

IAMI Project Report

Rishi Singhal, 2019194, rishi19194@iiitd.ac.in

Mini-Project 1: U-Net Implementation

As part of the U-Net project, I have used the LUNA 16 dataset to do the task of lung segmentation.

Dataset Description

Since the dataset size was quite huge for Google Colab, so I used 5 mhd files for the training set, 1 mhd file for the validation set, and 1 mhd for the testing set. Furthermore, I have only used 50% of the slices of each of the mhd files due to resources issue. Each slice is of size 512x512x1. Thus, we got the following sets size for both the lungs images and their corresponding segmented masks:-

Training set - (530, 512, 512, 1)

Validation set - (66, 512, 512, 1)

Testing set - (97, 512, 512, 1)

Drive link for the dataset is also attached [here](#).

Dataset Pre-processing

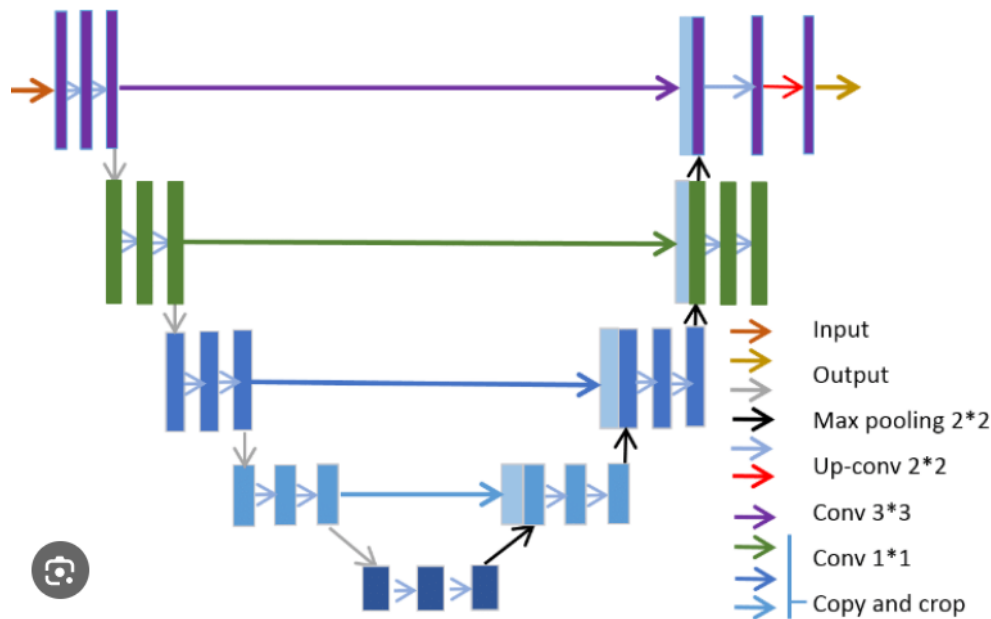
In the data pre-processing section we did the following steps:-

1. The intensity values in CT ranges between -1024 to 3071. However, for the task of lung segmentation, I can specifically focus on clipping the range between -1000 to 200.
2. I therefore truncated all the pixel values in the range [-1000 to 200].
3. Now, I normalized the output image between 0-1.

Some of the sample slices and their corresponding maps are as follows:

U-Net Model Architecture

I made a very simple and default U-Net model initially as shown in the picture below which takes in a 512x512x1 image returns a 512x512x1 segmented mask.



The 1st section is the encoder section that utilizes the 2D Convolution double blocks, followed by maxpooling. The 2D convolution double blocks utilizes filters of different size and kernels. I initially kept the filter sizes as [64,128,256] and kernel size as (3,3). The maxpooling kernel is also (2,2).

The 2nd section is the middle section that only has 2D convolution blocks with 512 filter size but no maxpooling.

