

Metodi del calcolo scientifico

Progetto 2

Authors of the paper:

Nicolò Nicholas Zagami 829888

Pietro Bressan 852260

A.A. 2023-2024

Indice

1	Introduzione	2
1.1	La DCT nell'elaborazione dei segnali	2
1.2	Scopo dell'analisi	3
2	Prima parte	4
2.1	Implementazione	4
2.2	Risultati	7
3	Seconda parte	9
3.1	Implementazione	9
3.2	Risultati	10
4	Conclusioni	14

Capitolo 1

Introduzione

1.1 La DCT nell'elaborazione dei segnali

Nell'ambito della gestione e della compressione dei dati, uno strumento molto potente è la Discrete Cosine Transform. Si tratta di una procedura ricavata a partire dalla Trasformata di Fourier, che ne semplifica notevolmente il calcolo per via delle assunzioni fatte sulla funzione che si sta studiando.

Applicando questa trasformazione ad un segnale, inteso come una sequenza ordinata di valori discreti (quindi sia esso un'immagine piuttosto che un video o un audio), si è in grado di "estrapolare" una serie di descrittori (coefficienti) che ne approssimano l'andamento. Ciascuno dei coefficienti è associato ad una particolare frequenza, per cui l'insieme di essi ci dice quali frequenze concorrono nel formare il segnale completo e con quale intensità si presentano. Un aspetto fondamentale è che le frequenze impiegate dalla trasformata forniscono informazione sul segnale a livello globale e non puntuale; ciò significa che tecnicamente ogni descrittore prodotto dalla DCT è utile per modellare la funzione nella sua interezza.

Successivamente attraverso la Inverse Discrete Cosine Transform (IDCT), l'operazione inversa alla DCT, siamo in grado di utilizzare questi coefficienti insieme con le corrispettive frequenze per risalire al segnale di partenza, senza perdita di dati alcuna. Esiste dunque una corrispondenza univoca tra il segnale e la sua trasformata dei coseni (di questo però non ci dobbiamo stupire, dal momento che a livello algebrico la DCT esegue a tutti gli effetti una conversione di base).

Il punto di forza della DCT risiede proprio nella natura dei descrittori: infatti per via delle proprietà di cui gode la trasformazione è possibile risalire al segnale originale (o per meglio dire ad una sua ragionevole approssimazione) utilizzando solo un insieme ridotto di questi valori. Questa particolarità rende possibile comprimere un segnale semplicemente scartando alcune delle sue componenti, dopo aver applicato la Discrete Cosine Transform. Chiaramente questo troncamento dei coefficienti ha delle ripercussioni sulla funzione che viene ricostruita: i valori che vengono scartati rappresentano delle frequenze che non sono più utilizzate nella ricostruzione, per cui il segnale ottenuto applicando la IDCT non sarà perfettamente identico a quello di partenza. Escludendo acuni dei valori approssimati si ha inevitabilmente una perdita di informazione dovuta a delle frequenze mancanti, che può essere più o meno notevole a seconda di quanti coefficienti decidiamo di eliminare in fase di compressione.

Tuttavia è possibile notare come nella maggior parte dei casi la IDCT permette di ottenere un segnale ricostruito soddisfacente, anche a fronte di una grande quantità di coefficienti scartati. A seconda dell'ambito di applicazione, la perdita di dati o di qualità nel segnale può essere contemplata a patto che il contenuto sia ancora identificabile.

1.2 Scopo dell'analisi

Nel corso della relazione ci soffermeremo a considerare le potenzialità di questa tecnica di compressione rispetto ad immagini a toni di grigio.

Il primo progetto verterà nel comparare tra di loro i tempi di esecuzione di due diverse implementazioni della DCT e della sua versione bidimensionale: una realizzata dai membri del gruppo ed una offerta da una libreria di calcolo scientifico nota.

Il secondo invece riguarderà la realizzazione di un'applicazione grafica per la visualizzazione dei risultati della compressione. Nella seconda parte, per il calcolo dei coefficienti, verrà utilizzata la libreria di calcolo esterna nella modalità fast.

