The University of Edinburgh

Masters Dissertation Report

Learning to image without ground truth

Author: Lunyuan Cao *Supervisor: Dr Michael Davies*

A thesis submitted in fulfilment of the requirements for the degree of MSc. in Signal Processing and Communications

in the

School of Engineering

August 2021

Declaration of Authorship

I, Lunyuan Cao, declare that this dissertation report titled, 'Learning to image without ground truth' and the work presented in it is my own. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Lunyuan Cao

Date: 12/08/2021

Abstract

Recently deep learning techniques have achieved state-of-the-art results in many inverse problems such as image denoising, super-resolution, CT and MRI image reconstruction. Most of these techniques require the ground truth data for learning and these approaches are called supervised learning. However, in real-world applications, clean ground truth data is scarce and even impossible to obtain, which makes the training of the neural network become a significant practical challenge. Although some self-learning approaches have been proposed to learn the images without ground truth, most of them can only deal with denoising problem in which the forward operator has trivial nullspace. In this project, we implement a self-supervised learning method called Equivariant Imaging (EI) to solve the sparse-view CT reconstruction where the forward operator has non trivial nullspace. By comparing with the supervised learning method, the results shows the EI method can learn the CT image reconstruction without ground truth very well.

Keywords: Image processing, Inverse problems, Model-based optimization, Deep learning.

Acknowledgements

I am very grateful to my supervisors, without whom this project would be challenging to complete in a short period. They guide me step by step with a lot of patience and answer my questions immediately. I learn much knowledge from them about how to figure out problems and learn from papers. I am also thankful to the University of Edinburgh for providing this fantastic project to me.

Contents

List of Figures

List of Tables

Abbreviations

CS	**C**ompressive **S**ensing
CT	**C**omputed **T**omography
MRI	**M**agnetic **R**esonance **I**maging
MSE	**M**ean **S**quare **E**rror
NN	**N**eural **N**etwork
EI	**E**quivalent **I**maging
ML	**M**aximum **L**ikelihood
MAP	**M**aximum **A** **P**osterior
SURE	**S**tein's **U**nbiased **R**isk **E**stimator
GSURE	**G**eneralized **S**tein's **U**nbiased **R**isk **E**stimator
LDAMP	**L**earned **D**enoising-Based **A**pproximate **M**essage **P**assing
FBP	**F**iltered **B**ack **P**rojection
DBP	**D**eep **B**asis **P**ursuit
FT	**F**ourier **T**ransform
IFT	**I**nverse **F**ourier **T**ransform
2D	**2** **D**imensional
3D	**3** **D**imensional
ART	**A**lgebraic **R**econstruction **T**echnique
EM	**E**xpectation **M**aximization
SART	**S**imultaneous **A**lgebraic **R**econstruction **T**echnique
LEARN	**L**earned **E**xperts' **A**ssessment-based **R**econstruction **N**etwork
RELU	**R**ectified **L**inear **U**nit
ANNs	**A**lgebraic **N**eural **N**etworks
CNNs	**C**onvolutional **N**eural **N**etworks
RNNs	**R**ecurrent **N**eural **N**etworks

PSNR	**P**eak **S**ignal-to-Noise **R**atio

Symbols

A	forward operator
A^{-1}	inverse forward operator
$A^{\dagger}$	pseudoinverse forward operator
θ	parameters
f_θ	reconstruction function
$\mathcal{G}$	group of transformations
$\mathcal{L}$	error function

Chapter 1

Introduction

Inverse problems are an old topic in mathematics and often appear in image and signal processing, such as compressive sensing (CS), computed tomography (CT), and magnetic resonance imaging (MRI) [2–4]. The goal of these problems is to recover original signal x from measurements y

$$y = Ax + \epsilon \tag{1.1}$$

and this process is often referred to as signal or image reconstruction. Due to the ill-conditioned operator A and the noise ϵ, this is usually a challenging task. Traditional model-based approaches often use some prior knowledge such as smoothness, sparsity in some dictionary or basis to reduce the set of plausible reconstructions [1]. Although these model-based paradigms usually have good theoretical properties, they can be very computationally expensive due to the optimization program that has to be run during the test time. Recently, deep learning techniques that rely on well-labelled data pairs (x, y) and supervised training have achieved state-of-the-art results on these image reconstruction problems [5]. By training with enough data, the neural network can reconstruct the original signals faster and better than classical model-based methods. However, all of these methods require ground truth signals x for learning the reconstruction function f_θ, which ignores the fact that in real-world applications, ground truth signals x is normally impossible or difficult to obtain. Therefore, some self-supervised learning methods which only use measurements y for learning have been proposed. SURE [6] which provides an unbiased estimate of the Mean Square Error (MSE) of a mean estimator, was proposed to recover the original signal x with only noisy measurements. It can be used to solve denoising and CS problems with a neural network (NN) without ground truth. Chen et al. [7] introduced an end-to-end self-supervised approach called Equivalent Imaging (EI) which overcomes the limitation of scarce ground-truth or even no ground-truth by using the equivariance present in natural signals such as rotation or shift invariance. This method can be used to solve CT reconstruction and image inpainting problems, and the results show that this approach performs almost as well as the supervised learning approach with the ground truth.

In this project, we implement the EI self-supervised learning method on the specific CT reconstruction inverse problem and show how it can learn the reconstruction function from compressed measurement y alone and evaluate the performance compared with a supervised learning approach.

There are mainly 6 chapters in this thesis. In Chapter 2, We introduce the basic concept of inverse problems and some typical applications of the inverse problems. Both traditional model-based approaches and modern deep learning techniques for solving inverse problems are clearly demonstrated. The principles of X-ray projection and some strategies for CT reconstruction are illustrated then. In the end, we demonstrate the concept of neural networks and how they work and some popular neural network architectures.

In Chapter 3, we implement the EI method to solve the CT reconstruction problem without ground truth and supervised method for comparison.

In Chapter 4, experiment setup and implementation are first illustrated. Then several experiments are done to estimate the performance of the EI method and adjust parameters to achieve the best performance.

In Chapter 5, we discuss the main results in our experiments and the potential and limitation of the EI method.

In Chapter 6, we give a brief conclusion of the project and some directions for future work.

Chapter 2

Background and theory

2.1 Inverse problems

Inverse problems are about reconstructing an unknown signal or image from observations, and the observations are acquired through a forward process. Inverse problems are often ill-posed since there is no unique solution to the equation Ax=0, and the forward operator A has a non-trivial null space. The ill-posed forward operator A also makes it very difficult or even impossible to reconstruct the original signals without any prior knowledge about the data [8]. To solve an ill-posed problem, we normally need to use regularization to transform the ill-posed problem into a well-posed problem, such as Tikhonov regularization [9] and total variation regularization [10]. Inverse problems have many applications such as image denoising, deconvolution, inpainting, CS, CT, MRI and the forward operator of these applications are shown in table 2.1.

Application	**Forward operator**	**Description**
Denoising [11]	A = I	I is the identity matrix
Deconvolution [12]	A(x) = h*x	h is a known blur kernel and * denotes convolution
Superresolution [13]	A = SB	S is a subsampling operator (identity matrix with missing rows) and B is a blurring operator corresponding to convolution with a blur kernel
Inpainting [14]	A = S	S is a diagonal matrix where S (i,j) = 1 for the sampled pixels and S (i,j) = 0 for the unsampled pixels
CS [15]	A = SF	S is a subsampling operator (identity matrix with missing rows) and F is discrete Fourier transform matrix
MRI [16]	A =SFD	S is a subsampling operator (identity matrix with missing rows) and F is discrete Fourier transform matrix, and D is a diagonal matrix representing a spatial domain multiplication with the coil sensitivity map
CT [11]	A = R	R is the discrete Radon transform

Table 2.1: Foward operators for inverse problems applications [1]

2.1.1 Traditional model-based approaches

To solve an inverse problem (recover images x from measurements y), we need to start from the beginning equation (1.1). This equation can be rewritten as $\boldsymbol{y} = \mathcal{N}(\mathcal{A}(\boldsymbol{x}))$, where $\mathcal{N}(\cdot)$ samples from the a noise distribution. Therefore, if we know the distribution of the noise ϵ and the forward matrix A, we can use a maximum likelihood (ML) estimation to recover image x.

$$\hat{\boldsymbol{x}}_{\mathrm{ML}} = \arg\max_{\boldsymbol{x}} p(\boldsymbol{y} \mid \boldsymbol{x}) = \arg\min_{\boldsymbol{x}} -\log p(\boldsymbol{y} \mid \boldsymbol{x}) \tag{2.1}$$

where $p(\boldsymbol{y} \mid \boldsymbol{x})$ is the likelihood of observing measurements y given the ground truth x. This method can solve a denoising problem well as the rank of A is full rank. However, it cannot handle other ill-posed inverse problems where the forward operator A is not of full rank, as there are non-unique solutions.

Hence, some prior knowledge about the ground truth x, such as smoothness and sparsity, is used to solve the ill-posed inverse problems. For example, we might expect x to be smooth, which means the distribution of x, g(x), is uniformly distributed. Then we can use Bayes' theorem to calculate the posterior distribution of x

$$p(x \mid y) = \frac{p(y \mid x)g(x)}{\int_{\Theta} p(y \mid \vartheta)g(\vartheta)d\vartheta} \tag{2.2}$$

where g is the density function of x, Θ is the domain of g. We can see the denominator of the posterior distribution is always positive and does not depend on x, so the equation (2.2) can be rewritten as

$$p(x \mid y) = p(y \mid x)g(x) \tag{2.3}$$

Then we can derive the maximum a posterior (MAP) estimation as

$$\hat{x}_{MAP} = \underset{x}{\operatorname{argmax}}\, p(x|y) = \underset{x}{\operatorname{argmax}}\, p(y|x)g(x) = \underset{x}{\operatorname{argmin}} -lnp(y|x) - ln(g(x)) \tag{2.4}$$

and use it to recover x. Moreover, when the noise is additive white Gaussian noise, the MAP formula leads to

$$\frac{1}{2}\left\|A(x) - y\right\|_2^2 + r(x) \tag{2.5}$$

where r(x) is proportional to the negative log-prior of x. There have been many examples that use this kind of framework, such as sparsity regularization on some basis [17], Tikhonov regularization [9], and total variation regularization [10]. Although MAP estimation can be used to solve a lot of inverse problems, difficulties occur when (1) the distribution of the signal x is not known, (2) the statistics of the noise are not known, (3) the forward operator A is not known or only partially known.

In a nutshell, the traditional model-based approaches mainly constrain the space of plausible solutions based on prior knowledge about the data (regularization). As discussed above, they have some drawbacks and are computationally expensive in some inverse problems with projective measurements like CT and MRI.

2.1.2 Deep learning techniques

In recent years, deep learning techniques have significantly improved in solving inverse problems, especially those with projective measurements such as CT and MRI. Some approaches start from the MAP formulation and try to learn a function of r(x), while others attempt to learn a direct mapping from measurements y to ground truth x. Deep learning methods can be broke down into two classes. The first class of deep learning is what we call supervised learning. The main idea is to match the ground truth x with the measurements y, which can be done easily by using the forward operator on ground truth x to create measurements y. The second class is unsupervised learning. This kind of learning does not rely on the matched pairs (x, y) and can be divided into three groups: (1) use unpaired ground truth signals x and measurements y, (2) use ground truth signals x only, (3) use measurements y only, also called self-supervised learning.

Forward operator A plays a significant role in inverse problems. The state of the forward operator A decides what kinds of deep learning should be used to solve the inverse problems. There are four states of the forward operator A: (1) A is known from the beginning, (2) A is not known during training but known during testing, (3) A is partially known, (4) A is never known. According to different states of the forward operator A, different types of deep learning techniques are adopted. The overall taxonomy is shown in table 2.2. Here, we only discuss the case that the forward matrix A is known in detail as our experiments are all down in this case. Many deep learning methods can be adopted to solve inverse problems when forward operator A is known. The main two types are supervised learning and unsupervised learning that use measurements y only (self-supervised learning).

	Supervised with matched (x,y) pairs	**unpaired ground truth x and measurement y**	**ground truth x only**	**measurements y only**
A fully known	U-Net [18] Unrolled optimization [19, 20]			SURE [6] EI [7]
A only known at test time			CSGM [21] OneNet [22]	
A partially known		Cycle GAN [23]	Blind deconv -olution with GAN's [24]	AmbientGAN [25]
A unknown	AUTOMAP [26]			

Table 2.2: Different Categories of learning methods

2.1.2.1 Supervised learning

For supervised learning, the learning task is to learn a reconstruction network f_θ which maps measurements y to ground truth image x. Different deep learning techniques use different ideas to parameterize the network f_θ. The forward operator A plays a significant role in defining the architecture of f_θ. For instance, we can use an approximate inverse of a linear operator A, denoted as A^{-1}, to first map the measurements y back to the image domain and then remove the artefacts from the reconstructed images by training the neural network. For different inverse problems, the specific choice of A^{-1}

is also different. For example, in CT reconstruction, a common choice of A^{-1} is filtered back projection (FBP) [27]; in image inpainting, a common choice is a diagnose matrix [14]; in super-resolution, a common choice is upsampling by bicubic interpolation [28]. It can simplify learning by using A^{-1} to measurements compared with directly using measurements as input. Moreover, a residual (skip) connection is often added to the reconstruction network to learn the difference between the input and output. According to [29], the residual connection can mitigate the vanishing gradient problem in the training process and achieve a noticeable increase in performance compared with the same network without it. Some networks even use more complicated hierarchical residual connections, such as U-Net [30]. The whole architecture of this supervised learning method is shown in Figure 2.1. However, this approach only incorporates the forward operator in the first layer. Some Unrolled methods which incorporate the forward operator into multiple layers of the reconstruction network have been proposed [31]. An unrolled method that optimizes the MAP formula has achieved good performance in a sparse coding context [32].

