

Multimedia

Homework 1

Alessandro Trigolo

7 Maggio 2024

Indice

1 Codifica semplice	4
1.1 Caricamento dell'immagine	4
1.2 Entropia dell'immagine	4
1.3 Codifica con dizionario	6
1.4 Discussione risultati parziali	6
1.5 Implementazione della codifica semplice	7
1.6 Entropia dell'errore di predizione	8
1.7 Codifica esponenziale di Golomb	8
1.8 Conclusioni	10
2 Codifica avanzata	12
2.1 Implementazione della codifica avanzata	12
2.2 Entropia dell'errore di predizione	14
2.3 Codifica esponenziale di Golomb	15
2.4 Confronto con codifica semplice	15
3 Conclusioni	16

Elenco delle figure

1 Estrazione della luminanza da un'immagine a colori.	5
2 Rappresentazione del modulo dell'errore di predizione nella codifica semplice.	8
3 Immagini scelte per effettuare le analisi.	10
4 Rappresentazione del modulo dell'errore di predizione nella codifica avanzata.	14
5 Rappresentazione della differenza tra l'errore della codifica semplice e della codifica avanzata.	14
6 Immagini scelte per effettuare le analisi.	16
7 Differenza di errore tra codifica semplice e avanzata dell'immagine della scacchiera.	17

Introduzione

Il presente report dettaglia il lavoro svolto per incontrare le richieste dell'*homework 1*, focalizzandosi sull'esplorazione e il confronto di diverse tecniche di codifica lossless per la compressione d'immagini in scala di grigi. L'obiettivo principale di questo studio è analizzare le prestazioni di codifiche predittive semplici e avanzate, valutando la loro efficacia nella compressione d'immagini con differenti caratteristiche. Attraverso l'analisi dell'entropia delle immagini, la valutazione del tasso di compressione ottenuto tramite la codifica con dizionario e l'analisi dell'errore di predizione, si mira a comprendere l'efficienza e l'adattabilità di queste tecniche in contesti diversi.

Il lavoro è stato svolto utilizzando il linguaggio di programmazione Python per implementare le diverse tecniche di codifica. All'interno del report non è presente tutto il codice prodotto ma sono presenti solo le parti fondamentali per il calcolo dei parametri richiesti. Ciò nonostante il codice è pubblico, accessibile al seguente [link](#).

1 Codifica semplice

La prima parte dell'homework è dedicata alla codifica semplice, la quale è una codifica predittiva di base per la compressione d'immagini in scala di grigi. L'obiettivo principale di questa parte del report è esplorare il funzionamento della codifica predittiva semplice, valutando la sua efficacia nella riduzione dell'errore di predizione e nel tasso di compressione ottenuto.

1.1 Caricamento dell'immagine

La prima richiesta dell'homework consiste nel caricare un'immagine a livelli di grigio oppure un'immagine a colori ed estrarne la luminanza, approssimabile come la media tra i tre canali dell'immagine (*Red*, *Green* e *Blue*). Il frammento di codice seguente incontra esattamente le richieste della prima task dove, attraverso la funzione `imread` del modulo `matplotlib.image`, l'immagine viene letta correttamente ed, eventualmente, ne viene estratta la luminanza.

```
1 # Prepare to load the image
2 img_file_name = "spiderman"
3 img_extension = ".jpg"
4 current_dir = os.getcwd()
5
6 # path to reach the img
7 path_to_img = os.path.join(current_dir, "multimedia", "hw-1",
8     "script", "imgs") + "/"
9
10 # loads the colored image
11 gray_img = mpimg.imread(path_to_img + img_file_name +
12     img_extension).astype(np.int16)
13
14 # extracts the luminance if RGB
15 if gray_img[0][0].size > 1:
16     gray_img = np.dot(gray_img[..., :3], [1, 1, 1]) / 3
```

Scegliendo un'immagine a colori è quindi possibile verificare il corretto funzionamento del codice, come mostrato nella figura 1 dove si è scelto come riferimento quella di *Spider-Man*.

