Chest X-Ray Lung Segmentation using Deep Learning

Analytics Academy: Level-3 Deep Learning (Image Analytics) Certification Report

Author: Shivaprasad Kallimani (GE Power, SSO: 105035167)

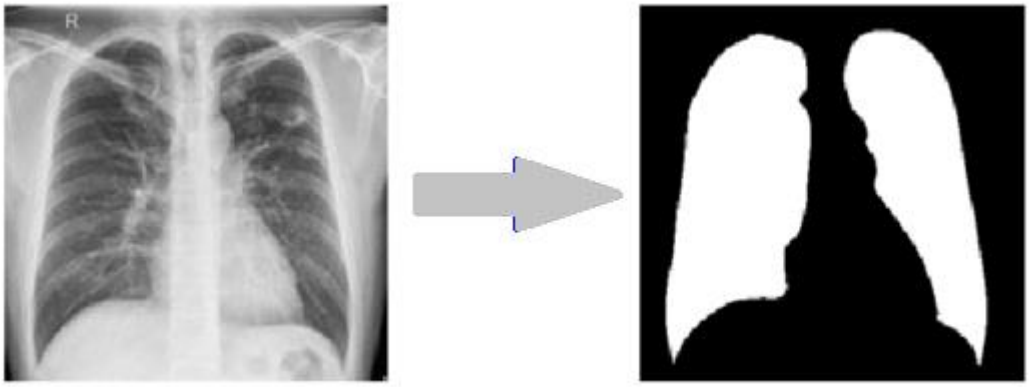


TABLE OF CONTENTS

X-Ray Lung Segmentation using Deep Learning

1. Preface

Chest X-ray (CXR) is an important imaging test used in the detection and diagnosis of many pulmonary diseases affecting the respiratory system like pneumothorax, tuberculosis, pneumonia, COVID-19 etc. As an example, tuberculosis (TB) is a chronic infectious disease worldwide (8 million cases per year) and remains a major cause of death globally. TB is especially common in populations which are resource poor and have weak healthcare systems. In such a scenario, it is important for a radiologist to have access to tools that can screen patients for diseases. An important first step in any computer-aided (CAD) system to detect pulmonary diseases is the automatic segmentation of lungs. An altered or compromised shape of the lung is a clear indication of abnormality and helps in quick triage of cases.

Automatic lung segmentation from X-ray images is a challenge because of the variety in contrasts, anatomical differences across subjects, multitude of scanning protocols and inhomogeneity in lung region. In this exercise, the participants will use a deep learning-based approach to segment both the left and right lung from an X-ray image.

2. Problem definition

Develop an algorithm to segment the x-ray images of lung to develop a mask that enables extraction of section portion of the image that is associated with the lung.

2.1 Datasets used

1. X-rays collected under Montgomery County's Tuberculosis screening program. Dataset consists of total 138 Main lung images, corresponding manual masks and clinical readings.
2. JSRT Dataset consisting of 247 lung images and corresponding manual masks.
3. Shenzhen dataset consisting of 476 X-Ray images and manual masks.
4. Belarus X-Ray dataset with 77 X-Rays and corresponding manual masks.

2.2 Objective of current assignment

1. Develop a U-Net deep learning model to perform the segmentation task for both the left and the right lung. Use Montgomery dataset for this step to train the deep learning model.

2. To check domain adaptation ability of previously developed deep learning model, test it on a new, but slightly different data set, and see how the network performs. If the performance degrades, develop additional algorithm to improve the performance. Use complete JSRT dataset as an unseen test data for this step. If the performance degrades on these datasets, then either tune the original model/method to improve accuracy or use 30% of JSRT dataset as training data for domain transformation tasks and re-test the accuracy using previously developed model. If the model accuracy is still unacceptable, tune and/or enhance base leaner model for acceptable accuracy on Montgomery, JSRT, Shenzhen and Belarus sets.

3. Ensure that developed models are stable and performance should remain same for any randomly chosen data subsets. Use Sørensen–Dice coefficient as metric to evaluate the similarity between predicted segmentation mask and the actual ground truth mask.

3. Datasets used, Methodology and Output metrics

3.1 Datasets used in this report

There are 4 different X-Ray datasets along with masks used in this report – Montgomery (138), JSRT (246), Shenzhen (476) and Belarus (77 X-rays). The X-Rays in each of these datasets look different in appearance, texture, contrast etc. This helps to check robustness and adaptation capability of segmentation model developed in this study.

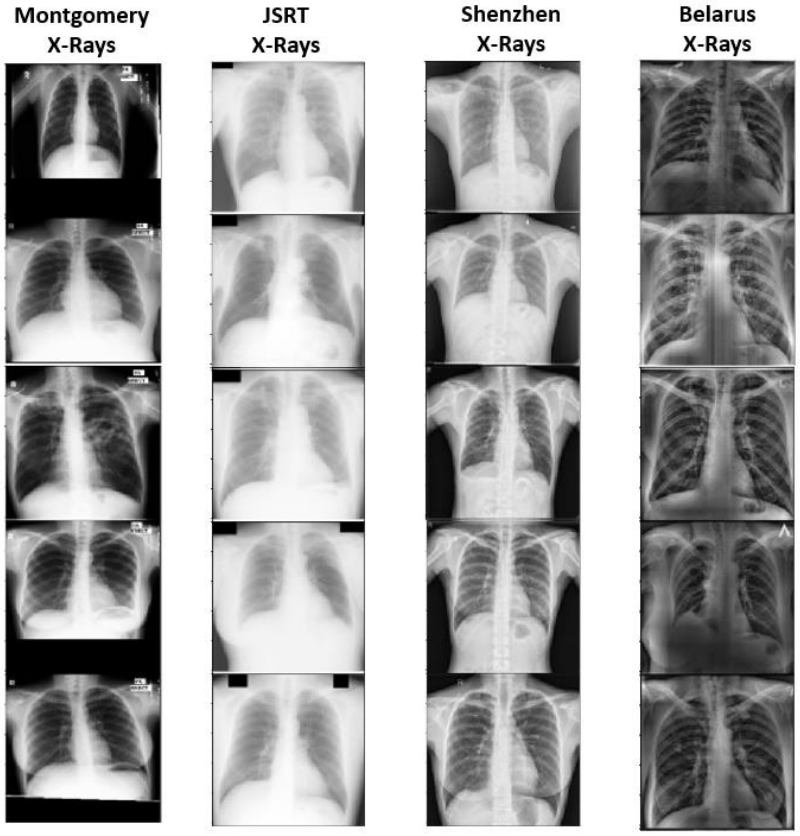


Figure 1: Chest X-Ray Datasets from different domain

3.2 Assumptions and method of solution

1. Pre-processing of images performed only to convert to square size and downsize to 256X256 pixels. **No image transforms or histogram equalization algorithms like clahe etc., applied initially as these images will be used for domain translation later. The intent is to develop a generalized deep learning model which suits various domains of segmentation for same task.**

2. Montgomery X-Ray dataset is used as the source for training the model. This dataset consists of 138 images which are split into ratio of 0.80 : 0.20 for training and testing respectively. This gives 110 X-Ray images for training and 38 images or testing. **The model is trained using K-fold cross validation - training data (110 images) is split into 5 folds.** Minor data augmentation on images like rotation, translation, shear and scaling will be applied during training process. Evaluation metric chosen is "Dice coefficient" which is same as F1 score in classification.

3. U-Net Deep learning architecture is used for segmentation task. This base learner model is trained from scratch using only Montgomery images and tested both on Montgomery and full JSRT datasets. The dice coefficient and segmentation plots are documented as outputs.

4. The model is expected to show poor performance on unseen domains like JSRT images. This is because the images are different in terms of texture, pixel intensity, brightness and contrast compared to Montgomery images. Hence, a domain transformation technique is applied using GAN (Generative Adversarial Networks). Specifically, CycleGAN is used to transform JSRT images to "look like" Montgomery images, and then the model is tested on transformed JSRT images to check whether model accuracy improves. Approximately 30% of JSRT dataset (80 images) and 80 images from Montgomery set are used for training CycleGAN. Use the remaining 70% of JSRT images and use CycleGAN model to translate to Montgomery like images and re-test on the base learner model.

5. If the improvement in segmentation task is not satisfactory, **base learner model (U-Net) needs to be enhanced and trained (without using JSRT, Shenzhen and Belarus data)**. In this report, base learner model is upgraded by adding up-sampling layers similar to U-Net++ architecture and replacing down sampling layers with higher level transfer learning models like EfficientNets as backbone. The features extracted at different levels of EfficientNet-B7 architecture are used as encoders for U-Net which greatly helps is segmentation accuracy. This method increases the computation time multiple times compared to base learner model, though the accuracy of model will be much better.

6. The enhanced base learner model U-Net+EfficientNetB7 is trained on Montgomery images and tested on JSRT data similar to previous steps. Additionally, model is tested on two more datasets - Shenzhen and Belarus X-Ray datasets. Also, check if model performance further increases when image enhancements like "clahe" is applied and model is retrained.

3.3 Performance Evaluation Metric – IOU or Dice?

Intersection over Union (IoU or Jaccard Index) and Dice Coefficient (F1 Score) are two commonly used metrics for semantic segmentation. Both these metrics range between 0–1 (0–100%) with 0 signifying no overlap and 1 signifying perfectly overlapping segmentation.

Difference between these metrics is that IoU tends to penalize bad segmentations more than Dice and overall errors look inflated compared to what Dice calculates. IoU tends to show numerically smaller values than Dice even though both are similar representation of segmentation performance estimations.

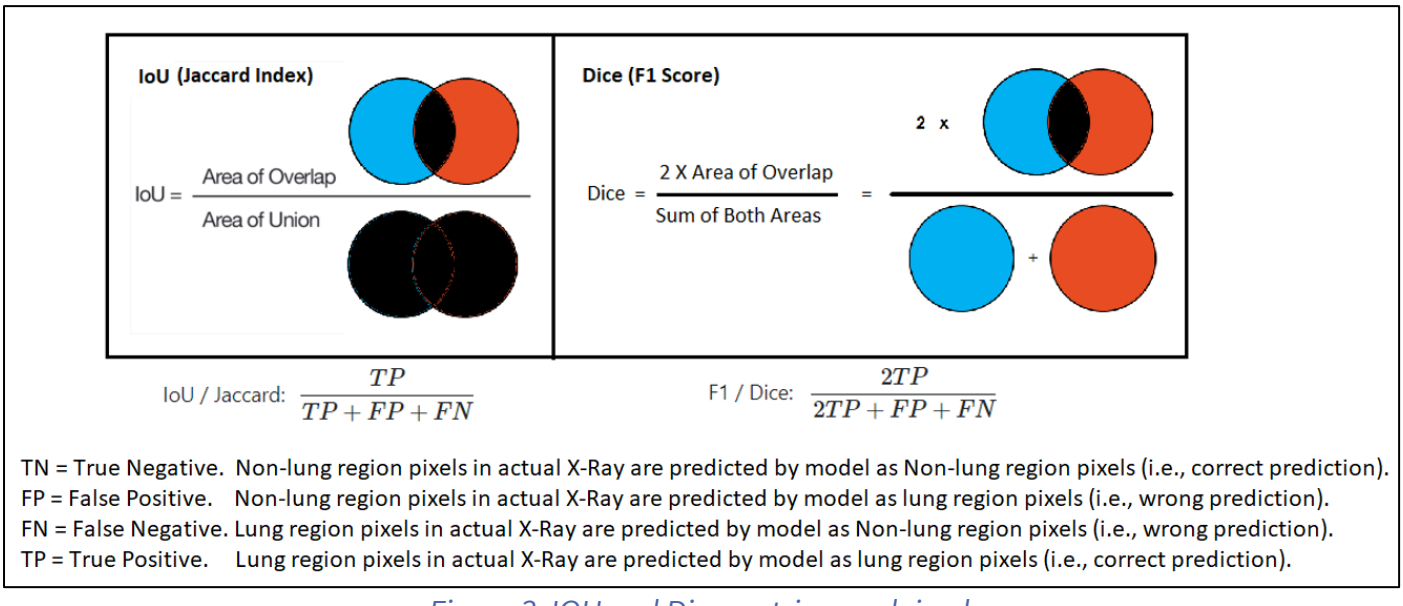


Figure 2: IOU and Dice metrics explained

In current report, "Dice Coefficient" will be used as performance measurement metric for lung segmentation tasks.

4. Develop and train a U-Net Architecture from scratch

U-Net CNN is built with grayscale as input image with dimensions (256,256,1) and trained from scratch on 80% Montgomery X-Ray images (110 images) using 5-fold cross validation approach. Model is tested to extract segmented lung regions on 28 Montgomery X-Rays, and also directly on another dataset of JSRT consisting of 246 X-Rays. Dice coefficient and standard error is calculated by comparing the predicted lung segmentations with ground truth lung masks.

4.1 U-Net Architecture details

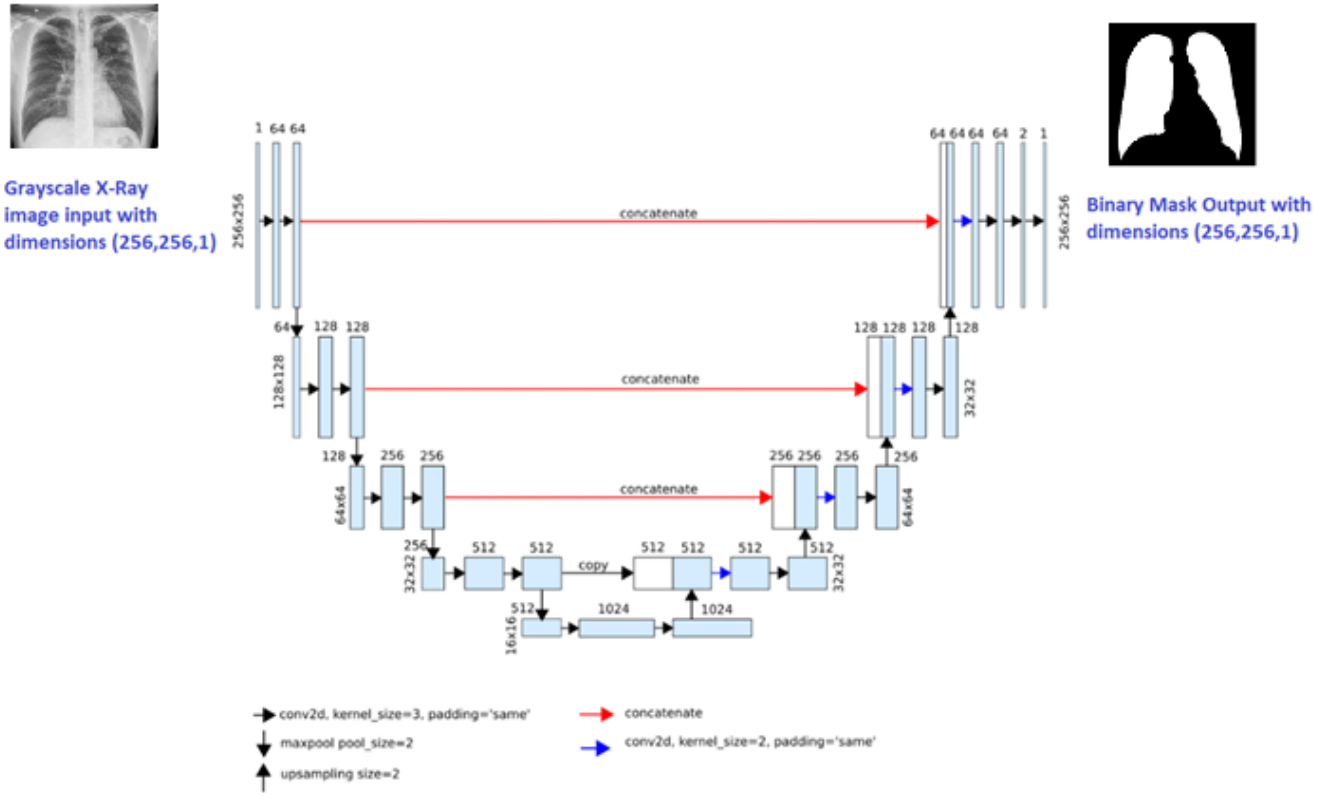


Figure 3: U-Net Architecture

Figure shows simple U-Net Architecture applied to a grayscale image to calculate a binary segmentation mask as the output. Given an input image, in this case a grayscale X-Ray image, the U-Net model produces a binary mask of 1s and 0s, where 1 indicates a lung region and 0 indicates non-lung region. This is a **semantic segmentation task** where model assigns an object category label to each pixel in the image. In this dataset of X-Ray images, lung pixels are coloured white and background pixels are coloured black.

U-Net is an end-to-end fully convolutional network (FCN); it only contains convolutional layers and does not contain any Dense layer because of which it can accept image of any size. The network consists of a contracting path and an expansive path, which gives it the u-shaped architecture. The contracting path is a typical convolutional network that consists of repeated application of convolutions, each followed by a rectified linear unit (ReLU) and a max pooling operation.

The main idea is to supplement a usual contracting network by successive layers, where pooling operations are replaced by up-sampling operators. Hence these layers increase the resolution of the output. A successive convolutional layer can then learn to assemble a precise output based on this information.

One important modification in U-Net is that there are a large number of feature channels in the up-sampling part, which allow the network to propagate context information to higher resolution layers. As a consequence, the expansive path is more or less symmetric to the contracting part, and yields a u-shaped architecture. The network only uses the valid part of each convolution without any fully connected layers. This tiling strategy is important to apply the network to large images, since otherwise the resolution would be limited by device memory.

U-Net architecture basically works by combining the features learnt in lower level of network with features learnt in higher level. During the contraction (down-sampling), the spatial information is reduced while feature information is increased. The expansive (up-sampling) pathway combines the feature and spatial information through a sequence of up-convolutions and concatenations with high-resolution features from the contracting path.

4.2 Training details, Test Results & Discussions:

U-Net model is trained using 'Adam' optimizer, metric as 'accuracy' and loss function of binary cross entropy. Learning rate is set using keras class **ReduceLROnPlateau** which benefits the model by reducing the learning rate when learning stagnates without improvement in metric.

The trained model weights for every fold are saved through keras checkpoints and later used for averaged segmentation predictions on test X-Rays. Dice coefficient or F1 score is used to evaluate segmentation (pixel level classification) accuracy between predicted and ground truth masks.

Below image shows accuracy and loss values improving over epochs for one of the folds during training.

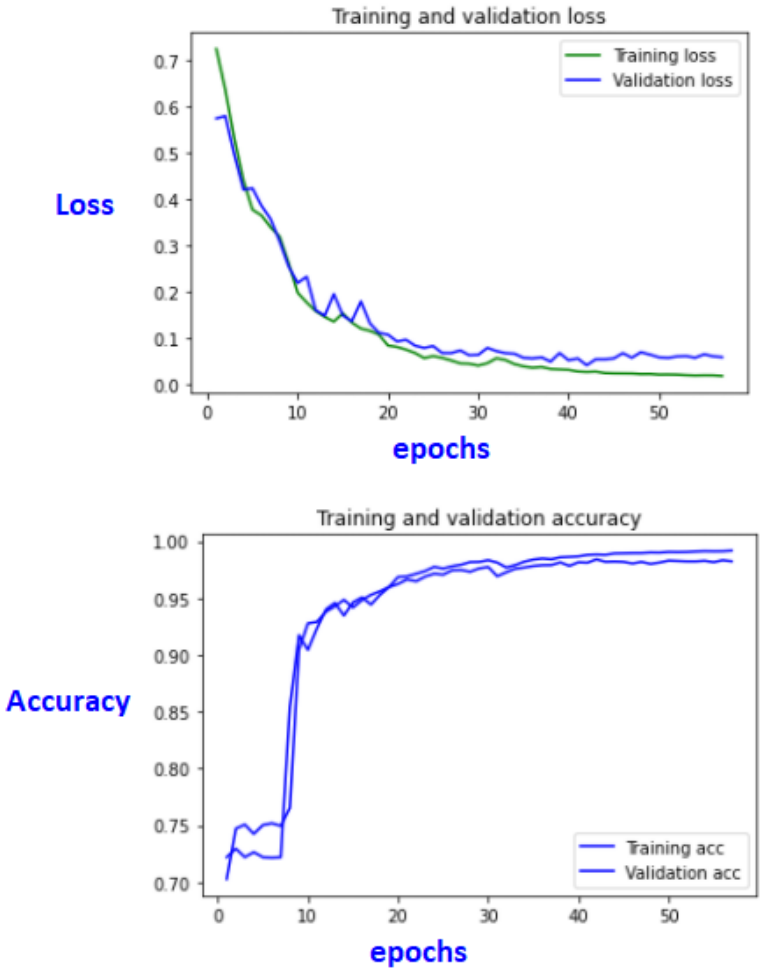


Figure 4: Training and Validation metrics (Increasing accuracy and reduced loss) plotted

	Simple U-Net Architecture trained from scratch	DICE Metric on test set		
Trials	Description	Montgomery	JSRT	
0	**Baseline - SupportNET Architecture (GRC) Estimations**	**0.956**	**0.964**	**Benchmark**
1	**U-Net with 256X256 single channel image. WITH 5-FOLD CV.**	**0.97427 +/-0.009**	**0.8879 +/-0.047**	

