Laplacian pyramid-based complex neural network learning for fast MR imaging

a Neural Network project

Ketbjano Vocaj - 1652134
Matteo Russo - 1664715
Paolo Tarantino - 1666228

October 2020

1 Introduction

In this report we'll talk about the methods and the steps followed for implementing the paper *Laplacian pyramid-based complex neural network learning for fast MR imaging.* First of all we'll give an overview on the specific Biomedical theme in which we are involved (by defining some technical terminologies), we'll proceed by analyzing the Network implementation and finally we'll discuss the results acquired in our work.
The goal in *Magnetic Resonance Imaging* researches is to find methodologies for reconstructing high-resolution images in the fastest way: the results achieved so far, show that k-space undersampling is one most popular method that can produce near to perfect images.
Furthermore, in this framework the Laplacian pyramid decomposition is proved to perform better than the other frameworks used for this purpose (such as U-Net, KIKI-Net, Cascade-Net etc...).

2 Magnetic Resonance Imaging

From wikipedia.org: *"Magnetic resonance imaging (MRI) is a medical imaging technique used in radiology to form pictures of the anatomy and the physiological processes of the body"*.
The images are saved in raw data known as **k-space**, which is the 2D or 3D Fourier transform of the MR image measured in complex representation (so we have 2 dimensions each one specifies the real and the imaginary part).

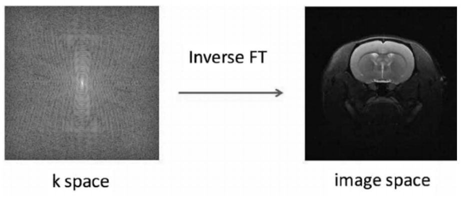


Figure 1: Fourier transform

As we can see in **Figure 1** the anatomical image is obtained by applying the Inverse of the Fourier Transform on the k-space; on the other hand, k-space is the result of the Fourier Transform applied on the image.

2.1 Undersampling

Now, since the scan time is the limiting factor in many magnetic resonance imaging applications, one approach to reduce it is to undersample the k-space: this is done by keeping a subset of the full-sampled data.
More precisely, we take the fully sampled k-space data and we apply on it a "noisy" mask in order to obtain the undersampled k-space data; this will be the input of the Inverse Fourier Transform which returns the undersampled real image that we will call **zero filled** image. Note that the image proposed so far has some "imperfections" compared to the fully sampled one (as shown below in **Figure 2**) but, as we will clarify later on, it's a crucial part for accelerating scan time since the network has to deal with less data (refer to **Section 5** for the implementation).

