

UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

# Compressione di immagini tramite autoencoder, stato dell'arte e sviluppi futuri

**Relatore**

Prof. Cagnazzo Marco

**Laureando**

Stella Filippo

ANNO ACCADEMICO 2022-2023

Data di laurea 16/11/2023



*Con questo elaborato concludo un percorso di studi che ha rappresentato un'importante tappa della mia vita. È un traguardo che in molti momenti ho pensato di non raggiungere, ma grazie al supporto e l'incoraggiamento di molte persone sono riuscito a raggiungerlo.*

*Innanzitutto, desidero ringraziare il mio relatore, il Professor Marco Cagnazzo, per la sua preziosa guida e il suo costante supporto. Grazie alla sua disponibilità e alla sua competenza, ho potuto approfondire le mie conoscenze e sviluppare un lavoro che sono orgoglioso di presentare.*

*Un ringraziamento particolare va ai miei genitori, che mi hanno sempre sostenuto e incoraggiato durante questi anni di studio.*

*Vorrei ringraziare i miei amici, in particolare Marco e Matteo, che sono stati sempre presenti e che hanno condiviso con me gioie e fatiche, dandomi una mano ad alleviare le tensioni e superare anche i momenti più faticosi.*

*Vorrei ringraziare Chiara, la mia fidanzata, che mi ha sopportato e supportato in questo percorso dandomi una mano nei momenti di bisogno.*

*Vorrei ringraziare tutti i miei colleghi di corso, in particolar modo Marco, Leonardo, Nicolo, Lucrezia e Riccardo. Per avermi spronato ad andare avanti e non rinunciare a questo obiettivo, anche nei momenti in cui ero prossimo a rassegnarmi.*

*Infine, vorrei ringraziare tutti i miei professori, che mi hanno aiutato a crescere e a maturare come persona e come professionista.*

*Grazie a tutti voi, grazie per avermi reso migliore, come persona e come professionista.*



# Sommario

Questo elaborato esplora il campo emergente della compressione delle immagini utilizzando l'intelligenza artificiale. L'obiettivo principale è studiare e analizzare le tecniche esistenti per la compressione delle immagini, nello specifico quelle che fanno uso di autoencoder.

Nel primo capitolo, viene fornita una panoramica dello stato dell'arte della compressione delle immagini utilizzando tecniche tradizionali, fornendo una rapida spiegazione del funzionamento di queste tecniche.

Il secondo capitolo si concentra sull'analisi delle tecniche per la compressione delle immagini tramite intelligenza artificiale. Viene fornita una rapida panoramica dell'attuale stato della ricerca. Successivamente viene fornita una spiegazione del principio di funzionamento degli autoencoder. Infine vengono presentate alcune ricerche che hanno portato un significativo apporto nella ricerca di queste tecniche.

Nel terzo capitolo vengono analizzate le prestazioni di alcuni metodi in termini di qualità oggettiva e soggettiva dell'immagine, compressione e tempo di compressione.

Nel quarto e ultimo capitolo vengono esplorati i possibili sviluppi futuri nel campo della compressione delle immagini tramite l'utilizzo degli autoencoder. Si discute inoltre di come le tecniche attuali potrebbero essere migliorate e si ipotizzano nuove direzioni di ricerca.

In conclusione, questa tesi fornisce una panoramica completa dello stato dell'arte della compressione delle immagini tramite autoencoder e offre spunti per future ricerche in questo campo.



# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Metodi tradizionali</b>	<b>5</b>
1.1 Codifica JPEG . . . . .	5
1.2 Codifica JPEG 2000 . . . . .	5
1.3 Codifica BPG . . . . .	6
1.4 Codifica VVC . . . . .	6
<b>2 Metodi con reti neurali</b>	<b>11</b>
2.1 Funzionamento degli autoencoder . . . . .	12
2.2 Reti di Compressione . . . . .	13
2.2.1 Variational compression with a scale hyperprior, Ballé 2018 . . . . .	14
2.2.2 Discretized gaussian mixture likelihoods, Cheng 2020 . . . . .	16
2.2.3 Neural data-dependent transform, Wang 2022 . . . . .	18
<b>3 Valutazione delle prestazioni</b>	<b>23</b>
3.1 Metriche utilizzate . . . . .	24
3.1.1 BPP . . . . .	24
3.1.2 Tempo di codifica . . . . .	25
3.1.3 PSNR . . . . .	25
3.1.4 MS-SSIM . . . . .	27
3.1.5 LPIPS . . . . .	28
3.2 Presentazione dei risultati . . . . .	29
<b>4 Sviluppi futuri</b>	<b>37</b>
4.1 Possibile utilizzo di SADAM . . . . .	37
4.2 Utilizzi in dispositivi mobili con Slim CAE . . . . .	39
<b>5 Conclusioni</b>	<b>41</b>



# Elenco delle figure

1	Diagramma di compressione Lossy . . . . .	3
1.1	Confronto PNG con JPEG a 0.167 bpp . . . . .	8
1.2	Confronto PNG con JPEG 2000 a 0.171 bpp . . . . .	8
1.3	Confronto PNG con BPG a 0.156 bpp . . . . .	9
1.4	Confronto PNG con VVC a 0.144 bpp . . . . .	9
2.1	Diagramma generico di un autoencoder, immagine presa dal documento [8] . .	13
2.2	Diagramma rete Ballé 2018 et al., immagine presa dal documento [11] . . . .	15
2.3	Diagramma rete Cheng 2020 et al., immagine presa dal documento [12] . . . .	17
2.4	Varianti degli attention modules sviluppati da Cheng 2020 et al., immagine presa dal documento [12] . . . . .	18
2.5	Diagramma rete Wang 2022 et al., immagine presa dal documento [13] . . . .	19
2.6	Moduli aggiuntivi creati dal tem di Wang et al. per il loro modello, immagini prese dal documento [13] . . . . .	20
2.7	Convoluzione di pesi e dati nel processo di decodifica nella rete Wang 2022, immagine presa dal documento [13] . . . . .	20
2.8	Confronto PNG con Ballé2018 a 0.145 bpp . . . . .	21
2.9	Confronto PNG con Cheng2020 a 0.123 bpp . . . . .	22
3.1	Tempi di compressione a 0.07 bpp . . . . .	30
3.2	Tempi di compressione a 0.16 bpp . . . . .	30
3.3	Tempi di compressione a 0.21 bpp . . . . .	31
3.4	Tempi di compressione a 0.34 bpp . . . . .	31
3.5	Tempi di compressione a 0.41 bpp . . . . .	32
3.6	Grafico PSNR . . . . .	33
3.7	Grafico MS-SSIM . . . . .	34
3.8	Grafico LPIPS con AlexNet . . . . .	35
4.1	Diagramma funzionamento slimCAE, immagine presa dal documento [26] . .	40



# Introduzione

Nell'era moderna la crescente quantità di dati di cui usufruiamo ogni giorno sta ricevendo, dagli esperti del settore, molte attenzioni. In quanto il throughput e la quantità di memoria di cui disponiamo sui nostri dispositivi sono, seppur ad oggi ampiamente sufficienti, comunque limitate. Un'altra ragione di questa attenzione è la crescente diffusione di servizi in tempo reale che quindi richiedono di scambiare quantità di dati considerevoli in pochissimo tempo. Un esempio di tali servizi potrebbe essere l'uso della realtà aumentata in ambito medico o di ricerca. Comprimere i dati è quindi ormai una necessità, che si farà sempre più impellente con il crescere delle dimensioni dei dati che andremo a gestire. A partire dai primi anni novanta infatti si sono iniziate a sviluppare alcune tecniche, a cui oggi si fa riferimento come tecniche tradizionali, per comprimere le immagini. Stiamo parlando ed esempio di JPEG e del suo successore JPEG 2000, di più recente sviluppo sono invece i codec BPG e VVC. Più recentemente l'attenzione dei ricercatori si è spostata su metodi basati su deep learning. Questi metodi presentano diversi vantaggi rispetto ai metodi tradizionali, infatti molte volte permettono di ottenere performance migliori rispetto ai metodi classici.

In questo documento ci proponiamo di fornire una panoramica dei metodi di compressione tradizionali più usati, e di quelli basati su deep learning che hanno fornito un maggiore contributo allo sviluppo di questi ultimi. Dopo aver presentato le varie tecniche vogliamo fornirne una valutazione delle prestazioni, in modo da poterli comparare oggettivamente.

La compressione è un processo che mira a minimizzare il numero di bit utilizzati per rappresentare una certa informazione senza intaccarne drasticamente la qualità. Questo obiettivo viene raggiunto riducendo le ridondanze e eliminando i dati irrilevanti. Tutti i framework di compressione consistono di una coppia codificatore-decodificatore, in cui il codificatore ha il compito appunto di codificare l'immagine in una rappresentazione ridotta e il decodificatore trasforma la rappresentazione ridotta nuovamente in un'immagine.

Tutti gli algoritmi di compressione possono essere raggruppati in due macro categorie. Gli algoritmi lossy o con perdita e quelli lossless o senza perdita. Come si può già intuire dal nome gli algoritmi lossless comprimono senza scartare informazioni, si limitano quindi ad applicare delle trasformate per ottenere delle nuove rappresentazioni più efficienti. Gli algoritmi lossy invece

ammettono la possibilità di scartare delle informazioni superflue che non vanno ad intaccare drasticamente la qualità percepita dell’immagine, in modo da poter comprimere ulteriormente. Se andiamo ad osservare gli algoritmi di codifica lossy possiamo scomporli tutti in almeno tre blocchi principali [1], con l’aggiunta di un ulteriore blocco negli algoritmi più recenti, come possiamo vedere nel diagramma 1.

Un primo blocco si occupa di convertire l’immagine in una rappresentazione latente, tramite l’applicazione di una trasformata, in un altro dominio che permette di rappresentare l’informazione da comprimere in modo più sparso.

Successivamente si ha un blocco che si occupa di quantizzazione, ovvero di mappare i valori in ingresso in un insieme finito di dimensione più piccola rispetto a quello di ingresso. In questo passaggio si realizza la perdita di informazioni caratteristica della codifica lossy.

Un ultimo blocco si occupa di effettuare la codifica entropica, in modo da comprimere ulteriormente l’informazione mappando i simboli usati più spesso con pochi bit.

Il quarto blocco, che è stato aggiunto successivamente, si occupa di effettuare predizione spaziale, ovvero di sfruttare le ridondanze e le regioni omogenee delle immagini per comprimere ulteriormente, nonostante questo blocco sia stato introdotto per ultimo va a posizionarsi all’inizio, prima del blocco che applica la trasformata. Data quindi un’immagine da comprimere  $x$ , il codificatore è composto sempre almeno da una trasformata  $\varepsilon$  e una funzione di quantizzazione  $Q$ . Negli algoritmi più recenti possiamo trovare anche la funzione di predizione  $P$  come espresso nella formula 1, dove  $\theta_\varepsilon$  denota i parametri del codificatore

$$y = Q(\varepsilon(P(x; \theta_\varepsilon))) \quad (1)$$

Infine per riottenere la rappresentazione dell’immagine il decodificatore ricostruisce l’immagine  $\hat{x}$  dal codice  $y$  ripercorrendo gli stessi passi in modo inverso, come possiamo vedere nell’equazione 2, dove  $\theta_D$  denota i parametri del decodificatore. [2]

$$\hat{y} = D(y; \theta_D) = D(Q(\varepsilon(P(x; \theta_\varepsilon))); \theta_D) \quad (2)$$

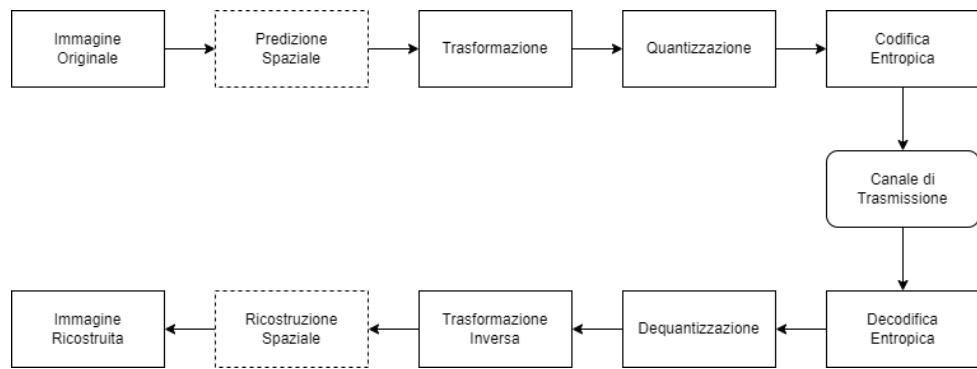


Figura 1: Diagramma di compressione Lossy



# **Capitolo 1**

## **Metodi tradizionali**

La caratteristica degli algoritmi di compressione tradizionali è quella di usare trasformate statistiche all'interno del primo blocco, messe a punto in numerosi anni di ricerca. Questa staticità non permette a questi metodi di adattarsi dinamicamente a tutti i tipi di contenuti delle immagini. Inoltre rende lo sviluppo di un nuovo algoritmo di compressione un processo lungo che richiede anni di studi e progettazione. [3] Andiamo ora a parlare brevemente dei metodi che andremo a considerare quando valuteremo le prestazioni dei vari algoritmi per poterli confrontare.

