Neural Computation Source Code and Report

Report

Group Member Name: Zhangda Xu, Xiaoyu Xia, Lizhao Zhou, Zhun Wang, Wenzhou Liu

1. Introduction

1) fast MRI ¶

Magnetic resonance imaging (MRI) plays an important role in advanced medical test. However, the long acquisition time in MRI often exceeds half an hour which results in low patient throughput and high patient anxiety. Hence, a faster MRI process is required to reduce the examination cost and to provide a more comfortable experience for patients. People gradually pay more attention on safety problems caused by its long duration in MRI machines as well. Some acute diseases intrinsically requires the treatment measures to provide fast and valid judgment on the state of illness since long diagnosis time will deteriorate the patients' condition and lead to secondary injury.

MRI is an indirect process which produces cross-sectional images of the subject's anatomy by their frequency and phase measurements, whereas typical direct process involves spatially-related measurements. The measurements are regularly correlated to a multidimensional Fourier-space representation of an imaged body. This is known as k-space.

Faster MRI process can be achieved by scanning a undersampled part of the k-space. A 4-fold acceleration can reduce the scanning time significantly from over half an hour to only a few minutes. Comparing with classical methods, image reconstruction with neural network can yield better imaging results and handle image data more efficiently when larger image with higher resolution are processed, whereas the cost of classical inverse optimisation approach increases non-linearly. The spatially-resolved image m is then estimated from the undersampled k-space by performing an inverse multidimensional Fourier transform, where $\hat{m}$ is a noise-corrupted estimate of the true image m, P() is a mask matrix selection over full k-space y:

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

2) Datasets

The datasets used in the fast MRI coursework are provided in prior with baseline functions, which is extracted from the open fastMRI Dataset(http://fastmri.med.nyu.edu/ (http://fastmri.med.nyu.edu/)), including a training dataset and a test dataset. By inspecting the key values of the HDF5 files, we find out that the training data is the raw k-space image generated from thousands of MRI examination, which is used for both training and validation. Each data in the test set consists of a 4-fold and 8-fold undersampled k-space data with the masks that generate them, respectively.

3) task

Essentially, our learning task in the coursework is to develop a method to reconstruct accurate images from the masked k-space data with both undersampling rates in the test set. The results are to be measured by the similarity to the ground truth images generated by the original test data.

The main goal we want to achieve in the coursework is to propose a deep learning approach which is capable of reconstructing high accuracy images even from undersampled scan images with large acceleration rates. By training a fine-tuned U-net neural network, we obtain over 40% relative improvement of reconstructed image quality using 4-fold and 8-fold accelerated scan than using the masked data directly.

2. Design

1) neural network description and justification

The coursework is considered as an image reconstruction problem. The neural network is fed with samples of input images and their better constructed peer. It is expected that, by learning a deep structure of layers, the trained model learns to find the intrinsic phases and magnitude among local neighbourhood between the pairs of images so that it can be applied to the reconstruction of other undersampled input.

In addition, the fast MRI problem is defined by, albeit the structure similarity stands, a deeper connection that the k-space data that generates the input image is a spatial subset of the k-space data that generates the ground truth image. Due to the Fourier Transformation process, it gives them spatio-temporal dependencies to each other. In another word, an image reconstruction method should be able to 'peek' through fractions of waves on certain frequencies to generate the full image.

To develop a specific methodology on the problem, we first consider Convolutional Neural Network(CNN) for its capability of learning spatial properties and reducing imaging noise. U-net is later introduced to the team because its unique structure is good at partitioning different pixels. It is smartly built with blocks of convolutional layers in an autoencoder style. The team decide to build baseline models on CNN first for a better understanding of the problem, and go for U-net architecture later if it yields better performance.

1) naive CNN model

The convolution layer introduces a concept of local perception, which tells that in the process of the human brain recognizing a picture, the whole picture is not recognized at the same time, but each feature in the picture is first perceived locally, and then the local operation is comprehensively performed at a higher level to obtain global information.

To fit the image into the model, the input is expected to have 4 dimensions. The raw file data is first transformed and loaded as input in iteration by a data loader. The implementation process before training will be discussed later.

We then develop our CNN model with three 2D convolutional layers, the first two of them with a kernel size of 5. The number of output channels starts from 16, which is doubled in the second layer. The last convolutional layer is applied with a kernel size of 1 to reduce the channels of feature maps in a single perceptron.

Each convolutional layer is followed by an activation layer. The activation function makes a non-linear mapping of the output of the convolution layer. In this model, we choose the ReLU function to make every iteration faster. After activation, feature space is reduced by a pooling layer. The pooling layer is mainly used for feature dimension reduction, by compressing the number of data and parameters. It helps to reduce model overfitting, and improves the fault tolerance of the model. Here we choose a max pooling over a (2, 2) window.

Finally, the feature maps are converted to a 1-dimensional vector by up-sampling and the third convolution layer. Since the features are in 1 dimension now, we feed them to 3 fully connected layers to produce the final result. After several previous convolutions, activations, and pooling, the model will fully learn a high-quality feature picture fully connected layer.

CNN can directly process grayscale images and can be used directly to process image-based classification. CNN has unique advantages in image processing due to its special structure of local weight sharing. The layout is closer to the real biological neural network. Weight sharing reduces the complexity of the network, especially for multidimensional input vectors. The feature that allows images to be directly entered on the network avoids the complexity of rebuilding data when retrieving and classifying features. Maybe the performance of CNN used for MRI processing is not well.

