una matrice di zeri
49 Hk2 = [eye(k) z; z.' Hk2]; % Costruisce Hk2
50 V = V * Hk2; % Aggiorna V
51 end
52
53 % Gestisce il caso in cui n > m
54 if n > m
55     k = m;
56     x = A(k:n, k); % Estrai il vettore dalla k-esima colonna
56 % alla fine
57     e1 = zeros(length(x), 1);
58     e1(1) = 1;
59     alpha = norm(x);
60     v = x + sign(x(1)) * alpha * e1; % Calcola il vettore di
60 % Householder
61     v = v / norm(v); % Normalizza v
62
63 % Calcola la matrice di Householder Hk per righe
64 Hk = eye(n-k+1) - 2 * (v * v');
65
66 % Applicare Hk a sinistra (aggiornare le righe di A)
67 A(k:n, k:m) = Hk * A(k:n, k:m);
68
69 z = zeros(k-1, n-k+1); % Crea una matrice di zeri
70 Hk = [eye(k-1) z; z.' Hk]; % Costruisce Hk includendo la
70 % parte superiore
71 U = U * Hk;
72 end
73
74 % La matrice A ora si trova in forma bidiagonale

```

```

75 B = A; % Assegno a B la matrice bidiagonale ottenuta
76 end

```

Listing 2: Bidiagonalizzazione di Householder

```

1 function [U, S, V] = qr_iteration_bidiag(X, tol)
2 % Ottieni la dimensione di X
3 [M, N] = size(X);
4 loopmax = 100 * max(size(X));
5 loopcount = 0; % Inizializza il contatore delle iterazioni
6 U = eye(M);
7 S = X'; % Trasposta di X per facilitare l'algoritmo QR
8 V = eye(N);
9 Err = realmax; % Inizializza un errore molto grande per
    entrare nel ciclo
10
11 while (Err > tol && loopcount < loopmax)
12     [Q, S] = qr(S');
13     U = U * Q; % Aggiorna U con il prodotto di Q
14     [Q, S] = qr(S');
15     V = V * Q; % Aggiorna V con il prodotto di Q
16     e = triu(S, 1); % Estrai la parte triangolare superiore
        di S
17     E = norm(e(:)); % Calcola la norma delle componenti
        superiori di S
18     F = norm(diag(S)); % Calcola la norma della diagonale di S
19     Err = E / F; % Calcola l'errore relativo tra la parte
        superiore e la diagonale
20     loopcount = loopcount + 1; % Incrementa il contatore
        delle iterazioni
21 end
22
23 % Gestione dei segni nella matrice diagonale S
24 SS = diag(S); % Estrai la diagonale di S
25 S = zeros(M, N); % Crea una matrice nulla di dimensione MxN
    per S
26 for n = 1:length(SS)
27     SSN = SS(n); % Estrai ogni elemento della diagonale
28     S(n, n) = abs(SSN); % Imposta il valore assoluto nella
        diagonale di S
29     if SSN < 0 % Se il valore risulta negativo,
30         % cambia il segno della colonna corrispondente di U
31         U(:, n) = -U(:, n);
32     end
33 end
34 end

```

Listing 3: Iterazione QR

2.2 Reti neurali

Introduzione

Le reti neurali artificiali (ANN, Artificial Neural Networks) sono modelli computazionali ispirati al funzionamento dei neuroni umani, utilizzati per risolvere

problemi complessi come il riconoscimento di pattern, la classificazione e l'ottimizzazione. Si basano su una struttura composta da nodi (*neuroni*), organizzati in strati:

- **Strato di input**, che riceve i dati iniziali.
- **Strati nascosti**, che elaborano i dati attraverso trasformazioni non lineari.
- **Strato di output**, che restituisce il risultato finale.

Struttura di Base di un Neurone Artificiale

Un singolo neurone artificiale prende in ingresso un vettore di input $\mathbf{x} = [x_1, x_2, \dots, x_n]^\top$, lo combina linearmente usando un vettore di pesi $\mathbf{w} = [w_1, w_2, \dots, w_n]^\top$, aggiunge un termine di bias b , e applica una funzione di attivazione $\phi(\cdot)$. La sua uscita y è calcolata come:

$$y = \phi \left(\sum_{i=1}^n w_i x_i + b \right) = \phi (\mathbf{w}^\top \mathbf{x} + b). \quad (2)$$

Dove:

- $\phi(\cdot)$ è una funzione non lineare, ad esempio la funzione sigmoide, ReLU (*Rectified Linear Unit*), definita come $\text{ReLU}(z) = \max(0, z)$, o la funzione \tanh (tangente iperbolica).
- b è un parametro che permette di spostare l'output della funzione di attivazione.

Reti a Singolo Strato

Una rete neurale a singolo strato consiste di uno strato di input e uno strato di output, senza strati nascosti. Le reti più comuni di questo tipo sono i percetroni o i percetroni a soglia.

Formula Generale

Se abbiamo m neuroni nello strato di output, l'output della rete è un vettore $\mathbf{y} = [y_1, y_2, \dots, y_m]^\top$, dove ogni componente è calcolata come:

$$y_j = \phi \left(\sum_{i=1}^n w_{ji} x_i + b_j \right), \quad \text{con } j = 1, 2, \dots, m. \quad (3)$$

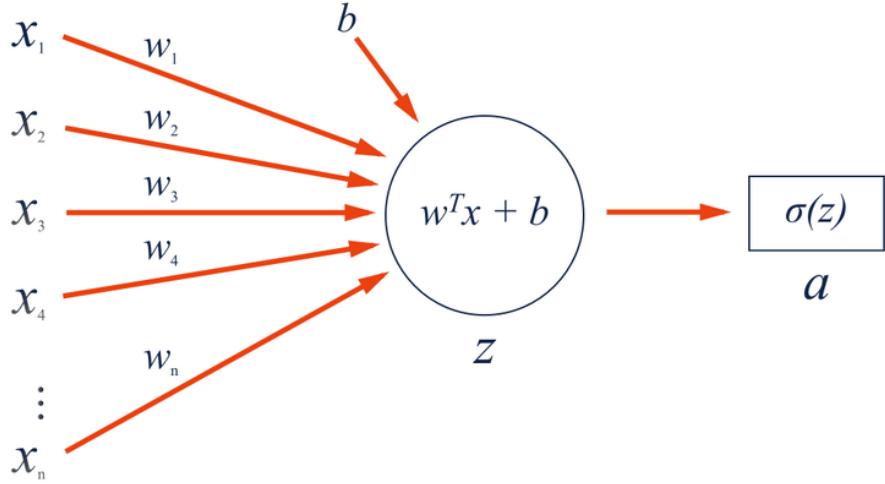


Figura 3: Struttura di un neurone

Limitazioni

Le reti a singolo strato sono limitate nella capacità di apprendere funzioni non lineari. Ad esempio, una rete neurale single-layer non è in grado di risolvere problemi di classificazione su dati non linearmente separabili, come il problema dello XOR, per il quale sono necessarie delle reti multi-layer.

Reti Multistrato

Le reti multistrato (o *Multi-Layer Perceptron*, MLP) o deep neural network, consistono in uno strato di input, diversi strati nascosti e uno strato di output. Ogni strato elabora i dati ricevuti applicando una combinazione lineare seguita da una funzione di attivazione non lineare. Questo approccio consente alla rete di apprendere rappresentazioni via via più complesse dei dati.

Consideriamo una rete con un vettore di input $\mathbf{x} \in R^d$, L strati nascosti e uno strato di output che produce un vettore $\mathbf{y} \in R^m$. Il processo di calcolo inizia dallo strato di input e attraversa gli strati nascosti fino a raggiungere lo strato di output.

Il primo strato nascosto riceve il vettore di input e calcola:

$$\mathbf{z}^{(1)} = \phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}),$$

dove $\mathbf{W}^{(1)} \in R^{h_1 \times d}$ è la matrice dei pesi che collega l'input al primo strato nascosto, $\mathbf{b}^{(1)} \in R^{h_1}$ è il vettore dei bias associati e ϕ rappresenta una funzione di attivazione non lineare, come la ReLU o la sigmoide. Il risultato, $\mathbf{z}^{(1)} \in R^{h_1}$, è l'output del primo strato nascosto.

Ogni strato nascosto successivo, indicato con $l = 2, \dots, L$, prende come input l'output dello strato precedente e calcola:

$$\mathbf{z}^{(l)} = \phi \left(\mathbf{W}^{(l)} \mathbf{z}^{(l-1)} + \mathbf{b}^{(l)} \right),$$

dove $\mathbf{W}^{(l)} \in R^{h_l \times h_{l-1}}$ è la matrice dei pesi tra lo strato $l - 1$ e lo strato l , e $\mathbf{b}^{(l)} \in R^{h_l}$ è il vettore dei bias associato allo strato l . Il risultato, $\mathbf{z}^{(l)} \in R^{h_l}$, rappresenta l'output dello strato nascosto l .

Infine, lo strato di output combina l'output dell'ultimo strato nascosto per produrre il risultato finale:

$$\mathbf{y} = \phi \left(\mathbf{W}^{(L+1)} \mathbf{z}^{(L)} + \mathbf{b}^{(L+1)} \right),$$

dove $\mathbf{W}^{(L+1)} \in R^{m \times h_L}$ è la matrice dei pesi tra l'ultimo strato nascosto e lo strato di output, mentre $\mathbf{b}^{(L+1)} \in R^m$ è il bias associato. Il vettore risultante \mathbf{y} rappresenta l'output complessivo della rete. Risulta molto importante in questo caso la scelta della funzione di attivazione del layer di uscita; infatti, se la rete assolve compiti di classificazione, dovrà essere impiegata una funzione di tipo Softmax, che associa ad ogni classe un valore di probabilità di appartenenza alla stessa. Alternativamente, per compiti di regressione, è possibile omettere la funzione di attivazione dell'uscita, in quanto ci si aspettano degli output continui. Il punto di forza di una rete multi-layer è che questa riesce ad apprendere dei pattern molto nascosti dai dati di addestramento, pattern che difficilmente risultano visibili a noi umani. Questo rende tali strumenti delle cosiddette *black box*, ovvero rende difficile identificarne il preciso funzionamento, anche analizzando dettagliatamente ogni singolo valore dei pesi appresi in fase di training. Questa struttura gerarchica permette alla rete di modellare funzioni complesse, anche non lineari o non continue, sfruttando la capacità degli strati nascosti di trasformare progressivamente i dati per rivelare le caratteristiche rilevanti ai fini del compito specifico.

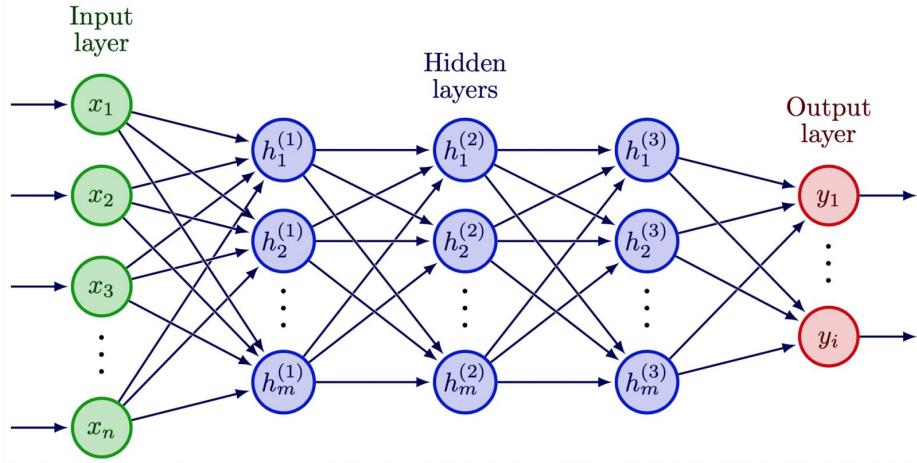


Figura 4: Deep neural network

Differenze tra Singolo Strato e Multistrato

Le reti neurali a singolo strato e quelle a multistrato differiscono significativamente in termini di capacità, struttura e applicazioni.

Per quanto riguarda la capacità di rappresentazione, una rete a singolo strato è limitata a rappresentare funzioni linearmente separabili. Al contrario, una rete multistrato è in grado di approssimare qualunque funzione, grazie al **Teorema di universalità dell'approssimazione**, purché abbia un numero sufficiente di neuroni negli strati nascosti. Questo rende le reti multistrato molto più versatili per affrontare problemi complessi.

Un'altra differenza chiave risiede nelle funzioni di attivazione. Le reti a singolo strato utilizzano tipicamente funzioni a soglia o sigmoide mentre le reti multistrato impiegano funzioni di attivazione come la ReLU, al fine di evitare problemi di scomparsa del gradiente, che affliggono le reti molto profonde.

Infine, le applicazioni delle due tipologie di rete riflettono queste differenze. Le reti a singolo strato sono adatte a problemi semplici, come la classificazione di dati linearmente separabili. D'altra parte, le reti multistrato sono indispensabili per affrontare problemi complessi, come il riconoscimento delle immagini, l'elaborazione del linguaggio naturale e la generazione di immagini. Grazie alla loro capacità di rappresentare relazioni non lineari e apprendere strutture nascoste nei dati, le reti multistrato sono alla base di molte delle tecnologie moderne basate sull'intelligenza artificiale.

Addestramento delle Reti Neurali

L'addestramento consiste nell'ottimizzazione dei parametri (pesi e bias) per minimizzare l'errore tra l'output previsto \hat{y} e quello desiderato y .

1. Forward Propagation

