Subtle Medical Coding Challenge

Gokce Sarar
University of California, San Diego
PhD Student in ECE
06/11/2019

Task I: DICOM I/O

task1_part1.py and task_part2.py are the command line tools which are written for Task 1 and Task 2 as described. They accept the arguments as described in the problem description. Some of the necessary functions are imported from utils.py.

In order to check if two DICOM files are identical, task1_check.py can be used, which takes the following arguments:

- --input-dicom1, --i1 path to first input DICOM directory
- --input-dicom2, --i2, path to second input DICOM directory

The scripts prints 'True' if the two DICOMs are identical and 'False' if they are different.

After following the instructions, the DICOM created by the second script was same as the original DICOM, as we are asked to check.

Task II: Simulating Fast Acquisitions

task2.py is used for this part. This script reads a DICOM file, blurs it and saves as another DICOM file. The arguments are:

- --input-dicom, --i path to input DICOM directory
- --output-dicom, --o, path to output DICOM directory

In Fig. 1, the original and the blurred version of the central slice of P1_dcm can be seen. In the figure, the blurred image is the direct output of Gaussian blurring filter, but when the 3D volume is saved as hdf5 file, it is normalized to 0-1 range.

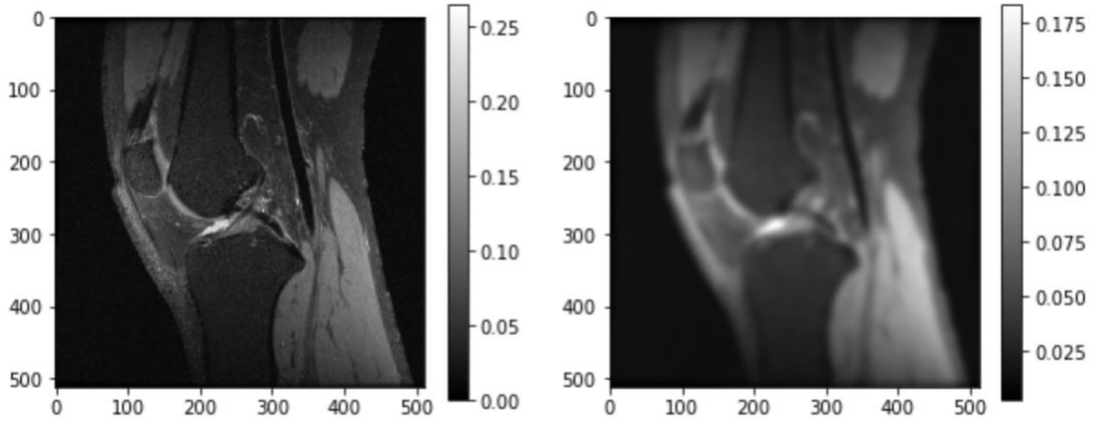


Fig 1. Central slice of the 3D volume of P1_dcm. (Left) original image, (right) blurred image

Task III: Super Resolution Using Deep Learning

In this part, our task is implementing image enhancement with DNNs. I implemented the pipeline and the model in Pytorch. The model takes a blurry 3D volume from Task II as the input and predicts an outcome that is as close as to the original 3D volume. Since we did the blurring in slice-level, I implemented a model which takes one slice and increases the resolution of that slice. The test results, I am going to report is the average on the whole test data set, which are the volumes 17, 18, 19 and 20.

For the inference I wrote test_model.py, where the user can enter the name of .hdf5 file of the volume and the result is saved as another .hdf5 file. Here I used another data_loader, since in real inference time, the ground truth can't be found. Additionally, here I also take into account that I would work on a volume so the used sampler is sequential.

Model Selection

Given the time restriction, I put my baseline as a simple CNN with 4 layers. I followed the model in [1], where the problem to solve is achieving super resolution from downsampled and blurred images, which are natural images. Even if the problem setting is very different, I used this model to setup the whole pipeline. The model I implemented can be found in modelSR.py, which I implemented to be tunable: the user can specify the number of out channels per layer, where implicitly the depth of the network is also specified. The user can also specify the kernel sizes and lastly the non-linearity operation. I only implemented for tanh and ReLI, but new non-linearities can be incorporated very easily. The reason I implemented these two is, in [1] they originally implemented ReLU and then showed that tanh is better performing.

