POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Matematica

Tesi di Laurea Magistrale

Learning Transforms for Image Compression

Relatore:
prof. Enrico Magli

Candidato:
Adele Guasco

Tutore Aziendale
CRITS Rai
Ing. Roberto Iacoviello

A.A. 2017-2018

Ringraziamenti

Vorrei ringraziare tutte e tutti coloro che hanno contribuito significativamente affinché potessi raggiungere questo traguardo. Innanzitutto, ringrazio il mio relatore, il professor Magli e il mio tutor aziendale l'ingegner Iacoviello, per avermi dato l'opportunità di realizzare questa tesi all'interno della Rai e per avermi guidata nella sua stesura. Vorrei ringraziare anche i docenti di tutti corsi che ho seguito per i loro insegnamenti, per essere stati sempre disponibili a sciogliere dubbi e fornire chiarimenti. Non posso non ringraziare le mie compagne e i miei compagni di studi, in particolare Virginia, Federica, Giada, Luca e il mio fidanzato Matteo, per essere stati un sostegno sia morale sia concreto nello studio. Infine ringrazio di cuore la mia famiglia che mi supporta e mi incoraggia da sempre, in particolare mia madre, mio padre, mio fratello e i miei nonni, anche se alcuni di loro non ci sono più so che questo mio risultato li renderebbe orgogliosi.

Sommario

Questa tesi è stata realizzata in collaborazione con il CRITS-Rai, nello specifico all'interno del gruppo audio-video. In un'azienda di broadcast, come la Rai, il centro ricerche ha lo scopo di studiare le nuove tecnologie, sperimentando i nuovi sistemi e servizi, per adattarle in modo efficiente ai bisogni dei clienti. Attualmente gli standard di compressione utilizzano nei loro algoritmi trasformate che per bassi valori del bit-rate mostrano artefatti di compressione piuttosto fastidiosi. Per risolvere questo problema una possibile soluzione che si sta considerando negli ultimi anni è l'utilizzo di reti neurali per il calcolo delle trasformate, considerato il loro potenziale in svariati altri ambiti.

L'obiettivo di questa tesi è sperimentare questo tipo di architettura, per esplorarne le potenzialità nel calcolo di trasformate e confrontarne le prestazioni con una delle trasformate più note, la DCT (Discret Cosine Transform), in termini di bit-rate e PSNR (Peak Signal to Noise Ratio). Considerando una possibile applicazione al video sono stati svolti esperimenti considerando immagini partizionate in blocchi, poiché questo tipo di elaborazione è necessaria per tutte le tecniche video di motion estimation. Nel caso di compressione lossy questo significa sviluppare trasformate in grado di minimizzare sia il rate sia la distorsione del segnale ricostruito, che in pratica si traduce in un problema di ottimizzazione del trade-off tra le due grandezze. Il sistema di compressione per immagini considerato calcola un set di trasformate tramite strati di filtri convoluzionali. L'obiettivo è quello di usare questo sistema, che di fatto è una CNN (Convolutional Neural Network), per ottimizzare la curva rate-distortion e di confrontare la performance con quella dello standard JPEG, che utilizza la DCT come trasformata. La funzione di attivazione usata è la GDN (Generalized Divisive Normalization), ispirata al modello dei neuroni nel sistema visivo. Il risultato di questa trasformata viene poi arrotondato da un quantizzatore scalare uniforme. L'immagine compressa viene ricostruita usando la trasformata inversa. Per ogni scelta del parametro di trade-off del rate-distortion, i parametri della trasformata diretta e di quella inversa sono ottimizzati con il metodo SGD (Stochastic Gradient Descendent). Per via del quantizzatore scalare uniforme la funzione di costo è discontinua, ossia con gradiente nullo quasi ovunque. Per risolvere questo inconveniente si riformula la funzione da ottimizzare con un rilassamento continuo. Infine per valutare la performance di questa trasformata rispetto alla DCT si stima il bit-rate, con un codificatore aritmetico.

Dai risultati che si sono ottenuti, si può affermare che le trasformate di tipo learned, apprese mediante questa particolare architettura, sono una valida alternativa nell'ottimizzazione del trade-off, per la compressione immagini, rispetto alle trasformate fin ora utilizzate dagli standard più noti. L'apprendimento di trasformate per la compressione è una tematica molto promettente, che richiede sicuramente ulteriori approfondimenti e miglioramenti visti i risultati ottenuti negli ultimi anni. Questo tipo di studi sta ispirando l'ambito direttamente legato alla compressione immagini con Machine Learning, cioè quello della compressione video.

Indice

Ringraziamenti II

Sommario III

1 Introduzione 1
1.1 Richiami alla Compressione Dati . . . 1
1.1.1 Compressione *lossless* e *lossy* . . . 2
1.1.2 Le *quality metrics* . . . 3
1.1.3 Ridondanza dei dati . . . 3
1.2 Cenni di Teoria dell'Informazione . . . 3
1.2.1 I limiti dell'entropia . . . 4
1.2.2 Teoria del *rate-distortion* . . . 4
1.3 Trasformate . . . 5
1.3.1 Trasformata Karhunen-Loeve (KLT) . . . 6
1.3.2 Discrete Cosine Transform (DCT) . . . 6
1.3.3 Sterable-DCT (SDCT) . . . 6
1.3.4 Mode Directional Dependent Transform (MDDT) . . . 7
1.4 Codificatore Entropico . . . 7
1.4.1 Codifica Huffman . . . 8
1.4.2 Codifica Aritmetica . . . 8
1.5 Reti Neurali . . . 9
1.5.1 Gradient Descendent (GD) . . . 10
1.5.2 Algoritmo Back Propagation (BP) . . . 11
1.5.3 Reti Neurali Convoluzionali (CNN) . . . 13
1.6 Struttura della tesi . . . 15

2 Progettazione e sviluppo 16
2.1 Ottimizzazione . . . 17
2.2 Architettura del sistema di trasformate . . . 19
2.2.1 General Divisive Normalization e trasformate . . . 21
2.3 Codice della catena di elaborazione . . . 21
2.3.1 Codice Matlab per immagini in scala di grigi . . . 22
2.3.2 Codice Matlab per immagini a colori . . . 28
2.3.3 Codificatore entropico aritmetico per calcolare bit-rate e PSNR . . . 28
2.4 Requisiti del progetto . . . 31

3 Test e Risultati 32
3.1 Risultati su immagini singole . . . 33
3.1.1 Immagine "girl" . . . 33
3.1.2 Immagine "Lena" . . . 35
3.1.3 Immagine "monkey" . . . 36
3.1.4 Immagine "pepper" . . . 38
3.1.5 Confronto della percezione delle ricostruite . . . 39
3.2 Risultati in media . . . 45
3.3 Confronto tra le due trasformate . . . 47
3.4 Risultati con immagini HD e 4K . . . 49
3.4.1 Immagine HD "run" . . . 50
3.4.2 Immagine 4K "football" . . . 51

4 Conclusioni e sviluppi futuri 54

Riferimenti bibliografici 56

Capitolo 1

Introduzione

In questo capitolo introduttivo si trattano brevemente alcuni argomenti che servono da richiami teorici per comprendere meglio il lavoro svolto. Inizialmente ci si sofferma sugli aspetti principali della compressione immagini e poi quelli relativi alle reti neurali. Nell'ultimo paragrafo è specificata la struttura dei capitoli successivi. Per questa parte sulla compressione si è fatto riferimento a [12] e per le reti neurali a [15].

1.1 Richiami alla Compressione Dati

La compressione dati è definita come la scienza che cerca di rappresentare l'informazione nella forma più compatta possibile. Inizialmente, questa esigenza era dovuta al fatto che le tecniche di trasmissione e memorizzazione dei dati non avevano ancora raggiunto i livelli attuali. Attualmente, grazie agli sviluppi tecnologici, queste capacità stanno costantemente aumentando. Tuttavia, contemporaneamente si assiste anche all'aumentare, più velocemente ancora, del bisogno di trasmettere e memorizzare grandi quantità di dati. Per fare un esempio, nella trasmissione digitale di un secondo di video in alta definizione, senza compressione, occorrono più di 3 Gb/s, mentre per il 4K ne occorrono 12 Gb/s. Dunque è facile vedere, moltiplicando questa quantità per il numero di secondi in un film, come la compressione dati abbia un ruolo fondamentale all'interno dell'innovazione tecnologica, se ad esempio questo video deve essere trasmesso da un'emittente televisiva oppure scaricato da un'applicazione smartphone.

Per poter elaborare, comprimere e trasmettere un'immagine è necessario rappresentarla in formato digitale, cioè come una sequenza di bit. Ciò è possibile tramite le operazioni di campionamento e quantizzazione.

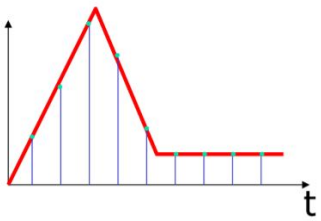

Figura 1.1: Campionamento di un segnale continuo.

Il campionamento consiste nel prelevare solo alcuni campioni del segnale nel tempo, si veda l'immagine 1.1, mentre la quantizzazione nell'assegnare un numero discreto di valori alle frequenze reali misurate che invece richiederebbero una rappresentazione a precisione infinita, si veda

la figura 1.2. Quando si parla di algoritmi di compressione in realtà ci si riferisce implicitamente a due algoritmi. Il primo, che dato il segnale X in input, lo comprime e ne genera una rappresentazione Y più compatta, quindi con meno bit. Il secondo, opera nel senso inverso, data la rappresentazione Y ricostruisce il segnale di partenza con $\hat{X}$, che non coincide necessariamente con l'originale X.

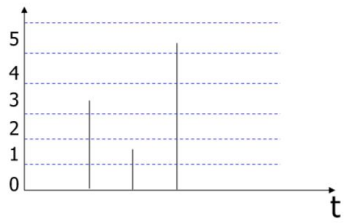

Figura 1.2: Quantizzazione di un segnale campionato.

1.1.1 Compressione *lossless* e *lossy*

In base al risultato della fase di ricostruzione, gli algoritmi di compressione si dividono in due categorie.

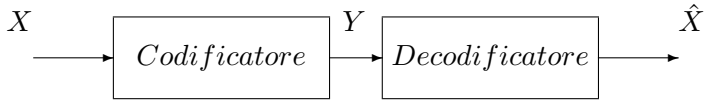

Figura 1.3: Schema di un generico algoritmo di compressione. X è il dato iniziale, Y è la rappresentazione compressa di X e $\hat{X}$ è il dato ricostruito.

La prima è chiamata compressione *lossless*, cioè senza perdita d'informazione, ed è il caso in cui $X = \hat{X}$. In questo caso il dato originale si ricava completamente dal dato compresso Y. Non essendo tollerata nessuna variazione tra dato in input e dato ricostruito in output, questo genere di algoritmi trova importanti applicazione in ambiti come la compressione testuale, in contesti di tipo bancario, e la diagnostica medica che non verrà qui trattata.

La seconda invece è chiamata compressione *lossy*, cioè con perdita di informazione, dove $X \approx \hat{X}$. Nonostante non siano uguali dato in input e dato in output, è tollerata una certa quantità di errore purché l'informazione non sia stravolta per la percezione umana. Chiaramente è facile capire che questo tipo di algoritmi permette di raggiungere un maggiore rapporto di compressione. Quest'ultimo è definito come il rapporta tra la quantità che occupa il dato originale e la quantità che occupa il dato compresso. Questo tipo di algoritmi è più diffuso poiché ha un campo di applicazioni decisamente più vasto del primo. In seguito si farà sempre riferimento a quest'ultima tipologia e non alla precedente.

Una cosa che hanno in comune tutti gli algoritmi di compressione sono la fase di *modeling* e di *coding*. La prima riguarda la ricerca di un modello che sia in grado di descrivere la ridondanza presente nei dati, che sarà condiviso tra il codificatore e il decodificatore. L'altra invece riguarda la scelta dei parametri del modello e come rappresentare la differenza tra dato input e dato ricostruito che proviene dal modello, cioè il *residual coding*.

1.1.2 Le *quality metrics*

Per confrontare i diversi algoritmi ci sono diversi criteri che riguardano vari aspetti, come la complessità computazionale, la quantità di memoria richiesta, la velocità di esecuzione, il livello di compressione o la bontà della ricostruzione del segnale. Qui si considereranno gli ultimi due criteri. Il livello di compressione consiste nella valutazione del rapporto di compressione oppure nel numero medio di bit richiesti per rappresentare un singolo campione, cioè il rate.

Nella compressione *lossy*, siccome il dato ricostruito è diverso dall'originale, è importante quantificare questa differenza in termini di distorsione. Per le immagini le misure di qualità più usate sono il *mean square error* (MSE) e il *peak signal to noise ratio* (PSNR), definite con le seguenti formule, dove N è il numero di pixel dell'immagine, $X[n]$ e $\hat{X}[n]$ sono i valori che assume il pixel n-esimo dell'immagine in input e nella ricostruita, sulla scala di grigio che va da 0 a 255, per immagini in bianco e nero, con 8 bit per campione:

$$MSE(X, \hat{X}) = \frac{1}{N} \sum_{n=1}^{N} (X[n] - \hat{X}[n])^2 \tag{1.1}$$

$$PSNR(X, \hat{X}) = 10 \log_{10} \frac{max^2(X)}{MSE(X, \hat{X})} [dB] \tag{1.2}$$

1.1.3 Ridondanza dei dati

L'obiettivo di questi algoritmi può anche definirsi come il tentativo di ridurre la ridondanza intrinseca nei dati, per trovarne una rappresentazione più compatta, ossia che richieda un minor numero di bit per essere memorizzata o trasmessa. La ridondanza è presente in vari aspetti.

Il primo riguarda la codifica dei campioni, ad esempio nel caso di un'immagine in bianco e nero ciò vuol dire quanti bit si assegnano per rappresentare ogni livello della scala di grigi. Se si definisce $p(k) = n_k/n$ la probabilità di occorrenza del livello k-esimo con $k = 0,1, ..., L - 1$, nell'esempio dell'immagine in bianco e nero $L = 255$, e $l(k)$ il numero di bit con cui è rappresentato il livello k-esimo, si può calcolare il numero medio di bit per pixel con la seguente formula:

$$L_{avg} = \sum_{k=0}^{L-1} l(k)p(k). \tag{1.3}$$

Se si assegna lo stesso numero di bit per ogni livello $l(k) = m$, allora $L_{avg} = m$, quindi non si sta sfruttando la ridondanza. Di solito si cerca di applicare un *variable length coding* (VLC), che consiste nell'assegnare pochi bit ai livelli che hanno una probabilità di occorrenza più alta e più bit a quelli che hanno una probabilità di occorrenza bassa.

Sicuramente in un'immagine i pixel adiacenti hanno intensità correlate tra loro, basta pensare alle zone uniformi naturalmente presenti. Questo è un secondo aspetto dove è possibile sfruttare la ridondanza in un'immagine per mezzo di modelli predittivi o di trasformate.

1.2 Cenni di Teoria dell'Informazione

La teoria dell'informazione offre gli elementi che servono come fondamento matematico per la compressione. Permette di definire in modo quantitativo l'informazione presente in un dato, rappresentarla in modo efficiente e trasmetterla attraverso i canali di comunicazione. Per poter fare tutto questo si consideri una sorgente di informazione, senza memoria e discreta, in grado di generare simboli da un insieme $X = \{x_i\}_{i=1}^{M}$ di M elementi, detto alfabeto. Questo significa che genera simboli statisticamente indipendenti. Inoltre, a ogni simbolo è associata la sua probabilità

di occorrenza con $p_i, \forall i = 1,2,...,M$. A questo punto l'informazione del i-esimo simbolo x_i è definita dalla formula 1.4 e la sua unità di misura sono i bit.

$$I(x_i) = log_2 \frac{1}{p_i} = -\log_2 p_i \tag{1.4}$$

La quantità d'informazione di una sequenza di simboli sarà inversamente proporzionale alla sua probabilità, cioè $I(x_i) > I(x_j)$ se $p_i < pj$. Quindi simboli con una probabilità di occorrenza bassa, cioè rari, hanno una quantità di informazione maggiore. Invece la probabilità di occorrenza di due simboli, essendo questi statisticamente indipendenti è $P(x_i, x_j) = P(x_i)P(x_j)$, da cui ne consegue che la rispettiva informazione $I(x_i x_j)$ è data dalla somma dell'informazione di ogni singolo simbolo $I(x_i) + I(x_j)$.

Una seconda grandezza utile è il valore atteso dell'informazione della sorgente X, detta entropia, e calcolata con la formula 1.5 e misurata in bit/simbolo.

$$H(X) = \sum_{i=1}^{M} p_i I(x_i) = \sum_{i=1}^{M} p_i \log_2 \frac{1}{p_i} \tag{1.5}$$

Meno uniforme è la distribuzione delle occorrenze più basso è il valore dell'entropia.

1.2.1 I limiti dell'entropia

Un risultato teorico importante è il calcolo esplicito del limite superiore dell'entropia di X, definito nella formula 1.6, in cui l'uguaglianza si verifica nel caso dei simboli equiprobabili.

$$H(X) \leq \log_2 M \tag{1.6}$$

Ciò permette di codificare, senza perdita, una sorgente di informazione X con entropia $H(X)$, usando $H(X) + \epsilon$ bit/simbolo. La ϵ è una quantità che può essere arbitrariamente piccola se si considerano blocchi di simboli sempre più grandi. Da questo risultato si possono dedurre due fatti. Primo, la lunghezza media in bit per codificare un simbolo [1.3] non può essere minore dell'entropia. Secondo, l'entropia è il target ottimale in termini di numero medio di bit/simbolo a cui tende un codificatore *lossless*, al considerare blocchi di simboli che hanno dimensione crescente.

In pratica, l'approccio che consiste nel considerare blocchi di simboli di taglia via via crescente per avere un numero medio di bit/simbolo vicino al valore ottimale è troppo complesso, perché si deve anche considerare la correlazione tra simboli. In alternativa, risulta più pratico usare modelli che siano in grado di sfruttare la correlazione, generando una nuova sequenza di simboli indipendenti tra loro, e successivamente codificare i nuovi simboli individualmente.