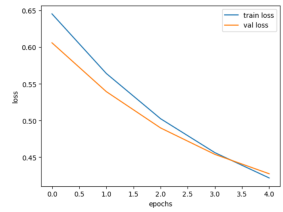
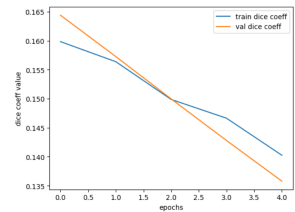
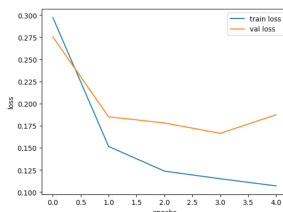
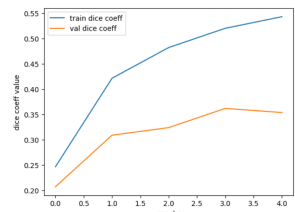
The 3rd section is the decoder section which does upsampling using 2D convolution blocks and `Conv2DTranspose()`. Also, there are skip connections from previous layers to preserve the loss from the previous layers so that they reflect stronger on the overall values. They are also scientifically proven to produce better results and lead to faster model convergence. In the final convolution block, we have a couple of convolutional layers followed by the final convolution layer.

I have initially made use of Binary Cross Entropy loss and $lr = 1e-4$ with Adam as the optimizer. Furthermore, I train my U-Net model for 4 epochs initially.

Results & Analysis

NOTE:- In the hyper-parameters tuning section I made use of a greedy approach, where I varied one hyper-parameter at a time and fixed the others. Then I choose the hyper-parameter on which I got the best results and went with the next hyper-parameters tuning.

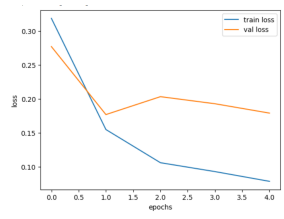
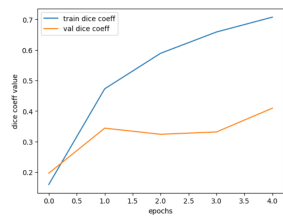
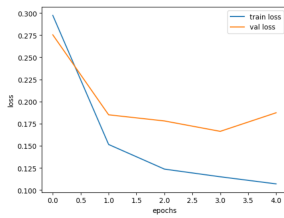
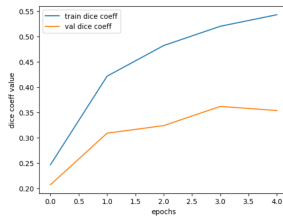
A.) Changing the filter sizes

Filter Sizes	Dice Values	Loss plots	Dice Coeff Plots
[64,128,256,512,1024]	Train - 0.1403 Val - 0.1358 Test - 0.1554		
[32,64,128,256,512]	Train - 0.5431 Val - 0.3536 Test - 0.4183		

I observe that using smaller filter sizes at each layer resulted in better dice coefficient values for train, val and test sets and also the loss is decreasing quite good. Smaller filter sizes mean that the model is not very complex because more complex models can lead to overfitting. Thus, I now fix the filter sizes as [32,64,128,256,512] and go to the next hyper-parameter.

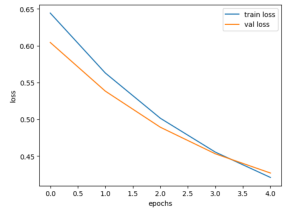
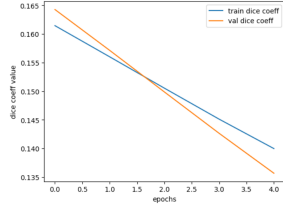
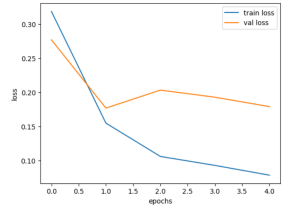
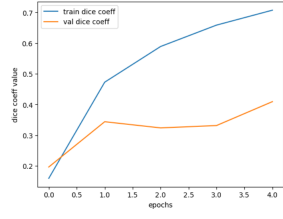
B.) Changing the learning rate

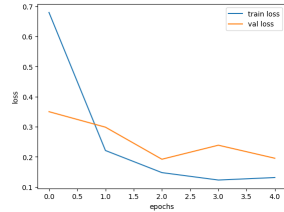
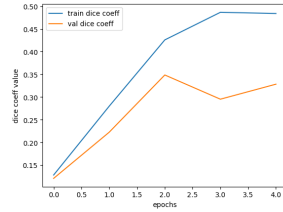
Learning Rate	Dice Values	Loss plots	Dice Coeff Plots
1e-2	Train - 0.5177 Val - 0.3435 Test - 0.3829		

1e-3	Train - 0.7071 Val - 0.4092 Test - 0.5046		
1e-4	Train - 0.5431 Val - 0.3536 Test - 0.4183		

I see that slightly increasing the learning rate, I get better dice coefficient values - i.e. is faster to converge to a local/global minima loss. But increasing it too much doesn't increase the dice coeff value as it simply moves up and down from the minima point and not converging quite enough. Thus, now I fixed the learning rate as 1e-3 and move forward.

