Università degli Studi di Padova

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in
Ingegneria Informatica

Confronto tra la Compressione con Quantizzazione, Machine Learning con reti lineari e non lineari

Relatore:
Prof. Stefano Tomasin

Laureando:
Marco Martinelli
1218666

Correlatore:
Dott. Alberto Rech

Anno Accademico: 2021/2022
Data di laurea: 20 settembre 2022

Abstract

In questa tesi viene affrontato il tema della compressione dati con perdite, effettuata attraverso l'algoritmo di Linde-Buzo-Gray (LBG) e l'autoencoder (AE).
I due metodi sono prima stati analizzati analiticamente in termini di efficienza di compressione, complessità computazionale e occupazione di memoria.
In seguito, i due metodi sono stati testati su due dataset, per andare a valutare la qualità della compressione e le regioni di decisione create.
É emerso che, a parità di bit usati per la compressione, l'autoencoder effettua una ricostruzione equivalente o migliore dell'algoritmo LBG, risultando vantaggioso anche in termini di occupazione di memoria e complessità computazionale.
Si nota inoltre che, nonostante la componente di non linearità introdotta dal quantizzatore, in generale un autoencoder lineare riesce ad ottenere una qualità di compressione equivalente a quella di autoencoder non lineari.

Indice

Acronyms

AE autoencoder. i, 4, 6, 7, 9, 12, 13, 15, 23–31, 37–39, 41–43, 45–49, 51, 56–59, 61–63, 66–69, 71–73, 75

DSQ differentiable soft quantization. 8, 9, 12–14, 29–31, 45, 51, 65, 73

LBG Linde-Buzo-Gray. i, 4–6, 18, 22, 23, 27–31, 46–48, 51, 56, 66–68, 71, 72, 74, 75

MSE mean-squared error. 4, 7, 9, 11, 14, 17, 18, 27, 29–31, 33–36, 45, 47, 48, 51, 56, 65, 67, 68, 71, 73

Capitolo 1

Compressione dei dati

La compressione dei dati è una tecnica che consente di ridurre la quantità di bit necessari per rappresentare una qualche informazione.
Considerando un generico vettore $\boldsymbol{x}$ di numeri reali, esistono due tipologie di compressione:

1. senza perdite (*lossless*);
2. con perdite (*lossy*).

Definiamo le funzioni *f(.)* e *g(.)* che effettuano rispettivamente la compressione e la decompressione di un generico vettore di numeri reali.

$$\boldsymbol{y} = f(\boldsymbol{x}) \qquad \text{funzione di compressione,} \tag{1.1}$$

$$\boldsymbol{x'} = g(\boldsymbol{y}) = g(f(\boldsymbol{x})) \qquad \text{funzione di decompressione.} \tag{1.2}$$

Nel caso di una compressione *lossless*, è possibile ricostruire esattamente la versione originale di un vettore compresso, dato che questa conserva tutta l'informazione contenuta ed è un processo completamente reversibile.
In una compressione di tipo *lossy*, invece, una parte dell'informazione del vettore originale viene persa permanentemente, rendendo questo processo di compressione irreversibile, ovvero in generale:

$$\boldsymbol{x'} = g(f(\boldsymbol{x})) \neq \boldsymbol{x} \tag{1.3}$$

Risulta pertanto impossibile riuscire a ricostruire esattamente il vettore originale. L'obiettivo di una compressione di questo tipo sarà minimizzare la distorsione tra il vettore in input e il vettore ricostruito con la decompressione. Nel corso della

tesi verranno considerate sempre compressioni di tipo *lossy*.
In particolare, si vuole trovare una funzione di compressione *f(.)* che riesca a mappare un vettore

$$\boldsymbol{\nu} = [\nu_1, ..., \nu_n] \tag{1.4}$$

di dimensione n in una sequenza $\boldsymbol{b(\nu)}$ di pochi bit, e una funzione di decompressione *g(.)* che, data una sequenza $\boldsymbol{b}$ riesca a risalire ad un vettore $\boldsymbol{\nu'}$ quanto più simile a quello di partenza.
Per misurare la distorsione tra il vettore in ingresso e il vettore ricostruito verrà usata come come metrica il mean-squared error (MSE), definito tra due vettori $\boldsymbol{\nu_1}$ e $\boldsymbol{\nu_2}$ come segue:

$$d(\boldsymbol{\nu_1}, \boldsymbol{\nu_2}) = \frac{1}{n}\sum_{i=1}^{n}(\nu_{1,i} - \nu_{2,i})^2 \tag{1.5}$$

Nella tesi verranno confrontati due metodi che permettono di modellare il sistema di compressione e decompressione descritto sopra:

1. l'algoritmo di Linde-Buzo-Gray;
2. l'autoencoder.

In entrambi i casi le funzioni di compressione *f(.)* e decompressione *g(.)* sono ottenute con una procedura di apprendimento effettuata su un training set T di N vettori.

1.1 Algoritmo di Linde-Buzo-Gray

L'algoritmo di Linde-Buzo-Gray è un algoritmo iterativo che genera un codebook $C = \{\boldsymbol{c}_1, ..., \boldsymbol{c}_K\}$ a partire da un training set T. Con *codebook* si intende un insieme K di vettori che vengono usati, una volta terminato l'addestramento, per determinare le regioni di decisione relative ai vettori contenuti in T.
L'algoritmo LBG opera due fasi sui vettori del codebook, una di ottimizzazione e una di divisione.
Partendo da un codebook vuoto $C_0 = \emptyset$, si alternano fasi di ottimizzazione a fasi di divisione, fino a quando il numero di vettori (e conseguentemente di regioni) all'interno del codebook è uguale a K, che deve essere potenza intera di 2.
La codifica viene effettuata come segue:

$$y = f(\boldsymbol{x}) = \underset{i}{\operatorname{argmin}}\, d(\boldsymbol{c}_i, \boldsymbol{x}) \tag{1.6}$$

Per quanto riguarda invece la decodifica, è sufficiente calcolare:

$$\boldsymbol{x'} = \boldsymbol{c}_y \tag{1.7}$$

1.1.1 Fase di ottimizzazione

La fase di ottimizzazione si compone di più iterazioni, identificate con l'indice j inizialmente posto a 0, le quali hanno l'obiettivo di migliorare progressivamente il codebook C fino a portarlo a convergenza.
Indichiamo con $C_{i,0}$ il codebook di partenza risultante dall'antecedente fase di divisione.
È possibile dividere a sua volta la fase di ottimizzazione in altre tre sottofasi:

1. calcolo delle regioni ottime rispetto al codebook;
2. calcolo della distorsione media sull'intero insieme T;
3. calcolo dell'errore relativo tra l'iterazione corrente e quella precedente.

Calcolo delle regioni ottime rispetto al codebook

La regola applicata per il calcolo delle regioni ottime rispetto ai vettori contenuti nel codebook $C_{i,j}$, dove l'indice i rappresenta il numero totale di iterazioni dell'algoritmo LBG svolte mentre l'indice j rappresenta il numero totale di iterazioni svolte nella corrente fase di ottimizzazione, è la seguente:

$$R_{k,i,j} = \{\boldsymbol{\nu} \in T \,|\, d(\boldsymbol{\nu}, \boldsymbol{c}_{k,i,j} = \min_{\boldsymbol{c_l} \in C_{i,j}} d(\boldsymbol{\nu}, \boldsymbol{c}_l)\},\ k = 1, ..., 2^i \tag{1.8}$$

Calcolo della distorsione media sull'intero insieme T

Una volta calcolate le regioni ottime, si calcola la distorsione media sull'intero insieme:

$$D_{i,j} = \frac{1}{N} \sum_{k=1}^{2^i} \sum_{\boldsymbol{\nu} \in R_{k,i,j}} d(\boldsymbol{\nu}, \boldsymbol{c}_{k,i,j}) \tag{1.9}$$

Calcolo dell'errore relativo tra l'iterazione corrente e quella precedente

Si calcola l'errore relativo tra l'iterazione corrente e la precedente come:

$$\varepsilon_{\text{rel}} = \frac{D_{i,j-1} - D_{i,j}}{D_{i,j}} \tag{1.10}$$

A questo punto, per verificare se il codebook è arrivato a convergenza, si verifica se l'errore relativo è minore di un certo valore di soglia δ:

$$\varepsilon_{\text{rel}} < \delta \tag{1.11}$$

Se la relazione (1.11) è verificata e il numero di vettori all'interno del codebook è uguale a K, allora si termina l'algoritmo, altrimenti si procede operando un'ulteriore fase di divisione.
Se invece la relazione (1.11) non è verificata, si opera un'altra iterazione della fase di ottimizzazione, incrementando il valore dell'indice j.
In questo caso il codebook $C_{i,j+1}$ è composto dai centroidi $\boldsymbol{c}_{k,i,j}$ delle regioni $R_{k,i,j}$ calcolati attraverso la formula:

$$\boldsymbol{c}_{k,i,j+1} = \frac{1}{|R_{k,i,j}|} \sum_{\boldsymbol{\nu} \in R_{k,i,j}} \boldsymbol{\nu}\,, \quad k = 1, ..., 2^i, \tag{1.12}$$

dove $|R_{k,i,j}|$ rappresenta il numero di vettori compresi nella k-esima regione alla i-esima iterazione dell'algoritmo LBG nella j-esima iterazione della fase di ottimizzazione.

1.1.2 Fase di divisione

La fase di divisione viene operata quando il codebook arriva a convergenza (1.11) ma non contiene ancora il numero K di vettori desiderato.

$$C_{i+1} = \{C_i^+\} \cup \{C_i^-\}, \tag{1.13}$$

dove

$$\begin{cases} C_i^+ = C_i + \boldsymbol{\varepsilon}_+ \\ C_i^- = C_i + \boldsymbol{\varepsilon}_- \end{cases} \tag{1.14}$$

I vettori costanti $\boldsymbol{\varepsilon}_+$ e $\boldsymbol{\varepsilon}_-$ spostano i vettori del codebook C_i in modo tale da dividere ulteriormente le regioni e calcolarne di nuove.
Alla fine di questa fase si ottiene quindi un codebook C_{i+1} di dimensione 2^{i+1}.

1.2 Autoencoder

Con autoencoder si intende una rete neurale feed-forward completamente connessa, qui usata con lo stesso scopo dell'algoritmo LBG: riuscire a codificare un

vettore e ricostruirlo minimizzando le perdite (in termini di mean-squared error). Per questa ragione, nel training set che viene utilizzato per l'addestramento della rete, l'input fornito coincide con il target (output desiderato). La rete dovrà imparare, durante l'addestramento, sia a comprimere sia a decomprimere i vettori in maniera efficiente. In altre parole, dovrà sia imparare a rappresentare l'input $\boldsymbol{x}$ in uno spazio a dimensionalità inferiore rispetto a quella iniziale, sia imparare a decomprimere il vettore codificato $\boldsymbol{y}$ ricostruendo un $\boldsymbol{x'}$ che abbia un MSE rispetto al vettore in input $\boldsymbol{x}$ quanto più piccolo possibile.
È possibile vedere un autoencoder come diviso in due parti:

1. Encoder: $\boldsymbol{y} = f(\boldsymbol{x})$, che si occupa della compressione dell'input, cioè del passaggio da n dimensioni del vettore $\boldsymbol{x}$ in input a m dimensioni del vettore $\boldsymbol{y}$ codificato, con $m < n$.

2. Decoder: $\boldsymbol{x'} = g(\boldsymbol{y})$, che effettua la decompressione del vettore codificato dall'encoder, cioè del passaggio da m dimensioni del vettore $\boldsymbol{y}$ codificato a n dimensioni del vettore $\boldsymbol{x}$ in input, con $n > m$.

Il parametro m, che rappresenta il numero di dimensioni del vettore $\boldsymbol{y}$ codificato, regola la qualità della compressione e indica che le componenti del vettore possono assumere soltanto 2^m valori.
Effettuare una riduzione della dimensione del vettore tutela anche dal rischio che la rete neurale, anzichè derivare delle proprietà dall'input per minimizzare il MSE tra uscita e ingresso, impari semplicemente a copiare l'input, vista la struttura del train-set [1].

1.2.1 Struttura della rete neurale di un autoencoder

In questa tesi verrà utilizzata, come architettura della rete neurale, una struttura simmetrica rispetto al layer centrale, ossia il layer che in un certo senso "segna il confine" tra encoder e decoder, nel quale viene applicata la funzione di quantizzazione.
Definendo L come il numero di layer della rete, i primi $\frac{L+1}{2}$ layer compongono l'encoder, mentre i rimanenti $\frac{L-1}{2}$ formano il decoder.
Relativamente alla funzione di quantizzazione citata sopra, con una rete neurale risulta difficile gestire una quantizzazione uniforme: dato che questa funzione, essendo definita a tratti, presenta punti di non derivabilità che ostacolano la convergenza della rete durante l'addestramento.

Per ovviare a questo problema è stata utilizzata un'approssimazione continua e derivabile del quantizzatore uniforme, che sfrutta la differentiable soft quantization. [2].

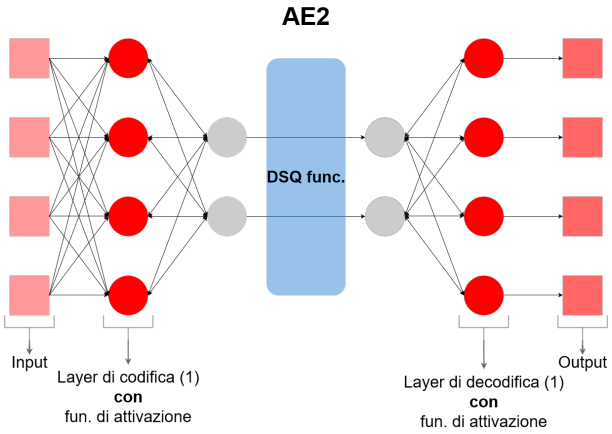


Figura 1.1: Struttura di un autoencoder a 2 layer

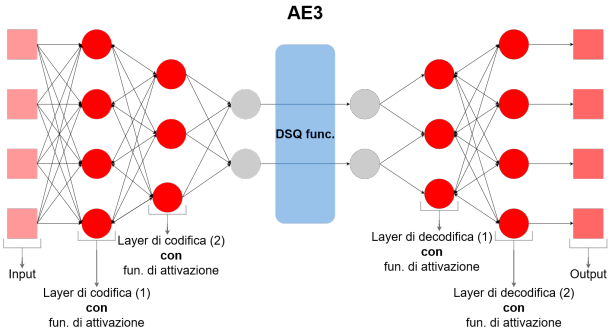


Figura 1.2: Struttura di un autoencoder a 3 layer

1.2.2 Autoencoder single layer/lineare

Con autoencoder lineare si intende un autoencoder con un singolo layer di encoding e un singolo layer di decoding, tra i quali non abbiamo una funzione di attivazione.
L'autoencoder lineare ha la stessa struttura di un deep autoencoder a 2 layer, ma senza funzione di attivazione tra i 2 layer.

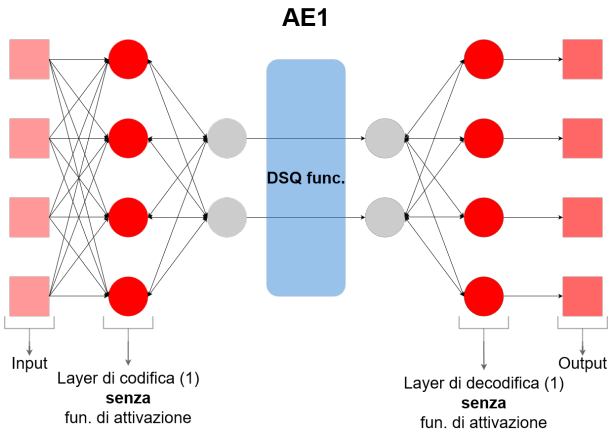


Figura 1.3: Struttura di un autoencoder lineare

Come dimostrato nella pubblicazione [3], nel caso non vi sia una funzione di quantizzazione tra encoder e decoder, in generale un autoencoder single layer funziona meglio di un deep autoencoder, ossia un autoencoder con funzione di attivazione tra i vari layer, in termini di minimizzazione del MSE, occupazione di memoria e complessità computazionale.
Quello che verrà studiato è se queste considerazioni rimangono valide anche andando ad inserire una funzione di quantizzazione DSQ tra encoder e decoder, che introduce l'unica componente di non linearità nella rete di un autoencoder single layer.

1.2.3 Addestramento

L'addestramento di una rete neurale feed-forward completamente connessa si compone di due passi:

1. *Forward propagation*: usato per propagare un tensore in avanti, dal layer l al layer $l + 1$. Questo passo viene usato sia durante l'addestramento sia durante l'impiego della rete neurale addestrata.

2. *Backward propagation*: usato per propagare un tensore all'indietro, dal layer l al layer l-1. Questo passo viene usato soltanto durante l'addestramento.

Per comprendere facilmente questi due passi, adottiamo una notazione matriciale per la rete neurale [4]:

$\boldsymbol{X}_l$	Matrice di input del layer l
$\boldsymbol{Y}_l$	Matrice dei prodotti scalari tra i layer l e $l+1$
$\boldsymbol{Z}_l$	Matrice di output del layer l
$\boldsymbol{G}$	Matrice che rappresenta l'output desiderato della rete
$\boldsymbol{W}_l$	Matrice dei pesi tra i layer l e $l+1$

Forward Propagation

Il passo di forward propagation viene usato per propagare in avanti l'input partendo dal layer di input fino ad arrivare al layer di output della rete. Più semplicemente, questo passo consente l'elaborazione dell'input da parte della rete.
Consideriamo due layer consecutivi l e $l+1$ di un autoencoder. I pesi tra i due layer sono rappresentati dalla matrice $\boldsymbol{W}_l$ e l'input del layer $l+1$ è rappresentato dalla matrice $\boldsymbol{X}_{l+1}$. Il passo di forward propagation consiste quindi nelle operazioni di:

- Applicazione dei pesi alla matrice di input:

$$\boldsymbol{Y}_{l+1} = \boldsymbol{W}_l \boldsymbol{X}_{l+1}, \tag{1.15}$$

- Applicazione della funzione di attivazione α per ottenere l'output del layer $l+1$:

$$\boldsymbol{Z}_{l+1} = \alpha(\boldsymbol{Y}_{l+1}), \tag{1.16}$$

Queste due operazioni vengono eseguite sequenzialmente per ogni coppia di layer consecutivi della rete.

Backward propagation

Il passo di backward propagation è il processo simmetrico rispetto a quello visto sopra di forward propagation. Con questo passo la rete, basandosi sull'errore sull'output prodotto dal passo di forward propagation, affina i pesi delle matrici $\boldsymbol{W}_l$ per andare a minimizzare l'errore relativamente alla metrica di distorsione adottata, nel nostro caso mean-squared error.
Mentre nella forward propagation si propagava l'input in avanti fino ad arrivare al layer di output, nella backward propagation si propaga l'errore sull'output partendo dal layer di output e terminando al layer di input.
Anche questa fase può essere scomposta in tre parti:

1. calcolo dell'errore rispetto all'output desiderato;
2. calcolo delle variazioni dell'errore;
3. aggiornamento dei pesi.

