```
In [ ]:  from google.colab import drive
         drive.mount("/content/gdrive", force_remount=True)
```

```
In [ ]:  %cd /content/gdrive/MyDrive/chestct
```

```
In [ ]:  import os
         os.environ['KAGGLE_CONFIG_DIR'] = "/content/gdrive/MyDrive/chestct"
```

```
In [ ]:  !kaggle datasets download -d nikhilpandey360/chest-xray-masks-and-label
```

```
Downloading chest-xray-masks-and-labels.zip to /content/gdrive/MyDriv
e/chestct
100% 9.57G/9.58G [01:25<00:00, 136MB/s]
100% 9.58G/9.58G [01:25<00:00, 120MB/s]
```

```
In [ ]:  !unzip /content/gdrive/MyDrive/chestct/chest-xray-masks-and-labels.zip
```

```
In [ ]:  !pip install -q mlflow
```

import numpy as np import os import cv2 from tqdm import tqdm import matplotlib.pyplot as plt import random from sklearn.model_selection import train_test_split %matplotlib inline

import warnings warnings.filterwarnings("ignore")

Load Dataset

```
In [ ]:  image_path_train = '/content/gdrive/MyDrive/chestct/Lung Segmentation/C
         mask_path_train = '/content/gdrive/MyDrive/chestct/Lung Segmentation/ma
         image_path_test = '/content/gdrive/MyDrive/chestct/Lung Segmentation/te
```

In [ ]:
```
# Get the list of file names in the training image directory
images = os.listdir(image_path_train)
# Get the list of file names in the training mask directory
mask = os.listdir(mask_path_train)
# Process mask file names: Extract base names by removing ".png" extensi
mask = [fName.split(".png")[0] for fName in mask]
# Process image file names: Extract base names by removing "_mask" suffi
image_file_name = [fName.split("_mask")[0] for fName in mask]
```

The code checks for masks with a modified name, specifically those containing the substring "mask" in their file names. The count of such masks is then printed.

In [ ]:
```
# Check for masks that have a modified name
check = [i for i in mask if "mask" in i]
# Print the total number of masks with a modified name
print("Total mask that has modified name:", len(check))
```

```
Total mask that has modified name: 566
```

Testing_files will contain the set of files that are common between the image and mask directories, representing files available for testing. training_files will contain the list of masks with modified names, which will be used for training.

In [ ]:
```
# Create a set of common files between the image and mask directories fo
testing_files = set(os.listdir(image_path_train)) & set(os.listdir(mask_
# Use the list of masks with modified names as training files
training_files = check
```

Images and masks are read and preprocessed based on the specified file naming conventions.

```
In [ ]:  def getData(X_shape, flag = "MONT"):
             """
             Get image and mask data for model training or testing.

             Parameters:
             - X_shape (int): Desired shape for the images.
             - flag (str): Flag to specify whether to get data for 'MONT' (testir

             Returns:
             - im_array (list): List of preprocessed images.
             - mask_array (list): List of corresponding masks.
             """
             im_array = []
             mask_array = []
             shape = (X_shape, X_shape)

             if flag == "MONT":

                 # Iterate through files in the testing set
                 for i in tqdm(testing_files):

                     # Read and preprocess image
                     im = cv2.imread(os.path.join(image_path_train, i), cv2.IMRE
                     im = cv2.resize(im, shape)
                     im = cv2.equalizeHist(im)

                     # Read and preprocess mask
                     mask = cv2.imread(os.path.join(mask_path_train, i), cv2.IMRI
                     mask = cv2.resize(mask, shape)

                     # Append to respective lists
                     im_array.append(im)
                     mask_array.append(mask)

             if flag == "SHEN":
                 # Iterate through files in the training set
                 for i in tqdm(training_files):

                     # Read and preprocess image
                     im = cv2.imread(os.path.join(image_path_train, i.split("_ma
                     im = cv2.resize(im, shape)
                     im = cv2.equalizeHist(im)

                     # Read and preprocess mask
                     mask = cv2.imread(os.path.join(mask_path_train, i + ".png")
                     mask = cv2.resize(mask, shape)

                     # Append to respective lists
                     im_array.append(im)
                     mask_array.append(mask)

             # Return lists of preprocessed images and masks
```

```
    return im_array, mask_array
```

In [ ]:
```
def get_test(X_shape, n_samples = 100):
    """
    Get test images for model evaluation.

    Parameters:
    - X_shape (int): Desired shape for the test images.
    - n_samples (int): Number of test samples to retrieve.

    Returns:
    - im_array (list): List of preprocessed test images.

    Note: The function randomly selects test samples from the specified
    """
    im_array = []
    shape = (X_shape, X_shape)

    # Randomly select test files
    test_files = random.choices(list(os.listdir(image_path_test)), k=n_

    # Iterate through selected test files
    for i in tqdm(test_files):
        im = cv2.imread(os.path.join(image_path_test, i), cv2.IMREAD_GR
        im = cv2.resize(im, shape)
        im = cv2.equalizeHist(im)
        im_array.append(im)

     # Return the list of preprocessed test images
    return im_array
```

Loading images and masks

This code part of the data preparation process for training and evaluating a model. The training data for both SHEN and MONT datasets are obtained, and a set of preprocessed test images is retrieved for model evaluation. The getData function is assumed to be a custom function that retrieves training data, and get_test is a function to obtain test data.

In [ ]:
```
# Setting the dimension and number of samples
dim, n_samples = 256, 50

# Retrieving training data for SHEN dataset
image_shen, mask_shen = getData(dim, flag = "SHEN")

# Retrieving training data for MONT dataset
image_mont, mask_mont = getData(dim, flag = "MONT")

# Retrieving test data
X_test = get_test(dim, n_samples = n_samples)
```

```
100%|██████████| 566/566 [12:14<00:00,  1.30s/it]
100%|██████████| 138/138 [03:06<00:00,  1.35s/it]
100%|██████████| 50/50 [01:11<00:00,  1.42s/it]
```

This code prepares the training and test datasets by reshaping them to the required format for further processing, ensuring they align with the input shape expected by the model. The verification print statements confirm the shapes of the reshaped arrays.

