21-coursework

December 13, 2019

Group 21 Rebuild MRI in the methods of DeblurGAN

Group 21 Member Name: Huang, ZIJIAN Ma, Yuhao Zhang, Zongxu Wan, Zhenyu Tzeng, JIUN-RUNG Song, YEWEI

Weblink: https://drive.google.com/open?id=1UeXd93w3XCnAOJGZGu8t3vQJkvHyCGts

1 Introductuon

1.1 MRI dataset

The MRI dataset that our group use contains raw k-space data from 100-dimensional volumes, each having about 30-40 two-dimension slices. For this experiment, the data is divided into two parts, 70% of data for training and 30% of data for testing. For the training set, each volume contains fully sampled k-space data. We can compute ground truth images and simulate under sampled data (either 4-fold AF or 8-fold AF) by using the random mask generator based on the training set. For the test set, we use 4-fold and 8-fold masks to generate 4-fold and 8-fold under sampled k-space data based on fully sampled k-space data, respectively. At last, the format of the dataset that we use is .h5 files.

1.2 Machine Learning Task

The machine learning is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention. The data MRI scanner generate is a complex-valued k-space data. K-space is the spatial frequency information (Fourier coefficients) of the object imaged. Therefore, we can perform the inverse Fourier transform (DFT) to the k-space data in order to reconstruct the image, $y\ \mathbb{C}^N$,

$$m = F^{-1}(y)$$

Where $m\ \mathbb{C}^N$ is complex-valued image and F^{-1} is DFT.

1.3 1.3 Aim to achieve

The aim of this experiment is reconstructing images using Magnetic Resonance Imaging (MRI) to reduce long acquisition time compare to Compressed Sensing based MRI (CS-MRI). Our group compare the image that we predicted with the ground truth image that origin dataset given. We

designed the neural network and adjusted the loss function to reconstruct the image in a shorter time.

2 2. Design

The network structure of our group applies the Deblur generative adversarial network which is based on the generative adversarial network.

2.1 2.1 Generative Adversarial Network

Generative adversarial network is the basic deep learning network. The framework is to define a game between two competing networks: the discriminator and the generator. (Orest Kupyn, n.d.) These two modules contest with each other to produce quite good output. Generative module is a network that generates images, it receives a random noise to generate the image from this noise. Discriminative module is a network that discriminates whether the image is 'real'. During the training process, generative module generates the 'real' image to trick the discriminative module, then discriminative module to divide the real image and fake image. At last, the generative module and discriminative module arrives an ideal situation that the image produced by generative module cannot discriminate by discriminative module. Then we get a good generative network.

2.2 2.2 Deblur Generative Adversarial Network

This kind of network can restore more pictures texture detail compare to normal generative adversarial network.

2.2.1 2.2.1 Generator network

The goal of the generator network is tricking the discriminator network by generating the images which cannot discriminate by discriminator network. The input of the generator network is the image of the MRI dataset. The deblur generative adversarial network generator architecture is applying in our group project. This architecture eleven blocks which include two stride convolution blocks, nine residual blocks and two transposed convolution blocks. The following is a schematic of cascade DeblurGAN generator architecture used in this project:

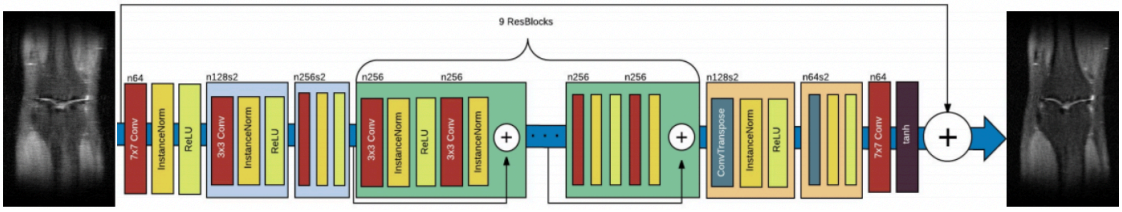

Figure 1. The architecture of generator

2.2.2 2.2.2 Discriminator network

The goal of the discriminator network is distinguishing between the real image and generated image by generator network. Then when the probability of judging a true image is close to 0.5, it means the generator network is already done.

2.2.3 2.2.3 Residual neural network

Residual neural network usually called ResNet. The structure of ResNet can speed up the training of neural networks extremely fast and improve the accuracy of the model. The main idea of ResNet is to add a direct connection channel in the network which is the idea of Highway Network. ResNet allow raw input information to be passed directly to subsequent layers.

2.3 2.3 Loss Function

In this coursework, our deblur generative adversarial network uses Wasserstein distance with the gradient penalty and perceptual loss.

2.3.1 2.3.1 Wasserstein loss

Wasserstein loss function try to increase the gap between the real image and generated image that generator produced. The critical loss is calculated in term of the difference between average score on real image and average score on generated image. This is the way to achieve the loss in deep learning framework such as PyTorch. Then we use gradient descent to minimize loss. This loss function used in our 4-fold AF discriminator.