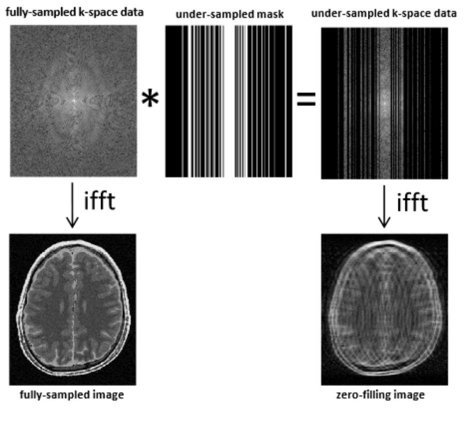


Figure 2: Undersampling procedure

3 Complex Laplacian Decomposition

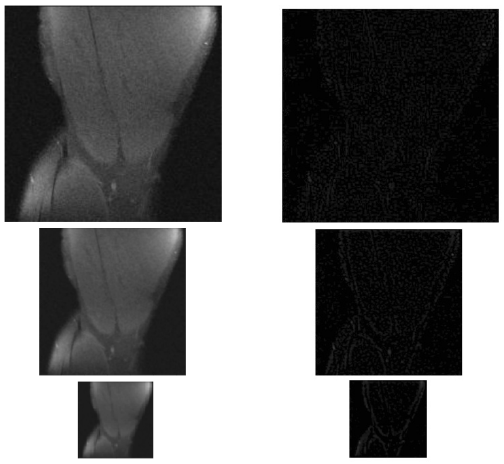
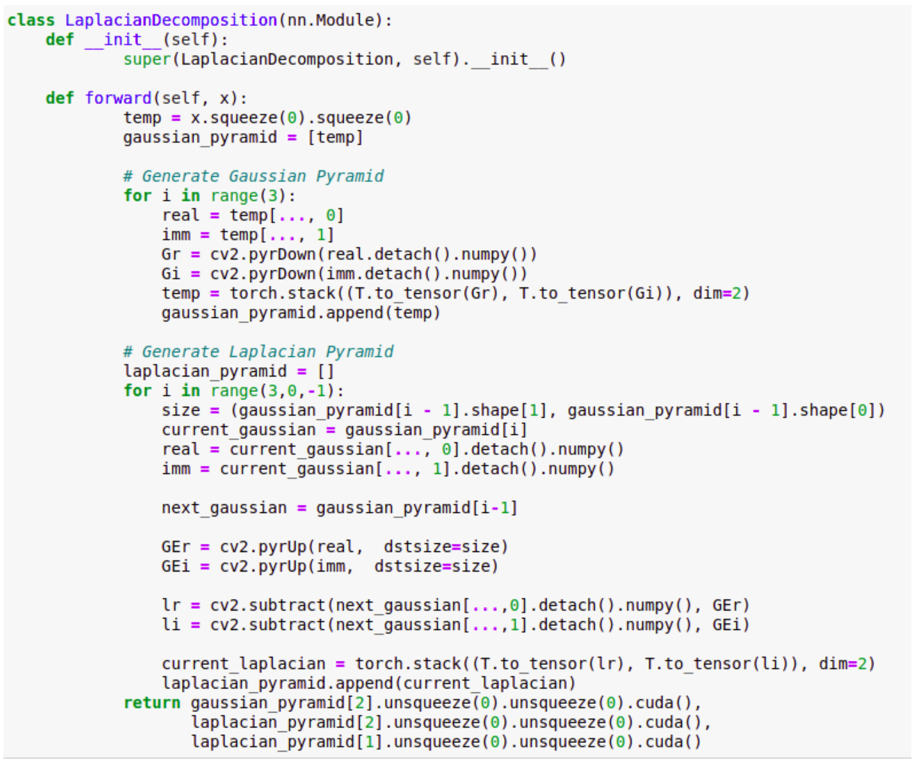


Figure 3: Gaussian and Laplacian pyramids

The Complex Laplacian Decomposition is the main work's core: the input image is undergone different operations, that will result in having two Laplacian error images and one Gaussian image. Starting from an input image (the zero filled one), we compute the Gaussian pyramid by sequentially scaling down the input in dimensions. Then, the smallest Gaussian image is taken and then upscaled up to the input image dimension previously given; this will result in having a pyramid of images that presents some form of blur, due to the upscaling. The two Laplacian error images can be computed by subtracting from the two biggest images of the Gaussian pyramid the two biggest blurred upscaled images. For reconstructing the Laplacian pyramid is needed to substract from each Gaussian image its upscaled version. However due to paper's specification we only cared about the first two Laplacian results.
For this step, we used the OpenCV (`cv2` in **Figure 4**) library that offered good tools for the upscaling and downscaling operations.



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

    def forward(self, x):
            temp = x.squeeze(0).squeeze(0)
            gaussian_pyramid = [temp]

            # Generate Gaussian Pyramid
            for i in range(3):
                real = temp[..., 0]
                imm = temp[..., 1]
                Gr = cv2.pyrDown(real.detach().numpy())
                Gi = cv2.pyrDown(imm.detach().numpy())
                temp = torch.stack((T.to_tensor(Gr), T.to_tensor(Gi)), dim=2)
                gaussian_pyramid.append(temp)

            # Generate Laplacian Pyramid
            laplacian_pyramid = []
            for i in range(3,0,-1):
                size = (gaussian_pyramid[i - 1].shape[1], gaussian_pyramid[i - 1].shape[0])
                current_gaussian = gaussian_pyramid[i]
                real = current_gaussian[..., 0].detach().numpy()
                imm = current_gaussian[..., 1].detach().numpy()

                next_gaussian = gaussian_pyramid[i-1]

                GEr = cv2.pyrUp(real,  dstsize=size)
                GEi = cv2.pyrUp(imm,  dstsize=size)

                lr = cv2.subtract(next_gaussian[...,0].detach().numpy(), GEr)
                li = cv2.subtract(next_gaussian[...,1].detach().numpy(), GEi)