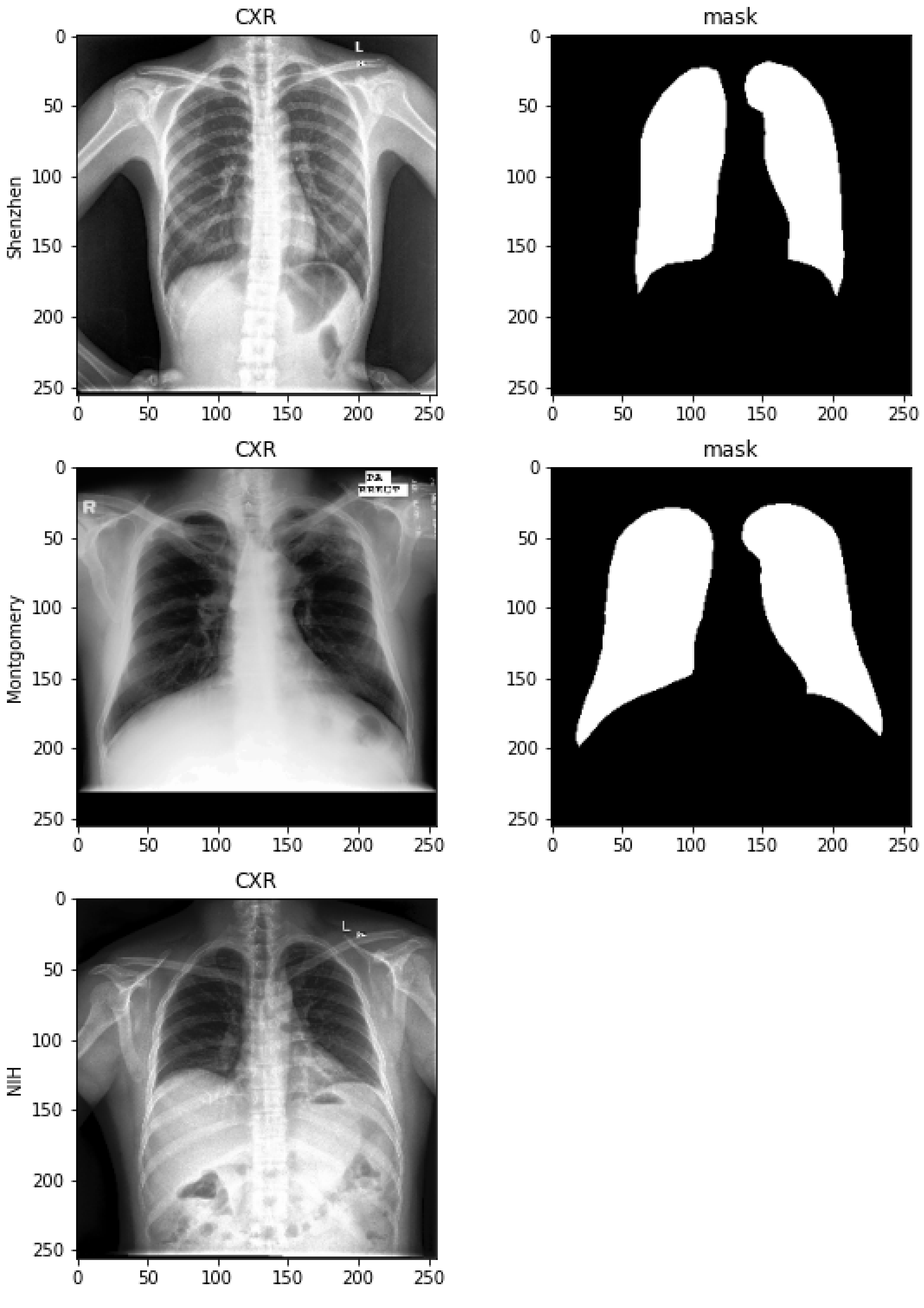
In [ ]:
```
# Reshaping the training and test data arrays to conform to the input sh

# Reshape training data for the SHEN dataset
image_shen = np.array(image_shen).reshape(len(image_shen), dim, dim, 1)
mask_shen = np.array(mask_shen).reshape(len(mask_shen), dim, dim, 1)

# Reshape training data for the MONT dataset
image_mont = np.array(image_mont).reshape(len(image_mont), dim, dim, 1)
mask_mont = np.array(mask_mont).reshape(len(mask_mont), dim, dim, 1)

# Reshape test data
X_test = np.array(X_test).reshape(len(X_test), dim, dim, 1)

print(image_shen.shape, mask_shen.shape)
print(image_mont.shape, mask_mont.shape)
print(X_test.shape)
```

```
(566, 256, 256, 1) (566, 256, 256, 1)
(138, 256, 256, 1) (138, 256, 256, 1)
(50, 256, 256, 1)
```

This code provides a visual representation of sample images and masks from different datasets using matplotlib subplots.

In [ ]:

```
# Visualizing sample images and masks from the datasets using matplotlib

i = 20
fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(9, 13))
axs[0, 0].imshow(image_shen[i].reshape(256, 256), cmap='gray')
axs[0, 1].imshow(mask_shen[i].reshape(256, 256), cmap='gray')
axs[0, 0].set_ylabel('Shenzhen')

axs[1, 0].imshow(image_mont[i].reshape(256, 256), cmap='gray')
axs[1, 1].imshow(mask_mont[i].reshape(256, 256), cmap='gray')
axs[1, 0].set_ylabel('Montgomery')

axs[2, 0].imshow(X_test[i].reshape(256, 256), cmap='gray')
axs[2, 0].set_ylabel('NIH')

axs[0, 0].set_title('CXR')
axs[1, 0].set_title('CXR')
axs[2, 0].set_title('CXR')

axs[0, 1].set_title('mask')
axs[1, 1].set_title('mask')

fig.delaxes(axs[2, 1])
```



The assertions are used to check whether the shapes of image arrays match their corresponding mask arrays for SHEN and MONT datasets. If the shapes do not match, an AssertionError will be raised, providing a safety check for data consistency.The 'np.concatenate' function is then used to combine the image and mask arrays along the first axis (axis=0), effectively stacking them vertically. The resulting 'images' and 'masks' arrays represent the combined dataset from the SHEN and MONT datasets.Finally, the shapes of the concatenated arrays are printed to confirm the successful concatenation.