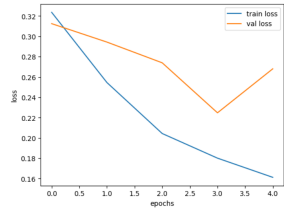
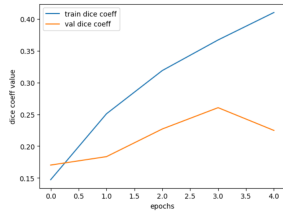
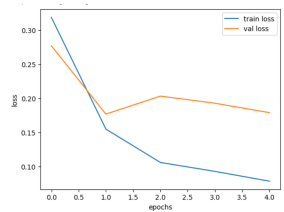
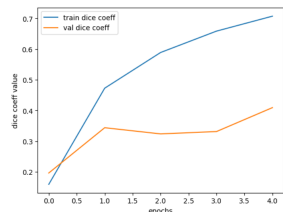
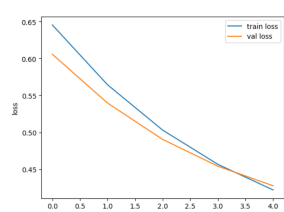
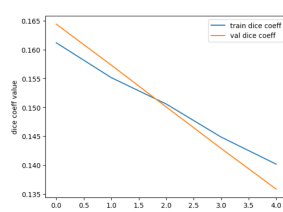
C.) Changing the Kernel Size

Kernel Size	Dice Values	Loss plots	Dice Coeff Plots
2	Train - 0.1402 Val - 0.1357 Test - 0.1365		
3	Train - 0.7071 Val - 0.4092 Test - 0.5046		

4	Train - 0.4840 Val - 0.3283 Test - 0.3800		
---	---	--	---

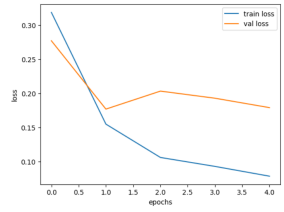
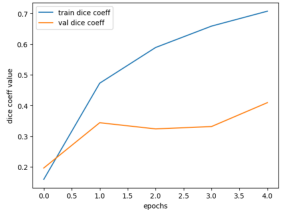
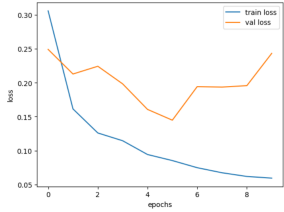
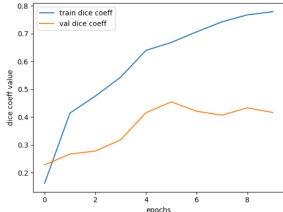
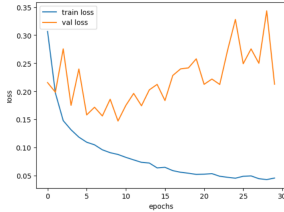
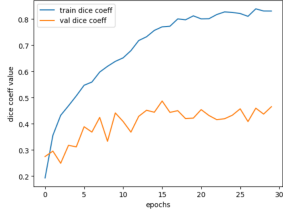
I observe that decreasing the filters kernel size is not good for the setup. However also increasing it beyond 3 is also not that beneficial. Thus, the default setting of kernel size of 3 is the best.

D.) Changing the Depth of the U-Net Model

Depth and Filter sizes	Dice Values	Loss plots	Dice Coeff Plots
4 [32,64,128,256]	Train - 0.4103 Val - 0.2247 Test - 0.3198		
5 [32,64,128,256,512]	Train - 0.7071 Val - 0.4092 Test - 0.5046		
6 [32,64,128,256,512,1024]	Train - 0.1401 Val - 0.1358 Test - 0.1366		

I observe that increasing and decreasing the model depth significantly is not that good, as it could lead to overfitting/underfitting. Thus, the default model depth of 5 is sufficient - resulting in the best metric values.