                current_laplacian = torch.stack((T.to_tensor(lr), T.to_tensor(li)), dim=2)
                laplacian_pyramid.append(current_laplacian)
            return gaussian_pyramid[2].unsqueeze(0).unsqueeze(0).cuda(),
                   laplacian_pyramid[2].unsqueeze(0).unsqueeze(0).cuda(),
                   laplacian_pyramid[1].unsqueeze(0).unsqueeze(0).cuda()
```

Figure 4: Laplacian Decomposition Code

We take the input `x` and we perform two squeeze operations in order to have a 3-dimension tensor. In the

first `for` loop, we split the real from the imaginary part, we perform the downscale on each chunk, and then re-obtaining a new 3-dimension tensor by doing the `stack` operation.
We append the new `upscaled` tensor to the gaussian pyramid list and we proceed to the next step.
In the second `for` loop, we want to either upscale each Gaussian tensor and to perform the subtract operation to produce each Laplacian error image. We then store two different gaussians at each cycle: the `current_gaussian` is used for computing the upscaled version of the image(called GEr and GEi), while `next_gaussian` is used for computing the subtract operation. The subtract is computed taking into account the real and imaginary part separately: the recomposed image is obtained using `torch.stack`. As the final step we return the smallest gaussian, the biggest and the middle laplacian error images respectively.

4 Network implementation

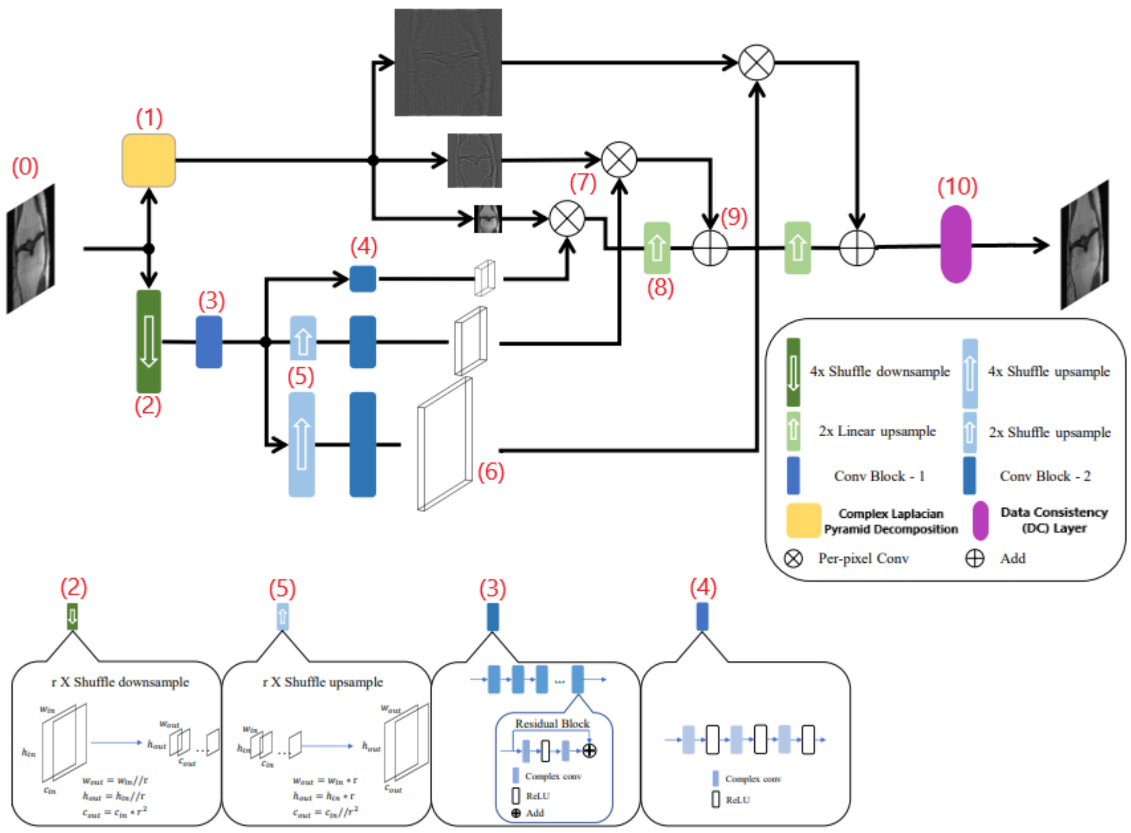


Figure 5: CLP-Net

In this section we provide the practical functions and methodologies used to implement the **CLP-Net** [1]. Before describing in details the network structure, it is important to notice that some network characteristics described in the paper have been modified in order to fit our hardware means.
