Exploring FastMRI

Nikhil Santokhi

September 2025

Contents

1 Introduction

Magnetic Resonance Imaging (MRI) is a non-invasive imaging technique that provides high-quality images of the body's anatomy. It is a powerful diagnostic tool used for the detection of various conditions but is limited by its slow acquisition time.

The slow scans cause several issues such as patient discomfort, which causes involuntary movement, especially in elderly and critically ill patients. This creates motion artifacts in the final image. If the final image isn't discernible, a doctor wouldn't be able to distinguish any precise conclusions, and thus the scan would be a waste of time, money, and possibly put patients' lives at risk if any disorders go unnoticed. Clinical throughput is also very slow, meaning fewer patients are scanned per day, leaving the vulnerable waiting. Due to these implications, accelerated MRI techniques have been developed to help reduce scan time.

Inside the scanners there are multiple receiver coils placed around the patient for better coverage of the body. Each coil is sensitive to signals from nearby tissue, providing slightly different perspectives of the same anatomy.

The machine generates a strong magnetic field to align the hydrogen protons in the body, and a radiofrequency pulse is sent in, knocking the protons out of alignment. As the protons relax back into alignment, they emit signals, which are picked up by the coils. These signals are processed, then converted into raw data known as k-space.

The scanner acquires the body in slices. Each slice corresponds to a thin cross-section of the anatomy. For every slice, each coil acquires its own k-space. That means if you scan with 12 coils and acquire 30 slices, you obtain 12 separate k-spaces for each slice (360 in total). When all 30 slices have been converted into a real image from the k-spaces acquired from the coils, you can view it as a 2D stack of 30 images or put it into software that will present it as a 3D volume.

To reduce scan time, the MRI machine is set to retrieve undersampled data, meaning the scan is quicker but the k-space obtained data will contain less information than a usual scan would. But these undersampled k-spaces lead to aliasing and blurring in the final images when reconstructed normally. To combat this, advanced reconstruction algorithms have been developed to recover high-quality images from undersampled data.

Classical approaches that address these issues are Parallel Imaging (PI) and Compressed Sensing (CS) [1]. These two methods can be effective but often require iterative solvers and don't generalise well to different anatomies. The combination of PI and CS has the potential to improve MRI scan time and image construction but is still held back by their limitations. The reconstruction from the under-sampled k-space data has remained a major problem in this domain.

Deep learning has emerged as a powerful solution for accelerated MRI [2]. Recent studies and advances have introduced data-driven methods and MRI physics integration for image reconstruction using severely under-sampled data, outperforming traditional techniques in terms of speed, image quality, and robustness. This paper explores different deep learning methods, the accompanying GitHub repo can be found here.

2 fastMRI

2.1 Dataset

The fastMRI dataset is large-scale public dataset created by the NYU School of Medicine consisting of MRI data from the knee, brain, prostate and breast. A dataset made for the intended use of research and development in improving the reconstruction of MR images from under-sampled data [3]. In this paper the focus will be on the multicoil knee MRIs.

The multicoil knee MRIs are all HDF5 files. Each file corresponds to one MRI scan and contains the k-space data, ground truth or mask, and some meta data related to the scan. The train and validation files have the same structure, whereas the test files have a slightly different one. Each of them contain three pieces of information with two being shared across all three sets.

Train, test and validation data all consist of: `ismrmrd_header` and `kspace`. The train and validation sets have `reconstruction_rss` whereas the test set contains `mask`.

- **ismrmrd_header:** Contains information about the MRI machine such as time of scan or the position of the patient but all patient information has been anonymised via conversion to the vendor neutral ISMRMRD format [3, 4].
- **kspace:** The training and validation k-space data is fully sampled and will need to be under-sampled during training, the k-spaces in the test set are already under-sampled and don't require any other preparation. Both have the shape (number of slices, number of coils, height, width). MRIs are acquired as 3D volumes, so the first dimension is the number of 2D slices.
- **reconstruction_rss:** Fully constructed, fully-sampled MR images which act as the ground-truth labels only in the train and validation HDF5 files. These images have been cropped to 320 x 320.
- **mask:** Holds the information that defines the under-sampled k-space mask. Found only in the test files.

In the GitHub repo, along with this paper, there is a script called *h5_inspect.py* that displays this information and more.

```
ismrmrd_header (Dataset)
  || Shape: ()
  || Dtype: object
  || (Could not read data: 'bytes' object has no attribute 'size')
kspace (Dataset)
  || Shape: (36, 15, 640, 368)
  || Dtype: complex64
  || Preview: [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j] ...
mask (Dataset)
  || Shape: (368,)
  || Dtype: bool
  || Preview: [False False  True  True False] ...
...
...
```

Above is an example output using the file *file1000683.h5* which comes from the test set so it has information on `mask` instead of `reconstruction_rss`.

Knee Dataset	Volumes	Slices
Train	973	34742
Validation	199	7134
Test	118	4092

Table 1: Split of the dataset

Download the dataset from here then follow the instructions and complete the form. An email will be sent with details on how to download [3].

2.2 fastMRI API

The fastMRI GitHub repo can be found here.

Inside the repo there are low-level utility functions, for example Fourier Transforms. These will be discussed in greater detail in Section 4.

fastMRI utilises high-level libraries such as PyTorch to create neural networks and PyTorch Lightning to train them.

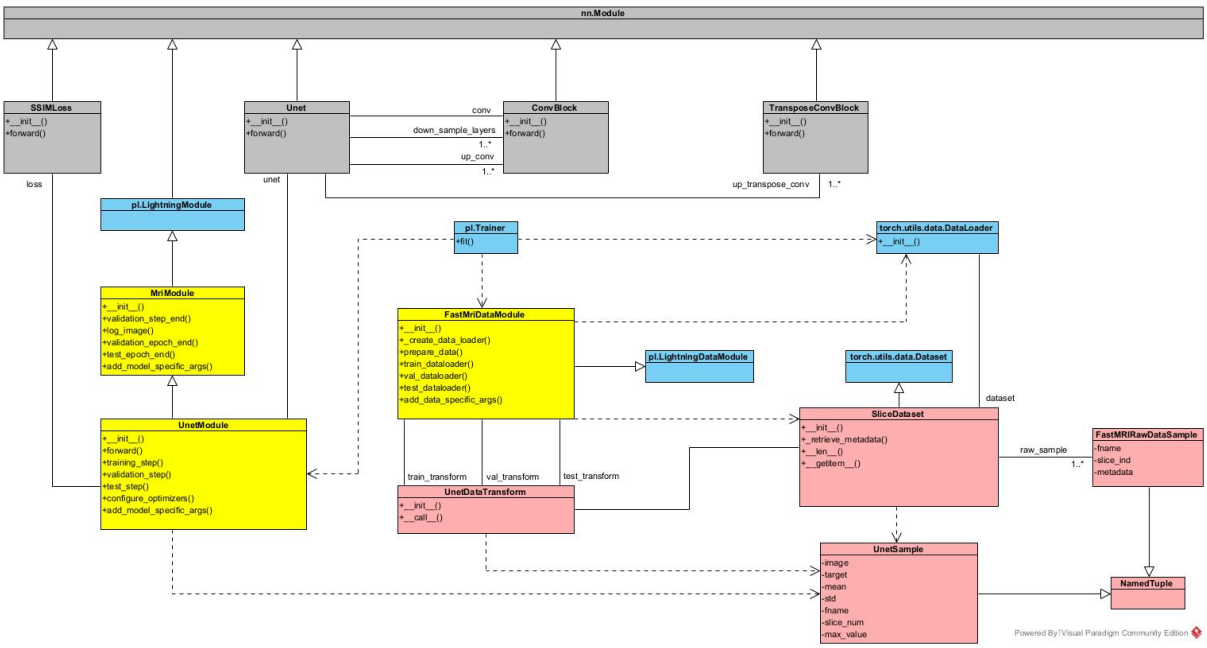


Figure 2.1: UML diagram of the Unet model

All 3 models covered later in Section 5 follow the same design pattern using PyTorch and PyTorch Lightning for training and loading the dataset.

The grey blocks symbolise the neural network classes, the yellow ones are the PyTorch Lightning derived classes, and the pink classes are dataset-related.

When `pl.Trainer.fit()` is called, the model goes through a startup process explained below.

`FastMriDataModule` inherits from PyTorch Lightning's `pl.LightningDataModule` to create a class that prepares the data, and defines parameters such as: batch size, number of workers, etc. After defining the parameters inside the `__init__()` method, each data loader must be configured using `_create_data_loader()` shown by the dependency between the `FastMriDataModule` and `torch.utils.data.DataLoader`. Refer to the startup sequence diagram below to see the order of operations; the instance of `SliceDataset` is different for each data loader. The instance for `train_dataloader()` will contain the training data and the instance for `val_dataloader()` will contain the validation data.

To load the dataset we need a new PyTorch derived class, `SliceDataset` which inherits from `torch.utils.data.Dataset`. Inside this class, there are three required methods: `__init__()`, `__len__()`, `__getitem__()`.

Inside the `__init__()` method, the dataset is initialised by loading the file path, targets, defining any data transforms which in this case come from `UnetDataTransform` and any other related metadata.

The `__len__()` method simply returns the number of samples in the dataset to let the data loaders know how many batches to produce.

Then `__getitem__()` takes an index as an input and returns a single data sample corresponding to that index from the dataset which is accessed from memory. Once the sample has been loaded, the predefined transforms will be applied to it, and then the sample will be returned as a dictionary.

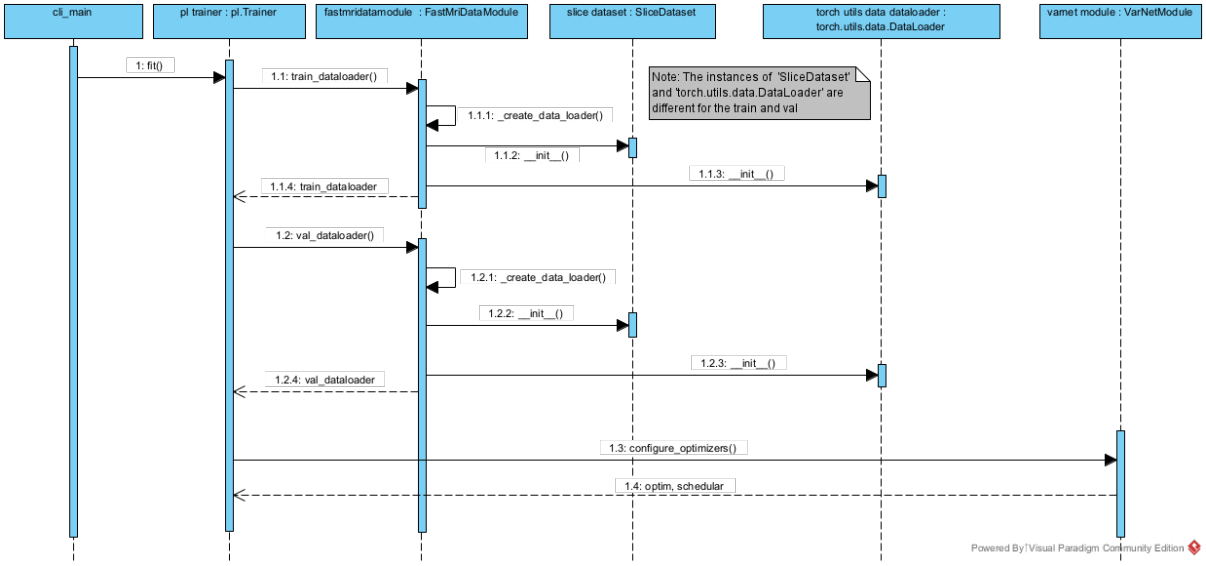


Figure 2.2: Sequence diagram showing VarNet startup

Training step-by-step:

1. First `__getitem__()` grabs and returns a sample.
2. A batch is then passed through the `training_step()`.
3. The `training_step()` calls the `forward()` method of the module.
4. Which in turn calls the `forward()` methods of the neural network.
5. After each epoch, loss is calculated and backpropagated to update the model parameters.
6. Steps 1-5 repeat until all epochs are completed.

