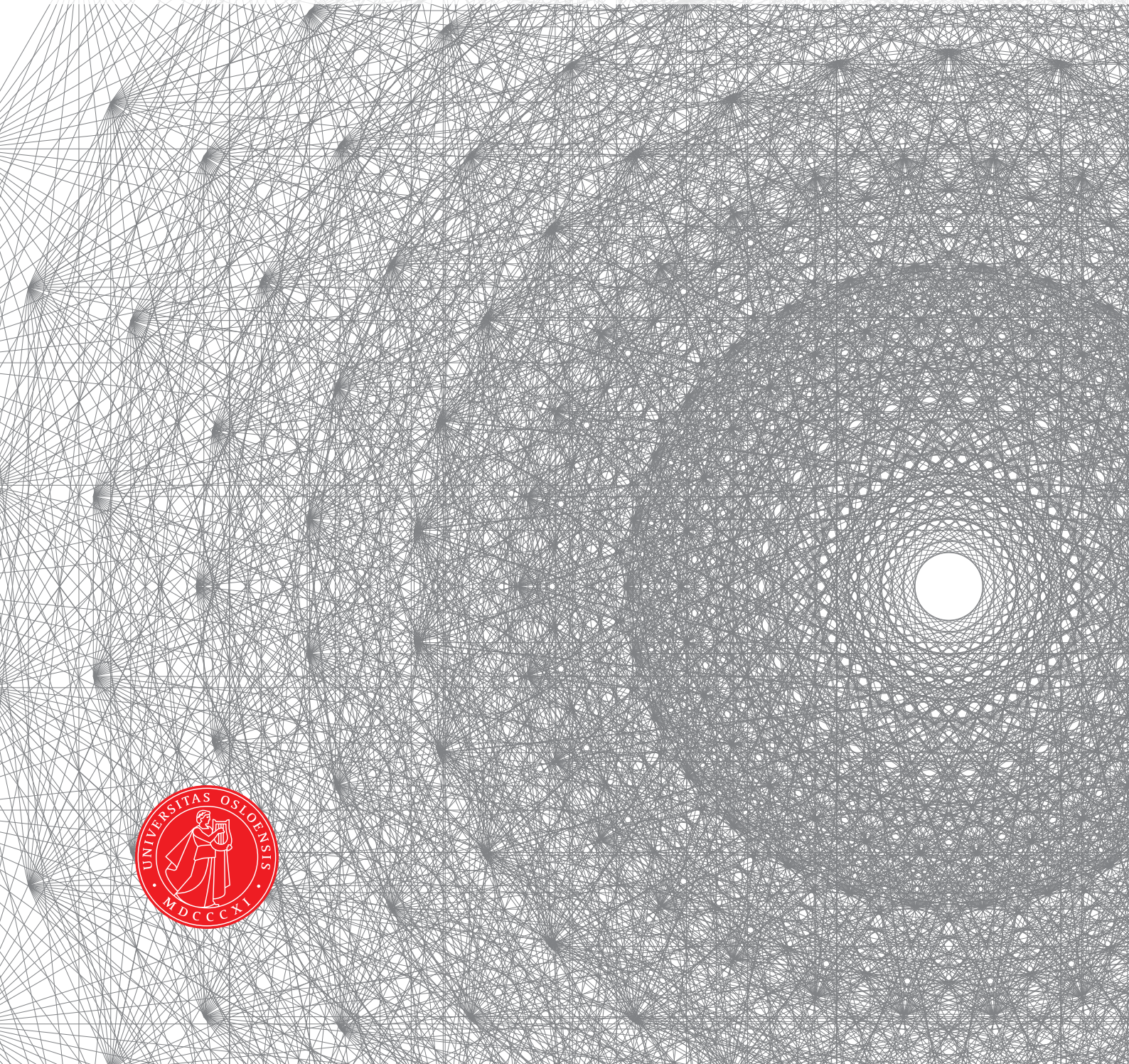
UiO : Department of Mathematics
University of Oslo

Parseval Reconstruction Networks

Improving Robustness of Deep Learning Based MRI Reconstruction Towards Adversarial Attacks

Mathias Lohne

Master's Thesis, Spring 2019



This master's thesis is submitted under the master's program *Computational Science and Engineering*, with program option *Computational Science*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

Recovering signals from undersampled measurements is a well-studied topic in mathematics. During the last decade, many attempts have been made to solve this problem using machine learning, with resulting reconstruction models that report remarkable performance. However, recent work have revealed major systematic stability issues with these models, such as the instability towards adversarial noise. That is, given an image which a neural network can recover correctly, we can easily create a tiny perturbation so that the perturbed image produces severe artifacts during recovery.

Similar phenomena are well-established for classification networks, and subsequently several regularization methods for reducing the instabilities of classification networks have been proposed. In this thesis we investigate Parseval networks, in which the every layer is constrained to be a contraction, thus limiting how much a perturbation can be amplified through the network. We adapt these techniques to image reconstruction networks and show that while we seem to sacrifice some performance, the resulting networks do not exhibit the same instabilities.

Acknowledgments

First of all, I would like to thank my two supervisors: Øyvind Ryan for introducing me to Compressive Sensing and being an actual superhero during the writing process – your regular critique of my drafts have been extremely helpful. And Anders C. Hansen for dragging me straight to the forefront of current research and pushing me to strive for good results. It has been both fun and motivating to get to work on current, unsolved issues, and your input have been much appreciated. I also need to give Vegard Antun a special mention. Even though you have your own PhD to worry about, you have taken the time to be like a third supervisor to me, in addition to being a good colleague and an overall nice human, and I'm extremely thankful.

I've been very lucky to get to know so many nice people during my stay at the University of Oslo, and naming everyone would simply take up too much space, but I want to particularly mention Kristian Monsen Haug and Vegard Stikbakke. I'm so glad I got to meet you early on during our studies, and that I've always had friends with me at every course I've ever taken at the University. Without your help and support I would not have gotten to where I am now. I would also like to thank Keith Zhou for all the conversations we've had in the study hall and for help with proofreading, and Eirik Ramsli Hauge for helping me to wrap my head around MR imaging. Turns out knowing a physicist can sometimes be quite helpful, especially when my mathematically inclined mind had to deal with the nasty real world for a moment.

Thanks to everyone at Studentorchesteret Biørneblæs for being so inclusive and giving me a network of friends at campus, for reminding me to take breaks and have fun, and generally being an awesome bunch of amazing people. Also, thanks for letting me boss you around during the last year. It's been fun.

Finally, a huge thanks to my wife, Lily. Thank you for supporting me, for listening to me go on in unnecessary detail about some obscure part of my thesis, for celebrating with me when results were good and for dragging me out of bed when motivations were low. You're literally the best.

Mathias Lohne
Oslo, May 2019

Contents

List of Symbols

$\mathbf{F}_n$	-	$n \times n$ Fourier matrix
Ω	-	Sampling pattern
P_Ω	-	Sampling operator using Ω as sampling pattern
$\mathbf{\Psi}$	-	Discrete Wavelet Transform operator
$\mathbf{A}$	-	Measurement operator
F	-	True model in supervised learning
θ	-	Set of all model parameters
$\hat{F}$	-	Estimated model in supervised learning
$\hat{F}_\theta$	-	Estimated model in supervised learning using θ as parameters
Φ	-	A neural network
Φ_θ	-	A neural network using θ as parameters
W_l	-	Layer l of a neural network, ie an affine mapping
$\mathbf{W}_l$	-	Weight matrix of layer l
$\mathbf{b}_l$	-	Bias vector of layer l
ρ_l	-	Activation function at layer l
$\boldsymbol{\rho}_l$	-	ρ_l acting element-wise on vectors
ReLU	-	Rectified Linear Unit
LReLU_α	-	Leaky Rectified Linear Unit with slope α
tanh	-	Hyperbolic tangent
σ	-	Sigmoid function
$\mathbf{a}_l$	-	Activations at layer l
N_l	-	Output dimension of layer l
L	-	Number of layers in a neural network
η	-	Step size / Learning rate
$\mathcal{L}$	-	Loss function
$\mathbf{U}(\cdot)$	-	Unfolding operator

List of Acronyms

Adam Adaptive Moments Optimizer. 26

CNN Convolutional Neural Network. 31

CS Compressive Sensing. 7

DC Data Consistency layer. 31

DL Deep Learning. 19

DWT Discrete Wavelet Transform. 13

GAN Generative Adversarial Network. 33

MRI Magnetic Resonance Imaging. 1

MSE Mean Squared Error. 23

NN Neural Network. 18

NSP Null Space Property. 10

ReLU Rectified Linear Unit. 19, 21

SGD Stochastic Gradient Descent. 26

UAT Universal Approximation Theorem. 20, 21

Provided Code

The main result of this thesis is the proposal of Parseval regularizers for image reconstruction networks. To make this method as accessible as possible, we have released the code necessary for anyone to include these techniques into their own work as an open source Python package, available through the Python Package Index. To install it on your own machine, the following `pip` command should suffice:

```
$ pip install parsnet
```

The `parsnet` package should now be available on your system. Depending on your Python installation, you might have to run the above command as root, or with the `--user` flag. This package introduces the `tight_frame` class which implements the methods derived in Chapter 5 as a plug-and-play extension to TensorFlow. Details on implementation, licensing and how to include the proposed method in your own work are found in Appendix B.1.

In addition we have released the total body of code as a separate git repository, consisting of all the scripts necessary to reproduce any of our presented results. This is available at the authors GitHub page[1].

[1] `https://github.com/mathialo/master_code`

CHAPTER 1

Introduction

In Magnetic Resonance Imaging (MRI), signal acquisition time often pose a problem. This is both expensive for the hospital, and uncomfortable for the patients as one have to lie still for the whole procedure.

In short, MRI works by subjecting a patient to a strong magnetic field. This forces hydrogen protons to align either with or against the direction of the magnetic field, which is usually the same as the z-axis. However, due to angular momentum, the magnetic moment of each each proton will rotate *around* the z-axis instead of aligning perfectly. Hence, the proton's magnetic moment will rotate at individual phases, causing the net magnetization of all protons to be along the z-axis as all other directions will cancel out. The MRI machine then sends a pulse of Radio Frequencies (RF) which cause the hydrogen protons magnetic moments to synchronize their phases. This in turn causes a component of the net magnetization vector in the xy-plane which can be detected by the MRI machine as an RF wave. Once the RF pulse stops, the individual magnetic moments will de-phase, causing the net magnetization vector to once again be along the z-axis. Different types of tissue will de-phase differently, and an MRI machine detects these RF frequencies [Flo12, Sec. 7.6].

Hence, an MRI machine does not sample image pixels directly, but rather as frequencies, which from a mathematical perspective is the same as saying that an MRI machine measures the signal in the 2D Fourier basis, instead of the standard basis. An inverse Fourier transform is performed on the measured frequencies, yielding the resulting image.

Traditional signal processing gives a lower bound on the number of MRI measurements required through Shannon and Nyquist's sampling theorem. However, many attempts have been made to cheat this bound, some of the most notable methods are Compressive Sensing (CS) and Deep Learning (DL).

Introduced in [CT06; CRT06; Don06], Compressive Sensing places some additional assumptions on the signal we are recovering, and can make do with way fewer measurements than traditional signal processing. We will discuss CS in further detail in Section 2.1, but in short CS makes fewer measurements, and solves a costly optimization problem to recover the signal. Hence, we can trade off time spent by the patient in an MR scanner, with computation time afterwards.

This is a huge improvement for both the patient and the hospital, but the long computation times in modern CS have led to searches for other approaches to undersampled signal recovery. With the explosion in popularity around

Machine Learning (ML) in the last years, another approach have emerged, namely using Deep Learning Denoising.

In this approach, we first do a very quick and crude recovery of the image using the adjoint of the measurement operator as an estimate for the inverse. This reconstruction will be very fast, but will leave many artifacts on the resulting image due to aliasing from the severe undersampling (as shown in Figures 2.1 and 2.3). These artifacts can be then removed using a Deep Learning Denoiser.

Several researchers have found apparent success with this approach, and report remarkable reconstruction capabilities [Sch+18; Yan+18; MJU17]. These recovery algorithms will typically use several days to *train*, but once they are trained they can recover images from measurements in mere seconds.

However, recent work have uncovered a systematic flaw in these recovery methods, namely stability issues [Ant+19]. The authors discuss three different kinds of instabilities in modern state-of-the-art ML-based recovery schemes:

1. Instability to adversarial attacks
2. Inability to recover unexpected details successfully
3. Instability to sampling rate and patterns

So-called Adversarial attacks regard finding a perturbation $\boldsymbol{\delta}$ for an image $\mathbf{x}$ such that the recovery of $\mathbf{x}$ works fine while the recovery of $\mathbf{x} + \boldsymbol{\delta}$ leads to severe recovery errors, even though $\mathbf{x}$ and $\mathbf{x} + \boldsymbol{\delta}$ might be nearly indistinguishable. This is a known flaw of Deep Learning-based classifiers, where a small change in the input image can cause the classifier to misclassify the image in often very unexpected ways [FMF17; MFF16].

The inability to recover unexpected details may not be very surprising to readers with a statistical background, but is still a very important form of instability. The authors of [Ant+19] performed experiments were the text "can u see it ♤" were superimposed on real MR images and fed through different ML-based recovering schemes, and in most cases the text came out unreadable. As these algorithms have never seen such text before, it is not very surprising that the recovery does not work well[1]. However, these experiments demonstrate an important fact: The main reason to do an MRI scan is to check for abnormalities. If some unexpected detail, such as a tumor, can cause the reconstruction to fail, this should be taken very seriously.

One can of course argue that this instability can easily be fixed by making sure that any kind of abnormality is presented to the algorithm during training. However, this will not fix the underlying problem that previously unseen details can cause instability in the reconstruction.

The instability towards the choice of sampling pattern (i.e. how the signal is sampled) is somewhat expected, as different sampling patterns produce very different artifacts (see Figure 2.3). The instability towards sampling rate, however, may seem more unintuitive. In some cases, such as for [Yan+18], the authors of [Ant+19] report that an increase in sampling rate results in a decrease of reconstruction capability.

[1] The nature of all machine learning is to extrapolate patterns from examples, and when none of your given examples contain a feature it is not surprising that this feature is not handled correctly

State-of-the-art Compressive Sensing does not exhibit these same instabilities, as we have clear bounds on how large an error can amplify during the reconstruction (see the *Further reading* section at page 12). The authors of [Ant+19] also provided examples of reconstruction of perturbed inputs using modern Compressive Sensing techniques to illustrate this fact[2].

In this thesis we will focus mainly on the instability towards adversarial attacks.

On Notation and Terminology

During the exploding interest of Machine Learning the last decade, the community have suffered from some growing pains. As there have been a huge push to publish new methods, in many cases being the first is more important than making sure all the details are correct. Further, many central, important and widely cited works are not published in traditional peer-reviewed journals, but as conference papers, or even merely uploaded to an online preprint archive such as arXiv.

This have led to some important problems with the Machine Learning literature [LS18], especially to readers with a mathematical background. Most notably:

Misuse and overloading of terminology Several terms found in ML literature are used to mean different things than their original mathematical definition. For example, the use of the term *convolution* in Convolutional Neural Networks (see Remark 3.7 on page 22) is used to mean a correlation, and *deconvolution* is used to mean *transposed convolutions*, or to be precise, *transposed correlations* (see Footnote 11 on page 33).

Whenever such differences exist between the use of a term in ML literature, and the actual mathematical meaning of the term we will give a remark, and specify which of these we will continue to use.

In addition, notions with an already established term are given new names, such as *learning rate*, which is used to mean the *step length* in an optimization problem. Other terms can be used to mean two different things, such as the word *adversarial*, which can mean two networks trained together in competition (see the subsection for the DAGAN network on page 33), or the creation of malicious attacks against a neural network (see Section 4.2).

Lack of proofs and theoretical justification In many papers, the authors often omit theoretical results in favor of empirical results and showing examples.

In some cases, a proof is given, but the nature of the result and proof makes it of little to no use for practical Machine Learning. Some examples of this include the constant referring to the Universal Approximation Theorem (UAT) (Theorem 3.2), which does not cover modern neural networks (see Remarks 3.4 to 3.6 on page 21), is non-constructive, and merely states that neural networks are dense in $C(S)$ (the space of continuous functions on a compact subset

[2] Although it is worth noting that the perturbations they test against were constructed with the deep learning models in mind.

$S \subset \mathbb{R}^n$)[3]. Another example is the paper introducing the Adam optimizer [KB14], which gives a proof of convergence when the objective function is convex. However, for training neural networks, the objective function is rarely convex, thus the result does not actually prove that Adam will work well for training neural networks.

In other cases, techniques are used with no formal proofs or theoretical justification. Revisiting Adam, the use of momentum and rescaling in combination have no clear theoretical motivation [GBC16, Sec. 8.5.3], but seem to work well in practice.

Mathematical mistakes Some times, mathematical mistakes or oversights are included in published works. The most notable example is perhaps the previously mentioned proof in [KB14], which were later shown to be wrong in [RKK18].

Another example we found while working with this thesis is the oversight that two of the loss functions in [Yan+18] are identically equal, even though they are presented as different functions (see Remark 3.9 on page 35).

Our Contributions

In this thesis we will explore how recent work regarding the stability of deep learning classification against adversarial attacks [Cis+17] can be adapted and applied to Deep Learning Denoisers. We will introduce Parseval reconstruction networks as a proposed method to reduce the instabilities of MRI reconstruction networks with regards to adversarial attacks, and provide empirical experiments to test their effectiveness.