2) U-Net

Then, we will train the neural network model by U-net based on CNN. When it comes to classification, the information provided by the pixels is always taken into account. However, this information generally includes two types: one is environmental field information, and the other is detailed information. The pixel-based approach has a great deal of uncertainty about the choice of the form. Choosing a size that is too large not only requires more pooling layers to make the environmental information appear, but also loses the local detailed information. But U-net uses a network structure that includes both down sampling and up sampling. Down sampling is used to gradually display the environmental information, and the process of up sampling is to combine the down sampling information of each layer and up sampling input information to restore the detailed information, and gradually restore the image accurately. Besides, U-Net combines the location information from the down sampling path to finally obtain a general information combining localization and context, which is necessary to predict a good segmentation map.

By changing our naive CNN structure a bit, we first construct the basic block used in the U-Net. This convolutional block is consisted of 2 convolutional layers, each of them operate a stride-2 convolution. Afterwards, each convolution layer is followed by an instance normalization, ReLU activation and dropout sequentially. The default kernel size for all the convolutional layers is set to be 3 * 3. By adding padding of size 1, we can keep the layers' shape unchanged as the input shape throughout the block. The number of channels after the first block can be tuned to fit different scenarios, which is suggested to be power of 2 starting from 32.

U-Net model is briefly composed of two parts : down sampling and up sampling. In the down sampling part, which is also called the feature extraction phase, U-Net starts with an input in a single channel. By changing the number of pooling layers, we can control how deep the model goes. By setting the channels after the first layer, the model can be either more robust or faster. It is implemented by default that the model goes a block deeper, the number of channels doubles at the down sampling stage so that the input and output channels of each layers fit with each other, whereas the pooling layer between blocks keeps halving the feature space. The baseline model is set to have 4 pooling layers and 32 channels after the first block. At the final step of feature extraction phase, the size of the feature map is compressed into a 20 x 20 space. At the same time, 256 channels of kernels are trained on the single layer. A 1 x 1 convolution is added to the end of down sampling stage to assemble the information in all channels.

Up sampling, which is also known as deconvolution, or transpose convolution, is a critical part of U-net. There is a method, known as bilinear interpolation, gets the feature map merged at the same scale as the number of channels corresponding to the feature extraction part. Therefore, before merging, the model will copy and paste the feature map in the shallow layers, which results in the final output is a region in the center of the input. After the first 3x3 convolution operation after each merging, the number of 3X3 convolution kernels is halved. Unlike down sampling, up sampling doubles the size of feature map. And then passing through ConvBlock. Finally, in the last layer, we use the kernel 1 * 1 to retain the shape of an output image to 320 x 320 as required. It is essential in the U-Net architecture that by adding back the under sampling layers, a proportion of shallow information is well kept even in the deepest layers. In its' application to fast MRI image reconstruction tasks, this structure enables the model to learn over a range of frequencies and combines them together. It helps to generalize the result and prevent overfitting, as well.

U-Net is derived from CNN. U-shaped structure is the biggest feature of U-Net. U-Net first performs down sampling and use the contact connection on the corresponding layer, which ensures that the reconstructed feature maps incorporate more low-level features. Up sampling layers also make the information such as edge restoration of the image more detailed.

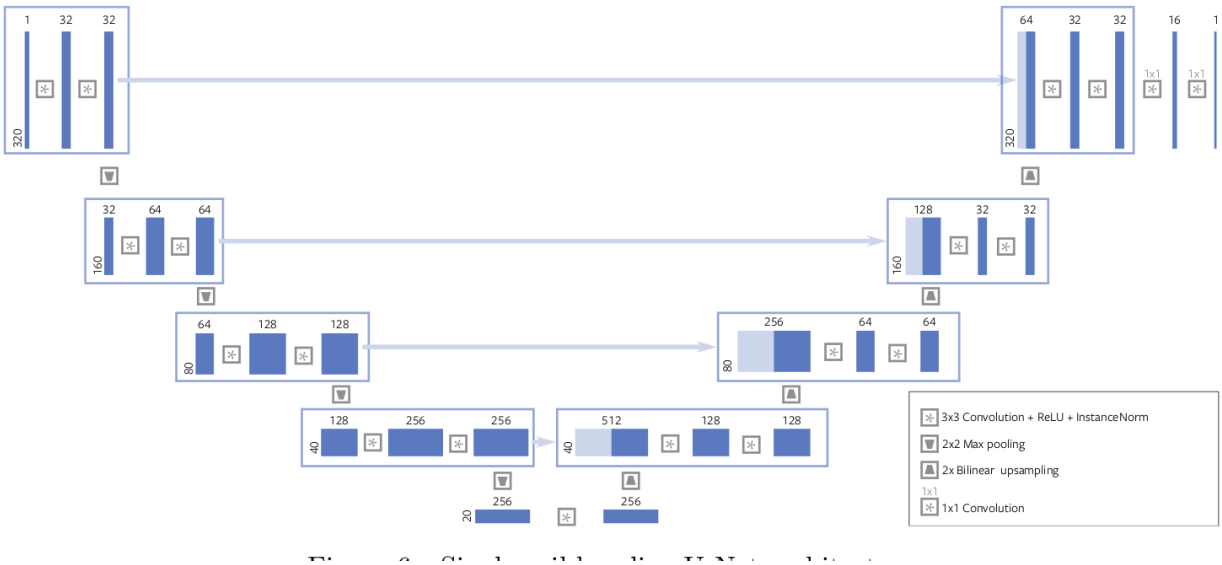

fig.1. U-Net structure

2) experiment factors

Hyperparameters in both baseline models are set according to the designed structure at the beginning. It it clear to see that U-Net architecture needs more experiment to optimise the performance because more hyperparameters are available to tune. First, the performance of two baselines should be compared. For both models, the optimisation function and loss function in the training session should be selected according to their performance on the dataset. In addition, the learning rate will be tested, starting from default value, to get a better convergence on models. Explicitly for the U-Net model, the number of output channels after first convolution block and the number of pooling layers need to be tuned in this case. Lastly, to improve our reconstruction and reduce noise and artifact in the image, a set of data preprocessing methods will be tested.