### **1.1 Codifica JPEG**

Questo metodo di compressione sviluppato nel 1992, sviluppato dall'omonimo gruppo, è diventato in poco tempo il formato di compressione più diffuso. Nonostante i numerosi tentativi di sostituirlo con formati più moderni questo è rimasto ancora ad oggi il formato più usato, principalmente grazie alla sua facilità implementativa. Il JPEG si basa sull'utilizzo della Discrete Cosine Transform o DCT per realizzare la rappresentazione sparsificata dell'immagine originale. L'immagine viene divisa in blocchi di dimensione  $8 \times 8$  e ad ogni blocco viene applicata la trasformata, i valori prodotti dall'applicazione della trasformata vengono poi quantizzati e codificati con due metodi denominati zig-zag scan e run length coding, al termine viene applicata la codifica entropica. [4]

Possiamo vedere un esempio di compressione con JPEG nella figura 1.1

### **1.2 Codifica JPEG 2000**

Nel 2001 con la crescente diffusione di internet, con l'aumento di dimensione e la richiesta di una maggiore qualità delle immagini viene sviluppato, sempre dal Joint Photograph Experts Group (JPEG), questo nuovo formato chiamato appunto JPEG 2000 in virtù dell'anno in cui è

stato sviluppato.

JPEG 2000 non utilizza la DCT come il suo predecessore ma viene introdotta una nuova trasformata, la DWT o Discrete Wavelet Transform, che si propone di meglio identificare e comprimere i bordi delle figure che compongono le immagini, ovvero il dettaglio che ci permette di distinguere le varie regioni all'interno di un'immagine. Un'ulteriore differenza di JPEG 2000 è la sua capacità di comprimere le immagini con qualità progressiva, ovvero permettere di tagliare la stringa di bit in posizioni diverse per ottenere diversi livelli di qualità. Se provassimo a tagliare invece la rappresentazione in bit di JPEG, otterremo un'immagine incompleta.

JPEG 2000 quindi voleva essere un formato di qualità superiore con una compressione più efficiente.[5]

Possiamo vedere un esempio di compressione con JPEG 2000 nella figura 1.2

### 1.3 Codifica BPG

Il formato Better Portable Graphics è stato sviluppato da Fabrice Bellard nel 2014 con l'obiettivo di sostituire l'ormai affermato formato JPEG. Questo metodo si basa sulla codifica intra-frame del codec HEVC o H.265 [6].

H.265 a differenza di JPEG e JPEG 2000 utilizza anche tecniche di predizione spaziale intra-frame per comprimere ulteriormente l'immagine, queste tecniche sfruttano la ridondanza spaziale, come ad esempio le regioni omogenee. Durante la codifica le varie tecniche vengono provate e viene scelta quella migliore. Inoltre la dimensione dei blocchi in cui viene suddivisa l'immagine non è più fissa ma può variare durante la codifica. La trasformata e la codifica entropica rimangono invece concettualmente simili ai metodi precedenti, ma utilizzano degli algoritmi più avanzati.

L'obiettivo di Bellard era quello di realizzare un formato molto leggero che potesse fornire immagini più compresse rispetto a JPEG, ma con una qualità superiore, di cui possiamo vedere un esempio nella figura 1.3

### 1.4 Codifica VVC

Versatile Video Coding (VVC) o H.266 è lo standard di codifica video più recente, finalizzato nel luglio 2020. È stato sviluppato dal Joint Video Experts Team (JVET) dell'ITU-T Video Coding Experts Group (VCEG) e dell'ISO/IEC Moving Picture Experts Group (MPEG) per soddisfare la crescente richiesta di una migliore compressione video, per supportare una più ampia gamma di contenuti multimediali attuali e applicazioni emergenti come contenuti in High Dynamic Range (HDR), a 360°, per la Realtà Virtuale (VR) o la Realtà Aumentata (AR)

[7].

Sebbene sia stato sviluppato per la compressione video, fornisce ottimi risultati anche per la compressione di immagini con metodo intra.

Come il suo predecessore, H.266 utilizza tecniche di predizione spaziale intra-frame e blocchi a dimensione variabile. A differenza del suo predecessore però le tecniche di predizione sono molte di più e più elaborate, e la dimensione di blocco può variare con più libertà. La trasformata e la codifica entropica rimangono simili ai metodi precedenti, ma utilizzano degli algoritmi più avanzati.

L'algoritmo di codifica VVC rappresenta l'attuale stato dell'arte per la compressione di video ed immagini, ne possiamo vedere un esempio nella figura 1.4



(a) Originale



(b) JPEG

Figura 1.1: Confronto PNG con JPEG a 0.167 bpp



(a) Originale



(b) JPEG 2000

Figura 1.2: Confronto PNG con JPEG 2000 a 0.171 bpp



(a) Originale



(b) BPG

Figura 1.3: Confronto PNG con BPG a 0.156 bpp



(a) Originale



(b) VVC

Figura 1.4: Confronto PNG con VVC a 0.144 bpp



# Capitolo 2

## Metodi con reti neurali

Negli ultimi anni abbiamo assistito ad uno sviluppo rapidissimo dell'intelligenza artificiale. Nonostante non sia una novità, i recenti avanzamenti tecnologici e la sempre più crescente disponibilità di dati di addestramento, hanno permesso lo sviluppo rapidissimo di queste tecnologie. Anche l'ambito della computer vision e nello specifico della compressione ha seguito questo andamento, infatti negli ultimi anni sono state pubblicate numerose ricerche che vanno a proporre metodi sempre più efficienti per la compressione di immagini, appunto tramite l'uso di intelligenze artificiali.

Le immagini naturali contengono al loro interno molte ridondanze spaziali che possono essere sfruttate per comprimere senza diminuire la qualità percepita. Le reti neurali sono eccellenti per questo compito, in quanto permettono di ridurre di molto le ridondanze ed identificare le regioni più significative, in modo da operare con maggiore precisione su esse. Partiamo quindi fornendo una rapida introduzione su cosa sia una rete neurale e quali sono i suoi componenti fondamentali.

Il componente fondamentale di una rete neurale è l'unità di elaborazione dati chiamata neurone, e più neuroni sono connessi tra di loro tramite connessioni pesate. Solitamente le reti sono composte da più livelli di neuroni e più livelli ci sono più si dice che la rete è profonda. L'analogia con la biologia umana non è casuale, infatti le reti neurali cercano di riprodurre la struttura interna del cervello umano per poterne emulare i processi cognitivi [1].

Per rendere utilizzabile una rete non è sufficiente crearla definendone la struttura ed altri parametri come le funzioni di attivazione, l'algoritmo di ottimizzazione, il learning rate ed altri iper parametri, Ma va addestrata per il compito specifico per la quale la volgiamo utilizzare. Durante la fase di addestramento vengono aggiustati i pesi relativi alle connessioni tra i vari neuroni, questi pesi vengono modificati fino a quando viene trovata la migliore configurazione che permette di raggiungere il miglior risultato possibile nel compito che ci interessa.

Le reti neurali sono uno dei metodi più comunemente usati per compiti di regressione e classifi-

cazione [1], il che le rende sfruttabili per cercare delle funzioni di regressione che permettano di comprimere le informazioni in delle loro rappresentazioni latenti più compresse. Durante gli anni si sono sviluppate molte tipologie di reti neurali che si differenziano per configurazione o per metodo di addestramento, le strutture che vengono maggiormente utilizzate sono CNN, GAN e Autoencoder. Dei primi due metodi forniremo una breve descrizione per illustrarne le differenze principali, mentre sugli Autoencoder ci concentreremo maggiormente in quanto oggetto di questo studio. Le reti neurali convoluzionali o CNN sono reti molto indicate per lavorare con immagini o video in quanto estraggono le caratteristiche dai dati di input tramite una serie di livelli convoluzionali e di pooling. Queste caratteristiche vengono poi utilizzate per operazioni di classificazione o identificazione.

Le reti avversarie generative o GAN vengono solitamente utilizzate per operazioni di generazione. Queste reti sono costituite da due parti, un generatore e un discriminatore. Il generatore cerca di generare, grazie a ciò che ha appreso dai dati di esempio, nuovi esempi il più realistici possibili per cercare di ingannare il discriminatore. Il discriminatore invece cerca di capire quali esempi sono reali e quali generati.

## 2.1 Funzionamento degli autoencoder

Gli autoencoder o autocodificatori sono un modello, le cui prime applicazioni risalgono al 1980, che consiste di due parti. Un codificatore si occupa di effettuare l'embedding dell'input, ovvero convertire l'input in una sua rappresentazione latente di dimensione inferiore. Da questa rappresentazione la seconda parte, il decodificatore, cerca di ricostruire l'input nel modo più fedele possibile. Lo schema classico di un autoencoder è visibile nell'immagine 2.1

Queste reti sono allenate utilizzando delle funzioni di perdita che vanno a misurare la differenza tra i dati in ingresso al codificatore e il prodotto del decodificatore. In base al risultato della funzione di perdita aggiustano i pesi delle reti fino a raggiungere il miglior risultato possibile, ovvero fino a realizzare una ricostruzione il più simile possibile ai dati di input. Gli autoencoder non sono però sufficienti per costruire un algoritmo di compressione che, come illustrato precedentemente, consiste di almeno tre elementi e più recentemente quattro.

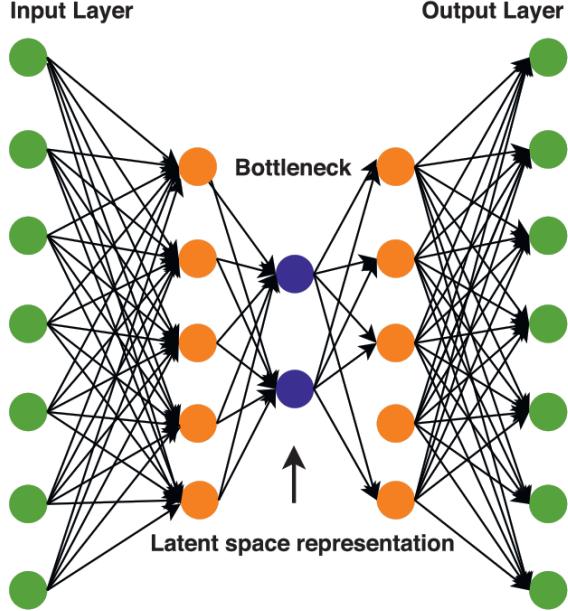


Figura 2.1: Diagramma generico di un autoencoder, immagine presa dal documento [8]

## 2.2 Reti di Compressione

Per ovviare a questo problema sono stati quindi sviluppati i Compressive AutoEncoder o CAE, che non sono nient’altro che un’estensione degli autoencoder in cui oltre alla rete neurale viene aggiunto un modello di probabilità che si occupa di assegnare un numero di bit alle rappresentazioni in base alla loro frequenza. Questa aggiunta permette di effettuare la codifica entropica della rappresentazione latente. [9]

La funzione di perdita di questa nuova architettura 2.1 varia leggermente rispetto quella dei soli autoencoder, in quanto viene aggiunto un termine che tiene conto del numero di bit utilizzati.

$$L = \underbrace{-\log_2(Q([f(x)]))}_{\text{numero di bit}} + \beta \cdot \underbrace{d(x, g([f(x)]))}_{\text{distorsione}} \quad (2.1)$$

Come possiamo vedere, l’equazione 2.1 consiste di due termini principali, il termine di destra rappresenta la distorsione tra ingresso e uscita della rete. Il secondo termine a sinistra viene introdotto appositamente per le CAE e stima il numero di bit utilizzati, questo permette di ottimizzare la rete non solo in base alla distorsione ma anche in base al numero di bit. Per controllare il compromesso tra numero di bit e distorsione viene introdotto il parametro  $\beta$ .