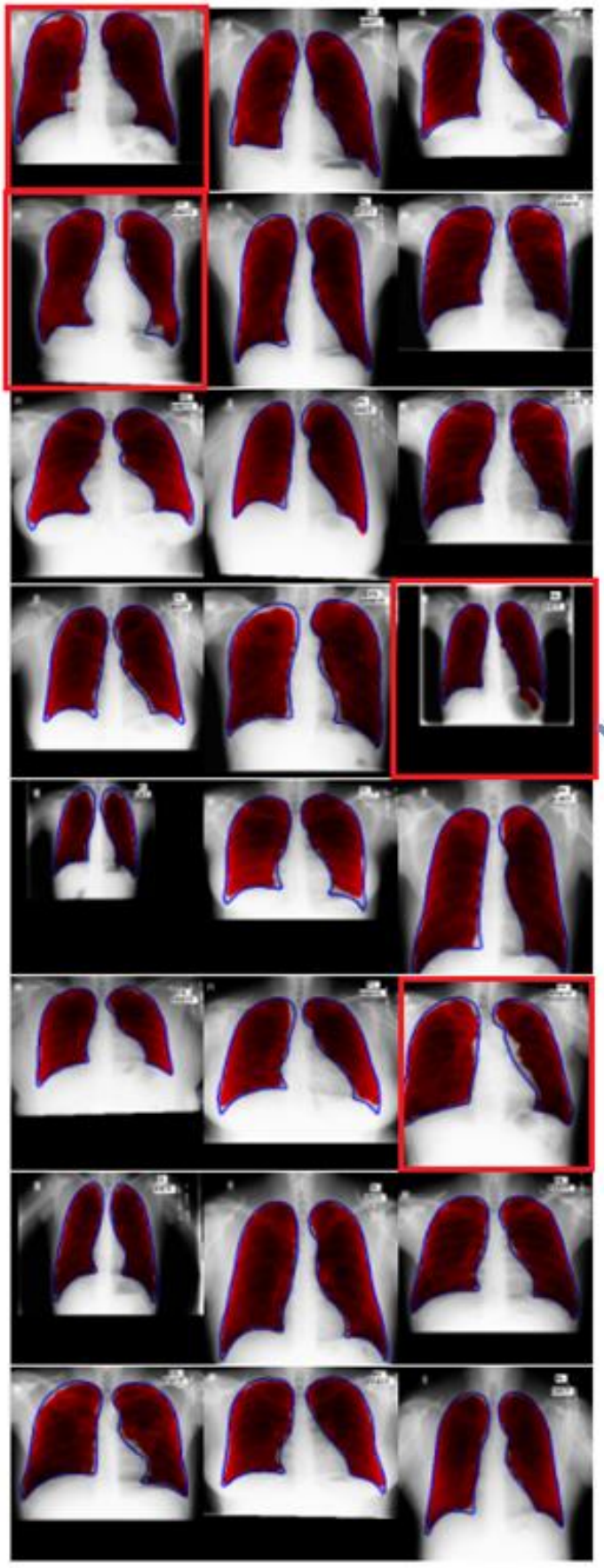
Table 1: U-Net model results compared with SupportNet

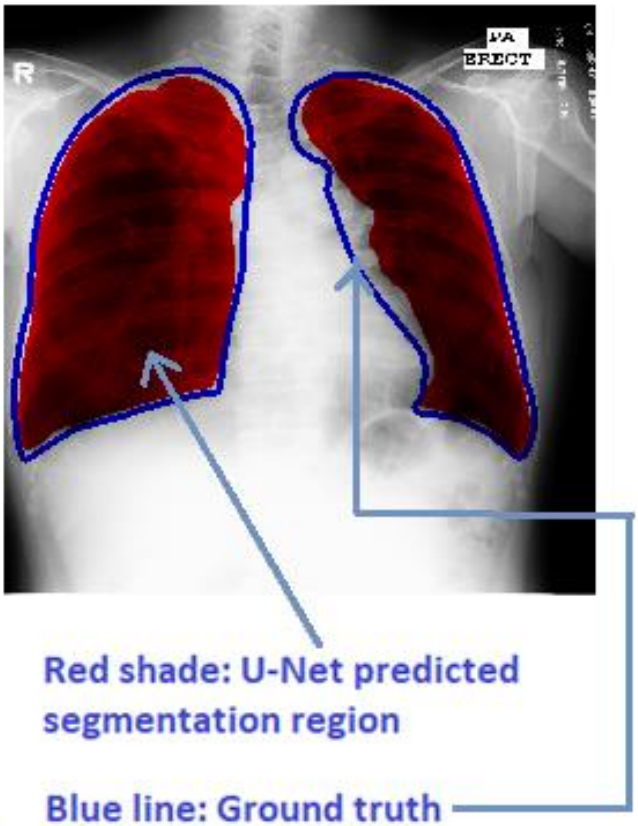
Table above shows the results for base learner U-Net model tested on 28 Montgomery X-Rays and 246 JSRT X-Rays. Mean Dice coefficient is calculated individually for both sets and **compared with results of GRC SupportNet model (considered as benchmark for highest accuracy).**

Mean Dice coefficient calculated for Montgomery test set is 0.9743, produces well segmented lung masks for most X-Rays as shown in section 4.3. This appears better compared to GRC SupportNet at first look, however, the dice calculated on entire JSRT set is poor - 0.8879, and this does not produce acceptable segmentation for JSRT X-Rays which is from other domain (due to different device technical characteristics which lead to different contrast, noise etc., in X-Ray images). Few predictions for JSRT X-Ray lung masks are presented in section 4.4 to observe how poor the segmentation is being calculated using this model.

The base learner U-Net model overfits and works only on Montgomery data, and fails to generalize on datasets from different domain like JSRT.

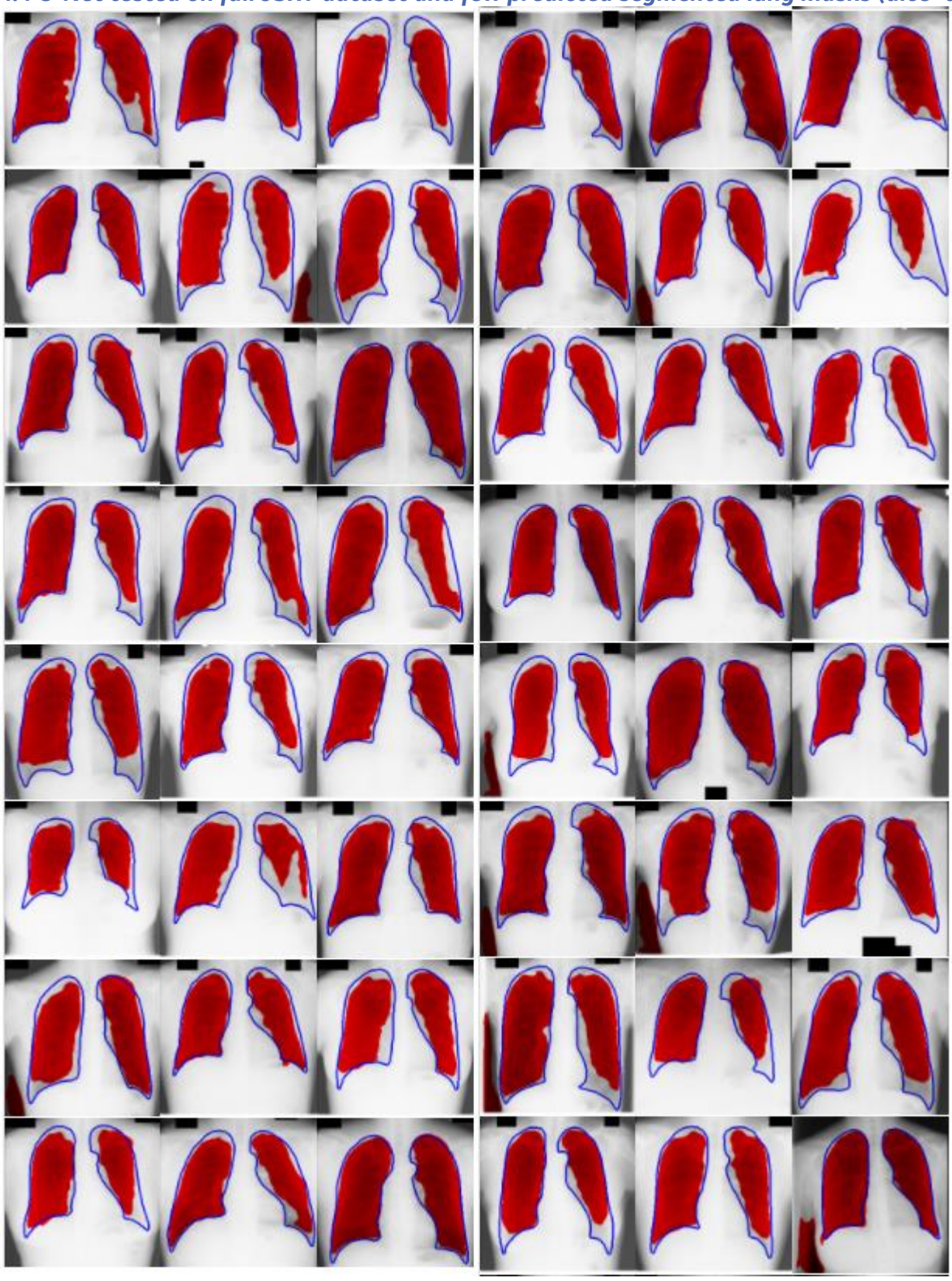
4.3 U-Net tested on Montgomery test data and predicted segmented lung masks (dice=0.9743):





Most of Montgomery X-Rays appear to be well segmented for lung masks except for 2-3 cases.

4.4 U-Net tested on full JSRT dataset and few predicted segmented lung masks (dice=0.8879):



As discussed earlier, JSRT X-Rays are from different domain and not used in training the base learner U-Net. The U-Net is purely trained on Montgomery images and fails to generalize on different domain data like JSRT. **As shown above, most of JSRT X-Rays show unacceptable lung mask segmentations.**

In next section, an attempt has been made to convert JSRT X-Rays to "look like" Montgomery X-Rays through a deep learning network called CycleGAN. This is expected to increase the segmentation accuracy on JSRT set.

5. Domain Transformation using Generative Adversarial Networks

CNNs for medical imaging suffer from performance degradation for new medical images with different pixel intensity distributions compared to training data sets. This unstable performance change makes it impossible to generalize that CNNs for medical domain, because it is impossible to obtain a data set that considers all conditions in the image-shooting environment. Also, because of the infinite number of new data sets with a variety of pixel intensities, training new ones to the network each time is a very expensive task. Therefore, in order to solve this problem practically, generalization of new data sets needs to be considered.

The task of transforming image data from an arbitrary domain into a target domain is known as image-to-image translation, a kind of domain adaptation. Traditional methods like histogram equalization and histogram matching are used to adjust similarity of intensity distribution to source dataset and this does not always work well for all datasets. This is where GAN (Generative Adversarial Networks) come into play.

5.1 GAN application for domain transformation to improve JSRT segmentations

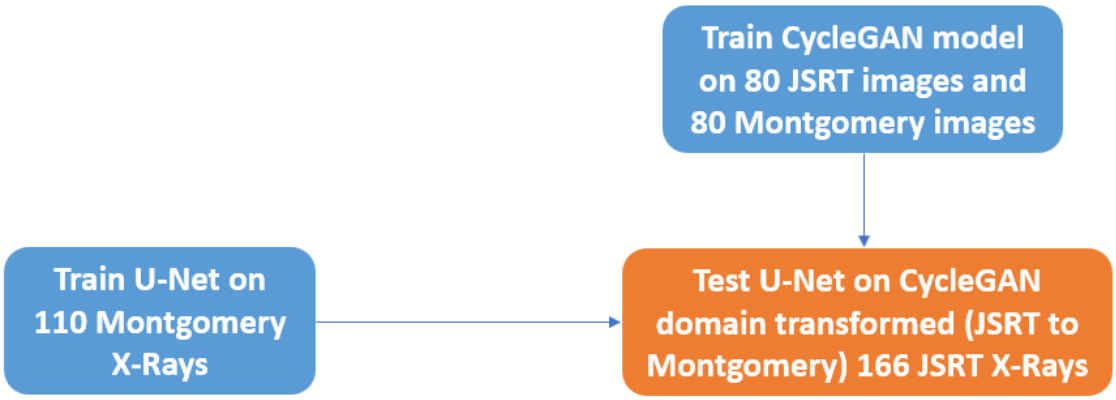


Figure 5: Train and Test plan for CycleGAN domain transformed JSRT images

In this study, a GAN structure called CycleGAN is applied on JSRT images which performs image to image translation to convert the JSRT X-Rays to “appear like” Montgomery images. Once the JSRT images are transformed through CycleGAN, the base learner U-Net model is tested on converted images, and results show improved segmentation accuracy for JSRT X-Rays.

To train CycleGAN, 30% of JSRT images (80 X-Rays) and 80 images from Montgomery set are used. The trained CycleGAN model is then used to convert the remaining 166 JSRT images to Montgomery like images. Subsequently, the U-Net model trained during first stage (section 4) used as-is for segmentation prediction on CycleGAN converted 166 JSRT X-Rays.

In next sections, fundamentals of CycleGAN structure and how it effectively works on image-to-image translation is explained.

5.2 CycleGAN Unpaired Image to image translation:

Most basic GAN (pix2pix) performs image-to-image translation using a paired dataset. For each image in the original domain, the paired data set contains an image converted to the target domain. While training the networks, it is not easy to get paired dataset for all domains of medical imaging. For such domain transformation tasks of unpaired image to image translation, a generalized method is to apply CycleGAN.

The CycleGAN is the widely used network for style transfer tasks. For medical images, structure of a CycleGAN can be used to generalize the intensity distribution.

For this project, we want to convert JSRT domain images to Montgomery domain images (since Montgomery dataset was used for training). Hence source domain is JSRT (referred as 'J') and target domain is Montgomery (referred as 'M'). Generator '$G_{J->M}$' translates JSRT image to Montgomery image, and '$G_{M->J}$' translates Montgomery image to JSRT image. Below sequence of operations describe how CycleGAN works.

1. First, the **generator $G_{J->M}$** learns to transform image from JSRT to Montgomery, and generates $G_{J->M}(J)$ a fake Montgomery image. The **discriminator 'D_M'** compares this generated fake image w.r.t actual Montgomery domain and classifies as real or fake. Next, **another generator $G_{M->J}$** obtains a reconstructed image, $G_{M->J}$ ($G_{J->M}$ (J)), using previously transformed (fake Montgomery) image into the original JSRT domain. By reducing the loss between the original source JSRT domain image and the reconstructed JSRT image, the network maintains the characteristics of the original domain. The discriminator and generator models are trained in an adversarial zero-sum process, like normal GAN models. The objective of the generators is to learn to better fool the discriminators and the discriminator should learn to better detect fake images. Together, the models find an equilibrium during the training process, and simultaneously mean absolute error between real image vs. final reconstructed images should be minimized.

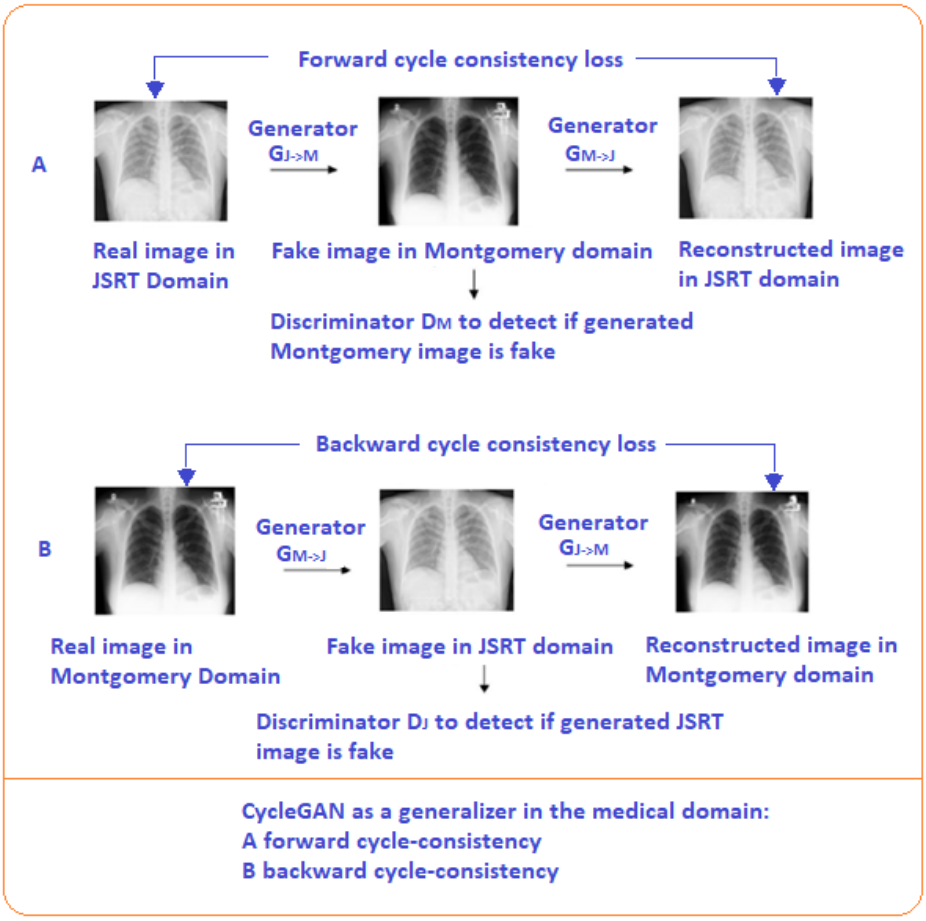


Figure 6: CycleGAN domain transformation Process

2. Conversely, the same method applies to converting from a target Montgomery domain to the original JSRT domain. Generator $G_{M->J}$ used in step 1, transforms the Montgomery image to a fake JSRT image. And then generator $G_{J->M}$ used in step 1, reconstructs the fake Montgomery image to reconstructed Montgomery image.

3. The discriminator models are trained directly on real and generated images, whereas the generator models are not. Instead, the generator models are trained via their related discriminator models. Specifically, they are updated to minimize the loss predicted by the discriminator for generated images marked as "real", called **adversarial loss**. As such, they are encouraged to generate images that better fit into the target domain.

4. The generator models are also updated based on '**cycle loss**' which measures how effective the generator models are at the regeneration of a source image when used with the other generator model. Since there are two cycles of image translation as explained in step 1 and 2, two separate cycle losses will be calculated by comparing original vs reconstructed image. Loss in step1, is **Forward cycle-consistency loss** while converting JSRT domain to Montgomery domain and then retransforming back to JSRT domain. Loss in step 2, is **Backward cycle-consistency loss**, is the loss when converting from the target Montgomery domain to the source JSRT domain and then back to the target Montgomery domain. CycleGAN combines both forward–backward cycle-consistency losses.

5. Apart from adversarial and cycle consistency losses, there will also be **identity loss**. Identity loss says that, if you fed a target image itself to generator which translates source to target, it should yield the real target image or something close to target image. Ex., if you fed a Montgomery image itself to generator which translates JSRT to Montgomery, it should yield the real Montgomery image or something close to Montgomery image.

6. Altogether, each generator model is optimized via the combination of four outputs with four loss functions. Ex., below list of losses is for step 1 where JSRT image is translated to Montgomery.

   - Adversarial Loss (L2 or mean squared error):
     JSRT -> [Generator-$G_{J->M]}$ -> Montgomery -> Discriminator-$D_{Montgomery}$ -> [real/fake]
   - Identity loss (L1 or mean absolute error):
     Montgomery -> [Generator-$G_{J->M]}$ -> Montgomery
   - Forward cycle loss (L1 or mean absolute error):
     JSRT -> [Generator-$G_{J->M]}$ -> Montgomery -> [Generator-$G_{M->J]}$ -> JSRT
   - Backward cycle loss (L1 or mean absolute error):
     Montgomery -> [Generator-$G_{M->J]}$ -> JSRT -> [Generator-$G_{J->M]}$ -> Montgomery

For current CycleGAN model, two subcases will be run, first using a U-Net CNN as generator and second using ResNet CNN as generator. Both CycleGAN models (CycleGAN_UNet and CycleGAN_ResNet) will be applied on JSRT images and tested for lung segmentation accuracy using base learner U-Net developed is section 4.

5.3 Results of CycleGAN transformed JSRT images (to Montgomery domain):

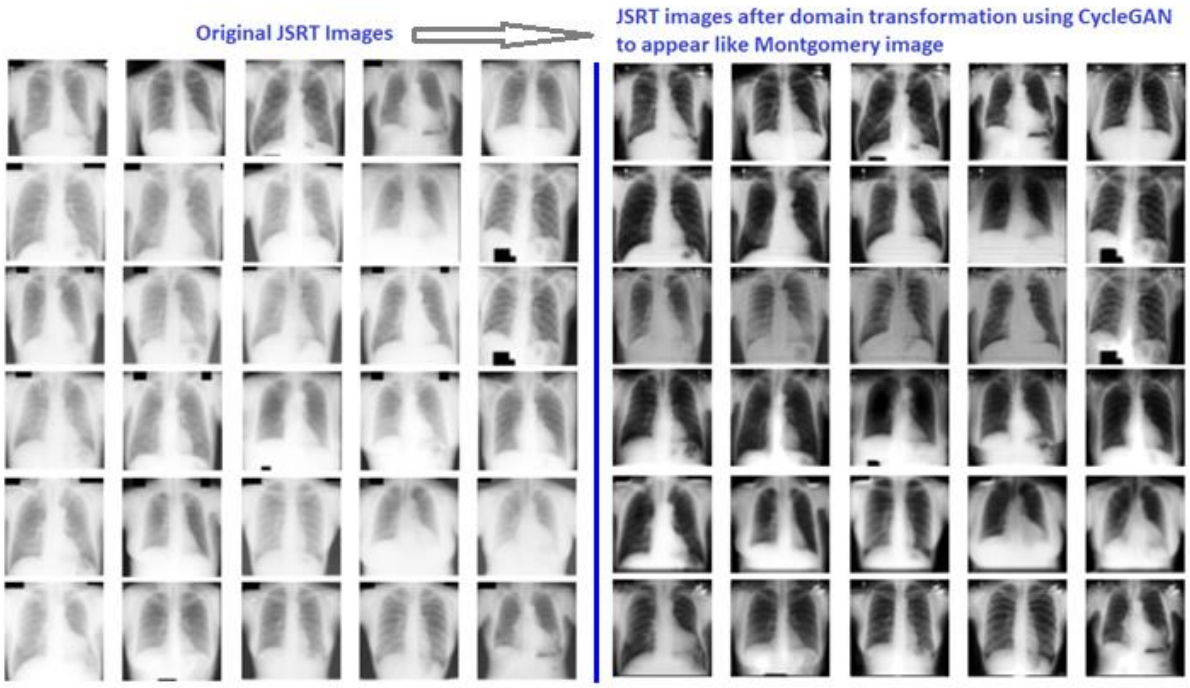


Figure 7: JSRT X-Rays transformed to Montgomery domain through CycleGAN

Above set of images show original JSRT images transformed to look like Montgomery domain images after applying CycleGAN. The transformation works very well by increasing image contrast between lung and non-lung regions. Original JSRT images appear quite hazy lung regions. After CycleGAN transformation, lungs look darker similar to Montgomery images and this results in better pixel level classifications and segmentation accuracy.