3. Implementation

For data loader, train images in k-space need to be read by h5py.File() to store in a h5py format. Show_slices() can visualize slices in the sample as k-space images. By using functions to_tensor(), ifft2() and complex_abs(), k-space data can be transformed to real images. As the whole dataset needs to be divided into two parts, train data and validation data to avoid overfitting, Load_data_path consisting of data_path_train and data_path_val aims at helping go through each subset and list all the file names, the file paths and the slices of subjects in the training and validation sets to generate a data list. A data list must include fname, raw data and slices. Then, raw data needs to be reshaped to 3 or more dimensions for further use and analysis. In MRIdataset, acceleration, center fraction and seed need to be defined for data processing which will be talked about in Experiment section.

Turn to train the model, I used the dataloader function from PyTorch. Because the size of the input and target data is the same, so the loss function we can choose could be l1 or l2(MSE). Firstly I choose l1 and SDG for the optimizer. I set the learning rate 0.001 after data processing like before and begin to train. Because of the limit of time and the GPU of my laptop, I just trained 2 epochs and save the model. However, I observe the loss has a remarkable fluctuation and keep unchanged in the end.

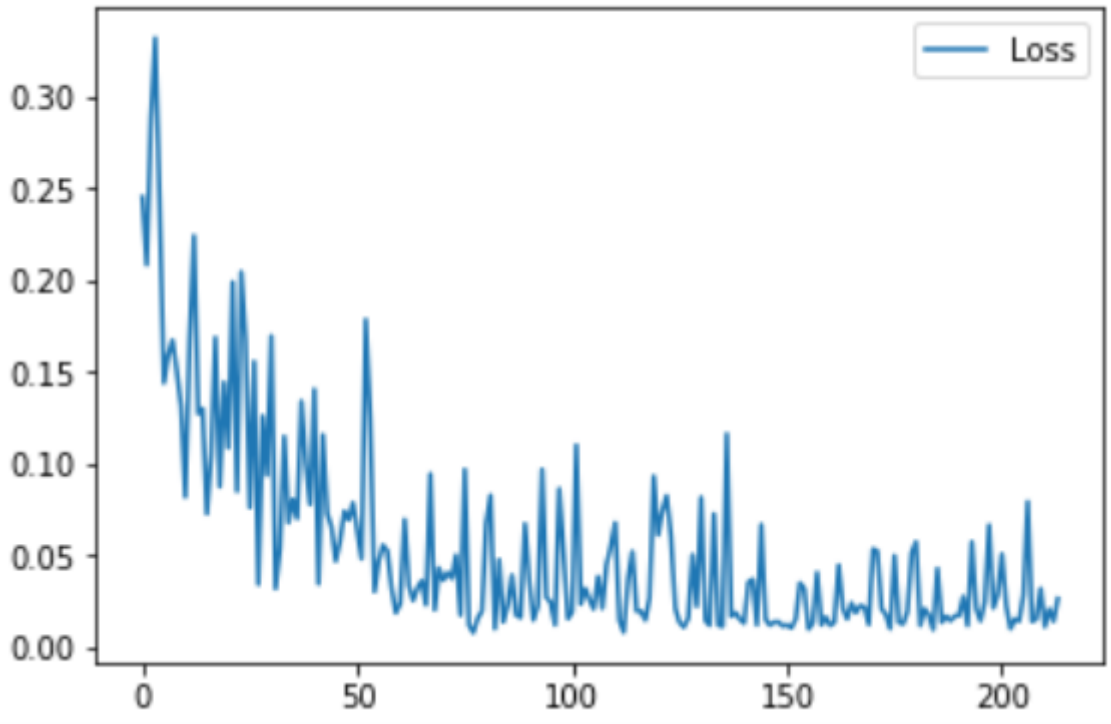

Fig.2. The reduction of loss curve

By testing the model, the performance of my model is not good. I input the testing data, set AF = 8 and use ssim function test, the average ssim is just 0.45 and the image is below. So I try to modify the network and adjust hyperparameter to optimize the model. Firstly, I change the loss function to MSE and reduce the learning rate to 0.0001. Besides them, I also add a convolution layer to extract 64 features and two 1 $/times$1 convolutions to reduce size progressively. After training 2 epochs and testing, I find the performance is still not good, but I do not have enough to continue to adjust the model, and we decide to use U-NET model. Fig.2. The left is the image with undersampling rate 8, the center is the target image and the right is the image after inputting in model.