1.2.2 Teoria del *rate-distortion*

Come si è detto precedentemente, la differenza degli algoritmi *lossy* rispetto a quelli di tipo *lossless*, sta nel fatto che è accettato un certo grado di distorsione nel segnale ricostruito. In questo caso, invece di concentrarsi solo sul numero medio di bit per pixel, cioè il rate, ci si concentra anche sulla quantità di informazione persa, cioè la distorsione. L'ideale sarebbe ottenere poca distorsione con un altrettanto basso bit-rate, tuttavia è ovvio che ciò che si fa non è altro che cercare un trade-off tra il rate e la distorsione. Questa parte della teoria dell'informazione è anche detta teoria del *rate-distortion*. Per studiare questo trade-off si potrebbe cercare di esprimere il rate come una funzione per un particolare valore di distorsione, ma purtroppo non è così semplice e quindi si deve affrontare il problema diversamente. Per cominciare è utile dare alcune definizioni.

Si consideri un processo random discreto e i.i.d., si definisce una metrica di distorsione per una realizzazione ricostruita $\hat{X}$ data la realizzazione originale X come:

$$D(X, \hat{X}) = \frac{1}{m} \sum_{j=1}^{m} d(x_j, \hat{x}_j) \tag{1.7}$$

dove m è la lunghezza della realizzazione e $d(x_j, \hat{x}_j) = (x_j - \hat{x}_j)^2$ è una misura di distorsione componente per componente. Da notare che la 1.1 e la 1.2 sono degli esempi.

All'inizio paragrafo 1.2 è stato introdotto il concetto di alfabeto X. Si consideri ora sempre X ma che invece di essere discreto è continuo e si consideri un secondo alfabeto $\hat{X}$, l'alfabeto ricostruito che in generale è diverso dal primo ma con le stesse caratteristiche di indipendenza. Se entrambe le sorgenti di informazioni che si originano da X e $\hat{X}$ si distribuiscono come una Gaussiana a media zero e varianza σ^2, si può dimostrare che esiste una formula esplicita della curva per il rate in funzione della distorsione che è la seguente:

$$R(D) = \begin{cases} \frac{1}{2} \log \frac{\sigma^2}{D}, \; se \; D < \sigma^2 \\ 0, \; altrimenti \end{cases} \tag{1.8}$$

Fissato un particolare valore per l'attesa della distorsione come segue:

$$\mathbb{E}[D(X, \hat{X})] \leq D \tag{1.9}$$

il valore della funzione $R(D)$ è il minimo rate per cui è garantita l'esistenza di una sorgente di informazione che rappresenti X con $\hat{X}$, tale che $\hat{X}$ è codificato con un rate pari a R.

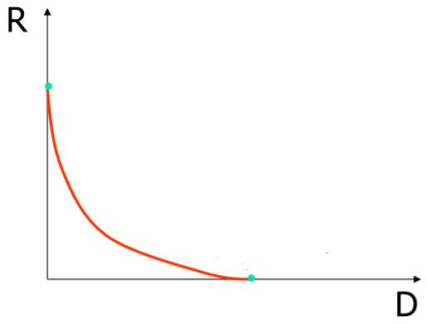

Figura 1.4: Plot del rate in funzione della distorsione.

Questo risultato è utile, perché essendo più bassa della funzione *rate-distortion* di qualunque altra distribuzione, fornisce un limite superiore nel caso in cui non è possibile calcolare esplicitamente la curva. In quest'ultimo caso si calcola la *operational R-D function*, un insieme di punti *rate-distortion* raggiungibili in pratica per una specifica realizzazione della sorgente e per una specifica codifica.

1.3 Trasformate

Per migliorare la quantizzazione e la codifica di un segnale vettoriale X, questo viene rappresentato in un dominio diverso da quello dei dati, con Y. In questo modo si modellizza la correlazione tra i campioni di X per concentrare la maggior parte dell'energia nei primi coefficienti di Y. In altre parole, una trasformazione lineare unitaria T, applica una proiezione di X, ossia una decomposizione in termini di funzioni base. Di seguito si specificheranno alcune trasformate d'interesse per il lavoro svolto.

1.3.1 Trasformata Karhunen-Loeve (KLT)

La trasformata di Karhunen-Loeve è anche conosciuta con il nome di Analisi delle Componenti Principali (PCA). È una trasformata adattativa, che si basa sulla decomposizione agli autovettori della matrice di covarianza C_X di X, in modo che l'i-esima colonna di T corrisponda al i-esimo autovettore di C_X. Una volta applicata T a X, si ottiene un vettore risultante, $Y = TX$, i cui elementi nei coefficienti trasformati sono scorrelati e la cui matrice di covarianza $C_Y = diag\{\lambda_1, \lambda_2, \ldots, \lambda_N\}$ è una matrice diagonale, i cui elementi sono gli autovettori di C_X.

Rispetto ad altre trasformate è piuttosto onerosa da calcolare, ma possiede una proprietà importante. Se si mantengono solo i primi K coefficienti del vettore trasformato Y, che corrispondono ai K autovettori più grandi, allora l'MSE tra il vettore originale e il ricostruito troncato è teoricamente ottimo rispetto a un troncamento dei primi K coefficienti di qualunque altra trasformata.

1.3.2 Discrete Cosine Transform (DCT)

È la trasformata che viene utilizzata dallo standard JPEG e che prende il nome dal fatto che le funzioni base della trasformata sono funzioni di coseni ed è definita nel modo seguente:

$$y(u) = \alpha(u) \sum_{n=0}^{N-1} x(n) \cos\left(\frac{(2n+1)u\pi}{2N}\right) \tag{1.10}$$

dove $u = 0,1,\ldots,N-1$ e $\alpha(u)$ è una costante.

Quella che si applica alle immagini è una 2-D DCT, che si ottiene calcolando la 1-D DCT indipendentemente alle righe e poi alle colonne. L'immagine viene precedentemente partizionata in blocchi di pixel 8×8. Il risultato della trasformata applicata ad un blocco dell'immagine è una matrice 8×8, i cui valori corrispondono alla quantità presente nel blocco di ogni funzione base della trasformata. Inoltre, bisogna far presente il fatto che la correlazione tra i bordi dei blocchi non viene eliminata. La DCT possiede la proprietà per cui se $N \to \infty$, la matrice dei coefficienti tende ad una matrice diagonale che è quella della trasformata KLT.

Essendo una trasformata separabile per righe e colonne riesce a comprimere l'energia nelle discontinuità orizzontali e verticali. Quindi nel caso di discontinuità arbitrariamente orientate non si ottengono delle buone performance. Per questo motivo sono state implementate altre trasformate che sono in grado di adattarsi alle specifiche direzioni caratteristiche dell'immagine, che sono di tipo direzionale o di tipo learned.

1.3.3 Sterable-DCT (SDCT)

In [6] è descritto nel dettaglio l'implementazione di questa trasformata che ha la peculiarità di poter essere orientata in qualunque direzione. Viene descritto il legame tra DCT e la *Graph Fourier Transform*, in particolare, se si considera il grafo a griglia $n \times n$ si può dimostrare che gli autovettori della matrice Laplaciana L sono gli stessi n vettori base della DCT. È anche possibile notare che la 2D-DCT non è l'unica base di autovettori per il Laplaciano del grafo a griglia, questo è dovuto al fatto che non tutti gli autovalori hanno molteplicità algebrica pari a 1. Quindi si parte da queste considerazioni per cercare altre basi di autovettori, scegliendo quella che meglio si adatta alle caratteristiche di uno specifico blocco da comprimere. Dato l'autovettore $\lambda_{k,l}$ di L, avente spazio vettoriale di dimensione 2, e dati i vettori della 2D-DCT $\boldsymbol{v}^{(k,l)}$ e $\boldsymbol{v}^{(l,k)}$ che sono anche gli autovettori di L corrispondenti a $\lambda_{k,l}$, allora si può ottenere qualunque altra base dello spazio vettoriale corrispondente a $\lambda_{k,l}$ come il risultato di una rotazione dei due vettori.

$$\begin{bmatrix} \boldsymbol{v}^{(k,l)'} \\ \boldsymbol{v}^{(l,k)'} \end{bmatrix} = \begin{bmatrix} \cos\theta_{k,l} & \sin\theta_{k,l} \\ -\sin\theta_{k,l} & \cos\theta_{k,l} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}^{(k,l)} \\ \boldsymbol{v}^{(l,k)} \end{bmatrix} \tag{1.11}$$

dove $0° \leq \theta_{k,l} \leq 90°$.

In questo modo si potrebbe ruotare ciascun autospazio con una angolo differente, scegliendo per ogni coppia di autovettori quello che comprime maggiormente il segnale, ma risulterebbe troppo oneroso, per cui si ruotano tutti gli autospazi della stessa quantità. La scelta di quest'unico angolo è fatta ottimizzando una funzione obiettivo, che misura ad esempio la sparsità dei coefficienti trasformati, tramite ricerca esaustiva.

1.3.4 Mode Directional Dependent Transform (MDDT)

È una tecnica legata alle trasformate con modi di predizione. Invece di usare un'unica trasformata per tutti i residui, la MDDT fornisce diverse trasformate a seconda dei modi di predizione [5]. Le trasformate sono apprese da un training set iniziale di dati, sulla base della KLT e possono essere implementate in modo separabile e non. Nel caso separabile si assume che gli elementi delle colonne e delle righe siano indipendenti e identicamente distribuiti (i.i.d.). Nel caso separabile, che è quello più pratico, la formula è la seguente:

$$Y = CXR \tag{1.12}$$

dove C e R sono le matrici della trasformazione delle colonne e delle righe rispettivamente. Essendo matrici ricavate da un training set di dati iniziale con la Singular Values Decomposition (SVD), le funzioni di base sono numeri float, successivamente arrotondati ad interi. Nello schema della MDDT, ogni modalità di intra-prediction ha una trasformata propria. La modalità migliore è selezionata in base al valore della funzione di rate-distortion.

1.4 Codificatore Entropico

Serve per realizzare l'ultimo passo della codifica, ossia, una volta che il segnale è stato rappresentato in un dominio più adatto e poi quantizzato, occorre che venga rappresentato in una sequenza di bit, ricordando quanto è già stato detto precedentemente in 1.1.3 (per capire meglio le varie fasi vedere gli schemi 1.5 e 1.6).

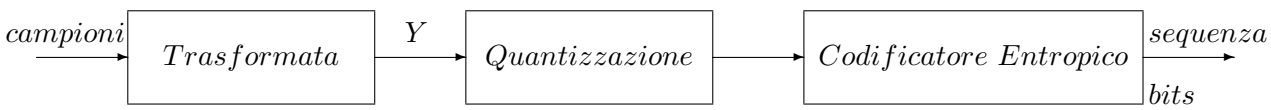

Figura 1.5: La catena di codifica del segnale.

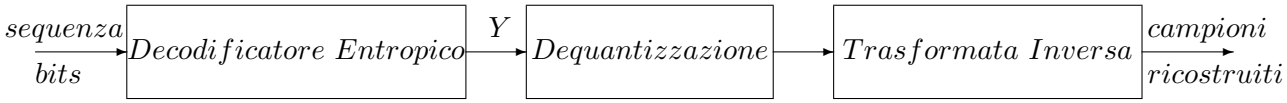

Figura 1.6: La catena di decodifica del segnale.

Riassumendo, dato un alfabeto $X = \{a_i\}_{i=0}^{N-1}$ di N simboli con probabilità di occorrenza $p(a_i) = p_i \ \forall i$, si vuole definire una rappresentazione compatta in bit di tali simboli, ossia le codeword, indicate con $c(a_i) \ \forall i$. Il criterio con cui farlo è tenere bassa la quantità $\sum_i l_i p_i$, indicata con $\bar{l}$, dove l_i è la lunghezza di a_i. Per cui i simboli più probabili avranno codeword corte contrariamente per quelli più rari, garantendone univocamente la decodifica tramite la regola del prefisso. Di seguito sono riportati i tipi di codificatori entropici utili al lavoro svolto.

1.4.1 Codifica Huffman

Si basa sull'idea di codifica a prefisso libero, cioè nessuna codeword può essere la parte iniziale di un'altra codeword. Il codice si rappresenta con un albero binario in cui solamente le foglie sono le codeword, un esempio è raffigurato nell'immagine 1.7. Ai nodi con profondità minore sono associati simboli con una probabilità più alta, poiché la lunghezza della codeword è più piccola, mentre ai nodi con profondità maggiore quelli più rari, con lunghezza più grande.

	cod. 1	cod. 2
a	0	0
b	01	10
c	11	11

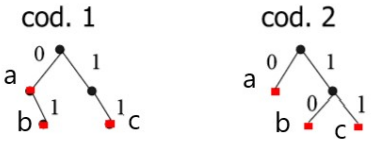

Figura 1.7: A sinistra la tabella delle codeword per due diverse codifiche. A destra i rispettivi alberi di codifica. Le codeword sono assegnate dalle foglie in sù. Si noti che cod.1 non è a prefisso libero mentre cod.2 lo è.

Data la massima probabilità p_{max} tra tutti i simboli, per il codificatore di Huffman vale la seguente relazione:

$$\bar{l} < \begin{cases} H(X) + p_{max}, \ se \ p_{max} \geq 0.5 \\ H(X) + p_{max} + 0.086, \ se \ p_{max} < 0.5 \end{cases} \tag{1.13}$$

dove la prima disuguaglianza si ha nel caso di una grossa subottimalità, invece la seconda è il caso da preferire e si verifica tendenzialmente quando il numero di simboli è elevato. Se $N = 2$, allora nella 1.13 ci si ritrova nel primo caso, dove la subottimalità non è buona. Per questa ragione si estende la tecnica considerando combinazioni congiunte di singoli simboli, in modo da far crescere di fatto la numerosità di simboli e ottenere una buona subottimalità. Lo svantaggio è che crescendo la lunghezza dei blocchi di simboli considerati aumenta notevolmente la complessità e se le statistiche di X dovessero cambiare bisognerebbe ricalcolare tutte le codeword.

1.4.2 Codifica Aritmetica

Si basa sulla corrispondenza biunivoca tra la sequenza corrente x e il valore corrispondente della funzione di distribuzione cumulativa $F_X(i) = \sum_{k=0}^{i-1} P(X = x_k)$. Ad esempio, sia $X = \{0, 1\}$ una

sorgente i.i.d., con $P(X = 0) = p_0$ e $P(X = 1) = 1 - p_0$. Per codificare un blocco di simboli di lunghezza m non è necessario calcolare tutte le 2^m possibili codeword, ma solo quelle relative ad una specifica realizzazione in input, si veda 1.8. L'intervallo [0,1] viene partizionato seguendo i valori delle probabilità dei simboli, iterativamente per m volte. Per determinare la codeword di x_i si sceglie una qualunque rappresentazione binaria di un numero $c(x_i) \in I$, ad esempio il punto medio di I.

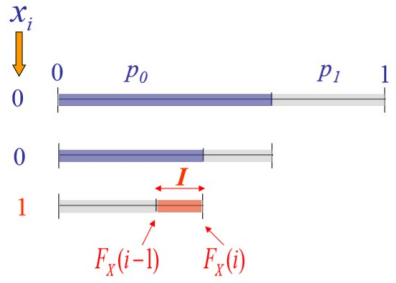

Figura 1.8: Processo iterativo di codifica per $m = 3$.

Quello che si cerca di fare è scegliere il numero con la rappresentazione binaria più corta per ottenere la codeword più corta. Una volta che si è trovata la codeword $c(x_i)$, per decodificare in modo univoco è necessario conoscere il numero m di iterazioni e il simbolo di fine file.

In questo caso vale la seguente relazione:

$$H(X) \leq \bar{l} \leq H(X) + \frac{2}{m} \tag{1.14}$$

dove $H(X)$ è l'entropia della sorgente, $\bar{l}$ la media della lunghezza delle codeword e m il numero di simboli considerati. Al crescere di m migliora l'efficienza della codifica ma aumenta anche linearmente la complessità. Con questa tecnica risulta facile adattare le statistiche del modello alle caratteristiche della sorgente, ad ogni iterazione.

1.5 Reti Neurali

Le reti neurali o NNs (dall'inglese Neural Netwoks), sono architetture di programmazione ispirate al funzionamento del sistema nervoso, che permettono ad un computer di apprendere da osservazioni di dati. Se impiegate come tecniche di Deep Learning, permettono di raggiungere notevoli soluzioni per problemi di classificazione legati al riconoscimento delle immagini, riconoscimento del parlato e l'elaborazione del linguaggio naturale, in cui sono coinvolte grandi quantità di dati da elaborare. A differenza di altri algoritmi di Machine Learning, le NNs sono in grado di apprendere il modello e le regole di decisione. Tuttavia, non vi è una teoria statistica o matematica che sia in grado di spiegare gli output.

Matematicamente una rete neurale implementa una complessa funzione multidimensionale non lineare, la cui unità base computazionale è il *perceptron*, che si vede nell'immagine 1.9. Questa unità prende una decisione in base al valore output della media pesata dei fattori in input nella regola in 1.15, quando l'output è 1 l'unità si dice attiva. Una definizione più formale della 1.15 è la 1.16, dove la media pesata è sostituita dal prodotto scalare tra il vettore x in input, il vettore w dei pesi e la soglia *threshold* dal vettore di bias b spostato a sinistra della disequazione. Il numero di input in ingresso può essere anche molto grande e i pesi w_i dipendono da questi input.

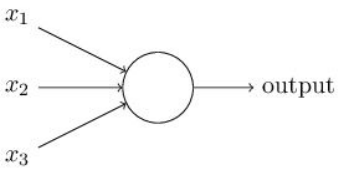

Figura 1.9: Unità computazionale di base di una NN. $x_i \in \boldsymbol{x}$ dove $\boldsymbol{x}$ è un vettore a valori reali e w_i sono i pesi.

$$ouput = \begin{cases} 0, \; se \; \sum_i w_i x_i \leq threshold \\ 1, \; se \; \sum_i w_i x_i > threshold \end{cases} \tag{1.15}$$

$$ouput = \begin{cases} 0, \; se \; \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} \leq 0 \\ 1, \; se \; \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} > 0 \end{cases} \tag{1.16}$$

Si possono realizzare decisioni più complesse, come nell'immagine 1.10 usando:

- un layer in input, per rapprensentare i dati iniziali
- diversi hidden layer, per elaborare i dati in input
- un layer di output, per dare la risposta.