Il problema con questa nuova funzione nasce dal fatto che durante la fase di backpropagation,

con tecniche come Stochastic Gradient Descent (SGD) dove è necessario calcolare il gradiente. La funzione 2.1 sopra illustrata non può essere utilizzata in quanto composta da funzioni non differenziabili, ovvero la funzione  $Q$  e l’arrotondamento  $[ \cdot ]$ . Per ovviare a questo inconveniente è necessario trovare delle alternative differenziabili.

Theis et al. [9] propone due soluzioni possibili, basate sul lavoro di Ballé et al. (2016) [10]. La prima soluzione si propone di risolvere il problema della funzione di arrotondamento, mentre Ballé et al. (2016) proponeva invece di utilizzare del rumore additivo gaussiano. 2.2.

$$[f(x)] \approx f(x) + u \quad (2.2)$$

Theis propone invece di rimpiazzare la derivata durante la fase di backpropagation con una sua approssimazione lineare. Empiricamente hanno verificato che l’arrotondamento lineare  $r(y) = y$  funziona alla pari di molte altre funzioni più elaborate.

La seconda soluzione risolve invece il problema della non differenziabilità della funzione di quantizzazione  $Q$ , utilizzando un’approssimazione continua 2.3.

$$Q(z) = \int_{[-.5,.5]^M} q(z + u) du \quad (2.3)$$

In questa approssimazione  $q$  rappresenta la densità di probabilità della distribuzione  $Q$ .

Lo sviluppo dei compressive autoencoder o CAE e le soluzioni proposte per rendere possibile l’esecuzione di algoritmi di addestramento, che richiedono il calcolo dei gradienti delle funzioni di perdita. Hanno permesso di sviluppare, negli anni successivi, modelli e nuove soluzioni in grado di ottenere buone prestazioni rispetto agli algoritmi tradizionali. Una lista che comprende tutti i metodi sviluppati fino al 2022 è presente nel lavoro di Mishra et al. [8], noi ci concentreremo sui tre che hanno fornito il maggiore contributo alla ricerca.

### 2.2.1 Variational compression with a scale hyperprior, Ballé 2018

Il primo metodo che andiamo ad analizzare è quello proposto nel 2018 da Ballé et al. [11] che prosegue il lavoro di Ballé et al. del 2016 [10], lavoro che ha ispirato anche la soluzione 2.3 offerta da Theis un anno prima.

Il metodo proposto da Ballé et al. nel 2018 utilizza un modello di probabilità misto a scala gaussiana o GSM dove i parametri di scala sono condizionati da un iperparametro. Questo modello permette l’addestramento end-to-end, ciò include anche l’ottimizzazione della rappresentazione quantizzata dell’iper parametro, del modello dell’entropia e dell’autoencoder principale.

L'innovazione principale di questo modello è l'aggiunta dell'iper parametro nell'informazione trasmessa, ciò permette al decodificatore di usare un modello dell'entropia condizionato dall'iper parametro. Questo consente di avere un modello per l'entropia dipendente dall'immagine. L'immagine 2.2 mostra l'architettura ad alto livello del modello di compressione, come si può osservare il modello sia composto da due sottoreti.

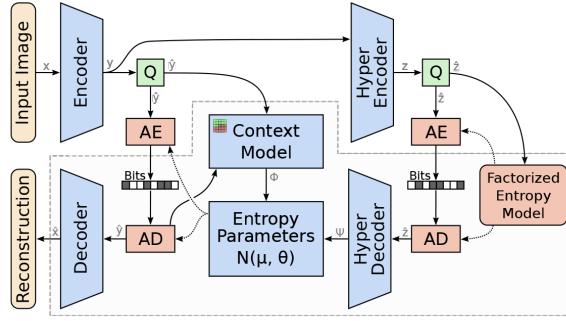


Figura 2.2: Diagramma rete Ballé 2018 et al., immagine presa dal documento [11]

La prima sottorete è l'autoencoder principale che ricava una rappresentazione latente delle immagini. La seconda invece ricava un modello di probabilità per effettuare la codifica entropica della rappresentazione latente.

I dati provenienti dalle due sottoreti vengono combinati dalla rete per i parametri dell'entropia, generando la media e la scala per il modello gaussiano dell'entropia.

La funzione di perdita utilizzata da questo modello 2.4 è molto simile a quella riportata in precedenza per i CAE, dove il parametro  $\beta$  viene sostituito dal moltiplicatore di Lagrange  $\lambda$  che regola il compromesso tasso-distorsione. Come di consueto l'obbiettivo di addestramento è quello di minimizzare la funzione di perdita rispetto ai parametri del modello.

$$R + \lambda \cdot D = \underbrace{\mathbb{E}_{x \sim p_x}[-\log_2 p_{\hat{y}}(\lfloor f(x) \rfloor)]}_{\text{numero di bit}} + \lambda \cdot \underbrace{\mathbb{E}_{x \sim p_x}[d(x, g(\lfloor f(x) \rfloor))]}_{\text{distorzione}} \quad (2.4)$$

L'entropia di ogni rappresentazione latente  $\hat{y}_i$  viene modellata come una gaussiana convoluta con una distribuzione uniforme, dunque il modello dell'entropia ha la formulazione 2.5.

$$p_{\hat{y}}(\hat{y}|\hat{z}) = \prod_i (\mathcal{N}(\mu_i, \sigma_i^2) * \mathcal{U}(-\frac{1}{2}, \frac{1}{2}))(\hat{y}_i) \quad (2.5)$$

Per predire media e scala della gaussiana invece vengono utilizzati sia l'iper distribuzione  $\hat{z}$ , sia il contesto della rappresentazione latente  $\hat{y}$ . In questo modello ci aspettiamo che l'iper codificatore e l'iper decodificatore apprendano due funzioni leggermente differenti in quanto

lavorano in combinazione con una rete autoregressiva che determina i parametri del modello dell’entropia. Ed essendo sia la rappresentazione latente che la iper-latente parte dell’informazione compressa generata dalla rete, l’equazione 2.4 deve essere leggermente espansa nell’equazione 2.6, per includere il costo di codifica degli iper-latenti e utilizzare come distorsione la distanza quadratica media.

$$R + \lambda \cdot D = \underbrace{\mathbb{E}_{x \sim p_x}[-\log_2 p_{\hat{y}}(\hat{y})]}_{\text{bit(latenti)}} + \underbrace{\mathbb{E}_{x \sim p_x}[-\log_2 p_{\hat{z}}(\hat{z})]}_{\text{bit(iper latenti)}} + \lambda \cdot \underbrace{\mathbb{E}_{x \sim p_x} \|x - \hat{x}\|_2^2}_{\text{distorzione}} \quad (2.6)$$

Il team di Ballé fornisce anche i dettagli implementativi per i livelli della rete, che abbiamo riportato nella tabella 2.1.

Codificatore	Decodificatoree	Iper Codificatore	Iper Decodificatore	Predittore di Contesto	Parametri Entropia
Conv: 5x5 c192 s2 GDN	Deconv: 5x5 c192 s2 IGDN	Conv: 3x3 c192 s1 Leaky ReLU	Deconv: 5x5 c192 s2 Leaky ReLU	Masked: 5x5 c384 s1	Conv: 1x1 c640 s1 Leaky ReLU
Conv: 5x5 c192 s2 GDN	Deconv: 5x5 c192 s2 IGDN	Conv: 5x5 c192 s2 Leaky ReLU	Deconv: 5x5 c288 s2 Leaky ReLU		Conv: 1x1 c512 s1 Leaky ReLU
Conv: 5x5 c192 s2 GDN	Deconv: 5x5 c192 s2 IGDN	Conv: 5x5 c192 s2 Leaky ReLU	Deconv: 3x3 c384 s1		Conv: 1x1 c384 s1
Conv: 5x5 c192 s2	Deconv: 5x5 c3 s2				

Tabella 2.1: Ad ogni riga della tabella corrisponde un livello del modello generalizzato, i dati della tabella sono stati ricavati dal documento [11]

Possiamo vedere degli esempi di compressione con questa rete nelle immagini 2.8b e 2.8c, le due immagini sono state prodotte da due versioni diverse della rete, rispettivamente con media della gaussiana variabile e con media della gaussiana fissata a zero.

## 2.2.2 Discretized gaussian mixture likelihoods, Cheng 2020

Il secondo metodo che approfondiremo è quello proposto da Cheng et al. nel 2020 [12], questo metodo prende come punto di partenza il lavoro di Ballé et al. [11] e lo migliora apportando delle modifiche al modello di probabilità delle rappresentazioni latenti.

Il lavoro di Ballé et al. [11] utilizza una distribuzione Gaussiana con media e scala congiunta con un modello autoregressivo 2.5. I test svolti dal team di Cheng hanno evidenziato della ridondanza spaziale residua, ridondanza che se possibile eliminare porterebbe a rappresentazioni leggermente ridotte e quindi più efficienti.

La loro proposta è quindi quella di utilizzare una miscela di gaussiane con la formulazione 2.7

$$p_{\hat{y}}(\hat{y}|\hat{z}) = \left( \sum_{k=1}^K w_i^{(k)} \mathcal{N}(\mu_i^{(k)}, \sigma_i^{2(k)}) * \mathcal{U}(-\frac{1}{2}, \frac{1}{2}) \right) (\hat{y}_i) \quad (2.7)$$

Quindi ogni miscela è caratterizzata da tre parametri, un peso  $w_i^{(k)}$ , la media  $\mu_i^{(k)}$  e la scala  $\sigma_i^{2(k)}$ . Con questo metodo sono comunque presenti delle ridondanze, ma l'aggiunta dei pesi  $w_i$  permette al modello di adattarsi alle diverse regioni delle immagini. Inoltre le scale prodotte con questo metodo sono più piccole rispetto al metodo di Ballé et al. [11], il che rende il modello più accurato, risultando in una rappresentazione compressa dell'immagine che richiede meno bit.

L'architettura della rete proposta da Cheng et al. [12] è rappresentata nell'immagine 2.3, come possiamo osservare anche qui sono presenti due sottoreti. Una responsabile di ricavare la rappresentazione latente dell'immagine, l'altra di ricavare gli iperparametri.

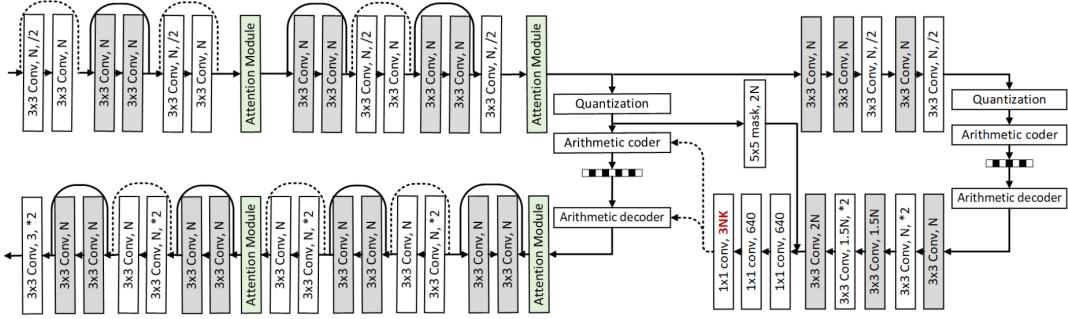


Figura 2.3: Diagramma rete Cheng 2020 et al., immagine presa dal documento [12]

Questo modello introduce due nuovi moduli, gli Attention Module, questi servono per aiutare la rete a prestare maggiore attenzione alle parti più complesse delle immagini e ridurre i bit necessari per rappresentare invece le parti più semplici. Uno schema che illustra il funzionamento di un attention module è visibile nell'immagine 2.4a.