Per implementare i progetti si è deciso di utilizzare il linguaggio Python.

Capitolo 2

Prima parte

2.1 Implementazione

L'implementazione della compressione di immagini attraverso la DCT è stata realizzata sfruttando le seguenti librerie:

- Libreria Numpy: libreria per la gestione dei vettori e delle matrici su cui eseguire la compressione.
- Libreria Scipy: libreria che implementa la Discrete Cosine Transform e la sua inversa in modalità "fast", utile per il confronto delle tempistiche.

La libreria sviluppata si compone di un unico modulo chiamato customdct che contiene al suo interno 4 funzioni elementari:

1. *compute_dct(values)* : funzione base utilizzata per ritornare i coefficienti corrispondenti alla base dei coseni associati alla funzione discreta (*values*) passata in ingresso. Il calcolo dei coefficienti segue la formula

$$a_k = \frac{v \cdot w^k}{w^k \cdot w^k} = \frac{\sum_{i=1}^N \cos(\pi k \frac{2i+1}{2N}) v_i}{w^k \cdot w^k} \quad (2.1)$$

Dove sappiamo che il prodotto $w_k \cdot w_k$ equivale a:

$$\begin{cases} N & \text{se } k = 0 \\ \frac{N}{2} & \text{se } k \neq 0 \end{cases} \quad (2.2)$$

Il codice della funzione è:

```
1 def compute_dct(values):
2     num_of_values = len(values)
3     steps = np.array([(2 * step + 1) / (2 * num_of_values) * np.pi for
4                       step in range(num_of_values)])
5     cosine_base = np.array([np.cos(steps * freq) for freq in range(
6                               num_of_values)])
7     cos_values = np.zeros(num_of_values)
8     cos_values[0] = np.dot(values, cosine_base[0]) / num_of_values
9     cos_values[1:num_of_values] = [np.dot(values, cosine_base[i]) * 2 /
10                                    num_of_values for i in range(1, num_of_values)]
11    return cos_values
```

[righe 2-4] Per prima cosa andiamo a costruire la base dei coseni che utilizzeremo per ricavare i valori. Per farlo identifichiamo degli "steps", ovvero dei passi, sui quali andiamo a valutare la funzione coseno. Rispetto alla formula riportata sopra questo vuol dire calcolare i vettori $\cos(\pi k \frac{2i+1}{2N})$.

[righe 5-8] Successivamente sfruttiamo la formula per calcolare i coefficienti e ritorniamo il risultato. Notare che distinguiamo i casi in base all'indice del vettore risultato, come indicato dalla formula generale.

2. *compute_idct(values)*: Questa funzione viene utilizzata per applicare la Inverse Discrete Cosine Transform ad un vettore di coefficienti, di modo da ottenere i valori originali relativi alla base canonica. In questo caso possiamo eseguire questo calcolo sfruttando la definizione di vettore generato da una base, utilizzando i valori passati in ingresso si ha:

$$v = \sum_{i=1}^N a_k w^k \quad (2.3)$$

Dove v rappresenta il vettore target, w^k indica il $k - \text{esimo}$ vettore nella base dei coseni, N è il numero di elementi presenti in v (o analogamente il numero di valori presenti in $values$) e a_k è il $k - \text{esimo}$ coefficiente presente nel vettore passato come parametro alla funzione ($values$).

Il codice che effettua questa operazione è il seguente:

```

1 def compute_idct(values):
2     num_of_values = len(values)
3     canonical_values = np.zeros(num_of_values)
4     steps = np.array([(2 * step + 1) / (2 * num_of_values) * np.pi
5                       for step in range(num_of_values)])
6     cosine_base = np.array([np.cos(steps * freq) for freq in range(
7         num_of_values)])
8     for i in range(num_of_values):
9         canonical_values += values[i]*cosine_base[i]
10    return canonical_values

```

[righe 2-5] Come per la Discrete Cosine Transform ricaviamo i vettori della base canonica che ci aiuteranno ad ottenere il vettore target. Il procedimento è lo stesso, si calcolano degli steps e si valuta la funzione coseno su di essi.