In [ ]:
```python
# Checking the shapes of image and mask arrays and concatenating them

assert image_shen.shape == mask_shen.shape
assert image_mont.shape == mask_mont.shape
images = np.concatenate((image_shen, image_mont), axis=0)
masks  = np.concatenate((mask_shen, mask_mont), axis=0)

print(images.shape, masks.shape)
```

```
(704, 256, 256, 1) (704, 256, 256, 1)
```

Data Augmetation

The apply_brightness_contrast function adjusts the brightness and contrast of an input image based on the specified levels. The create_contrast_images_v1 function applies contrast adjustments to a list of input images, generating a new list of images with varying contrast levels. These functions can be used as part of data augmentation to create a more diverse dataset for training machine learning models. Adjusting brightness and contrast can help the model generalize better to different lighting conditions

```
In [ ]:  # Data Augmentation: Brightness and Contrast Adjustment


         # Function to apply brightness and contrast adjustments to an input imag
         def apply_brightness_contrast(input_img, brightness = 0, contrast = 0):

             """
             Apply brightness and contrast adjustments to an input image.

             Parameters:
                 input_img (numpy.ndarray): Input image.
                 brightness (int): Brightness adjustment level.
                 contrast (int): Contrast adjustment level.

             Returns:
                 numpy.ndarray: Processed image with brightness and contrast adju
             """
             if brightness != 0:
                 if brightness > 0:
                     shadow = brightness
                     highlight = 255
                 else:
                     shadow = 0
                     highlight = 255 + brightness
                 alpha_b = (highlight - shadow) / 255.0
                 gamma_b = shadow

                 buf = cv2.addWeighted(input_img, alpha_b, input_img, 0, gamma_b
             else:
                 buf = input_img.copy()

             if contrast != 0:
                 f = 131 * (contrast + 127) / (127 * (131 - contrast))
                 alpha_c = f
                 gamma_c = 127 * (1 - f)

                 buf = cv2.addWeighted(buf, alpha_c, buf, 0, gamma_c)

             return buf

         # Function to create a list of images with varying contrast levels
         def create_contrast_images_v1(b, c):
             """
             Create a list of images with varying contrast levels.

             Parameters:
                 b (int): Brightness adjustment level.
                 c (int): Contrast adjustment level.

             Returns:
                 List[numpy.ndarray]: List of images with applied contrast adjust
             """
             contrast_images = []
             for i in tqdm(range(len(images)), "contrast_images"):
                 contrast_images.append(apply_brightness_contrast(images[i], bri
    return contrast_images
```

This section is useful for visually inspecting the effects of contrast adjustments on the images and checking that the resulting dataset has the expected shape for training machine learning models.

```
In [ ]:
# Applying Contrast Adjustment to Images with Specified Brightness and
b, c = -40, -120

contrast_images_v1 = create_contrast_images_v1(b, c)
contrast_images_v1 = np.array(contrast_images_v1).reshape(len(contrast_
print(f'\nshape = {contrast_images_v1.shape}')
```

```
contrast_images: 100%|██████████| 704/704 [00:00<00:00, 6635.93it/s]


shape = (704, 256, 256, 1)
```

This code is an alternative method for applying contrast adjustments to the images using OpenCV's addWeighted function, providing flexibility in adjusting the contrast levels

```
In [ ]:
# Function to Create Contrast-Adjusted Images Using OpenCV addWeighted

def create_contrast_images_v2(alpha, beta):
    """
    Create contrast-adjusted images using the OpenCV addWeighted functio

    Parameters:
    - alpha: The weight of the original image
    - beta: The weight of the contrast-adjusted image

    Returns:
    - contrast_images_v2: List of contrast-adjusted images
    """
    contrast_images_v2 = []
    for i in tqdm(range(len(images)), "contrast_images"):
        contrast_images_v2.append(cv2.addWeighted(images[i], alpha, imag
    return contrast_images_v2
```

This code segment demonstrates how to set contrast adjustment parameters, apply contrast adjustments to images, and inspect the shape of the resulting array containing the contrast-adjusted images.

```
In [ ]:
alpha = 1.5 #@alpha
beta = 0.7 #@beta

contrast_images_v2 = create_contrast_images_v2(alpha, beta)
contrast_images_v2 = np.array(contrast_images_v2).reshape(len(contrast_
print(f'\nshape = {contrast_images_v2.shape}')
```

```
contrast_images: 100%|██████████| 704/704 [00:00<00:00, 14611.26it/s]


shape = (704, 256, 256, 1)
```

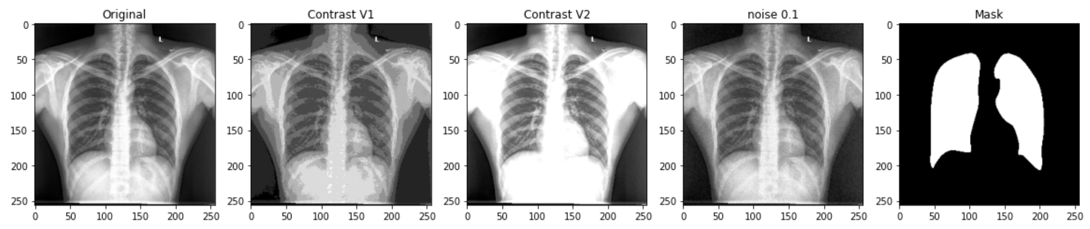
This code segment demonstrates how to add random noise to images in the dataset and inspect the shape of the resulting array.

```
In [ ]:
# A Function to Generate Random Noise
def noise(i: int = len(images)):
    return np.random.randint(0, 255, size=(i, 256, 256, 1))

# A Function to Add Noise to Images
def noise_images(epsilon: float = 0.1):
    noised = noise()
    noised_img = []
    for i in tqdm(range(len(images)), "noise_images"):
        noised_img.append(noised[i] * epsilon + images[i])

    return noised_img

# Created Noised Images Using the noise_images Function
noised_images = noise_images(epsilon=0.1)

# Reshape the Noised Images
noised_images = np.array(noised_images).reshape(len(noised_images), 256
print(f'\nshape = {noised_images.shape}')
```

```
noise_images: 100%|██████████| 704/704 [00:00<00:00, 2326.65it/s]


shape = (704, 256, 256, 1)
```

This code segment visualizes a set of images for comparison, displaying the original image, images with contrast adjustments (V1 and V2), a noised image, and the corresponding mask.This visualization helps compare different image transformations and their impact on the original image, including contrast adjustments and the addition of noise.