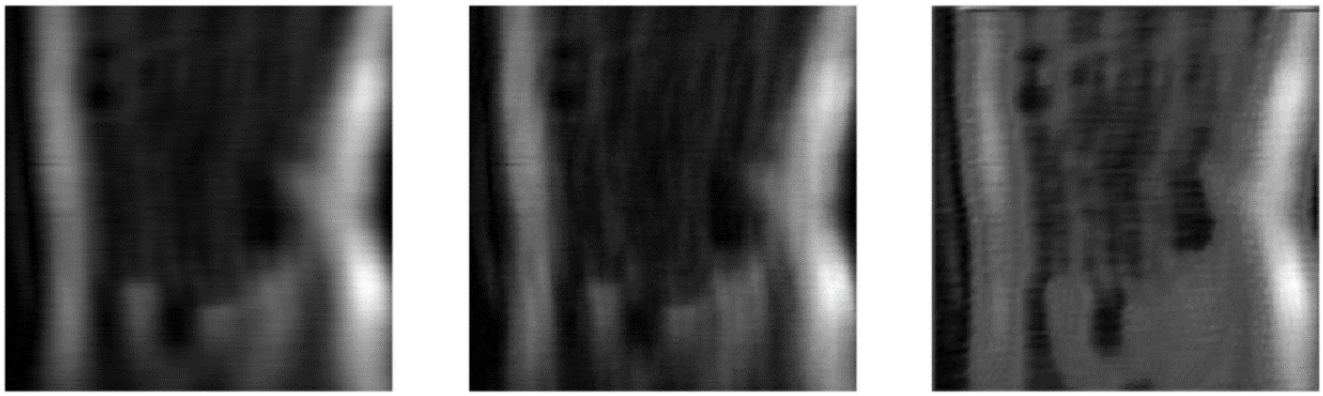

Fig.3. Imaging result of CNN

The U-NET model has been described in the design part. So, in this part, I decide show some details. The channels of input images and output images are set one, and the kernels' channels are 64 which means after we did the stride-2 convolutional operation, leading to 64 channels images. Because the value of drop probability is 0.5，each layer will randomly drop approximate 50 percent cells. Besides, experimenters set the value of number of pool layers is 3.Three pool layers means the ConvBlock will be operated three times in down sampling and twice in up sampling. In the training part, we choose the L1 loss function. In addition, we adopt the Stochastic Gradient Decent(SGD) optimizer which is the most basic optimization method and common training methods. Then, the value of learning rate we set is 0.01.Finally, we can train the data by forward propagation and back propagation.

2) Performance analysis mechanism

Since the unavailablity of the original data, we can validate our reconstruction method only on the training set. To evaluate the training session, an updated loss value list in the iteration of training is recorded, which is plotted to illustrate the process of training. By comparing the validation results in table, we can compare the performance of different models in the test.

After gaining the output images, we can transform them to real images to make a comparison. And the value range of SSIM is [0,1], which means the larger the value is, the better the performance is. Structural similarity (SSIM) is also a full-reference image quality evaluation index. It measures the similarity of two images from the aspects of brightness, contrast, and structure. The SSIM algorithm is designed to take into account the visual characteristics of the human eye and it is more in line with the human eye's visual perception than traditional methods. MSE or PSNR algorithms are both evaluations of absolute errors. For the fuzzy changes in the human's perception of the structural information of the image, the model also introduces some perception phenomena related to the changes in perception, including the brightness mask and the contrast mask. The structural information refers to the internal dependency between pixels, especially among pixels that are close in space. These dependencies carry important information on the target's visual perception.

4. Experiments

1) Description of experiments

The first experiment is to determine which baseline model we should use in the following development. By implementing both customised CNN model and the constructed U-net model to the training dataset, two models are trained over 20 epochs with learning rate of 0.001. The optimizer and loss function are set by default as Stochastic Gradient Descent(SGD) and L1-loss, respectively. To evaluate the baseline over a comparable condition, the number of output channels of the first convolution layer in U-Net is set to be 32. As shown in figure[4], when two models are trained for a limited epochs, the performance is similar. However, with the increase of total samples trained, the U-Net architecture promises a better convergence, thanks to its deeper layers. We then conclude that U-Net is a better choice for further development.

	loss	ssim
CNN	0.0284	0.338
U-NET	0.0941	**0.4053**

fig.4. Comparison of CNN and U-Net model performance

The next step is to select the loss function and the optimizer for U-Net in the training session. From previous experiences in image reconstruction, L1, Mean Squared Error (MSE) and Smooth L1 loss are listed as candidates for loss function, whereas SGD, Adam and RMSprop are selected for training optimizer. Then we diverge the training session on the U-Net into 9 pairs of combinations, respectively. Each session trains the model on the training set for 20 epochs with default learning rate set to optimizers. By comparing a performance grid of average SSIM and loss on the validation set in figure[5], it shows that choosing RMSprop at learning rate of 0.001 as the optimizer yields best performance on minimizing the loss function of MSE. However, all the models differ in their training time and overfitting issues. Nevertheless, we decide to choose MSE loss with RMSprop optimizer and to improve their performance in the following experiments.

	SGD	RMSprop	Adam
MSE	Avg loss: 0.1019 Avg SSIM: 0.3615	**Avg loss: 0.0775** **Avg SSIM: 0.4488**	Avg loss: 0.0791 Avg SSIM: 0.4362
L1	Avg loss: 0.0993 Avg SSIM: 0.3788	Avg loss: 0.0857 Avg SSIM: 0.4182	Avg loss: 0.0832 Avg SSIM: 0.4262
Smooth L1	Avg loss: 0.1023 Avg SSIM: 0.3574	Avg loss: 0.0064 Avg SSIM:0.4281	Avg loss: 0.0814 Avg SSIM: 0.4317

fig.5. Evaluation over different optimizer and loss function

To address the problems of unsuited learning rate, a test of varying learning rate for RMSprop is set over controlled conditions. We choose learning rate of 0.01, 0.001, 0.0003 and 0.0001. Four trained models' performance in training session are compared. As shown in the plot, the model trained with learning rate of 0.001 performs best on convergence.

Inferencing from 3 experiments above, we set MSE loss as the loss function, RMSprop as the optimizer and learning rate at 0.001 in U-Net model training. Although all the hyperparameters are optimised, there is an unneglectable difference in the trend of average loss and average SSIM metric of each epoch, where indicates overfitting. Therefore, the preprocessing methods are modified to test if it could be mitigated by different normalization. Normalizing the input images can also improve the reconstruction accuracy by reducing the noise and artifect acquired in MR. We compare the performance of max-min normalization and z-value normalization on the input images in figure[6].