The model has the basic block of convolutional layer with padding and a following the non-linearity layer. Based on the given depth, this block is repeated and the last layer is again a Convolutional layer without a non-linearity. Since the input and output image size is the same, there isn't any max-pooling operation. The biggest difference from [1] is our input and output images have the same sizes, so there isn't a sub-pixel convolutional layer in my model. I did my experiments with the following architecture:

Conv2d (1,64,5)→ Non-linearity → Conv2d(64,64,3) → Non-linearity→ Conv2d(64,32,3) → Non-linearity→ Conv2d(32,1,3)

The parenthesis means: (in channels, out channels, kernel size)

In the medical imaging literature GANs [2, 3] and U-nets[4,5] have shown significant success. I didn't experiment with GANs due to the difficulty in their training procedure. Some of the difficulties:

- In the training procedure the discriminator can get ahead of generator, that's why generator needs to be trained k times between each training iteration of discriminator.

- Mode collapse can happen, where generator learns to generate one good sample and tricks the discriminator, but actually doesn't learn to represent the underlying image space.

Especially Wasserstein-GAN solved the training difficulties in a significant way, I didn't think GAN training would be time efficient for the current problem.

U-nets have been successful in image segmentation and enhancement. They are given this name, due to their "U" shape. It can be though as an autoencoder, where we gradually decrease the size of the image by maxpooling after convolution layers and then increase it by upsampling to the original size. Nevertheless, in such setting the biggest problem is the lost fine details during upsampling. If we call the down-sampling path as encoder and upsampling path as decoder, U-net solves this problem by having the skip connections between the same-sized images in encoder and decoder path by concatenating them. Moreover as the size of the image gets smaller, the number of channels get larger, so that when we upsample, the information in larger amount of channels is combined to less number of channels so that information can be protected to a very good degree.

U-net would be a natural choice in this task. For this model, I adapted the code from [6] and kept the original code to a high degree, which can be found in unet.py. I just wanted to experiment with this model to get a sense of its performance in this task, but since I decided to try another model additional to what I originally implemented (modelSR.py), I had to adapt from this source due to time constraint. Since the training was much slower than my implementation due to increased number of parameters, I had to stop the training before it reaches the stopping criteria. As can be seen in results section, its performance is worse than simple CNN network, but it might be due to lack of convergence. In this project, a U-net of depth 5 is used and the channels are created as exactly described as in [5].

Loss Function

Since the aim for the model is bringing the input image as close as to the output image, the straight-forward loss function is Mean Sqaured Error (MSE) loss.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Minimizing MSE also maximizes the peak signal-to-noise ratio (PSNR), which is a common measure used to evaluate and compare Super Resolution algorithms.

PSNR=10log10(MAX_I/MSE)

where MAX_I means the maximum possible pixel intensity value. However, as explained in [7], pixel-wise loss functions such as MSE have problems to operate with the uncertainty inherent in recovering lost high-frequency details such as texture. Minimizing MSE encourages finding pixel-

wise averages of plausible solutions and they are typically overly-smooth with poor perceptual quality. In [7] they used GANs to overcome this, but another possible loss function to overcome the shortcomings of MSE is Structural Similarity (SSIM) index. SSIM is a perception-based method, which incorporates luminance, contrast and structure. It changes between -1 and 1 and 1 can be reached if there are two identical sets of data and therefore it indicates perfect structural similarity [10].

Pytorch doesn't have a built-in SSIM loss function as tensorflow has, but in the community [8] is highly used, and that's why I used that repository as loss function. The loss can be entered as command line argument as "SSIM" or "MSE".

(Other than SSIM loss and U-Net model codes, all the code for this challenge is written by me)

For the stopping criteria, I used early stopping based on a patience of 5 epochs. I did validation loss error in ever 60 batch iteration to save the best model, if the current validation loss is the lowest, but for stopping criteria, I used validation loss at the end of the epochs, since with mini-batches the loss progress can be very noisy(oscillating) and checking frequently and counting based on the frequent checks might be misleading. That's why after checking at the end of each epoch if there isn't any progress in 5 epochs, the training is stopped. I didn't look for consecutive 5 epochs, since the loss can increase oscillating, but the important point would be minimum value.

