Metodi del calcolo scientifico

Compressione di immagini con DCT

Volpato Mattia 866316 *
Stefano Andreotti 851596 †

Appello di Giugno 2024

*m.volpato4@campus.unimib.it
†s.andreotti7@campus.unimib.it

Contents

1 Prima parte - DCT2 benchmark

1.1 Obiettivo

In questa prima parte dell'elaborato si richiede di fornire una propria implementazione dell'algoritmo ***DCT2*** **(Discrete Cosine Transform 2-dimensional)** in ambiente **open source** e di confrontare le performance ottenute (in termini di **tempo di esecuzione**) con un'implementazione fornita da una libreria dell'ambiente open source.

A tal fine si è scelto di utilizzare il linguaggio di programmazione **python** e le librerie ***scipy***, dedicata al **calcolo scientifico**, e **numpy**, descritte nel dettaglio nel paragrafo 1.2.

Successivamente, al fine di determinare l'impatto di implementazioni più o meno efficienti sui **tempi di esecuzione**, sono state fornite tre diverse implementazioni personali dell'algoritmo ***DCT2***, trattate nel paragrafo 1.3.

Come indicato nelle specifiche, per semplicità verranno trattate solo matrici quadrate.

1.2 Libreria

La libreria **scipy** mette a disposizione il modulo **fftpack**, contenente una serie di funzioni che permettono di calcolare diverse varianti della *Discrete Cosine Transform*.

In particolare, nella stesura di questo elaborato è stata utilizzata unicamente la funzione *dctn*, che permette di specificare uno o più assi lungo i quali eseguire la *DCT*. Per ottenere lo scaling indicato nel progetto sono stati usati i parametri:

```
        fft.dctn(x, axes=(0, 1), type=2, norm='ortho')
```

Implementando la *versione fast* dell'algoritmo, ci si aspetta di ottenere una complessità temporale dell'ordine $O(n^2 log(n))$.

1.3 Implementazioni

Come già indicato, al fine di analizzare le differenze di perfomance derivanti dall'utilizzo di implementazioni che variano per efficienza, sono state fornite **tre diverse** versioni della *DCT* 'fatta in casa':

- ***dct_naive***: implementazione diretta della definizione di *DCT* in python, utilizzando due cicli *for* innestati

```
def dct_naive(x:np.array) -> np.array:
    N = len(x)
    sqrt_N, sqrt_2 = np.sqrt(N), np.sqrt(2)
    dct_x = np.zeros(N, dtype=np.float64)
    coeff = np.pi / (2 * N)

    for k in range(N):
        a_k = 0.0
        coeff_k = coeff * k
        for i, x_i in enumerate(x):
            a_k += np.cos(coeff_k * (2 * i + 1)) * x_i
        dct_x[k] = a_k / sqrt_N * sqrt_2

    dct_x[0] /= sqrt_2

    return dct_x
```

- ***dct_outer***: sfrutta al massimo le **operazioni tra matrici** ottimizzate messe a disposizione da ***numpy***, pre-computando tutti i coefficienti

$$cos(\pi k \frac{2i+1}{2N}), 0 \leq k, i \leq N-1$$

attraverso il ***prodotto esterno*** $\otimes$ (*riga 8*)

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \otimes \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \underline{v_1} \otimes \underline{v_2} = \underline{v_1} \cdot \underline{v_2}^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 & b_2 & \dots & b_m \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \dots & a_1 \cdot b_m \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \dots & a_2 \cdot b_m \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \dots & a_n \cdot b_m \end{bmatrix} \tag{1}$$

e successivamente calcolando gli a_k tramite il ***prodotto vettore-matrice*** (*riga 10*)

```
1  def dct_outer(x:np.array) -> np.array:
2      N = len(x)
3      k = np.arange(N)
4      n = np.arange(N)
5  
6      const_coeff = np.pi / (2 * N)
7      var_coeff = np.outer(k, 2 * n + 1)
8      transform_matrix = np.cos(const_coeff * var_coeff)
9  
10     result = transform_matrix @ x
11 
12     result[0] *= np.sqrt(1 / N)
13     result[1:] *= np.sqrt(2 / N)
14     return result
```

Questo approccio porta al dover salvare una matrice addizionale di dimensione $(N \times N)$, portando la complessità spaziale a $\theta(N^2)$.