	4af	8af
max-min	Avg loss: 0.0072 Avg SSIM: 0.5211	Avg loss: 0.0133 Avg SSIM: 0.4155
standard	Avg loss: 0.0519 Avg SSIM: 0.5319	Avg loss: 0.0893 Avg SSIM: 0.4308

fig.6. Performance over two methods of normalization

After choose the best hyperparameters, we train the fine-tuned model on the training set and measure their performance on the validation set. To best out the performance, we increase the number of channels in the model to 64 and 128 for seperate results. The performance can be improved moderately by increasing the number of channels in each layer. The best reconstruction performance for 4-fold and 8-fold undersampled input images are xxx and yyy, respectively.

The training set provided with the coursework is divided into training dataset and validation dataset with a 80% to 20% ratio. In every validation, the model is set to evaluation mode to make sure that no training is made on validation data. After the model is optimised, we feed the test datasets to reconstruct the images from undersampled k-space data.

2) Discussion

The performance of our model is discussed and re-evaluated by the team. Summarising the experiments, we find the final result is not good enough. The reconstruction is not as accurate as the masked k-space data. We find out that it is because the mismatch of value range of input, output and ground truth images. In the preprocessing part, the input and target images are first normalized by the max-min method, which clamp all of the data from [-255, 255] to the range of [-1,1]. Normalization before training is encouraged because due to the number of parameters computed in the neural network, unnormalized data could make the model very unstable and cost a large amount of time to train.

However, in the architecture of our model, no activation function is used as the output layer to remap all the value on the output data to the [-1,1] interval, but return random large numbers. In the attempt to fix the data preprocessing, the instant-wise remapping method is applied to the output, which is not effective to deal with large data images.

Due to the constraint of time for this coursework, this model is not improved further for a better performance. Several potential solutions are discussed. The best one is to apply a max-min normalization on both input and ground truth images, clamping the data into range [-1, 1]. The output is then activated by the last layer to be mapped from random large numbers to the [-1, 1] interval. Before measuring similarity and applying the model, the normalization of output data images should be inversed back to the original value range. To clarify, not normalizing input and output image will not cause the mismatch of value range, but it is not practical, since the training will be both unstable and very costly.

5. Conclusions

In this paper, we have implemented a CNN-based and U-Net-based deep learning method on undersampled k-space data to reconstruct real images. Various deep-learning techniques based on CNN have been proposed to address the problem of reconstructing MRI from undersampled k-space data. Therefore, we tried CNN as one of our solution to reconstruct the images. However, in comparison to U-Net model, CNN obtain relatively low performance and quality of reconstructed images. For the results acquired from U-Net model, we conclude that the models with very large capacities trained on large amounts of data can enable high acceleration factors and a robust U-Net model with more channels and pooling layers shows better performance.

Although the final performance of the model is not very satisfying, we propose a method which can potentially fix the neural network by applying a max-min normalization to the input and target images and adding an activation layer to the last layer in neural network. With no mismatch on value ranges, our model promises a stronger performance on fast MRI image reconstruction.

6. Description of Contribution

1. Introduction [10%] (Xia, Xu)

2. Design [15%] (Zhou)

3. Implementation [20%] (Wang, Liu, Zhou)

4. Experiments [45%] (Xu, Liu, Zhou)

5. Conclusions [10%] (Xu, Xia)

6. Coding (Xu, Liu)

7. Literature reference(Xia)

References

[1] A. H. Baykan, S. Sahin, I. Inan, S. Kafadar, and S. M. Erturk. RapidSequence MRI Analysis of Acute Abdominal Pain.ResearchGate, 92019.

[2] K. G. Hollingsworth. Reducing acquisition time in clinical MRI bydata undersampling and compressed sensing reconstruction.Physicsin Medicine and Biology, 60(21):R297–R322, oct 2015.

[3] B. J. Kim, H. G. Kang, H.-J. Kim, S.-H. Ahn, N. Y. Kim, S. Warach,and D.-W. Kang. Magnetic Resonance Imaging in Acute IschemicStroke Treatment.J. Stroke, 16(3):131, 9 2014.

[4] J. Liu and D. Saloner. Accelerated MRI with CIRcular CartesianUnderSampling (CIRCUS): a variable density Cartesian samplingstrategy for compressed sensing and parallel imaging.Quant. ImagingMed. Surg., 4(1):57–67, 2 2014.

[5] A. S. Lundervold and A. Lundervold. An overview of deep learningin medical imaging focusing on MRI.Z. Med. Phys., 29(2):102–127,5 2019.

[6] K. Nael and W. Kubal. Magnetic Resonance Imaging of Acute Stroke.Magn. Reson. Imaging Clin. N. Am., 24(2):293–304, 5 2016.

[7] I. Oksuz, B. Ruijsink, E. Puyol-Anton, A. Bustin, G. Cruz, C. Prieto,D. Rueckert, J. A. Schnabel, and A. P. King. Deep Learning using K-space Based Data Augmentation for Automated Cardiac MR MotionArtefact Detection.arXiv, 8 2018.

[8] P. Putzky and M. Welling. Invert to Learn to Invert.arXiv, 11 2019.