Recalling **Figure 5**, the network's input is the undersampled image **(0)** that goes through two paths: in the first one we compute the pyramid decomposition **(1)** which returns 3 images (as said in section 3) that will be used later on in the *Per-Pixel Convolution* operation. In the second path we compute the *4xShuffleDownsample* **(2)** that produces 16 Feature Maps which will be sent to the *Convolutional-Block-1*. The *Convolutional-Block 1* **(3)** is composed by two residual blocks, each having two complex convolution layers and one ReLU activation layer. The output shallow features are then used as input for each of the 3 branches that follow. The upper branch is the only one that does not present a reshuffle operation, it only has a *Convolutional-Block-2* **(4)**, while the others have an Upsample operation **(5)** performing a reduce on the input channels and a resize on the Feature Maps. Both the operations are done according to a scale factor which is 2 for the middle branch and 4 for the lower one. The second Convolutional Block is a sequential module that consists of three complex convolution layers and three ReLU activation layers. All the branches are then resized **(6)** following the kernel size=5, for the paper specification, and the outputs are then used for the *Per-Pixel Convolution* operation **(7)** that involves the output images coming from the Laplacian Decomposition in the first path. The Per-Pixel

[1]Pytorch v1.6.0 library was used for this purpose

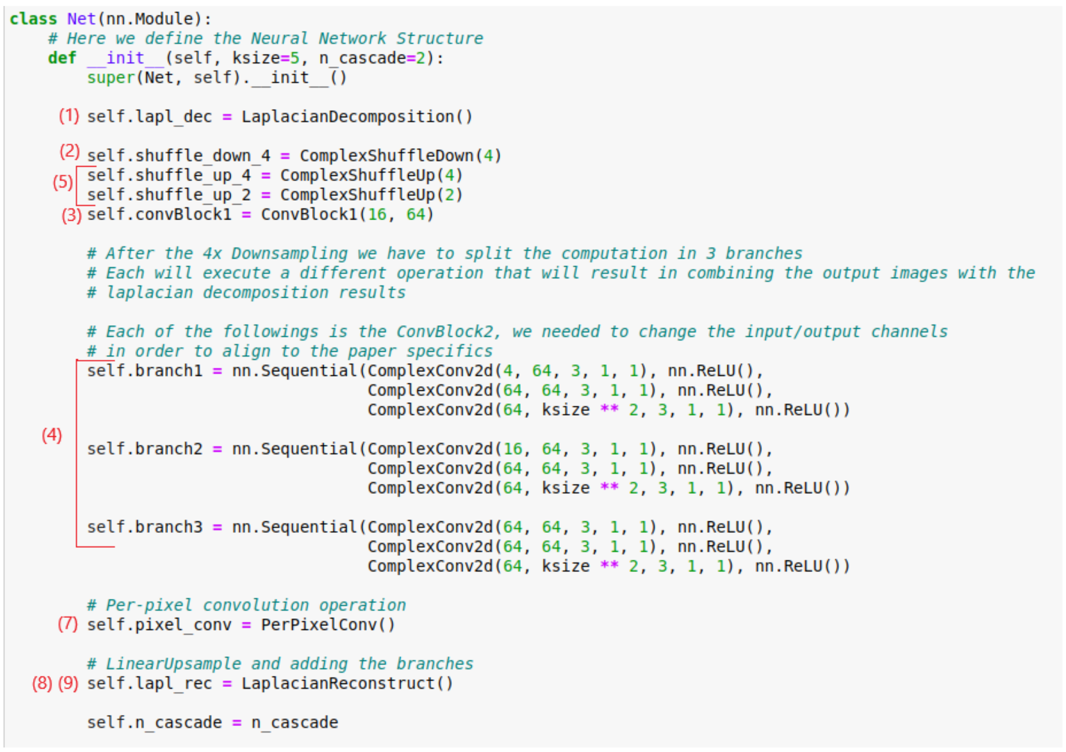
Convolution performs the following operation:

$$F(m,n) = I_{neighbour}(m,n) \otimes K(m,n)$$

where $\mathbf{K(m,n)}$ means the pixel value of the "Kernel" at position (m,n), and $\mathbf{I_{neighbour}(m,n)}$ is the neighborhood of the pixel of the Laplacian error maps or the Gaussian maps at position (m,n). F(m,n) is the pixel of the acquired feature maps F at position (m,n), and $\otimes$ means inner product operation. The resultant image is then upsampled **(8)** one more time, added to the neighbour **(9)** and finally passed to the *Data Consistency Layer*. The Data Consistency Layer **(10)** takes as input the