- ***dct_cpp***: implementazione diretta della definizione della *DCT* in ***C++***, che rispetto a **python** permette una gestione più efficiente delle **strutture dati** e presenta dei **cicli** molto **più performanti**. Codice disponibile in questa repository.

Per tutte e tre le versioni, la funzione *DCT2* è stata ottenuta applicando la *DCT* prima per righe e poi per colonne come segue:

```
def dct2(x:np.array, dct_functor:callable) -> np.array:
    dct2_temp = np.apply_along_axis(dct_functor, axis=0, arr=x)
    dct2_x = np.apply_along_axis(dct_functor, axis=1, arr=dct2_temp)
    return dct2_x
```

Ottenendo:

- ***dct2_naive***:

```
dct2_naive = lambda x : dct2(x, dct_naive)
```

- ***dct2_outer***:

```
dct2_outer = lambda x : dct2(x, dct_outer)
```

- ***dct2_cpp***

- ***dct2_lib***, l'implementazione della libreria:

```
dct2_lib = lambda x : fft.dctn(x, axes=(0, 1), type=2, norm='ortho')
```

Per tutte e tre le implementazioni *custom* ci si aspetta una **complessità temporale** $T(n) = \theta(n^3)$. Tutto il codice è disponibile in questa repository.

1.4 Benchmark

1.4.1 Matrici e modalità di utilizzo

Il **benchmark** è stato eseguito utilizzando matrici $N \times N$ di dimensione crescente, con il fine di individuare un *andamento asintotico regolare* nei **tempi di esecuzione**.

Partendo da una dimensione minima di 25×25, si sono ripetutamente raddoppiate le singole dimensioni (e quindi quadruplicate le dimensioni totali delle matrici), fermandosi quando le diverse implementazioni iniziavano a richidere troppo tempo.

Tutte le matrici sono state inizializzate casualmente con valori interi compresi tra 0 e 255, in maniera da simulare uno dei principali scenari applicativi della *DCT2*; inoltre, al fine di ridurre la varianza dei risultati, tutte le esecuzioni sono state ripetutute tre volte e se ne è considerato il **valore medio**.

1.4.2 Risultati

A seguire vengono riportati i risultati ottenuti, prima in forma tabellare e poi come grafico (in scala logaritmica); i **tempi** riportati *in rosso* nella tabella (e *tratteggiati* nel grafico) sono solo stimati, utilizzando un **modello di regressione polinomiale** (per un polinomio di grado 3).

Times (s)				
N	***dct_naive***	***dct_outer***	***dct_cpp***	***dct_lib***
25	0.0499	0.0023	0.0003	0.0001
50	0.2557	0.0065	0.0037	0.0001
100	2.0294	0.0416	0.0267	0.0002
200	14.174	0.1899	0.1820	0.0010
400	98.035	1.8742	1.4267	0.0020
800	705.54	14.671	11.636	0.0070
1600	$5.3 \cdot 10^3$	113.59	95.239	0.0350
3200	$4.1 \cdot 10^4$	887.62	772.96	0.1936
6400	$3.2 \cdot 10^5$	$7.0 \cdot 10^3$	$6.2 \cdot 10^3$	1.1590
12800	$2.5 \cdot 10^6$	$5.6 \cdot 10^4$	$5.0 \cdot 10^4$	5.1695

Table 1: Risultato del benchmark (i tempi in rosso sono solo stimati)