[9] M. E. Ryan, A. Jaju, J. D. Ciolino, and T. Alden.Rapid MRIevaluation of acute intracranial hemorrhage in pediatric head trauma.Neuroradiology, 58(8):793–799, 8 201

Link

https://1drv.ms/u/s!AoY13IiLu1kIg79GbFyNF9U-BKlRfw?e=h98XkU (https://1drv.ms/u/s!AoY13IiLu1kIg79GbFyNF9U-BKlRfw?e=h98XkU)

In [ ]:

```
import h5py, os

import numpy as np
from scipy.io import loadmat
from matplotlib import pyplot as plt
from skimage.measure import compare_ssim

import torch
from torch import nn
from torch import optim
from torch.nn import functional as F
from torch.utils.data import DataLoader
Transforms
def tensor_to_complex_np(data):

    data = data.numpy()
    return data[..., 0] + 1j * data[..., 1]


def to_tensor(data):

    if np.iscomplexobj(data):
        data = np.stack((data.real, data.imag), axis=-1)
    return torch.from_numpy(data)


def apply_mask(data, mask_func, seed=None):

    shape = np.array(data.shape)
    shape[:-3] = 1
    mask = mask_func(shape, seed)
    return torch.where(mask == 0, torch.Tensor([0]), data), mask


def fft2(data):

    assert data.size(-1) == 2
    data = ifftshift(data, dim=(-3, -2))
    data = torch.fft(data, 2, normalized=True)
    data = fftshift(data, dim=(-3, -2))
    return data


def ifft2(data):

    assert data.size(-1) == 2
    data = ifftshift(data, dim=(-3, -2))
    data = torch.ifft(data, 2, normalized=True)
    data = fftshift(data, dim=(-3, -2))
    return data


def complex_abs(data):

    assert data.size(-1) == 2
    return (data ** 2).sum(dim=-1).sqrt()


def root_sum_of_squares(data, dim=0):
```

```
    return torch.sqrt((data ** 2).sum(dim))


def center_crop(data, shape):

    assert 0 < shape[0] <= data.shape[-2]
    assert 0 < shape[1] <= data.shape[-1]
    w_from = (data.shape[-2] - shape[0]) // 2
    h_from = (data.shape[-1] - shape[1]) // 2
    w_to = w_from + shape[0]
    h_to = h_from + shape[1]
    return data[..., w_from:w_to, h_from:h_to]


def complex_center_crop(data, shape):

    assert 0 < shape[0] <= data.shape[-3]
    assert 0 < shape[1] <= data.shape[-2]
    w_from = (data.shape[-3] - shape[0]) // 2
    h_from = (data.shape[-2] - shape[1]) // 2
    w_to = w_from + shape[0]
    h_to = h_from + shape[1]
    return data[..., w_from:w_to, h_from:h_to, :]


def normalize(data, mean, stddev, eps=0.):

    return (data - mean) / (stddev + eps)


def normalize_instance(data, eps=0.):

    mean = data.mean()
    std = data.std()
    return normalize(data, mean, std, eps), mean, std


def roll(x, shift, dim):

    if isinstance(shift, (tuple, list)):
        assert len(shift) == len(dim)
        for s, d in zip(shift, dim):
            x = roll(x, s, d)
        return x
    shift = shift % x.size(dim)
    if shift == 0:
        return x
    left = x.narrow(dim, 0, x.size(dim) - shift)
    right = x.narrow(dim, x.size(dim) - shift, shift)
    return torch.cat((right, left), dim=dim)


def fftshift(x, dim=None):

    if dim is None:
        dim = tuple(range(x.dim()))
        shift = [dim // 2 for dim in x.shape]
    elif isinstance(dim, int):
        shift = x.shape[dim] // 2
    else:
```

```
        shift = [x.shape[i] // 2 for i in dim]
    return roll(x, shift, dim)


def ifftshift(x, dim=None):

    if dim is None:
        dim = tuple(range(x.dim()))
        shift = [(dim + 1) // 2 for dim in x.shape]
    elif isinstance(dim, int):
        shift = (x.shape[dim] + 1) // 2
    else:
        shift = [(x.shape[i] + 1) // 2 for i in dim]
    return roll(x, shift, dim)
Subsample mask
class MaskFunc:

    def __init__(self, center_fractions, accelerations):

        if len(center_fractions) != len(accelerations):
            raise ValueError('Number of center fractions should match number of
 accelerations')

        self.center_fractions = center_fractions
        self.accelerations = accelerations
        self.rng = np.random.RandomState()

    def __call__(self, shape, seed=None):

        if len(shape) < 3:
            raise ValueError('Shape should have 3 or more dimensions')

        self.rng.seed(seed)
        num_cols = shape[-2]

        choice = self.rng.randint(0, len(self.accelerations))
        center_fraction = self.center_fractions[choice]
        acceleration = self.accelerations[choice]

        num_low_freqs = int(round(num_cols * center_fraction))
        prob = (num_cols / acceleration - num_low_freqs) / (num_cols - num_low_f
reqs)
        mask = self.rng.uniform(size=num_cols) < prob
        pad = (num_cols - num_low_freqs + 1) // 2
        mask[pad:pad + num_low_freqs] = True

        mask_shape = [1 for _ in shape]
        mask_shape[-2] = num_cols
        mask = torch.from_numpy(mask.reshape(*mask_shape).astype(np.float32))

        return mask
Data loader
def load_data_path(train_data_path, val_data_path):

    data_list = {}
    train_and_val = ['train', 'val']
    data_path = [train_data_path, val_data_path]

    for i in range(len(data_path)):
        data_list[train_and_val[i]] = []
        which_data_path = data_path[i]
```

```
        for fname in sorted(os.listdir(which_data_path)):
            subject_data_path = os.path.join(which_data_path, fname)
            if not os.path.isfile(subject_data_path): continue

            with h5py.File(subject_data_path, 'r') as data:
                num_slice = data['kspace'].shape[0]

            data_list[train_and_val[i]] += [(fname, subject_data_path, slice) fo
r slice in range(5, num_slice)]

    return data_list
class MRIDataset(DataLoader):

    def __init__(self, data_list, acceleration, center_fraction, use_seed):
        self.data_list = data_list
        self.acceleration = acceleration
        self.center_fraction = center_fraction
        self.use_seed = use_seed

    def __len__(self):
        return len(self.data_list)

    def __getitem__(self, idx):
        subject_id = self.data_list[idx]

        return get_epoch_batch(subject_id, self.acceleration, self.center_fracti
on, self.use_seed)
Preprocessing
def get_epoch_batch(subject_id, acc, center_fract, use_seed=True):

    fname, rawdata_path, slice = subject_id

    with h5py.File(rawdata_path, 'r') as data:
        rawdata = data['kspace'][slice]

    slice_kspace = to_tensor(rawdata).unsqueeze(0)
    S, Ny, Nx, ps = slice_kspace.shape
    shape = np.array(slice_kspace.shape)

    mask_func = MaskFunc(center_fractions=[center_fract], accelerations=[acc])
    seed = None if not use_seed else tuple(map(ord, fname))
    mask = mask_func(shape, seed)

    masked_kspace = torch.where(mask == 0, torch.Tensor([0]), slice_kspace)
    masks = mask.repeat(S, Ny, 1, ps)

    img_und = ifft2(masked_kspace)
    img_und = complex_center_crop(img_und, [320, 320])
    img_und = complex_abs(img_und)
    norm = img_und.max()

    img_und, mean, std = normalize_instance(img_und, eps=1e-11)
    img_und = img_und.clamp(-6, 6)

    img_gt = ifft2(slice_kspace)
    img_gt = complex_center_crop(img_gt, [320, 320])
    img_gt = complex_abs(img_gt)
    img_gt = normalize(img_gt, mean, std, eps=1e-11)
    img_gt = img_gt.clamp(-6, 6)
```

```
    return img_gt, img_und, masked_kspace.squeeze(0), masks.squeeze(0), mean, st
d, norm
Metrics
def ssim(gt, pred):

    return compare_ssim(gt.transpose(1, 2, 0), pred.transpose(1, 2, 0),
                        multichannel=True, data_range=gt.max())
Reconstruction
def reconstructions(test_loader, model_log):
    model.load_state_dict(torch.load(root + model_log))
    for iter, fname in enumerate(test_loader):
        output = model(images)
def save_reconstructions(reconstructions, out_dir):

    for fname, recons in reconstructions.items():
        subject_path = os.path.join(out_dir, fname)
        print(subject_path)
        with h5py.File(subject_path, 'w') as f:
            f.create_dataset('reconstruction', data=recons)
def show_slices(data, slice_nums, cmap=None):
    fig = plt.figure(figsize=(15,10))
    for i, num in enumerate(slice_nums):
        plt.subplot(1, len(slice_nums), i + 1)
        plt.imshow(data[num], cmap=cmap)
        plt.axis('off')
U-net model
class ConvBlock(nn.Module):


    def __init__(self, in_chans, out_chans, drop_prob):

        super().__init__()

        self.in_chans = in_chans
        self.out_chans = out_chans
        self.drop_prob = drop_prob

        self.layers = nn.Sequential(
            nn.Conv2d(in_chans, out_chans, kernel_size=3, padding=1),
            nn.InstanceNorm2d(out_chans),
            nn.ReLU(),
            nn.Dropout2d(drop_prob),
            nn.Conv2d(out_chans, out_chans, kernel_size=3, padding=1),
            nn.InstanceNorm2d(out_chans),
            nn.ReLU(),
            nn.Dropout2d(drop_prob)
        )

    def forward(self, input):

        return self.layers(input)

    def __repr__(self):
        return f'ConvBlock(in_chans={self.in_chans}, out_chans={self.out_chans},
' \
            f'drop_prob={self.drop_prob})'


class UnetModel(nn.Module):