5.4 Results and Discussions:

As explained in section 5.1 through 5.3, the CycleGAN transformed JSRT images will be tested using base learner U-Net model developed in section 4. Dice coefficient is calculated on 166 JSRT test images and reported in table below for trial 2 and 3.

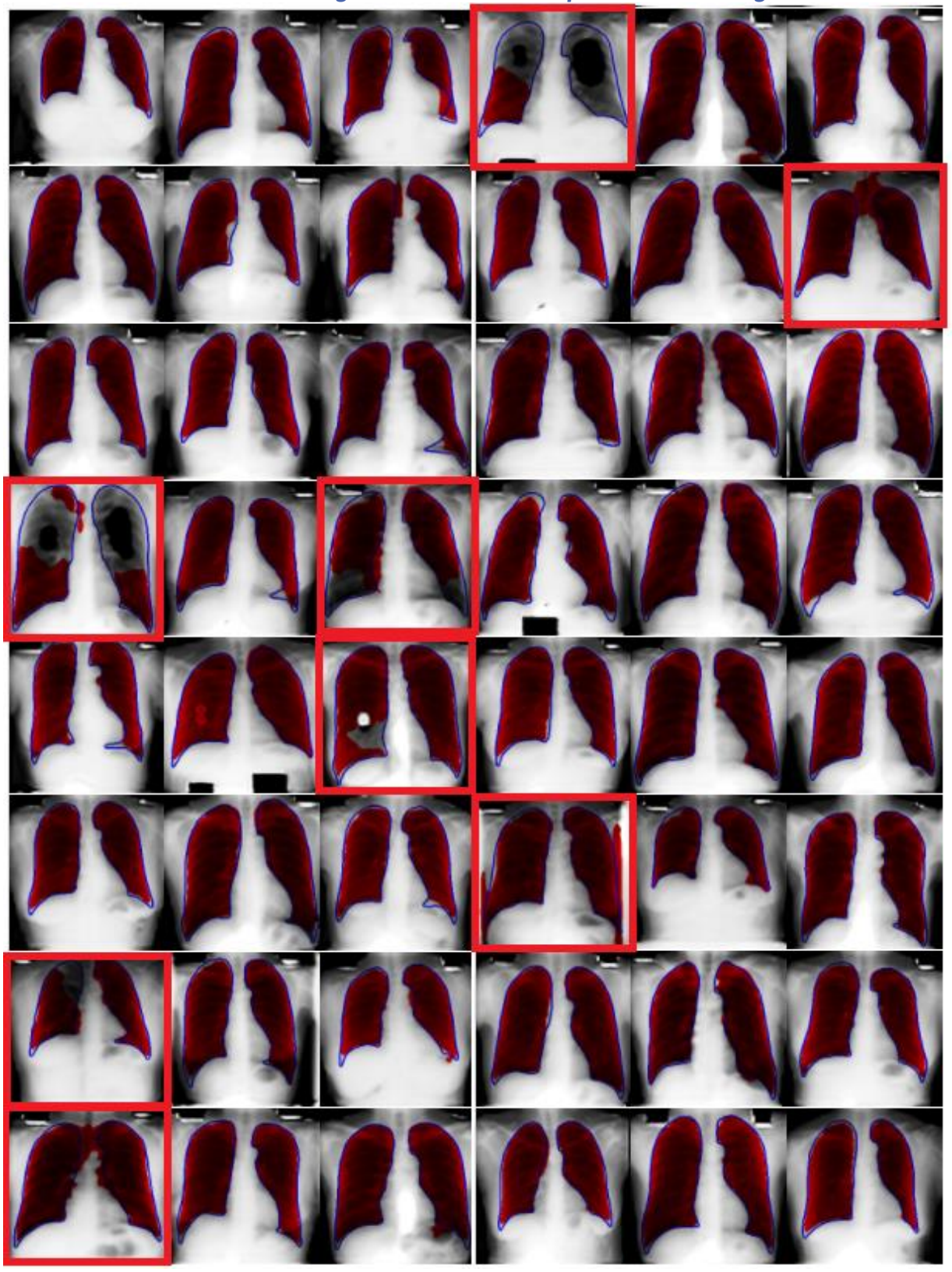
	Simple U-Net Architecture trained from scratch	DICE Metric on test set		
Trials	Description	Montgomery	JSRT	
0	Baseline - SupportNET Architecture (GRC) Estimations	0.956	0.964	Benchmark
1	U-Net with 256X256 single channel image. WITH 5-FOLD CV.	0.97427 +/-0.009	0.8879 +/-0.047	
2	CycleGAN_UNET transformed JSRT images tested using U-Net model in Trial 1	Not applicable	0.93599 +/-0.107	
3	CycleGAN_RESNET transformed JSRT images tested using U-Net model in Trial 1		0.93839 +/-0.081	

Table 2: JSRT Result improvements using CycleGAN Domain transformation

The dice metrics for trial 3 and 4 show much improvement compared to trial 1 which was a direct application of base learner U-Net model on original JSRT X-Rays. This indicates that CycleGAN transformed JSRT images to source Montgomery domain helped in better segmentation. Further, between two subcases of CycleGAN generators, **the CycleGAN based on ResNet generator showed slightly better segmentation accuracy compared to CycleGAN using U-Net generator**. The actual segmented images are shown in section 5.4.1 and 5.4.2 which are much better than previous predictions presented in section 4.4.

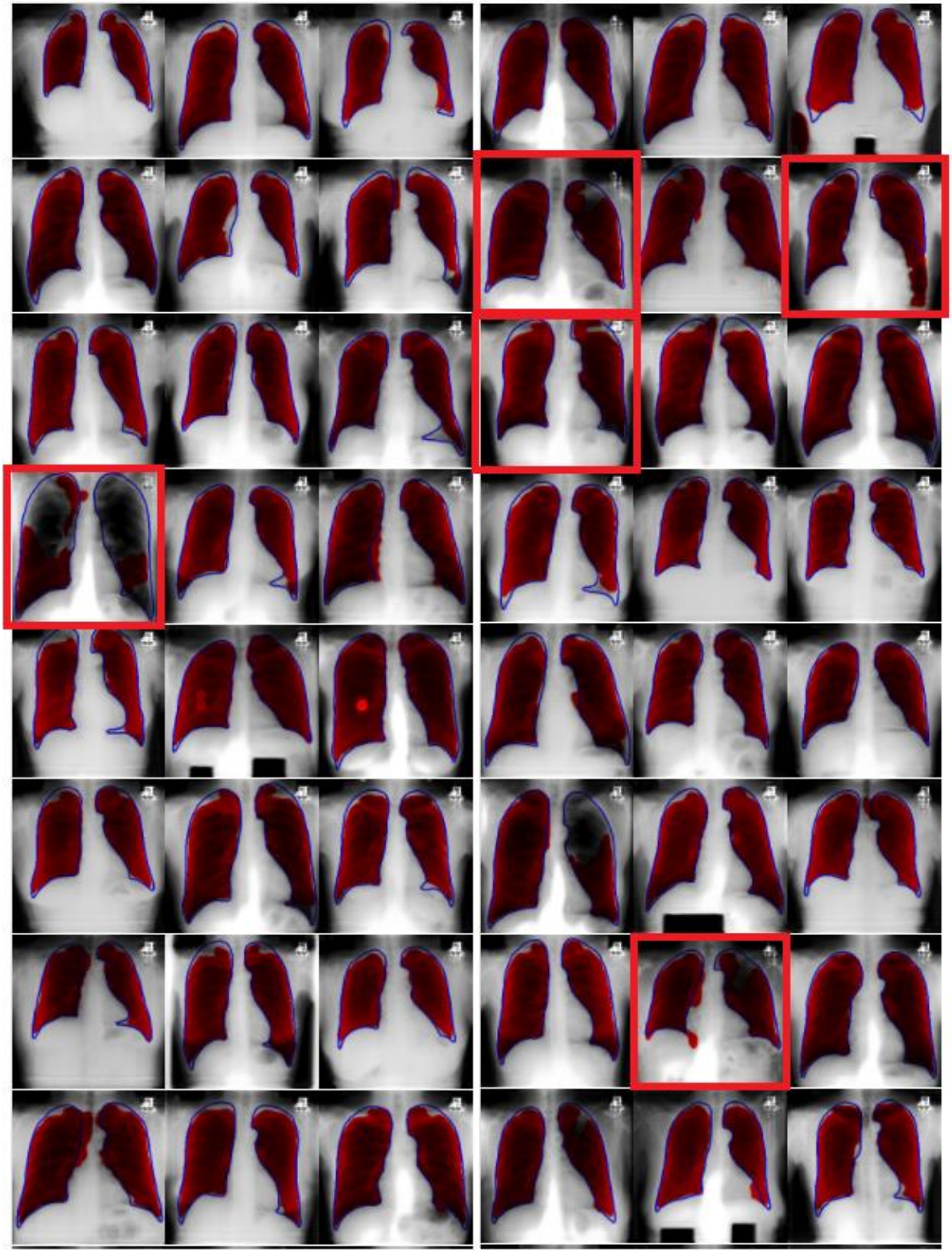
However, improved dice of 0.9384 obtained through CycleGAN transformed JSRT images is still a poor score. Base learner U-Net model must be tuned, or be enhanced using higher level pre-trained models like EfficientNet to obtain much robust and higher segmentation accuracy as explained in section 6.

5.4.1 Base learner U-Net tested on CycleGAN_UNet transformed JSRT images (dice=0.9360):



The CycleGAN_UNet transformed JSRT images results in better segmentation compared to results presented in section 4.4 for original JSRT images without using CycleGAN. However, many segmented images indicated in red boxes, can be seen with poor segmentation results.

5.4.2 Base learner U-Net tested on CycleGAN_ResNet transformed JSRT images (dice=0.9384):



The CycleGAN_ResNet transformed JSRT images results in highest accuracy for segmentation compared to other trials presented in section 5.4. Also, compared to CycleGAN_UNet results in section 5.4.1, the segmentation appears slightly better. However, many segmented images indicated in red boxes, can be seen with poor segmentation results.

For further improvement in results, base learner U-Net model needs to be enhanced as discussed in next section.

6. Base learner enhancements to improve segmentation accuracy

In this section, base learner U-Net model is enhanced using some ideas drawn from U-Net++ architecture and also using popular deep learning network EfficientNet as encoder to replace down-sampling layer of U-Net. Details of U-Net++ and EfficientNet architectures, and how these networks yield superior segmentation accuracy when combined is discussed in following sections.

6.1 U-Net++ Architecture:

U-Net++ is a deeply supervised encoder-decoder network where the encoder and decoder sub-networks are connected through a series of nested, dense skip pathways. The re-designed skip pathways aim at reducing the semantic gap between the feature maps of the encoder and decoder sub-networks, yielding significant performance gain over simple U-Net model.

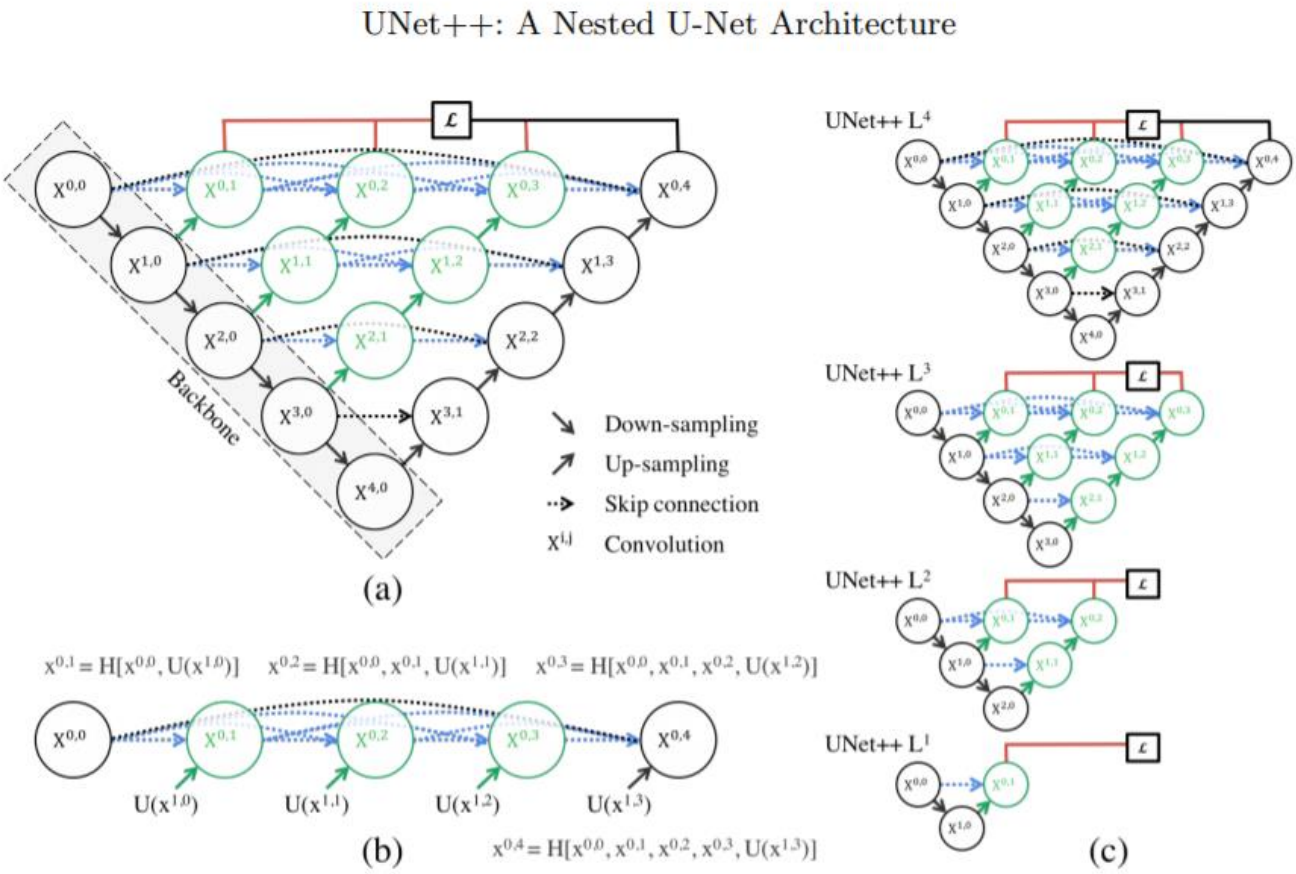


Figure 8: U-Net++ Architecture Explained

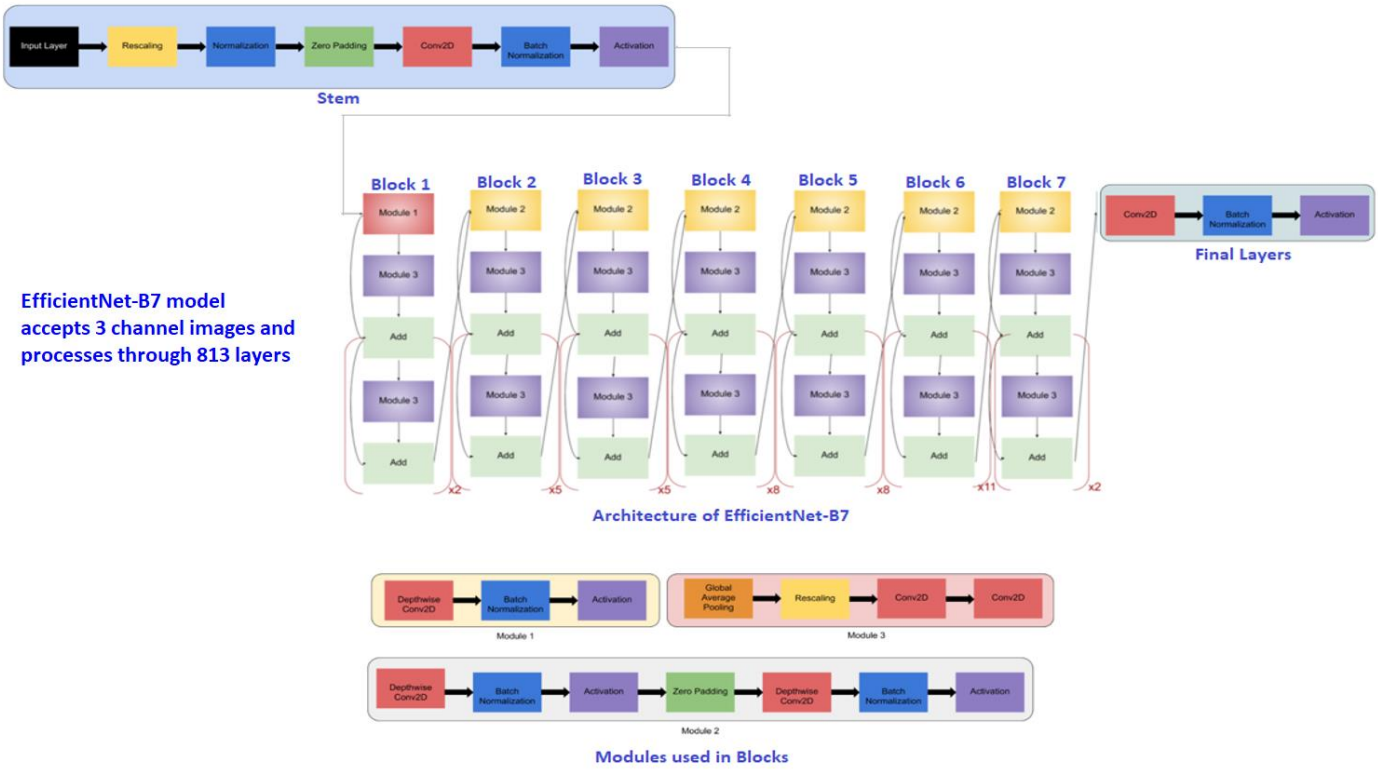
As shown in fig. (a), UNet++ consists of an encoder and decoder that are connected through a series of nested dense convolutional blocks. The main idea behind UNet++ is to bridge the semantic gap between the feature maps of the encoder and decoder prior to fusion. For example, the semantic gap between ($X^{0,0}$,$X^{1,3}$) is bridged using a dense convolution block with three convolution layers. In the graphical abstract, black indicates the original U-Net, green and blue show dense convolution blocks on the skip pathways, and red indicates deep supervision. Red, green, and blue components distinguish UNet++ from U-Net. Detailed analysis of the first skip pathway of U-Net++ is shown as in fig.(b and c) to explain how nesting works with dense connections. Fig.(c) shows loss function calculated and added at successive stages of UNet for deep supervision.

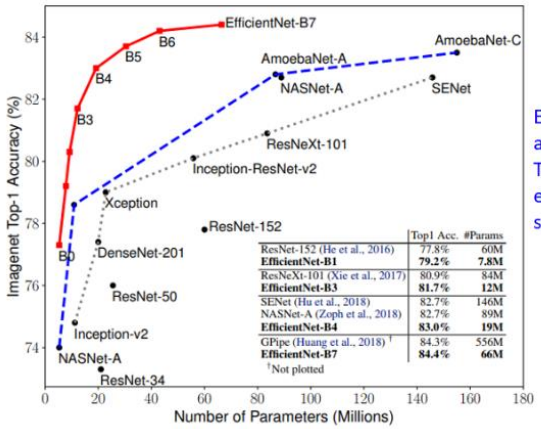
The 'back-bone' layer for this U-Net++ architecture can be normal down-sampling convolution layers as applied in simple U-Net, or can be replaced by feature extractions obtained through other pre-trained deep learning models like ResNet, EfficientNet etc.

For current lung segmentation task performed, U-Net++ architecture is not directly used (too many parameters to compute and needs high-end gpu). Instead, the simple baseline U-Net is used as-is and several up-sampling layers are being added with skip connections instead of nesting the layers. Also, one of the latest deep learning models called EfficientNet (pre-trained on imagenet) will be used as encoder as explained in section 6.3.

6.2 EfficientNet-B7 Architecture:

EfficientNet is a convolutional neural network architecture with compound scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient. Unlike conventional practice that arbitrary scales these factors, the EfficientNet scaling method uniformly scales network width, depth, and resolution with a set of fixed scaling coefficients in a principled way. The compound scaling method is justified by the intuition that if the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image. All EfficientNet networks are based on the inverted bottleneck residual blocks, in addition to squeeze-and-excitation blocks. Below figure shows layout of EfficientNet-B7 architecture, and a chart shows its class leading performance on imagenet database.





EfficientNet -B7 model shows class-leading accuracy on Imagenet database.
This model can be used as Encoder (feature extractor for X-Ray images) for the segmentation model developed.