Reconstructed Image of the previous step along with the mask and the undersampled k-space data. In particular, on the input reconstructed MR images will be performed a Fourier Transform and multiplied by a inverse mask afterwards. The inverse mask is obtained by (1 - mask). After that, we can obtain the reconstructed k-space data by adding the undersampled k-space data to the result of the previous step, and we will perform inverse Fourier Transform on the reconstructed k-space data to produce the final reconstructed MR image.
And the whole above process will be repeated several times, resulting into a cascaded structure.

4.1 Code Implementation

As mentioned before, the paper's Neural Network structure is not feasible to be fitted by our hardware mean. More specifically, we changed the cascades number `n_cascade` from five to two. The cascade number represents how many times the Network loops over itself using the new reconstructed image each time.
The Residual Block number is 16 in the original implementation, while we changed this parameter to be 2. In **Figure 6** we defined all the network's layers in the manner we saw previously[2] by initializing the proper classes and pointing out the input and output channels of every single Convolution (each followed by a ReLU activation function).
The actual run and computation are done in the `forward()` function defined in **Figure 7**: we pass the `mr_img`, `mk_space` and `mask` computed in the `data_transform` before training (**Section 5**).
We start by looping over the cascades number, then we proceed by following the two paths we already mentioned in **Section 4**: the Laplacian decomposition was performed using the OpenCV library, the resulting three images are stored in `gaussian_3, lap_1, lap_2` variables.



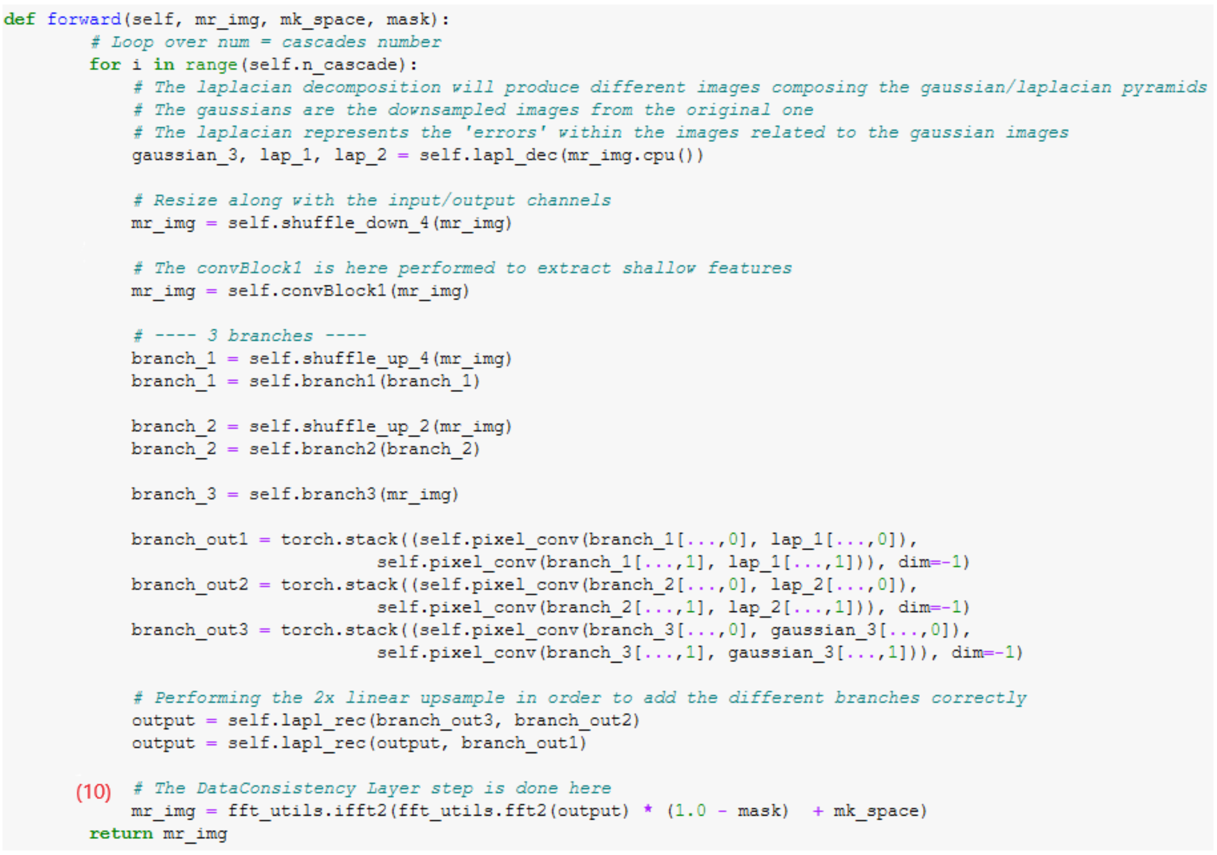
```
class Net(nn.Module):
    # Here we define the Neural Network Structure
    def __init__(self, ksize=5, n_cascade=2):
        super(Net, self).__init__()