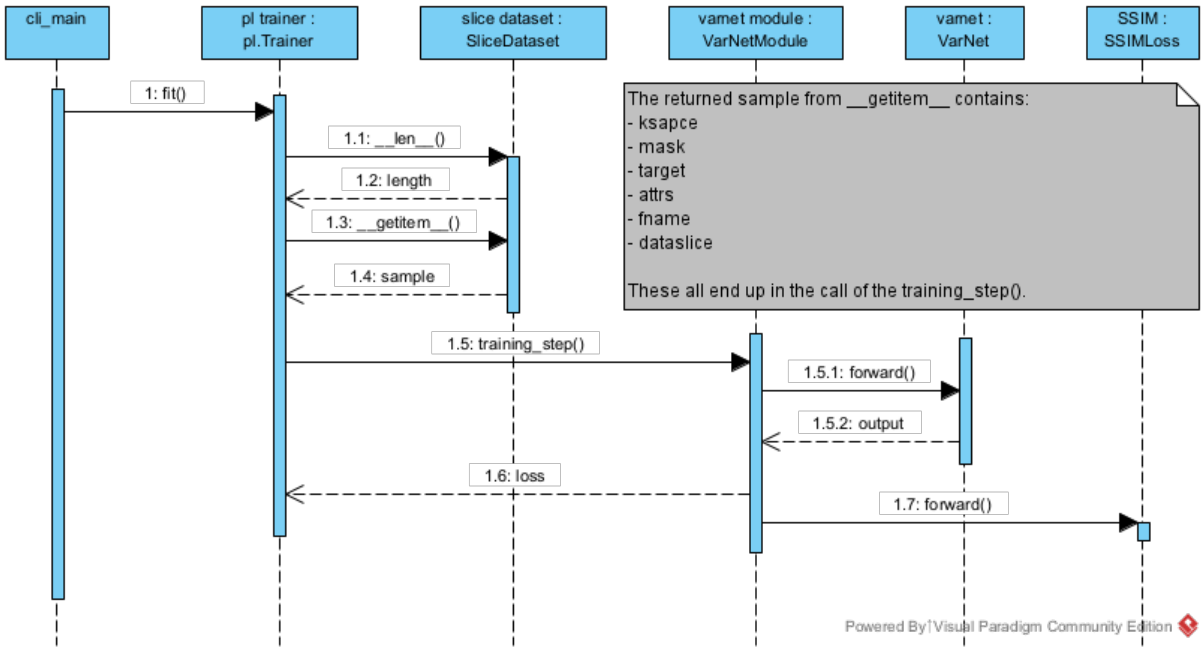


Figure 2.3: Sequence diagram showing the training loop of VarNet

3 K-Space and Masking

In this section, we look at a more in-depth analysis of k-spaces and how to undersample them for utilising fastMRI.

All of the k-space figures in this section and the next come from coils 0, 5 and 10 from slice 20 in the file *file1000167.hdf5* which is from the training set. It's shape is (30, 15, 640, 372).

3.1 What is a k-space?

In Section 1 it was mentioned that k-spaces are obtained from an MRI machine, but what exactly is a k-space?

As previously mentioned k-spaces are frequency encoded-data. The centre contains high energy with low frequencies and on the other hand the outer edges are the opposite, low energy with high frequency. This means that most of the information extracted from the k-space is in the centre, and therefore it usually remains fully sampled.

Due to their extremely dynamic range k-spaces are displayed using logarithms, this is so they can actually be visualised. The difference in frequencies from the centre to the outer areas would be too large displaying the k-spaces this way. The logs compress the range of the k-spaces. Low frequencies control the contrast and overall shape and on the other hand the high frequency controls the edges and finer details. Without utilising logs k-spaces would practically be invisible on a normal display and information would most definitely be lost. (When coding 1e-9 is added to prevent logs of 0 - as it is undefined).

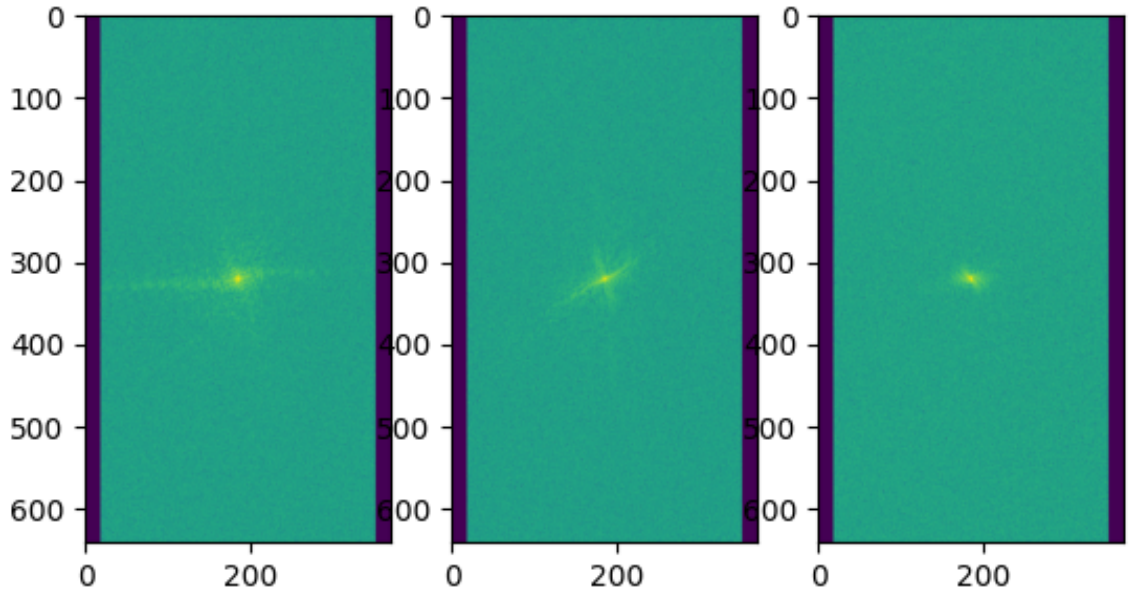


Figure 3.1: Fully sampled k-spaces

3.2 Undersampling

To obtain undersampled k-spaces a technique called masking is used which applies a binary mask over the k-space.

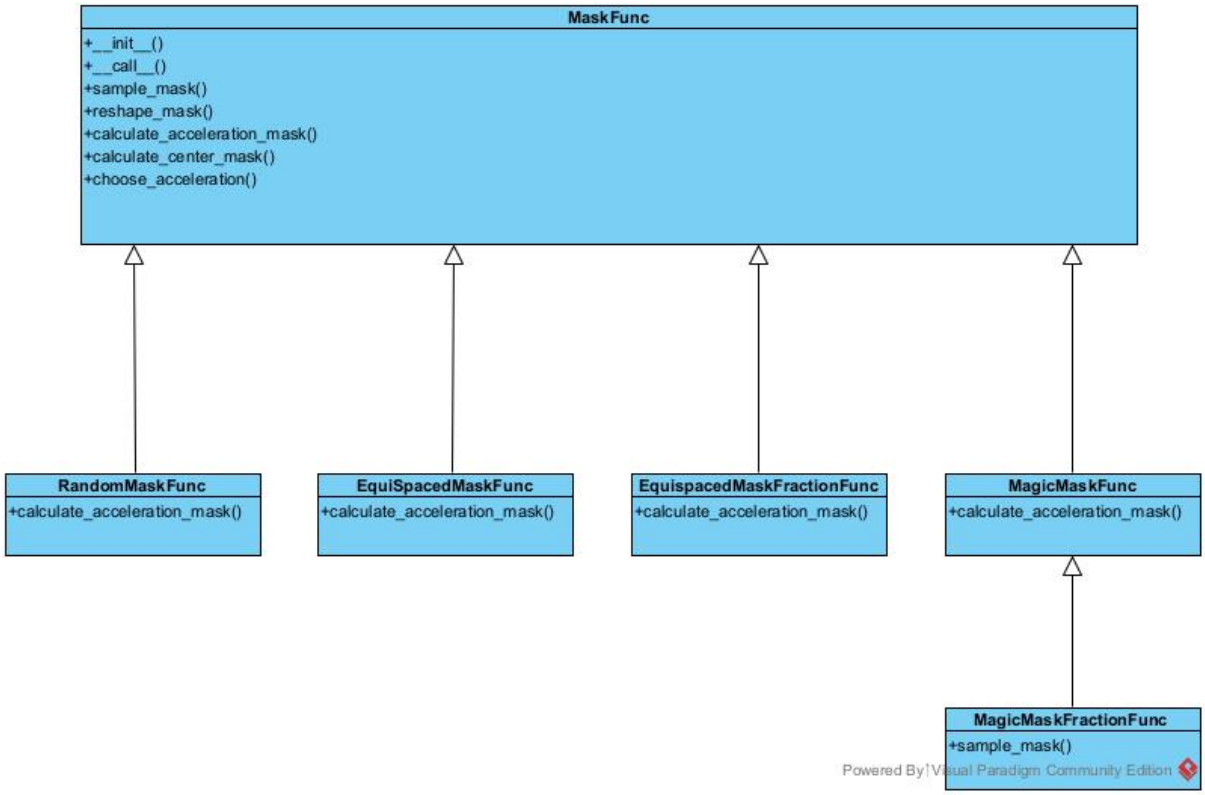


Figure 3.2: UML of the Mask Classes

Each different type of mask class is inherited from the abstract class `MaskFunc`. Every mask has its own method for acceleration, which is the amount of under-sampling applied to a k-space.

So the different k-space masking methods can be categorised based on their sampling patterns and how they control acceleration.

Acceleration selects certain vertical lines of the k-space, some to keep and some to remove for the mask. Speed of acceleration determines the amount of data that is kept and the amount which is lost.

So the higher the acceleration the faster the scan, which means more data is missing. After masking is applied, a subset of the previously fully sampled k-space is obtained.

The `MaskFunc` class serves as the base class for all k-space masking functions. It defines the general structure for generating subsampling masks and includes methods for selecting accelerations and centre fractions. Subclasses implement specific strategies for k-space sampling.

The `RandomMaskFunc` class creates a probabilistic k-space mask by randomly selecting high frequency lines while ensuring the expected number of selected lines matches the desired acceleration. A fully sampled centre region is always included.

The `EquispacedMaskFunc` class generates a mask with equally spaced k-space lines, similar to standard GRAPPA-style acquisitions. It selects lines at fixed intervals and allows an optional random offset to vary the starting position.

The `EquispacedMaskFractionFunc` class builds on equispaced sampling by adjusting the acceleration rate based on the number of low-frequency lines. This ensures that the total number of selected lines matches the intended acceleration, even if the spacing is not perfectly uniform.

The `MagicMaskFunc` class applies an equispaced mask while incorporating an offset for conjugate symmetry in k-space. By mirroring samples across the centre, it improves efficiency in reconstructions that can exploit approximate symmetry. However, the method typically results in a lower effective acceleration than initially intended.

The `MagicMaskFractionFunc` class enhances the `MagicMaskFunc` by precisely matching the target acceleration. It adjusts offsets and ensures that the number of selected k-space lines aligns with the desired acceleration rate while maintaining conjugate symmetry properties.

The script for all of this is in the fastMRI repo: *fastmri/data/subsample.py*

The `apply_mask` function from *fastmri/data/transforms.py* is responsible for subsampling k-space data by applying a given mask function. It takes as input a tensor data representing k-space, a `mask_func` that generates a mask based on the input shape, an optional offset for the mask pattern, an optional seed for reproducibility, and an optional padding argument to control which parts of k-space remain masked.