As [Cis+17] did not provide any code to reproduce and further develop their findings, we have reimplemented all the necessary functionality and released it as a free, easy-to-use software package for Python 3 and TensorFlow, licensed under the LGPLv3 license, and available at the author's GitHub page[4] or through the Python Package Index. See Appendix B.1 for details.

The total body of code used throughout the thesis to perform computations, experiments, generate figures and so on have been made available as a separate GitHub repository[5].

Thesis Outline

- In **Chapter 2** we will define reconstruction of MR Images as Inverse Problems, and briefly discuss how to solve them using the traditional Compressive Sensing theory.

- **Chapter 3** is an introduction to Neural Networks, and how modern research are using them to solve MRI reconstruction.

[3] Polynomials are also dense in $C(S)$, they are however not as frequently used in machine learning as neural networks. If the UAT were the sole reason Neural Networks are performing so well, we should in principle be able to achieve the same results using only polynomials.

[4] `https://github.com/mathialo/parsnet`

[5] `https://github.com/mathialo/master_code`

- In **Chapter 4** we will look at numerical stability. We will discuss weaknesses in the methods from Chapter 3, particularly how the addition of carefully picked noise can produce severe artifacts in the reconstruction.
- **Chapter 5** will propose Parseval reconstruction networks for recovering undersampled MR images, heavily influenced by recent work in classification networks [Cis+17]. We will constrain each layer in the Neural Network to be a *contraction*, limiting the amount an error can amplify.
- **Chapter 6** contains experimental validation of the methods outlined in Chapter 5.
- In **Chapter 7** we conclude, and point to future work.

CHAPTER 2

Undersampled Signals

Traditional signal processing follows Shannon and Nyquist's sampling theorem. It states that if the sampling frequency is at least twice the highest frequency present in a signal, the signal can be recovered perfectly from samples by interpolating with sinc functions. However, by changing the recovery method from sinc interpolation to other techniques, we can in some circumstances undersample the signal while still being able to fully recover it.

We begin our study of undersampled signals by formulating the problem as an *inverse problem*. Let $\mathbf{x} \in \mathbb{R}^n$ be some image[1], let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a matrix representing how measurements are taken, and let $\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{R}^m$ be measurements of $\mathbf{x}$. If $m = n$, there is a chance that the measurement operator $\mathbf{A}$ is invertible, and we could recover the original image as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$. However, when working with undersampled signals, we have that $m \ll n$, so $\mathbf{A}$ cannot possibly be invertible.

However, we would still like to recover $\mathbf{x}$ from $\mathbf{y}$. Solving the inverse problem amounts to constructing a mapping $\mathbf{B}\colon \mathbb{R}^m \to \mathbb{R}^n$ which estimates an inverse of $\mathbf{A}$, at least on a certain subset of $\mathbb{R}^m$.

One immediate idea for a solution is to use the adjoint of the measurement operator as an estimate for the inverse. This however gives back a rather noisy image, resulting from aliasing due to the severe undersampling. This effect is illustrated in Figure 2.1.

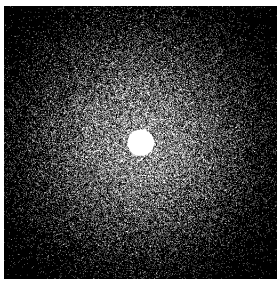
In the remainder of this chapter we will present Compressive Sensing (CS) as a general recovery scheme for undersampled signals, and briefly show how we can apply CS to MRI reconstruction. Later, in Section 3.4, we will look at how Deep Learning can be used to solve the same problem.

2.1 Compressive Sensing

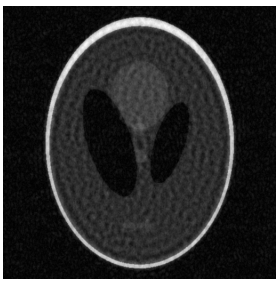
The field of Compressive Sensing has exploded in the last decade, after the initial publications by Candès, Tao, Romberg and Donoho [CT06; CRT06; Don06]. CS has proven itself to be a reliable way to recover undersampled signals, and in 2017 the United States Food and Drug Administration approved the use of CS in commercial MRI machines [FDA17; Sie17].

[1] Normally, one thinks of images as two-dimensional signals (ie, as matrices). However, for the time being, we will consider a vector where all the columns of the original image matrix have been stacked.

(a) The Shepp–Logan phantom, ie our input signal $\mathbf{x}$



(b) Sampling pattern Ω



(c) Naive reconstruction: $\tilde{\mathbf{x}} = \mathbf{A}^*\mathbf{y} = \mathbf{A}^*\mathbf{A}\mathbf{x}$

Figure 2.1: Reconstruction with the adjoint.

Before we introduce the main concepts in CS, the following definitions will introduce some necessary terminology. As we will switch back and forth between real and complex cases, we will introduce the first notions for a general field $\mathbb{K}$, which can be either $\mathbb{R}$ or $\mathbb{C}$.

Definition 2.1. The support of a vector $\mathbf{v} \in \mathbb{K}^n$ is defined as the index set of its non-zero entries, that is:

$$\operatorname{supp} \mathbf{v} = \{i \in \{1, 2, \dots, n\} \mid v_i \neq 0\}$$

Throughout the thesis we will use $\|\mathbf{v}\|_0 = \left|\operatorname{supp} \mathbf{v}\right|$ to denote the cardinality of the support set, ie the number of non-zero entries. We will call this the *ℓ_0-norm*, even though $\|\cdot\|_0$ is strictly speaking not a norm as it fails to comply with the scaling property for norms. This misuse of terms is quite normal in the CS literature.

Definition 2.2. A vector $\mathbf{v} \in \mathbb{K}^n$ is said to be s-sparse if it has no more than s non-zero entries, that is if $\|\mathbf{v}\|_0 \leq s$.

To recover an undersampled signal with CS, we will assume the original signal to be sparse. Since the main topic for this thesis is not CS, we will give a slightly simplified introduction. At the end of this section, we will discuss these simplifications, and point to further reading on how to circumvent them.

Given an original signal $\mathbf{x} \in \mathbb{C}^n$, a measurement operator $\mathbf{A} \in \mathbb{C}^{m \times n}$, and measurements $\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{C}^m$, we seek to reconstruct $\mathbf{x}$ from $\mathbf{y}$. However, since $m < n$ the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{y} \tag{2.1}$$

is under-determined and has an infinite number of solutions. In order to pick a solution from this solution space we will assume the original signal to be sparse, and pick the sparsest solution to Equation (2.1). In other words, we wish to solve the following optimization problem:

$$\min_{\mathbf{z}\in\mathbb{C}^N} \|\mathbf{z}\|_0 \quad \text{subject to } \mathbf{A}\mathbf{z} = \mathbf{A}\mathbf{x} \tag{P_0}$$

However, ℓ_0 optimization is known to be NP-hard in general[2]. Since this makes (P_0) intractable, we will solve the convex relaxation of (P_0) instead:

$$\min_{\mathbf{z}\in\mathbb{C}^N} \|\mathbf{z}\|_1 \quad \text{subject to } \mathbf{A}\mathbf{z} = \mathbf{A}\mathbf{x} \tag{P_1}$$

Solving inverse problems by solving (P_1) is known as Basis Pursuit (BP) [FR13, Chapter 4].

Guaranteeing correctness of recovered signals

Guaranteeing the success of Basis Pursuit can be split into two sub-problems. First, we must guarantee that (P_0) has a unique solution. Second, we must ensure that the relaxation (P_1) also has a unique solution, and that the signal realizing the solution to (P_1) also realizes the solution to (P_0).

We begin by showing when (P_0) has unique solutions:

Theorem 2.3. *If the following equality holds:*

$$\left\{\mathbf{z} \in \mathbb{C}^N \mid \mathbf{A}\mathbf{z} = \mathbf{A}\mathbf{x}, \|\mathbf{z}\|_0 \leq s\right\} = \{\mathbf{x}\}$$

That is, if $\mathbf{x}$ *is the unique* s*-sparse solution to* (P_0)*, then the number of measurements* m *(ie, the number of rows in* $\mathbf{A}$*) must satisfy* $m \geq 2s$*.*

Before proving this theorem, we need the following lemma (stated as part of Theorem 2.13 in [FR13]):

Lemma 2.4. *Every* s*-sparse vector* $\mathbf{x}$ *is the unique* s*-sparse solution to* (P_0) *with* $\mathbf{y} = \mathbf{A}\mathbf{x}$ *if and only if every set of* $2s$ *columns of* $\mathbf{A}$ *is linearly independent.*

Proof. Assume that every s-sparse vector $\mathbf{x}$ is the unique s-sparse solution to (P_0) with $\mathbf{y} = \mathbf{A}\mathbf{x}$. Let $\mathbf{v} \in \ker \mathbf{A}$ be $2s$-sparse, and let $\mathbf{x}$ and $\mathbf{z}$ be two s-sparse vectors with $\operatorname{supp}\mathbf{x} \cap \operatorname{supp}\mathbf{z} = \emptyset$ such that $\mathbf{v} = \mathbf{x} - \mathbf{z}$. Then,

$$\mathbf{0} = \mathbf{A}\mathbf{v} = \mathbf{A}(\mathbf{x} - \mathbf{z}) = \mathbf{A}\mathbf{x} - \mathbf{A}\mathbf{z} \implies \mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{z}$$

and by assumption, $\mathbf{x} = \mathbf{z}$. Since $\mathbf{x}$ and $\mathbf{z}$ have disjoint supports, it follows that $\mathbf{x} = \mathbf{z} = \mathbf{0}$ and that $\mathbf{v} = \mathbf{0}$. Hence the only $2s$-sparse vector in $\ker \mathbf{A}$ is $\mathbf{0}$. Thus, for any set S with $|S| = 2s$ we have that the linear set of equations

$$\mathbf{A}_S\mathbf{x} = \mathbf{0}$$

has a unique solution, and by the Invertible Matrix Theorem, it follows that any selection of $2s$ columns of $\mathbf{A}$ must be linearly independent.

[2] A proof of the NP-hardness of (P_0) can be found in Section 2.3 of [FR13], and is obtained by reducing (P_0) to the *exact cover by 3-sets* problem, which is known to be NP-complete.

Conversely, assume that every set of $2s$ columns of $\mathbf{A}$ is linearly independent. By the Invertible Matrix Theorem we have that

$$\ker \mathbf{A} \cap \left\{ \mathbf{v} \in \mathbb{C}^n \mid \|\mathbf{v}\|_0 \leq 2s \right\} = \{\mathbf{0}\}$$

Now, let $\mathbf{x}, \mathbf{z}$ be s-sparse with $\mathbf{Ax} = \mathbf{Az}$. Then, $\mathbf{x} - \mathbf{z}$ is $2s$-sparse, and

$$\mathbf{Ax} = \mathbf{Az} \implies \mathbf{0} = \mathbf{Ax} - \mathbf{Az} = \mathbf{A}(\mathbf{x} - \mathbf{z})$$

and since the kernel of $\mathbf{A}$ does not contain any other $2s$-sparse vectors than $\mathbf{0}$, we have that $\mathbf{x} = \mathbf{z}$, which establishes the uniqueness of the solution and concludes the proof. □

Proof of Theorem 2.3. Assume that it is possible to uniquely recover any s-sparse vector $\mathbf{x}$ from the knowledge of its measurement vector $\mathbf{y} = \mathbf{Ax}$. Then, by Lemma 2.4, we have that every set of $2s$ columns of $\mathbf{A}$ must be linearly independent. This implies that $\operatorname{rank} \mathbf{A} \geq 2s$. From linear algebra we know that the rank of a matrix can not be bigger that the number of rows m, hence $\operatorname{rank} \mathbf{A} \leq m$. Combining this, we get that

$$2s \leq \operatorname{rank} \mathbf{A} \leq m$$

which concludes the proof. □

Next up, we will show when Basis Pursuit actually solves (P_0). We begin by introducing the Null Space Property:

Definition 2.5. A matrix $\mathbf{A} \in \mathbb{K}^{m \times N}$ is said to satisfy the *Null Space Property (NSP) relative to* a set $S \subset \{1, 2, \ldots, N\}$ if

$$\|\mathbf{v}_S\|_1 < \|\mathbf{v}_{\overline{S}}\|_1 \quad \text{for all } \mathbf{v} \in \ker \mathbf{A} \setminus \{\mathbf{0}\}$$

It is said to satisfy the *Null Space Property of order s* if it satisfies the null space property relative to any set $S \subset \{1, 2, \ldots, N\}$ with $|S| \leq s$

The following theorem and corollary shows that the NSP will be a sufficient condition for our measurement operator in order to ensure the success of Basis Pursuit.

Theorem 2.6 [FR13, Thm. 4.4]**.** *Given a matrix $\mathbf{A} \in \mathbb{K}^{m \times N}$, every vector $\mathbf{x} \in \mathbb{K}^N$ supported on a set S is the unique solution to (P_1) with $\mathbf{y} = \mathbf{Ax}$ if and only if $\mathbf{A}$ satisfies the NSP relative to S.*

Proof. We will begin by proving that if a vector $\mathbf{x}$ supported on S uniquely solves (P_1), then $\mathbf{A}$ satisfies the NSP relative to S.

Given an index set S, assume that every vector $\mathbf{x} \in \mathbb{K}^N$ supported on S is the unique solution to

$$\min_{\mathbf{z} \in \mathbb{C}^N} \|\mathbf{z}\|_1 \quad \text{subject to } \mathbf{Az} = \mathbf{Ax} \tag{P_1}$$

Since $\ker \mathbf{A}$ is a subspace of $\mathbb{K}^N$, it is clear that for any $\mathbf{v} \in \ker \mathbf{A} \setminus \{\mathbf{0}\}$, the vector $\mathbf{v}_S$ is the unique solution to

$$\min_{\mathbf{z} \in \mathbb{C}^N} \|\mathbf{z}\|_1 \quad \text{subject to } \mathbf{A}\mathbf{z} = \mathbf{A}\mathbf{v}_S \tag{2.2}$$

Because $\mathbf{v} \in \ker \mathbf{A}$, we have that $\mathbf{A}\mathbf{v} = \mathbf{0}$, which means that $\mathbf{A}(\mathbf{v}_S + \mathbf{v}_{\overline{S}}) = \mathbf{0}$, giving us that $\mathbf{A}(-\mathbf{v}_{\overline{S}}) = \mathbf{A}\mathbf{v}_S$. Hence it is clear that $-\mathbf{v}_{\overline{S}}$ is also a feasible solution to (2.2), but since $\mathbf{v}_S$ is assumed to be the *unique* optimal solution to (2.2), we get that $\|\mathbf{v}_S\|_1 < \|-\mathbf{v}_{\overline{S}}\|_1$. Since $\|\cdot\|_1$ is a norm, we have that $\|-\mathbf{v}_{\overline{S}}\|_1 = |-1|\,\|\mathbf{v}_{\overline{S}}\|_1 = \|\mathbf{v}_{\overline{S}}\|_1$. Thus, we arrive at the following inequality:

$$\|\mathbf{v}_S\|_1 < \|\mathbf{v}_{\overline{S}}\|_1$$

This establishes the NSP for $\mathbf{A}$, relative to S.

To prove the other implication, assume first that the NSP holds for $\mathbf{A}$, relative to a given set S. Let $\mathbf{x}$ be a vector in $\mathbb{K}^N$ supported on S. Let $\mathbf{z} \in \mathbb{K}^N$ be a vector that satisfies $\mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{z}$, and assume that $\mathbf{x} \neq \mathbf{z}$. Our goal will be to show that $\|\mathbf{z}\|_1$ must be strictly bigger than $\|\mathbf{x}\|_1$, which will prove the uniqueness of the solution.

Define $\mathbf{v} = \mathbf{x} - \mathbf{z}$. Since $\mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{z}$, we have that

$$\mathbf{0} = \mathbf{A}\mathbf{x} - \mathbf{A}\mathbf{z} = \mathbf{A}(\mathbf{x} - \mathbf{z}) = \mathbf{A}\mathbf{v}$$

This means that $\mathbf{v} \in \ker \mathbf{A}$. Since $\mathbf{x} \neq \mathbf{z}$, we also have that $\mathbf{v} \neq \mathbf{0}$. If we use the triangle inequality of norms, as well as the definition of $\mathbf{v}$, we obtain

$$\|\mathbf{x}\|_1 = \|\mathbf{x} - \mathbf{z}_S + \mathbf{z}_S\|_1 \leq \|\mathbf{x} - \mathbf{z}_S\|_1 + \|\mathbf{z}_S\|_1 = \|\mathbf{v}_S\|_1 + \|\mathbf{z}_S\|_1$$

Now, using the assumption that $\mathbf{A}$ satisfies the NSP relative to S we get the next inequality

$$\|\mathbf{v}_S\|_1 + \|\mathbf{z}_S\|_1 < \|\mathbf{v}_{\overline{S}}\|_1 + \|\mathbf{z}_S\|_1$$

Using the definition of $\mathbf{v}$ and $\mathbf{z}$ again, we arrive at our final result:

$$\|\mathbf{v}_{\overline{S}}\|_1 + \|\mathbf{z}_S\|_1 = \|\mathbf{x}_{\overline{S}} - \mathbf{z}_{\overline{S}}\|_1 + \|\mathbf{z}_S\|_1 = \|-\mathbf{z}_{\overline{S}}\|_1 + \|\mathbf{z}_S\|_1 = \|\mathbf{z}\|_1$$

This proves that $\|\mathbf{x}\|_1 < \|\mathbf{z}\|_1$ for any $\mathbf{z} \in \mathbb{K}^N$ satisfying $\mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{z}$ and $\mathbf{x} \neq \mathbf{z}$. This establishes the required minimality of $\|\mathbf{x}\|_1$, and thus the uniqueness of the solution. □

Theorem 2.6 is not in itself a sufficient guarantee of correctness, but if we let the set S vary, it immediately yields a more general result:

Corollary 2.7 [FR13, Thm. 4.5]**.** *Given a matrix $\mathbf{A} \in \mathbb{K}^{m \times N}$, every s-sparse vector $\mathbf{x} \in \mathbb{K}^N$ is the unique solution to* (P_1) *with $\mathbf{y} = \mathbf{A}\mathbf{x}$ if and only if $\mathbf{A}$ satisfies the NSP of order s.*

Before we prove this result, we will give a small remark: Corollary 2.7 shows that if $\mathbf{A}$ satisfies the NSP of order s, the ℓ_1-minimization strategy of (P_1) will actually solve (P_0) for all s-sparse vectors.