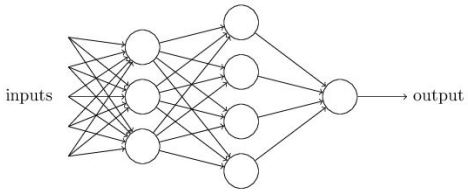

Figura 1.10: Rete Neurale di tipo Feedforward Fully Connected: non sono presenti loop tra layer e ogni nodo è collegato a tutti i nodi del layer successivo.

Per l'apprendimento bisogna avere a disposizione un training set, $\{\boldsymbol{x_i}, y_i\}$, piuttosto numeroso, in cui $\boldsymbol{x_i}$ rappresenta il dato i-esimo e y_i il label associato al dato, la rete poi mappa ciascun dato con un label. La validazione serve per stabilire quanto correttamente vengono mappati i dati. Perciò sarebbe desiderabile che un piccolo cambiamento nei pesi generasse un altrettanto piccolo cambiamento nell'output, cosa che non avviene nel caso 1.16. Per risolvere questo problema si può usare come funzione di attivazione la funzione sigmoide, $\sigma(z) = \frac{1}{1+\exp^{-z}}$, con $z = w \cdot x + b$, che è continua e differenziabile. Nella pratica questa è la più usata ma ne vengono usate anche altre, ad esempio la tangente iperbolica o la ReLU (Rectified Linear Unit).

Nei paragrafi seguenti si espongono i metodi che vengono usati per allenare una rete neurale e la tipologia di reti neurali convoluzionali. Per approfondire questi aspetti consultare [15].

1.5.1 Gradient Descendent (GD)

Per allenare la rete bisogna trovare i valori ottimi di w e b, in modo che l'output approssimi la classificazione vera $y(x)$ tramite l'ottimizzazione di una funzione di costo. Si possono usare diverse funzioni di costo ad esempio la funzione Cross-entropy o quella quadratica 1.17. Nella teoria la più usata è l'ultima delle due perché è più intuitiva.

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a(x,w,b)||^2 \tag{1.17}$$

dove a indica la risposta all'input x.

Per minimizzare la 1.17 un metodo usato è il gradiente discendente o GD. Questo metodo iterativo nel caso di funzioni convesse garantisce di trovare come soluzione il minimo globale, mentre per funzioni non convesse si rischia di avere come soluzione un minimo locale, che potrebbe non essere una buona soluzione. L'idea che si applica è quella di cambiare il valore della variabile v della funzione $C(v)$ in modo che questa diminuisca. Le direzioni di variazione sono data dal gradiente della funzione. Per cui se $\Delta C \approxeq \nabla C \cdot \Delta v$ scegliendo $\Delta v = -\eta \nabla C$ la variazione della funzione è data da:

$$\Delta C = -\eta ||\nabla C||^2 \tag{1.18}$$

dove η prende il nome di *learning rate* e deve essere abbastanza piccolo affinché valga l'espansione di Taylor. L'aggiornamento delle variabili per una rete neurale è il seguente:

$$\begin{cases} w_k \leftarrow w_k - \eta \dfrac{\partial C}{\partial w_k} \\ b_l \leftarrow b_l - \eta \dfrac{\partial C}{\partial b_l} \end{cases} \tag{1.19}$$

La stima del gradiente di C su tutto un training set può essere molto lenta, per cui si può usare una variante del GD che è lo stochastic-GD o SGD. Si divide il training set in $m << n$ sottoinsiemi casuali di campioni,$X_1, X_2, \ldots, X_m$, ad ogni iterazione si usa solo uno di questi insiemi per stimare il gradiente di C. Questa approssimazione introduce errore che serve per velocizzare le tempistiche di apprendimento e per riuscire a non rimanere bloccati in minimi locali.

1.5.2 Algoritmo Back Propagation (BP)

È un algoritmo iterativo che usando l'espressione delle derivate parziali della funzione C rispetto a w e b, propaga i valori aggiornati con il GD dei pesi e dei bias su tutta la rete. Prima di procedere si consideri la seguente notazione, utile per il seguito, illustrata nelle immagini 1.11 e 1.12:

- $w^l_{j,k} \in \boldsymbol{W^l}$: il peso dal k-esimo neurone nel $(l-1)$-esimo layer al j-esimo neurone nel j-esimo layer
- $b^l_j \in \boldsymbol{b^l}$: bias del j-esimo neurone nel l-esimo layer
- $a^l_j \in \boldsymbol{a^l}$: attivazione del j-esimo neurone nel l-esimo layer,
  dove $\sigma(z)$ è una generica funzione di attivazione e $\boldsymbol{a^l} = \sigma(\boldsymbol{W^l a_{l-1}} + \boldsymbol{b^l}) = \sigma(\boldsymbol{z^l})$ il vettore di attivazione del layer.

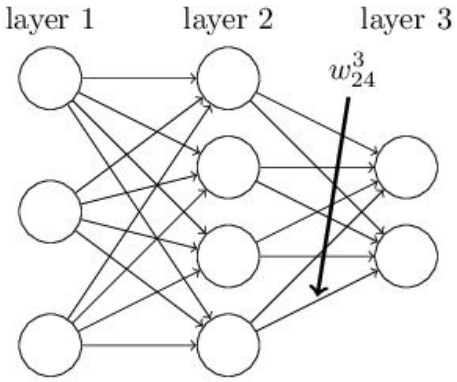

Figura 1.11

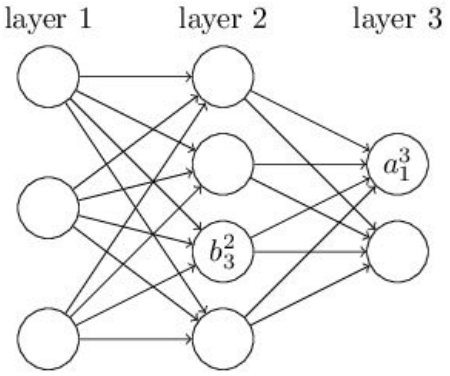

Figura 1.12

Se $z^l_j = \sum_k w^l_{j,k} a^{l-1}_k + b^l_j$, è possibile definire l'errore del neurone j-esimo nel l-esimo layer come:

$$\delta j^l = \frac{\partial C}{\partial z^l_j} \tag{1.20}$$

con il rispettivo vettore $\boldsymbol{\delta^l}$.

Il punto chiave dell'algoritmo è nel calcolare l'errore per ogni layer e le derivate parziali di C rispetto a w e b. È possibile ricavare le quattro formule per queste quantità, riportate in 1.21, 1.22, 1.23 e 1.24, in modo che il calcolo non risulti oneroso ma che coinvolga semplici somme e moltiplicazioni di variabili note.

$$\boldsymbol{\delta^L} = \nabla_a C \odot \sigma'(\boldsymbol{z^L}) \tag{1.21}$$

dove L indica l'ultimo layer, cioè quello di output, mentre $\odot$ il prodotto vettoriale componente per componente.

L'equazione successiva in 1.22 si ricava dalla 1.21 con la chain rule:

$$\boldsymbol{\delta^l} = [(\boldsymbol{W^{l+1}})^T \boldsymbol{\delta^{l+1}}] \odot \sigma'(\boldsymbol{z^l}), \tag{1.22}$$

Le ultime due equazioni esprimono il tasso con cui varia la funzione di costo C:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{1.23}$$

$$\frac{\partial C}{\partial w_{j,k}^l} = a_k^{l-1}\delta_j^l. \tag{1.24}$$

A questo punto è possibile descrivere i vari passaggi dell'algoritmo:

1. Dato l'input x e i valori iniziali per $\boldsymbol{W^l}$ e $\boldsymbol{b^l}$ $\rightarrow$ si calcola $\boldsymbol{a^1}$
2. Fase Feedforward $\rightarrow$ $\forall l = 2,3,\ldots,L$ si calcolano $\boldsymbol{z^l} = \boldsymbol{W^l a_{l-1}} + \boldsymbol{b^l}$ e $\boldsymbol{a^l} = \sigma(\boldsymbol{z^l})$
3. Errore in output $\rightarrow$ si calcola $\boldsymbol{\delta^L} = \nabla_a C \odot \sigma'(\boldsymbol{z^L})$
4. Fase Backpropagation $\rightarrow$ $\forall l = L-1, L-2, \ldots, 1$ si calcola $\boldsymbol{\delta^l} = [(\boldsymbol{W^{l+1}})^T \boldsymbol{\delta^{l+1}}] \odot \sigma'(\boldsymbol{z^l})$
5. Output $\rightarrow$ $\forall l, j, k$ si calcola $\dfrac{\partial C}{\partial w_{j,k}^l} = a_k^{l-1}\delta_j^l$ e $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$ $\Rightarrow$G.D. $\begin{cases} w_{j,k}^l \leftarrow w_{j,k}^l - \eta \dfrac{\partial C}{w_{j,k}^l} \\ b_j^l \leftarrow b_j^l - \eta \dfrac{\partial C}{b_j^l} \end{cases}$

Una rete può avere molti layer interni, da cui prende il nome di deep NN, un esempio in figura 1.13. Questo implica che la fase di training potrebbe richiedere delle tempistiche molto elevate, a ciò si possono aggiungere anche altri due problemi. Il primo è l'overfitting, che si può raggirare con tecniche di early stopping, regolarizzazione o drop out. Il secondo, detto vanishing gradient, riguarda il fatto che i primi layer tendono ad apprendere molto più lentamente degli ultimi, perché il gradiente della funzione costo C diminuisce sempre di più nella fase di Backpropagation. Si può risolvere il problema inserendo alla fine di ogni layer un batch normalization layer.

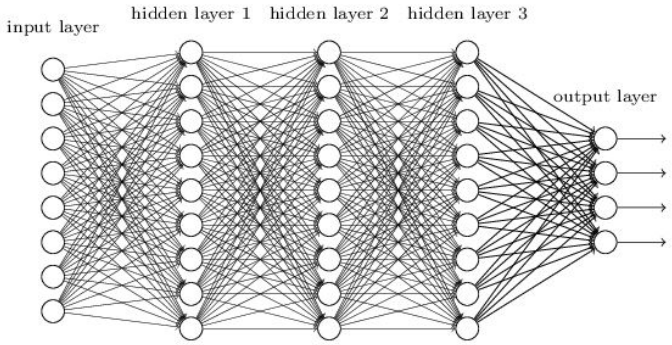

Figura 1.13

1.5.3 Reti Neurali Convoluzionali (CNN)

Le CNNs sono reti neurali che per via della loro architettura vengono molto usate nella classificazione d'immagini, ad esempio per riconoscere numeri scritti a mano. A differenza delle NNs Feed Forward, in cui ogni input è collegato a ogni neurone del layer interno, qui si collegano al neurone interno solo piccole regioni dell'immagine, ad esempio 5×5, quindi a un numero limitato di input. Questa regione nell'immagine in input è chiamata *local receptive field* per il neurone interno, come è mostrato nella figura 1.14. Il *local receptive field* si fa scorrere, orizzontalmente

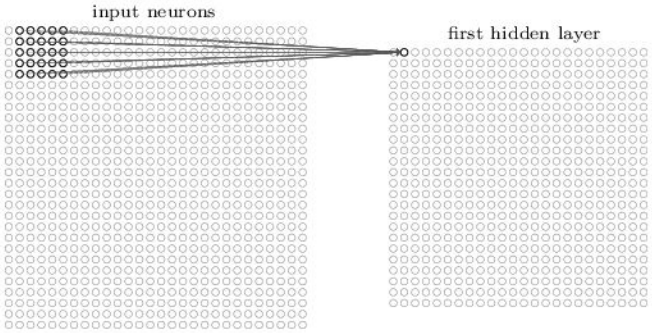

Figura 1.14

e verticalmente, lungo tutta l'immagine, la quantità di spostamento è chiamata *stride length*, e a ogni spostamento corrisponde un neurone nel layer interno.

Le connessioni dal *local receptive field* al neurone corrispondente hanno tutte gli stessi pesi e bias, nel caso 5×5, la funzione di attivazione per ogni neurone nella posizione (j, k) è data dalla formula 1.25.

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right) \tag{1.25}$$

Con questa architettura si individuano esattamente le stesse caratteristiche in diverse posizioni all'interno dell'immagine, per sfruttare eventuali contenuti invariati che vengono traslati. La condivisione degli stessi pesi e bias del layer definisce un filtro. In generale un layer completo di una CNN è costituito da diversi filtri, stesse dimensioni del *local receptive field* ma diversi pesi e bias, il cui numero ne definisce la profondità. Per snellire la quantità d'informazione in uscita dal layer convoluzionale, visto prima, si applica un pooling layer che opera una sorta di downsampling, indipendentemente per ogni filtro, l'immagine 1.15 è un esempio. Il più noto è il max-pooling layer, ogni unità di pooling ha come valore output il massimo dell'attivazione di una piccola regione (es. 2×2) del layer convoluzionale.

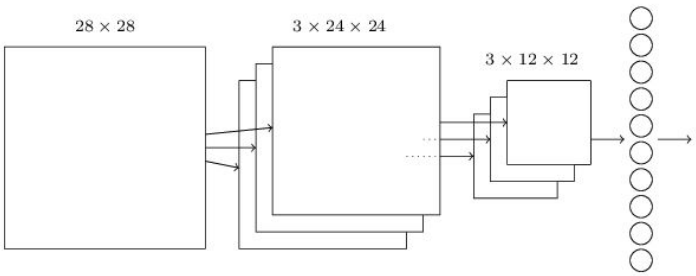

Figura 1.15: Esempio di una CNN con un layer convoluzionale di profondità 3, 3 pooling layers per estrarre le caratteristiche dell'immagine. Per ultimo un fully connected layer per avere in output la classificazione.

A differenza delle classiche Deep NNs con molti hidden layer e parametri, che rendono più ostica la fase di training, le CNNs per la loro caratteristica di condivisione dei pesi hanno meno parametri, quindi il problema dell'overfitting in questo caso è minore. La fase di training di una CNN si può fare con l'algoritmo BP, con alcuni accorgimenti per i layer convoluzionali e di pooling, di solito facendo un pre-training dei primi layer.

1.6 Struttura della tesi

Dopo questa introduzione su i richiami teorici, la struttura dell'elaborato è organizzata nel modo seguente:

- Nel capitolo 2 si presenta l'analisi di un sistema per la compressione basato su strati di filtri convoluzionali, la sua fase di ottimizzazione, la sua architettura e il codice di implementazione.
- Nel capitolo 3 si confrontano le trasformate ottimizzate con la DCT variando la dimensione del blocco della trasformata e il passo di quantizzazione.
- Nel capitolo 4 si discutono i risultati ottenuti.

Capitolo 2

Progettazione e sviluppo

Questo capitolo riguarda lo studio del sistema di compressione immagini in [11], costituito da una trasformata di codifica non lineare, un quantizzatore uniforme e una trasformata di decodifica non lineare. L'architettura usata è una CNN, con una particolare funzione di attivazione, la GDN (Generalized Divisive Normalization), che risulta essere efficace nella modellizzazione della densità dell'immagine ([9]). Il sistema è ottimizzato congiuntamente per il rate-distortion, usando una variante del SGD (Stochastic Gradient Descendent). La funzione di costo che si ricava dal quantizzatore è discontinua per cui si deve riformulare l'ottimizzazione con un rilassamento continuo.

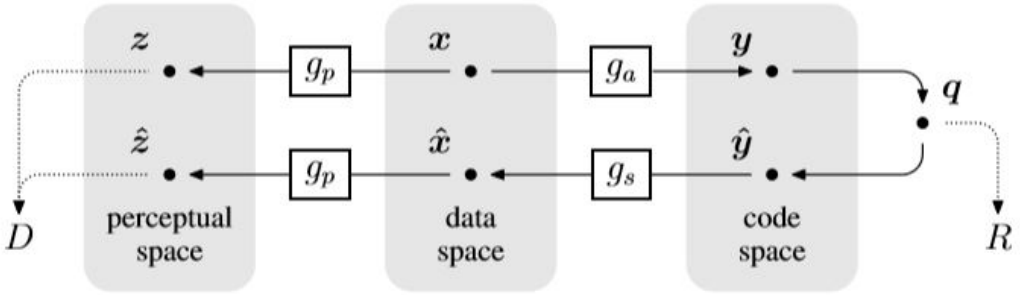

Figura 2.1: Schema generale del sistema.

Nell'articolo [10] era già stato presentato lo schema di un sistema end-to-end per la compressione di immagini, basato sull'ottimizzazione di generiche trasformate non lineari e combinate con una qualunque metrica percettiva differenziabile. In tale articolo si dimostra che se si ottimizza il sistema per una metrica di distorsione si ottengono risultati migliori che con l'ottimizzazione tramite l'MSE, poiché quest'ultimo è noto non rappresentare adeguatamente la percezione visiva dell'occhio umano. Nell'articolo [11] però si usa come misura di distorsione l'errore quadratico concentrandosi sulla trasformata. Lo schema nella figura 2.1 può essere descritto partendo da un vettore immagine, $\boldsymbol{x} \in \mathbb{R}^N$, successivamente mappato nello spazio latente di codifica, tramite una trasformata di analisi parametrica multivariata, $\boldsymbol{y} = g_a(\boldsymbol{x}; \boldsymbol{\phi})$. Tale rappresentazione è quantizzata, ottenendo un vettore di valori discreti, $\boldsymbol{q} \in \mathbb{Z}^M$, che poi viene compresso in maniera entropica. Il rate, R, del codificatore discreto è limitato inferiormente dall'entropia della distribuzione di probabilità discreta del vettore, $H(P_{\boldsymbol{q}})$. Nel processo di ricostruzione invece, si considerano i valori discreti di q che sono reinterpretati come i valori continui del vettore $\hat{\boldsymbol{y}}$, che poi attraverso una trasformata parametrica di sintesi è riportato nello spazio dei dati, $\hat{\boldsymbol{x}} = g_s(\hat{\boldsymbol{y}}; \boldsymbol{\theta})$. La distorsione è calcolata con la norma $D(\boldsymbol{x}, \hat{\boldsymbol{x}})$, in questo caso non è necessario

una trasformata per la valutazione nello spazio percettivo, perché si è detto che si usa l'errore quadratico. Perciò con questa assunzione la funzione $g_p(\cdot)$ è la funzione identità e $\boldsymbol{z} = \boldsymbol{x}$ come $\hat{\boldsymbol{z}} = \hat{\boldsymbol{x}}$. I vettori dei parametri, $\boldsymbol{\phi}$ e $\boldsymbol{\theta}$ vengono ottimizzati per una somma pesata del rate e della distorsione su un insieme di immagini, si veda la figura 2.1.