In [ ]:
```python
# Visualize Images for Comparison

i = 15
fig, (ax1, ax2, ax3, ax4, ax5) = plt.subplots(1, 5, figsize=(20, 10))

ax1.imshow(images[i].reshape(256, 256), cmap='gray')
ax1.set_title('Original')

ax2.imshow(contrast_images_v1[i].reshape(256, 256), cmap='gray')
ax2.set_title('Contrast V1')

ax3.imshow(contrast_images_v2[i].reshape(256, 256), cmap='gray')
ax3.set_title('Contrast V2')

ax4.imshow(noised_images[i].reshape(256, 256), cmap='gray')
ax4.set_title('noise 0.1')

ax5.imshow(masks[i].reshape(256, 256), cmap='gray')
ax5.set_title('Mask');
```



This preparation is crucial for training a machine learning model on a more extensive and varied dataset, which may enhance the model's ability to generalize to different types of input images.

In [ ]:
```python
# Concatenate Images and Masks
all_images = np.concatenate((images, contrast_images_v1, contrast_image
all_masks  = np.concatenate((masks, masks, masks, masks), axis=0)

all_images.shape, all_masks.shape
```

Out[22]:
```
((2816, 256, 256, 1), (2816, 256, 256, 1))
```

This preprocessing is essential for preparing the data for a machine learning model, ensuring that the pixel values are within a suitable range for training and evaluation.

In [ ]:
```
X_train, X_val, Y_train, Y_val = train_test_split((all_images - 127.0) 
                                                  (all_masks > 127).ast
                                                  test_size = 0.2,
                                                  random_state = 2018)
X_testNorm = (X_test - 127.0) / 127.0
```

Residual U-Net

[https://www.sciencedirect.com/science/article/abs/pii/S0924271620300149 (https://www.sciencedirect.com/science/article/abs/pii/S0924271620300149)](https://www.sciencedirect.com/science/article/abs/pii/S0924271620300149)

A Residual U-Net is a hybrid neural network architecture that combines the strengths of Residual Networks (ResNets) and U-Net architectures, designed particularly for image segmentation tasks. The key idea of residual learning is, it contains shortcut connections that allow the network to learn residual mappings, making it easier to train deep networks. In a Residual U-Net, residual connections are strategically added to both the encoder and decoder paths of the U-Net (which is Similar to Res-UNet). These connections facilitate the direct flow of information between layers, allowing the model to learn residuals, or differences, between input and output more effectively. ResUNet has an encoder-decoder structure with skip connections. The difference lies in the use of residual blocks in the encoder.The final layer typically uses a sigmoid or softmax activation function to produce pixel-wise predictions. Implementation involves adding residual blocks to the encoding and decoding paths, typically consisting of convolutional layers with skip connections. This architecture enables the training of deeper networks while maintaining efficient information flow, leading to improved segmentation performance, even The use of residual blocks helps mitigate the vanishing gradient problem, enabling the training of deeper networks(especially in medical imaging scenarios).

The "Chest-X-Ray-Image_Segmentation_ResUNet" project aims to perform lung segmentation in chest X-ray images using a Residual U-Net architecture. The overarching goal involves not only achieving accurate semantic segmentation of lung regions but also incorporating feature extraction and tuberculosis case diagnosis.Residual U-Net architecture is specifically tailored for the complexities of medical image segmentation, providing a robust and effective framework for accurately identifying and segmenting lung regions in chest X-ray images.

The model's significance lies in its ability to extract intricate features effectively, addressing challenges associated with deep networks such as the vanishing gradient problem. By incorporating residual connections inspired by ResNets, the model ensures efficient information flow, enabling the extraction of both local and global features. The Res-UNet architecture contributes to precise localization, especially crucial for medical image segmentation tasks. The skip connections in the decoder path facilitate the fusion of high-level features and fine-grained details, enhancing the reconstruction of detailed structures in the segmented lung regions. With a final layer using the sigmoid activation function for binary segmentation, the model is well-suited for identifying whether each pixel belongs to the lung region. Additionally, the inclusion of batch normalization and activation functions

In [ ]:

```python
import tensorflow as tf
import tensorflow.keras
from tensorflow.keras.models import *
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import *
from tensorflow.keras import backend as K
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateSche
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateSche
from tensorflow.keras.callbacks import ModelCheckpoint
import mlflow
import mlflow.tensorflow
```

These metrics are crucial for assessing the accuracy and performance of the Residual U-Net model in lung segmentation. The goal is to maximize these coefficients, indicating a high degree of overlap and similarity between the predicted and true lung segmentations. The loss functions, inversely proportional to the coefficients, guide the training process to improve the model's segmentation accuracy.

In [ ]:

```python
#Calculate the Dice coefficient.

def dice_coef(y_true, y_pred):
    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 + 1) / (K.sum(y_true_f) + K.sum(y_pred_f)

#Calculate the Dice coefficient loss.
def dice_coef_loss(y_true, y_pred):
    return 1 - dice_coef(y_true, y_pred)

#Calculate the Jaccard coefficient.
def jaccard_coef(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (intersection + 1.0) / (K.sum(y_true_f) + K.sum(y_pred_f) -

#Calculate the Jaccard coefficient.
def jaccard_coef_loss(y_true, y_pred):
    return 1 - jaccard_coef(y_true, y_pred)
```

These functions define the building blocks used in the Residual U-Net architecture. The stem block, residual block, and upsample concatenation block are fundamental components for creating a U-Net with residual connections, facilitating effective feature learning and information flow.

```
In [ ]:  #Batch normalization followed by an optional ReLU activation.
         def bn_act(x, act=True):
             x = tensorflow.keras.layers.BatchNormalization()(x)
             if act == True:
                 x = tensorflow.keras.layers.Activation("relu")(x)
             return x

         #Convolutional block: Batch normalization, ReLU activation, and Conv2D l
         def conv_block(x, filters, kernel_size=(3, 3), padding="same", strides=1
             conv = bn_act(x)
             conv = tensorflow.keras.layers.Conv2D(filters, kernel_size, padding=
             return conv

         #Stem block: Initial convolutional block with a residual connection.
         def stem(x, filters, kernel_size=(3, 3), padding="same", strides=1):
             conv = tensorflow.keras.layers.Conv2D(filters, kernel_size, padding=
             conv = conv_block(conv, filters, kernel_size=kernel_size, padding=pa

             shortcut = tensorflow.keras.layers.Conv2D(filters, kernel_size=(1, 1
             shortcut = bn_act(shortcut, act=False)

             output = tensorflow.keras.layers.Add()([conv, shortcut])
             return output

         #Residual block: Two convolutional blocks with a residual connection.
         def residual_block(x, filters, kernel_size=(3, 3), padding="same", stric
             res = conv_block(x, filters, kernel_size=kernel_size, padding=paddir
             res = conv_block(res, filters, kernel_size=kernel_size, padding=padc

             shortcut = tensorflow.keras.layers.Conv2D(filters, kernel_size=(1, 1
             shortcut = bn_act(shortcut, act=False)

             output = tensorflow.keras.layers.Add()([shortcut, res])
             return output

         #Up-sampling and concatenation block: Up-sample the input and concatenat

         def upsample_concat_block(x, xskip):
             u = tensorflow.keras.layers.UpSampling2D((2, 2))(x)
             c = tensorflow.keras.layers.Concatenate()([u, xskip])
             return c
```