Hyperparameters

As optimizer I used Adam optimizer and for the initial learning rate I chose 0.001, since in the original paper this is the suggested parameter[9]. I chose this optimizer, since for each parameter the optimizer adapts the learning rate individually, while keeping the average of recent magnitudes of individual weights into account. It has been shown that it improves learning procedure significantly.

For this problem I didn't use any regularization, since I didn't have enough time to experiment and apply regularization to improve the performance in case there is need (ie in case of overfitting, l2-regularization improves the performance).

I used a batch size of 16 with the simple CNN network and 4 with U-Net since the GPU memory didn't allow more in both cases. Stochastic Gradient Descent has shown to be better then using the whole training dataset for each step, since there might be redundancy in the data so that by using less data we can move faster. In other words, instead of using the whole data set and taking one step, using one sample and moving faster even if in a more oscillating way has shown to be better for training. Nevertheless, in order to decrease the oscillations in the optimization trajectory using mini-batches is proposed and is excessively used in practice.

Lastly, the kernel size is another choice to be done for convolutional layers. There are two aspects: first, using a larger kernel can give relationships of larger regions. However, using small

kernels are advantageous: the convolution operation can be written as matrix multiplication of the image with a circulant matrix, whose columns are created by the zero-padded kernel. Circulant matrices are diagonalizable with discrete Fourier transform (DFT), where the diagonals of the diagonal matrix is the Fourier transform coefficients of the kernel. Due to space-frequency duality, if a filter is short, it covers more frequencies (dirac delta has all frequency components). Similarly if a filter is long, then it is not going to cover most of the frequency component, which will likely to end up having more 0 Fourier coefficients. Having a weight matrix, which has 0s on the diagonal after diagonalization, will likely to end up vanishing gradient, so that using shorter kernels are advantageous. Lastly, using larger kernels also computationally challenging. That's why usually in the literature kernel sizes of 3 or 5 are used. I followed [1] for having 5x5 kernels in the first layer and then 3x3 kernels in the following layers.

Results

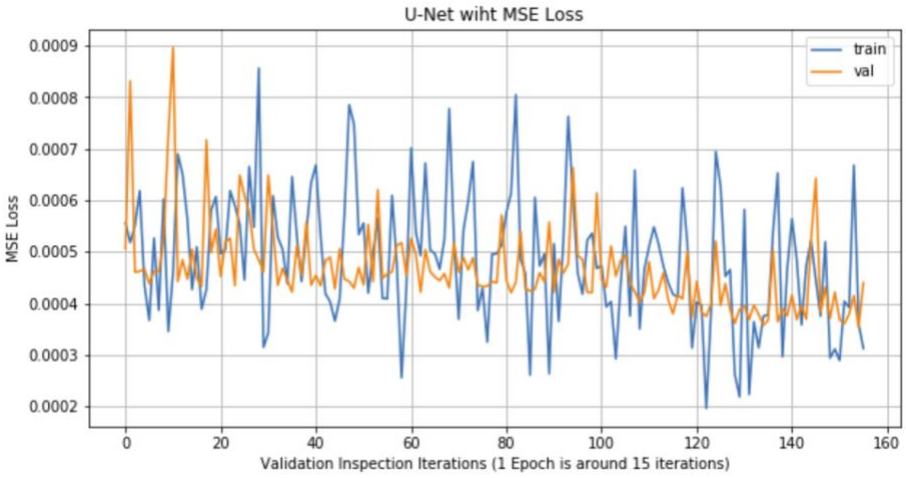
I prepared hdf5 files by concataneting all training volumes (and test volumes respectively) and then training data is split as 88% of the first 15 data and the rest is used for validation. I inspected validation loss in every 60 * batch_size iterations, which caused the graphs to be different scale for U-net and simple CNN architecture.

