Elaborazione di Segnali Multimediali

Elaborazioni nel dominio spaziale

L.Verdoliva, D.Cozzolino

In questa lezione presenteremo le principali elaborazioni puntuali, cioè quelle operazioni che modificano singolarmente l'intensità di ogni pixel dell'immagine per migliorarne l'aspetto complessivo (*Image Enhancement*). Inoltre, studieremo le operazioni aritmetiche e quelle geometriche. Di seguito le istruzioni che ci permettono di importare i moduli Python che useremo in questa esercitazione:

```
# attiva la modalita interattiva di matplotlib
%matplotlib qt

import numpy as np              # importa Numpy
import matplotlib.pyplot as plt # importa Matplotlib
import scipy.ndimage as ndi     # importa Scipy per le immagini
import skimage.io as io         # importa il modulo Input/Output di SK-Image
```

1 Operazioni puntuali

Le elaborazioni puntuali si applicano ad ogni pixel dell'immagine senza considerare interazioni e dipendenze tra pixel vicini, e sono definite in funzione dell'intensità del pixel. Vedremo operazioni lineari e non lineari.

1.1 Offset additivo e cambiamento di scala

Cominciamo col valutare l'effetto di un offset additivo su un'immagine, realizziamo cioè la trasformazione: $y = x+b$, possiamo usare le immagini `vista_aerea.jpg` e `granelli.jpg`. La visualizzazione dell'istogramma può aiutarvi a scegliere il valore da assegnare all'offset b. In particolare, per la fotografia aerea un valore ragionevole è $b = -50$, quindi scrivendo `y = x-50` si ottiene l'immagine elaborata. Se si vuole fare in modo che l'immagine in uscita abbia la stessa dinamica dell'ingresso, bisogna fare attenzione al fatto che i valori dell'immagine potrebbero diventare negativi. Si può scrivere allora il seguente codice:

```
mask = x>0      # crea una matrice di vero (True) e falso (False)
x[mask] = 0     # si annullano tutti i valori minori di 0
```

In questo modo si gestiscono solo i valori negativi, per evitare anche di avere valori più grandi di 255:

```
x[x>255] = 255
```

Visualizzate la maschera per vedere in quali punti eventualmente si ha un effetto di saturazione.

Notate come con questa operazione le immagini presentino globalmente un aspetto più o meno luminoso, tuttavia l'istogramma, che è stato traslato verso sinistra o destra, mostra come non tutti i valori siano stati usati. Provate ad effettuare un cambiamento di scala $y = ax$ sulle due immagini proposte, determinando il valore appropriato di a in entrambi i casi e confrontando questa soluzione con quella precedente.

1.2 Negativi di un'immagine

Considerate l'immagine della mammografia `mammografia.jpg`, realizzatene il negativo ($y = K - 1 - x$) con $K = 256$.

1.3 Full Scale Histogram Stretch

Il Full Scale Histogram Stretch – o Contrast Stretch – è una delle operazioni più utili nell'ambito dell'elaborazione e della visualizzazione di immagini digitali. Scrivete una funzione con il seguente prototipo: `def fshs(x)` che realizza questa operazione nell'ipotesi in cui $K = 256$. Applicate questa funzione ad immagini il cui istogramma non copre l'intera dinamica e visualizzate il risultato. Come già detto Python realizza automaticamente questa operazione attraverso il comando `plt.imshow(x,clim=None,cmap='gray')`, provate allora a confrontare il risultato ottenuto con quello che avreste visualizzando l'immagine originale con tale comando. Per calcolare massimo e minimo potete usare i comandi: `xmin = np.min(x)` e `xmax = np.max(x)`.

1.4 Trasformazione logaritmo e potenza

Applichiamo la trasformazione logaritmo $y = \log(x + 1)$ in modo da migliorare il contrasto nelle zone scure per la trasformata di Fourier rappresentata nell'immagine `spettro.jpg`:

```
y = np.log(x+1)
plt.figure(); plt.imshow(y,clim=None,cmap='gray');
```

Provate invece ad applicare la trasformata potenza $y = x^\gamma$ all'immagine di `vista_aerea.jpg`, con $\gamma = 3$ e all'immagine di `spina_dorsale.jpg` (risonanza magnetica di una spina dorsale fratturata) con $\gamma = 0.3$ (ricordate di realizzare sempre il contrast stretch dopo l'operazione non lineare).

```
y = x ** 3
z = fshs(y)
```

Realizzate l'enhancement di immagini troppo chiare o scure, con $\gamma > 1$ e $\gamma < 1$, rispettivamente, visualizzando il risultato al variare di γ.