Questi moduli però richiedono una grande quantità di tempo in fase di addestramento, per cercare di ovviare a questo problema viene proposta una versione semplificata 2.4b in cui è stato rimosso il blocco contenente le informazioni non locali.

Possiamo vedere degli esempi di compressione con questa rete nelle immagini 2.9b e 2.9c, le due immagini sono state prodotte da due versioni diverse della rete, rispettivamente senza Attention module e con.

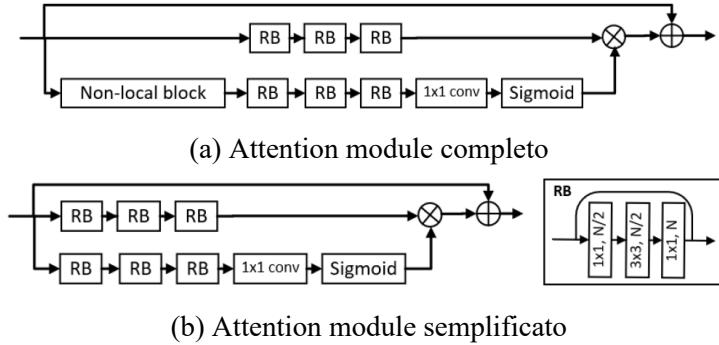


Figura 2.4: Varianti degli attention modules sviluppati da Cheng 2020 et al., immagine presa dal documento [12]

### 2.2.3 Neural data-dependent transform, Wang 2022

Il terzo ed ultimo metodo che andremo ad approfondire è quello proposto nel 2022 da Wang et al. [13], questo metodo propone un nuovo approccio per rendere la compressione di un’immagine dipendente dall’immagine stessa. Per fare ciò hanno preso ispirazione dai precedenti lavori e dai modelli di codifica ibridi, il framework proposto cerca la migliore trasformata per comprimere l’immagine specifica, la proposta di Wang et al. si differenzia dai metodi sviluppati in precedenza nei seguenti aspetti.

Durante la fase di codifica vengono prodotti sia la rappresentazione compressa sia una rappresentazione neuro-sintattica, in grado di catturare informazioni astratte sul contesto dell’immagine che possano aiutare a proiettare la rappresentazione compressa in un sottospazio in cui i coefficienti sono più compatti.

Il modello per la codifica entropica si differenzia da quelli esistenti in quanto le due rappresentazioni latenti  $\hat{z}_s$  e  $\hat{z}_c$  vengono codificate separatamente 2.8 in due stringhe di bit compresse  $b_s$  e  $b_c$ .

$$b_c = EC(\hat{z}_c), \quad b_s = EC(\hat{z}_s) \quad (2.8)$$

La compressione separata permette un controllo più preciso del processo di codifica. Simmetricamente, avviene il processo di decodifica 2.9.

$$\hat{z}_c = ED(b_c), \quad \hat{z}_s = ED(b_s) \quad (2.9)$$

La funzione di decodifica proposta è dipendente dai dati, in quanto per immagini di input diverse  $x$  si ottengono diverse rappresentazioni neuro-sintattiche  $\hat{z}_s$ , in modo da generare trasformate di decodifica più specifiche.

La funzione di perdita che deve quindi essere ottimizzata in fase di addestramento della rete ha la struttura 2.10. Dove  $D$  rappresenta la metrica di distorsione,  $R$  misura il bit-rate,  $\hat{z}_h$  rappresenta l'iper distribuzione di  $\hat{z}_s$  e  $\hat{z}_c$ , mentre  $\lambda$  è l'iper parametro che regola il compromesso tra tasso di codifica e distorsione.

$$L = D(x, \hat{x}) + \lambda(R(\hat{z}_c) + R(\hat{z}_s) + R(\hat{z}_h)) \quad (2.10)$$

La struttura del modello proposto da Wang et al. è visibile nell'immagine 2.5, come è possibile osservare ritroviamo la solita struttura composta da due sottoreti, una per ricavare la rappresentazione latente dell'immagine, l'altra per ricavare gli iperparametri. La novità introdotta in questo modello è la presenza di una trasformata dipendente dai dati in ingresso, questa capacità della rete corrisponde al percorso evidenziato in rosso all'interno della figura 2.5.

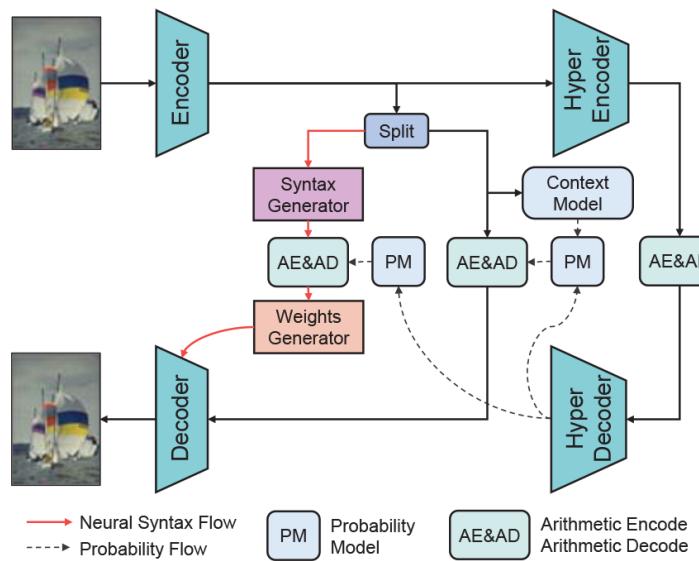


Figura 2.5: Diagramma rete Wang 2022 et al., immagine presa dal documento [13]

In questo modello vengono introdotti due nuovi moduli, un Syntax Generator e un Weights Generator di cui possiamo osservare una rappresentazione nelle immagini 2.6a e 2.6b.

Il modulo di Syntax Generation 2.6a è stato creato per estrarre delle informazioni di sintassi dall'input in fase di codifica, dopo averle estratte queste informazioni vengono mappate in un vettore unidimensionale  $\hat{z}_s$  con un operazione di pooling globale. I vettori  $\hat{z}_s$  vengono quindi

concatenati, quantizzati e codificati.

Il secondo modulo 2.6b si occupa invece di mappare la rappresentazione neuro-sintattica  $\hat{z}_s$ ,

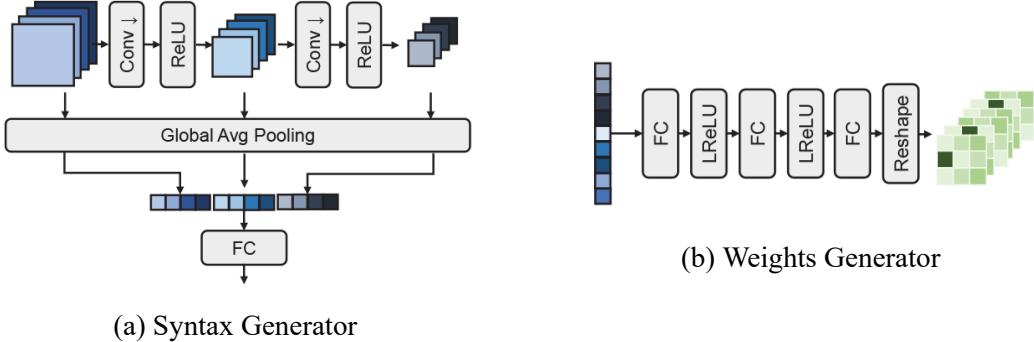


Figura 2.6: Moduli aggiuntivi creati dal team di Wang et al. per il loro modello, immagini prese dal documento [13]

ottenuta durante la codifica, in parametri per i kernel della rete di decodifica, l'estrazione di questi parametri è quello che permette a questa rete di creare delle trasformate dipendenti dai dati.

Questi parametri vengono utilizzati dalla rete di decodifica, come possiamo vedere nella figura 2.7, in quanto solo i parametri dell'ultimo livello della rete sono fissati, mentre quelli dei livelli precedenti vengono generati dal Weights Generator, in modo da permettere alla rete di adattarsi ad immagini diverse.

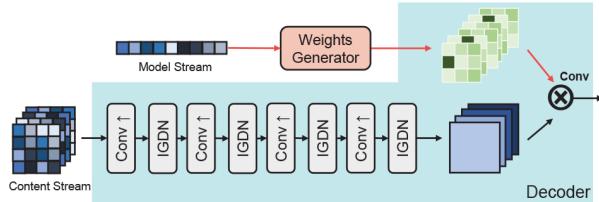


Figura 2.7: Convoluzione di pesi e dati nel processo di decodifica nella rete Wang 2022, immagine presa dal documento [13]

Il team di Wang et al. ha inoltre sviluppato un modulo aggiuntivo di post-processing, utile per i casi in cui è richiesta una ricostruzione più dettagliata, anche questa rete fa uso della rappresentazione neuro-sintattica per adattarsi maggiormente alle varie immagini. Grazie a questa aggiunta il team di Wang et al. ha riportato di aver superato anche il codec VVC, che attualmente rappresenta lo stato dell'arte per la compressione di immagini.

Di questo modello non possiamo fornire degli esempi di compressione in quanto il team di Wang et al. [13] ha rilasciato il modello ma non la rete pre allenata e non avendo a disposizione

dell'equipaggiamento adeguato per eseguire l'allenamento del modello, ci è stato impossibile utilizzare questa rete nei nostri esperimenti.



Figura 2.8: Confronto PNG con Ballé2018 a 0.145 bpp



(a) Originale



(b) Cheng2020



(c) Cheng2020 attention

Figura 2.9: Confronto PNG con Cheng2020 a 0.123 bpp

# Capitolo 3

## Valutazione delle prestazioni

In questo capitolo vogliamo comparare gli algoritmi tradizionali descritti nel capitolo 1 con due degli algoritmi descritti nel capitolo 2, nello specifico Ballé et al. [11] e Cheng et al. [12]. Purtroppo non siamo stati in grado di testare anche Wang et al. [13] in quanto, nonostante abbiano reso il modello disponibile, non ne hanno fornito una versione pre addestrata.

L'hardware su cui sono stati eseguiti i test è un computer dotato di processore AMD Ryzen 5 5600X con 6 core e 12 thread a 3.7GHz, munito di 16GB di ram DDR4 a 3200MHz e come sistema operativo una distribuzione Fedora Linux 38 con kernel linux 6.5.7-200. Vogliamo sottolineare anche che per realizzare delle misurazioni più vicine alla realtà possibili abbiamo deciso di non utilizzare GPU, in quanto non tutti i computer sono dotati di GPU su cui è possibile eseguire Tensorflow o PyTorch e non abbiamo preso particolari precauzioni per garantire l'esecuzione esclusiva del codice, abbiamo lasciato che il codice concorresse con il sistema operativo e le applicazioni in background per l'allocazione del processore, tutto per simulare un ambiente più simile ad un caso reale.

Per effettuare i vari test sono stati realizzati dei notebook in python 3.11 su jupyterlab 4.0.7. I codificatori che abbiamo utilizzato sono i seguenti. Per comprimere le immagini con JPEG abbiamo utilizzato Pillow 10.0.1 [14] sviluppato da Jeffrey A. Clark. Per comprimere le immagini con JPEG 2000 abbiamo utilizzato OpenJPEG 2.5.0 [15] sviluppato dall' Université de Louvain. Per comprimere con BPG abbiamo usato il framework messo a disposizione da F.Bellard [6] nella versione 0.9.8. Per comprimere con VVC abbiamo usato il framework distribuito da Fraunhofer HHI [16] che comprende l'encoder vvencapp 1.9.1 e il decocder vvdecapp 2.1.2. Per comprimere con le due reti abbiamo utilizzato la libreria compressai 1.2.4 [17] in cui sono presenti le implementazioni pre allenate dei due modelli che ci interessano. Dei due modelli sono presenti due versioni, per Ballé et al. [11] è fornita un implementazione in cui la media della distribuzione gaussiana viene bloccata a 0 e una in cui la media è un parametro determinato durante la compressione, per Cheng at al. 2020 [12] viene fornita una versione che

fa uso degli attention module e una in cui sono disabilitati.