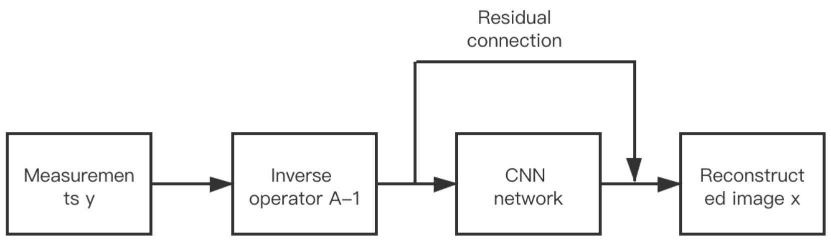


FIGURE 2.1: Supervised learning architecture for inverse problems

2.1.2.2 Self-supervised learning

For self-supervised learning, we usually need to use some prior knowledge about forwarding operator A, noise ϵ, and some properties or laws about the data. Tamir et al. [33] proposed a self-supervised deep learning method called Deep Basis Pursuit (DBP), which use the noise statistics for each training data and transform the problem to a basis pursuit denoising extension with a deep convolutional neural network. Stein et al. [6] introduced a technique called SURE to calculate the mean square error of a mean estimator. For the reconstruction network f_θ, SURE can estimate the mean square error of f_θ with only measurements y as

$$\begin{aligned}\mathbb{E}_{\boldsymbol{\varepsilon}}\left[\frac{1}{n}\left\|\boldsymbol{x}^{\star}-f_{\theta}(\boldsymbol{y})\right\|^{2}\right]=&\mathbb{E}_{\epsilon}\left[\frac{1}{n}\left\|\boldsymbol{y}-f_{\theta}(\boldsymbol{y})\right\|^{2}\right]\\&+2\frac{\sigma^{2}}{n}\operatorname{div}_{\boldsymbol{y}}\left(f_{\theta}(\boldsymbol{y})\right)-\sigma^{2}\end{aligned} \tag{2.6}$$

where $\theta \sim \mathcal{N}\left(0, \sigma^2 \mathbf{I}\right)$ and $\operatorname{div}_{\boldsymbol{y}}\left(f_\theta(\boldsymbol{y})\right) := \sum_{i=1}^n \frac{\partial f_\theta(\boldsymbol{y})}{\partial y_i}$. Then we can use gradient descent to find a good estimator θ^* if the estimators $\{f_\theta\}_{\theta \in \Theta}$ are differentiable with respect to θ. And finally we can use $f_{\theta^*}(y)$ to estimate the reconstructed image $x^\star$.

SURE can be generalized to GSURE which was proposed in [34] to estimate x from a linear measurement $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{w}$, where $\mathbf{A} \neq \mathbf{I}$, and $\mathbf{w}$ can be any distribution from the exponential family and has known covariance. When the noise $\mathbf{w}$ is i.i.d. Gaussian distributed noise, the estimate can be written as

$$\begin{aligned}
&\mathbb{E}_\varepsilon\left[\frac{1}{n}\left\|P_A\left(\boldsymbol{x}^\star - f_\theta(\boldsymbol{y})\right)\right\|^2\right] \\
&= \mathbb{E}_\varepsilon\left[\frac{1}{n}\left\|P_A \boldsymbol{x}^\star\right\|^2 + \frac{1}{n}\left\|P_A f_\theta(\boldsymbol{y})\right\|^2\right. \\
&\quad \left. - \frac{2}{n} f_\theta(\boldsymbol{y})^T A^\dagger \boldsymbol{y} + \frac{2\sigma^2}{n} \operatorname{div}_{\boldsymbol{y}}\left(f_\theta(\boldsymbol{y})\right)\right]
\end{aligned} \tag{2.7}$$

where $A^\dagger$ is the pseudoinverse of A and $P_A = A^\dagger A$ is projection onto the row space of A.

SURE has also been applied to train the DnCNN [35] and Learned Denoising-Based Approximate Message Passing (LDAMP) networks [36] for denoising and compressive sensing tasks.

2.1.2.3 EI

EI is an end-to-end deep learning technique that use some invariant properties presented in images to achieve self-supervised learning, such as shift invariance for image inpainting, rotation invariance for CT reconstruction. Unlike SURE, which can only deal with denoising tasks, the EI method can be used to solve inverse problems in which the forward operator has non trivial nullspace. The aim of this approach is also to learn a reconstruction network f_θ such that $f_\theta(y) = x$, similar to supervised learning, to recover the ground truth image x. The difference is that it uses measurement consistency and invariant set consistency instead of mapping from measurement to ground truth as there is no ground truth.

Measurement consistency

Given the measurements y, the linear forward operator A:$\mathbb{R}^n \rightarrow \mathbb{R}^m$, and the reconstruction function $f_\theta : \mathbb{R}^m \rightarrow \mathbb{R}^n$ that needed to be learned, the measurement consistency is

$$A f_\theta(y) = y \tag{2.8}$$

The measurement consistency can make sure the reconstruction function f_θ is consistent in the measurement domain. However, it cannot capture information about X (the set of plausible signals x) outside the range of operator A as there are many f_θ that can satisfy (2.7) from Proposition 1.

Proposition 1: For any reconstruction function $f(y) : \mathbb{R}^m \mapsto \mathbb{R}^n$ of the form

$$f(y) = A^\dagger y + v(y) \tag{2.9}$$

where $v(y) : \mathbb{R}^m \mapsto \mathcal{N}$ is any function whose image domain belongs to the nullspace of A satisfies the measurement consistency.

Proof:The formula (2.7) can be rewritten as

$$Af(y) = AA^\dagger y + Av(y) \tag{2.10}$$

where the first term is y as $AA^\dagger$ is the identity matrix, and second term is 0 as $Av(y) = 0$ for all $v(y)$ in the nullspace of A. This proposition tells that it is impossible to learn the nullsapce component of A without ground truth signals.

Invariant set consistency

Measurement consistency is not enough to learn inverse mapping, so we need to use some prior knowledge about X. It is commonly assumed that some images are invariant to a group of transformations. For instance, natural images are invariant to shift transformations, and CT images are invariant to rotation transformations.

Therefore, we can say that the set of signals X is invariant to a group of transformations $\mathcal{G} = \left\{g_1, \ldots, g_{|\mathcal{G}|}\right\}$ which are unitary matrices $T_g \in \mathbb{R}^{n\times n}$, so for all $x \in \mathcal{X}$, we have

$$T_g x \in \mathcal{X} \tag{2.11}$$

for $\forall g \in \mathcal{G}$. Based on this transformation invariance, we can get another equation for composition $f \circ A$

$$f\left(AT_g x\right) = T_g f(Ax) \tag{2.12}$$

for all $x \in \mathcal{X}$, and all $g \in \mathcal{G}$.

By using the invariant set consistency, we also have

$$y = Ax = AT_g T_g^\top x = A_g \tilde{x} \tag{2.13}$$

where $A_g = AT_g$ and $\tilde{x} = T_g^\top x$. Since $\tilde{x}$ is also a signal in $\mathcal{X}$, we can learn the range space of the operator A_g.Equivalently, we can think that we rotate the range space A_g

by a group of $\mathcal{G}$.

$$\mathcal{R}_{A_g} = T_g \mathcal{R}_A \tag{2.14}$$

From Theorem 1, if we want to recover a unique signal set $\mathcal{X}$, we need to make sure the concatenation of the operator A_g spans the full space $\mathbb{R}^n$. In other words, A_g has trivial nullspace.

Theorem 1: A necessary condition for recovering the signal model X from compressed observations is that the matrix

$$M = \begin{bmatrix} AT_1 \\ \vdots \\ AT_{|\mathcal{G}|} \end{bmatrix} \in \mathbb{R}^{|\mathcal{G}|m \times n} \tag{2.15}$$

is of rank n.

Proof: Based on the matrix M and $y_g = AT_g x$ for all g, we can stack all the measurements together into $\tilde{y} = Mx$ where $\tilde{y} \in \mathbb{R}^{|\mathcal{G}|m}$. Therefore, if we want to recover x, M needs to be rank of n. This theorem also tells that as long as the range of A itself is not invariant to all T_g, the invariant set consistency allows us to learn beyond the range space of A.

So far, we have illustrated the basic concepts of model-based optimization methods and Deep CNN-based methods and introduced some famous algorithms in both methods. It is also very important to know the common pros and cons of these two kinds of methods as shown in table 2.3

	Model-based optimization methods	**Deep CNN-based methods**
Pros	1. Can handle different inverse problems 2. Have clear physical meanings	1. Data driven end-to-end learning 2. Wide coverage and good adaptability stage
Cons	1. The prior knowledge may not be strong enough 2. Can be time-consuming for the optimization process	1. Limited generality of the learned models 2. Limited interpretability for the learned models

TABLE 2.3: Pros and cons of model based optimization methods and deep CNN based methods for inverse problems.

2.2 Computed Tomography

2.2.1 X-ray projection

Due to the non-destructive diagnostic value, CT has been widely used in clinical medicine, industrial manufacturing, and other fields. CT images are 2-dimensional (or 3-dimensional) models of a decrease of attenuation of X-ray beam when passing through the object, which an X-ray detector can directly measure. The attenuation coefficient is determined by the nature of the medium (ex. bone, blood, muscle for the human body). The object gradually rotates 360 degrees during the scanning process, and the detector will record the projection image at each angle. The process for two angles projection is shown in Figure 2.2 [37].

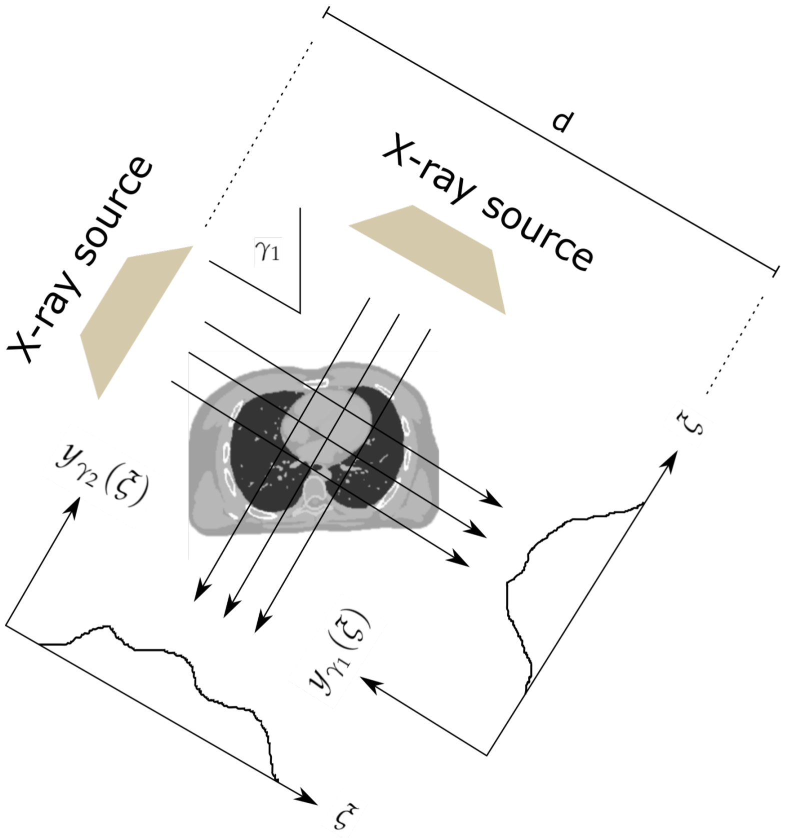


FIGURE 2.2: CT projection process

The object in Figure 2.2 is a 2-dimensional (2-D) image, so at a given angle, the projection is the line integral. We normally use Radon transform to calculate the projection (sinogram) of an object. Then we can do the Fourier Transform (FT) and inverse Fourier Transform (IFT) to the sinogram to reconstruct the 2-D image. In this case, the IFT is also called inverse Radon transform. According to the Fourier slice theorem, if we have an infinite number of projections taken on an infinite number of angles of the object, we can perfectly reconstruct the original object.

In practice, the object is 3-dimensional (3D), which means there is a set of line integrals at a given angle. And the projection for this given angle is made up of this set of line integrals. Moreover, a set of many such projections under different angles organized in 2D is called a sinogram of a 3-D object. Then we can do the FT to these 2-D projection images to get the slices of the FT of the 3-D density of the object according to the Fourier slice theorem. The slices can be interpolated to build up a complete FT of that 3-D density. After that, we can use IFT to recover the 3-D density of the object.

2.2.2 CT reconstruction

As a typical inverse problem, CT reconstruction is to obtain the 3D internal structure information from the X-ray projection images. Back projection is a standard approach to reconstructing CT images. The central idea of the back projection is to smear all the projections to get the CT slice image [38]. The back-projection process will blur the final image; therefore, a particular type of filter is used to solve the blurring problem. The whole process is called filtered back projection (FBP). FBP can acquire CT images fast and convenient, and the reconstructed images are close to real images. However, the reconstructed images may be severely distorted when the number of views (projections) is not enough. To overcome this issue, a lot of iterative techniques have been proposed to use some prior knowledge about the images in the reconstruction process, such as algebraic reconstruction technique (ART) [39], expectation maximization (EM) [40], and simultaneous algebraic reconstruction technique (SART) [41]. Nevertheless, these iterative methods are computationally expensive and time-consuming and cannot reconstruct well in many challenging cases. In recent years, deep learning techniques have achieved state-of-the-art results on CT reconstruction. However, only a few deep learning approaches were proposed for sparse-view CT reconstruction. Zhu et al. [42] introduced a deep neural network called "AUTOMAP" for MRI reconstruction from k-space data. There is no doubt that this approach can be used for CT reconstruction since CT projection data and image data are connected, just like how MRI k-space data and image data are connected. However, this network requires a lot of memory and produces a very computational challenging task for CT data. In addition, the fully-connected layer does not suit the intrinsic structure of the CT imaging process. The fully-connected layer was later replaced by a back-projection operator to reduce the computational burden [43], but the back-projection process is no longer learnable. Hu et al. [44] proposed a learned experts' assessment-based reconstruction network (LEARN) for sparse-view CT images, and the results show the feasibility and merits of the network.

