Report for the exam of Neural Networks
Pyramid convolutional neural network for MRI reconstruction

Alessio Devoto 1701081

November 3, 2020

Contents

1 Introduction

Magnetic Resonance Imaging (MRI) as a non-invasive approach has many advantages over other imaging techniques. However, due to the physic limitations of MRI, the data acquisition is inherently slow. One common approach to accelerate MRI data acquisition is to take fewer measurements, generating an undersampled k-space. Recovering the image from the heavily undersampled k-space data is challenging and has been an active research field. Compressed sensing (CS) has been accepted for MR image reconstruction in current clinical practice. However, CS often requires the use of optimization algorithms which are iterative by nature, thus taking a relatively long computation time, which hinders or even prohibits MRI in certain clinical applications. In addition, CS based methods generally include some hyper-parameters and improper values of those hyper-parameters usually result in over-smoothed images or artifacts. Therefore, it takes great efforts to manually tune those hyper-parameters in real practice for CS based methods.

To avoid above drawbacks of CS, deep learning based methods have been proposed for MRI reconstruction in recent years. However, it is a great challenge for current deep learning

methods to recover the high frequency signals (i.e., fine details) from undersampled data, especially with a high acceleration rate. Nevertheless, the details in the reconstructed image are crucial for clinical diagnosis.

2 The problem

Mathematically, the data acquisition process of MRI can be formulated as follows:

$$y = Ax + \epsilon \tag{1}$$

where $x \in C^M$ is the image we want to reconstruct, $y \in C^N$ is the observed k-space, and ϵ is the noise. Both x and y are data from one coil and represented in vector form. A is the forward operator and often includes the multiplication of the Fourier transform matrix F , the binary undersampling matrix D and coil sensitivity matrix S. We don't use coil sensitivity maps and ignore S in the following. F is a factor in the forward matrix A because the k-space is a representation of the MR signal in the spatial frequency domain. Hence, to transform the signal into a real image (and vice versa), we use the Fourier transform. In particular, the Fast Fourier Transform (FFT) and inverse Fast Fourier Transform (IFFT) are applied to "move between" the two domains (Figure 1 for an example of k-space transformation). The binary undersampling D can be seen as a mask, with 0 and 1 values, corresponding to hidden/not hidden features. **The goal of MRI image reconstruction is to estimate image x from observed k-space y**. This can be preliminarily solved as an optimization problem:

$$\underset{x}{\operatorname{argmin}} \frac{1}{2}\|y - Ax\|_2^2 + \lambda R \tag{2}$$

where $R(x)$ is the regularization term and λ is the weight.

Many optimization algorithms can be used to minimize the objective function in Eqn. 2. As long as R(x) is differential, we can solve Eqn. 2 using gradient descent in an iterative fashion. At each step, the following operation is performed:

$$x^{k+1} = x^k + \alpha[AT(y - Ax^k) + \nabla R(x^k)] \; for \; k = 0, 1, ..., K \tag{3}$$

where k is the index for iteration and α is the learning rate.

The above iterative optimization process can be modeled by a Convolutional Recurrent Neural Network (ConvRNN) model.

$$\hat{x} = ConvRNN(\tilde{x}, y, D, \theta) \tag{4}$$

where $\tilde{x}$ is the input image (undersampled image), y is the undersampled k-space, D is the sampling mask, and θ are the parameters of the ConvRNN model.

3 The dataset: fastMRI

In order to train and evaluate the network, we used the FastMRI dataset provided by Facebook AI Research (FAIR) and NYU Langone Health. The fastMRI dataset contains data from MRI acquisitions of knees and brains. In particular, the following acquisitions are of interest for our study:

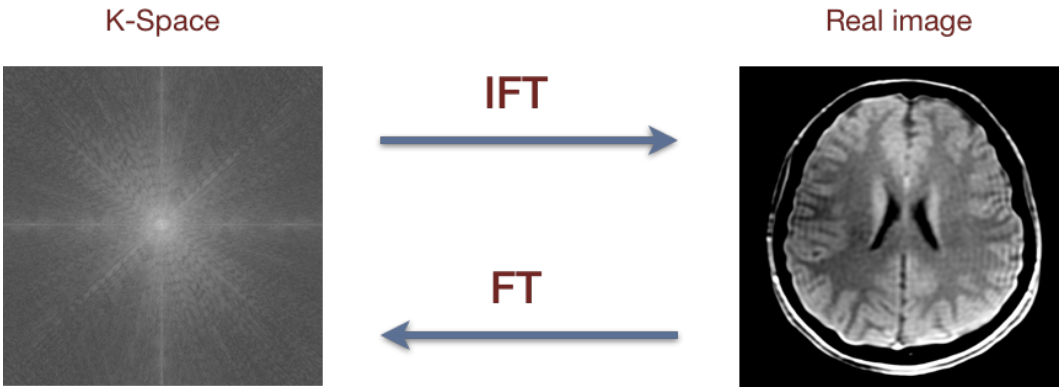

Figure 1: k-space and its corresponding real image

- **Raw multi-coil k-space data:** unprocessed complex-valued multi-coil MR measurements.
- **Emulated single-coil k-space data:** combined k-space data derived from multi-coil k-space data in such as way as to approximate single-coil acquisitions, for evaluation of single-coil reconstruction algorithms.
- **Ground-truth images:** real-valued images reconstructed from fully-sampled multi-coil acquisitions using the simple root-sum-of-squares method detailed below. These may be used as references to evaluate the quality of reconstructions.

This data was designed to enable two distinct types of tasks:

1. **Single-coil reconstruction task:** reconstruct images approximating the ground-truth from undersampled single-coil data.
2. **Multi-coil reconstruction task:** reconstruct images approximating the ground-truth from undersampled multi-coil data.

For each task, an official split into training and validation subsets is provided. It comprehends fully-sampled acquisitions, as well as test and challenge subsets which contain k-space data that have been subjected to undersampling masks. *Each MRI acquisition contains roughly 30 slices, corresponding to different depths in the 3D volume of the analyzed body region.*

Unfortunately, the size of the datasets that are used in the paper is out of reach for us. The smallest training set, i.e. the one for single-coil knee challenge, is roughly 90 GB. The multi-coil for both knee and brain is 1 TB. It has been clear from the beginning that without expensive and more powerful tools, the implementation of the project would have been impossible. In the light of the above, we decided to train the model solely for the **single-coil knee** challenge (paper also uses multi-coil knee) and on a reduced dataset of 20 GB, thus including "only" **6000 images** in the experiment, resulting from roughly 200 different MRI acquisitions. To further speed up the process, we had to crop the images in the data loading step. Instead of considering 320 x 320 cropped images (this was the authors' choice), we used a 240 x 240 central cropping.

3.1 Preprocessing the data

As mentioned above, the data loading process of our model has been a bit different from the one adopted in the paper, in that we used reduced size images (240 x 240 instead of 320 x 320) and a smaller number of samples. No other major changes were made in this step other than that.

For each MRI acquisitions, roughly 30 slices are provided, corresponding to different depths of the 3D volume of the knee (See Figure 2). In particular, *for each slice*, we have at our disposal the **initial k-space** and the **target image** as a one channel gray scale real image. The steps of the data loading are:

1. **Masking:** The k-space is masked using a masking function. Masked k-space and corresponding mask are output of this step. Both these tensors are in spatial frequency domain. The mask, which is nothing but a binary tensor of same shape as the k-space, has to be stored in memory, so as to be reused in the DC layer in the network. The masking specifically consists in the element wise multiplication (Hadamard product) between the k-space and a mask produced by a masking function, that allows for many parameters, such as acceleration factor. In our case, the acceleration factor was set to 2 and 4.

2. **Transformation:** The masked k-space can now be transformed into a complex image of the same size through the Inverse Fast Fourier Transform and then center cropped to the proper size. The resulting image is of course blurred and is fed as input to the network, that will try to reconstruct it. Therefore the tensor we use as input represents a complex image with two channels: real and imaginary.

