Lab 3: Spatial Error Concealment Techniques

Kyeong Soo (Joseph) Kim
Department of Communications and Networking
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
3 May 2024

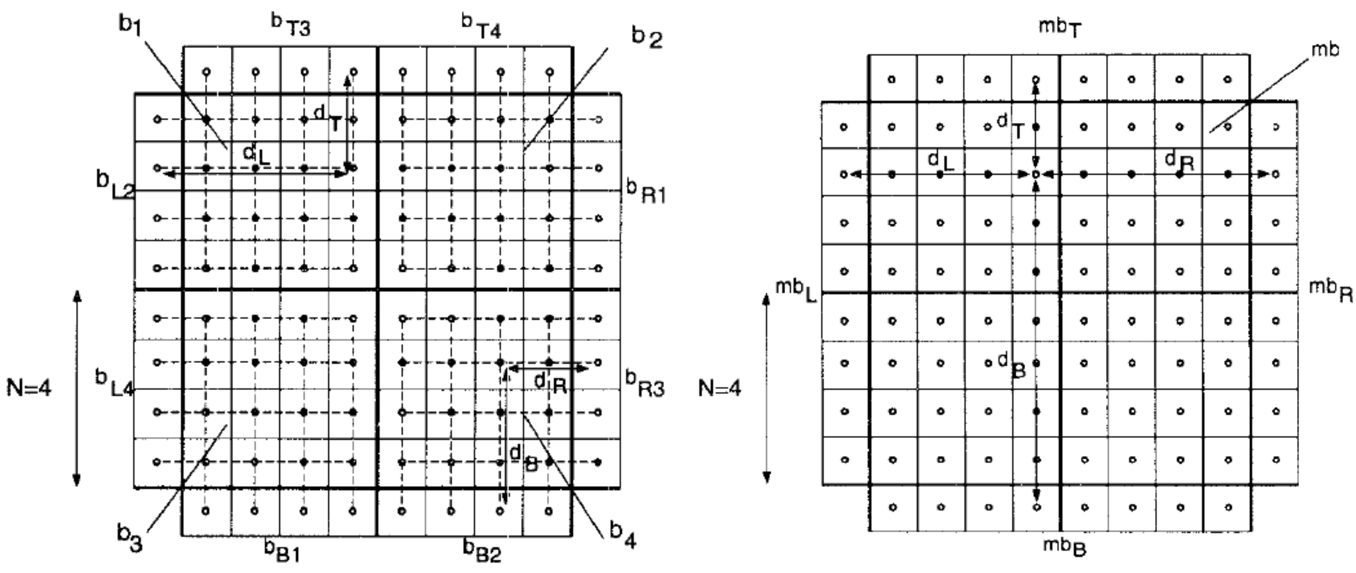
I. Introduction

In this Lab, you are to carry out spatial error concealment techniques for intra-coded frame. For simplicity, we do not consider quantization and block coding but directly work in the image domain. You are provided two grayscale 592×848 images — i.e., an original ('peppers) and a damaged ('peppers_damaged') — as two-dimensional arrays of type numpy.uint8 stored in the NumPy NPY files **"peppers.npy"** and **"peppers_damaged.npy"**; the impact of packet losses are modeled as several 16×16 macroblocks whose values are set to zeros in the damaged image. A Python script "macroblocks.py" is also provided to demonstrate the conversion between an image array and a corresponding two-dimensional array of macroblocks and processing of those macroblocks to model packet loss effect.

You need to submit a Lab report and Python script(s) through the Learning Mall Online by the end of Sunday, 26 May 2024.

II. Review of Spatial Error Concealment Techniques

Here we briefly review two simple spatial error concealment techniques proposed for MPEG-2 video codec [1], which are illustrated in Fig. 1 for 4x4 blocks within an 8x8 macroblock.



(a) (b)

Fig. 1. Spatial interpolation techniques for error concealment: (a) Block based. (b) Macroblock based (from [1]).