This architecture is beneficial for image segmentation tasks as it balances capturing detailed features and maintaining spatial information. The residual connections aid in gradient flow during training, contributing to better convergence and performance.

```
In [ ]:  def ResUNet():
             # Number of filters at each level of the network
             f = [16, 32, 64, 128, 256]

             # Input layer for the network with dimensions (dim, dim, 1)
             inputs = tensorflow.keras.layers.Input((dim, dim, 1))

             ## Encoder
             e0 = inputs
             # Initial convolutional block
             e1 = stem(e0, f[0])
             # Residual blocks with downsampling
             e2 = residual_block(e1, f[1], strides=2)
             e3 = residual_block(e2, f[2], strides=2)
             e4 = residual_block(e3, f[3], strides=2)
             e5 = residual_block(e4, f[4], strides=2)

             ## Bridge
             # Convolutional block applied to the deepest encoder level
             b0 = conv_block(e5, f[4], strides=1)
             b1 = conv_block(b0, f[4], strides=1)

             ## Decoder
             # Upsampling and concatenation with skip connections
             u1 = upsample_concat_block(b1, e4)
             d1 = residual_block(u1, f[4])

             u2 = upsample_concat_block(d1, e3)
             d2 = residual_block(u2, f[3])

             u3 = upsample_concat_block(d2, e2)
             d3 = residual_block(u3, f[2])

             u4 = upsample_concat_block(d3, e1)
             d4 = residual_block(u4, f[1])

             ## Output layer
             outputs = tensorflow.keras.layers.Conv2D(1, (1, 1), padding="same",
             # Construct the model
             model = tensorflow.keras.models.Model(inputs, outputs)
             return model
```

These metrics and loss functions are essential for training and evaluating the performance of the semantic segmentation model in this project. They provide insights into how well the model is capturing relevant patterns and making accurate predictions.

In [ ]:
```python
# Define evaluation metrics for the model
metrics = [dice_coef, jaccard_coef,
           'binary_accuracy',
           tf.keras.metrics.Precision(),
           tf.keras.metrics.Recall()]

# Define loss functions for training the model
loss = [dice_coef_loss,
        jaccard_coef_loss,
        'binary_crossentropy']
```

Automatic logging with mlflow.autolog() is a convenient way to track and organize experiments, results, and models. It simplifies the process of recording essential information during model training without explicitly adding manual logging statements. The logged information is then accessible through the MLflow UI, providing a comprehensive view of different runs and their associated metrics and artifacts.

In [ ]:
```python
# Enable automatic logging of metrics, parameters, and models with MLflo
mlflow.autolog()
```

```
2022/05/21 06:16:09 INFO mlflow.tracking.fluent: Autologging successfu
lly enabled for sklearn.
2022/05/21 06:16:09 INFO mlflow.tracking.fluent: Autologging successfu
lly enabled for tensorflow.
```

This code initializes a U-Net model, configures it for training with the Adam optimizer, specifies the loss function and evaluation metrics, and prints a summary of the model architecture.

In [ ]:

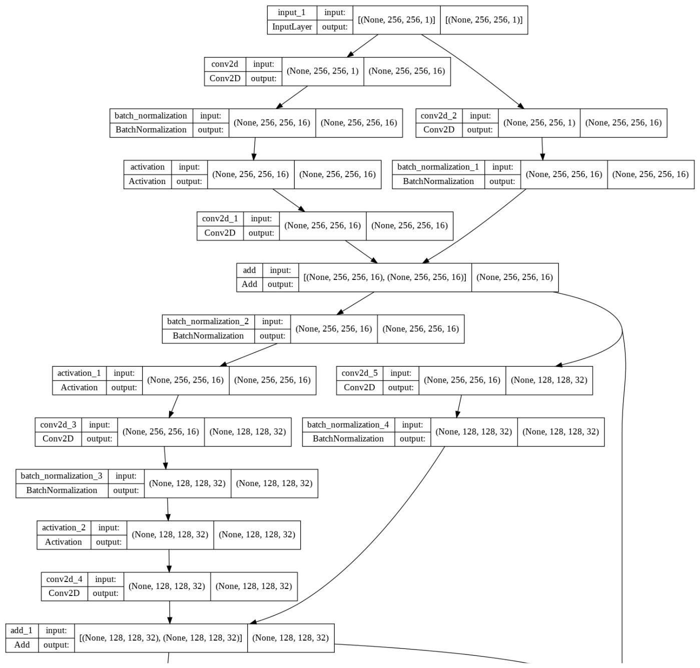
```python
#Creates an instance of the ResUNet model
model = ResUNet()

#An Adam optimizer is instantiated.
adam = tensorflow.keras.optimizers.Adam()

#A method configures the model for training
model.compile(optimizer=adam, loss=loss, metrics=metrics)

#prints the summary
model.summary()
```

```
Model: "model"
_________________________________________________________________________________

__________________________________
 Layer (type)                   Output Shape         Param #
Connected to
=================================================================================
==================================
 input_1 (InputLayer)           [(None, 256, 256, 1  0
[]
                                )]

 conv2d (Conv2D)                (None, 256, 256, 16  160
['input_1[0][0]']
                                )

 batch_normalization (BatchNorm  (None, 256, 256, 16  64
['conv2d[0][0]']
 alization)                     )
```

Generating a visual representation of the model architecture is a helpful practice for understanding the structure of complex neural networks.This visualization is beneficial for model debugging, sharing model details with collaborators, and explaining the model's architecture in presentations or documentation.

```
In [ ]:  from keras.utils.vis_utils import plot_model

         # Generate a visual representation of the model and save it to a file
         plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer
```

