SYNTHETIC CT GENERATION FROM MR IMAGE

1. INTRODUCTION

Medical image reconstruction aims to acquire high-quality medical images for clinical usage at minimal cost and risk to the patients. Deep learning and its applications in medical imaging, especially in image reconstruction have received considerable attention in the literature in recent years. More specifically, synthetic CT (computed tomography) images generated from MR (magnetic resonance) data are becoming more popular because they offer several potential advantages over traditional CT scans.

One advantage is that MR scans do not use ionizing radiation, which can be harmful to the body. This makes MR scans a safer option for patients, particularly for those who may be pregnant or have a history of radiation exposure.

Another advantage is that MR scans provide more detailed images of soft tissues and organs, which can be especially useful for certain types of examinations. Synthetic CT images generated from MR data can be used to create detailed 3D models of organs and other structures, which can be helpful for planning surgical procedures or for visualizing abnormalities.

In addition, synthetic CT images can be generated quickly and easily from MR data, which can save time and reduce the need for multiple scans. This can be especially beneficial in emergency situations where rapid diagnosis is critical.

Overall, the use of synthetic CT images generated from MR data is increasing because they offer a number of potential advantages over traditional CT scans, including improved safety, greater detail, and the ability to generate 3D models for surgical planning.

2. OBJECTIVES

To build a deep learning model which is capable of generating accurate synthetic CT images from corresponding MR images.

3. PREREQUISITES

- tensorflow 2.6.4
- numpy 1.21.6
- openCV 4.5.4
- matplotlib 3.5.3
- pandas 1.3.5
- keras 2.6.0

4. IMPLEMENTATIONS

- Data preprocessing methods: N4 bias correction, histogram matching and mask generation
- U-Net for CNN model
- Encoder of the U-Net is initialized by pretrained VGG19 weights
- Best model is saved based on perceptual loss in validation data

5. DATASET

Initially the dataset which I found on kaggle contained around 4500 2D MRI-CT slices. But upon further investigation, I found that the data did not contain enough details to obtain proper results.

So, I contacted the author of the paper which I was following to implement this model to provide me with the dataset which was previously not publicly available. The obtained dataset contains 367 paired images of brain MRI and CT scan. It is randomly divided into training, validation and test data. The raw dataset can be downloaded from here.

6. DATA PREPROCESSING

The raw data obtained needed some preprocessing to be fed into the model. Preprocessing methods include N4 bias correction, histogram matching and mask generation. A utils script was created for the preprocessing methods as well as to access files.

Each MR image was corrected for intensity non-uniformity using the N3 bias field correction algorithm. All MR images were then histogram-matched to a randomly chosen template to help standardize image intensities across different patients using the method of Cox et al.

To better evaluate the accuracy of sCT estimation, a binary head mask was automatically derived from each MR image to separate the head region from the non-anatomical, background region of the image. This was achieved by applying the Otsu auto-thresholding method on each MR image. A morphological closing operator was employed to fill in gaps around the nasal cavities and the ear canals, the largest connected component of which then produced the head mask. The CT images also had the head frame which was clearly visible. In order to get accurate results, the head frame from each CT image was artificially removed.

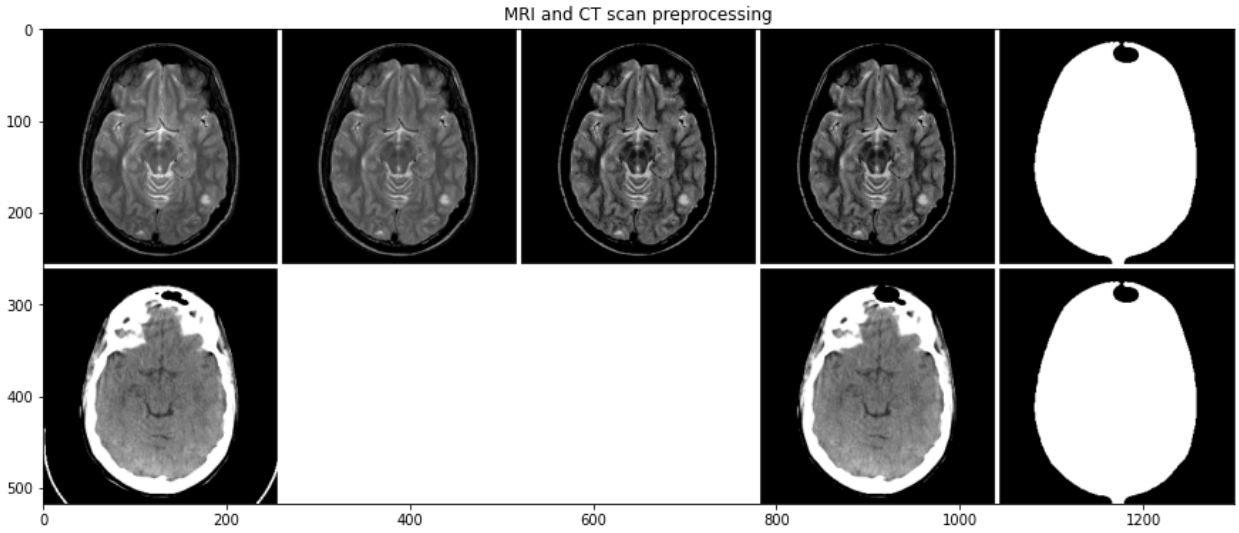

Fig. MRI and CT scan preprocessing steps

After applying the preprocessing methods, the final processed MR and CT images along with their corresponding masks is illustrated below:

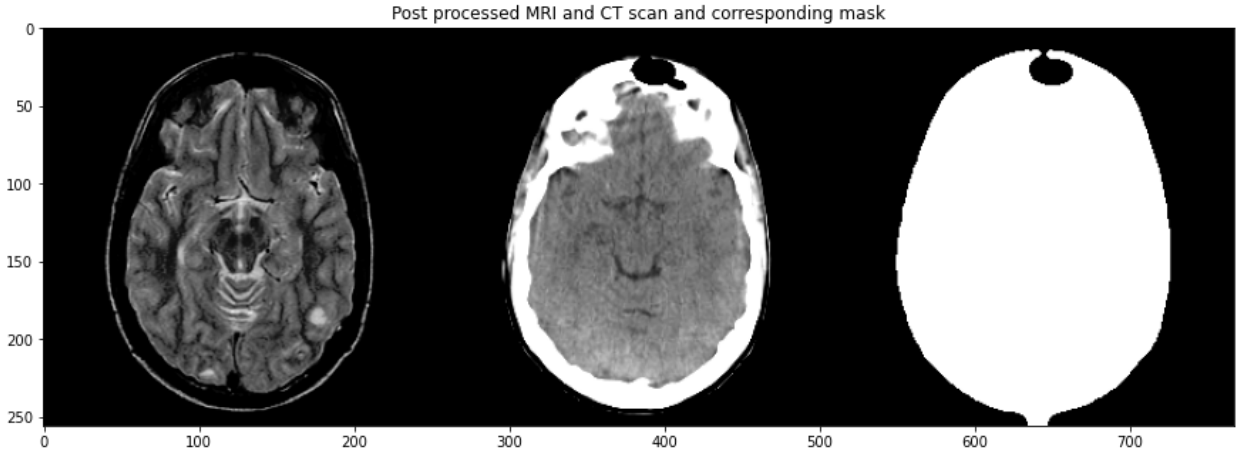

Fig. Post processed images

After data preprocessing, the dataset was further divided into training, test and validation sets with each containing both MR and CT images. 256 images were used for training, 64 images were used for validation and 47 images were used for test sets.

7. DATA AUGMENTATION

Due to the limited dataset, it was necessary to augment the data to make the model more robust. Hence, simple data augmentation is also performed to artificially increase the number of training data during model training, which applies a random translation of up to 20 pixels in each spatial dimension for each pair of MR and CT images or randomly flips the images.

Method	Default	Adjusted
Horizontal Flip	None	True
Width Shift Range	-	0.05
Height Shift Range	-	0.05
Shear Range	-	0.05
Zoom Range	-	0.05
Rotation Range	-	0.2
Fill Mode	None	'nearest'

Fig. Table showing the different data augmentation techniques applied

8. METHOD

In this implementation, I developed a 2D DCNN model to directly learn a mapping function for converting a 2D MR slice to its equivalent 2D CT. This work was built upon the developments in semantic image segmentation where a deep CNN model can be trained from end to end to directly produce a dense label map for object segmentation in a 2D image. In particular, the U-net architecture that was proposed by Ronneberger was adopted and modified and the resulting architecture is shown.