2.3.2 2.3.2 Perceptual loss

Actually, the perceptual loss that we use in this group project is a simple L2-loss (Square loss function). However, the perceptual loss is related to the generate image and real image CNN feature maps, the difference of them is the perceptual loss. The goal of perceptual loss is restoring general content of image.

2.3.3 2.3.3 L1 loss

L1 loss function also known as mean absolute error (MAE). MAE usually as a loss function for regression models. MAE is the sum of the absolute differences between the target variable and predict variable. Cause MAE measures the average error in a set of predictions and don't need to consider the direction of predictions. The value of MAE is between 0 to ∞. The mean absolute error is given by:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

Where y_i is the prediction and x_i is the true value.

2.3.4 2.3.4 Structural Similarity

Structural similarity also called SSIM. SSIM is a method to measure the similarity of two images. Structural similarity compares the similarity of the two images x, y in three dimensions which are luminance, contrast and structure. The similarity of two images is the function of three dimensions shown below.

$$S(x, y) = f(l(x, y), c(x, y), s(x, y))$$

2.4 2.4 Summary

In general, the network structure of our coursework project uses the Deblur generative adversarial network. We define the generator and discriminator to train the dataset. In order to get better result, we apply several loss functions like Wasserstein loss and Perceptual loss in the process. At last, our group uses the structural similarity loss function to evaluate the score of our neural network.

3 3. Implementation:

Our teams using the frame of GAN (Generative adversarial networks) and using the nine layers generator network models of DeblurGan which base on standard Deep RES-NET architectures. The reason why our team choose DeblurGAN is that compared with U-net, DeblurGan has more layers and complexity which is able to get more accusative results.

The reason of our team choose DeblurGan is that we ought to implement the task through tanning the data base on GAN (Generative Adversarial Networks). In the research of Goodfellow et al. (June 2014) points out that how to build up a GAN. First, designing a generator, which generate samples and a discriminator, and a discriminator, which try to distinguish the result of generators. Second, compose competing networks by the generator and the discriminator and use data to train them. In order to fit the goal that the discriminatory accuracy is 0.5. In other words, the training will be end up until the discriminator are not able to distinguish the real images and the images generated by the generator. Our team adopt the concept of GAN to train a generator which is able to rebuild the loosing K-space data into MRI images that most close to the result images which rebuilt from the complete K-space data.

Before choosing DeblurGan as our team's main process of our course work. Several methods also have been considered. According to the research of Orest Kupyn et al. (2018) mentioned that DeblurGan has the ability to preserve more details in the result images which makes the result much close to the real images is the main reason our team adopt it. Before choosing DeblurGan as our team's K-space rebuilt method, DAGAN (Deep De-Aliasing Generative Adversarial Networks) and GANCS (Deep Generative Adversarial Networks for Compressed Sensing) were also became as the choice of rebuilt function because all of them are based on GAN network that are able to fit the result we need. Nevertheless, our team finally decide not using GANCS to implement the task which mentioned above because of several reasons.

In one hand, According to the research of Guang Yang et al.(JUNE 2018) points out that DAGAN to implement MRI Reconstruction base on U-Net generator and announced that each image can be reconstructed in 5 ms. Compared to DeblurGan DAGAN is more emphasized the efficient instead

of accurate rate. Thus, our team choose DeblurGan instead because instead of U-NET which DAGAN adopt, RES-NET become better choice since it has more layers to decrease the influence of Degradation problems and fit our team's goal, high accuracy.

In other hands, compared to MSE and MAE, DeblurGan is able to restore better texture details to get sharper images. In other words, Using DeblurGan is suitable to fix the MRI images which rebuilt from the loss K-space data.

4 4. Experiment

4.1 4.1 Losses function

In this case, the loss function is vital. The loss function in this experiment is divided into two part to get the result more accurate.The reason is that the GAN and CNN all may create loss during the training. On one hand, GAN may create some loss in the recognize process, which may affect the result. Unlike the normal CGAN, the DeblurGAN use the WGAN-GP as critic function, which is shown to be robust to the choice of generator architecture(Orest K., Volodymyr B. etc., 2018). The biggest difference is WGAN can get the work done with less possibility broken rate and good result. On the other hand, CNN may also create some loss in picking up the feature in the images.