Il problema di ottimizzazione del rate-distortion rilassato ricorda quello relativo all'apprendimento di modelli generativi per immagini ([13]), ma differisce per i vincoli applicati, qui sono imposti in modo che si approssimi il problema discreto per ogni valore trade-off del rate-distortion.

2.1 Ottimizzazione

L'obiettivo è minimizzare la somma pesata del rate e della distorsione, $R + \lambda D$, rispetto alla trasformata di analisi, alla trasformata di sintesi e al codificatore entropico, dove λ è il parametro che regola il trade-off (figura 2.2). La funzione obiettivo è definita direttamente in termini di entropia:

$$L[g_a, g_s] = -\mathbb{E}[\log_2 P_{\boldsymbol{q}}] + \lambda \mathbb{E}||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2 \tag{2.1}$$

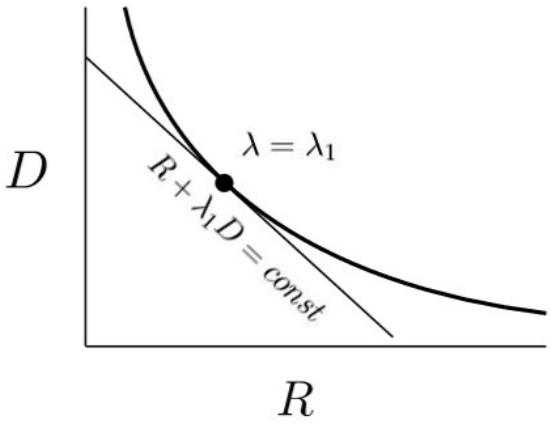

Figura 2.2: Rate-distortion trade-off.

Entrambe le attese in 2.1 sono approssimate da medie su un training set di immagini. Se si ipotizza di applicare un quantizzatore uniforme fisso con grandezza di bin pari a 1 e i valori rappresentati sono al centro del bin si ottiene:

$$\hat{\boldsymbol{y}}_i = \boldsymbol{q}_i = \text{round}(\boldsymbol{y}_\text{i}) \tag{2.2}$$

dove $\boldsymbol{y}_i$ indica l'elemento della rappresentazione latente, $\hat{\boldsymbol{y}}_i$ l'elemento quantizzato. A questo punto la densità marginale di $\hat{\boldsymbol{y}}_i$ si ricava come un treno di masse di probabilità discrete (funzioni delta di Dirac, figura 2.3), con pesi pari al valore della funzione di massa di probabilità(2.3) di $\boldsymbol{q}_i$:

$$P_{q_i}(n) = \int_{n-1/2}^{n+1/2} p_{y_i}(t)dt, \forall n \in \mathbb{Z}. \tag{2.3}$$

Entrambi i termini in 2.1 dipendono dai valori quantizzati, mentre la derivata della funzione 2.2 è nulla quasi ovunque, rendendo di fatto il GD (Gradient Descendent) inapplicabile. Per riuscire a ottimizzare con il GD si sostituisce il quantizzatore con una sorgente di rumore additivo

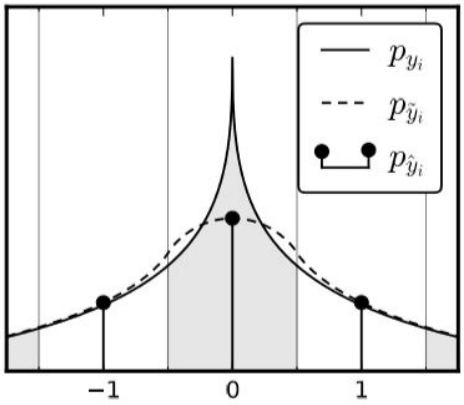

Figura 2.3: Relazione in una dimensione delle diverse densità.

uniforme i. i. d., $\boldsymbol{\Delta y}$, avente la stessa ampiezza del bin di quantizzazione, ossia 1. In questo modo si definisce $\tilde{\boldsymbol{y}}_i$ come l'elemento perturbato con rumore uniforme e si ottengono due proprietà desiderabili.

- La funzione di densità di $\tilde{\boldsymbol{y}}$ è un rilassamento continuo della funzione di massa di probabilità di $\boldsymbol{q}$:

$$p_{\tilde{\boldsymbol{y}}_i}(\boldsymbol{n}) = P_{\boldsymbol{q}}(\boldsymbol{n}), \forall \boldsymbol{n} \in \mathbb{Z}^M, \tag{2.4}$$

  perciò l'entropia differenziale di $\tilde{\boldsymbol{y}} = \boldsymbol{y} + \boldsymbol{\Delta y}$ può essere usata come un'approssimazione dell'entropia discreta di $\boldsymbol{q}$.

- Il rumore uniforme e indipendente approssima l'errore di quantizzazione per quanto riguarda i suoi momenti marginali.([7]). Questo stesso approccio si applica alla misura di distorsione.

È stato mostrato empiricamente, in [11], che l'entropia differenziale di $\tilde{\boldsymbol{y}}$, ottenuta dall'ottimizzazione sopra, stima l'entropia discreta di $\boldsymbol{q}$ (figura 2.4).

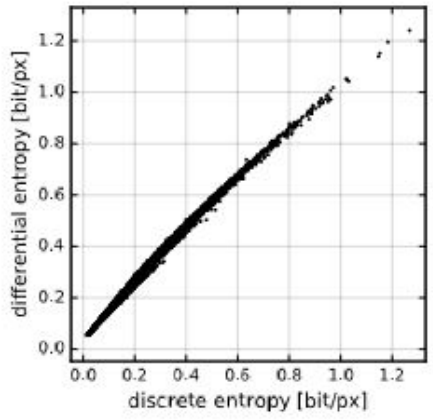

Figura 2.4: $H[P_{q_i}]$ vs. $h[p_{\tilde{y}_i}]$, sommati su tutti gli i.

La scelta del modello dell'errore può influire sulla stima dell'approssimazione, perciò è stato usato un modello frequentista e non distorto di stima di $p_{\tilde{y}}$ invece di quello Bayesiano. Inoltre, si è assunto indipendenza nelle marginali, nello spazio di codifica, sia per il modello di probabilità rilassato sia per il codificatore entropico. Viene usato un modello non parametrico per le marginali $p_{\tilde{y}_i}$ costituito da campioni di funzioni lineari a tratti, (es. una spline lineare) usando 10 punti di campionamento per intervallo. Quindi se $p_{\tilde{y}_i} = p_{y_i} * U(0,1)$ è smussata da un filtro, l'errore del modello può essere reso arbitrariamente piccolo riducendo l'intervallo di campionamento. Il vettore dei parametri $\boldsymbol{\psi}^{(i)}$ rappresenta il valore di $p_{\tilde{y}_i}$ nel punto i-esimo di campionamento.

La funzione di costo finale per ottimizzare $\boldsymbol{\theta}$ e $\boldsymbol{\phi}$ si scrive nel modo seguente:

$$\begin{aligned} L(\boldsymbol{\theta}, \boldsymbol{\phi}) = \; & -\mathbb{E}_{\boldsymbol{x},\boldsymbol{\Delta y}} \left[\sum_i \log_2 p_{\tilde{y}_i}(g_a(\boldsymbol{x};\boldsymbol{\phi})\boldsymbol{\Delta y}; \boldsymbol{\psi}^{(i)}) \right] \\ & + \lambda \, \mathbb{E}_{\boldsymbol{x},\boldsymbol{\Delta y}} \left\| g_p(g_s(g_a(\boldsymbol{x};\boldsymbol{\phi}) + \boldsymbol{\Delta y};\boldsymbol{\theta})) - g_p(\boldsymbol{x}) \right\|^2 \end{aligned} \tag{2.5}$$

Questa funzione è continua e differenziabile rispetto ai parametri, quindi è adatta per applicare il metodo di SGD.

È stato verificato empiricamente ([11]) che la funzione di costo rilassata 2.5 fornisce una buona approssimazione dei valori effettivi del rate-distortion. La verifica è avvenuta confrontando i valori della funzione obiettivo con i valori della sua versione rilassata, si veda l'immagine 2.5.

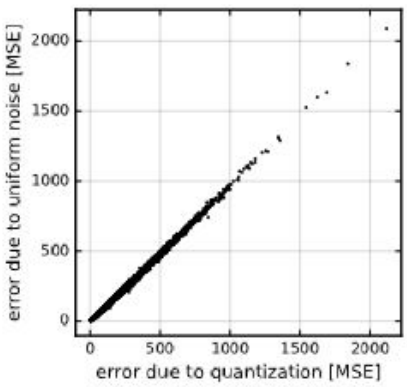

Figura 2.5: Valutazione della distorsione, per $\hat{y}$ vs. $\tilde{y}$, sommati su tutti gli i.

2.2 Architettura del sistema di trasformate

Come è stato detto in precedenza, le trasformate di analisi e sintesi del sistema sono realizzate con tre stage di convoluzione, sottocampionamento e GDN(figura 2.6). Dove:

- *conv.*: layer convoluzionale, è indicata la dimensione del filtro e il numero di features in input/output.
- *down-/upsample*: sotto-/sovracampionamento implementato congiuntamente con la convoluzione adiacente.
- *add*: per ogni feature viene sommato un bias per ogni elemento.
- *GDN/IGDN*: viene applicato il generalized divisive normalization e la sua inversa approssimata.

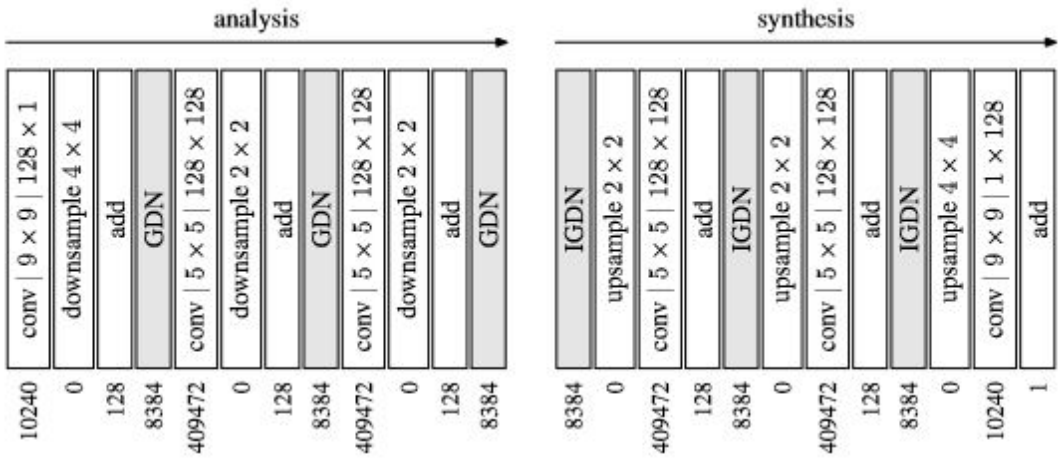

Figura 2.6: Architettura della trasformata di analisi, g_a, e di sintesi,g_s, per immagini in scala di grigi. In basso sono indicati il numero di parametri per ogni fase.

Il modello completo è ottimizzato su un sottoinsieme di 6507 immagini preso dal database ImageNet([3]). Sono stati applicati una serie di fasi di pre-elaborazione per ridurre artefatti e altre indesiderate contaminazioni dalle immagini. Per prima cosa sono state eliminate le immagini con un alto livello di saturazione, poi è stato aggiunto del rumore uniforme corrispondente alla quantizzazione del valore del pixel alle restanti immagini. Infine le immagini sono state sottocampionate e tagliate in modo che dimensione finale fosse di 256×256 pixel, in cui la quantità di sotto campionamento e selezione della parte da tagliare è random ma dipendente dalla dimensione dell'immagine iniziale. Per ridurre il rumore alle alte frequenze e artefatti di compressione, il fattore di resampling non è mai più grande di 0.75. Le immagini non abbastanza grandi sono state scartate a priori.

Per ottenere un'ottimizzazione sufficientemente veloce e stabile le tecniche usate sono le seguenti:

- l'algoritmo di ottimizzazione di Adam ([13]) per ottenere i valori dei parametri $\boldsymbol{\phi}$ e $\boldsymbol{\theta}$, iniziando con $\alpha = 10^{-4}$, e successivamente diminuendolo di un fattore 10 ogni volta che si verificava un ristagno nel miglioramento sia del rate sia della distorsione, fino a $\alpha = 10^{-7}$;

- parametrizzazione dei filtri lineari usando i coefficienti della loro trasformata DCT, perché migliorava la velocità di convergenza rispetto alla DFT;

- parametrizzazione del parametro $\boldsymbol{\beta}$ nel GDN/IGDN,

$$\beta_i = (\beta_i^{'})^2 - 2^{-10}. \tag{2.6}$$

  Questo assicura che il gradiente sia più piccolo quando il parametro è vicino a 0, situazione in cui l'ottimizzazione sarebbe instabile. Ciascun β_i è mappato univocamente nell'intervallo $[2^{-5}, \infty)$ ad ogni passo del gradiente. La stessa procedura è usata per gli elementi di $\boldsymbol{\gamma}$, in più si fa la media di $\boldsymbol{\gamma}^{'}$ con la sua trasposta in modo che il risultato sia simmetrico ad ogni step;

- rimozione dell'ambiguità tra ogni trasformazione lineare e la sua successiva non linearità (o precedente per la trasformata di sintesi), rinormalizzando i filtri lineari dopo ogni passo del gradiente, dividendo ciascun filtro per la radice quadrata della somma dei suoi coefficienti

al quadrato. Nell'analisi la somma è sullo spazio delle features e sui loro input, mentre nella sintesi sullo spazio delle features e sui loro output.

2.2.1 General Divisive Normalization e trasformate

La maggior parte dei sistemi di compressione si basa su trasformate lineari ortogonali, scelte in modo da ridurre la correlazione nei dati, allo scopo di semplificare l'elaborazione nel codificatore entropico. Ma le statistiche congiunte della risposta di questi filtri lineari mostra ad alti ordini forti dipendenze. Questa dipendenza può essere significativamente ridotta attraverso l'applicazione di operatori di controllo del gain, congiunti e non lineari([17], [14] e [18]), ispirati ai modelli dei neuroni visivi([8] e [2]). Si può mostra che l'uso del GDN risulta essere efficiente nel mappare in una distribuzione gaussiana le distribuzioni statistiche congiunte di immagini naturali, molto di più che una cascata di trasformazioni lineari seguite da una non-linearità marginale([11] e [9]). Qui la trasformata di analisi utilizzata, g_a, consiste in tre passi di convoluzione, sottocampionamento e GDN. Quest'ultimo è definito nel modo seguente:

$$v_i = \frac{u_i}{\left(\beta_i + \sum_j \gamma_{ij} u_j^2\right)^{1/2}}, \tag{2.7}$$

dove i vettori $\boldsymbol{u}$ e $\boldsymbol{v}$ sono rispettivamente le attivazioni lineari e normalizzate in un punto preciso dello spazio delle features. Il vettore $\boldsymbol{\beta}$ e la matrice simmetrica $\boldsymbol{\gamma}$ sono i parametri da ottimizzare. La trasformata di sintesi, g_s, è realizzata con l'inversa approssimata, applicando un principio conosciuto dall'algoritmo LISTA. La trasformata di sintesi consiste in tre passi di IGDN (Inverse Generalized Divisive Normalization), sovracampionamento e convoluzione:

$$v_i = u_i \left(\beta_i + \sum_j \gamma_{ij} u_j^2\right)^{1/2}, \tag{2.8}$$

dove i parametri sono gli stessi della 2.7, ma ottimizzati separatamente.

La struttura convoluzionale delle trasformate è realizzata con 128 filtri nel primo stage, ciascuno sottocampionato con un fattore 4 verticalmente e orizzontalmente. Negli altri due stage invece si sono usati 128 filtri da applicare ad ogni fetures di input dello stage precedente, e sottocampionati con un fattore 2 in ciascuna delle direzioni. Come è stato anticipato precedentemente, la trasformata percettiva, g_p, in questo caso è l'identità e come metrica si usa l'errore quadratico.

2.3 Codice della catena di elaborazione

Il codice per implementare questo sistema è disponibile sia per Matlab sia per Python al link [4].

Per il liguaggio Python non sono disponibili i parametri ma è disponibile il codice per effettuare il training.

Per il linguaggio Matlab oltre al codice dell'architettura, per immagini in bianco e nero, sono disponibili anche i parametri per immagini in scala di grigi e per immagini a colori, ma non l'architettura per le immagini a colori o per rieseguire la fase di training e neppure i parametri per alti bit-rate. Per la parte a colori è stato necessario scrivere il codice partendo da quello a disposizione. Differisce dal codice in Python nella gestione dei bordi. Inoltre non è stato implementato il codificatore entropico aritmetico, tuttavia la stima entropica del bit-rate e i valori del PSNR nelle immagini decodificate sono molto simili a quelli riportati nell'articolo [11] e nel link [4].

Per usare il codice Matlab è necessario scaricare i parametri pre-trained del modello e il pacchetto *matlabPyrTools*, sottolineando il fatto che l'implementazione disponibile è adatta solo per immagini in scala di grigi, ma i parametri per l'implementazione per le immagini a colori sono anch'essi forniti. Ciascuna sottocartella corrisponde ad un valore del parametro di trade-off, λ, che specifica un punto lungo la curva della funzione rate-distortion. In particolare più il valore di λ è alto più è alta la qualità dell'immagine che si sta ricostruendo. All'interno di ciascuna sottocartella ci sono due file che contengono i parametri per ciascuno stage della trasformata di analisi/sintesi. Ogni file contiene i parametri come nell'equazione 2.1. In particolare:

- ***h*** **:** array 4-D (tensore) contenente i kernel del filtro lineare per lo stage corrispondente. Per gli stage dell'analisi la prima dimensione corrisponde al canale di output, la seconda dimensione a quello di input mentre la terza e la quarta alla posizione spaziale (y, x) all'interno del filtro. Al contrario per la trasformata di sintesi, la prima corrisponde al canale di input e la seconda a quello di output, mentre terza e quarta sono come per l'analisi.
- ***c*** **:** vettore che contiene i parametri additivi di bias per ogni canale.
- ***strides*** **:** il fattore (y, x) di sotto/sovracampionamento per ogni canale.
- ***beta*** **:** il vettore contenente i valori costanti di β nel denominatore presente nella funzione di normalizzazione, per ogni canale.
- ***gamma*** **:** array bidimensionale contenente i pesi γ presenti nella funzione di normalizzazione. Le dimensioni numerano i canali di input/output ma l'ordine non è consequenziale perché la matrice deve essere simmetrica per definizione.