In maniera analoga a quanto fatto per l'analisi del passo di forward propagation, andiamo a considerare due layer consecutivi *l* e *l+1*, dove il layer *l+1* è l'ultimo layer della rete.
L'**errore rispetto all'output desiderato** si calcola come:

$$\boldsymbol{E}_{l+1} = \alpha'(\boldsymbol{Y}_{l+1}) \odot (\boldsymbol{Y}_{l+1} - \boldsymbol{G}) \tag{1.17}$$

dove $\alpha'(\boldsymbol{Y}_{l+1})$ indica il gradiente dei prodotti scalari al layer *l+1* e il simbolo $\odot$ indica la moltiplicazione elemento per elemento.
Nell'ipotesi in cui invece il layer *l+1* sia un layer interno, allora si può dire con certezza che è seguito da un layer *l+2*. In questo caso si calcola l'errore rispetto all'output desiderato come:

$$\boldsymbol{E}_{l+1} = \alpha'(\boldsymbol{Y}_{l+1}) \odot (\boldsymbol{W}_{l+1}\boldsymbol{E}_{l+2}) \tag{1.18}$$

Questo errore viene poi propagato all'indietro, quindi al layer *l+2*.
Nel calcolo della variazione dell'errore e nell'aggiornamento dei pesi non vi sono differenze a seconda che il layer considerato sia interno o esterno.
Considerati quindi due layer consecutivi qualsiasi *l* e *l+1*, definiamo le **variazioni dell'errore** come:

$$\boldsymbol{D}_l^T = \boldsymbol{E}_{l+1}\boldsymbol{Z}_l^T \tag{1.19}$$

Infine, definito con λ il learning rate della rete, l'**aggiornamento dei pesi** viene calcolato con:

$$\boldsymbol{W}_l^T = \boldsymbol{W}_l^T - \lambda \boldsymbol{D}_l^T \tag{1.20}$$

1.2.4 Quantizzatore DSQ

Come accennato in precedenza, il DSQ è un quantizzatore con una funzione caratteristica che approssima in maniera continua e derivabile la funzione caratteristica di un quantizzatore uniforme.
Grazie alle caratteristiche di continuità e derivabilità, questo tipo di quantizzazione risulta particolarmente adatta al processo di addestramento per le reti neurali degli autoencoder, specialmente per quanto riguarda il calcolo del gradiente (equazioni (1.17) e (1.18)).
I parametri necessari per un quantizzatore DSQ, come per un quantizzatore uniforme, sono:

- l'intervallo di quantizzazione: $[t, u]$
- il numero di bit di quantizzazione: s

Partendo da questi parametri, si riescono a determinare:

- gli intervalli di quantizzazione

$$P_w \,,\; w = 0, ..., 2^s - 2 \tag{1.21}$$

- il passo di quantizzazione

$$\Delta = \frac{u - t}{2^s - 1} \tag{1.22}$$

Il quantizzatore DSQ che verrà usato durante gli addestramenti sostituisce ad ogni gradino del quantizzatore uniforme una tangente iperbolica, indicata con $\phi(x)$, le quali trattano i punti che ricadono negli intervalli P_w:

$$\phi(x) = \beta \tanh(p(x - m_w)) \quad , \quad \text{con } x \in P_w \tag{1.23}$$

dove:

$$\beta = \frac{1}{\tanh(0.5p\Delta)} \tag{1.24}$$

$$m_w = t + \Delta(w + 0.5) \tag{1.25}$$

Scendendo nei dettagli di questi parametri, il parametro β è un fattore di scala che assicura la continuità tra le varie tangenti iperboliche, mentre m_w rappresenta il punto medio dell'intervallo P_w.
Il parametro p specifica invece la pendenza delle tangenti iperboliche.
Si può notare come, al crescere del valore di p, il grafico della funzione DSQ diventi quasi totalmente identico a quello del quantizzatore uniforme.
Gli autoencoder dovranno imparare come minimizzare il numero di punti che cadono sulle alzate del DSQ.
Questi sono i punti dove si ha la maggiore incertezza nel passaggio da DSQ a quantizzatore uniforme, appunto perchè con un quantizzatore uniforme i punti possono stare unicamente sulle pedate e non sulle alzate.

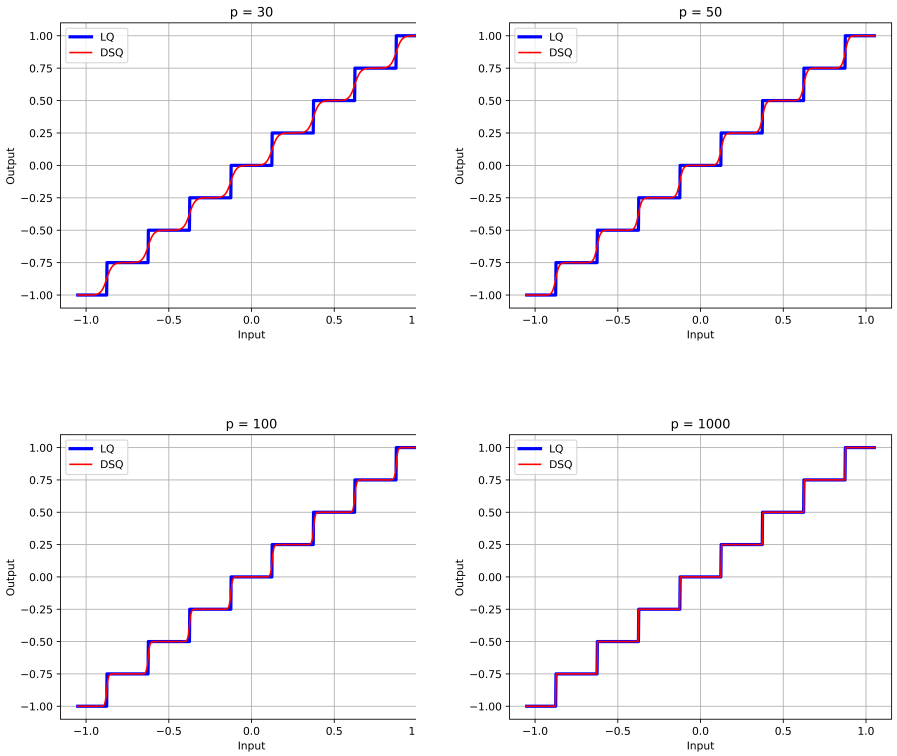


Figura 1.4: Variazione del grafico della funzione del DSQ in relazione al valore della pendenza p

Ora che sono stati specificati tutti i parametri necessari alla quantizzazione, è possibile procedere a definire la funzione di quantizzazione DSQ come:

$$Q_x = \begin{cases} t & x < t \\ u & x > u \\ t + \Delta(w + \frac{\phi(x)+1}{2}) & x \in P_w \ , \ w = 0, ..., 2^s - 2 \end{cases} \tag{1.26}$$

Durante l'addestramento verrà quindi usata l'equazione (1.26), mentre nella fase di test, per andare a replicare il comportamento di un quantizzatore uniforme che non permettere di andare a collocarsi sulle alzate dei gradini, si applicherà la funzione sign a $\phi(x)$:

$$Q_x = \begin{cases} t & x < t \\ u & x > u \\ t + \Delta(w + \frac{\text{sign}(\phi(x))+1}{2}) & x \in P_w \ , \ w = 0, ..., 2^s - 2 \end{cases} \tag{1.27}$$

Gestione dell'incremento del valore della pendenza del DSQ

Come detto in precedenza, durante l'addestramento viene fatto variare il valore del parametro p che regola la pendenza del quantizzatore DSQ, per arrivare a renderlo quasi un quantizzatore uniforme.

Si nota empiricamente che, per valori bassi di p, anche un incremento unitario può portare ad un incremento nella minimizzazione del MSE notevole, mentre per osservare ciò con valori alti di p bisogna incrementare il suo valore di molto.
Per questo, negli addestramenti il valore di p verrà incrementato in maniera proporzionale al suo valore. In altre parole, per valori di p piccoli si avrà un incremento unitario, mentre in seguito al crescere di p aumenterà anche l'incremento.

Per gestire questa situazione, è stata definito un metodo *incrementP(int p)* che si occupa, come suggerisce il nome, di incrementare il valore della pendenza p passato come argomento:

```
incrementP(p:Int):
    if(p<50):
        return p+1
    if(p>=50 and p<100):
        return p+10
    if(p>=100 and p<1000):
        return p+100
```

```
8     if(p>1000):
9       return p+1000
10    return p+1
```

Listing 1.1: funzione per l'incremento di p

1.2.5 Apprendimento

L'apprendimento di un autoencoder può essere visto sia come supervisionato sia come non supervisionato.
Guardando all'autoencoder nel suo complesso, partendo dal layer di input e arrivando a quello di output, considerando quindi come input $\boldsymbol{x}$ e come output $\boldsymbol{x'}$, si ha un apprendimento di tipo supervisionato.
Questo perchè durante l'addestramento vengono forniti sia l'input sia l'output desiderato.
Se invece si guarda solo alla parte dell'encoder, considerando quindi come input $\boldsymbol{x}$ e come output $\boldsymbol{y}$, allora l'apprendimento è di tipo non supervisionato, in quanto la rete non riceve alcuna indicazione sull'output desiderato.

Capitolo 2

Confronto analitico

In questo capitolo verranno confrontati i due metodi sopra descritti in maniera analitica, ossia indipendente dai risultati degli addestramenti.
Gli aspetti sotto i quali i due metodi verranno confrontati sono:

1. complessità computazionale: analisi asintotica con costo unitario per ogni operazione aritmetica tra scalari (addizione e moltiplicazione);

2. occupazione di memoria: viene valutata solo la memoria necessaria a salvare le strutture di compressione e decompressione;

3. qualità della compressione: viene valutato il rapporto tra compressione e distorsione tra output e input in termini di MSE.

In tutte le analisi verranno usate le seguenti variabili:

S	Numero di bit necessari a rappresentare un numero reale.
F	Occupazione di memoria di una funzione (in bit).
γ	Numero di bit necessari a rappresentare il risultato della compressione.
τ	Complessità computazionale.
L	Numero di layer dell'autoencoder.
η_l	Numero di neuroni dell'l-esimo layer dell'autoencoder, $l = 1, ..., L$.
τ_{add}	Complessità di una addizione tra due numeri reali.
τ_{mul}	Complessità di una moltiplicazione tra due numeri reali.
τ_{conf}	Complessità di un confronto tra due numeri reali.

2.1 Algoritmo di Linde, Buzo, Gray

2.1.1 Complessità computazionale

Dall'articolo [5] sappiamo che la complessità computazionale dell'algoritmo Linde-Buzo-Gray (LBG) è data da:

$$\tau_{\text{LBG,art}} = \psi KN((2n-1)t_{\text{add}} + nt_{\text{mul}}) + \psi N(K-1)\tau_{\text{conf}}, \tag{2.1}$$

dove con ψ viene indicato il numero di iterazioni totali svolte dall'algoritmo.
Si può vedere che la somma è divisa in due termini:

1. il primo termine si riferisce al confronto di tutti gli N vettori del training set T con tutti i K vettori n-dimensionali del codebook;
2. il secondo termine si riferisce invece alla formazione delle regioni ottime secondo il codebook.

In questa fase, per ogni vettore del training set T bisogna trovare la distorsione minima fra tutti i vettori del codebook. Ricordando che i vettori del codebook sono K, allora per ognuno degli N vettori del training-set occorrerà effettuare $K-1$ confronti.
Volendo estrarre dalla (2.1) la complessità asintotica, si trova:

$$\tau_{\text{LBG,art,asint}} = \mathcal{O}(\psi KNn + \psi KN). \tag{2.2}$$

Il secondo termine di questa somma, in maniera analoga a quanto visto sopra, rappresenta la parte relativa ai $K-1$ confronti.
Per calcolare la complessità computazionale complessiva dell'algoritmo LBG effettivamente implementato e utilizzato negli addestramenti, è possibile analizzare separatamente le varie funzioni che vengono invocate durante un addestramento, per poi calcolare la complessità totale, sommandole.
Le funzioni a cui si farà riferimento sono:

1. centroid($\boldsymbol{V}$): funzione che calcola il centroide di un insieme di vettori V.
2. distortion($\boldsymbol{v}, \boldsymbol{w}$): funzione che calcola il MSE tra due vettori $\boldsymbol{v}$ e $\boldsymbol{w}$ n-dimensionali.
3. split($\mathcal{C}, \boldsymbol{\varepsilon}_+, \boldsymbol{\varepsilon}_-$): funzione che sdoppia il codebook $\mathcal{C}$, come mostrato in (1.14).

4. compress($\boldsymbol{v}, \boldsymbol{\mathcal{C}}$): funzione di compressione del vettore v in base al codebook $\boldsymbol{\mathcal{C}}$.

Funzione Centroid($\boldsymbol{V}$)

Pseudocodice

Algorithm 1 Funzione che calcola il centroide di un insieme di vettori V

```
1: function CENTROID(V)
2:     sum⃗ ← 0⃗
3:     for all v⃗ ∈ V do
4:         sum⃗ ← sum⃗ + v⃗
5:     return sum⃗/|V|
```

Complessità computazionale

Nel caso peggiore la funzione deve calcolare il centroide relativo all'intero training set T:

$$\tau_{\text{centroid}} = \mathcal{O}(Nn + 1) = \mathcal{O}(Nn) \tag{2.3}$$

Funzione Distortion($\boldsymbol{v}, \boldsymbol{w}$)

Pseudocodice

Algorithm 2 Funzione che calcola il *MSE* tra due vettori n-dimensionali

```
1: function DISTORTION(v⃗, w⃗)
2:     mse ← 0
3:     for i ← 0 to |v⃗| do
4:         d ← v_i − w_i
5:         d ← d · d
6:         mse ← mse + d
7:     return mse/|v⃗|
```

Complessità computazionale

La complessità computazionale di questa funzione è data da:

$$\tau_{\text{distortion}} = \mathcal{O}(3n + 1) = \mathcal{O}(n) \tag{2.4}$$

Funzione Split($\mathcal{C}, \boldsymbol{\varepsilon}_+, \boldsymbol{\varepsilon}_-$)

Pseudocodice

Algorithm 3 Funzione che sdoppia il codebook

```
1: function SPLIT(C, ε⃗_+, ε⃗_−)
2:     C_new ← ∅
3:     for all c⃗ ∈ C do
4:         C_new ← C_new ∪ (c⃗ + ε⃗_+)
5:         C_new ← C_new ∪ (c⃗ + ε⃗_−)
       return C_new
```

Complessità computazionale

La funzione SPLIT viene chiamata ogni volta che è necessario effettuare una fase di divisione del codebook (sezione 1.1.2).
Ricordando che abbiamo definito con K il numero di vettori finale che il codebook deve contenere, allora la funzione durante l'intero algoritmo verrà chiamata un numero di volte pari a $\log_2 K$.
La complessità computazionale di questa funzione è quindi:

$$\tau_{\text{split}} = \mathcal{O}\left(2n \sum_{i=0}^{log_2 K - 1} 2^i\right) = \mathcal{O}(n) \tag{2.5}$$

Funzione Compress($\boldsymbol{v}, \boldsymbol{\mathcal{C}}$)

Pseudocodice

Algorithm 4 Funzione di compressione

```
1: function COMPRESS(v⃗, C)
2:     d_min = DISTORTION(v⃗, C[0])
3:     index ← 0
4:     for m ← 0 to length(C) − 1 do
5:         if DISTORTION(v⃗, C[m]) < d_min then
6:             d_min ← DISTORTION(v⃗, C[m])
7:             index ← m
8:     return index
```

Complessità computazionale

Questa funzione si occupa di calcolare la distorsione tra i vettori del codebook e ricercare linearmente il minimo.
Nel caso peggiore, quando il codebook contiene il massimo numero di vettori K,

ha complessità:

$$\tau_{compress} = \mathcal{O}(K\tau_{distortion} + (K-1)) = \mathcal{O}(Kn + K) = \mathcal{O}(Kn) \tag{2.6}$$

Complessità computazionale complessiva dell'algoritmo LBG

Pseudocodice

Algorithm 5 Algoritmo di Linde, Buzo, Gray

```
1: function LindeBuzoGray(T, δ, ε⃗+, ε⃗−, K)
2:     C ← {0⃗}
3:     D_prev ← +∞
4:     error ← +∞
5:     while length(C) ≤ K and error > δ do
6:         R ← [R_1, ..., R_length(C)]
7:         for all v⃗ ∈ T do
8:             r ← COMPRESS(v⃗, C)
9:             R[r] ← R[r] ∪ v⃗
10:        D ← 0
11:        for r ← 0 to length(C) do
12:            for all v⃗ ∈ R[r] do
13:                D ← D + DISTORTION(v⃗, C[r])
14:        D ← D/length(T)
15:        error ← (D_prev − D)/D
16:        if error > δ then
17:            C ← ∅
18:            for r ← 0 to length(C) do
19:                C ← C ∪ CENTROID(R[r])
20:            D_prev ← D
21:        else
22:            if length(C) < K then
23:                C ← SPLIT(C, ε⃗+), ε⃗−)
24:                D_prev ← +∞
25:                error = +∞
```

Complessità computazionale

I contributi complessivi dati dalle funzioni viste sopra sono:

- CENTROID esegue K iterazioni $\rightarrow \quad K \cdot \tau_{\text{centroid}}$
- DISTORTION con i cicli annidati scorre tutti i vettori del training set $\rightarrow$ $N \cdot \tau_{\text{distortion}}$

- SPLIT comprende già nella sua complessità (vedi (2.5)) il contributo della funzione alla complessità computazionale totale dell'algoritmo $\rightarrow \quad \tau_{\text{split}}$

- COMPRESS scorre gli N vettori totali del training set $T \rightarrow \quad N \cdot \tau_{\text{compress}}$

La presenza del *while* alla riga 5 introduce a tutte le righe di codice contenute nel ciclo un fattore moltiplicativo ψ, pari al numero totale di iterazioni svolte dall'algoritmo.
Sommando tutti i contributi si ottiene:

$$\begin{aligned} &\tau_{\text{LBG}} = \tau_{\text{split}} + \psi(N\tau_{\text{compress}} + N\tau_{\text{distortion}} + K\tau_{\text{centroid}}) \\ &= \mathcal{O}(n) + \psi(N\mathcal{O}(Kn + K)N\mathcal{O}(n) + K\mathcal{O}(Nn)) \\ &= \mathcal{O}(\psi KNn + \psi KN) \end{aligned} \tag{2.7}$$

Che conferma quanto mostrato nella (2.1). Volendo semplificare ulteriormente la complessità asintotica in fase di addestramento si trova che questa vale:

$$\tau_{\text{LBG}} = \mathcal{O}(\psi KNn) \tag{2.8}$$

2.1.2 Occupazione di memoria

Per poter compiere compiere le operazioni di compressione e decompressione, è necessario salvare le rispettive strutture, che sono la funzione di compressione (4) e la funzione di decompressione definita come:

Algorithm 6 Funzione di decompressione

```
1: function DECOMPRESS(C, index)
2:     return C[index]
```

Entrambe le funzioni, avendo tra i parametri di chiamata il codebook $\mathcal{C}$, devono averlo a disposizione.
Ricordando che era stata definita con F l'occupazione di memoria di una funzione in bit, con K la dimensione desiderata del codebook, con n il numero di dimensioni dei vettori del codebook e con S il numero di bit necessari a rappresentare un numero reale, si avrà che l'occupazione di memoria totale (in bit) dell'algoritmo LBG è data da:

$$\mu_{\text{LBG}} = KnS + 2F \text{ bit} \tag{2.9}$$

2.2 Autoencoder

2.2.1 Complessità computazionale

In maniera simile a quanto fatto per l'algoritmo LBG, per analizzare la complessità computazionale complessiva dell'addestramento di un autoencoder (AE), analizzeremo separatamente le varie fasi/passi di cui si compone l'addestramento:

1. *Forward propagation*: illustrato nella sezione 1.2.3.
2. *Backward propagation*: come illustrato nella sezione 1.2.3, composto dalle operazioni di:
   - calcolo dell'errore sull'output;
   - calcolo delle variazioni dell'errore;
   - aggiornamento dei pesi.