Experiments extract loss values from two levels for 4AF image: the end of the generator and the end of the entire model.The first is the perceptual loss, which is directly calculated on the output of the generator, which can ensure that the GAN model is oriented to the deblurring task.it can recover finer texture details and to achieve visually good looking results because if we take the with deep retrain network and like the features of deeper layers.It would correspond to some general image features like shapes general textures and other so minimizing for that loss would let us to recover the general information in the image and adding the additional gain component.It compares the output of the first convolution of the VGG convolution. The second loss is the Wasserstein loss, which is performed on the output of the entire model. It is taken from the average of the gap between the two images and is known for optimizing the convergence of the GAN network. A simple expression is developed to calculate it. By adding two expression together, the overall loss can be measured accurately. Because there are some difference between two expression, one expression multiply a constant, which is 100 in this experiment. Finally, L1 loss function is applied in 8AF image, because the difference between its predicted value and the training value is too large, the previous instant function is not applicable.

[ ]:
```
import keras.backend as K
from keras.applications.vgg16 import VGG16
from keras.models import Model
import numpy as np

# Note the image_shape must be multiple of patch_shape
image_shape = (256, 256, 3)


def l1_loss(y_true, y_pred):
    return K.mean(K.abs(y_pred - y_true))
```

```
#100 based loss
def perceptual_loss_100(y_true, y_pred):
    return 100 * perceptual_loss(y_true, y_pred)


def perceptual_loss(y_true, y_pred):
    vgg = VGG16(include_top=False, weights='imagenet', input_shape=image_shape)
    loss_model = Model(inputs=vgg.input, outputs=vgg.get_layer('block3_conv3').
 ↪output)
    loss_model.trainable = False
    return K.mean(K.square(loss_model(y_true) - loss_model(y_pred)))


def wasserstein_loss(y_true, y_pred):
    return K.mean(y_true*y_pred)

def gradient_penalty_loss(self, y_true, y_pred, averaged_samples):
    gradients = K.gradients(y_pred, averaged_samples)[0]
    gradients_sqr = K.square(gradients)
    gradients_sqr_sum = K.sum(gradients_sqr,
                              axis=np.arange(1, len(gradients_sqr.shape)))

    gradient_l2_norm = K.sqrt(gradients_sqr_sum)
    gradient_penalty = K.square(1 - gradient_l2_norm)

    return K.mean(gradient_penalty)
```

4.2 4.2 Main model

The main model is design as a resnet based GAN.

Model selection The first is why we use the resnet in our experiment? Compared with other network structures, the deep residual network model is simpler and clearer. Its core contribution is to effectively solve the degradation problem of convolutional neural networks due to gradient dispersion when the network layer deepens through the residual block structure. Constructing a neural network with deeper network layers improves the accuracy, and can maintain good generalization when the network layer deepens.

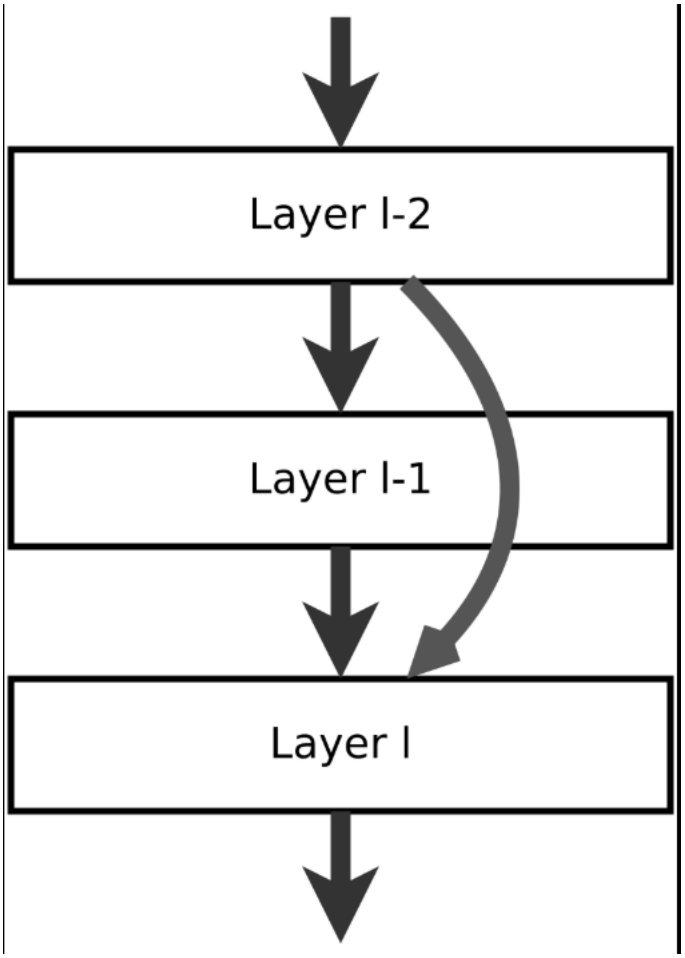

Figure 2. The model of Residual Network

For the reason that using resnet can deeply extract the graph feature, which can perform better in picture generation. During solving the output of network, the activation function sigmoid and relu is used frequently.But the Sigmoid function may cause gradient saturation and gradient disappearance problems when the gradient of the convolutional neural network is in the process of backpropagation. The value of a in the randomized ReLU activation function obeys a uniform distribution during the training phase.

$$\mathrm{ReLU}(x)=\begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Afterword, the Batch Normalization operation is taken to the output of the convolution to get the result easy to be trained in the later step.In the end, BN carries multiple samples of information during the training process, which is equivalent to a certain amount of noise itself, which plays a regularization effect, so the need for Dropout is basically eliminated.