1.2 Entropia dell'immagine

La seconda task dell'homework richiede di calcolare l'entropia dell'immagine in scala di grigi. L'entropia di una variabile aleatoria X (in questo caso l'immagine) è definita come l'**informazione media** degli eventi della sorgente; l'informazione di un evento è descritta dalla funzione seguente:

$$I(X) = \log_2 \left(\frac{1}{p_i} \right)$$



Figura 1: Estrazione della luminanza da un’immagine a colori.

L’informazione media diminuisce all’aumentare della probabilità dell’evento p_i . Questo è ragionevole in quanto più un evento è improbabile(quindi $p_i \rightarrow 0$) e più la sua informazione è alta ($I(X) \rightarrow +\infty$). Assumendo che gli eventi della sorgente siano indipendenti, l’informazione media si traduce nella seguente formula:

$$H(X) = E[I(X)] = \sum_{i=1}^M p_i \log_2 \left(\frac{1}{p_i} \right) = - \sum_{i=1}^M p_i \log_2 p_i$$

Dove con M si indica il numero di elementi nell’insieme X . Questa formula si riassume nel seguente script, dove la variabile contenente l’immagine in bianco e nero viene trasposta e convertita in un vettore monodimensionale, detto `raster_scan`. In secondo luogo attraverso la funzione `numpy.histogram` vengono contate il numero di occorrenze per ogni valore di pixel. Successivamente, per calcolare la probabilità, si divide il numero di occorrenze per il numero totale di occorrenze, escludendo eventuali valori diversi da zero. Una volta calcolare la probabilità, attraverso le funzioni `numpy.sum` e `numpy.log2` si ottiene il valore dell’entropia $H(X)$.

```

1 # flatten the transposed matrix to read pixels row by row
2 raster_scan = np.transpose(gray_img).flatten()
3
4 # count the occurrences of each pixel value
5 occurrences = np.histogram(raster_scan, bins=range(256))[0]
6

```

```

7   # calculate the relative frequencies
8   rel_freq = occurrences / np.sum(occurrences)
9
10  # remove zero-values of probability
11  p = rel_freq[rel_freq > 0]
12
13  # compute the entropy
14  entropy_x = - np.sum(p * np.log2(p))

```

Dopo aver eseguito lo script, l'entropia dell'immagine scelta è di **7.581 bpp**.

1.3 Codifica con dizionario

La terza task chiede di utilizzare una compressione a dizionario, come `zip` nel caso di *Windows* per poi calcolare il *bitrate* risultante. Lo script necessario per soddisfare la richiesta è presentato nel frammento di codice sottostante. In particolare le prime righe si occupano di "zippare" il file mentre le istruzioni seguenti estraggono la dimensione dell'immagine compressa (in byte). Infine nelle ultime linee di codice viene computato l'effettivo *bitrate* dividendo la dimensione del file compresso con la dimensione dell'immagine originale.

```

1   # change the current working directory to the directory containing
2   # the image
3   os.chdir(path_to_img)
4
5   # zip the image
6   cmd = f"zip {img_file_name}.zip {img_file_name}{img_extension}"
7   os.system(cmd)
8
9   # get the zip bytes
10  zip_bytes = os.stat(f"{img_file_name}.zip").st_size
11
12  # get img size
13  height, width = gray_img.shape
14  img_size = width * height
15
16  # get the bitrate
17  zip_bitrate = zip_bytes * 8 / img_size

```

<https://www.google.it/maps/@45.4105155,11.8792821,3306m/data=!3m1!1e3?entry=ttu>
Dunque, dopo aver eseguito lo script si ottiene il valore del bitrate, corrispondente a **1.329 bpp**.