[righe 6-8] Cicliamo su tutti gli elementi presenti in $values$ e li moltiplichiamo per i corrispettivi valori nella base dei coseni, realizzando così la sommatoria in formula. Il vettore target altro non sarà che la combinazione lineare di questi vettori. Infine ritorniamo il vettore target.

3. *compute_dct2(encoded_matrix, win_size = 8)*: Questa funzione si occupa di applicare la DCT su immagini a toni di grigio. L'immagine viene suddivisa a blocchi della dimensione $win_size \times win_size$.

Andiamo ad analizzare passo per passo gli aspetti implementativi di questo metodo:

- **Efficienza nel calcolo:** Per rendere più efficiente il calcolo dei coefficienti, i valori presenti nella matrice di ingresso sono decrementati di 128. In questo modo la media dei valori associati ad ogni pixel dell'immagine è 0.

```

1 approx_matrix = (matrix.astype("float64") - 128)

```

- **Padding:** Come detto in precedenza l'immagine viene suddivisa a blocchi di dimensione 8x8; bisogna quindi rendere la matrice divisibile per 8 qualora già non lo fosse, per fare questo aggiungiamo una "cornice". Alla fine del metodo, prima di ritornare la matrice la riportiamo alla sua dimensione originale.

```

1 # Inserimento del padding (se necessario)
2 rows, cols = matrix.shape
3 padding_rows = win_size*ceil(rows/win_size)-rows
4 padding_cols = win_size*ceil(cols/win_size)-cols
5 if(padding_cols != 0 or padding_rows != 0):
6     approx_matrix = np.pad(approx_matrix, ((0,padding_rows), (0,
7         padding_cols)), mode='edge')
8
# ... prima di ritornare il risultato rimuovo il padding riportando
# la matrice alla sua dimensione originale.
9 approx_matrix = approx_matrix[:rows, :cols]

```

- **Applicazione della DCT bidimensionale:** Partendo dalle coordinate (0,0) dell'immagine, discendo tutti i blocchi eseguendo su uno ad uno la DCT. Occorre però fare delle osservazioni nel caso bidimensionale: infatti se nel caso dei vettori ci bastava applicare la DCT una sola volta, con immagini (matrici) l'operazione viene svolta più volte. In particolare per ogni blocco analizzato, la DCT viene eseguita prima sulle righe e successivamente sulle colonne dello stesso blocco (il procedimento può anche essere scambiato valutando prima le colonne e poi le righe).

```

1 #Scorro l'immagine a blocchi 8x8 partendo da (0,0) e
2     discendendo l'immagine
3 for row in range(0, approx_matrix.shape[0], win_size):
4     for col in range(0, approx_matrix.shape[1], win_size):
5         #Recupero una porzione (finestra) dell'immagine
6         for sample_col in range(col, col+win_size):
7             approx_matrix[row:row+win_size, sample_col] =
8                 compute_dct(approx_matrix[row:row+win_size,
9                     sample_col])
10            for sample_row in range(row, row+win_size):
11                approx_matrix[sample_row, col:col+win_size] =
12                    compute_dct(approx_matrix[sample_row, col:col+
13                        win_size])

```

4. *compute_idct2(encoded_matrix, win_size = 8):* Come nel caso monodimensionale esiste una funzione per il calcolo inverso della DCT su immagini a toni di grigio. L'implementazione è pressoché analoga alla DCT eseguita su matrici, in quanto l'immagine è ancora una volta suddivisa in blocchi di dimensione *win_size* x *win_size*, eventualmente bordata, sui quali iterativamente viene calcolata la Inverse Discrete Cosine Transform (eseguendola una volta per colonne e poi per righe).

Dal momento che la funzione restituisce un'immagine visualizzabile, è necessario assicurarsi che i valori ottenuti applicando la IDCT alla matrice siano validi. Per questo motivo l'immagine prima di essere ritornata viene incrementata di 128 (riportando la media dei valori a 128 appunto) e viene approssimata di modo che ogni cella della matrice contenga un valore intero nell'intervallo [0, 255].