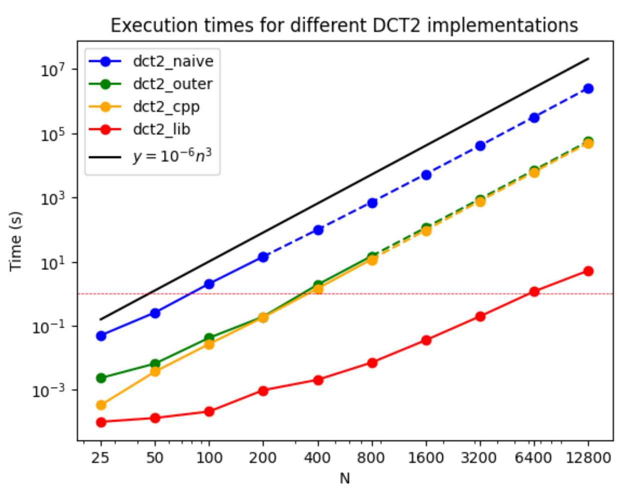

Figure 1: Risultato del benchmark su scala logaritmica (i *tempi tratteggiati* sono solo stimati)

Dal grafico 1 si nota come i risultati della sperimentazione confermino l'attesa teorica di una **complessità**

temporale $T(n) = \theta(n^3)$ per le implementazioni *custom*, mentre l'implementazione *fast* della libreria mostra una capacità di scalare decisamente superiore, sebbene non sia immediato riconoscere l'andamento asintotico.

Confrontando invece solo le tre versioni *custom*, risulta evidente che, a parità di complessità computazionale, le implementazioni ***dct_outer*** e ***dct_cpp*** siano nettamente superiori a ***dct_naive***, al punto da riuscire a trattare matrici *16 volte più grandi* con tempi dagli *stessi ordini di grandezza*.

Inoltre, si evince come sia possibile ottenere ottime prestazioni da un linguaggio di programmazione *interpretato*, orientato alla *portabilità* e alla *semplicità di utilizzo* come **python**: nonostante i limiti dovuti alla sua struttura, è possibile raggiungere prestazioni molto vicine a quelle del **C/C++**, a patto di fare ampio utilizzo di librerie specializzate ed efficienti (e sviluppate in linguaggi performanti) come ***numpy*** (sviluppata in C).

2 Seconda parte - Compressione di immagini

2.1 Obiettivo

In questa seconda parte la richiesta è lo sviluppo di un piccolo *sistema software* che implementi una versione semplificata della *compressione* con *perdità di qualità* **JPEG**, facendo utilizzo della versione fast della ***DCT2***.

Anche in questo caso si è scelto di utilizzare il linguaggio **python** insieme ad un'architettura **MVC** (**Model-View-Controller**), descritta nel dettaglio nella sezione 2.3.

Inoltre, al fine di avvicinarsi il più possibile al vero formato **JPEG**, si è scelto di trattare sia immagini in **toni di grigi** che a colori (**RGB**) e di creare un proprio (banale) *formato di serializzazione* (sezione 2.2);

Infine, sono stati effettuati alcuni esperimenti su immagini in formato **bmp** di varie dimensioni (sezione 2.4).

Tutto il codice è disponibile in questa repository.

2.2 Formato JPUG

Come già detto, si è deciso di definire un formato personalizzato che andasse a simulare il formato **JPEG**, chiamato (in maniera fantasiosa) **JPUG**. Il formato salva banalmente in formato binario e in maniera sequenziale le tre componenti necessarie a ricostruire l'immagine compressa:

- il coefficiente $\boldsymbol{F}$, rappresentante la dimensione dei blocchi in cui è suddivisa l'immagine;
- il coefficiente $\boldsymbol{d}$, rappresentante la prima antidiagonale delle entrate da tagliare;
- il valore delle **entrate** (nella base dei coseni) da mantenere, che vengono linearizzate per righe in un **vettore monodimensionale**, come nel seguente esempio (per $F = 5$ e $d = 4$):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & - \\ a_{21} & a_{22} & a_{23} & - & - \\ a_{31} & a_{32} & - & - & - \\ a_{41} & - & - & - & - \\ - & - & - & - & - \end{bmatrix} \Longrightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{21} & a_{22} & a_{23} & a_{31} & a_{32} & a_{41} \end{bmatrix}$$

Il **numero $\boldsymbol{n}$ di entrate** della matrice **da mantenere** (**dimensione del vettore linearizzato**) sarà dato da:

- Per $0 \leq d \leq F$:
$$n = \sum_{i=0}^{d-1} i + 1 = \sum_{i=1}^{d} i = \frac{d(d+1)}{2} \tag{2}$$
- Per $F < d \leq 2F - 1$:
$$n = \frac{F(F+1)}{2} + \sum_{k=1}^{d-F} F - k \tag{3}$$

  dove

$$\begin{aligned} \sum_{k=1}^{d-F} F - k = \sum_{k=1}^{d-F} F - \sum_{k=1}^{d-F} k = \\ &= F(d-F) - \frac{(d-F)(d-F+1)}{2} = \\ &= Fd - F^2 - \frac{1}{2}d^2 + \frac{1}{2}Fd - \frac{1}{2}d + \frac{1}{2}Fd - \frac{1}{2}F^2 + \frac{1}{2}F = \\ &= -\frac{3}{2}F^2 + \frac{1}{2}F + 2Fd - \frac{1}{2}d^2 - \frac{1}{2}d \end{aligned}$$

e di conseguenza

$$n = \frac{1}{2}F^2 + \frac{1}{2}F - \frac{3}{2}F^2 + \frac{1}{2}F + 2Fd - \frac{1}{2}d^2 - \frac{1}{2}d =$$
$$= 2Fd - \frac{1}{2}d^2 - F^2 - \frac{1}{2}d + F$$

Riassumendo:

$$n(F,d) = \begin{cases} \frac{d(d+1)}{2} & \text{se } 0 \leq d \leq F \\ 2Fd - \frac{1}{2}d^2 - F^2 - \frac{1}{2}d + F & \text{se } F < d \leq 2F - 1 \end{cases} \tag{4}$$

La **percentuale di entrate salvate** (o **tasso di compressione**) δ sarà invece dato da:

$$\delta(F,d) = \frac{F^2 - n}{F^2} = 1 - \frac{n}{F^2} \tag{5}$$

che vale

- $0 \leq d \leq F$:

$$\delta(F,d) = 1 - \frac{d(d+1)}{2F^2} \tag{6}$$

- $F < d \leq 2F - 1$:

$$\delta(F,d) = 1 - \frac{4Fd - d^2 - 2F^2 - d + 2F}{2F^2} = 2 - \frac{4Fd - d^2 - d + 2F}{2F^2} \tag{7}$$

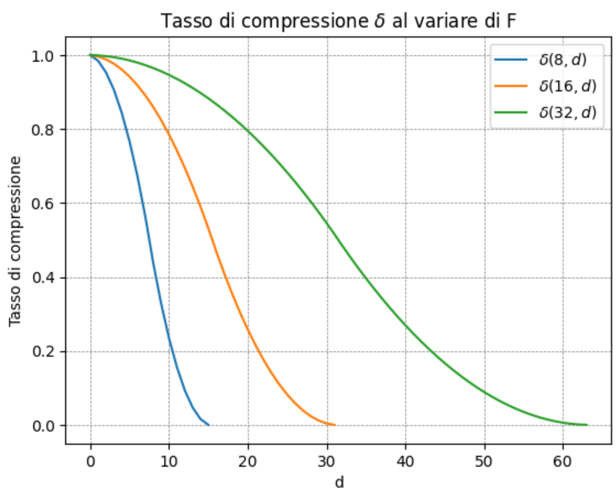

Figure 2: **Tasso di compressione** δ per diversi valori di F

Si noti che sia n sia δ fanno riferimento al *numero di elementi* dei blocchi ma **non direttamente** alla *quantità di spazio effettivo (in byte)*: infatti, data l'assenza di utilizzo di una **matrice di quantizzazione**, questo formato 'naive' salva gli elementi dei vettori compressi come **numeri a virgola mobile**, che quindi occupano **2, 4** o **8 bytes** (a seconda della precisione richiesta), a differenza dei singoli **byte** delle immagini originali. Questo comporta che, per valori di d non abbastanza piccoli, l'effetto sia quello di **aumentare la dimensione dell'immagine** anzichè diminuirla (nonostante ci sia comunque perdita di qualità).

Nel caso di immagini **RGB**, vengono compresse e salvate tutte e tre le matrici **R**, **G** e **B** in maniera indipendente.