Proof of Corollary 2.7. Assume every s-sparse vector $\mathbf{x} \in \mathbb{K}^N$ is the unique solution to (P_1). Then, for every set S with $|S| \leq s$ we can find a vector $\mathbf{x}' \in \mathbb{K}$ supported on S which is the unique solution to (P_1). By Theorem 2.6 we then have that $\mathbf{A}$ must satisfy the NSP relative to S. Since this is true for all S with $|S| \leq s$, $\mathbf{A}$ must satisfy the NSP of order s.

Conversely, assume that $\mathbf{A}$ satisfies the NSP of order s. Then, from Definition 2.5, we have that $\mathbf{A}$ satisfies the NSP relative to S for every set S with $|S| \leq s$. From Theorem 2.6 we have that a vector $\mathbf{x} \in \mathbb{K}^N$ is supported on S only if it is the unique solution to (P_1). Since this is true for any set S with $|S| \leq s$, it is true for any s-sparse vector. □

Further reading: In this introduction, we have made some simplifying assumptions which will not hold up in the real world. First, we have assumed the signal to be *uniformly sparse*. That is, the non-zero entries in the signal have been assumed to be located at any index with the same probability throughout. Any reader familiar with sparsifying transforms, such as the discrete wavelet transform, will know that this is not the case. Normally, we will have some areas in the signal with a higher density of non-zero entries[3]. Recent work in CS have approached this problem by introducing *sparsity in levels* and associated results [Adc+17]. This is often referred to as *Structured Compressive Sensing*.

Second, we have assumed the signal to be sparse. However, in reality coefficients are rather almost 0 than exactly 0. We call such vectors *compressible*. Also, when working with real-valued numbers on a computer, measurement errors are unavoidable. This poses several challenges, such as the equality constraint in (P_1). We can solve this issue by introducing a small error term $\varepsilon > 0$, and defining the following revision of (P_1):

$$\min_{\mathbf{z} \in \mathbb{C}^N} \|\mathbf{z}\|_1 \quad \text{subject to} \|\mathbf{A}\mathbf{z} - \mathbf{A}\mathbf{x}\| \leq \varepsilon \tag{$P_{1,\varepsilon}$}$$

Extensions to the NSP exist, such as the *Robust NSP* [FR13, Def. 4.17] which gives an upper bound on the reconstruction error when compressibility and measurement errors are taken into account [FR13, Thm. 4.19].

Modern, state-of-the-art CS also sometimes employs regularization techniques like Total (Generalized) Variation, alone or in combination with a sparsifying transform [Ant+19; MM16]. Further, in some cases a *Shearlet* or *Curvelet* transform is preferred over a Wavelet transform[4] as a sparsifying transform [Ant+19].

Current challenges with compressive sensing

The main challenge with modern state-of-the-art CS are long runtimes. The optimization problem in CS is often very large, consisting of a huge amount of variables. Running 1000 iterations of FISTA, a popular algorithm for ℓ_1 minimization [BT09], on a 2048×2048 image using a modern desktop computer will often take around half an hour to complete.

[3] see Figures 2.2b and 2.2c for an example, note how the upper left corner contains vastly more information than any other part of the image

[4] The amalgamation of these techniques is often called X-lets.

(a) Original image of Lily **(b)** Result of a 1-level DWT **(c)** Result of a 2-level DWT

Figure 2.2: Example of a Haar DWT

However, it is worth noting that many of the modern implementations of these algorithms are sequential, and written for CPUs. Since many of the computations involved are easily parallelizable and well-suited for GPU computations (such as matrix-vector multiplications, FFTs, etc), one could expect large speedups by implementing these algorithms in a different way. In fact, a recent implementation of FISTA using TensorFlow brought the computation time down to around 90 seconds when executing on a GPU [Hau19].

2.2 Measurement operators

In this section we will study our measurement operators in more detail. First, recall that MRI measurements are in the *Fourier domain*. That is, if we performed full sampling, our measurement operator $\mathbf{A}$ would simply be the n-point Fourier matrix $\mathbf{F}_n$ (that is, the shift of coordinates matrix from the n-dimensional standard basis to the n-dimensional Fourier basis).

However, we wish to obtain fewer samples. Let Ω be a set of the indices for the samples to include. Thus, the number of samples m is the size of this set:

$$m = |\Omega|$$

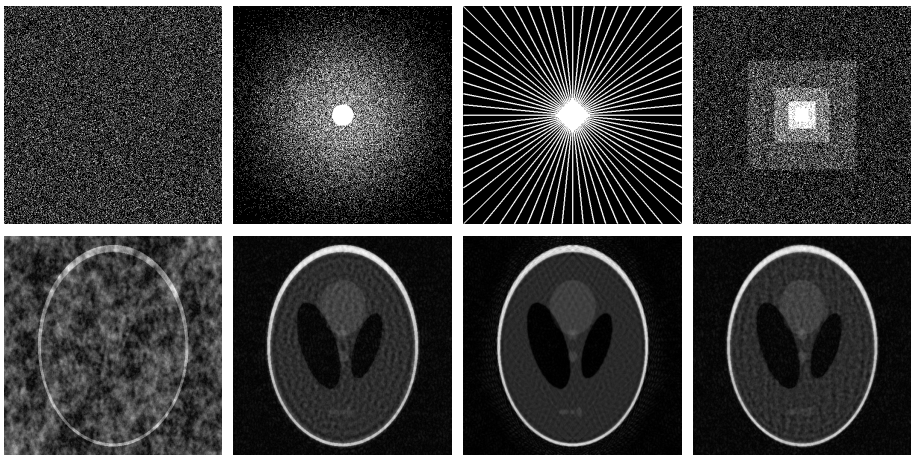
Let $\mathbf{P}_\Omega \colon \mathbb{K}^n \to \mathbb{K}^m$ be the projection to the subspace indexed by Ω. Thus, a possible measurement operator becomes:

$$\mathbf{A} = \mathbf{P}_\Omega \mathbf{F}_n \tag{2.3}$$

This is the measurement operator typically used in Deep Learning-based reconstruction.

However, since Compressive Sensing assumes the signal to be sparse, the measurement operator in Equation (2.3) is not adequate since the Fourier spectrum of a natural image is rarely sparse.

We circumvent this issue by applying a sparsifying transform. A sparsifying transform $\mathbf{\Psi}$ is an invertible operator that maps a non-sparse signal $\mathbf{x}$ to sparse representation $\mathbf{\Psi x}$. One possible sparsifying transform is a *Discrete Wavelet Transform (DWT)*.



(a) Uniform **(b)** Gaussian **(c)** Radial **(d)** Level

Figure 2.3: Different sampling patterns (top) and the reconstruction of the Shepp–Logan phantom (from Figure 2.1a) using the respective adjoint operators (bottom). The sampling rate is fixed at 20%

The key idea in a wavelet transform is to take some signal, expressed in a high resolution wavelet basis, and express it in terms of a lower resolution basis, and a detail basis. A wavelet is defined by a mother wavelet ψ and a scaling function ϕ. By shifting and scaling these functions we obtain a basis for the low resolution wavelet space (from the ϕ's) and the detail space (from the ψ's). A Discrete Wavelet Transform is then a change of coordinates from the higher resolution wavelet basis, to a lower resolution and detail basis. An example of a DWT is found in Figure 2.2. For a complete introduction to wavelets we refer to Chapter 4 and onwards in [Rya19].

Since wavelet coefficients are often sparse (or at least very compressible), we can adapt CS to MRI reconstruction by recovering the wavelet coefficients of the Fourier measurements. Hence, the following operator is a suitable measurement operator for CS:

$$\mathbf{A} = \mathbf{P}_\Omega \mathbf{F}_n \mathbf{\Psi}^{-1} \tag{2.4}$$

Sampling patterns

How we distribute the samples in Ω can have a huge impact on the ability of our reconstruction scheme to successfully recover the signal.

Some popular choices of sampling patterns, and the reconstruction using their respective adjoints, are depicted in Figure 2.3. Perhaps the easiest sampling pattern to conceive of is uniform sampling:

- **Uniform sampling** draws samples randomly from a uniform distribution. Meaning the indices (i, j) to include are drawn from two independent uniform distributions until a desired sampling rate is achieved.

However, since most of the energy in the signal is often centered around the origin (when viewed in the frequency domain), we wish to include more samples from around the origin. Several sampling schemes achieve this:

- **Gaussian sampling** draws samples randomly, but from a truncated normal distribution rather than a uniform one. The distribution is centered around the origin, and has variance σ^2. This variance acts as a spread parameter, determining how center-heavy the sampling pattern should be. Indices to include are drawn from this distribution until a desired sampling rate is achieved. It is also common to add a ball around the origin where every sample is included.
- **Radial sampling** draws equiangular lines from the origin to the edges. The number and thickness of these lines determine the sampling rate.
- **Level sampling** performs uniform sampling with increasing sampling rates closer to the origin. The innermost level typically has a sampling rate of 1.

CHAPTER 3

Neural Networks

The use of Neural Networks in machine learning can be tracked back several decades, however the last decade shows a huge increase in their popularity. The success of AlexNet [KSH12] on the 2012 ImageNet Large Scale Visual Recognition Challenge[1] in many ways marks the beginning of the neural network revolution. AlexNet won by a clear margin, with an error rate more than 10% lower than the closest runner-up. Since AlexNet, all winners of the ImageNet Challenge have been variations of a Neural Network, with current error rates under 5% [He+16; Sze+17].

The success of Neural Networks as one-size-fits-all classifiers has led researchers to adapt them to other problems than visual recognition. In this thesis, we will mainly focus on the use of Neural Networks to reconstruct undersampled images. To do this, several different approaches have been proposed [MJU17], however the denoising approach of [Sch+18; Yan+18] seems to be most common for MRI reconstruction. We will explore this in detail in Section 3.4.

We begin this chapter by giving a general introduction to supervised learning, the framework for which Neural Networks are most often used. We will then present Neural Networks, and give details on how to construct them in practice. We finish the chapter with an introduction on how to adapt Neural Networks to inverse problems for MRI reconstruction, and present two contemporary networks [Sch+18; Yan+18].

3.1 Supervised Learning

Before we introduce neural networks properly, we will give a brief introduction to general supervised learning. Given a set of data tuples $\{(x_i, y_i)\}_{i=1}^n$, usually referred to as the *training data*, the main assumption in supervised learning is that there exists some mapping F such that

$$y_i = F(x_i) + \varepsilon_i \quad \text{for all } i \in \{1, 2, \dots, n\} \tag{3.1}$$

where $\varepsilon_i \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$. The goal of supervised learning algorithms is to find an approximator $\hat{F}$ for this mapping. Given a new, previously unseen input x, we can predict what the output will be using this approximator:

$$\hat{y} = \hat{F}(x) \qquad \text{([ISL, Eq. (2.2)])}$$

[1] A competition for software to classify more several million images of 1000 different objects.

We often call $\hat{F}$ a *model* and $\hat{y}$ a *prediction*.

Two big applications for supervised learning are regression and classification. In regression, x and y are both continuous variables, often some vectors in $\mathbb{R}^n$ or $\mathbb{C}^n$[2]. In classification, the input x is often some vector, but the output y (usually called the *label* of x in this case) is categorical. I.e., the output is some element in a finite set of known possibilities.

Inverse Problems as Regression

As a little tangent at the end of the section, we will look at how we can use the statistical framework of supervised learning to solve inverse problems directly.

Given a set of training data, $\{\mathbf{x}_i\}_{i=1}^n$ we can create the expected output by applying the measurement operator $\mathbf{A}$. Thus our full training set becomes $\{(\mathbf{A}\mathbf{x}_i, \mathbf{x}_i)\}_{i=1}^n$, or by defining $\mathbf{y}_i = \mathbf{A}\mathbf{x}_i$, we get $\{(\mathbf{y}_i, \mathbf{x}_i)\}_{i=1}^n$. Note that the original signal $\mathbf{x}$ now plays the role of the *output* of the learning scheme, previously denoted y, and vice versa for $\mathbf{y}$. The model $\hat{F}$ will in this case be fitted to act as an inverse of $\mathbf{A}$ on the given training data.

Some researchers have found success using this approach on reconstructing undersampled MRI images [Zhu+18]. Recent work have however shown [Zhu+18] to suffer from major instabilities [Ant+19]. The denoising approach described in Section 3.4 remain the most used [MJU17], and will be the main focus of this thesis.

3.2 Neural Networks

The study of neural networks can be traced back several decades, but has gained popular ground during the last decade after some enormous success stories, especially on the image classification problem. Neural networks are a family of functions known to be good at approximating arbitrary functions. Contrary to many other classical approaches, neural networks do not place any assumptions on F, and are thus well suited as a one-size-fits-all approximator.

Giving a formal definition of Neural Networks turns out to be very difficult, as it is often not done by the community. Already in 1999, mathematician Allan Pinkus describes the difficulty of giving a definition that includes all contemporary Neural Networks [Pin99]. We will, however, give a definition, which we will later refine when necessary:

Definition 3.1. Let $L, N_0, \dots, N_L \in \mathbb{N}$, let $W_l \colon \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$ for $l = 1, \dots, L$ be affine maps. Let $\rho_1, \dots, \rho_L \colon \mathbb{R} \to \mathbb{R}$ be some non-linear, differentiable functions, and let $\boldsymbol{\rho}_1 \dots \boldsymbol{\rho}_L$ be the same functions acting element-wise on vectors. A map $\Phi \colon \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ given by

$$\Phi(\mathbf{x}) = \boldsymbol{\rho}_L(W_L(\boldsymbol{\rho}_{L-1}(W_{L-1}(\cdots \boldsymbol{\rho}_1(W_1(x)) \cdots)))) \tag{3.2}$$

is called a *Neural Network (NN)*.

[2] All though in many cases, but not in ours, the output y is a singe number and not a vector.

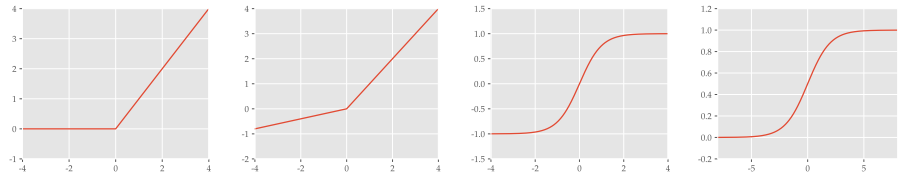


Figure 3.1: Different choices of activation functions. From left to right: Rectified Linear Unit (ReLU), Leaky ReLU with slope 0.2 ($\text{LReLU}_{0.2}$), hyperbolic tangent (tanh) and sigmoid (σ).

We often refer to the ρ_l-s as *activation functions* or *non-linearities*, and the W_l-s as *layers*[3] of the network. The total specification of the number of layers, the size of each layer, the choice of activation functions and so on is referred to as the *architecture* of the network.

Typically, the number of layers varies somewhere between 20 and 50 [SZ14; He+15], with some extreme edge cases [He+16], and is referred to as the *depth* of the network. Using deep neural networks to solve the supervised learning problem is often dubbed Deep Learning (DL).

Some popular choices for non-linearities are

$$\begin{aligned} \text{ReLU}(x) &= \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \\ \text{LReLU}_\alpha(x) &= \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \\ \tanh(x) &= \frac{e^{2x} - 1}{e^{2x} + 1} \\ \sigma(x) &= \frac{1}{1 + e^{-x}} \end{aligned} \tag{3.3}$$

Plots of these functions are found in Figure 3.1. The Rectified Linear Unit (ReLU) and its variations are by far the most used in state-of-the-art Deep Learning.

To simplify notation, we refer the output of the non-linearities as *activations*, and introduce the notation $\mathbf{a}_l$ for the activations at layer l. Thus, we can rewrite Equation (3.2) as

$$\begin{aligned} \mathbf{a}_1 &= \rho_1(W_1(\mathbf{x})) \\ \mathbf{a}_2 &= \rho_2(W_2(\mathbf{a}_1)) \\ \mathbf{a}_3 &= \rho_3(W_3(\mathbf{a}_2)) \\ &\vdots \\ \mathbf{a}_L &= \rho_L(W_L(\mathbf{a}_{L-1})) \\ \Phi(\mathbf{x}) &= \mathbf{a}_L \end{aligned} \tag{3.4}$$

[3] This distinction between layers and non-linearities is not always found in the literature. Some include the non-linearity as a part of the layer, so that $\rho_l(W_l(\cdot))$ is referred to as a layer. In this thesis however, we will separate the non-linearity from the layer and only refer to the affine mapping.