1.5 Equalizzazione dell'istogramma

L'equalizzazione dell'istogramma permette di migliorare il contrasto di un'immagine, quando sono stati usati tutti i valori appartenenti alla dinamica, ma la distribuzione appare fortemente sbilanciata. Caricate l'immagine marte.jpg, visualizzate il suo istogramma e poi effettuate l'equalizzazione dell'istogramma usando la funzione `skimage.exposure.equalize_hist` del Scikit-Image.

1.6 Uso delle statistiche locali

In questa sezione realizziamo l'enhancement solo su una parte dell'immagine `filamento.jpg`, allo scopo di evidenziare una struttura nascosta. Vogliamo effettuare un'elaborazione (amplificazione del valore) solo dei pixel appartenenti alle aree scure, che abbiano basso contrasto, ma non nelle regioni piatte. Questo si traduce in termini di condizioni su media e deviazione standard locale dell'immagine. Un pixel viene elaborato se:

1. la luminosità media attorno al pixel (media locale) è opportunamente minore della luminosità media dell'intera immagine;
2. la deviazione standard locale è relativamente bassa (indice di basso contrasto) e non è troppo piccola (per non enfatizzare le aree omogenee).

A questo punto solo se la media locale di un pixel è minore di quella globale opportunamente scalata ($\mu_l \leq k_0\mu$ per individuare zone scure) e se la deviazione standard locale soddisfa la disuguaglianza ($k_1\sigma \leq \sigma_l \leq k_2\sigma$ per assicurare un basso contrasto), con k_0, k_1 e k_2 costanti, il pixel va amplificato di un fattore E. Assumeremo $k_0 = 0.4$, $k_1 = 0.02$, $k_2 = 0.4$ e $E = 4$. Per realizzare il codice che effettua tali operazioni bisogna determinare la matrice di test (maschera), che vale 1 laddove il pixel va elaborato, 0 altrimenti: Nell'ipotesi di aver già calcolato media, `med`, e deviazione standard, `std`, globali dell'immagine, di conoscere l'immagine delle medie locali, `MED`, e quella delle deviazioni standard, `DEV`, su blocchi 3×3, il comando che genera la maschera è:

```
mask = ((MED<=0.4*med)) & (DEV<=0.4*dev)) & (DEV>=0.02*dev)))
```

Scrivete uno script dal nome `local_enhanc.m` che realizza le operazioni appena descritte e che richiama due funzioni per il calcolo delle medie e delle deviazioni standard, rispettivamente. Ricordate di visualizzare l'immagine originale e quella finale.

1.7 Esercizi proposti

Si vuole migliorare la visualizzazione di strutture nascoste nell'immagine quadrato.tif.

1. Realizzate l'enhancement globale dell'immagine con equalizzazione dell'istogramma, scrivendo la funzione `function y = glob_equaliz(x)`, e mostrate l'immagine elaborata;
2. poiché nemmeno questa operazione risolve il problema ricorrete all'enhancement locale, effettuate cioè l'equalizzazione dell'istogramma su blocchi 3×3 dell'immagine e conservate solo il pixel centrale del blocco elaborato. A tale scopo scrivete la funzione `function y = loc_equaliz(x)`, e mostrate l'immagine elaborata.

2 Bit-plane slicing

Consideriamo un'immagine in cui ogni livello è rappresentato su 8 bit. E' possibile suddividere l'immagine in bit-plane, cioè in piani in cui si rappresentano ognuno dei bit da quello meno significativo (bit-plane 0) a quello più significativo (bit-plane 7). La decomposizione di un'immagine digitale in bit-plane (*bit-plane slicing*) è molto utile per comprendere l'importanza che ogni bit ha nel rappresentare l'immagine e quindi se il numero di bit usato nella quantizzazione è adeguato. Estraiamo i bit-plane dell'immagine `frattale.jpg` usando la funzione `bitget` del file `bitop.py` fornito:

```
from bitop import bitget
B = bitget(x, 7)                        # estrazione bit-plane più significativo
plt.imshow(B, clim=[0,1], cmap='gray') # visualizzazione del bit-plane
```

Scrivete uno script dal nome `bit_plane.py` in cui estraete e visualizzate tutti i bit-plane dell'immagine. Potete memorizzare i bit-plane in una struttura tridimensionale in cui `bitplane[:,:,i]`.