Per semplicità, andremo ad analizzare prima la complessità computazionale nell'utilizzo di una rete addestrata, per la quale è sufficiente considerare solamente la complessità del passo di *Forward Propagation* (sezione 1.2.3).

Rete già addestrata

Come visto in precedenza, una rete addestrata dovrà unicamente processare gli input forniti al layer di input della rete fino al layer di output della stessa, pertanto andrà ad utilizzare unicamente il passo di *forward propagation*.
Indicando con L il numero di livelli della rete, questo passo verrà ripetuto *L-1* volte.
Come faremo anche in seguito, per semplicità andremo sempre a calcolare la complessità computazionale relativa ad un input singolo.
La complessità computazionale totale è quindi data da:

$$\tau_{test,forward} = \mathcal{O}(\sum_{l=1}^{L-1} \eta_l \eta_{l+1}) \tag{2.10}$$

Addestramento della rete

In una rete da addestrare vengono eseguiti sia il passo di *forward propagation* sia quello di *backward propagation*.
Il passo di *forward propagation* viene eseguito, in maniera equivalente a quanto

visto sopra per una rete già addestrata, per tutti i vettori in input del training set T e per tutte le epoche ξ.
Considerando per semplicità che all'interno del training set T vi sia un solo vettore, la complessità computazionale totale data dai passi di *forward propagation* è pari a:

$$\tau_{\text{training,forward}} = \mathcal{O}(\xi \sum_{l=1}^{L-1} \eta_l \eta_{l+1}) \tag{2.11}$$

Per quanto riguarda il passo di *backward propagation*, andiamo ad analizzare separatamente le fasi in cui è diviso.

Calcolo dell'errore

Come descritto nella sezione 1.2.3, per il calcolo dell'errore si possono verificare due situazioni, a seconda che il layer della coppia considerata sia il layer di output o un layer interno.
Nel primo caso, mostrato nella (1.17), abbiamo solamente operazioni elemento per elemento:

$$\tau_{\text{errore,output}} = \mathcal{O}(\eta_{l+1} + \eta_{l+1}) = \mathcal{O}(\eta_{l+1}) \tag{2.12}$$

Nel secondo caso, mostrato nella (1.18), viene eseguita una moltiplicazione elemento per elemento e una moltiplicazione tra matrici:

$$\tau_{\text{errore,intern}} = \mathcal{O}(\eta_{l+1} + \eta_{l+1}\eta_{l+2}) = \mathcal{O}(\eta_{l+1} + \eta_{l+2}) \tag{2.13}$$

Nel caso peggiore, il calcolo dell'errore sull'output ha complessità pari alla complessità massima tra la (2.12) e la (2.13):

$$\tau_{\text{errore}} = \max(\tau_{\text{errore,output}}, \tau_{\text{errore,intern}}) = \tau_{\text{errore,intern}} = \mathcal{O}(\eta_{l+1}\eta_{l+2}) = \mathcal{O}(\eta_l \eta_{l+1}) \tag{2.14}$$

Come si noterà anche in seguito, la complessità computazionale dell'addestramento di un AE, come anche quella di utilizzo di un AE già addestrato, è strettamente correlata al numero totale di layer L.

Calcolo delle variazioni dell'errore

Il calcolo delle variazioni dell'errore, come mostrato nella (1.19), si compone di una singola moltiplicazione tra matrici. Questa operazione ha quindi una complessità pari a:

$$\tau_{\text{variazioni}} = \mathcal{O}(\eta_l \eta_{l+1}) \tag{2.15}$$

Aggiornamento dei pesi

Nell'aggiornamento dei pesi, illustrato nella (1.20), vengono eseguite solamente operazioni elemento per elemento:

$$\tau_{\text{aggiornamento}} = \mathcal{O}(\eta_l \eta_{l+1} + \eta_l \eta_{l+1}) = \eta_l \eta_{l+1} \tag{2.16}$$

Complessità computazionale complessiva del passo di backward propagation

Applicando una volta il passo di backward propagation a tutta la rete, rimanendo nell'ipotesi che il training set T contenga un singolo vettore, abbiamo una complessità pari a:

$$\tau_{\text{training,backward}} = \tau_{\text{errore}} + \tau_{\text{variazione}} + \tau_{\text{aggiornamento}} = \mathcal{O}(\sum_{l=1}^{L-1} \eta_l) \tag{2.17}$$

Andando a generalizzare, definendo con N il numero di vettori del training set T, si ottiene una complessità computazionale per l'addestramento dell' AE pari a:

$$\tau_{\text{training,AE}} = N(\tau_{\text{training,forward}} + \tau_{\text{training,backward}}) = \mathcal{O}(N\xi \sum_{l=1}^{L-1} \eta_l \eta_{l+1}) \tag{2.18}$$

2.2.2 Occupazione di memoria

L'occupazione di memoria di un AE, come in generale di una qualsiasi rete neurale, è data dal peso di:

- neuroni,
- pesi,
- bias.

L'occupazione di memoria in bit di un AE è quindi riassumibile con:

$$\mu_{\text{AE}} = \mu_{\text{neuroni}} + \mu_{\text{pesi}} + \mu_{\text{bias}}. \tag{2.19}$$

Il numero di neuroni totali della rete è dato dalla somma dei neuroni di ogni layer della rete:

$$\eta = \sum_{l=1}^{L} \eta_l \tag{2.20}$$

Ricordando che abbiamo indicato con F l'occupazione di memoria di una funzione in bit, e sapendo che i neuroni sono delle funzioni, l'occupazione di memoria dei neuroni è data da:

$$\mu_{\text{neuroni}} = \eta F \tag{2.21}$$

Ancora, ricordando che abbiamo indicato con S il numero di bit necessari a rappresentare un numero reale, l'occupazione di memoria data dai bias è:

$$\mu_{\text{bias}} = \eta S \tag{2.22}$$

Per quanto riguarda i pesi, essendo la rete completamente connessa, come spiegato alla sezione 1.2, avremo un peso per ciascuna connessione presente tra i neuroni.
Pertanto, il numero di pesi è pari al numero delle connessioni presenti tra i neuroni, da cui si ricava:

$$\mu_{\text{pesi}} = S \sum_{l=1}^{L-1} \eta_l \eta_{l+1} \text{bit} \tag{2.23}$$

Si può quindi concludere che la memoria totale occupata da un AE è pari a:

$$\mu_{\text{AE}} = S(\eta + \sum_{l=1}^{L-1} \eta_l \eta_{l+1}) + \eta F \text{bit} \tag{2.24}$$

2.2.3 Osservazioni

Ricapitolando i risultati trovati in questo capitolo, si ha la seguente tabella riassuntiva:

Complessità computazionale dell'addestramento	
Algoritmo LBG	**AE**
$\mathcal{O}(\psi KNn)$	$\mathcal{O}(N\xi \sum_{l=1}^{L-1} \eta_l \eta_{l+1})$
Complessità computazionale dell'utilizzo	
Algoritmo LBG	**AE**
$\mathcal{O}(Kn)$	$\mathcal{O}(\sum_{l=1}^{L-1} \eta_l \eta_{l+1})$
Occupazione di memoria in bit	
Algoritmo LBG	**AE**
$KnS + 2F$	$S(\eta + \sum_{l=1}^{L-1} \eta_l \eta_{l+1}) + \eta F$

Tabella 2.1: Tabella riassuntiva delle complessità computazionali e occupazioni di memoria dei metodi studiati

Osservazioni sulle complessità computazionali

Notiamo come nelle complessità computazionali ci siano molti parametri in gioco, la maggior parte indipendenti fra loro.
Per quanto riguarda l' AE, in generale notiamo che la complessità computazionale dipende dal numero di layer della rete e dal numero di neuroni per layer.
Chiaramente quindi, a parità di minimizzazione del MSE, è da preferire l'utilizzo di un AE single layer rispetto ad un deep AE.
In caso la minimizzazione del MSE dell' AE single layer siano invece peggiori rispetto a quelle di un deep AE, occorre valutare se la perdita di informazione è comunque poco rilevante rispetto al miglioramento delle prestazioni.

Osservazioni sulle occupazioni di memoria

Relativamente ad entrambe le occupazioni di memoria, possiamo individuare due componenti nelle somme, la prima relativa all'occupazione di memoria data dai numeri reali (S), la seconda relativa all'occupazione di memoria data dalle funzioni (F).
Per quanto riguarda lo spazio occupato dalle funzioni, si nota come le funzioni nell' autoencoder siano maggiori in numero rispetto a quelle dell'algoritmo LBG, di conseguenza occupano più spazio.
Per quanto concerne invece lo spazio per la memorizzazione dei numeri reali,

l'algoritmo LBG dovrà salvare i vettori del codebook, mentre l' AE dovrà memorizzare i pesi e i bias.
In questo caso, in maniera equivalente a quanto detto per le complessità computazionali, non si riesce a determinare se un metodo occupi meno dell'altro.
Valgono anche le stesse considerazioni fatte sopra relativamente ad un AE single layer.

Capitolo 3

Addestramento e test dei modelli

Nelle simulazioni effettuate per testare i due metodi, sono stati utilizzati due dataset differenti:

1. dataset di gaussiane correlate;

2. dataset di due mixture-gaussian correlate.

Ciascuno di questi dataset è pensato per minimizzare eventuali simmetrie nelle distribuzioni di probabilità dei vettori contenuti al loro interno, allo scopo di evitare di ricadere in uno *sweet spot* e quindi ottenere risultati falsati.
I due metodi descritti e confrontati in precedenza sono stati implementati in linguaggio *Python*, in particolare per il codice relativo agli autoencder e al quantizzatore DSQ è stato utilizzato il framework *PyTorch* [8].
Nelle simulazioni verrà effettuato, per ogni compressione, un confronto tra LBG, autoencoder lineare e deep autoencoder, in termini di:

- minimizzazione del MSE con i modelli addestrati,

- convergenza dell'addestramento, relativamente alle epoche/iterazioni di addestramento;

- regioni di decisioni create.

La convergenza degli autoencoder verrà poi confrontata anche in relazione ai valori della pendenza p della funzione di quantizzazione DSQ (1.2.4).
Verrà infine studiato il rapporto tra layer dell'autoencoder, bit di quantizzazione e prestazioni, per capire il rapporto tra questi.

3.1 Parametri adottati per gli addestramenti

3.1.1 Parametri per l'algoritmo LBG

Parametro	Test con input 10-dimensionali	Test con input bidimensionali
K	16, 64	8, 16
δ	10^{-5}	$10^{-}5$
ε_{+}	$10^{-5} \cdot \mathbf{1}$	$10^{-}5 \cdot \mathbf{1}$
ε_{-}	$10^{-5} \cdot -\mathbf{1}$	$10^{-}5 \cdot -\mathbf{1}$

Tabella 3.1: Parametri adottati negli addestramenti per l'algoritmo LBG

3.1.2 Parametri per l'autoencoder

Parametro	Test con input 10-dimensionali	Test con input bidimensionali
λ	10^{-5}	10^{-5}

Tabella 3.2: Parametri adottati negli addestramenti per l'autoencoder

3.1.3 Parametri per il quantizzatore DSQ

Parametro	Test con input 10-dimensionali	Test con input bidimensionali
s	2, 3, 4	3, 4
t	-1	-1
u	1	1

Tabella 3.3: Parametri adottati negli addestramenti per il DSQ

3.2 Procedura per l'analisi della minimizzazione del MSE

Per ciascuno dei dataset utilizzati, verranno analizzati i risultati delle simulazioni sui modelli **addestrati** in termini di distorsione MSE.
I metodi sono stati testati con una compressione a 4 e a 6 bit: questo significa che l'algoritmo LBG dovrà trovare, rispettivamente, $2^4 = 16$ e $2^6 = 64$ regioni.
Per quanto riguarda invece gli autoencoder, avendo due neuroni di quantizzazione (figure 1.2, 1.3, 1.1), questi avranno nel primo caso 2 bit di quantizzazione e 3

nel secondo.
Le compressioni analizzate a tutto tondo nelle varie sezioni saranno appunto quelle a 4 e a 6 bit, mentre inizialmente verranno esaminate anche quelle a 2 e a 8 bit in termini di minimizzazione del MSE del modello addestrato, per capire il rapporto tra bit di quantizzazione e numero di layer dell'autoencoder in relazione al dataset utilizzato.

3.3 Procedura per l'analisi della convergenza dell'addestramento

Per ciascuno dei dataset utilizzati, verranno riportati i risultati delle simulazioni per verificare la minimizzazione del MSE durante l'addestramento.
Per l'algoritmo LBG la distorsione verrà raccolta ad ogni iterazione dell'algoritmo mentre per gli autoencoder verrà raccolta ogni 15 epoche, cioè ad ogni incremento del valore della pendenza p del quantizzatore DSQ (come definito nello snippet 1.2.4).
Per quanto riguarda l'autoencoder, verrà poi calcolato di quanto migliora la minimizzazione del MSE nelle ultime 300 epoche, passando da una pendenza p del DSQ di $79 \cdot 10^3$ ad una finale di $99 \cdot 10^3$, per capire se è possibile considerare l'addestramento dell'autoencoder giunto a convergenza.

Capitolo 4

Simulazioni di gaussiane correlate

Una prima fase di test è stata effettuata con un dataset i cui valori sono dati dalla correlazione di due variabili aleatorie gaussiane indipendenti x_1 e x_2, con media nulla e varianza unitaria, mediante la relazione:

$$\boldsymbol{y} = \boldsymbol{M}_n \, \boldsymbol{x}, \tag{4.1}$$

dove n indica il numero di dimensioni del vettore in ingresso/uscita.
Per valutare il MSE è stata utilizzata la matrice:

$$\boldsymbol{M}_{10} = \begin{bmatrix} -0.6 & -0.2 \\ 0.5 & -0.3 \\ 0.6 & -0.2 \\ -0.7 & -0.4 \\ 0.6 & 0.3 \\ -0.7 & 0.1 \\ 0.4 & 0.3 \\ -0.1 & -0.8 \\ 0.5 & -0.6 \\ 0.2 & 0.2 \end{bmatrix} \tag{4.2}$$

Mentre per l'analisi delle regioni di decisione è stata usata la matrice:

$$\boldsymbol{M}_2 = \begin{bmatrix} 0.7 & -0.2 \\ -0.8 & 0.1 \end{bmatrix} \tag{4.3}$$

Entrambi i dataset sono stati generati tramite *Python* e sono complessivamente composti da 70000 vettori: 60000 vettori per il training set e 10000 per il test set.

4.1 Minimizzazione del MSE con i modelli addestrati

4.1.1 Compressione a 2 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	59	-8.3
$AE1$	2310	-7.6
$AE2$	2310	-8.3
$AE3$	2310	-8.2
$AE4$	2310	-8.3
$AE5$	2310	-7.5
$AE6$	2310	-7.0
$AE7$	2310	-7.1
$AE8$	2310	-6.9
$AE9$	2310	-6.7

Tabella 4.1: Minimizzazione del MSE ottenuta con una compressione a 2 bit

4.1.2 Compressione a 4 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	203	-13.5
$AE1$	2310	-14.6
$AE2$	2310	-14.5
$AE3$	2310	-13.7
$AE4$	2310	-7.7
$AE5$	2310	-8.8
$AE6$	2310	-5.2
$AE7$	2310	-7.7
$AE8$	2310	-6.0
$AE9$	2310	-4.9

Tabella 4.2: Minimizzazione del MSE ottenuta con una compressione a 4 bit

4.1.3 Compressione a 6 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	429	-19.1
$AE1$	2310	-19.7
$AE2$	2310	-18.6
$AE3$	2310	-18.6
$AE4$	2310	-14.2
$AE5$	2310	-10.7
$AE6$	2310	-12.3
$AE7$	2310	-9.9
$AE8$	2310	-11.6
$AE9$	2310	-9.5

Tabella 4.3: Minimizzazione del MSE ottenuta con una compressione a 6 bit

4.1.4 Compressione a 8 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	606	-25.1
$AE1$	2310	-23.5
$AE2$	2310	-22.8
$AE3$	2310	-22.5
$AE4$	2310	-23.5
$AE5$	2310	-18.3
$AE6$	2310	-15.6
$AE7$	2310	-13.7
$AE8$	2310	-9.6
$AE9$	2310	-8.1

Tabella 4.4: Minimizzazione del MSE ottenuta con una compressione a 8 bit

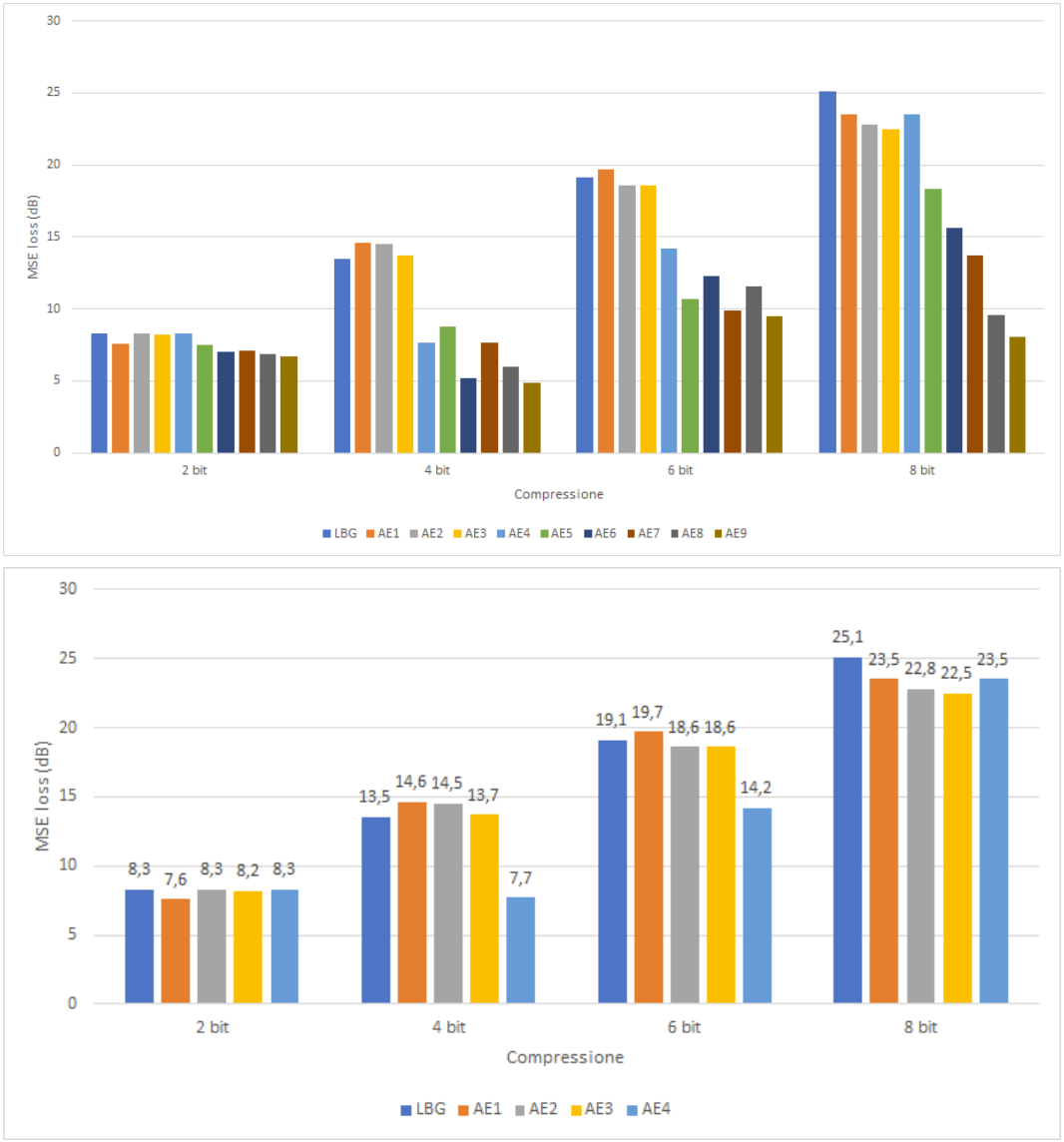


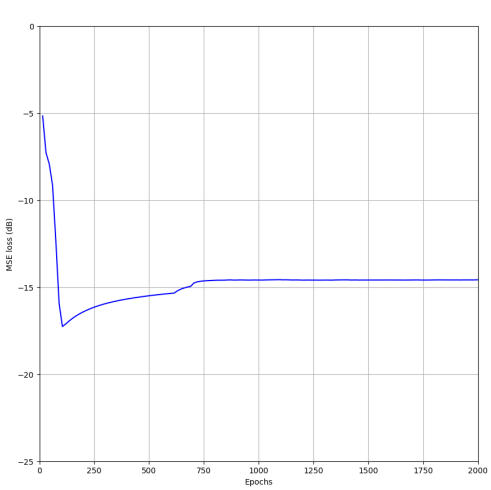
Figura 4.1: Confronto tra la minimizzazione del MSE dei modelli in relazione al numero di bit di compressione

Dai risultati sopra si può notare che utilizzare autoencoder con più di 3 layer non porta a particolari vantaggi, indipendentemente dal numero di bit di quantizzazione utilizzati.
Pertanto, nelle sezioni successive, non andremo ad utilizzare deep autoencoders con 4 o più layers.