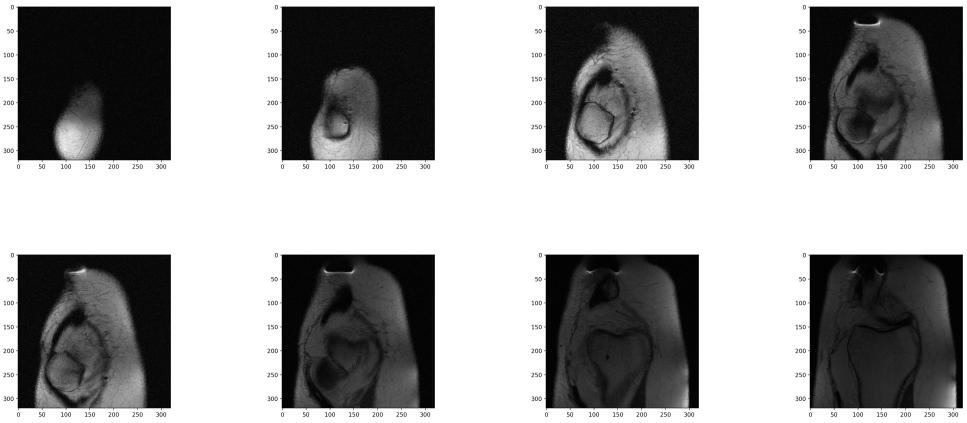

Figure 2: Slices from same MRI acquisition

All the above tensors were normalized. It is important to stress that the k-space, the mask, the masked k-space cannot be simply discarded after the input for the net had been computed: they are to be retained for later use in the DC layer in each iteration, as we will describe in next section. Listing 1 shows transformations applied in data loading.

```
def data_transform(kspace, mask_function, target, data_attributes, filename,
    slice_num):

    kspace_t = transforms.to_tensor(kspace)
    # normalize
    kspace_t = transforms.normalize_instance(kspace_t)[0]

    # apply mask: returns masked space and generated mask
    masked_kspace, mask = transforms.apply_mask(data=kspace_t, mask_func=
    mask_func)
    # apply Inverse Fourier Transform to get the complex image
    masked_image = fastmri.ifft2c(masked_kspace)
    # center crop masked image
    masked_image = transforms.complex_center_crop(masked_image, (IMG_SIZE,
    IMG_SIZE))
    # permuting the masked image for pytorch n x c x h x w format
    masked_image = masked_image.permute(2, 0, 1)
    # normalize
    masked_image = transforms.normalize_instance(masked_image)[0]

    target = transforms.to_tensor(target)
    # normalize
    target = transforms.normalize_instance(target)[0]
    # add dummy dimension
    target = torch.unsqueeze(target, 0)

    return kspace_t, masked_image, target, mask, data_attributes['max'],
    slice_num
```

Listing 1: Data Loader and transforms

4 The network

To improve recovery of fine details, the network reconstructs the image at multiple scales. We consider the image x as the combination of images x_i in different scales.

$$x = f(x_1, x_2, ..., x_s) \tag{5}$$

where x_s indicates image x at scale s and f is a function to integrate images from different scales. Then we can reconstruct each x_s as:

$$\hat{x}_s = g_s(\tilde{x}_s) \tag{6}$$

where $\tilde{x}_s$ is the image with artefacts and gs is the function for reconstruction. Each g_s is specialized in modeling the signal at a certain scale, especially the high frequency signals. The network features three specially designed ConvRNN modules to model data in different scales (ConvRNN1-4x, ConvRNN2-2x and ConvRNN3-1x). Each ConvRNN module consists of an encoder g_{enc}, an decoder g_{dec}, a basic RNN cell (ResBlock) g_{res} including two residual convolutions (ResConv), and a data consistency (DC) layer, as shown in Figure 1 and 2.

The input images are zero-filled undersampled complex images $F^{-1}DTy$ with real and imaginary values as two channels. The output of $(k+1)th$ iteration of ConvRNNi can be derived as follows:

$$x^{(k+1)} = DC(g_i(x_i^{(k)}, h_i^{(k)}, y, D, \theta_i)) \tag{7}$$

$$= F^{-1}[Dy + (1-D)Fg_i^{dec}(g_i^{res}(h_i^{(k)}) + g_i^{enc}(x_i^{(k)}))] \tag{8}$$

where $h_i^{(k)} = g_i^{res}(h_i^{(k-1)}) + g_i^{enc}(x_i^{(k-1)})$ is the hidden state from previous iteration and $h(0) = 0$.

