
Università degli Studi di Salerno
Dipartimento di Informatica

---

# Lossless Dataset Compression

*M. Barowsky, A. Mariona, F. P. Calmon*
*(ICASSP 2021)*

Progetto Compressione Dati

Anno accademico 2021/2022

Docente
Prof. Bruno Carpentieri

Studenti
Alessio Ambruoso, 0522500985
Silvio Esposito, 0522501031
Paolo Panico, 0512105254

# Sommario

# 1 Introduzione

La compressione senza perdita di dati è un problema di notevole interesse, soprattutto per la ricerca scientifica, ove bisogna memorizzare ed inviare grandi collezioni di dati. Lo scopo del paper[1] scritto da *Barowsky*, *Mariona* e *Calmon* è quello di dimostrare che gli algoritmi di codifica sorgente esistenti sono subottimali per i problemi dove il posizionamento non ha importanza, e che è possibile migliorare notevolmente il bitrate di codifica se permettiamo alla compressione di un dataset originale di decomprimere in una qualsiasi permutazione dei dati.

Esistono vari casi d'uso in cui ciò è utile:

- grandi insiemi di dati utilizzati per l'addestramento di modelli nel machine learning;
- collezioni di foto;
- archivi di ricerca;
- directory di file;
- database statistici tabulari;
- eccetera

Gli autori si concentrano sul primo elemento del sopracitato elenco, i dataset.
Definiscono la compressione dei dataset come il problema di creare codifiche a partire dalla sorgente che permettano di ricostruire i dataset fino ad una permutazione delle loro voci iniziali, ed introducono un quadro teorico ed un algoritmo pratico per la compressione senza perdita di dati. I contributi principali sono quattro:

1. Presentazione di una nuova formulazione teorica per il problema della compressione dei dataset come un'estensione della canonica compressione lossless.
2. Formulazione e dimostrazione di un teorema che riduce il compito di comprimere un dataset a quello di comprimere una struttura sui dati che sia invariante rispetto alla permutazione.
3. Nuova procedura di codifica predittiva per il dataset: riordina il dataset per minimizzare le differenze tra gli elementi e addestra un modello predittivo alle caratteristiche degli elementi adiacenti.
4. Costruzione del pacchetto software che implementi la codifica predittiva per la compressione.

# 2 Idea degli autori

L'idea degli autori descrive una metodologia per la compressione lossless dei dataset. La procedura estende le tecniche di codifica predittiva presenti in letteratura e utilizzate, ad esempio, in `JPEG-LS`. L'approccio ha tre passaggi fondamentali:

- Riordino degli elementi del dataset;
- Codifica predittiva;
- Codifica entropica.

Si desidera riordinare le voci del dataset per massimizzare l'accuratezza della codifica predittiva, che è più efficace quando la stringa di errore del predittore è concentrata intorno allo 0, ovvero quando il modello funziona bene. Gli esperimenti confermano che l'ordinamento per ridurre la distanza euclidea tra le voci successive del dataset determina una compressione migliore rispetto all'ordinamento casuale. Per raggiungere questo obiettivo, gli autori hanno sviluppato un metodo chiamato ordinamento `kNN-MST`, descritto di seguito. (*k-Nearest Neighbor Minimum Spanning Tree*).

Innanzitutto, occorre notare che riordinare gli elementi del dataset per massimizzare la somiglianza tra di essi equivale a trovare un percorso Hamiltoniano minimo sul grafo dei *nearest-neighbors* (Figura 1).

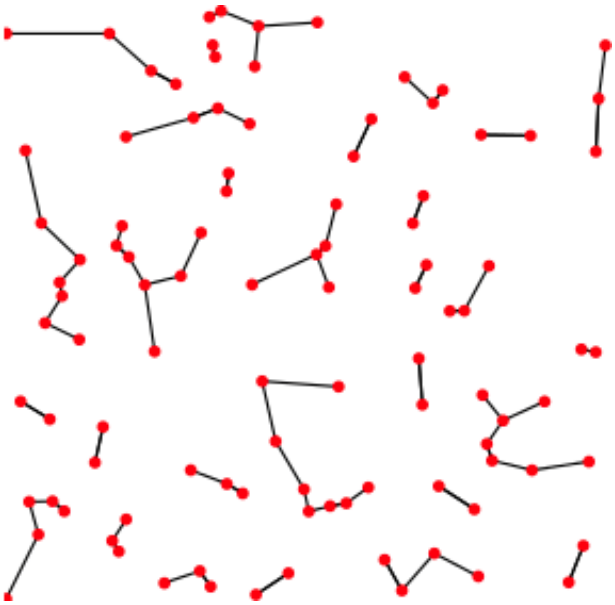

Figura 1: Nearest neighbor graph

Il calcolo di un attraversamento ottimale globale di un neighbor graph è il problema del *Metric Traveling Salesman Problem*, che sebbene sia *NP-hard*, esistono buone

approssimazioni. Semplicemente attraversando un albero di copertura minimo (MST) del *neighbor graph* completo si ottiene una soluzione che è entro il doppio del costo ottimale, che sappiamo essere una costante trascurabile ai fini della complessità computazionale. Naturalmente, il calcolo dell'MST sul *neighbor graph* completo di un grande dataset può essere computazionalmente impossibile.