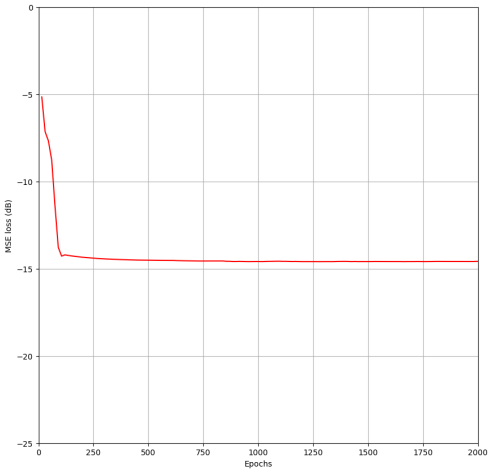
4.2 Convergenza dell'addestramento

4.2.1 Compressione a 4 bit

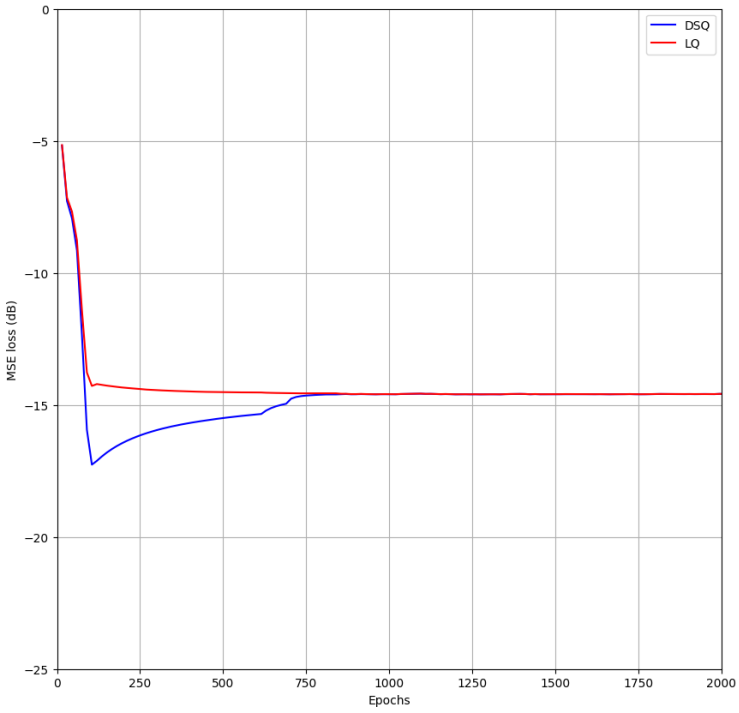
AE1



(a) minimizzazione MSE con DSQ



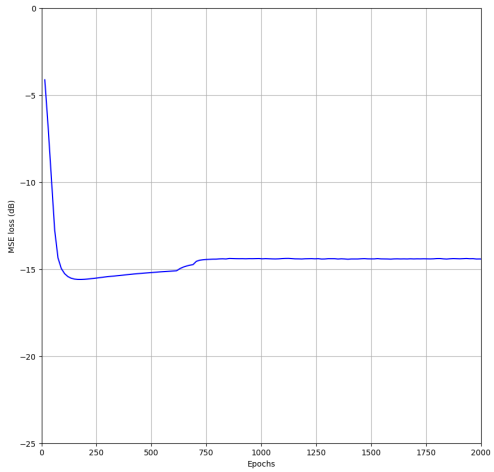
(b) Minimizzazione MSE con quantizzatore uniforme



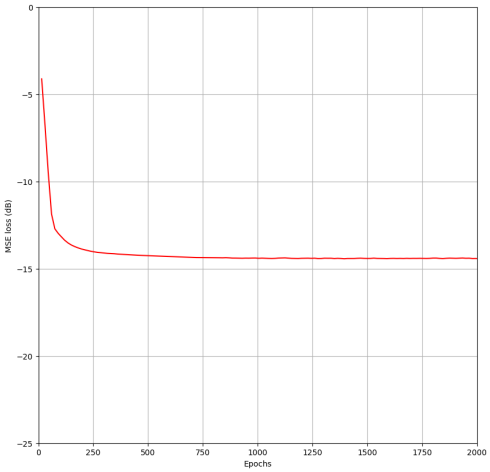
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 4.2: Convergenza dell'addestramento di un AE single layer con una compressione a 4 bit

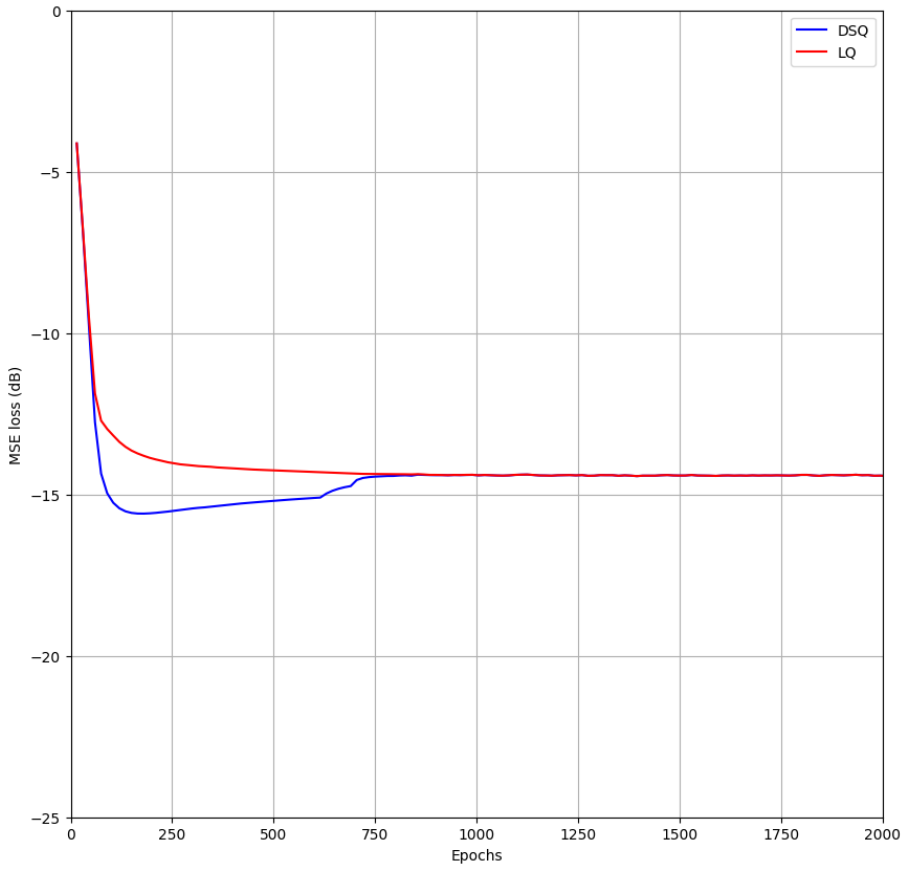
AE2



(a) minimizzazione MSE con DSQ



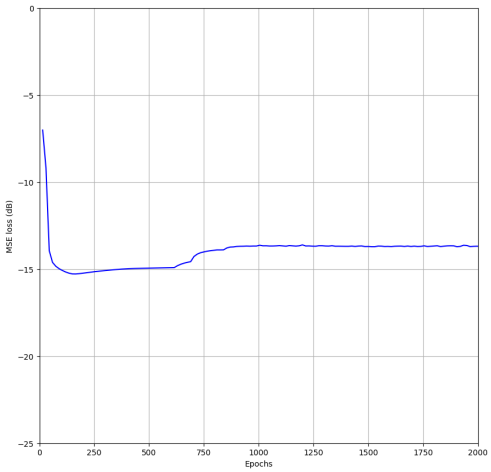
(b) Minimizzazione MSE con quantizzatore uniforme



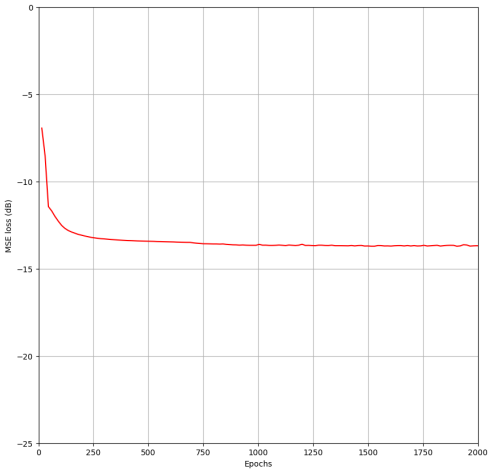
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 4.3: Convergenza dell'addestramento di un AE a 2 layer con una compressione a 4 bit

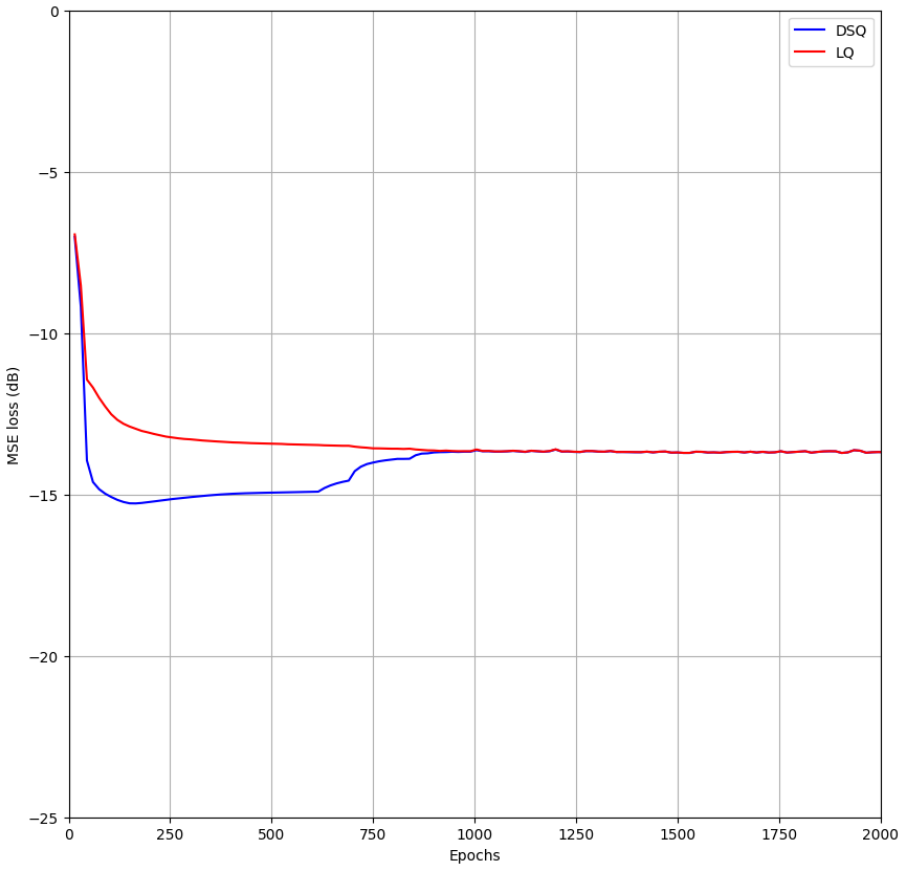
AE3



(a) minimizzazione MSE con DSQ



(b) Minimizzazione MSE con quantizzatore uniforme



(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 4.4: Convergenza dell'addestramento di un AE a 3 layer con una compressione a 4 bit

Confronto tra LBG e autoencoder

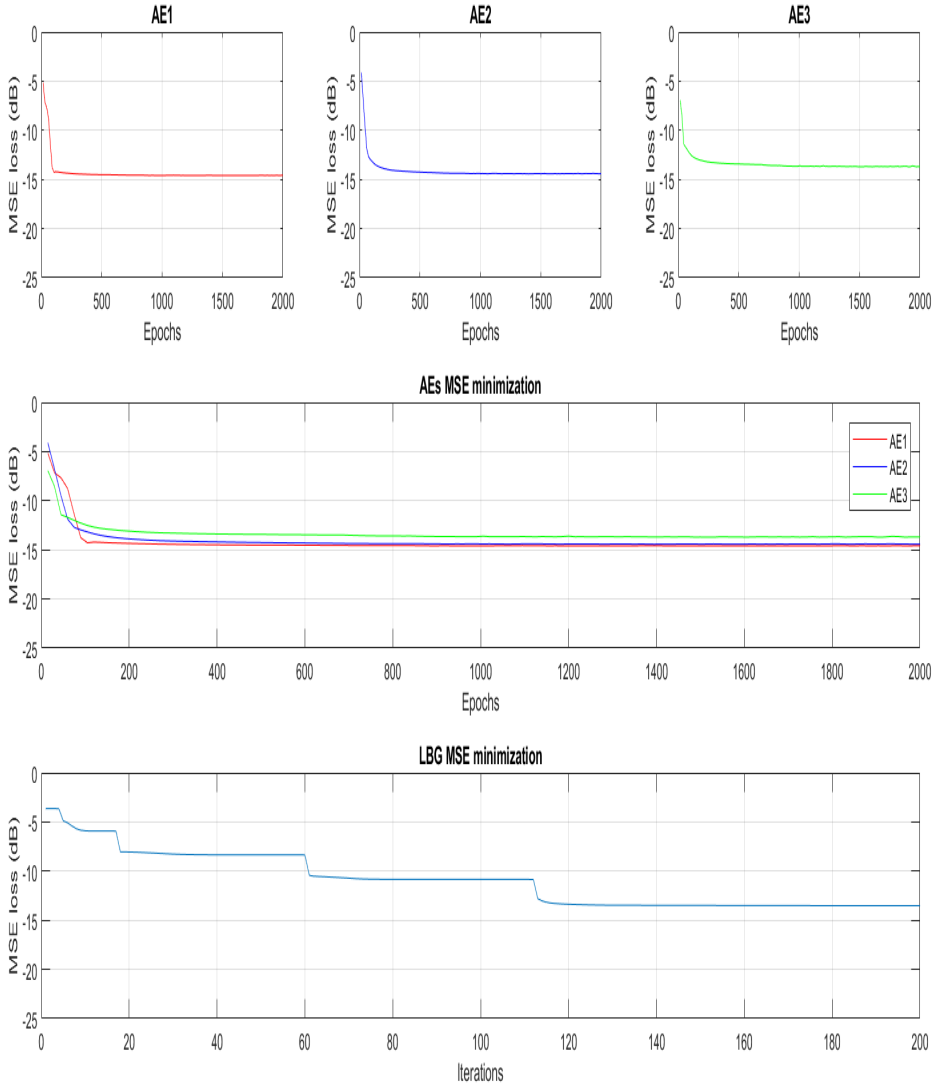
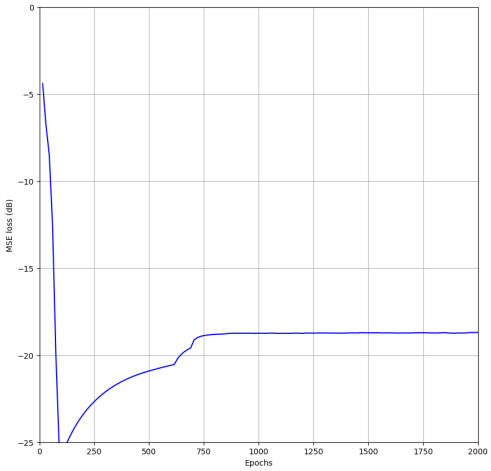


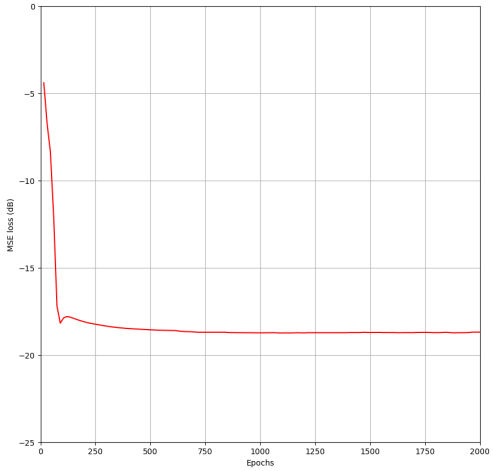
Figura 4.5: Minimizzazione MSE con compressione a 4 bit

4.2.2 Compressione a 6 bit

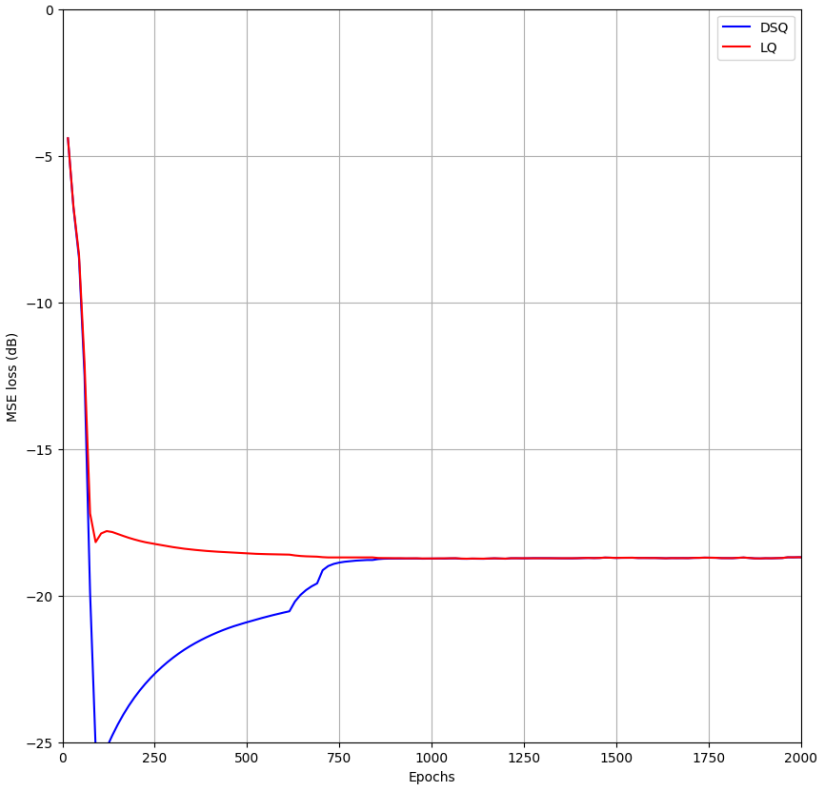
AE1



(a) minimizzazione MSE con DSQ



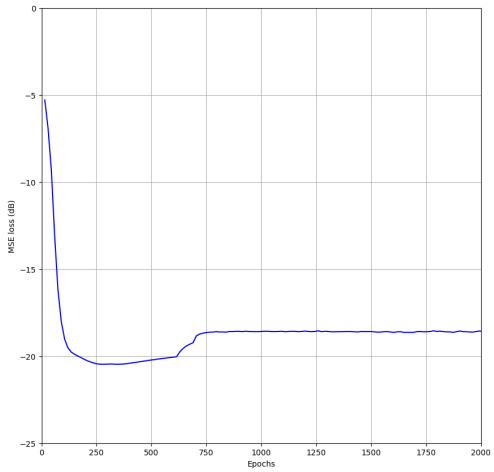
(b) Minimizzazione MSE con quantizzatore uniforme