Of the two techniques, the first one interpolates each single $N \times N$ block in one macroblock as shown in Fig. 1 (a). This interpolation can be described as follows: For $i,\ k = 1, \ldots, N$,

$$\begin{aligned} b_1(i,\ k) &= \frac{d_T b_{L2}(i,\ N) + d_L b_{T3}(N,\ k)}{d_L + d_T} \\ b_2(i,\ k) &= \frac{d_T b_{R1}(i,\ 1) + d_R b_{T4}(N,\ k)}{d_R + d_T} \\ b_3(i,\ k) &= \frac{d_B b_{L4}(i,\ N) + d_L b_{B1}(1,\ k)}{d_L + d_B} \\ b_4(i,\ k) &= \frac{d_B b_{R3}(i,\ 1) + d_R b_{B2}(1,\ k)}{d_R + d_B} \end{aligned} \tag{1}$$

where $b_l,\ l = 1, \ldots, 4$ is the l^{th} block of the current macroblock, $b_{Xl},\ l = 1, \ldots, 4$ with $X = L,\ R,\ T,\ B$ is the l^{th} block of the neighboured macroblock (Left, Right, Top, Bottom) and d_x with $X = L,\ R,\ T,\ B$ is the distance from the respective pixel of the block b_{Xl} to the current pixel $b_l(i,\ k)$.

The second technique interpolates each pixel of the whole macroblock with the adjacent pixels of the four neighbouring macroblocks. Fig. 1 (b) shows the macroblock with the boundary pixels of the neighbouring macroblocks. Each pixel of the current macroblock with the size $2N \times 2N$ will be concealed by simple interpolation of the four pixels of the surrounding macroblocks, i.e., for $i,\ k = 1, \ldots, 2N$,

$$\begin{aligned} mb(i,\ k) = \frac{1}{d_L + d_R + d_T + d_B} \Big(& d_R mb_L(i, 2N) + d_L mb_R(i, 1) + \\ & d_B mb_T(2N, k) + d_T mb_B(1, k) \Big) \end{aligned} \tag{2}$$

where mb is the current macroblock, mb_X with $X = L,\ R,\ T,\ B$ is the respective neighboured macroblock (Left, Right, Top, Bottom) and d_X with $X = L,\ R,\ T,\ B$ is the distance from the respective pixel of the macroblock mb_X to the current pixel $mb(i,\ k)$. This technique works better if the surrounding macroblocks exist. If some of the macroblocks do not exist for interpolation (e.g., if one whole stripe of macroblocks is damaged), the corresponding distance will be set to zero (for instance if mb_L do not exist d_R will be set to zero).

With only two available macroblocks mb_T and mb_B the equation (2) reduces to: For $i,\ k = 1, \ldots, 2N$,

$$mb(i,\ k) = \frac{d_B mb_T(2N, k) + d_T mb_B(1, k)}{d_T + d_B} \tag{3}$$

III. Task: Macroblock-Based Spatial Interpolation

For this task, you need to submit a Lab report with Python script(s) for the following activities:

1) (5 points) Block-based spatial interpolation.
   - Conceal the lost macroblocks of "peppers_damaged" using the block-based spatial interpolation technique described in Section II.
   - Display both the original ("peppers") and the error-concealed images from step (1) for comparison (refer to the "macroblocks.m" and Fig. 2 in this regard).
   - Calculate PSNR of the error-concealed image with respect to the original image (again, refer to the "macroblocks.m" in this regard).
2) (5 points) Repeat 1) but this time using the macroblock-based spatial interpolation technique.

Note that The following files are provided on Learning Mall Online for this task:

- *peppers.npy*: A NumPy NPY file for the original peppers image.
- *peppers_damaged.npy*: A NumPy NPY file for the damaged peppers image.

Fig. 2. Original and damaged peppers images.

- *macroblocks.py*: A Python script demonstrating the conversion between an image array and a corresponding cell array of macroblocks and processing of those macroblocks to model packet loss effect.