Facciamo quindi la seguente osservazione della teoria dei grafi:

$$Sia\ G\ un\ grafo\ V, E.$$
$$Per\ ogni\ k\ che\ va\ da\ uno\ alla\ cardinalità\ dell'insieme\ dei\ vertici,$$
$$se\ kNN(G) \text{è}\ connesso,$$
$$allora\ un\ MST\ di\ kNN(G) \text{è}\ un\ MST\ di\ G$$

Questo lemma ci permette di trovare $k$ tale che il grafo kNN di $G$ sia completamente connesso e quindi calcolare un riordino del dataset attraversando l'MST (Figura 2). Il $k$ minimo necessario per la connettività kNN nelle simulazioni era $O(log(n))$, il che permette di impostare il $k$ iniziale uguale al logaritmo naturale di $n$ e aumentare solo se il grafo kNN risultante è disconnesso.

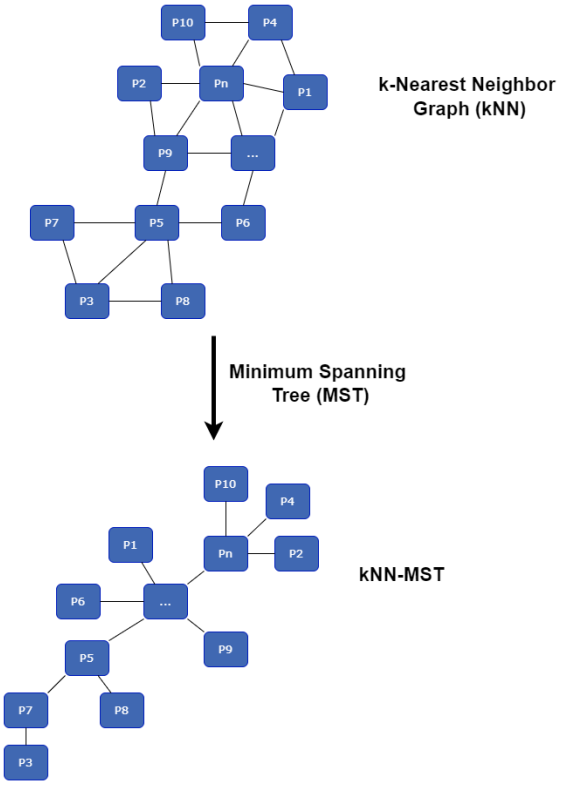


Figura 2: esempio grafico kNN-MST

L'idea centrale nella codifica predittiva per la compressione del dataset è quella di utilizzare le caratteristiche del campione corrente e dei campioni vicini prodotti

dall'ordinamento `kNN-MST` del dataset. Per addestrare il predittore, estraiamo quindi una stringa di contesto per pixel da ogni immagine nel dataset, in modo da migliorare la previsione quando sono presenti più immagini simili.

L'accuratezza della funzione predittiva è cruciale per ridurre al minimo l'entropia nella stringa di errore e migliorare la velocità di compressione.

Innanzitutto, viene generata la stringa di contesto per ciascun pixel nel dataset, vengono quindi usate queste informazioni per addestrare il modello predittivo su tutti gli elementi.
La rete viene addestrata fino a quando non va in *overfit*, perché non è necessario considerare come l'accuratezza si generalizzi ai campioni oltre il dataset.
Successivamente il modello viene applicato a ciascun pixel registrando l'errore di predizione oppure 0 se la predizione è corretta.

Infine, la stringa di errore prodotta viene compressa con la codifica `Huffman`.

Tutto questo è riassunto nell'algoritmo in Figura 3.

**Algorithm 1** Predictive Dataset Compression

**Input:** a dataset $\mathcal{D} = \{x_i\}_{i=1}^{n}$
$(x_{i_j})_{j=1}^{n} = \text{KNN-MST-REORDER}(x^n)$
$\{\{\text{context}_a, a\}_{a \in x_t}\}_{t=1}^{n} = \text{EXTRACTCONTEXT}(\mathcal{D})$
$h = \text{TRAINPREDICTIVEMODEL}(\{\{\text{context}_a, a\}_{a \in x_t}\}_{t=1}^{n})$
error_string = [ ]
**for** $x_t$ in $\mathcal{D}$ **do**
  **for** $a$ in $x_t$ **do**
    error_string += $h(\text{context}_a) - a$
  **end for**
**end for**
HUFFMANENCODE(metadata, f.params, error_string)

Figura 3: Algoritmo in pseudocodice di *Barowsky et al.*

# 3 Implementazione

Il nostro compito era quello di studiare il paper di *Barowsky et al.* e di implementare il sistema di compressione proposto.
Per realizzare l'implementazione abbiamo scelto di utilizzare *Python*, come suggerito anche dagli autori stessi.
Di seguito cercheremo di descrivere il sistema da noi implementato, facendo alcune considerazioni e raccontando i problemi incontrati durante lo sviluppo.

Abbiamo realizzato uno script *Python* denominato `DatasetCompressor.py` eseguibile da linea di comando. Il file prende in input la modalità di esecuzione che può essere:

- `train`,
- `encode`,
- `decode`,

ed un secondo parametro che varia in base al primo.