```

```
    def __init__(self, in_chans, out_chans, chans, num_pool_layers, drop_prob):

        super().__init__()

        self.in_chans = in_chans
        self.out_chans = out_chans
        self.chans = chans
        self.num_pool_layers = num_pool_layers
        self.drop_prob = drop_prob

        self.down_sample_layers = nn.ModuleList([ConvBlock(in_chans, chans, drop
_prob)])
        ch = chans
        for i in range(num_pool_layers - 1):
            self.down_sample_layers += [ConvBlock(ch, ch * 2, drop_prob)]
            ch *= 2
        self.conv = ConvBlock(ch, ch, drop_prob)

        self.up_sample_layers = nn.ModuleList()
        for i in range(num_pool_layers - 1):
            self.up_sample_layers += [ConvBlock(ch * 2, ch // 2, drop_prob)]
            ch //= 2
        self.up_sample_layers += [ConvBlock(ch * 2, ch, drop_prob)]
        self.conv2 = nn.Sequential(
            nn.Conv2d(ch, ch // 2, kernel_size=1),
            nn.Conv2d(ch // 2, out_chans, kernel_size=1),
            nn.Conv2d(out_chans, out_chans, kernel_size=1),
        )

    def forward(self, input):

        stack = []
        output = input
        # Apply down-sampling layers
        for layer in self.down_sample_layers:
            output = layer(output)
            stack.append(output)
            output = F.max_pool2d(output, kernel_size=2)

        output = self.conv(output)

        # Apply up-sampling layers
        for layer in self.up_sample_layers:
            output = F.interpolate(output, scale_factor=2, mode='bilinear', alig
n_corners=False)
            output = torch.cat([output, stack.pop()], dim=1)
            output = layer(output)
        return self.conv2(output)
Training and validation
def train(num_epochs, interval, model_save, model_log):

    if not model_log is None:
        model.load_state_dict(torch.load(root + model_log))

    for epoch in range(1, num_epochs + 1):
        model.train()

        avg_loss = 0
        train_loss = 0
        iter_loss = []
```

```
        for iteration, sample in enumerate(train_loader):
            img_gt, img_und, masked_kspace, masks, mean, std, norm = sample

            X = img_und
            Y = img_gt
            X, Y = X.to(device), Y.to(device)

            output = model(X)
            loss = F.mse_loss(output, Y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss = 0.99 * train_loss + 0.01 * loss.item() if iteration > 0
else loss.item()
            iter_loss.append(train_loss)

            if iteration % interval == 0:
                print(epoch, iteration, train_loss)

        avg_loss = np.mean(iter_loss)
        print('\nTrain Epoch: {} Average training loss: {:.4f} \n'.format(epoch,
avg_loss))
        torch.save(model.state_dict(), root + model_save)


def val(model_log):

    model.load_state_dict(torch.load(root + model_log))
    model.eval()
    model.to(device)
    val_loss = []
    avg_loss = 0
    val_ssim = 0

    with torch.no_grad():
        for sample in val_loader:
            img_gt, img_und, masked_kspace, masks, mean, std, norm = sample

            show_slices(img_und.squeeze(0),[0])

            X = img_und
            Y = img_gt
            X, Y = X.to(device), Y.to(device)

            result = model(X)
            mean = mean.unsqueeze(1).unsqueeze(2).to(device)
            std = std.unsqueeze(1).unsqueeze(2).to(device)
            Y = Y * std + mean
            result = result * std + mean

            norm = norm.unsqueeze(1).unsqueeze(2).to(device)
            loss = mse_loss(result / norm, Y / norm, size_average=False)
            val_loss.append(loss.item())

            result, Y = result.cpu(), Y.cpu()
            val_ssim += ssim(Y.squeeze(0).numpy(), result.squeeze(0).numpy())

    avg_loss = np.mean(val_loss)
    val_ssim /= len(val_dataset)
    print('\nAverage loss: {:.4f} \nAverage SSIM: {:.4f}'.format(avg_loss, val_s
```

```
sim))
if __name__ == '__main__':

    root = '/home/kevinxu/NC2019MRI/fastMRI/models/'
    data_path_train = '/home/kevinxu/Documents/NC2019MRI/train/'
    data_path_val = '/home/kevinxu/Documents/NC2019MRI/train/'
    data_list = load_data_path(data_path_train, data_path_val)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    acc = 8
    cen_fract = 0.04
    seed = False
    num_batch = 1
    num_workers = 12

    train_dataset = MRIDataset(data_list['train'], acceleration=acc,
                    center_fraction=cen_fract, use_seed=seed)
    train_loader = DataLoader(train_dataset, shuffle=True,
                    batch_size=num_batch, num_workers=num_workers)

    val_dataset = MRIDataset(data_list['val'], acceleration=acc,
                    center_fraction=cen_fract, use_seed=seed)
    val_loader = DataLoader(val_dataset, shuffle=True,
                    batch_size=num_batch, num_workers=num_workers)

    model = UnetModel(in_chans=1, out_chans=1, chans=32, # 64, 128
                        num_pool_layers=4, drop_prob=0.5).to(device)

    optimizer = optim.RMSprop(params=model.parameters(), lr=0.001)



%matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt


    def plot_curve(train_log, val_log, interval):

    train_df, val_df = pd.read_csv(train_log), pd.read_csv(val_log)
    train_loss, val_loss = train_df[train_df.columns[2]], val_df[val_df.columns[
2]]
    train_loss_interval, val_loss_interval = train_loss.iloc[::interval], val_lo
ss.iloc[::interval]
    plt.title('Learning Curve')
    plt.plot(train_loss_interval, 'b')
    plt.plot(val_loss_interval, 'r')
    plt.show()
```