Training Graphs

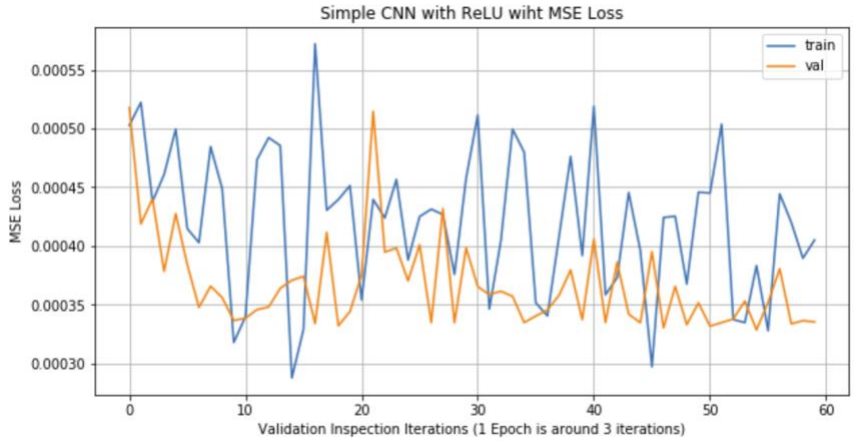
In this section, the training process of all models with the corresponding loss can be seen. I will comment on them as a last paragraph after giving all of the figures. In the x axis, the iteration numbers are labeled along with the explanation of where an epoch would correspond, where epoch is defined as 1 iteration through whole training data set.

U-Net with MSE Loss

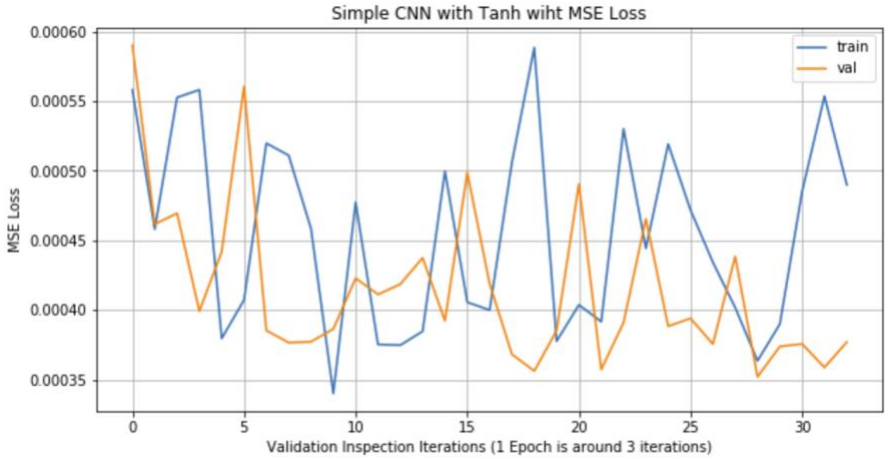
I stopped training after 10 epochs due to time restrictions before convergence.



Simple CNN with ReLU and MSE Loss



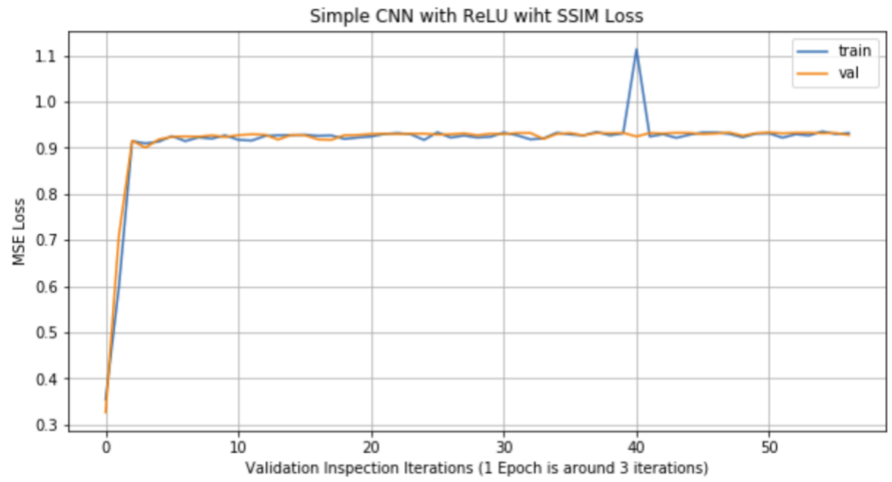
Simple CNN with Tanh and MSE Loss



Simple CNN with ReLU and SSIM Loss

As can be seen in the figure, model took little longer to come to convergence level, but since the range of the y-axis is larger than the other graphs, this training seems to be smoother, but it is due to the changed range.