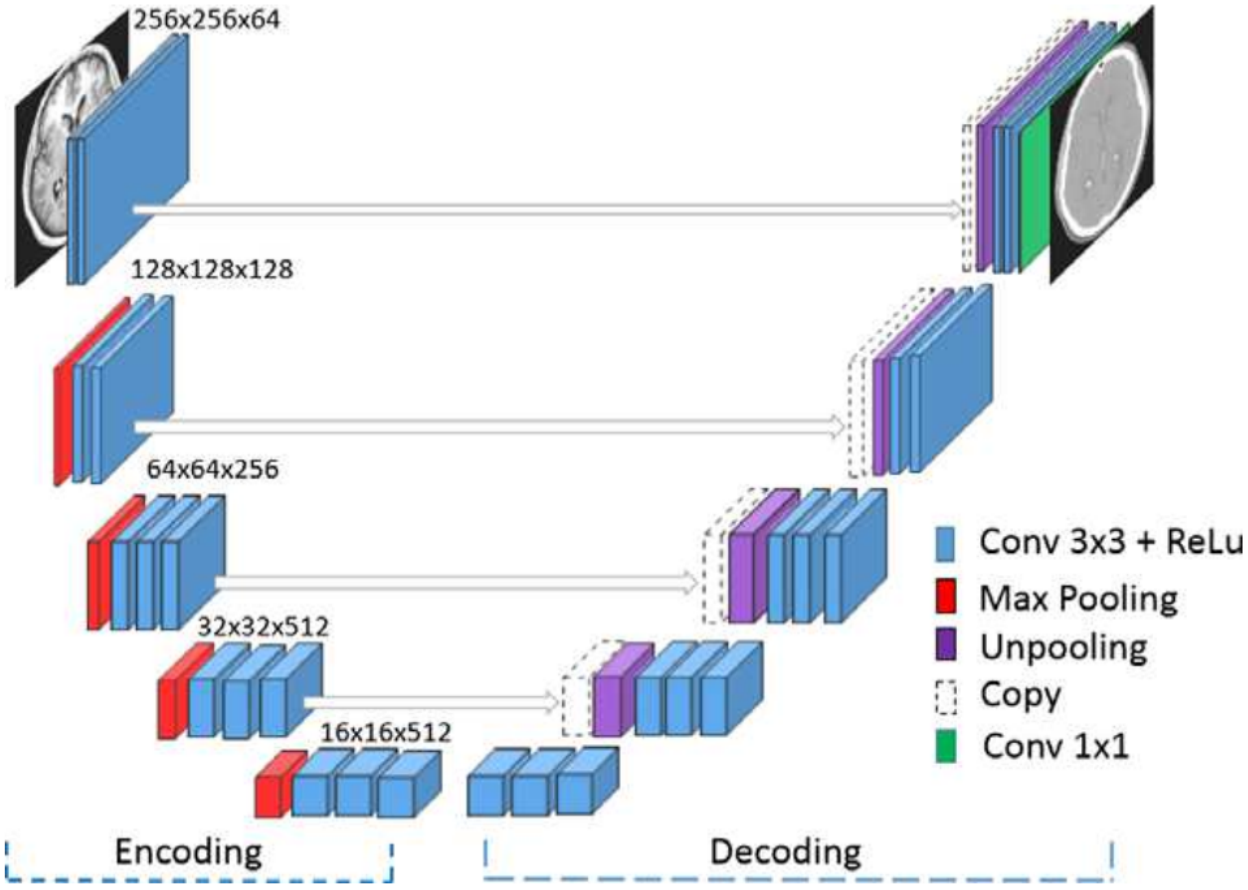

Fig. U-net architecture

Similar to Ronnerberger, the model can be seen as consisting of two main parts: an encoding part (left half) and a decoding part (right half). The encoding part behaves as traditional CNNs that learn to extract a hierarchy of increasingly complex features from an input MR image. The decoding part transforms the features and gradually reconstructs the sCT prediction from low to high resolution. The final output of the network is a 2D image with the same size as the input image.

Different from Ronnerberger, the encoding part was designed to follow the same architecture as the popular VGG19 model. Although, people who have worked with medical data feel uneasy about using ImageNet pretrained weights as a starting point because medical data looks different than ImageNet data, it is seen that it makes a lot more sense to use pretrained weights than to build the model from scratch because practitioners have achieved human level accuracy using imagenet pretrained weights for biomedical image research purpose. Moreover, the choice to particularly use VGGNet for our model is the result of the paper "Performance Comparison of Different Pre-Trained Deep Learning Models in Classifying Brain MRI Images" which compared models such as Inception V3, R-CNN, CNN-UNet, VGG-16, GoogleNet and other

custom CNNs and the conclusion was that VGG16 was the most successful among all architectures. Hence, I initially used VGG16 as pretrained weights for the encoder part which showed significant results but later, I also implemented the model using VGG19 weights which showed much better results, hence that is what was used for the final implementation of the model.

8.1. Loss Function

Since, a synthetic CT image is generated from MR image using the model, it is important to compare the similarity between the generated image and the actual CT and for that purpose, perceptual loss performs the best since it is used to compare high level differences, like content and style discrepancies, between images. A perceptual loss function is very similar to the per-pixel loss function, as both are used for training feed-forward neural networks for image transformation tasks. The perceptual loss function is a more commonly used component as it often provides more accurate results regarding style transfer.

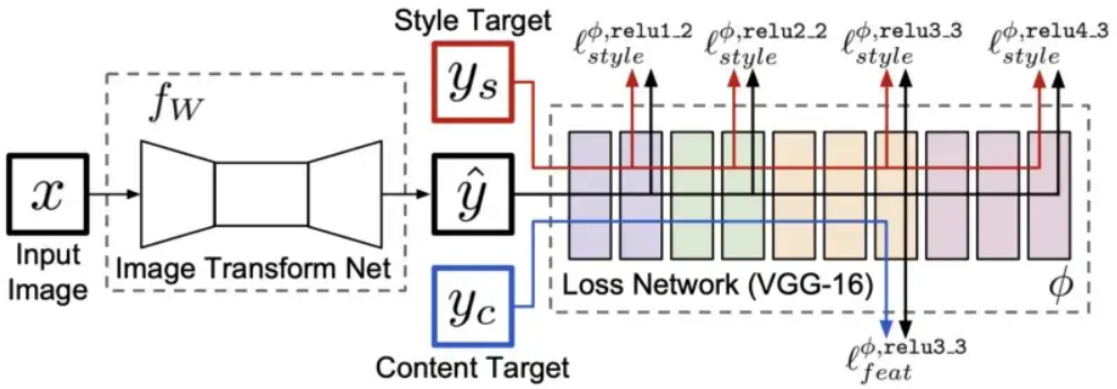

Fig. Perceptual Loss

8.2. Model implementation details

The model is trained using backpropagation with the Adam stochastic optimization method which offers faster convergence than standard stochastic gradient descent methods. The stochastic optimization method randomly selects a subset of training samples (each consists of an MR slice and corresponding CT slice) at each iteration of the optimization which is known as a mini-batch. Since the mini-batch is limited by available GPU memory, we use an effective mini-batch size of 32 for model training. As mentioned earlier, the model parameters for the encoder part of the network are initialized using corresponding weights from a pre-trained 19 layer VGG model as available from the keras.applications package. After the model is built, the parameters are as follows:

Total params:	28,812,225
Trainable params:	28,808,285
Non-trainable params:	3840

The training parameters for the model are as follows:

Parameters	Values
EPOCHS	500
BATCH_SIZE	32
learning_rate	1e-4
decay_rate	Learning_rate / EPOCHS
STEPS_PER_EPOCH	len(df_train) / BATCH_SIZE
SAVE_PERIOD	50
input_size	(256, 256, 3)

9. CODE IMPLEMENTATION

9.1. Utils Script

A utility script is defined to implement basic methods to access files, preprocess data and get corresponding masks of the MR and CT images. The utility file can be accessed in the following way:

```
from utils import *
```

```
# Script
import os
import cv2
import numpy as np
import SimpleITK as sitk
from scipy.stats import pearsonr


def all_files_under(path, extension='png', append_path=True, sort=True):
    if append_path:
        if extension is None:
            filenames = [os.path.join(path, fname) for fname in
os.listdir(path)]
        else:
            filenames = [os.path.join(path, fname) for fname in
os.listdir(path) if fname.endswith(extension)]
    else:
        if extension is None:
            filenames = [os.path.basename(fname) for fname in os.listdir(path)]
        else:
            filenames = [os.path.basename(fname) for fname in os.listdir(path)
if fname.endswith(extension)]

    if sort:
        filenames = sorted(filenames)

    return filenames


def histogram(img, bins=256):
    h, w = img.shape
    hist = np.zeros(bins)
    for i in range(h):
        for j in range(w):
            a = img.item(i, j)
            hist[a] += 1

    return hist


def cumulative_histogram(hist, bins=256):
    cum_hist = hist.copy()
    for i in range(1, bins):
        cum_hist[i] = cum_hist[i-1] + cum_hist[i]
```

```
    return cum_hist


def n4itk(img):
    ori_img = img.copy()
    mr_img = sitk.GetImageFromArray(img)
    mask_img = sitk.OtsuThreshold(mr_img, 0, 1, 200)

    # Convert to sitkFloat32
    mr_img = sitk.Cast(mr_img, sitk.sitkFloat32)
    # N4 bias field correction
    num_fitting_levels = 4
    num_iters = 200
    try:
        corrector = sitk.N4BiasFieldCorrectionImageFilter()
        corrector.SetMaximumNumberOfIterations([num_iters] *
num_fitting_levels)
        cor_img = corrector.Execute(mr_img, mask_img)
        cor_img = sitk.GetArrayFromImage(cor_img)

        cor_img[cor_img<0], cor_img[cor_img>255] = 0, 255
        cor_img = cor_img.astype(np.uint8)
        return ori_img, cor_img  # return origin image and corrected image
    except (RuntimeError, TypeError, NameError):
        print('[*] Catch the RuntimeError!')
        return ori_img, ori_img


def histogram_matching(img, ref, bins=256):
    assert img.shape == ref.shape

    result = img.copy()
    h, w = img.shape
    pixels = h * w

    # histogram
    hist_img = histogram(img)
    hist_ref = histogram(ref)
    # cumulative histogram
    cum_img = cumulative_histogram(hist_img)
    cum_ref = cumulative_histogram(hist_ref)
    # normalization
    prob_img = cum_img / pixels
    prob_ref = cum_ref / pixels
```

```
    new_values = np.zeros(bins)
    for a in range(bins):
        j = bins - 1
        while True:
            new_values[a] = j
            j = j - 1

            if j < 0 or prob_img[a] >= prob_ref[j]:
                break

    for i in range(h):
        for j in range(w):
            a = img.item(i, j)
            b = new_values[a]
            result.itemset((i, j), b)

    return result


def get_mask(image, task='m2c'):
    # Bilateral Filtering
    # img_blur = cv2.bilateralFilter(image, 5, 75, 75)
    img = image.copy()
    # gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
    th, img_thr = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)

    img_mor = img_thr.copy()

    # For loop closing
    for ksize in range(21, 3, -2):
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (ksize, ksize))
        img_mor = cv2.morphologyEx(img_mor, cv2.MORPH_CLOSE, kernel)

    # Copy the thresholded image.
    im_floodfill = img_mor.copy()

    # Mask used to flood filling.
    # Notice the size needs to be 2 pixels than the image.
    h, w = img_mor.shape
    mask = np.zeros((h + 2, w + 2), np.uint8)

    # Floodfill from point (0,0)
    cv2.floodFill(im_floodfill, mask, (0, 0), 255)
```

```
    # Invert floodfilled image
    img_floodfill_inv = cv2.bitwise_not(im_floodfill)

    # Combine the two images to get the foreground.
    pre_mask = img_mor | img_floodfill_inv

    # Find the biggest contour
    mask = np.zeros((h, w), np.uint8)
    max_pix, max_cnt = 0, None
    contours, hierarchy = cv2.findContours(pre_mask, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)
    for cnt in contours:
        num_pix = cv2.contourArea(cnt)
        if num_pix > max_pix:
            max_pix = num_pix
            max_cnt = cnt

    cv2.drawContours(mask, [max_cnt], 0, 255, -1)

    if task.lower() == 'm2c':
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
        mask = cv2.dilate(mask, kernel, iterations=2)

    return mask
```