The function first determines the shape of the mask based on the last three dimensions of data. It then calls `mask_func` to generate a sampling mask and obtain the number of low-frequency lines included. If padding is specified, the function zeroes out specific regions of the mask. The mask is then applied to the k-space data by performing an element-wise multiplication, ensuring that masked-out regions are set to zero. The addition of 0.0 ensures that masked-out values remain numerically clean.

The function returns three values: the masked k-space data, the mask itself, and the number of low frequency samples. This setup ensures flexibility, allowing different types of masks (random, equispaced, symmetry-aware, etc.) to be applied while maintaining control over the subsampling behaviour.

3.3 Different Types of Masking

3.3.1 Random Mask

The `RandomMaskFunc` selects k-space lines randomly while ensuring the expected number of sampled lines matches the desired acceleration. It retains a fraction of low-frequency components and randomly selects the remaining samples with a calculated probability. This method introduces randomness, making it well-suited for deep learning-based reconstructions, but unsuitable for structured parallel imaging methods like GRAPPA.

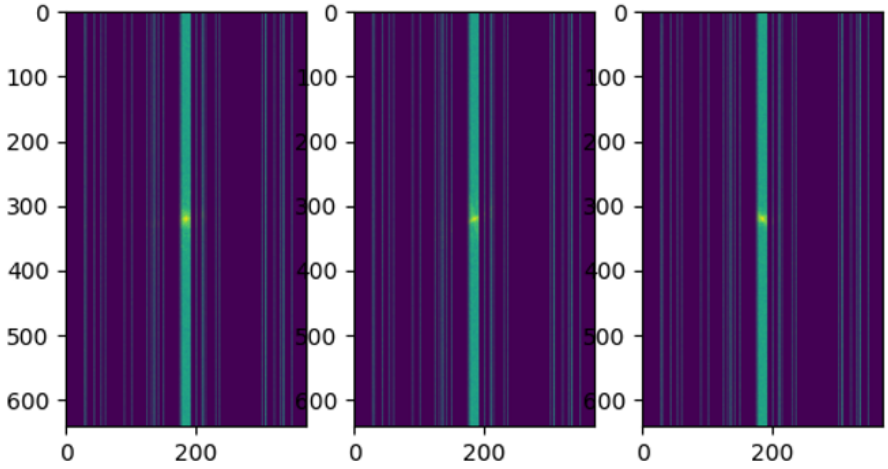


Figure 3.3: Random masked k-spaces

3.3.2 Equispaced Mask and Equispaced Fraction Mask

The `EquispacedMaskFunc` samples k-space lines at fixed, equal intervals. This structured approach is commonly used in GRAPPA-style acquisitions. However, since the central region is densely sampled, the effective acceleration may be greater than the intended value. If no offset is specified, a random offset is introduced to vary the sampling position. This method is ideal for traditional parallel imaging but lacks flexibility for deep learning applications.

The `EquispacedMaskFractionFunc` improves on the equispaced approach by adjusting the acceleration rate to account for the number of low-frequency lines. This ensures that the expected number of sampled lines matches the intended acceleration more precisely. However, unlike true equispaced sampling, the selected lines may not be perfectly uniform, which can impact GRAPPA-based reconstructions.

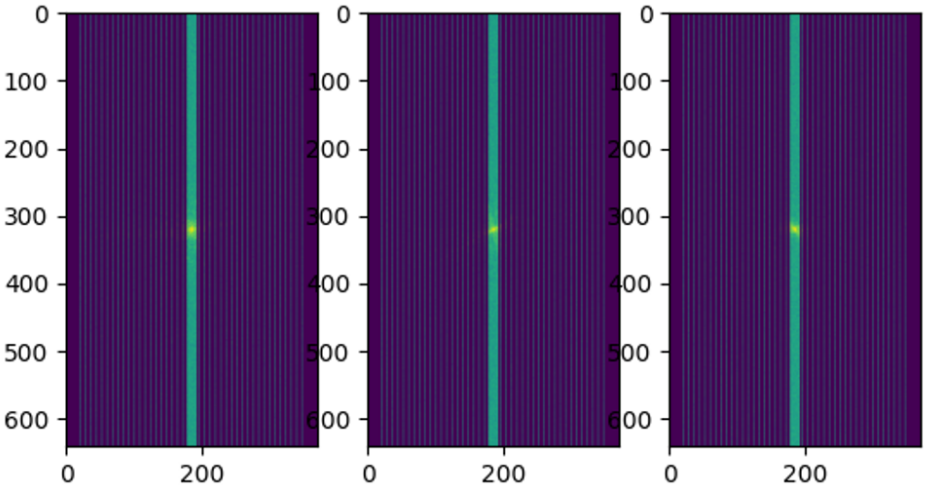


Figure 3.4: Equispaced masked k-spaces

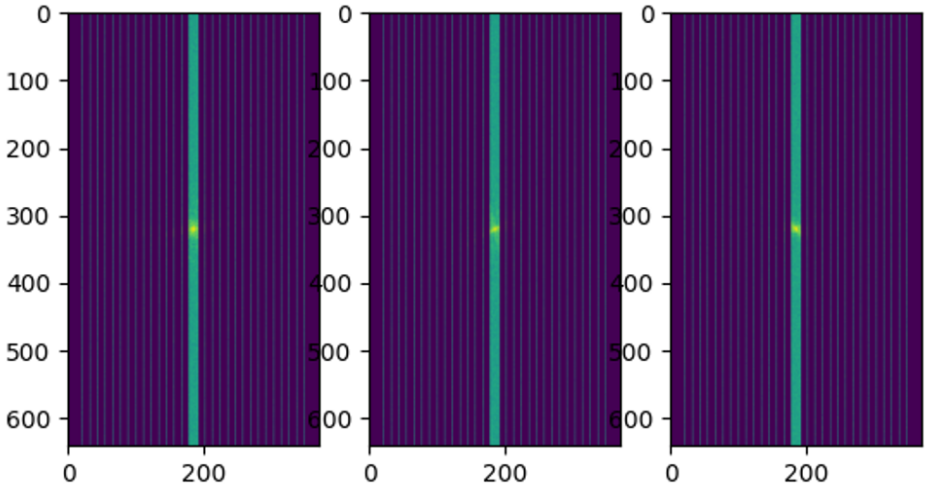


Figure 3.5: Equispaced fraction masked k-spaces

3.3.3 Magic Mask and Magic Fraction Mask

The `MagicMaskFunc` leverages conjugate symmetry by offsetting samples on both sides of k-space. Since MRI data often exhibits approximate symmetry, this mask improves efficiency compared to standard equispaced masks. The sampling pattern is similar to equispaced masking but with additional constraints to exploit symmetry. Like equispaced masks, it tends to underestimate the target acceleration.

The `MagicMaskFractionFunc` refines the `MagicMaskFunc` by precisely matching the target acceleration. It does this by adjusting offsets and ensuring that the total number of sampled columns aligns with the acceleration rate. This method is particularly useful when combining conjugate symmetry-based subsampling with deep learning. [5]

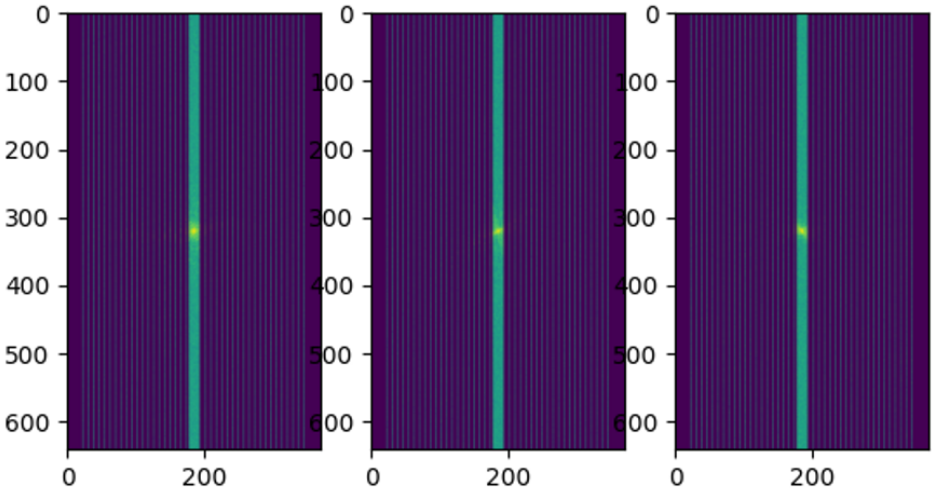


Figure 3.6: Magic masked k-spaces

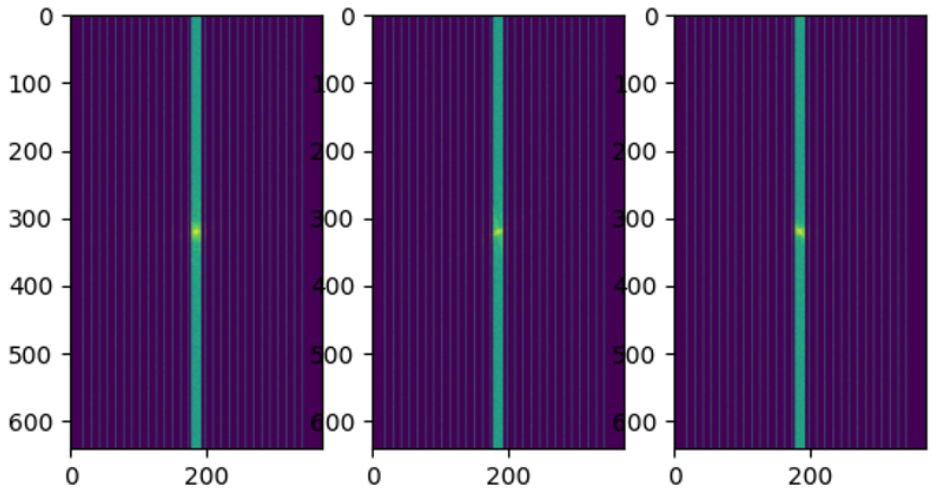


Figure 3.7: Magic fraction masked k-spaces

3.3.4 ACS Masking

The fully-sampled central lines of a masked k-space is called the auto-calibration signal (ACS) region. Therefore ACS masking keeps only this fully-sampled central region, and discards the rest.

Sensitivity maps for each coil are derived from the ACS region as they provide enough information to compute smooth coil profiles. ACS masking is mentioned in Section 5.2 and sensitivity maps will be explained further in Section 4.1.

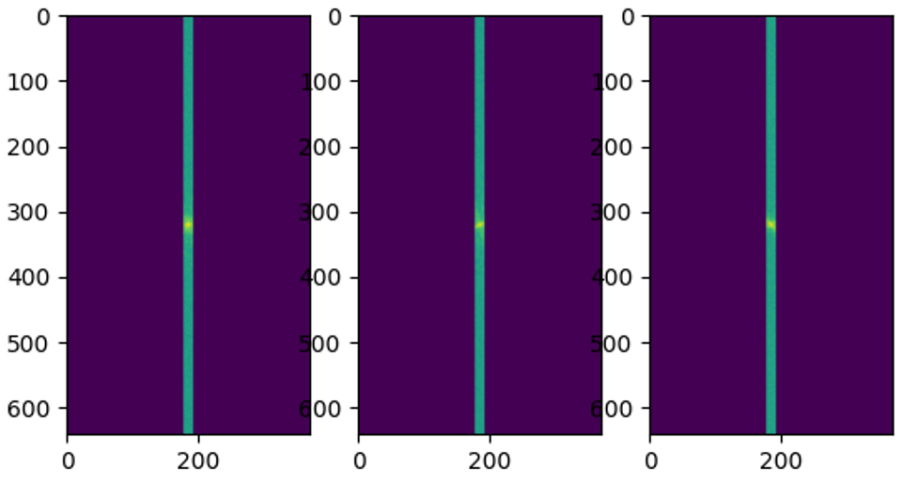


Figure 3.8: ACS masked k-spaces

4 Image Space

Section 1 had a very brief explanation of the image construction process, in this section we take a deeper look into this process.

