Universidad Rey Juan Carlos

Estándares de Codificación de Audio y Video

Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

PRÁCTICA 2: FUNDAMENTOS DE CODIFICACIÓN DE IMAGEN Y VÍDEO

Parte I – Codificación de imagen en Python

El objetivo de esta parte de la práctica es dar a conocer al alumno los principios básicos de la compresión de imágenes, llevando a cabo la compresión completa de un bloque de tamaño 8x8, que es el tamaño estándar que se utiliza, tanto en la compresión de imágenes fijas JPEG, como en la de imágenes móviles MPEG. Cualquier imagen, por compleja que sea, siempre se descompondrá en bloques elementales de tamaño 8x8. Este bloque se podrá codificar modificando el grado de compresión que se desea conseguir. Para cada una de las situaciones consideradas, se analizará el error cometido, pudiendo observarse que cuanto más alto sea el nivel de compresión, más baja será la calidad obtenida en el bloque reproducido.

Para todas estas operaciones se utilizará el lenguaje Pyhton, que en la actualidad es de los más utilizados para realizar tratamiento de imágenes, por su simplicidad, por la gran cantidad de librerías de las que dispone para trabajar con este tipo de señales y por ser de libre distribución.

Para realizar esta parte de la práctica hemos utilizado el código proporcionado por el Prof. Dr.-Ing. Johannes Maucher (https://www.hdm-stuttgart.de/~maucher/Python/MMCodecs/html/index.html)

Importamos los paquetes que vamos a necesitar:

```
import cv2
```

```
import numpy as np

from matplotlib import pyplot as plt

from matplotlib.colors import Normalize

import matplotlib.cm as cm
```

En OpenCV las imágenes en color se importan como BGR. Lo primero que hacemos es convertirla en RGB intercambiando los canales rojo y azul para poder representar la imagen con *pyplot.imshow():*

```
B=8 # Tamaño del bloque

Imagen= 'zelda.bmp'

img1 = cv2.imread(Imagen,cv2.CV_LOAD_IMAGE_UNCHANGED)

h,w=np.array(img1.shape[:2])/B * B

img1=img1[:h,:w]

#Convert BGR to RGB

b,g,r = cv2.split(img1)

img2 = cv2.merge((r,g,b))

plt.figure()

plt.imshow(img2)
```

A continuación, se pide que marquemos una región en la imagen mostrada. El bloque formado se muestra en color azul.

```
point=plt.ginput(1)

block=np.floor(np.array(point)/B) #first component is col,
second component is row

print "Coordinates of selected block: ",block

scol=block[0,0]

srow=block[0,1]
```

```
plt.plot([B*scol,B*scol+B,B*scol+B,B*scol,B*scol],
[B*srow,B*srow,B*srow+B,B*srow+B,B*srow])

plt.axis([0,w,h,0])
```

Transformamos la imagen BGR en YCrCb y aplicamos submuestreo sobre las componentes de color.

```
transcol=cv2.cvtColor(img1, cv2.cv.CV_BGR2YCrCb)

SSV=2

SSH=2

crf=cv2.boxFilter(transcol[:,:,1],ddepth=-1,ksize=(2,2))

cbf=cv2.boxFilter(transcol[:,:,2],ddepth=-1,ksize=(2,2))

crsub=crf[::SSV,::SSH]

cbsub=cbf[::SSV,::SSH]

imSub=[transcol[:,:,0],crsub,cbsub]
```

La variable `SSH` define el factor de submuestreo en dirección horizontal y `SSV` en dirección vertical. Recordar:

- con 4:4:4, no se reduce la información de color respecto a la de brillo → la imagen no sufre pérdidas.

- con 4:2:2, se reduce la información de color en un factor de 2 en dirección horizontal → el color tiene la mitad de resolución (en horizontal) y el brillo sigue intacto.

- con 4:2:0, la información de el color se reduce en un factor de 2 en ambas direcciones, horizontal y vertical → el color tiene la cuarta parte de resolución y el brillo sigue intacto.