In caso venga eseguito in modalità `train`, lo script necessita sia della posizione del dataset adibito al *training* sia di quella della parte di dataset adibita al *testing*.

Nel caso che invece venga eseguito in modalità `encode`, lo script necessita della posizione del dataset da comprimere. In questa modalità è possibile inserire sia la posizione dell'intero dataset, ma anche la posizione della parte di *training* e quella di *testing* che poi lo script provvederà ad unire e preparare per la compressione.

Infine, nel caso venga eseguito in modalità `decode`, lo script necessita della posizione del file compresso.

La nostra implementazione utilizza sia per i dataset che per il file compresso il formato `.npy`, tipico di *Python*, che rappresenta un file di array `NumPy`.

## 3.1 Riordino del dataset

Seguendo il processo degli autori, per effettuare il riordino del dataset abbiamo per prima cosa calcolato il kNN-MST sul dataset.

Per calcolare il *neighbor graph* abbiamo utilizzato una funzione di sklearn[2] che permette di settare la modalità di creazione dell'albero, e la metrica utilizzata per il calcolo di quella modalità. Nel nostro caso abbiamo scelto la modalità basata sulla distanza e la metrica euclidea per il calcolo.

Una volta ottenuto il kNN dobbiamo applicare l'algoritmo MST su questo grafo e lo abbiamo fatto tramite la funzione minimum_spanning_tree di scipy[3] che prende in input un grafo connesso e restituisce un MST.

Così facendo abbiamo ottenuto il nostro kNN-MST. Adesso bisognava riuscire a trovare il path minimo per percorrere tutti i nodi di questo grafo, in modo da rendere vicini i nodi più simili e quindi ordinarli secondo la distanza euclidea.

Gli autori consigliavano di eseguire il problema MTSP (*Metric Traveller Salesman Problem*) per risolvere questo passo.

Purtroppo, con dataset di grandi dimensioni, questo algoritmo occupa una grande quantità di spazio e risorse. Abbiamo provato ad utilizzare diversi algoritmi, sia deterministici che probabilistici ed anche varie tecniche, come quella basata sulla programmazione dinamica, una delle migliori in termini di performance. Purtroppo, non siamo riusciti ad effettuare in nessun modo l'algoritmo TSP sul dataset.

Dopo varie ricerche e vari studi, siamo giunti alla conclusione che avendo a disposizione un MST e non necessitando di dover tornare al nodo iniziale dopo aver visitato l'ultimo nodo, potessimo sfruttare un algoritmo DFS che avrebbe dato gli stessi risultati di risolvere TSP.

Con questa modifica siamo riusciti a calcolare un path ottimale del grafo prodotto in precedenza in tempi quasi nulli e senza sprecare risorse. Per il calcolo della DFS abbiamo utilizzato la funzione dfs_preorder_nodes di NetworkX[4] che prende in input il grafo su cui eseguire la ricerca e il nodo di partenza dello stesso.

Al termine di questi processi abbiamo potuto finalmente riordinare il dataset originale sfruttando il path ottenuto dalla DFS.

```
#KNN-MST
knn = kneighbors_graph(train_X, width, mode='distance', metric='euclidean')
knn_mst= minimum_spanning_tree(knn)

#TSP
g= nx.from_scipy_sparse_matrix(knn_mst)
path= nx.dfs_preorder_nodes(g,0)
tourOpt=list(path)

#REORDER DATASET
df= pd.DataFrame(train_X)
newdf=df.loc[tourOpt]
newTrain= newdf.to_numpy()
```

Figura 4: Codice relativo a kNN-MST, TSP e riordino del dataset

## 3.2 Training del modello

Per il *training* del modello abbiamo utilizzato la libreria `keras`[5] sfruttando i layer forniti e creando una rete neurale.

Abbiamo utilizzato il layer `RNN` a cui abbiamo fornito una cella *GRU* di 128 hidden layers e il dataset in taglie da 250.

Per comprendere meglio ciò che descriveremo, diamo delle veloci definizioni necessarie.

Una `RNN` (*Recurrent Neural Network*), è una classe di rete neurale artificiale che include neuroni collegati tra loro a formare un loop. Tipicamente sfrutta l'output dello strato superiore come input per lo strato successivo.

Una cella *GRU* è paragonabile ad una memoria a lungo termine con un parametro che indica il punto dopo il quale gli è permesso di resettare la memoria. Viene spesso sfruttata insieme alle reti `RNN`, poiché permette di conservarne gli output elaborandoli attraverso pesi e biases, prima del prossimo input.

`Adam` è una tecnica di ottimizzazione stocastica basata sulla riduzione del gradiente che utilizza una stima adattiva ad ogni passo basata sui pesi forniti.

La *Cross-Entropy* è una misura di performance dei modelli il cui output è una probabilità compresa tra 0 e 1. Questo valore aumenta quanto la probabilità di predizione diverge dal valore reale.

Nel nostro modello, la cella *GRU* prende in input il valore 128 che rappresenta il numero di neuroni o hidden layers che si intendono utilizzare. A questo punto la RNN sfrutta i 128 neuroni per dare differenti soluzioni al problema dato in input. Nel nostro caso, non avendo appositamente dato in input delle label da associare al dataset, il problema da risolvere è esattamente quello di ricordare il dataset.