4.1 Converting K-Spaces into Real Valued Images

To convert a k-space into a real image an inverse fast Fourier transform is applied to it. This also works in reverse; apply a fast Fourier transform to an IFFT image and it will turn back into a k-space.

Fourier transforms are used because a k-space is a measurement of spatial frequency, with the centre having low frequencies and the outer edges high frequency as said before in Section 3. So k-spaces encode spatial frequencies and signal intensities and applying an IFFT to a k-space will turn it from frequency domain to spatial domain. K-spaces can be viewed as the Fourier domain representation of the anatomy, which is why using it works so well.

After the IFFT is applied to a k-space, the image space is complex valued because the coils carry both amplitude and phase, so the absolute value of the image is taken so it is able to be visualised.

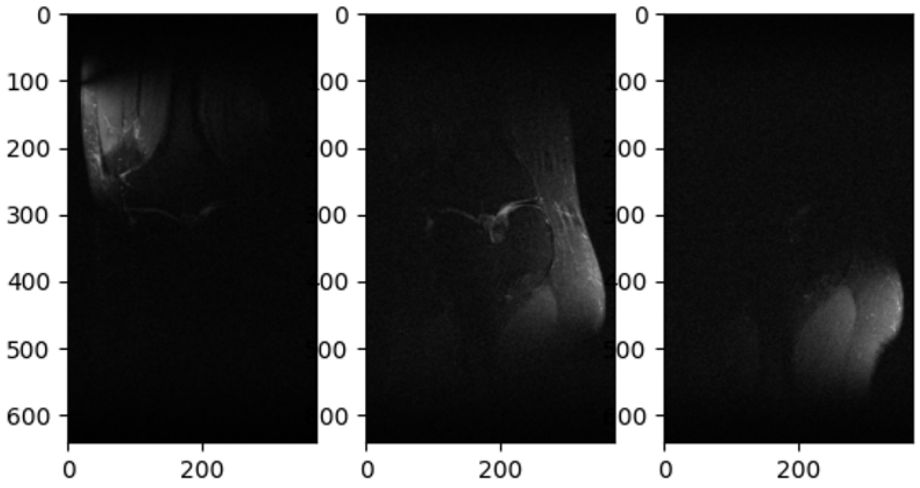
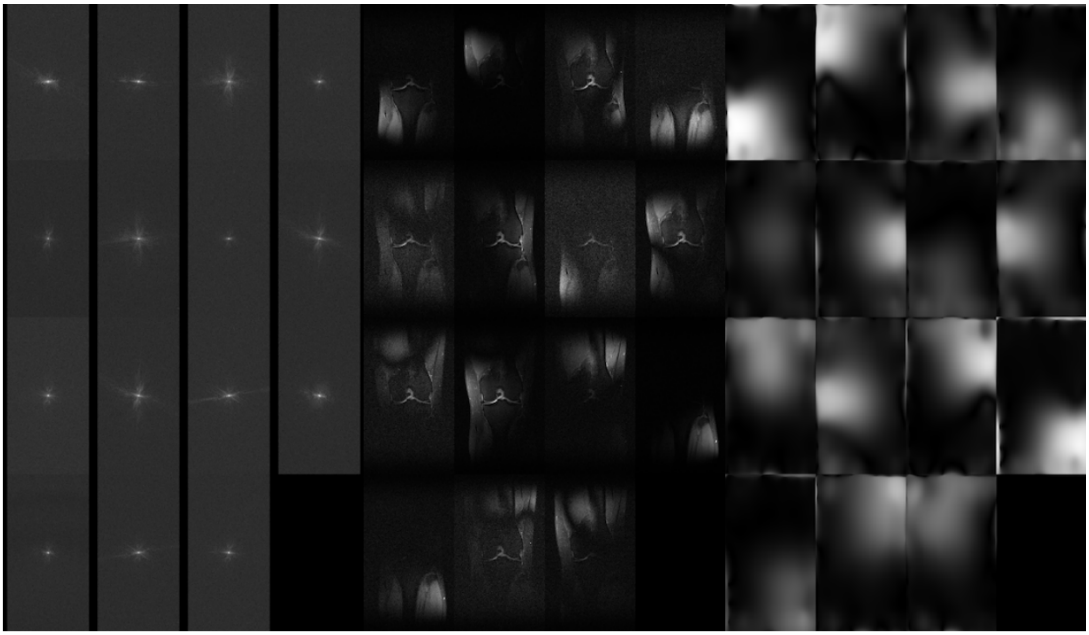


Figure 4.1: Fully Sampled IFFT images

Each coil sees different parts of the anatomy being scanned as there are many around the patient, which is why each IFFT image focuses on a different region. Each coil has its own spatial sensitivity profile which remains the same for different slices. They are more sensitive to signals near it and become weaker to the signals further away. These sensitivity profiles are visualised by 'sensitivity maps' with the focused area being 'shaded' in, they show the relative spatial weight for the region of which each coil is responsible for. Sensitivity maps are vital aspect in Section 5.2.



(a) k-space data from 15 coils

(b) Individual coil spatial images from fully sampled data

(c) Coil sensitivity map magnitudes given by ESPIRiT

Figure 4.2: Multi coil MRI reconstruction showing examples of sensitivity maps (c) [3]

The masked images will contain aliasing and be blurred - this is obvious as there is missing information from the k-space.

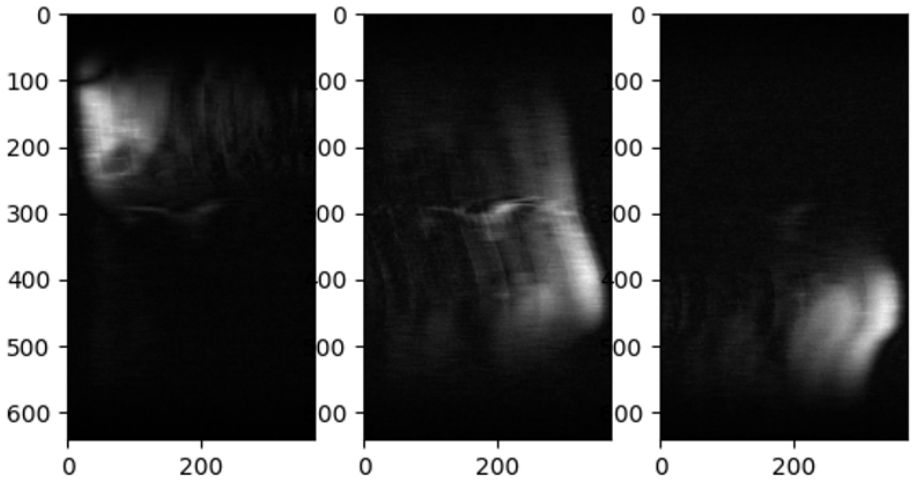


Figure 4.3: Equispaced mask IFFT images

4.2 Final Image Construction

As each coil focuses on a different region of the image, they need to be combined together to create the full image. This is done using the root-sum-of-squares (RSS) transform.

There are other ways of reconstruction such as SENSE, ESPIRIT, GRAPPA and CS but RSS is the preferred method in this scenario because it is simple and robust. Most scanners use RSS as the standard way to display the final MRI image.

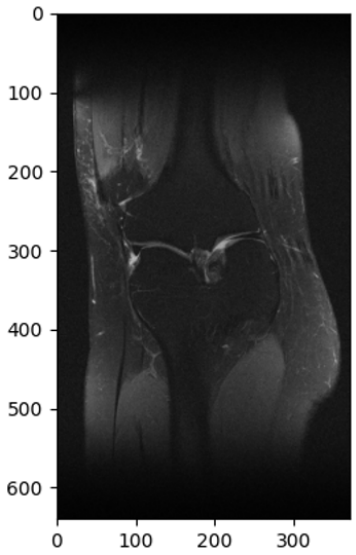


Figure 4.4: No mask RSS images

This is the complete 20th slice (out of 30) from the same file as before from all 15 coils.

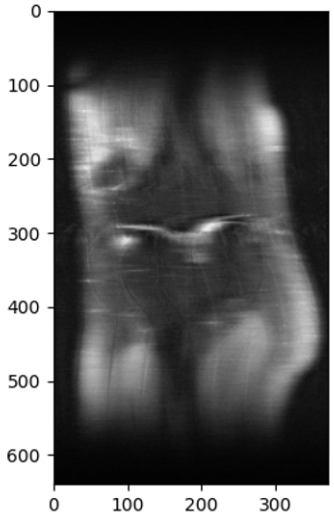


Figure 4.5: Magic mask RSS image

Just like the IFFT images from a masked k-space, the RSS image will also contain aliasing and be blurred.

Figure 4.6: Flowchart outlining the image construction process with shapes included

The k-space and image space are complex valued whereas the full MRI image is real.

These shapes can be seen when using *h5_ inspect.py*, once again coming from *file1000167.h5* which is the same file that was used in the previous sections.

4.3 Code Implementation

The following code is from *fastMRI_ tutorial.py* which is inside the fastMRI repo, it works through the implementation of everything covered so far this section. I have adapted this to create a file called *fastMRI_ masking.py* which explores all of the masking techniques previously shown in Section 3.3, conversion using IFFT and the final image construction with RSS which can be found in the repo for this paper.

The fastMRI repo contains some utility functions to aid in converting k-space into image space. These functions only work on PyTorch tensors, so they must be converted from numpy tensors to PyTorch tensors, this is done using the `to_tensor` function (located in *fastmri/data/transforms.py*).

The utility functions can be found in the *fastmri* folder, the important ones of note are: `ifft2c` which applies the inverse fast Fourier transform found inside of *fftc.py*, `complex_abs` which calculates the complex absolute value found in *math.py* and `rss` which performs the RSS transform located in *coil_ combine.py*.

The start of the tutorial file starts by going over loading and preparing the k-space data from the h5 file, but that will not be covered here.

Firstly, `show_coils` is a function which does exactly what it says, next the absolute value of the k-space is taken to remove the complex numbers. After this take the log of the k-space to actually visualise it and then finally add 1e-9 to prevent log 0 (explained in Section 3.1).

```
show_coils(np.log(np.abs(slice_kspace) + 1e-9), [0, 5, 10])
```

The utility functions explained before used in action:

```
slice_kspace2 = T.to_tensor(slice_kspace)
slice_image = fastmri.ifft2c(slice_kspace2)
slice_image_abs = fastmri.complex_abs(slice_image)
```

Below is the final line for image construction using the `rss` function, `np.abs` is just used as defensive coding, it is not mathematically required as the absolute value was taken before.

```
slice_image_rss = fastmri.rss(slice_image_abs, dim=0)
```

Simulate under-sampled data by creating the mask object and applying it to the k-space data. Then the same utility functions are applied to the masked k-space to construct the final MRI image.

```
from fastmri.data.subsample import RandomMaskFunc
mask_func = RandomMaskFunc(center_fractions=[0.04], accelerations=[8])
masked_kspace, mask, _ = T.apply_mask(slice_kspace2, mask_func)
```

```
sampled_image = fastmri.ifft2c(masked_kspace)
sampled_image_abs = fastmri.complex_abs(sampled_image)
sampled_image_rss = fastmri.rss(sampled_image_abs, dim=0)
```

5 Models

Section 2.2 explained each model's design pattern, but in this section, we take a deep look into each model's architecture. The loss function for all models is the structural similarity index (SSIM) [6].