¿A qué formato de submuestreo de color corresponden los valores de SSH=2 y SSV=2?

Antes de submuestrear, las componentes de color se pasan por un filtro promedio de tamaño 2 x 2. La imagen YCrCb se almacena en la variale `ImSub`:

Para realizar la cuantificación de los coeficientes transformados, se utilizan las matrices de cuantificación definidas en el estándar JPEG (una para la componente de luminancia y otra para las componentes de crominancia):

```
QY=np.array([[16,11,10,16,24,40,51,61,
[12,12,14,19,26,48,60,55],[14,13,16,24,40,57,69,56],
[14,17,22,29,51,87,80,62],[18,22,37,56,68,109,103,77],
[24,35,55,64,81,104,113,92],[49,64,78,87,103,121,120,101],
[72,92,95,98,112,100,103,99]])
```

```
QC=np.array([[17,18,24,47,99,99,99,99]
[18,21,26,66,99,99,99,99],[24,26,56,99,99,99,99,99],
[47,66,99,99,99,99,99,99],[99,99,99,99,99,99,99,99],
[99,99,99,99,99,99,99,99],[99,99,99,99,99,99,99,99],
[99,99,99,99,99,99,99,99]])
```

Para obtener diferentes niveles de calidad de la cuantifiación se puede definir un factor de calidad QF. A partir de de este factor de calidad se puede calcular un parámetro de escala. Las matrices definidas arriba se multiplican por el parámetro de escala. Una valor bajo de QF implica un valor alto del parámetro de escala → cuantificación gruesa o burda→ baja calidad pero alta tasa de compresión. En la lista Q almacenamos las 3 matrices cuantificadas y escaladas que van a ser aplicadas a los coeficientes de la DCT:

```
QF=99.0

if QF < 50 and QF > 1:

     scale = np.floor(5000/QF)

elif QF < 100:

     scale = 200-2*QF

else:

     print "Quality Factor must be in the range [1..99]"

scale=scale/100.0

Q=[QY*scale,QC*scale,QC*scale]
```

A continuación, realizamos la DCT + cuantficación sobre cada uno de los canales. Como se define en el estándar, antes de realizar la DCT, se resta el valor de 128 para que todos los valores queden en el rango [-128, 127].

```
TransAll=[]

TransAllQuant=[]

ch=['Y','Cr','Cb']

plt.figure()

for idx,channel in enumerate(imSub):

     plt.subplot(1,3,idx+1)

     channelrows=channel.shape[0]

     channelcols=channel.shape[1]

     Trans = np.zeros((channelrows,channelcols),
np.float32)

     TransQuant = np.zeros((channelrows,channelcols),
np.float32)

     blocksV=channelrows/B

     blocksH=channelcols/B

     vis0 = np.zeros((channelrows,channelcols),
np.float32)

     vis0[:channelrows, :channelcols] = channel

     vis0=vis0-128

     for row in range(blocksV):

          for col in range(blocksH):

               currentblock = cv2.dct(vis0[row*B:
(row+1)*B,col*B:(col+1)*B])

               Trans[row*B:(row+1)*B,col*B:
(col+1)*B]=currentblock
```

```
                TransQuant[row*B:(row+1)*B,col*B:
(col+1)*B]=np.round(currentblock/Q[idx])

     TransAll.append(Trans)

     TransAllQuant.append(TransQuant)

     if idx==0:

          selectedTrans=Trans[srow*B:(srow+1)*B,scol*B:
(scol+1)*B]

     else:

          sr=np.floor(srow/SSV)

          sc=np.floor(scol/SSV)

          selectedTrans=Trans[sr*B:(sr+1)*B,sc*B:(sc+1)*B]


     plt.imshow(selectedTrans,cmap=cm.jet,interpolation='n
     earest')

     plt.colorbar(shrink=0.5)


     plt.title('DCT of '+ch[idx])
```

`TransAll` contiene los coeficientes DCT de los tres canales, *`TransAllQuant`* contiene los coeficientes DCT de los tres canales cuantificados. Notar que los canales tienen diferentes tamaños debido al submuestreo de las componentes de color. En la última parte del bucle, los coeficientes de la DCT del bloque seleccionado y resaltado son mostrados.