9.2. Prerequisites

```
import pandas as pd
from sklearn.model_selection import train_test_split
from keras import backend as K
from keras.applications import VGG19
from keras.layers import Conv2D, BatchNormalization, Activation, MaxPool2D,
Conv2DTranspose, Concatenate, Input
from keras.models import Model
from keras.applications import VGG19
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

9.3. Accessing dataset

```
data = '/path/to/dataset/'
```

```
# All files
filenames = all_files_under(data+'raw', extension='png')
mri_files = all_files_under(data+'mri', extension = 'png')
ct_files = all_files_under(data+'ct', extension = 'png')
mask_files = all_files_under(data+'mask', extension = 'png')
preprocessed_files = all_files_under(data+'preprocessing', extension = 'png')
post_files = all_files_under(data+'post', extension = 'png')
```

9.4. Preprocessing Images

Preprocessing methods including N4 bias correction, histogram matching and mask generation are applied to the images and the resulting images are saved in a folder if not already saved. This step allows us to create separate folders containing process MR images, CT images and corresponding masks.

```
# temp_id = template image id for histogram matching
save_path = '/path/to/folder'
def imshow(ori_mr, cor_mr, his_mr, masked_mr, mask, ori_ct, masked_ct,
size=256, delay=0, himgs=2, wimgs=5, margin=5):
    """
    Input parameters:
    ori_mr : Original MR image
    cor_mr: N4ITK corrected MR image
    his_mr: Histogram matched MR image
    masked_mr: Mask of the MR image
    ori_ct: Original CT image
    masked_ct: Mask of the CT image

    Result: A concatenated image of each preprocessing step output
    """
    canvas = 255 * np.ones((himgs * size + (himgs-1) * margin, wimgs * size +
(wimgs-1) * margin), dtype=np.uint8)

    first_rows = [ori_mr, cor_mr, his_mr, masked_mr, mask]
    second_rows = [ori_ct, 255*np.ones(ori_ct.shape),
255*np.ones(ori_ct.shape), masked_ct, mask]
    for idx in range(len(first_rows)):
        canvas[:size, idx*(margin+size):idx*(margin+size)+size] =
first_rows[idx]
        canvas[-size:, idx*(margin+size):idx*(margin+size)+size] =
second_rows[idx]
```

```
    # "N4 Bias Field Correction",
    # cv2_imshow( canvas)
    if cv.waitKey(delay) & 0XFF == 27:
        sys.exit('[*] Esc clicked!')

    return canvas

def main(data, temp_id=2, size=256, delay=0, is_save=True):
    """
    Takes the dataset path as input parameter.
    """
    preprocessing_folder = os.path.join(os.path.dirname(data), 'preprocessing')
    if is_save and not os.path.exists(preprocessing_folder):
        os.makedirs(preprocessing_folder)

    post_folder = os.path.join(os.path.dirname(data), 'post')
    if is_save and not os.path.exists(post_folder):
        os.makedirs(post_folder)

    ct_folder = os.path.join(os.path.dirname(data), 'ct')
    if is_save and not os.path.exists(ct_folder):
        os.makedirs(ct_folder)

    mri_folder = os.path.join(os.path.dirname(data), 'mri')
    if is_save and not os.path.exists(mri_folder):
        os.makedirs(mri_folder)

    mask_folder = os.path.join(os.path.dirname(data), 'mask')
    if is_save and not os.path.exists(mask_folder):
        os.makedirs(mask_folder)


    # read all files paths
    filenames = all_files_under(data, extension='png')

    # read template image
    temp_filename = filenames[temp_id]
    ref_img = cv.imread(temp_filename, cv.IMREAD_GRAYSCALE)
    ref_img = ref_img[:, -size:].copy()
    _, ref_img = n4itk(ref_img)  # N4 bias correction for the reference image

    for idx, filename in enumerate(filenames):
        # print('idx: {}, filename: {}'.format(idx, filename))

        img = cv.imread(filename, cv.IMREAD_GRAYSCALE)
```

```
        ct_img = img[:, :size]
        mr_img = img[:, -size:]

        # N4 bias correction
        ori_img, cor_img = n4itk(mr_img)
        # Dynamic histogram matching between two images
        his_mr = histogram_matching(cor_img, ref_img)
        # Mask estimation based on Otsu auto-thresholding
        mask=get_mask(his_mr, task='m2c')
        # Masked out
        masked_ct = ct_img & mask
        masked_mr = his_mr & mask
        canvas = imshow(ori_img, cor_img, his_mr, masked_mr, mask, ct_img,
masked_ct, size=size, delay=delay)
        canvas2 = np.hstack((masked_mr, masked_ct, mask))

        if is_save:
            cv.imwrite(os.path.join(preprocessing_folder,
os.path.basename(filename)), canvas)
            cv.imwrite(os.path.join(post_folder, os.path.basename(filename)),
canvas2)
            cv.imwrite(os.path.join(ct_folder, os.path.basename(filename)),
masked_ct)
            cv.imwrite(os.path.join(mri_folder, os.path.basename(filename)),
masked_mr)
            cv.imwrite(os.path.join(mask_folder, os.path.basename(filename)),
mask)
main('/path/to/dataset')
```

9.5. Create data frame and split data on train set, validation set and test set

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.DataFrame(data={"mri": mri_files, 'ct' : ct_files})
df_train, df_test = train_test_split(df,test_size = 0.128)
df_train, df_val = train_test_split(df_train,test_size = 0.2)
```