E.) Changing the number of epochs

Number of epochs	Dice Values	Loss plots	Dice Coeff Plots
4	Train - 0.7071 Val - 0.4092 Test - 0.5046		
10	Train - 0.7790 Val - 0.4196 Test - 0.5216		
30	Train - 0.8305 Val - 0.4651 Test - 0.5094		

By increasing the number of epochs, I observe that epochs of 30 resulted in over-fitting as the validation loss shoots too much.

But with epochs 4 the model seems to not getting trained that much.

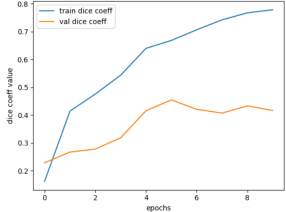
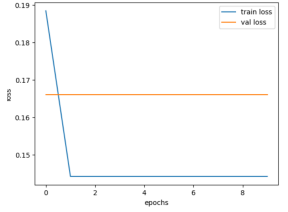
Thus, training the model with 10 epochs resulted in better metrics values.

F.) Changing the loss function

I made use of 2 more different loss functions - dice loss and SSIMLoss other than the Binary Cross Entropy Loss.

The structural similarity (SSIM) index provides a measure of the similarity by comparing two images based on luminance similarity, contrast similarity and structural similarity information.

While, Dice coefficient is a measure of overlap between two masks. 1 indicates a perfect overlap while 0 indicates no overlap. Dice Loss = 1 — Dice Coefficient.

Loss function	Dice Values	Loss plots	Dice Coeff Plots
Dice Loss	Train - 0.0076 Val - 1.033e-4 Test - 3.56e-5		
Cross-Entropy Loss	Train - 0.7790 Val - 0.4196 Test - 0.5216		
SSIM Loss	Train - 0.0152 Val - 1.033e-4 Test - 3.56e-5		

I observe that both SSIM-Loss and Dice Loss doesn't perform well for the given lung segmentation task. Thus, I proceed with Cross-Entropy loss itself.

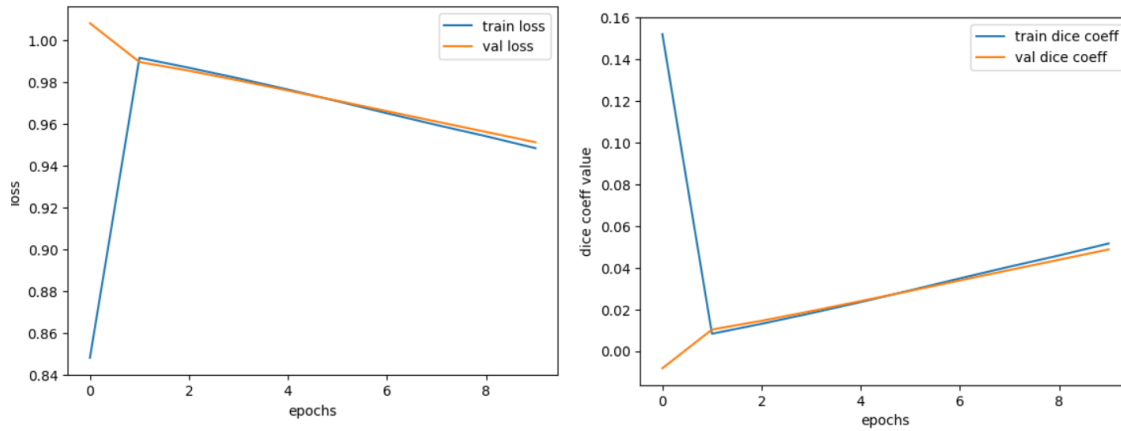
I also tried one more hypothesis proposed by [Nieradzik et al.](#) where they claim that tanh activation function is better than sigmoid function for dice loss at the end layer. Thus, I changed the activation function accordingly and trained and test the model.

Train Dice Coeff - 0.0517

Val Dice Coeff - 0.0488

Test Dice Coeff - 0.0676

From comparing the dice values in the sigmoid case, the hypothesis does hold true.



G.) Adding Dropout

I added dropout after every pooling and Conv2DTranspose() functions to see if I observe any improvement in my metrics.