The final output after K iteration of all three ConvRNN are derived recursively:

$$x1 = ConvRNN1(x_0, y, D, \theta_1) \tag{9}$$

$$x2 = ConvRNN1(x_1, y, D, \theta_2) \tag{10}$$

$$x3 = ConvRNN1(x_2, y, D, \theta_3) \tag{11}$$

where $x_0 = F^{-1}DTy$ is the zero-filled undersampled image and θ are parameters of the networks.

To ensure each ConvRNN can extract features at different scales (downsampling the feature map size by 4x, 2x, 1x) and reconstruct the image accordingly, the contracting and expanding factors (strides) of encoders and decoders are varied. Two (de-)convolution layers with stride=2 in the encoder and decoder of first ConvRNN are used to perform the coarse reconstruction and one or none (de-)convolution layers for second and last ConvRNN to fill in fine details.

Finally, a CNN final module is applied to combine the three reconstructed images x_1, x_2, x_3 and derive the final reconstruction $\hat{x}$:

$$\hat{x} = DC(CNN(x_1, x_2, x_3)) \tag{12}$$

Figure 3 shows a shallow (not unrolled nor zoomed-in) picture of the net. The complete picture is represented in Figure 4, whereas the detailed description of parameters for each layer is described in Figure 6. We now describe in a more detailed fashion the layers of the network, and show how they were implemented using PyTorch.

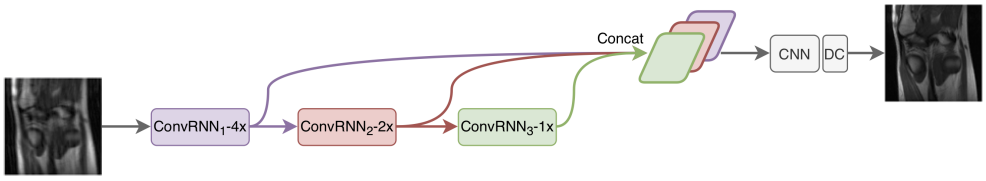

Figure 3: Pyramid convolutional neural network, without zoom in

4.1 ConvRNN block

The network comprises three ConvRNN block (ConvRNN1-4x, ConvRNN2-2x and ConvRNN3-1x) specialized for reconstructing the image at different scales (4x, 2x and 1x respectively). Each ConvRNN block is in fact a recurrent unit, and can be further decomposed into 4 sub blocks: **encoder**, **residual convolution (ResBlock)**, **decoder**, **data consistency layer (DC)**. The tensor input image enters the **encoder** first, where it is downsampled through two convolutional blocks with proper slides and padding values. The output of the encoder is then added to the output of the **ResBlock** from the previous iteration, to form the current hidden state of the ConvRNN. The first **hidden state** is a tensor of zeroes.

Subsequently, the current hidden state is fed as input to the **decoder**, that attempts to upsample the resulting image through 2 deconvolutions (transposed convolutions) and reconstruct fine details. Finally, the resulting tensor is processed by the **Data Consistency Layer**, that makes sure that only initially masked frequencies be reconstructed by the net. The complete code for the first of the three convolutional blocks is shown in Listing 2. We now focus on the details of a ConvRNN generic block.

```
class ConvRNN1(nn.Module):
    def __init__(self):
        super(ConvRNN1, self).__init__()

        # encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(in_channels=2, out_channels=384,
         kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=384, out_channels=384,
        kernel_size=3, stride=2, padding=1),
            nn.ReLU()
        )
        # residual convolutions
        self.resblock = ResBlock(in_channels= 384, out_channels=384,
            kernel_size=3, stride=1,padding=1)
        # decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(in_channels=384, out_channels=384,
                           kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=384,
                       out_channels=2,
            kernel_size=4,
            stride=2,
            padding=1),
            nn.ReLU()
        )
        # data consistency
        self.dc = DataConsistencyLayer()


    def forward(self, input, hidden_input, kspace, mask):
      # encoder
        enc_out = self.encoder(input)
        # residual convolutions
        resblock_out = self.resblock(hidden_input)
        # add hidden state to ouput of encoder
        current_hidden_state = enc_out.add(resblock_out)
        #decoder
        dec_out = self.decoder(current_hidden_state)
        # data consistency
        dc_out = self.dc(dec_out, kspace, mask)

        return dc_out, current_hidden_state
```