Una volta che la rete precedente dà in output le differenti previsioni, le compariamo con quelle corrette e modifichiamo il peso del contributo di ogni neurone nella decisione finale a seconda di quanto sia stato preciso, sfruttando come tecnica di ottimizzazione Adam e come tecnica di *loss* la Cross-Entropy.

Infine, non resta che calcolare l'*accuracy* e la *loss* per l'epoca corrente e ripetere il tutto per un numero sufficiente di epoche, nel nostro caso 80 epoche sono più che sufficienti per avere una buona *accuracy*.

L'obiettivo della nostra rete, come già accennato, è quello di memorizzare il dataset, in modo da ricordare la sequenza di pixel di ogni immagine e permettere quindi una compressione maggiore. Questo obiettivo, richiede che le previsioni della rete siano deterministiche. Infatti, noi non salviamo la rete addestrata come si farebbe di solito, ma conserviamo lo stato della rete nella sua epoca migliore. Questo ci permette di rieseguirla con gli stessi parametri ogni volta, ottenendo sempre lo stesso output.

Al termine dell'addestramento con il dataset MNIST abbiamo ottenuto un *accuracy* di circa il 95.434%

Al seguito mostriamo i grafici singoli delle quattro performance del predittore:
*Train Accuracy*, *Test Accuracy*, *Train Loss*, *Test Loss*.