1.4 Discussione risultati parziali

Osservando il valore di entropia ottenuto nel punto 1.2 si osserva che quest'ultima è molto vicina ad 8, indicando che l'immagine trovata, in ogni pixel, contiene 7.581 bit di informazione su un limite massimo di 8 bpp. Il motivo per cui tale valore non è

esattamente 8 può dipendere dal fatto che l'immagine (figura 1) contiene zone molto uniformi rispetto ad altre, per esempio il cielo e i grattacieli, i quali sono tutti della stessa tonalità di grigio e sono più o meno uniformi rispetto al personaggio raffigurato.

Il fatto che la compressione attraverso la codifica con dizionario abbia un bitrate di 1.329 bpp (punto 1.3) sta a significare che, nonostante l'entropia dell'immagine fosse alta, la compressione è riuscita a sfruttare in maniera ottimale la ridondanza dei dati, precedentemente puntualizzata. Infatti si osserva che dividendo l'entropia iniziale con il bitrate ottenuto dalla codifica con dizionario si ottiene un **tasso di compressione** di 5.706, che indica una compressione particolarmente efficace, nonostante l'elevata entropia iniziale.

1.5 Implementazione della codifica semplice

La quinta richiesta dell'homework è quella di effettuare una codifica predittiva *semplificata*. In particolare, data l'immagine, definita come un vettore chiamato $x(n)$, la codifica predittiva è definita come segue:

$$x(n) = \begin{cases} 128 & \text{se } n = 0 \\ x(n - 1) & \text{altrimenti} \end{cases}$$

Di conseguenza l'errore di predizione $y(n)$ sull'immagine $x(n)$ è dato da:

$$y(n) = \begin{cases} x(n) - 128 & \text{se } n = 0 \\ x(n) - x(n - 1) & \text{altrimenti} \end{cases}$$

Tale codifica si traduce nel seguente codice Python, dove viene utilizzato il vettore `raster_scan`, calcolato nel punto 1.2, contenente il vettore dell'immagine in scala di grigi.

```

1 # calculate prediction error
2 simple_coding_error = np.concatenate(([raster_scan[0] - 128],
3                                     np.diff(raster_scan.astype(float))))
4
5 # plot error graph
simple_coding_error_img =
    np.transpose(np.reshape(np.abs(simple_coding_error),
                           (width, height)))

```

Dopo aver eseguito lo script soprastante si ottiene l'immagine mostrata nella figura 2. Dall'immagine si può notare che le zone più chiare, ovvero le zone in cui il predittore ha fatto più errori, sono le zone dei contorni, in particolare del personaggio raffigurato. Questo perché la differenza dei colori tra una sezione e l'altra è particolarmente accentuata. D'altro canto, le zone di colore blu scuro sono quelle in cui i colori sono più uniformi: i palazzi e il cielo sono quasi dello stesso colore, suggerendo una zona dove la variazione di colore - e quindi di informazione - è molto bassa.

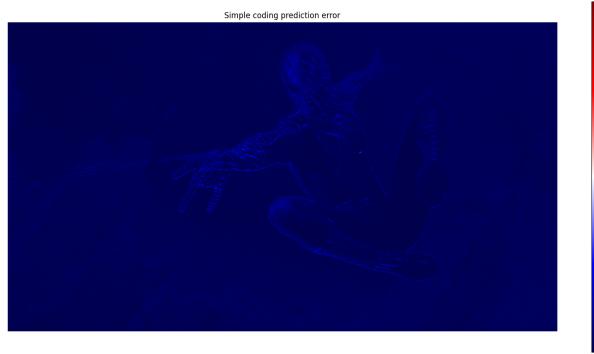


Figura 2: Rappresentazione del modulo dell’errore di predizione nella codifica semplice.