Per valutare i vari metodi abbiamo utilizzato 24 immagini non compresse di dimensione  $2048 \times 3072$  con spazio di colore RGB a 8 bit per canale, prese dal database di Kodak [18], abbiamo deciso di utilizzare questo dataset in quanto è il dataset maggiormente utilizzato dalla comunità scientifica per valutare le prestazioni di algoritmi che lavorano su immagini.

I vari algoritmi sono stati eseguiti più volte con vari livelli di qualità, che andremo ad approfondire successivamente, ed infine sono state calcolate le metriche sui risultati ottenuti.

## 3.1 Metriche utilizzate

Per valutare le prestazioni facciamo affidamento ad alcune metriche che ci permettono di confrontare le varie tecniche. Molte di queste metriche sono oggettive, alcune invece cercano di quantificare il più fedelmente possibile quella che è la qualità percepita da un osservatore umano. Passiamo ora ad introdurre brevemente le metriche e come vengono calcolate.

### 3.1.1 BPP

Per valutare quanto un’immagine sia stata compressa utilizziamo il numero di bit necessari per rappresentare un pixel dell’immagine  $x$ , o bit per pixel. Il calcolo 3.1 di questo parametro è molto semplice ed intuitivo.

$$bpp(x) = \frac{taglia(x)}{larghezza(x) \cdot altezza(x)} \quad (3.1)$$

Dove con taglia indichiamo il numero di bit occupati dall’immagine compressa, con larghezza intendiamo il numero di pixel in una riga orizzontale dell’immagine e con altezza intendiamo il numero di pixel in una riga verticale dell’immagine.

Lo spezzone di codice 3.1 mostra come è stato calcolata la metrifica bit per pixel.

Listing 3.1: Spezzone di codice per il calcolo dei bit per pixel

```
from PIL import Image
import os

image = Image.open(file)
file_size = os.path.getsize(file) * 8
pixels = image.width * image.height
bits_per_pixel = file_size / pixels
```

### 3.1.2 Tempo di codifica

Per valutare la velocità con la quale un algoritmo di codifica riesce a produrre la rappresentazione compressa di un’immagine andiamo a misurare il tempo che intercorre tra la chiamata al codificatore e il termine dell’esecuzione del processo di codifica. Avendo utilizzato python per richiamare i codificatori abbiamo usato il modulo timeit di per calcolare i tempi di esecuzione. Lo spezzone di codice 3.2 mostra come è stata usata la libreria per calcolare i tempi di esecuzione.

Listing 3.2: Spezzone di codice per il calcolo del tempo di compressione

```
import timeit

starttime = timeit.default_timer()
call_to_encoder() #Chiamata encoder
endtime = timeit.default_timer()
execution_time = endtime - starttime #Calcolo del tempo in secondi
```

### 3.1.3 PSNR

L’ultima delle metriche oggettive che andiamo a considerare è il Peak Signal to Noise Ratio o PSNR. Questa metrica, espressa in scala logaritmica, rappresenta il rapporto tra la potenza del segnale originale e la potenza del rumore introdotto dal processo di compressione, dunque più alto è il valore del PSNR più l’immagine compressa è fedele all’originale.

Per poter calcolare il PSNR occorre definire cosa sia e come si calcola l’MSE per immagini a colori, ovvero con più componenti.

L’ Errore Quadratico Medio o MSE indica la distanza al quadrato tra il valore di un pixel dell’immagine e il valore dello stesso pixel nell’immagine distorta. Per calcolare l’MSE di una singola componente di colore si utilizza la formula 3.2, dove  $M$  ed  $N$  indicano rispettivamente la larghezza e l’altezza in pixel delle immagini orginale  $R$  e compressa  $C$ .

$$MSE(R, C) = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \|R(i, j) - C(i, j)\|^2 \quad (3.2)$$

Nel caso di immagini a colori però non abbiamo una sola componente ma ne abbiamo al minimo due, bisogna dunque definire come combinare gli MSE delle singole componenti, in base allo spazio di colore scelto, in un'unica metrica globale per l'immagine.

Nel nostro caso, avendo scelto lo spazio di colore YUV, la combinazione degli MSE delle singole componenti si ottiene con la formula 3.3, nella quale possiamo osservare che l'MSE globale dell'immagine non è altro che la somma pesata degli MSE delle singole componenti.

Questi pesi sono utilizzati per attribuire più importanza al canale  $Y$ , in quanto le informazioni in questo canale sono quelle maggiormente responsabili della qualità dell'immagine.

$$MSE(R, C) = \left(\frac{3}{4}\right) \cdot MSE_Y(R, C) + \left(\frac{1}{8}\right) \cdot MSE_U(R, C) + \left(\frac{1}{8}\right) \cdot MSE_V(R, C) \quad (3.3)$$

Avendo definito come si calcola l'MSE pesato, il calcolo del PSNR pesato si ottiene con la formula 3.4, in cui  $I$  indica il massimo valore che il pixel di un canale può assumere, nel nostro caso essendo i canali nello spazio di colore YUV rappresentati con 8 bit ciascuno, il valore di  $I$  è 255.

$$PSNR(R, C) = 10 * \log \frac{I^2}{MSE(R, C)}_{10} \quad (3.4)$$

Questa metrica è uno standard nella comunità scientifica per l'analisi della qualità dei sistemi che operano con immagini, solitamente viene calcolata per immagini con gli spazi di colore YUV o YCbCr.

Per il calcolo abbiamo usato la libreria scikit-image 0.22.0 per calcolare l'MSE dei singoli canali per poi combinarli e calcolare il PSNR, come possiamo vedere nello spezzone di codice 3.3.

Listing 3.3: Spezzone di codice per il calcolo del PSNR pesato

```
from skimage import metrics

YMSE = metrics.mean_squared_error(OY, Y)
UMSE = metrics.mean_squared_error(OU, U)
VMSE = metrics.mean_squared_error(OV, V)
MSE = (3/4)*YMSE + (1/8)*UMSE + (1/8)*VMSE
psnr = 10 * numpy.log10((255*255) / MSE)
print('PSNR: '+str(msssim)+'dB')
```

### 3.1.4 MS-SSIM

La prima metrica che cerca di fornire un indice di qualità percepita che andremo a considerare è il MultiScale Structural Similarity for IMage quality assessment o MS-SSIM. [19].

Lo sviluppo di questo indice si basa sull'assunto che il sistema visivo umano è altamente adatto per estrarre informazioni strutturali dalle immagini, dunque una misura di similarità strutturale dovrebbe fornire una buona approssimazione della qualità percepita da un osservatore umano. Per il calcolo dell'MS-SSIM, come possiamo vedere nell'equazione 3.5, si deve eseguire il prodotto pesato di tre indici, un indice di luminanza  $l$ , un indice di contrasto  $c$  e un indice di struttura  $s$ , per la stessa immagine scalata  $M$  volte. Questi indici sono pesati da tre esponenti, rispettivamente  $\alpha_M \beta_j \gamma_j$ , e come possiamo osservare solamente due dei componenti vengono pesati per ogni scalatura dell'immagine, la luminanza invece viene calcolata solo per l'indice di qualità  $M$ .

$$MS - SSIM(R, C) = [l_M(R, C)]^{\alpha_M} \cdot \prod_{j=1}^M [c_j(R, C)]^{\beta_j} [s_j(R, C)]^{\gamma_j} \quad (3.5)$$

L'indice  $M$  rappresenta la scalatura delle immagini in ingresso, se l'indice è uguale a 1 ci stiamo riferendo esattamente alle immagini da valutare. Ad ogni incremento di  $M$  all'immagine originale viene applicato un filtro passa basso e viene sotto campionata di un fattore 2. Dopo esattamente  $M - 1$  iterazioni l'immagine non sarà più riducibile e la computazione termina.

Questa è la seconda metrica standard che viene utilizzata dalla comunità scientifica per valutare le prestazioni di algoritmi che operano su immagini, in quanto a differenza del PSNR, questa rappresenta un giudizio più soggettivo.

Per il calcolo dell'MS-SSIM abbiamo usato la libreria pytorch-msssim 1.0.0 che fornisce una semplice funzione per il calcolo di questa metrica, l'unica accortezza da dover prendere è quella di convertire le immagini da valutare in tensori normalizzati prima di passare le immagini alla funzione, come possiamo vedere nello spezzone di codice 3.4.

Listing 3.4: Spezzone di codice per il calcolo dell'MS-SSIM

```
import torch
from skimage import metrics
from torchvision import transforms

device = 'cuda' if torch.cuda.is_available() else 'cpu'

reference = transforms.ToTensor()(reference_image).unsqueeze(0).to(device)
compressed = transforms.ToTensor()(compressed_image).unsqueeze(0).to(device)
msssim = ms_ssim(reference, compressed, data_range=1, size_average=True)
print('MS-SSIM: '+str(msssim))
```

### 3.1.5 LPIPS

L'ultima metrica che andremo ad utilizzare è anch'essa soggettiva e utilizza delle reti neurali per valutare la distanza percepita tra l'immagine originale e l'immagine compressa, la tecnica proposta da Zhang et al. nel 2018 [20] prende il nome di Learned Perceptual Image Patch Similarity o LPIPS.

L'osservazione su cui si basa lo sviluppo di questa metrica è il fatto che l'attivazione interna dei neuroni di reti utilizzate per compiti di classificazione di immagini ad alto livello possono essere utilizzate per calcolare una distanza percepita tra due immagini. Durante lo sviluppo di questa metrica il team di Zhang et al. ha scoperto non solo che questa si rivela essere un'ottima metrica, ma riesce anche ad emulare molto bene i giudizi dati da degli osservatori umani, la maggior parte delle volte anche in modo migliore rispetto ad altre metriche più affermate, come il precedentemente citato MS-SSIM.

Le reti che sono state valutate dal team di Zhang et al. sono SqueezeNet, AlexNet e VGG, a queste reti vengono fornite le due immagini  $x$  e  $x_0$ , rispettivamente originale e distorta. Dalle reti vengono poi estratte le feature da  $L$  livelli, rispettivamente  $\hat{y}^l$  e  $\hat{y}_0^l$  e vengono normalizzate rispetto alla dimensione dei canali. Le attivazioni dei vari canali vengono poi pesate con un vettore di pesi  $w^l$  e ne viene calcolata la distanza  $L_2$ , come possiamo vedere nell'equazione 3.6.

$$d(x, x_0) = \sum_l \frac{1}{H_l W_l} \sum_{h,w} \|w_l \cdot (\hat{y}_{hw}^l - \hat{y}_{0,hw}^l)\|_2^2 \quad (3.6)$$

Noi abbiamo scelto di utilizzare la metrica LPIPS con rete AlexNet, in quanto dai risultati sperimentali, nonostante non sia la tecnica più avanzata, fornisce la valutazione più simile a quella data da degli osservatori umani.

Il team di Zhang et al. fornisce anche una libreria per python che abbiamo utilizzato per i nostri test nella versione 0.1.4, nello spezzone di codice 3.5 mostriamo l'uso di tale libreria.

Listing 3.5: Spezzone di codice per il calcolo di LPIPS con AlexNet

```
import torch
from torchvision import transforms
import lpips

loss_fn_alex = lpips.LPIPS(net='alex')

reference = transforms.ToTensor()(reference_image).unsqueeze(0).to(device)
compressed = transforms.ToTensor()(compressed_image).unsqueeze(0).to(device)
d = loss_fn_alex(transforms.Normalize(mean=(0.5, 0.5, 0.5),
                                      std=(0.5, 0.5, 0.5))(reference), transforms.Normalize(mean=(0.5, 0.5, 0.5),
                                      std=(0.5, 0.5, 0.5))(compressed))[0].item()

print('LPIPS: '+str(d))
```

## 3.2 Presentazione dei risultati

Andiamo ora a presentare i risultati sperimentali ottenuti dalla compressione delle immagini del dataset Kodak con gli algoritmi precedentemente descritti. Per ogni metodo di compressione vengono analizzati cinque diversi livelli di qualità comparati con le metriche appena descritte. I livelli vengono scelti andando ad agire sui parametri di qualità degli encoder, per cercare di mantenere il confronto equo abbiamo cercato di ottenere circa gli stessi livelli per ogni metodo di compressione. Tutte i valori che andremo a presentare sono le medie delle metriche sulle 24 immagini del dataset.