2.3 Neural Networks architectures

The neural network in deep learning is a parametric model that was first inspired by the biological structure of neurons in the human brain. By training with enough data, a neural network can simulate different functions with almost any complexity. The neural network originated from the neural network model first established by Warren McCulloch and Walter Pitts in 1943 [45]. In 1957, Frank Rosenblatt created the first model that can perform pattern recognition, the Perceptron, which is an early form of feed forward neural network [46]. Marvin Minsky and Seymour Papert pointed out two main limitations of perceptrons in their book ”Perceptrons” in 1969: 1. Perceptrons cannot handle XOR problems, 2. Processing large neural networks requires a long calculation time [47]. In 1975, Paul Werbos proposed back propagation, which solved the XOR problem and made neural network learning more efficient. Although the classification effect of the initial neural network is good, due to its strict requirements on data and computing resources, there is no suitable application scenario; it has been relatively unpopular for a long time. Until the 21st century, with the rapid development of big data and computing performance (GPU), neural networks began to blossom, especially widely used in computer vision, natural language processing, and robotics.

A typical neural network normally has three layers: input layer, hidden layer, and output layer. Each layer has many neural network nodes, as simply illustrated in Figure 2.3. The nodes in the input layer store the input values. For the nodes in the hidden layer and output layer, each of them is a linear regression model which has the formula:

$$Y = W^{*}X + B \tag{2.16}$$

where X = $\{x_i\}_{i=1,\dots,N}$ is the input values, W = $\{w_i\}_{i=1,\dots,N}$ is the weights, B = $\{b_i\}_{i=1,\dots,N}$ (optional) is the bias, and Y = $\{y_i\}_{i=1,\dots,N}$ is the output values. The output value Y is then passed to a nonlinear active function which determines the final output. The architecture is shown in Figure 2.4. The nonlinear active function is a significant component in a neural network due to its properties:

1. Non-linearity: When the activation function is linear, the neural network can only deal with linearly separable data. When the activation function is nonlinear, a two-layer neural network can approximate any function, which means it can handle both linear separable and nonlinearly separable data.

2. Differentiability: When the optimization method is based on gradients such as back-propagation, this property is necessary.

3. Monotonicity: A single-layer network can guarantee a convex function.

Rectified linear unit (ReLU) is the most commonly used activation function in modern neural networks as it has a high convergence rate and fast calculation speed in the gradient descent algorithm. However, there is a Dead ReLU Problem that when x¡0, the gradient is 0, the network parameters will never be updated.

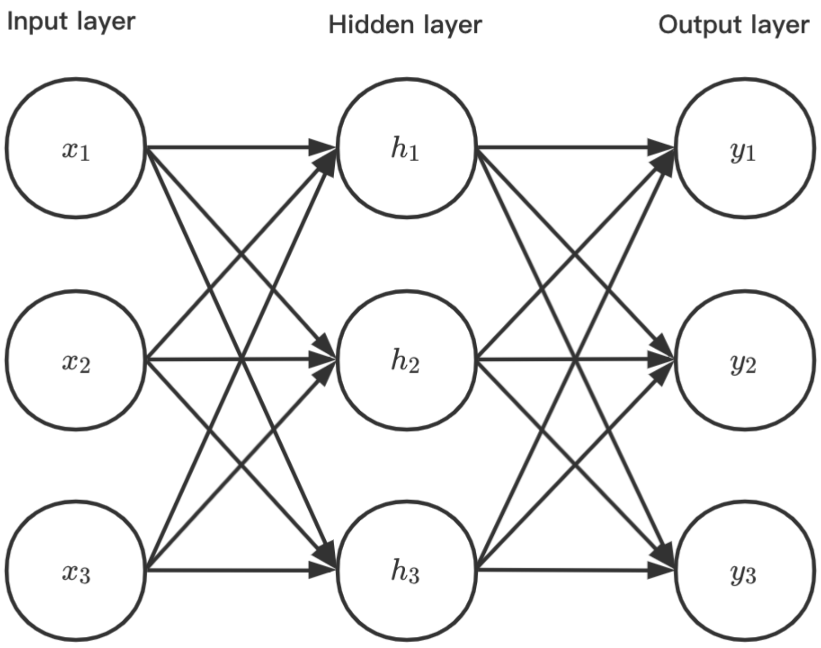


Figure 2.3: A simple three-layer neural network

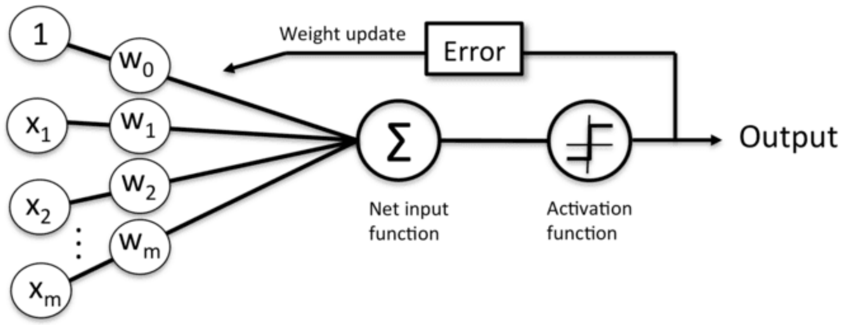


Figure 2.4: The architecture of a single node in the hidden layer and output layer

2.3.1 CNN

There are various neural networks such as original artificial neural networks (ANNs), Convolutional neural networks (CNNs), and Recurrent neural networks (RNNs). CNNs has been widely used in image processing, including image reconstruction. A typical CNN network has an input layer, multiple hidden layers, and an output layer. The hidden layers normally consist of convolutional layers, activation function (ReLU) layers, pooling layers (Max Pooling), and fully connected layers. The convolutional layer uses some convolution kernels to extract features from the image. Each kernel represents different features and will be passed to the convolutional layer, and our training process is to train these different kernels. During the convolution process, we often use padding such as zero padding and circular padding to make the image size the same before and after convolution and utilize the edge information. The activation function layer is to use activation functions to activate each neural node in the network. And in the pooling layer, we normally use downsampling to reduce the network training parameters and the degree of overfitting of the model. CNNs can extract the local features from images very well and require much less preprocessing, thus bringing significant image processing improvements, especially in image segmentation and recognition. A basic CNN model is shown in Figure 2.5 [48].

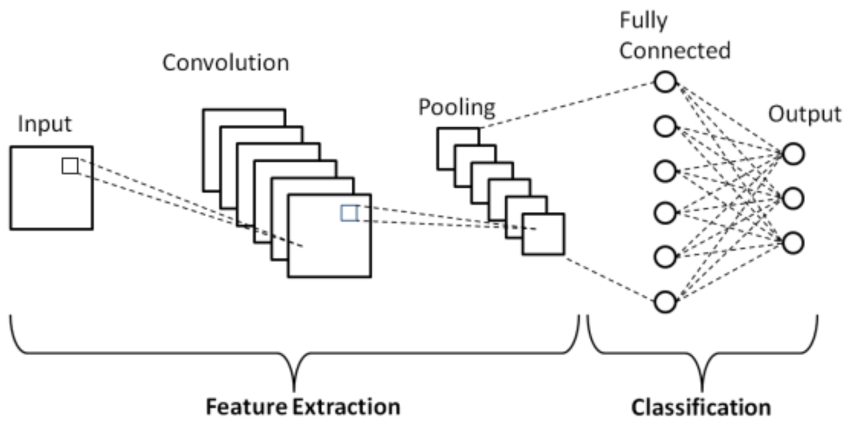


FIGURE 2.5: Basic CNN architecture

2.3.2 UNet

Based on the concept of CNNs that convert images to vectors for classification, UNet [18] was born for image segmentation. UNet first converts images to vectors and then converts the vectors back to images. This perseveres the structural integrity of the image, which reduces the distortion enormously. The UNet architecture is shown below.

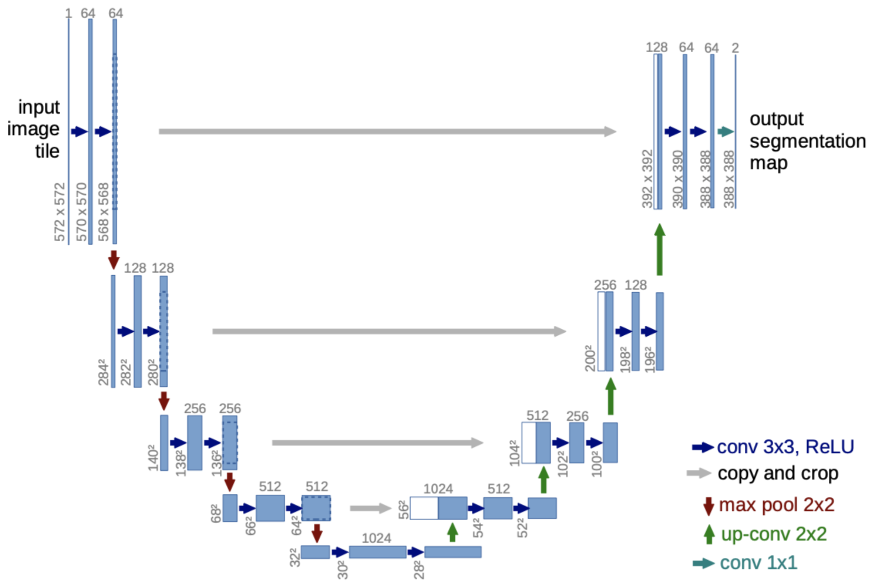


FIGURE 2.6: UNet architecture

From Figure 2.5, we can see that the whole network can be divided into two parts. The left part is the Encoder, and the right part is the Decoder. Each layer in the Encoder uses a convolution function (3*3) to extract the feature maps and a maxpooling function (2*2) to reduce the complexity of the network, just like a typical CNN. At every step, the number of channels is doubled the spatial dimension is halved. In the decoder part, we use an upsampling function (2*2) to upsample the feature maps and a feature concatenation to combine the upsampled feature maps with the feature maps in the same layer in the Encoder. After that, we do a convolution operation to extract feature maps. At every step, the spatial dimension is doubled, and the number of channels is halved.

Basically, there are two types of information (features) in the UNet:

Low-level (deep) information: Low-level information is the multiple downsampled low-resolution information. The contextual semantic information of the segmentation can be obtained from the low-level information and this information can be used for classification of objects. So low-level information is very important in the classification problem.

High-level (shallow) information: High-level information is the high-resolution information that is directly transferred from the Encoder to the Decoder at the same layer after concatenation operation. It can provide more refined features for segmentation, such as gradients.

UNet architecture is widely used in medical images such as CT, MRI mainly due to :

1. Medical images have blurry boundaries and complex gradients, requiring more high-level information. High-level information is used for precise segmentation.

2. The semantics of human body are simple and clear due to the relatively fixed internal structure of the human body and the regular distribution of segmentation targets in the human body images. Therefore, low-level information can be used for target object recognition.

UNet combines low-level information (providing a basis for object category recognition) and high-level information (providing a basis for precise segmentation and positioning), which is perfectly suitable for medical image segmentation [49].

2.3.3 FBPConvNet

Based on the UNet and FBP concepts, FBPConvNet [30] was proposed for CT reconstruction. FBPConvNet first applies the discretized FBP to the measurements and then use this as the input of a UNet network to simplify the learning complexity. The UNet network is trained to regress the FBP results to the ground truth images. A residual (skip) connection between input and output is added in this network compared with UNet. The architecture of FBPConvNet is shown in Figure 2.7.