1.6 Entropia dell’errore di predizione

La richiesta successiva è quella di calcolare il valore dell’entropia dell’errore di predizione $y(n)$. Utilizzando uno script del tutto simile a quello utilizzato nella task 1.2 è quindi possibile calcolare l’entropia richiesta.

```

1 # count the occurrences of each prediction error value
2 occ, _ = np.histogram(simple_coding_error, bins = range(-255,
256))
3
4 # calculate the relative frequencies and remove any probability == 0
5 freqRel = occ / np.sum(occ)
6 p = freqRel[freqRel > 0]
7
8 # calculate the entropy
9 entropy_y = - np.sum(p * np.log2(p))

```

Dopo l’esecuzione dello script si ottiene che l’entropia dell’errore di predizione semplice è **4.382 bpp**.

1.7 Codifica esponenziale di Golomb

Seguendo le direttive della task numero 7, calcoliamo la codifica esponenziale di Golomb (detta anche *exp-Golomb*), la quale dato un numero intero ritorna la sua codifica. Quest’ultima è più efficace rispetto a una normale codifica binaria in quanto durante la codifica vengono risparmiati diversi bit, migliorando quindi la compressione.

Tale codifica può essere esaminata prima affrontando il caso in cui tutti i numeri da codificare siano interi positivi e poi tale concetto si può generalizzare. La codifica di Golomb per interi positivi, detta anche *exponential Golomb unsigned*, è definita come segue.

$$\text{eg_unsigned}(n) = \begin{cases} 1 & \text{se } n = 0 \\ \text{zeros}(\lfloor \log_2(n + 1) \rfloor) + \text{dec2bin}(n + 1) & \text{altrimenti} \end{cases}$$

Dove `zeros(k)` indica una funzione che ritorna una stringa di k zeri mentre la funzione `dec2bin(n)` ritorna il valore binario del numero naturale n . A questo, dopo aver definito la codifica senza segno, la codifica con segno, detta anche *exponential Golomb signed*, è immediata:

$$\text{eg_signed}(n) = \begin{cases} \text{eg_unsigned}(2n - 1) & \text{se } n > 0 \\ \text{eg_unsigned}(-2n) & \text{altrimenti} \end{cases}$$

Dopo aver definito matematicamente la codifica, è possibile tradurla in due funzioni Python molto semplici, che riassumono esattamente quanto già osservato.

```

1  def exp_golomb_signed(n: int) -> str:
2      # Computes the Exp-Golomb code for signed integers
3
4      if n > 0:
5          return exp_golomb_unsigned(2 * n - 1)
6
7      return exp_golomb_unsigned(-2 * n)
8
9
10 def exp_golomb_unsigned(n: int) -> str:
11     # Computes the Exp-Golomb code for non-negative integers
12
13     # handle the case where N is zero
14     if n == 0:
15         return '1'
16
17     # returns the coded string of bits
18     return '0' * int(math.floor(math.log2(n + 1))) +
           format(n + 1, 'b')
```

Possiamo quindi occuparci di calcolare quanto richiesto dalla consegna. Per calcolare il numero di bit necessari utilizzando la codifica esponenziale di Golomb, è sufficiente calcolare la codifica per ogni errore della predizione semplice. La lunghezza delle codifica di ciascun valore viene sommata ottenendo quindi il numero totale di bit necessari per codificare l'errore. Infine, per ottenere il bitrate di codifica è necessario dividere il numero totali di bit per la grandezza dell'immagine. Il seguente script ricopre esattamente queste direttive.

```

1  def exp_golomb_count(vector: list) -> int:
2      # Calculate the bitrate based on the exponential Golomb code for
3      # a given vector.
4
5      bit_count = 0
6      for symbol in vector:
```

```

6     bit_count += len(exp_golomb_signed(int(symbol)))
7
8     return bit_count
9
10    exp_golomb_bit = exp_golomb_count(simple_coding_error)
11
12    exp_golomb_bpp = exp_golomb_bit / img_size

```

Dopo l'esecuzione dello script, il numero di bit necessari per la codifica è 41 305 140 mentre il bitrate di codifica ha il valore di **4.980 bpp**.

