SIMON FRASER UNIVERSITY

Assignment 2: Image Inpainting and Image Compression

Zhiqin Chen, Ali Mahdavi Amiri

February 13, 2019

1 OVERVIEW

In this assignment you will implement:

1) Poisson Blending (10 points)

2) Inpainting by U-net (9 points)

3) Inpainting by Resnet (8 points)

4) Image Compression by SVD (3 points)

5) Wavelet Compression (6 points)

The report is required although I did not assign points for it. The penalty is -1 point for each missing figure or comment in the report.

If you have questions, try to Google them first. If you cannot find an answer, you can send me an email. I will not respond to the questions of which you can simply find the answers in Pytorch documents.

There are probably mistakes in this assignment. If you find any, please email me as soon as possible. I will give you bonus points if it is a severe mistake, thanks.

Please train your networks on GPUs in the lab, otherwise it may take a loooooong time.

When submitting to Canvas, please zip everything you have (codes, report) into one single file “yourname-studentnumber-ass2.zip”.

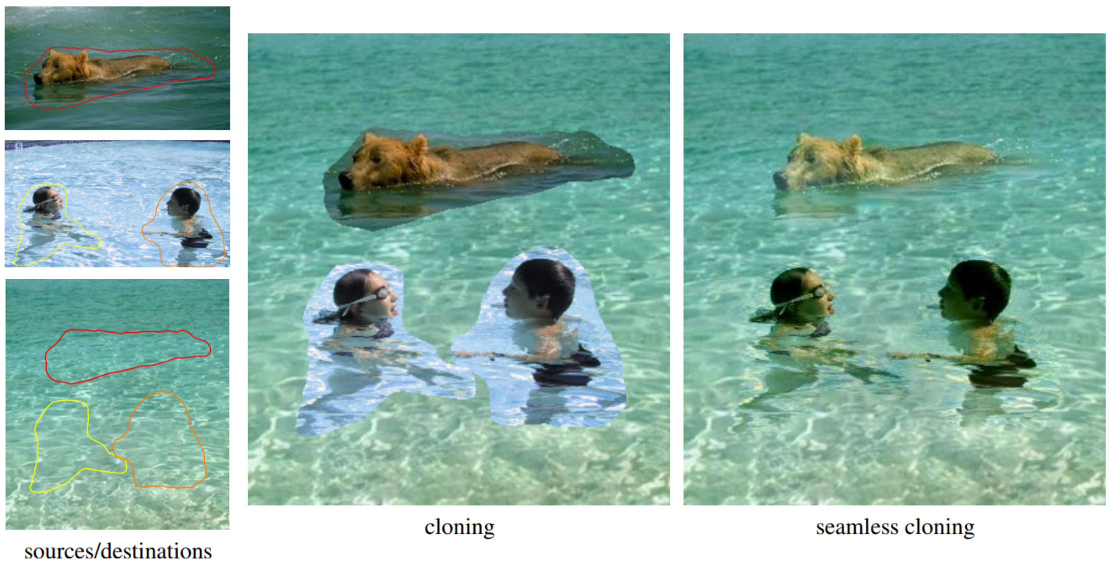

Figure 2.1: Poisson image blending sample from [2].

2 Poisson blending

In this section, you will implement Poisson blending [2]: given an image patch in the source image and its destination in the target image, Poisson blending can change the color of this patch but retain its content so that it can seamlessly blend into the target image.

2.1 Image reconstruction

We already implemented image reconstruction from first-order derivatives by solving a least squares problem in the tutorial. This time we reconstruct the image by referring to its second-order derivatives. We denote the source image as S and the reconstructed image as V. Denote the intensity of the pixel (x, y) in image S as $s_{(x,y)}$ and the pixel in image V as $v_{(x,y)}$. Note that S is given so $s_{(x,y)}$ are constants, and $v_{(x,y)}$ are variables. To obtain variables $v_{(x,y)}$, we need to solve the equations (for each pixel):

$$V^g_{up} + V^g_{down} + V^g_{left} + V^g_{right} = S^g_{up} + S^g_{down} + S^g_{left} + S^g_{right} \tag{2.1}$$