El proceso completo de codificación intracuadro (que reduce la redundancia espacial) terminaría aplicando la codificación entrópica (RLE + VLC) a cada bloque de coeficientes DCT cuantificados, que no vamos a ver en esta práctica.

Para comprobar la calidad de la codificación intracuadro (basada en JPEG) vamos a tartar de reconstruir la imagen original a partir de la codificada. Para cada componente (YCrCb) realizamos la decodificación en sentido inverso a la codificación: DCT inversa, desnormalización (rango de valores en [0,255]). Además, las componentes de

crominancia son interpoladas, usando el método `resize()` de `opencv,` para recuperar su tamaño original.

```
ImagDecodificada=np.zeros((h,w,3), np.uint8)

for idx,channel in enumerate(TransAllQuant):

     channelrows=channel.shape[0]

     channelcols=channel.shape[1]

     blocksV=channelrows/B

     blocksH=channelcols/B

     back0 = np.zeros((channelrows,channelcols), np.uint8)

     for row in range(blocksV):

          for col in range(blocksH):

               dequantblock=channel[row*B:(row+1)*B,col*B:
(col+1)*B]*Q[idx]

               currentblock =
np.round(cv2.idct(dequantblock))+128

               currentblock[currentblock>255]=255

               currentblock[currentblock<0]=0

               back0[row*B:(row+1)*B,col*B:
(col+1)*B]=currentblock

     back1=cv2.resize(back0,(w,h))

     ImagDecodificada[:,:,idx]=np.round(back1)
```

El array `ImagDecodificada` contiene la imagen reconstruida en formato YCrCb. Finalmente, la imagen se transforma a BGR, se presenta y se almacena en una fichero. La calidad se mide calculando el SSE (Sum of Square Error) entre la imagen original y la imagen decodificada.

```
reImg=cv2.cvtColor(ImagDecodificada, cv2.cv.CV_YCrCb2BGR)
```

```
cv2.cv.SaveImage('BackTransformedQuant.jpg',
cv2.cv.fromarray(reImg))

plt.figure()

img3=np.zeros(img1.shape,np.uint8)

img3[:,:,0]=reImg[:,:,2]

img3[:,:,1]=reImg[:,:,1]

img3[:,:,2]=reImg[:,:,0]

plt.imshow(img3)

SSE=np.sqrt(np.sum((img2-img3)**2))

print "Sum of squared error: ",SSE

plt.show()
```

Ejecutar el código anterior considerando distintos valores de QF (entre 1 y 99), distintos formatos de submuestreo de color (4:4:4, 4:2:2, 4:2:0) y obtener el valor de SSE correspondiente. Comente los resultados obtenidos para las imágenes proporcionadas.

Parte II – Codificación de video

Material:

Para realizar esta práctica vamos a utilizar tres programas: ffmpeg, vlc y Avidemux. Todos son gratuitos y se pueden descargar de las siguientes páginas web:

https://sourceforge.net/projects/avidemux/

https://ffmpeg.org/download.html

http://www.videolan.org/vlc/

Vamos a utilizar dos secuencias de video, una con poco movimiento (de dibujos animados de la serie "*Peppa Pig*") y otra con mucho (de la película de "*Avatar*"). Ambas secuencias están codificadas en x264 (códec H.264), que actualmente es el códec más usado, y se encuentran disponibles junto al guión de la práctica.

Objetivo 1 (O1): Entender los parámetros de codificación vistos en clase (resolución, fps, etc.)

Tarea 1 (T1): Comentar los parámetros encontrados resaltando lo que caracteriza a cada tipo de secuencia de video.

Pasos a realizar para alcanzar el objetivo O1:

Para visualizar las dos secuencias, vamos a emplear el programa VLC. En el menú de herramientas > Información Multimedia, se encuentra información del codec, y en su pestaña de estadísticas se muestran los parámetros de interés en tiempo real.