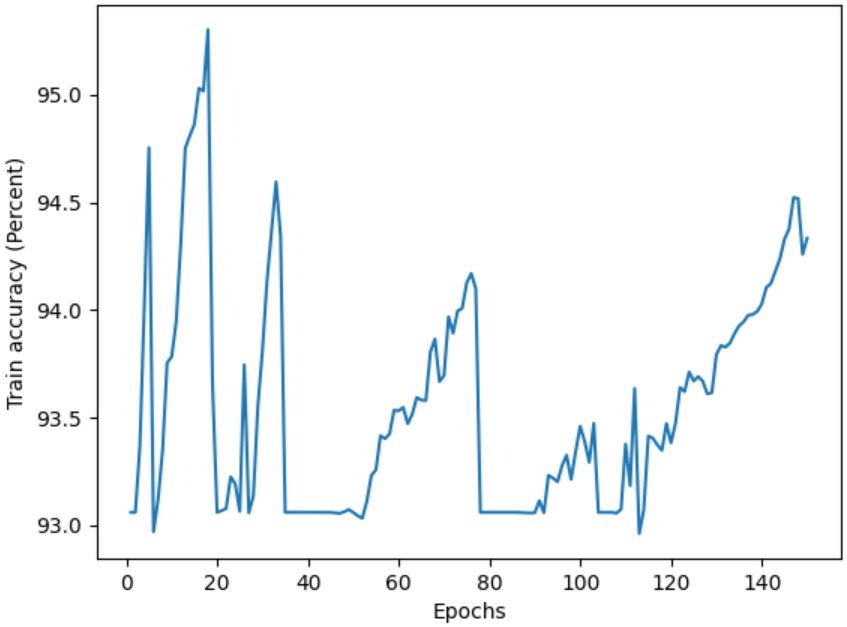


Figura 5: Grafico di *Train Accuracy* sulle epoche

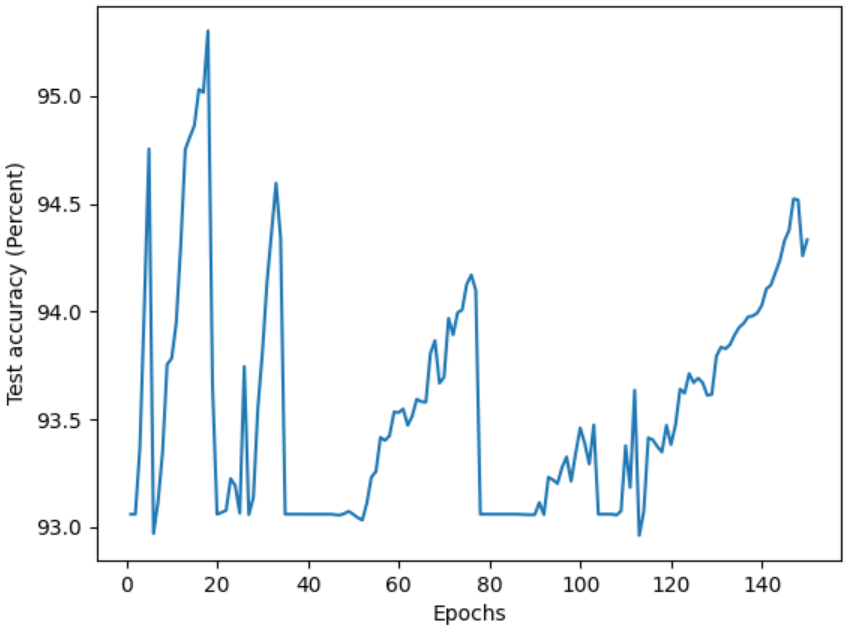


Figura 6: Grafico di *Test Accuracy* sulle epoche

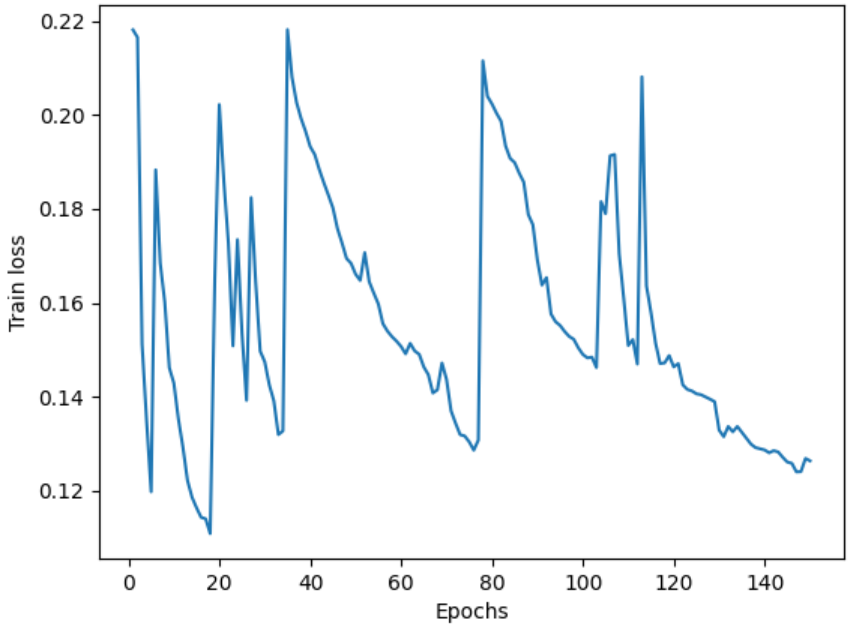


Figura 7: Grafico di *Train Loss* sulle epoche

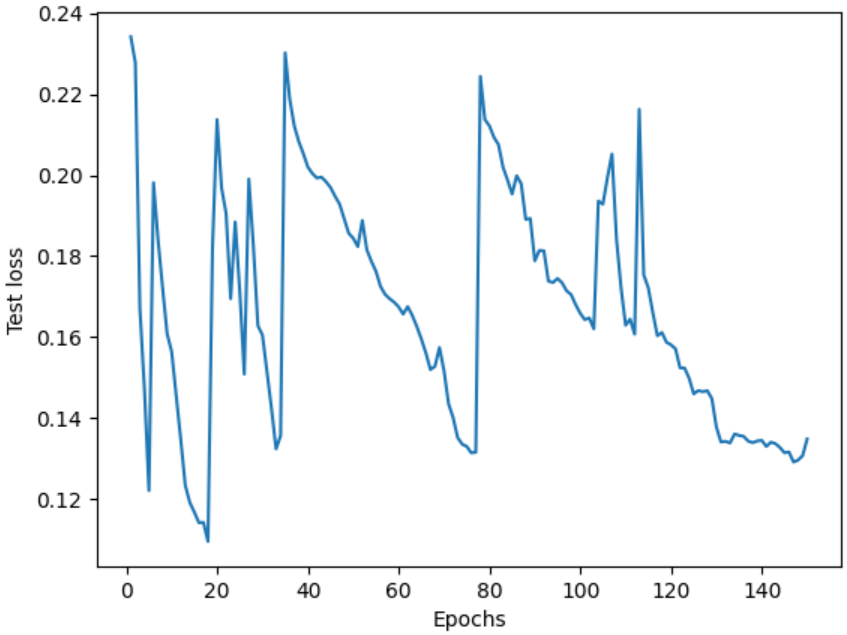


Figura 8: Grafico di *Test Loss* sulle epoche

Adesso mostriamo un confronto tra l'*Accuracy* del *Training* e del *Testing*.

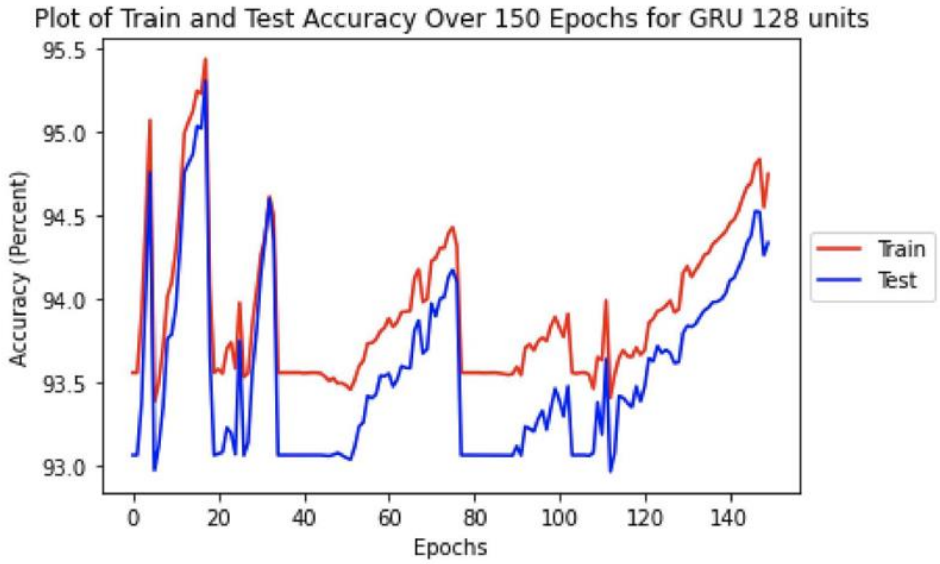


Figura 9: Grafico di *Train Accuracy* e relativo *Test Accuracy* sulle epoche

Qui è invece riportato un confronto tra le *Loss* del *Training* e dell'*Accuracy*.