Figure 9: Simple layout of EfficientNet-B7 Deep learning model and Class leading accuracy on Imagenet

6.3 Enhanced U-Net with EfficientNet B7 Encoder Backbone:

The lung segmentation model trained on Montgomery data did not perform well on JSRT data with simple U-Net and even after applying CycleGAN for domain transformation, the segmentation results don't look satisfactory. The better approach to solve this problem is to enhance the base learner U-Net model which is able to learn and extract better features from images irrespective of domain variation on test data.

An important update for the model is to use imagenet pre-trained EfficientNet-B7 (with 813 layers) encoded (down-sampled) features to replace the down-sampling layer which acts as the backbone for U-Net architecture. The imagenet pre-trained layers extract preliminary information from X-Ray images and provides as input to simple U-Net backbone (conv1, conv2, conv3 and conv4 in figure below) and additionally, up-sampling layers and skip connections added to base U-Net model helps to extract much more semantic content at deeper layers for better segmentation.

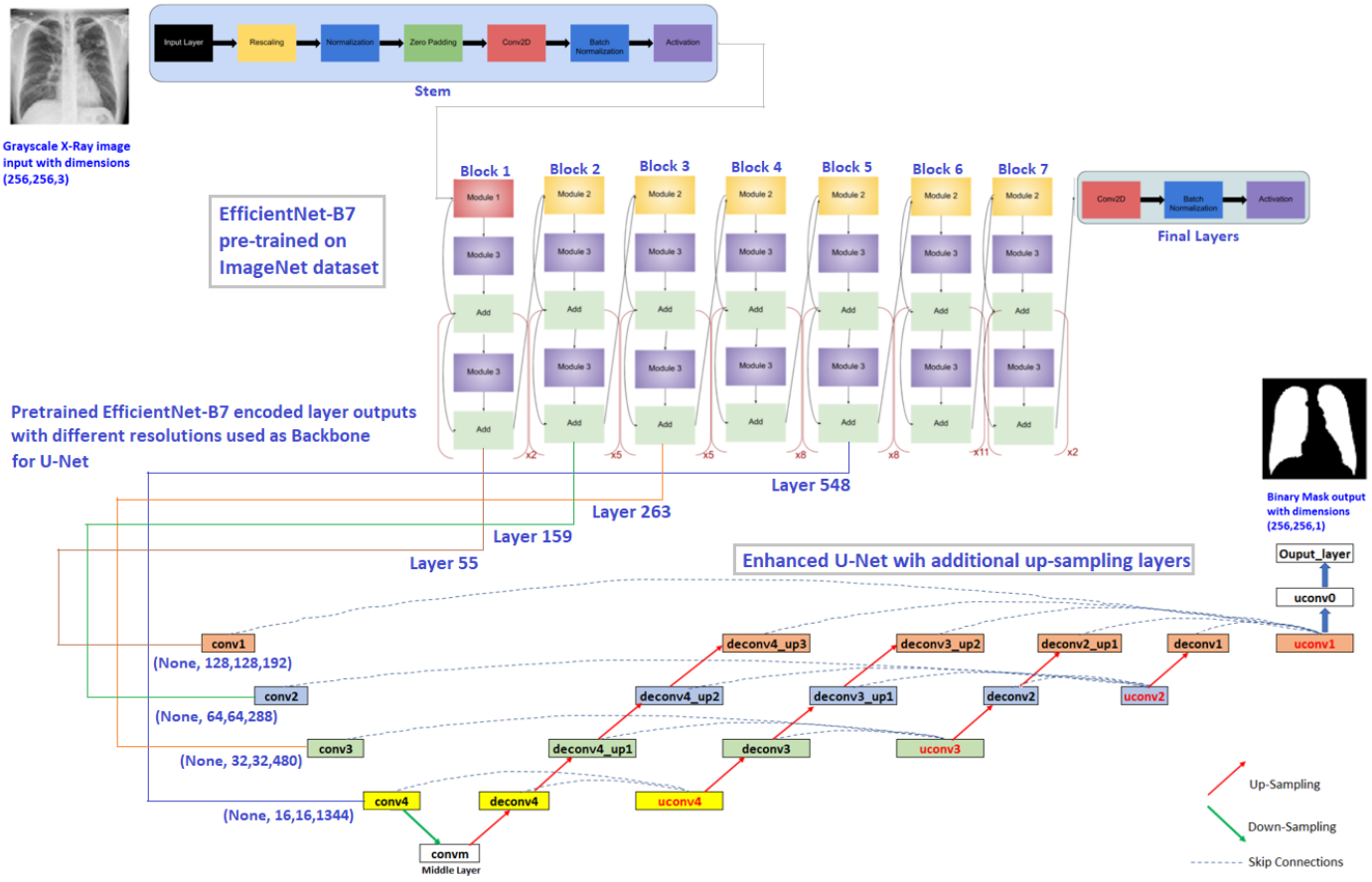


Figure 10: Enhanced U-Net with EfficientNet-B7 Encoder and addition of up-sampling layers

6.4 Training details, Test Results & Discussions

The complete network is setup as described in section 6.3. EfficientNet-B7 needs 3 channel images as input and ImageNet pretrained weights will be used for encoding features. The grayscale -Rays are converted to 3 channel image and encoded features are extracted at different layers of EfficientNet-B7. The extracted feature sets with different dimensions will be supplied as inputs to enhanced U-Net model which replaces all down-sampling layers.

The U-Net region of the model will be trained over 250 epochs for every fold. The "Dice loss" and "dice coefficient" functions will be used as custom functions optimized to produce segmented masks as outputs. The custom functions mentioned above show much better correlation for the metric used for the task in hand, yielding better performance compared to using binary cross entropy and accuracy in previous base learner U-Net model in section 4.

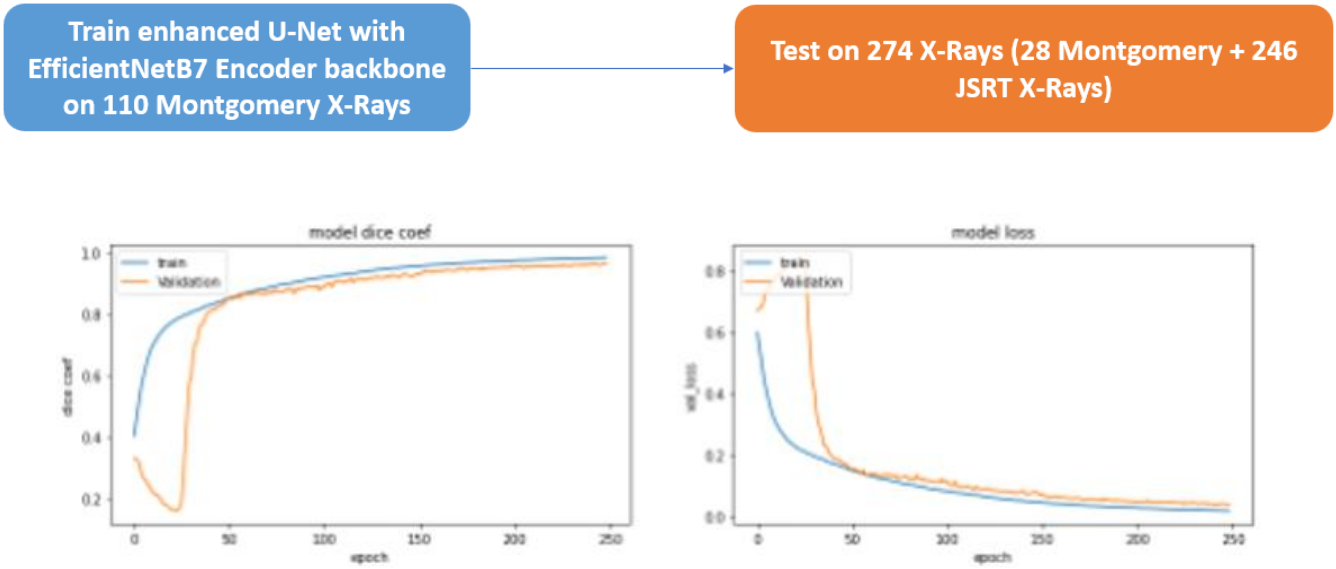


Figure 11: Enhanced U-Net training performance over epochs for one of the fold

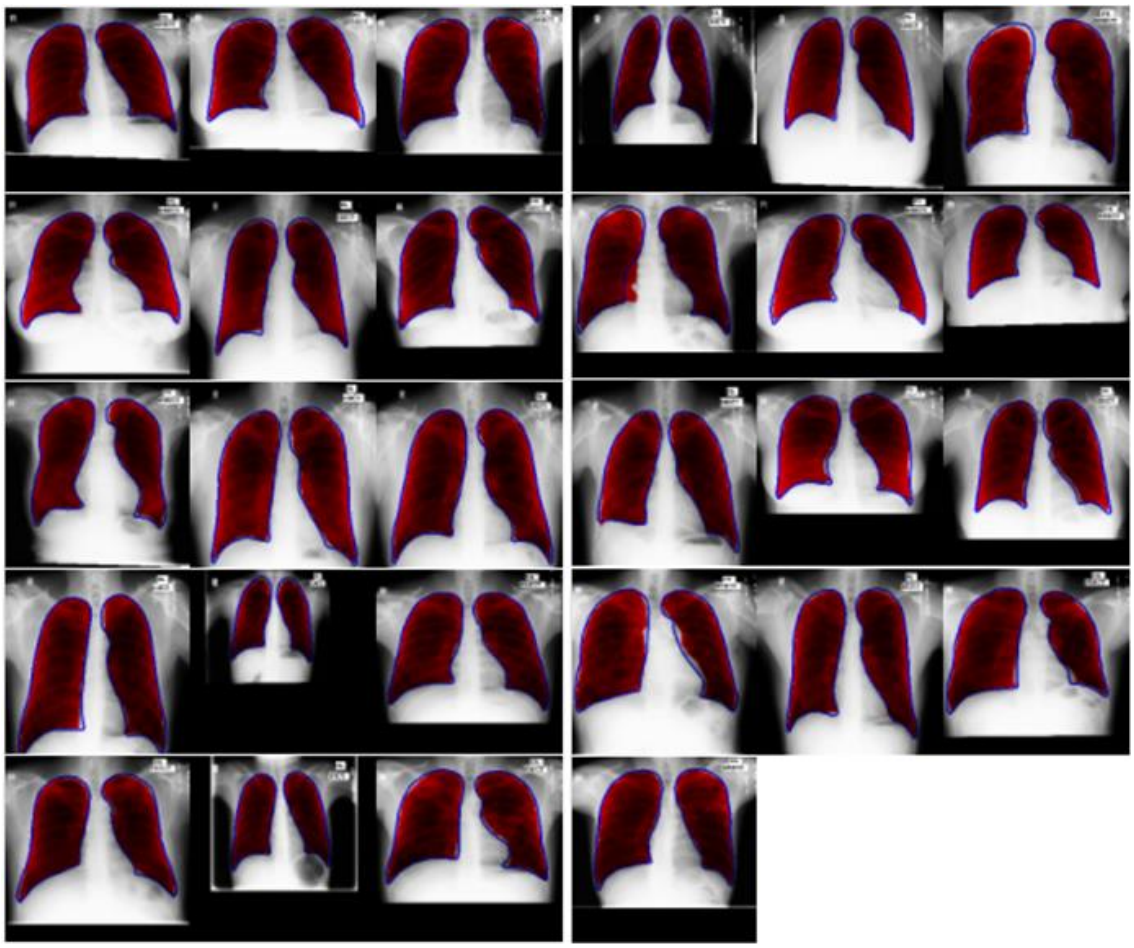
	Deep Learning Models for Lung Segmentation Task	DICE Metric on test set		
Trials	Description	Montgomery	JSRT	
0	Baseline - SupportNET Architecture (GRC) Estimations	0.956	0.964	Benchmark
1	U-Net with 256X256 single channel image. WITH 5-FOLD CV.	0.97427 +/-0.009	0.8879 +/-0.047	
2	CycleGAN_UNET transformed JSRT images tested using U-Net model in Trial 1	Not applicable	0.93599 +/-0.107	
3	CycleGAN_RESNET transformed JSRT images tested using U-Net model in Trial 1		0.93839 +/-0.081	
4	Enhanced U-Net with EfficientNet-B7 Encoder. WITH 5-FOLD CV.	0.98161 +/-0.007	0.9665 +/-0.013	Best Model developed
5	CycleGAN_UNET transformed JSRT images tested using Enhanced U-Net model in Trial 4	Not applicable	0.9634 +/-0.012	
6	CycleGAN_RESNET transformed JSRT images tested using Enhanced U-Net model in Trial 4		0.9626 +/-0.011	

Table 3: Enhanced U-Net model results compared against SupportNet

Trial 4, 5 and 6 show results calculated using enhanced U-Net and compared with base learner model results. Dice coefficient calculated using the enhanced U-Net model (Montgomery=0.9816 and JSRT=0.9665) shows superior performance compared to all other trials performed. **The performance of enhanced U-Net with EfficientNet-B7 encoder model appears as good or slightly better performance than SupportNet** with much simpler workflow (this does not need support of texture and shape features, and no memory augmentation needed).

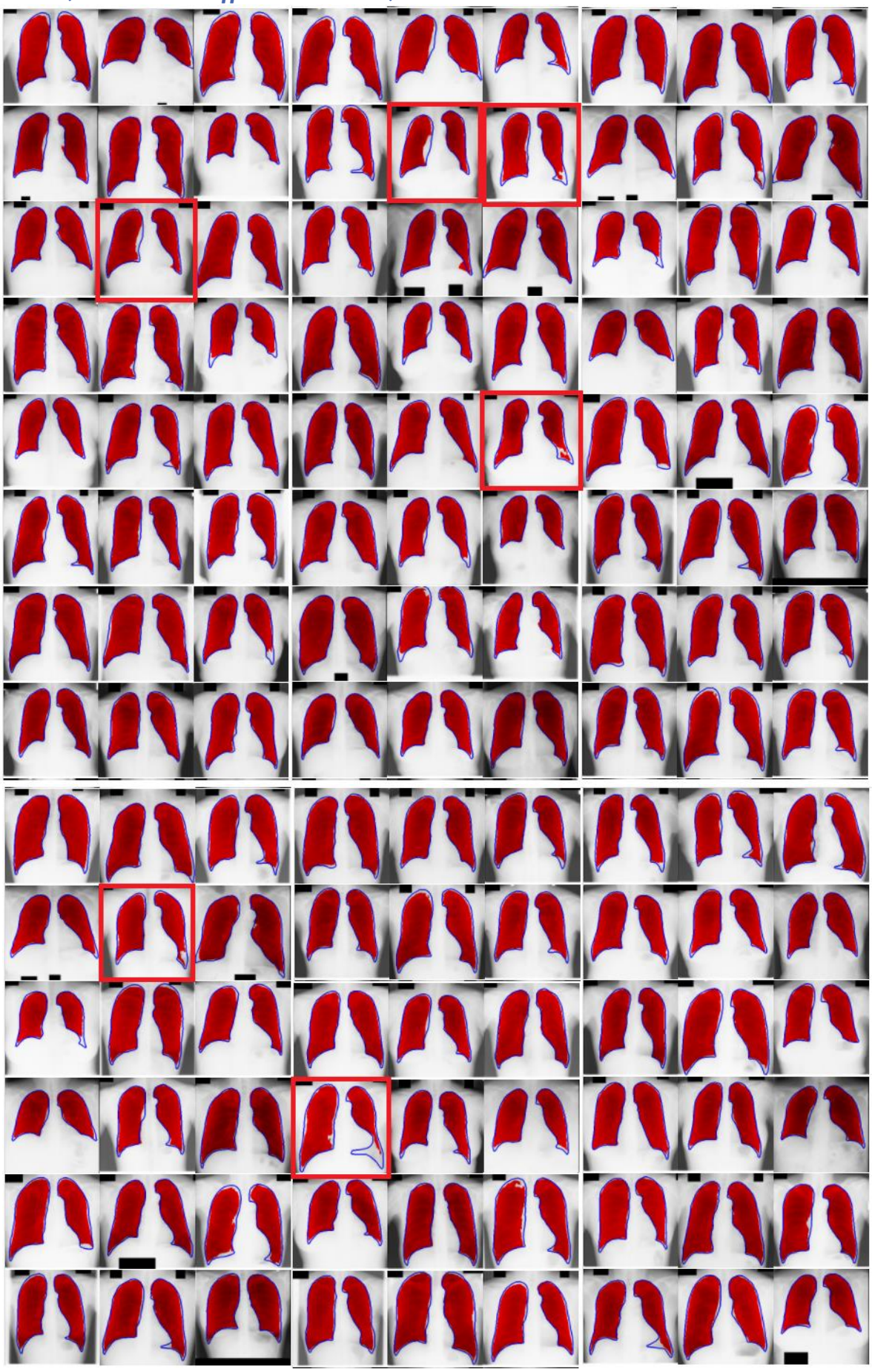
Further applying CycleGAN on JSRT images, did not show any difference to original JSRT dice results. **The enhanced U-Net with EfficientNet-B7 encoder appears very robust in segmentation task by strong feature extractions in the early layers and much better semantic content at deeper layers**. Hence this model is declared as best model developed for lung segmentation task in current report. Few examples of segmented lung masks are shown in section 6.4.1 through 6.4.3.

6.4.1 Enhanced U-Net with EffNet-B7 Encoder backbone trained and tested on Montgomery data (Mean Dice coefficient = 0.9816):



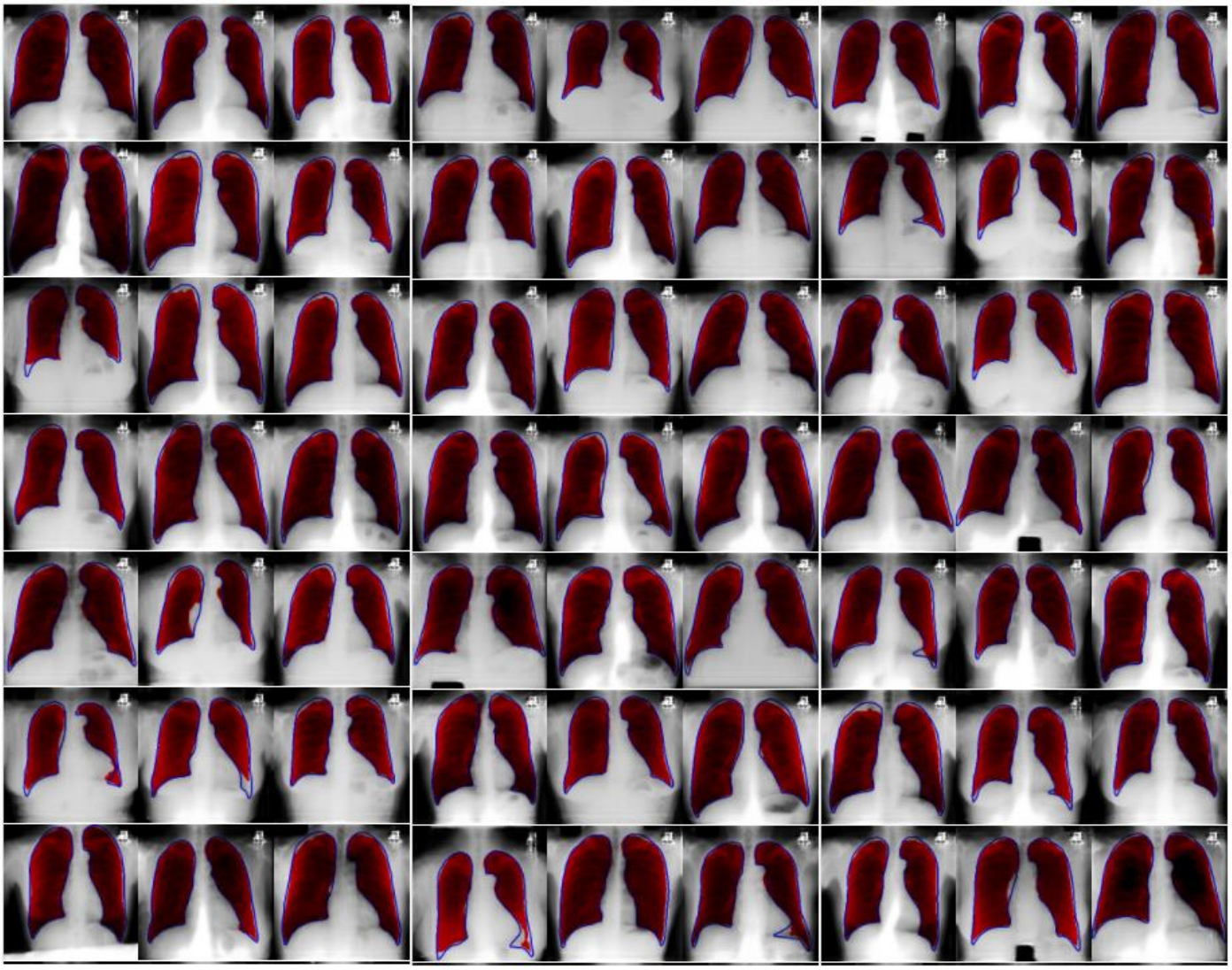
Montgomery test images show very accurately segmented lung masks .

6.4.2 Enhanced U-Net with EffNet-B7 Encoder backbone trained on Montgomery images and tested on JSRT data (Mean dice coefficient = 0.9665):



Enhanced U-Net with EfficientNet-B7 encoder based JSRT lung segmentations show very large improvement in segmentation accuracy compared to base learner U-Net with CycleGAN_ResNet in section 5.4.2.

6.4.3 Enhanced U-Net with EffNet-B7 Encoder backbone trained on Montgomery images and tested on CycleGAN_ResNet transformed JSRT data (Mean dice coefficient = 0.9626):



There was no effect of CycleGAN transformation on JSRT to Montgomery images on segmentation accuracy.