5.1 Baseline Model - Unet

This model is based on the original paper, *U-Net: Convolutional Networks for Biomedical Image Segmentation (Ronneberger et al., 2015)* [7], which serves as the backbone for the models to come. It proposed the use of a fully convolutional encoder–decoder network specifically designed for biomedical image segmentation. Despite not being directly related to MRI reconstruction, the key ideas and contributions can still be applied, as both tasks require high spatial precision and the ability to balance global context with local detail preservation. The success of U-Net in biomedical segmentation laid the groundwork for its widespread adoption and adaptation in MRI reconstruction research.

The architecture of a U-Net consists of two pathways: a contracting path, the encoder, and an expanding path, the decoder, with skip connections between the two to preserve fine-grained spatial details, which would otherwise be lost during downsampling.

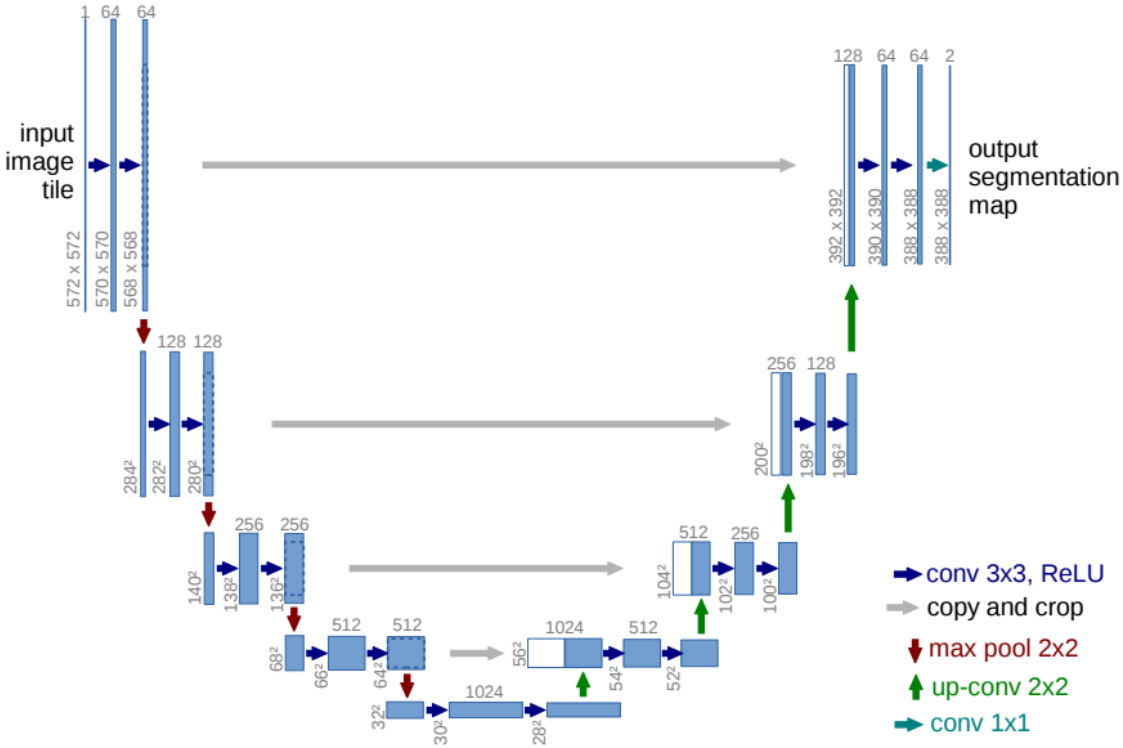


Figure 5.1: U-Net architecture [7].

The contracting path progressively reduces the spatial dimensions of the input by increasing the feature channels, all while capturing semantic information using convolutions. The expanding path reconstructs the spatial resolution using up-convolutions and skip connections to preserve finer details and to combat the vanishing gradients.

The contracting path uses an amalgamation of convolution (unpadded) and max pooling layers. The encoder starts off by applying two 3x3 convolution layers followed by a ReLU activation to the input image, which has one channel (greyscale image). These convolutions increase the number of channels to 64 to capture higher-level features. After this, a 2x2 max pooling layer with a stride of 2 is applied; this downsamples the feature map by half its size [8].

This downsampling process of applying convolutions and max pooling is repeated three more times, with the number of channels being doubled, reaching 1024, and the feature maps being downsampled by half their size every time to get to 28x28.

Next is the bottleneck, which processes the deepest features of the image. It is the connection between the encoder and decoder, where the spatial resolution of the image is at its smallest, but the feature representation is at its richest.

After the bottleneck, it's time for the upsampling process; this is done using both convolution and up-convolution operations to combine the learned high-level features along with the low-level features sent from the encoder path using skip connections. The decoder starts by applying a 2x2 up-convolution, which halves the number of channels and doubles the spatial dimension. After the up-convolution, a skip connection from the corresponding feature map from the encoder is concatenated to the upsampled feature map (the concatenated image must be cropped to match the decoder path's dimensions), which doubles the channels back to what they were before, so two 3x3 unpadded convolutions followed by a ReLU are applied to reduce the channels; then another 2x2 up-convolution is applied again, doubling spatial resolution and halving channels. This is repeated until the number of channels is back to 64, which in turn leads to the final 1x1 convolution layer to reduce the number of channels to the desired number of classes, followed by an activation. In medical imaging, a sigmoid activation is used for binary classification [8].

To understand how upsampling works using transpose convolutions, see *Up-Sampling with Transposed Convolution (Shibuya, 2017)* [9].

This is just an overview of a simple U-Net model, but many elements can be changed and improved. For example, the model used for fastMRI has the convolution layers being followed by an instance normalisation, LeakyReLU, and then a dropout layer instead of just a convolution followed by a ReLU. And the transpose convolution is also followed by an instance normalisation, Leaky ReLU, and then finally a dropout layer.

Refer to Figure 2.1 to see how the neural network functions and how each class interacts with one another.

U-Net is advantageous for MRI image reconstruction because it enables the recovery of high-quality images from under-sampled data, which is suitable for proper clinical use. But alone, U-Net is insufficient, so it is used as the foundational baseline for other models, shaping the trajectory of the field and used as the refinement backbone in state-of-the-art models.

5.2 E2E-VarNet

The End-to-End Variational Network [10] (E2E-VarNet) model builds on the foundations of the original VarNet model. To learn about VarNet, see *Learning a variational network for reconstruction of accelerated MRI data (Hammernik et al., 2017)* [11].

E2E-VarNet improves upon VarNet by learning the entire reconstruction process end-to-end, including the sensitivity maps. The architecture is built from cascades of refinement steps that operate in k-space rather than traditional methods that operate in image space. The model is structured as a sequence of cascades and consists of three modules: Sensitivity Map Estimation (SME), Refinement (R) and Data Consistency (DC).

The SME module starts by taking the ACS lines from the input k-space using an ACS mask. An inverse Fourier transform is then applied to the masked k-space to obtain the image space, which then goes through a CNN, specifically a U-Net. The output image space from the U-Net is divided by the RSS transform to acquire the final outputs for this module, which are the estimated sensitivity maps. These estimated sensitivity maps are used in the Refinement module.

The Refinement module is the core of the network that refines the current reconstruction. Input k-space is turned into image space using an inverse Fourier transform, the image space is then reduced into one image using the estimated sensitivity maps from the SME module. Next, the image is passed through a U-Net, which acts as a denoiser, after this, the image is expanded back into image space, once again using the estimated sensitivity maps. The last part is applying a Fourier transform, so the final output of the module is in k-space. The U-Net works on the image space of the data, but the refinement result is injected back into k-space.

The Data Consistency module is needed to prevent the intermediate k-space data from the Refinement module from drifting away from the actual acquired k-space data. To enforce this, the DC module computes a correction map that brings the intermediate k-space closer to the measured k-space values [10].

These modules work together in tandem with each other to create the end-to-end aspect of the network. The SME module estimates the sensitivity maps once at the start. Next, the network works through a sequence of cascades. Each cascade goes through the Refinement module, converting the data between k-space and image space, then the DC module applies its corrections to the intermediate k-space from the refinement step. The input for the Refinement module in the next cascade will be the output from the DC in the previous cascade. After several cascades, 10-12, the final k-space will have an IFT applied to it, then an RSS, and lastly be cropped for the final output.

E2E-VarNet integrates sensitivity estimation, learned image refinement, and strict data consistency into a unified, cascaded model, allowing it to reconstruct high-quality MRI images from heavily undersampled multi-coil data in a fully end-to-end fashion.

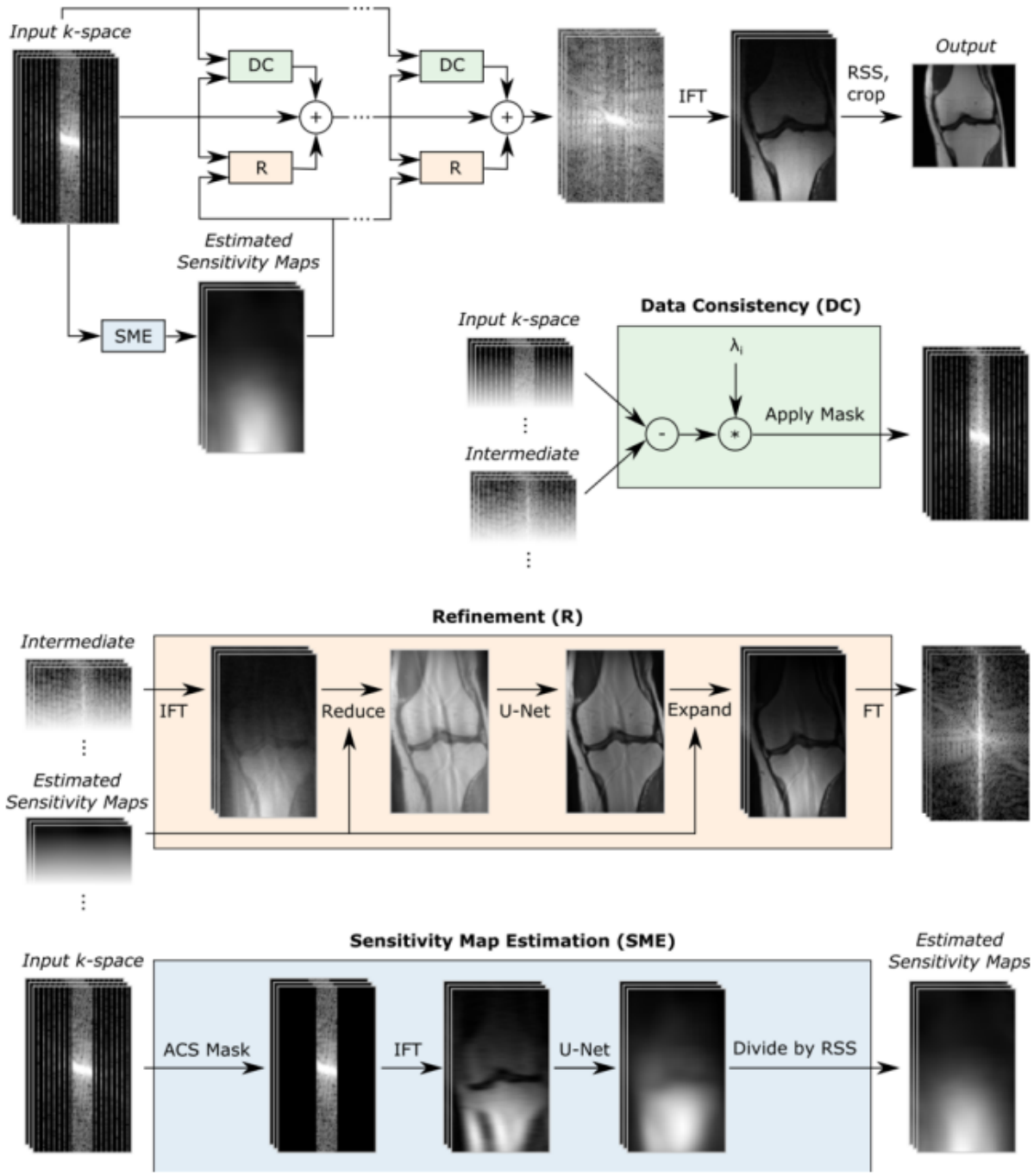


Figure 5.2: E2E-VarNet Architecture [10]

E2E-VarNet presents clear advantages as the whole construction pipeline is trained end-to-end, the sensitivity maps are learned directly from data without the need of external estimations. The model has had state-of-the-art performance on the fastMRI dataset, outperforming other models such as the previously aforementioned VarNet and U-Net baselines.