9.6. Data generator

Data is generated by first applying normalization techniques to the images which are further augmented in various ways.

```
from keras.preprocessing.image import ImageDataGenerator

# Normalize data

def adjust_data(img,ct):
    img = img / 255
    mask = ct / 255
#     mask[mask > 0.5] = 1
#     mask[mask <= 0.5] = 0

    return (img, mask)

def train_generator(data_frame, batch_size, aug_dict,
        image_color_mode="rgb",
#         mask_color_mode="grayscale",
        mri_save_prefix="mri",
        ct_save_prefix="ct",
        save_to_dir=None,
        target_size=(256,256),
        seed=1):
    '''
    can generate mri and ct image at the same time use the same seed for
    image_datagen and ct_datagen to ensure the transformation for mri
    and ct image is the same.If you want to visualize the results of generator,
    set save_to_dir = "your path"
    '''
    mri_datagen = ImageDataGenerator(**aug_dict)
    ct_datagen = ImageDataGenerator(**aug_dict)

    mri_generator = mri_datagen.flow_from_dataframe(
        data_frame,
        x_col = "mri",
        class_mode = None,
        color_mode = image_color_mode,
        target_size = target_size,
        batch_size = batch_size,
        save_to_dir = save_to_dir,
        save_prefix  = mri_save_prefix,
        seed = seed)
```

```
    ct_generator = mri_datagen.flow_from_dataframe(
        data_frame,
        x_col = "ct",
        class_mode = None,
        color_mode = image_color_mode,
        target_size = target_size,
        batch_size = batch_size,
        save_to_dir = save_to_dir,
        save_prefix  = ct_save_prefix,
        seed = seed)

    train_gen = zip(mri_generator, ct_generator)

    for (mri, ct) in train_gen:
        mri, ct = adjust_data(mri, ct)
        yield (mri,ct)
```

9.7. Define Loss Function and metrics

Perceptual loss function is defined.

```
from tensorflow.keras import backend as K
smooth=100

tf.compat.v1.disable_eager_execution()
from tensorflow.keras.applications import VGG19

def perceptual_loss_vgg19(y_true, y_pred):
    # must ensure that y_true and y_pred come from same distribution
    # nice to have - similar distribution to vgg19 inputs
    y_pred = tf.image.grayscale_to_rgb(y_pred, name=None) #need to convert as
UNet outputs single channel
    vgg = VGG19(include_top=False, weights="imagenet", input_shape=(256, 256,
3))
    loss_model = Model(
        inputs=vgg.input, outputs=vgg.get_layer("block3_conv3").output
    )
    return K.mean(K.square(loss_model(y_true) - loss_model(y_pred)))
```

9.8. Define model architecture

The model is defined using the pretrained VGG19 weight for the encoder part of the U-Net.

```
# Defining the architectural blocks

def conv_block(input, num_filters):
    x = Conv2D(num_filters, 3, padding="same")(input)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    x = Conv2D(num_filters, 3, padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    return x

def decoder_block(input, skip_features, num_filters):
    x = Conv2DTranspose(num_filters, (2, 2), strides=2, padding="same")(input)
    x = Concatenate()([x, skip_features])
    x = conv_block(x, num_filters)
    return x

def build_vgg19_unet(input_shape):
    """ Input """
    inputs = Input(input_shape)

    """ Pre-trained VGG19 Model """
    vgg19 = VGG19(include_top=False, weights="imagenet", input_tensor=inputs)

    """ Encoder """
    s1 = vgg19.get_layer("block1_conv2").output         ## (512 x 512)
    s2 = vgg19.get_layer("block2_conv2").output         ## (256 x 256)
    s3 = vgg19.get_layer("block3_conv3").output         ## (128 x 128)
    s4 = vgg19.get_layer("block4_conv3").output         ## (64 x 64)

    """ Bridge """
    b1 = vgg19.get_layer("block5_conv3").output         ## (32 x 32)

    """ Decoder """
    d1 = decoder_block(b1, s4, 512)                     ## (64 x 64)
    d2 = decoder_block(d1, s3, 256)                     ## (128 x 128)
    d3 = decoder_block(d2, s2, 128)                     ## (256 x 256)
    d4 = decoder_block(d3, s1, 64)                      ## (512 x 512)
```

```
    """ Output """
    outputs = Conv2D(1, 1, padding="same", activation="sigmoid")(d4)

    model = Model(inputs, outputs, name="VGG19_U-Net")
    return model
```

9.9. Build and compile model

```
input_size = (256,256,3)

vgg19_unet = build_vgg19_unet(input_size)
vgg19_unet.summary()
```

Now, compile the model using Adam optimization and save the model every 50 epochs.

```
# Define the parameters
EPOCHS = 500
BATCH_SIZE = 32
learning_rate = 1e-4
decay_rate = learning_rate / EPOCHS
STEPS_PER_EPOCH = len(df_train) / BATCH_SIZE
SAVE_PERIOD = 50
```

```
# Save model every 10 epochs
model_file_format = "model_unet256_vgg19.{epoch:03d}.hdf5"
opt = keras.optimizers.Adam(learning_rate=learning_rate,beta_1=0.9,
beta_2=0.999, epsilon = 0.01, decay=decay_rate, amsgrad=False)

checkpointer = ModelCheckpoint(model_file_format, verbose=1, save_freq=
int(SAVE_PERIOD * STEPS_PER_EPOCH))

model.compile(optimizer=opt, loss=perceptual_loss_vgg19,
metrics=[perceptual_loss_vgg19,"accuracy"])
```