Where $V^g_{left} = v_{(x,y)} - v_{(x-1,y)}$, $V^g_{right} = v_{(x,y)} - v_{(x+1,y)}$, $V^g_{up} = v_{(x,y)} - v_{(x,y-1)}$, $V^g_{down} = v_{(x,y)} - v_{(x,y+1)}$ are the gradients of $v_{(x,y)}$ in the four directions. The others are defined similarly, for example $S^g_{down} = s_{(x,y)} - s_{(x,y+1)}$. In order to compute it, we have:

$$4 \times v_{(x,y)} - v_{(x+1,y)} - v_{(x-1,y)} - v_{(x,y+1)} - v_{(x,y-1)} = S^g_{up} + S^g_{down} + S^g_{left} + S^g_{right} \tag{2.2}$$

Where the red part is made of variables, and the blue part is a constant. By representing the coefficients of V as a matrix A and the constants as b, we have $AV = b$. If the image has k pixels, V will be a $k \times 1$ matrix, A will be $k \times k$, and b will be $k \times 1$.

Note: If a pixel is an edge pixel, for example $(1,2)$, it does not have a left neighbor, therefore it does not have horizontal second-order derivative, so in the equation you only need to consider vertical gradients ($V^g_{up} + V^g_{down} = S^g_{up} + S^g_{down}$). If a pixel is a corner pixel, it has no second-order derivative ($0 = 0$). We have four corner pixels, so there will be four all-zero rows in A and the corresponding four zeros in b. This is why we cannot get the reconstructed image by solving the current equations. We need to provide four "control points" to make A has rank k so that solving $AV = b$ will give us a unique solution. The four "control points" control the global lightness and the global gradients. For this assignment, you need to choose the four corner points as the four "control points" for better control, i.e., $(1,1)$, $(1,n)$, $(m,1)$ and (m,n).

$$\begin{aligned} v_{(1,1)} &= s_{(1,1)} \\ v_{(1,n)} &= s_{(1,n)} \\ v_{(m,1)} &= s_{(m,1)} \\ v_{(m,n)} &= s_{(m,n)} \end{aligned} \tag{2.3}$$

Considering the four extra constraints, V will be a $k \times 1$ matrix, A will be $(k+4) \times k$, and b will be $(k+4) \times 1$. You may make A $k \times k$ by removing the four all-zero rows, but it is not necessary since in any case we can solve the equations.

You need to write down and solve the equations using $A = sparse(i, j, v)$ (5 points), and then copy those variable pixels to the appropriate positions in the output image. The framework is given (same to the one used in the tutorial).

Note: As a validation, print the error $sum(abs(AV - b))$ after you get the solution V. If your implementation is correct, this error should be very small (like $1e^{-11}$ or $1e^{-12}$), no matter what values you have assigned to the control points.

What to submit: the code main.m.

What to show in the report: Since you have 4 "control points" now, you can manipulate the ground truth values to control the global lightness and gradients. Show 5 reconstructed images in the report: one similar to the ground truth, one globally brighter, one brighter on the left side, one brighter on the bottom side, and one brighter on right bottom corner. Some examples are shown in Figure 2.2.

2.2 Poisson Blending

The provided matlab code already gives you a framework that allows you to:

First, load a source image and select a region you want to copy. You can click multiple times to specify the boundaries of your patch, then press "Q" to get a closed patch.

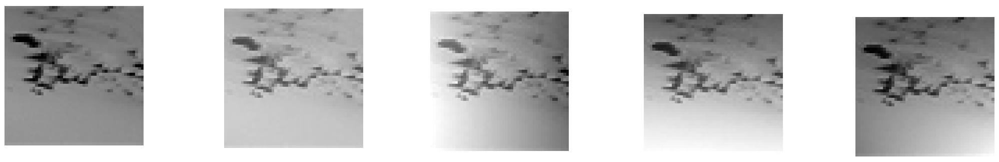

Figure 2.2: Sample results for image reconstruction.

Second, load a target image and place your patch onto the image. Press “W”“S” to resize and “A”“D” to rotate. Press “Q” to finish.

Tips: try to select the boundaries in the featureless regions for both images, e.g. sky.

You need to complete poisson_blend.m. In the code you are given the source image S, the target image T and the Mask M, where in M a pixel is 1 if it is in the selected region and 0 otherwise. Your task is to implement Poisson blending, which is to solve the equations (for each pixel in the selected region):

$$V^{tg}_{up} + V^{tg}_{down} + V^{tg}_{left} + V^{tg}_{right} = S^{g}_{up} + S^{g}_{down} + S^{g}_{left} + S^{g}_{right} \tag{2.4}$$

This time, we only consider the pixels in the selected region. Suppose we have k pixels in the selected region (that means mask M has k non-zero entries), In $AV = b$, V will be a $k \times 1$ matrix, A will be $k \times k$, and b will be $k \times 1$. No extra constraint is required.