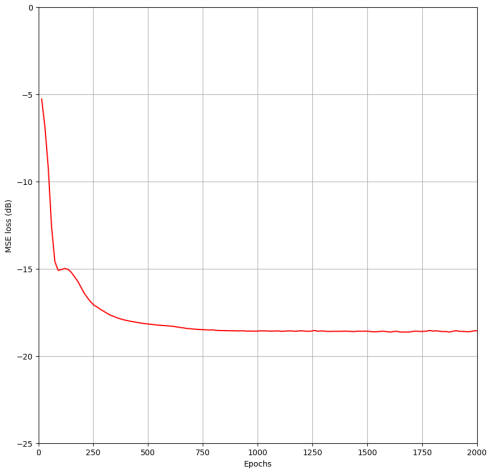
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 4.6: Convergenza dell'addestramento di un AE single layer con una compressione a 6 bit

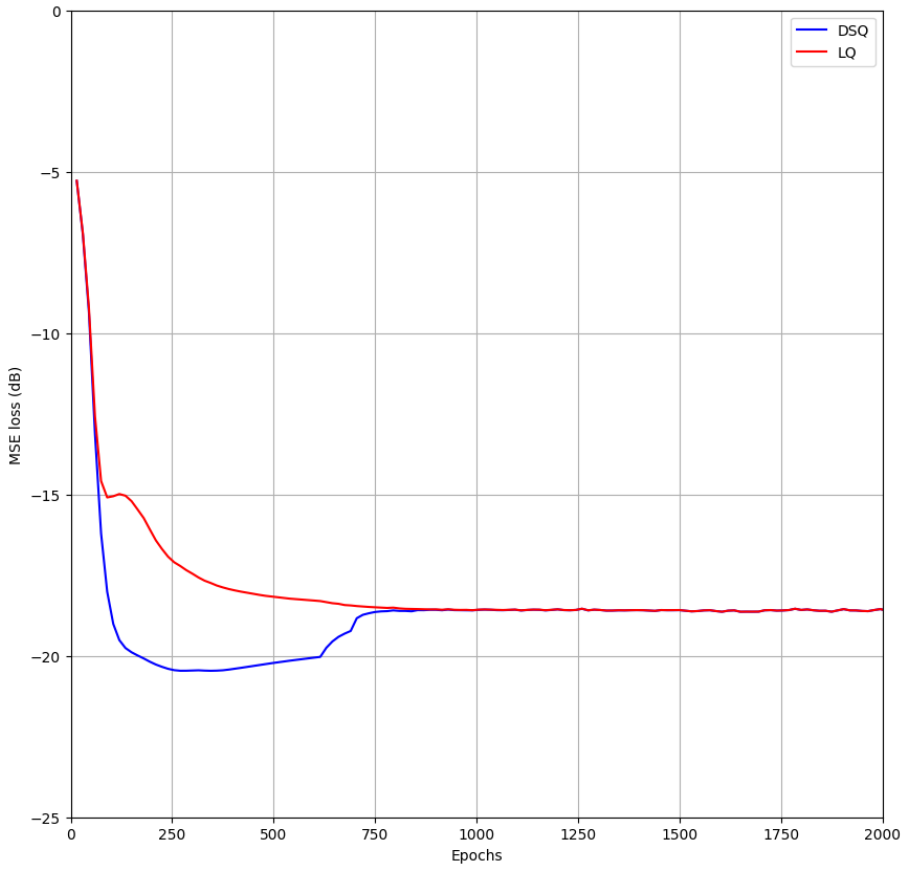
AE2



(a) minimizzazione MSE con DSQ



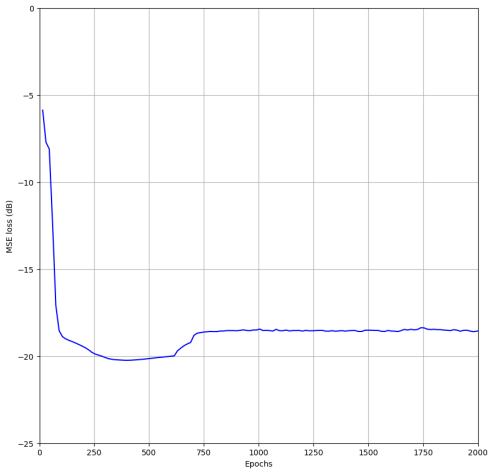
(b) Minimizzazione MSE con quantizzatore uniforme



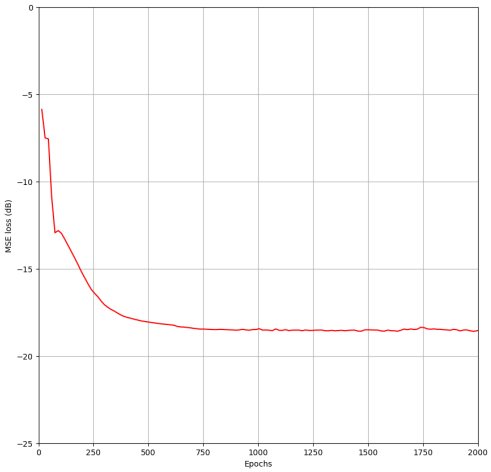
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 4.7: Convergenza dell'addestramento di un AE a 2 layer con una compressione a 6 bit

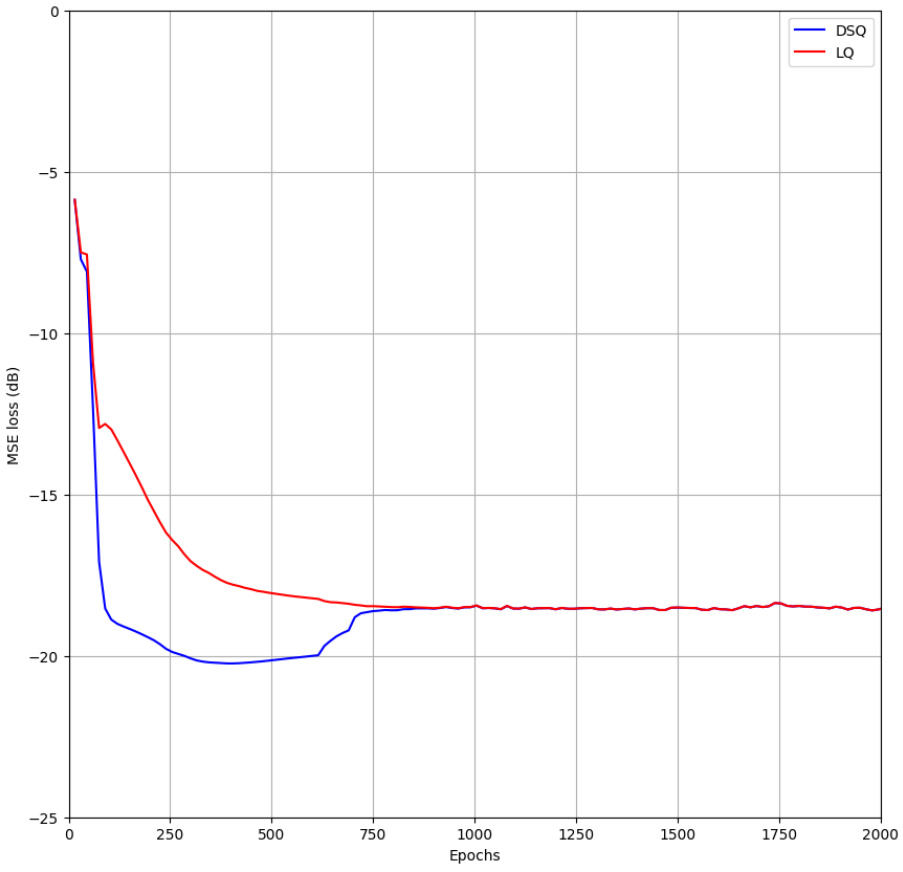
AE3



(a) minimizzazione MSE con DSQ



(b) Minimizzazione MSE con quantizzatore uniforme



(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 4.8: Convergenza dell'addestramento di un AE a 3 layer con una compressione a 6 bit

Confronto tra LBG e autoencoder

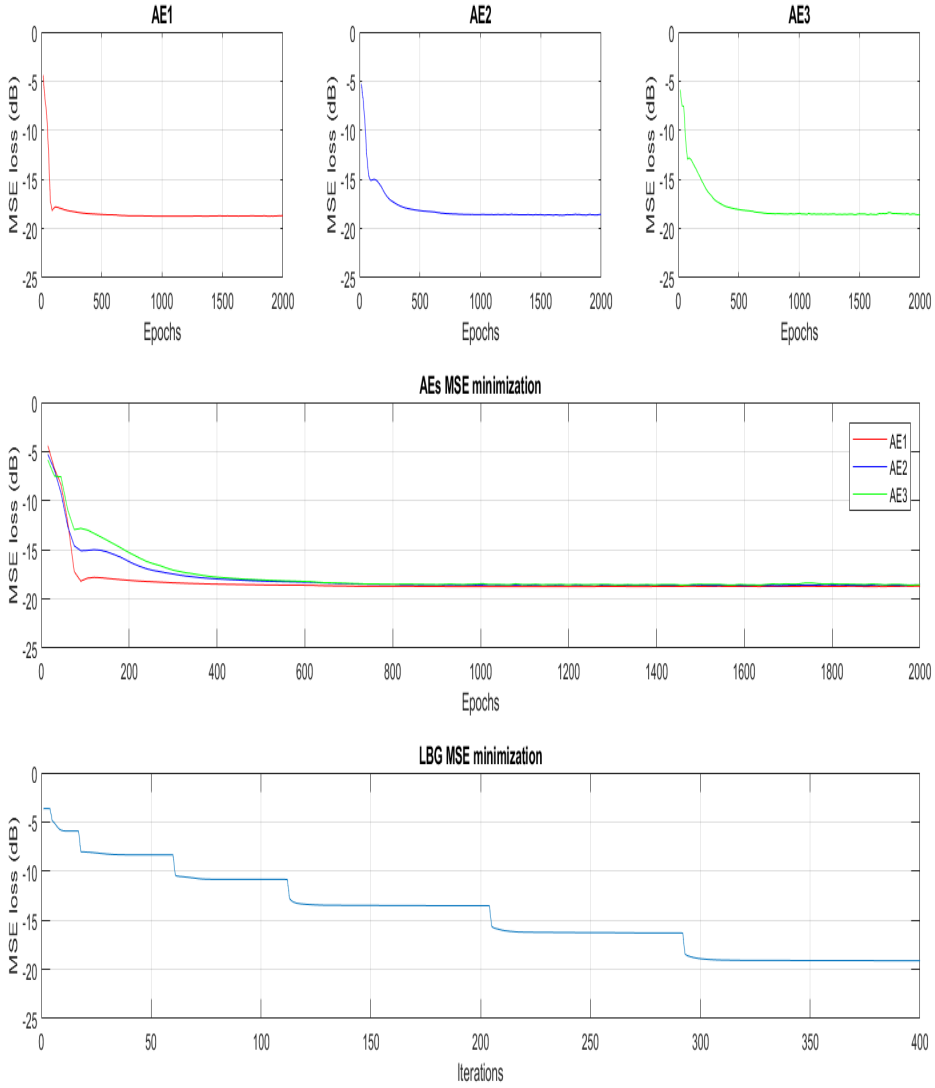


Figura 4.9: Minimizzazione MSE con compressione a 6 bit

4.2.3 Analisi analitica della convergenza

Modello	Bit di compressione	$\Delta_{MSE\,loss}$ ultime 300 epoche	Miglioramento % rispetto al MSE finale
$AE1$	4 bit	$4{\cdot}10^{-5}$	0.13 %
$AE2$	4 bit	$6{\cdot}10^{-5}$	0.15 %
$AE3$	4 bit	$2{\cdot}10^{-5}$	0.05 %
$AE1$	6 bit	$2{\cdot}10^{-5}$	0.2 %
$AE2$	6 bit	$7{\cdot}10^{-5}$	0.7 %
$AE3$	6 bit	$6{\cdot}10^{-5}$	0.6 %

Tabella 4.5: Miglioramento della minimizzazione del MSE nelle ultime 300 epoche

Dai grafici e dalla tabella sopra, è possibile affermare che gli autoencoder siano giunti a convergenza, dato che con un incremento della pendenza p del quantizzatore DSQ pari a $20 \cdot 10^3$ i miglioramenti nella minimizzazione del MSE sono quasi irrilevanti.
Si nota inoltre che la minimizzazione del MSE del DSQ in tutti i casi evolve in maniera diversa rispetto a quella del quantizzatore uniforme.
Questo perchè la rete, come spiegato nella sezione 1.2.4, minimizza il più possibile il numero di punti che cadono sulle alzate del DSQ.
Inizialmente la rete andrà ad aggiustare i pesi e i bias per ridurre l'errore, in seguito, all'aumentare del valore della pendenza p, andrà a lavorare sulla distribuzione di probabilità dei punti in ingresso al quantizzatore DSQ per andare a renderlo sempre più compatibile con il quantizzatore uniforme.

4.3 Regioni di decisione

Per andare a graficare le regioni di decisione in maniera semplice, sono stati forniti sia agli autoencoder sia all'algoritmo LBG degli input bidimensionali.
Gli autoencoder utilizzati a questo scopo sono l'autoencoder lineare e l'autoencoder a 2 layer, modificati rispetto a quelli mostrati nelle figure (1.3) e (1.1) in maniera tale che il layer di quantizzazione abbia un solo neurone:

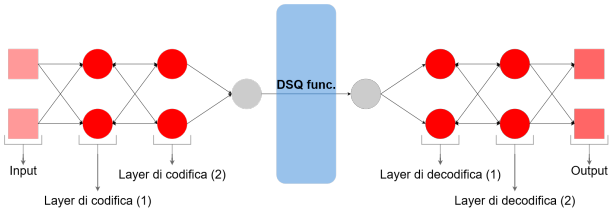


Figura 4.10: Struttura di un autoencoder con input bidimensionale

4.3.1 Compressione a 3 bit

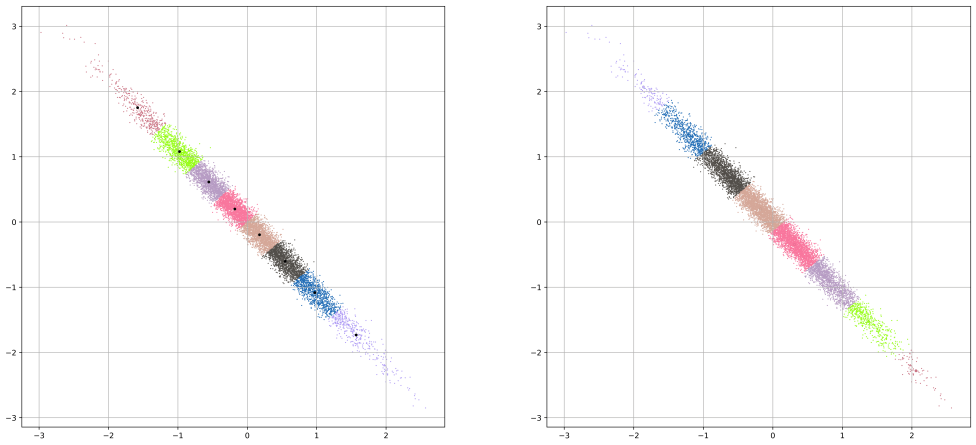


Figura 4.11: Regioni create, e relativi centroidi, dall'algoritmo LBG (a sinistra) e dall'autoencoder a 2 layer (a destra), con una compressione a 3 bit

Minimizzazione del mean-squared error:

- autoencoder a 2 layer: -13.7 dB;
- algoritmo LBG: -12.4 dB.

4.3.2 Compressione a 4 bit

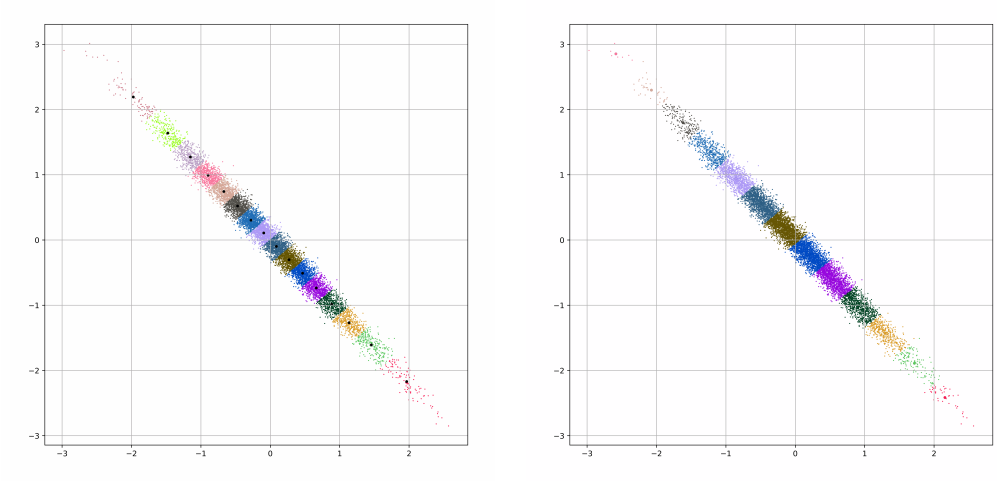


Figura 4.12: Regioni create, e relativi centroidi, dall'algoritmo LBG (a sinistra) e dall'autoencoder a 2 layer (a destra), con una compressione a 4 bit

Minimizzazione del mean-squared error:

- autoencoder a 2 layer: -19.9 dB
- algoritmo LBG: -17.5 dB

4.4 Stime asintotiche

Conoscendo le caratteristiche del dataset, oltre alla struttura e i parametri degli autoencoder utilizzati, è possibile fornire per ciascuno dei due metodi delle stime riguardo la complessità computazionale τ in termini di operazioni tra scalari e l'occupazione di memoria μ in bit viste nel capitolo (2).