9.10. Train model

First training, test and validation data are generated using the train_generator method.

```
train_generator_args = dict(rotation_range=0.2,
                            width_shift_range=0.05,
                            height_shift_range=0.05,
                            shear_range=0.05,
                            zoom_range=0.05,
                            horizontal_flip=True,
                            fill_mode='nearest')
train_gen = train_generator(df_train, BATCH_SIZE,
                                train_generator_args,
                                target_size=(256, 256))

val_gen = train_generator(df_val, BATCH_SIZE,
                                dict(),
                                target_size=(256, 256))

test_gen = train_generator(df_test, BATCH_SIZE,
                                dict(),
                                target_size=(256, 256))
```

Now, train the model for 500 epochs and store the history in the 'history' variable.

```
history = model.fit(train_gen,
                    steps_per_epoch=len(df_train) / BATCH_SIZE,
                    epochs=EPOCHS,
                    callbacks = [checkpointer],
                    validation_data = val_gen,
                    validation_steps=len(df_val) / BATCH_SIZE)
```

10. RESULTS

After training the model for 500 epochs, the following results were obtained.

Data Type	Perceptual Loss	Accuracy
Training Data	5.5441	67.99
Validation Data	6.1970	70.81

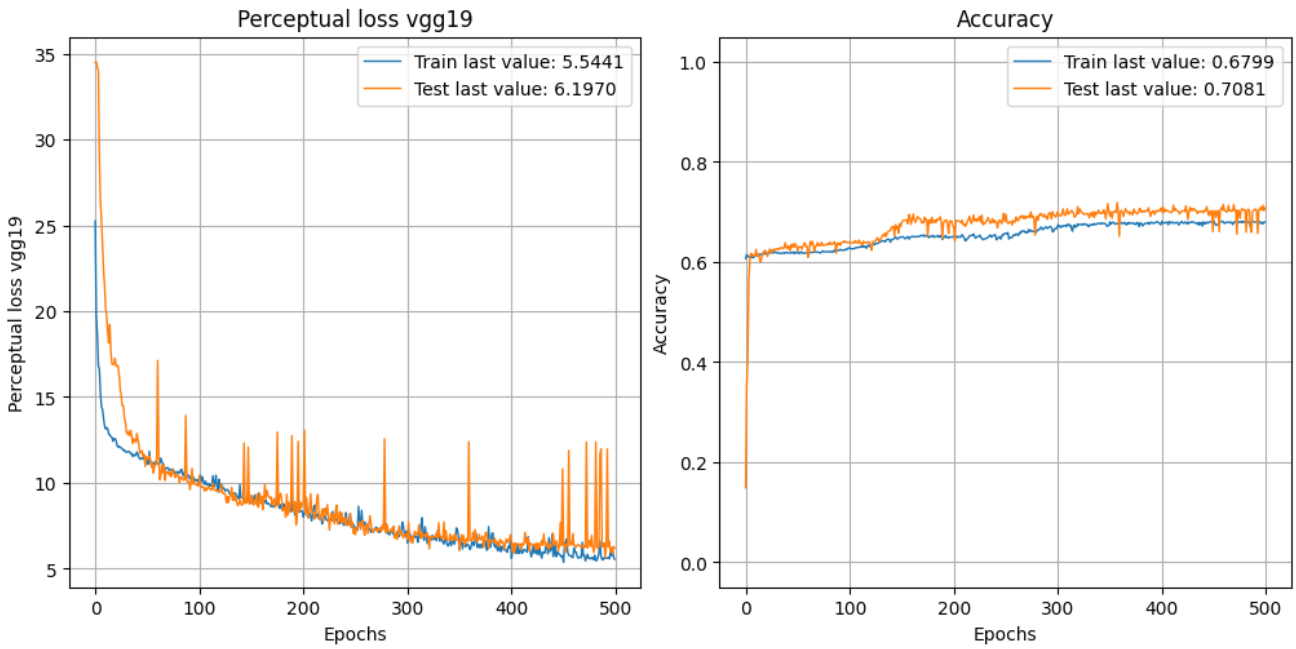

Fig.: Visualization of accuracy and loss functions of the U-Net model with pre-trained VGG19 weights over 500 epochs

In the perceptual loss function, it is observed that there are suddens spikes in validation data which is perfectly normal due to multiple reasons:

- Using mini batches will introduce some additional noise, as the gradients for each weight update iteration will not be calculated on all available training data, but only a limited subset (batch). This extra noise is a small price to pay given the considerable speedups mini-batch yields, leading to faster convergence.

- The other reason may be because the validation metrics may be noisy since the validation set is a much smaller sample than the training set and is not the target for optimization (training/validation distributions may not overlap completely).

Moreover, the results from pretrained VGG16 weights are also visualized to compare why the VGG19 generates better results.