Listing 2: Code for ConvRNN block

Encoder layer

The encoder is made up of two convolutional layers. The padding and stride of the convolution are carefully chosen so as to analyse the image at a proper scale. In particular, the

shape of the input tensor is altered in the convolution, according to the usual formula:

$$W_{out} = \frac{(W_{in} - W_k + 2P)}{S} + 1 \tag{13}$$

where W_{out} is the width of the output image, W_{in} is the width of input image, W_k is the kernel width P is the padding value and S represents the stride. The same formula can be used to compute the height.

Residual Convolutional layer

The residual convolutional layer is made up of two residual convolutions. The parameters of the convolutions depend on the specific ConvRNN block. These convolutions are introduced in order to mitigate the effect of vanishing gradient. In these layers, the output of the convolutions is added to the input before the convolution itself is applied. A Batch Normalization is eventually applied to the resulting tensor. ResBlock class is shown in Listing 3.

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size = 3, stride =
    1, padding=1):
        /... /

    def forward(self, input):
        #ResConv1
        identity1 = input                #skip connection
        out = self.conv1(input)
        out += identity1                 #apply skip connection
        out = self.act1(out)
        #ResConv2
        identity2 = out                  #skip connection
        out = self.conv2(out)
        out += identity2                 #apply skip connection
        out = self.act2(out)
        out = self.norm(out)    #apply batch normalization
        return out
```

Listing 3: Code for Residual convolution block

Decoder layer

The encoder takes as input the current hidden state of the net, i.e. the sum of the output of the previous iteration recurrent block and the output of the encoder. It comprises two transposed convolution layers, whose parameters are tuned to reconstruct an image of the initial size.

Data Consistency layer

The data consistency layer is positioned at the end of a ConvRNN block, as a last step for each iteration and as a last layer of the net as well. It is used to ensure that only details that are supposed to be reconstructed be actually produced by the net. In particular, the DC layer takes as input the initial **k-space** (frequency domain), **the mask** that was applied to the kspace (frequency domain), and the **current complex image** as it was reconstructed up to the current iteration. In this layer, the image is first padded to be the same size of the k-space and mask, and then transformed into the frequency domain through the Fast

Fourier Transform. The image is then masked with an "anti mask", so as to *hide only frequencies that were not hidden in the first place during data loading.* The anti mask is obtained by computing the difference between a tensor of ones and the initial mask. The masked kspace and the "anti masked" current reconstruction are the multiplied element wise and the resulting image is anti transformed (IFFT) and center cropped to the initial image size. This operation ensures that only details that were initially masked are actually reconstructed by the network, whereas **other reconstructed values are replaced by initial correct values**.

```
class DataConsistencyLayer(nn.Module):
    def __init__(self):
        super(DataConsistencyLayer, self).__init__()

    def proper_padding(self, prediction, k_space_slice):
      #pad prediction to be same size as k_space
        h = prediction.shape[-3]
        w = prediction.shape[-2]
        w_pad = (k_space_slice.shape[-2] - w) // 2
        h_pad = (k_space_slice.shape[-3] - h) // 2
        return torch.nn.functional.pad(prediction,
                    (0,0,w_pad,w_pad,h_pad,h_pad),
                    'constant', 0)

    def data_consistency_kspace(self, prediction, k_space_slice, mask):
      prediction = prediction[:,:,0:IMG_SIZE, 0:IMG_SIZE]
        prediction = prediction.permute(0,2,3,1)
        # padding:
        prediction = self.proper_padding(prediction, k_space_slice)
        # transform to kspace domain:
        k_space_prediction = fastmri.fft2c(prediction)