APPENDIX
MACROBLOCKS.PY: PYTHON SCRIPT FOR PROCESSING IMAGE USING MACROBLOCKS

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
##
# @file      macroblocks.py
# @author    Kyeong Soo (Joseph) Kim <kyeongsoo.kim@gmail.com>
# @date      2020-04-22 created for "mandrills" image
#            2024-04-26 updated for "peppers" image
#
# @brief     Convert a grayscale image into 8x8 macroblocks and
#            process them to model packet loss impact.
#


import math
import matplotlib.pyplot as plt
import numpy as np


def psnr(img1, img2):
    """
    Calculate PSNR of two grayscale images.

    Args:
        img1 (2x2 numpy array of uint8): grayscale image 1
        img2 (2x2 numpy array of uint8): grayscale image 2
    """
    mse = np.mean((img1 - img2)**2)
    try:
        psnr = 20 * math.log10(255 / math.sqrt(mse)) # 255 is for the max. value of uint8
    except ZeroDivisionError as e:
        print("Error: " + str(e))
        print("PSNR cannot be defined for identical images, so we just return 100.\n")
        return 100 # meaningless value!
    return psnr

```

```python

def random_discard_mbs(img, mb_size=16, mb_number=128):
    """
    Discard macroblocks randomly.

    Args:
        img (2x2 numpy array of numpy.uint8): grayscale image
        mb_size: macroblock size; default is 16
        mb_number: number of macroblokcs to discard; default is 128
    """
    imgx, imgy = img.shape # image dimensions
    mbx = imgx // mb_size # number of MBs in X
    mby = imgy // mb_size # number of MBs in Y

    # create macroblocks from the original image
    # N.B.: based on the techniques: https://towardsdatascience.com/efficiently-splitting-an-
    # N.B.: copy() is needed; otherwise, mbs would be a different "view" of the original imag
    mbs = np.copy(img.reshape(mbx, mb_size, mby, mb_size).swapaxes(1, 2))
    # N.B.: the above is equivalent to the following (but much faster, of course):
    #---------------------------------------------------------------------
    # mbs = np.empty((mbx, mby), dtype=object) # object array storing MBs
    # for i in range(mbx):
    #     for j in range(mby):
    #         mbs[i ,j] = img[i*mb_size:(i+1)*mb_size, j*mb_size:(j+1)*mb_size]
    #---------------------------------------------------------------------

    # random discard of MBs to model packet loss impact
    xs = np.random.randint(mbx, size=mb_number)        # x index of MBs to discard
    ys = np.random.randint(mby, size=mb_number)        # y index of MBs to discard
    mbs_damaged = mbs
    for i in range(mb_number):
        mbs_damaged[xs[i], ys[i]] = np.zeros((mb_size, mb_size)) # set values of discarded MB

    # convert damaged MBs back to an image
    img_damaged = mbs_damaged.swapaxes(1, 2).reshape(imgx, imgy)
    # N.B.: again, the above is equivalent to the following:
    #---------------------------------------------------------------------
    # img_damaged = np.empty((imgx, imgy), dtype=np.uint8)
    # for i in range(mbx):
    #     for j in range(mby):
    #         img_damaged[i*mb_size:(i+1)*mb_size, j*mb_size:(j+1)*mb_size] = mbs_damaged[i,
    #---------------------------------------------------------------------

    return img_damaged # convert to uint8 as a grayscale image


if __name__ == "__main__":
    img = np.load("peppers.npy") # load original peppers image (592x848 uint8 array)
    img_damaged = random_discard_mbs(img, mb_size=16, mb_number=128) # damaged image

    fig, axs = plt.subplots(1, 2)
    axs[0].imshow(img, cmap='gray')
    axs[0].title.set_text('Original')
    axs[0].set_axis_off()
    axs[1].imshow(img_damaged, cmap='gray')
    axs[1].title.set_text('Damaged')
    axs[1].set_axis_off()
```

```
    plt.tight_layout()
    plt.show()

    print(f'PSNR = {psnr(img_damaged, img):.4f}') # PSNR of damaged image
```

REFERENCES

[1] S. Aign and K. Fazel, "Temporal & spatial error concealment techniques for hierarchical MPEG-2 video codedc," in Proc. 1995 IEEE ICC, Seattle, WA, USA, Jun. 1995, pp. 1778–1783.