2.2 Risultati

Dopo aver presentato l'implementazione della "custom" DTC2 analizziamo ora i tempi di esecuzione rapportati alle grandezze delle matrici utilizzate in input con le due implementazioni; questo esperimento parte dall'assunzione che i tempi relativi alla "custom" DCT2 dovrebbero essere proporzionali a N^3 e $N^2 \log(N)$ per la versione da libreria.

Le matrici sono state costruite a partire dalle immagini (file .bmp) di dimensione variabile utilizzando `np.array()`

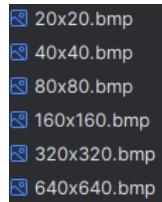


Figura 2.1: Immagini su cui sono state costruite le matrici.

Successivamente misuriamo i tempi con entrambe le implementazioni, i cui risultati sono i seguenti:

Dimensione (NxN)	Tempo custom DCT2 (s)	Tempo SciPy DCT2 (s)
20x20	0.0036676	0.0001599
40x40	0.0094442	0.0000533
80x80	0.0391304	0.0001267
160x160	0.1550706	0.0002462
320x320	0.6121832	0.0008621
640x640	2.3973159	0.0046158

Tabella 2.1: Tempi di esecuzione per DCT2 custom e FFT DCT2

Per visualizzare meglio l'andamento delle due implementazioni abbiamo prodotto un grafico. Inoltre conoscendo l'andamento teorico dei due algoritmi abbiamo deciso di utilizzare `curve_fit`: metodo di **SciPy** in grado di approssimare il modello teorico ai dati ottenuti.

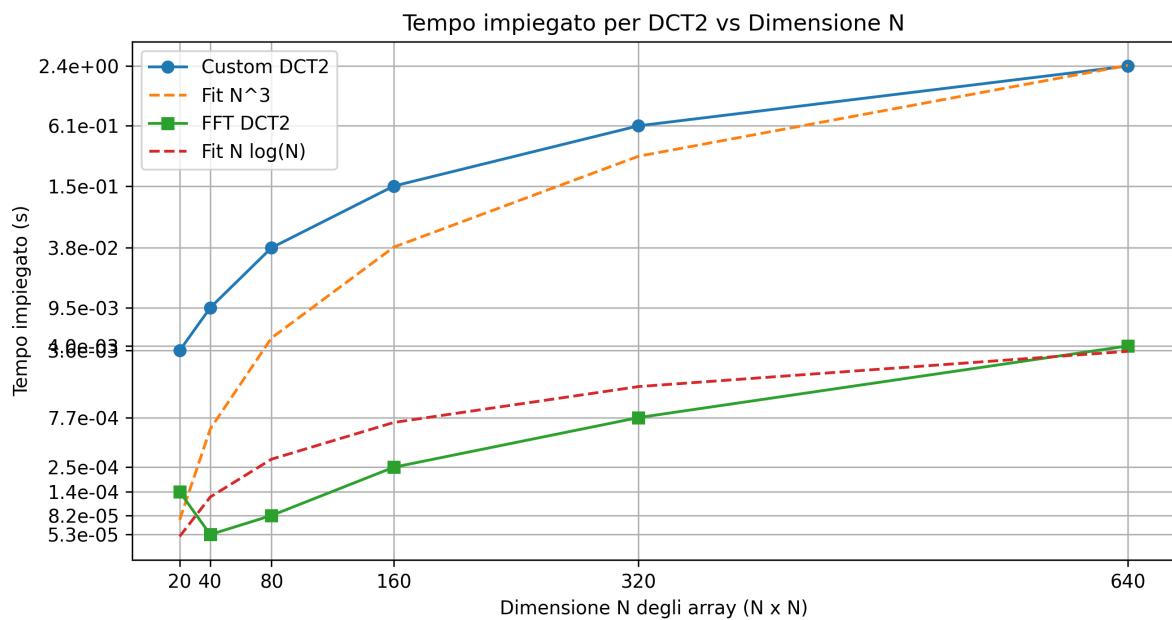


Figura 2.2: DCT2 "Custom" vs DCT2 FFT (libreria).