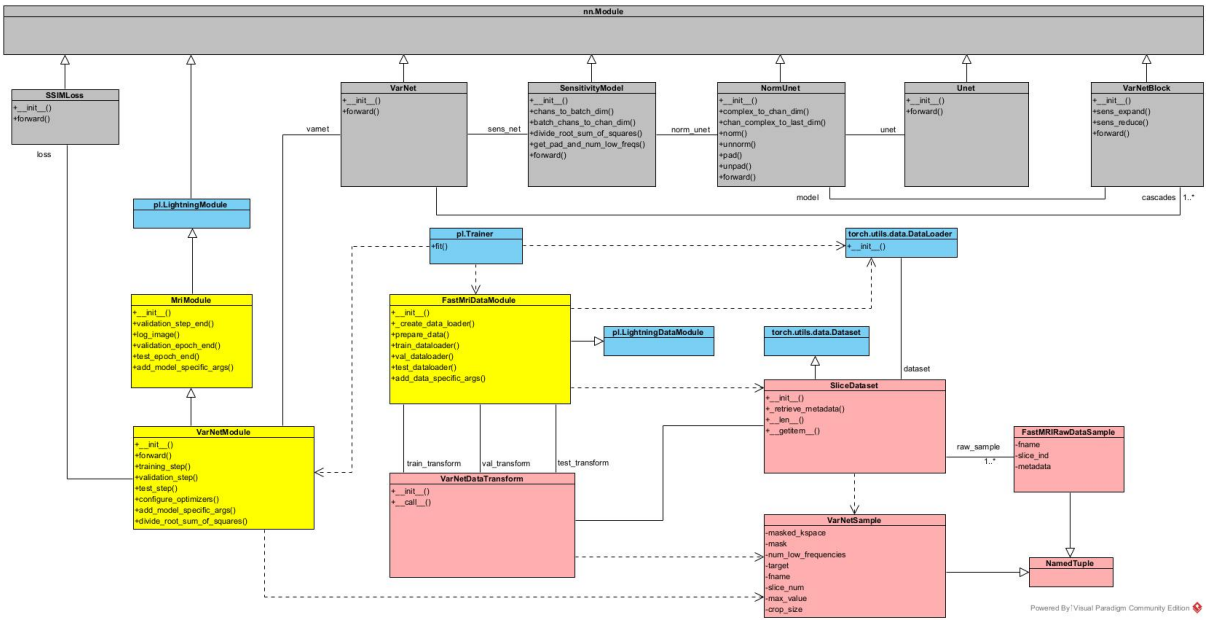


Figure 5.3: UML Diagram of E2E VarNet

```
    def forward(
        self,
        masked_kspace: torch.Tensor,
        mask: torch.Tensor,
        num_low_frequencies: Optional[int] = None,
    ) -> torch.Tensor:
        sens_maps = self.sens_net(masked_kspace, mask, num_low_frequencies)
        kspace_pred = masked_kspace.clone()

        for cascade in self.cascades:
            kspace_pred = cascade(kspace_pred, masked_kspace, mask, sens_maps)

        return fastmri.rss(fastmri.complex_abs(fastmri.ifft2c(kspace_pred)), dim=1)
```

The `(forward()` method above is from `VarNet`. It first calls `SensitvityModel` to calculate the sensitivity maps using `NormUnet` and `Unet`. The sensitivity maps are passed into each of the cascades, which in turn use their own `NormUnet` and `Unet` to perform the refinement steps.

5.3 Feature-Image VarNet

Accelerated MRI reconstructions via variational network and feature domain learning (Giannakopoulos et al., 2024) [12] presents two models, which add to and improve the performance of E2E-VarNet by adding three different architectural modifications, with motivation stemming from the idea that most of the high-level features are discarded in the last convolutional layers of each cascade when applying the data consistency. The number of output channels decreases from 32 to 2, and these remaining features could contain useful information for the reconstruction.

The two models proposed are Feature VarNet and Feature-Image VarNet. Feature VarNet has the same structure and works the same way as E2E-VarNet but with different cascades and data consistency, the SME remains the same. Feature VarNet introduces feature cascades with block-wise attention instead of the refinement cascades from E2E-VarNet, and the data consistency is applied using an encoder and decoder.

Before the feature cascades are applied, the input k-space is encoded to a 32-channel feature tensor. Once the k-space has been projected into the higher-dimensional feature tensor, it goes through the feature cascade steps. It starts by going through a block-wise attention module and is then passed into a U-Net, which denoises and reorganises the features while staying in the 32-channel feature domain. After the cascade, the feature tensor is decoded into 2-channels and DC is enforced, the result is then re-encoded back into 32-channel feature space. When all cascades are complete, the output feature tensor is reduced into a 2-channel k-space, which is then inverse Fourier transformed. And finally, RSS is applied to obtain the final image.

The encoder and decoder inside of the data consistency are a single convolution layer with a kernel size of 5 and padding of 2. The encoder maps the 2 input channels (real and imaginary part of the image) to a chosen number of feature channels, which in this case is 32. The decoder maps the 32 channels into 2 channels. Both encoder and decoder are used without an activation function [12].

Aliasing artifacts in Cartesian undersampling occur in predictable locations because every missing line in a k-space causes the aliasing in the image space. These artifacts repeat every R pixels (where R is the acceleration factor) as that the periodicity of the missing k-space lines. To exploit this, a block-wise attention [13] module is introduced before the U-Net in each cascade. Positional encodings are added to the input features to provide spatial context to the attention mechanism, and the query, key, and value embeddings are initialised using dilated convolutions. The attention mechanism computes attention weights by comparing the query and key embeddings, these weights are then used to attend to the value embeddings and produce the output features. The process starts by reshaping the feature tensor into blocks where aliasing repeats every N pixels ($N = W/R$ where W is the image width). These blocks are then flattened into column vectors and passed through the attention mechanism to produce refined feature representations. The output is projected back onto the same shape as the original input features using a 1×1 convolution, and the two are added together to form the final output, which becomes the input for the U-Net [12]. See the green classes in the UML diagram in Figure 5.5 for the feature cascades.

Feature-Image VarNet is a hybrid architecture that combines the strengths of Feature VarNet and E2E-VarNet, integrating both feature space and image space reconstruction techniques to enhance performance. For simplicity, the cascades inherited from E2E-VarNet are referred to as image cascades. As in previous models, the sensitivity maps are estimated first. The feature space cascades are then executed, the resulting feature space is decoded into k-space. This k-space output is passed through the image cascades. After the image cascades are completed, the final image is reconstructed as usual.

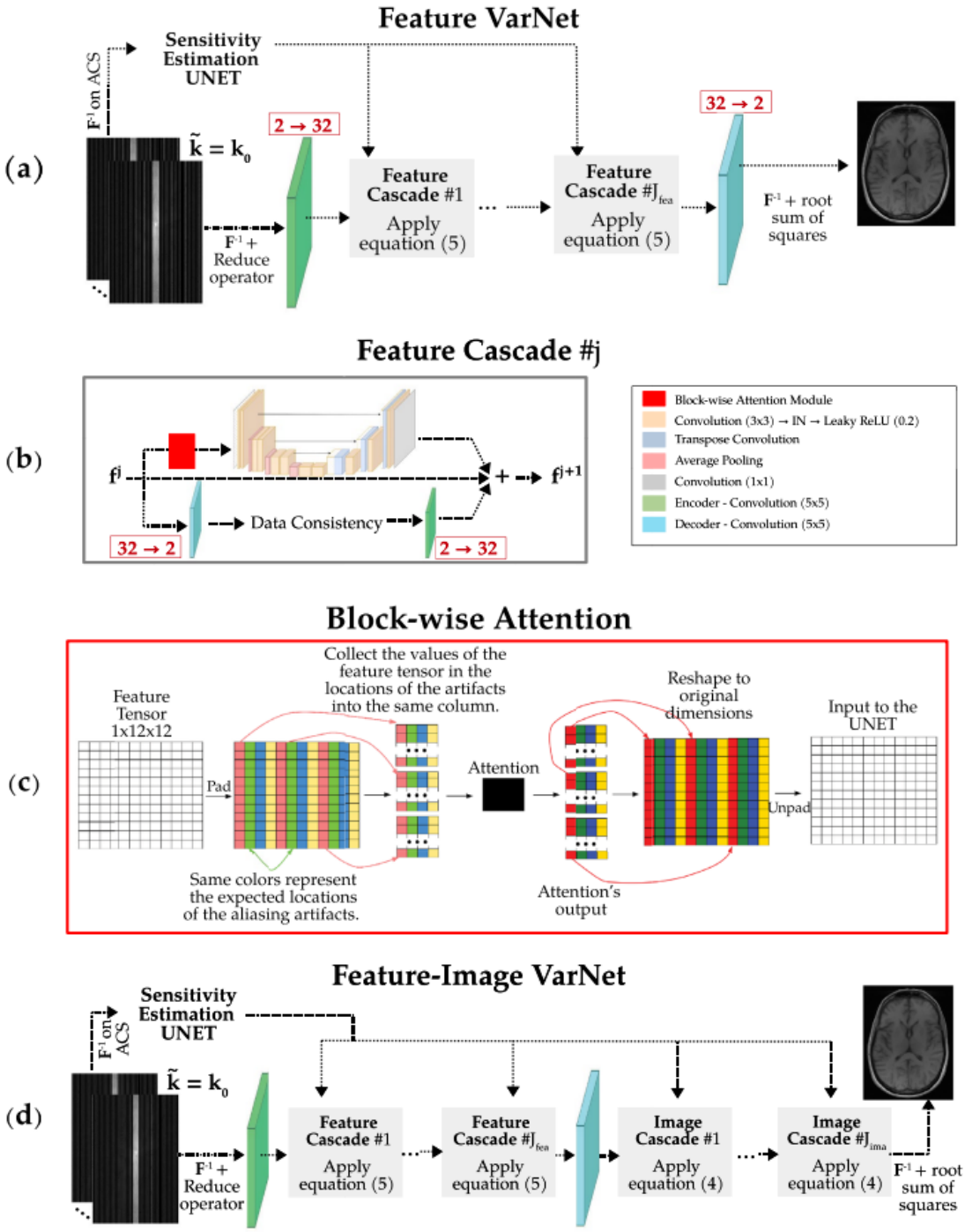


Figure 5.4: Feature VarNet and FI VarNet Architecture [12]

Feature-Image VarNet represents an important step forward in MRI reconstruction by combining feature space cascades with image space updates. Its application of block-wise attention enhances artifact suppression by directly targeting predictable aliasing patterns. Beyond improvements in SSIM, the model has earned neuroradiologists, with several years of clinical experience, seal of approval. Clinicians consistently rated its reconstructions as sharper and more reliable for assessing fine anatomical details, confirming its potential for real-world deployment.

Below is the python implementation of the neural network classes and the interaction between the said classes. The UML diagram is the same as the E2E-VarNet with the addition of the feature cascades (green classes).