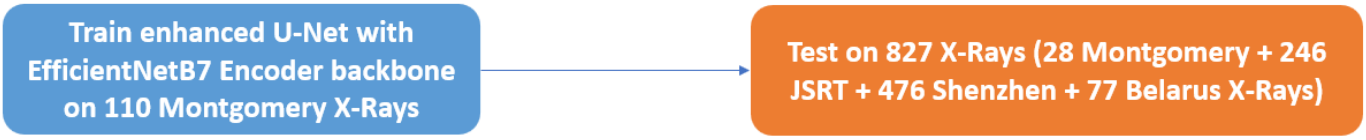
The enhanced U-Net+EffNet-B7 model proves that the model generalizes and adapts very well between different domain datasets. In next section, we will add two more X-Ray datasets (Shenzhen and Belarus X-Rays) to check domain adaptation limits of this model.

7. Domain adaptation capability of Enhanced U-Net model

The enhanced U-Net+EffNet-B7 model emerged as best developed model in current study and worked well on JSRT domain images. To check the adaptation limits of this model, two additional datasets from different domains will be used – Shenzhen and Belarus TB X-Ray images.

Shenzhen TB dataset has total 662 X-Rays and lung masks are available only for ~560 images. Not all available lung masks are accurately segmented for lung regions. Hence, at-least 80 X-Rays with inaccurate lung masks from the dataset are rejected through visual inspection and only 476 X-Rays are used for testing. For Belarus TB dataset, only 77 X-Ray images are available with lung masks.

The Enhanced U-Net model explained in previous section 6, will be directly for testing. Hence the model trained purely on 110 Montgomery X-Rays will be tested on total 827 images belonging to 4 different domains.



7.1 Results and discussions

	Enhanced U-Net with EfficientNet-B7 Encoder	DICE Metric on test set			
Trials	Description	Montgomery	JSRT	Shenzhen	Belarus
7	Enhanced U-Net + EffNet-B7 (Trial 4 extended to include Shenzhen and Belarus datasets)	0.98161 +/-0.007	0.9665 +/-0.013	0.9610 +/-0.018	0.9505 +/-0.021
8	Trial 7 With Clahe applied on input images (clip limit = 3, Tile size = 8,8)	0.9797 +/-0.007	0.9688 +/-0.009	0.9596 +/-0.019	0.9506 +/-0.018

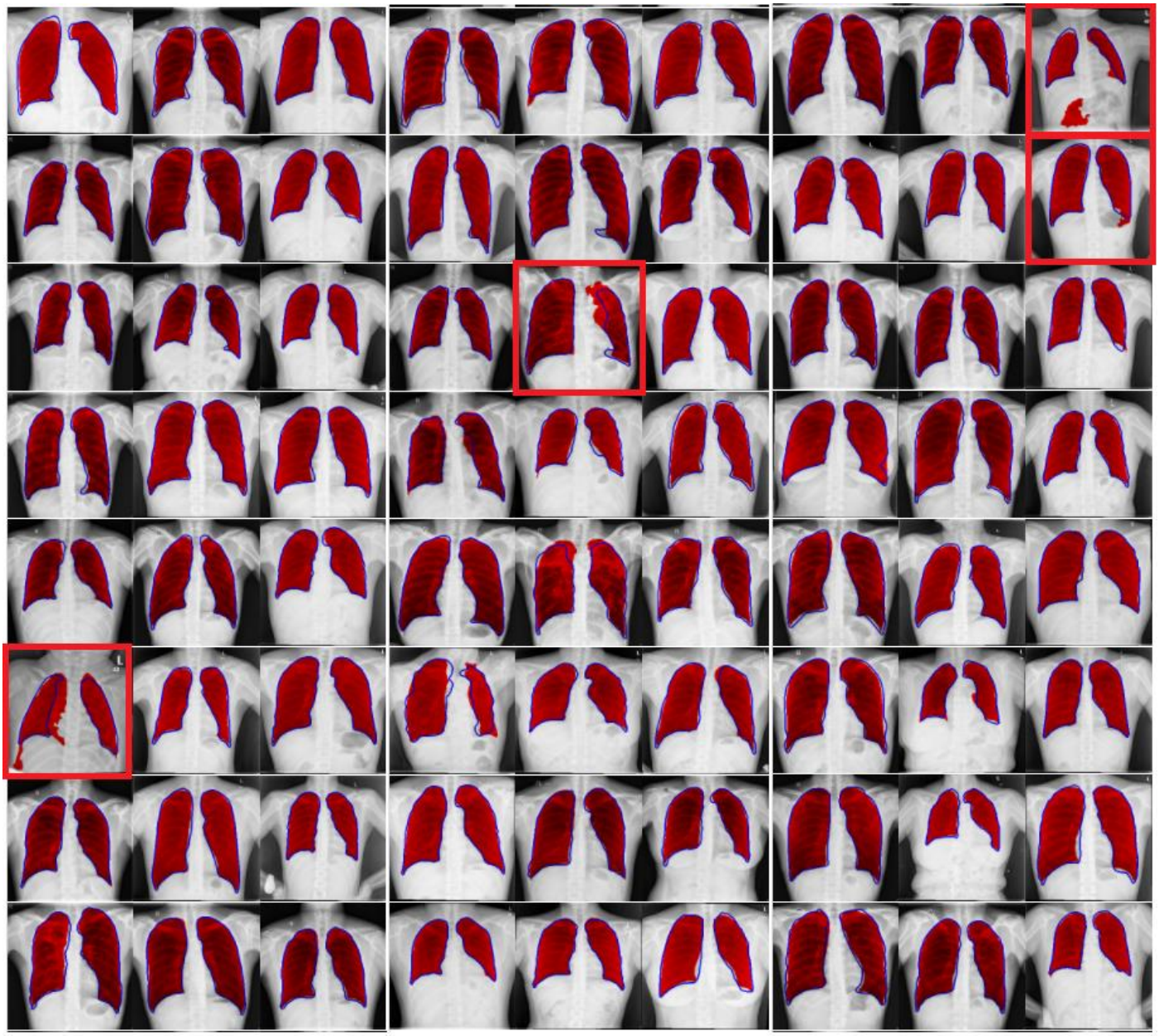
Table 4: Enhanced U-Net model domain adaption ability test and effect of clahe on results

Trial 7 in above table is essentially same model as trial 4 explained in previous section and only addition is extending testing on two extra datasets – Shenzhen and Belarus.

The enhanced U-Net model discussed in section 6, was additionally tested on Shenzhen and Belarus datasets in trial 7 and it resulted in good dice scores. The segmentation results looked acceptable for Shenzhen dataset as presented in section 7.2. For Belarus set, segmentation results are show in section 7.3, appeared acceptable except for few images. Belarus set X-Rays are much different (darker with less contrast) in appearance compared to other three datasets and it could be the reason for a slightly degraded performance.

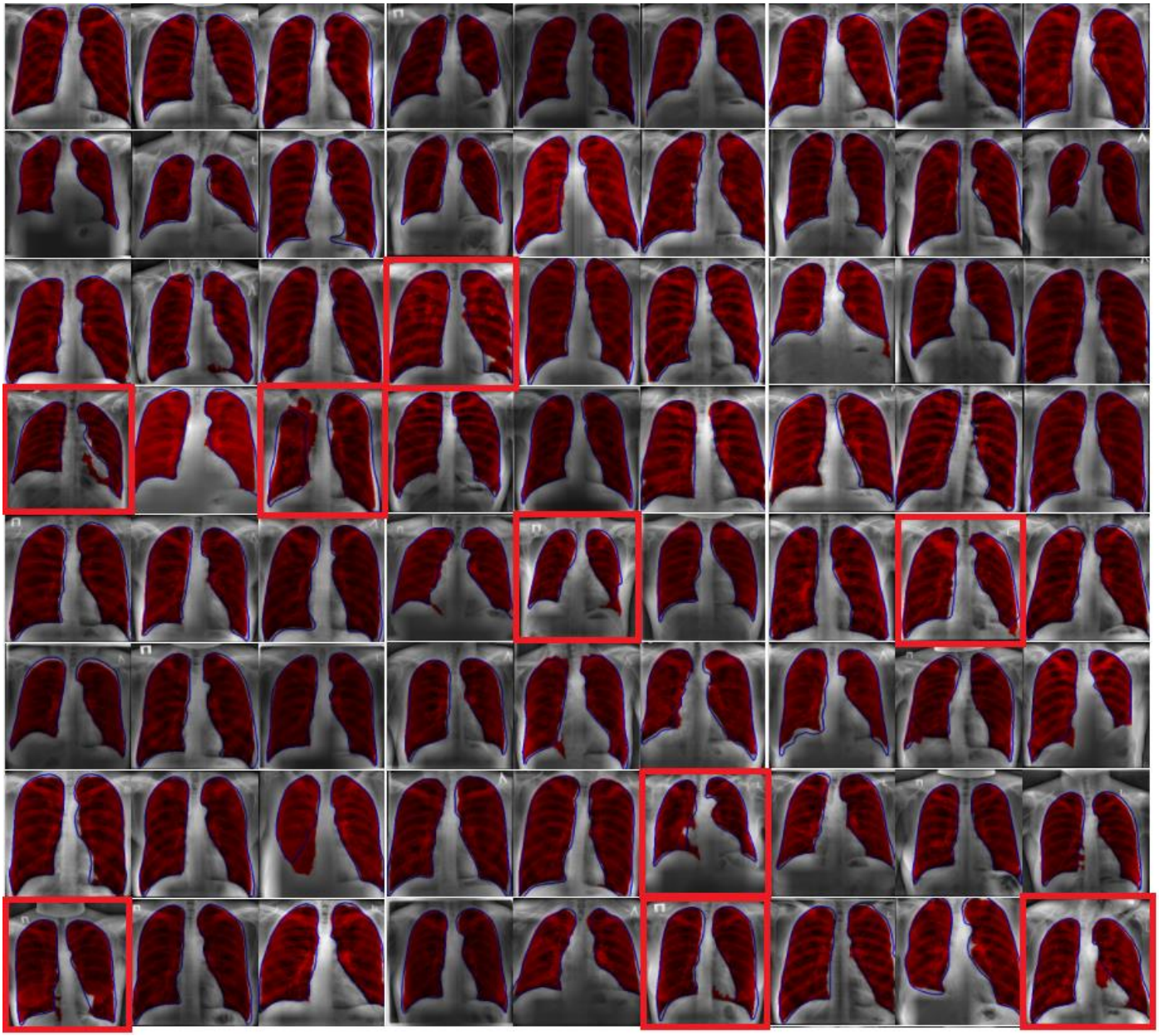
Additionally, in trial 8, image enhancement effect like clahe (Contrast Limited Adaptive Histogram Equalization) is applied and model is re-trained. However, this had no effect or slightly negative effect on performance of model. **Hence, trial 7 shows that image pre-processing or enhancement is not necessary for the model to achieve best segmentation accuracy and can be considered as best final model results from current study.**

7.2 Enhanced U-Net with EffNet-B7 Encoder backbone trained on Montgomery images and tested on Shenzhen data (Mean dice coefficient = 0.9610): No Clahe applied for training and testing



Shenzhen X-Ray masks appear well segmented except for few cases. Segmentation performance is almost same as JSRT set.

7.3 Enhanced U-Net with EffNet-B7 Encoder backbone trained on Montgomery images and tested on Belarus data (Mean dice coefficient = 0.9505): No Clahe applied for training and testing.



Belarus X-Rays are well segmented for majority of images. Yet, few images show segmented X-Ray lung region spreading out of the actual lung regions. It is yet to be seen if CycleGAN domain transformation from Belarus to Montgomery domain helps to improve segmentation accuracy for this dataset, however, it will not be done in current study and recommended as future scope for further improvement.

8. Conclusions and Deliverables

8.1 Conclusions

Compiling all results together, table below explains the sequence of trials performed and corresponding outputs to compare. **The enhanced U-Net with EfficientNet-B7 encoder explained in trial 7 (same as trial 4) is the best model developed in this study.**

	Deep Learning Models for Lung Segmentation Task	DICE Metric on test set				
Trials	Description	Montgomery	JSRT	Shenzhen	Belarus	
0	Baseline - SupportNET Architecture (GRC) Estimations	0.956	0.964	Not Included		
1	U-Net with 256X256 single channel image. WITH 5-FOLD CV.	0.97427 +/-0.009	0.8879 +/-0.047			
2	CycleGAN_UNET transformed JSRT images tested using U-Net model in Trial 1	Not applicable	0.93599 +/-0.107			
3	CycleGAN_RESNET transformed JSRT images tested using U-Net model in Trial 1		0.93839 +/-0.081			
4	Enhanced U-Net with EfficientNet-B7 Encoder. WITH 5-FOLD CV.	0.98161 +/-0.007	0.9665 +/-0.013			
5	CycleGAN_UNET transformed JSRT images tested using Enhanced U-Net model in Trial 4	Not applicable	0.9634 +/-0.012			
6	CycleGAN_RESNET transformed JSRT images tested using Enhanced U-Net model in Trial 4		0.9626 +/-0.011			
7	Enhanced U-Net + EffNet-B7 (Trial 4 extended to include Shenzhen and Belarus datasets)	0.98161 +/-0.007	0.9665 +/-0.013	0.9610 +/-0.018	0.9505 +/-0.021	Best Model
8	Trial 7 With Clahe applied on input images (clip limit = 3, Tile size = 8,8)	0.9797 +/-0.007	0.9688 +/-0.009	0.9596 +/-0.019	0.9506 +/-0.018	

Table 5: Comparison of results for all experiments performed in this study

- Base learner U-Net model trained from scratch on Montgomery domain images worked well only for Montgomery domain and did not generalize on images of another domain like JSRT.

- CycleGAN domain transformation tested on JSRT to Montgomery domain, effectively results in higher segmentation accuracy by increasing the contrast between lungs and non-lung regions. For base learner U-Net model CycleGAN resulted in significant improvement accuracy.

- CycleGAN tested with two generator networks – ResNet and U-Net. Both models show similar performance and accuracy for domain transformation.

- **Enhanced U-Net model with EfficientNet-B7 Encoder and several up-sampled layers, results in best accuracy and adaptation across different domains. This works well even with no image enhancements like clahe etc., applied as pre-processing step**. Also, it was observed that experiment of applying CycleGAN and clahe image pre-processing made no significant difference to the results.

- **Compared to benchmark GRC SupportNet architecture model, the Enhanced U-Net + EfficientNet-B7 encoder showed similar or slightly better results with simpler workflow** (with no need of support texture and shape features, and no continuous memory augmentation needed). However, this comparison is limited to only two test datasets - Montgomery and JSRT. It is expected that, ensembled techniques between SupportNet and Enhanced U-Net+EfficientNet-B7 models would further improve segmentation accuracy.

8.2 Deliverables

1. **What is the best model you got? What is the best layers and parameter setting that gives best results?**

- As discussed in section 6.3, Enhanced U-Net with EfficientNet-B7 encoder is the best model. The X-Ray features extracted from EfficientNet-B7 down-sampling layer numbers 55, 159, 263, and 548 replace the down-sampling layers of U-Net model. Also, the up-sampling layers are added to U-Net model which extract better semantic details from image. For training, 'Adam' optimizer is used with learning rate of 0.0001 and keras callback 'ReduceLROnPlat' with minimum learning rate variation of 1e-6 is used to adjust the learning rate and trained over 250 epochs.

2. **What is the best loss functions (entropy, MSE, DICE) to get best results?**

- Keras in-built loss function - Binary cross Entropy was used in base learner U-Net model which produces average segmentation accuracy. In Enhanced U-Net model, a custom function is defined called Dice loss, which specifically calculates better gradients for backpropagation process for the segmentation task in hand, and simultaneously helps to improve the accuracy metric Dice coefficient resulting in better segmentation.

3. **What are the effects of data augmentation on results?**

- Data augmentation introduces slightly modified versions of existing training images by performing techniques like cropping, padding and flipping etc. This enables significant increase in diversity of data and eliminates the need of additional data. In current study, training dataset is very small consisting of just 110 images and data augmentation like rotation, shift, shear and zoom are used. During training process over different epochs, data augmentation supplies the diversified images and helps to eliminate the model overfitting on training images. In this report, all experiments are performed using data augmentation (comparison results for with and without data augmentation not available).

4. **How did you validate that your model was good and reliable? What is the error distribution across sets? How do you test overfitting? Is your model stable in its results?**

- K-fold cross validation strategy is used during training by splitting the data into 5 random subsets. This helps preventing model overfitting on training images and poorly performing on unseen test or validation images. The training and validation losses are plotted and compared after each fold is trained and the results slightly vary depending on easy or difficult to segment images present in corresponding subset. Both training and validation loss and accuracy improve together as explained in figure 4 and 11 in section 4.2 and 6.4. Finally, averaged performance all folds is used as output prediction and reported as dice coefficient. The final results presented in trial 7 indicate the model generalizes well and performs consistently between different domains. Hence using K-fold CV along with data augmentation makes sure the model developed is robust, stable and reliable.

5. **A portion of the randomly selected dataset (20%) should be used as test dataset and evaluated through the use of Dice Metric2. You should get a dice value of greater than 0.8 on the test set to pass this test problem.**

- The test data used in this study is very large compared to training data explained in section 7.1. The training data used is just 110 images from Montgomery set, and for testing, massive set of 827 images from 4 different domains are used. Trial 7 in section 8.1, shows dice results for different datasets. Enhanced U-Net model is the best model developed and resulting in dice coefficients as high as 0.9816, 0.9665, 0.9610 and 0.9505 on Montgomery, JSRT, Shenzhen and Belarus datasets respectively.

9. References

[1]. O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation, in International Conference on Medical image computing and computer-assisted intervention, pp. 234{241, Springer, 2015.

[2]. “Tuberculosis chest x-ray image data sets." https://ceb.nlm.nih.gov/repositories/tuberculosis-chest-x-ray-image-data-sets/. Accessed: 04-06-2018.

[3]. Xin Yi, Ekta Walia, Paul Babyn, “Generative adversarial network in medical imaging: A review”, ELSEVIER Medical Image Analysis 58 (2019) 101552.

[4]. Vardhan Agrawal, “Complete Architectural Details of all EfficientNet Models”, Towards Data Science.

[5]. Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang, “UNet++: A Nested U-Net Architecture for Medical Image Segmentation”.

[6]. Tawsifur Rahman, Amith Khandakar, Yazan Qiblawey, Anas Tahir , “Exploring the Effect of Image Enhancement Techniques on COVID-19 Detection using Chest X-rays Images”. Department of Biomedical Physics & Technology, University of Dhaka Dhaka-1000, Bangladesh

[7]. GRC Bangalore, “Memory Augmented Continuous Learning for Segmentation”.

[8]. Rahul Venkataramani, Hariharan Ravishankar, Saihareesh Anamandra, “Towards Continuous Domain adaptation for Healthcare”

[9]. Siddartha, Kaggle Master, “https://www.kaggle.com/meaninglesslives/nested-unet-with-efficientnet-encoder”.

[10]. Bhakti Baheti, Shubham Innani, Suhas Gajre, Sanjay Talbar , “Eff-UNet: A Novel Architecture for Semantic Segmentation in Unstructured Environment”

[11]. Dong-Ho Lee, Yan Li and Byeong-Seok Shin , “Generalization of intensity distribution of medical images using GANs”

[12]. “Machine Learning Mastery”, All books by Jason Brownlee.

[13]. Shenzhen Dataset – “https://www.kaggle.com/raddar/tuberculosis-chest-xrays-shenzhen”

[14]. Faizan Munawar, Shoaib Azmat, Talha Iqbal, Christer Grönlund, And Hazrat Ali, “Segmentation of Lungs in Chest X-Ray Image Using Generative Adversarial Networks”.