V^{tg} is defined according to the mask. For (x, y), if $(x-1, y)$ is in the selected region, we have $V^{tg}_{left} = (v_{(x,y)} - v_{(x-1,y)})$; if $(x-1, y)$ is outside, $V^{tg}_{left} = (v_{(x,y)} - t_{(x-1,y)})$. The gradients for the other directions are defined in a similar way.

After implementing *Image reconstruction*, it should be straightforward to implement poisson blending. For each pixel in the selected region, consider its four neighbors. If a neighbor is inside the selected region, treat it as a variable; if it is outside, treat it as a constant defined by T.

You need to write down and solve the equations using $A = sparse(i, j, v)$ (3 points), and then copy those variable pixels to the appropriate positions in the output image to obtain the blended image (1 point). For RGB images, process each channel separately and combine the results together in the end. Write this in an recursive or iterative way so that poisson_blend can handle both grayscale and color images (1 point).

Note: As you can see, this time A again is a full rank matrix, therefore solving $AV = b$ will give us a unique solution. Print the error $sum(abs(AV - b))$ after you get the solution V. If your implementation is correct, this error should be very small.

What to submit: the code poisson_blend.m. Do not change other files, except for the image names in main.m.

What to show in the report: Show 3 pairs of blending results in the form of Figure 2.1. You

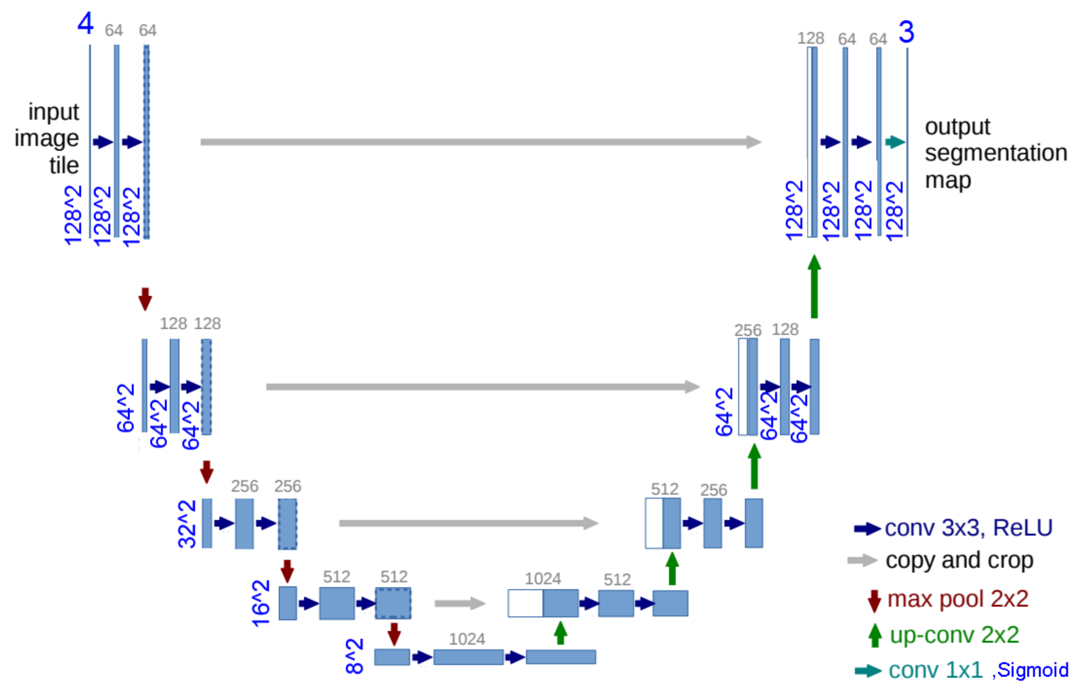

Figure 3.1: Modified U-net structure.

can just blend two images instead of three. No need to draw the lines, just show the source, the target, the cloning and the blending result. Choose your images from the internet. Do not use the images I provided, be creative :)

3 Inpainting by U-net

After assignment 1, I'm sure you are familiar with Pytorch and U-net [3] by now. In this section, you will do some modification to the code you already completed in assignment 1 to make the network do inpainting.