        # apply mask:
        k_space_out = (1 - mask) * k_space_prediction + mask * k_space_slice
        # back to complex image
        prediction = fastmri.ifft2c(k_space_out)
        # crop image
        prediction = transforms.complex_center_crop(prediction, (IMG_SIZE,
    IMG_SIZE))
        # back to 1x2xhxw
        prediction = prediction.permute(0,3,1,2)
        return prediction

    def forward(self, prediction, original_kspace_slice, mask):
        out = self.data_consistency_kspace(prediction,
                    original_kspace_slice,
                       mask)
        return prediction
```

Listing 4: Code for data consistency layer

4.2 Concatenation Layer and Final convolutional layer

In the Concatenation layer, the outputs of the three ConvRNN blocks are stacked, to form a 6 channels tensor. This is then fed as input to 4 final convolutions and finally to an additional DC layer. The final prediction of the network is a complex image, with real and imaginary values, whereas the target is a real valued image. Hence, in this final step we also transform the prediction into a real image by considering its magnitude.

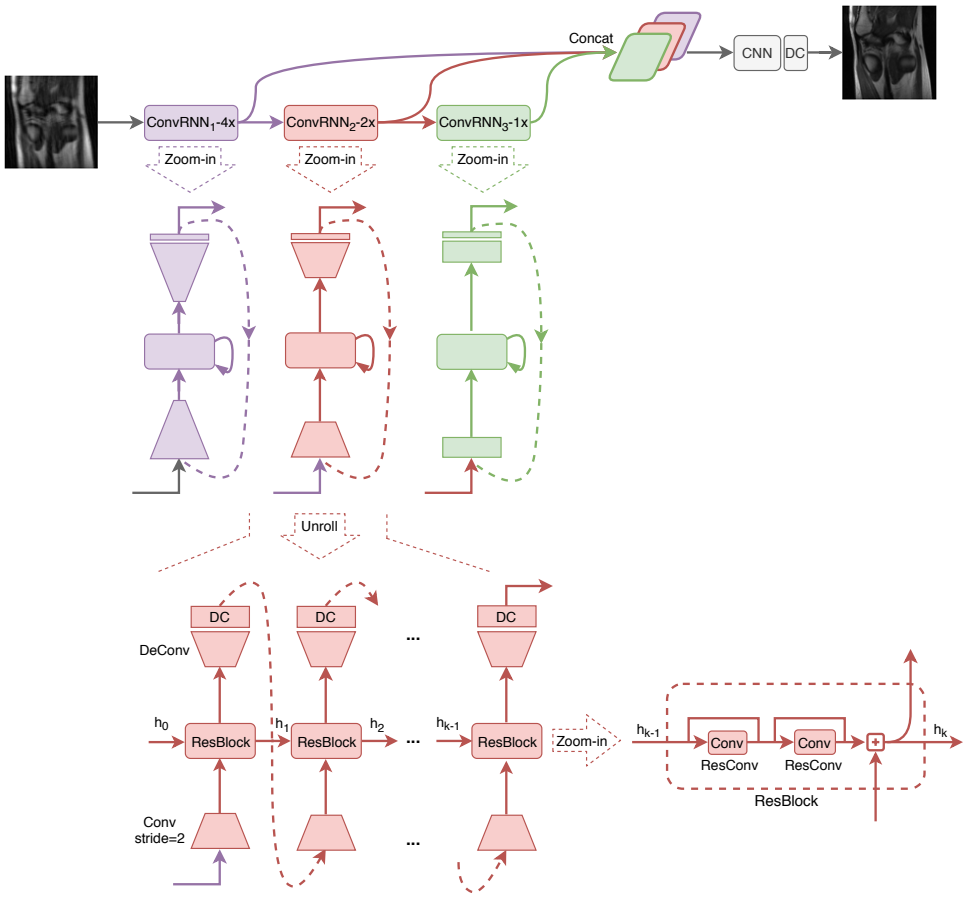

Figure 4: Pyramid convolutional neural network, with zoom in

5 Training

The network was trained for 60 epochs, and the number of iterations for each ConvRNN block was set to 5.

Loss function

For each slice, the output of the net was compared to the target provided in the dataset. The total loss function is defined as a linear combination of Normalised Mean Square Error (NMSE) loss and the Structural Similarity Index (SSIM) loss. The total loss function is derived as follows:

$$\mathcal{L}_{NMSE}(\hat{x}, x) = \frac{\|\hat{x} - x\|_2^2}{\|v\|_2^2} \tag{14}$$

$$\mathcal{L}_{SSIM}(\hat{x}, x) = \frac{(2\mu_{\hat{x}}\mu_x + c_1)(2\sigma_{\hat{x}x} + c_2)}{(\mu_{\hat{x}}^2 + \mu_x^2 + c_1)(\sigma_{\hat{x}}^2 + \sigma_x^2 + c_1)} \tag{15}$$

$$\mathcal{L}(\hat{x}, x) = \mathcal{L}_{NMSE}(\hat{x}, x) + \beta \mathcal{L}_{SSIM}(\hat{x}, x) \tag{16}$$