The choice of activation function is important for the performance and training of the neural network. Some activation functions, such as the sigmoid and the hyperbolic tangent, have the problem of *vanishing gradients*. I.e., if the input value is too large or too small, the derivative (important for training, see Section 3.3) will be almost 0.

Even though neural networks are wildly used in machine learning, the rigorous study into the mathematics of contemporary Deep Learning is somewhat lacking [LS18]. Some results exist, however, most notably the Universal Approximation Theorem (UAT), introduced in [Cyb89; Hor91].

The UAT as originally stated in [Cyb89] covers *sigmoidal* activation functions, which are functions f that satisfies $\lim_{x\to\infty} f(x) = 1$ and $\lim_{\mathbf{x}\to-\infty} f(x) = 0$. Extensions to the UAT for non-sigmoidal cases have been proven later [Pin99; SM17], we will however cover the UAT as originally stated.

Theorem 3.2 (Universal Approximation Theorem)**.** *Let $\rho\colon \mathbb{R} \to \mathbb{R}$ be a bounded, measurable, sigmoidal function. Let $\mathcal{S} \subset \mathbb{R}^n$ be a compact set and let $C(\mathcal{S})$ denote the vector space of continuous functions on $\mathcal{S}$.*

Then for any $f \in C(\mathcal{S})$ and any $\varepsilon > 0$, there exists a set of parameters $N \in \mathbb{N}$, $v_1, \dots, v_N \in \mathbb{R}$, $\mathbf{w}_1, \dots, \mathbf{w}_N \in \mathbb{R}^n$, $b_1, \dots, b_N \in \mathbb{R}$ such that $f : \mathbb{R}^n \to \mathbb{R}$ defined as

$$f(\mathbf{x}) = \sum_{i=1}^{N} v_i \rho(\mathbf{w}_i^T \mathbf{x} + b_i) \tag{3.5}$$

satisfies

$$\|f(\mathbf{x}) - g(\mathbf{x})\| < \varepsilon$$

for all $\mathbf{x} \in \mathcal{S}$. In other words, dense neural networks with one hidden layer are dense in $C(\mathcal{S})$

Before proving the theorem, we must introduce the notion of *discriminatory functions*. First, let $M(\mathcal{S})$ be the set of signed Borel measures on $\mathcal{S}$.

Definition 3.3. A function ρ is *discriminatory* if for a measure $\mu \in M(\mathcal{S})$, we have that

$$\int_{\mathcal{S}} \rho(\mathbf{w}^T \mathbf{x} + b) d\mu(\mathbf{x}) = 0 \quad \text{for all } \mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R}$$

implies that $\mu = 0$.

Proof of Theorem 3.2 (as given in [Cyb89]). Let $\mathcal{F} \subset C(\mathcal{S})$ be the space of functions on the form given in Equation (3.5). We must show that the closure $\overline{\mathcal{F}}$ of $\mathcal{F}$ is $C(\mathcal{S})$.

Assume for contradiction that $\overline{\mathcal{F}} \neq C(\mathcal{S})$, that is $\mathcal{F}$ is a closed proper subspace of $C(\mathcal{S})$. Then, by the Hahn–Banach theorem [MW13, Thm. 14.1], there exists a bounded linear functional L on $C(\mathcal{S})$ such that $L \neq 0$, but $L(\overline{\mathcal{F}}) = L(\mathcal{F}) = 0$.

By the Riesz Representation Theorem [MW13, Thm. 13.15], L can be expressed as

$$L(h) = \int_{\mathcal{S}} h(x) d\mu(x) \quad \text{for all } h \in C(\mathcal{S})$$

for some $\mu \in M(\mathcal{S})$. Particularly, since $\rho(\mathbf{w}^T \mathbf{x} + b) \in \mathcal{F}$ for all $\mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R}$, we have that

$$\int_{\mathcal{S}} \rho(\mathbf{w}^T \mathbf{x} + b) d\mu(x) = 0$$

and since all bounded, measurable and sigmoidal functions are discriminatory [Cyb89, Lemma 1] this implies that $\mu = 0$, which in turn implies that $L = 0$ which is a contradiction. Thus, the closure $\overline{\mathcal{F}} = C(\mathcal{S})$ which concludes the proof. □

This result provides some theoretical justification for the success of neural networks as universal approximators. However, we will give a couple of remarks to the UAT as originally presented:

Remark 3.4. Theorem 3.2 covers single-layer dense neural networks. However, in practice we often work with much deeper nets, and with convolutional layers instead of dense layers. The importance of depth were mentioned in [KSH12], but not formally proven.

Remark 3.5. The most used activation function by far is the Rectified Linear Unit (ReLU), which is not a bounded function. Recent work has shown that the UAT can be extended to certain non-bounded cases, such as the ReLU [SM17].

Remark 3.6. Theorem 3.2 is non-constructive. It only tells us that a certain network *exists*, but does not relate to learning.

Convolutional layers

Modern networks usually apply other kinds of layers in addition to, or instead of, the dense layers described in Definition 3.1. Most notable are convolutional layers. These are achieved by restricting the matrix multiplication in the affine maps W_l to be a cascade of convolutions, as such:

$$\text{Dense layers:} \quad W_l(\mathbf{a}_{l-1}) = \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l \tag{3.6}$$

$$\text{Convolutional layers:} \quad W_l(\mathbf{a}_{l-1}) = \mathbf{w}_l * \mathbf{a}_{l-1} + \mathbf{b}_l \tag{3.7}$$

Networks applying convolutional layers are called Convolutional Neural Networks (CNNs). Note that since convolutions can be written as linear operators (see page 44), convolutional layers are still affine mappings and fit well with Definition 3.1.

To keep our theory closely aligned with practical implementations, we will treat dimensionality in the same way as TensorFlow[4]. Hence, when applying neural networks to images, we will handle dimensionality differently for dense and convolutional layers. In dense layers, the columns of the

[4] One of the most popular Deep Learning frameworks [TF].

image matrix are stacked as a vector making the image a 1-dimensional signal. While in convolutional layers, the dimensionality is kept, and a 2-dimensional convolution is performed. In addition to this, natural images often have different colors, and as hinted above we will use an array of convolutions performed separately. We also introduce a forth dimension for batch calculations. Thus, the input of a convolutional layer is a 4-dimensional tensor $\mathbf{a}_{\text{in}} \in \mathbb{R}^{M \times m \times n \times c_{\text{in}}}$ (batch size, height, width, number of input channels) and with output $\mathbf{a}_{\text{out}} \in \mathbb{R}^{M \times (m/s_y) \times (n/s_x) \times c_{\text{out}}}$ and the weight tensor is a 4-dimensional tensor $\mathbf{w} \in \mathbb{R}^{d_h \times d_w \times c_{\text{in}} \times c_{\text{out}}}$ (filter height, filter width, number of input channels, number of output channels).

We then define the convolution in convolutional layers as the following:

$$(\mathbf{w} * \mathbf{a})_{b,i,j,k_2} = \sum_{i'} \sum_{j'} \sum_{k_1} w_{i',j',k_1,k_2} \cdot a_{b,i+s_y i',j+s_x j',k_1} \tag{3.8}$$

The s_x and s_y are called the *strides*, and depict the step length in the convolution. If $s_x = s_y = 1$ we obtain normal unstrided convolution. Note that strides > 1 decrease the signal dimensionality, for example $s_x = s_y = 2$ means that the width and height dimensions of the output tensor is half that of the input.

Remark 3.7. The observant reader might recognize Equation (3.8) as a correlation, and not a convolution. This misuse of terms is very common in the Deep Learning literature, and we will therefore use this terminology throughout. However, since the coefficients are learned and not set, it is actually not too misleading to refer to the operation as a convolution. The coefficients saved in the weights of the network will simply be the time-reversed complex-conjugate of the actual filter being applied.

Residual blocks

We will conclude this section with a layer variant which doesn't fit our initial definition of Neural Networks, namely residual blocks, popularized by [He+16].

In a residual block, the original signal is fed through several layers unaltered, in addition to the processed signal. An example of a 3-layered residual block may look like the following:

$$B(\mathbf{a}) = \rho_3(W_3(\rho_2(W_2(\rho_1(W_1(\mathbf{a})))))) + \mathbf{a}) \tag{3.9}$$

Note that it is usual to add the identity before applying the final non-linearity. Even though this type of block doesn't fit our initial definition of Neural Networks, we can expand Definition 3.1 to allow such blocks in addition to dense and convolutional layers.

As networks become deeper, several problems begin to occur. When feeding signals forward through the network, *signal degradation* comes into play, and experimental results have shown that the performance of a sufficiently deep network will decrease when adding more layers [He+16].

The motivation for residual blocks is that by passing the identity through, the effects of degradation and vanishing gradients are reduced, allowing deeper networks. In [He+16] the authors present a well-performing 152-layered

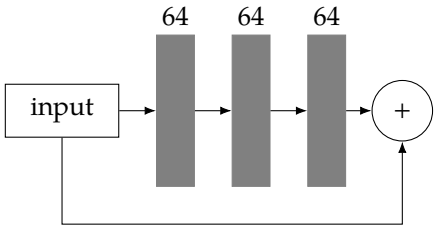


Figure 3.2: A typical graphical representation of a 3-layered residual block with 64-channeled convolutional layers

network for image classification, and produce experimental networks with over 1200 layers.

3.3 Training

Now that we have seen how neural networks are good candidates for approximating functions, the question becomes how to set the weights and biases in Equations (3.6) and (3.7). This is where the training data mentioned in Section 3.1 is used.

Loss functions

First, we must define a *cost* or a *loss function*. A loss function is a measure on how wrong a certain model $\hat{F}$ is, compared to the underlying F. Of course, we don't know F a priori, so the loss function must approximate the difference between $\hat{F}$ and F. To simplify notation, we will let θ be defined as the set of all trainable parameters. For a neural network this will typically be:

$$\theta = \{\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_L, \mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_L\}$$

We will sometimes simply refer to θ as the *parameters* of the network, and use the term *hyper-parameters* to refer to fixed, untrainable parameters. We write $\hat{F}_\theta$ to emphasize that this model uses θ as its parameters.

A popular choice of loss is the Mean Squared Error (MSE), also known as the mean ℓ_2-error. Given training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$, a set of parameters θ and a proposed model $\hat{F}_\theta$, the MSE loss of those parameters is defined as:

$$\mathcal{L}(\theta \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{y}_1, \ldots, \mathbf{y}_n) = \frac{1}{n} \sum_{i=1}^{n} \|\hat{F}_\theta(\mathbf{x}_i) - \mathbf{y}_i\|_2^2 \tag{3.10}$$

Sometimes it is more convenient to define the loss as a function of the model directly:

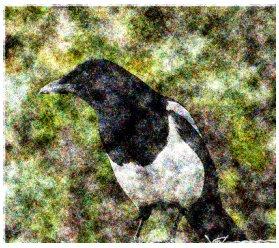
$$\mathcal{L}(\hat{F}_\theta \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{y}_1, \ldots, \mathbf{y}_n) = \frac{1}{n} \sum_{i=1}^{n} \|\hat{F}_\theta(\mathbf{x}_i) - \mathbf{y}_i\|_2^2 \tag{3.11}$$

The difference only being the notation. Throughout the thesis we will use both definitions, however it will always be clear from context which definition that is used.

(a) Base image

(b) Translation shifted

(c) Add $\varepsilon = 0.32$ to every pixel



(d) Noisy

(e) Another image of a bird

(f) An entirely different image

Image	(a)	(b)	(c)	(d)	(e)	(f)
ℓ_2 distance to (a)	0	215.04	204.80	167.26	216.44	193.15

Figure 3.3: Problems with ℓ_2 loss as a measure of perceptual likeness. Images (b) through (f) all have similar ℓ_2 distances to (a). The images were scaled so every color value is in the unit interval and cropped to 350×400 pixels. Images from the *ImageNet Large Scale Visual Recognition Challenge* dataset [Rus+15].

Even though the MSE is widely used, it is worth noting that MSE is not a good measure on *perceptual likeness*. Two images may look very much alike, and still have a large ℓ_2 distance between them. For example, given an $M \times N$ image $\mathbf{x}$ and a small value ε, the image $\mathbf{x} + \varepsilon\mathbf{1}$ may look like the same image – only a tiny bit brighter – while the ℓ_2 distance between them may be very large depending on the size of the image:

$$\|\mathbf{x} - (\mathbf{x} + \varepsilon\mathbf{1})\|_2 = \varepsilon\|\mathbf{1}\|_2 = \varepsilon\sqrt{MN}$$

Other issues include shifts in translation or rotation, which exhibits the same behavior where images that look alike may have large ℓ_2 distances, see Figure 3.3. In particular, note that Figures 3.3a and 3.3f are closer in an ℓ_2 sense than Figures 3.3a and 3.3c

When a loss function is picked, we can create a neural network based on the training data. First, fix an architecture, then pick the network parameters minimizing the loss over the dataset. That is, let the network parameters be the solution to the following minimization problem:

$$\theta = \arg\min_{\theta'} \mathcal{L}(\theta' \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{y}_1, \ldots, \mathbf{y}_n) \tag{3.12}$$

However, the loss as a function of the network parameters is a often highly non-convex function. Thus, solving Equation (3.12), even for relatively small networks, is highly intractable. In most cases we must settle for a local minimum instead of the optimal values.

Back-propagation and optimization

Before we discuss different optimization algorithms suited for finding a local minimum of Equation (3.12), we will show how to find the gradient of the loss function with respect to the network parameters.

To simplify notation, we will only consider finding the gradient given a single data point, although the method is easily generalizable to using several data points as well. That is, we will consider the slightly simplified case of deriving an expression for

$$\nabla_\theta \mathcal{L}(\theta \mid \mathbf{x}, \mathbf{y}) = \nabla_\theta \|\Phi_\theta(\mathbf{x}) - \mathbf{y}\|_2^2$$

Note that since

$$\nabla_\theta \|\Phi_\theta(\mathbf{x}) - \mathbf{y}\|_2^2 = 2(\Phi_\theta(\mathbf{x}) - \mathbf{y})^T \nabla_\theta \Phi_\theta(\mathbf{x})$$

the real challenge is to differentiate the neural network with respect to the parameters θ.

Because of the composite nature of Neural Networks (recall Equation (3.2)), the chain rule for differentiation is a natural choice of differentiation technique when differentiating Neural Networks. In order to further simplify notation, we will use the activation notation from Equation (3.4). By applying the chain rule to a Neural Network Φ_θ, we get

$$\begin{aligned}
\nabla_\theta \Phi_\theta(\mathbf{x}) &= \nabla_\theta \mathbf{a}_L \\
&= \nabla_\theta(\boldsymbol{\rho}_L(W_L(\mathbf{a}_{L-1}))) \\
&= \boldsymbol{\rho}_L'(W_L(\mathbf{a}_{L-1})) \cdot \mathbf{W}_L \cdot \nabla_\theta \mathbf{a}_{L-1}
\end{aligned}$$

We continue with the same approach on $\mathbf{a}_{L-1}$:

$$\begin{aligned}
&= \boldsymbol{\rho}_L'(W_L(\mathbf{a}_{L-1})) \cdot \mathbf{W}_L \cdot \nabla_\theta(\boldsymbol{\rho}_{L-1}(W_{L-1}(\mathbf{a}_{L-2}))) \\
&= \boldsymbol{\rho}_L'(W_L(\mathbf{a}_{L-1})) \cdot \mathbf{W}_L \cdot \boldsymbol{\rho}_{L-1}'(W_{L-1}(\mathbf{a}_{L-2})) \cdot \mathbf{W}_{L-1} \cdot \nabla_\theta \mathbf{a}_{L-2}
\end{aligned}$$

We continue applying the chain rule in this recursive fashion until the base case of $\mathbf{a}_1 = \boldsymbol{\rho}_1(W_1(\mathbf{x}))$. For each recursive step we find the derivative of the network with respect to the weights of the next layer.

In Deep Learning literature, this is often called Back-propagation, coined in [RHW86]. For a further explanation of Back-propagation with details on how the algorithm is implemented and typically interpreted by the machine learning community we refer to Section 6.5 of [GBC16].

The gradient of a function tells us in which direction the function grows most rapidly, hence the negative of the gradient tells us in which direction the function declines most rapidly. This is the main principle behind *Gradient Descent*[5]. This algorithm uses the entire dataset to compute the gradient of

[5] Also known as *Steepest Descent* in some mathematics literature.

Algorithm 3.1 Stochastic Gradient Descent [GBC16, Alg. 8.1]

Input: Initial parameters θ, step length η, number of epochs E, batch size M, training set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$.

1. For $e = 1, 2, \ldots, E$, do
   - 1.1 Create a new permutation $p = \{p_1, p_2, \ldots, p_N\}$ of $\{1, 2, \ldots, N\}$
   - 1.2 For $n = 0, 1, \ldots, \lceil N/M \rceil - 1$, do
     - 1.2.1 Sample M examples from shuffled data sets, $\left\{(\mathbf{x}_{p_j}, \mathbf{y}_{p_j})\right\}_{j=nM}^{(n+1)M}$
     - 1.2.2 Compute gradient of loss using sampled mini-batch, $\mathbf{g} \leftarrow \nabla \mathcal{L}(\theta \mid \mathbf{x}_{p_{nM}}, \ldots, \mathbf{x}_{p_{(n+1)M}}, \mathbf{y}_{p_{nM}}, \ldots, \mathbf{y}_{p_{(n+1)M}})$
     - 1.2.3 Apply update to parameters, $\theta \leftarrow \theta - \eta \mathbf{g}$

Output: Trained parameters θ

the network w.r.t. the parameters. Then, it subtracts a scaled version of the gradient from all the parameters. The scaling factor is often called the *step length* in mathematical optimization literature, or *learning rate* in Deep Learning literature.