Out[31]:



input_1 | InputLayer | input: [(None, 256, 256, 1)] | output: [(None, 256, 256, 1)]
conv2d | Conv2D | input: (None, 256, 256, 1) | output: (None, 256, 256, 16)
batch_normalization | BatchNormalization | input: (None, 256, 256, 16) | output: (None, 256, 256, 16)
conv2d_2 | Conv2D | input: (None, 256, 256, 1) | output: (None, 256, 256, 16)
activation | Activation | input: (None, 256, 256, 16) | output: (None, 256, 256, 16)
batch_normalization_1 | BatchNormalization | input: (None, 256, 256, 16) | output: (None, 256, 256, 16)
conv2d_1 | Conv2D | input: (None, 256, 256, 16) | output: (None, 256, 256, 16)
add | Add | input: [(None, 256, 256, 16), (None, 256, 256, 16)] | output: (None, 256, 256, 16)
batch_normalization_2 | BatchNormalization | input: (None, 256, 256, 16) | output: (None, 256, 256, 16)
activation_1 | Activation | input: (None, 256, 256, 16) | output: (None, 256, 256, 16)
conv2d_5 | Conv2D | input: (None, 256, 256, 16) | output: (None, 128, 128, 32)
conv2d_3 | Conv2D | input: (None, 256, 256, 16) | output: (None, 128, 128, 32)
batch_normalization_4 | BatchNormalization | input: (None, 128, 128, 32) | output: (None, 128, 128, 32)
batch_normalization_3 | BatchNormalization | input: (None, 128, 128, 32) | output: (None, 128, 128, 32)
activation_2 | Activation | input: (None, 128, 128, 32) | output: (None, 128, 128, 32)
conv2d_4 | Conv2D | input: (None, 128, 128, 32) | output: (None, 128, 128, 32)
add_1 | Add | input: [(None, 128, 128, 32), (None, 128, 128, 32)] | output: (None, 128, 128, 32)

Saving the best weights of the model. It uses the format 'BEST_res_unet.h5', where 'BEST' could be a placeholder for any additional information.ReduceLROnPlateau: it monitors the 'val_loss' (validation loss) and reduces the learning rate by a factor of 0.5 if there is no improvement in validation loss for 4 consecutive epochs. Various parameters control its behavior, including patience (number of epochs with no improvement) and min_lr (lower bound on the learning rate).ModelCheckpoint: It saves the model weights to the specified file (weight_path) whenever there is an improvement in the monitored metric ('val_loss'). The save_best_only parameter ensures that only the best weights (minimum validation loss) are saved.

In [ ]:
```
# Define a file name for saving the best weights of the model
weight_path="{}_res_unet.h5".format('BEST')

# Reduce learning rate when a metric has stopped improving
reduceLROnPlat = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
                                   patience=4,
                                   verbose=1, mode='min', epsilon=0.0001

# Model checkpoint callback to save the best weights during training
checkpoint = ModelCheckpoint(weight_path, monitor='val_loss', verbose=1,
                             save_best_only=True, mode='auto')

# List of callbacks to be applied during training
callbacks_list = [checkpoint, reduceLROnPlat]
```

```
WARNING:tensorflow:`epsilon` argument is deprecated and will be remove
d, use `min_delta` instead.
```

This section initializes random seeds for Python's built-in random module, NumPy (np), and TensorFlow (tf). Setting seeds to specific values ensures that the random number generation is reproducible, making it possible to obtain the same results each time the code is run.Then,the model is trained using the fit method. The training data (X_train and Y_train) are used, and the validation data (X_val and Y_val) are used to evaluate the model's performance during training.

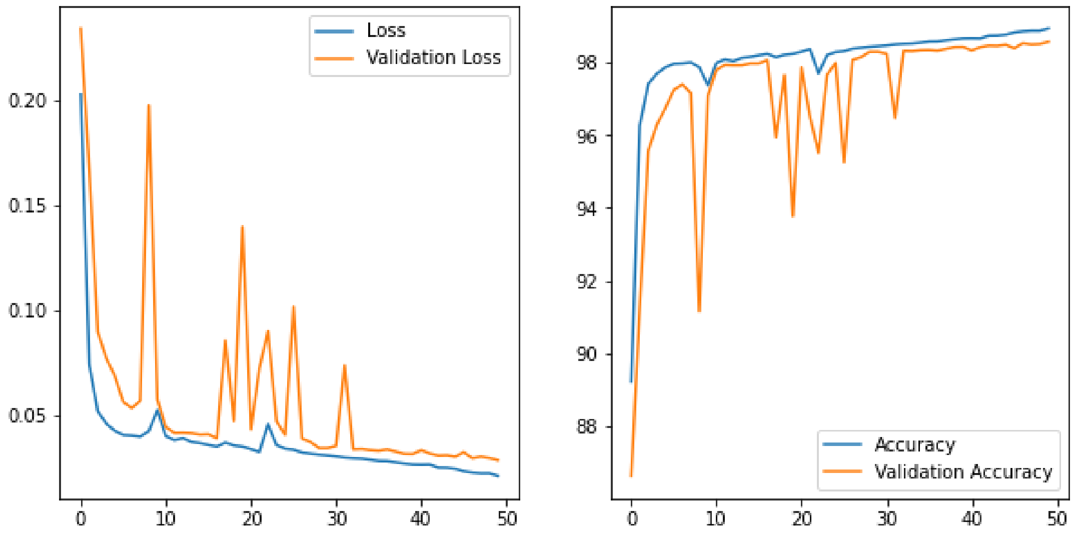
In [ ]:
```
#initialize random seeds so results are repeatable
import random
import tensorflow as tf

random.seed(1)
np.random.seed(1)
tf.random.set_seed(1)
```

In [ ]:
```
res = model.fit(X_train, Y_train,
                validation_data=(X_val, Y_val),
                batch_size=32, epochs=50,
                callbacks=callbacks_list)
```

plot model response

This code is useful for visually inspecting the training and validation trends, helping to identify potential overfitting or underfitting. The loss curves show how well the model is learning, while the accuracy curves indicate the model's performance on both the training and validation sets.

```
In [ ]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 5))
        ax1.plot(res.history['loss'], '-', label = 'Loss')
        ax1.plot(res.history['val_loss'], '-', label = 'Validation Loss')
        ax1.legend()

        ax2.plot(100 * np.array(res.history['binary_accuracy']), '-',
                 label = 'Accuracy')
        ax2.plot(100 * np.array(res.history['val_binary_accuracy']), '-',
                 label = 'Validation Accuracy')
        ax2.legend();
```