10. APPENDIX:

List of Python Codes attached:

1. Base learner U-Net trained from scratch on Montgomery images using K-fold CV
2. Base learner U-Net inference on Montgomery and JSRT images
3. Read and Convert JSRT dataset to NPY format
4. CycleGAN domain transformation (JSRT to Montgomery) using ResNet as Encoder
5. CycleGAN domain transformation (JSRT to Montgomery) using U-Net as Encoder
6. CycleGAN inference code (Convert JSRT to Montgomery domain)
7. Enhanced U-Net with EfficientNet-B7 backbone encoder

1. U-Net trained from scratch on Montgomery images using K-fold CV:

```
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold

import os
import numpy as np
from skimage import io, exposure
from skimage.filters import laplace
from skimage.transform import resize
import cv2

#create a new output folder
#os.mkdir('/working/CXR_Mask/')

# define location of dataset for Main lung CXR images
folder = '../input/montgomery-data/Main Problem/MainLungImages/'
CXR, CXR_Mask = list(), list()
im_shape = (256, 256)

def make_masks():
    path = '../input/montgomery-data/Main Problem/MainLungImages/'
    for i, filename in enumerate(os.listdir(path)):
        # load CXR image
        photo = cv2.imread(folder + filename, 0)  # 0 means grayscale
        photo = photo.astype('float32')/255.0
        #photo = laplace(photo)
        photo = cv2.resize(photo, dsize=im_shape, interpolation=cv2.INTER_LINEAR)
        CXR.append(photo)

        left = cv2.imread('../input/montgomery-data/Main Problem/leftMask/' + filename[:-4] + '
.png', 0)
        right = cv2.imread('../input/montgomery-data/Main Problem/rightMask/' + filename[:-4] +
'.png', 0)
        fullmaskarray = np.clip(left + right, 0, 255).astype('float32')/255.0
        fullmaskarray = cv2.resize(fullmaskarray, dsize=im_shape, interpolation=cv2.INTER_LINEA
R)
        CXR_Mask.append(fullmaskarray)
        #print ('CXR_Mask', i, filename)

make_masks()
# convert to numpy arrays
CXR = asarray(CXR)
#CXR -= CXR.mean()
#CXR /= CXR.std()
CXR_Mask = asarray(CXR_Mask)
```

```
CXR = CXR.reshape(CXR.shape[0], 256, 256, 1)
CXR_Mask = CXR_Mask.reshape(CXR_Mask.shape[0], 256, 256, 1)

# Split into train and test data(20%)
X=np.arange(0,138)
X_train, X_test = train_test_split(X, test_size=0.2, random_state=999)

CXR_train=[]
CXR_Mask_train=[]
for i in X_train:
    CXR_train.append(CXR[i,:,:,:])
    CXR_Mask_train.append(CXR_Mask[i,:,:,:])
CXR_train=np.array(CXR_train)
CXR_Mask_train=np.array(CXR_Mask_train)

print(CXR.shape, CXR_Mask.shape, CXR_train.shape, CXR.min(), CXR.max(), CXR_Mask.min(), CXR_Mas
k.max())
#save the reshaped CXR_Mask
#save('CXR.npy', CXR)
#save('CXR_Mask.npy', CXR_Mask)

#=======================================BUILD UNET MODEL=================================
=======
from keras.models import Model
from keras.layers.merge import concatenate
from keras.layers import Input, Convolution2D, MaxPooling2D, UpSampling2D


def UNET(inp_shape, k_size=3):
    merge_axis = -1 # Feature maps are concatenated along last axis (for tf backend)
    data = Input(shape=inp_shape)
    conv1 = Convolution2D(filters=32, kernel_size=k_size, padding='same', activation='relu')(da
ta)
    conv1 = Convolution2D(filters=32, kernel_size=k_size, padding='same', activation='relu')(co
nv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu')(po
ol1)
    conv2 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu')(co
nv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu')(po
ol2)
    conv3 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu')(co
nv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='relu')(p
ool3)
    conv4 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='relu')(c
onv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

    conv5 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='relu')(p
ool4)

    up1 = UpSampling2D(size=(2, 2))(conv5)
    conv6 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='relu')(u
p1)
    conv6 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='relu')(c
onv6)
    merged1 = concatenate([conv4, conv6], axis=merge_axis)
```

```
    conv6 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='relu')(m
erged1)

    up2 = UpSampling2D(size=(2, 2))(conv6)
    conv7 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='relu')(u
p2)
    conv7 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='relu')(c
onv7)
    merged2 = concatenate([conv3, conv7], axis=merge_axis)
    conv7 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='relu')(m
erged2)

    up3 = UpSampling2D(size=(2, 2))(conv7)
    conv8 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='relu')(u
p3)
    conv8 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='relu')(c
onv8)
    merged3 = concatenate([conv2, conv8], axis=merge_axis)
    conv8 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='relu')(m
erged3)

    up4 = UpSampling2D(size=(2, 2))(conv8)
    conv9 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu')(up
4)
    conv9 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu')(co
nv9)
    merged4 = concatenate([conv1, conv9], axis=merge_axis)
    conv9 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu')(me
rged4)

    conv10 = Convolution2D(filters=1, kernel_size=k_size, padding='same', activation='sigmoid')
(conv9)

    output = conv10
    model = Model(data, output)
    return model
#=============================================================================================
=========

#================================Start kfold validations and model build+train+evaluations====
======
#Using 5 folds; train 4 batches 22*4=88 images and test 1 batch 22 images
num_train_samples = 88
train_batch_size = 16
train_steps = np.ceil(num_train_samples / train_batch_size)

num_val_samples = 22
val_batch_size = 4
val_steps = np.ceil(num_val_samples / val_batch_size)

#=============================================DATA AUGMENTATION=============================
==========
from keras.preprocessing.image import ImageDataGenerator
from IPython.display import clear_output
from keras.utils.vis_utils import plot_model
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, EarlyStopping, ReduceLROnPl
ateau

dg_args = dict(    rescale=1.,
                   rotation_range = 10,
                   width_shift_range = 0.1,
                   height_shift_range = 0.1,
                   shear_range = 0.01,
                   zoom_range = [0.8, 1.2],
                   fill_mode = 'nearest',
                   data_format = 'channels_last')
```

```
image_gen = ImageDataGenerator(**dg_args)

def gen_augmented_pairs(in_cxr, in_seg, batch_size):
    while True:
        seed = np.random.choice(range(9999))
        # keep the seeds syncronized otherwise the augmentation to the images is different from
the masks
        g_cxr = image_gen.flow(in_cxr, batch_size = batch_size, seed = seed)
        g_seg = image_gen.flow(in_seg, batch_size = batch_size, seed = seed)
        for i_cxr, i_seg in zip(g_cxr, g_seg):
            yield i_cxr, i_seg

def get_model_name(k):
    return 'UNET_'+str(k)+'.h5'

#VALIDATION_ACCURACY = []
#VALIDAITON_LOSS = []
inp_shape = CXR_train[0].shape

#save_dir = '/working/'
num_folds=5
kf = KFold(n_splits=num_folds, shuffle=True)
fold_var = 1
Y = np.arange(0,len(CXR_train))

for train_index, val_index in kf.split(Y):
        train_gen = gen_augmented_pairs(CXR_train[train_index], CXR_Mask_train[train_index], b
atch_size = train_batch_size)
        image_gen = ImageDataGenerator(rescale=1.)
        valid_gen = gen_augmented_pairs(CXR_train[val_index], CXR_Mask_train[val_index], batch
_size = val_batch_size)

        # Build model
        UNet = UNET(inp_shape)
        UNet.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

        # CREATE CALLBACKS
        checkpoint = ModelCheckpoint(get_model_name(fold_var), monitor='val_loss', verbose=1,
                                     save_best_only=True, mode='min', save_weights_only = Fals
e)

        reduceLROnPlat = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
                                           patience=15,
                                           verbose=1, mode='min', epsilon=0.0001, cooldown=2,
min_lr=1e-6)
        early = EarlyStopping(monitor="val_loss",
                              mode="min",
                              patience=15) # probably needs to be more patient, but kaggle tim
e is limited
        callbacks_list = [checkpoint, early, reduceLROnPlat]
        # FIT THE MODEL
        history = UNet.fit_generator(train_gen,
                          steps_per_epoch = train_steps,
                          epochs = 80, verbose=1,
                          validation_data = valid_gen,
                          validation_steps = val_steps,
                          callbacks = callbacks_list
                         )

        #PLOT HISTORY
        #                :
        #                :

        # LOAD BEST MODEL to evaluate the performance of the model
```

```
        #UNet.load_weights("/working/UNET_"+str(fold_var)+".h5")

        #results = UNet.evaluate(valid_gen)
        #results = dict(zip(UNet.metrics_names,results))

        #VALIDATION_ACCURACY.append(results['accuracy'])
        #VALIDATION_LOSS.append(results['loss'])

        #tf.keras.backend.clear_session()

        fold_var += 1
```

2. U-Net inference on Montgomery test images:

```
def Dice(y_true, y_pred):
    """Returns Dice Similarity Coefficient for ground truth and predicted masks."""
    assert y_true.dtype == bool and y_pred.dtype == bool
    y_true_f = y_true.flatten()
    y_pred_f = y_pred.flatten()
    intersection = np.logical_and(y_true_f, y_pred_f).sum()
    return (2. * intersection + 1.) / (y_true.sum() + y_pred.sum() + 1.)

def masked(img, gt, mask, alpha=1):
    """Returns image with GT lung field outlined with red, predicted lung field
    filled with blue."""
    rows, cols = img.shape
    color_mask = np.zeros((rows, cols, 3))
    #convert to numpy float and then subtract
    #boundary = morphology.dilation(gt, morphology.disk(3)) - gt
    a1 = morphology.dilation(gt, morphology.disk(3))
    a2 = gt
    a1 = a1.astype(np.float32)
    a2 = a2.astype(np.float32)
    boundary = a1 - a2

    color_mask[mask == 1] = [1, 0, 0]          #predicted mask - Red color area
    color_mask[boundary == 1] = [0, 0, 1]      # Actual mask - Ground truth - Blue curve
    img_color = np.dstack((img, img, img))

    img_hsv = color.rgb2hsv(img_color)
    color_mask_hsv = color.rgb2hsv(color_mask)

    img_hsv[..., 0] = color_mask_hsv[..., 0]
    img_hsv[..., 1] = color_mask_hsv[..., 1] * alpha

    img_masked = color.hsv2rgb(img_hsv)
    return img_masked

def remove_small_regions(img, size):
    """Morphologically removes small (less than size) connected regions of 0s or 1s."""
    img = morphology.remove_small_objects(img, size)
    img = morphology.remove_small_holes(img, size)
    return img


# For inference standard keras ImageGenerator is used.
test_gen = ImageDataGenerator(rescale=1.)

n_test = CXR_test.shape[0]
dices = np.zeros(n_test)
im_shape = (256, 256)

i = 0
count=0
for xx, yy in test_gen.flow(CXR_test, CXR_Mask_test, batch_size=1, seed=999):
```

```
    count=count+1
    img = np.squeeze(xx)
    UNet.load_weights('../input/kfold-unet-ver0/UNET_1.h5')
    pred1 = UNet.predict(xx)[..., 0].reshape(inp_shape[:2])
    UNet.load_weights('../input/kfold-unet-ver0/UNET_2.h5')
    pred2 = UNet.predict(xx)[..., 0].reshape(inp_shape[:2])
    UNet.load_weights('../input/kfold-unet-ver0/UNET_3.h5')
    pred3 = UNet.predict(xx)[..., 0].reshape(inp_shape[:2])
    UNet.load_weights('../input/kfold-unet-ver0/UNET_4.h5')
    pred4 = UNet.predict(xx)[..., 0].reshape(inp_shape[:2])
    UNet.load_weights('../input/kfold-unet-ver0/UNET_5.h5')
    pred5 = UNet.predict(xx)[..., 0].reshape(inp_shape[:2])
    pred = (pred1 + pred2 + pred3 + pred4 + pred5)/5.0
    mask = yy[..., 0].reshape(inp_shape[:2])

    # Binarize masks
    gt = mask > 0.5
    pr = pred > 0.5

    # Remove regions smaller than 2% of the image
    pr = remove_small_regions(pr, 0.02 * np.prod(im_shape))

    #io.imsave('./'+str(count), masked(img, gt, pr, 1))
    export_path = '../working/'
    image_filename = str(count) + str('.png')
    io.imsave(os.path.join(export_path, image_filename), masked(img, gt, pr, 1))

    dices[i] = Dice(gt, pr)
    print(str(count), dices[i])

    i += 1
    if i == n_test:
        break

print('Mean Dice:', dices.mean())
```

3. JSRT data – Convert to NPY format

```
import os
import numpy as np
from numpy import asarray
from numpy import save
from skimage import transform, io, exposure
from matplotlib import image


#NOTE: the "img" and "gif" format files have to be processed only through below process.
#  The JSRT inferenced on UNET shows very good accuracy only using below processing

im_shape = (256,256)

CXR=[]
CXR_Mask=[]

folder1 = 'D:/Users/105035167/GasTurbine/Deep_Learning_L3_Certification/JSRT_Original_Data/Chest_XRays/'
```

```
folder2 = 'D:/Users/105035167/GasTurbine/Deep_Learning_L3_Certification/JSRT_Original_Data/Left_lungMask/'
folder3 = 'D:/Users/105035167/GasTurbine/Deep_Learning_L3_Certification/JSRT_Original_Data/Right_lungMask/'


for i, filename in enumerate(os.listdir(folder1)):
    img = 1 - np.fromfile(folder1 + filename[:-4] + '.img', dtype='>u2').reshape((2048, 2048))*1./4096
    img = transform.resize(img, im_shape)
    img = np.expand_dims(img, -1)
    CXR.append(img)


    left = image.imread(folder2 + filename[:-4] + '.gif')
    right = image.imread(folder3 + filename[:-4] + '.gif')
    img = np.clip(left + right, 0, 255)
    img = transform.resize(img, im_shape)
    img = np.expand_dims(img, -1)
    CXR_Mask.append(img)


# convert to numpy arrays
CXR = asarray(CXR)
CXR_Mask = asarray(CXR_Mask)


print(CXR.shape, CXR_Mask.shape, CXR.min(), CXR.max(), CXR_Mask.min(), CXR_Mask.max())
#save the reshaped CXR_Mask
save('CXR.npy', CXR)
save('CXR_Mask.npy', CXR_Mask)
```

4. CycleGAN domain transformation (JSRT to Montgomery) with ResNet Encoder:

```
!pip install -qq git+https://www.github.com/keras-team/keras-contrib.git
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split

import os
import numpy as np
from skimage import io, exposure
from skimage.filters import laplace
from skimage.transform import resize
import cv2

#create a new output folder
#os.mkdir('/working/CXR_Mask/')

# define location of dataset for Main lung CXR images
folder1 = '../input/jsrt-data/'
folder2 = '../input/montgomery-data-npy-format/'

#Load all the chest Xrays from respective datasets
```

```
jsrt_CXR = np.load(folder1 + 'CXR.npy')
montgomery_CXR = np.load(folder2 + 'CXR.npy')

print(jsrt_CXR.shape,  montgomery_CXR.shape)


from numpy import savez_compressed

# Split into train and test data(67%)
X=np.arange(0,246)
X_train, X_test = train_test_split(X, test_size=0.67, random_state=999)
jsrt_CXR_train = jsrt_CXR[X_train]
jsrt_CXR_test = jsrt_CXR[X_test]

X=np.arange(0,138)
X_train, X_test = train_test_split(X, test_size=0.4, random_state=999)
montgomery_CXR_train = montgomery_CXR[X_train]
montgomery_CXR_test = montgomery_CXR[X_test]

print(jsrt_CXR_train.shape, montgomery_CXR_train.shape)
#combine and save datasets together
savez_compressed('cyclegan_CXR_train.npz', jsrt_CXR_train, montgomery_CXR_train)

save('jsrt_CXR_train.npy', jsrt_CXR_train)
save('montgomery_CXR_train.npy', montgomery_CXR_train)
save('jsrt_CXR_test.npy',jsrt_CXR_test)
save('montgomery_CXR_test.npy', montgomery_CXR_test)


from random import random
from numpy import load
from numpy import zeros
from numpy import ones
from numpy import asarray
from numpy.random import randint
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
from matplotlib import pyplot

# load and prepare training images
def load_real_samples(filename):
        # load the dataset
        data = load(filename)
        # unpack arrays
        X1, X2 = data['arr_0'], data['arr_1']
        # scale from [0,255] to [-1,1]
        X1 = (X1 * 2.0) - 1.0
        X2 = (X2 * 2.0) - 1.0
        return [X1, X2]

# define the discriminator model
def define_discriminator(image_shape):
        # weight initialization
        init = RandomNormal(stddev=0.02)
        # source image input
        in_image = Input(shape=image_shape)
        # C64
```

```
        d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image
)
        d = LeakyReLU(alpha=0.2)(d)
        # C128
        d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # C256
        d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # C512
        d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # second last output layer
        d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # patch output
        patch_out = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
        # define model
        model = Model(in_image, patch_out)
        # compile model
        model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5), loss_weights=[0.5])
        return model

# generator a resnet block
def resnet_block(n_filters, input_layer):
        # weight initialization
        init = RandomNormal(stddev=0.02)
        # first layer convolutional layer
        g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(input_layer)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # second convolutional layer
        g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(g)
        g = InstanceNormalization(axis=-1)(g)
        # concatenate merge channel-wise with input layer
        g = Concatenate()([g, input_layer])
        return g

# define the standalone generator model
def define_generator(image_shape, n_resnet=9):
        # weight initialization
        init = RandomNormal(stddev=0.02)
        # image input
        in_image = Input(shape=image_shape)
        # c7s1-64
        g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # d128
        g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # d256
        g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # R256
        for _ in range(n_resnet):
                g = resnet_block(256, g)
        # u128
```

```
        g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init
)(g)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # u64
        g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)
(g)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # c7s1-3
    #g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
        g = Conv2D(1, (7,7), padding='same', kernel_initializer=init)(g)   # 1 channel for gra
yscale
        g = InstanceNormalization(axis=-1)(g)
        out_image = Activation('tanh')(g)
        # define model
        model = Model(in_image, out_image)
        return model

# define a composite model for updating generators by adversarial and cycle loss
def define_composite_model(g_model_1, d_model, g_model_2, image_shape):
        # ensure the model we're updating is trainable
        g_model_1.trainable = True
        # mark discriminator as not trainable
        d_model.trainable = False
        # mark other generator model as not trainable
        g_model_2.trainable = False
        # discriminator element
        input_gen = Input(shape=image_shape)
        gen1_out = g_model_1(input_gen)
        output_d = d_model(gen1_out)
        # identity element
        input_id = Input(shape=image_shape)
        output_id = g_model_1(input_id)
        # forward cycle
        output_f = g_model_2(gen1_out)
        # backward cycle
        gen2_out = g_model_2(input_id)
        output_b = g_model_1(gen2_out)
        # define model graph
        model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
        # define optimization algorithm configuration
        opt = Adam(lr=0.0002, beta_1=0.5)
        # compile model with weighting of least squares loss and L1 loss
        model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10], optimize
r=opt)
        return model

# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
        # choose random instances
        ix = randint(0, dataset.shape[0], n_samples)
        # retrieve selected images
        X = dataset[ix]
        # generate 'real' class labels (1)
        y = ones((n_samples, patch_shape, patch_shape, 1))
        return X, y

# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, dataset, patch_shape):
        # generate fake instance
        X = g_model.predict(dataset)
        # create 'fake' class labels (0)
        y = zeros((len(X), patch_shape, patch_shape, 1))
        return X, y
```

```
# save the generator models to file
def save_models(step, g_model_AtoB, g_model_BtoA):
	# save the first generator model
	filename1 = 'g_model_AtoB.h5'
	g_model_AtoB.save(filename1)
	# save the second generator model
	filename2 = 'g_model_BtoA.h5'
	g_model_BtoA.save(filename2)
	print('>Saved: %s and %s' % (filename1, filename2))

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, trainX, name, n_samples=5):
	step = step/trainX.shape[0]
	# select a sample of input images
	X_in, _ = generate_real_samples(trainX, n_samples, 0)
	# generate translated images
	X_out, _ = generate_fake_samples(g_model, X_in, 0)
	# scale all pixels from [-1,1] to [0,1]
	X_in = (X_in + 1) / 2.0
	X_out = (X_out + 1) / 2.0
	X_in = np.squeeze(X_in)    #convert to one channel (3D to 1D)
	X_out = np.squeeze(X_out)  #convert to one channel (3D to 1D)
	# plot real images
	for i in range(n_samples):
		pyplot.subplot(2, n_samples, 1 + i)
		pyplot.axis('off')
		pyplot.imshow(X_in[i], cmap='gray')
	# plot translated image
	for i in range(n_samples):
		pyplot.subplot(2, n_samples, 1 + n_samples + i)
		pyplot.axis('off')
		pyplot.imshow(X_out[i], cmap='gray')
	# save plot to file
	filename1 = '%s_generated_plot_%06d.png' % (name, (step+1))
	pyplot.savefig(filename1)
	pyplot.close()

# update image pool for fake images
def update_image_pool(pool, images, max_size=50):
	selected = list()
	for image in images:
		if len(pool) < max_size:
			# stock the pool
			pool.append(image)
			selected.append(image)
		elif random() < 0.5:
			# use image, but don't add it to the pool
			selected.append(image)
		else:
			# replace an existing image and use replaced image
			ix = randint(0, len(pool))
			selected.append(pool[ix])
			pool[ix] = image
	return asarray(selected)

# train cyclegan models
def train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset
):
	# define properties of the training run
	n_epochs, n_batch, = 150, 1
	# determine the output square shape of the discriminator
	n_patch = d_model_A.output_shape[1]
	# unpack dataset
	trainA, trainB = dataset
```

```
        # prepare image pool for fakes
        poolA, poolB = list(), list()
        # calculate the number of batches per training epoch
        bat_per_epo = int(len(trainA) / n_batch)
        # calculate the number of training iterations
        n_steps = bat_per_epo * n_epochs
        # manually enumerate epochs
        for i in range(n_steps):
                # select a batch of real samples
                X_realA, y_realA = generate_real_samples(trainA, n_batch, n_patch)
                X_realB, y_realB = generate_real_samples(trainB, n_batch, n_patch)
                # generate a batch of fake samples
                X_fakeA, y_fakeA = generate_fake_samples(g_model_BtoA, X_realB, n_patch)
                X_fakeB, y_fakeB = generate_fake_samples(g_model_AtoB, X_realA, n_patch)
                # update fakes from pool
                X_fakeA = update_image_pool(poolA, X_fakeA)
                X_fakeB = update_image_pool(poolB, X_fakeB)
                # update generator B->A via adversarial and cycle loss
                g_loss2, _, _, _, _  = c_model_BtoA.train_on_batch([X_realB, X_realA], [y_real
A, X_realA, X_realB, X_realA])
                # update discriminator for A -> [real/fake]
                dA_loss1 = d_model_A.train_on_batch(X_realA, y_realA)
                dA_loss2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
                # update generator A->B via adversarial and cycle loss
                g_loss1, _, _, _, _ = c_model_AtoB.train_on_batch([X_realA, X_realB], [y_realB
, X_realB, X_realA, X_realB])
                # update discriminator for B -> [real/fake]
                dB_loss1 = d_model_B.train_on_batch(X_realB, y_realB)
                dB_loss2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
                # summarize performance
                print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (i+1, dA_loss1,dA_loss
2, dB_loss1,dB_loss2, g_loss1,g_loss2))
                # evaluate the model performance every so often
                if (i+1) % (bat_per_epo * 1) == 0:
                        # plot A->B translation
                        summarize_performance(i, g_model_AtoB, trainA, 'AtoB')
                        # plot B->A translation
                        summarize_performance(i, g_model_BtoA, trainB, 'BtoA')
                if (i+1) % (bat_per_epo * 5) == 0:
                        # save the models
                        save_models(i, g_model_AtoB, g_model_BtoA)


from keras.utils.vis_utils import model_to_dot
import os
from IPython.display import Image

#Image(model_to_dot(g_model_AtoB, show_shapes=True).create_png())

# load image data
dataset = load_real_samples('cyclegan_CXR_train.npz')
print('Loaded', dataset[0].shape, dataset[1].shape)
# define input shape based on the loaded dataset
image_shape = dataset[0].shape[1:]
# generator: A -> B
g_model_AtoB = define_generator(image_shape)
# generator: B -> A
g_model_BtoA = define_generator(image_shape)
# discriminator: A -> [real/fake]
d_model_A = define_discriminator(image_shape)
# discriminator: B -> [real/fake]
d_model_B = define_discriminator(image_shape)
# composite: A -> B -> [real/fake, A]
c_model_AtoB = define_composite_model(g_model_AtoB, d_model_B, g_model_BtoA, image_shape)
# composite: B -> A -> [real/fake, B]
c_model_BtoA = define_composite_model(g_model_BtoA, d_model_A, g_model_AtoB, image_shape)
```

```
# train models
train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset)
```