2.1 Esercizi proposti

1. *Ricostruzione mediante bit-plane*. Ponete a zero i bit-plane meno significativi di un'immagine (usate la funzione `bitset` del file `bitop.py`) e visualizzate il risultato al variare del numero di bit-plane che utilizzate nel processo di ricostruzione. Questo esperimento vi permette di stabilire fino a che punto (almeno da un punto di vista percettivo) sia possibile diminuire il numero di livelli usati nel processo di quantizzazione.

2. *Esempio di Watermarking*. Provate adesso a realizzare una forma molto semplice di watermarking, che consiste nell'inserire una firma digitale all'interno di un'immagine. Sostituite il bit-plane meno significativo dell'immagine lena.y con l'immagine binaria marchio.y. Quest'ultima ha dimensioni 350×350 quindi è necessario estrarre una sezione delle stesse dimensioni dell'immagine lena.y. Provate poi a ricostruire l'immagine e visualizzatela, noterete che da un punto di vista visivo l'immagine non ha subito modifiche percettibili.

   Ripetete l'esperimento inserendo il watermark in un diverso bit-plane.

3 Operazioni aritmetiche

Le operazioni aritmetiche (somma/sottrazione, prodotto/divisione) coinvolgono una o più immagini e si effettuano pixel per pixel. Per esempio, fare la sottrazione tra due immagini vi permette di scoprire le differenze che esistono tra due immagini. Provate allora a visualizzare l'immagine frattale.jpg, e quella in cui sono stati posti a zero i 4 bit-plane meno significativi. Noterete come da un punto di vista visivo sono molto simili, fatene allora la differenza e visualizzatela a schermo.

4 Operazioni geometriche

Le operazioni geometriche consentono di ottenere, a partire da un'immagine x una nuova immagine y nella quale non si modificano i valori di luminosità, ma solo la posizione dei pixel.

4.1 Ridimensionamento

Rimpicciolire un'immagine di un fattore intero è estremamente semplice da realizzare in Numpy. Supponiamo per esempio di volerla ridurre di un fattore 2 lungo entrambe le direzioni, allora:

```
y = x[::2,::2]                          # decimazione per 2
plt.imshow(y, clim=[0,255], cmap='gray'); # visualizzazione
```

Questa operazione ci permette di modificare la risoluzione spaziale dell'immagine e consiste di fatto nell'abbassare (in numerico) la frequenza di campionamento del segnale. Se volessimo invece rimpicciolirla di un fattore non intero, realizzando per esempio la trasformazione $y[m,n] = x\left[\frac{3}{2}m, \frac{3}{2}n\right]$ bisogna fare più attenzione perché occorre assegnare correttamente i valori di intensità in uscita, come per esempio $y[1,1] = x\left[\frac{3}{2}, \frac{3}{2}\right]$. Quest'ultimo valore non è definito nell'immagine in ingresso per cui bisogna determinarlo mediante interpolazione. Possiamo utilizzate la funzione `rescale` del modulo `skimage.transform` che ridimensiona l'immagine con interpolazione bilineare:

```
from skimage.transform import rescale
y = rescale(x, 2/3, order=1)
plt.imshow(y, clim=[0,255],cmap='gray');
```

Il parametro `order` è il tipo di interpolazione. Con l'opzione `order=0` si effettua un'interpolazione nearest neighbor, mentre con `order=1` di tipo bilineare. Chiaramente si può anche ingrandire l'immagine se il fattore scelto è maggiore di 1; inoltre, con la funzione `skimage.transform.resize` si possono fissare le dimensioni che deve avere la nuova immagine (se però non si fa attenzione a conservare il rapporto d'aspetto, si crea distorsione nell'immagine).

In Python è possibile effettuare una trasformazione geometrica affine specificando direttamente la matrice di trasformazione $\mathbf{A}$ attraverso la funzione di `skimage.transform.warp`. Supponiamo di voler ingrandire una sezione di 25×50 pixel intorno all'occhio di lena:

```
from skimage.transform import warp

x = np.float32(io.imread('lena.jpg'))
x = x[252:277,240:290];
M = x.shape[0]; N = x.shape[1]

A = np.array([ [0.5,0,0], [0,0.5,0], [0,0,1]], dtype=np.float32)
y1 = warp(x, A, output_shape=(2*M,2*N), order = 0)
y2 = warp(x, A, output_shape=(2*M,2*N), order = 1)
y3 = warp(x, A, output_shape=(2*M,2*N), order = 3)

plt.subplot(3,1,1);
plt.imshow(y1,clim=[0,255],cmap='gray'); plt.title('interpolazione nearest');
plt.subplot(3,1,2);
plt.imshow(y2,clim=[0,255],cmap='gray'); plt.title('interpolazione bilinear');
plt.subplot(3,1,3);
plt.imshow(y3,clim=[0,255],cmap='gray'); plt.title('interpolazione bicubic');
```