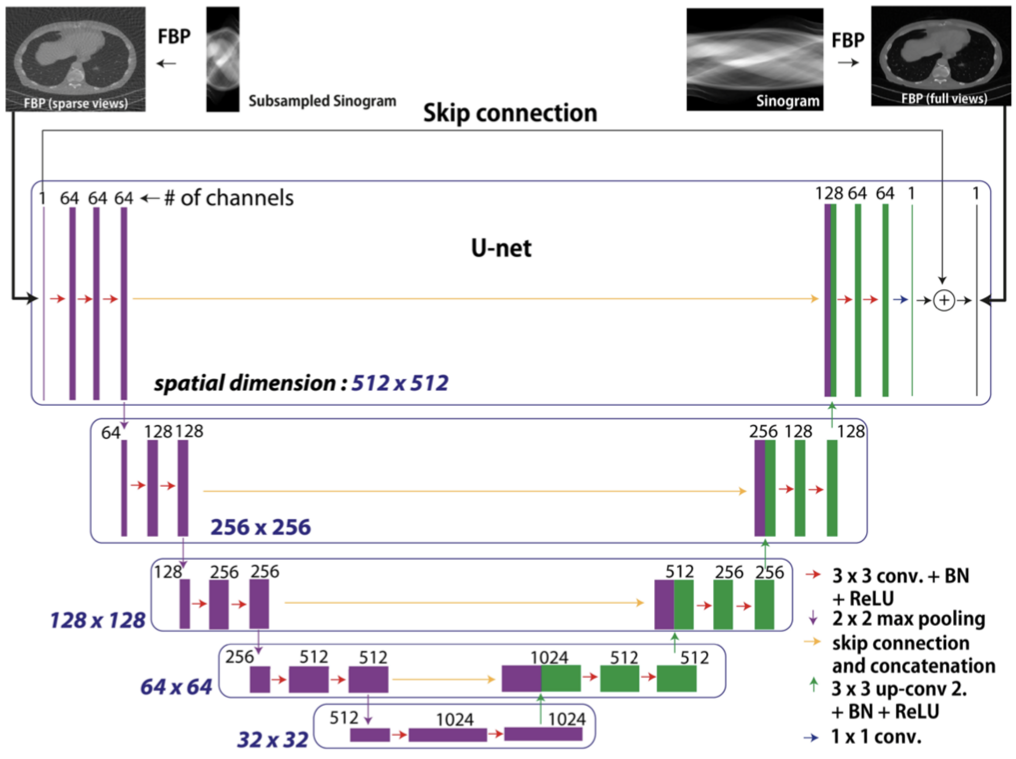


FIGURE 2.7: FBPConvNet architecture

There are few strategies of the FBPConvNet:

1. A residual (skip) connection between input and output is added in this network compared with the original UNet. The network can actually learn the difference between output and input. As we talked about in the supervised learning method, this can mitigate the vanishing gradient problem in the training process and achieve a noticeable increase in performance compared with the same network without it.

2. Zero padding is used in each convolution layer to ensure that the image size does not decrease after every convolution. This can also preserve some edge information.

3. The last layer in the original UNet is replaced by a 1*1 convolutional layer to reduce the 64 channels to a single output image, as the original UNet results in two channels: foreground and background.

Chapter 3

Methods

3.1 Overview approaches

The problem is to reconstruct spare-view CT images from measurements only. Here, in order to evaluate the performance of the self-supervised learning method, we also use a supervised learning approach for comparison. Therefore, we present two approaches in this project as follows:

1. EI approach: Train a reconstruction function f_θ by using measurement consistency and invariant set consistency.

2. Supervised approach: Train a reconstruction function f_θ by using (x,y) pairs.

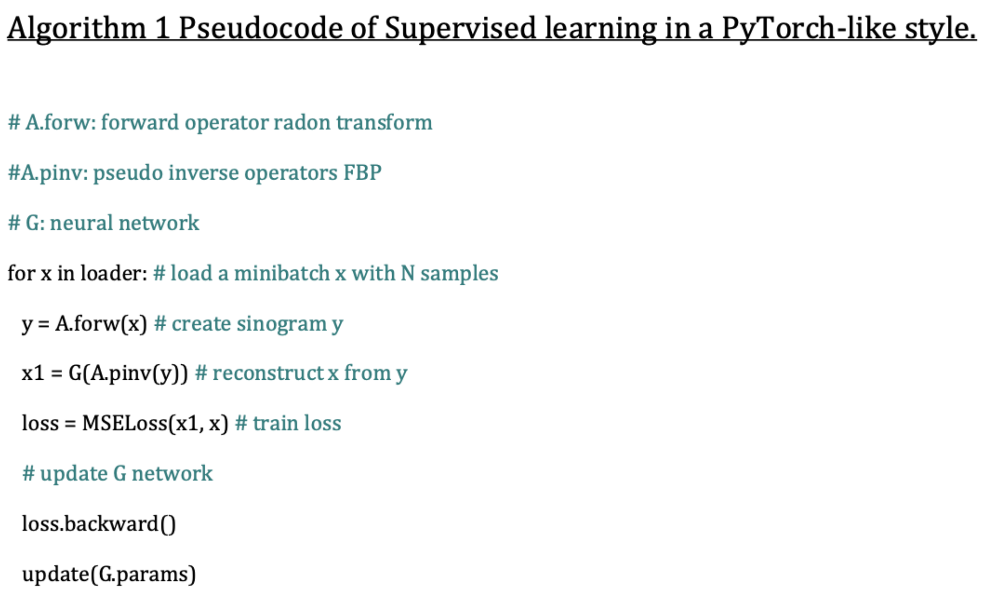
For both approaches, we use a trainable deep neural network G_θ:$\mathbb{R}^n \rightarrow \mathbb{R}^n$ and an approximate linear inverse $A^\dagger \in \mathbb{R}^{n\times m}$ to define the reconstruction function as $f_\theta = G_\theta \circ A^\dagger$. Practically $A^\dagger$ can be chosen as any functions that is cheap to compute. Here we chose the FBP function as the linear inverse $A^\dagger$ to first project y into $\mathbb{R}^n$ to simplify the learning complexity.

3.2 Supervised approach

In this approach, since we have the ground truth x and measurements y, we only need to calculate the error between reconstructed image $\hat{\boldsymbol{x}} = f_\theta(y)$ and ground truth x as the loss function. Then we can use gradient descent to adjust the parameters of the network according to the gradient of the loss function. The optimization is shown below:

$$\arg\min_\theta \mathbb{E}_y \mathcal{L}\left(\hat{\boldsymbol{x}}, x\right) \tag{3.1}$$

The pseudocode is shown in Figure 3.1.



Algorithm 1 Pseudocode of Supervised learning in a PyTorch-like style.

```
# A.forw: forward operator radon transform
#A.pinv: pseudo inverse operators FBP
# G: neural network
for x in loader: # load a minibatch x with N samples
  y = A.forw(x) # create sinogram y
  x1 = G(A.pinv(y)) # reconstruct x from y
  loss = MSELoss(x1, x) # train loss
  # update G network
  loss.backward()
  update(G.params)
```

FIGURE 3.1: Supervised learning pseudocode

3.3 EI approach

In this approach, we need to use measurement consistency and invariant set consistency to constrain the plausible reconstructions.

Therefore, given compressed measurement y, we first need to use $f_\theta(y)$ to calculate the reconstructed image $x^{(1)}$ as an estimate of the ground truth x. Then we can use the measurement consistency between $Af_\theta(y)$ and y to make sure $Ax^{(1)}$ stays close to the input measurement y. However, we cannot learn beyond the range space of A. So we use the invariant set consistency to get $x^{(2)} = T_g x^{(1)}$, where T_g is a group of rotation transformations, and pass $x^{(2)}$ to $A \circ f$ to get $x^{(3)}$. The whole process is illustrated in Figure 3.2.

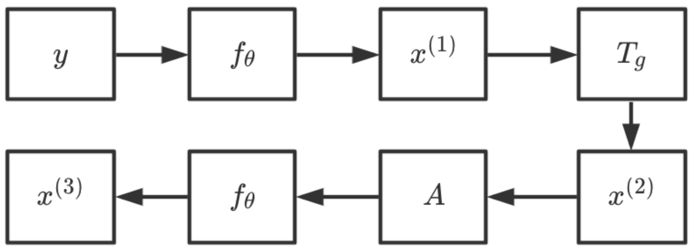


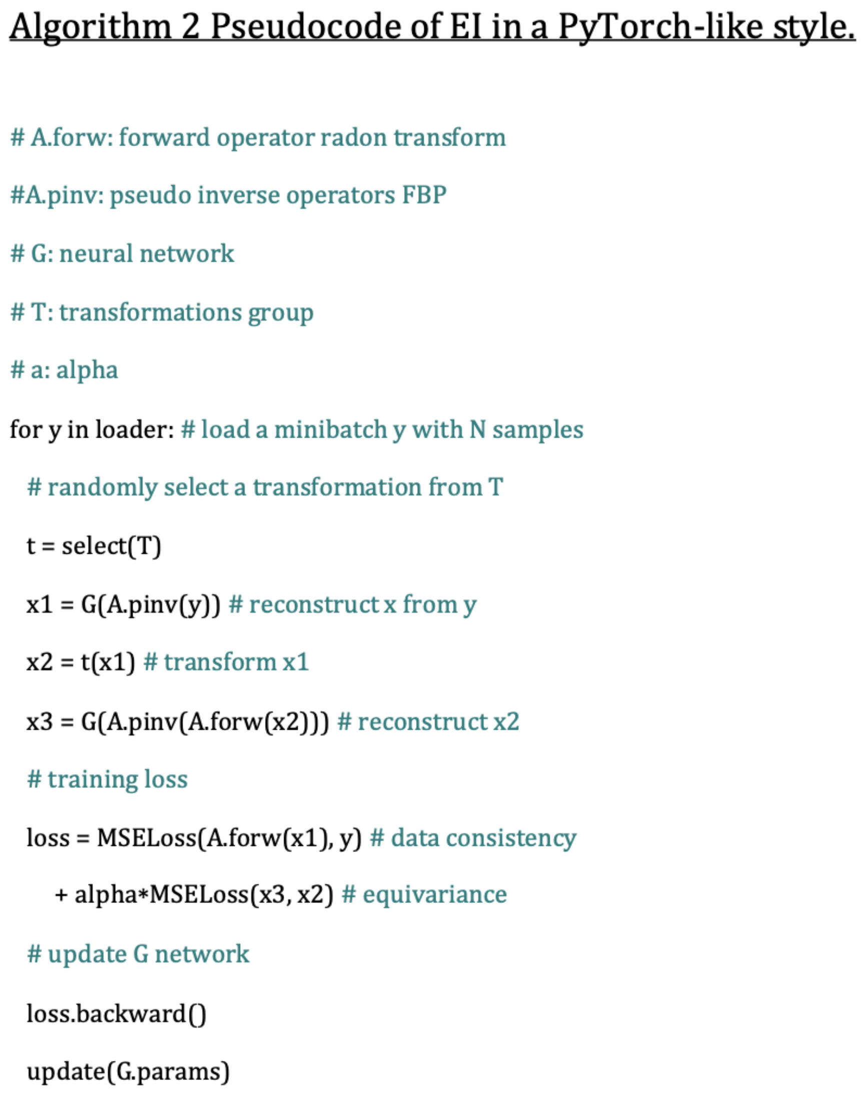
FIGURE 3.2: EI strategy

The parameters of the network are updated according to the error between $Ax^{(1)}$ and y, and the error between $x^{(2)}$ and $x^{(3)}$. The optimization is shown below:

$$\arg\min_{\theta} \mathbb{E}_{y,g} \left\{ \mathcal{L}\left(Ax^{(1)}, y\right) + \alpha \mathcal{L}\left(x^{(2)}, x^{(3)}\right) \right\} \tag{3.2}$$

where the first term $\mathcal{L}\left(Ax^{(1)}, y\right)$ is for data consistency and the second term $\alpha \mathcal{L}\left(x^{(2)}, x^{(3)}\right)$ is for invariant set consistency, α is the trade-off parameter to control the strength of invariant set consistency, and $\mathcal{L}$ is the error function.

And the pseudocode is shown in Figure 3.3



Algorithm 2 Pseudocode of EI in a PyTorch-like style.

```
# A.forw: forward operator radon transform
#A.pinv: pseudo inverse operators FBP
# G: neural network
# T: transformations group
# a: alpha
for y in loader: # load a minibatch y with N samples
  # randomly select a transformation from T
  t = select(T)
  x1 = G(A.pinv(y)) # reconstruct x from y
  x2 = t(x1) # transform x1
  x3 = G(A.pinv(A.forw(x2))) # reconstruct x2
  # training loss
  loss = MSELoss(A.forw(x1), y) # data consistency
     + alpha*MSELoss(x3, x2) # equivariance
  # update G network
  loss.backward()
  update(G.params)
```

FIGURE 3.3: EI pseudocode

Chapter 4

Experiments and Results

We apply the self-supervised learning method EI on sparse-view CT image reconstruction problem and evaluate the performance compare to the fully supervised learning method. We designed our experiments to address the following questions: (i) how well does the EI compare to supervised learning? (ii) how does the number of views affect the performance of both EI and supervised learning? (iii) what value of hyper-parameter α should be to achieve the best performance? (iv) how does the number of transformations (rotation) affect the performance of EI?