Dropout Value	Dice Values	Loss plots	Dice Coeff Plots
0	Train - 0.7790 Val - 0.4196 Test - 0.5216		
0.25	Train - 0.5813 Val - 0.3739 Test - 0.4650		
0.5	Train - 0.1165 Val - 0.1075 Test - 0.1103		

I see that adding dropout doesn't seem to improve the performance - maybe the model is fitting fine with no dropout. But yes, I can maybe reduce the number of dropouts used - i.e., add dropout at only some layers (future work)

Thus, finally after all the hyper-parameter tuning I got the best dice coefficient metrics values of:

Train - 0.7790

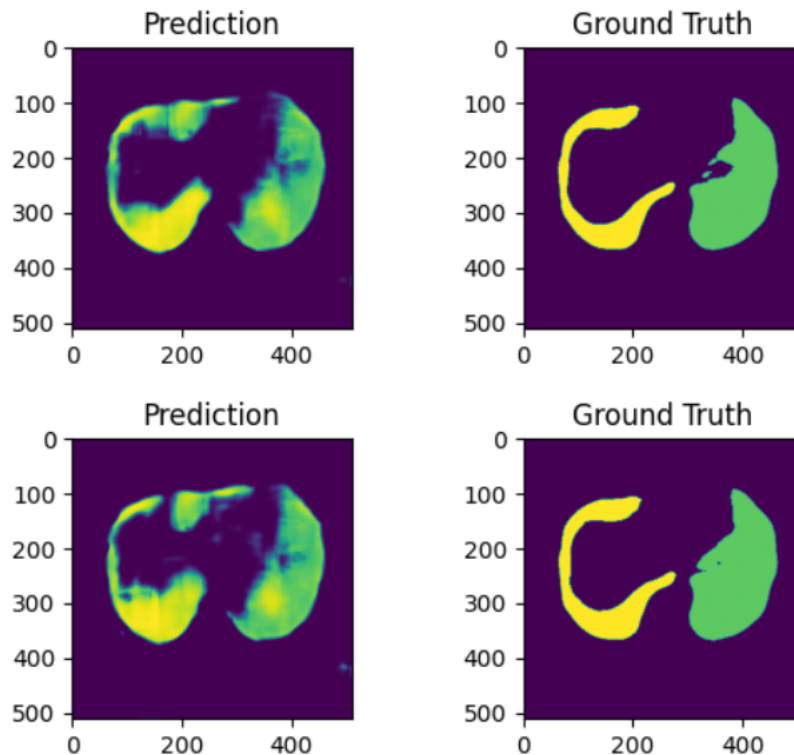
Val - 0.4196

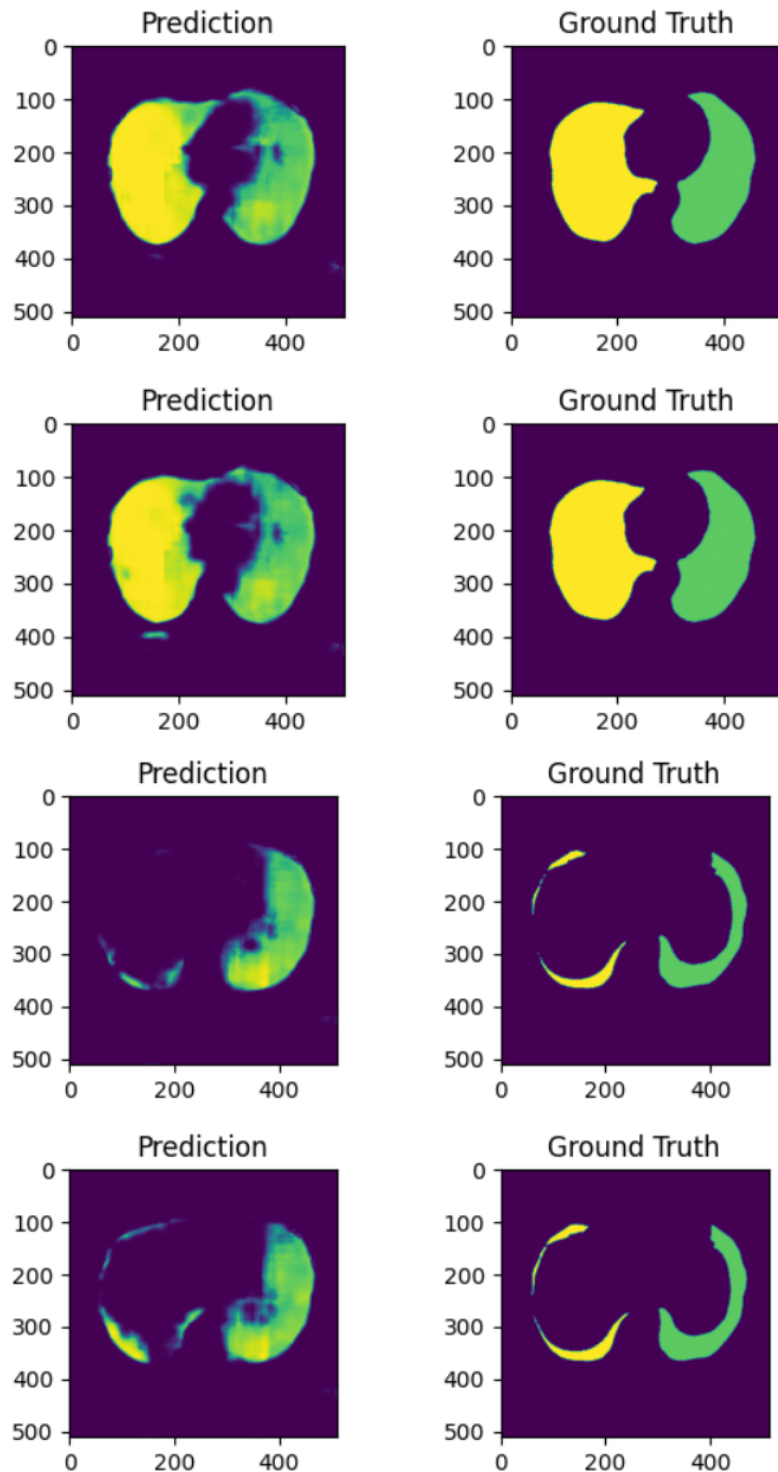
Test - 0.5216

With the following hyper-parameters:-

- 1.) Model depth - 5
- 2.) Filter sizes - [32,64,128,256,512]
- 3.) Learning rate - 1e-3 and Adam Optimizer
- 4.) Filter kernel size - 3
- 5.) Binary Cross-Entropy Loss function
- 6.) Number of epochs = 10
- 7.) No dropout, i.e., 0 dropout value

Also, I am attaching a few of the predicted lung masks prediction by my best model





As part of the future I would like to work more with the dice loss and SSIM Loss functions as they are supposed to be good loss functions for image segmentation task. I would also like to increase the sizes of train, validation and test sets to see how well the model generalize for a bigger dataset.

Mini-Project 2: AutoMap Implementation

We will use the same lungs CT dataset as used in Project 1, i.e., 14 train files, 4 test files, and 2 validation files. Here each file (mhd file) has $512 \times 512 \times 164$ 2D sample slices.

A.) Image Resizing

Now, for the implementation of auto-map without exceeding memory limits we will resize the slice to size 64×64 . To do so, we have 2 different methods:

1.) Cropping the data to 64×64 patches:

Here I would crop the original images into 64×64 patches and then use it for computing the complex FFT data.

2.) Resampling the data to 64×64 patches:

Resampling 2D data refers to changing the resolution or size of the data by interpolating. For example, if we have a 2D image with dimensions 256×256 pixels and we want to resample it to a smaller size of 64×64 pixels, we can achieve this by applying a resampling algorithm that reduces the number of pixels in the image while preserving the essential features of the image.

Regarding which is better, I don't think we can comment on which will be the best in total because of the following reasons.