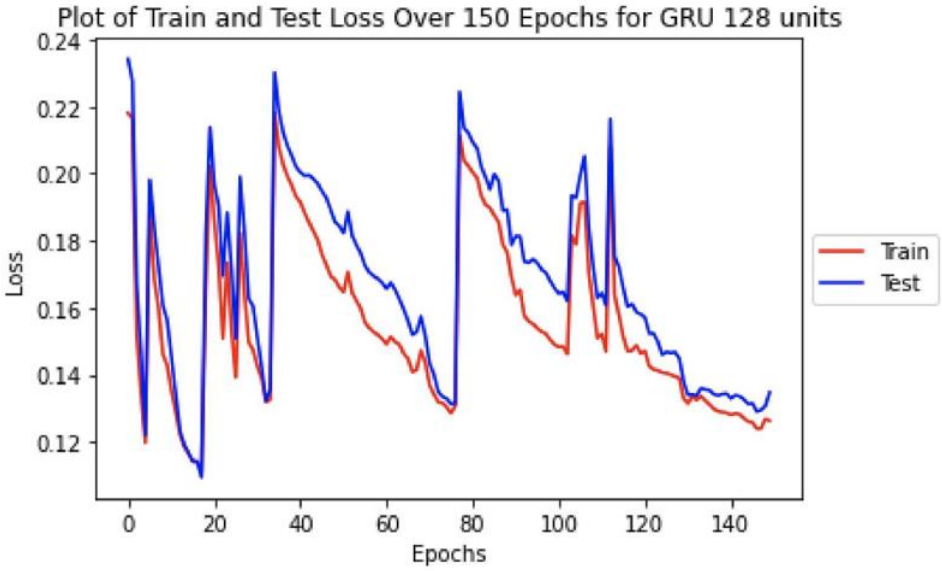


Figura 10: Grafico di *Train Loss* e relativa *Test Loss* sulle epoche

A dimostrazione del fatto che, all'aumentare dell'*Accuracy* si ha una corrispondente diminuzione delle *Loss*, i seguenti due grafici:

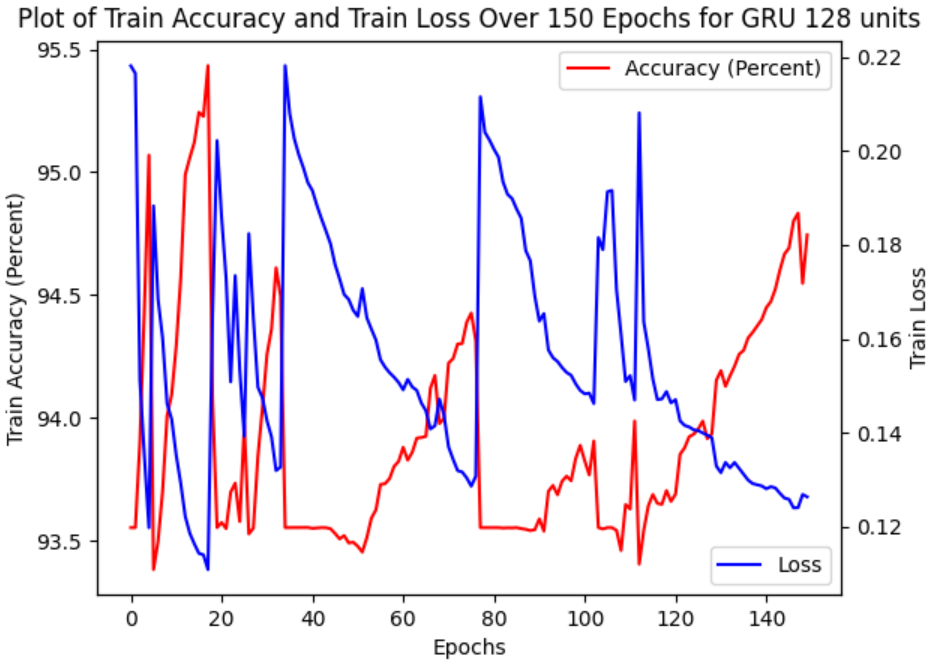


Figura 11: Grafico di *Train Accuracy* e *Train Loss* sulle epoche

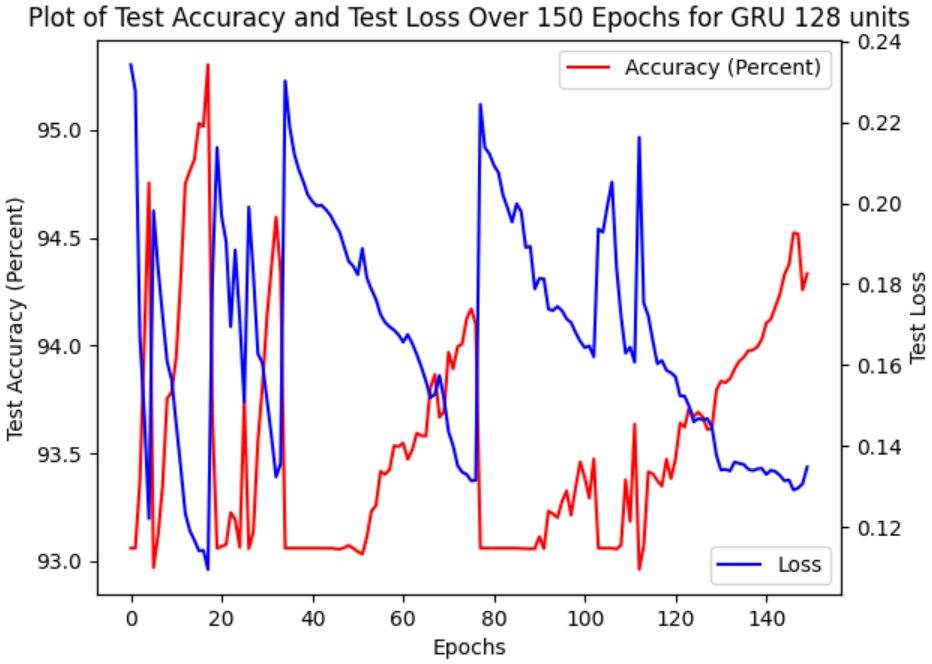


Figura 12: Grafico di *Test Accuracy* e *Test Loss* sulle epoche

## 3.3 Predizione del dataset

Per la predizione del dataset abbiamo sfruttato il migliore stato dell'addestramento precedentemente illustrato. La rete per la predizione è costituita dagli stessi *layer* di quella per il *training*, con l'unica differenza che in questo caso il nostro input è di 1 pixel, il primo pixel di ogni immagine del dataset.

La rete prende in input questa maschera di dimensione $1x1$ e restituisce in output il prossimo pixel della sequenza che sfrutterà poi come input per il prossimo passo.

Per ogni immagine abbiamo quindi un numero di predizioni uguali alle dimensioni dell'immagine stessa meno 1. Infatti, solo il primo pixel è stato fornito manualmente alla rete.

Come abbiamo illustrato per la fase di *training* l'obiettivo della rete è quello di memorizzare il dataset, questo vuol dire che nella fase di predizione è poco importante l'input che viene fornito alla rete poiché essa si comporterà sempre allo stesso modo.

Dovendo necessariamente dare un input alla rete per ogni immagine, abbiamo scelto di fornire il primo pixel poiché è un valore piccolo e semplice da conservare come informazione.

Al termine di questo processo disponiamo di un dataset predetto che provvediamo a confrontare con quello originale per calcolare una stringa di errore ed avremo anche una lista di pixel che rappresenta il primo pixel per ogni immagine del dataset.

Per calcolare la stringa di errore facciamo questo controllo:

- Se il pixel predetto è uguale a quello originale, allora aggiungiamo uno 0 alla stringa;
- Se il pixel predetto è diverso da quello originale, allora aggiungiamo alla stringa la differenza tra il pixel dell'immagine originale e quello dell'immagine predetta

## 3.4 Codifica

In questa fase siamo in possesso della nostra stringa di errore e la lista dei pixel.
Secondo l'idea fornita dagli autori dovremmo comprimere la stringa di errore con `Huffman`, infatti, più il predittore è stato preciso, più questa stringa conterrà degli 0 che permetteranno una compressione migliore.

Per la codifica `Huffman` abbiamo utilizzato il modulo `dahuffman`[6].

Per prima cosa abbiamo provveduto ad addestrare il modello di `Huffman` sulla nostra stringa di errore costruendone quindi la tabella di codifica.

Successivamente abbiamo concatenato la stringa di errore alla lista dei pixel e abbiamo compresso il risultato.

Abbiamo poi aggiunto alcune meta informazioni alla nostra compressione, quali il numero di immagini nel dataset, la larghezza e l'altezza dei campioni, la lunghezza della stringa di errore e la tabella di codifica creata.

Le prime tre informazioni, quali il numero di campioni, larghezza e altezza, ci serviranno per ridistribuire i valori della stringa di errore per ricreare il dataset.

La lunghezza della stringa di errore ci servirà per separare dal file compresso la stringa di errore dalla lista dei pixel ed infine la tabella di codifica ci servirà anche per la decodifica.

Al termine di queste operazioni abbiamo salvato la nostra compressione con le informazioni associate all'interno di un file `.npy`.

```
error_string, pixels= Predict(newTrain, width, height, MODE)

print("end prediction")

#HUFFMAN CODING
errors=np.array(error_string)
errors=errors.reshape(-1)
length= errors.shape[0]
codec=HuffmanCodec.from_data(errors)
all_err= np.concatenate((errors, pixels),0)
encoded=codec.encode(all_err)

#meta informations
meta=[n_samples, width, height, length, codec]
#save to file 0:meta(n_samples, width, height, errorLength) 1:encoded(errors, firstPixels)
toSave= np.array([meta, encoded], dtype=object)
np.save('datasetEncoded.npy', toSave)
```

Figura 13: Codice relativo alla predizione e alla codifica `Huffman`

# 3.5 Decodifica

Abbiamo provveduto a realizzare anche un sistema di decodifica che prendendo in input il file compresso restituisse il dataset originale senza nessuna perdita di informazione.