4.1 Experiment Setup and implementation

All of our codes were written in Python and implemented in PyTorch. The forward operator A of the X-ray CT is the discrete radon transform, and the FBP (iradon transform) is used to approximate the inverse operator $A^{\dagger}$. Both radon transform function and FBP function are from the package TorchRadon [50] which is a PyTorch extension written in CUDA that implements differentiable routines for solving (CT) reconstruction problems. It should be noted that the package TorchRadon is only building for PyTorch 1.5 to 1.7. The dataset we used is the CT100 dataset [51] which is a public real CT clinic dataset. This dataset consists of 100 middle slices of CT images taken from 69 different patients. We resize the image from 512*512 to 128*128 as the ground truth to reduce the complexity of the neural network and the time for learning. Then we use the *radon* on the ground truth to create 50-views sinograms. The first 90 sinograms are used for training, and the last 10 sinograms are used for testing. The network we used in this project is a simplified version of FBPConvNet [7] which only has four layers as our input images have been resized to 128*128 resolution. The architecture of the FBPConvNet is shown below,

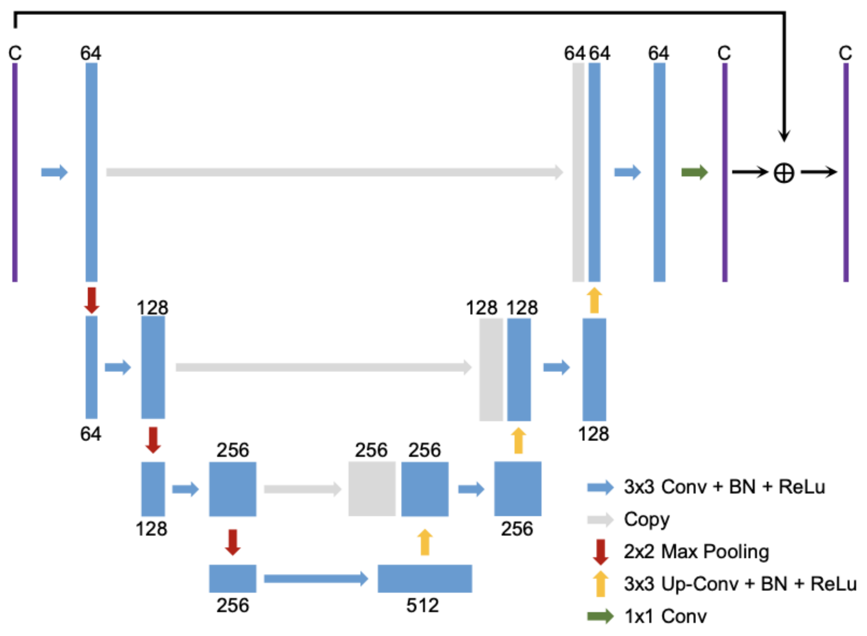


FIGURE 4.1: Simplified FBPConvNet architecture

And the error function (loss function) $\mathcal{L}$ we used in both supervised and EI methods is the mean squared error (MSE) as the CT images reconstruction is a regression model. The optimizer we used in both methods is the Adam [52] optimizer with a batch size of 2 and an initial learning rate of 0.0005 as it is very efficient and only requires first-order gradients with little memory requirement. The weight decay of the optimizer is 10^{-8}. We train the networks of both methods for 5000 epochs and keep the learning rate invariant for the first 2000 epochs and then shrink it by a factor of 0.1 every 1000 epochs. We store the trained model for every 100 epochs. As for the invariant set consistency in EI, we use the rotation invariance of the CT images and $\mathcal{G}$ is the group of rotations by 1 degree ($|\mathcal{G}| = 360$). For all experiments, we use Peak Signal-to-Noise Ratio (PSNR) to evaluate the performance of our methods.

4.2 Supervised method VS EI method

In this section, we cover the experiment of how well the EI method can achieve compared to the supervised method. We also calculate the FBP ($A^{\dagger}$y) for comparison. The final result of 50-views CT images reconstruction is shown in Figure 4.2. We choose α as 100 since this is a good value to achieve the equivariance in the EI method as illustrated in Section 4.4.

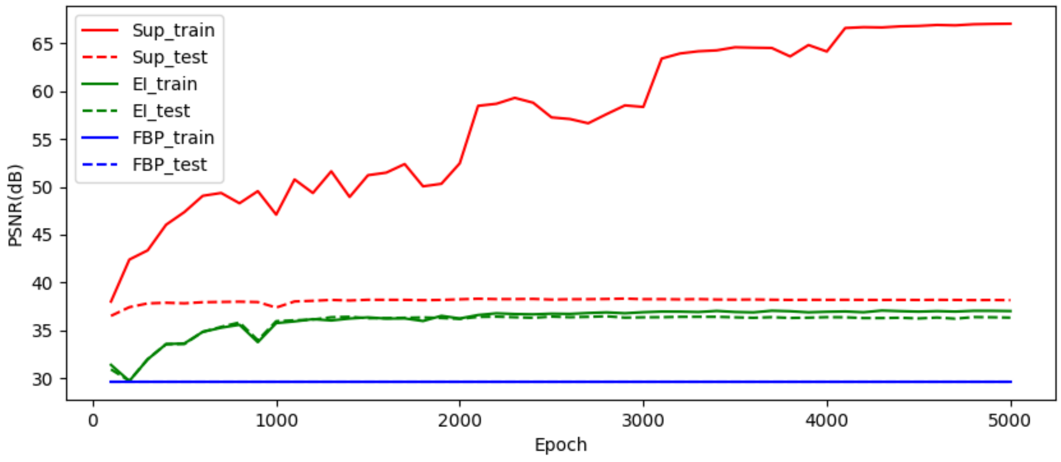


FIGURE 4.2: Reconstruction performance of 50-view measurements for training and testing

From Figure 4.2, we have two main observations:

(1) The supervised learning method is better than the EI method as expected, since the supervised learning method has the ground truth. However, the gap between these two methods is small, which means the performance of the EI method without ground truth is almost as good as the performance of the supervised method with ground truth. And the EI performance is about 7dB better than the FBP, which also demonstrates the EI method can learn the null space of the images.

(2) There is a huge gap between training and testing errors in the supervised learning method, implying that the network may be overfitting. However, this phenomenon did not occur in the EI method. This can be due to the fact that the EI method constrains the network to a set of functions (those equivariant functions on data) and thus has better generalization properties.

An examples of 50-view CT image reconstruction is shown in Figure 4.3.

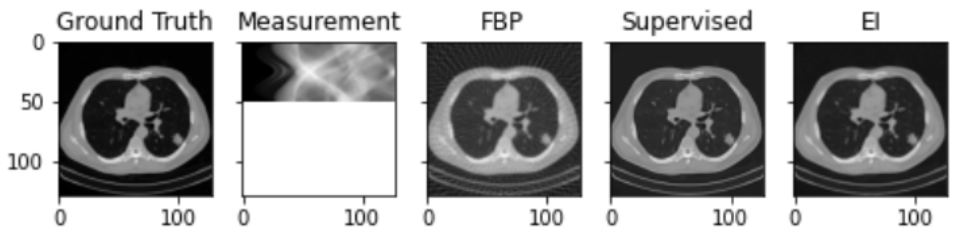


FIGURE 4.3: An examples of 50-view CT image reconstruction

From the figure above, we can see the EI method has removed the artefacts shown in the FBP and has a good performance on CT reconstruction.

4.3 The effect of the number of views

As we talked about in Section 2.2.1, if we have an infinite number of views from an infinite number of angles, we can perfectly reconstruct the original CT images. Therefore, in this section, we explore the effect of the number of views on both supervised and EI methods.

We use the measurements from 10 views to 120 views with 10 views intervals for training and testing. The value of hyperparameter α is 100 for all numbers of views. The results of the testing reconstruction PSNR is shown in Figure 4.4

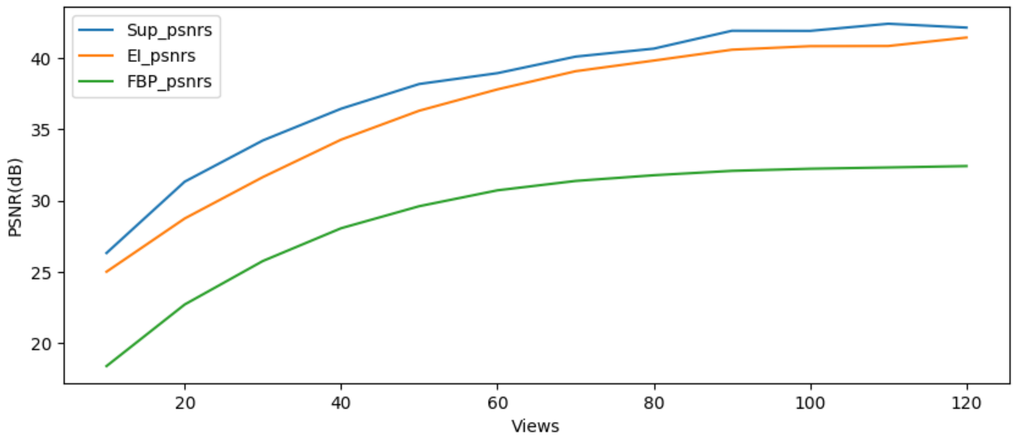


FIGURE 4.4: Reconstruction performance of different number of views

The figure above shows that the reconstruction PSNR continuously increases as the number of views grows, which verifies that the performance of CT images reconstruction is strongly related to the number of views. The more number of views, the better performance of reconstruction. Moreover, we can also observe that the growing speed of the reconstruction performance decreases as the number of views increases. Another observation is that the performance of the supervised method is always better than the EI method, but the gap between them is also always small, which proves that the EI method has a good performance no matter the number of views.

4.4 The value selection of hyperparameter α

The hyperparameter α is very important in the EI approach. It controls the strength of equivariance. If we set the value too large, it may limit the power of measurement consistency. If we set the value too small, the equivariance may be negligible. Moreover, the number of views may also affect the choice of the value of α. Therefore, this section explores how to choose the value of α to achieve the best performance in the EI method.

We test five different values of α (0.1,1,10,100,1000) for five different number of views (10,30,50,70,90). The results of the testing reconstruction PSNR is shown below

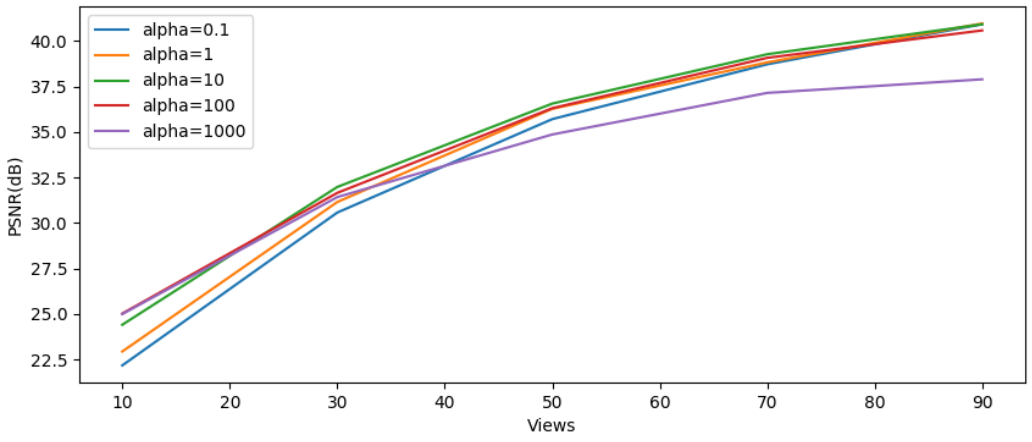


FIGURE 4.5: Reconstruction performance of 50-view measurements for training and testing

From Figure 4.5, we can see that the number of views does affect the choice of the value of α. For the low number of views from 10 to 30, we can choose a big value such as 100 or an even bigger value of 1000 for α to achieve better performance. For the high number of views from 70 to 90, we can choose a small value like 10,1 or an even smaller value 0.1 for α to achieve better performance. We can draw a conclusion from the above observations that the importance of equivariance in the low number of views is greater than that in the high number of views. However, for all different numbers of views, a middle value of α like 10 or 100 can achieve good performance. Therefore, we can choose the value of α as 10 or 100 for simplicity in our learning process.

4.5 The effect of the number of transformations

In Section 3.3, we use a group of rotation transformations T_g to achieve the invariant set consistency. However, how many transformations should we use to achieve the best performance? Does the number of transformations affect the final performance a lot? Is the number of transformations related to the number of views? We do not know the answer yet. Therefore, in this section, we cover the experiment of how the number of transformations affects the reconstruction performance.

We set the α as 100 as before and use five different numbers of transformations (1,3,5,7,10) for three different numbers of views (10,30,50). The transformations are randomly selected from the transformation group. The results of the testing reconstruction PSNR is shown in Figure 4.6

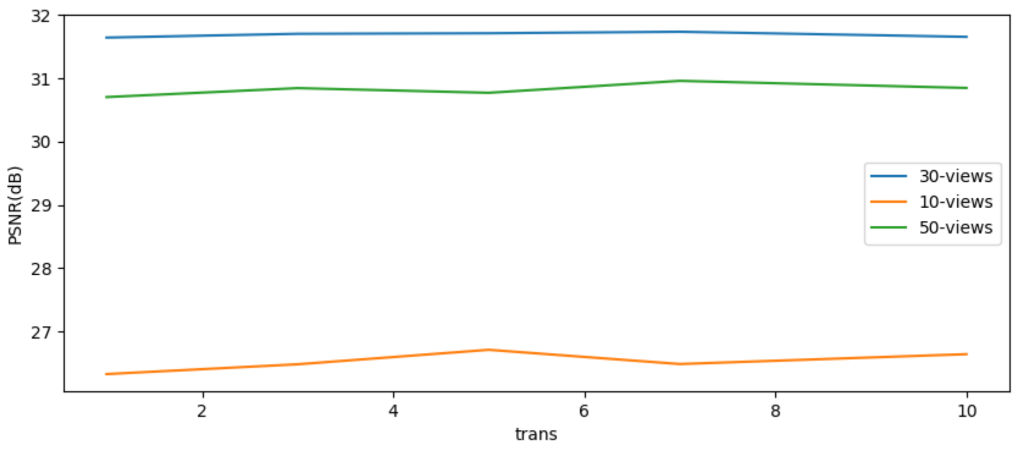


FIGURE 4.6: Reconstruction performance of different number of transformations for different number of views

The figure above shows that the number of transformations has little effect on the reconstruction performance for three different numbers of views. In other words, the EI method is insensitive to the number of transformations regardless of the number of views. Hence, we can choose a small number of transformations such as 5 to reduce complexity and save time while keeping diversity.