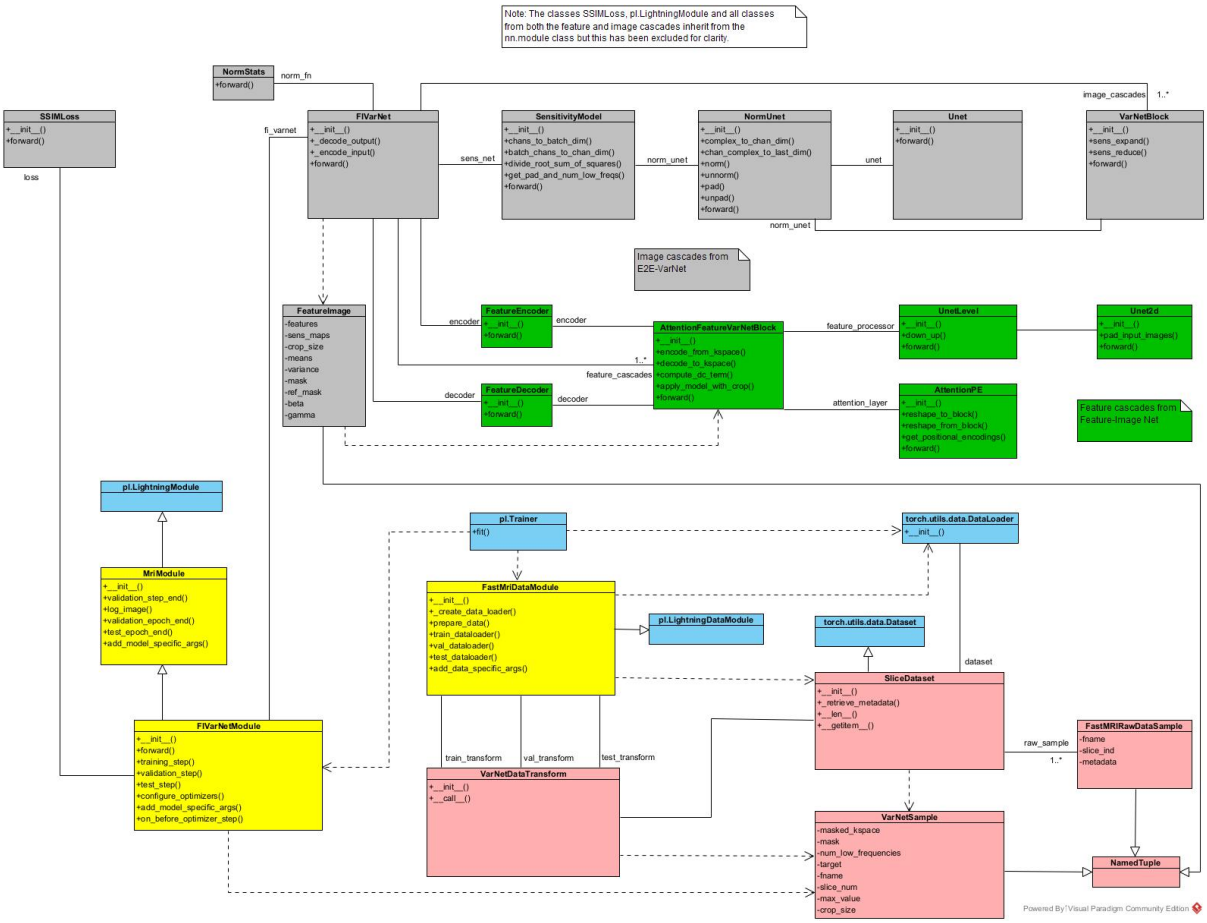


Figure 5.5: UML Diagram of Feature-Image VarNet

The following code is from the `forward()` method inside of `AttentionPE`, outlining the attention mechanism.

First, the model needs to be given spatial information of the image. This is done using `pos_enc` which adds position encodings to the feature tensor.

Next query (`q`), key (`k`), and value (`v`) are initiliased using dilated convolutions.

Then `q` and `k`, which are feature tensors projected from `h_`, are reshaped into blocks. Their dimensions change from `(1, 32, 640, 368)` to `(58880, 32, 4)`, since 640×368=235,520

spatial positions and dividing by the acceleration factor R=4 gives 235,520/4=58,880 blocks. Dividing by the acceleration as every fourth line in k-space is missing, which produces aliasing that repeats every fourth pixel in image space. Reshaping this way groups together pixels that are aliased with one another, ensuring the attention mechanism learns correlations in the correct locations.

Attention weights are calculated next. Although the attention formulation is usually written as QK^T, this implementation permutes Q instead of transposing K so that the tensor dimensions align for batched matrix multiplication in PyTorch, mathematically the two are equivalent. The shape of `q` is now `(58880, 4, 32)` and the resulting shape of `w_` is `(58880, 4, 4)`.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Figure 5.6: Scaled-Dot Product Attention [13]

In the final step, `v` is first reshaped into blocks, after which the attention weights are permuted to ensure correct value and key alignment before application. The weights are then applied to `v`, resulting in `h_` having the shape `(58880, 32, 4)`. The attended features are reshaped back from the blocks into the original shape of `(1, 32, 640, 368)`. They are then projected into the correct feature dimension, and combined with the original input via a residual connection to preserve features while enhancing them through attention.

```
247     def forward(self, x: Tensor, accel: int) -> Tensor:
248         im_size = (x.shape[2], x.shape[3])
249         h_ = x
250         h_ = self.norm(h_)
251 
252         pos_enc = self.get_positional_encodings(x.shape[2], x.shape[3], h_.device.type)
253         h_ = h_ + pos_enc
254 
255         q = self.dilated_conv(self.q(h_))
256         k = self.dilated_conv(self.k(h_))
257         v = self.dilated_conv(self.v(h_))
258 
259         # compute attention
260         c = q.shape[1]
261         q = self.reshape_to_blocks(q, accel)
262         k = self.reshape_to_blocks(k, accel)
263         q = q.permute(0, 2, 1)  # b,hw,c
264         w_ = torch.bmm(q, k)  # b,hw,hw    w[b,i,j]=sum_c q[b,i,c]k[b,c,j]
265         w_ = w_ * (int(c) ** (-0.5))
266         w_ = torch.nn.functional.softmax(w_, dim=2)
267 
268         # attend to values
269         v = self.reshape_to_blocks(v, accel)
270         w_ = w_.permute(0, 2, 1)  # b,hw,hw (first hw of k, second of q)
271         h_ = torch.bmm(v, w_)  # b, c,hw (hw of q) h_[b,c,j] = sum_i v[b,c,i] w_[b,i,j]
272         h_ = self.reshape_from_blocks(h_, im_size, accel)
273         h_ = self.proj_out(h_)
274 
275         return x + h_
```

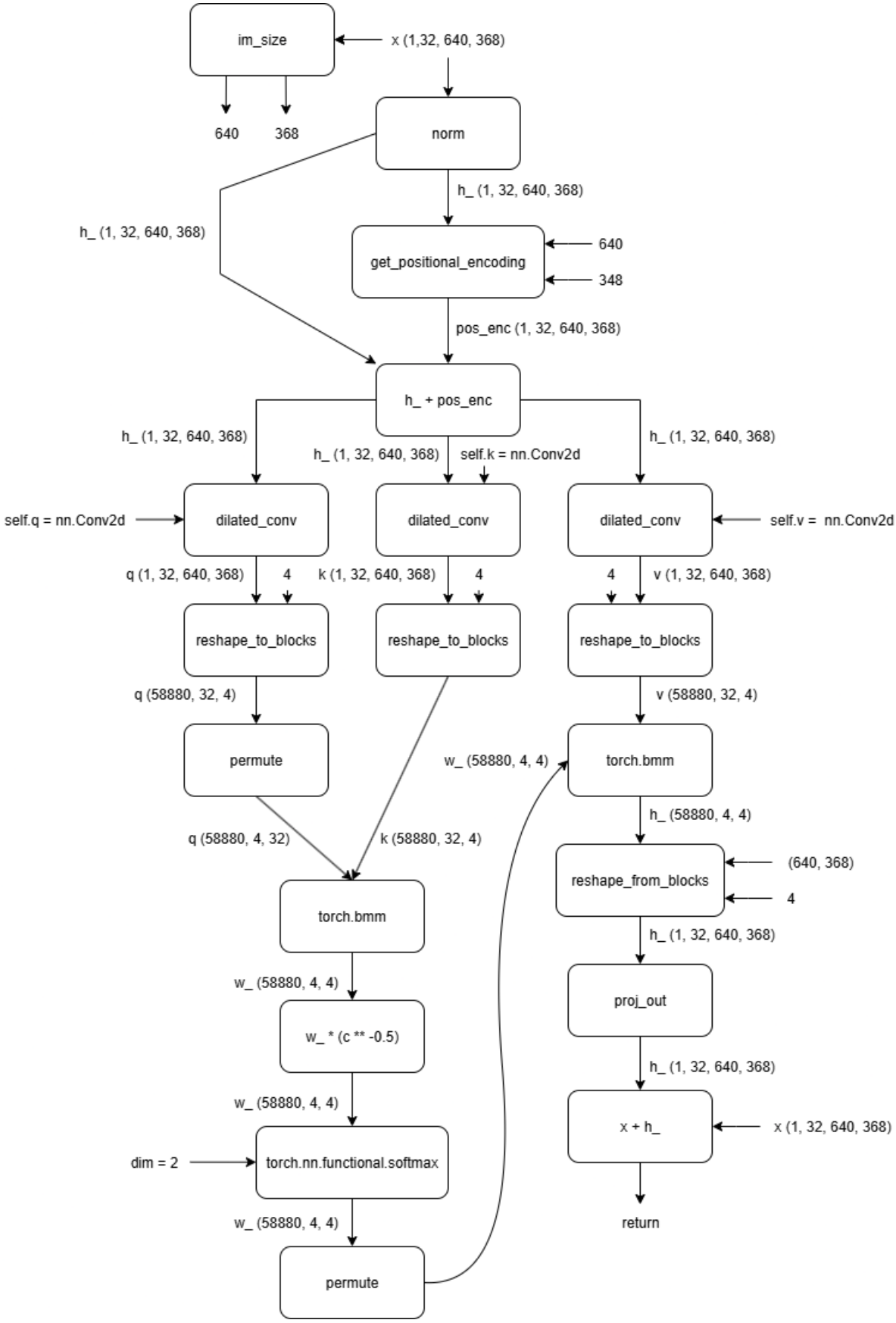


Figure 5.7: Flowchart of the `AttentionPE forward()` method

6 Conclusions

In reviewing the progression from U-Net baselines to models such as E2E-VarNet and Feature-Image VarNet, it is clear that each approach offers distinct advantages and trade-offs for accelerated MRI reconstruction.

U-Net remains the simplest and most common baseline for MRI reconstruction because it is easy to implement, computationally efficient, and effective at learning mappings from undersampled to fully sampled images. If U-Net was used alone it will be held back by its limitations of operating entirely in the image domain, with no physics-based enforcement of data consistency. Details can be hallucinated causing anatomically implausible features, which is unacceptable in clinical practice. All while performance degrades significantly at higher accelerations. Despite the limitations, it has shown that deep learning could outperform CS, motivating the development of models like E2E-VarNet and Feature-Image VarNet. Retaining U-Net as their refinement module inside cascades, but surround it with data consistency and domain specific innovations.

E2E-VarNet was a breakthrough in physics based learning ensuring reconstructions remained faithful to the acquired k-space data, with the end-to-end sensitivity estimation being computed in house removing the need to rely on external methods like ESPIRiT. When first introduced in 2020, it was established as the reference model on the fastMRI dataset by placing highly in both knee and brain data on the leaderboard at R=4, outperforming CS and plain U-Nets. However, drawbacks of the model are that it mainly operates in the 2-channel domain at each cascade, which limits feature representation, and at higher accelerations of R=8, aliasing and blurring were observed. It is also computationally heavy as multiple cascades are costly in terms of memory and runtime, E2E-VarNet consists of 20.1M trainable parameters. Nonetheless it remains a strong benchmark for physics informed reconstruction methods, as seen with Feature VarNet and Feature-Image VarNet.