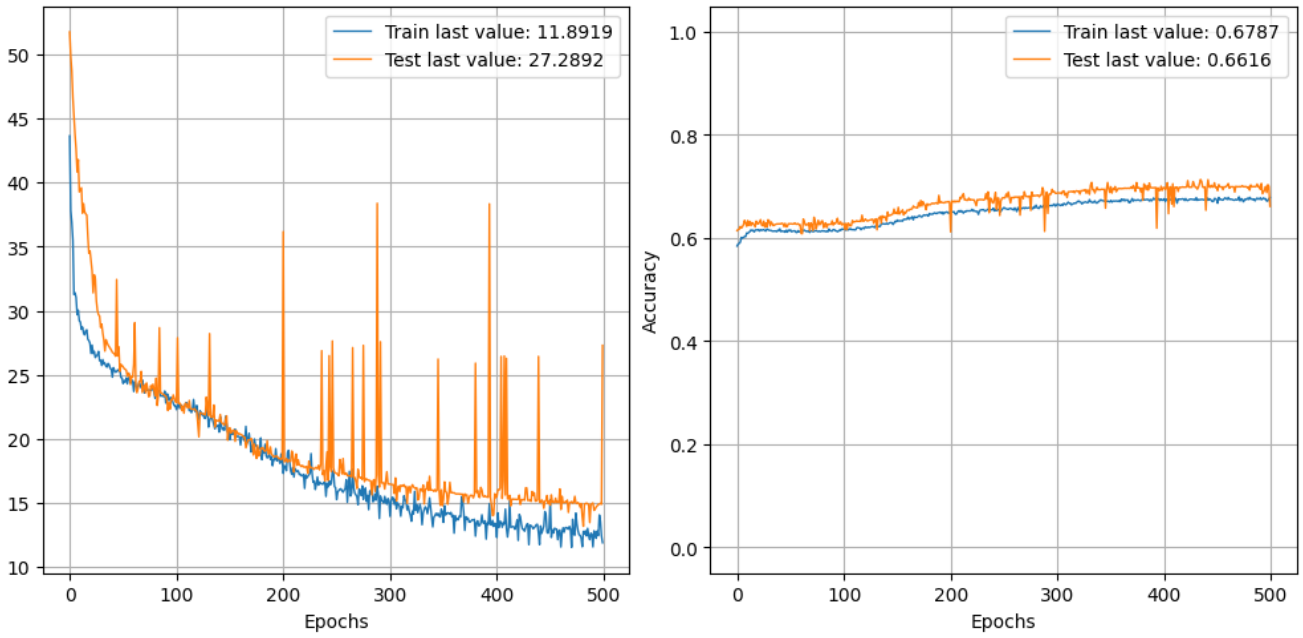

Fig. Visualization of accuracy and loss functions of the U-Net model with pre-trained VGG16 weights over 500 epochs

Model	Data Type	Perceptual Loss	Accuracy
VGG16	Training Data	11.8919	67.87
	Validation Data	27.2892	66.16
VGG19	Training Data	5.5441	67.99
	Validation Data	6.1970	70.81

It is clear that the spikes in the loss function are much larger in VGG16 compared to the VGG19 pretrained model and the validation loss after 500 epochs is almost 5 times larger than that compared to the VGG19 model.

It was also analyzed how the model performs on test data and the results were as follows:

Model	Perceptual Loss	Accuracy
VGG16	14.428	71.53
VGG19	6.21	69.03

The model was then used to predict synthetic CT images from MR images using test data and the results were as follows:

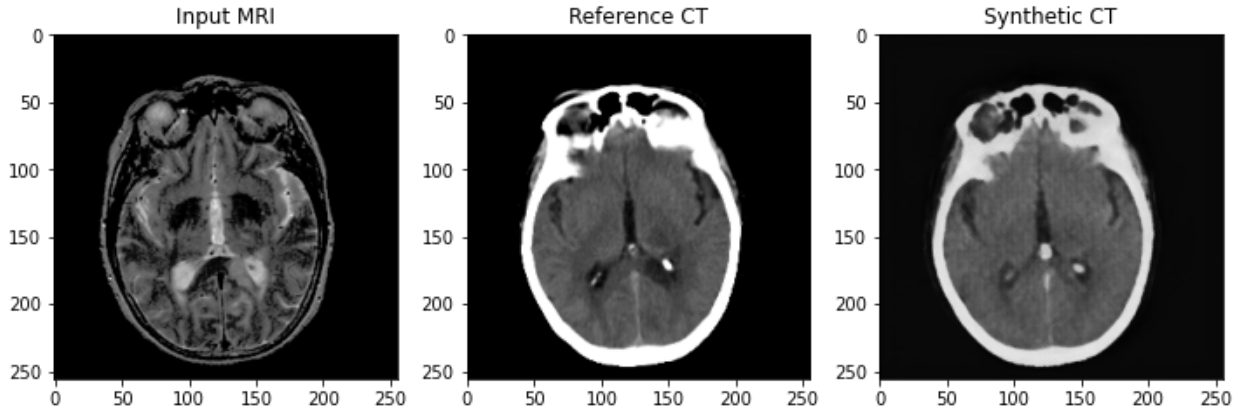

Fig. The input MR, reference CT and the synthetic CT produced by the model

11. CONCLUSION

The U-Net network with pretrained VGG19 weights was evaluated to analyze whether the model was able to generate comparable synthetic CT images from corresponding MR images.

It was observed that after 500 epochs the model performed well with about an accuracy of 70% with a perceptual loss of about 6 which is impressive considering how different the input image and generated image are. I think the model would be able to perform much better if the dataset can be expanded since currently, the model is working with only 367 images.