This is however very ineffective in practice as using the entire dataset for every parameter update will make the parameter updates very costly to compute. Thus, pure Gradient Descent is rarely used in practice.

Instead, we will separate the training data into several batches and use one batch to compute one iteration. This means that the gradient computed will not be the true gradient of the loss function, but an approximation. However, since every iteration will be much cheaper to compute, the overall training speed will increase. When all the batches have been used once, we shuffle the training set and draw new batches. One cycle through the dataset in this fashion is referred to as an *epoch*. This algorithm is called Stochastic Gradient Descent (SGD), and is presented in detail in Algorithm 3.1.

We conclude this section on optimization with a brief introduction of the Adam (**Ada**ptive **M**oments) optimizer. Like with SGD, we separate the training set into batches. In addition, Adam enjoys two important features: momentum and adaptive step lengths. Using momentum is a technique developed to increase training speed on surfaces with high curvature [GBC16, Sec. 8.3.2]. As an example, let us consider adding momentum to SGD. We introduce a momentum hyper-parameter $\alpha \in [0, 1)$, and replace step 1.2.3 in Algorithm 3.1 with

- 1.2.3a Compute update: $\Delta\theta \leftarrow \alpha\Delta\theta - \eta\mathbf{g}$
- 1.2.3b Apply update: $\theta \leftarrow \theta + \Delta\theta$

This way, subsequent iterations will not substantially differ in direction, which makes the algorithm is less vulnerable to rapidly changing gradients. The term momentum is an analogy from physics, where energy is needed to change the trajectory of an object in motion [GBC16, p. 288]. Adam uses rescaling of

Algorithm 3.2 Adam [GBC16, Alg 8.7]

Input: Initial parameters θ, step length η, decay rates $\alpha_1, \alpha_2 \in [0, 1)$, small constant ε, number of epochs E, batch size M, training set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$.

1. Initialize time step $t \leftarrow 0$
2. Initialize 1st and 2nd moment variables $\mathbf{m}_1 \leftarrow 0$, $\mathbf{m}_2 \leftarrow 0$
3. For $e = 1, 2, \ldots, E$, do
   - 3.1 Create a new permutation $p = \{p_1, p_2, \ldots, p_N\}$ of $\{1, 2, \ldots, N\}$
   - 3.2 For $n = 1, 2, \ldots, N/M$, do
     - 3.2.1 Sample M examples from shuffled data sets, $\left\{(\mathbf{x}_{p_j}, \mathbf{y}_{p_j})\right\}_{j=nM}^{(n+1)M}$
     - 3.2.2 Compute gradient of loss using sampled mini-batch, $\mathbf{g} \leftarrow \nabla\mathcal{L}(\theta \mid \mathbf{x}_{p_{nM}}, \ldots, \mathbf{x}_{p_{(n+1)M}}, \mathbf{y}_{p_{nM}}, \ldots, \mathbf{y}_{p_{(n+1)M}})$
     - 3.2.3 Update time step, $t \leftarrow t + 1$
     - 3.2.4 Update biased first moment estimate, $\mathbf{m}_1 \leftarrow \alpha_1\mathbf{m}_1 + (1 - \alpha_1)\mathbf{g}$
     - 3.2.5 Update biased second moment estimate, $\mathbf{m}_2 \leftarrow \alpha_2\mathbf{m}_2 + (1 - \alpha_2)\mathbf{g}^2$
     - 3.2.6 Correct bias in first moment estimate, $\mathbf{m}_1 \leftarrow \mathbf{m}_1/(1 - \alpha_1^t)$
     - 3.2.7 Correct bias in first moment estimate, $\mathbf{m}_2 \leftarrow \mathbf{m}_2/(1 - \alpha_2^t)$
     - 3.2.8 Compute update, $\Delta\theta = -\eta\mathbf{m}_1/(\sqrt{\mathbf{m}_2} + \varepsilon)$
     - 3.2.9 Apply update to parameters, $\theta \leftarrow \theta + \Delta\theta$

Output: Trained parameters θ

the gradient in combination with momentum, which has no clear theoretical justification but seems to work well in practice [GBC16, Sec. 8.5.3].

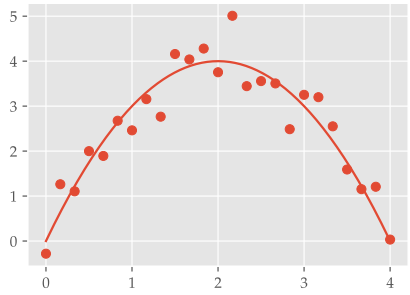
Second, Adam uses adaptive step lengths. The motivation for this technique is that if we set the step length in SGD to be too small, the training will be unnecessarily slow. If we set it to high, we will not be able to fine-tune the parameters sufficiently. One possible solution is to periodically reduce the step length η in Algorithm 3.1. The Adam optimizer automates this process. The full algorithm is presented in Algorithm 3.2.

For further reading on other optimization algorithms we refer to Sections 8.3, 8.5 and 8.6 of [GBC16].

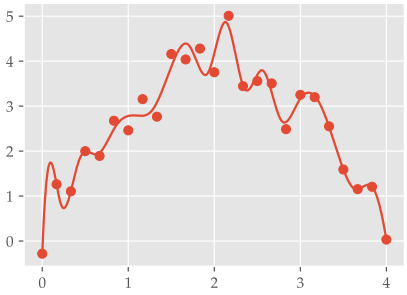
Initialization

Both Algorithms 3.1 and 3.2 require initial parameters. In this section we will briefly mention some of the most used initialization methods for neural networks.

Xavier [GB10] and He [He+15] initialization both follow the same principle. They draw the entries of the weight matrices from a uniform or normal distribution with parameters chosen so that the scale of the gradients are



(a) Noisy measurements along with the true underlying model.



(b) An extreme case of overfitting using cubic splines. See page 81 for details.

Figure 3.4: Example of overfitting.

similar throughout the layers. During the derivation of the expressions for the distribution parameters, [GB10] assumes linear activation, while [He+15] assumes ReLU activation. These techniques are the most common in state-of-the-art Deep Learning.

We will also include orthogonal initialization [SMG13], even though it is not as widely used as He or Xavier, as it will be useful when applying Parseval constraints in Chapters 5 and 6. Orthogonal initialization gives the weight matrices orthogonal rows or columns, depending on whether the matrix have more rows than columns or more columns than rows. Generating a random $m \times n$ matrix with orthogonal rows can be done by generating a square $n \times n$ matrix with entries drawn from a normal distribution, performing a QR-factorization and drawing the first m rows of the Q matrix.

Overfitting

In addition to the lack of convexity, overfitting poses a real challenge when training neural networks. The universality of neural networks is a double-edged sword. While able to fit any function, we risk learning the local noise as well. An example of overfitting can be seen in Figure 3.4.

Recall the underlying assumption for supervised learning (Equation (3.1)). Since we only know the value of y_i, there is no way of distinguishing the contribution of the underlying model from the noise [ISL, p. 22].

There are a few techniques available for detecting and combating overfitting. The most important one is to separate the available data into several datasets. One set will be used for training, one for validation and one for testing [ISL, Section 2.2]. When solving Equation (3.12) (or an estimate of it), we will use the training set to compute each iteration of the optimization algorithm, but after each step (or every nth step if one wish to save some computation time) we will compute the cost with regards to the validation set as well. The main idea is that an overfitted model will have low training cost, but high validation cost since it is trained on the noise from the training set. Hence we will look for the moment where these two values start to diverge [ISL, p. 32]. The test set it

not used until the very end to give a fair indication on the real-life performance for the model on previously unseen data.

To further combat overfitting and not merely detecting it, we must limit the generality of the approximation method in some way. Consider again the example in Figure 3.4. The model in Figure 3.4b was created minimizing the MSE between the spline and the measurements. If we wanted to achieve a smoother spline, we could add some *regularization term* to this loss function, for example punishing large values on the spline's derivative[6]. Thus, the loss for a specific spline f would be

$$\mathcal{L}(f) = \underbrace{\frac{1}{n}\sum_{i=1}^{n}\|f(x_i) - y_i\|_2^2}_{\text{MSE}} + \underbrace{\lambda\|f'\|}_{\substack{\text{Regularization}\\ \text{term}}} \tag{3.13}$$

The parameter λ is often referred to as a *regularization parameter*. Popular regularization terms for neural networks include the ℓ_1 or ℓ_2 norm of the weight matrices.

Other techniques more tailored to neural networks exist too. Among them are batch normalization [IS15] and dropout [Sri+14]. We will use some of them later in Chapter 6, but we will not go into more detail on how they work. In Chapter 6 we will also see how the Parseval constraint developed in Chapter 5 makes networks less prone to overfitting.

Data Augmentation

A dataset can be extended by creating artificial data based on the true data and adding it to the dataset. Data augmentation refers to any process where we create a new data point by altering an existing one in such a way that we do not change the nature of the data point, that is, the relation between the measured value and the expected output (Equation (3.1)) do not change [GBC16, Section 7.4].

Examples of data augmentation on MR images includes rotating the image, flipping the image along one or more axes, zooming in or introducing artificial noise. These alterations can be done beforehand, or on the fly during training.

3.4 Deep Learning Denoising

We begin our study of Deep Learning Denoising by showing how we can use Neural Networks to remove noise from an image.

Denoising

If we let the output dimensions of the network be the same as the input dimensions, we can train the network to detect and remove noise.

[6] We are ignoring spline-specific methods such as shrinking the spline space by e.g. manipulating the knot vector or lowering the degree as they have no direct equivalent for neural networks.

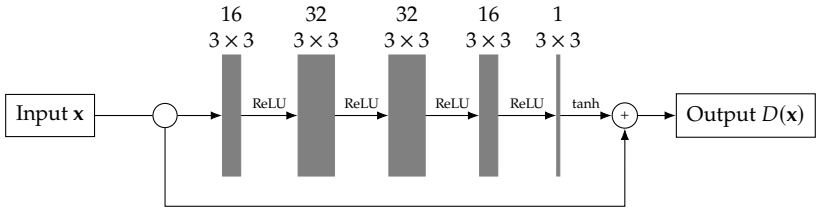


Figure 3.5: Architecture of the denoiser in Example 3.8.

Let $\mathbf{x}$ be some image, and let $\tilde{\mathbf{x}}$ be a noisy version of the same image. Let Φ be some Neural Network. We can create a denoiser D by summing the output of the Neural Network with the original image as such:

$$D(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}} + \Phi(\tilde{\mathbf{x}}) \tag{3.14}$$

We will then train the network Φ so that the difference between $D(\tilde{\mathbf{x}})$ and $\mathbf{x}$ is small. In other words, we train the network to output the negative of the noise in the image[7]. Ideally, we would like to minimize the perceptual difference between $D(\tilde{\mathbf{x}})$ and $\mathbf{x}$. However, as discussed in Section 3.3, measuring perceptual likeness is very difficult. It is very common to use the MSE as loss [MJU17, p. 91].

To frame denoising as a supervised learning problem, our true model F is a function that takes in a noisy image $\tilde{\mathbf{x}} = \mathbf{x} + \boldsymbol{\varepsilon}$ and returns the original, noiseless image $\mathbf{x}$. The denoiser D acts as the approximation model $\hat{F}$.

Example 3.8. To illustrate how a CNN can be used to remove noise, we will build a network to remove synthetic Gaussian noise from images. In our example we will use the MNIST dataset of handwritten numbers [LeC+98], consisting of 60 000 black and white 28×28 images, with pixel values in $[0, 1]$.

In order to speed up training time, we will consider a relatively small model of 5 layers. The layers consist of 16, 32, 32, 16 and 1 convolutions with 3×3 kernels. We use ReLU activation, except for in the last layer which uses tanh activation to allow negative output, and to ensure that the output is in $[-1, 1]$. The output of the network is then added to the original image as shown in Equation (3.14). The total architecture of our denoiser is shown in Figure 3.5. As a loss function we opted for the MSE between the original image and the result from the denoiser.

During training, we separated the dataset into batches of 100 images. For each batch we created noisy versions of the images by adding a random variable to each pixel of every image. The noise was drawn i.i.d. from $\mathcal{N}(0, 0.1)$ We trained the network using the Adam optimizer for 200 epochs, resulting in a total of 12 000 iterations.

The effectiveness of the resulting denoiser is shown in Figure 3.6. The images used in this test are from an independent test set of images, not used in the training. ♣

[7] Some define the denoiser as $D(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}} - \Phi(\tilde{\mathbf{x}})$ instead, and let the network learn the actual noise. However, most of the literature concerning MRI reconstruction follows the setup in Equation (3.14)

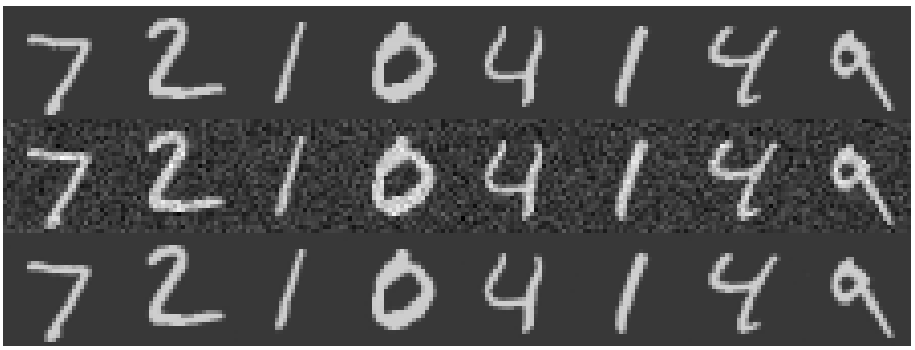


Figure 3.6: Denoising on generated Gaussian noise using the CNN described in Example 3.8 and Figure 3.5. *Top:* original images, unseen by the denoiser during training, *Middle:* images with Gaussian noise, *Bottom:* result of denoising.

MRI Reconstruction Using Deep Learning Denoising

We will now shift our focus from general neural networks and denoising to our specific application of MRI reconstruction.

Recall from the intro of Chapter 2 that one can use the adjoint of the measurement operator as a crude estimate for the inverse to create a noisy image, but usually with the details present (except for the case of uniform sampling). Thus, for some measurements $\mathbf{y}$ we find the noisy reconstruction $\tilde{\mathbf{x}}$ as

$$\tilde{\mathbf{x}} = \mathbf{A}^*\mathbf{y} = \mathbf{A}^*\mathbf{A}\mathbf{x} = \mathbf{F}_n^*\mathbf{P}_\Omega^*\mathbf{P}_\Omega\mathbf{F}_n\mathbf{x}$$

Note that $\mathbf{F}_n^* = \mathbf{F}_n^{-1}$ since $\mathbf{F}_n$ is unitary, and that $\mathbf{P}_\Omega^*\mathbf{P}_\Omega$ is given as

$$(\mathbf{P}_\Omega^*\mathbf{P}_\Omega\mathbf{y})_i = \begin{cases} y_i & \text{if } i \in \Omega \\ 0 & \text{otherwise} \end{cases}$$

So in our synthetic experiments, we find $\tilde{\mathbf{x}}$ from a training image $\mathbf{x}$ by taking the DFT of $\mathbf{x}$, zeroing out all the indices not contained in the sampling pattern Ω, and taking the IDFT.

Since MRI machines measure complex signals, the image $\mathbf{x}$, and by extension $\tilde{\mathbf{x}}$, will be complex valued. Further, since we cannot guarantee our sampling pattern to be symmetric around the origin, we could still have imaginary parts in $\tilde{\mathbf{x}}$ even though $\mathbf{x}$ was real. Thus, our denoising networks must take complex inputs. Since the support for complex numbers in many neural network frameworks is rather limited, we will some times interpret these images as two-channeled real-valued signals instead of single-channeled complex-valued signals.

Case Study: DeepMRINet

Introduced in [Sch+18], the DeepMRINet makes two important contributions to the field. Namely, using a *cascade of CNNs* to de-alias the image instead of a singe CNN, and the introduction of the Data Consistency layer (DC).