Different than MSE, for SSIM, the loss is maximized and the maximum possible value is 1. During training it got a value larger than 1, which might be due to a bug in the code I got from repository. As a next step, I will try to debug it.



With tanh non-linearity, the stopping criteria is reached earlier than the ReLU, regardless of the loss function and as can be seen in the next section, the results are worse with tanh. Either the stopping criteria should be changed for this model or a different learning rate can be applied.

As stated before U-Net is stopped externally due to the long training duration, that's why a direct comparison about training wouldn't be correct.

Test Dataset Results

As can be seen from the test results the best model performing in MSE does not necessarily the best in SSIM. MSE and PSNR are directly related, that's why they are giving the same information. The best MSE performing model is Simple CNN w ReLU, which is trained with MSE loss, but it is the worst in SSIM loss. The best SSIM loss performing model is Simple CNN w ReLU, which is trained with SSIM, but it is the second best in MSE loss. Even if it is expected that the models which are trained on the specific loss are outperforming each other in that class, the fact that Simple CNN w ReLU(SSIM) is the second best in MSE, shows that SSIM is a better loss to train.

Models	MSE	PSNR (dB)	SSIM
Simple CNN w ReLU, Loss: MSE	0.000322	35.107	0.9185
Simple CNN w Tanh, Loss: MSE	0.000354	34.685	0.9194
U-Net, Loss: MSE	0.000364	34.590	0.9204
Simple CNN w ReLU, Loss: SSIM	0.000346	34.846	0.9262

*Loss in the left column indicates the training loss.

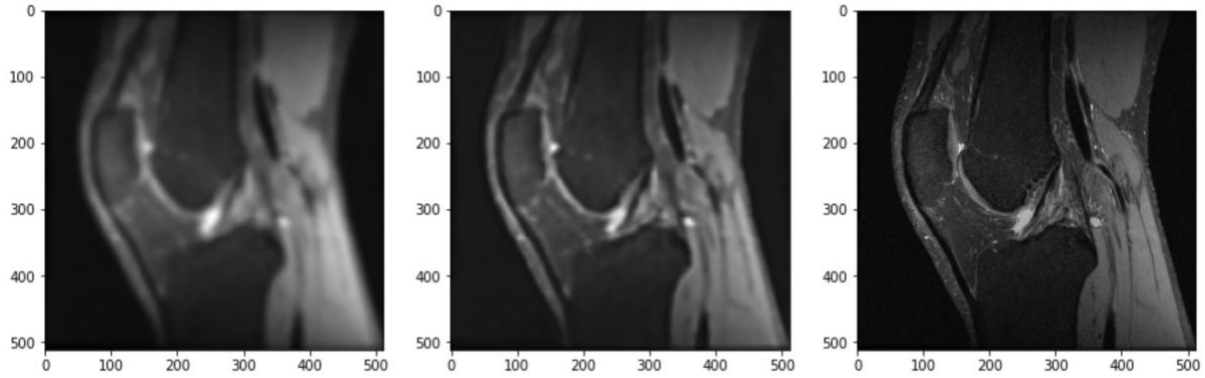
Image example

Here I'm showing the 126th slice (mid-slice) of 17th volume (middle) cleaned with each model alongside the blurred(left) and original versions(right). The metrics are shown above the images.

In general, there is significant difference between the cleaned image and the original image for all models. Nevertheless for Simple CNN w ReLU, Loss: MSE and Simple CNN w ReLU, Loss: SSIM, the perceptual quality is much better. With the other two, especially the luminance is apparently worse. For this slice Simple CNN w ReLU, Loss: SSIM gave the best results in all metrics, which I would also agree perceptually.