É da notare che l'ultimo stage dell'analisi (**analysis-03**) e il primo della sintesi (**synthesis-00**) contengono solo il parametro di bias (***c***), allo scopo di ricentrare i valori nello spazio di codifica per poi poter applicare il quantizzatore (es. la funzione "round").

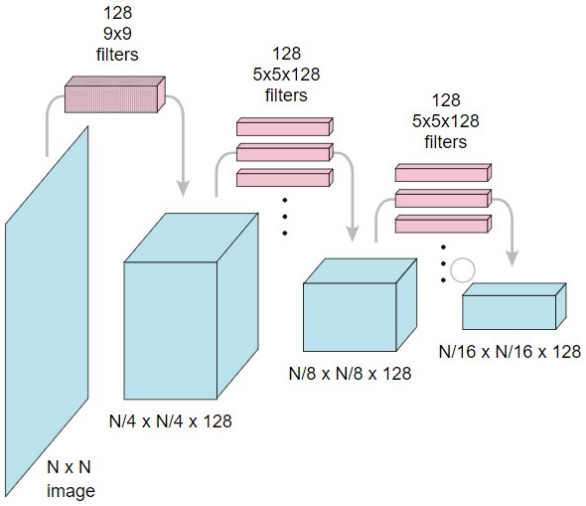

Figura 2.7: Tre step della trasformata di analisi per immagini in scala di grigi

2.3.1 Codice Matlab per immagini in scala di grigi

Di seguito è riportato nei listati 2.1, 2.2, 2.3 e 2.4 il codice che implementa la trasformata di analisi completa, con una funzione Matlab, è riportato anche il suo schema nella figura 2.7. Il codice completo si trova nel link al riferimento[4].

La funzione riceve in ingresso i seguenti input:

- **im**: un'immagine (matrice) di valori di pixel scala di grigi,
- **lambda** : determina la quantità di compressione, cioè il trade-off tra bit e qualità, e assume valori pari a 2^k, per $k \in [5:12]$,
- **noQuantize** : se viene specificato questo campo, in output verranno restituiti valori float invece di interi.

Mentre in uscita restituisce nella variabile **outRep** il tensore (matrice 3-D) contenente i 128 canali che costituiscono la rappresentazione codificata dell'immagine, quantizzati di default a valori interi.

Preliminarmente si controlla che l'immagine in input sia nel formato desiderato, cioè in scala di grigi e con valori nel range [0,1] e che lo sia anche il parametro lambda nel range [5:12]. Se i valori dei pixel non sono compresi tra [0,1] vengono riscalati in modo da rientrarci e lo stesso è fatto con lambda.

Come prima cosa, 2.1, viene realizzato un padding dell'immagine di 41 più il multiplo di 16 pixel rispetto alla grandezza dell'immagine per poter effettuare delle convoluzioni valide, e si accede alla cartella dei parametri di analisi per i valori in scala di grigi corrispondente al lambda dato in input.

Listing 2.1: funzione di analisi 'compress'

```
function [outRep] = compress(im, lambda, noQuantize)

% lambda deve essere una potenza di due con esponente che assume
    valori nel range [5:12], altrimenti si riscala.

pad = 41 + 16 + 16*ceil(size(im)/16) - size(im);

bdry = [floor(pad(1)/2), floor(pad(2)/2), ceil(pad(1)/2), ceil(pad(2)
    /2)];% ex [28 28 29 29]
im = [im(bdry(1):-1:1,bdry(2):-1:1), im(bdry(1):-1:1,:), im(bdry(1)
    :-1:1,end:-1:(end-bdry(4)+1));...
im(:,bdry(2):-1:1), im, im(:,end:-1:(end-bdry(4)+1));...
im(end:-1:(end-bdry(3)+1),bdry(2):-1:1), im(end:-1:(end-bdry(3)+1),:)
    , im(end:-1:(end-bdry(3)+1),end:-1:(end-bdry(4)+1))];

myPath = fileparts(mfilename('fullpath'));
paramDir = sprintf('%s/parameters/gray/lambda-%06d/', myPath, lambda)
    ;
```

Poi, in 2.2, si inizializza la rappresentazione di input ('inRep') con l'immagine, la dimensione dell'input ('inDims') con la dimensione dell'immagine e il numero di canali di output ('inDepth') a 1.

Dopodiché inizia il ciclo che svolge i tre stage di convoluzione. All'inizio del ciclo (cioè dello stage) vengono caricati tutti i parametri di analisi ('h', 'stides', 'c', 'beta', 'gamma'), necessari per quel determinato stage. Si controlla che il numero di canali di output('inDepth') sia uguale alla seconda dimensione del filtro 'h', altrimenti dà errore. Si calcolano le dimensioni di output ('outDims'), la prima e l'ultima locazione valida per effettuare la convoluzione ('inStart' e 'inStop') e si assegna come numero di canali di output('outDepth') la prima dimensione del

filtro 'h'. Si inizializza la rappresentazione in uscita dal filtro ('outRep') con una variabile che abbia come dimensioni la concatenazione delle dimensioni di output ('outDims') e del numero di canali di output ('outDepth').

Listing 2.2: cont. funzione di analisi 'compress'

```
15 inRep = im;
16 inDims = size(im);
17 inDepth = 1;
18
19 for stage = 0:2
20         load(sprintf('%sanalysis-%02d.mat', paramDir, stage), 'h', '
               strides', 'c', 'beta', 'gamma');
21         filters = double(h);
22         clear h;
23         fDims = [size(filters,3),size(filters,4)];
24         downSampling = double(strides);
25         clear strides;
26         const = double(c);
27         clear c;
28         beta = double(beta);
29         gamma = double(gamma)
30
31         if (inDepth ~= size(filters,2))
32                 error(sprintf('Stage %d: depth of input %d does not
                       match filter set',stage,inDepth));
33         end
34         outDims = floor((inDims-(fDims-1)+(downSampling-1))./
               downSampling);
35         inStart = 1+(fDims-1)/2;
36         inStop = inDims-(fDims-1)/2;
37         outDepth = size(filters,1);
38         outRep = zeros([outDims, outDepth]);
```

A questo punto, in 2.3, si cicla, doppiamente sulla profondità in output ('outDepth') e poi sulla profondità di input ('inDepth'). Per poter calcolare ogni canale della rappresentazione di output questo viene prima sottocampionato e successivamente sommato ad ogni iterazione su i canali di input alla correlazione tra il filtro e il canale corrispondente, in modo da simulare una convoluzione. Una volta terminati il ciclo interno si somma il valore di bias. Se l'immagine da comprimere è ad esempio di 512×512, al primo stage la variabile 'outRep' avrà dimensione 141×141×128, al secondo 69×69×128 e all'ultimo 33×33×128.

Una volta ottenuta la rappresentazione di output per poter terminare lo stage si applica, all'esterno dei due cicli, il GDN.

Listing 2.3: cont. funzione di analisi 'compress'

```
40         for n=1:outDepth
41                 for m=1:inDepth
42                         filt = squeeze(filters(n,m,:,:));
43                         edges = 'reflect1';
```

```
44                              outRep(:,:,n) = outRep(:,:,n) + corrDn(inRep
                                    (:,:,m), filt, edges, downSampling,
                                    inStart, inStop);
45                      end
46                      outRep(:,:,n) = outRep(:,:,n) + const(n);
47              end
48
49              norm2 = reshape(outRep.^2, [prod(outDims), outDepth]);
50              norm2 = norm2 * gamma + ones(size(norm2,1),1) * beta;
51              outRep = outRep ./ reshape(sqrt(norm2), size(outRep));
```

Prima di iniziare con un nuovo stage si riassegnano, in 2.4, i valori della rappresentazione di input ('inRep') e la dimensione dell'input ('inDims'), rispettivamente con la rappresentazione ottenuta dallo stage appena svolto ('outRep'), con la dimensiona di questa ('outDims'). Il numero di canali di output ('inDepth') con quello calcolato durante lo stage ('outDepth').

L'ultimo stage, consiste nel sommare il bias e quindi è svolto in un altro ciclo. Infine i valori della trasformata vengono quantizzati con valori interi se non si era specificato diversamente nell'input della funzione.

Listing 2.4: cont. funzione di analisi 'compress'

```
53              inRep = outRep;
54              inDims = outDims;
55              inDepth = outDepth;
56      end
57
58      stage = 3;
59      load(sprintf('%sanalysis-%02d.mat', paramDir, stage), 'c');
60      const = double(c);
61      clear c;
62
63      for n=1:outDepth
64              outRep(:,:,n) = outRep(:,:,n) + const(n);
65      end
66
67      if (~noQuantize)
68              outRep = round(outRep);
69      end
70
71      return
```

Di seguito è riportato nei listati 2.5, 2.6 e 2.7 il codice che implementa la trasformata di sintesi completa, con una funzione Matlab, per ricostruire un'immagine dalla sua rappresentazione codificata generata dalla funzione 'compress'. Gli input che si ricevono in ingresso sono i seguenti,

- **codedIm** : tensore (matrice 3-D) contenente i 128 canali che costituiscono la rappresentazione codificata dell'immagine,
- **lambda** : determina la quantità di compressione, cioè il trade-off tra bit e qualità, e assume valori pari a 2^k, per $k \in [5 : 12]$, deve essere lo stesso valore che è usato per generare la 'codedIm',

- **imDims** : vettore di due elementi [y,x] che specifica le dimensioni dell'immagine da ricostruire, dovrebbero coincidere con quelle dell'immagine originale.

Mentre in uscita restituisce nella variabile **outIm** l'immagine ricostruita.

Preliminarmente si controlla che il lambda in input appartenga al range, in caso contrario il valore viene riscalato.

Come prima cosa si accede alla cartella dei parametri di sintesi per i valori in scala di grigi corrispondente al lambda, in 2.5. Poi si inizializza la rappresentazione di input ('inRep') con la rappresentazione codificata dell'immagine in input, la dimensione dell'input ('inDims') con un vettore che contiene il valore delle prime due dimensioni di 'inRep' e il numero di canali di output ('inDepth') pari alla terza dimensione di 'inRep'.

Il primo stage, specularmente a quanto viene fatto nella trasformata di analisi, consiste nel ciclare sul numero di canali ('inDepth') della rappresentazione ('inRep') per sommare a ciascuno il bias.

Listing 2.5: funzione di sintesi 'uncompress'

```
1  function [outIm] = uncompress(codedIm, lambda, imDims)
2
3  myPath = fileparts(mfilename('fullpath'));
4  paramDir = sprintf('%s/parameters/gray/lambda-%06d/', myPath, lambda)
       ;
5
6  inRep = codedIm;
7  inDims = [size(inRep,1), size(inRep,2)];
8  inDepth = size(inRep,3);
9
10 stage = 0;
11 load(sprintf('%ssynthesis-%02d.mat', paramDir, stage), 'c');
12 const = double(c);
13 clear c;
14
15 for n=1:inDepth
16         inRep(:,:,n) = inRep(:,:,n) + const(n);
17 end
```

Dopodiché, in 2.6, inizia il ciclo che svolge i tre stage di convoluzione. All'inizio del ciclo (cioè dello stage) vengono caricati tutti i parametri di sintesi ('h', 'stides', 'c', 'beta', 'gamma'), necessari per quel determinato stage. Si controlla che il numero di canali di output('inDepth') sia uguale alla prima dimensione del filtro 'h', altrimenti dà errore. Si calcolano le dimensioni di output ('outDims'), la prima e l'ultima locazione valida per effettuare la convoluzione ('inStart' e 'inStop') e si assegna come numero di canali di output('outDepth') la seconda dimensione del filtro 'h'. Si inizializza la rappresentazione in uscita dal filtro ('outRep') con una variabile che abbia come dimensioni la concatenazione delle dimensioni di output ('outDims') e del numero di canali di output ('outDepth'), in maniera molto simile all'analisi.

Successivamente si applica il GDN, diversamente da quanto visto nella funzione di 'compress' in cui l'applicazione della convoluzione e del GDN si invertono.

Listing 2.6: cont. funzione di sintesi 'uncompress'

```
19
20 for stage = 1:3
```

```
21          load(sprintf('%ssynthesis-%02d.mat', paramDir, stage), 'h', '
                strides', 'c', 'beta', 'gamma');
22          filters = double(h);
23          clear h;
24          fDims = [size(filters,3),size(filters,4)];
25          upSampling = double(strides);
26          clear strides;
27          const = double(c);
28          clear c;
29          beta = double(beta);
30          gamma = double(gamma);
31
32          if (inDepth ~= size(filters,1))
33                  error(sprintf('Stage %d: depth of input %d does not
                        match that of filter set',stage,inDepth));
34          end
35          outDims = (inDims-1).*upSampling + fDims;
36          outStart = 1+(fDims-1)/2;
37          outStop = outDims-(fDims-1)/2;
38          outDepth = size(filters,2);
39
40          norm2 = reshape(inRep.^2, [prod(inDims), inDepth]);
41          norm2 = norm2 * gamma + ones(size(norm2,1),1) * beta;
42          inRep = inRep .* reshape(sqrt(norm2), size(inRep));
43          outRep = zeros([outDims, outDepth]);
```

A questo punto, in 2.7, si cicla doppiamente sulla profondità in output ('outDepth') e poi sulla profondità di input ('inDepth'). Per poter calcolare ogni canale della rappresentazione di output questo viene prima sovracampionato e successivamente sommato ad ogni iterazione su i canali di input alla correlazione tra il filtro e il canale corrispondente, in modo da simulare una convoluzione. Una volta terminati il ciclo interno si somma il valore di bias.

Prima di iniziare con un nuovo stage si riassegnano i valori della rappresentazione di input ('inRep') e la dimensione dell'input ('inDims'), rispettivamente con la rappresentazione ottenuta dallo stage appena svolto ('outRep'), con le dimensione di questa ('outDims'). Il numero di canali di output ('inDepth') con quello calcolato durante lo stage ('outDepth').

La dimensione dell'immagine finale restituita dalla funzione viene regolata per coincidere con le dimensioni specificate inizialmente in input.

Listing 2.7: cont. funzione di sintesi 'uncompress'

```
45
46          for n=1:outDepth
47                  outIm = zeros(outDims);
48                  for m=1:inDepth
49                          filt = squeeze(filters(m,n,:,:));
50                          edges = 'reflect1';
51                          upConv(inRep(:,:,m), filt, edges, upSampling,
                                outStart, outStop, outIm);
52                  end
53                  outRep(:,:,n) = outIm + const(n);
```

```
54          end
55
56          inRep = outRep;
57          inDims = outDims;
58          inDepth = outDepth;
59  end
60
61  bdry = [floor((size(outRep)-imDims)/2), ceil((size(outRep)-imDims)/2)
        ];
62  outIm = outRep((bdry(1)+1):(size(outRep,1)-bdry(3)), ((bdry(2)+1):(
        size(outRep,2)-bdry(4))));
63
64  return
```

2.3.2 Codice Matlab per immagini a colori

Il codice per l'elaborazione delle immagini a colori, non essendo disponibile, è stato implementato partendo da quello già fornito per immagini in scala di grigi nel riferimento [4].

Le differenze più importanti sono,

- il numero di filtri delle trasformate non è più 128 ma 192;
- lambda è sempre una potenza di 2 ma i suoi esponenti variano nel range [7:12];
- cambia l'indirizzo per accedere alla cartella dei parametri, che non è più quello relativo ai parametri ottimizzati in scala di grigi ma ai parametri a colori;
- nella funzione compress, che realizza la trasformata di analisi, il numero di canali di output inizializzati nella variabile 'inDepth' non è più uno ma 3.

2.3.3 Codificatore entropico aritmetico per calcolare bit-rate e PSNR

Siccome non era disponibile il codice per il codificatore aritmetico, che serve per calcolare il bit-rate dell'immagine compressa, anche questo, come la versione a colori delle trasformate è stato necessario implementarlo a parte. Il codificatore aritmetico converte i coefficienti quantizzati in forma binaria e li comprime leggendoli per piani di bit. In 2.8 e 2.9 sono riportati gli pseudo-codici del codificatore aritmetico, rispettivamente per la DCT e per le trasformate ottimizzate.

In entrambi i casi, la variabile 'blk_size' è la dimensione del blocco quadrato con cui si vuole partizionare l'immagine 'I' in input. Per la DCT 'T' è la variabile bidimensionale con i coefficienti della DCT delle stesse dimensioni dei blocchi dell'immagine. Prima di procedere con la funzione 'zigzag' che scannerizza i blocchi trasformati per calcolarne il rate, i coefficienti della trasformata sono quantizzati con un passo di quantizzazione pari a 'quant_step', che determina il trade-off tra rate e distorsione. Invece, per le trasformate ottimizzate, il trade-off tra rate e distorsione è 'lambda'. Al variare di quest'ultimo cambiano i parametri delle trasformate. Per entrambe le trasformate una volta ottenuto il blocco trasformato se ne calcola il rate con la funzione 'ae_coder'. Dopodiché una volta ricostruita l'intera immagine si calcola il PSNR. Entrambe le funzioni restituiscono il rate, il PSNR e l'immagine ricostruita con la rispettiva trasformata utilizzata.

```
1  [rate_dct, psnr_dct, I_rec_dct] = computingDctRateDist(I, blk_size,
       quant_step)
```

```
2
3  l1 = size(I,1)/blk_size;
4  l2 = size(I,2)/blk_size;
5  T = dctmtx(blk_size);
6  for J=1:l1
7      for K=1:l2
8          I_blk = double(I(blk_size*(J-1)+1:blk_size*J, blk_size*(K-1)
               +1:blk_size*K));
9          coef = T * I_blk * T';
10         coef = round(coef/quant_step);
11         coef = zigzag(coef);
12
13         rate_dct = rate_dct + ae_coder(coef',1);
14
15         coef_rec = invzigzag(coef*quant_step, blk_size, blk_size);
16         I_blk_rec = T' * coef_rec * T;
17         I_blk_rec = reshape(I_blk_rec,blk_size,blk_size);
18         I_rec_dct(blk_size*(J-1)+1:blk_size*J, blk_size*(K-1)+1:
               blk_size*K) = I_blk_rec;
19     end
20 end
21 psnr_dct = PSNR(I, I_rec_dct);
22 rate_dct = rate_dct/(size(I,1)*size(I,2));
```

Listing 2.8: Calcolo bit-rate per la DCT.

```
1  [rate_opt psnr_opt, I_rec_opt] = computingOptRateDist(I, blk_size,
       lambda)
2
3  l1 = size(I,1)/blk_size;
4  l2 = size(I,2)/blk_size;
5  for J=1:l1
6      for K=1:l2
7          I_blk = double(I(blk_size*(J-1)+1:blk_size*J, blk_size*(K-1)
               +1:blk_size*K));
8          coef = compress(I_blk, lambda);
9          sizeCoef = size(coef);
10         coef_rec = zeros(sizeCoef);
11         for i = 1:sizeCoef(3)
12             layer = coef(:,:,i);
13             layer = zigzag(layer);
14
15             rate_opt = rate_opt + ae_coder(layer',1);
16
17             coef_rec(:,:,i) = invzigzag(layer, sizeCoef(1), sizeCoef
                   (2));
18         end
19         I_blk_rec = uncompress(coef_rec, lambda, [blk_size blk_size])
               ;
20         I_blk_rec = reshape(I_blk_rec,blk_size,blk_size);
```

```
21          I_rec_opt(blk_size*(J-1)+1:blk_size*J, blk_size*(K-1)+1:
                blk_size*K) = I_blk_rec;
22
23      end
24
25  end
26  psnr_opt = PSNR(I, I_rec_opt);
27  rate_opt = rate_opt/(size(I,1)*size(I,2));
```

Listing 2.9: Calcolo bit-rate per le trasformate ottimizzate.