4.4.1 Complessità computazionale

Modello	$\boldsymbol{\tau}$ addestramento	$\boldsymbol{\tau}$ utilizzo
MSE loss con compressione a 4 bit		
Algoritmo LBG	$1.95 \cdot 10^9$	160
Autoencoder AE1	$6.10 \cdot 10^9$	44
Autoencoder AE2	$6.10 \cdot 10^9$	44
Autoencoder AE3	$1.72 \cdot 10^{10}$	124
MSE loss con compressione a 6 bit		
Algoritmo LBG	$1.65 \cdot 10^{10}$	640
Autoencoder AE1	$6.10 \cdot 10^9$	44
Autoencoder AE2	$6.10 \cdot 10^9$	44
Autoencoder AE3	$1.72 \cdot 10^{10}$	124
Regioni di decisione con compressione a 3 bit		
Algoritmo LBG	$4.90 \cdot 10^8$	16
Autoencoder AE1	$6.10 \cdot 10^9$	22
Autoencoder AE2	$6.10 \cdot 10^9$	22
Regioni di decisione con compressione a 4 bit		
Algoritmo LBG	$1.07 \cdot 10^8$	32
Autoencoder AE1	$6.10 \cdot 10^9$	22
Autoencoder AE2	$6.10 \cdot 10^9$	22

Tabella 4.6: Tabella riassuntiva per la complessità computazionale dei modelli

4.4.2 Occupazione di memoria

Modello	$\boldsymbol{\mu}$ [bit]
MSE loss con compressione a 4 bit	
Algoritmo LBG	$160S + 2F$
Autoencoder AE1	$68S + 24F$
Autoencoder AE2	$68S + 24F$
Autoencoder AE3	$158S + 34F$
MSE loss con compressione a 6 bit	
Algoritmo LBG	$640S + 2F$
Autoencoder AE1	$68S + 24F$
Autoencoder AE2	$68S + 24F$
Autoencoder AE3	$158S + 34F$
Regioni di decisione con compressione a 3 bit	
Algoritmo LBG	$16S + 2F$
Autoencoder AE1	$37S + 15F$
Autoencoder AE2	$37S + 15F$
Regioni di decisione con compressione a 4 bit	
Algoritmo LBG	$32S + 2F$
Autoencoder AE1	$37S + 15F$
Autoencoder AE2	$37S + 15F$

Tabella 4.7: Tabella riassuntiva per l'occupazione di memoria dei modelli

4.5 Considerazioni e osservazioni

In primo luogo, dai risultati mostrati nelle tabelle nella Sezione 4.1, si nota come i vari autoencoder, sia il lineare che i non lineari, riescano ad ottenere una minimizzazione del MSE pari o migliore di quella ottenuta con l'algoritmo LBG, indipendentemente dal numero di bit di compressione utilizzati.
Si vede inoltre che l'autoencoder lineare, nonostante la componente di non linearità introdotta dal quantizzatore DSQ, riesce ad ottenere una qualità di ricostruzione equivalente o migliore a quella dei deep autoencoder.
Una ragione potrebbe essere che la rete utilizza due neuroni per la quantizzazione (Figura 1.3). Utilizzando un dataset composto da due gaussiane correlate (Sezione 4), la rete può utilizzare i neuroni di quantizzazione per apprendere le due variabili aleatorie e come esse sono correlate.
In seguito verranno utilizzati dataset che eliminano situazioni di questo tipo, così da andare a vedere se l'autoencoder lineare continua ad avere prestazioni simili o migliori di quelle dei deep autoencoder.
Guardando invece ai risultati esposti nella Sezione 4.4, si vede come gli autoencoder in fase di utilizzo abbiano una complessità computazionale minore dell'algoritmo LBG, mentre in fase di addestramento questo non vale per l'autoencoder a 3 layer.
Per quanto concerne invece l'occupazione di memoria, tendenzialmente gli autoencoder hanno un'occupazione di memoria maggiore rispetto all'algoritmo LBG. Non è comunque possibile astrarre una vera e propria conclusione riguardo questo parametro, dato che dipende strettamente dal numero di bit necessari a rappresentare un numero reale (S) e dall'occupazione di memoria di una funzione (F).
Osserviamo infine come le regioni di decisione mostrate nella Sezione 4.3 create dall'algoritmo LBG e dall'autoencoder siano simili. L'unica differenza apprezzabile è una diversa suddivisione dello spazio: mentre l'algoritmo LBG tende a creare regioni più piccole nella zona centrale e più grandi verso le zone esterne, l'autoencoder mostra regioni di circa uguali dimensioni, in accordo con la presenza del quantizzatore uniforme.
Considerando quindi tutti questi fattori, si può concludere che, con questo specifico dataset, risulta più conveniente sotto tutti i punti di vista utilizzare un autoencoder lineare.

Capitolo 5

Simulazioni con due mixture-gaussian

La seconda fase di test è stata effettuata con un dataset i cui valori hanno una densità di probabilità data dalla combinazione lineare di due densità di probabilità di gaussiane in una mixture-gaussian:

1. x_1, con media 0 e varianza 0.4 ;
2. x_2, con media 2 e varianza 0.7 .

Con mixture-gaussian si intende un modello la cui densità di probabilità è data dalla somma di due o più densità di probabilità di variabili aleatorie gaussiane indipendenti [6].
La densità di probabilità di questo modello è data da:

$$p(x) = \sum_{i=1}^{2} \frac{1}{2} \cdot \mathcal{N}(x \,|\, \mu_i, \sigma_i) \quad , \tag{5.1}$$

Il motivo per cui andiamo ad utilizzare questo dataset è testare la robustezza dei modelli con dataset aventi una statistica dei simboli di molto più elaborata rispetto a quella vista nel Capitolo 4.
Per generare valori da una statistica di questo tipo è stato definito il metodo *generate2mix(num: Int, dim: Int)*, dove *num* indica il numero di vettori da generare mentre *dim* rappresenta le dimensioni di questi:

```
1 def generate2mix(num: int, dim: int):
2   mu = [0, 2]
3   sigma = [0.4, 0.7]
4   dataset = []
```

```
  for i in range(num): #numero di vettori da generare
    tmp = []
    for j in range(dim): #dimensione dei vettori da generare
      n_rnd = random.randint(0, 1)
      val_generated = np.random.normal(mu[n_rnd], sigma[n_rnd])
      tmp.append(val_generated)
    dataset.append(tmp)

return dataset
```

Listing 5.1: funzione per generare vettori dalla distribuzione 2 mixture gaussian

Entrambi i dataset sono stati generati tramite *Python* e sono composti da 70000 vettori complessivamente: 60000 vettori per il training set e 10000 per il test set.

5.1 Minimizzazione del MSE con i modelli addestrati

5.1.1 Compressione a 2 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	117	-2.6
$AE1$	2310	-0.5
$AE2$	2310	-0.4
$AE3$	2310	-2.7

Tabella 5.1: Minimizzazione del MSE ottenuta con una compressione a 2 bit

5.1.2 Compressione a 4 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	288	-3.7
$AE1$	2310	-4.0
$AE2$	2310	-3.9
$AE3$	2310	-3.6

Tabella 5.2: Minimizzazione del MSE ottenuta con una compressione a 4 bit

5.1.3 Compressione a 6 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	540	-4.8
$AE1$	2310	-5.1
$AE2$	2310	-5.0
$AE3$	2310	-4.3

Tabella 5.3: Minimizzazione del MSE ottenuta con una compressione a 6 bit

5.1.4 Compressione a 8 bit

Modello	Numero di epoche/iterazioni	MSE finale (dB)
LBG	745	-6.1
*AE*1	2310	-5.6
*AE*2	2310	-5.2
*AE*3	2310	-4.8

Tabella 5.4: Minimizzazione del MSE ottenuta con una compressione a 8 bit

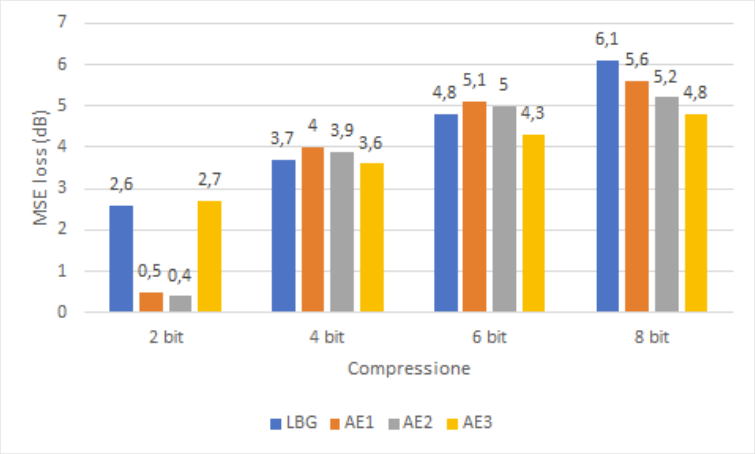


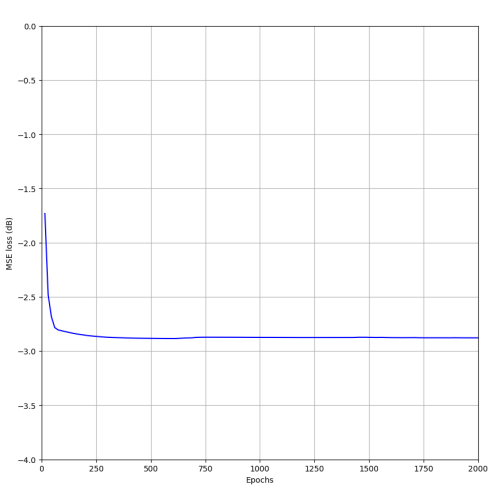
Figura 5.1: Confronto tra le minimizzazioni dei modelli in relazione ai bit di compressione

Con questo dataset, a differenza di quanto visto con il dataset della sezione precedente, non si ha un dominio assoluto dell'autoencoder lineare con tutte le compressioni.
In particolare, emerge che nelle compressioni a 2 e 4 bit l'autoencoder lineare minimizza il MSE in maniera migliore rispetto ai deep autoencoder, mentre con le compressioni a 6 e 8 bit maggiore è il numero di layer dell'autoencoder migliore è la minimizzazione del MSE.
Si nota inoltre che le minimizzazioni del MSE che si ottiengono con i modelli addestrati sono decisamente peggiori rispetto a quelle viste nella Sezione precedente 4.1. Questo perchè, avendo entrambe le gaussiane varianza minore di uno, i valori del training set saranno più ravvicinati, di conseguenza risulterà più facile assegnare un valore ad una regione di decisione sbagliata.
Inoltre, l'algoritmo LBG riesce sempre a minimizzare il MSE meglio rispetto ai vari autoencoder, indipendentemente dal numero di bit di compressione.

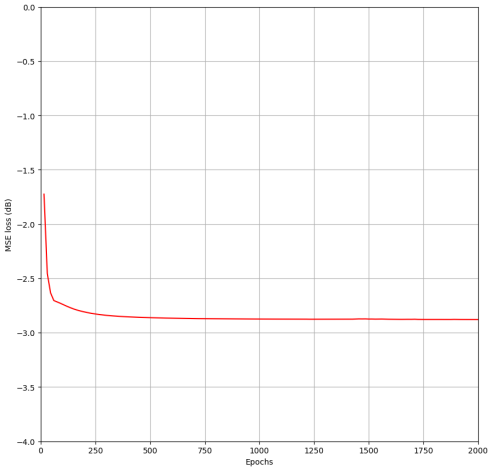
5.2 Convergenza dell'addestramento

5.2.1 Compressione a 4 bit

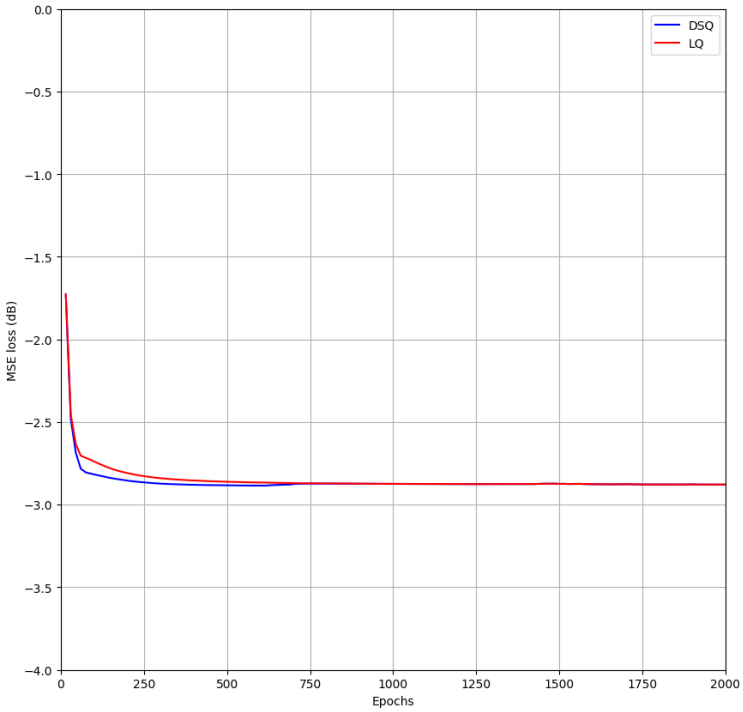
AE1



(a) minimizzazione MSE con DSQ



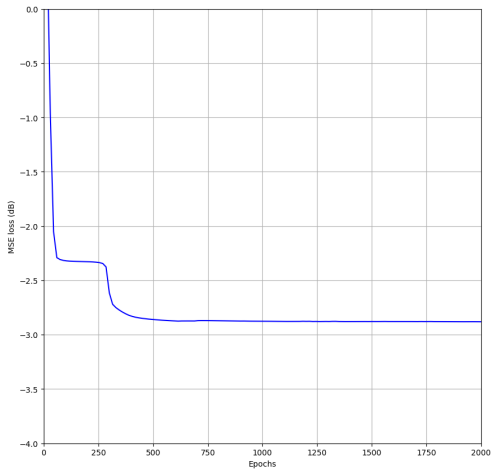
(b) Minimizzazione MSE con quantizzatore uniforme



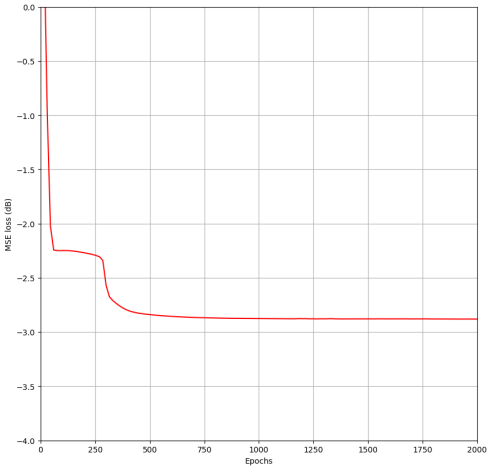
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 5.2: Convergenza dell'addestramento di un AE single layer con una compressione a 4 bit

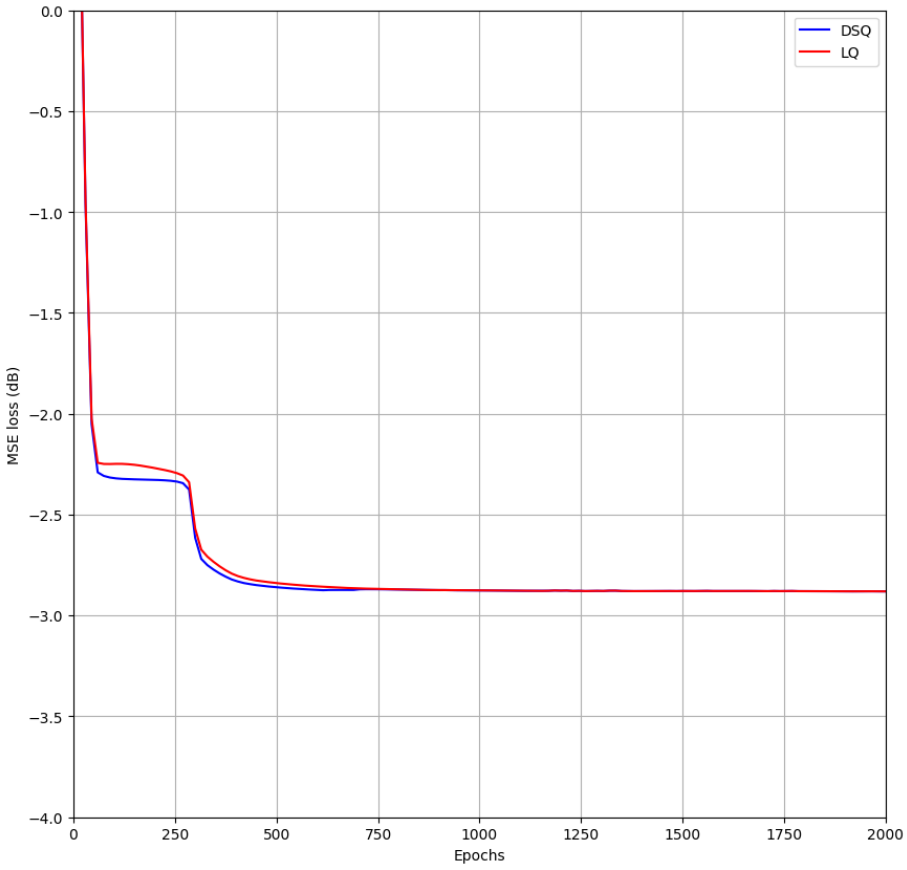
AE2



(a) minimizzazione MSE con DSQ



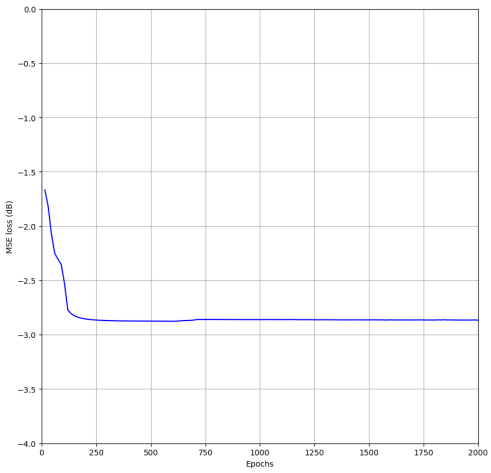
(b) Minimizzazione MSE con quantizzatore uniforme



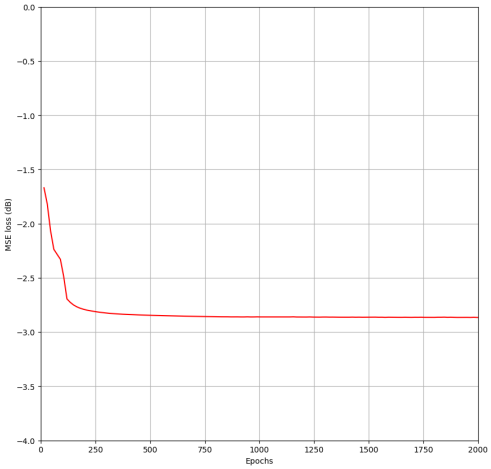
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 5.3: Convergenza dell'addestramento di un AE a 2 layer con una compressione a 4 bit