1.8 Conclusioni

Cerchiamo ora di ottenere dei dati statistici che ci permettano di trarre delle informazioni utili. Si sono scelte sei immagini, mostrate nella figura 3, alcune delle quali volte proprie a testare le capacità della codifica semplice (e poi di quella avanzata). In particolare le prime tre immagini hanno peculiarità uniche: la prima, completamente nera, rappresenta l'assenza di entropia, la seconda (la scacchiera) è composta da blocchi mentre la terza rappresenta del rumore Gaussiano, con l'intendo di avere un'immagine con alta entropia. Le altre immagini invece sono normali immagini da comparare.

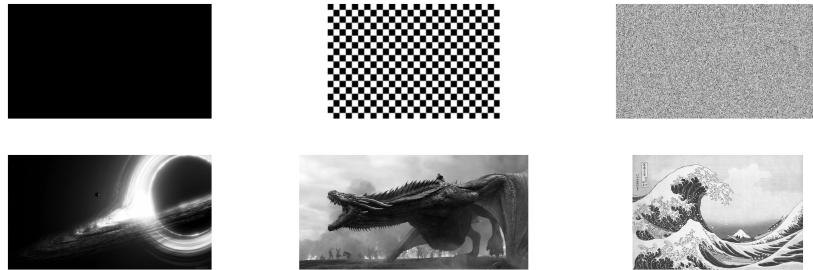


Figura 3: Immagini scelte per effettuare le analisi.

Per ciascuna delle immagini si è eseguito il codice e i risultati trovati sono stati riassunti nella seguente tabella (1).

Osservando la tabella riassuntiva si possono notare valori particolarmente diversi per ciascuna immagine. Chiaramente l'immagine completamente nera ha entropia nulla e di conseguenza lo è anche l'errore di predizione; anche il tasso di codifica esponenziale di Golomb suggerisce che è necessario solo un bit per rappresentare l'errore. Per quanto riguarda la seconda immagine, quella della scacchiera, si ha un'entropia più alta ma comunque ridotta rispetto alle altre immagini. Questo perché i due colori dominanti sono il bianco e il nero, senza altre variazioni di conseguenza il segnale non è sparsificato come negli altri casi. Ciò nonostante il tasso di codifica è comunque abbastanza elevato. Nella terza immagine, raffigurante il rumore Gaussiano, possiamo notare che sia l'entropia, ma ancora di più l'errore di predizione sono molto elevati. Questo perché i pixel dell'immagine sono casuali rendendolo un segnale difficile da predire. Proprio per questo motivo il suo tasso di codifica è così elevato.

img	.pgm	.zip	error	EG-coding
black	0.000	0.001	0.000	1.000
chessboard	5.539	2.218	5.420	6.364
gaussian-noise	7.338	3.254	7.663	11.454
interstellar	6.816	0.274	2.300	2.332
dragon	7.659	1.255	3.599	3.567
great-wave	7.055	3.022	6.175	7.947
spiderman	7.581	1.329	4.382	4.980

Tabella 1: Dati ottenute dalle varie istanze di esecuzione dello script, con immagini diverse.

Nel caso delle altre immagini, i risultati non sono così inaspettati, a parte nel caso della *Great wave* dove l'entropia dell'errore di predizione e il tasso di codifica sono comunque particolarmente elevati rispetto alle altre immagini.

2 Codifica avanzata

La seconda sezione del report, dedicata alla codifica avanzata, si concentra sull'analisi e la valutazione di un tecnica di codifica predittiva più sofisticata rispetto a quella precedentemente esaminata. Come vedremo, la codifica avanzata permette di ottenere risultati più precisi e più efficienti rispetto alla codifica semplice, in particolare per immagini con caratteristiche più complesse.