I dati di input \mathbf{x} attraversano la rete, generando gli output intermedi. Per una rete a due strati:

$$\mathbf{z}^{(1)} = \phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}), \quad (4)$$

$$\hat{\mathbf{y}} = \phi(\mathbf{W}^{(2)}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}), \quad (5)$$

dove $\phi(\cdot)$ è una funzione di attivazione non lineare, come la ReLU o la sigmoide.

2. Calcolo della Perdita

La funzione di perdita misura l'errore tra l'output previsto $\hat{\mathbf{y}}$ e quello desiderato \mathbf{y} . Alcuni esempi:

- **Errore Quadratico Medio (MSE):**

$$L = \frac{1}{N} \sum_{i=1}^N \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2. \quad (6)$$

- **Cross Entropy** (per problemi di classificazione):

$$L = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c}). \quad (7)$$

3. Backpropagation

Si calcolano i gradienti della perdita rispetto ai parametri della rete usando la regola della catena. Per un peso generico w :

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}, \quad (8)$$

dove η è il tasso di apprendimento (*learning rate*).

4. Aggiornamento dei Parametri

L'ottimizzazione iterativa utilizza algoritmi basati sui metodi di discesa, applicati al gradiente dell'errore, come:

- **Stochastic Gradient Descent (SGD):** tecnica di aggiornamento dei pesi particolarmente adatta all'addestramento con batch.
- **Adam:** combina gradienti con scale adattative per una convergenza più rapida.

2.3 Tipi di Reti Neurali

Esistono diversi tipi di reti neurali, ognuna adatta a specifici tipi di problemi. Di seguito vengono descritti brevemente i principali tipi di reti neurali:

Reti Ricorrenti (Recurrent Neural Networks, RNN)

Le reti neurali ricorrenti (RNN) sono particolarmente utili per l'elaborazione di sequenze temporali o dati sequenziali, come il testo o i segnali audio. A differenza delle reti feedforward, le RNN hanno connessioni cicliche che permettono di memorizzare informazioni da passi precedenti della sequenza. Le RNN sono utilizzate in applicazioni come la traduzione automatica, il riconoscimento vocale e la generazione di testo.

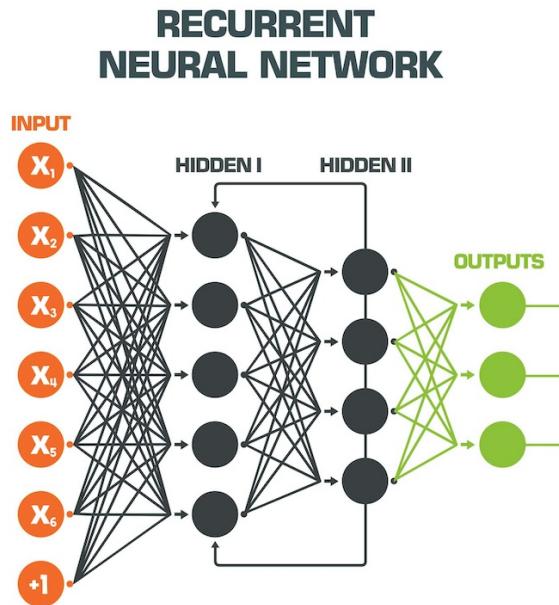


Figura 5: Struttura Reti ricorrenti

Reti Convoluzionali (Convolutional Neural Networks, CNN)

Le reti neurali convoluzionali (CNN) sono progettate specificamente per il trattamento delle immagini, utilizzano operazioni di convoluzione per applicare filtri ai dati in modo da estrarre caratteristiche rilevanti, come bordi e forme. Le CNN sono particolarmente efficaci nel riconoscimento di pattern visivi e sono ampiamente utilizzate in compiti come il riconoscimento delle immagini, la segmentazione e la classificazione.

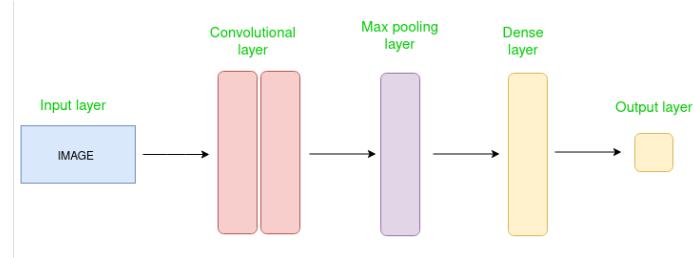


Figura 6: Struttura reti convoluzionali

Le reti neurali convoluzionali, o CNN (Convolutional Neural Networks), rappresentano un avanzamento cruciale nel campo dell'apprendimento profondo. Progettate per gestire dati con struttura spaziale, come immagini, video o segnali audio, le CNN sono oggi tra gli strumenti più potenti e diffusi per la visione artificiale. Dopo la loro introduzione negli anni '80, queste reti sono diventate fondamentali in molte applicazioni, dal riconoscimento di oggetti alla segmentazione semantica, grazie alla loro capacità di apprendere automaticamente caratteristiche rilevanti dai dati grezzi.

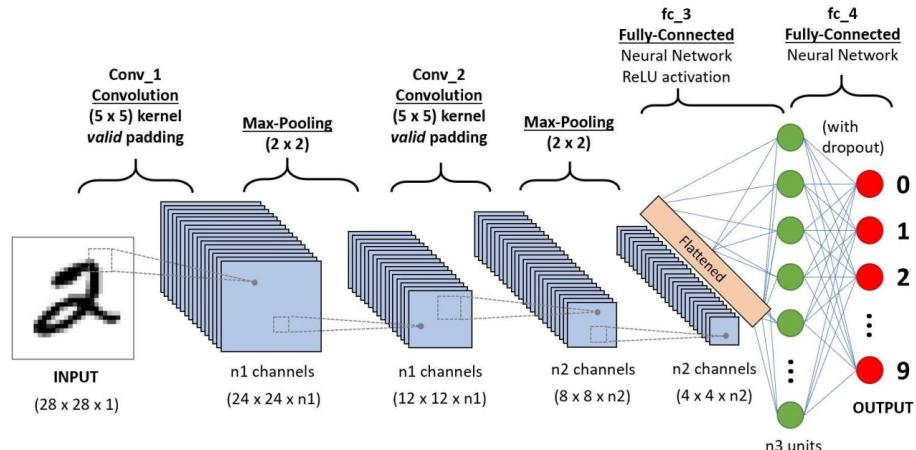


Figura 7: Struttura di una CNN

Struttura di una CNN

Una CNN elabora i dati in ingresso attraverso una sequenza di operazioni strutturate in strati. Questi strati, che operano in modo sequenziale o parallelo, collaborano per estrarre e trasformare le caratteristiche rilevanti, partendo da

dettagli elementari (ad esempio, bordi e angoli) fino a strutture più complesse (come forme o oggetti interi).

Lo **strato convoluzionale**, il cuore della CNN, applica un'operazione chiamata convoluzione per estrarre caratteristiche locali dai dati in ingresso. Durante questa fase, una serie di filtri (o kernel) viene fatta scorrere su porzioni dei dati. Matematicamente, l'operazione può essere rappresentata come:

$$z_{i,j}^{(l)} = \sum_{c=1}^{C^{(l-1)}} \sum_{p=1}^{k_h} \sum_{q=1}^{k_w} x_{i+p-1,j+q-1,c}^{(l-1)} \cdot K_{p,q,c}^{(l)} + b^{(l)},$$

dove:

- $x_{i,j,c}^{(l-1)}$ è il valore del pixel nella posizione (i, j) del canale c nello strato $l-1$;
- $K_{p,q,c}^{(l)}$ rappresenta i valori del kernel di dimensione $k_h \times k_w$;
- $b^{(l)}$ è il bias associato al filtro;
- $z_{i,j}^{(l)}$ è l'output della convoluzione nella posizione (i, j) per lo strato l .

Questa operazione viene ripetuta per ogni filtro, generando un insieme di mappe di attivazione che rappresentano le risposte locali ai filtri applicati.

Per controllare le dimensioni delle mappe di attivazione risultanti, si utilizzano due tecniche fondamentali: il **padding**, che aggiunge bordi artificiali attorno ai dati in ingresso, e lo **stride**, che definisce il passo di spostamento del filtro. Le dimensioni dell'output risultante si calcolano come:

$$H_{\text{out}} = \frac{H_{\text{in}} + 2p - k_h}{s} + 1, \quad W_{\text{out}} = \frac{W_{\text{in}} + 2p - k_w}{s} + 1,$$

dove $H_{\text{in}}, W_{\text{in}}$ rappresentano le dimensioni originali dell'immagine, p il padding, s lo stride, e k_h, k_w le dimensioni del kernel.

Dopo la convoluzione, la rete applica una **funzione di attivazione** non lineare, come la ReLU (*Rectified Linear Unit*), definita come:

$$a_{i,j}^{(l)} = \max(0, z_{i,j}^{(l)}).$$

Un altro componente cruciale è lo **strato di pooling**, che riduce la dimensionalità delle mappe di attivazione mantenendo le informazioni essenziali. Ad esempio, nel *max pooling*, si seleziona il valore massimo in una finestra di dimensione $p_h \times p_w$:

$$p_{i,j}^{(l)} = \max_{p,q} a_{i+p-1,j+q-1}^{(l)},$$

mentre nel *average pooling* si calcola la media dei valori nella finestra:

$$p_{i,j}^{(l)} = \frac{1}{p_h \cdot p_w} \sum_{p=1}^{p_h} \sum_{q=1}^{p_w} a_{i+p-1,j+q-1}^{(l)}.$$

Questi passaggi contribuiscono a rendere la rete più robusta alle variazioni nei dati, come traslazioni o rotazioni, migliorando l'invarianza spaziale del modello.

Dalla Convoluzione all'Output Finale

Dopo una serie di strati convoluzionali e di pooling, le mappe di attivazione vengono appiattite in un vettore. Questo vettore attraversa uno o più **strati completamente connessi**, che combinano tutte le informazioni apprese per produrre l'output finale. Il calcolo in uno strato completamente connesso, come visto in precedenza, può essere espresso come:

$$y = \phi(Wv + b),$$

dove:

- W è la matrice dei pesi;
- v è il vettore appiattito delle caratteristiche;
- b è il bias;
- ϕ è una funzione di attivazione, come la *softmax* per la classificazione.

Addestramento e Ottimizzazione

L'obiettivo dell'addestramento di una CNN è minimizzare una funzione di perdita, come la *Cross-Entropy Loss* per la classificazione:

$$L = - \sum_{c=1}^C y_c \log(\hat{y}_c),$$

dove y_c è il valore target e \hat{y}_c è la probabilità predetta per la classe c . Per problemi di regressione, si può utilizzare il *Mean Squared Error* (MSE):

$$L = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2.$$

Durante l'addestramento, si sfrutta l'algoritmo di backpropagation per calcolare i gradienti della funzione di perdita rispetto ai parametri della rete (K, W, b). Questi gradienti vengono poi utilizzati per aggiornare i parametri attraverso tecniche di ottimizzazione come il Gradient Descent:

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta},$$

dove θ rappresenta un parametro della rete e η il tasso di apprendimento.

Vantaggi e Applicazioni

Grazie alla loro architettura, le CNN combinano efficienza ed efficacia, riducendo il numero di parametri rispetto alle reti completamente connesse e garantendo invarianza spaziale. Ciò le rende adatte a una vasta gamma di applicazioni, tra cui visione artificiale, riconoscimento facciale, elaborazione video e guida autonoma. Nonostante alcune sfide, come la necessità di grandi quantità di dati e risorse computazionali elevate, le CNN rimangono uno degli strumenti più potenti nell'apprendimento automatico.

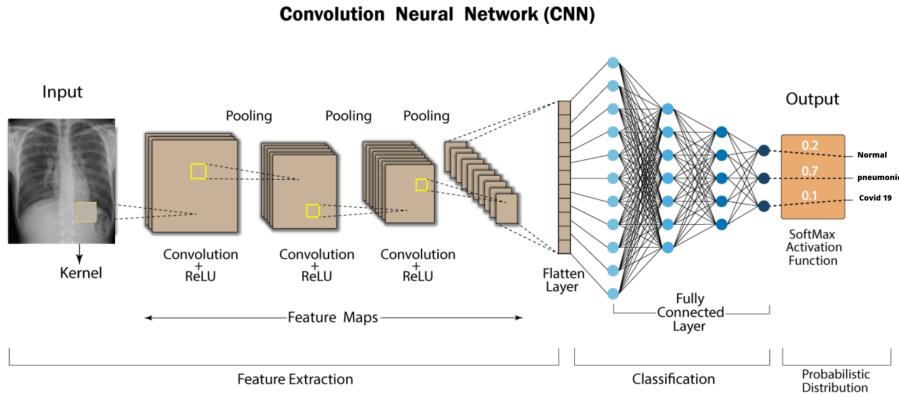


Figura 8: Esempio di CNN

3 Caso di applicazione: dati multidimensionali

In questo caso di applicazione l'obiettivo è mostrare come l'applicazione di una tecnica di SVD a dei dati numerici permetta di ridurre la dimensionalità di questi dati con una minima perdita di informazione, dando grandi vantaggi in termini di risorse richieste per memorizzare grandi quantità di dati. In particolare si procederà all'addestramento di una rete neurale su un dataset numerico a piena dimensionalità, successivamente si procederà a ridurre la dimensionalità tramite SVD e si ritenterà l'addestramento, confrontando infine i risultati.

3.1 Introduzione al dataset e alla rete utilizzata

Il dataset utilizzato è reperibile dall'archivio UCI (disponibile online), e si tratta di "Optical Recognition of Handwritten Digits" che è stato prodotto nel 1998, utilizzando programmi di pre-elaborazione messi a disposizione dal NIST per estrarre bitmap normalizzati di cifre scritte a mano da un modulo prestampato. Di un totale di 43 persone coinvolte, 30 hanno contribuito al set di addestramento e 13 diverse al set di test. I bitmap 32x32 delle cifre vengono divisi in blocchi non sovrapposti di 4x4 e il numero di pixel attivi viene conteggiato in

ciascun blocco. Questo genera una matrice di input 8x8 dove ogni elemento è un numero intero nel range 0..16. Questo riduce la dimensionalità e conferisce invarianza a piccole distorsioni. Di seguito il codice commentato per l'addestramento della rete neurale, con i dati non trattati da alcuna tecnica, e i tre addestramenti eseguiti sul dataset:

```

1 % Leggere i file di dati
2 file1 = readmatrix("optdigits.tes", FileType="text"); % Legge il
   file di test
3 file2 = readmatrix("optdigits.tra", FileType="text"); % Legge il
   file di training
4
5 % Unire i dataset
6 dataset=[file2; file1]; % Combina i due dataset in un unico array
7
8 % Estrarre etichette e caratteristiche
9 labels=dataset(:, 65);      % L'ultima colonna contiene le
   etichette (valori target)
10 features=dataset(:, 1:64);   % Le prime 64 colonne sono le feature
   (valori input)
11
12 train_ratio = 0.8; % 80% dei dati sara usato per il training
13 % Numero totale di campioni
14 num_samples = size(features, 1); % Numero totale di righe nel
   dataset
15
16 % Indici casuali per il training e il test
17 random_indices = randperm(num_samples); % Mescola casualmente gli
   indici dei dati
18 num_train = round(train_ratio * num_samples); % Calcola il numero
   di campioni per il training
19 train_indices = random_indices(1:num_train); % Indici per i dati
   di training
20 test_indices = random_indices(num_train+1:end); % Indici per i
   dati di test
21
22 % Dividere dati in training e test
23 X_train = features(train_indices, :); % Feature per il training
24 y_train = labels(train_indices);        % Etichette per il training
25 X_test = features(test_indices, :);    % Feature per il test
26 y_test = labels(test_indices);         % Etichette per il test
27
28 % Creazione della rete neurale
29 layers = [
30     featureInputLayer(size(X_train, 2), "Name", "featureinput") %
       Livello di input per 64 feature
31     fullyConnectedLayer(40, "Name", "fc1") %
       Primo livello fully-connected con 40 neuroni
32     reluLayer("Name", "relu1") %
       Livello ReLU
33     batchNormalizationLayer("Name", "batchnorm1") %
       Batch normalization
34     dropoutLayer(0.1, "Name", "dropout1") %
       Dropout con probabilita 0.1
35     fullyConnectedLayer(20, "Name", "fc2") %
       Secondo livello fully-connected con 20 neuroni
36     reluLayer("Name", "relu2") %

```

```

    Livello ReLU
37 batchNormalizationLayer("Name", "batchnorm2") %  

    Batch normalization
38 dropoutLayer(0.1, "Name", "dropout2") %  

    Dropout con probabilita 0.1
39 fullyConnectedLayer(10, "Name", "fc3") %  

    Terzo livello fully-connected con 10 neuroni (numero di  

    classi)
40 softmaxLayer("Name", "softmax") %  

    Livello softmax per calcolare probabilita per ciascuna  

    classe
41 classificationLayer("Name", "classification")]; %  

    Livello di classificazione per calcolare l'errore
42
43 % Opzioni di allenamento
44 options = trainingOptions("adam", ...
45     MaxEpochs=300, ... % Numero  

        massimo di epoche
46     InitialLearnRate=0.0005, ... % Velocita di  

        apprendimento iniziale
47     GradientThreshold=1, ... % Soglia per il  

        gradiente (per evitare esplosioni del gradiente)
48     ValidationData=[X_test, categorical(y_test)], ... % Dati di  

        validazione (feature e etichette)
49     Shuffle="every-epoch", ... % Mescola i  

        dati a ogni epoca
50     Plots="training-progress", ... % Mostra il  

        grafico dei progressi di allenamento
51     MiniBatchSize=1000, ... % Dimensione  

        del mini-batch
52     Verbose=false); % Disattiva  

        output dettagliati nella console
53
54 % Allenare la rete
55 trainedNet = trainNetwork(X_train, categorical(y_train), layers,  