2.4 Requisiti del progetto

Per poter utilizzare nella pratica questo sistema è stato necessario scaricare sul PC i seguenti elementi:

- Matlab versione 2017 o più recente,
- matlabPyrTools,
- i parametri pre-trained del modello,
- codificatore entropico aritmetico.

Capitolo 3

Test e Risultati

Questo capitolo è dedicato alla fase di sperimentazione e analisi dei risultati. L'obiettivo è confrontare il set di trasformate ottenuto dal training dell'architettura descritta nel capitolo 2 con la DCT, calcolando le prestazioni rate-distortion. Inizialmente gli esperimenti sono stati fatti sulle singole immagini di 512×512 pixel ciascuna, in 3.1, 3.2, 3.4 e 3.3, e successivamente su un set di 23 immagini, reperibili in [4], da cui è stata calcolata la media per creare i grafici. Ciascuna immagine da elaborare, prima di applicarne la trasformata, è stata partizionata in blocchi quadrati. La taglia del blocco è stata fatta variare tra i valori di 32, 64, 128, 256, e 512 pixel. Infine è stato fatto un ulteriore confronto tra i due metodi considerando alcune immagini HD e 4K.

Figura 3.1: girl

Figura 3.2: Lena

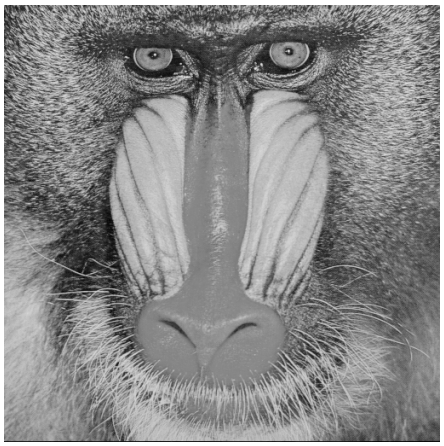

Figura 3.3: monkey

Figura 3.4: pepper

Per ogni immagine è stato calcolato il bit-rate e il PSNR, al variare della dimensione del blocco e del parametro di trade-off, che nel caso della DCT è lo step di quantizzazione, che può assumere come valori $[40, 50, 60, 70, 80, 90]$ e nel caso delle trasformate ottimizzate è lambda, che assume valori 2^k $k = 7, ..., 12$.

È importante avere presente, per l'interpretazione dei risultati, che il training è stato fatto considerando immagini intere non partizionate a blocchi. E che oltre al PSNR a parità di bit-rate bisogna anche fare una valutazione della qualità della percezione dell'immagine ricostruita.

3.1 Risultati su immagini singole

Di seguito verranno mostrati i grafici con il rate e il PSNR per ciascuna delle immagini 3.1,3.2,3.4 e 3.3.

3.1.1 Immagine "girl"

I grafici 3.5 e 3.6 sono relativi all'immagine 3.1. A parità di dimensione le curve sono calcolate tenendo fissa la dimensione del blocco e facendo variare il parametro di trade-off. Nel grafico 3.5 che riporta i valori per la DCT, più diminuisce la dimensione del blocco più la qualità dell'immagine migliora a parità di step di quantizzazione.

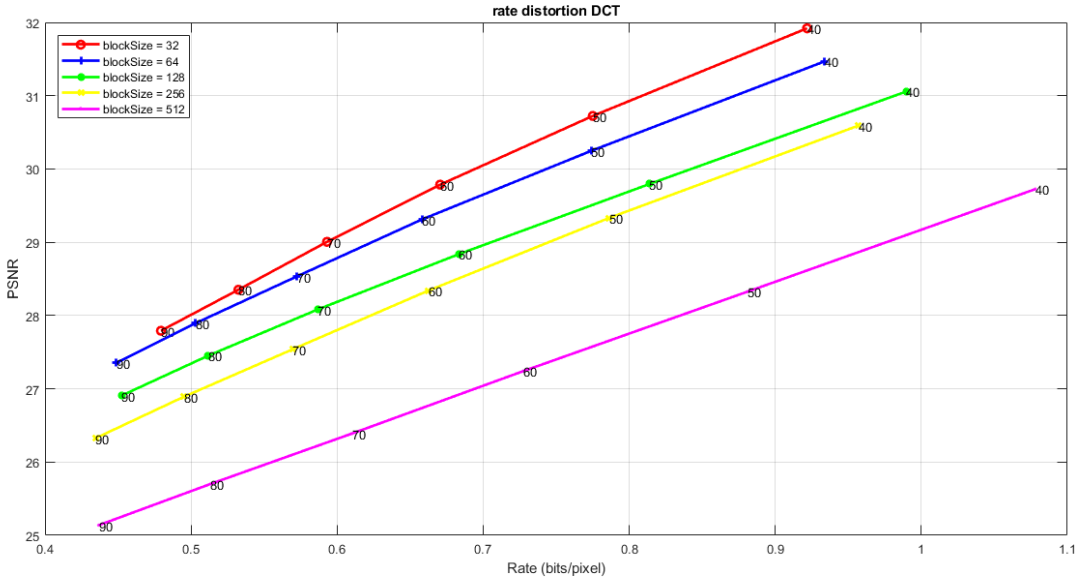

Figura 3.5

Nel grafico 3.6 che riporta i valori per le trasformate ottimizzate, il comportamento è differente dalla DCT, più la dimensione del blocco è piccola più la qualità dell'immagine tende a diminuire e nonostante ciò il rate aumenta.

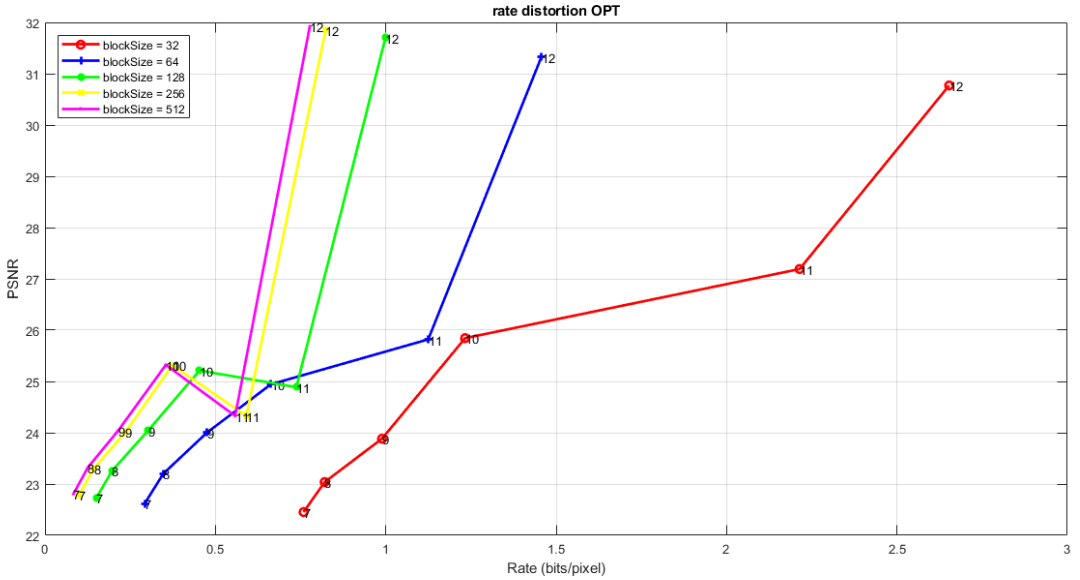

Figura 3.6

Complessivamente il miglior caso della DCT e il miglior caso delle trasformate ottimizzate sono praticamente equivalenti per il PSNR, anche se la DCT riporta un rate leggermente più alto.

3.1.2 Immagine "Lena"

I grafici 3.7 e 3.8 sono relativi all'immagine 3.2. A parità di dimensione le curve sono calcolate tenendo fissa la dimensione del blocco e facendo variare il parametro di trade-off. Nel grafico 3.7 che riporta i valori per la DCT, si nota che per rate inferiori a 0.5 circa, i risultati migliori sono raggiunti nel caso in cui la dimensione del blocco è massima, cioè 512. Per valori di rate maggiori di 0.5, la trasformata ottiene performance migliori per dimensione di blocco pari a 32.

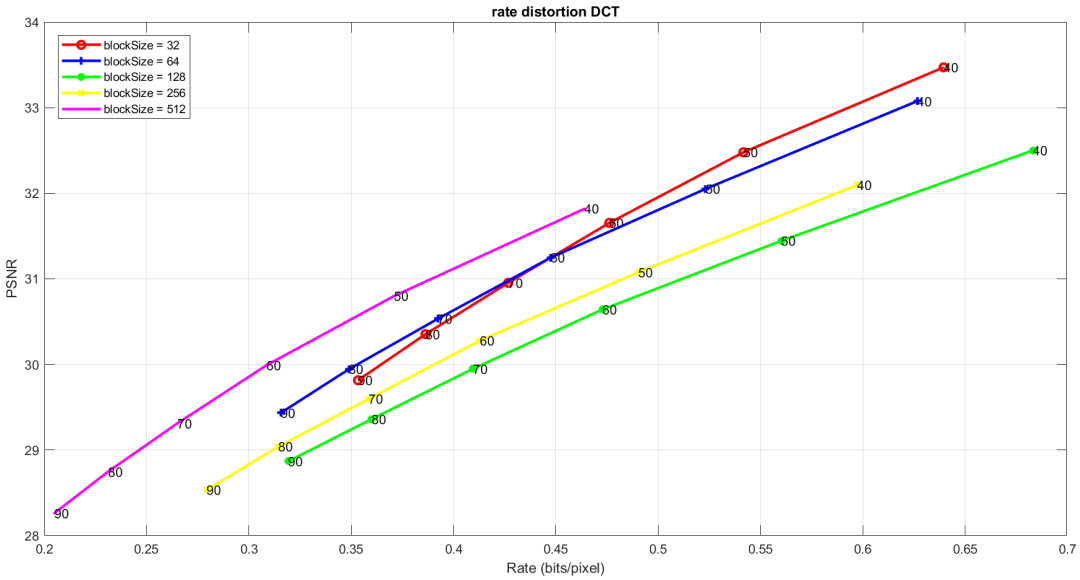

Figura 3.7

Nel grafico 3.8 che riporta i valori per le trasformate ottimizzate più la dimensione del blocco è piccola più la qualità dell'immagine tende a diminuire e nonostante ciò il rate aumenta, come per l'immagine 3.1.

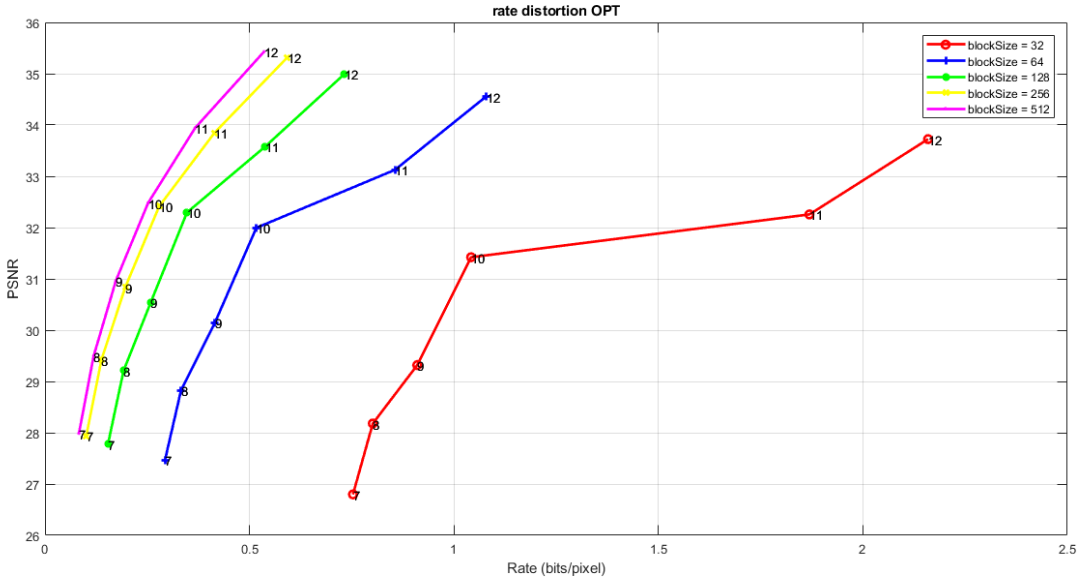

Figura 3.8

Nel grafico 3.7 il PSNR massimo è circa 33.5 con un rate di 0.55 e il range di variazione del rate non supera mai lo 0.7. Nel grafico 3.8, il PSNR massimo è 35.5 con un rate simile alla DCT, ma nei casi con blocchi di dimensione 32 e 64 pixel, il rate assume valori maggiori di 2 e di 1 rispettivamente, quindi non si ottiene un buon compromesso tra qualità della ricostruita e rate.

3.1.3 Immagine "monkey"

I grafici 3.9 e 3.10 sono relativi all'immagine 3.3. A parità di dimensione le curve sono calcolate tenendo fissa la dimensione del blocco e facendo variare il parametro di trade-off. Nel grafico 3.9 che riporta i valori per la DCT, si nota che ci sono due gruppi di curve, il primo composto dalle curve che riportano i risultati per blocchi di 32 e 64 pixel, il secondo invece composto dalle curve relative a blocchi di dimensione 128, 256 e 512 pixel. Il primo gruppo a parità di step di quantizzazione tende ad assumere un rate simile al secondo ma con un PSNR sempre migliore.

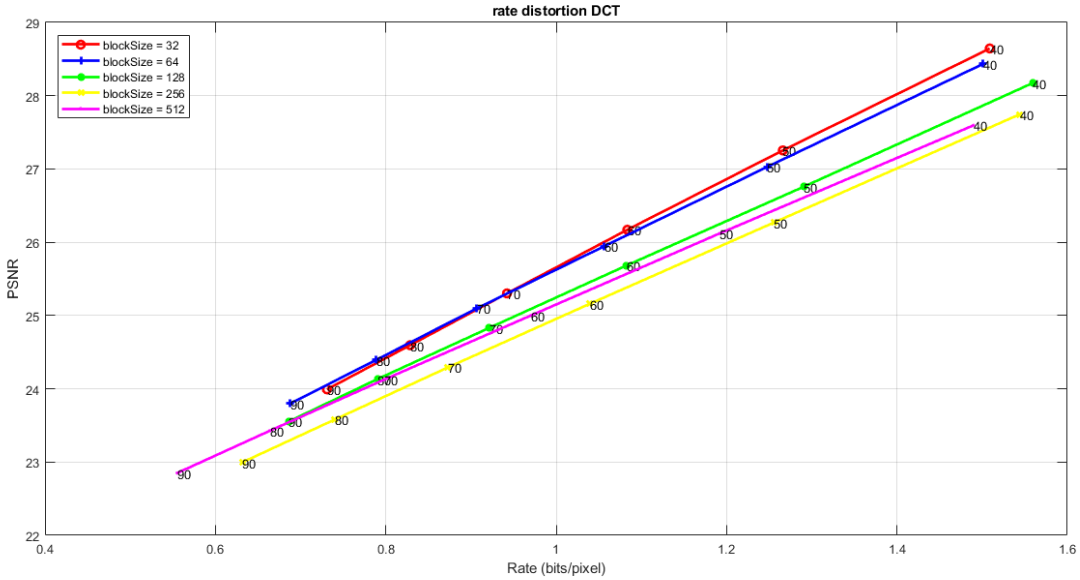

Figura 3.9

Nel grafico 3.10 che riporta i valori per le trasformate ottimizzate, come ci si aspetterebbe avendo presente i risultati delle immagini precedenti, per blocchi piccoli, da 32 e 64 pixel, non si ottengono dei risultati accettabili di rate-distortion.