2.3 Architettura di sistema

2.4 Esperimenti

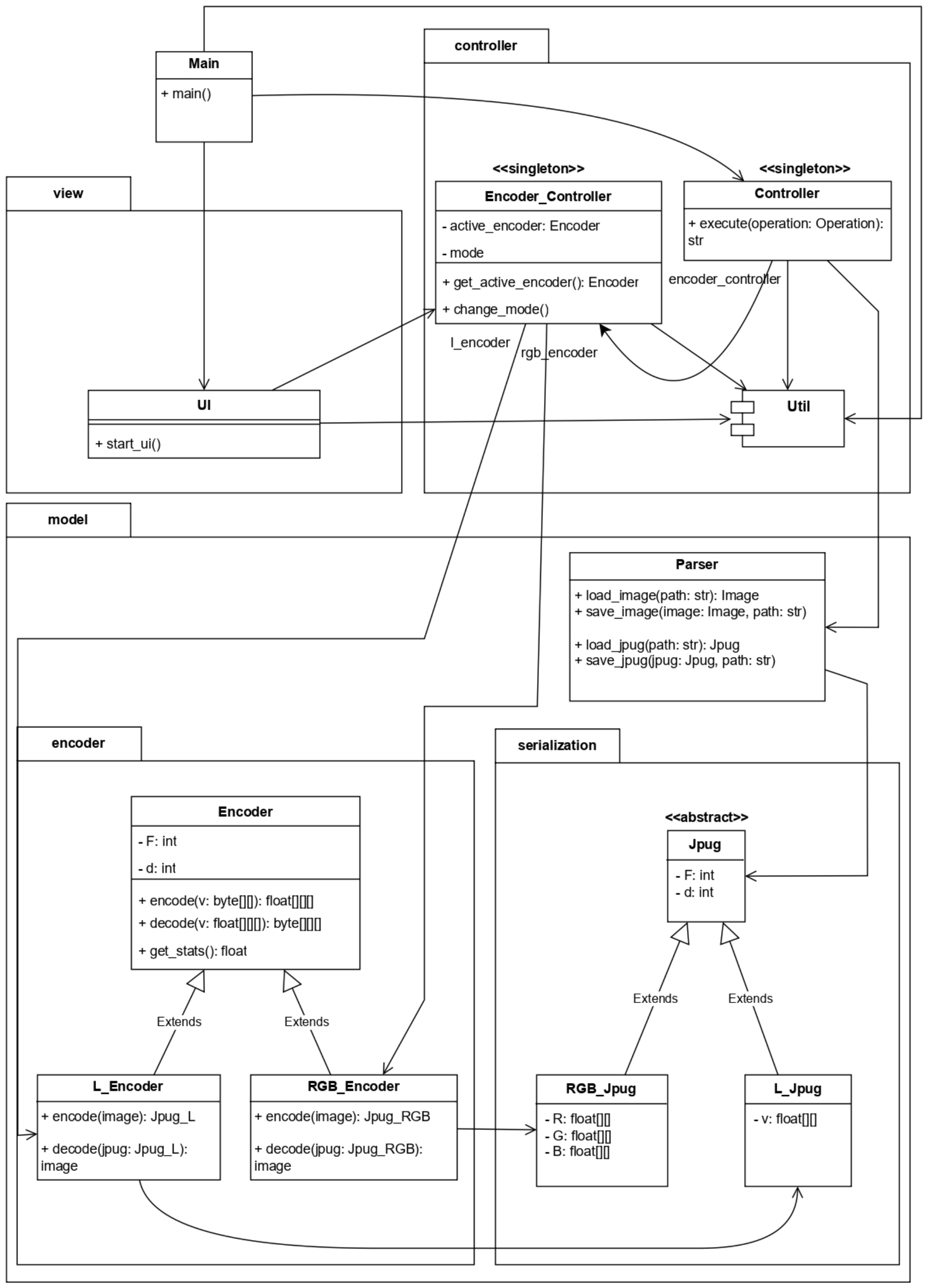

Figure 3: Diagramma delle classi