Tutto il codice scritto per effettuare i test e tutte le immagini comprimate con le relative metriche sono disponibili presso il repository su Github [21].

Il primo livello significativo si trova in corrispondenza di  $0.16\text{bpp}$ , i tempi di compressione per questo livello di qualità sono visibili nell'immagine 3.2. Il secondo livello si trova in corrispondenza di  $0.21\text{bpp}$ , dei cui tempi di compressione sono riportati nell'immagine 3.3. Il terzo livello che andiamo a considerare si ha per  $0.34\text{bpp}$ , i tempi di compressione per questo livello sono riportati nel grafico 3.4.

Un ulteriore livello di qualità si ha in corrispondenza di  $0.07\text{bpp}$ , JPEG non riesce però a comprimere fino a questi livelli, mentre la misurazione di JPEG 2000 non è stata fatta perché si è preferito prendere una misura per un livello di qualità più elevato. I tempi di compressione per questo livello sono stati comunque calcolati in quanto è interessante osservare il comportamento del codec VVC, e sono riportati nell'immagine 3.1.

L'ultimo livello di qualità invece si ha per  $0.41\text{bpp}$  dove però non abbiamo le misurazioni delle reti, in quanto non sono presenti i modelli pre addestrati per queste qualità. I tempi di compressione per quest'ultimo livello sono visibili nella figura 3.5.

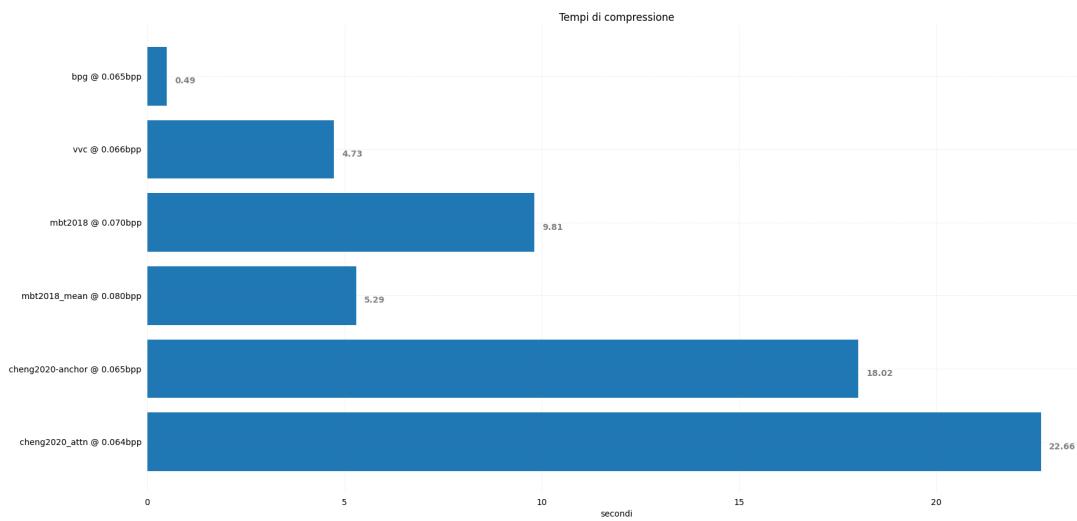


Figura 3.1: Tempi di compressione a 0.07 bpp

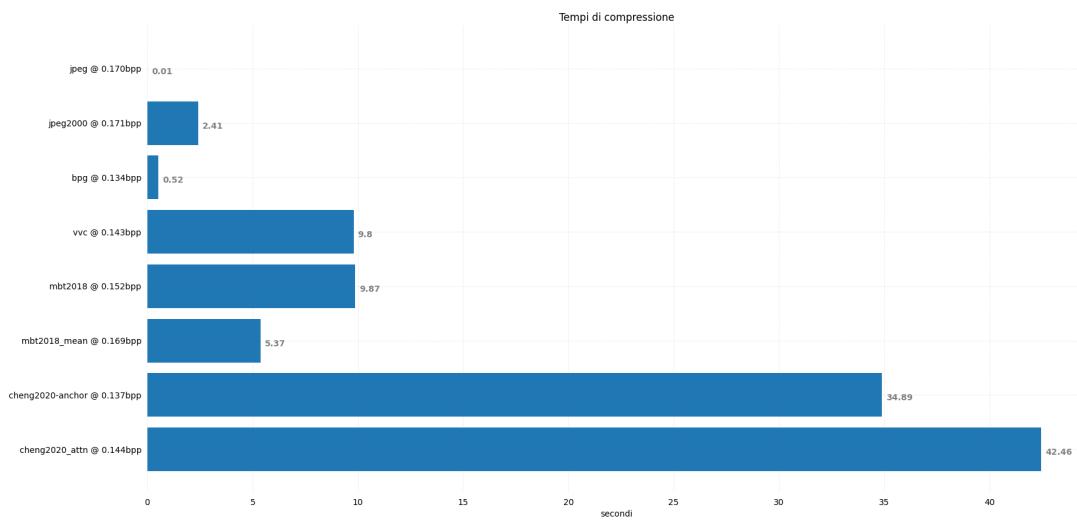


Figura 3.2: Tempi di compressione a 0.16 bpp

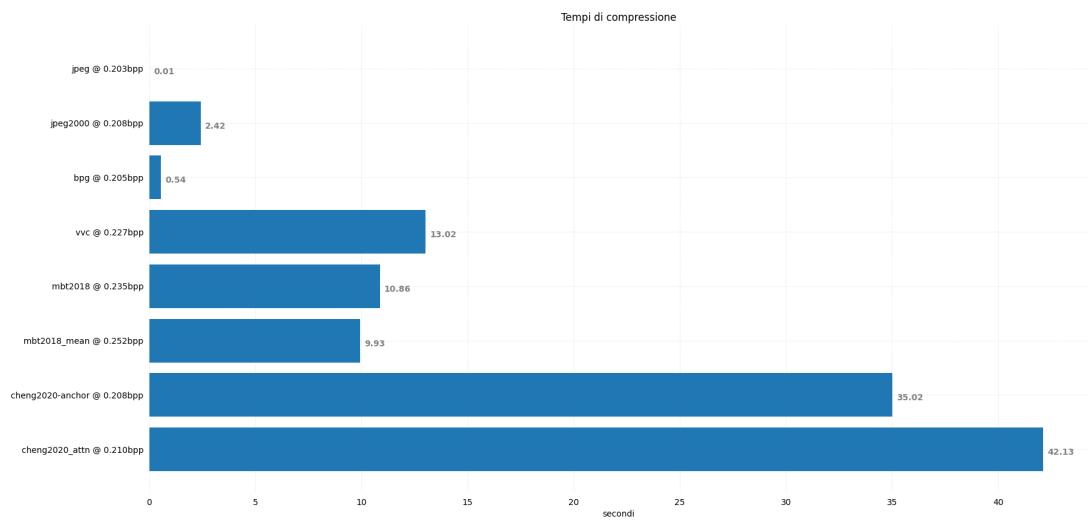


Figura 3.3: Tempi di compressione a 0.21 bpp

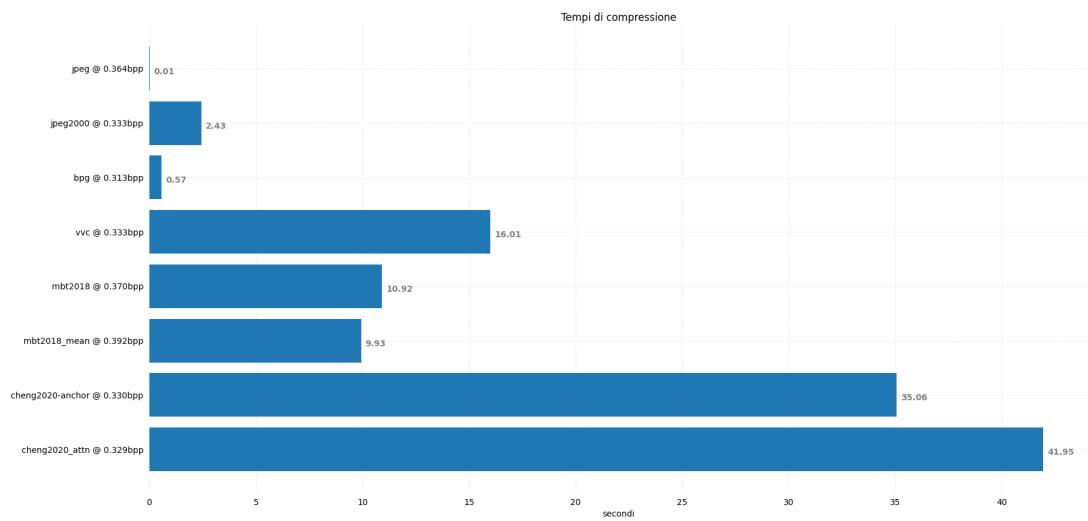


Figura 3.4: Tempi di compressione a 0.34 bpp

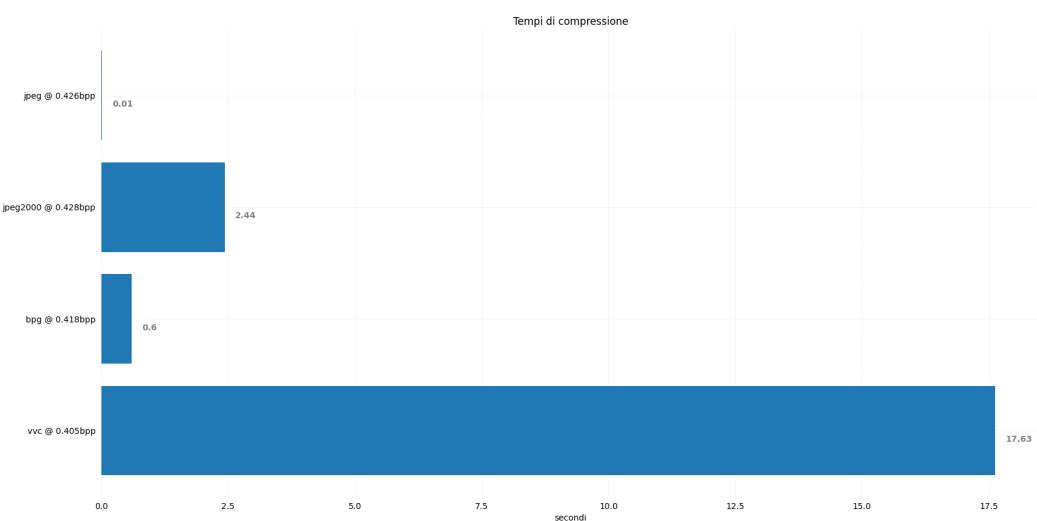


Figura 3.5: Tempi di compressione a 0.41 bpp

Come possiamo osservare nei grafici i tempi di compressione delle reti aumentano all'aumentare della qualità, i tempi di compressione dei metodi tradizionali invece non variano sensibilmente all'aumentare della qualità. Il codec VVC costituisce però un'eccezione a questo comportamento costante dei metodi tradizionali, in quanto possiamo vedere che il tempo aumenta all'aumentare della qualità. Questo comportamento si ha per la complessità di H.266, in quanto deve trovare il miglior metodo di compressione tra quelli a sua disposizione.

Passiamo ora a presentare le metriche di qualità dell'immagine, partiamo dal grafico del PSNR 3.6, come possiamo vedere nel grafico, la migliore qualità di compressione si ha con il codec VVC, seguito poi da Cheng 2020, Ballé 2018, BPG, JPEG 2000 ed infine JPEG. H.266 si conferma quindi l'attuale stato dell'arte per la compressione di immagini osservando il grafico del PSNR, che ricordiamo essere una metrica oggettiva.