    (1) self.lapl_dec = LaplacianDecomposition()

    (2) self.shuffle_down_4 = ComplexShuffleDown(4)
    (5) self.shuffle_up_4 = ComplexShuffleUp(4)
        self.shuffle_up_2 = ComplexShuffleUp(2)
    (3) self.convBlock1 = ConvBlock1(16, 64)

        # After the 4x Downsampling we have to split the computation in 3 branches
        # Each will execute a different operation that will result in combining the output images with the
        # laplacian decomposition results

        # Each of the followings is the ConvBlock2, we needed to change the input/output channels
        # in order to align to the paper specifics
        self.branch1 = nn.Sequential(ComplexConv2d(4, 64, 3, 1, 1), nn.ReLU(),
                                     ComplexConv2d(64, 64, 3, 1, 1), nn.ReLU(),
                                     ComplexConv2d(64, ksize ** 2, 3, 1, 1), nn.ReLU())

    (4) self.branch2 = nn.Sequential(ComplexConv2d(16, 64, 3, 1, 1), nn.ReLU(),
                                     ComplexConv2d(64, 64, 3, 1, 1), nn.ReLU(),
                                     ComplexConv2d(64, ksize ** 2, 3, 1, 1), nn.ReLU())

        self.branch3 = nn.Sequential(ComplexConv2d(64, 64, 3, 1, 1), nn.ReLU(),
                                     ComplexConv2d(64, 64, 3, 1, 1), nn.ReLU(),
                                     ComplexConv2d(64, ksize ** 2, 3, 1, 1), nn.ReLU())

        # Per-pixel convolution operation
    (7) self.pixel_conv = PerPixelConv()

        # LinearUpsample and adding the branches
(8) (9) self.lapl_rec = LaplacianReconstruct()

        self.n_cascade = n_cascade
```

Figure 6: Network's layers

The other path is composed by the `self.shuffle_down` followed by the first Convolutional Block. The output feature maps are then sent through the 3 branches, named in code as `branch1, branch2, branch3`,

[2]We labeled the steps with red circled numbers so it's easier to visually follow the steps and the related implementation.

each of them outputs some shallow features using a fixed output channels value according with the kernel size parameter (`ksize`) defined at the beginning of the network, as seen in the `__init__` method.
The outputs shallow features of each branch are then used for the per-pixel convolution step, taking into account the real and imaginary part of each branch output; then a stack operation is performed in order to have a tensor of 5 dimension at the end of the step (with the form of [1,1,H,W,2], with H and W that can vary).
Then we have the reconstruction step, which performs the Upsample method along with the ADD operation between images. As the last step, the Data Consistency Layer takes the Net's input parameters `mk_space` and `mask` to produce the resultant image.

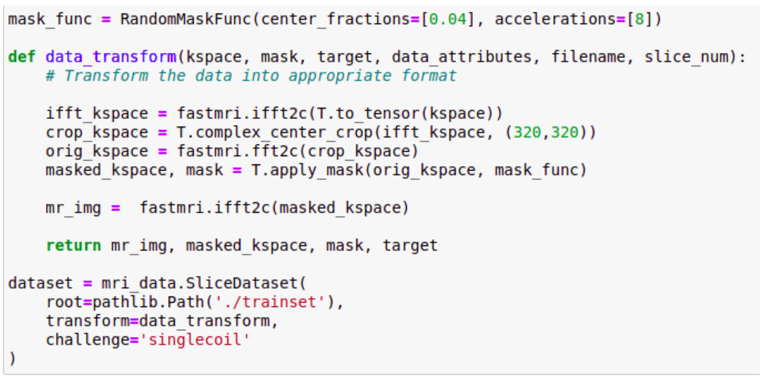


```python
def forward(self, mr_img, mk_space, mask):
        # Loop over num = cascades number
        for i in range(self.n_cascade):
            # The laplacian decomposition will produce different images composing the gaussian/laplacian pyramids
            # The gaussians are the downsampled images from the original one
            # The laplacian represents the 'errors' within the images related to the gaussian images
            gaussian_3, lap_1, lap_2 = self.lapl_dec(mr_img.cpu())

            # Resize along with the input/output channels
            mr_img = self.shuffle_down_4(mr_img)

            # The convBlock1 is here performed to extract shallow features
            mr_img = self.convBlock1(mr_img)

            # ---- 3 branches ----
            branch_1 = self.shuffle_up_4(mr_img)
            branch_1 = self.branch1(branch_1)

            branch_2 = self.shuffle_up_2(mr_img)
            branch_2 = self.branch2(branch_2)

            branch_3 = self.branch3(mr_img)

            branch_out1 = torch.stack((self.pixel_conv(branch_1[...,0], lap_1[...,0]),
                                       self.pixel_conv(branch_1[...,1], lap_1[...,1])), dim=-1)
            branch_out2 = torch.stack((self.pixel_conv(branch_2[...,0], lap_2[...,0]),
                                       self.pixel_conv(branch_2[...,1], lap_2[...,1])), dim=-1)
            branch_out3 = torch.stack((self.pixel_conv(branch_3[...,0], gaussian_3[...,0]),
                                       self.pixel_conv(branch_3[...,1], gaussian_3[...,1])), dim=-1)

            # Performing the 2x linear upsample in order to add the different branches correctly
            output = self.lapl_rec(branch_out3, branch_out2)
            output = self.lapl_rec(output, branch_out1)