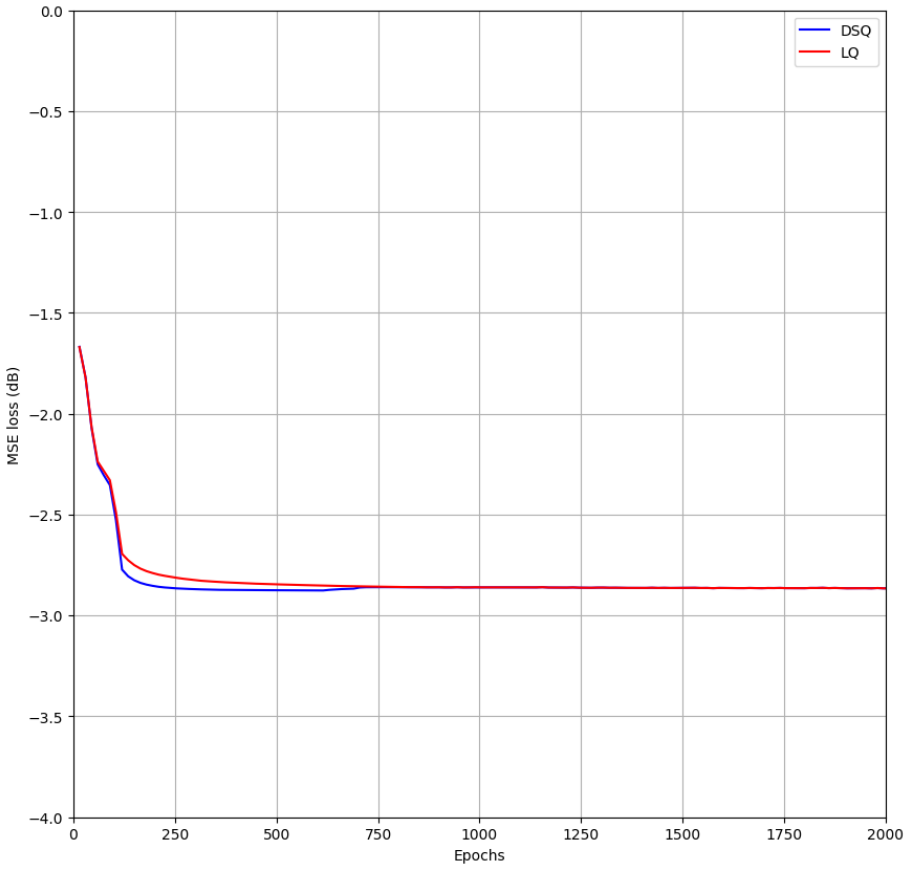
AE3



(a) minimizzazione MSE con DSQ



(b) Minimizzazione MSE con quantizzatore uniforme



(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 5.4: Convergenza dell'addestramento di un AE a 3 layer con una compressione a 4 bit

Confronto tra LBG e autoencoder

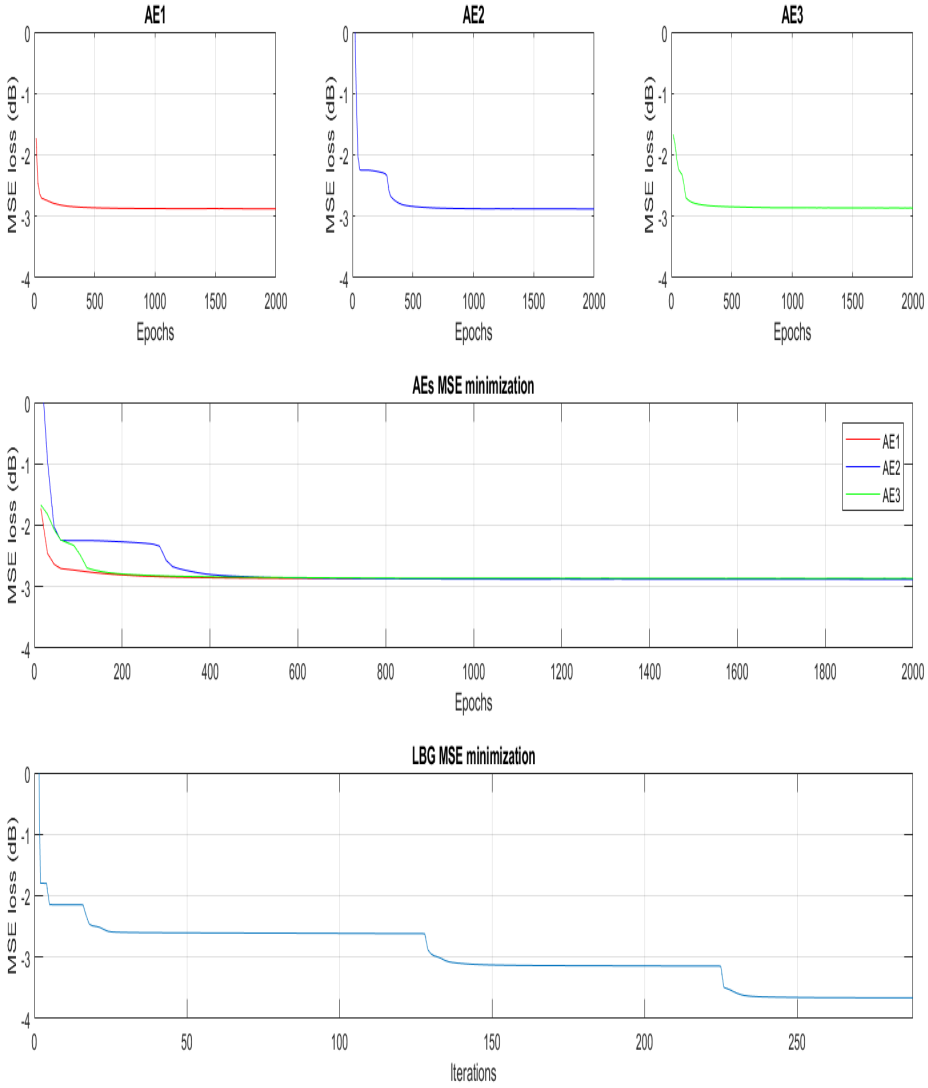
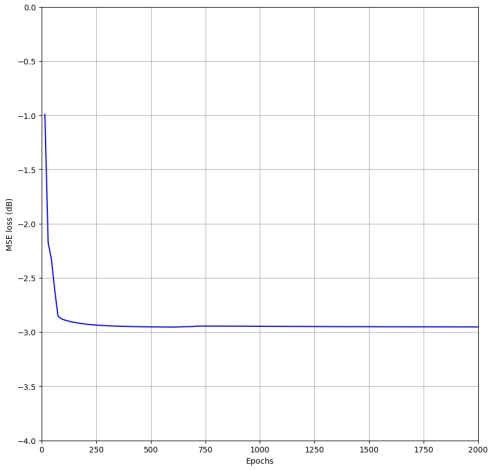


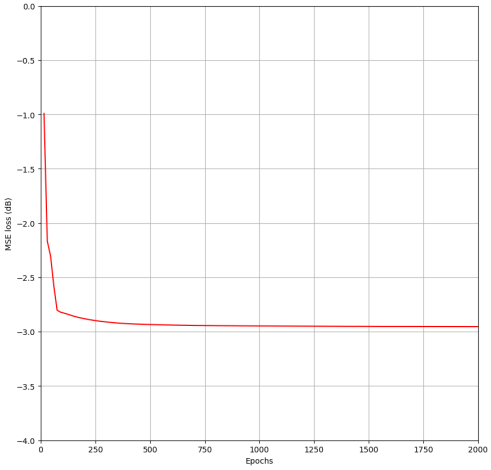
Figura 5.5: Minimizzazione MSE con compressione a 4 bit

5.2.2 Compressione a 6 bit

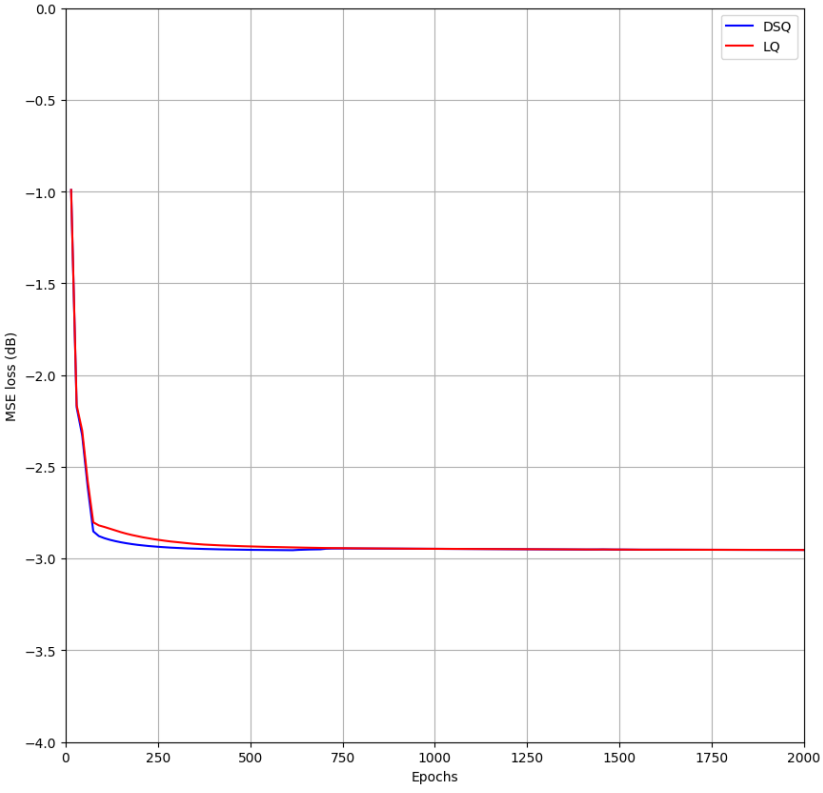
AE1



(a) minimizzazione MSE con DSQ



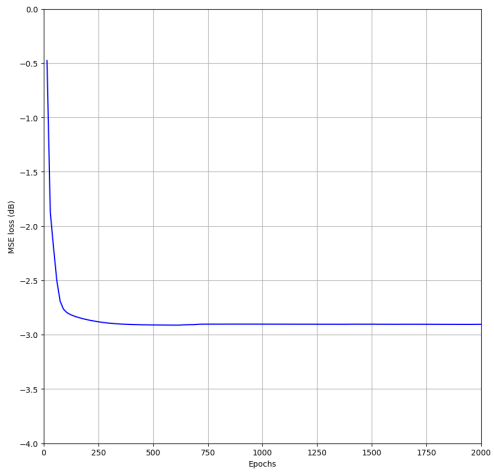
(b) Minimizzazione MSE con quantizzatore uniforme



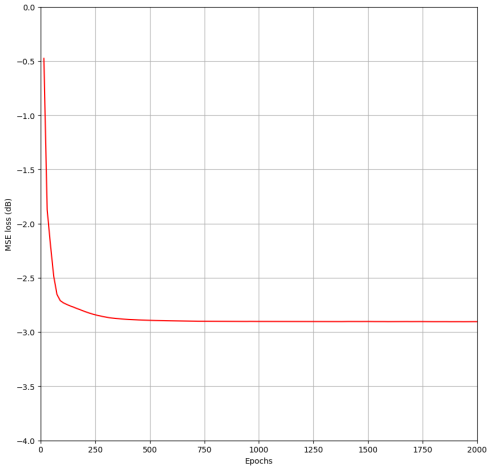
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 5.6: Convergenza dell'addestramento di un AE single layer con una compressione a 6 bit

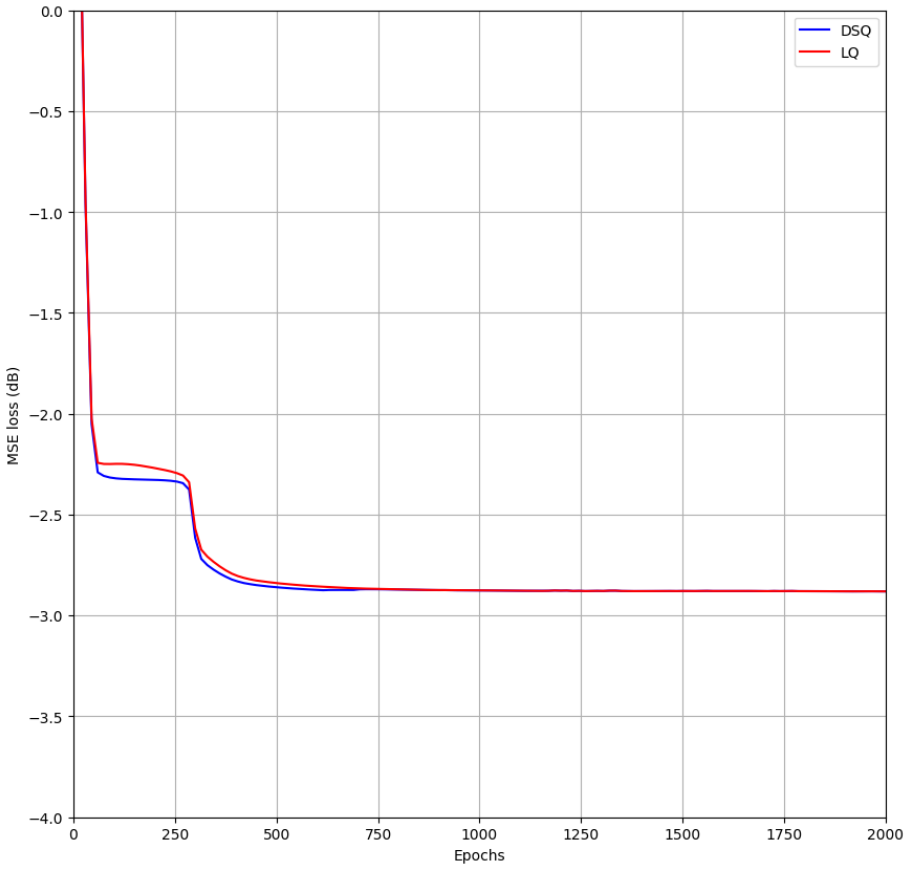
AE2



(a) minimizzazione MSE con DSQ



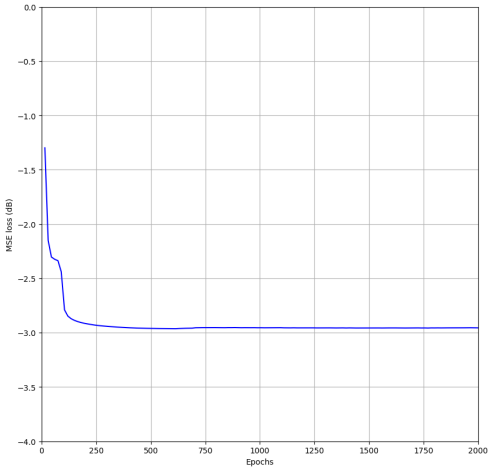
(b) Minimizzazione MSE con quantizzatore uniforme



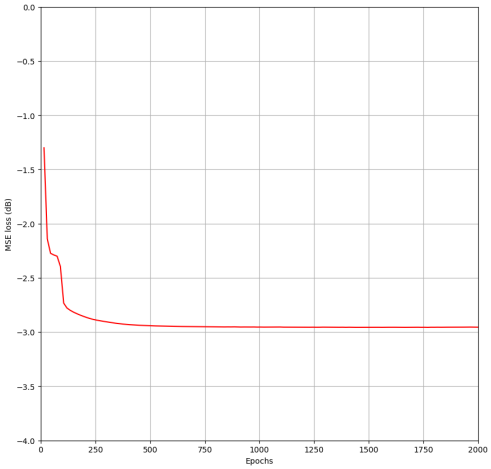
(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 5.7: Convergenza dell'addestramento di un AE a 2 layer con una compressione a 6 bit

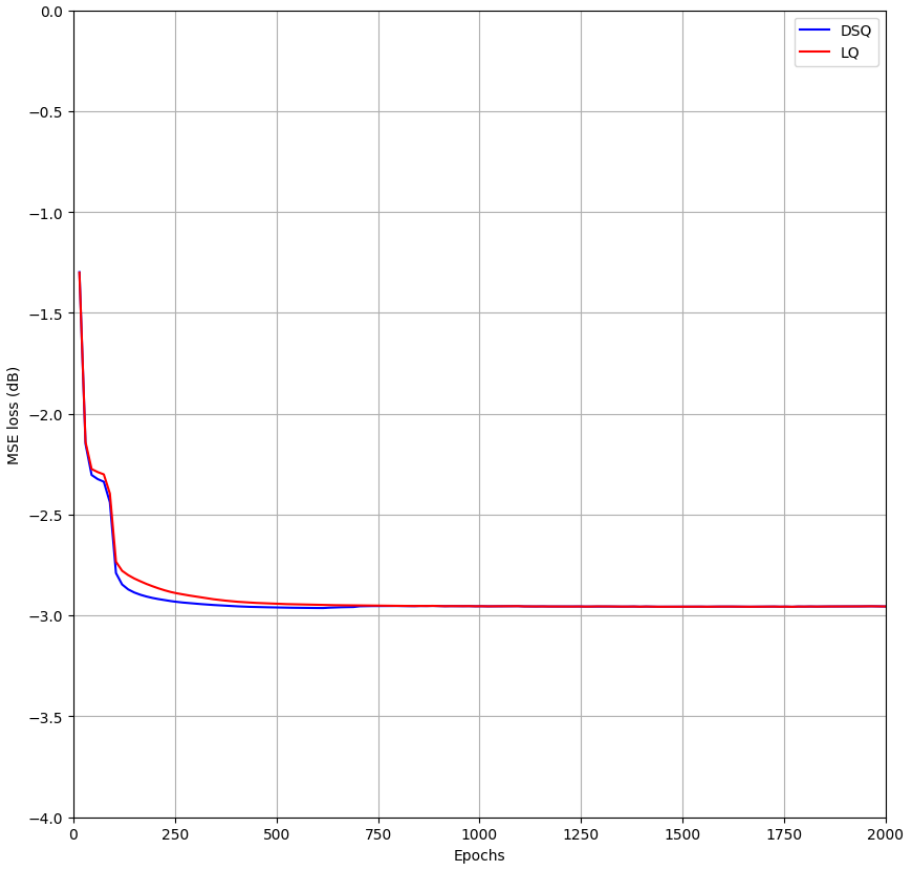
AE3



(a) minimizzazione MSE con DSQ



(b) Minimizzazione MSE con quantizzatore uniforme



(c) Minimizzazione MSE con quantizzatore uniforme e DSQ

Figura 5.8: Convergenza dell'addestramento di un AE a 3 layer con una compressione a 6 bit