1. Both the input size and the output size of the network have changed to 128^2. See Figure 3.1. The main change is that the layer sizes are the same throughout each level, for example in the first level the layers are all 128^2, in the second level the layers are all 64^2. Therefore there is no need to centercrop the layers when doing skip-connections. Also, you need to add correct padding in each convolution layer, to make sure the input and output sizes are the same. The images in assignment 1 are gray-scale. The input/output layers are modified to adapt the network for the processing of colored images. Note: the input should contain 4 channels for RGB and mask, and the output should contain 3 channels for RGB. Note: the last layer is conv 1×1 then Sigmoid, not Softmax. (2 points)

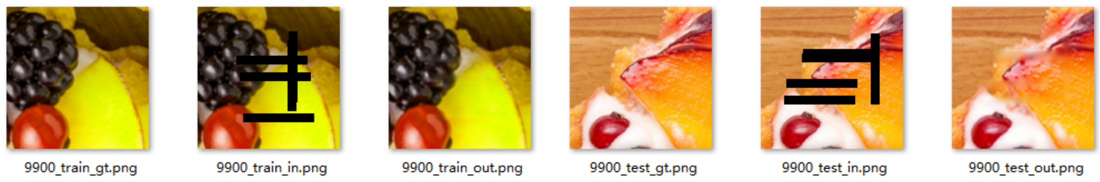

Figure 3.2: Sample saved images. From left to right: (99 epochs) train_groundtruth, train_input, train_output, test_groundtruth, test_input, test_output

2. The dataset I provided is a colored image. Modify the data-loader to randomly crop a small colored patch of the image for the network input. You need to apply data augmentations, including random horizontal and vertical flip, color jitter, random rotation, random resizing and random crop. Please refer to torchvision-transforms. Note that you need to place augmentation blocks in a correct order. (2 points)

3. The images in the dataset are complete, so we need to "augment" the dataset by introducing random holes to make them incomplete before feeding them to the network. Add another component in the data-loader, to randomly place small black rectangles in the image, and automatically generate a mask for this image. The rectangles should have random widths and heights in an appropriate range. In this assignment we fix a rectangle to be either 8×64 or 64×8, and we put 5 such rectangles randomly on the image. See an example in Figure 3.2. The mask should have white background and black rectangles indicating the areas that need inpainting. The data-loader should give out both network inputs (4 channels: RGB and mask) and targets (3 channels: RGB). Note: make sure the rectangle is not blurry on the boundary. (2 points)

4. In assignment 1, we use batch size 1, that is, in each iteration, we only input one image and output one image. For this assignment, you need to make the batch size to be 16. The dataloader should output $[16, 4, 128, 128]$ for the input and $[16, 3, 128, 128]$ for the ground truth. Modify the network if needed. Note: Before feeding to the network, the images and masks need to be normalized to $[0, 1]$. Important: for better results, please remove all batch normalization layers if you have any in your network. (1 point)

5. Modify the train.py, so that in the end of each training epoch, the network will save 6 images in the "samples" folder: train_groundtruth, train_input, train_output, test_groundtruth, test_input, test_output. train_groundtruth is a image patch from the training set; train_input is the "augmented" image after step 3; train_output is the network output. (1 point)

6. Modify the loss function to be MSE loss. (1 point)

7. Train the network for 100 epochs, each epoch has 100 iterations (since we only have one image for training).

What to submit: zip your codes into unet.zip.

What to show in the report: Show five sets of results in step 5: training 1 epochs, 5 epochs, 10 epochs, 50 epochs and 100 epochs. Organize the images in a similar way as Figure 3.2.

4 INPAINTING BY RESNET

In this section you will implement a resnet [1] for inpainting. I made it up for practicing purposes, so there's no guarantee that this network works better than U-net. Please do not deviate from the given network structure.

1. Implement residual block as shown in Figure 4.1 (4 points). For new layers please refer to TORCH.NN.

Note: conv or deconv layers have no bias term, set "bias=False".

Note: you should implement it as ONE class:

```
class resBlock(nn.Module):
    def __init__(self, inchannel, outchannel, downsample=False, upsample=False):
```

and call it as:

```
self.block1 = resBlock(inchannel, outchannel, False, False)
```

2. Implement the network as shown in Figure 4.2. (4 points)

3. Train the network for 100 epochs, each epoch has 100 iterations.

What to submit: zip your codes into resnet.zip.