[ ]:
```
from keras.layers import Input, Activation, Add, UpSampling2D
from keras.layers.advanced_activations import LeakyReLU
```

```
from keras.layers.convolutional import Conv2D
from keras.layers.core import Dense, Flatten, Lambda
from keras.layers.normalization import BatchNormalization
from functions.layer_utils import ReflectionPadding2D
from keras.layers.core import Dropout
# the paper defined hyper-parameter:chr
channel_rate = 64
# Note the image_shape must be multiple of patch_shape
image_shape = (256, 256, 3)
patch_shape = (channel_rate, channel_rate, 3)

ngf = 64   #Define the number of original convolution kernels in the generator␣
 ↪model
ndf = 64   #Define the number of original convolution kernels in the␣
 ↪discriminator model
input_nc = 3  #number of input channel
output_nc = 3  #number of output channel
input_shape_generator = (256, 256, input_nc)  #Define size of input
input_shape_discriminator = (256, 256, output_nc) # Define size of output
n_blocks_gen = 9 #Define the number of residual layers

#Define residual block function
def res_block(input, filters, kernel_size=(3, 3), strides=(1, 1),␣
 ↪use_dropout=False):
    """
    Instanciate a Keras Resnet Block using sequential API.

    :param input: Input tensor
    :param filters: Number of filters to use
    :param kernel_size: Shape of the kernel for the convolution
    :param strides: Shape of the strides for the convolution
    :param use_dropout: Boolean value to determine the use of dropout
    :return: Keras Model
    """
#Use a convolution operation with a step size of 1 to keep the size of the␣
 ↪input data same
    x = ReflectionPadding2D((1, 1))(input)
    x = Conv2D(filters=filters,
               kernel_size=kernel_size,
               strides=strides,)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    if use_dropout:
        x = Dropout(0.5)(x)
#Do convolution operation again with a step size of 1
    x = ReflectionPadding2D((1, 1))(x)
```

```
    x = Conv2D(filters=filters,
               kernel_size=kernel_size,
               strides=strides,)(x)
    x = BatchNormalization()(x)

    merged = Add()([input, x])
    return merged
```

4.2.1 4.2.1 Generator

Using ResNet may create a deep network which can build something much simpler that can work faster and work better but the problem is how to restore the lost information.Generative adversarial Network and generative models can help us to deal with that problem. So the main idea of the experiment is that Gan framework can be used to hallucinate some details that the resulting image would look realistic and it can deal with the fact that some information is lost. Therefore,adopt GAN method is the main theory of the process of image reconstruction. this kind of generative model where you have two networks during the learning process a generator and a discriminator the generator takes some noise vectors that sample from some distribution as an input and outputs an image the discriminator receives real image as an input and a fake image and is trying to distinguish between them and all the learning process is done in an iterative manner so the discriminate was the generator and discriminator.They're good at producing some good realistically image out of noise.Instead, we have a problem where we have an input image,not a noise, so this is a kind of conditional adversarial networks that can deal with that. So the whole difference is that in standard generative adverse area networks the generator receives noise back to set as an input and in conditional there are several networks. Generator G and discriminator D is the minimax objective: where Pr is the data distribution and Pg is the model distribution, defined by ~x = G(z); z v P(z), the input z is a sample from a simple noise distribution.

$$\min_G \max_D \mathop{\mathbb{E}}_{x \backsim \mathbb{P}_r} [\log(D(x))] + \mathop{\mathbb{E}}_{\tilde{x} \backsim \mathbb{P}_g} [\log(1 - D(\tilde{x}))]$$

The input and output are complex-valued images of the same size and each include three channels for real and imaginary components.

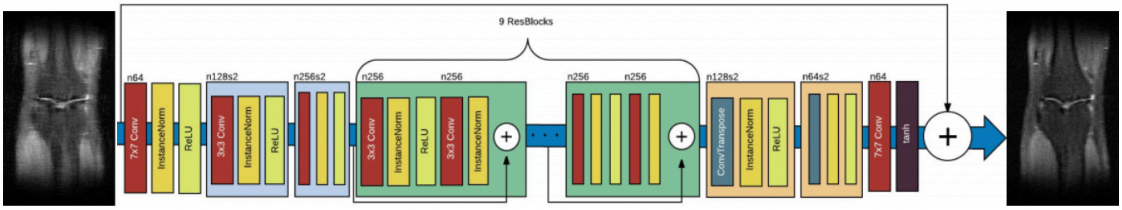

Figure 3. The architecture of generator

The network framework of the generator is shown above. The purpose of the generator is to reproduce sharp images. The neural network is fast based on the code of the ResNet network, and it continuously tracks the evolution of the original blurred image applied. The residual block of each