    options);

```

3.2 Addestramento e risultati

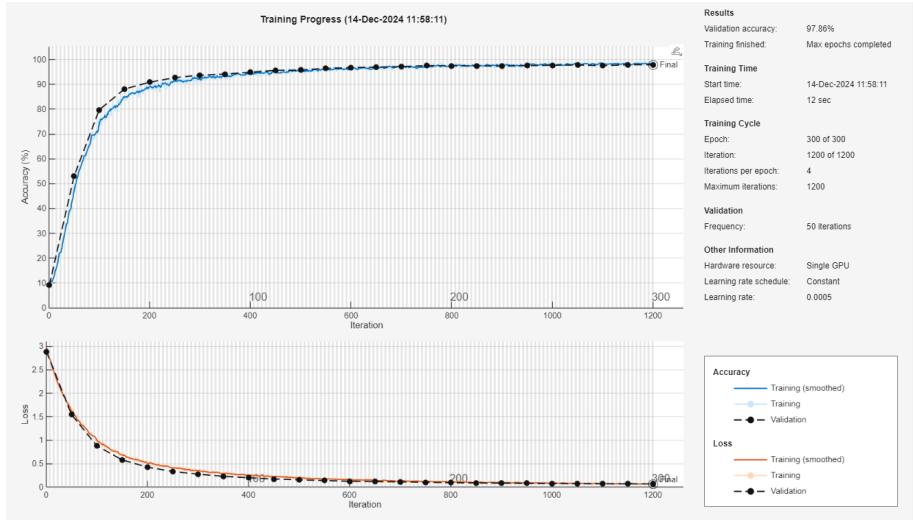


Figura 9: Primo addestramento effettuato

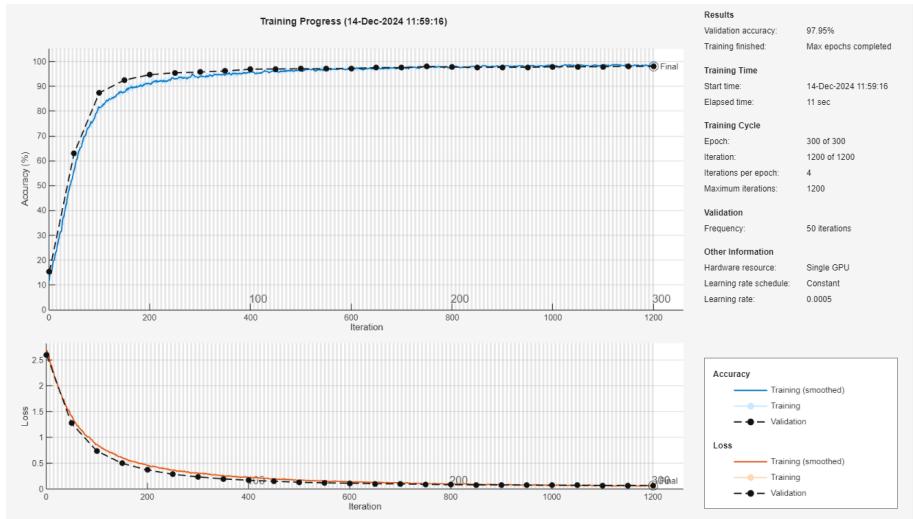


Figura 10: Secondo addestramento effettuato

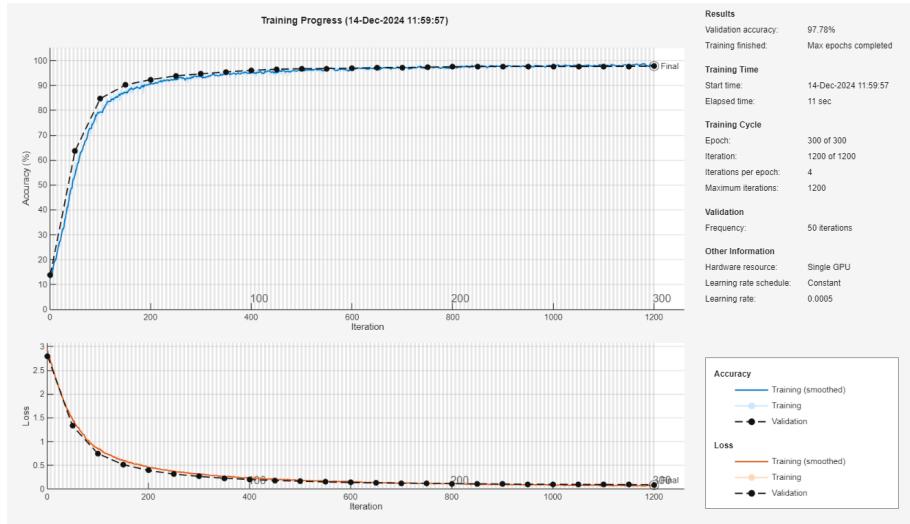


Figura 11: Terzo addestramento effettuato

Come si può notare dalle immagini raffiguranti gli addestramenti, in media il classificatore ha raggiunto il 97.86% di precisione sui dati di test, un ottimo risultato. In media l'addestramento ha richiesto 11.3s per terminare 300 epoche su una GPU discreta. Successivamente procediamo ad applicare la riduzione di dimensionalità con SVD ai dati.

3.3 Applicazione di SVD al dataset

Di seguito il codice MATLAB utilizzato (la parte di costruzione della rete neurale e di addestramento risultano invariate).

```

1 function [U, B, V] = householder_bidiagonalization(A)
2 function [U, S, V] = qr_iteration_bidiag(X, tol)
3 function [U, S, V] = svd_algorithm(X)
4
5 % Leggere i file di dati
6 file1 = readmatrix("optdigits.tes", FileType="text"); % Legge il
    file di test
7 file2 = readmatrix("optdigits.tra", FileType="text"); % Legge il
    file di training
8
9 % Unire i dataset
10 dataset = [file2; file1]; % Combina i due dataset in un unico array
11
12 % Estrarre etichette e caratteristiche
13 labels = dataset(:, 65); % L'ultima colonna contiene le
    etichette (valori target)
14 features = dataset(:, 1:64); % Le prime 64 colonne sono le
    feature (valori input)
15
16 [U, S, V] = svd_algorithm(features);

```

```

17 k = 5; %numero di valori singolari da selezionare
18 V_k = V(:, 1:k);
19 features = features * V_k; %riduzione della dimensionalita
20 eigenvalues = diag(S).^2; %calcolo degli autovalori
21 quality = sum(eigenvalues(1:k))/sum(eigenvalues) %valutazione
22     della qualita rispetto ai dati originali
23 size(features)
24
25 % Percentuale di dati da usare per il training
26 train_ratio = 0.8; % 80% dei dati sara usato per il training
27
28 % Numero totale di campioni
29 num_samples = size(features, 1); % Numero totale di righe nel
30     dataset
31
32 % Indici casuali per il training e il test
33 random_indices = randperm(num_samples); % Mescola casualmente gli
34     indici dei dati
35 num_train = round(train_ratio * num_samples); % Calcola il numero
36     di campioni per il training
37 train_indices = random_indices(1:num_train); % Indici per i dati
38     di training
39 test_indices = random_indices(num_train+1:end); % Indici per i
40     dati di test
41
42 % Dividere dati in training e test
43 X_train = features(train_indices, :); % Feature per il training
44 y_train = labels(train_indices); % Etichette per il training
45 X_test = features(test_indices, :); % Feature per il test
46 y_test = labels(test_indices); % Etichette per il test
47
48 % Creazione della rete neurale
49 layers = [
50     featureInputLayer(size(X_train, 2), "Name", "featureinput") %
51         Livello di input per 64 feature
52     fullyConnectedLayer(40, "Name", "fc1") %
53         Primo livello fully-connected con 40 neuroni
54     reluLayer("Name", "relu1") %
55         Livello ReLU
56     batchNormalizationLayer("Name", "batchnorm1") %
57         Batch normalization
58     dropoutLayer(0.1, "Name", "dropout1") %
59         Dropout con probabilita 0.1
60     fullyConnectedLayer(20, "Name", "fc2") %
61         Secondo livello fully-connected con 20 neuroni
62     reluLayer("Name", "relu2") %
63         Livello ReLU
64     batchNormalizationLayer("Name", "batchnorm2") %
65         Batch normalization
66     dropoutLayer(0.1, "Name", "dropout2") %
67         Dropout con probabilita 0.1
68     fullyConnectedLayer(10, "Name", "fc3") %
69         Terzo livello fully-connected con 10 neuroni (numero di
70             classi)
71     softmaxLayer("Name", "softmax") %
72         Livello softmax per calcolare probabilita per ciascuna
73             classe

```