Comentar cómo cambian los parámetros encontrados dependiendo de la secuencia de fotogramas que hay en el video.

Nota: Para utilizar ffmpeg copia el ejecutable `ffmpeg` en la carpeta donde has descargado las secuencias de video. A continuación, abre un terminal en esa carpeta. Recuerda que para lanzarlo debes utilizar "`./ffmpeg`".

Con ffmpeg, vamos a reducir el tiempo de ambas secuencias a 30 segundos, sin audio, para así trabajar más cómodamente. Además, aprovecharemos para quitar toda compresión que haya, convirtiéndola a formato raw.

Sabiendo los fps de cada secuencia (del apartado anterior), ¿Cuántos fotogramas son necesarios para conseguir 30 segundos de cada película?

En la ventana de comandos, ejecutamos:

```
./ffmpeg -i peppapig.mp4 –vframes Numero_Fotogramas –c:v
rawvideo -pix_fmt yuv420p peppapig.yuv
```

Comprueba que el tamaño del archivo raw obtenido es el resultado de multiplicar el número de fotogramas por la resolución de cada uno y por los bits empleados en cada píxel.

Realiza la misma operación con la otra película.

Objetivo 2 (O2): Entender el funcionamiento del estándar mpeg-2.

Tarea 2 (T2): Codificar en mpeg-2

Pasos a realizar para alcanzar el objetivo O2:

Ejecuta:

```
./ffmpeg -f rawvideo -pix_fmt yuv420p -s:v 640x360 -r 25 -i
peppapig.yuv -c:v mpeg2video peppapig_mpeg2.avi
```

Como se observa, al codificar desde un video en formato raw es imprescindible indicar el tipo de codificación de los píxeles, así como la resolución de las imágenes y la tasa de imágenes por segundo.

Sin indicar ningún parámetro más al codificador, ¿qué efecto se observa?

El parámetro "-qscale:v 10", está indicando que aplique un valor de compresión de 10, para esa implementación del códec. En ffmpeg, las librerías de los códecs empleadas usan habitualmente valores entre 1 y 31 para graduar el tipo de compresión, pero eso es algo específico de la implementación realizada del estándar.

Los valores bajos de compresión buscarán la compresión únicamente en escenas en las que haya mucho movimiento, mientras que los valores altos aplicarán incluso a las tramas I. Localiza cuáles son esos valores límites para las diferentes partes de cada secuencia de video.

```
./ffmpeg -f rawvideo -pix_fmt yuv420p -s:v 640x360 -r 25 -i
peppapig.yuv -c:v mpeg2video –qscale:v 10
peppapig_mpeg2_10.avi
```

¿La codificación obtenida es suficientemente buena? Puedes utilizar zoom para acercarte. Cambia el qscale a 5. Comenta los resultados.

Vamos a generar una imagen png de un instante de la cada secuencia, una vez codificada en mpeg2, y vamos a compararla con la misma imagen fija obtenida de la secuencia original o de la raw. De esta forma, se puede hacer un zoom mayor.

Para seleccionar un instante, emplea el parámetro "-ss":

```
./ffmpeg -i peppapig_mpeg2_5.avi –ss 00:00:01.440 –vframes
1 foto_peppa_mpeg2.png
```

```
./ffmpeg -f rawvideo -pix_fmt yuv420p -s:v 640x360 -r 25 -i
peppapig.yuv –ss 00:00:01.440 –vframes 1 foto_peppa_yuv.png
```

Con Avidemux vamos a poder ver de qué tipo es cada fotograma en cada secuencia de video (en la parte inferior de la pantalla, se indica cuál es el tipo de fotograma en el que está detenida la película). Localiza fotogramas de tipo I y P. Observa en qué casos se generan fotogramas tipo I (*key-frames*).

Determina cuál es el GOP de las diferentes versiones que hemos codificado. ¿Es fijo o variable? ¿Cuándo se insertan tramas I? Nota: analiza al menos los 10 primeros segundos.