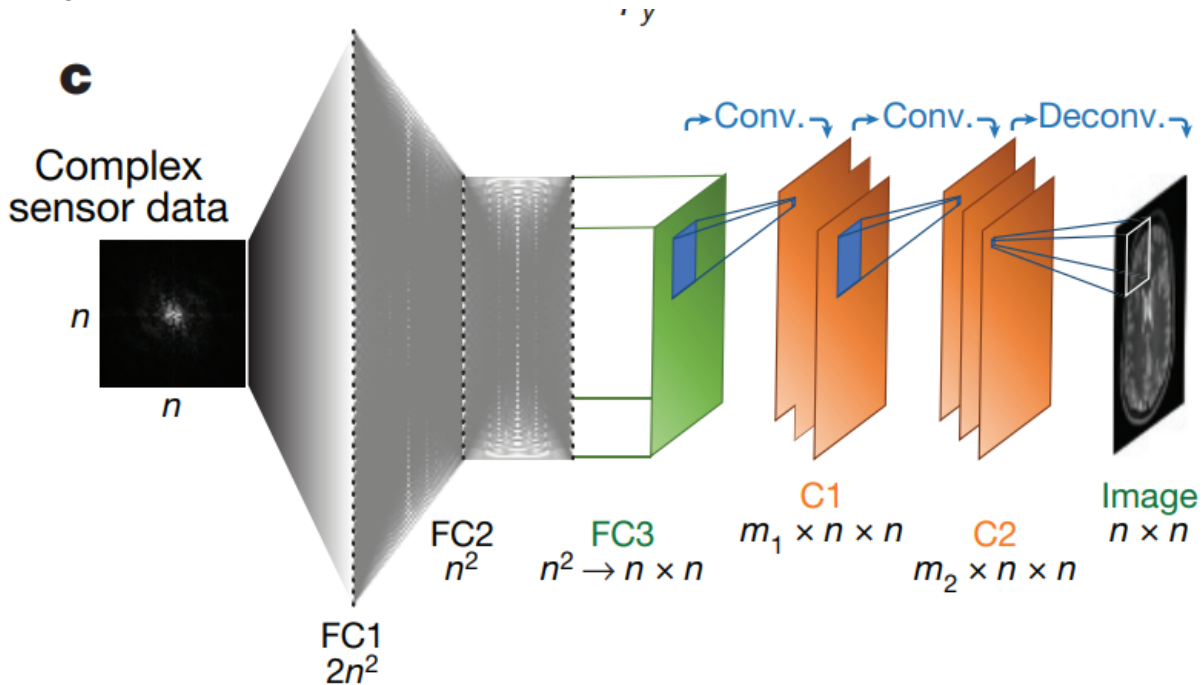
Cropping the data to a smaller size can help remove noise and irrelevant information from the input data, leading to a cleaner signal and potentially more accurate results. However, if the crop size is too small, important information may be lost, which could negatively affect the learned Fourier transform.

On the other hand, resampling the data to a fixed patch size can ensure that all data points have the same dimensions and resolution, which can simplify the learning process and potentially improve the generalization of the learned Fourier transform. However, this method could also introduce artifacts and distortions in the data, which could negatively affect the results.

Thus, we can't really conclude which strategy would be the best without putting into effect in the code for the given task.

B.) Model Architecture

For the automap model implementation I will use the model architecture proposed by [Zhu et al.](#). The figure for their model is as follows:



After getting complex FFT data the AUTOMAP model passes the $N \times N$ complex data through a FC1 layer of size $n^2 \times 2n^2$ and then passes it through FC2 of size $2n^2 \times n^2$. Both FC1 and FC2 uses tanh-activation function. Then through another FC3 layer of $n^2 \times n^2$.

Then comes convolution blocks with ReLU function as the activation function where it uses 2 convolution blocks of m_1 and m_2 filters. Finally, there's a deconvolution block which results in a $n \times n$ reconstructed image.

For ease of implementation, we can refer to the code available at [Github](#) that makes use of the above model architecture itself.

The Github code uses the following model configurations:

1.) At first, they read all the images in the dataset folder. For each image they resize it to 64x64 using `cv2.resize()` function and then stores all these resized images in a list.

```

# Resize images
final_images = np.abs(final_images)
final_images_resized = np.zeros((64,64,final_images.shape[2]))
for i in range(final_images.shape[2]):
    final_images_resized[:, :, i] = cv2.resize(final_images[:, :, i], (64,64))

```

2.) Then for each resized image it computes the k-space data by taking the resized image FFT. The complex data is of the form (1,64,64,2) where the complex data has both real and imaginary part channels of size 1x64x64.

3.) It considers the model architecture as follows:-

Step 1.) It first flattens the (num_imgs x 64 x 64 x 2) to (num_imgs x 64*64*2) format. Then it passes it through a FC1 with input size of (num_imgs x 64*64*2) and output size of (num_imgs x 64*64). The FC1 layer uses tanh activation function.

Step 2.) Then, it passes it through FC2 with input size of (num_imgs x 64*64) and output size of (num_imgs x 64*64) with tanh as the activation function.