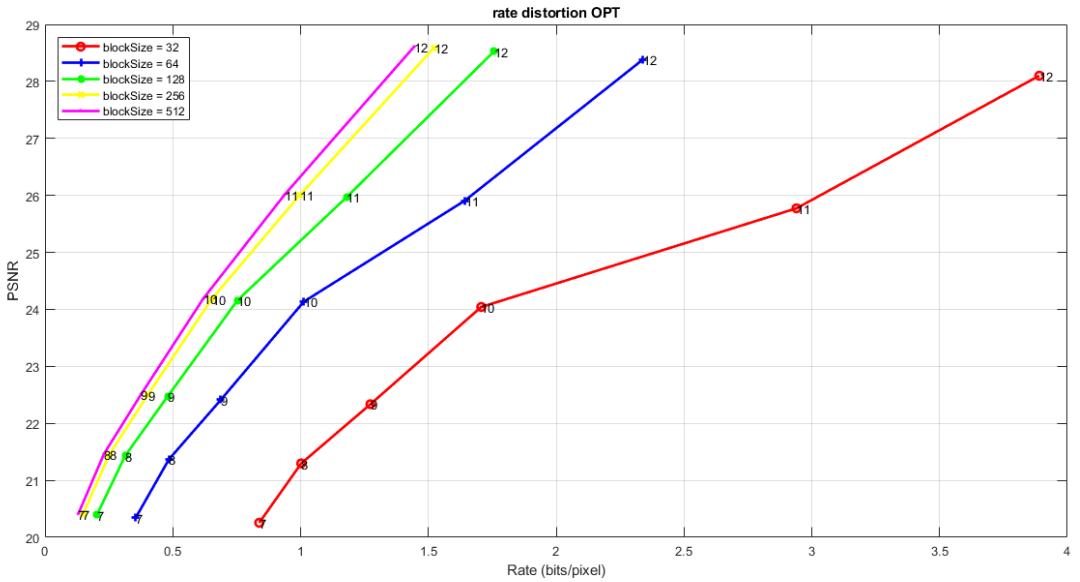

Figura 3.10

Nel caso di questa immagine, differentemente dalle precedenti, per alti PSNR il rate in entrambe le trasformate è maggiore di 1, quindi si potrebbe dedurre che questo non è dovuto al tipo di trasformata utilizzata ma è intrinseco all'immagine.

3.1.4 Immagine "pepper"

I grafici 3.11 e 3.12 sono relativi all'immagine 3.4. A parità di dimensione le curve sono calcolate tenendo fissa la dimensione del blocco e facendo variare il parametro di trade-off. Il grafico 3.11 che riporta i valori per la DCT mostra che si ottengono risultati migliori quando l'immagine è compressa a blocchi di 512 pixel dal punto di vista del rate-distortion per rate minori di 0.25. Il valore più alto di PSNR è raggiunto nel caso in cui l'immagine è partizionata in blocchi di 32 pixel a fronte di un rate quasi doppio.

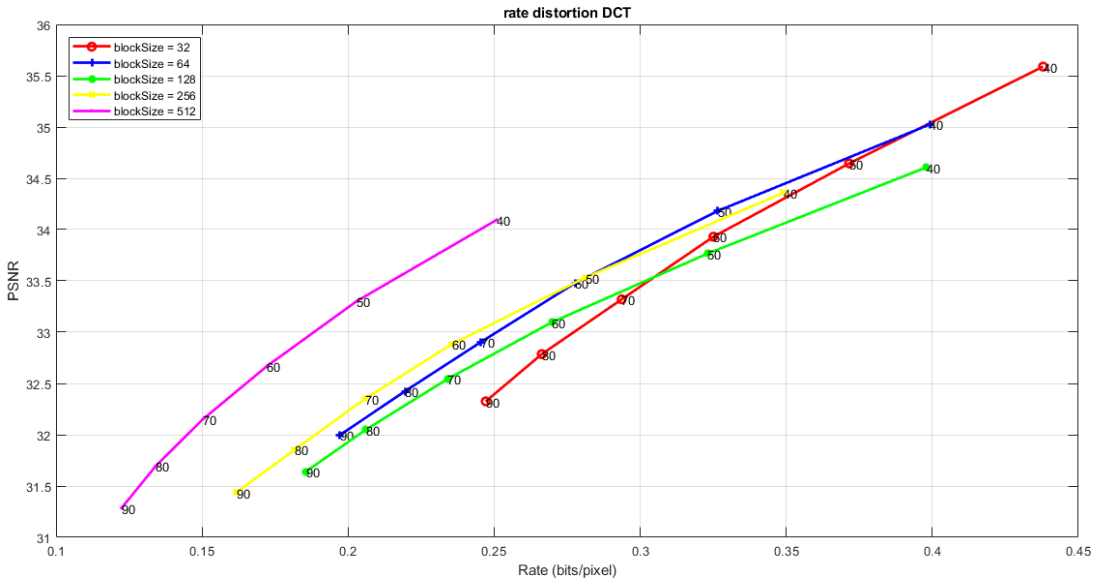

Figura 3.11

Nel grafico 3.12 che riporta i valori per le trasformate ottimizzate, più la dimensione del blocco è piccola più la qualità dell'immagine tende a diminuire a parità di lambda. Mentre nelle altre immagini il PSNR per lambda pari a 12 assumeva un valore costante, indipendentemente dalla taglia del blocco, in questo caso la differenza è decisamente più evidente.

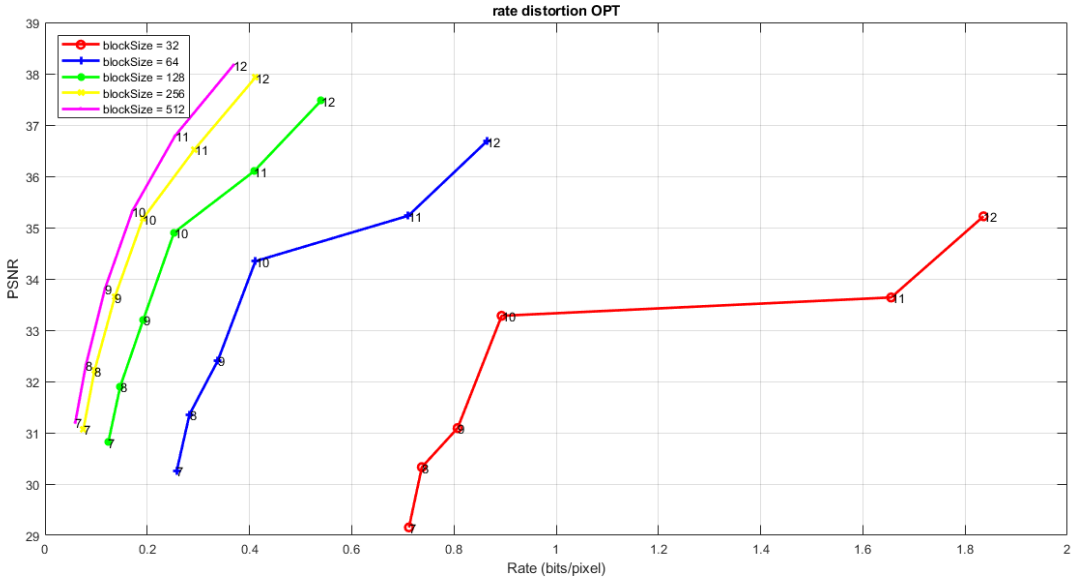

Figura 3.12

Confrontando i due grafici si nota che quello delle trasformate ottimizzate in diversi casi supera il PSNR massimo ottenuto dalla DCT, pur mantenendo un rate più basso.

3.1.5 Confronto della percezione delle ricostruite

Come è stato detto precedentemente, oltre al calcolo del rate e del PSNR è importante anche la valutazione percettiva delle immagini ricostruite per poter valutare la bontà delle trasformate. Di seguito sono riportate solamente alcune delle immagini ricostruite per dare un'idea della differenza tra le due trasformate.

Figura 3.13: immagine originale

L'immagine 3.14 è stata ottenuta utilizzando la DCT applicandola a tutta l'immagine 3.13 intera e con uno step di quantizzazione di 40. I valori corrispondenti di rate e di PSNR sono 0.73 e 27.

Figura 3.14: immagine ottenuta con la DCT

L'immagine 3.15 è stata ottenuta utilizzando le trasformate ottimizzate considerando tutta l'immagine 3.13 intera e con un lambda pari a 9. Il valore del rate è 0.21 e quello del PSNR 24.

Figura 3.15: immagine ottenuta con le trasformate ottimizzate

Figura 3.16: immagine originale

L'immagine 3.17 è stata ottenuta utilizzando la DCT, partizionando l'immagine 3.16 in blocchi di 32 pixel e con uno step di quantizzazione di 40. In questo modo i valori che si sono ottenuti sono 0.39 per il rate e 30 per il PSNR.

Figura 3.17: immagine ottenuta con la DCT

L'immagine 3.18 è la ricostruita dell'immagine 3.16 considerata intera e con un lambda pari a 9, da cui si è ottenuto un rate di 0.12 e un PSNR di 29.

Figura 3.18: immagine ottenuta con le trasformate ottimizzate

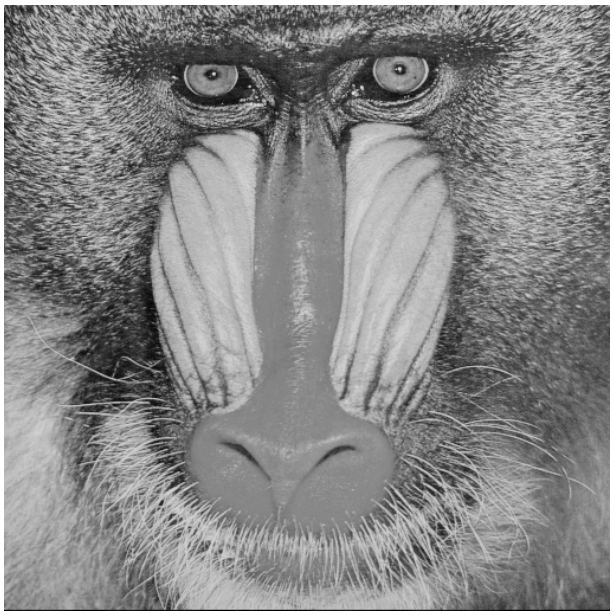

Figura 3.19: immagine originale

La ricostruita corrispondente all'applicazione della DCT all'immagine 3.19 partizionata in blocchi di 32 pixel e usando uno step di quantizzazione pari a 80, è mostrata in 3.20 il cui rate risulta essere 0.83 e il PSNR 25.

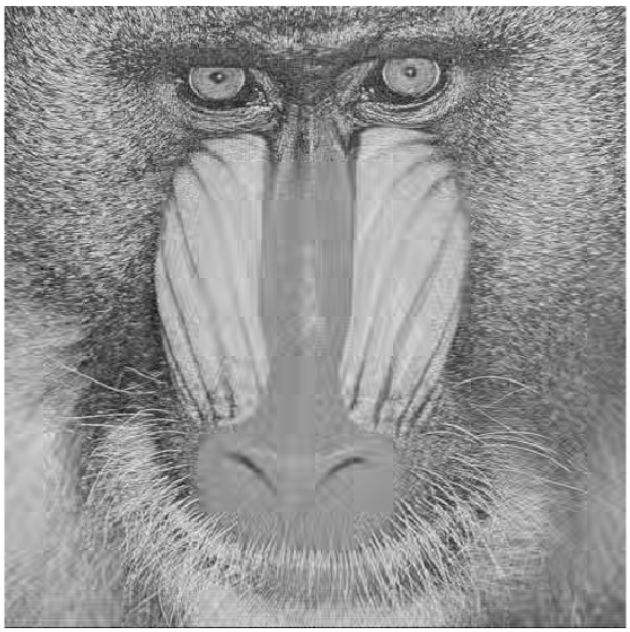

Figura 3.20: immagine ottenuta con la DCT

L'immagine 3.21 invece è ottenuta dalle trasformate ottimizzate considerando sempre l'immagine 3.19 intera e con un lambda di 9. Il risultato è un rate di 0.23 e un PSNR di 21.

Figura 3.21: immagine ottenuta con le trasformate ottimizzate

Figura 3.22: immagine originale

L'ultimo caso mostrato è relativo all'immagine 3.22, considerando per la DCT l'immagine partizionata in blocchi di 32 e con step di quantizzazione di 80, la cui ricostruita è l'immagine 3.23, con un rate di 0.27 e un PSNR di 33.

Figura 3.23: immagine ottenuta con la DCT

L'immagine 3.24 è stata ottenuta dall'immagine 3.22, utilizzando le trasformate ottimizzate sull'immagine intera e con un lambda di 9, ottenendo come risultati un rate di 0.08 e un PSNR di 32.

Figura 3.24: immagine ottenuta con le trasformate ottimizzate

3.2 Risultati in media

I seguenti grafici, 3.25 e 3.26, mostrano i valori medi su un set di 23 immagini, in cui è stato fatto variare il parametro di trade-off e la dimensione del blocco.

Nel grafico in figura3.25, è mostrato l'andamento del rate-distortion per la DCT, dove si può vedere che i risultati peggiori si verificano nel caso di immagini partizionate a blocchi di 126 e 256 pixel. Invece si ottengono i risultati migliori per 'blockSize' pari a 32 e 64. Questo risultato si era già notato quando sono state analizzate le immagini singolarmente ed è un fatto noto che la DCT sia più performante quando l'immagine è partizionata in blocchi di piccole dimensioni, infatti lo standard JPEG elabora blocchi di immagini di 8×8 pixel.

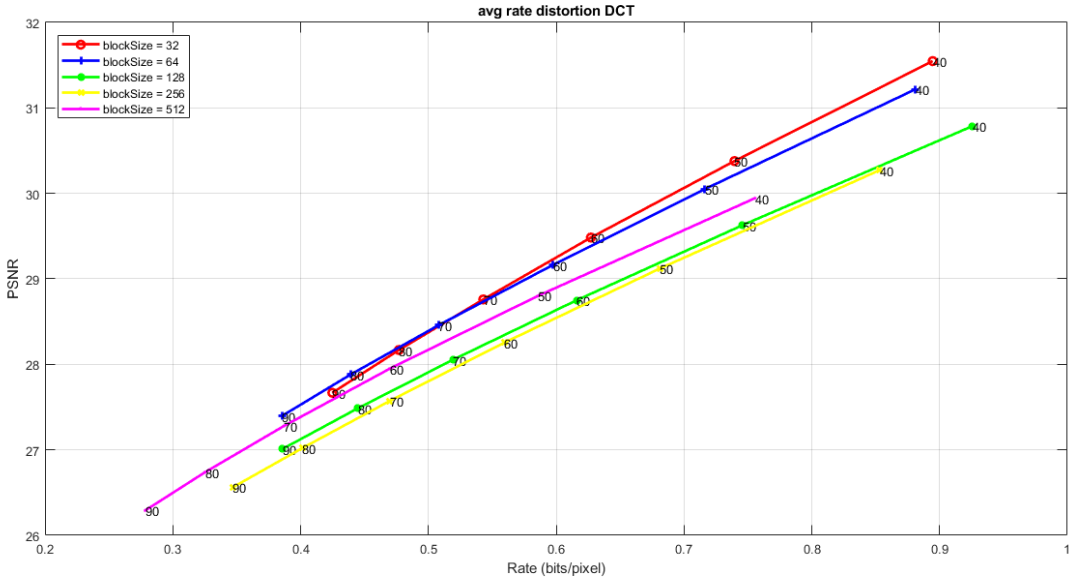

Figura 3.25

Il grafico 3.26 illustra l'andamento del rate-distortion per le trasformate ottimizzate. Come si era già visto nei casi di immagini singole, più aumenta la dimensione del blocco più si ottengono risultati migliori. A questa considerazione va ricordato che il training della CNN è stato svolto su immagini intere e non partizionate. Tuttavia si è voluto fare un'analisi su immagini partizionate, poiché nel caso della compressione video i frame vengono suddivisi in blocchi per realizzare l'applicazione della motion estiamtion.

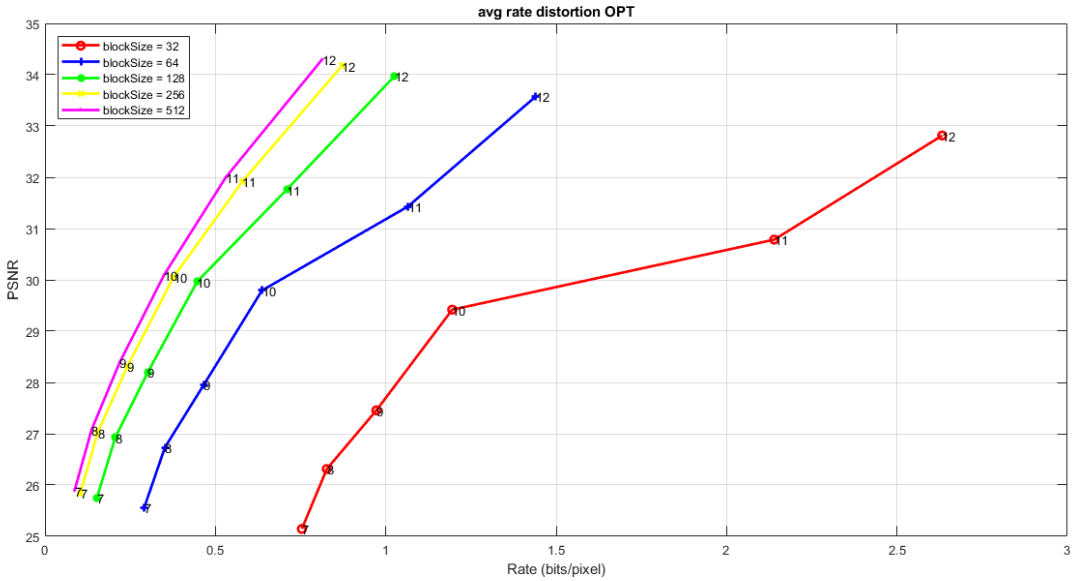

Figura 3.26

Analizzando i risultati dei due tipi di trasformate si conferma il fatto che la DCT ottiene

prestazioni migliori per immagini partizionate in blocchi di piccole dimensioni, mentre le trasformate ottimizzate per immagini non partizionate. Considerando il caso migliore per ciascuna della due, emerge che la DCT a parità di rate assume valori di PSNR più bassi di almeno due ordini di grandezza. Quindi le trasformate ottimizzate comprimono in maniera migliore l'immagine e non presentano artefatti di blocchettizzazione.

3.3 Confronto tra le due trasformate

Di seguito sono mostrati i grafici delle due trasformate messe a confronto a parità di dimensione del blocco e facendo variare il parametro di trade-off.

È evidente dai grafici che all'aumentare della dimensione dei blocchi cambia l'efficacia delle trasformate. Nel primo grafico,3.27, relativo a blocchi di dimensione 32, la DCT ottiene risultati migliori a parità di rate. Nel grafico 3.28 le curve delle due trasformate iniziano a sovrapporsi, ma la DCT continua ad avere prestazioni migliori sempre per quanto riguarda il rate. Dal grafico 3.29 e poi nei successivi, 3.30 e 3.31, la curva dei risultati delle trasformate ottimizzate è sempre migliore di quella della DCT.