Il sistema di decodifica sfrutta lo stesso procedimento di quello di codifica.

Per prima cosa, sfruttando il file in input e le informazioni in esso contenute decomprime i dati e separa la stringa di errore dalla lista dei pixel che poi ridistribuisce secondo le dimensioni del dataset.

Una volta preparati entrambi i valori, passa la lista dei pixel al predittore come input per la rete e la stringa di errore per il confronto.

Il predittore opererà allo stesso modo di prima, l'unica differenza riguarderà il processo per calcolare il dataset originale che sommerà il valore del pixel nella stringa di errore a quello predetto, per ottenere nuovamente il valore originale, infatti:

- Se il pixel nella stringa di errore è uguale a 0, ciò significa che il valore predetto dalla rete era giusto e sommandolo a 0, si ottiene nuovamente il valore giusto;
- Se il pixel nella stringa di errore è diverso da 0, significa che il valore predetto dalla rete era errato e abbiamo calcolato l'errore come:
  $pixelOriginale - pixelPredetto = errorePixel$.
  Segue quindi che: $pixelOriginale = errorePixel + pixelPredetto$.

Come nella fase di codifica, una volta ottenuto il dataset originale, lo salviamo in un file di tipo `.npy`.

```
#LOAD AND DECODE DATASET
dataset= np.load(TRAIN_PATH, allow_pickle=True )
meta=dataset[0]
encoded= dataset[1]
codec=meta[4]
all_dataset=codec.decode(data= encoded)
errors= all_dataset[:meta[3]]
pixels= all_dataset[meta[3]:]

#PREPARE FOR PREDICTION
errors= np.array(errors)
pixels= np.array(pixels)
errors= errors.reshape((meta[0], meta[1]*meta[2]))
pixels= pixels.reshape((meta[0], 1))

#PREDICT ORIGINAL DATASET
originalImages, _= Predict(errors, meta[1], meta[2], MODE, pixels)
newDataset= np.array(originalImages)
newDataset= newDataset.reshape((meta[0], meta[1], meta[2]))

#SAVE MODEL
np.save('dataset.npy', newDataset)
```

Figura 14: Codice relativo al caricamento, decodifica e predizione del dataset

# 4 Risultati e confronto

Abbiamo testato il nostro modello sul dataset MNIST che ha una dimensione di 52.3MB.

Nonostante il nostro predittore non sia molto preciso, portando quindi ad una stringa di errore con molti valori che se pur si avvicinano allo 0 non lo sono, siamo comunque riusciti ad ottenere un'ottima compressione.

Infatti, il nostro dataset compresso, compreso di metadati pesa solo 14.8MB, risultando in una compressione ci circa il 71% e senza nessuna perdita di informazioni, che è inoltre più piccola di JPEG-LS di circa il 7%.

La compressione che abbiamo ottenuto è inoltre circa la stessa in termini di percentuale di quella ottenuta dagli autori.

| Compressore | Peso MNIST (MB) |
|---|---|
| Non compresso | 52.3 |
| JPEG-LS | 18.43 |
| Metodo Lossless Dataset Compression (risultato riportato dagli autori) | 15.07 |
| Metodo Lossless Dataset Compression (nostra implementazione) | 14.80 |

# 5 Conclusioni

In conclusione, troviamo che questo metodo di compressione presentato da *Barowsky*, *Mariona* e *Calmon* si renda molto efficace e spesso necessario in contesti di dataset di grandi dimensioni.

La nostra implementazione cerca di essere quanto più vicina alle informazioni fornite in linea teorica dagli autori se pur con qualche piccola modifica resa necessaria.

Purtroppo, date le nostre scarse conoscenze in termini di intelligenza artificiale la rete che abbiamo sviluppato è spesso poco precisa, il che potrebbe essere migliorato in futuro, ottenendo risultati ancora più performanti.

Vediamo in breve aspetti positivi e negativi di questo metodo di compressione lossless.

L'aspetto negativo maggiore riguarda il tempo di esecuzione che richiede circa 20 ore per l'addestramento della rete e circa 1 ora per la compressione, costringendo quindi ad un'attesa pressoché infinita. Si potrebbe pensare per una futura pubblicazione di conservare dei modelli pre-addestrati per almeno i dataset più famosi in circolazione, in questo modo si potrebbe riuscire a saltare la fase più lunga del processo, quella di *training*.

L'aspetto positivo è logicamente la grande percentuale di compressione che si riesce ad ottenere da un dataset senza perdere nessuna informazione. Basti pensare che quando si parla di compressione lossless di immagini è spesso considerata buona già una compressione di 2 ad 1, mentre con questo metodo eseguito sul dataset `MNIST` siamo riusciti ad ottenere una compressione di circa 7 ad 1.

# 6 Riferimenti bibliografici

[1]
https://ieeexplore.ieee.org/document/9413447
https://www.researchgate.net/publication/352172034_Predictive_Coding_for_Lossless_Dataset_Compression

[2]
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.neighbors

[3]
https://docs.scipy.org/doc/scipy/reference/sparse.csgraph.html

[4]
https://networkx.org/documentation/stable/reference/algorithms/traversal.html

[5]
https://keras.io/

[6]
https://github.com/soxofaan/dahuffman