The idea behind the DC layer is that we know the true value for the measurements included in our sampling pattern Ω, and we only need to

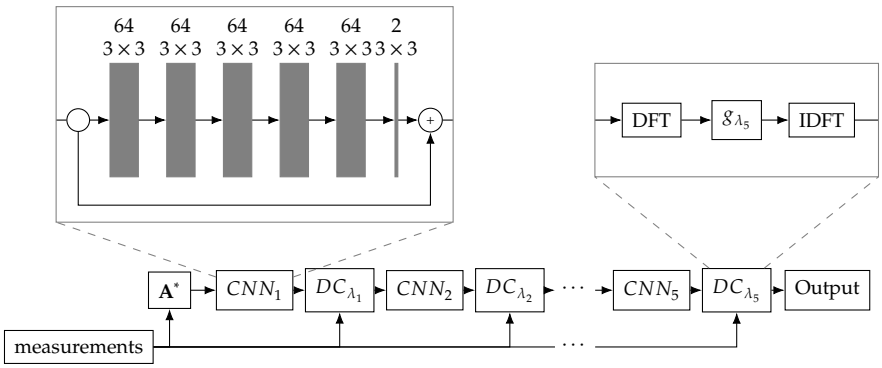


Figure 3.7: The DeepMRINet architecture.

reconstruct the unknown frequencies. Hence, we will include a layer in our architecture that will pull back the values for the known indices to the corresponding frequency by performing a convex combination between the two. Thus, for an input $\mathbf{a}$, the DC layer becomes

$$\mathrm{DC}_\lambda(\mathbf{a}) = \mathrm{IDFT}(g_\lambda(\mathrm{DFT}(\mathbf{a}) \mid \mathbf{y}, \Omega)) \tag{3.15}$$

Where DFT and IDFT denotes the Discrete Fourier Transform and the Inverse Discrete Fourier Transform respectively, and g_λ is given as

$$g_\lambda(\mathbf{z} \mid \mathbf{y}, \Omega) = \begin{cases} z_k & \text{if } k \notin \Omega \\ \frac{z_k + \lambda y_k}{1+\lambda} & \text{if } k \in \Omega \end{cases} \tag{3.16}$$

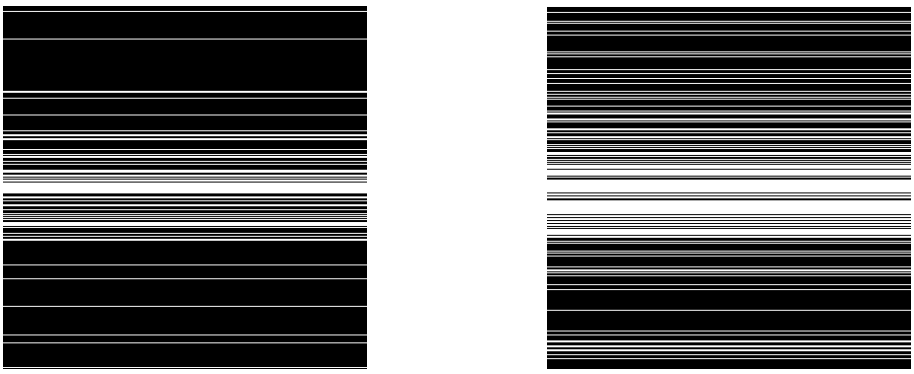
Here, $\mathbf{z}$ depicts the reconstructed image in the frequency domain, and $\mathbf{y}$ depicts the measurements as obtained by the sampling operator $\mathbf{A}$. The λ is interpreted as the amount we pull back the reconstructed measurements to the original. It can be set in one of two ways. Either as a parameter of the network, meaning a variable the optimization algorithm can change during training, or as a hyper-parameter, meaning a set variable that cannot change. The original paper [Sch+18] does not discuss the choice of λ[8]. When implementing the DeepMRINet in Chapters 4 and 6, we will treat λ as a hyper-parameter, and include a possibility for a different λ for each DC layer.

The full architecture of the DeepMRINet is found in Figure 3.7. The total network as described in [Sch+18] consists of 5 denoising CNNs following the residual setup in Equation (3.14), interlaced with DC layers. Each CNN consists of 5 convolutional layers, each performing 64 convolutions with 3×3-filters, all of them followed by ReLUs. The input and output of each CNN is a 2-channeled image, where the first channel depicts the real part and the second channel depicts the imaginary part of the image.

The original DeepMRINet were trained on the dataset from [Cab+14]. It consists of 30 images of 10 patients[9] yielding a total of 300 images. From these

[8] However, by examining the published code at the author's GitHub page it is clear that λ is chosen to be ∞

[9] The original dataset really had only one complete short-axis cardiac cine scan per patient, which was later split into layers, yielding 30 individual 2D images.



(a) 17% **(b)** 33%

Figure 3.8: Example sampling patterns used in [Sch+18].

fully-sampled images, synthetic MRI measurements were made by Fourier transforming the images and subsampling the resulting frequencies. The sampling patterns were created by randomly selecting rows from the Fourier spectrum with decreasing probability further away from the origin. The sampling patterns were redrawn during training. An example sampling pattern is shown in Figure 3.8.

Due to the limited amount of training data, different rigid transformations were used as data augmentation. The dataset was not split into a dedicated training, validation and test set. Instead, the authors used 2-fold cross validation[10] during training to synthesize a test loss.

The authors of [Sch+18] provides example recoveries and compare their method to a combination of Compressive Sensing and Dictionary Learning. An example recovery of the original DeepMRINet is shown in Figures 4.2a and 4.2b.

Because of the remarkable performance reported in [Sch+18], the rest of this thesis will focus mainly on the DeepMRINet. However, the techniques developed in Chapter 5 can be applied to any Neural Network.

Case Study: DAGAN

The Deep De-Aliasing Generative Adversarial Network (DAGAN) was introduced in [Yan+18]. The structure of DAGAN is based on a Generative Adversarial Network (GAN) where the total network is separated into a *generator* and a *discriminator*. For our purposes the generator occupies the role of the denoiser, while the discriminator is part of the loss function. We will discuss the loss function used in DAGAN later.

The architecture of the generator is largely based on that of the U-Net [RFB15]. This architecture is characterized by a series of strided convolutional layers in the first half of the network, and strided *transposed convolutions*[11], thus ending up with the same height and width dimensions in the first and last

[10] For an in-depth exploration of this technique we refer to [ISL, Section 5.1].

[11] Also known as *upconvolutions*. Several papers, including [Yan+18], refer to this as *deconvolutions*.

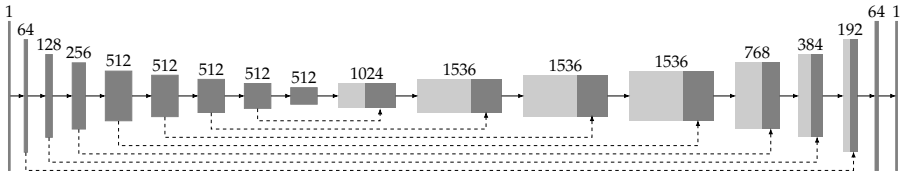


Figure 3.9: The architecture of the generator in DAGAN.

layers. The U-net also uses so-called *skip-connections*. Unlike the residual blocks outlined in Equation (3.9) where we added the identity, U-net and DAGAN *concatenates* the identity in the channel-dimension. The results from layer 7 is concatenated at the end of layer 9, as these have the same height and width. The results from layer 6 is concatenated at the end of layer 10, and so on. These skip-connections are shown with dashed lines in Figure 3.9.

DAGAN projects the image from $\mathbb{C}^n$ to $\mathbb{R}^n$ before applying the denoiser. Thus, the input layer to DAGAN is a single-channeled gray-scale image. The first 8 layers of the generator uses 4×4 convolutions with stride $s_x = s_y = 2$, meaning the sliding window of the convolution moves 2 steps in each direction instead of 1. Layers 1 through 8 uses the Leaky ReLU with a slope parameter of 0.2. Layers 9 through 16 uses 4×4 transposed convolutions with stride $s_x = s_y = 2$, thus undoing the downsizing resulting from the striding in the first half. These layers uses ReLU as the non-linearity. The last layer is a 1×1-convolution with no strides and tanh activation, which effectively turns it into a mapping from 64 to 1 channels, and scaling the output of the network to be in $[-1, 1]$.

The output of the generator is then summed with the input image as shown in Equation (3.14), producing the final output.

The loss function used in DAGAN is a sum of four terms. For a single image $\mathbf{x}$ and the measured $\mathbf{y} = \mathbf{A}\mathbf{x}$, the *pixel loss* of a proposed set of parameters θ is defined as the normal MSE:

$$\mathcal{L}_{\text{iMSE}}(\theta) = \frac{1}{2}\|\mathbf{x} - G_\theta(\mathbf{A}^*\mathbf{y})\|_2^2 \qquad \text{([Yan+18, Eq. (9)])}$$

Here, G_θ refers to the generator using θ as parameters. The authors of [Yan+18] also introduce the *frequency loss*:

$$\mathcal{L}_{\text{fMSE}}(\theta) = \frac{1}{2}\|\text{DFT}(\mathbf{x}) - \text{DFT}(G_\theta(\mathbf{A}^*\mathbf{y}))\|_2^2 \qquad \text{([Yan+18, Eq. (10)])}$$

which is simply the MSE in the frequency domain. The authors also use the first few layers of a pre-trained version of the VGG classification network [SZ14] to construct a so-called *perceptual loss*:

$$\mathcal{L}_{\text{VGG}}(\theta) = \frac{1}{2}\|f_{\text{VGG}}(\mathbf{x}) - f_{\text{VGG}}(G_\theta(\mathbf{A}^*\mathbf{y}))\|_2^2 \qquad \text{([Yan+18, Eq. (11)])}$$

The idea being that if the VGG classificator predicts different labels for the original image and the measured and reconstructed, they do not look much

However, in signal processing, a deconvolution has a very specific definition of *undoing* a convolution [Fer+10], which is not the same as applying a transposed stride.

alike. DAGAN do not use the full classificator, but rather measure the MSE between the activations in the fourth convolutional layer.

Finally, [Yan+18] introduce the *adversarial loss* created by the discriminator, which is a neural network in itself trained to predict whether a given image is a reconstruction of an undersampled signal, or a fully sampled one. If we can fool this discriminator to give a reconstruction of an undersampled image a high probability of being fully sampled, that would be an indication of a working reconstruction scheme. Thus, the adversarial loss is defined as

$$\mathcal{L}_{\text{GEN}}(\theta) = -\log(D_\theta(G_\theta(\mathbf{A}^*\mathbf{y}))) \qquad \text{([Yan+18, Eq. (12)])}$$

The total loss is then a weighted sum of the four different losses:

$$\mathcal{L}_{\text{total}}(\theta) = \alpha\mathcal{L}_{\text{iMSE}}(\theta)+\beta\mathcal{L}_{\text{fMSE}}(\theta)+\gamma\mathcal{L}_{\text{VGG}}(\theta)+\mathcal{L}_{\text{GEN}}(\theta) \quad \text{([Yan+18, Eq. (13)])}$$

The suggested parameters in the original paper are $\alpha = 15$, $\beta = 0.1$ and $\gamma = 0.0025$ [Yan+18, Fig. 11].

Remark 3.9. Since the discrete Fourier transform is a unitary operator, we have that

$$\begin{aligned}
\mathcal{L}_{\text{fMSE}}(\theta) &= \frac{1}{2}\|\text{DFT}(\mathbf{x}) - \text{DFT}(G_\theta(\mathbf{A}^*\mathbf{y}))\|_2^2 \\
&= \frac{1}{2}\|\text{DFT}(\mathbf{x} - G_\theta(\mathbf{A}^*\mathbf{y}))\|_2^2 \\
&= \frac{1}{2}\|\mathbf{x} - G_\theta(\mathbf{A}^*\mathbf{y})\|_2^2 \\
&= \mathcal{L}_{\text{iMSE}}(\theta)
\end{aligned}$$

Which means that the *"pixel loss"* and *"frequency loss"* are really the same, even though [Yan+18] presents them as different.

CHAPTER 4

Stability

In this chapter we will begin by introducing an important concept regarding numerical stability, namely Lipschitz constants, and give some initial results which will be useful during our derivation of Parseval networks in Chapter 5. Later, in Section 4.2, we will introduce so-called *adversarial attacks* as a way to systematically create small perturbations where modern neural networks for MRI reconstruction will fail.

4.1 Numerical Stability

We begin our study of stability by specifying what we mean with a *stable recovery*. First, we introduce the notion of Lipschitz constants.

Definition 4.1 (Lipschitz constants)**.** Given two normed spaces $(A, \|\cdot\|_A)$ and $(B, \|\cdot\|_B)$ and a function $f : A \to B$. $L \in \mathbb{R}$ is said to be *a Lipschitz constant* of f if

$$\|f(x) - f(y)\|_B \leq L\|x - y\|_A \quad \text{for all } x, y \in A$$

The smallest possible L to satisfy the above equation is said to be *the Lipschitz constant* of f.

The Lipschitz constant of a function gives a bound of how much an error in the input to a function can change the outcome. Consider a function $\hat{F} : \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times n}$ which estimates the inverse of a measurement operator, and let L be a Lipschitz constant of $\hat{F}$. If we perturb the input image to $\hat{F}$ by a small perturbation $\boldsymbol{\delta}$ with $\|\boldsymbol{\delta}\|_2 < \varepsilon$, we now have a bound on how much this will affect the reconstruction:

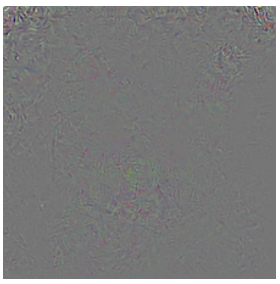
$$\|\hat{F}(\mathbf{x} + \boldsymbol{\delta}) - \hat{F}(\mathbf{x})\|_2 \leq L\|\mathbf{x} + \boldsymbol{\delta} - \mathbf{x}\|_2 = L\|\boldsymbol{\delta}\|_2 < L\varepsilon \tag{4.1}$$

If the Lipschitz constant of a function is ≤ 1, then we know that the function cannot amplify errors. That is, if the perturbation on the input has norm ε, the on the error on the output (compared to the unperturbed input) is $\leq \varepsilon$ for any input. We call such functions *contractions*.

Since Neural Networks are composite functions of the layers, the following two results will become useful later.

(a) Original image, correctly classified as a *tiger cat*

(b) Perturbed image, misclassified as an *American bald eagle*



(c) Added perturbation, magnified to be visible

Figure 4.1: Example of an adversarial attack against a classification network. Here we are using the ResNet-50 network [He+16] trained on the ImageNet dataset [Rus+15]. See page 81 for details.

Proposition 4.2. *Let $(X, \|\cdot\|)$ be a normed space, and let $f\colon X \to X$ and $g\colon X \to X$ be two functions. Define $h\colon X \to X$ as the composition of f and g, ie $h = f \circ g$. If L_f is a Lipschitz constant of f and L_g is a Lipschitz constant of g, then $L_f L_g$ is a Lipschitz constant of h.*

Proof. Let $x, y \in X$, then

$$\|h(x) - h(y)\| = \|f(g(x)) - f(g(y))\| \leq L_f \|g(x) - g(y)\| \leq L_f L_g \|x - y\|$$

which shows that $L_f L_g$ is a Lipschitz constant of h. □

The next corollary follows immediately by induction on n.

Corollary 4.3. *Let $(X, \|\cdot\|)$ be a normed space, and let $f_1, f_2, \ldots, f_n\colon X \to X$ be functions with respective Lipschitz constants $L_1, L_2, \ldots, L_n$. Define $g\colon X \to X$ as $g = f_1 \circ f_2 \circ \cdots \circ f_n$. Then, $\prod_{i=1}^{n} L_i$ is a Lipschitz constant for g.*

Throughout the rest of the thesis, we will consider Lipschitz constants in $\mathbb{R}^n$ with regards to the ℓ_2 norm.

4.2 Adversarial Attacks

It is well established that classifiers using neural networks are very vulnerable to a certain kind of perturbations to the input [Big+13; Sze+13; MFF16; FMF17]. Given an image $\mathbf{x} \in \mathbb{R}^n$ with label $c \in \{1, 2, \ldots, C\}$ and a neural network based classifier $f\colon \mathbb{R}^n \to \{1, 2, \ldots, C\}$, one can often find a perturbation $\boldsymbol{\delta}$ with a small $\|\boldsymbol{\delta}\|_2$ such that $f(\mathbf{x}) = c$ while $f(\mathbf{x} + \boldsymbol{\delta}) \neq c$, even though $\mathbf{x}$ and $\mathbf{x} + \boldsymbol{\delta}$ might be indistinguishable by the human eye, see Figure 4.1.

Earlier work have developed algorithms to systematically create such noise for classification networks [MFF16]. Constructing these perturbations is often

Algorithm 4.1 Finding adversarial noise for inverse problems.

Input: Image $\mathbf{x}$, Neural Network Φ, measurement operator $\mathbf{A}$, maximum number of iterations N, step length η, perturbation size regularization parameter λ, initial perturbation size τ.

1. Initialize $\mathbf{v}_0 \leftarrow \mathbf{0}$ and $\boldsymbol{\delta}_0 \in B(\mathbf{0}, \tau)$
2. For $i = 0, 1, \ldots, N-1$, do
   2.1 $\mathbf{v}_{i+1} \leftarrow \gamma \mathbf{v}_i + \eta \nabla Q(\boldsymbol{\delta}_i)$
   2.2 $\boldsymbol{\delta}_{i+1} \leftarrow \boldsymbol{\delta}_i + \mathbf{v}_{i+1}$

Output: $\boldsymbol{\delta}_N$

called *adversarial attacks*, or in the concrete case for classification it is also known as *fooling*. In this section we will present a way to create adversarial attacks for denoisers, introduced in [Ant+19].

Generating Adversarial Noise

We begin by looking at why we can not use the approaches from classification fooling in inverse problems. Given a classifier $\hat{F}\colon \mathbb{R}^n \to \{1, 2, \ldots, C\}$ and an image $\mathbf{x}$ which $\hat{F}$ will normally classify correctly, it is clear that the optimal perturbation $\boldsymbol{\delta}'$ for fooling $\hat{F}$ on $\mathbf{x}$ is

$$\boldsymbol{\delta}' = \arg\min_{\boldsymbol{\delta}} \|\boldsymbol{\delta}\|_2 \quad \text{subject to } \hat{F}(\mathbf{x} + \boldsymbol{\delta}) \neq \hat{F}(\mathbf{x}) \tag{4.2}$$

The benefit of working with classifiers in this case is that we have a clear indication of success – namely misclassification. However, solving Equation (4.2) in practice is intractable because of the non-linear constraint.