What to show in the report: Show five sets of results: training 1 epochs, 5 epochs, 10 epochs, 50 epochs and 100 epochs. Organize the images in a similar way as Figure 3.2.

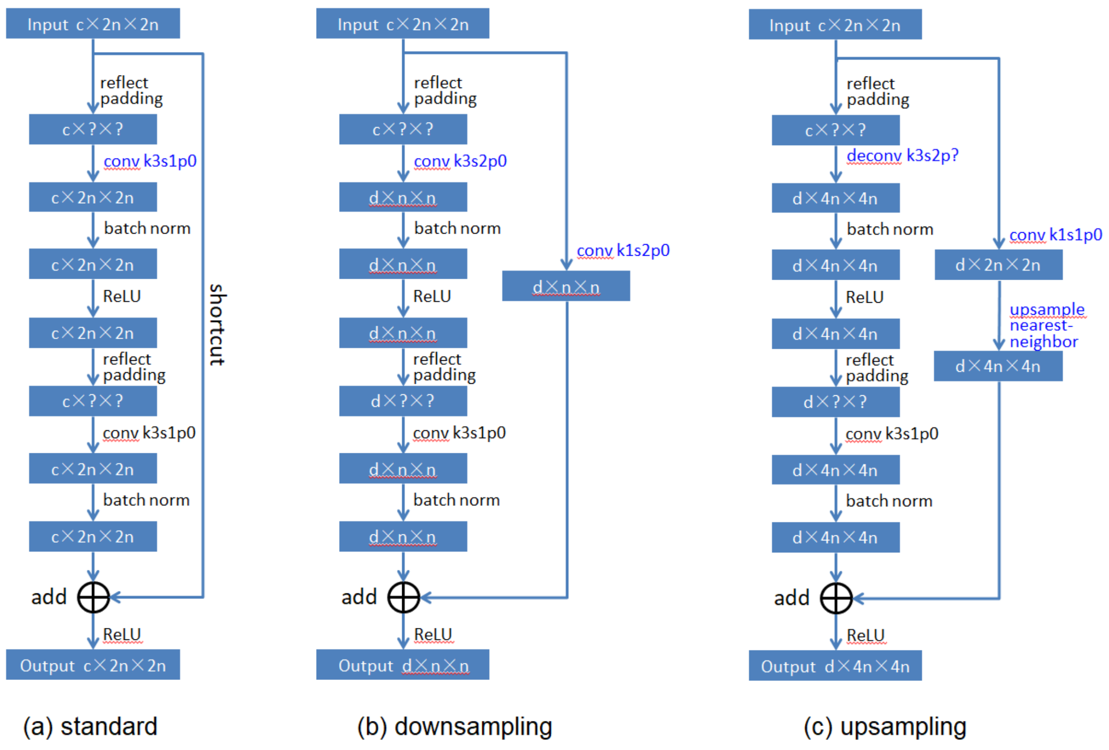

Figure 4.1: The structure of a residual block. We only consider three cases here: (a) input and output have the same shape; (b) downsample and change the number of channels; (c) upsample and change the number of channels. The differences between the three structures are highlighted so that you can integrate them into one single class. $convk3s1p0$ means a convolution layer with 3×3 kernel, 1×1 stride and 0×0 padding. The question mark represents some number you need to figure out.