residual network contains two 3 * 3 convolution kernels and 256 feature map layers. Then perform batch normalization and RELU activation function processing. Nine Resnet blocks are applied to the under-sampled version of the input. We added a connection from input to output and divided the result by 2 to keep the standardized output different from the above architecture diagram, the process of deconvolution is replaced by the UpSampling2D operation. Under-Sampling2D can be regarded as the reverse operation of Pooling, which is to use Nearest Neighbor interpolation to enlarge, which is to copy the data of rows and columns to expand the size of the feature map. Undersampling discards most counter-example data. Models that can cause large deviations. However, undersampling may cause data sample points, reducing the calculation time.

[ ]:
```
#Building the generator mode
def generator_model():
    """Build generator architecture."""
    # Current version : ResNet block
    inputs = Input(shape=image_shape)
#Use a convolution operation with a step size of 1 to keep the size of the␣
 ↪input data same
    x = ReflectionPadding2D((3, 3))(inputs)
    x = Conv2D(filters=ngf, kernel_size=(7, 7), padding='valid')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    n_downsampling = 2
    for i in range(n_downsampling):
        mult = 2**i
        x = Conv2D(filters=ngf*mult*2, kernel_size=(3, 3), strides=2,␣
 ↪padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)

    mult = 2**n_downsampling
    for i in range(n_blocks_gen):
        x = res_block(x, ngf*mult, use_dropout=True)

    for i in range(n_downsampling):
        mult = 2**(n_downsampling - i)
        # x = Conv2DTranspose(filters=int(ngf * mult / 2), kernel_size=(3, 3),␣
 ↪strides=2, padding='same')(x)
        x = UpSampling2D()(x)
        x = Conv2D(filters=int(ngf * mult / 2), kernel_size=(3, 3),␣
 ↪padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)

    x = ReflectionPadding2D((3, 3))(x)
    #Convolution operation with a step size of 1
    x = Conv2D(filters=output_nc, kernel_size=(7, 7), padding='valid')(x)
```

```
    x = Activation('tanh')(x)

    ##Complete aresidual with the outermost input
    outputs = Add()([x, inputs])
    # outputs = Lambda(lambda z: K.clip(z, -1, 1))(x)
    #Prevent the eigenvalue range from being too large, and divide by 2 (take␣
↪the mean residual)
    outputs = Lambda(lambda z: z/2)(outputs)

    model = Model(inputs=inputs, outputs=outputs, name='Generator')
    return model
```

4.2.2 4.2.2 Discriminator

The idea of discriminator is similar to the idea of generator. Because of its function is making choice, it may not require the same complexity as the generator and also the output of the discriminator is not concerned. Like the generator, the inside of the discriminator is the CNN algorithm and the basic idea is doing the convolution, normalization and activation. The final layer is combine all the output above into one. In the end, by using the dense function, the output is equal to either 0 or 1.

Overall, The goal of the discriminator is to determine if the input image was created manually. Therefore, the architecture of the discriminator is convolutional and outputs a single value. The structure of the discriminator model is relatively simple. (1) The size of the input data is reduced by 4 downsampling convolutions. (2) Compress the channel after two 1×1 convolutions with the same size. (3) After two layers of fully connected networks, a discrimination result (0 or 1) is generated.

[ ]:
```
def discriminator_model():#Build discriminator model
    """Build discriminator architecture."""
    n_layers, use_sigmoid = 3, False
    inputs = Input(shape=input_shape_discriminator)
    #Downsampling convolution
    x = Conv2D(filters=ndf, kernel_size=(4, 4), strides=2,␣
↪padding='same')(inputs)
    x = LeakyReLU(0.2)(x)

#Continue 3 times in downsampling convolutions
    nf_mult, nf_mult_prev = 1, 1
    for n in range(n_layers):
        nf_mult_prev, nf_mult = nf_mult, min(2**n, 8)
        x = Conv2D(filters=ndf*nf_mult, kernel_size=(4, 4), strides=2,␣
↪padding='same')(x)
        x = BatchNormalization()(x)
        x = LeakyReLU(0.2)(x)
#Convolution operation with a step size of 1 without changing the size
```

```
    nf_mult_prev, nf_mult = nf_mult, min(2**n_layers, 8)
    x = Conv2D(filters=ndf*nf_mult, kernel_size=(4, 4), strides=1,␣
 ↪padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.2)(x)
#Convolution operation with a step size of 1, does not change the size.␣
 ↪Compress channel is 1
    x = Conv2D(filters=1, kernel_size=(4, 4), strides=1, padding='same')(x)
    if use_sigmoid:
        x = Activation('sigmoid')(x)
#Two layers are fully connected and output the discrimination result
    x = Flatten()(x)
    x = Dense(1024, activation='tanh')(x)
    x = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=inputs, outputs=x, name='Discriminator')
    return model
```