While working with inverse problems, declaring success is not that straight forward since "successful recovery" is a somewhat subjective term. As discussed in Section 3.3 it is also very difficult to measure mathematically whether two images look alike.

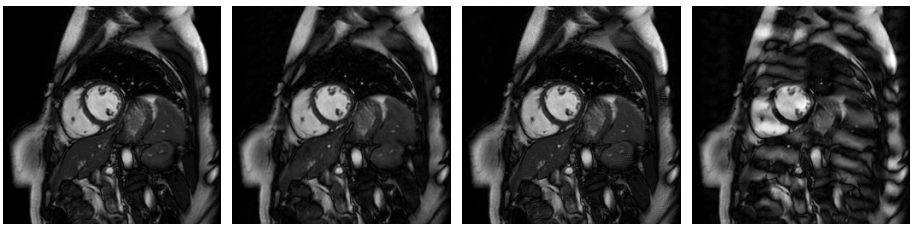
In [Ant+19], the authors propose a way to adapt Equation (4.2) to inverse problems. Instead of constraining the output to be different, we maximize the difference between the reconstruction of the original image and the perturbed image. We also constrain the size of the perturbation using a quadratic regularization term. Hence, given a denoiser D and an input image $\mathbf{x}$ which D can successfully recover, the optimal perturbation is given as

$$\boldsymbol{\delta}' = \arg\max_{\boldsymbol{\delta}} \frac{1}{2}\|D(\mathbf{A}(\mathbf{x} + \boldsymbol{\delta})) - D(\mathbf{A}\mathbf{x})\|_2^2 - \frac{\lambda}{2}\|\boldsymbol{\delta}\|_2^2 \tag{4.3}$$

The λ acts as a regularization parameter on the perturbation, and controls how large the perturbation can be.

However, solving Equation (4.3) in practice is highly intractable. Again, we will have to settle for a local minimum. First, we define our objective function:

$$Q(\boldsymbol{\delta}) = \frac{1}{2}\|D(\mathbf{A}(\mathbf{x} + \boldsymbol{\delta})) - D(\mathbf{A}\mathbf{x})\|_2^2 - \frac{\lambda}{2}\|\boldsymbol{\delta}\|_2^2 \tag{4.4}$$



(a) Original $\mathbf{x}$ **(b)** Reconstruction $D(\mathbf{A}\mathbf{x})$ **(c)** Perturbed $\mathbf{x} + \boldsymbol{\delta}$ **(d)** Reconstruction $D(\mathbf{A}(\mathbf{x} + \boldsymbol{\delta}))$

Figure 4.2: Instabilities in the original DeepMRINet as shown in [Ant+19]. This version of DeepMRINet samples at 33% with a sampling pattern similar to Figure 3.8b. Images from [Ant+19], used with permission.

Let $\mathbf{y} = \mathbf{A}\mathbf{x}$ and $\mathbf{u} = \mathbf{y} + \mathbf{A}\boldsymbol{\delta}$. Using the chain rule, the gradient of Q can be expressed as

$$\nabla_{\boldsymbol{\delta}} Q(\boldsymbol{\delta}) = \mathbf{A}^* \nabla_{\mathbf{u}} \|D(\mathbf{u}) - D(\mathbf{y})\|_2^2 - \lambda \boldsymbol{\delta} \tag{4.5}$$

We can interpret $D(\mathbf{y})$ as a constant as it is not dependent on $\mathbf{u}$ or $\boldsymbol{\delta}$, thus we only need to find the gradient for $D(\mathbf{u})$, which can be done with back-propagation as D is a neural network.

The final algorithm for generating adversarial perturbations is then to combine Equation (4.5) with some variant of Gradient Descent to achieve a local minimum of Equation (4.4). The authors of [Ant+19] propose using Gradient Ascent with Nesterov momentum. A detailed description of the algorithm presented in [Ant+19] is found in Algorithm 4.1.

The authors of [Ant+19] provides example perturbations for several modern MRI reconstruction networks, among them the DeepMRI network. See Figure 4.2 for an example of an adversarial attack on DeepMRINet with the original weights provided by [Sch+18].

Current Solutions

One possible cause of instability is overfitting. Recall the overfitted example in Figure 3.4 on page 28. By only nudging the value slightly in the horizontal direction, we see that the value in the vertical direction can vary dramatically in an unexpected way. Thus, some of the current techniques to limit instabilities address limiting overfitting, while others address the issue of adversarial attacks directly.

Some work have gone into the robustness of classifiers towards adversarial attacks, and we will conclude this chapter with a brief presentation on some of them [FMF17]. In Chapter 5 we will introduce a newer approach in more detail [Cis+17].

An initial approach is to augment the dataset with adversarial examples. In [MFF16] the authors propose training a classifier on a normal dataset, and when the classifier is done training, one creates a new training set consisting of adversarial attacks on the trained model. The model is then further trained on this new data set with a decreased step length for a few epochs.

Another approach is to consider the Jacobian matrix of a network Φ_θ, that is

$$J_{\mathbf{x}}(\Phi_\theta) = \frac{\partial \mathbf{a}_L}{\partial \mathbf{x}}$$

where $\mathbf{a}_L$ denote the activations in the last layer and $\mathbf{x}$ denotes the input to the network. Large elements in this matrix are undesirable as it means that the output of the network can change drastically on small changes to the input. We can thus add the norm of the Jacobian as a regularization term to our loss function [FMF17; GR14]. This approach is analogous to the regularization term discussed in Equation (3.13) on page 29.

Other forms of regularization can also help prevent overfitting, which can in turn reduce instabilities. By adding some scaled norm of the weight matrices as a regularization term we limit how large the weights can be. The motivation for this is to not let a single feature become too important in the classification [ISL, Sec. 6.2; GBC16, Sec. 7.1].

Any type of regularization is designed reduce the generality of the network. This might introduce some bias to the model, but can in turn reduce the variance. This trade-off, often called the bias-variance trade-off, is a well studied topic in statistics [ISL, Sec. 2.2.2]. It states that the MSE of a model can be decomposed into three parts, namely model variance, model bias and observation variance. Since we cannot control the observation variance, we are most interested in the model variance and bias.

Recall the underlying assumption for supervised learning from Equation (3.1) on page 17. Given an observation $(\mathbf{x}, \mathbf{y})$ from a true underlying model $F(\mathbf{x}) = \mathbf{y} + \varepsilon$, $\varepsilon \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, we can express the *expected MSE* for a model $\hat{F}$ as (adapted for notation from [ISL, Eq. (2.7)]):

$$E\left[\mathrm{MSE}(\hat{F} \mid \mathbf{x}, \mathbf{y})\right] = \underbrace{\mathrm{Var}\left[\hat{F}(\mathbf{x}_i)\right]}_{\text{Model variance}} + \underbrace{E\left[\|\hat{F}(\mathbf{x}_i) - F(\mathbf{x}_i)\|_2^2\right]}_{\text{Model bias}} + \underbrace{\sigma^2}_{\substack{\text{Observation} \\ \text{variance}}} \tag{4.6}$$

Hence, if the regularization causes the variance to drop more than the bias increases, the overall value of the loss function will decrease.

Even though these approaches can help reduce overfitting and instabilities in neural networks in some cases, the topic of stability towards adversarial attacks remain an open and unsolved problem [FMF17].

CHAPTER 5

Parseval Networks

Parseval networks were introduced in [Cis+17] as a new way to combat overfitting and instabilities in classification networks.

The main idea behind Parseval networks is to limit the Lipschitz constant of each layer to be less than 1. This is done for two main reasons. First, by making each layer a contraction, we restrict the amount a feature can impact an activation in the network, which can combat overfitting. Second, by making the entire network a contraction we limit the amount of change on the output logits of the network. Our motivation is that this may improve the network's stability against adversarial attacks.

5.1 Lipschitz Constants of Neural Networks

In order to constrain the Lipschitz constants of Neural Networks, we must first derive the Lipschitz constants of the different kinds of layers. From Corollary 4.3 we have that the Lipschitz constant of a Neural Network is bounded above by the product of the Lipschitz constants of its layers. Thus, if we limit every layer to be a contraction, the whole network will be a contraction.

Dense Layers

Recall that a dense layer is an affine mapping $W_l \colon \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$, given as

$$\mathbf{a}_l = W_l(\mathbf{a}_{l-1}) = \mathbf{W}_l\mathbf{a}_{l-1} + \mathbf{b}_l$$

We wish to limit the Lipschitz constant of this mapping. The following proposition will help us achieve this bound:

Proposition 5.1. *The Lipschitz constant of an affine mapping $W \colon \mathbb{R}^m \to \mathbb{R}^n$ given as $W(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$ is the largest singular value of $\mathbf{W}$.*

Proof. Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$, then

$$\|W(\mathbf{x}) - W(\mathbf{y})\|_2 = \|\mathbf{W}\mathbf{x} + \mathbf{b} - (\mathbf{W}\mathbf{y} + \mathbf{b})\|_2 = \|\mathbf{W}(\mathbf{x} - \mathbf{y})\|_2 \leq \|\mathbf{W}\|_2\|\mathbf{x} - \mathbf{y}\|_2$$

We know the last inequality to be sharp. Hence, we see that the Lipschitz constant of W is the operator norm $\|\mathbf{W}\|_2$. It remains to show that this norm is bounded by the largest singular value of $\mathbf{W}$.

Let $\mathbf{W} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ be the singular value decomposition of $\mathbf{W}$, so $\mathbf{\Sigma} = \mathbf{U}^T\mathbf{W}\mathbf{V}$. Let $\mathbf{x} \in \mathbb{R}^m$ with $\|\mathbf{x}\|_2 = 1$, we must show that $\|\mathbf{W}\mathbf{x}\|_2^2 \leq \sigma_1^2$. Since the columns of $\mathbf{V}$ forms an orthonormal basis for $\mathbb{R}^m$, there exists a $\mathbf{c} \in \mathbb{R}^m$ with $\|\mathbf{c}\|_2 = 1$ such that $\mathbf{x} = \mathbf{V}\mathbf{c}$. Thus:

$$\|\mathbf{W}\mathbf{x}\|_2^2 = \|\mathbf{W}\mathbf{V}\mathbf{c}\|_2^2 = \|\mathbf{U}^T\mathbf{W}\mathbf{V}\mathbf{c}\|_2^2 = \|\mathbf{\Sigma}\mathbf{c}\|_2^2 = \sum_{i=1}^{n} \sigma_i^2 |c_i|^2 \leq \sigma_1^2 \sum_{i=1}^{n} |c_i|^2 = \sigma_1^2$$

Which concludes the proof. □

Convolutional Layers

To derive Lipschitz constants for convolutional layers, we will begin by showing how we can express convolutional layers as affine mappings. In traditional signal processing theory, convolutions are expressed with circulant Toepliz matrices [Rya19, p. 99]. However, recall from Remark 3.7 that convolutional layers do not comprise actual convolutions. Hence, the traditional theory does not apply directly.

We will begin by presenting how [Cis+17] solves this problem for unstrided, 1D convolutional layers, and working out some important details omitted from the original paper. Then we will generalize their result to cover strided convolutions and higher dimensional input tensors as well.

First of all, note that we can ignore the batch dimension b in Equation (3.8) without loss of generality. From our perspective the batch dimension is merely to do the same set of operations on different input. To further simplify notation, assume that the length of the convolution kernel is odd, that is the kernel size is $2d + 1$ for some $d \in \mathbb{N}$. Throughout this section, we will rearrange the order of the dimensions of the input so that the channels is first instead of last (contrary to the subsection at page 21). This will not alter the end results, but make some of the matrix expressions later on tidier.

Hence, we have our input $\mathbf{a} \in \mathbb{R}^{c_{\text{in}} \times n}$ (input channels, length) instead of $\mathbf{a} \in \mathbb{R}^{b \times n \times c_{\text{in}}}$ (batch, length, input channels). Our weight tensor is still $\mathbf{w} \in \mathbb{R}^{(2d+1) \times c_{\text{in}} \times c_{\text{out}}}$ (kernel size, input channels, output channels), with the filter kernels indexed from $-d$ to d. One-dimensional, unstrided convolutions without the batch dimension is then given as

$$(\mathbf{w} * \mathbf{a})_{k_2,i} = \sum_{i'} \sum_{k_1} w_{i',k_1,k_2} \cdot a_{k_1,i+i'} \tag{5.1}$$

For an input activation $\mathbf{a}$ to a layer, let the *unfolding operator* applied to our input signal $\mathbf{U}(\mathbf{a})$ be the $(2d+1)c_{\text{in}} \times n$ matrix where the jth column is given as

$$\mathbf{U}(\mathbf{a})_{*,j} = \begin{bmatrix} a_{1,j-d} \\ a_{2,j-d} \\ \vdots \\ a_{c_{\text{in}},j-d} \\ a_{1,j-d+1} \\ \vdots \\ a_{c_{\text{in}},j+d} \end{bmatrix}$$

We define $a_{i,j} = 0$ whenever i or j is out of bounds[1]. Note that $\mathbf{U}(\cdot)$ is clearly a linear operator. Further, let the *unfolded weight matrix* $\mathbf{W}$ be the $c_{\text{out}} \times (2d+1)c_{\text{in}}$ matrix where row i is given as

$$\mathbf{W}_{i,*} = \begin{bmatrix} w_{-d,1,i} & w_{-d,2,i} & \cdots & w_{-d,c_{\text{in}},i} & w_{-d+1,1,i} & \cdots & w_{d,c_{\text{in}},i} \end{bmatrix}$$

Note that $\mathbf{W}$ is merely a reshaping of original weight tensor $\mathbf{w}$.

We can now rewrite Equation (5.1) on matrix form as

$$(\mathbf{w} * \mathbf{a}) = \mathbf{W}\mathbf{U}(\mathbf{a}) \tag{5.2}$$

Thus, a convolutional layer is given as

$$\mathbf{a}_l = \mathbf{W}_l \mathbf{U}(\mathbf{a}_{l-1}) + \mathbf{b}_l$$

From the first half of the proof of Proposition 5.1 we have that the Lipschitz constant of a convolutional layer is the operator norm of the composite $\|\mathbf{W}\mathbf{U}\|_2^2$, and combined with Proposition 4.2 this gives that

$$\|\mathbf{W}\mathbf{U}\|_2^2 \leq \|\mathbf{W}\|_2^2 \|\mathbf{U}\|_2^2$$

From the second part of the proof of Proposition 5.1 we have that $\|\mathbf{W}\|_2^2$ is bounded by the largest singular value of $\mathbf{W}$. Since $\mathbf{U}$ is an operator that repeats shifted versions of the input signal $2d+1$ times, we have that

$$\|\mathbf{U}(\mathbf{x}) - \mathbf{U}(\mathbf{y})\|_2^2 = \|\mathbf{U}(\mathbf{x} - \mathbf{y})\|_2^2 \leq (2d+1)\|\mathbf{x} - \mathbf{y}\|_2^2$$

Giving that $\sqrt{2d+1}$ is a Lipschitz constant for $\mathbf{U}$.

We summarize the above in the following proposition:

Proposition 5.2. *The Lipschitz constant of a convolutional layer $W : \mathbb{R}^{n \times c_{in}} \to \mathbb{R}^{n \times c_{out}}$ with kernel size $2d+1$ and weight tensor $\mathbf{w}$ is bounded by $\sqrt{(2d+1)}\sigma_1$ where σ_1 denotes the largest singular value of the unfolded weight matrix $\mathbf{W}$.*

Extension to Strided Convolutions

One possible way to introduce strides to Equation (5.2) is to linearly sample the columns of the resulting matrix. However, we will use another approach were we redefine the unfolding operator, as this aligns more closely with Equation (3.8) as well as how convolutional layers are typically implemented in practice.

[1] Often dubbed zero-padding in DL literature.

The unfolding operator for a 1D convolutional layer using stride s will be the $(2d+1)c_{\text{in}} \times n/s$ matrix where column j is given as

$$\mathbf{U}(\mathbf{a})_{*,j} = \begin{bmatrix} a_{1,j-ds} \\ a_{2,j-ds} \\ \vdots \\ a_{c_{\text{in}},j-ds} \\ a_{1,j-(d-1)s} \\ \vdots \\ a_{c_{\text{in}},j+ds} \end{bmatrix}$$

By omitting entries from the input signal, it is clear that the Lipschitz constant can not be larger than for the unstrided case.

Extension to Higher Dimensions

Extension to higher dimensions will largely follow the same strategy as the extension to strided convolutions, . We will only show how to extend Equation (5.2) to 2D convolutions, but the following technique can be repeated to achieve convolutions of even higher dimensional signals. However, since we will only deal with 2D images in this thesis, 2D convolutional layers will suffice.

First, note that the input to a 1D convolutional layer is a 2D tensor $\mathbf{a} \in \mathbb{R}^{c_{\text{in}} \times n}$. Thus, the input to 2D convolutional layer is a 3D tensor $\mathbf{a} \in \mathbb{R}^{c_{\text{in}} \times m \times n}$. To simplify notation, assume that the kernel is square and of odd length, that is the kernel size is $(2d+1) \times (2d+1)$ for a $k \in \mathbb{N}$. Our filter tensor is now a 4-dimensional tensor $\mathbf{w} \in \mathbb{R}^{(2d+1)\times(2d+1)\times c_{\text{in}} \times c_{\text{out}}}$

Intuitively, our approach is to stack the columns in the image vertically, thus turning the signal into a 2D tensor. This will however demand some restructuring of both the unfolding operator and the unfolded weight matrix.