```

55    classificationLayer("Name", "classification")];           %
      Livello di classificazione per calcolare l'errore
56
57 % Opzioni di allenamento
58 options = trainingOptions("adam", ...
59     MaxEpochs=300, ...                                     % Numero
60     massimo di epoche
61     InitialLearnRate=0.0005, ...                         % Velocita di
62     apprendimento iniziale
63     GradientThreshold=1, ...                            % Soglia per il
64     gradiente (per evitare esplosioni del gradiente)
65     ValidationData={X_test, categorical(y_test)}, ... % Dati di
66     validazione (feature e etichette)
67     Shuffle="every-epoch", ...                          % Mescola i
68     dati a ogni epoca
69     Plots="training-progress", ...                      % Mostra il
70     grafico dei progressi di allenamento
71     MiniBatchSize=1000, ...                            % Dimensione
72     del mini-batch
73     Verbose=false);                                  % Disattiva
74     output dettagliati nella console
75 % Allenare la rete
76 trainedNet = trainNetwork(X_train, categorical(y_train), layers,
77 options);

```

3.4 Addestramento con dati a dimensionalità ridotta

Infine sono stati eseguiti dei tentativi di addestramento della stessa rete neurale sui dati, con differente numero di caratteristiche, e di conseguenza con una differente qualità dell'approssimazione con SVD. In particolare, i tentativi sono stati eseguiti dimezzando il numero di colonne per ogni iterazione. Di seguito i risultati:

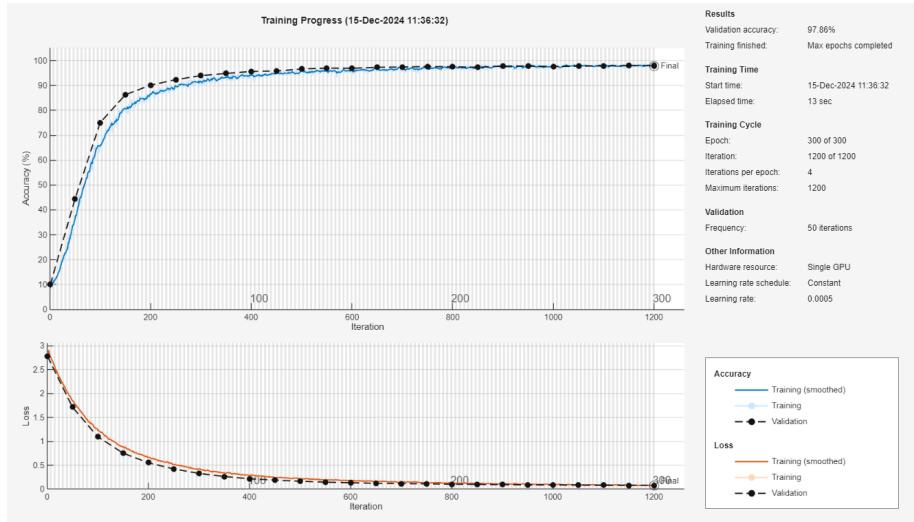


Figura 12: Addestramento con 32 features

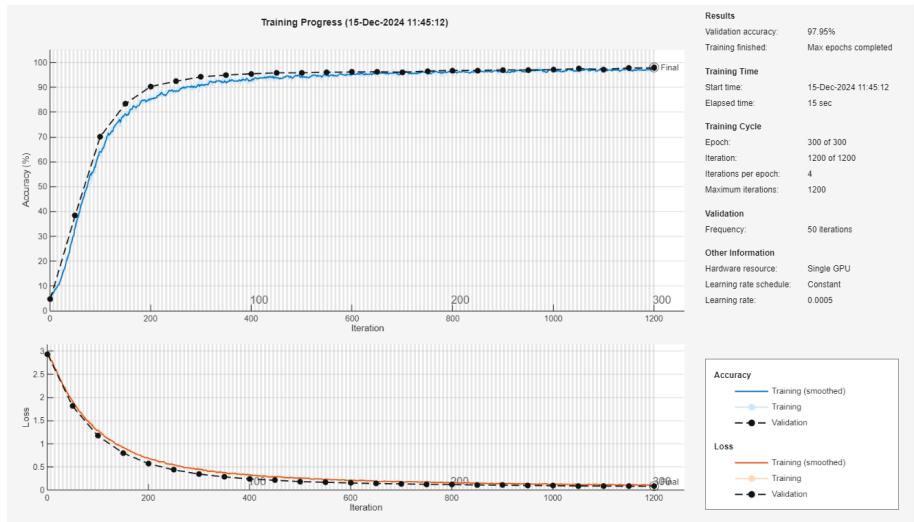


Figura 13: Addestramento con 16 features

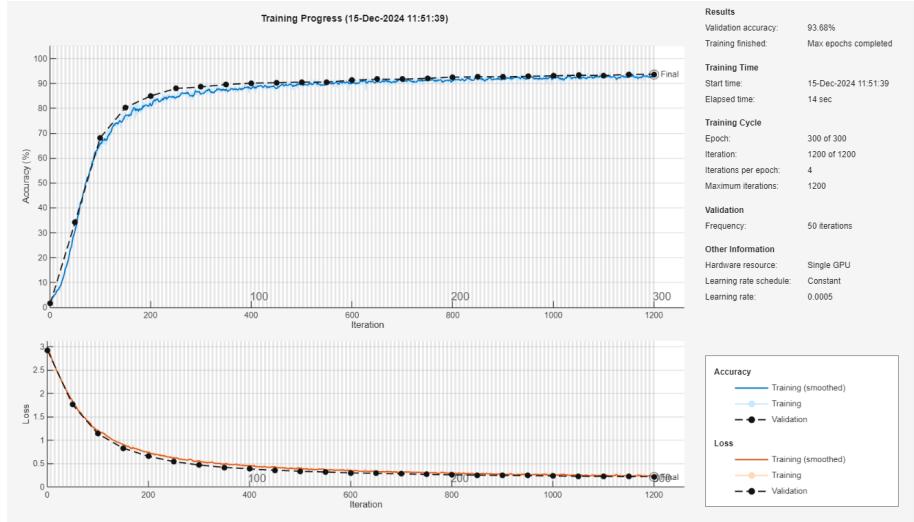


Figura 14: Addestramento con 8 features

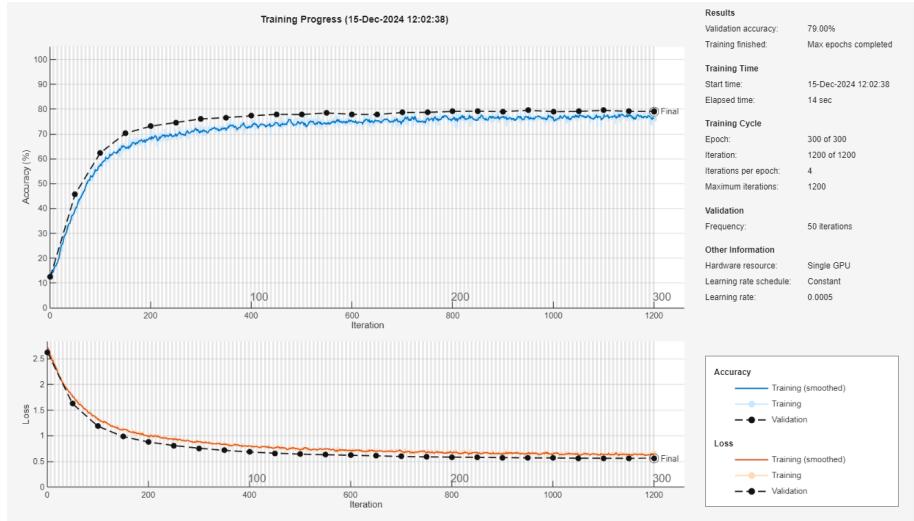


Figura 15: Addestramento con 4 features

Nel primo tentativo di addestramento, usando 32 features, la qualità dell'approssimazione è stata del 98.91%, infatti la rete neurale ha appreso correttamente dai dati e ha raggiunto il 97.86% di accuratezza sui dati di test. Nel secondo tentativo, usando 16 features, la qualità è risultata del 95.24%, e nonostante sia ridotta rispetto al caso precedente, la rete ha totalizzato comunque il 97.95% di accuratezza sui dati di test; risulta quindi evidente che l'informazione

contenuta nei dati di addestramento è ancora più che sufficiente. Nel terzo tentativo, usando 8 features, la qualità è scesa al 89.45%, e l'accuratezza sui dati di test si è arrestata al 93.68%, dal quale possiamo dedurre che la ridotta quantità di informazioni contenuta nei dati di training stia effettivamente iniziando ad intaccare l'addestramento. Tutto ciò risulta molto più evidente selezionando solo 4 features, infatti in questo caso la qualità dell'SVD scende all' 82.26%, e a seguito dell'addestramento l'accuratezza sul test si è ridotta al 79.00%, che è molto più bassa rispetto ai tentativi precedenti. Le ragioni di questo fenomeno si possono proprio individuare nell'ordinamento dei valori singolari operato dall'SVD, infatti si può paragonare il lavoro dell'algoritmo all'ordinamento del contenuto informativo dei dati, in modo tale che, non considerando gli ultimi valori singolari, la quantità di informazione si riduca di una piccola quantità. La drastica diminuzione dell'accuratezza dell'addestramento nell'ultimo tentativo è dovuta al fatto che non sono stati considerati dei valori singolari piuttosto importanti, che hanno portato a ridurre considerevolmente la quantità di informazione nei dati. Nonostante ciò è stato possibile addestrare la rete neurale con un numero di features ridotto di un quarto, senza praticamente intaccare l'accuratezza sui dati di test, il che costituisce in ogni caso un ottimo risultato.

4 Caso di applicazione: computer vision

4.1 Introduzione al dataset e alla rete utilizzata

Per testare gli effetti dell'applicazione di SVD su un dataset di immagini per l'addestramento di una rete neurale convoluzionale, utilizzeremo un problema di classificazione. L'obiettivo che la rete convoluzionale dovrà raggiungere sarà il riconoscimento della presenza di un incendio in una fotografia scattata da varie angolazioni e in differenti condizioni ambientali. A tale scopo è stato utilizzato un dataset di immagini di incendi, prevalentemente boschivi, disponibile online (vedere le fonti). Il dataset presenta due classi: "fire", con 477 immagini, e "no-fire", con 991 immagini. Di seguito un esempio della classe fire e un esempio della classe no-fire:



Figura 16: Immagine di classe fire



Figura 17: Immagine di classe no-fire

Per quanto riguarda la rete neurale scelta, si tratta della ResNet18, una CNN pre-addestrata per problemi di classificazione sul dataset ImageNet. Nel nostro caso d'uso performeremo il cosiddetto transfer learning, cioè bloccheremo l'addestramento di tutti i layer tranne l'ultimo, così da sfruttare tutti i vantaggi dell'addestramento precedente, modificando soltanto il modo in cui le features individuate vengono effettivamente associate alle due classi finali. Eseguiremo l'addestramento prima con le immagini senza applicazione di SVD, successivamente procederemo ad addestrare la rete, senza variare alcun tipo di iperparametro, sulle stesse immagini a cui è stata applicata la tecnica di SVD.

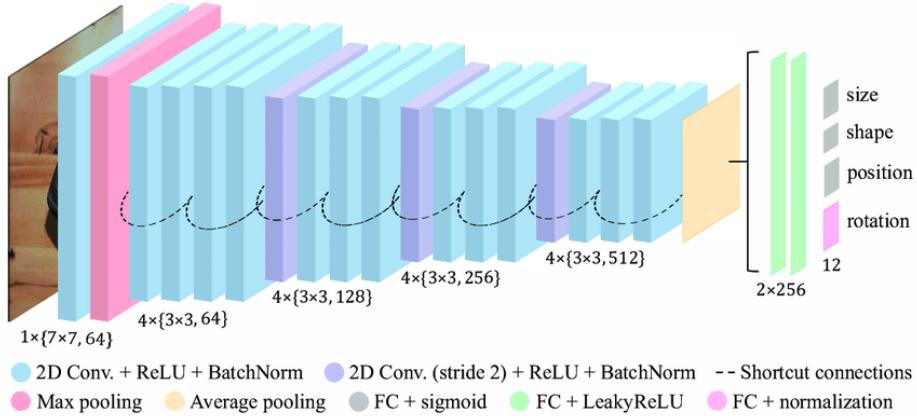


Figura 18: Struttura di ResNet18

4.2 Preprocessing del dataset

Prima di addestrare qualsiasi rete neurale convoluzionale è spesso necessario applicare tecniche di resizing e di augmentation al dataset. La prima problematica da fronteggiare è che la rete ResNet18 ha una dimensione di input fissata a [224, 224, 3], quindi accetta immagini come 3 matrici (canali) di dimensione 224x224. Nel dataset scelto le immagini sono di dimensioni differenti e tipicamente molto più grandi. Per risolvere questo problema è stato necessario effettuare un resizing di ogni immagine del dataset, utilizzando tecniche interpolatorie. Di seguito il codice utilizzato:

```

1 % Creazione di un imageDatastore per il dataset
2 imds = imageDatastore("corrected_wildfires_dataset\",
    IncludeSubfolders=true, LabelSource="foldernames");
3
4 % - "imageDatastore" carica tutte le immagini contenute nella
5 % - cartella "corrected_wildfires_dataset".
6 % - "IncludeSubfolders=true" permette di includere immagini in
7 % - eventuali sottocartelle.
8 % - "LabelSource='foldernames'" assegna etichette alle immagini in
9 % - base al nome delle cartelle in cui si trovano.

```

```

8 % Directory di output
9 outputFireFolder = "fire"; % Cartella per immagini con etichetta
10    "fire"
11 outputNoFireFolder = "nofire"; % Cartella per immagini con
12    etichetta "nofire"
13
14 % Controllo ed eventuale creazione delle cartelle se non esistono
15 if ~exist(outputFireFolder, 'dir')
16    mkdir(outputFireFolder); % Crea la cartella "fire" se non
17        esiste
18 end
19
20 % Reset del contatore dell'ImageDatastore per assicurarsi che la
21    lettura inizi dalla prima immagine
22 reset(imds);
23
24 % Iterazione su tutte le immagini nel datastore
25 while hasdata(imds)
26    % Lettura dell'immagine e delle informazioni associate
27    [img, info] = read(imds);
28    % Determinazione dell'etichetta dell'immagine
29    label = info.Label;
30
31    % Ottenimento del nome del file e dell'estensione
32    [~, filename, ext] = fileparts(info.Filename);
33
34    % Ridimensionamento dell'immagine alla dimensione 224x224 pixel
35    resizedImg = imresize(img, [224, 224]);
36
37    % Salvataggio delle immagini preprocessate nella cartella
38    % corrispondente all'etichetta
39    if label == "nofire"
40        imwrite(resizedImg, fullfile(outputNoFireFolder,
41            [filename, ext])); % Salva nella cartella "nofire"
42    elseif label == "fire"
43        imwrite(resizedImg, fullfile(outputFireFolder, [filename,
44            ext])); % Salva nella cartella "fire"
45    end
46 end

```

In questo caso in realtà è presente un ulteriore problema, ovvero lo sbilanciamento delle due classi, infatti le immagini di classe fire sono 477, a fronte delle immagini di classe no-fire che sono praticamente il doppio. Questo problema tipicamente porta a performance peggiori del classificatore, a seguito dell'addestramento. Per risolvere questo problema si utilizza una tecnica di oversampling, in cui si duplicano randomicamente delle immagini della classe svantaggiata, in modo da rendere bilanciati sia il train set che il test set. Di seguito la procedura utilizzata per importare il dataset e successivamente per bilanciare sia il train set che il test set.