[ ]:
```
#Combine the discriminator model with the generator model to form the complete␣
 ↪structure of the DeblurGAN model
def generator_containing_discriminator(generator, discriminator):
    inputs = Input(shape=image_shape)
    generated_image = generator(inputs)  #Call generator
    outputs = discriminator(generated_image) # Call discriminator
    model = Model(inputs=inputs, outputs=outputs) #Construct a discriminator␣
 ↪model model using the Model class of the tf.keras interface.
    #When using model, you can set trainable parameters to control the internal␣
 ↪structure of the model
    return model

def generator_containing_discriminator_multiple_outputs(generator,␣
 ↪discriminator):
    inputs = Input(shape=image_shape)
    generated_image = generator(inputs)
    outputs = discriminator(generated_image)
    model = Model(inputs=inputs, outputs=[generated_image, outputs])
    return model
```

4.3 4.3 Train procedure

4.3.1 4.3.1 The weight function

This part of the programming is about saving the result of the training as the weights matrix form. It is include the discriminator and generator and both of them is saved as the H5 form. Because of the limit of the memory, the whole data may not be processed at once. So the dataset is separated as the different file like the input file and the output is the weights matrix of the partical file.

```
[ ]: from keras.callbacks import TensorBoard
     from keras.optimizers import Adam
     import numpy as np
     import os
     import datetime
     import h5py
     import torch
     BASE_DIR = 'weights/'
     def save_all_weights(d, g, epoch_number, current_loss):
         now = datetime.datetime.now()
         save_dir = os.path.join(BASE_DIR, '{}{}'.format(now.month, now.day))
         if not os.path.exists(save_dir):
             os.makedirs(save_dir)
         g.save_weights(os.path.join(save_dir, 'generator_{}_{}.h5'.
      ↪format(epoch_number, current_loss)), True)
         d.save_weights(os.path.join(save_dir, 'discriminator_{}.h5'.
      ↪format(epoch_number)), True)
```

4.3.2 4.3.2 The training kernal

This is the main training algorithm, the basic idea is to combine the code and function together and make it running. Because the idea of some details about the kernals has been described in the function above, this part only focus on the operation to the dataset.There are three 'for' loop in the code, so there will be explaination for the three 'for' loop below. Outside the 'for' loop, there are many perpare, including the detail of the loss function. They are well talken above, so this part will skip that.

The first 'for' loop decide the deep of the neural network. The basic of the algorithm will be talken below in the second and third 'for' loop part.

The second 'for' loop is for getting the generator training. The generator can generate an image based on the model, then the image will be sent to the the third 'for' loop, which will be talken below. Then, the loss will be calculated and eventually be printed. This loop is taking every image of the dataset out and train individually.

The third 'for' loop is for calculating the loss function value and training for the discriminator. The basic idea is to calculate the average of the loss function for judging true and the loss function for judging for false. Then, put the loss function together as a prove of training. The function of this loop is for Runing a single gradient update on a single batch of data for multiply times, in order to upgrade the discriminator.

Eventually, the model is saved as matrix form in h5 form file.

```
[ ]: def train_multiple_outputs(dataA, dataB, batch_size, log_dir, epoch_num,␣
      ↪critic_updates=5):
         x_train, y_train = dataA, dataB
         g = generator_model()
         d = discriminator_model()
```

```
    d_on_g = generator_containing_discriminator_multiple_outputs(g, d)

    d_opt = Adam(lr=1E-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
    d_on_g_opt = Adam(lr=1E-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08)

    d.trainable = True
    d.compile(optimizer=d_opt, loss=wasserstein_loss)
    d.trainable = False
    loss = [perceptual_loss, wasserstein_loss]
    loss_weights = [100, 1]
    d_on_g.compile(optimizer=d_on_g_opt, loss=loss, loss_weights=loss_weights)
    d.trainable = True

    output_true_batch, output_false_batch = np.ones((batch_size, 1)), -np.
↪ones((batch_size, 1))

    log_path = './logs'
    tensorboard_callback = TensorBoard(log_path)

    for epoch in range(epoch_num):
        permutated_indexes = np.random.permutation(x_train.shape[0])

        d_losses = [] train
        d_on_g_losses = []
        for index in range(int(x_train.shape[0] / batch_size)):
            batch_indexes = permutated_indexes[index*batch_size:
↪(index+1)*batch_size]
            image_blur_batch = x_train[batch_indexes]
            image_full_batch = y_train[batch_indexes]

            generated_images = g.predict(x=image_blur_batch,␣
↪batch_size=batch_size)

            for _ in range(critic_updates):
                d_loss_real = d.train_on_batch(image_full_batch,␣
↪output_true_batch)
                d_loss_fake = d.train_on_batch(generated_images,␣
↪output_false_batch)
                d_loss = 0.5 * np.add(d_loss_fake, d_loss_real)
                d_losses.append(d_loss)

            d.trainable = False
            d_on_g_loss = d_on_g.train_on_batch(image_blur_batch,␣
↪[image_full_batch, output_true_batch])
            d_on_g_losses.append(d_on_g_loss)
            d.trainable = True
```

```
        print(np.mean(d_losses), np.mean(d_on_g_losses))
        with open('log.txt', 'a+') as f:
            f.write('{} - {} - {}\n'.format(epoch, np.mean(d_losses), np.
↪mean(d_on_g_losses)))

        save_all_weights(d, g, epoch, int(np.mean(d_on_g_losses)))
```