Chapter 5

Discussion

The experiments results indicate that the EI approach can achieve good performance on CT image reconstruction in different numbers of views by choosing an appropriate value of the hyperparameter α and a small number of transformations. The value selection of α is affected by the number of views. We can choose a big value for α for the low number of views, a medium value for α for the middle number of views, and a small value for α for the high number of views to achieve the best performance. This can be explained as for the low number of views, the information about the original CT images is also scarce, the strength of equivariance should be high to learn more information beyond the range space of A. For the high number of views, we have a lot of information about the original CT images. The strength of equivariance should be low in order to not disturb the measurement consistency. Or, we can choose a medium value for α for simplicity as it has good performance for all numbers of views. The reconstruction performance is insensitive to the number of rotation transformations, so we should choose a small number of transformations to save time and resources. This can be illustrated as long as we have enough epochs, we can reach the final reconstruction performance as each epoch may represent different transformations. And the reason we use a small number of transformations rather than one transformation is to ensure the diversity of the transformations as we select the transformation randomly from the transformation group. Although the performance of EI cannot reach that of the supervised method, the gap between them is tiny, which implies the success of the EI method as a self-supervised learning method.

In fact, we can add an adversarial network to the EI method to ensure the reconstructed image $x^{(1)}$ from $f_\theta(y)$ and rotation transformed $x^{(2)}$ are identically distributed or add the invariant set consistency to the supervised learning method. However, both of these two strategies provide a very slight improvement, as illustrated in [7]. We did not do the experiments about these two strategies in this project, but it is necessary to know their existence.

The EI approach can also be applied for image inpainting tasks by using the shift invariance exited in natural images. The recovery performance is as well as CT reconstruction. Although EI has the potential to be used in other imaging tasks, we should notice the necessary condition for EI method that the range space of A is not invariant to all T_g. For instance, the shift invariance can not be applied to Fourier based measurement operators such as deblurring, superresolution, and MRI as A would be invariant to the shifts.

Chapter 6

Conclusion

We implement the EI method to solve the CT reconstruction inverse problem without ground truth and compare it with the supervised learning method to estimate the performance. We also study the effects of different numbers of views, hyperparameter α, number of transformations on the performance of the EI approach. With an appropriate value of α and a small number of transformations, the EI method can achieve good reconstruction performance, which is very close to that of the supervised learning method no matter the number of views. Most of the self-supervised learning methods can only solve denoising problems where the forward matrix is identity. The EI method is a new self-supervised learning method that can deal with the inverse problems that the forward operator A has a non-trivial nullspace. While the equivariance may be used for other image reconstruction tasks, we should keep in mind that not any combination of A and group transformation $\mathcal{G}$ is useful for learning beyond the range space.

In future work, we can apply mixed types of transformations during the training and verify whether it can help improve the convergence time and performance. It is also fascinating to see whether the EI method can be used to solve nonlinear imaging problems.

Appendix A

Appendix

The whole codes can be divided into 6 python files: data.py, Netwrok_arch.py, Utils.py, train.py, test.py and main.py.

The data.py file stores a Dataset function that is used for loading the original CT images and two Dataloader functions to create the dataloaders for training data and testing data.

The Network_arch.py file stores the network architecture we used for our project.

The Utils.py file stores some functions that we used for calculating the PSNR and plotting reconstructed CT images.

The train.py file stores the training loops and training functions for both supervised learning and EI.

The test.py file stores the testing loops and testing functions for both supervised learning and EI.

And the main.py file stores the codes for environments setup, packages import, and the main codes for the experiments.

The 6 files are shown below

Data.py

```
# -*- coding: utf-8 -*-
"""data.ipynb

Automatically generated by Colaboratory.

Original file is located at
```

```
    https://colab.research.google.com/drive/1IKebxF4LkLaslpNT2ti1cbGeCA1LXBNn
"""

import scipy.io as io
from torch.utils.data import DataLoader, Dataset
import torch
import numpy as np
class MyDataset(Dataset):
   def __init__(self, gt_path):
       data = io.loadmat(gt_path)   #load original dara
       self.ground_truth = torch.FloatTensor(data['Groud_Truth'])   #convert
    to float tensor

   def __getitem__(self, index):
       x = self.ground_truth[index]  #get data by index
       return x

   def __len__(self):
       return len(self.ground_truth)  #get the length of data

def dataloader_train(path,batch):
    train_dataset = MyDataset(path)
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                               batch_size=batch,
                                               shuffle=True)    # create the
    dataloader for specific batch_size
    return train_loader,train_dataset
def dataloader_test(path,batch):
    test_dataset = MyDataset(path)
    test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                               batch_size=batch,
                                               shuffle=False)   # create the
    dataloader for specific batch_size
    return test_loader,test_dataset
```

Network_arch.py

```
# -*- coding: utf-8 -*-
"""Network_arch.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1sZnvltw9io3JPv1a--V2xVLMDMYroQhr
```

```
"""

import torch
import torch.nn as nn
import torch.nn.init as init

class UNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=1):
        super(UNet, self).__init__()
        self.Maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.Conv1 = conv_block1(ch_in=in_channels, ch_out=64)
        self.Conv2 = conv_block(ch_in=64, ch_out=128)
        self.Conv3 = conv_block(ch_in=128, ch_out=256)
        self.Conv4 = conv_block(ch_in=256, ch_out=512)   # Encoder
        #self.Conv5 = conv_block(ch_in=512, ch_out=1024)

        #self.Up5 = up_conv(ch_in=1024, ch_out=512)
        #self.Up_conv5 = conv_block(ch_in=1024, ch_out=512)

        self.Up4 = up_conv(ch_in=512, ch_out=256)
        self.Up_conv4 = conv_block(ch_in=512, ch_out=256)

        self.Up3 = up_conv(ch_in=256, ch_out=128)
        self.Up_conv3 = conv_block(ch_in=256, ch_out=128)

        self.Up2 = up_conv(ch_in=128, ch_out=64)
        self.Up_conv2 = conv_block(ch_in=128, ch_out=64)    #Decoder

        self.Conv_1x1 = nn.Conv2d(in_channels=64, out_channels=out_channels,
    kernel_size=1, stride=1, padding=0)


    def forward(self, x):
        # encoding path
        cat_dim = 1
        input = x
        x1 = self.Conv1(input)

        x2 = self.Maxpool(x1)
        x2 = self.Conv2(x2)

        x3 = self.Maxpool(x2)
        x3 = self.Conv3(x3)
        x4 = self.Maxpool(x3)
        x4 = self.Conv4(x4)

        # x5 = self.Maxpool(x4)
        # x5 = self.Conv5(x5)

        # #decoding + concat path
        # d5 = self.Up5(x5)
        # d5 = torch.cat((x4, d5), dim=cat_dim)

        # d5 = self.Up_conv5(d5)

        d4 = self.Up4(x4)
        d4 = torch.cat((x3, d4), dim=cat_dim)
        d4 = self.Up_conv4(d4)
        d3 = self.Up3(d4)
        d3 = torch.cat((x2, d3), dim=cat_dim)
        d3 = self.Up_conv3(d3)

        d2 = self.Up2(d3)
        d2 = torch.cat((x1, d2), dim=cat_dim)
        d2 = self.Up_conv2(d2)

        # d2 = self.Up2(d3)
        # d2 = torch.cat((x1, d2), dim=cat_dim)
        # d2 = self.Up_conv2(d2)

        d1 = self.Conv_1x1(d2)
        d1 = d1 + input
        return d1


class conv_block(nn.Module):
    def __init__(self, ch_in, ch_out):
        super(conv_block, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1,
    bias=True,padding_mode='zeros'), # zero padding
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True),
            nn.Conv2d(ch_out, ch_out, kernel_size=3, stride=1, padding=1,
    bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x
class conv_block1(nn.Module):
    def __init__(self, ch_in, ch_out):
        super(conv_block1, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1,
    bias=True, padding_mode='circular'), # circular padding
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True),
            nn.Conv2d(ch_out, ch_out, kernel_size=3, stride=1, padding=1,
    bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x

class up_conv(nn.Module):
    def __init__(self, ch_in, ch_out):
        super(up_conv, self).__init__()
        self.up = nn.Sequential(
            nn.Upsample(scale_factor=2),
            nn.Conv2d(ch_in, ch_out, kernel_size=3, stride=1, padding=1,
    bias=True),
            nn.BatchNorm2d(ch_out),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.up(x)
        return x
```

Utils.py

```
# -*- coding: utf-8 -*-
"""Utils.ipynb
```

```
Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/14oDHThNjD9RekhGr9cfK2bu5LuhNd59Z
"""

import matplotlib.pyplot as plt
import torch
import numpy as np

def PSNR_cal(original, compressed,max_pixel):
    mse = torch.mean((original - compressed) ** 2)
    if(mse == 0):  # MSE is zero means no noise is present in the signal .
                  # Therefore PSNR have no importance.
        return 100
    psnr = 20 * torch.log10(max_pixel / torch.sqrt(mse))
    return psnr.detach().cpu().numpy()
def show_images(imgs, titles=None, keep_range=True, shape=None, figsize=(8,
    8.5)):
    imgs = [x.cpu().numpy() for x in imgs]
    combined_data = np.array(imgs)

    if titles is None:
        titles = [str(i) for i in range(combined_data.shape[0])]

    # Get the min and max of all images
    if keep_range:
        _min, _max = np.amin(combined_data), np.amax(combined_data)
    else:
        _min, _max = None, None

    if shape is None:
        shape = (1, len(imgs))

    fig, axes = plt.subplots(*shape, figsize=figsize, sharex=True, sharey=True)
    ax = axes.ravel()
    for i, (img, title) in enumerate(zip(imgs, titles)):
        ax[i].imshow(img, cmap=plt.cm.Greys_r, vmin=_min, vmax=_max)
        ax[i].set_title(title)
```