where $\|v\|_2^2$ represents squared Euclidean norm of the volume v that x belongs to, and $c1 = (k1L)^2$, $c2 = (k2L)^2$ in SSIM. Window size is 7×7, and $k1 = 0.01, k2 = 0.03$, and define L as the maximum magnitude value of the target image x, $L = max(\|x\|)$. We used $\beta = 0.5$ to balance the two loss functions.

Training routine

The neural network was trained with the lookahead version of the Adam optimization algorithm. Adam (adaptive moment estimation) is a variation of the classical stochastic gradient descent. In the paper, the learning rate was set to 10^{-5} for the first epoch as the training warmup and increased to 10^{-4}, and was then reduced by a factor of 2 every 10 epochs. We, on the other hand, set the learning rate to 10^{-4} and then reduced it by a factor of 2 every 15 epochs(using a scheduler), since those settings seemed to yield the best results with our different configuration. There are no specifications in the paper about the batch size that was used during the training. We used a batch size of 1, which was also the only size that our GPU was able to support.

As mentioned above, the acceleration factors used for masking functions are $2X$ and $4X$. These values represent the "amount" of masking. Even though the $2X$ is not used in other models, it was coherent with the choice of using a reduced dataset.

6 Evaluation and results

Unsurprisingly, the results we obtained are not much close to the ones achieved in the original paper. Even though the network converges, and is able to reduce the total loss, it is still far from reconstructing fine details that were lost due to the application of the mask. This is easily understood, because we used a reduced dataset, with size less than a half of the original one and we cropped the images, too. The limitations imposed by the hardware were significant and made it impossible for us to train the network for days, which was for sure possible at Facebook, where the original model was trained. We can claim that the net, if trained on a decently large dataset, with powerful enough GPUs or TPUs, would outperform the other algorithms and models and attain the same results as the original version. Unfortunately in this case, we were only able to achieve a $SSIM$ of 0.643 for $4X$ acceleration and 0.725 for $2X$, as shown in Table 1. Figure 5 shows the reconstruction of a $4X$ masked image with five different models.