      (10)  # The DataConsistency Layer step is done here
            mr_img = fft_utils.ifft2(fft_utils.fft2(output) * (1.0 - mask)  + mk_space)
        return mr_img
```

Figure 7: Forward function

5 Training the network

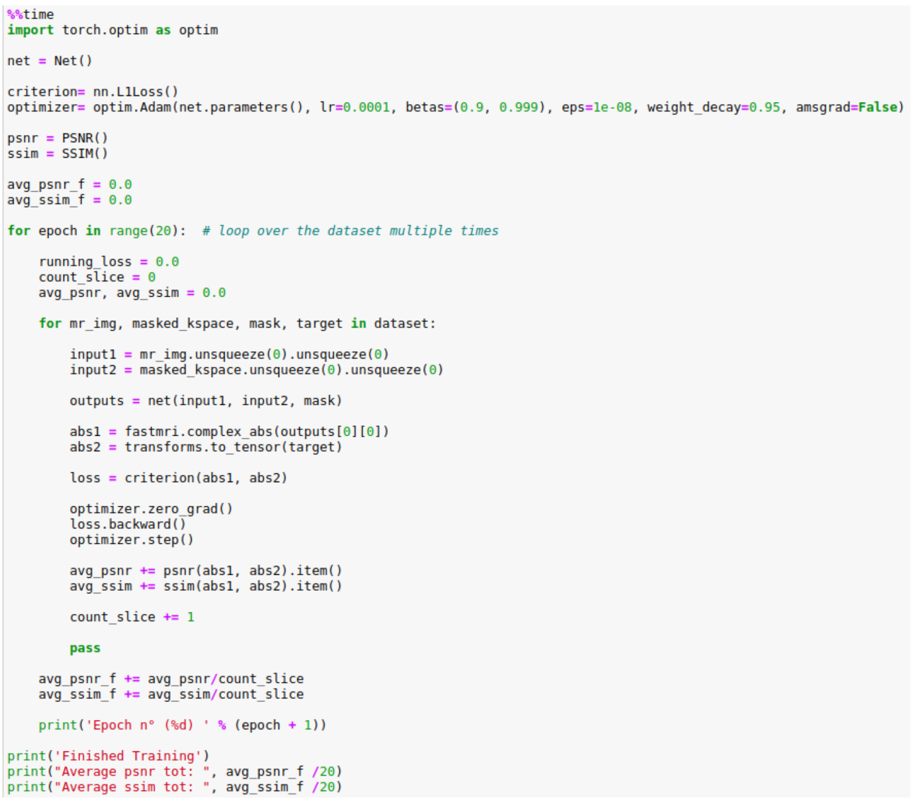
The *fastMRI* dataset has files in the `.h5` format, each file has a number of slices that represent the different levels of deepness of a knee scan; we used a total of 361 images as input for the net. Before training the network we defined a function that applies the transformations introduced in **Section 2** to every element on the dataset, so we read each file`.h5` and call the function `data_transform()` on each single layer.
This function takes many parameters but we will focus mainly on the kspace, the target image (fully sampled) and the layers number; since the raw images are in the format $(640, 372)$ we firstly need to crop the kpace in the format $(320, 320)$, then we apply the mask on it through a `Random Mask Function` which gives us two outputs: the masked kspace and the mask generated. We compute the Inverse Fourier Transform (`ifft2c`) on the masked kspace for obtaining the undersampled image `mr_img` which is returned at the end of the function together with the masked kspace (the undersampled kspace), the mask and the target that will be used for comparing our results (**Figure 8**).



```python
mask_func = RandomMaskFunc(center_fractions=[0.04], accelerations=[8])

def data_transform(kspace, mask, target, data_attributes, filename, slice_num):
    # Transform the data into appropriate format

    ifft_kspace = fastmri.ifft2c(T.to_tensor(kspace))
    crop_kspace = T.complex_center_crop(ifft_kspace, (320,320))
    orig_kspace = fastmri.fft2c(crop_kspace)
    masked_kspace, mask = T.apply_mask(orig_kspace, mask_func)

    mr_img =  fastmri.ifft2c(masked_kspace)

    return mr_img, masked_kspace, mask, target

dataset = mri_data.SliceDataset(
    root=pathlib.Path('./trainset'),
    transform=data_transform,
    challenge='singlecoil'
)
```

Figure 8: Data Transform Function

Once we are done in converting our data, we begin the real training by specifying in the beginning our loss function and the optimizer (i.e. **L1 loss** and **Adam algorithm** respectively) using the parameters provided in the paper; then we run the loop on each layer for 20 epochs. On each epoch we also keep track on the average of **PSNR** and **SSIM** metrics that will help us on evaluating the work done in **Section 6**.

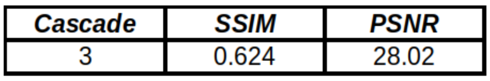