Train.py

```
# -*- coding: utf-8 -*-
"""train.ipynb
```

```
Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1dGXTh7Mw5M4peNh5zqbXcWUCZjciYt5q
"""

import warnings
import torch
import numpy as np
from utils import PSNR_cal
from kornia import rotate
from random import sample
from network_arch import UNet
import os
def EI_train_loop(dataloader, model, loss_fn, optimizer,radon,alpha,trans):
    model.train()
    train_loss, train_psnr=[], []
    for batch,X in enumerate(dataloader):
        X = X.cuda()
        y = radon.forward(X)
        filtered_sinogram = radon.filter_sinogram(y)
        x1 = model(radon.backprojection(filtered_sinogram))  # calculate the
    FBP
        rotate_angles = np.linspace(1, 360, 360, endpoint=True)

        if trans == 1 :

          angles=sample(list(rotate_angles),1)
          x2_1 = rotate(x1, torch.Tensor([angles[0]]).cuda())
          # x2 = torch.stack([x2_1,x2_2,x2_3,x2_4,x2_5],dim=0)
          sinogram1 = radon.forward(x2_1)
          x3_1 = model(radon.backprojection(radon.filter_sinogram(sinogram1)))
          loss_trans=loss_fn(x3_1,x2_1)
          # sinogram = radon.forward(x2)
          # x3 = model(radon.backprojection(radon.filter_sinogram(sinogram)))
          # loss_trans=loss_fn(x3,x2)
          loss = loss_fn(radon.forward(x1), y)+alpha*loss_trans
          train_loss.append(loss.item())
          train_psnr.append(PSNR_cal(X, x1,torch.max(x1)))
          # Backpropagation
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()
        if trans == 3 :

          angles=sample(list(rotate_angles),3)
          x2_1 = rotate(x1, torch.Tensor([angles[0]]).cuda())
          x2_2 = rotate(x1, torch.Tensor([angles[1]]).cuda())
          x2_3 = rotate(x1, torch.Tensor([angles[2]]).cuda())

          # x2 = torch.stack([x2_1,x2_2,x2_3,x2_4,x2_5],dim=0)
          sinogram1 = radon.forward(x2_1)
          sinogram2 = radon.forward(x2_2)
          sinogram3 = radon.forward(x2_3)

          x3_1 = model(radon.backprojection(radon.filter_sinogram(sinogram1)))
          x3_2 = model(radon.backprojection(radon.filter_sinogram(sinogram2)))
          x3_3 = model(radon.backprojection(radon.filter_sinogram(sinogram3)))


    loss_trans=(loss_fn(x3_1,x2_1)+loss_fn(x3_2,x2_2)+loss_fn(x3_3,x2_3))/3
          # sinogram = radon.forward(x2)
          # x3 = model(radon.backprojection(radon.filter_sinogram(sinogram)))
          # loss_trans=loss_fn(x3,x2)
          loss = loss_fn(radon.forward(x1), y)+alpha*loss_trans
          train_loss.append(loss.item())
          train_psnr.append(PSNR_cal(X, x1,torch.max(x1)))
          # Backpropagation
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()

        if trans == 5 :

          angles=sample(list(rotate_angles),5)
          x2_1 = rotate(x1, torch.Tensor([angles[0]]).cuda())
          x2_2 = rotate(x1, torch.Tensor([angles[1]]).cuda())
          x2_3 = rotate(x1, torch.Tensor([angles[2]]).cuda())
          x2_4 = rotate(x1, torch.Tensor([angles[3]]).cuda())
          x2_5 = rotate(x1, torch.Tensor([angles[4]]).cuda())
          # x2 = torch.stack([x2_1,x2_2,x2_3,x2_4,x2_5],dim=0)
          sinogram1 = radon.forward(x2_1)
          sinogram2 = radon.forward(x2_2)
          sinogram3 = radon.forward(x2_3)
          sinogram4 = radon.forward(x2_4)
          sinogram5 = radon.forward(x2_5)
          x3_1 = model(radon.backprojection(radon.filter_sinogram(sinogram1)))
          x3_2 = model(radon.backprojection(radon.filter_sinogram(sinogram2)))
          x3_3 = model(radon.backprojection(radon.filter_sinogram(sinogram3)))
          x3_4 = model(radon.backprojection(radon.filter_sinogram(sinogram4)))
          x3_5 = model(radon.backprojection(radon.filter_sinogram(sinogram5)))

    loss_trans=(loss_fn(x3_1,x2_1)+loss_fn(x3_2,x2_2)+loss_fn(x3_3,x2_3)+loss_fn(x3_4,x2_4)+lo
          # sinogram = radon.forward(x2)
          # x3 = model(radon.backprojection(radon.filter_sinogram(sinogram)))
          # loss_trans=loss_fn(x3,x2)
          loss = loss_fn(radon.forward(x1), y)+alpha*loss_trans
          train_loss.append(loss.item())
          train_psnr.append(PSNR_cal(X, x1,torch.max(x1)))
          # Backpropagation
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()

        if trans == 7 :

          angles=sample(list(rotate_angles),7)
          x2_1 = rotate(x1, torch.Tensor([angles[0]]).cuda())
          x2_2 = rotate(x1, torch.Tensor([angles[1]]).cuda())
          x2_3 = rotate(x1, torch.Tensor([angles[2]]).cuda())
          x2_4 = rotate(x1, torch.Tensor([angles[3]]).cuda())
          x2_5 = rotate(x1, torch.Tensor([angles[4]]).cuda())
          x2_6 = rotate(x1, torch.Tensor([angles[5]]).cuda())
          x2_7 = rotate(x1, torch.Tensor([angles[6]]).cuda())

          # x2 = torch.stack([x2_1,x2_2,x2_3,x2_4,x2_5],dim=0)
          sinogram1 = radon.forward(x2_1)
          sinogram2 = radon.forward(x2_2)
          sinogram3 = radon.forward(x2_3)
          sinogram4 = radon.forward(x2_4)
          sinogram5 = radon.forward(x2_5)
          sinogram6 = radon.forward(x2_6)
          sinogram7 = radon.forward(x2_7)

          x3_1 = model(radon.backprojection(radon.filter_sinogram(sinogram1)))
          x3_2 = model(radon.backprojection(radon.filter_sinogram(sinogram2)))
          x3_3 = model(radon.backprojection(radon.filter_sinogram(sinogram3)))
          x3_4 = model(radon.backprojection(radon.filter_sinogram(sinogram4)))
          x3_5 = model(radon.backprojection(radon.filter_sinogram(sinogram5)))
          x3_6 = model(radon.backprojection(radon.filter_sinogram(sinogram6)))
```

```
            x3_7 = model(radon.backprojection(radon.filter_sinogram(sinogram7)))


      loss_trans=(loss_fn(x3_1,x2_1)+loss_fn(x3_2,x2_2)+loss_fn(x3_3,x2_3)+loss_fn(x3_4,x2_4)+lo
            # sinogram = radon.forward(x2)
            # x3 = model(radon.backprojection(radon.filter_sinogram(sinogram)))
            # loss_trans=loss_fn(x3,x2)
            loss = loss_fn(radon.forward(x1), y)+alpha*loss_trans
            train_loss.append(loss.item())
            train_psnr.append(PSNR_cal(X, x1,torch.max(x1)))
            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

          if trans == 10 :

            angles=sample(list(rotate_angles),10)
            x2_1 = rotate(x1, torch.Tensor([angles[0]]).cuda())
            x2_2 = rotate(x1, torch.Tensor([angles[1]]).cuda())
            x2_3 = rotate(x1, torch.Tensor([angles[2]]).cuda())
            x2_4 = rotate(x1, torch.Tensor([angles[3]]).cuda())
            x2_5 = rotate(x1, torch.Tensor([angles[4]]).cuda())
            x2_6 = rotate(x1, torch.Tensor([angles[5]]).cuda())
            x2_7 = rotate(x1, torch.Tensor([angles[6]]).cuda())
            x2_8 = rotate(x1, torch.Tensor([angles[7]]).cuda())
            x2_9 = rotate(x1, torch.Tensor([angles[8]]).cuda())
            x2_10 = rotate(x1, torch.Tensor([angles[9]]).cuda())
            # x2 = torch.stack([x2_1,x2_2,x2_3,x2_4,x2_5],dim=0)
            sinogram1 = radon.forward(x2_1)
            sinogram2 = radon.forward(x2_2)
            sinogram3 = radon.forward(x2_3)
            sinogram4 = radon.forward(x2_4)
            sinogram5 = radon.forward(x2_5)
            sinogram6 = radon.forward(x2_6)
            sinogram7 = radon.forward(x2_7)
            sinogram8 = radon.forward(x2_8)
            sinogram9 = radon.forward(x2_9)
            sinogram10 = radon.forward(x2_10)
            x3_1 = model(radon.backprojection(radon.filter_sinogram(sinogram1)))
            x3_2 = model(radon.backprojection(radon.filter_sinogram(sinogram2)))
            x3_3 = model(radon.backprojection(radon.filter_sinogram(sinogram3)))
            x3_4 = model(radon.backprojection(radon.filter_sinogram(sinogram4)))
            x3_5 = model(radon.backprojection(radon.filter_sinogram(sinogram5)))
            x3_6 = model(radon.backprojection(radon.filter_sinogram(sinogram6)))
            x3_7 = model(radon.backprojection(radon.filter_sinogram(sinogram7)))
            x3_8 = model(radon.backprojection(radon.filter_sinogram(sinogram8)))
            x3_9 = model(radon.backprojection(radon.filter_sinogram(sinogram9)))
            x3_10 =
    model(radon.backprojection(radon.filter_sinogram(sinogram10)))

    loss_trans=(loss_fn(x3_1,x2_1)+loss_fn(x3_2,x2_2)+loss_fn(x3_3,x2_3)+loss_fn(x3_4,x2_4)+lo
            # sinogram = radon.forward(x2)
            # x3 = model(radon.backprojection(radon.filter_sinogram(sinogram)))
            # loss_trans=loss_fn(x3,x2)
            loss = loss_fn(radon.forward(x1), y)+alpha*loss_trans
            train_loss.append(loss.item())
            train_psnr.append(PSNR_cal(X, x1,torch.max(x1)))
            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()


    print(f"Train : \n  Avg loss: {np.mean(train_loss):>8f} \n")
    print(f"  Avg psnr: {np.mean(train_psnr):>8f} \n")
    return np.mean(train_psnr),np.mean(train_loss)


def Supervised_train_loop(dataloader, model, loss_fn, optimizer, radon):
    model.train()
    train_loss, train_psnr=[], []
    for batch,X in enumerate(dataloader):
        X = X.cuda()
        y = radon.forward(X)
        filtered_sinogram = radon.filter_sinogram(y)
        fbpy = radon.backprojection(filtered_sinogram)  # calculate the FBP
        pred = model(fbpy.float())
        loss = loss_fn(pred, X)
        train_loss.append(loss.item())
        train_psnr.append(PSNR_cal(X, pred,torch.max(pred)))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Train : \n  Avg loss: {np.mean(train_loss):>8f} \n")
    print(f"  Avg psnr: {np.mean(train_psnr):>8f} \n")
    return np.mean(train_psnr),np.mean(train_loss)
```

```
def
    EI_train(train_dataloader,model,path,loss_fn,optimizer,radon,alpha,epochs,scheduler,trans)
    warnings.filterwarnings("ignore",category=DeprecationWarning)
    epoch_psnr_train,epoch_psnr_loss = 0,0
    lists = os.listdir(path)  # list all the files and store to lists
    if lists:                                    # if there is a check points,
    start training from the newest check point
      lists.sort(key=lambda fn: os.path.getmtime(path + "/" + fn))  # arrange
    by time
      file_new = os.path.join(path, lists[-1])                      # get the
    newest file
      ckpt = torch.load(file_new)
      model.load_state_dict(ckpt['net_state_dict'])
      epoch = ckpt['epoch']
      for e in range(epoch,epochs):
        print(f"Epoch {e+1}\n-------------------------------")
        epoch_psnr_train,epoch_psnr_loss = EI_train_loop(train_dataloader,
    model, loss_fn, optimizer,radon,alpha,trans)
        scheduler.step()
        if path is not None and (e + 1) % 100 == 0 :
                model.eval()
                ckpt = {
                    'epoch': e + 1,
                    'epoch_psnr_train': epoch_psnr_train,
                    'epoch_psnr_loss' : epoch_psnr_loss,
                    'net_state_dict': model.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                }
                torch.save(ckpt, os.path.join(path,
    'ckp_epoch_{}.pt'.format(e+1)))
    else:                                       # if there is no check point,
    start from 0
      for e in range(epochs):
        print(f"Epoch {e+1}\n-------------------------------")
        epoch_psnr_train,epoch_psnr_loss = EI_train_loop(train_dataloader,
    model, loss_fn, optimizer,radon,alpha,trans)
        scheduler.step()
        if path is not None and (e + 1) % 100 == 0 :
                model.eval()
                ckpt = {
                    'epoch': e + 1,
                    'epoch_psnr_train': epoch_psnr_train,
                    'epoch_psnr_loss' : epoch_psnr_loss,
                    'net_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
            }
            torch.save(ckpt, os.path.join(path,
'ckp_epoch_{}.pt'.format(e+1)))
    print("\nTraining is Done.\tTrained model saved at {}".format(path))


def
    Supervised_train(train_dataloader,model,path,loss_fn,optimizer,epochs,scheduler,radon):
    epoch_psnr_train,epoch_psnr_loss = 0,0
    lists = os.listdir(path)          # list all the files and store to lists
    if lists:                                    # if there is a check
points, start training from the newest check point
      lists.sort(key=lambda fn: os.path.getmtime(path + "/" + fn))   # arrange
by time
      file_new = os.path.join(path, lists[-1])                   # get the
newest file
      ckpt = torch.load(file_new)
      model.load_state_dict(ckpt['net_state_dict'])
      epoch = ckpt['epoch']
      for e in range(epoch,epochs):
        print(f"Epoch {e+1}\n-------------------------------")
        epoch_psnr_train,epoch_psnr_loss =
Supervised_train_loop(train_dataloader, model, loss_fn, optimizer,radon)
        scheduler.step()
        if path is not None and (e + 1) % 100 == 0 :
            model.eval()
            ckpt = {
                'epoch': e + 1,
                'epoch_psnr_train': epoch_psnr_train,
                'epoch_psnr_loss' : epoch_psnr_loss,
                'net_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
            }
            torch.save(ckpt, os.path.join(path,
'ckp_epoch_{}.pt'.format(e+1)))
    else:                                        # if there is no check
point, start from 0
      for e in range(epochs):
        print(f"Epoch {e+1}\n-------------------------------")
        epoch_psnr_train,epoch_psnr_loss =
Supervised_train_loop(train_dataloader, model, loss_fn, optimizer,radon)
        scheduler.step()
        if path is not None and (e + 1) % 100 == 0 :
            model.eval()
```

```
                ckpt = {
                    'epoch': e + 1,
                    'epoch_psnr_train': epoch_psnr_train,
                    'epoch_psnr_loss' : epoch_psnr_loss,
                    'net_state_dict': model.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                }
                torch.save(ckpt, os.path.join(path,
    'ckp_epoch_{}.pt'.format(e+1)))
    print("\nTraining is Done.\tTrained model saved at {}".format(path))
```

Test.py

```
# -*- coding: utf-8 -*-
"""test.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1RrDMlR9vJiv6tlJjP-tBRWuFgZfHAyRG
"""

import torch
import numpy as np
import matplotlib.pyplot as plt
from utils import PSNR_cal,show_images
from network_arch import UNet
import os

def Supervised_test_loop(dataloader, model, loss_fn,radon):
    model.eval()
    test_loss, test_psnr=[], []
    with torch.no_grad():
        for X in dataloader:
            X = X.cuda()
            y = radon.forward(X)
            filtered_sinogram = radon.filter_sinogram(y)
            fbpy = radon.backprojection(filtered_sinogram)
            pred = model(fbpy.float())
            test_loss.append(loss_fn(pred, X).item())
            test_psnr.append(PSNR_cal(X, pred,torch.max(pred)))
    print(f"Test : \n  Avg loss: {np.mean(test_loss):>8f} \n")
    print(f"  Avg psnr: {np.mean(test_psnr):>8f} \n")
    return np.mean(test_psnr)
```

```
def EI_test_loop(dataloader, model, loss_fn,radon):
    #test_loss, correct = 0, 0
    model.eval()
    test_loss, test_psnr=[], []
    #i=0
    with torch.no_grad():
        for X in dataloader:
            X = X.cuda()
            y = radon.forward(X)
            filtered_sinogram = radon.filter_sinogram(y)
            x1 = model(radon.backprojection(filtered_sinogram))
            test_loss.append(loss_fn(x1, X).item())
            test_psnr.append(PSNR_cal(X, x1,torch.max(x1)))


    print(f"Test : \n  Avg loss: {np.mean(test_loss):>8f} \n")
    print(f"  Avg psnr: {np.mean(test_psnr):>8f} \n")
    return np.mean(test_psnr)

def EI_test(dataloader, path, loss_fn,radon,epochs):
    psnrs_train = []
    psnrs_test = []
    psnrs_epoch = []
    loss_train = []
    checkpoint_model_dir=path
    for e in range(epochs):
        if checkpoint_model_dir is not None and (e + 1) % 100 == 0 :
                print(f"Epoch {e+1}\n-------------------------------")
                path=os.path.join(checkpoint_model_dir,
    'ckp_epoch_{}.pt'.format(e+1))
                ckpt = torch.load(path)
                test_model=UNet().cuda()
                test_model.load_state_dict(ckpt['net_state_dict'])
                psnrs_train.append(ckpt['epoch_psnr_train'])
                loss_train.append(ckpt['epoch_psnr_loss'])
                psnrs_epoch.append(ckpt['epoch'])
                psnrs_test.append(EI_test_loop(dataloader, test_model,
    loss_fn,radon))
    print("Done!")
    return psnrs_train,psnrs_test

def Supervised_test(dataloader, path, loss_fn,epochs,radon):
    psnrs_train = []
    psnrs_test = []
    psnrs_epoch = []
    loss_train = []
    checkpoint_model_dir=path
    for e in range(epochs):
        if checkpoint_model_dir is not None and (e + 1) % 100 == 0 :
                print(f"Epoch {e+1}\n-------------------------------")
                path=os.path.join(checkpoint_model_dir,
    'ckp_epoch_{}.pt'.format(e+1))
                ckpt = torch.load(path)
                test_model=UNet().cuda()
                test_model.load_state_dict(ckpt['net_state_dict'])
                psnrs_train.append(ckpt['epoch_psnr_train'])
                loss_train.append(ckpt['epoch_psnr_loss'])
                psnrs_epoch.append(ckpt['epoch'])
                psnrs_test.append(Supervised_test_loop(dataloader, test_model,
    loss_fn,radon))
    print("Done!")
    return psnrs_train,psnrs_test
```