5. CycleGAN domain transformation (JSRT to Montgomery) using U-Net as Encoder

```
!pip install -qq git+https://www.github.com/keras-team/keras-contrib.git
```

```
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split

import os
import numpy as np
from skimage import io, exposure
from skimage.filters import laplace
from skimage.transform import resize
import cv2

#create a new output folder
#os.mkdir('/working/CXR_Mask/')

# define location of dataset for Main lung CXR images
folder1 = '../input/jsrt-data/'
folder2 = '../input/montgomery-data-npy-format/'

#Load all the chest Xrays from respective datasets
jsrt_CXR = np.load(folder1 + 'CXR.npy')
montgomery_CXR = np.load(folder2 + 'CXR.npy')

print(jsrt_CXR.shape,  montgomery_CXR.shape)


from numpy import savez_compressed

# Split into train and test data(67%)
X=np.arange(0,246)
X_train, X_test = train_test_split(X, test_size=0.67, random_state=999)
jsrt_CXR_train = jsrt_CXR[X_train]
jsrt_CXR_test = jsrt_CXR[X_test]

X=np.arange(0,138)
X_train, X_test = train_test_split(X, test_size=0.4, random_state=999)
montgomery_CXR_train = montgomery_CXR[X_train]
montgomery_CXR_test = montgomery_CXR[X_test]

print(jsrt_CXR_train.shape, montgomery_CXR_train.shape)
#combine and save datasets together
savez_compressed('cyclegan_CXR_train.npz', jsrt_CXR_train, montgomery_CXR_train)

save('jsrt_CXR_train.npy', jsrt_CXR_train)
save('montgomery_CXR_train.npy', montgomery_CXR_train)
save('jsrt_CXR_test.npy',jsrt_CXR_test)
save('montgomery_CXR_test.npy', montgomery_CXR_test)


from random import random
from numpy import load
from numpy import zeros
from numpy import ones
from numpy import asarray
from numpy.random import randint
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
```

```
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
from matplotlib import pyplot

# load and prepare training images
def load_real_samples(filename):
        # load the dataset
        data = load(filename)
        # unpack arrays
        X1, X2 = data['arr_0'], data['arr_1']
        # scale from [0,255] to [-1,1]
        X1 = (X1 * 2.0) - 1.0
        X2 = (X2 * 2.0) - 1.0
        return [X1, X2]

# define the discriminator model
def define_discriminator(image_shape):
        # weight initialization
        init = RandomNormal(stddev=0.02)
        # source image input
        in_image = Input(shape=image_shape)
        # C64
        d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image
)
        d = LeakyReLU(alpha=0.2)(d)
        # C128
        d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # C256
        d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # C512
        d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # second last output layer
        d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
        d = InstanceNormalization(axis=-1)(d)
        d = LeakyReLU(alpha=0.2)(d)
        # patch output
        patch_out = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
        # define model
        model = Model(in_image, patch_out)
        # compile model
        model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5), loss_weights=[0.5])
        return model

# Applying UNET for generator=====================================
from keras.models import Model
from keras.layers.merge import concatenate
from keras.layers import Input, Convolution2D, MaxPooling2D, UpSampling2D

# define the standalone generator model
def define_generator(image_shape):
        # weight initialization
        init = RandomNormal(stddev=0.02)
        # image input
        in_image = Input(shape=image_shape)
```

```
        # c7s1-64
        g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # d128
        g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # d256
        g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # R256
        #==============UNET STARTS=============================================
        k_size = 3
        merge_axis = -1 # Feature maps are concatenated along last axis (for tf backend)
        conv1 = Convolution2D(filters=32, kernel_size=k_size, padding='same', activation='relu
')(g)
        conv1 = Convolution2D(filters=32, kernel_size=k_size, padding='same', activation='relu
')(conv1)
        pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
        u1 = InstanceNormalization(axis=-1)(pool1)

        conv2 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu
')(u1)
        conv2 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu
')(conv2)
        pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
        u2 = InstanceNormalization(axis=-1)(pool2)

        conv3 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu
')(u2)
        conv3 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu
')(conv3)
        pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
        u3 = InstanceNormalization(axis=-1)(pool3)

        conv4 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='rel
u')(u3)
        conv4 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='rel
u')(conv4)
        pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)
        u4 = InstanceNormalization(axis=-1)(pool4)

        conv5 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='rel
u')(u4)

        up1 = UpSampling2D(size=(2, 2))(conv5)
        conv6 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='rel
u')(up1)
        conv6 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='rel
u')(conv6)
        u6 = InstanceNormalization(axis=-1)(conv6)
        merged1 = concatenate([conv4, u6], axis=merge_axis)
        conv6 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='rel
u')(merged1)

        up2 = UpSampling2D(size=(2, 2))(conv6)
        conv7 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='rel
u')(up2)
        conv7 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='rel
u')(conv7)
        u7 = InstanceNormalization(axis=-1)(conv7)
        merged2 = concatenate([conv3, u7], axis=merge_axis)
```

```
        conv7 = Convolution2D(filters=256, kernel_size=k_size, padding='same', activation='rel
u')(merged2)

        up3 = UpSampling2D(size=(2, 2))(conv7)
        conv8 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='rel
u')(up3)
        conv8 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='rel
u')(conv8)
        u8 = InstanceNormalization(axis=-1)(conv8)
        merged3 = concatenate([conv2, u8], axis=merge_axis)
        conv8 = Convolution2D(filters=128, kernel_size=k_size, padding='same', activation='rel
u')(merged3)

        up4 = UpSampling2D(size=(2, 2))(conv8)
        conv9 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu
')(up4)
        conv9 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu
')(conv9)
        u9 = InstanceNormalization(axis=-1)(conv9)
        merged4 = concatenate([conv1, u9], axis=merge_axis)
        conv9 = Convolution2D(filters=64, kernel_size=k_size, padding='same', activation='relu
')(merged4)

        conv10 = Convolution2D(filters=1, kernel_size=k_size, padding='same', activation='sigm
oid')(conv9)
        #================================UNET Ends====================
        # u128
        g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init
)(conv10)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # u64
        g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)
(g)
        g = InstanceNormalization(axis=-1)(g)
        g = Activation('relu')(g)
        # c7s1-3
    #g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
        g = Conv2D(1, (7,7), padding='same', kernel_initializer=init)(g)   # 1 channel for gra
yscale
        g = InstanceNormalization(axis=-1)(g)
        out_image = Activation('tanh')(g)
        # define model
        model = Model(in_image, out_image)
        return model

# define a composite model for updating generators by adversarial and cycle loss
def define_composite_model(g_model_1, d_model, g_model_2, image_shape):
        # ensure the model we're updating is trainable
        g_model_1.trainable = True
        # mark discriminator as not trainable
        d_model.trainable = False
        # mark other generator model as not trainable
        g_model_2.trainable = False
        # discriminator element
        input_gen = Input(shape=image_shape)
        gen1_out = g_model_1(input_gen)
        output_d = d_model(gen1_out)
        # identity element
        input_id = Input(shape=image_shape)
        output_id = g_model_1(input_id)
        # forward cycle
        output_f = g_model_2(gen1_out)
        # backward cycle
        gen2_out = g_model_2(input_id)
```

```
        output_b = g_model_1(gen2_out)
        # define model graph
        model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
        # define optimization algorithm configuration
        opt = Adam(lr=0.0002, beta_1=0.5)
        # compile model with weighting of least squares loss and L1 loss
        model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10], optimize
r=opt)
        return model

# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
        # choose random instances
        ix = randint(0, dataset.shape[0], n_samples)
        # retrieve selected images
        X = dataset[ix]
        # generate 'real' class labels (1)
        y = ones((n_samples, patch_shape, patch_shape, 1))
        return X, y

# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, dataset, patch_shape):
        # generate fake instance
        X = g_model.predict(dataset)
        # create 'fake' class labels (0)
        y = zeros((len(X), patch_shape, patch_shape, 1))
        return X, y

# save the generator models to file
def save_models(step, g_model_AtoB, g_model_BtoA):
        # save the first generator model
        filename1 = 'g_model_AtoB.h5'
        g_model_AtoB.save(filename1)
        # save the second generator model
        filename2 = 'g_model_BtoA.h5'
        g_model_BtoA.save(filename2)
        print('>Saved: %s and %s' % (filename1, filename2))

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, trainX, name, n_samples=5):
        step = step/trainX.shape[0]
        # select a sample of input images
        X_in, _ = generate_real_samples(trainX, n_samples, 0)
        # generate translated images
        X_out, _ = generate_fake_samples(g_model, X_in, 0)
        # scale all pixels from [-1,1] to [0,1]
        X_in = (X_in + 1) / 2.0
        X_out = (X_out + 1) / 2.0
        X_in = np.squeeze(X_in)    #convert to one channel (3D to 1D)
        X_out = np.squeeze(X_out)  #convert to one channel (3D to 1D)
        # plot real images
        for i in range(n_samples):
                pyplot.subplot(2, n_samples, 1 + i)
                pyplot.axis('off')
                pyplot.imshow(X_in[i], cmap='gray')
        # plot translated image
        for i in range(n_samples):
                pyplot.subplot(2, n_samples, 1 + n_samples + i)
                pyplot.axis('off')
                pyplot.imshow(X_out[i], cmap='gray')
        # save plot to file
        filename1 = '%s_generated_plot_%06d.png' % (name, (step+1))
        pyplot.savefig(filename1)
        pyplot.close()
```

```
# update image pool for fake images
def update_image_pool(pool, images, max_size=50):
	selected = list()
	for image in images:
		if len(pool) < max_size:
			# stock the pool
			pool.append(image)
			selected.append(image)
		elif random() < 0.5:
			# use image, but don't add it to the pool
			selected.append(image)
		else:
			# replace an existing image and use replaced image
			ix = randint(0, len(pool))
			selected.append(pool[ix])
			pool[ix] = image
	return asarray(selected)

# train cyclegan models
def train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset
):
	# define properties of the training run
	n_epochs, n_batch, = 200, 1
	# determine the output square shape of the discriminator
	n_patch = d_model_A.output_shape[1]
	# unpack dataset
	trainA, trainB = dataset
	# prepare image pool for fakes
	poolA, poolB = list(), list()
	# calculate the number of batches per training epoch
	bat_per_epo = int(len(trainA) / n_batch)
	# calculate the number of training iterations
	n_steps = bat_per_epo * n_epochs
	# manually enumerate epochs
	for i in range(n_steps):
		# select a batch of real samples
		X_realA, y_realA = generate_real_samples(trainA, n_batch, n_patch)
		X_realB, y_realB = generate_real_samples(trainB, n_batch, n_patch)
		# generate a batch of fake samples
		X_fakeA, y_fakeA = generate_fake_samples(g_model_BtoA, X_realB, n_patch)
		X_fakeB, y_fakeB = generate_fake_samples(g_model_AtoB, X_realA, n_patch)
		# update fakes from pool
		X_fakeA = update_image_pool(poolA, X_fakeA)
		X_fakeB = update_image_pool(poolB, X_fakeB)
		# update generator B->A via adversarial and cycle loss
		g_loss2, _, _, _, _  = c_model_BtoA.train_on_batch([X_realB, X_realA], [y_real
A, X_realA, X_realB, X_realA])
		# update discriminator for A -> [real/fake]
		dA_loss1 = d_model_A.train_on_batch(X_realA, y_realA)
		dA_loss2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
		# update generator A->B via adversarial and cycle loss
		g_loss1, _, _, _, _ = c_model_AtoB.train_on_batch([X_realA, X_realB], [y_realB
, X_realB, X_realA, X_realB])
		# update discriminator for B -> [real/fake]
		dB_loss1 = d_model_B.train_on_batch(X_realB, y_realB)
		dB_loss2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
		# summarize performance
		print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (i+1, dA_loss1,dA_loss
2, dB_loss1,dB_loss2, g_loss1,g_loss2))
		# evaluate the model performance every so often
		if (i+1) % (bat_per_epo * 1) == 0:
			# plot A->B translation
			summarize_performance(i, g_model_AtoB, trainA, 'AtoB')
			# plot B->A translation
			summarize_performance(i, g_model_BtoA, trainB, 'BtoA')
```

```
                if (i+1) % (bat_per_epo * 5) == 0:
                        # save the models
                        save_models(i, g_model_AtoB, g_model_BtoA)


from keras.utils.vis_utils import model_to_dot
import os
from IPython.display import Image

#Image(model_to_dot(g_model_AtoB, show_shapes=True).create_png())

# load image data
dataset = load_real_samples('cyclegan_CXR_train.npz')
print('Loaded', dataset[0].shape, dataset[1].shape)
# define input shape based on the loaded dataset
image_shape = dataset[0].shape[1:]
# generator: A -> B
g_model_AtoB = define_generator(image_shape)
# generator: B -> A
g_model_BtoA = define_generator(image_shape)
# discriminator: A -> [real/fake]
d_model_A = define_discriminator(image_shape)
# discriminator: B -> [real/fake]
d_model_B = define_discriminator(image_shape)
# composite: A -> B -> [real/fake, A]
c_model_AtoB = define_composite_model(g_model_AtoB, d_model_B, g_model_BtoA, image_shape)
# composite: B -> A -> [real/fake, B]
c_model_BtoA = define_composite_model(g_model_BtoA, d_model_A, g_model_AtoB, image_shape)
# train models
train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset)
```

6.CycleGAN inference to convert JSRT images to Montgomery domain:

```
# example of using saved cyclegan models for image translation
from keras.models import load_model
from numpy import load
from numpy import vstack
from matplotlib import pyplot
from numpy.random import randint
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
import numpy as np

# load and prepare training images
def load_real_samples(filename):
        # load the dataset
        data = load(filename)
        # unpack arrays
        X1, X2 = data['arr_0'], data['arr_1']
        # scale from [0,1] to [-1,1]
        X1 = (X1 * 2.0) - 1.0
        X2 = (X2 * 2.0) - 1.0
        return [X1, X2]

# select a random sample of images from the dataset
def select_sample(dataset, n_samples):
        # choose random instances
        ix = randint(0, dataset.shape[0], n_samples)
        print(ix)
        # retrieve selected images
        X = dataset[ix]
        return X

# plot the image, the translation, and the reconstruction
def show_plot(imagesX, imagesY1, imagesY2):
```

```
        images = vstack((imagesX, imagesY1, imagesY2))
        titles = ['Real', 'Generated', 'Reconstructed']
        images = np.squeeze(images)
        # scale from [-1,1] to [0,1]
        images = (images + 1) / 2.0
        # plot images row by row
        for i in range(len(images)):
                # define subplot
                pyplot.subplot(1, len(images), 1 + i)
                # turn off axis
                pyplot.axis('off')
                # plot raw pixel data
                pyplot.imshow(images[i], cmap='gray')
                # title
                pyplot.title(titles[i])
        pyplot.show()

# load dataset
A_data, B_data = load_real_samples('../input/cyclegan-ver5-outputs/cyclegan_CXR_train.npz')
print('Loaded', A_data.shape, B_data.shape)
print(A_data.min(), A_data.max())
# load the models
cust = {'InstanceNormalization': InstanceNormalization}
model_AtoB = load_model('../input/cyclegan-ver5-outputs/g_model_AtoB.h5', cust)
model_BtoA = load_model('../input/cyclegan-ver5-outputs/g_model_BtoA.h5', cust)
# plot A->B->A
A_real = select_sample(A_data, 1)
print(A_real.min(), A_real.max())
B_generated  = model_AtoB.predict(A_real)
print(B_generated.min(), B_generated.max())
A_reconstructed = model_BtoA.predict(B_generated)
show_plot(A_real, B_generated, A_reconstructed)
# plot B->A->B
B_real = select_sample(B_data, 1)
A_generated  = model_BtoA.predict(B_real)
B_reconstructed = model_AtoB.predict(A_generated)
show_plot(B_real, A_generated, B_reconstructed)
```

```
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split

import os
import numpy as np
from skimage import io, exposure
from skimage.filters import laplace
from skimage.transform import resize
import cv2

# define location of dataset for Main Lung CXR images
folder1 = '../input/cyclegan-ver5-outputs/'

#Load all the chest Xrays from respective datasets
jsrt_CXR = np.load(folder1 + 'jsrt_CXR_test.npy')
jsrt_CXR = (jsrt_CXR * 2) - 1
print(jsrt_CXR.shape, jsrt_CXR.min(), jsrt_CXR.max())
```

```
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
```

```
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split

import os
import numpy as np
from skimage import io, exposure
from skimage.filters import laplace
from skimage.transform import resize
import cv2

# define location of dataset for Main Lung CXR images
folder1 = '../input/cyclegan-ver5-outputs/'

#Load all the chest Xrays from respective datasets
jsrt_CXR = np.load(folder1 + 'jsrt_CXR_test.npy')
#convert array from [0,1] to [-1,1] as input for GAN inference
jsrt_CXR = (jsrt_CXR * 2) - 1
print(jsrt_CXR.shape, jsrt_CXR.min(), jsrt_CXR.max())

jsrt_GAN_CXR=[]
for i in range(0,(jsrt_CXR.shape[0])):
    A_real = jsrt_CXR[[i]]
    B_generated  = model_AtoB.predict(A_real)
    jsrt_GAN_CXR.append(B_generated[0])

from numpy import asarray
jsrt_GAN_CXR=asarray(jsrt_GAN_CXR)
# convert from [-1,1] to [0,1] for UNET inference
jsrt_GAN_CXR=(jsrt_GAN_CXR+1)/2
print(jsrt_GAN_CXR.shape, jsrt_GAN_CXR.min(), jsrt_GAN_CXR.max())


print(jsrt_GAN_CXR.shape)
save('jsrt_GAN_CXR.npy', jsrt_GAN_CXR)

from sklearn.model_selection import train_test_split

jsrt_CXR_Mask = np.load('../input/jsrt-data/CXR_Mask.npy')
# Split into train and test data(67%)
X=np.arange(0,246)
X_train, X_test = train_test_split(X, test_size=0.67, random_state=999)
jsrt_CXR_GAN_Mask = jsrt_CXR_Mask[X_test]
print(jsrt_CXR_GAN_Mask.shape)
```