2.1 Implementazione della codifica avanzata

La penultima richiesta, esige di creare un nuovo tipo di codifica, detta *avanzata* definita come segue:

$$y(n, m) = \begin{cases} x(n, m) - 128 & \text{se } n = m = 0 \\ x(n, m - 1) & \text{se } n = 0 \\ x(n - 1, m) & \text{se } m = 0 \\ \text{med}[x(n, m - 1), x(n - 1, m), x(n - 1, m - 1)] & \text{se } m = \text{MAX} \\ \text{med}[x(n, m - 1), x(n - 1, m), x(n - 1, m + 1)] & \text{altrimenti} \end{cases}$$

Dove in questo caso l'immagine anzichè essere vista come un vettore x è vista come una matrice di dimensione $n \times m$. Le richieste della codifica sono tradotte in uno script contenente due cicli annidati - il primo che itera lungo le righe della matrice e il secondo che itera lungo le colonne - dove all'intero sono presenti una serie di `if` statement che soddisfano la definizione di codifica avanzata precedentemente citata. Inoltre si osserva la mediana dei tre valori è calcolata tramite una funzione definita dall'utente, descritta anch'essa all'interno dello script.

```
1 # median function for advanced coding
2 def median(a, b, c):
3
4     v = [a, b, c]
5     v.sort()
6
7     return v[1]
8
9
10    # codes an image based on the guidelines directives
11    def advanced_coding(img):
12        # blank image
13        predicted_img = np.zeros_like(img)
14
15        # extracts the height and the width from the image
16        height, width = img.shape[0], img.shape[1]
17
18        # iterates through the rows (height)
19        for row in range(height):
```

```

21     # iterates through the cols (width)
22     for col in range(width):
23
24         if row == 0 and col == 0: # first pixel
25             predicted_img[row][col] = img[row][col] - 128
26
27         elif row == 0:           # first row
28             predicted_img[row][col] = img[row][col - 1]
29
30         elif col == 0:          # first col
31             predicted_img[row][col] = img[row - 1][col]
32
33         elif col == (width - 1): # last col
34             predicted_img[row][col] = median(img[row -
35                 1][col], img[row][col - 1], img[row - 1][col
36                 - 1])
37
38     else:                   # other cases
39         predicted_img[row][col] = median(img[row -
40                 1][col], img[row][col - 1], img[row - 1][col
41                 + 1])
42
43     return predicted_img
44
45
46     # performs advanced coding
47     predicted_img = advanced_coding(gray_img)
48
49     # calculates the prediction error
50     adv_coding_error_img = gray_img - predicted_img

```

Dopo aver eseguito lo script è possibile mostrare l'errore di predizione sottraendo l'immagine predetta `predicted_img` con l'immagine iniziale `img` e "plottando" il suo valore assoluto. L'immagine mostrata in figura 4 è ciò che risulta degli errori commessi durante la predizione.



Figura 4: Rappresentazione del modulo dell’errore di predizione nella codifica avanzata.

Si osserva che per quanto l’immagine 4 sia simile alla figura 2, è presente una differenza: è sufficiente mostrare un’immagine che contenga la differenza in modulo tra le varaiibili `simple_coding_error` e `adv_coding_error`. Dal risultato, mostrato nella figura 5, si può notare che non è di colore uniforme ma presenta diverse sfumature di blu, suggerendo quindi una differenza tra le due immagini di errore.

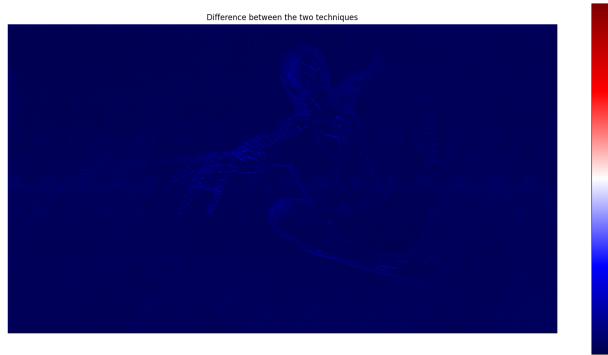


Figura 5: Rappresentazione della differenza tra l’errore della codifica semplice e della codifica avanzata.