Step 3.) Now, they reshape the output size to (num_imgs x 64 x 64 x 1) and passes it through a convolution block with input size of (num_imgs x 64 x 64 x 1) and output size of (num_imgs x 64 x 64 x W0). Here W0 is the number of filters which is 64 in this case with kernel size as 5x5. Also, the convolution block is followed by the ReLU activation function.

Step 4.) Next, it passes the output of the 1st convolution block through another convolution block consisting of 64 filters and kernel size of 7x7. The output size of the block is (num_imgs x 64 x 64 x W1), where W1 = 64. It is then followed by the ReLU activation function. Finally, the output of the 2nd convolution block is passed through another convolution block(or deconv block) with number of filters as 1 and kernel size of 7x7. The output of the same is (num_imgs x 64 x 64 x 1) - the reconstructed image.

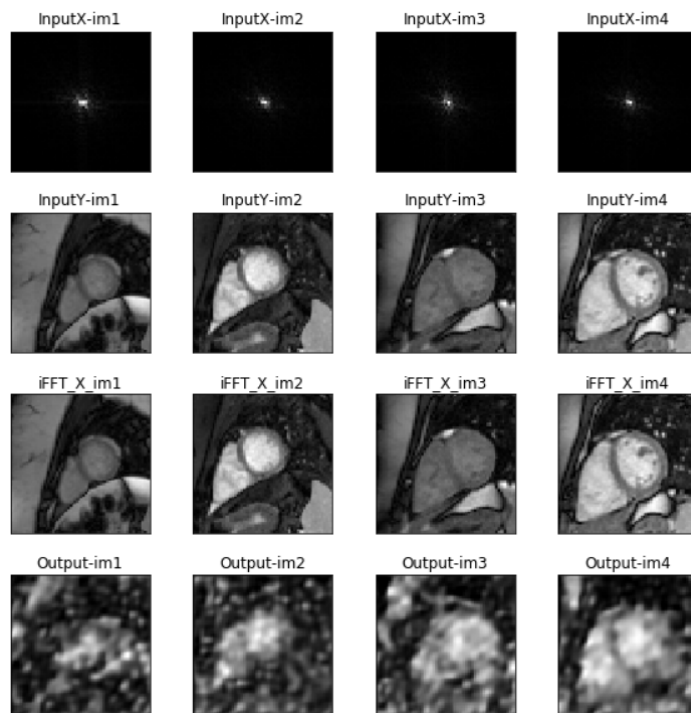
Step 5.) The model uses RMSOptimizer as the optimizer function with a learning rate of 0.0001 and learning decay of 0.9. As the loss function it makes use of L2 Loss.

```
def compute_cost(DECONV, Y):
    """
    Computes cost (squared loss) between the output of forward propagation and
    the label image
    :param DECONV: output of forward propagation
    :param Y: label image
    :return: cost (squared loss)
    """

    cost = tf.square(DECONV - Y)

    return cost
```

4.) The Github Code then compares the reconstructed image with the ground truth image & images reconstructed using iFFT. A few of their sample results are shown below.



They have said that their reconstructed images are blurring because they are using lesser images for training (~5000 images). Also, it could be the case that they haven't followed the model architecture proposed by [Zhu et al.](#)

C.) Bonus Question

1.) Why did we choose 64x64 patch size? What is limitation of AutoMap?

We choose 64x64 patch size (quite small) for AutoMap because in general, a smaller patch size can lead to faster training and inference times, as well as a reduction in the complexity of the model. In AutoMap the model complexity scales quadratically as the input size of the image increases. This is also because of the limitation of AutoMap as it's not scalable due to too many model parameters since it doesn't exploit the property that the original Fourier kernels are linearly separable.

2.) Can it be designed better?

Yes, AutoMap can be designed better as explained by [Schlemper et al.](#) They exploit the property that original Fourier kernels are linearly separable and thus replaced the FC layers with decomposed transform (DT) layers, which are fully-connected along one axis. For two-dimensional input, they applied the DT layer twice, once for each axis. Each DT block is activated by ReLU nonlinearity. DT blocks are followed by a sparse convolutional autoencoder. In this way their proposed dAutoMap is scalable as it has made the model complexity to almost linear with increasing image size. This is further explained in detail in their paper.

3.) Highlight differences training a crop data sample vs resampled data sample data network

The answer to this is same as made in Part A.