	$SSIM$		
	2×	4×	8×
PCRNN (original)	na	0.743	0.682
PCRNN (our implementation)	0.725	0.643	0.544
CS	na	0.570	0.484
UNet	na	0.723	0.654

Table 1: SSIM results of CS, UNet and PC-RNN on fastMRI validation data

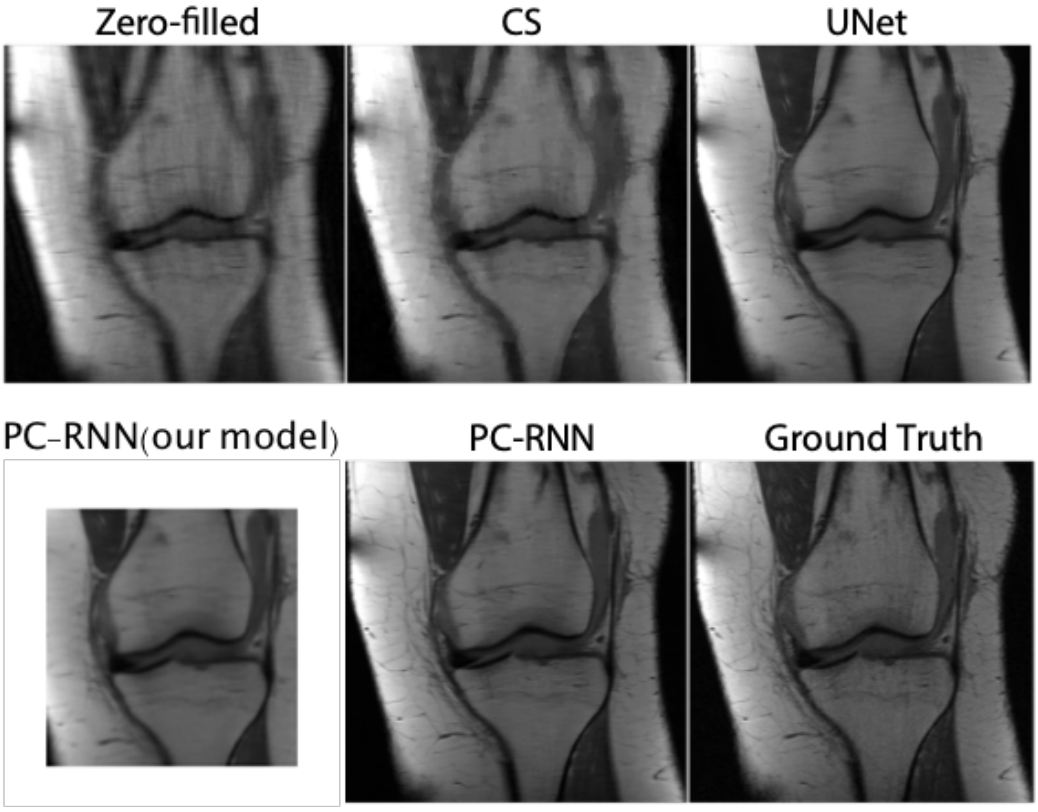

Figure 5: Comparison of the result of models on a 4X masked kspace

References

[1] Ian Goodfellow and Yoshua Bengio and Aaron Courvillel *Deep Learning*, 2016, MIT press

[2] Puyang Wang, Eric Z. Chen, Terrence Chen, Vishal M. Patel, Shanhui Sun *Pyramid Convolutional RNN for MRI Reconstruction*

[3] Schlemper, J., Caballero, J., Hajnal, J. V., Price, A., & Rueckert, D. *A Deep Cascade of Convolutional Neural Networks for MR Image Reconstruction.*

[4] Jure Zbontar, Florian Knoll, Anuroop Sriram et al. *fastMRI: An Open Dataset and Benchmarks for Accelerated MRI*

[5] Computer vision: Fourier transforms of images https://plus.maths.org/content/fourier-transforms-images

[6] Facebook github repository for FastMRI project *https://github.com/facebookresearch/fastMRI*

[7] ResNet explained *https://paperswithcode.com/method/resnet*

[8] Anuroop Sriram, Jure Zbontar, Tullie Murrell et al. *End-to-End Variational Networks for Accelerated MRI Reconstruction*

Module	Block	Type	Output size
	Input image	–	2×320×320
ConvRNN_1	Encoder	Conv(2,3,1)	384×160×160
		Conv(2,3,1)	384×80×80
	ResBlock	ResConv	384×80×80
		ResConv	384×80×80
	Decoder	DeConv(2,4,1)	384×160×160
		DeConv(2,4,1)	2×320×320
ConvRNN_2	Encoder	Conv(1,3,1)	192×320×320
		Conv(2,3,1)	192×160×160
	ResBlock	ResConv	192×160×160
		ResConv	192×160×160
	Decoder	DeConv(2,4,1)	192×320×320
		DeConv(1,4,1)	2×320×320
ConvRNN_3	Encoder	Conv(1,3,1)	96×320×320
		Conv(1,3,1)	96×320×320
	ResBlock	ResConv	96×320×320
		ResConv	96×320×320
	Decoder	DeConv(1,4,1)	96×320×320
		DeConv(1,4,1)	2×320×320
CNN	–	Conv(1,3,1)	6×320×320
		Conv(1,3,1)	96×320×320
		Conv(1,3,1)	96×320×320
		Conv(1,3,1)	2×320×320
	Output image	–	2×320×320
	ResConv	Input	C×M×N
		Conv(1,3,1)	C×M×N
		Conv(1,3,1)	C×M×N

Figure 6: Pyramid convolutional neural network layers