Si cambiamos ese GOP durante la codificación, ¿cómo influirá en el resultado? Para cambiarlo, ejecutar ffmpeg con el parámetro –g, es decir para usar un GOP de hasta 100, "-g 100". Compara el resultado tanto a nivel visual como a nivel de bytes comprimidos y comenta los resultados.

Repite ahora los pasos anteriores con la secuencia de video de Avatar.

La codificación en mpeg-2 también se puede realizar con Avidemux en lugar de con ffmpeg. Observa los parámetros de codificación que permite ajustar Avidemux para codificar el video en mpeg-2 y relaciónalos con lo visto en clase para dicho estándar. Prueba a codificar ambas secuencias cambiando los valores de los parámetros y comenta los resultados obtenidos (con respecto a calidad visual) con cada cambio.

Objetivo 2 (O3): Entender el funcionamiento del estándar H.264

Tarea 2 (T3): Codificar en H.264

Tarea 3 (T4): Evaluar la compatibilidad.

Pasos a realizar para alcanzar el objetivo O3:

Vamos a ejecutar:

```
./ffmpeg -f rawvideo -pix_fmt yuv420p -s:v 640x360 -r 25 -i peppapig.yuv -c:v libx264 peppapig_x264.mp4
```

¿Se observa pérdida de calidad? ¿Cuándo es más notable?

Usando Avidemux, localiza los tipos de fotogramas I, P y B en cada secuencia codificada con H.264 y compara con las obtenidas en mpeg-2.

En la implementación de H.264, los parámetros CRF (*Constant Race Factor*) y QP (*Quantization parameter*) tienen una importancia considerable. En el enlace http://slhck.info/articles/crf puedes encontrar la información que necesitas para entender este parámetro. Utiliza ffmpeg y/o Avidemux para estudiar la importancia de dicho parámetro sobre una de las secuencias de video del material y comenta los resultados obtenidos.

El estándar H.264 define perfiles de codificación, para ajustar la secuencia codificada a diferentes categorías de decodificadores (con más o menos capacidad de cálculo). Evalúa cuál es la compatibilidad de tu móvil con el estándar H.264, probando diferentes perfiles. Para ello, aplica las opciones indicadas en https://trac.ffmpeg.org/wiki/Encode/H.264#Compatibility

Evalúa ahora cómo se comportan estos perfiles cuando también limitas la tasa de bits o ajustas el CRF.

La codificación en H.264 también se puede realizar con Avidemux en lugar de con ffmpeg. Observa los parámetros de codificación que permite ajustar Avidemux para codificar el video en H.264 y relaciónalos con lo visto en clase para dicho estándar. Prueba a codificar ambas secuencias cambiando los valores de los parámetros y comenta los resultados obtenidos (con respecto a calidad visual) con cada cambio.

Con Avidemux, visualiza los vectores de movimiento y saca conclusiones. *Nota*: Avidemux nos permite visualizar los vectores de movimiento (en Video Decoder > Configurar, activar Show motion vectors)

Compresión sin pérdidas en H.264:

Vamos a generar un video a partir de un conjunto de imágenes fijas pertenecientes a la película de *Peppa pig*. Para obtener una imagen de tipo png cada segundo, ejecutamos:

```
./ffmpeg –i peppapig.mp4 –vf fps=1/1 img%03d.png
```

Para generar una secuencia de video, sin pérdida al comprimir (opción "-qp 0"), ejecutamos:

```
./ffmpeg -framerate 1/1 -i img%03d.png -c:v libx264 –qp 0
-r 1 -pix_fmt yuv420p salida1.mp4
```

La película resultante (salida1.mp4) no es visible en muchos reproductores de video, que no soportan codificación sin pérdidas, pero sí lo será en avidemux.

Si seleccionamos todas las imágenes generadas (29), el tamaño total que ocupan es mucho mayor que el archivo generado salida1.mp4, ¿por qué puede ser?