Confronto tra LBG e autoencoder

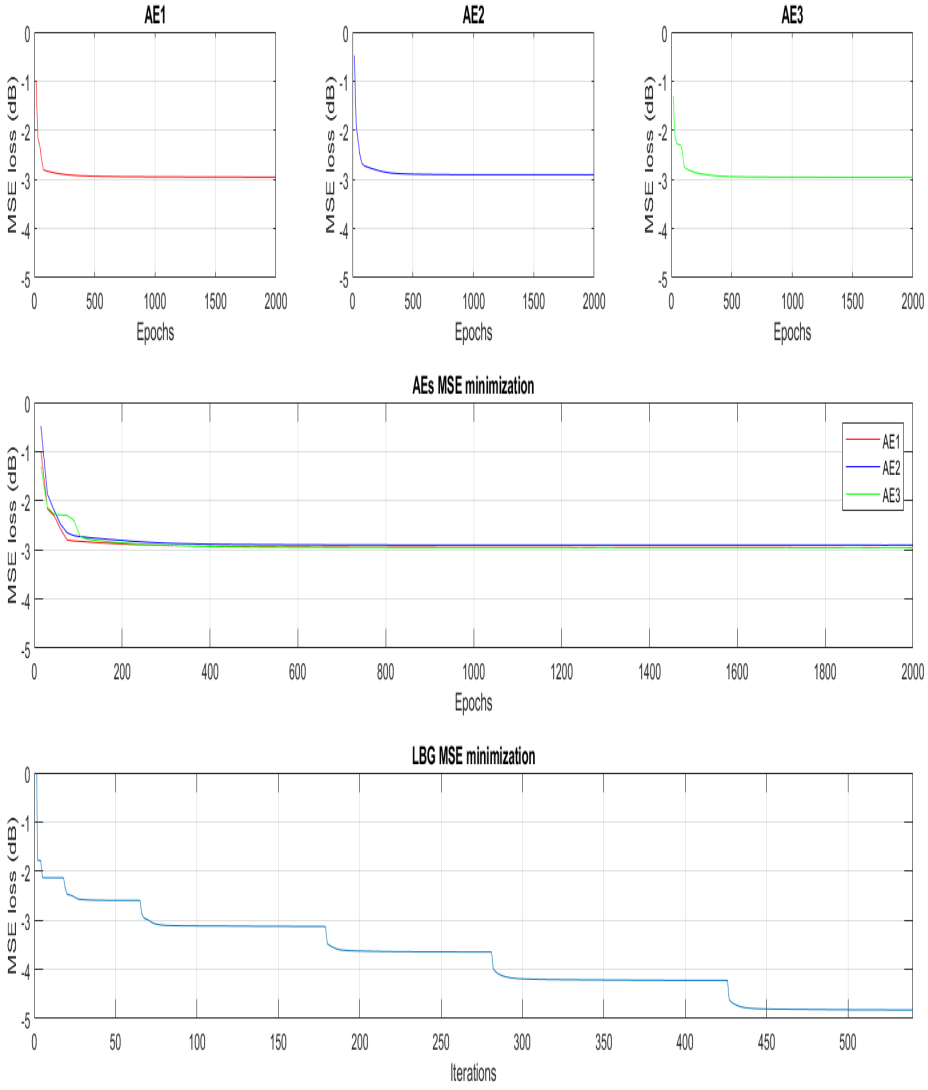


Figura 5.9: Minimizzazione MSE con compressione a 6 bit

5.2.3 Analisi analitica della convergenza

Modello	Bit di compressione	$\Delta_{MSE\,loss}$ ultime 300 epoche	Miglioramento % rispetto al MSE finale
$AE1$	4 bit	$7{\cdot}10^{-5}$	0.3 %
$AE2$	4 bit	$7{\cdot}10^{-5}$	0.5 %
$AE3$	4 bit	$4{\cdot}10^{-5}$	0.4 %
$AE1$	6 bit	$2{\cdot}10^{-6}$	0.3 %
$AE2$	6 bit	$6{\cdot}10^{-6}$	0.2 %
$AE3$	6 bit	$2{\cdot}10^{-5}$	0.6 %

Tabella 5.5: Miglioramento della minimizzazione del MSE nelle ultime 300 epoche

Dai grafici e dalla tabella, è possibile affermare che gli autoencoder siano giunti a convergenza, dato che con un incremento della pendenza p del quantizzatore DSQ pari a $20 \cdot 10^3$ i miglioramenti nella minimizzazione del MSE sono quasi irrilevanti.
Dai grafici relativi alla minimizzazione del MSE esposti sopra, si nota come l'addestramento con compressioni a 4 bit risulti particolarmente difficoltoso.
In particolare, dopo una prima "calibrazione" di pesi e bias, il DSQ impiega molte epoche prima di capire come modificare i punti che cadono sulle alzate. Ciò causa appunto la divergenza tra il grafico del DSQ e quello del quantizzatore che si può ad esempio vedere nelle zone centrali dei grafici relativi alla compressione a 4 bit.

5.3 Regioni di decisione

Per andare a graficare le regioni di decisione in maniera semplice, sono stati forniti sia agli autoencoder sia all'algoritmo LBG degli input bidimensionali.
L'autoencoder utilizzato a questo scopo è mostrato nella figura sottostante.

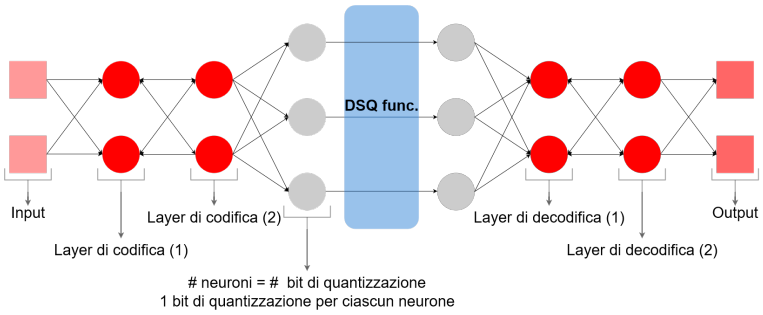


Figura 5.10: Struttura dell'autoencoder con input bidimensionale utilizzato per graficare le regioni di decisione con questo dataset

Si noti come con questo dataset è stata usata una struttura dell'autoencoder diversa rispetto a quella vista nella Sezione 4.3.
Questa scelta è dovuta al fatto che essendo la statistica di questo dataset più elaborata rispetto a quella vista al Capitolo 4, è conveniente massimizzare il numero di neuroni al layer interno.

5.3.1 Compressione a 3 bit

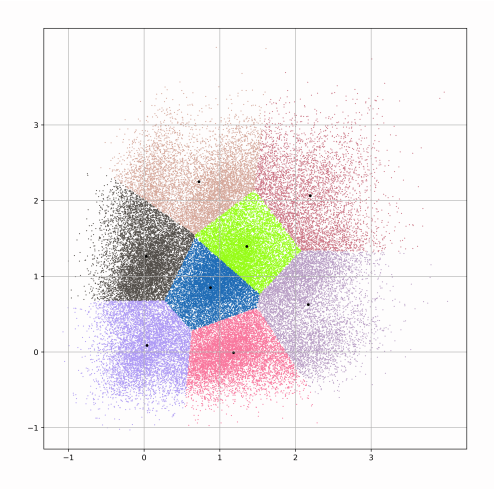
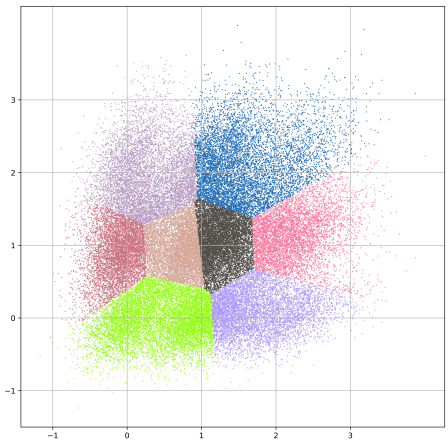



Figura 5.11: Regioni create, e relativi centroidi, dall'algoritmo LBG (a sinistra) e dall'autoencoder a 2 layer (a destra), con una compressione a 3 bit

Minimizzazione del mean-squared error:

- autoencoder a 2 layer: -9 dB;
- algoritmo LBG: -9.6 dB.

5.3.2 Compressione a 4 bit

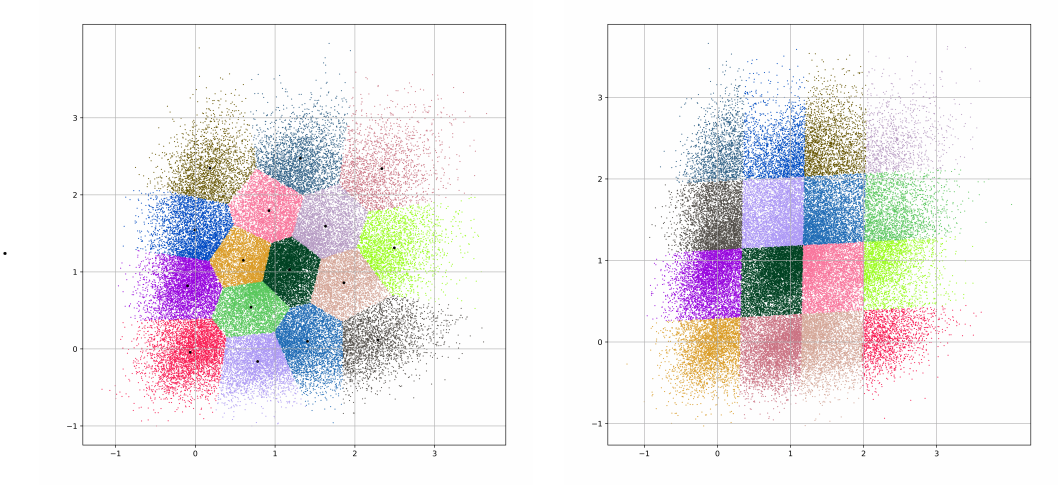


Figura 5.12: Regioni create, e relativi centroidi, dall'algoritmo LBG (a sinistra) e dall'autoencoder a 2 layer (a destra), con una compressione a 4 bit

Minimizzazione del mean-squared error:

- autoencoder a 2 layer: -12.2 dB;
- algoritmo LBG: -12.7 dB.

5.4 Stime asintotiche

Conoscendo le caratteristiche del dataset, oltre alla struttura e i parametri degli autoencoder utilizzati, è possibile fornire per ciascuno dei due metodi delle stime riguardo la complessità computazionale τ in termini di operazioni tra scalari e l'occupazione di memoria μ in bit viste nel Capitolo 2.

5.4.1 Complessità computazionale

Modello	$\boldsymbol{\tau}$ addestramento	$\boldsymbol{\tau}$ utilizzo
MSE loss con compressione a 4 bit		
Algoritmo LBG	$2.77 \cdot 10^9$	160
Autoencoder AE1	$6.10 \cdot 10^9$	44
Autoencoder AE2	$6.10 \cdot 10^9$	44
Autoencoder AE3	$1.72 \cdot 10^{10}$	124
MSE loss con compressione a 6 bit		
Algoritmo LBG	$5.18 \cdot 10^9$	640
Autoencoder AE1	$6.10 \cdot 10^9$	44
Autoencoder AE2	$6.10 \cdot 10^9$	44
Autoencoder AE3	$1.72 \cdot 10^{10}$	124
Regioni di decisione con compressione a 3 bit		
Algoritmo LBG	$4.90 \cdot 10^8$	16
Autoencoder AE1	$6.10 \cdot 10^9$	22
Autoencoder AE2	$6.10 \cdot 10^9$	22
Regioni di decisione con compressione a 4 bit		
Algoritmo LBG	$1.07 \cdot 10^8$	32
Autoencoder AE1	$6.10 \cdot 10^9$	22
Autoencoder AE2	$6.10 \cdot 10^9$	22

Tabella 5.6: Tabella riassuntiva per la complessità computazionale dei modelli

5.4.2 Occupazione di memoria

Modello	$\boldsymbol{\mu}$ [bit]
MSE loss con compressione a 4 bit	
Algoritmo LBG	$160S + 2F$
Autoencoder AE1	$68S + 24F$
Autoencoder AE2	$68S + 24F$
Autoencoder AE3	$158S + 34F$
MSE loss con compressione a 6 bit	
Algoritmo LBG	$640S + 2F$
Autoencoder AE1	$68S + 24F$
Autoencoder AE2	$68S + 24F$
Autoencoder AE3	$158S + 34F$
Regioni di decisione con compressione a 3 bit	
Algoritmo LBG	$16S + 2F$
Autoencoder AE1	$37S + 15F$
Autoencoder AE2	$37S + 15F$
Regioni di decisione con compressione a 4 bit	
Algoritmo LBG	$32S + 2F$
Autoencoder AE1	$37S + 15F$
Autoencoder AE2	$37S + 15F$

Tabella 5.7: Tabella riassuntiva per l'occupazione di memoria dei modelli

5.5 Considerazioni e osservazioni

Guardando innanzitutto ai risultati mostrati nella Sezione 5.1, si nota come, nel caso di una compressione a 2 bit, per ottenere con un autoencoder una qualità di ricostruzione uguale o migliore di quella ottenuta con l'algoritmo LBG risulta necessario utilizzare un deep autoencoder con almeno 3 layer (Figura 1.2).
Per quanto riguarda invece le altre compressioni, si riesce ad eguagliare la minimizzazione del MSE ottenuta con l'algoritmo LBG anche con un autoencoder a due layer o con un autoencoder lineare (Figure 1.1 e 1.3).
Per queste compressioni vale inoltre la medesima considerazione fatta alla Sezione 4.5, ossia che l'autoencoder lineare riesce ad ottenere una qualità di ricostruzione equivalente o migliore di quelle ottenute con i deep autoencoder.
Emerge quindi come all'aumentare del numero dei bit di quantizzazione progressivamente l'autoencoder lineare arrivi a eguagliare o migliorare la compressione effettuata dai deep autoencoder.
Una ragione dietro ciò sta nel fatto che aumentare il numero dei bit di quantizzazione equivale a rendere "più semplice" la compressione, in quanto la mappatura viene effettuata su un numero maggiore di regioni di decisione.
Analizzando ora le tabelle alla Sezione 5.4, si nota che l'algoritmo LBG ha una minore complessità computazionale di addestramento rispetto a tutti gli autoencoder utilizzati, mentre tendenzialmente gli autoencoder AE1 e AE2 ottengono una complessità di utilizzo minore di quella dell'algoritmo LBG.
Per quanto riguarda l'occupazione di memoria si ha che per i modelli aventi in input vettori a 10 dimensioni, gli autoencoder hanno una occupazione di memoria di molto minore rispetto a quella dell'algoritmo LBG.
Per quanto concerne i modelli aventi in input vettori a 2 dimensioni, si ha che l'algoritmo LBG ha una occupazione di memoria minore di quella degli autoencoder, in maniera particolarmente rilevante quando si considera una compressione a 3 bit.
Osservando infine le regioni di decisione mostrate nella Sezione 5.3 create dall'algoritmo LBG e dall'autoencoder, dalla minimizzazione del MSE si può dire che queste sono circa equivalenti in termini di qualità di ricostruzione dell'input, pertanto le differenze tra le regioni create dai due metodi sono dovute ai metodi stessi e al loro funzionamento.
Per quanto riguarda la compressione a 3 bit (Sezione 5.3.1), le regioni create sono simili nella forma e nella suddivisione dello spazio.
Guardando invece alle regioni relative alla compressione a 4 bit (Sezione 5.3.2),

in questo caso si nota innanzitutto che le forme delle regioni sono di molto diverse: le regioni create dall'algoritmo LBG sono esagonali mentre quelle create dall'autoencoder sono quadrilatere.
Analogamente a quanto osservato nella Sezione 4.5, emerge anche una diversa suddivisione dello spazio: mentre l'algoritmo LBG tende a creare regioni più piccole nella zona centrale e più grandi verso le zone esterne, le regioni dell'autoencoder sono tutte di circa uguali dimensioni, in accordo con la presenza del quantizzatore uniforme.
Si può quindi concludere, in maniera simile a quanto fatto nella Sezione 4.5, che anche per questo dataset risulta conveniente sotto tutti i punti di vista utilizzare un autoencoder lineare, ad eccezione del caso della compressione a 2 bit, nella quale risulta più conveniente sfruttare l'algoritmo LBG.

5.6 Confronto con i risultati ottenuti con i dataset di gaussiane correlate

La prima differenza tra i due dataset, che si riflette poi nelle differenze tra i risultati visti nei Capitoli 4 e 5, sta nella statistica dei simboli.
Mentre con il dataset di gaussiane correlate i simboli sono dati dalla correlazione di due gaussiane aventi gli stessi parametri, con il dataset di mixture gaussian i simboli sono dati dalla correlazione di due gaussiane aventi parametri differenti.
Questo porta, in primo luogo, ad avere una minimizzazione del MSE migliore con il dataset visto nel Capitolo 4.
In particolare, ciò è dovuto al fatto che le due gaussiane identiche che vengono correlate, a differenza di quelle del dataset del Capitolo 5, hanno una varianza maggiore, pertanto la distribuzione dei simboli sarà meno "concentrata" e di conseguenza la probabilità di assegnare un simbolo ad una regione di decisione non appropriata si ridurrà.
Si noti comunque come, nonostante la minimizzazione del MSE ottenuta al termine degli addestramenti sia differente tra i due dataset, la convergenza degli addestramenti segue lo stesso andamento nei due casi in esame. L'unica differenza apprezzabile è che con il dataset di mixuture gaussian le minimizzazioni del MSE ottenute con il DSQ e con il quantizzatore lineare durante l'addestramento tendono a coincidere, mentre con il dataset di gaussiane correlate si era notato che nelle prime epoche di addestramento si otteneva una minimizzazione del MSE molto migliore con il DSQ.
Guardando invece alle regioni di decisione create, notiamo intanto come la struttura dell'autoencoder usato per i due dataset sia differente (reference a figure).
Per il dataset del Capitolo 4 é sufficiente usare un solo neurone al layer centrale, avente un numero di bit pari al numero di bit di compressione utilizzati.
Per il dataset del Capitolo 5 risulta invece necessario andare a massimizzare il numero di neuroni al layer centrale, usando un numero di neuroni pari al numero di bit di compressione utilizzati, ciascuno avente un singolo bit di quantizzazione a disposizione.
La ragione di ciò risiede sempre nella maggiore complessità statistica dei simboli del dataset di mixture gaussian. Per la stessa ragione, anche le regioni di decisione relative ai due dataset sono differenti sia nella forma delle regioni stesse sia nella suddivisione dello spazio.
Si vede comunque che l'autoencoder in tutti i casi riesce a creare regioni molto

simili a quelle create dall'algoritmo LBG, a testimoniare la capacità di adattamento di questa rete neurale.
Infine, analizzando le complessità computazionali e le occupazioni di memoria, emerge che queste, essendo indipendenti dalla statistica dei simboli che compongono il dataset, sono le stesse per i due dataset (Sezioni 4.5 e 5.5).

Capitolo 6

Considerazioni e conclusioni finali

Dalle considerazioni effettuate nelle Sezioni 4.5, 5.5 e 5.6, emerge chiaramente che con entrambi i dataset studiati risulta sempre più conveniente utilizzare un AE single layer.
Si vede infatti come, tranne nello specifico caso della compressione a 2 bit con il dataset di due mixture gaussian correlate (Sezione 5.1), questa rete neurale riesce a produrre una ricostruzione equivalente o migliore rispetto a quella ottenuta con gli altri autoencoder e con l'algoritmo LBG, ottenendo inoltre in fase di addestramento ed utilizzo una complessità computazionale e una occupazione di memoria minore, tranne per quanto riguarda la complessità di addestramento con il dataset visto al Capitolo 5.
Possiamo quindi concludere che, con questi due specifici dataset aventi una statistica dei simboli originata da distribuzioni gaussiane, risulti nel complesso la scelta migliore utilizzare autoencoder lineari, confermando, nonostante la presenza di una funzione di quantizzazione nel layer centrale, quando esposto in [3].

Bibliografia

[1] I.Goodfellow,Y.Bengio,and A.Courville, *DeepLearning*, `www.deeplearningbook.org`, MIT Press, 2016.

[2] R.Gong et al., Differentiable Soft Quantization: Bridging Full-Precision and Low-Bit Neural Networks, 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 4851-4860, doi: 10.1109/ICCV.2019.00495.

[3] Harald Steck and Dario Garcia Garcia, On the Regularization of Autoencoders, arXiv:2110.11402v1, 2021

[4] Brian Dolhansky, 2014, Artificial Neural Networks: Matrix Form, `www.briandolhansky.com/blog/2014/10/30/artificial-neural-networks-matrix-form-part-5`

[5] J.Shanbehzadeh and P.O. Ogunbona, On the computational complexity of the LBG and PNN algorithms in IEEE Transactions on Image Processing, vol.6, no.4, pp.614-616, April 1997, doi:10.1109/83.563327.

[6] Buitinck L., Louppe G., Blondel M., Pedregosa F., Mueller A., Grisel O., Niculae V., Prettenhofer P., Gramfort A., Grobler J., Layton R., Vanderplas J., Joly A., Holt B., Varoquaux G., *API design for machine learning software: Experiences from the scikit-learn project*, 2013.

[7] *Convolutional Neural Networks*, `http://deeplearning.net/tutorial/lenet.html`, last consultation: 07/01/2020.

[8] *PyTorch framework*, `pytorch.org`, last consultation: 01/09/2022.