```

1 % Funzione per replicare casualmente i file di un array
2 function files = randReplicateFiles(files, numDesired)

```

```

3     n = numel(files); % Numero di file disponibili
4     ind = randi(n, numDesired, 1); % Genera indici casuali tra 1 e
5         n
6     files = files(ind); % Seleziona i file corrispondenti agli
7         indici generati
8 end
9
10 % Creazione di un imageDatastore per il dataset
11 imds = imageDatastore("resized_dataset\", IncludeSubfolders=true,
12 LabelSource="foldernames");
13
14 % Suddivisione del dataset in train (80%) e test (20%)
15 [imdsTrain, imdsTest] = splitEachLabel(imds, 0.8);
16
17 % Creazione di un istogramma delle etichette nel training set
18 histogram(imdsTrain.Labels);
19
20 % Bilanciamento del training set
21 labels = imdsTrain.Labels; % Ottiene le etichette delle immagini
22     nel training set
23 [G, classes] = findgroups(labels); % Raggruppa le immagini in base
24     all'etichetta
25
26 % Determina il numero di osservazioni (immagini) per ogni classe
27 numObservations = splitapply(@numel, labels, G);
28
29 % Trova il numero massimo di osservazioni tra tutte le classi
30 desiredNumObservationsPerClass = max(numObservations);
31
32 % Per ogni classe, replica casualmente i file per bilanciare le
33     classi
34 files = splitapply(@(x){randReplicateFiles(x,
35     desiredNumObservationsPerClass)}, imdsTrain.Files, G);
36
37 % Combina i file replicati in un singolo array
38 files = vertcat(files{:});
39
40 % Ricalcolo delle etichette per i file replicati
41 labels = []; % Inizializzazione di un array vuoto per le nuove
42     etichette
43 info = strfind(files, '\'); % Trova i separatori di directory per
44     ogni file
45 for i = 1:numel(files)
46     idx = info{i}; % Indici delle occorrenze del separatore per il
47         file corrente
48     dirName = files{i}; % Percorso del file
49     targetStr = dirName(idx(end-1)+1:idx(end)-1); % Nome della
50         cartella di etichettatura
51     targetStr2 = cellstr(targetStr); % Converte la stringa in cell
52         array
53     labels = [labels; categorical(targetStr2)]; % Aggiunge
54         l'etichetta come valore categoriale
55 end
56
57 % Assegna i nuovi file e le relative etichette al training set
58 imdsTrain.Files = files;
59 imdsTrain.Labels = labels;

```

```

47 % Visualizza un istogramma delle etichette nel training set
48 % bilanciato
49 histogram(imdsTrain.Labels);
50
51 %Esegue lo stesso processo per i dati di test
52 labels=imdsTest.Labels;
53 [G,classes] = findgroups(labels);
54 numObservations = splitapply(@numel,labels,G);
55 desiredNumObservationsPerClass = max(numObservations);
56 files = splitapply(@(x){randReplicateFiles(x,...,
57 desiredNumObservationsPerClass)},imdsTest.Files,G);
58 files = vertcat(files{:});
59 labels=[];info=strfind(files,'\'');
60 for i=1:numel(files)
61     idx=info{i};
62     dirName=files{i};
63     targetStr=dirName(idx(end)-1)+1:idx(end)-1];
64     targetStr2=cellstr(targetStr);
65     labels=[labels;categorical(targetStr2)];
66 end
67 imdsTest.Files = files;
68 imdsTest.Labels=labels;
69 histogram(imdsTest.Labels)

```

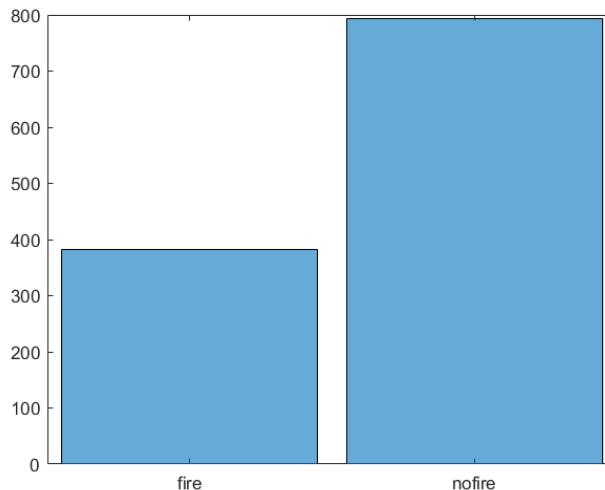


Figura 19: Train set prima dell'oversampling

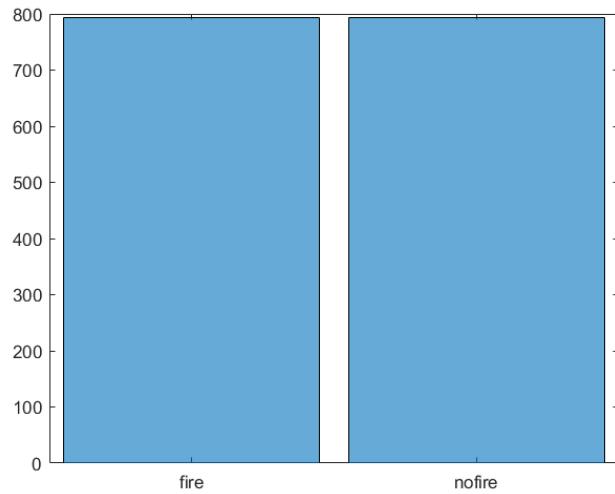


Figura 20: Train set dopo l'oversampling

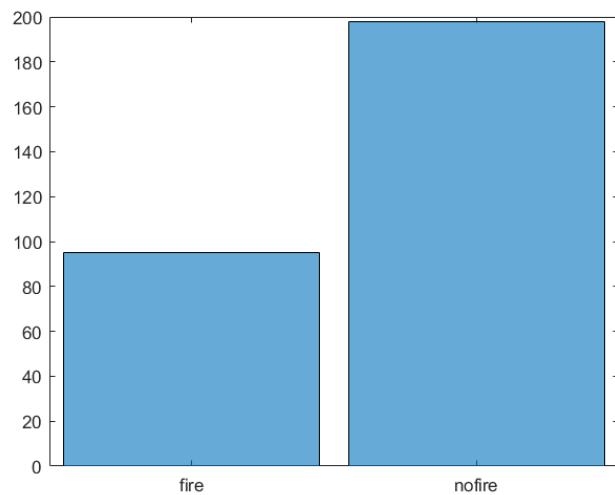


Figura 21: Test set prima dell'oversampling

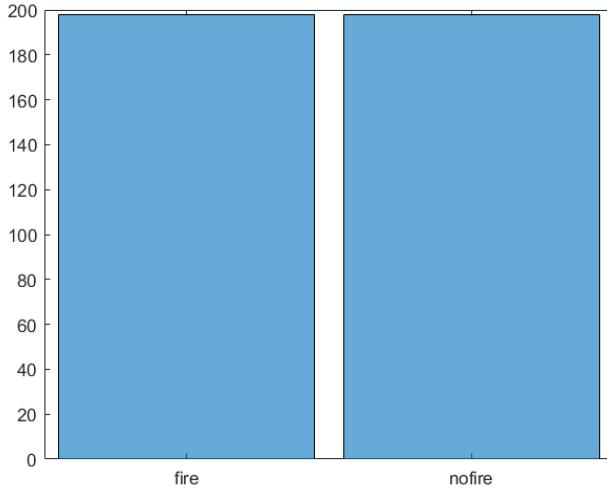


Figura 22: Test set dopo l'oversampling

Infine, per migliorare ulteriormente l'addestramento abbiamo usato delle semplici tecniche di data augmentation sul training set, al fine di migliorare il grado di generalizzazione del modello.

```

1 % Definizione dei parametri per l'augmentation
2 pixelRange = [-30 30]; % Intervallo di traslazione casuale (in
    % pixel) per l'asse X e Y
3 RotationRange = [-30 30]; % Intervallo per rotazioni casuali (in
    % gradi)
4 scaleRange = [0.8 1.2]; % Intervallo per scalatura casuale (tra il
    % 80% e il 120% delle dimensioni originali)
5
6 % Creazione di un oggetto " imageDataAugmenter " per applicare
    % trasformazioni casuali
7 imageDataAugmenter = imageDataAugmenter( ...
8     'RandXReflection', true, ... % Riflesso orizzontale casuale
9     'RandXTranslation', pixelRange, ... % Traslazione casuale
    % lungo l'asse X
10    'RandYTranslation', pixelRange, ... % Traslazione casuale
    % lungo l'asse Y
11    'RandXScale', scaleRange, ... % Scalatura casuale lungo l'asse
    % X
12    'RandYScale', scaleRange, ... % Scalatura casuale lungo l'asse
    % Y
13    'RandRotation', RotationRange ... % Rotazione casuale
);
14
15 % Creazione di un " augmentedImageDatastore " per il training set
16 resizeTrainImgs = augmentedImageDatastore([224 224], imdsTrain,
    % DataAugmentation", imageDataAugmenter); % Il primo parametro
    % rappresenta l'eventuale resizing, ma in questo caso non viene
    % effettuata alcuna operazione, dato che tale tecnica e stata

```

```

    applicata precedentemente, e le immagini hanno la dimensione
    224x224
18
19 % Creazione di un "augmentedImageDatastore" per il test set
20 resizeTestImgs = augmentedImageDatastore([224 224], imdsTest,
    "DataAugmentation", imageAugmenter);

```

4.3 Addestramento di ResNet18

Come già anticipato, è stato applicato il transfer learning, sostituendo l'ultimo layer di ResNet18 con un altro layer di soli due neuroni, così da poter ottenere dai due la probabilità di appartenenza alla rispettiva classe. Di seguito il codice per la configurazione della rete e degli iperparametri, con corrispondente addestramento.