Feature VarNet and Feature-Image VarNet built upon E2E-VarNet to process data in a higher dimensional space as not to waste any information, and acknowledges the aliasing periodicity by using attention to learn the artifact dense regions. And radiologist studies found Feature-Image VarNet to produce more visibly clear and sharper images. FI VarNet placed 2^{nd} at R=4 and 3^{rd} at R=8, exceeding E2E-VarNet in every metric. Ablation studies confirmed that feature cascades outperforms image cascades, block-wise attention improved artifact suppression and most importantly the combination of feature and image cascades yielded the strongest overall performance. But the added complexity requires more architectural components which can make training difficult. The large resource requirements due to larger feature tensors and attention mechanism leads directly into heavy memory usage and expensive training costs, Feature-Image VarNet consists of 187M trainable parameters.

It is important to note that while E2E VarNet and FI VarNet rank highly among open submissions on the fastMRI leaderboard, the very top positions are occupied by private models that are not publicly available. These submissions may employ undisclosed architectures or training strategies, making direct comparison challenging, but they indicate that further performance gains are possible beyond current published methods.

AIRS-Net is a closed-source model from AIRS Medical (Seoul, South Korea) which sits at the top for 4x and 8x acceleration [12].

Ranking	Model	SSIM	PSNR
4× acceleration			
1st	AIRS-Net	0.9632	42.1
2nd	FI VarNet	0.9607	41.5
3rd	DIRCN	0.9601	41.3
4th	E2E VarNet	0.9591	41.1
5th	dd	0.9591	41.1
6th	IR_FRestormerF11	0.9587	41.0
8× acceleration			
1st	AIRS-Net	0.9511	39.7
2nd	DIRCN	0.9455	38.6
3rd	FI VarNet	0.9453	38.6
4th	IR_FRestormerF72	0.9427	38.0
5th	E2E VarNet	0.9426	38.0
6th	dd	0.9426	38.0

Figure 6.1: fastMRI leaderboard for 4x and 8x accelerations [12]

Beyond variational networks and their extensions, other deep learning approaches for MRI reconstruction have emerged that show strong reliability. Diffusion models such as, cold diffusion [14] and score-based generative models [15], have recently been applied to undersampled MRI, which learn the gradient of the log-probability distribution of MR images, then carries out an iterative process of denoising samples from the prior. Generative adversarial networks (GANs) have also all been explored as alternatives or complements to physics-informed unrolled networks [16]. While these approaches are less mature on the fastMRI leaderboard compared to the VarNet derivatives, they represent promising directions that balance reconstruction quality, robustness, and data efficiency.

References

[1] Jeffrey A Fessler. "Optimization methods for MR image reconstruction (long version)". In: (June 2019). URL: http://arxiv.org/abs/1903.03510.

[2] Florian Knoll et al. "Deep Learning Methods for Parallel Magnetic Resonance Image Reconstruction". In: (Apr. 2019). URL: http://arxiv.org/abs/1904.01112.

[3] Jure Zbontar et al. "fastMRI: An Open Dataset and Benchmarks for Accelerated MRI". In: (Dec. 2019). URL: http://arxiv.org/abs/1811.08839.

[4] Souheil J. Inati et al. "ISMRM Raw data format: A proposed standard for MRI raw datasets". In: *Magnetic Resonance in Medicine* 77 (1 Jan. 2017), pp. 411–421. ISSN: 15222594. DOI: 10.1002/mrm.26089.

[5] Aaron Defazio. "Offset Sampling Improves Deep Learning based Accelerated MRI Reconstructions by Exploiting Symmetry". In: (Feb. 2020). URL: http://arxiv.org/abs/1912.01101.

[6] Jim Nilsson and Tomas Akenine-Möller. "Understanding SSIM". In: (June 2020). URL: http://arxiv.org/abs/2006.13846.

[7] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: (May 2015). URL: http://arxiv.org/abs/1505.04597.

[8] Alejandro Ito Aramendia. *The U-Net : A Complete Guide*. Feb. 2024. URL: https://medium.com/@alejandro.itoaramendia/decoding-the-u-net-a-complete-guide-810b1c6d56d8.

[9] Naoki Shibuya. *Up-sampling with Transposed Convolution*. Nov. 2017. URL: https://naokishibuya.github.io/blog/2017-11-14-up-sampling-with-transposed-convolution/.

[10] Anuroop Sriram et al. "End-to-End Variational Networks for Accelerated MRI Reconstruction". In: (Apr. 2020). URL: http://arxiv.org/abs/2004.06688.

[11] Kerstin Hammernik et al. "Learning a Variational Network for Reconstruction of Accelerated MRI Data". In: (Apr. 2017). URL: http://arxiv.org/abs/1704.00447.

[12] Ilias I. Giannakopoulos et al. "Accelerated MRI reconstructions via variational network and feature domain learning". In: *Scientific Reports* 14 (1 Dec. 2024). ISSN: 20452322. DOI: 10.1038/s41598-024-59705-0.

[13] Ashish Vaswani et al. "Attention Is All You Need". In: (Aug. 2023). URL: http://arxiv.org/abs/1706.03762.

[14] Guoyao Shen et al. "Learning to reconstruct accelerated MRI through K-space cold diffusion without noise". In: (Sept. 2024). DOI: 10.1038/s41598-024-72820-2.

[15] Hyungjin Chung and Jong C. Ye. "Score-based diffusion models for accelerated MRI". In: (July 2022). URL: https://arxiv.org/abs/2110.05243.

[16] Walid Al-Haidri et al. "A Deep Learning Framework for Cardiac MR Under-Sampled Image Reconstruction with a Hybrid Spatial and k-Space Loss Function". In: *Diagnostics* 13 (6 Mar. 2023). ISSN: 20754418. DOI: 10.3390/diagnostics13061120.

A Creating the Virtual Environment

I used Anaconda Navigator to create a virtual environment for fastMRI. First open the base terminal and enter the following to create a new virtual environment. I named my environment FastMRI.

```
(base) C:\Users\Nikhil>conda create --name FastMRI python=3.10
```

Next I created a new directory to git clone the original fastMRI repo.

```
D:\MRI>git clone https://github.com/facebookresearch/fastMRI.git
```

To learn the basics, I started off by working through *fastMRI_ tutorial.ipynb*. I converted it to a python script, *fastMRI_ tutorial.py*, which can be found in my GitHub repo.

Open the new environment terminal and pip install the following packages that are required to run *fastMRI_ tutorial.ipynb*. Other packages that are needed were automatically installed when the environment was created, for example numpy.

```
(FastMRI) C:\Users\Nikhil>pip install matplotlib
(FastMRI) C:\Users\Nikhil>pip install h5py
(FastMRI) C:\Users\Nikhil>pip install pandas
(FastMRI) C:\Users\Nikhil>pip install tqdm
```

To install PyTorch I went to *https://pytorch.org/get-started/locally* and selected: Stable (2.8.0), Windows, Pip, Python and CUDA 12.6 copied the command (removing the 3 after the pip) and pip installed it to my fastMRI environment.

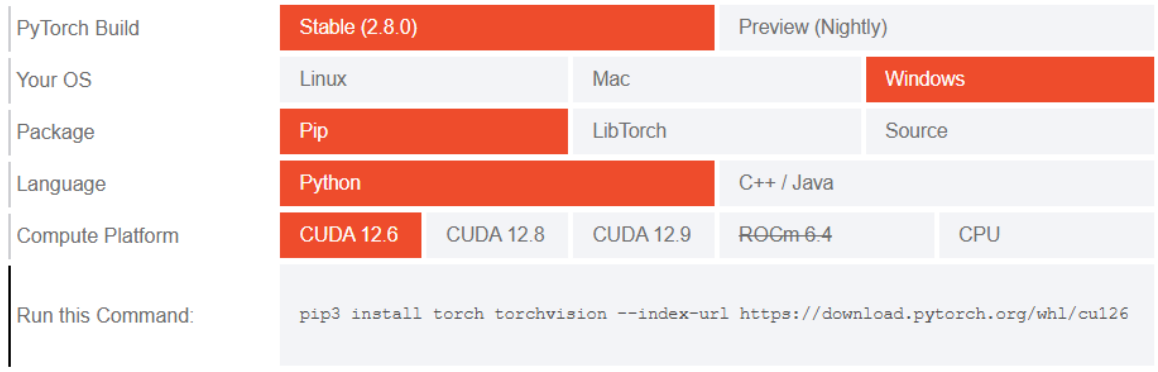


Figure A.1: Installing PyTorch

```
(FastMRI) C:\Users\Nikhil>pip install torch torchvision --index-url \
https://download.pytorch.org/whl/cu126
```

The extra packages needed to run *train_ varnet_ demo.py*.

```
(FastMRI) C:\Users\Nikhil>pip install pytorch-lightning
(FastMRI) C:\Users\Nikhil>pip install runstats
(FastMRI) C:\Users\Nikhil>pip install scikit-image
```

B Changes to Code

B.1 fastMRI_tutorial.py

As I was converting a python notebook to a python script some changes were necessary to get the script working.

In the show coils function added `plt.show(block=True)` to the end with the same indentation as the for loop so it looks like this:

```
def show_coils(data, slice_nums, cmap=None):
    fig = plt.figure()
    for i, num in enumerate(slice_nums):
        plt.subplot(1, len(slice_nums), i+1)
        plt.imshow(data[num], cmap=cmap)
    plt.show(block=True)
```

After the line `plt.imshow(np.abs(slice_image_rss.numpy()), cmap='gray')` added another `plt.show(block=True)`, so it ended up looking like this:

```
plt.imshow(np.abs(slice_image_rss.numpy()), cmap='gray')
plt.show(block=True)
```

Once again another `plt.show(block=True)` was needed, this time on the line after `plt.imshow(np.abs(sampled_image_rss.numpy()), cmap='gray')` so the end result is:

```
plt.imshow(np.abs(sampled_image_rss.numpy()), cmap='gray')
plt.show(block=True)
```

B.2 train_varnet_demo.py

The version of PyTorch Lightning I used was different from what the original code was made with, so I had to make a few changes to get the scripts working.

The first change happens on line 46.

```
46 distributed_sampler=False
```

On line 84 I had to change the directory. So that when I ran the script, the *fastmri_dirs.yaml* file that gets created was in the same directory as everything else to keep the project organised.

Next remove or comment lines 150 and 179-182.

```
150 # parser = pl.Trainer.add_argparse_args(parser)
```

```
179     # if args.resume_from_checkpoint is None:
180         # ckpt_list = sorted(checkpoint_dir.glob("*.ckpt"), key=os.path.getmtime)
181         # if ckpt_list:
182             # args.resume_from_checkpoint = str(ckpt_list[-1])
```

The last change in *train_varnet_demo.py* comes at line 67, where the variable 'trainer' is changed to:

```
67      trainer = pl.Trainer(
68          max_epochs=50,
69          accelerator='gpu',
70          devices=1
71      )
```

Note: If this change is made first it will change the number of the lines previously mentioned.

The final changes are in the files *mri_module.py* and *data_module.py* which are located in fastmri/pl_modules. These are needed so *train_varnet_demo.py* doesn't encounter any errors.

B.2.1 mri_module.py

Remove or comment the lines 154-221, the 'validation_epoch_end' method inside the 'MriModule' class.

```
154     # def on_validation_epoch_end(self, val_logs):
155     #     # aggregate losses
156     #     losses = []
157     #     mse_vals = defaultdict(dict)
158     #     target_norms = defaultdict(dict)
159     #     ssim_vals = defaultdict(dict)
160     #     max_vals = dict()
161     #
162     #     ...
```

B.2.2 data_module.py

The last 3 changes are in the 'FastMriDataModule' class. The first one is on line 107, second one at line 352 and finally line 436.

```
107 num_workers: int = 1,
```

```
352 default="multicoil",
```

```
436 default=0,
```