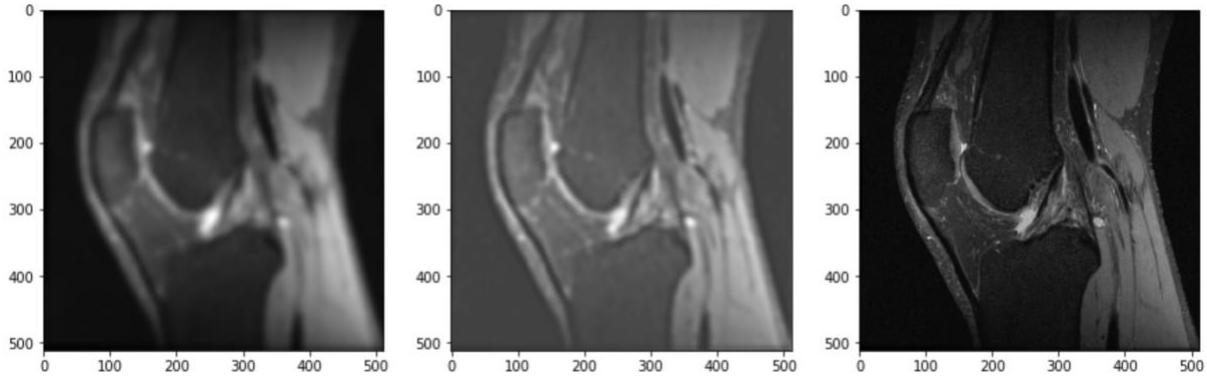
Simple CNN w ReLU, Loss: MSE

```
SSIM:0.7542              SSIM: 0.9153
PSNR: 23.908dB           PSNR: 37.815dB
MSE: 0.00407             MSE: 0.000165
```



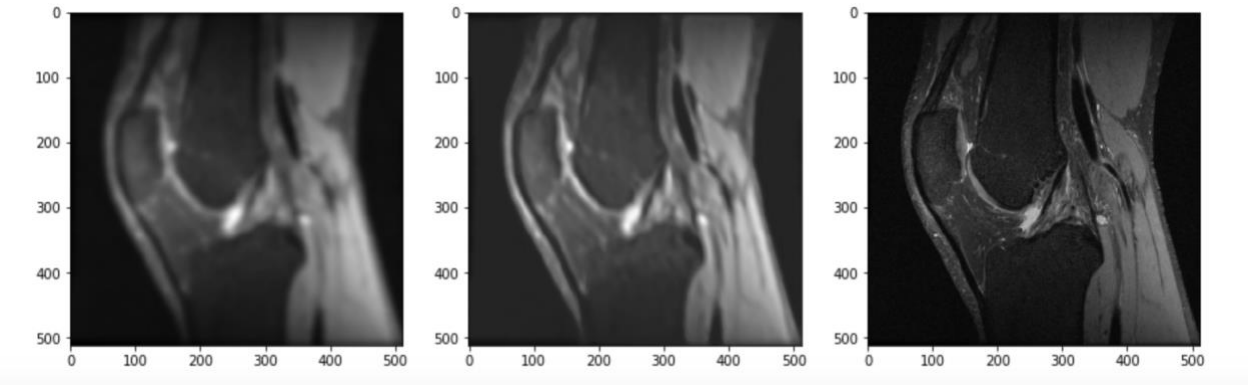
Simple CNN w Tanh, Loss: MSE

```
SSIM:0.7542              SSIM: 0.9118
PSNR: 23.908dB           PSNR: 37.650dB
MSE: 0.00407             MSE: 0.000172
```



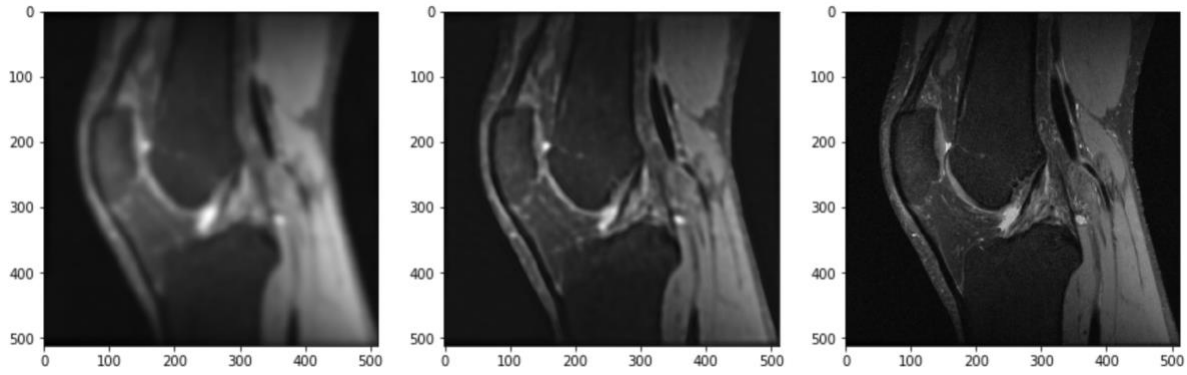
U-Net, Loss: MSE

```
SSIM:0.7542              SSIM: 0.9114
PSNR: 23.908dB           PSNR: 37.270dB
MSE: 0.00407             MSE: 0.000187
```