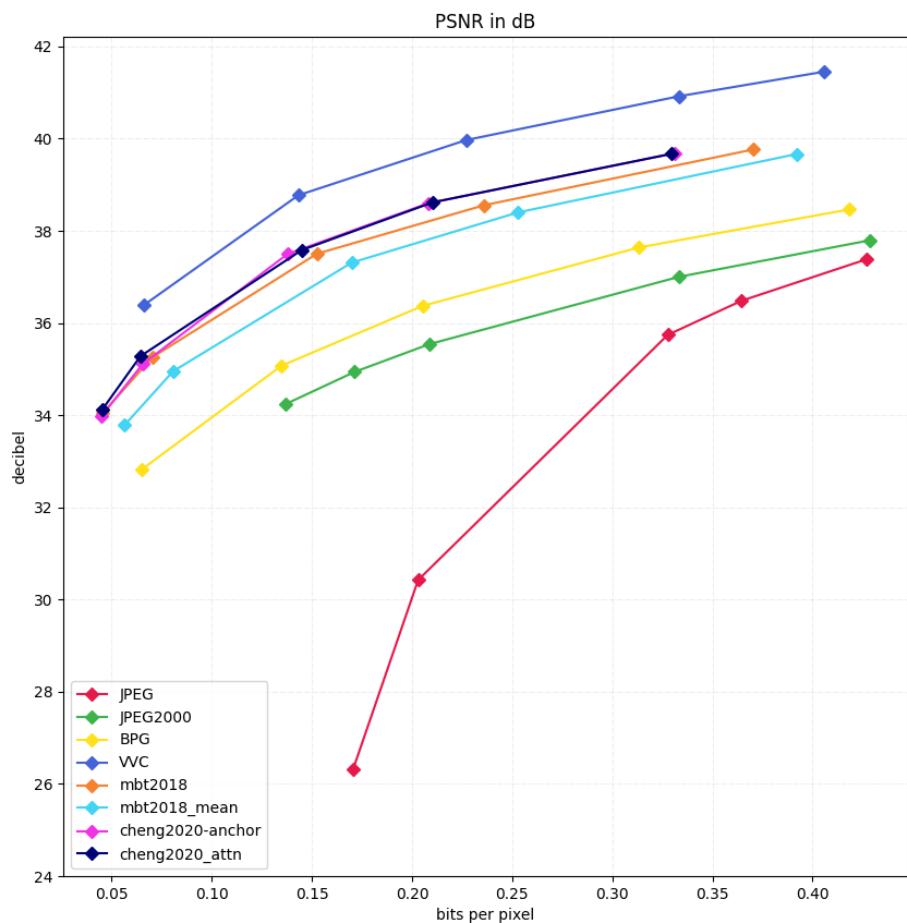


Figura 3.6: Grafico PSNR

Il prossimo grafico che andremo a commentare è il grafico dell'MS-SSIM 3.7, in questo grafico possiamo osservare come JPEG, JPEG 2000 e BPG si confermino i metodi che forniscono le qualità di compressione peggiori. Diversamente da quanto osservato nel grafico del PSNR, secondo questo grafico il miglior metodo di compressione è Cheng2020, seguito da Balle2018 e come terzo abbiamo VVC.

Ricordiamo che l'MS-SSIM è invece una metrica che cerca di replicare la valutazione del sistema visivo umano.

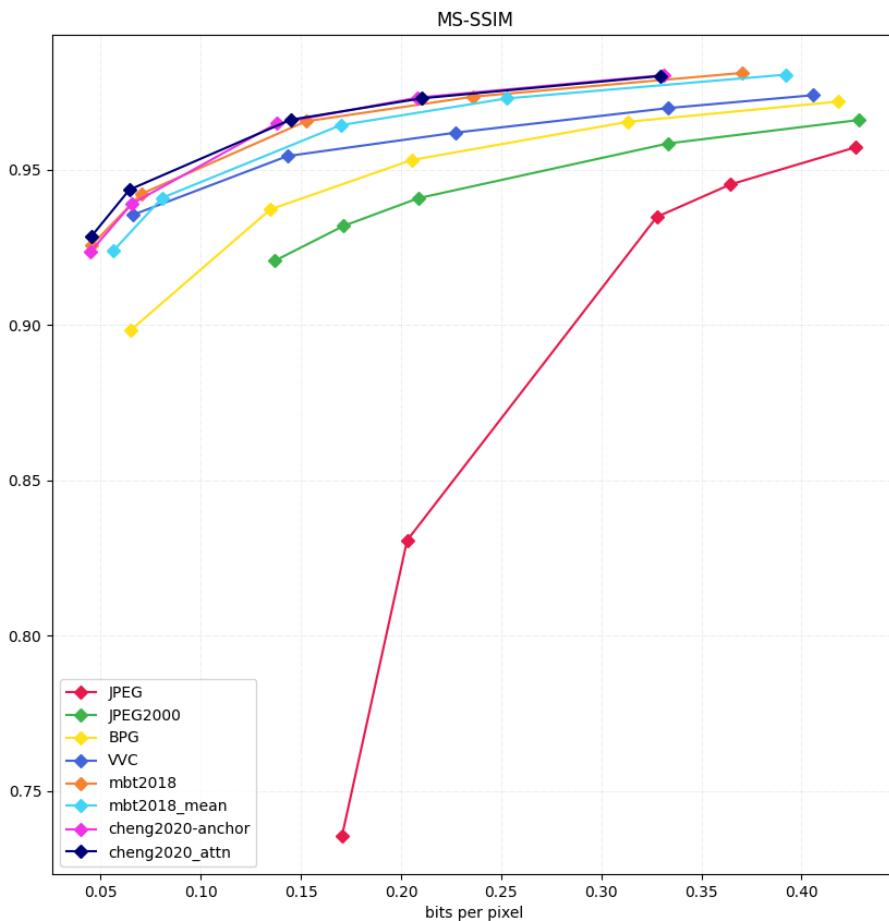


Figura 3.7: Grafico MS-SSIM

L'ultima metrica è invece la più recente, andiamo ora a presentare i risultati presenti nel grafico della metrica LPIPS con AlexNet 3.8. Come possiamo vedere in questo grafico VVC si conferma il metodo migliore, seguito da Cheng2020, Ballé2018, BPG, JPEG 2000. Per quanto

riguarda JPEG otteniamo un comportamento inaspettato in quanto i primi due punti sono in linea con le aspettative, il successivo invece è vicino a JPEG2000 e gli ultimi due lo superano addirittura.

Questo comportamento ci ha sorpreso in quanto ci aspettavamo che la qualità percepita fosse vicina a quella di JPEG 2000 ma non che la superasse.

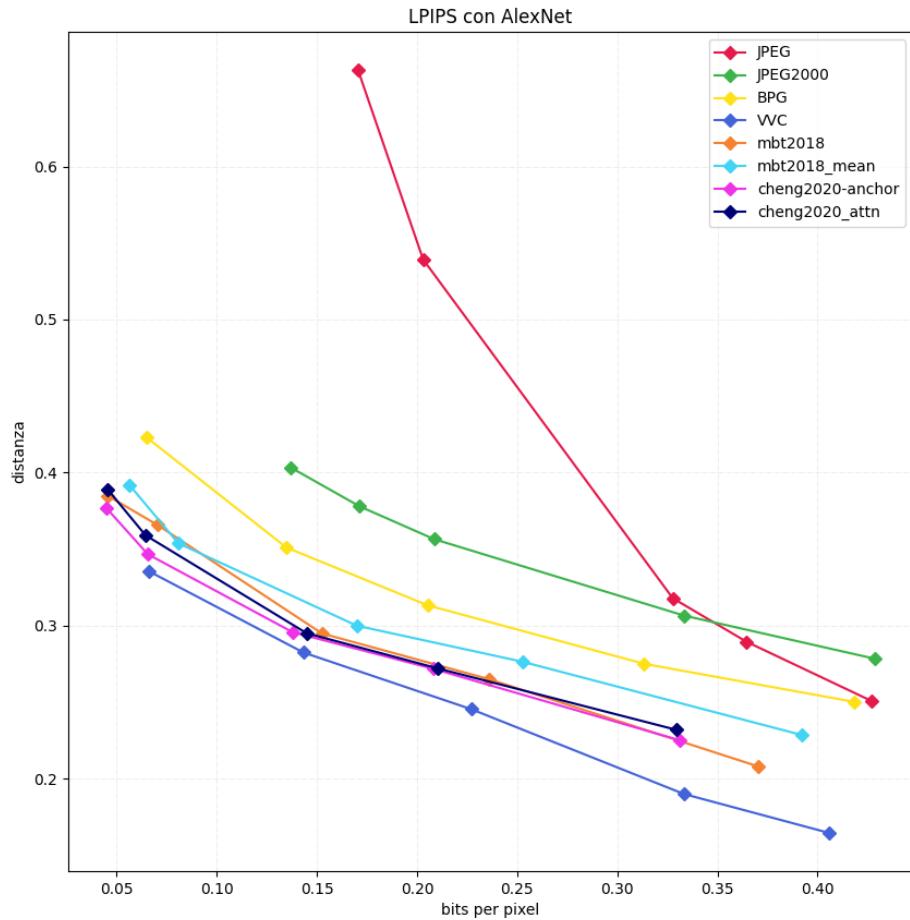


Figura 3.8: Grafico LPIPS con AlexNet

Dopo aver presentato e commentato questi grafici ci sentiamo di affermare che le reti da noi valutate svolgono un lavoro molto buono, che supera i metodi di compressione tradizionali più usati. Non riescono però ancora a superare il codec VVC che attualmente rimane lo stato dell'arte per la compressione di immagini.



# Capitolo 4

## Sviluppi futuri

### 4.1 Possibile utilizzo di SADAM

In una ricerca del 2018 Ballé introduce SpectralADAM (SADAM) [22], un’alternativa all’algoritmo ADAM [23], per l’addestramento di reti di compressione.

Se consideriamo una funzione di perdita  $L$  che consiste in una somma di funzioni di una proiezione lineare dei vettori in input  $x$  4.1, dove  $H$  è una matrice di filtri  $h_i$ . I filtri sono una parte dei parametri che devono essere ottimizzati in fase di addestramento, per ogni filtro la regola di aggiornamento del gradient descent 4.2 sottrae il gradiente della funzione di perdita moltiplicato per lo step size  $\rho$ .

$$L = \sum_x l(x) \text{ with } z = Hx \quad (4.1)$$

$$\Delta h = -\rho \frac{\partial L}{\partial h} = -\rho \sum_x \frac{\partial l}{\partial z} x \quad (4.2)$$

L’aggiornamento  $\Delta H$  consiste in somme scalari di vettori  $x$  proiettate sui filtri corrispondenti, quindi ereditano la maggior parte della struttura della covarianza dei dati in input. Di conseguenza se  $x$  fa parte di un insieme di immagini naturali abbiamo che lo spettro della potenza è inversamente proporzionale a quello della frequenza [24], dunque lo step size per componenti a basse frequenze è molto più grande rispetto a componenti ad alte frequenze, questo può portare a problemi di convergenza e in casi rari di stabilità.

Un metodo per risolvere questo problema è l’uso di un algoritmo diverso, simile al gradient descent, l’algoritmo ADAM [23], la cui funzione di aggiornamento 4.3 varia leggermente. Dove  $m_h$  è una media costantemente aggiornata della derivata  $\frac{\partial l}{\partial z}$  e  $C_h$  è una matrice diagonale

che rappresenta la stima della covarianza.

$$\Delta h = -\rho C_h^{-\frac{1}{2}} m_h \quad (4.3)$$

Essendo  $C_h$  forzata ad essere diagonale non riesce a rappresentare adeguatamente la struttura della covarianza per immagini naturali.