Prediction on Validation set

This code is loading the pre-trained Residual U-Net model from the specified file path, and the loaded model is stored in the variable model for further use. The compile=False argument is used to skip the compilation step, assuming that the model was already compiled and saved during training.

```
In [ ]: from keras.models import load_model
        model=load_model(r"/content/gdrive/MyDrive/chestct/BEST_res_unet.h5", c
```

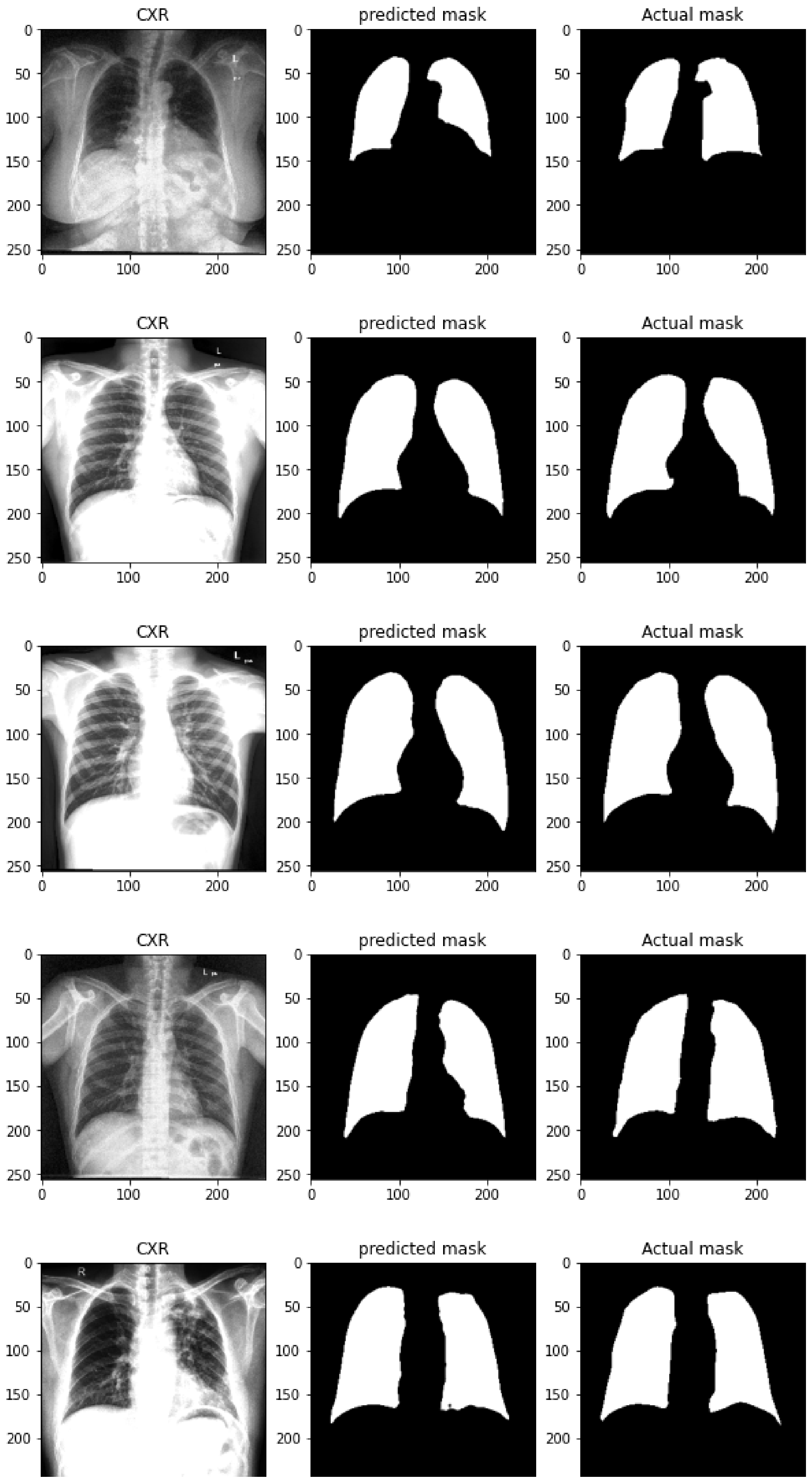
After loading the pre-trained Residual U-Net model, this line of code is using the model to predict the output for the validation dataset, and the predictions are stored in the variable preds_val for further analysis or evaluation.

```
In [ ]: preds_val = model.predict(X_val)
```

This code generates a visual comparison of original chest X-rays, predicted masks, and actual masks for a subset of the validation data in a 5x3 grid of subplots. Each row corresponds to a different sample in the dataset.

In [ ]:

```
fig, axs = plt.subplots(nrows=5, ncols=3, figsize=(10, 20))

for i in range(5):
    for j in range(3):
        if j == 0:
            axs[i, j].imshow(X_val[i + 10].reshape(256, 256), cmap='gray
            axs[i, j].set_title('CXR')
        elif j == 1:
            axs[i, j].imshow(preds_val[i + 10].reshape(256, 256), cmap=
            axs[i, j].set_title('predicted mask')

        elif j == 2:
            axs[i, j].imshow(Y_val[i + 10].reshape(256, 256), cmap='gray
            axs[i, j].set_title('Actual mask')
```