7.Enhanced U-Net with EfficientNet-B7 Backbone:

```
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split

import os
import numpy as np
from skimage import io, exposure
from skimage.filters import laplace
from skimage.transform import resize
import cv2

#============================
from skimage.transform import resize
```

```
import tensorflow as tf
import keras.backend as K
from keras.losses import binary_crossentropy

from tensorflow.keras.applications import EfficientNetB7
from keras.preprocessing.image import load_img
from keras import Model
from keras.callbacks import  ModelCheckpoint
from keras.layers import Input, Conv2D, Conv2DTranspose, MaxPooling2D, concatenate, Dropout,Bat
chNormalization
from keras.layers import Conv2D, Concatenate, MaxPooling2D
from keras.layers import UpSampling2D, Dropout, BatchNormalization
from tqdm import tqdm_notebook
from keras import initializers
from keras import regularizers
from keras import constraints
from keras.utils import conv_utils
from keras.utils.data_utils import get_file
from keras.engine.topology import get_source_inputs
from keras.engine import InputSpec
from keras import backend as K
from keras.layers import LeakyReLU
from keras.layers import ZeroPadding2D
from keras.losses import binary_crossentropy
import keras.callbacks as callbacks
from keras.callbacks import Callback
from keras.applications.xception import Xception
from keras.layers import multiply

from keras import optimizers
from keras.utils.generic_utils import get_custom_objects

from keras.engine.topology import Input
from keras.engine.training import Model
from keras.layers.convolutional import Conv2D, UpSampling2D, Conv2DTranspose
from keras.layers.core import Activation, SpatialDropout2D
from keras.layers.merge import concatenate
from keras.layers.normalization import BatchNormalization
from keras.layers.pooling import MaxPooling2D
from keras.layers import Input,Dropout,BatchNormalization,Activation,Add
from keras.regularizers import l2
from keras.layers.core import Dense, Lambda
from keras.layers.merge import concatenate, add
from keras.layers import GlobalAveragePooling2D, Reshape, Dense, multiply, Permute
from keras.optimizers import SGD, Adam
from keras.preprocessing.image import ImageDataGenerator

import glob
import shutil
import os
import random
from PIL import Image

#============================================

#create a new output folder
#os.mkdir('/working/CXR_Mask/')

# define location of dataset for Main lung CXR images
folder = '../input/montgomery-data/Main Problem/MainLungImages/'
CXR, CXR_Mask = list(), list()
im_shape = (256, 256)

def make_masks():
    path = '../input/montgomery-data/Main Problem/MainLungImages/'
    for i, filename in enumerate(os.listdir(path)):
```

```
        # load CXR image
        photo = cv2.imread(folder + filename)  #Read in image with all 3 channels
        photo = photo.astype('float32')/255.0
        photo = cv2.resize(photo, dsize=im_shape, interpolation=cv2.INTER_LINEAR)
        CXR.append(photo)

        left = cv2.imread('../input/montgomery-data/Main Problem/leftMask/' + filename[:-4] + '
.png',0)
        right = cv2.imread('../input/montgomery-data/Main Problem/rightMask/' + filename[:-4] +
'.png',0)
        fullmaskarray = np.clip(left + right, 0, 255).astype('float32')/255.0
        fullmaskarray = cv2.resize(fullmaskarray, dsize=im_shape, interpolation=cv2.INTER_LINEA
R)
        CXR_Mask.append(fullmaskarray)
        #print ('CXR_Mask', i, filename)

make_masks()
# convert to numpy arrays
CXR = asarray(CXR)
#CXR -= CXR.mean()
#CXR /= CXR.std()
CXR_Mask = asarray(CXR_Mask)

CXR = CXR.reshape(CXR.shape[0], 256, 256, 3)
CXR_Mask = CXR_Mask.reshape(CXR_Mask.shape[0], 256, 256, 1)

print(CXR.shape, CXR_Mask.shape, CXR.min(), CXR.max(), CXR_Mask.min(), CXR_Mask.max())
#save the reshaped CXR_Mask
#save('CXR.npy', CXR)
#save('CXR_Mask.npy', CXR_Mask)


# Split into train and test data(20%)
X=np.arange(0,138)
X_train, X_test = train_test_split(X, test_size=0.2, random_state=999)
#Now prepare a validation dataset
X_train, X_valid = train_test_split(X_train, test_size=0.15, random_state=999)

CXR_train=[]
CXR_Mask_train=[]
for i in X_train:
    CXR_train.append(CXR[i,:,:,:])
    CXR_Mask_train.append(CXR_Mask[i,:,:,:])
CXR_train=np.array(CXR_train)
CXR_Mask_train=np.array(CXR_Mask_train)

CXR_test=[]
CXR_Mask_test=[]
for i in X_test:
    CXR_test.append(CXR[i,:,:,:])
    CXR_Mask_test.append(CXR_Mask[i,:,:,:])
CXR_test=np.array(CXR_test)
CXR_Mask_test=np.array(CXR_Mask_test)

CXR_valid=[]
CXR_Mask_valid=[]
for i in X_valid:
    CXR_valid.append(CXR[i,:,:,:])
    CXR_Mask_valid.append(CXR_Mask[i,:,:,:])
CXR_valid=np.array(CXR_valid)
CXR_Mask_valid=np.array(CXR_Mask_valid)

print(CXR.shape, CXR_train.shape, CXR_valid.shape)

save('CXR_train.npy', CXR_train)
```

```
save('CXR_Mask_train.npy', CXR_Mask_train)
save('CXR_test.npy', CXR_test)
save('CXR_Mask_test.npy', CXR_Mask_test)
save('CXR_valid.npy', CXR_test)
save('CXR_Mask_valid.npy', CXR_Mask_test)


# Plot few images using preprocessed arrays

#plt.figure(1)
#subplot(r,c) provide the no. of rows and columns
#f, axarr = plt.subplots( nrows=5, ncols=2, sharex=True, sharey=True, figsize = (6.5,16), grids
pec_kw = {'wspace':0, 'hspace':0})

# use the created array to output your multiple images. In this case I have stacked 4 images ve
rtically
#t_img, m_img = CXR[0].squeeze(), CXR_Mask[0].squeeze(); axarr[0,0].imshow(t_img);  axarr[0,1].
imshow(m_img);
#t_img, m_img = CXR[10].squeeze(), CXR_Mask[10].squeeze(); axarr[1,0].imshow(t_img);  axarr[1,1
].imshow(m_img);
#t_img, m_img = CXR[59].squeeze(), CXR_Mask[59].squeeze(); axarr[2,0].imshow(t_img);  axarr[2,1
].imshow(m_img);
#t_img, m_img = CXR[104].squeeze(), CXR_Mask[104].squeeze(); axarr[3,0].imshow(t_img);  axarr[3
,1].imshow(m_img);
#t_img, m_img = CXR[134].squeeze(), CXR_Mask[134].squeeze(); axarr[4,0].imshow(t_img);  axarr[4
,1].imshow(m_img);

#plt.gray()
#plt.show()
#plt.tight_layout()
#plt.subplots_adjust(wspace=0, hspace=0)

# Input "X" or "CXR" are chest X Ray images . Output "y"s are the segmentaion mask images
CXR_train, seg_train = CXR_train, CXR_Mask_train
CXR_valid, seg_valid = CXR_valid, CXR_Mask_valid
CXR_test, seg_test = CXR_test, CXR_Mask_test

#Below code is to check if the code works on cpu
#CXR_train= CXR_train[0:48]
#seg_train= seg_train[0:16]

num_train_samples = len(CXR_train)
num_val_samples = len(CXR_valid)
num_test_samples = len(CXR_test)

train_batch_size = 8
train_steps = np.ceil(num_train_samples / train_batch_size)

val_batch_size = 8
val_steps = np.ceil(num_val_samples / val_batch_size)

test_batch_size = 8
test_steps = np.ceil(num_test_samples / test_batch_size)

from keras.preprocessing.image import ImageDataGenerator
dg_args = dict(    rescale=1.,
                   rotation_range = 10,
                   width_shift_range = 0.1,
                   height_shift_range = 0.1,
                   shear_range = 0.01,
                   zoom_range = [0.8, 1.2],
                   fill_mode = 'nearest',
                   data_format = 'channels_last')

image_gen = ImageDataGenerator(**dg_args)
```

```
def gen_augmented_pairs(in_cxr, in_seg, batch_size):
    while True:
        seed = np.random.choice(range(9999))
        # keep the seeds syncronized otherwise the augmentation to the images is different from
the masks
        g_cxr = image_gen.flow(in_cxr, batch_size = batch_size, seed = seed)
        g_seg = image_gen.flow(in_seg, batch_size = batch_size, seed = seed)
        for i_cxr, i_seg in zip(g_cxr, g_seg):
            yield i_cxr, i_seg

train_gen = gen_augmented_pairs(CXR_train, seg_train, batch_size = train_batch_size)

#image_gen = ImageDataGenerator(rescale=1.,validation_split=0.15)
image_gen = ImageDataGenerator(rescale=1.)
valid_gen = gen_augmented_pairs(CXR_valid, seg_valid, batch_size = val_batch_size)

image_gen = ImageDataGenerator(rescale=1.)
test_gen = gen_augmented_pairs(CXR_test, seg_test, batch_size = test_batch_size)

#only to check if above code has worked correctly,not used anywhere else
train_X, train_Y = next(train_gen)
test_X, test_Y = next(test_gen)
print(train_X.shape, train_Y.shape)
print(test_X.shape, test_Y.shape)


#====Defining Dice Loss
def dice_coef(y_true, y_pred, smooth=1):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth)

def dice_loss(y_true, y_pred):
    smooth = 1.
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = y_true_f * y_pred_f
    score = (2. * K.sum(intersection) + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth)
    return 1. -score

def bce_dice_loss(y_true, y_pred):
    return binary_crossentropy(y_true, y_pred) + dice_loss(y_true, y_pred)

def bce_logdice_loss(y_true, y_pred):
    return binary_crossentropy(y_true, y_pred) - K.log(1. - dice_loss(y_true, y_pred))


#=====================================BUILD UNET MODEL=================================
=======
def convolution_block(x, filters, size, strides=(1,1), padding='same', activation=True):
    x = Conv2D(filters, size, strides=strides, padding=padding)(x)
    x = BatchNormalization()(x)
    if activation == True:
        x = LeakyReLU(alpha=0.1)(x)
    return x

def residual_block(blockInput, num_filters=16):
    x = LeakyReLU(alpha=0.1)(blockInput)
    x = BatchNormalization()(x)
    blockInput = BatchNormalization()(blockInput)
    x = convolution_block(x, num_filters, (3,3) )
    x = convolution_block(x, num_filters, (3,3), activation=False)
    x = Add()([x, blockInput])
```

```
    return x

#=======Defining UEfficientNet Model
def UEfficientNet(input_shape=(None, None, 3),dropout_rate=0.0):

    backbone = EfficientNetB7(weights='imagenet',
                              include_top=False,
                              input_shape=input_shape)
    input = backbone.input
    start_neurons = 8

    conv4 = backbone.layers[548].output
    print(conv4.shape)
    conv4 = LeakyReLU(alpha=0.1)(conv4)
    pool4 = MaxPooling2D((2, 2))(conv4)
    pool4 = Dropout(dropout_rate)(pool4)

     # Middle
    convm = Conv2D(start_neurons * 32, (3, 3), activation=None, padding="same",name='conv_middl
e')(pool4)
    convm = residual_block(convm,start_neurons * 32)
    convm = residual_block(convm,start_neurons * 32)
    convm = LeakyReLU(alpha=0.1)(convm)

    deconv4 = Conv2DTranspose(start_neurons * 16, (3, 3), strides=(2, 2), padding="same")(convm
)
    deconv4_up1 = Conv2DTranspose(start_neurons * 16, (3, 3), strides=(2, 2), padding="same")(d
econv4)
    deconv4_up2 = Conv2DTranspose(start_neurons * 16, (3, 3), strides=(2, 2), padding="same")(d
econv4_up1)
    deconv4_up3 = Conv2DTranspose(start_neurons * 16, (3, 3), strides=(2, 2), padding="same")(d
econv4_up2)
    uconv4 = concatenate([deconv4, conv4])
    uconv4 = Dropout(dropout_rate)(uconv4)

    uconv4 = Conv2D(start_neurons * 16, (3, 3), activation=None, padding="same")(uconv4)
    uconv4 = residual_block(uconv4,start_neurons * 16)
#     uconv4 = residual_block(uconv4,start_neurons * 16)
    uconv4 = LeakyReLU(alpha=0.1)(uconv4)  #conv1_2

    deconv3 = Conv2DTranspose(start_neurons * 8, (3, 3), strides=(2, 2), padding="same")(uconv4
)
    deconv3_up1 = Conv2DTranspose(start_neurons * 8, (3, 3), strides=(2, 2), padding="same")(de
conv3)
    deconv3_up2 = Conv2DTranspose(start_neurons * 8, (3, 3), strides=(2, 2), padding="same")(de
conv3_up1)
    conv3 = backbone.layers[263].output
    print(conv3.shape)
    uconv3 = concatenate([deconv3,deconv4_up1, conv3])
    uconv3 = Dropout(dropout_rate)(uconv3)

    uconv3 = Conv2D(start_neurons * 8, (3, 3), activation=None, padding="same")(uconv3)
    uconv3 = residual_block(uconv3,start_neurons * 8)
#     uconv3 = residual_block(uconv3,start_neurons * 8)
    uconv3 = LeakyReLU(alpha=0.1)(uconv3)

    deconv2 = Conv2DTranspose(start_neurons * 4, (3, 3), strides=(2, 2), padding="same")(uconv3
)
    deconv2_up1 = Conv2DTranspose(start_neurons * 4, (3, 3), strides=(2, 2), padding="same")(de
conv2)
    conv2 = backbone.layers[159].output
    print(conv2.shape)
    uconv2 = concatenate([deconv2,deconv3_up1,deconv4_up2, conv2])

    uconv2 = Dropout(0.1)(uconv2)
```

```
    uconv2 = Conv2D(start_neurons * 4, (3, 3), activation=None, padding="same")(uconv2)
    uconv2 = residual_block(uconv2,start_neurons * 4)
#     uconv2 = residual_block(uconv2,start_neurons * 4)
    uconv2 = LeakyReLU(alpha=0.1)(uconv2)

    deconv1 = Conv2DTranspose(start_neurons * 2, (3, 3), strides=(2, 2), padding="same")(uconv2
)
    conv1 = backbone.layers[55].output
    print(conv1.shape)
    uconv1 = concatenate([deconv1,deconv2_up1,deconv3_up2,deconv4_up3, conv1])

    uconv1 = Dropout(0.1)(uconv1)
    uconv1 = Conv2D(start_neurons * 2, (3, 3), activation=None, padding="same")(uconv1)
    uconv1 = residual_block(uconv1,start_neurons * 2)
#     uconv1 = residual_block(uconv1,start_neurons * 2)
    uconv1 = LeakyReLU(alpha=0.1)(uconv1)

    uconv0 = Conv2DTranspose(start_neurons * 1, (3, 3), strides=(2, 2), padding="same")(uconv1)
    uconv0 = Dropout(0.1)(uconv0)
    uconv0 = Conv2D(start_neurons * 1, (3, 3), activation=None, padding="same")(uconv0)
    uconv0 = residual_block(uconv0,start_neurons * 1)
#     uconv0 = residual_block(uconv0,start_neurons * 1)
    uconv0 = LeakyReLU(alpha=0.1)(uconv0)

    uconv0 = Dropout(dropout_rate/2)(uconv0)
    output_layer = Conv2D(1, (1,1), padding="same", activation="sigmoid")(uconv0)

    model = Model(input, output_layer)
    model._name = 'u-xception'

    return model
```

```
import pandas as pd
from keras.utils.vis_utils import plot_model
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, EarlyStopping, ReduceL
ROnPlateau
from keras import backend as K

import matplotlib.pyplot as plt
import gc

IMG_WIDTH = 256
IMG_HEIGHT = 256
IMG_CHANNELS = 3

input_shape = (IMG_WIDTH, IMG_HEIGHT, IMG_CHANNELS)

#================================Start kfold validations and model build+train+evaluation
s==========
num_train_samples = 88
train_batch_size = 8
train_steps = np.ceil(num_train_samples / train_batch_size)

num_val_samples = 22
val_batch_size = 4
val_steps = np.ceil(num_val_samples / val_batch_size)

#============================================DATA AUGMENTATION=========================
==============
from keras.preprocessing.image import ImageDataGenerator
from IPython.display import clear_output

dg_args = dict(    rescale=1.,
```

```
                  rotation_range = 10,
                  width_shift_range = 0.1,
                  height_shift_range = 0.1,
                  shear_range = 0.01,
                  zoom_range = [0.8, 1.2],
                  fill_mode = 'nearest',
                  data_format = 'channels_last')

image_gen = ImageDataGenerator(**dg_args)

def gen_augmented_pairs(in_cxr, in_seg, batch_size):
    while True:
        seed = np.random.choice(range(9999))
        # keep the seeds syncronized otherwise the augmentation to the images is different
from the masks
        g_cxr = image_gen.flow(in_cxr, batch_size = batch_size, seed = seed)
        g_seg = image_gen.flow(in_seg, batch_size = batch_size, seed = seed)
        for i_cxr, i_seg in zip(g_cxr, g_seg):
            yield i_cxr, i_seg

def get_model_name(k):
    return 'UNET_EFFNETB7_'+str(k)+'.h5'

VALIDATION_ACCURACY = []
VALIDAITON_LOSS = []

#save_dir = '/working/'
num_folds=5
kf = KFold(n_splits=num_folds, shuffle=True)
fold_var = 0
Y = np.arange(0,CXR_train.shape[0])
countfold=0

for train_index, val_index in kf.split(Y):
    countfold += 1
    print("Running FOLD number ", countfold)

    train_gen = gen_augmented_pairs(CXR_train[train_index], seg_train[train_index], batch_
size = train_batch_size)
    image_gen = ImageDataGenerator(rescale=1.)
    valid_gen = gen_augmented_pairs(CXR_train[val_index], seg_train[val_index], batch_size
= val_batch_size)

    #Build Model
    K.clear_session()
    model = UEfficientNet(input_shape=input_shape,dropout_rate=0.0)
    # Build model
    opt = Adam(learning_rate=0.0001)    #0.0001 with 150 epochs with dropout 0.0 is curren
tly best working
    model.compile(loss=dice_loss, optimizer=opt, metrics=[dice_coef])

    # CREATE CALLBACKS
    checkpoint = ModelCheckpoint(get_model_name(fold_var), monitor='val_loss', verbose=1,
                                 save_best_only=True, mode='min', save_weights_only = True)

    reduceLROnPlat = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
                                       patience=50,
                                       verbose=1, mode='min', epsilon=0.0001, cooldown=2, min_
lr=1e-6)
    early = EarlyStopping(monitor="val_loss",
                          mode="min",
                          patience=50)
```

```
    callbacks_list = [checkpoint, early, reduceLROnPlat]

    # FIT THE MODEL
    history = model.fit_generator(train_gen,
                  steps_per_epoch = train_steps,
                  epochs = 250, verbose=1,
                  validation_data = valid_gen,
                  validation_steps = val_steps,
                  callbacks = callbacks_list
                 )

    # Plot train and validation iou metric curves

    plt.figure(figsize=(16,4))
    plt.subplot(1,2,1)
    plt.plot(history.history['dice_coef'][1:])
    plt.plot(history.history['val_dice_coef'][1:])
    plt.ylabel('dice coef')
    plt.xlabel('epoch')
    plt.legend(['train','Validation'], loc='upper left')

    plt.title('model dice coef')

    plt.subplot(1,2,2)
    plt.plot(history.history['loss'][1:])
    plt.plot(history.history['val_loss'][1:])
    plt.ylabel('val_loss')
    plt.xlabel('epoch')
    plt.legend(['train','Validation'], loc='upper left')
    plt.title('model loss')
    gc.collect()

    del model
    fold_var += 1
```

```
del model

K.clear_session()

input_shape = (IMG_WIDTH, IMG_HEIGHT, IMG_CHANNELS)
model = UEfficientNet(input_shape=input_shape,dropout_rate=0.0)
# Build model
opt = Adam(learning_rate=0.0001)
model.compile(loss=dice_loss, optimizer=opt, metrics=[dice_coef])

model.load_weights('./UNET_EFFNETB7.h5')
```

```
def Dice(y_true, y_pred):
    """Returns Dice Similarity Coefficient for ground truth and predicted masks."""
    assert y_true.dtype == bool and y_pred.dtype == bool
    y_true_f = y_true.flatten()
    y_pred_f = y_pred.flatten()
    intersection = np.logical_and(y_true_f, y_pred_f).sum()
    return (2. * intersection + 1.) / (y_true.sum() + y_pred.sum() + 1.)

def masked(img, gt, mask, alpha=1):
    """Returns image with GT lung field outlined with red, predicted lung field
    filled with blue."""
    rows, cols = img.shape
    color_mask = np.zeros((rows, cols, 3))
    #convert to numpy float and then subtract
```

```
    #boundary = morphology.dilation(gt, morphology.disk(3)) - gt
    a1 = morphology.dilation(gt, morphology.disk(3))
    a2 = gt
    a1 = a1.astype(np.float32)
    a2 = a2.astype(np.float32)
    boundary = a1 - a2

    color_mask[mask == 1] = [1, 0, 0]          #predicted mask - Red color area
    color_mask[boundary == 1] = [0, 0, 1]      # Actual mask - Ground truth - Blue curve
    img_color = np.dstack((img, img, img))

    img_hsv = color.rgb2hsv(img_color)
    color_mask_hsv = color.rgb2hsv(color_mask)

    img_hsv[..., 0] = color_mask_hsv[..., 0]
    img_hsv[..., 1] = color_mask_hsv[..., 1] * alpha

    img_masked = color.hsv2rgb(img_hsv)
    return img_masked

def remove_small_regions(img, size):
    """Morphologically removes small (less than size) connected regions of 0s or 1s."""
    img = morphology.remove_small_objects(img, size)
    img = morphology.remove_small_holes(img, size)
    return img
```

```
from skimage import morphology, color, io, exposure
import cv2

n_test = CXR_test.shape[0]
dices = np.zeros(n_test)
im_shape = (256, 256)

#del model
K.clear_session()
input_shape = (IMG_WIDTH, IMG_HEIGHT, IMG_CHANNELS)
model = UEfficientNet(input_shape=input_shape,dropout_rate=0.0)
# Build model
opt = Adam(learning_rate=0.0001)
model.compile(loss=dice_loss, optimizer=opt, metrics=[dice_coef])

i=0
count=0
for xx in range(len(CXR_test)):
    count=count+1
    img = CXR_test[xx]
    img = img[np.newaxis,...]

    model.load_weights('./UNET_EFFNETB7_0.h5')
    pred0 = model.predict(img)[..., 0].reshape(input_shape[:2])
    model.load_weights('./UNET_EFFNETB7_1.h5')
    pred1 = model.predict(img)[..., 0].reshape(input_shape[:2])
    model.load_weights('./UNET_EFFNETB7_2.h5')
    pred2 = model.predict(img)[..., 0].reshape(input_shape[:2])
    model.load_weights('./UNET_EFFNETB7_3.h5')
    pred3 = model.predict(img)[..., 0].reshape(input_shape[:2])
    model.load_weights('./UNET_EFFNETB7_4.h5')
    pred4 = model.predict(img)[..., 0].reshape(input_shape[:2])

    pred = (pred0 + pred1 + pred2 + pred3 + pred4)/5.0

    img = cv2.cvtColor(CXR_test[xx], cv2.COLOR_BGR2GRAY)
```

```
    yy = CXR_Mask_test[xx]
    mask = yy[..., 0].reshape(input_shape[:2])

    # Binarize masks
    gt = mask > 0.5
    pr = pred > 0.5

    # Remove regions smaller than 2% of the image
    pr = remove_small_regions(pr, 0.02 * np.prod(im_shape))

    #io.imsave('./'+str(count), masked(img, gt, pr, 1))
    export_path = '../working/'
    image_filename = str(count) + str('.png')
    io.imsave(os.path.join(export_path, image_filename), masked(img, gt, pr, 1))

    dices[i] = Dice(gt, pr)
    print(str(count), dices[i])

    i += 1
    if i == n_test:
        break

print('Mean Dice:', dices.mean())
```