Per risolvere questo problema Ballé propone una variante di questo algoritmo, SpectralADAM, dove invece di applicare l'algoritmo direttamente ad  $h$ , viene applicato alla sua Trasformata Discreta di Fourier con ingresso Reale (RDFT) riparametrizzando  $h$  4.4.

$$H = F^T g \text{ with } g = Fx \quad (4.4)$$

Per ottimizzare  $g$  usiamo la derivata 4.5 ed applichiamo la regola di aggiornamento 4.3.

$$\frac{\partial l}{\partial g} = F \frac{\partial L}{\partial h} = -\frac{\partial l}{\partial z} Fx \quad (4.5)$$

Essendo 4.4 lineare possiamo calcolare l'aggiornamento  $h$  come descritto nell'equazione 4.6, dove  $m_g$  e  $C_g$  sono delle medie costantemente aggiornate delle derivate 4.5

$$\Delta h = F^T (-\rho C_g^{-\frac{1}{2}} m_g) = -\rho F^T C_g^{-\frac{1}{2}} F m_h \quad (4.6)$$

La stima della covarianza  $F^T C_g F$  deve essere diagonale nel dominio della trasformata di Fourier, non più nel dominio dei coefficienti dei filtri, e fino a quando  $x$  è invariante per permutazioni, come tutti i dati spazio temporali, la base  $F$  è garantito dia una buona approssimazione degli autovettori della vera struttura della covarianza dell'input  $x$ .

Dai risultati sperimentali della ricerca di Ballé, SADAM stabilizza e velocizza il processo di addestramento delle reti, inoltre essendo Ballé un ricercatore per Google, aveva già implementato il codice per queste soluzioni alternative all'interno della popolare libreria Tensorflow [25].

Alla luce di questi risultati ottenuti da Ballé ci chiediamo come mai negli anni successivi questo metodo non sia stato utilizzato per addestrare le nuove reti, ma si sia preferito continuare ad usare ADAM, come possiamo vedere nei lavori di Cheng et al. [12] e Wang et al. [13]. Riteniamo quindi sarebbe interessante addestrare nuovamente queste reti utilizzando SADAM e valutare la differenza di prestazioni.

## 4.2 Utilizzi in dispositivi mobili con Slim CAE

Durante la ricerca delle fonti per la stesura di questo documento ci siamo imbattuti in una ricerca molto interessante del 2021 da parte di Yang et al. [26], dove propongono di utilizzare delle SlimCAE, in modo da poter ridurre la potenza di calcolo necessaria per la compressione senza rinunciare a troppa qualità, tutto questo per poter rendere questa tecnologia fruibile anche su dispositivi con ridotta potenza di calcolo, come gli smartphone o più in generale dei dispositivi mobili.

Per rendere gli autoencoder, la cui struttura è stata descritta nel capitolo 2, degli slimmable autoencoder è necessario rendere i vali livelli della rete slimmable. Un livello, per essere slimmable, deve realizzare un’operazione valida rimuovendo parte dei parametri di quel livello. Consideriamo gli slimAE come composti da  $K$  sotto autoencoders, ognuno caratterizzato da due parametri 4.7 ed ognuno con la sua funzione di perdita 4.8. Di conseguenza i parametri di tutta la rete sono contenuti in un vettore di  $K$  coppie di parametri e la funzione di perdita altro non è che la somma pesata, con dei pesi  $w^k$ , delle  $K$  funzioni di perdita.

$$\psi^k = (\theta^{(k)}, \phi^{(k)}) \in (\theta^{(1)}, \phi^{(1)}), \dots, (\theta^{(K)}, \phi^{(K)}) \quad (4.7)$$

$$L(\Psi, \chi) = \sum_k w^k L^k(\theta^k, \phi^k; \chi) \quad (4.8)$$

Per ottenere invece delle slimCAE dobbiamo rendere tutte le operazioni nei CAE non parametriche, riducibili o rimpiazzabili senza perdere efficienza. Nella rete proposta da Yang et al. la quantizzazione non è parametrica, i livelli convoluzionali sono stati implementati per essere riducibili, per GDN e IGND [22] ci sono varie alternative che possono essere intercambiate ed infine vengono utilizzati dei modelli per l’entropia scambiabili, in questo modo ogni sotto CAE ha i suoi parametri  $\nu^k$ .

Possiamo vedere uno schema di funzionamento di una slimCAE nell’immagine 4.1, osserviamo come andando ad incrementare la dimensione della rete possiamo comprimere più dettagli, realizzando quindi una codifica progressiva.

Dai risultati sperimentali ottenuti dal team di Yang et al. [26], la rete da loro proposta ottiene risultati al pari del modello proposto da Ballé et al. [11], nonostante le dimensioni ridotte della rete e i tempi di codifica ridotti.

Ci chiediamo quindi come mai questo modello non abbia guadagnato popolarità data la sua applicabilità in situazioni più vicine alla realtà e per quale motivo le nuove reti non cerchino di realizzare anche delle versioni riducibili per permettere a sempre più persone di usufruire di tale

tecnologia.

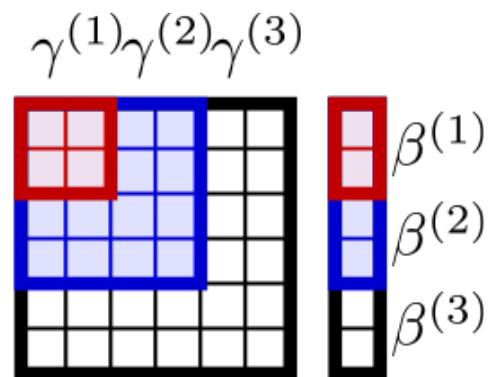


Figura 4.1: Diagramma funzionamento slimCAE, immagine presa dal documento [26]

# Capitolo 5

## Conclusioni

Questo lavoro presenta l'attuale situazione per quanto riguarda la compressione di immagini con metodi tradizionali e che fanno uso di reti neurali. Dati i recenti avanzamenti nell'ambito di ricerca ci sentiamo di affermare che, alla luce degli esperimenti da noi eseguiti, le attuali reti, Ballé et al. [11] e Cheng et al. [12], funzionano molto bene e potrebbero essere applicate nella vita di tutti i giorni, in quanto raggiungono prestazioni nettamente superiori rispetto ai metodi tradizionali più utilizzati, quali JPEG [4], JPEG 2000 [5] e BPG [6], a discapito di una compressione più lenta eseguita su CPU, come possiamo vedere dai grafici 3.1, 3.2, 3.3, 3.4.

In casi specifici, dove è richiesta una qualità di compressione superiore invece, i metodi tradizionali e nello specifico VVC [7], rappresentano ancora l'unica alternativa sensata in quanto permettono di comprimere di più rispetto alle reti, garantendo una migliore qualità oggettiva e soggettiva, come possiamo osservare rispettivamente nei grafici 3.6 e 3.8.

L'ambito di ricerca della compressione con reti neurali rimane comunque molto promettente e con la crescente disponibilità di potenza di calcolo e di esempi di addestramento non potrà far altro che continuare a migliorare ulteriormente. Alcune migliorie, che potrebbero essere oggetto di future ricerche, potrebbero essere l'uso di SpectralADAM [22] per l'addestramento delle reti di compressione e l'attenzione nello sviluppo di reti riducibili, come slimCAE [26] sviluppata dal team di Yang et al. nel 2021, per permettere l'uso di questi metodi anche su dispositivi con ridotta potenza di calcolo.

La naturale prosecuzione di questo lavoro sarebbe la valutazione di metodi che fanno uso di reti neurali per quanto riguarda la codifica video, valutando le prestazioni dei metodi tradizionali attualmente esistenti e compararli con metodi più recenti che fanno uso di intelligenza artificiale e metodi ibridi.



# Bibliografia

- [1] H. T. Sadeeq, T. H. Hameed, A. S. Abdi e A. N. Abdulfatah, «Image compression using neural networks: a review,» *International Journal of Online and Biomedical Engineering (iJOE)*, vol. 17, n. 14, pp. 135–153, 2021.
- [2] Y. Hu, W. Yang, Z. Ma e J. Liu, «Learning end-to-end lossy image compression: A benchmark,» *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, n. 8, pp. 4194–4211, 2021.
- [3] Z. Cheng, H. Sun, M. Takeuchi e J. Katto, «Deep convolutional autoencoder-based lossy image compression,» in *2018 Picture Coding Symposium (PCS)*, IEEE, 2018, pp. 253–257.
- [4] G. Wallace, «The JPEG still picture compression standard,» *IEEE Transactions on Consumer Electronics*, vol. 38, n. 1, pp. xviii–xxxiv, 1992. doi: 10.1109/30.125072.
- [5] A. Skodras, C. Christopoulos e T. Ebrahimi, «The JPEG 2000 still image compression standard,» *IEEE Signal Processing Magazine*, vol. 18, n. 5, pp. 36–58, 2001. doi: 10.1109/79.952804.
- [6] F. Bellard, *BPG Image format*, <https://bellard.org/bpg/>, Consultato: 17-10-2023.
- [7] B. Bross, Y.-K. Wang, Y. Ye et al., «Overview of the Versatile Video Coding (VVC) Standard and its Applications,» *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, n. 10, pp. 3736–3764, 2021. doi: 10.1109/TCSVT.2021.3101953.
- [8] D. Mishra, S. K. Singh e R. K. Singh, «Deep architectures for image compression: a critical review,» *Signal Processing*, vol. 191, p. 108346, 2022.
- [9] L. Theis, W. Shi, A. Cunningham e F. Huszár, «Lossy image compression with compressive autoencoders,» *arXiv preprint arXiv:1703.00395*, 2017.
- [10] J. Ballé, D. Minnen, S. Singh, S. J. Hwang e N. Johnston, «Variational image compression with a scale hyperprior,» *arXiv preprint arXiv:1802.01436*, 2018.

- [11] D. Minnen, J. Ballé e G. D. Toderici, «Joint autoregressive and hierarchical priors for learned image compression,» *Advances in neural information processing systems*, vol. 31, 2018.
- [12] Z. Cheng, H. Sun, M. Takeuchi e J. Katto, «Learned image compression with discretized gaussian mixture likelihoods and attention modules,» in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 7939–7948.
- [13] D. Wang, W. Yang, Y. Hu e J. Liu, «Neural data-dependent transform for learned image compression,» in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 17379–17388.
- [14] J. A. Clark e contributors, *Pillow, a PIL fork*, <https://python-pillow.org/>, Consultato: 17-10-2023.
- [15] U. de Louvain, *OpenJPEG*, <https://www.openjpeg.org/>, Consultato: 17-10-2023.
- [16] F. HHI, *H.266 / VVC*, <https://www.hhi.fraunhofer.de/en/departments/vca/technologies-and-solutions/h266-vvc.html>, Consultato: 17-10-2023.
- [17] I. InterDigital Communications, *CompressAI*, <https://interdigitalinc.github.io/CompressAI/>, Consultato: 25-10-2023.
- [18] E. K. Company, *True Color Kodak Images*, <https://r0k.us/graphics/kodak/>, Consultato: 17-10-2023.
- [19] Z. Wang, E. P. Simoncelli e A. C. Bovik, «Multiscale structural similarity for image quality assessment,» in *The Thirly-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, Ieee, vol. 2, 2003, pp. 1398–1402.
- [20] R. Zhang, P. Isola, A. A. Efros, E. Shechtman e O. Wang, «The unreasonable effectiveness of deep features as a perceptual metric,» in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 586–595.
- [21] F. Stella, *Image Compression AI*, <https://github.com/ilofX/ImageCompressionAI>, Consultato: 07-11-2023.
- [22] J. Ballé, «Efficient nonlinear transforms for lossy image compression,» in *2018 Picture Coding Symposium (PCS)*, IEEE, 2018, pp. 248–252.
- [23] D. P. Kingma e J. Ba, «Adam: A method for stochastic optimization,» *arXiv preprint arXiv:1412.6980*, 2014.
- [24] D. J. Field, «Relations between the statistics of natural images and the response properties of cortical cells,» *Josa a*, vol. 4, n. 12, pp. 2379–2394, 1987.

- [25] M. Abadi, A. Agarwal, P. Barham et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. indirizzo: <https://www.tensorflow.org/>.
- [26] F. Yang, L. Herranz, Y. Cheng e M. G. Mozerov, «Slimmable compressive autoencoders for practical neural image compression,» in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 4998–5007.