Sulle ascisse troviamo i valori crescenti di N , sulle ordinate invece i tempi di esecuzione misurati in secondi, dai risultati è possibile apprezzare che le due implementazioni rispettano la loro proporzionalità con le classi di complessità N^3 (per la versione custom) e $N^2 \log(N)$ (per la versione di libreria).

Capitolo 3

Seconda parte

3.1 Implementazione

Per la realizzazione dell'interfaccia grafica sono state utilizzate le seguenti librerie

- Libreria Numpy: libreria per la gestione dei vettori e delle matrici su cui eseguire la compressione.
- Libreria Tkinter: libreria per la creazione e la manipolazione di interfacce grafiche.
- Libreria Scipy: libreria che implementa la Discrete Cosine Transform e la sua inversa in modalità "fast".
- Libreria PIL: libreria per la gestione di immagini.

L'interfaccia si presenta con uno stile minimale

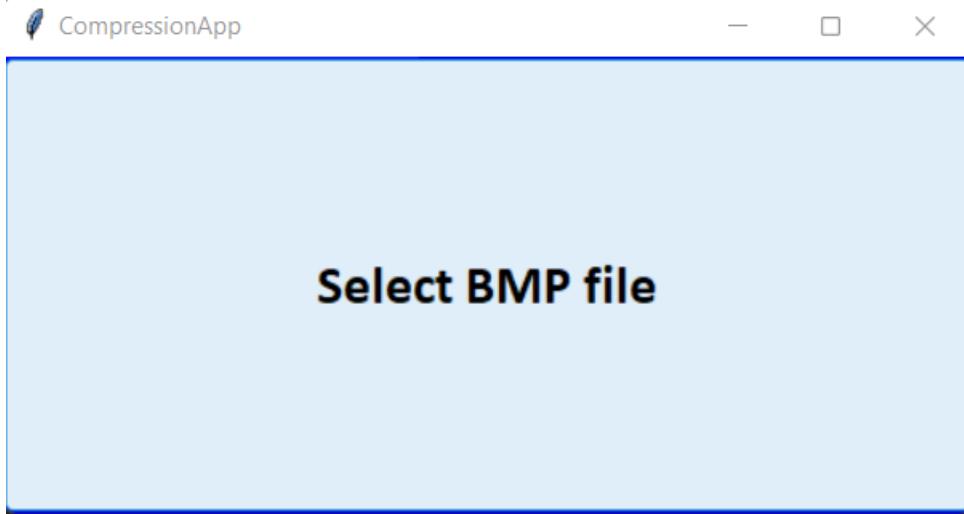
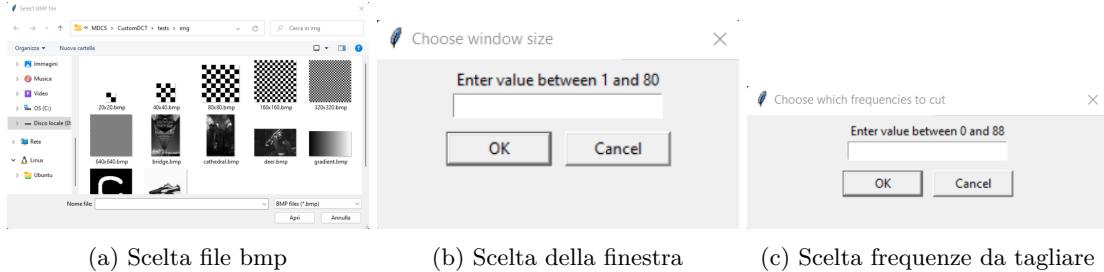


Figura 3.1: Schermata di apertura.