Simple CNN w ReLU, Loss: SSIM

```
SSIM:0.7542              SSIM: 0.9223
PSNR: 23.908dB           PSNR: 38.008dB
MSE: 0.00407             MSE: 0.000158
```



Appendix – Codes

- task1_check.py: For Task 1 and explained in the Report. It is a command line tool. If typed python task1_check.py –h, the arguments are explained.
- task1_part1.py: For Task 1 and explained in the Report. It is a command line tool. Using –h, the necessary arguments can be seen.
- task1_part2.py: For Task 1 and explained in the Report. It is a command line tool. Using –h, the necessary arguments can be seen.

- task2.py: For Task 2 and explained in the Report. It is a command line tool. Using –h, the necessary arguments can be seen.
- utils.py: The auxiliary functions for Task 1 and 2.
- blur_script.sh: The shell script I used to generate the 3D volumes.
- pytorch_ssim: Repository[8] for SSIM loss
- pytorch-ssim-master: Repository[8] for SSIM loss
- modelSR.py: The simple CNN model I wrote, which is tunable. The explanation is done in the report.
- unet.py: The U-net model
- main.py: The main function for training. Using –h the necessary arguments can be seen in the command line. However, the model should be defined in this script, either as U-net or simple CNN with desired layers, it is not defined as command line arguments.
- train_utils.py: The auxiliary functions for main.py. Since the training code takes the data in a specific format (concataneted 3D volume) with the ground truth data, the data location is hard-coded as location in the training code, nevertheless, I wrote another script only for inference 'clean_data_w_model.py' as command line tool, where any 3D volume as hdf5 file can be given to the model for cleaning. I created another data loader which takes only blurred data.
- clean_data_w_model.py: For cleaning blurred data, where the output is written as hdf5 file. By using –h in command line the necessary arguments can be seen.
- inference_utils.py: The auxiliary functions clean_data_w_model.py

References

[1] W. Shi, J. Caballero, F. Huszar, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[2] Y. Chen, F. Shi, A. G. Christodoulou, Y. Xie, Z. Zhou, and D. Li, "Efficient and Accurate MRI Super-Resolution Using a Generative Adversarial Network and 3D Multi-level Densely Connected Network," Medical Image Computing and Computer Assisted Intervention – MICCAI 2018 Lecture Notes in Computer Science, pp. 91–99, 2018.

[3] C. You et al., "CT Super-resolution GAN Constrained by the Identical, Residual, and Cycle Learning Ensemble (GAN-CIRCLE)," 2018. Available: https://arxiv.org/abs/1808.04256

[4] H. Dong, G. Yang, F. Liu, Y. Mo, and Y. Guo, "Automatic Brain Tumor Detection and Segmentation Using U-Net Based Fully Convolutional Networks," Communications in Computer and Information Science Medical Image Understanding and Analysis, pp. 506–517, 2017.

[5] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolu- tional networks for biomedical image segmentation. In MIC- CAI, pages 234–241. Springer, 2015

[6] https://github.com/jvanvugt/pytorch-unet/blob/master/unet.py

[7] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, et al. Photo-realistic single image super-resolution using a gener- ative adversarial network. arXiv preprint arXiv:1609.04802, 2016.

[8] https://github.com/Po-Hsun-Su/pytorch-ssim

[9] Kingma, Diederik P and Ba, Jimmy Lei. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[10] https://en.wikipedia.org/wiki/Structural_similarity