```python
%%time
import torch.optim as optim

net = Net()

criterion= nn.L1Loss()
optimizer= optim.Adam(net.parameters(), lr=0.0001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0.95, amsgrad=False)

psnr = PSNR()
ssim = SSIM()

avg_psnr_f = 0.0
avg_ssim_f = 0.0

for epoch in range(20):  # loop over the dataset multiple times

    running_loss = 0.0
    count_slice = 0
    avg_psnr, avg_ssim = 0.0

    for mr_img, masked_kspace, mask, target in dataset:

        input1 = mr_img.unsqueeze(0).unsqueeze(0)
        input2 = masked_kspace.unsqueeze(0).unsqueeze(0)

        outputs = net(input1, input2, mask)

        abs1 = fastmri.complex_abs(outputs[0][0])
        abs2 = transforms.to_tensor(target)

        loss = criterion(abs1, abs2)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        avg_psnr += psnr(abs1, abs2).item()
        avg_ssim += ssim(abs1, abs2).item()

        count_slice += 1

        pass

    avg_psnr_f += avg_psnr/count_slice
    avg_ssim_f += avg_ssim/count_slice

    print('Epoch n° (%d) ' % (epoch + 1))

print('Finished Training')
print("Average psnr tot: ", avg_psnr_f /20)
print("Average ssim tot: ", avg_ssim_f /20)
```

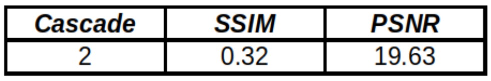
Figure 9: Training code

6 Testing and Results

We tested our network using 352 total images, collected in the same `.h5` format as the training and precomputed in the same `data_transform()` function.
During our test step, we used the two Quantitative Evaluation Indices used in the paper - i.e. PSNR (Peak Signal to Noise Radio) and SSIM (Structural Similarity Index Matrix) - in order to evaluate the quality between the reconstructed images and the fully sampled ones. We used the two functions already provided by the *fastMRI* dataset: the range of the PSNR return value is between [0, inf], but typically for a good evaluation it ranges between [20,40], while the range of the SSIM's one is between [0,1].
Since the better `kernel_size` according to the paper's specification tends to be 5, we tested our network results by keeping in mind this specific value. As already mentioned, we reduced the cascade number to 2 in our implementation, so we compare our results with the minimum cascade number of the paper's outputs.



Cascade	***SSIM***	***PSNR***
3	0.624	28.02



Cascade	***SSIM***	***PSNR***
2	0.32	19.63

Figure 10: On the left, the paper results. On the right, our results

These are the output images of our implementation of the Neural Network proposed in the paper.

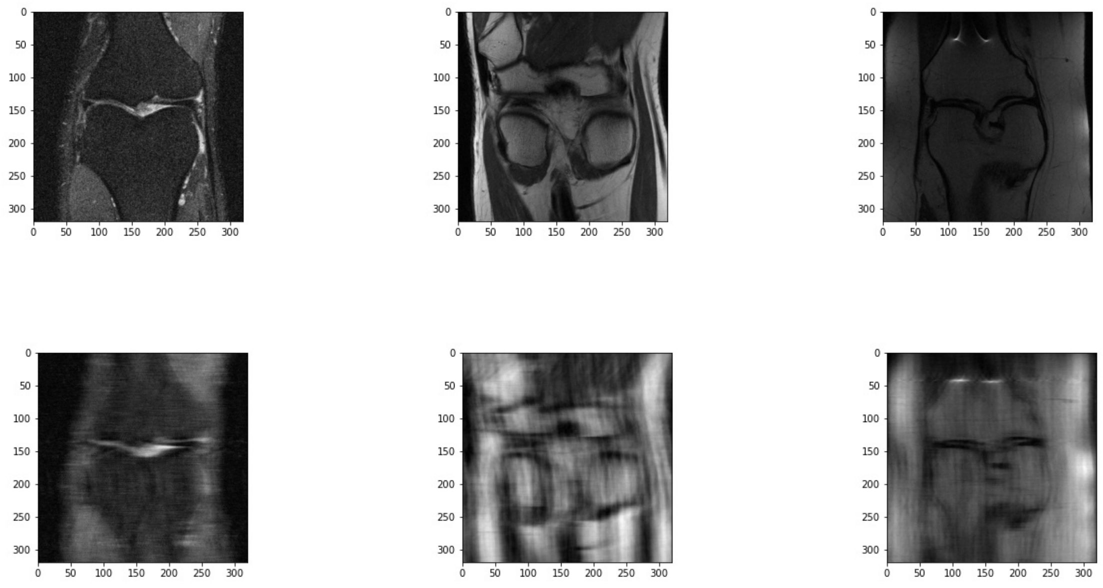


Figure 11: Comparing the outputs

7 Conclusions

The critical part of our work was the pre-training part, we needed to prepare our data to be feasible to the paper's specification; furthermore, working with complex data required more pre-computation load.
Even though our results are far from those expected due to the lack of computational power, we believe that by increasing the epochs, the cascades and residual blocks we could have obtained better results.
This work grown our interest on this field which we think will be preponderant in our everyday lives in the near future.

8 References

`https://fastmri.org`
`https://openreview.net/forum?id=0IeI8QS8N6`
`https://pytorch.org/docs/stable/index.html`