Da qui possiamo procedere tramite un' apposita dialog box e selezionare un'immagine a toni di grigio (estensione .bmp). Una volta scelta l'immagine il programma ci chiederà di inserire due valori: una dimensione della finestra in cui verrà suddivisa l'immagine e un intero che rappresenta le frequenze che verranno tagliate.



(a) Scelta file bmp

(b) Scelta della finestra

(c) Scelta frequenze da tagliare

Successivamente il programma prende in ingresso l’immagine a toni di grigio e la converte in una matrice bidimensionale a valori interi. Sulla matrice viene eseguita la Discrete Cosine Transform in modo iterativo: l’immagine viene scomposta in blocchi della dimensione della finestra selezionata. Per ciascuno dei blocchi viene chiamata la funzione *dct* dal modulo *scipy.fft* sia per le righe che per le colonne. Il blocco generato viene troncato di modo da annullare i coefficienti (o le frequenze) per cui la somma degli indici supera il valore di soglia selezionato. In questo modo possiamo volontariamente causare una perdita di dettagli nella figura originale.

L’immagine viene ricostruita applicando questa volta la IDCT nuovamente a tutti i blocchi. Avendo azzerato alcuni dei valori in fase di compressione, ci possiamo aspettare che l’operazione inversa peggiori qualità della nostra immagine, introducendo i fenomeni tipici associabili con questo tipo di operazione.

Per ultimo l’immagine originale e l’immagine su cui abbiamo effettuato la compressione e la decompressione sono mostrate l’una di fianco a l’altra. Qualora le due immagini poste vicino dovessero superare la dimensione dello schermo, queste vengono ridimensionate di modo da poterle osservare i risultati.

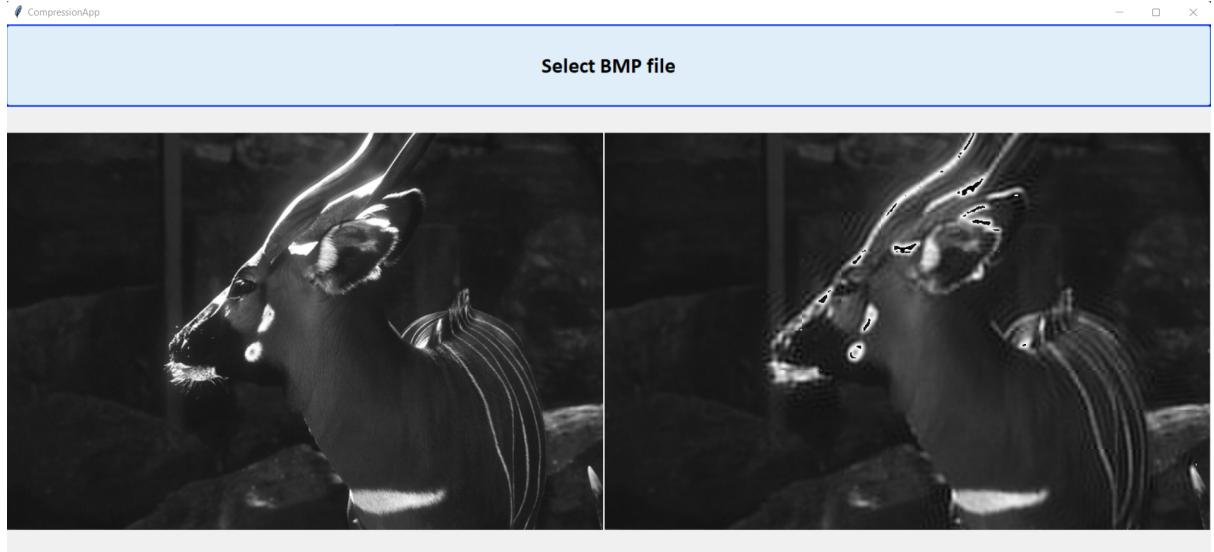


Figura 3.3: A sinistra l’immagine originale (1011x661 pixels), a destra l’output prodotto selezionando una finestra di dimensione 50 e un valore di taglio delle frequenze pari a 15.