4.3.3 4.3.3 preparing the test data and generate the output image

The usage of the test data is like the train data, because the form of the training data and the test data is similar. So, like the stage above, the file is opened and get a couple of slide out. Then, adding the mask is taken place in the image. After, the image is cropped and resized before send to the generator.

This part of the code is using the h5 result to generator the image, which based on the unclear input. Because of the training model using the 256x256 pixel image, the input of test data must crop and resize the image to fit the kernal. The image is put as the form of the matrix. Finally, the matrix is processed by the generator model and the result is printed as image.

[ ]:
```
from functions import transforms as T
from functions.subsample import MaskFunc

def get_gt(volume_kspace):
    volume_kspace2 = T.to_tensor(volume_kspace)
    volume_image = T.ifft2(volume_kspace2)
    volume_image_abs = T.complex_abs(volume_image)
    return volume_image_abs
```

This function is desided to get the training data. Because the dataset are given as the h5 form and without dividing the training and testing dataset, it is required to process first to make the algorithm running easier. So, the whole dataset is delivered to the data process program. The idea of the data process is to pick several slices from a K space dataset, which is a rectangle shade photograph. This part of the data is set into the label B. The transform has applied inverse Fourier Transform to get the complex image. In order to produce the data with poor image quality as input of the model, some masks are created to block some part of the dataset, which cause the image from the picked process become poor quality. We created two mask function objects which is used to perform multiple operations per point between the original k-space data and the grayscale, and use the Fourier transform to generate under-sampled images. This part of data is set into the label A.

The acc decide whether the 4af or 8af data is inputing, which decide the chaos in the image.

[ ]:
```
def open_train_file(subject_path, acc=4, size=256):
    if (acc == 4):
        mask_func = MaskFunc(center_fractions=[0.08], accelerations=[
                             4])  # Create the 4 mask function object
```

```
    else:
        mask_func = MaskFunc(center_fractions=[0.04], accelerations=[
                             8])  # Create the 8 mask function object
    with h5py.File(subject_path, "a") as hf:
        volume_kspace = hf['kspace'][()]
        volume_kspace2 = T.to_tensor(volume_kspace)
        gt_data = T.center_crop(get_gt(volume_kspace), [size, size]).
↪unsqueeze(3).repeat(1,1,1,3)
        mask = mask_func(np.array(volume_kspace2).shape, seed=0)
        masked_kspace = torch.where(mask == 0, torch.Tensor([0]),␣
↪volume_kspace2)
        masked_image = T.center_crop(get_gt(masked_kspace), [size, size]).
↪unsqueeze(3).repeat(1,1,1,3)
    return masked_image, gt_data
```

The dataset is given as the form of the h5 file in the train folder. Because the dataset is huge and the memory is limit, the cropping and resizing process is hard to finish at once. In this case, instead of processing the data in the original training dataset, the dataset is divided according to the input h5 file. And then using 'for' loop to connect the matrix together.

[ ]:
```
#Prepare dataset
file_path = './train/'
ie = 0
da=torch.empty(0)
db=torch.empty(0)
for fname in sorted(os.listdir(file_path)):
    subject_path = os.path.join(file_path, fname)
    mask_data, gt = open_train_file(subject_path, 4)
    da=torch.cat((da,gt),0) # Combine gt
    db=torch.cat((db,mask_data),0) #Combine mask_data
dal=da.numpy() #trans o numpy
dbl=db.numpy() #trans o numpy
```

Start the training by calling the function. Set the result above as input.