```

1 % Caricamento del modello ResNet-18 pre-addestrato
2 net = resnet18;
3
4 % Determinazione della dimensione di input del modello
5 inputSize = net.Layers(1).InputSize;
6
7 % Creazione di un grafo dei layer a partire dal modello
    pre-addestrato
8 lgraph = layerGraph(net);
9
10 learnableLayer = 'fc1000'; % Layer responsabile della
    classificazione finale
11
12 % Nome del layer di classificazione originale
13 classLayer = 'ClassificationLayer_predictions';
14
15 % Determinazione del numero di classi nel dataset
16 numClasses = numel(categories(imds.Labels));
17
18 % Creazione di un nuovo layer completamente connesso (fully
    connected)
19 newLearnableLayer = fullyConnectedLayer(numClasses, ...
    'Name','new_fc', ... % Nome del nuovo layer
    'WeightLearnRateFactor',10, ... % Fattore di apprendimento
        per i pesi (più rapido rispetto ai layer precedenti)
    'BiasLearnRateFactor',10); % Fattore di apprendimento per
        i bias (più rapido)
20
21 % Sostituzione del layer completamente connesso nel grafo dei layer
22 lgraph = replaceLayer(lgraph, learnableLayer, newLearnableLayer);
23
24 % Creazione di un nuovo layer di classificazione
25 newClassLayer = classificationLayer('Name','new_classoutput');
26
27 % Sostituzione del layer di classificazione nel grafo dei layer
28 lgraph = replaceLayer(lgraph, classLayer, newClassLayer);
29
30 %Impostazione degli iperparametri
31 miniBatchSize = 64; % Dimensione del mini-batch (64 immagini per
    aggiornamento dei pesi)

```

```

35 options = trainingOptions('sgdm', ...
36     'MiniBatchSize', miniBatchSize, ...
37     'MaxEpochs', 8, ...
38     'InitialLearnRate', 1e-4, ...
39     'Shuffle', 'every-epoch', ...
40     'Verbose', false, ...
41     'Plots', 'training-progress');
42
43 % Addestramento del modello
44 net = trainNetwork(resizeTrainImgs, lgraph, options);
45 save("net.mat", "net")

```

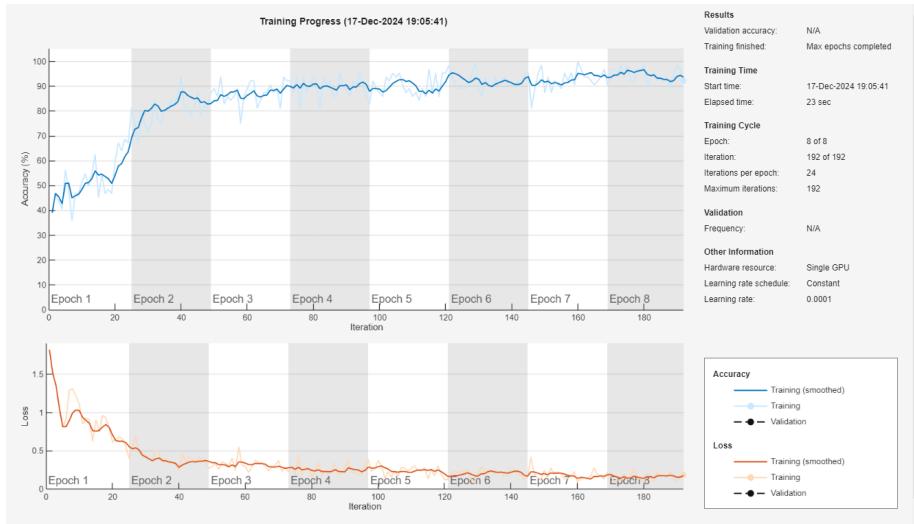


Figura 23: Training di ResNet18 con dataset senza SVD

4.4 Valutazione del modello

Per la valutazione del modello risultante, dato che si tratta di un classificatore, è stata utilizzata la matrice di confusione, prodotta sulla base dei risultati del modello sul test set. Di seguito codice e risultati.

```

1 % Classificazione delle immagini del test set
2 [YPred, probs] = classify(net, resizeTestImgs);
3
4 %Calcolo dell'accuratezza
5 accuracy = mean(YPred == imdsTest.Labels);
6
7 %Visualizzazione della Confusion Matrix
8 YTrue = imdsTest.Labels; % Etichette reali del test set
9 figure;
10 cm = confusionchart(YTrue, YPred);

```

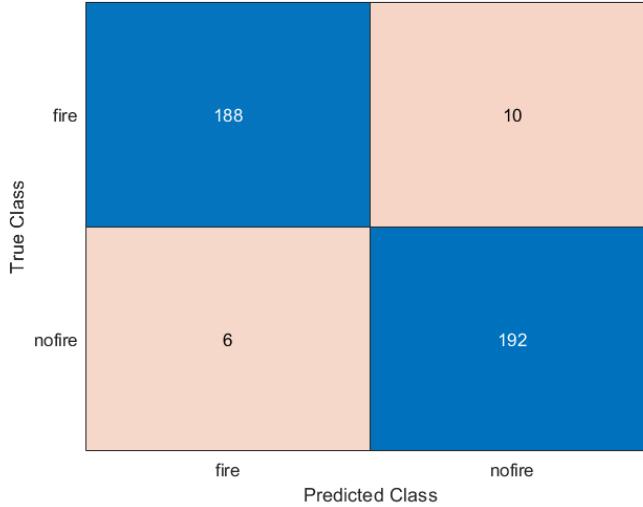


Figura 24: Matrice di confusione risultante senza SVD

In questo caso, l'accuratezza è risultata del 95.96%, che in generale è un ottimo risultato. Da notare però che un addestramento di questo tipo è un processo stocastico e, ad ogni tentativo, i risultati variano. Ad ogni modo con le immagini senza applicazione di SVD l'accuratezza è sempre risultata maggiore del 90%, tipicamente assestandosi su valori vicini al 92%. Per mostrare le feature prelevate dalla rete neurale, consideriamo un esempio di classe fire, e le sue corrispondenti attivazioni (cioè le immagini in uscita dai filtri convoluzionali, che identificano le features considerate dal modello).



Figura 25: Immagine di classe fire, correttamente classificata dal modello

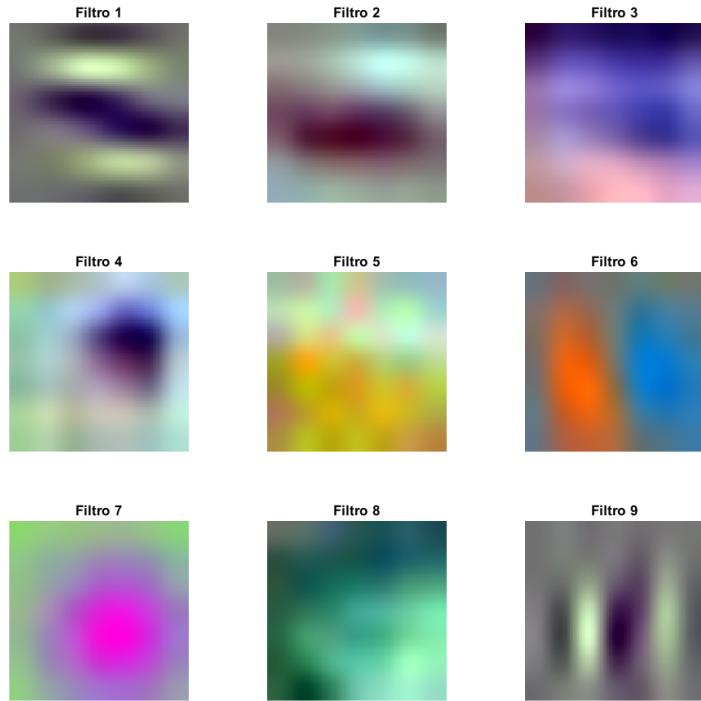


Figura 26: I primi 9 filtri del primo livello convoluzionale di ResNet18

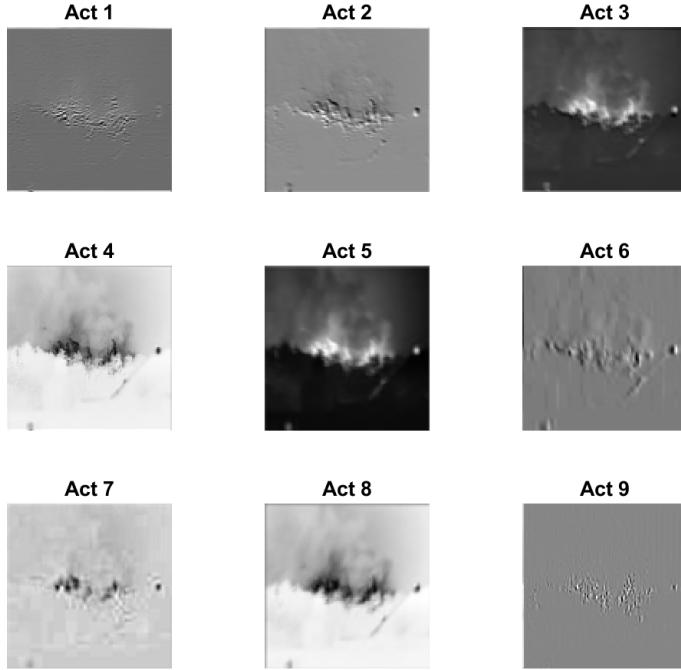


Figura 27: Attivazioni in uscita dai primi 9 filtri del primo livello convoluzionale di ResNet18

4.5 Applicazione dell'SVD al dataset

Per applicare la tecnica di compressione tramite SVD alle immagini del dataset si scomponete ogni immagine nei suoi 3 canali (R, G, B) che sono delle matrici 224x224, dopodiché di ogni canale viene calcolata l'SVD. Date le matrici U, Σ e V si selezionano le prime k colonne di U e di V e si estrae la sottomatrice $k \times k$ dalla S (selezionando così i primi k valori singolari), a questo punto si hanno a disposizione le matrici U_k, S_k, V_k , dal quale possiamo ottenere la ricostruzione approssimata (in questo caso compressa) del canale originale, come $\tilde{A} = U_k \Sigma_k V_k^T$. Una volta eseguita questa operazione per i 3 canali, questi vengono nuovamente uniti a formare una immagine in formato RGB. Di seguito il codice che esegue quanto descritto.

```

1 % Funzione per comprimere l'immagine utilizzando la Decomposizione
2 % ai Valori Singolari (SVD)
3 function compressedImg = compressWithSVD(img, k)
4
5 % Converte l'immagine in double
6 img = im2double(img);

```

```

6 % Ottiene il numero di righe e colonne dell'immagine
7 [rows, cols, ~] = size(img);
8
9 % Crea un'immagine vuota che conterra l'immagine compressa
10 compressedImg = gpuArray.zeros(rows, cols, 3);
11
12 % Itera sui 3 canali di colore (R, G, B)
13 for c = 1:3
14     % Estrae il canale corrente e lo trasferisce sulla GPU per
15     % elaborazione parallela
16     channel = gpuArray(img(:, :, c));
17
18     % Calcola la Decomposizione ai Valori Singolari (SVD) per
19     % il canale corrente
20     [U, S, V] = svd_algorithm(channel);
21
22     % Seleziona i primi k valori singolari (approssimazione)
23     U_k = U(:, 1:k);
24     S_k = S(1:k, 1:k);
25     V_k = V(:, 1:k);
26
27     % Ricostuisce il canale utilizzando solo i primi k valori
28     % singolari
29     channel = U_k * S_k * V_k';
30
31     % Assegna il canale ricostruito all'immagine compressa
32     compressedImg(:, :, c) = channel;
33 end
34
35 % Converte l'immagine compressa in formato uint8 e la
36 % trasferisce dalla GPU alla CPU
37 compressedImg = gather(im2uint8(compressedImg));
38
39 % Crea un oggetto imageDatastore per caricare il dataset delle
40 % immagini
41 imds = imageDatastore("resized_dataset\", IncludeSubfolders=true,
42 LabelSource="foldernames");
43
44 % Definisce le cartelle per salvare le immagini compressi in base
45 % alla loro etichetta
46 outputFireFolder = "fire";
47 outputNoFireFolder = "nofire";
48
49 % Crea la cartella "fire" se non esiste
50 if ~exist(outputFireFolder, 'dir')
51     mkdir(outputFireFolder);
52 end
53
54 % Crea la cartella "nofire" se non esiste
55 if ~exist(outputNoFireFolder, 'dir')
56     mkdir(outputNoFireFolder);
57 end
58
59 % Ripristina il puntatore all'inizio del dataset
60 reset(imds);

```