2.2 Entropia dell’errore di predizione

Dopo aver calcolato l’errore del predittore avanzato dell’immagine, è necessario calcolarne l’entropia. Le istruzioni necessarie per il calcolo dell’entropia sono le stesse

utilizzate per il calcolo dell'entropia del predittore semplice nel paragrafo 1.6.

```
1     occ, _ = np.histogram(adv_coding_error_img, bins = range(-255,
2                                         256))
3
4     # calculate the relative frequencies and remove any probability == 0
5     freqRel = occ / np.sum(occ)
6     p = freqRel[freqRel > 0]
7
8     # calculate the entropy
9     entropy_y = - np.sum(p * np.log2(p))
```

Dopo aver eseguito lo script, si ottiene il valore dell'entropia del valore di predizione avanzata il quale corrisponde a **4.195 bpp**, che è minore rispetto a quello ottenuto nell'errore di codifica semplice, suggerendo che la codifica avanzata è stata più efficace.

2.3 Codifica esponenziale di Golomb

Come per la codifica semplice, anche in questo caso è necessario calcolare la codifica esponenziale di Golomb sull'errore di predizione. Il codice utilizzato è esattamente lo stesso indicato sul paragrafo 1.7 con l'unica differenza che la funzione viene invocata sull'errore di predizione avanzata anziché su quella semplice.

```
1     # calculates EG bits
2     exp_golomb_bit =
3         exp_golomb_count(adv_coding_error_img.flatten())
4
5     # computes the bitrate
6     exp_golomb_bpp = exp_golomb_bit / img_size
```

In questo caso, per l'errore di predizione avanzata sull'immagine 1, il numero di bit di necessari è di 38 854 794 mentre il bitrate è **4.6845 bpp**, il quale, anche in questo caso, è minore rispetto a quello della codifica semplice.

2.4 Confronto con codifica semplice

Confrontando i valori ottenuti con la codifica avanzata con i valori della codifica semplice, si desume che le prestazioni della codifica avanzata sono migliori. In particolare, per quanto riguarda l'entropia dell'errore, si ha che nella codifica semplice l'entropia è di 4.382 bpp mentre nella codifica avanzata l'entropia corrisponde a 4.195 bpp. Questa differenza indica che la codifica avanzata ha una maggiore efficienza nella riduzione dell'errore di predizione che si traduce in una migliore rappresentazione dell'immagine originale.

In secondo luogo, si nota che anche il tasso di codifica si riduce tra la codifica semplice (che è di 4.980 bpp) e la codifica avanzata (4.685) sottolineando l'efficacia di quest'ultima. La differenza dei due valori indica che con la codifica avanzata si è in grado di rappresentare lo stesso numero di informazioni attraverso un numero minore di bit (infatti $38\ 854\ 794 < 41\ 305\ 140$).

Per quanto il tasso di codifica sia largamente minore rispetto all'entropia dell'immagine originale (7.581 bpp), rimane comunque molto più elevato rispetto al *bitrate* ottenuto zippando l'immagine (che corrisponde a 1.329 bpp). Questo è dovuto al fatto che la codifica avanzata implementata rimane comunque molto meno inefficiente rispetto alle tecniche ottimizzate della codifica con dizionario offerta dai sistemi operativi.

3 Conclusioni

Confrontiamo ora i valori ottenuti per la codifica avanzata con quelli precedentemente ottenuti nella codifica semplice nelle stesse immagini, illustrate per comodità nella figura 6 sottostante.