3.2 Risultati

Grazie all’interfaccia grafica riusciamo ad apprezzare la bontà della compressione eseguita dalla Discrete Cosine Transfrom. Particolarmente interessante è osservare cosa accade alla approssimazione di un’immagine al variare della dimensione della finestra e delle frequenze da tagliare, due parametri che concorrono nel definire la precisione con cui viene ricostruita la figura.

Infatti in un generico blocco di dimensione fissata, è contenuta una variabile quantità di informazione. Possiamo pensare all'informazione associata con un blocco come alla presenza di bordi al suo interno o a cambiamenti nella tonalità di grigio. Per informazione intendiamo qualsiasi elemento distinto presente nel blocco che richiede una sua rappresentazione, maggiori sono i dettagli contenuti in una finestra, maggiore sarà l'informazione associata ad essa. Viene naturale pensare che all'incrementare della dimensione con cui viene discretizzata l'immagine, dipenderà un aumento dell'informazione media contenuta per blocco.



Figura 3.4: Ingrandimento di due blocchi di dimensione 125x125. Il primo riquadro a sinistra si presenta con una tinta uniforme, il secondo a destra invece cattura diversi elementi legati all'occhio e al manto dell'animale.

Quando viene applicata la DCT alle singole porzioni, i coefficienti relativi ai pixel sono codificati di modo che i loro valori non si riferiscono più ad una gradazione di grigio, bensì all'intensità di una particolare frequenza. Andando ad azzerare alcune delle celle in un blocco di fatto si rimuovono dei dettagli che erano originariamente presenti. L'effetto prodotto è più o meno visibile a seconda dal numero di elementi distinguibili che esso contiene.

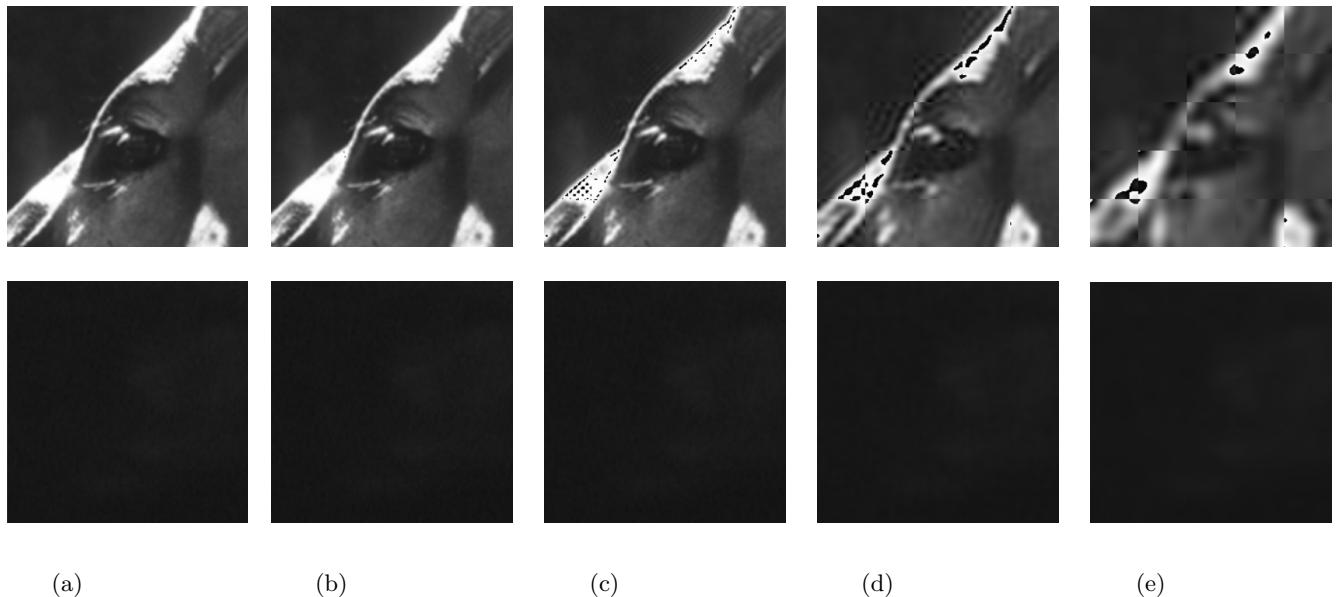


Figura 3.5: La compressione eseguita su due porzioni dell'immagine, con una dimensione della finestra pari al 20%. Da sinistra, la porzione originale e le seguenti compressioni eseguite tagliando di volta in volta la metà delle frequenze totali.