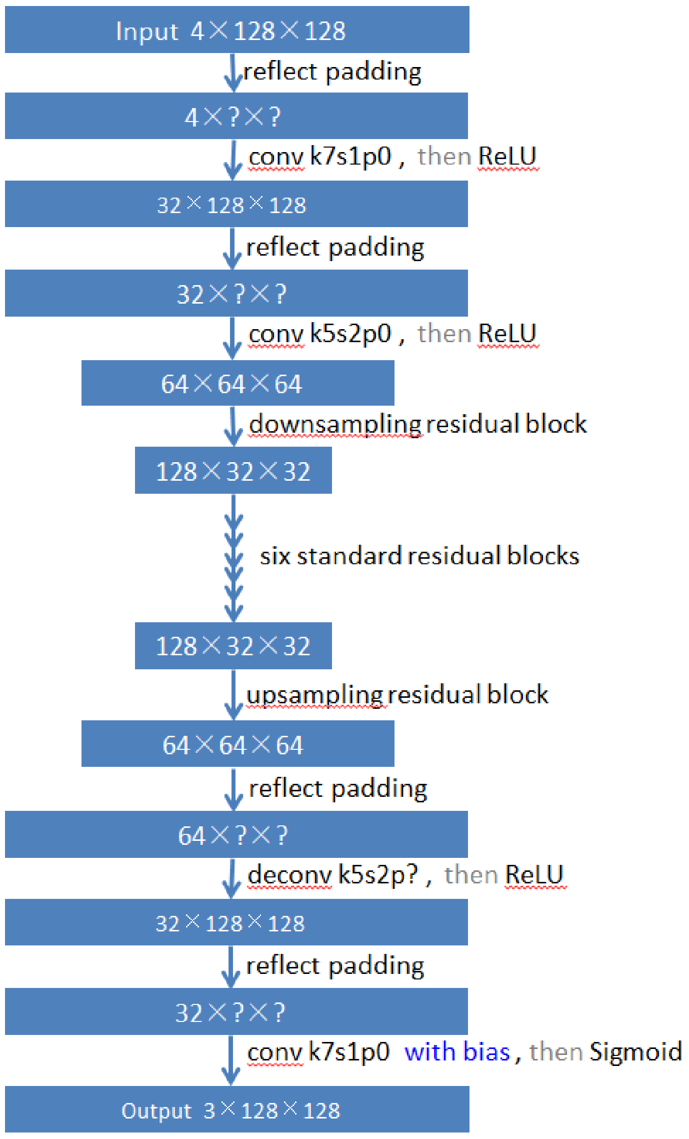

Figure 4.2: The structure of the inpainting network. Note that the last conv layer has bias term.

5 SVD Compression

In this section, you implement image compression using singular value decomposition. Use an $M \times N$ binary image I to form an SVD matrix, then drop 5, 10, and 15 smallest singular values and obtain a new image. Compare the PSNR of each of these cases against the initial image I. (3 marks)

What to submit: zip your codes into SVD.zip.

What to show in the report: The resulting compressed images for all three scenarios plus a table showing their PSNR.

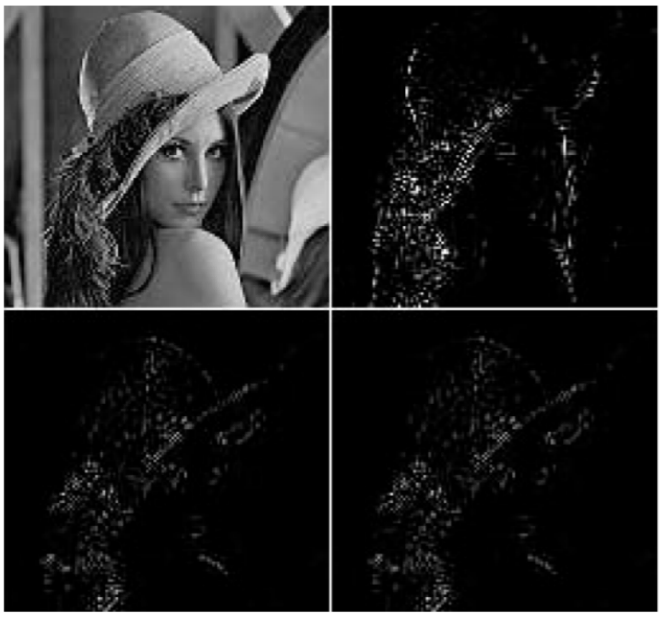

Figure 6.1: Lena image after applying Haar wavelets on rows and columns.

6 Wavelet Compression

In this section, you implement image compression using Haar wavelet.

1. Use a $2^n \times 2^n$ binary image I, implement binary Haar wavelet decomposition and reconstruction on rows and columns. You should be able to show the details (see Figure **??**). Your algorithm should be able to support Haar wavelet for n times. (2 marks)

2. Perform the same task as 1 but for **integer** Haar wavelet. (1 mark)

3. Calculate integer ternary Wavelet and show your work. Write an algorithm for reconstruction and decomposition based on filters you found. (2 mark)

4. Implement integer ternary Haar Wavelet and apply it on a $3^n \times 3^n$ image. Specification is the same as 1. (1 mark)

What to submit: zip your codes into Wavelet.zip.

What to show in the report: The resulting compressed images for all four scenarios plus a written your work for finding integer ternary Haar wavelet and its reconstruction and decomposition algorithms.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Transactions on graphics (TOG)*, 22(3):313–318, 2003.

[3] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.