```

56
57 % Ciclo per elaborare tutte le immagini nel datastore
58 while hasdata(imds)
59     % Legge un'immagine e le relative informazioni
60     [img, info] = read(imds);
61
62     % Ottiene l'etichetta dell'immagine (fire onofire)
63     label = info.Label;
64
65     % Estrae il nome del file e l'estensione
66     [~, filename, ext] = fileparts(info.Filename);
67
68     % Comprime l'immagine utilizzando la funzione compressWithSVD
69     % con k=112
70     compressedImg = compressWithSVD(img, 112);
71
72     % Salva l'immagine compressa nella cartella appropriata in
73     % base all'etichetta
74     if label == "nofire"
75         imwrite(compressedImg, fullfile(outputNoFireFolder,
76                                         [filename, ext]));
77     elseif label == "fire"
78         imwrite(compressedImg, fullfile(outputFireFolder,
79                                         [filename, ext]));
80     end
81 end

```

Tramite questo algoritmo è possibile comprimere il dataset di immagini 224x224 applicando SVD. Di seguito il confronto tra immagini con differenti rank di approssimazione.



Figura 28: Considerando 112 valori singolari

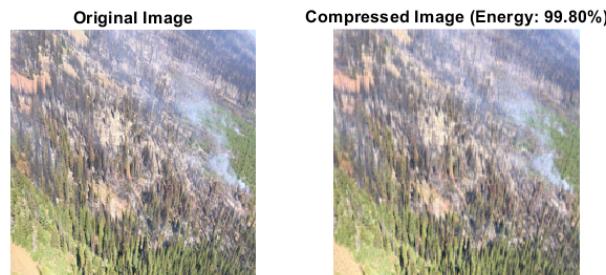


Figura 29: Considerando 56 valori singolari



Figura 30: Considerando 28 valori singolari

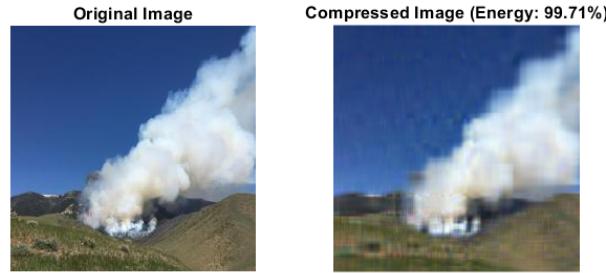


Figura 31: Considerando 14 valori singolari

Risulta importante notare come, nonostante l'applicazione di un'approssimazione anche molto invasiva (fino a soli 14 valori singolari a partire da 224), il contenuto informativo resta assolutamente elevatissimo, con una diminuzione quasi trascurabile, a fronte di un evidente cambiamento dell'immagine dal punto di vista visivo. Il fenomeno è ben visibile, analizzando l'andamento dei valori singolari tipico registrato.

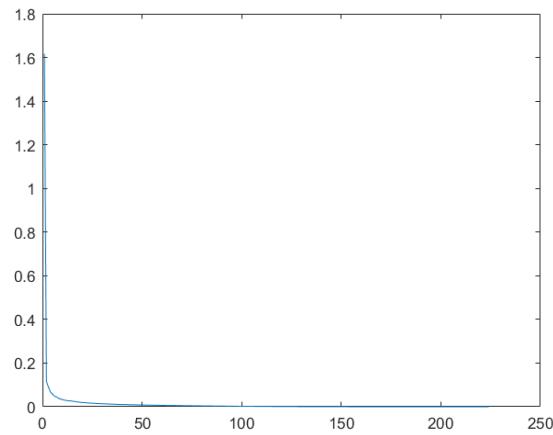


Figura 32: Andamento dei valori singolari tipico delle immagini del dataset

4.6 Addestramento della ResNet18 con immagini compresse

Utilizzando la rete vista in precedenza, senza alcuna variazione degli iperparametri, si procede adesso con la valutazione dei modelli addestrati con i dataset compressi da SVD, a vari livelli di approssimazione. Di seguito i risultati degli addestramenti.



Figura 33: Addestramento con dataset approssimato con 112 valori singolari

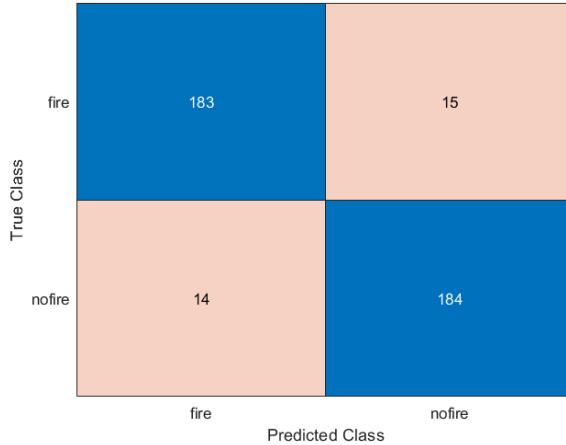


Figura 34: Matrice di confusione con 112 valori singolari



Figura 35: Addestramento con dataset approssimato con 56 valori singolari

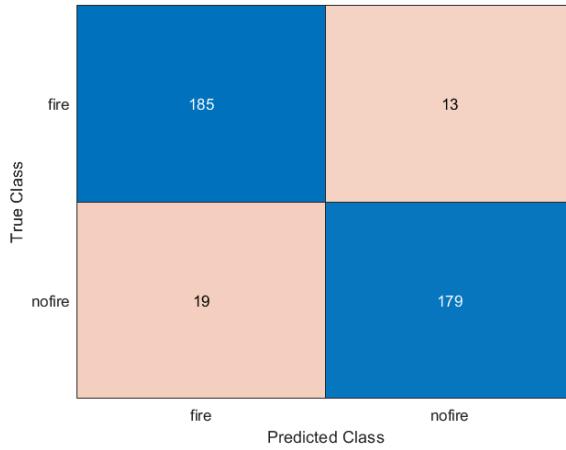


Figura 36: Matrice di confusione con 56 valori singolari



Figura 37: Addestramento con dataset approssimato con 28 valori singolari

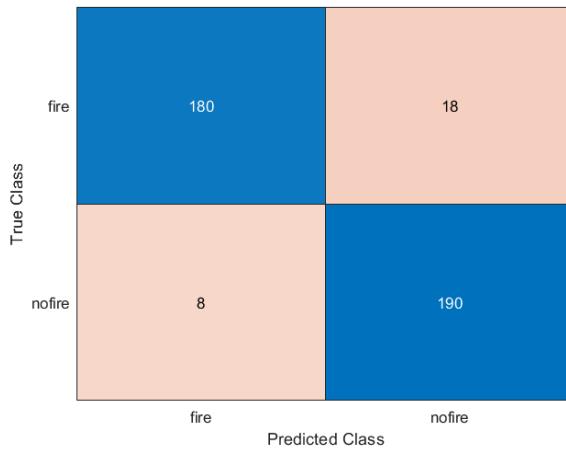


Figura 38: Matrice di confusione con 28 valori singolari

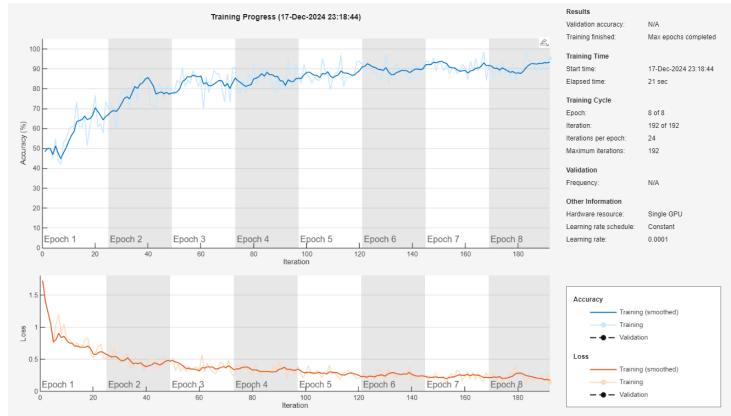


Figura 39: Addestramento con dataset approssimato con 14 valori singolari

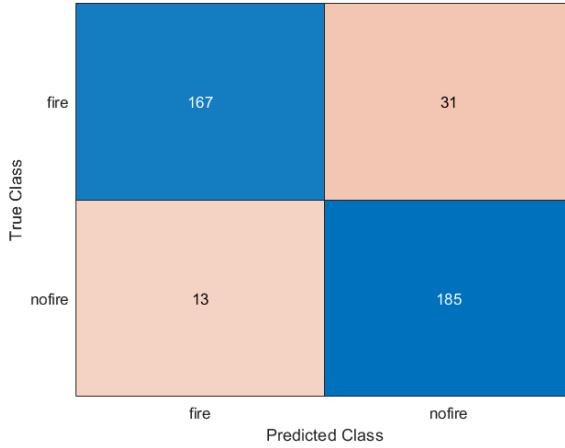


Figura 40: Matrice di confusione con 14 valori singolari

Come è possibile notare dagli addestramenti, la rete neurale, nonostante la compressione, è sempre riuscita a identificare i pattern tra le features individuate dai livelli convoluzionali, mostrando solo un lieve calo nei risultati dell'addestramento con la compressione a 14 autovvalori(che è l'unico a scendere sotto il 90% di accuratezza sul test set). Questo risultato è in accordo con i livelli di energia elevatissimi mantenuti dalle immagini a seguito dell'applicazione di SVD.

5 Conclusioni e considerazioni

Considerati i test effettuati sui dati con e senza applicazione di SVD, e considerati i risultati ottenuti, in generale possiamo affermare che la tecnica SVD risulta un validissimo strumento per la riduzione dello spazio necessario alla memorizzazione dei grandi dataset. In particolare, le immagini, che sono tipicamente più problematiche in quanto a spazio necessario per la memorizzazione, beneficiano moltissimo dell'applicazione di questa tecnica. Sarebbe possibile infatti memorizzare le matrici U, Σ, V , invece che l'immagine nella sua dimensione originale. Ad esempio, nel caso di applicazione analizzato, invece di memorizzare 3 canali di dimensione 224x224, cioè $3 \times 224 \times 224 = 150528$ valori, potremmo memorizzare le tre matrici risultanti dalla scomposizione dei tre canali: ipotizzando di selezionare 28 valori singolari, che comunque hanno permesso di ottenere ottimi risultati nella classificazione, avremmo $3 \times (224 \times 28 + 28 \times 28 + 224 \times 28) = 39984$ valori, cioè il 26.56% rispetto alla memorizzazione dell'intera immagine. Questo livello di compressione porterebbe ad una riduzione della necessità di un grande numero di data-center per il mantenimento di queste grandi quantità di dati, permettendo comunque di ottenere un validissimo risultato in termini di addestramento di modelli per tutti i possibili scopi.

6 Fonti

1. Dataset per addestramento della rete neurale su dati numerici: <https://archive.ics.uci.edu/dataset/80/optical+recognition+of+handwritten+digits>
2. Dataset per addestramento della rete convoluzionale: <https://www.kaggle.com/datasets/johanjohnthomas/corrected-wildfires>
3. Reti neurali: <https://www.deeplearningbook.org/>.
4. Oversampling: <https://it.mathworks.com/matlabcentral/fileexchange/78020-oversampling-for-deep-learning-classification-example>