In [ ]:

```
# CNN
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1,
                            out_channels=16,
                            kernel_size=3,
                            stride=2,
                            padding=1),
            torch.nn.BatchNorm2d(16),
            torch.nn.ReLU()
        )
        self.conv2 = torch.nn.Sequential(
            torch.nn.Conv2d(16,32,3,1,1),
            torch.nn.BatchNorm2d(32),
            torch.nn.ReLU()
        )
        self.conv3 = torch.nn.Sequential(
            torch.nn.Conv2d(32,64,3,1,1),
            torch.nn.BatchNorm2d(64),
            torch.nn.ReLU()
        )
        self.conv4 = nn.Conv2d(in_channels=64, out_channels=32, kernel_size=1,st
ride = 1)
        self.conv5 = nn.Conv2d(in_channels=32, out_channels=16, kernel_size=1,st
ride = 1)
        self.conv6 = nn.Conv2d(in_channels=16, out_channels=1, kernel_size=1,str
ide = 1)
        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear',align_corner
s=False)
#         self.fc1 = nn.Linear(1*32*80*80, 120)
#         self.fc2 = nn.Linear(in_features=120, out_features=84)
#         self.fc3 = nn.Linear(in_features=84, out_features=10)
    def forward(self, x):
        # Max pooling over a (2, 2) window
       # x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
      #  print(x.shape)
        # If the size is a square you can only specify a single number
     #   x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = self.conv1(x)
        #print(x.shape)
        x = self.conv2(x)
       # print(x.shape)
        x = self.conv3(x)

        x = self.upsample(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.conv6(x)
#         x = F.relu(self.fc1(x))
#         x = F.relu(self.fc2(x))
#         x = self.fc3(x)
        return x
    def num_flat_features(self, x):
        size = x.size()[1:]     # all dimensions except the batch dimension
        num_features = 1
```

```
        for s in size:
            num_features *= s
        return num_features
model = Net()
#print(model)
# Try a random 32x32 input
input = torch.randn(1, 1, 320, 320)
out = model(input)
print(out.shape)
# loss_func = nn.CrossEntropyLoss()
# opt = torch.optim.Adam(model.parameters(),lr=0.001)
loss_func = nn.MSELoss()
opt = torch.optim.SGD(model.parameters(), lr=0.0001)
```

In [ ]:

```
```