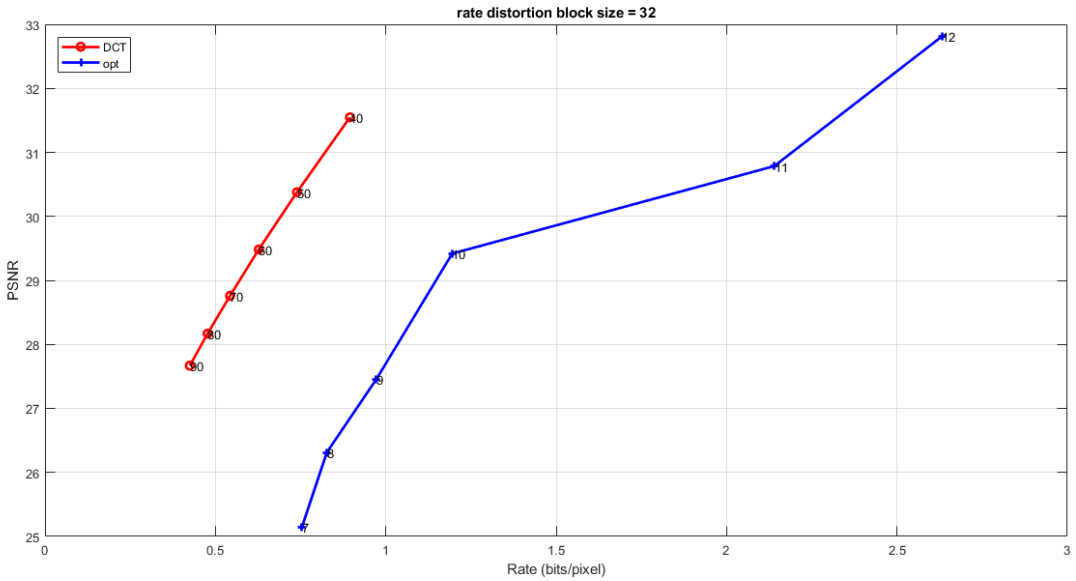

Figura 3.27

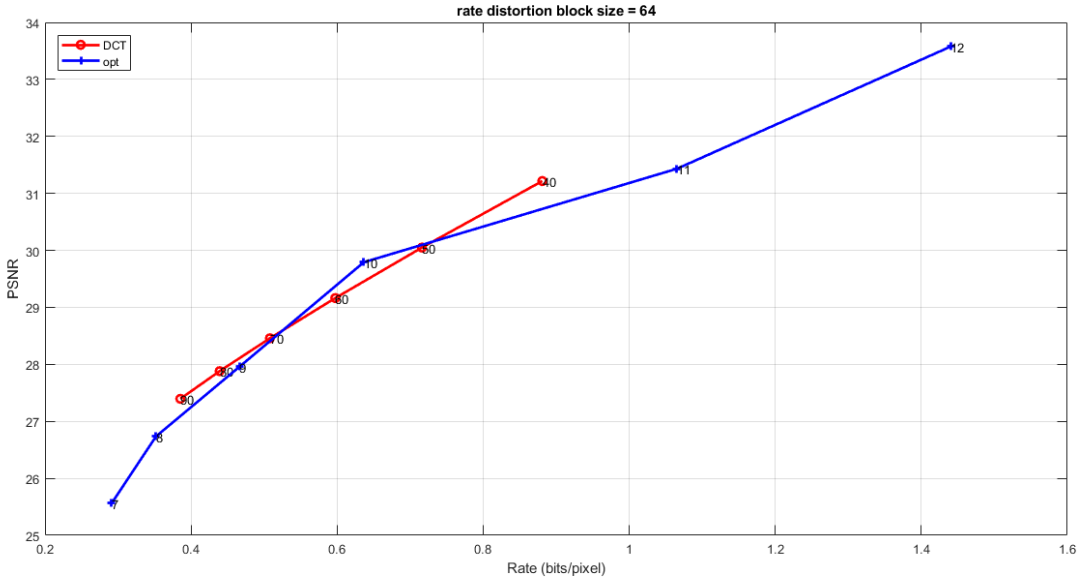

Figura 3.28

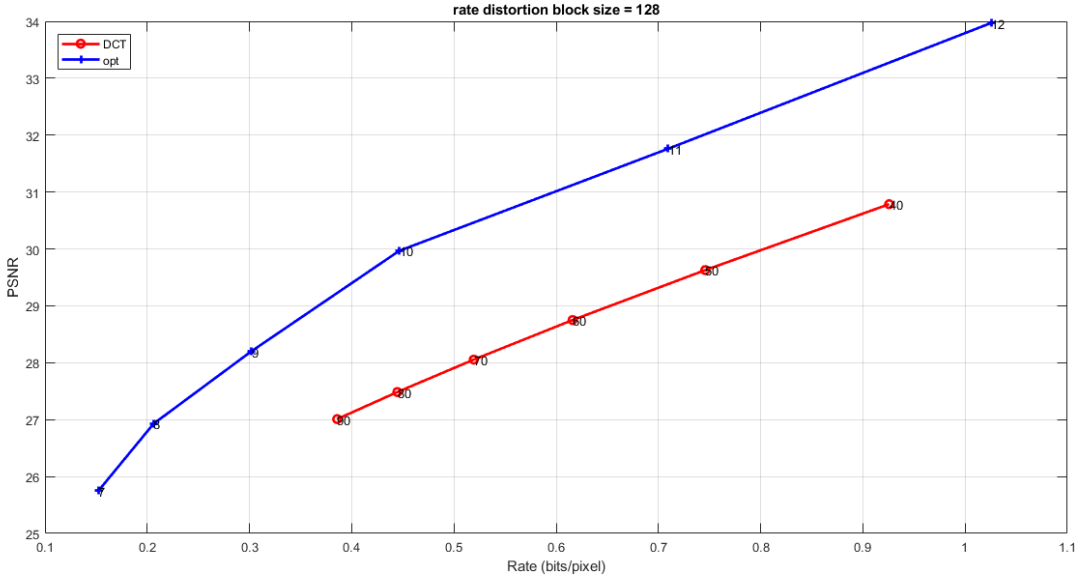

Figura 3.29

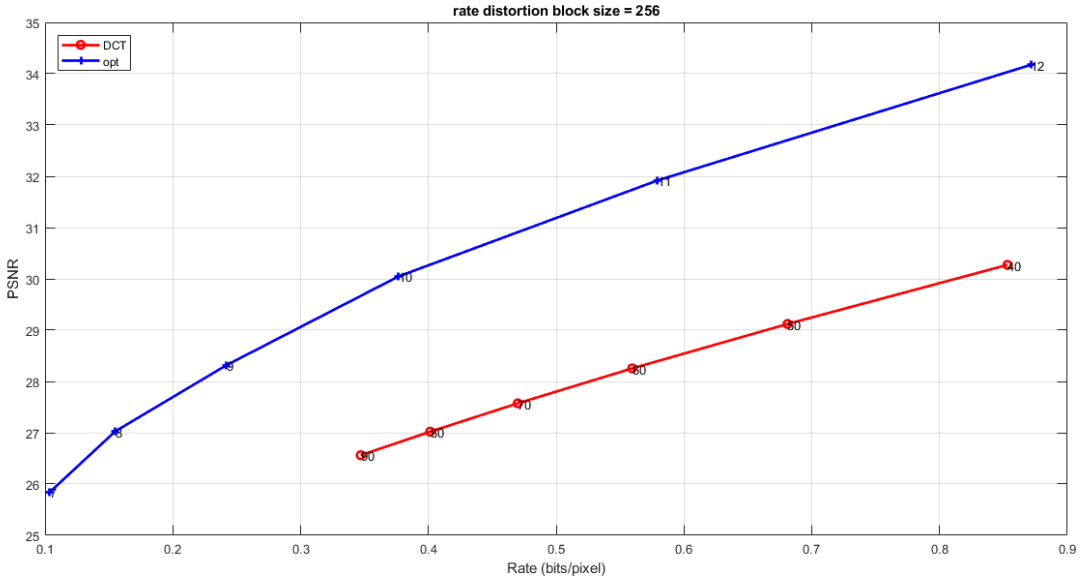

Figura 3.30

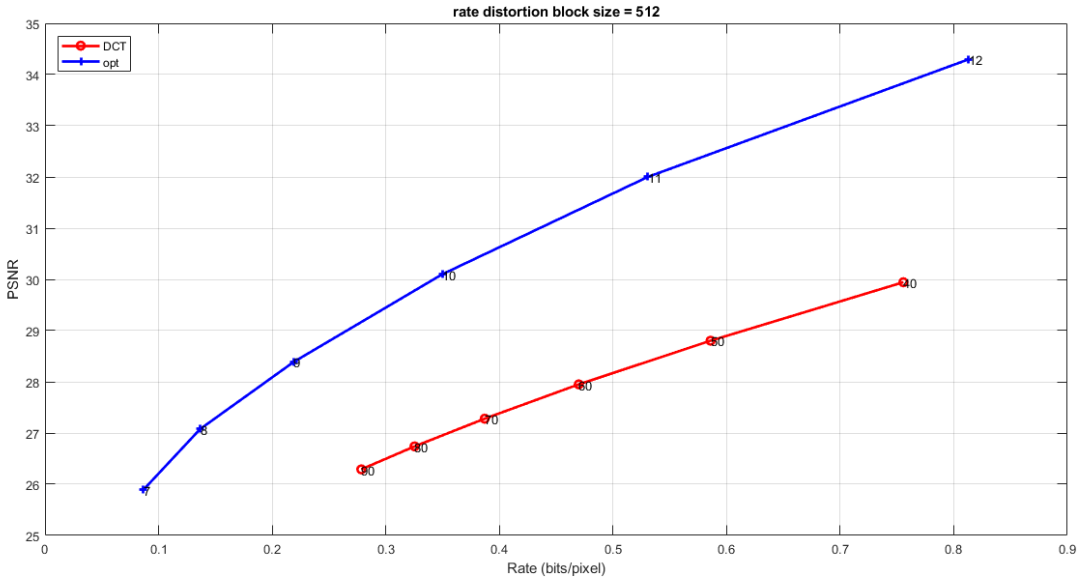

Figura 3.31

3.4 Risultati con immagini HD e 4K

Si è voluto fare alcuni test per valutare le prestazioni anche nel caso di immagini in HD e 4K non partizionate. Il training dell'architettura è sempre fatto su le immagini reperibili al link [4].

3.4.1 Immagine HD "run"

L'immagine 3.32 è un'immagine in HD di un evento sportivo. I valori ottenuti dalla DCT nell'elaborazione di questa immagine sono 1.1 di rate e 34 di PSNR. Mentre la ricostruita dalle trasformate ottimizzate ottiene 0.3 di rate e 30 di PSNR. Quindi, la DCT pur esibendo una qualità leggermente migliore presenta un rate di tre volte più alto delle trasformate ottimizzate. Le immagini 3.33, 3.34 e 3.35, mostrano rispettivamente un ritaglio di 256x256 pixel dell'immagine originale, della ricostruita con la DCT e con le trasformate ottimizzate.

Figura 3.32: originale

Figura 3.33: originale

Figura 3.34: DCT con step di quantizzazione 100

Figura 3.35: Trasformate ottimizzate con lambda 12

3.4.2 Immagine 4K "football"

L'immagine in 4K 3.36 è stata ripresa durante un match di calcio. Sono state ripetute le stesse elaborazioni dell'HD. In questo caso hanno fornito per la DCT un rate di 2 e un PSNR di 44. Mentre per le trasformate ottimizzate i valori di rate e PSNR sono rispettivamente 0.2 e 40. Come nel caso precedente la DCT ha un PSNR più alto, ma le trasformate ottimizzate ottengono un valore di rate molto più basso, nonostante la ricostruita non sia sostanzialmente peggiore della ricostruita della DCT. Le immagini 3.36, 3.37 e 3.39 sono delle porzioni di 584x404

pixel, rispettivamente dell'immagine originale, della ricostruita con la DCT e della ricostruita con le trasformate ottimizzate.

Figura 3.36: originale

Figura 3.37: originale

Figura 3.38: DCT con step di quantizzazione 100

Figura 3.39: Trasformate ottimizzate con lambda 12

Capitolo 4

Conclusioni e sviluppi futuri

Una prima considerazione da fare alla luce dei risultati nel capitolo 3, è che il set di trasformate ottimizzate ottiene performance migliori quando si considerano immagini non partizionate, che è piuttosto comprensibile dato che il training è stato fatto su immagini intere. È evidente comunque che le trasformate ottimizzate ottengono risultati migliori della DCT, infatti a parità di rate le immagini compresse hanno un PSNR maggiore e a parità di PSNR il rate è minore. Bisogna anche notare che le immagini ricostruite considerando l'immagine originale completa non presentano artefatti di blocchettizzazione, come nel caso della DCT. Quindi, nonostante le criticità di cui si parlerà in seguito con le possibili soluzioni, si può confermare ciò che ci si era chiesti all'inizio di questo lavoro, cioè che trasformate di tipo learned apprese mediante NNs sono una valida alternativa, nell'ottimizzazione del trade-off per la compressione immagini a trasformate, fin ora sempre adottate dagli standard, come la DCT.

Un ulteriore passo nell'analisi potrebbe essere ripetere gli esperimenti considerando anche la componente cromatica, (anche se negli esperimenti in HD e 4K sono state già usate un paio di immagini a colori), dato che si è riusciti a modificare il codice Matlab per immagini in scala di grigi anche per immagini a colori (si veda il paragrafo 2.3.2). Ciò che ci si dovrebbe aspettare da questo tipo di analisi sono risultati simili a quelli ottenuti considerando solo la componente luma. Se si volesse anche fare un'analisi per alti bit-rate, probabilmente l'approccio migliore sarebbe cambiare linguaggio di programmazione usando Python, per motivi di praticità. Questo perché non sono disponibili i parametri della CNN per alti bit-rate e quindi dovendo fare un nuovo training è più agevole utilizzare il codice per il training già disponibile per Python che realizzarlo da capo con Matlab. In questo caso per ottenere un modello che tratti alti bit-rate bisogna usare un numero di filtri pari a 256.

Riprendendo i risultati ottenuti, si può notare che quando le immagini sono partizionate, le trasformate ottimizzate, presentano due criticità. La prima è l'aumento notevole del rate a parità di PSNR, rispetto a quando si considera l'immagine completa. Che potrebbe essere risolta realizzando un training su patch di immagini e non su immagini intere, rivedendo opportunamente il numero di filtri. La seconda è legata alla visualizzazione dell'immagine, in cui sono decisamente evidenti i blocchi con cui si partiziona l'immagine, un esempio è mostrato in figura 4.1. In questo caso si potrebbero applicare tecniche di deblocking solamente sui i bordi dei blocchi, poiché si sa già a priori in quali pixel ci saranno delle discontinuità di luminosità.

Figura 4.1: Ricostruita di "girl" con 'blockSize' = 32

La ragione per cui si è voluto in ogni caso partizionare le immagini anche nel caso delle trasformate ottimizzate è dovuta alla naturale evoluzione di un sistema per comprimere immagini, ossia un sistema per comprimere video. Ad esempio, in [16] è proposto un nuovo sistema di compressione video con Machine Learning per bassa latenza, con performance migliori degli standard attuali.

Bibliografia

[1] J. Ballé. «Efficient Nonlinear Transforms for Lossy Image Compression». In: *:arXiv:1802.00847v2* (2018).

[2] M. Carandini e J. D. Headger. «Normalization as a canonical neural computation». In: *Nature Reviewes Neuroscience 13. DOI: 10.1038/nrn3136.* (2012).

[3] J. et al. Deng. «ImageNet: a large-scale hierarchical image database». In: *IEEE Conf. on Computer Vision and Pattern Recognition. DOI : 10.1038/nrn3136.* (2009).

[4] *End-to-end Optimized Image Compression*. 2017. URL: http://www.cns.nyu.edu/~lcv/iclr2017/.

[5] C. Pang J. Dai F. Zou O. C. Au. «Enhanced Intra Mode Dependent Directional Transform». In: *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference, p. 632-635.* (2010).

[6] G. Fracastoro e E. Magli. «Steerable Discrete Cosine Transform». In: *IEEE Transactions on Image Processing, vol. 26 , pp. 303-313.* (2015).

[7] R. M. Gray e D. L. Neuhoff. «Quantization». In: *IEEE Transactions on Information Theory 44.6. DOI : 10.1109/18.720541.* (1998).

[8] J. D. Headger. «Normalization of cell responses in cat striate cortex». In: *Visual Neuroscience 9.2. DOI: 10.1017/S0952523800009640.* (1992).

[9] E. P. Simoncelli J. Ballé V. Laparra. «Density Modeling of Images Using a Generalized Normalization Transformation». In: *arXiv e-prints. Presented at the 4th Int. Conf. for Learning Representations, 2016, arXiv: 1511.06281.* (2015).

[10] E. P. Simoncelli J. Ballé V. Laparra. «End-to-end optimization of nonlinear transform codes for perceptual quality». In: *arXiv e-prints. Accepted for presentation at the 32nd Picture Coding Symposium. arXiv:1607.05006.* (2016).

[11] E. P. Simoncelli J. Ballé V. Laparra. «End-to-end Optimized Image Compression». In: *5th International Conference on Learning Representations.* (2016).

[12] Sayood Khalid. *Introduction to data compression*. fourth edition. Morgan Kaufmann, 2012.

[13] D. P. Kingma e M. Welling. «Auto-Encoding Variational Bayes». In: *arXiv e-prints. arXiv: 1312.6114.* (2014).

[14] S. Lyu. «Divisive Normalization: Justification and Effectivness as Efficient Coding Transform». In: *Advances in Neural Information Processing Systems 10, pp. 815-821.* (2010).

[15] Nielsen Michael. *Neural Networks and Deep Learning*. A cura di Elsievier. 2017. URL: http://neuralnetworksanddeeplearning.com/.

[16] C. Lew S. Branson A. G. Anderson L. Bourdev O. Rippel S. Nair. «Learned Video Compression». In: *arXiv:1811.06981v1* (2018).

[17] O. Shwartz e E. P. Simoncelli. «Natural signal statistics and signal sensory gain control». In: *Nature Neuroscience 4.8. DOI: 10.1038/90526.* (2001).

[18] F. Sinze e M. Bethge. «What Is the Limit of Redundancy Reduction with Divisive Normalization?» In: *Neural Computation 25.11. DOI: 10.1162/NECO_a_00505.* (2013).