Notate l'effetto di blocchettatura causato dall'interpolazione con opzione 'nearest' rispetto a 'bilinear' e 'bicubic'. Alla funzione `warp` abbiamo fornito anche il parametro `output_shape` che indica le dimensioni dell'immagine di uscita. Se il parametro `output_shape` è omesso l'immagine ottenuta `y` avrà lo stesso numero di righe e colonne di `x` ottenendo l'ingrandimento solo di una parte dell'immagine. Fate attenzione al fatto che la matrice affine richiesta dalla funzione `warp` è diversa da quella che abbiamo definito in teoria. In particolare:

$$[m', n', 1] = [m, n, 1]\,\mathbf{T}$$

mentre per la funzione `warp` si ha:

$$[n', m', 1] = [n, m, 1]\,\mathbf{A}^t$$

Ci sono quindi due differenze fondamentali: un'inversione di righe e colonne e una trasposta della matrice stessa. I comandi Python che ci permettono di ottenere $\mathbf{A}$ a partire da $\mathbf{T}$ sono i seguenti:

```
A = T[[1,0,2],:][:,[1,0,2]].T
```

4.2 Traslazioni e Rotazioni

Proviamo a realizzare la traslazione di un'immagine ($m' = m + 50, n' = n + 100$):

```
x = np.float32(io.imread('lena.jpg'))
A = np.array([ [1,0,100], [0,1,50], [0,0,1]], dtype=np.float32)
y = warp(x, A, order = 1)
plt.subplot(1,2,1);
plt.imshow(x,clim=[0,255],cmap='gray'); plt.title('originale');
plt.subplot(1,2,2);
plt.imshow(y,clim=[0,255],cmap='gray'); plt.title('traslata');
```

E' anche possibile modificare il colore per i pixel esterni al dominio dell'immagine. Se per esempio si vuole che abbiano colore bianco:

```
y = warp(x, A, order=1, cval=255)
```

In questo modo si inserisce una gradazione di grigio specificando un valore tra 0 (nero) e 255 (bianco).
Per la rotazione di un'immagine si ha:

```
from skimage.transform import warp

x = np.float32(io.imread('lena.jpg'))

A = np.array([[np.cos(np.pi/4),np.sin(np.pi/4),0],
     [-np.sin(np.pi/4),np.cos(np.pi/4),0],
     [0,0,1]], dtype=np.float32)

y = warp(x, A, order = 1)
plt.subplot(1,2,1);
plt.imshow(x,clim=[0,255],cmap='gray'); plt.title('originale');
plt.subplot(1,2,2);
plt.imshow(y,clim=[0,255],cmap='gray'); plt.title('ruotata');
```

Notate che il centro di rotazione non è il centro dell'immagine, ma la posizione [0,0]. Confrontate questo risultato con quello che otterreste direttamente con la funzione `skimage.transform.rotate`.

4.3 Esercizi proposti

1. *Distorsione*. Scrivete la funzione che realizza la distorsione di un'immagine lungo la direzione verticale e orizzontale e che abbia il prototipo: `deforma(x,c,d)`. Scegliete un'immagine e al variare dei parametri `c` e `d` osservate il tipo di distorsione.

2. *Rotazione Centrale*. Scrivete una funzione con il prototipo `ruota(x, theta)` che utilizza la funzione `wrap` per ruotare di un'angolo `theta` l'immagine `x` rispetto al centro dell'immagine. A tal fine usarte una combinazione di traslazioni e rotazione. Ricordatevi che la combinazione di diverse trasformazioni

affini è ancora una trasformazione affine, che può essere ottenuta tramite il prodotto (matriciale) delle matrici che le definiscono.

3. *Combinazione di operazioni geometriche*. Scrivete una funzione dal prototipo `rot_shear(x,theta,c)` per realizzare una rotazione e poi una distorsione verticale (attenzione all'ordine!). Fate le due operazioni rispetto al centro dell'immagine. Create l'immagine di ingresso usando il seguente comando `x = np.float64(skimage.data.checkerboard())` in modo da generare una scacchiera su cui le modifiche risultano essere più facilmente visibili.