[ ]:
```
#train start
train_multiple_outputs(dal,dbl,5,'./log',5)
```

4.3.4 4.3.4 generate the output image

This part of the code is using the h5 result to generator the image, which based on the unclear input. Because of the training model using the 256x256 pixel image, the input of test data must crop and resize the image to fit the kernal. The image is put as the form of the matrix. Finally, the matrix is processed by the generator model and the result is printed as image.

```
[ ]: from functions.utils import deprocess_image
def gen_image(weight_path, input_dir):
    f = h5py.File('result.h5','w')
    g = generator_model()
    g.load_weights(weight_path)
    ima=torch.empty(0)
    for file_name in os.listdir(input_dir):
        print(file_name)
        with h5py.File(input_dir+'/'+file_name, "a") as hf2:
            volume_kspace = hf2['kspace_4af'][()]
            volume_kspace2 = T.to_tensor(volume_kspace)
            masked_image = T.center_crop(get_gt(volume_kspace2), [256, 256]).
 ↪unsqueeze(3).repeat(1,1,1,3)
            print(masked_image.shape)
            generated_images = g.predict(x=masked_image.numpy())
            generated = np.array([deprocess_image(img) for img in␣
 ↪generated_images])
            im=T.to_tensor(generated)
            ima=torch.cat((ima,im),0)
            print(ima.shape)
    f['recon_4af']=np.array(ima.numpy())
    f.close()
```

This function of this part is output the the test image after training.

```
[ ]: gen_image('generator4af.h5','test')
```

5 Conclusion

In order to fast constructed image, the research process focus on using deblurgan model to reduce the deblur. First of all, it is likely that GAN model is meaningful for extracting key information. The generator product clears images and discriminator is created to Distinguish between real and clear images and fake or blurred images. In the reconstruction of medical images, GANs seem to provide good performance. Some operations are added to the loss function to highlight texture details and special features.

Secondly, it can be found in this research that the VGG perceptual loss function is beneficial for 4AF images because of good visual perception. However, 8AF images was greatly affected by noise, which it had a giant bias. Therefore, the perceptual function is replaced by the L1 function to process images. It is shown in this figure, the left one is the 4AF one, the middle one is the 8AF one and the right one is the groundtruth image.

Figure 4. The result of the test dataset

The result of our project is shown in this table,

MASK	SSIM Value
4AF	0.37
8AF	0.21

Ultimately, the construction of deep networks is expensive, the training period is long, and the impact of the loss function is great. Explaining the relationship between input features and their outputs is difficult for researchers.

In conclusion, this model still has great optimization prospects, but it is limited to machines, data sets, and time, and cannot be further optimized for the time being.

For solving the problem of MRI reconstruction, finding a fast, accurate and more stable framework is the future research direction for scientists.

6 Contribution

Person	Contribution
Huang, ZIJIAN	Reading the paper and materials about the DeblurGAN. Figuring out the whole process of the algorithm, especially the discriminator and the training procress.Writing the experiment part of the report.
Ma, Yuhao	Reading related papers and materials about the DeblurGAN.Establish a testing and training environment and debug parameters. Evaluate training and optimize parameters with Yewei Song together, providing some design ideas to him.

Person	Contribution
Zhang, Zongxu	Reading related papers and materials about the DeblurGAN.Analyzed the gan framework and why use it. Completed the code interpretation of the discriminator and generator, the characteristics of the loss function and the reasons for using them. Completed the essay on experiment part ,which included loading data, loss function generator model and discriminator model sections.
Wan, Zhenyu	Research papers on GAN, DeblurGAN and U-net in neural networks. Focus on reading relevant materials about DeblurGAN neural networks. Research dataset, machine learning task and aim of our project. Understand the structure of entire neural network especially generator, discriminator and loss function factors. Writing the introduction and design of report.
Tzeng, JIUN-RUNG	Writing the implementation part of the report, reading the essay and make a compare and contrast of DAGAN and DeblurGAN. Explain the reason how and why our team use DeblurGAN and RES-NET instead of U-NET
Song, YEWEI	Design the GAN network, Choose the RESNET base unit, writing model train and test code, Design data handling process, evaluating train results, optimism the param

6.1 REFERENCE

1. Orest K., Volodymyr B., Mykola M., Dmytro M., Jiˇri M.: DeblurGAN:BlindMotionDeblurringUsingConditionalAdversarialNetworks(2018). Available at: https://arxiv.org/pdf/1711.07064.pdf (Accessed at: 8 December 2019).
2. Z. Wang, E. P. S. a. A. C. B., 2003. Multiscale structural similarity for image quality assessment. [Online] Available at: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1292216&isnumber=28784 [Accessed 11 12 2019].
3. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, "Generative adversarial networks," Advances in Neural Information Processing Systems, Montreal, Canada, Dec. 2019.
4. Mardani, M., Gong, E., Cheng, J., Vasanawala, S., Zaharchuk, G., Alley, M., Thakur, N., Han, S., Dally, W., Pauly, J. and Xing, L. (2019). Deep Generative Adversarial Networks for Compressed Sensing Automates MRI. [online] arXiv.org. Available at: https://arxiv.org/abs/1706.00051 [Accessed 12 Dec. 2019].
5. Prasanth Omanakuttan: Understanding CNN,GAN,AE and VAE (no date). Available at: https://blog.goodaudience.com/understanding-cnn-gan-ae-and-vae-3fdfe857ab98 (Accessed at: 8 December 2019).