Main.py

```
# -*- coding: utf-8 -*-
"""Main.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/14qs8IklLQbVDJ9VOm4miDyhFmZ-MdTbE
"""

import scipy.io as io
import os
import torch
import numpy as np
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms
import torch.nn as nn
from torch.autograd import Variable
import matplotlib.pyplot as plt
from torch_radon import Radon, RadonFanbeam
from kornia import rotate
from random import choice
from network_arch import UNet
from data import dataloader_train,dataloader_test
from utils import PSNR_cal,show_images
from train import EI_train,Supervised_train
from test import EI_test,Supervised_test

epochs=5000
batch_size=2
learning_rate=0.0005
weight_dec=1e-8
alpha=0.1
image_size=128
n_views=50
det_count = int(np.sqrt(2)*image_size + 0.5)
angles = np.linspace(0, np.pi, n_views, endpoint=False)
radon = Radon(image_size, angles, clip_to_circle=False, det_count=det_count)
loss_fn = nn.MSELoss().cuda()
Supervised_model = UNet().cuda()
EI_model = UNet().cuda()
optimizer_Sup = torch.optim.Adam(Supervised_model.parameters(),
    lr=learning_rate, weight_decay=weight_dec)
optimizer_EI = torch.optim.Adam(EI_model.parameters(), lr=learning_rate,
    weight_decay=weight_dec)
scheduler_Sup = torch.optim.lr_scheduler.MultiStepLR(optimizer_Sup,
    milestones=[2000,3000,4000], gamma=0.2)
scheduler_EI = torch.optim.lr_scheduler.MultiStepLR(optimizer_EI,
    milestones=[2000,3000,4000], gamma=0.2)
Sup_psnrs_train,Sup_psnrs_test= [],[]
EI_psnrs_train,EI_psnrs_test= [],[]
gt_path =
    ('/content/drive/MyDrive/Final_year_project/src/Dataset/Groud_Truth_train.mat')
gt_path_test =
    ('/content/drive/MyDrive/Final_year_project/src/Dataset/Groud_Truth_test.mat')

dataloader_train,dataset_train=dataloader_train(gt_path,batch_size)
dataloader_test,dataset_test=dataloader_test(gt_path_test,batch_size)

EI_Model_path =
    '/content/drive/MyDrive/Final_year_project/Trained_model/EI_model/50-views'
Supervised_Model_path =
    '/content/drive/MyDrive/Final_year_project/Trained_model/Supervised_model/50-views'

Supervised_train(dataloader_train,Supervised_model,Supervised_Model_path,loss_fn,optimizer_Su
EI_train(dataloader_train,EI_model,EI_Model_path,loss_fn,optimizer_EI,radon,alpha,epochs,sche
Sup_psnrs_train,Sup_psnrs_test = Supervised_test(dataloader_test,
    Supervised_Model_path, loss_fn,epochs,radon)
EI_psnrs_train,EI_psnrs_test = EI_test(dataloader_test, EI_Model_path,
    loss_fn,radon,epochs)

plt.figure(figsize=(10,4))
plt.plot(Sup_psnrs_train,label="Sup_psnrs_train")
plt.plot(Sup_psnrs_test,label="Sup_psnrs_test")
plt.plot(EI_psnrs_train,label="EI_psnrs_train")
plt.plot(EI_psnrs_test,label="EI_psnrs_test")
plt.xlabel('Epoch')
# Set the y axis label of the current axis.
plt.ylabel('PSNR(dB)')
# Set a title of the current axes.
#plt.title('Two or more lines on same plot with suitable legends ')
# show a legend on the plot
plt.legend()
# Display a figure.
plt.savefig('/content/drive/MyDrive/Final_year_project/Figures/50_views.png')
plt.show()
```

Bibliography

[1] G. Ongie, A. Jalal, C. A. Metzler, R. G. Baraniuk, A. G. Dimakis, and R. Willett, "Deep learning techniques for inverse problems in imaging," *INFORMATION THEORY*, vol. 1, no. 1, 2020.

[2] E. Y. Sidky and X. Pan, "Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization," vol. 53, no. 17, p. 4777, 2008.

[3] M. G. McGaffin and J. A. Fessler, "Alternating dual updates algorithm for x-ray ct reconstruction on the gpu," vol. 1, no. 3, p. 86–199, 2015.

[4] M. Lustig, D. Donoho, and J. M. Pauly, "parse mri: The application of compressed sensing for rapid mr imaging," vol. 58, no. 6, pp. 1182–1195, 2007.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," vol. 521, pp. 436–444, May 2015.

[6] C. A. Metzler, A. Mousavi, R. Heckel, and R. G. Baraniuk, "Unsupervised learning with stein's unbiased risk estimator," 2018.

[7] D. Chen, J. Tachella, and M. E. Davies, "Equivariant imaging: Learning beyond the range space," 2021.

[8] J.-L. Starck and M. J. Fadili, "An overview of inverse problem regularization using sparsity," in *2009 16th IEEE International Conference on Image Processing (ICIP)*, pp. 1453–1456, 2009.

[9] A. N. Tikhonov, "On the stability of inverse problems," vol. 39, no. 5, pp. 195–198, 1943.

[10] Y. Wang, J. Yang, W. Yin, and Y. Zhang, "A new alternating minimization algorithm for total variation image reconstruction," vol. 1, no. 3, pp. 248–272, 2008.

[11] R. C. Gonzalez and R. E. Woods, "Image processing," vol. 2, Mar 2007.

[12] E. Caroli, J. Stephen, G. D. Cocco, L. Natalucci, and A. Spizzichino, "Coded aperture imaging in x-and gamma-ray astronomy," vol. 45, p. 349–403, 1987.

[13] S. Farsiu, M. D. Robinson, M. Elad, and P. Milanfar, "Fast and robust multiframe super resolution," vol. 13, pp. 1327–1344, Oct 2004.

[14] M.Bertalmio, G.Sapiro, V.Caselles, and C.Ballester, "Image inpainting," pp. 417–424, 2000.

[15] E. J. Candes, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," vol. 59, no. 8, pp. 1207–1223, 2006.

[16] I.Elbakri and J.Fessler, "Statistical image reconstruction for polyenergetic x-ray computed tomography," vol. 21, pp. 89–99, Feb 2002.

[17] S. J. Wright, R. D. Nowak, and M. A. T. Figueiredo, "Sparse reconstruction by separable approximation," vol. 57, pp. 2479–2493, Jul 2009.

[18] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," vol. 1, pp. 234–241, May 2015.

[19] S. Diamond, V. Sitzmann, F. Heide, and G. Wetzstein, "Unrolled optimization with deep priors," 2017.

[20] T. Meinhardt, M. Moeller, C. Hazirbas, and D. Cremers, "Learning proximal operators: Using denoising networks for regularizing inverse imaging problems," pp. 1799–1808, 2017.

[21] A. Bora, A. Jalal, E. Price, and A. G. Dimakis, "Compressed sensing using generative models," *Proc. Int. Conf. Mach. Learn.*, p. 537–546, 2017.

[22] J. R. Chang, C.-L. Li, B. Poczos, B. V. Kumar, and A. C. Sankaranarayanan, "One network to solve them all—solving linear inverse problems using deep projection models," *Proc. IEEE Int. Conf. Comput. Vis.*, p. 5888–5897, 2017.

[23] D. Engin, A. Genç, and H. K. Ekenel, "Cycle-dehaze: Enhanced cyclegan for single image dehazing," p. 825–833, 2018.

[24] O. Kupyn, V. Budzan, M. Mykhailych, D. Mishkin, and J. Matas, "Deblurgan: Blind motion deblurring using conditional adversarial networks," *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, p. 8183–8192, 2018.

[25] A. Bora, E. Price, and A. G. Dimakis, "Ambientgan: Generative models from lossy measurements," *Proc. ICLR*, vol. 2, p. 5, 2018.

[26] B. Zhu, J. Z. Liu, S. F. Cauley, B. R. Rosen, and M. S. Rosen, “Image reconstruction by domain-transform manifold learning,” vol. 555, no. 7697, p. 487–492, 2018.

[27] M. T. McCann, K. H. Jin, and M. Unser, “Convolutional neural networks for inverse problems in imaging: A review,” *EEE Signal Process. Mag.*, vol. 34, p. 85–95, Nov 2017.

[28] C. Dong, C. C. Loy, K. He, , and X. Tang, “Image super-resolution using deep convolutional networks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, p. 295–307, Feb 2016.

[29] J. Kim, J. K. Lee, and K. M. Lee, “Accurate image super-resolution using very deep convolutional networks,” *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, p. 1646–1654, Jun 2016.

[30] K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, “Deep convolutional neural network for inverse problems in imaging,” vol. 26, pp. 4509–4522, Sep 2017.

[31] K. Gregor and Y. LeCun, “Learning fast approximations of sparse coding,” pp. 399–406, 2010.

[32] K. Gregor and Y. LeCun, “Learning fast approximations of sparse coding,” *Proc. Int. Conf. Mach. Learn.*, p. 399–406, 2010.

[33] J. I. Tamir, X. Y. Stella, and M. Lustig, “Unsupervised deep basis pur- suit: Learning reconstruction without ground-truth data,” *Proc. 27th Annu. Meeting ISMRM*, p. 4, 2019.

[34] Y. C. Eldar, “Generalized sure for exponential families: Applications to regularization,” *IEEE Transactions on Signal Processing*, vol. 57, no. 2, p. 471–481, 2009.

[35] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, “Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising,” *IEEE Trans. Image Process*, vol. 26, p. 3142–3155, July 2017.

[36] M. Zhussip, S. Soltanayev, and S. Y. Chun, “Training deep learn- ing based image denoisers from undersampled measurements without ground truth and without image prior,” *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, p. 10255–10264, 2019.

[37] T. Buzug, “Computed tomography,” *Berlin, Heidelberg: Springer Berlin Heidelberg*, 2008.

[38] H. Xie, H. Shan, W. Cong, C. Liu, X. Zhang, S. Liu, R. Ning, and G. Wang, “Deep efficient end-to-end reconstruction (deer) network for few-view breast ct image reconstruction,” *IEEE Access*, vol. 8, pp. 196633–196646, 2020.

[39] R. Gordon, R. Bender, and G. T. Herman, "Algebraic reconstruction techniques (art) for three-dimensional electron microscopy and x-ray photography," vol. 29, p. 471–481, Dec 1970.

[40] A.P.Dempster, N.M.Laird, and D.B.Rubin, "Maximum likelihood from incomplete data via the em algorithm," vol. 39, no. 1, pp. 1–38, 1977.

[41] A.Andersen, "Simultaneous algebraic reconstruction technique (sart): A superior implementation of the art algorithm," vol. 6, pp. 81–94, Jan 1984.

[42] B. Zhu, J. Z. Liu, S. F. Cauley, B. R. Rosen, and M. S. Rosen, "Image reconstruction by domain-transform manifold learning," vol. 555, pp. 487–492, Mar 2018.

[43] T. Wurfl, M. Hoffmann, V. Christlein, K. Breininger, Y. Huang, M. Unberath, and A. K. Maier, "Deep learning computed tomography: Learning projection-domain weights from image domain in limited angle problems," *IEEE Trans. Med. Imag.*, vol. 37, p. 1454–1463, Jun 2018.

[44] H. Chen, Y. Zhang, Y. Chen, J. Zhang, W. Zhang, H. Sun, Y. Lv, P. Liao, J. Zhou, and G. Wang, "Learn: Learned experts' assessment- based reconstruction network for sparse-data ct," *IEEE Trans. Med. Imag.*, vol. 37, p. 1333–1347, Jun 2018.

[45] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, p. 115–133, 1943.

[46] F. Rosenblatt, "The perceptron, a perceiving and recognizing automaton project para," *Cornell Aeronautical Laboratory*, 1957.

[47] M. Minsky and S. Papert, "Perceptrons," 1969.

[48] Phung and Rhee, "A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets," *Applied Sciences*, vol. 9, p. 4500, 10 2019.

[49] E. K. Wang, C.-M. Chen, M. M. Hassan, and A. Almogren, "A deep learning based medical image segmentation technique in internet-of-medical-things domain," *Future Generation Computer Systems*, vol. 108, pp. 135–144, 2020.

[50] M. Ronchetti, "Torchradon: Fast differentiable routines for computed tomography," *arXiv preprint arXiv:2009.14788*, 2020.

[51] K. Clark, B. Vendt, K. Smith, J. Freymann, J. Kirby, P. Koppel, S. Moore, S. Phillips, D. Maffitt, and M. P. et al., "The cancer imaging archive (tcia): maintaining and operating a public information repository," *Journal of digital imaging*, vol. 26, no. 6, p. 1045–1057, 2013.

[52] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.