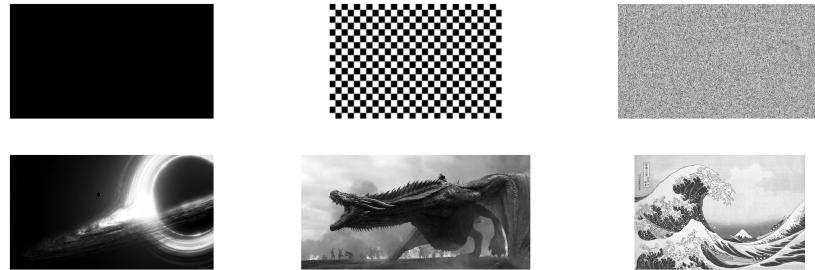


Figura 6: Immagini scelte per effettuare le analisi.

Dopo aver eseguito la codifica avanzata e aver calcolato il tasso di codifica dell'errore di predizione su ciascuna immagine, i risultati ottenuti sono riassunti nella tabella 2. Si può notare che in genere, casi isolati a parte, la codifica avanzata ha prodotto dei

Immagine	Entropia		Tasso di codifica	
	Semplice	Avanzata	Semplice	Avanzata
black	0.000	0.000	1.000	1.000
chessboard	5.420	5.529	6.364	6.457
gaussian-noise	7.663	7.459	11.454	11.092
interstellar	2.300	2.139	2.332	2.177
dragon	3.599	3.412	3.567	3.338
great-wave	6.175	6.056	7.947	7.801
spiderman	4.382	4.195	4.980	4.684

Tabella 2: Tabella di confronto riassuntiva.

risultati migliori rispetto alla codifica semplice. In genere, sia l'entropia che il tasso

di codifica sono minori di circa due decimi rispetto ai valori ottenuti della codifica semplice sottolineando ancora una volta che la codifica avanzata è più efficace rispetto a quella semplice.

Spostiamo l'attenzione ora sui casi isolati. In particolare si osserva che l'immagine completamente nera non ha alcun tipo di miglioramento o peggioramento tra una codifica e l'altra. Questo risultato è più che ragionevole in quanto, essendo l'immagine di entropia nulla, anche l'errore di predizione sarà nullo.

Un risultato invece più interessante riguarda l'immagine della scacchiera. Questa è l'unica istanza in cui la codifica semplice ha avuto un risultato migliore rispetto alla codifica avanzata. Osserviamo la figura 7 dove a destra è presente l'errore nella codifica semplice mentre a sinistra l'errore nella codifica avanzata. Si può notare

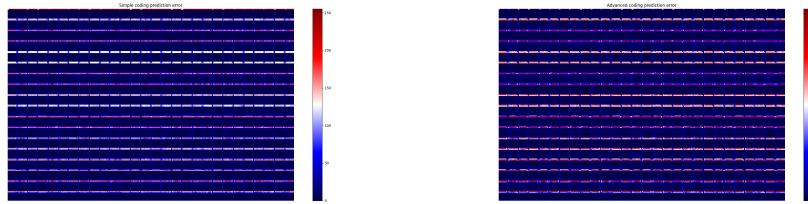


Figura 7: Differenza di errore tra codifica semplice e avanzata dell'immagine della scacchiera.

che in diverse zone la percentuale di rosso è più elevata nella figura di sinistra. In particolare si concentrano nella zona in cui si passa in una nuova riga della scacchiera, dove i colori sopra sono l'esatto contrario dei colori sotto. Proprio per questo motivo i valori della codifica avanzata sono maggiori rispetto ai valori della codifica semplice. Quest'ultima predice il pixel basandosi solamente su quello precedente (a sinistra) mentre la codifica avanzata prende il valore mediano tra tre pixel, di cui due sono della riga superiore. Proprio per questo motivo, nel momento in cui si cambia una nuova riga della scacchiera il valore che viene preso è l'opposto del valore effettivo del pixel (quando si cambia la riga della scacchiera il valore sopra è bianco mentre quello sotto è nero o viceversa) aumentando quindi l'entropia dell'errore.

Complessivamente, i risultati ottenuti dimostrano che la codifica predittiva avanzata è in grado di sfruttare in modo più efficiente la ridondanza presente nelle immagini, portando a una migliore compressione rispetto alla codifica semplice.