Dunque vi è un trade-off tra la dimensione dei blocchi con cui suddividiamo l'immagine e il numero di frequenze che possiamo eliminare, a patto di mantenere più o meno la stessa qualità dei dettagli.

Possiamo accorgerci di una scarsa compressione quando questa comporta la generazione di artefatti all'interno della figura, ovvero di elementi originariamente assenti nell'immagine ed introdotti a causa del taglio di frequenze. Tra questi i più noti sono legati alla comparsa di "quadrati" e al fenomeno di Gibbs.

Se discretizzando la matrice si azzerano una grande quantità di coefficienti in proporzione all'ampiezza della finestra, in fase di decompressione dell'immagine la IDCT avrà a disposizione pochi valori per ricostruire ciascun blocco. Solo poche frequenze saranno dunque impiegate per la rigenerazione della finestra, con conseguente perdita di omogeneità tra blocchi adiacenti. La figura che si ottiene sarà caratterizzata dalla presenza di una griglia.



Figura 3.6: Dettaglio con la testa dell'animale. Le sezioni con una maggiore quantità di elementi sono maggiormente influenzate dalla presenza di quadrati.

Il fenomeno di Gibbs invece avviene quando in una porzione di immagine vi è un cambio repentino nei valori assunti dai singoli pixel. Tipicamente lo si riscontra in prossima di bordi o al corrispondere di un salto nella tonalità del livello di grigio tra celle vicine. In quell'area la DCT fatica ad approssimare correttamente l'informazione contenuta e produce una serie di onde che diventano visibili una volta eseguita l'operazione inversa. Anche in questo caso la dimensione della finestra e il numero di frequenze tagliate giocano un ruolo importante nel definire quanto il fenomeno sia percepibile.



Figura 3.7: Immagine di una scarpa su cui viene eseguita la DCT. Notare come le zone dove vi è un passaggio repentino dal bianco a nero sono attorniate da delle linee che seguono la sagoma della figura. Lo stesso non si può dire per la regione del tacco che invece ha una gradazione di colore simile allo sfondo.

Capitolo 4

Conclusioni

Il documento analizza le implementazioni della Trasformata Discreta del Coseno (DCT) per la compressione di immagini, evidenziando le performance computazionali e i compromessi necessari tra compressione e qualità dell'immagine.

- **FFT DCT vs "Custom" DCT:** Le implementazioni della DCT studiate nel documento mostrano significative differenze nelle performance computazionali. In particolare, l'uso di librerie ottimizzate come SciPy consente di ottenere risultati molto più veloci rispetto alle implementazioni personalizzate, specialmente quando si lavora con grandi matrici.
- **Trade-off tra Compressione e Qualità:** per garantire che la perdita di qualità sia minimizzata, mantenendo al contempo un livello di compressione accettabile è necessario trovare il giusto compromesso con i parametri. Tuttavia, una compressione eccessiva può introdurre artefatti visibili come il fenomeno di Gibbs e la comparsa di "quadrati".
- **Parametri per la Qualità della Compressione:** Infine, il documento sottolinea l'importanza di alcuni parametri critici che influenzano la qualità della compressione: Due parametri fondamentali sono la dimensione dei blocchi in cui viene suddivisa l'immagine e il numero di frequenze eliminate. Blocchi di dimensioni maggiori e un numero minore di frequenze eliminate tendono a migliorare la qualità dell'immagine ricostruita, poiché preservano più informazioni dell'immagine originale; tuttavia, questo riduce il livello di compressione ottenibile.