CXR
predicted mask
Actual mask
0
50
100
150
200
250
0
100
200

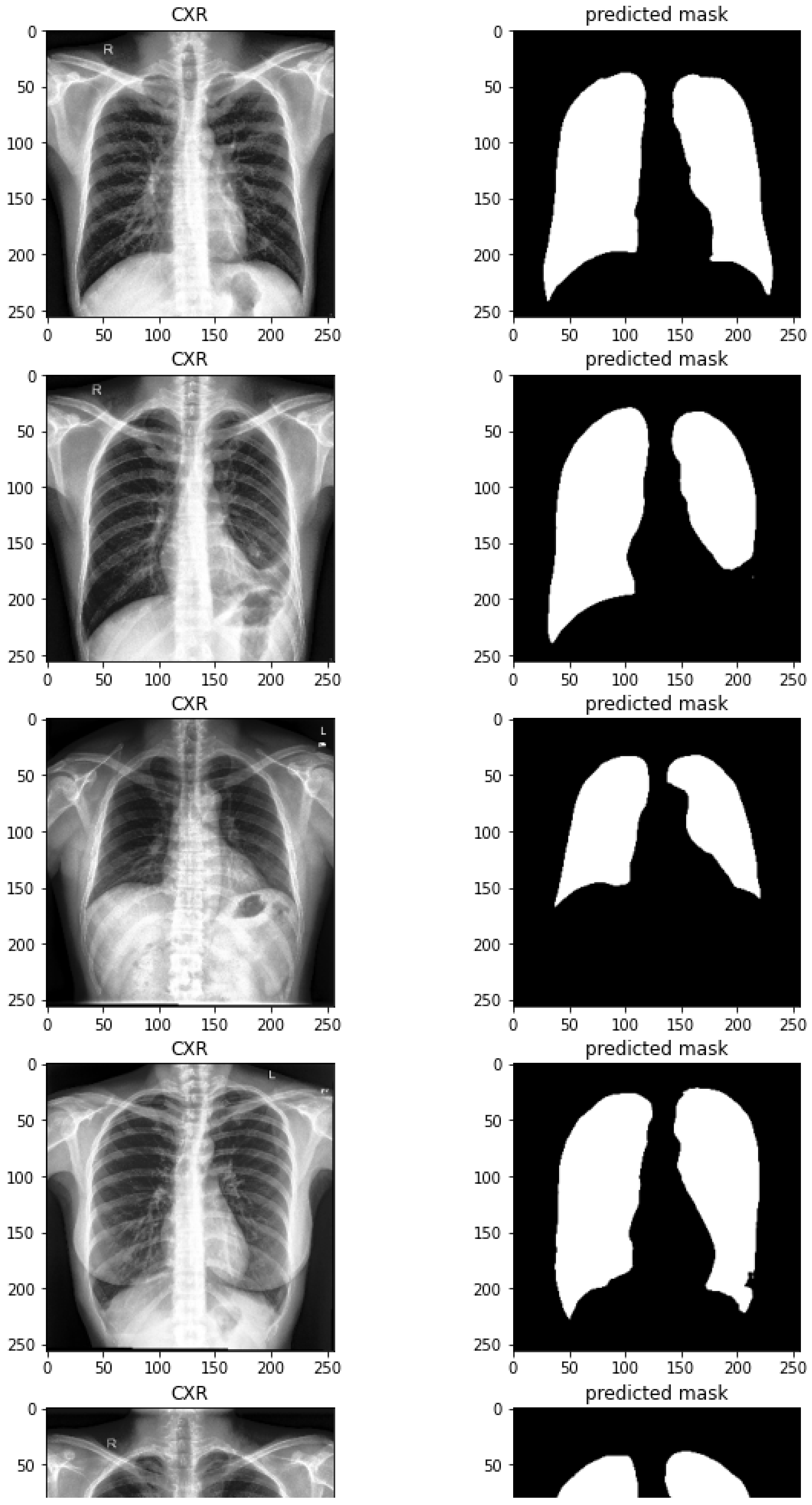
In [ ]:

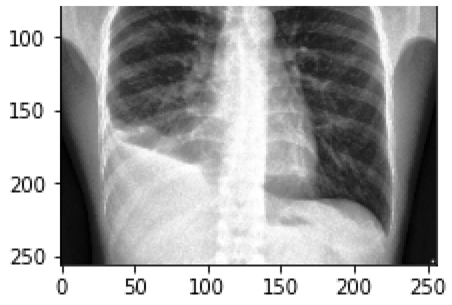
```
preds = model.predict(X_testNorm)
```

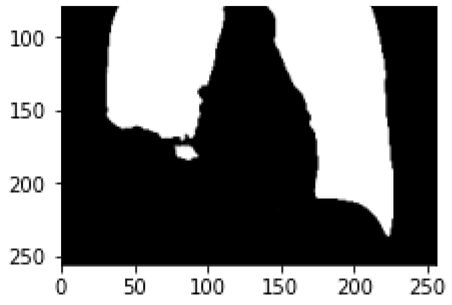
This code creates a visual comparison of original chest X-rays and their corresponding predicted masks for a subset of the test data in a 5x2 grid of subplots. Each row corresponds to a different sample in the test dataset.

In [ ]:

```
fig, axs = plt.subplots(nrows=5, ncols=2, figsize=(10, 20))

for i in range(5):
    for j in range(2):
        if j != 1:
            axs[i, j].imshow(X_testNorm[i + 10].reshape(256, 256), cmap=
            axs[i, j].set_title('CXR')
        else:
            axs[i, j].imshow(preds[i + 10].reshape(256, 256), cmap='gray
            axs[i, j].set_title('predicted mask')
```


CXR
predicted mask
CXR
predicted mask
CXR
predicted mask
L
CXR
predicted mask
L
CXR
predicted mask
R





```
In [ ]:
```

Conclusion:

The project demonstrates a comprehensive approach to medical image segmentation using a deep learning architecture tailored to the characteristics of chest X-ray data.The use of appropriate evaluation metrics ensures a meaningful assessment of model performance.Further insights into the project's success would depend on additional information, such as the diversity and size of the dataset, generalization to unseen data, and clinical validation.Overall, the project appears well-structured, addressing key aspects of medical image segmentation. The true effectiveness of the model would depend on rigorous evaluation, potentially involving collaboration with medical professionals and real-world clinical validation.

Possible Improvements:

Consideration of additional data augmentation techniques. Exploration of more advanced architectures or pre-trained models. Inclusion of a thorough analysis of false positives/negatives and clinical validation.

Chest X-ray image segmentation project has potential real-world applications in the field of medical imaging and healthcare.

The chest X-ray image segmentation project holds significant promise for real-world applications in the field of medical imaging and healthcare. By automating the segmentation of critical regions, such as the lungs, the project contributes to disease detection and diagnosis. The accurate segmentation results offer potential benefits in treatment planning and monitoring, assisting healthcare professionals in determining the extent of abnormalities and tracking changes over time. Beyond diagnosis, the model could optimize radiological workflows by providing preliminary segmentations, potentially expediting the diagnostic process. Acting as a decision support system, it offers valuable insights during the interpretation of X-ray images. The automated segmentation system becomes a crucial tool for remote healthcare assessments. Additionally, the project serves educational purposes, aiding medical students and professionals in understanding and visualizing chest X-ray segmentations. As a research tool, it contributes to clinical studies focused on lung diseases' patterns and characteristics. Successful deployment in clinical settings necessitates thorough validation, even, ethical considerations, including patient privacy and data security, should be paramount in the implementation of such systems in healthcare practice.

Thank You

```
In [ ]:
```