As before, we can ignore the batch dimension without loss of generality. Thus, for an input activation $\mathbf{a}$, we can express the convolution as

$$(\mathbf{w} * \mathbf{a})_{k_2,i,j} = \sum_{i'} \sum_{j'} \sum_{k_1} w_{i',j',k_1,k_2} \cdot a_{k_1,i+i',j+j'} \tag{5.3}$$

To simplify further notation, let $\mathbf{a}_{i,j} \in \mathbb{R}^{c_{\text{in}}}$ be the vector given by stacking the channel dimension as a column vector:

$$\mathbf{a}_{i,j} = \begin{bmatrix} a_{1,i,j} \\ a_{2,i,j} \\ \vdots \\ a_{c_{\text{in}},i,j} \end{bmatrix}$$

To express (5.3) on the same form as Equation (5.2), we will redefine the unfolding operator applied to our input signal $\mathbf{U}(\mathbf{a})$ to be the $(2d+1)^2 c_{\text{in}} \times mn$

matrix where the jth column is given as

$$\mathbf{U}(\mathbf{a})_{*,j} = \begin{bmatrix} \mathbf{a}_{\lfloor j/n \rfloor - d,\ (j \bmod n) - d} \\ \mathbf{a}_{\lfloor j/n \rfloor - d,\ (j \bmod n) - d + 1} \\ \vdots \\ \mathbf{a}_{\lfloor j/n \rfloor - d,\ (j \bmod n) + d} \\ \mathbf{a}_{\lfloor j/n \rfloor - d + 1,\ (j \bmod n) - d} \\ \vdots \\ \mathbf{a}_{\lfloor j/n \rfloor + d,\ (j \bmod n) + d} \end{bmatrix}$$

Likewise, let $\mathbf{w}_{i,j,k_2}^T \in \mathbb{R}^{c_{\text{in}}}$ be the column vector obtained by stacking the input channel dimensions, so that $\mathbf{w}$ is a row vector:

$$\mathbf{w}_{i,j,k_2} = \begin{bmatrix} w_{i,j,1,k_2} & w_{i,j,2,k_2} & \cdots & w_{i,j,c_{\text{in}},k_2} \end{bmatrix}$$

We redefine the unfolded weight matrix as the $c_{\text{out}} \times (2d+1)^2 c_{\text{in}}$ matrix where row i is given as

$$\mathbf{W}_{i,*} = \begin{bmatrix} \mathbf{w}_{-d,-d,i} & \mathbf{w}_{-d,-d+1,i} & \cdots & \mathbf{w}_{-d,d,i} & \mathbf{w}_{-d+1,-d,i} & \cdots & \mathbf{w}_{d,d,i} \end{bmatrix}$$

Note that $\mathbf{W}$ is still just a reshaping of $\mathbf{w}$. Thus, we can rewrite Equation (5.3) as

$$(\mathbf{w} * \mathbf{a}) = \mathbf{W}\mathbf{U}(\mathbf{a}) \tag{5.4}$$

using these redefined versions of the unfolding operator and unfolded weight matrix. We can limit $\|\mathbf{W}\|_2$ in the same manner as before. The unfolding operator now repeats the signal $(2d+1)^2$ times, and following the argument for the 1D case, $2d+1$ is a Lipschitz constant for $\mathbf{U}$.

Residual Blocks

Residual blocks pose a problem with Lipschitz constants. Recall that in a residual block the identity is fed through several layers, as shown in Equation (3.9) and Figure 3.2 on page 22 and on page 23. The problem with residual blocks is that the sum of two contractions is not necessarily a contraction. As a simple counterexample, consider $f : \mathbb{R} \to \mathbb{R}$, with $f(x) = x$. While f is clearly a contraction, $g = f + f$ is not.

Thus, even though we restrict every layer in the residual block to be a contraction, the block itself will not be a contraction. The solution to this problem is motivated by the following proposition:

Proposition 5.3. *Let $f, g : \mathbb{R}^m \to \mathbb{R}^n$ be two contractions. Any convex combination of f and g is also a contraction.*

Proof. Let $f, g : \mathbb{R}^m \to \mathbb{R}^n$ be two contractions, let $\lambda \in [0,1]$, and define $h : \mathbb{R}^m \to \mathbb{R}^n$ as

$$h(\mathbf{x}) = \lambda f(\mathbf{x}) + (1-\lambda) g(\mathbf{x})$$

Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$, then

$$\begin{aligned}
&\|h(\mathbf{x}) - h(\mathbf{y})\| \\
&= \|\lambda f(\mathbf{x}) + (1-\lambda)g(\mathbf{x}) - \lambda f(\mathbf{y}) - (1-\lambda)g(\mathbf{y})\| \\
&= \|\lambda(f(\mathbf{x}) - f(\mathbf{y})) + (1-\lambda)(g(\mathbf{x}) - g(\mathbf{y}))\| \\
&\leq \lambda\|f(\mathbf{x}) - f(\mathbf{y})\| + (1-\lambda)\|g(\mathbf{x}) - g(\mathbf{y})\| \\
&\leq \lambda\|\mathbf{x} - \mathbf{y}\| + (1-\lambda)\|\mathbf{x} - \mathbf{y}\| \\
&= \|\mathbf{x} - \mathbf{y}\|
\end{aligned}$$

Which shows that h is a contraction. □

In Parseval networks, we will redefine residual blocks to use convex combinations instead of sums, thus ensuring the residual block to be a contraction. An example 3-layered Parseval residual block is then given as

$$B(\mathbf{a}) = \rho_3(\lambda W_3(\rho_2(W_2(\rho_1(W_1(\mathbf{a})))))) + (1-\lambda)\mathbf{a}) \tag{5.5}$$

The parameter λ can either be interpreted as a fixed hyper-parameter, or as a trainable parameter.

Activation Functions

The following lemma will help us find upper bounds of Lipschitz constants for activation functions.

Lemma 5.4. *Let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function. Let $x, y \in \mathbb{R}$. If $d_{\max} = \max\left\{\left|f'(z)\right| \mid z \in \mathbb{R}\right\}$ exits, then*

$$\left|f(x) - f(y)\right| \leq d_{\max}\left|x - y\right|$$

In other words, $d_{\max}$ is a Lipschitz constant of f.

Proof. Let $d_{\max}$ be defined as above. Assume for contradiction that there exists a pair $x, y \in \mathbb{R}$ with $x \leq y$ such that

$$\left|f(x) - f(y)\right| > d_{\max}\left|x - y\right|$$

By the *Mean Value Theorem* we know that there exists a point $z \in \mathbb{R}$ such that

$$d_{\max} < \frac{\left|f(x) - f(y)\right|}{\left|x - y\right|} = \left|f'(z)\right|$$

This is a contradiction, since $d_{\max}$ is the defined to be the largest derivative of f. □

We will use this lemma to prove the following proposition.

Proposition 5.5. *ReLU, Leaky ReLU with slope ≤ 1, hyperbolic tangent and sigmoid are all contractions.*

Proof. Let $x, y \in \mathbb{R}$. If both $x, y \leq 0$, $\mathrm{ReLU}(x) = \mathrm{ReLU}(y) = 0$. Thus it is clear that 0 is a Lipschitz constant of ReLU restricted to $(-\infty, 0)$. If both $x, y > 0$, Lemma 5.4 gives that 1 is a Lipschitz constant of ReLU restricted to $(0, \infty)$. If $x < 0$ and $y > 0$, we have that

$$\left|\mathrm{ReLU}(y) - \mathrm{ReLU}(x)\right| = \left|\mathrm{ReLU}(y)\right| = y \leq \left|y - 0\right| = \left|y - x\right|$$

Thus, $\max\{0, 1\} = 1$ is a Lipschitz constant for the ReLU, so it is a contraction.

Likewise, for the leaky ReLU Lemma 5.4 gives that α is a Lipschitz constant of LReLU_α restricted to $(-\infty, 0)$ and that 1 is a Lipschitz constant of LReLU_α restricted to $(0, \infty)$. Following the same argument as above, $\max\{\alpha, 1\}$ is a Lipschitz constant of LReLU_α. Since we have assumed $\alpha \leq 1$, LReLU is a contraction.

For tanh, we observe the derivative

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

Since $\tanh^2(x) \geq 0$ for all x with $\tanh(0) = 0$, we have that

$$\max_{x \in \mathbb{R}} \frac{d}{dx}\tanh(x) = 1$$

And by Lemma 5.4, 1 is a Lipschitz constant of tanh, so it is a contraction.

Finally, for the sigmoid, we find the derivative:

$$\sigma'(x) = \frac{d}{dx}\left(\frac{1}{1 + e^{-x}}\right) = \frac{e^{-x}}{\left(1 + e^{-x}\right)^2} \tag{5.6}$$

and the second derivative:

$$\frac{d^2}{dx^2}\left(\frac{1}{1 + e^{-x}}\right) = \frac{d}{dx}\left(\frac{e^{-x}}{\left(1 + e^{-x}\right)^2}\right) = \frac{2e^{-2x}}{(1 + e^{-x})^3} - \frac{e^{-x}}{(1 + e^{-x})^2}$$

which has exactly one root, namely $x = 0$. Since $\sigma'(0) = 0.25$ and

$$\lim_{x \to \infty} \sigma'(x) = \lim_{x \to -\infty} \sigma'(x) = 0$$

we have that $x = 0$ is a global maximum for σ' with $\sigma'(0) = 0.25$. By Lemma 5.4, 0.25 is a Lipschitz constant of σ and it is a contraction. □

5.2 Tight Frames

This section will give a brief introduction on the topic of tight frames. As we will later see, this notion, in conjunction with the results of the previous section, will give us a way to enforce the layers to be contractions.

We will begin with the general definition of tight frames:

Definition 5.6 ([FR13, Definition 5.6])**.** A system of vectors $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_m\}$ in $\mathbb{R}^n$ is called a *tight frame* if there exists a constant $\lambda > 0$ such that one of the following equivalent conditions holds:

(a) $\|\mathbf{x}\|_2^2 = \lambda \sum_{i=1}^{m} |\langle \mathbf{x}, \mathbf{e}_i \rangle|^2 \quad \text{for all } \mathbf{x} \in \mathbb{R}^n$

(b) $\mathbf{x} = \lambda \sum_{i=1}^{m} \langle \mathbf{x}, \mathbf{e}_i \rangle \mathbf{e}_i \quad \text{for all } \mathbf{x} \in \mathbb{R}^n$

(c) $\mathbf{A}\mathbf{A}^T = \frac{1}{\lambda}\mathbf{I}_n$, where $\mathbf{A}$ is the matrix with columns $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_m\}$

Proof of equivalence. To see that (a), (b) and (c) are equivalent, notice first that the right hand side of (a) can be written as

$$\lambda \mathbf{x}^T \mathbf{A}\mathbf{A}^T \mathbf{x} \tag{5.7}$$

for a matrix $\mathbf{A}$ with columns $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_m\}$. Since $\mathbf{A}\mathbf{A}^T$ is symmetric, it is uniquely characterized by the quadratic form in Equation (5.7). Thus

$$\mathbf{x}^T \mathbf{x} = \|\mathbf{x}\|_2^2 = \lambda \mathbf{x}^T \mathbf{A}\mathbf{A}^T \mathbf{x}$$

must hold for all $\mathbf{x}$, giving that $\lambda \mathbf{A}\mathbf{A}^T = \mathbf{I}$, establishing the equivalence of (a) and (c). For the equivalence of (b) and (c), we have that

$$\sum_{i=1}^{m} \langle \mathbf{x}, \mathbf{e}_i \rangle \mathbf{e}_i = \mathbf{A} \begin{bmatrix} \langle \mathbf{x}, \mathbf{e}_1 \rangle \\ \langle \mathbf{x}, \mathbf{e}_2 \rangle \\ \vdots \\ \langle \mathbf{x}, \mathbf{e}_m \rangle \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathbf{e}_1^T \mathbf{x} \\ \mathbf{e}_2^T \mathbf{x} \\ \vdots \\ \mathbf{e}_m^T \mathbf{x} \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_m^T \end{bmatrix} \mathbf{x} = \mathbf{A}\mathbf{A}^T \mathbf{x}$$

Giving that

$$\mathbf{x} = \lambda \sum_{i=1}^{m} \langle \mathbf{x}, \mathbf{e}_i \rangle \mathbf{e}_i = \lambda \mathbf{A}\mathbf{A}^T \mathbf{x}$$

for all $\mathbf{x}$, hence $\lambda \mathbf{A}\mathbf{A}^T = \mathbf{I}$. □

In this thesis, we are concerned with a special kind of tight frames, namely Parseval frames:

Definition 5.7. A tight frame with $\lambda = 1$ is called a *Parseval frame*.

We are now ready to present the main result of this section:

Theorem 5.8. *Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m < n$, be a matrix with singular values $\sigma_1, \dots, \sigma_n$. If the the rows of $\mathbf{A}$ form a Parseval frame, the singular values are all* 1*.*

Proof. Let $\mathbf{A} = [\mathbf{e}_1\ \mathbf{e}_2\ \cdots\ \mathbf{e}_m]^T$. Then the columns of $\mathbf{A}^T$ forms a Parseval frame, and by Definition 5.6c we have that $\mathbf{A}^T\mathbf{A} = \mathbf{I}_n$. Since all the eigenvalues of $\mathbf{I}_n$ are 1, the singular values of $\mathbf{A}^T$ are all 1. By the invariance of singular values under transposition, the same is true for $\mathbf{A}$. □

Proposition 5.1 and Theorem 5.8 gives us a way to enforce every layer in the neural network to be contractions. We will constrain all the weight matrices to have Parseval frames as rows, hence the name of Parseval networks.

5.3 Parseval Training

Now that we have a tractable way to control that the layers of the network are contractions, we shift our focus to how we will enforce this criteria during the training of the network.

In this section, we will present *Parseval training* as introduced in [Cis+17]. Parseval training interlaces every iteration in the training optimization algorithm with a retraction step to ensure that the rows of the weight matrices (approximately) form a Parseval frame.

For a given weight matrix $\mathbf{W}$, the authors of [Cis+17] introduce the following regularization loss:

$$R(\mathbf{W}) = \frac{1}{2}\|\mathbf{W}^T\mathbf{W} - \mathbf{I}\|_2^2 \tag{5.8}$$

with gradient

$$\nabla_{\mathbf{W}} R(\mathbf{W}) = (\mathbf{W}\mathbf{W}^T - \mathbf{I})\mathbf{W} \tag{5.9}$$

We observe that $R(\mathbf{W}) = 0$ if and only if the rows of $\mathbf{W}$ form a Parseval frame. Hence, we will minimize Equation (5.8).

We will solve this minimization problem using gradient descent. This gives us the following variable update:

$$\mathbf{W} \leftarrow \mathbf{W} - \beta\nabla_{\mathbf{W}} R(\mathbf{W}) \tag{5.10}$$

The β will act as a step length for the optimization, and is interpreted as a type of regularization parameter. By inserting Equation (5.9) into Equation (5.10) and rearranging, we get

$$\mathbf{W} \leftarrow (1-\beta)\mathbf{W} - \beta\mathbf{W}\mathbf{W}^T\mathbf{W} \tag{5.11}$$

Since R is a convex function of $\mathbf{W}$, repeating Equation (5.11) until convergence would ensure that the rows of $\mathbf{W}$ are a Parseval frame. However, this is unpractical for a number of reasons.

First, solving a full optimization problem for every layer in each global optimization step is very computationally costly, and will slow the training dramatically. Second, a $\mathbf{W}$ that realizes the minimum of $R(\mathbf{W})$ might be far from the weights that actually minimizes the global loss function. Hence, running the regularization optimization all the way to convergence might negatively affect the performance of the network. Lastly, as we will see in more detail in Section 6.3, it does not seem to be necessary to do more than one step. Empirical results show that the singular values of the matrices are all (very close to) 1

Algorithm 5.1 Parseval training using gradient descent as the base algorithm.

Input: Initial parameters θ, step length η, retraction parameter β, number of epochs E, batch size M, training set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$.

1. For $e = 1, 2, \ldots, E$, do
   - 1.1 Create a new permutation $p = \{p_1, p_2, \ldots, p_N\}$ of $\{1, 2, \ldots, N\}$
   - 1.2 For $n = 1, 2, \ldots, N/M$, do
     - 1.2.1 Sample M examples from shuffled data sets, $\left\{(\mathbf{x}_{p_j}, \mathbf{y}_{p_j})\right\}_{j=nM}^{(n+1)M}$
     - 1.2.2 Compute gradient of loss using sampled mini-batch, $\mathbf{g} \leftarrow \nabla \mathcal{L}(\theta \mid \mathbf{x}_{p_{nM}}, \ldots, \mathbf{x}_{p_{(n+1)M}}, \mathbf{y}_{p_{nM}}, \ldots, \mathbf{y}_{p_{(n+1)M}})$
     - 1.2.3 Apply update to parameters, $\theta \leftarrow \theta - \eta \mathbf{g}$
     - 1.2.4 For each dense/convolutional layer k, do
       - 1.2.4.1 Retraction step, $\mathbf{W}_k \leftarrow (1 + \beta)\mathbf{W}_k - \beta \mathbf{W}_k \mathbf{W}_k^T \mathbf{W}_k$

Output: Trained parameters θ

even after only performing a single step of gradient descent (see Figure 6.8 on page 63).

Performing the retraction step in Equation (5.11) for each layer after every iteration in Algorithm 3.1 or Algorithm 3.2 yields Parseval training, described in detail in Algorithm 5.1.

Remark 5.9. Algorithm 5.1 is slightly different than the proposed algorithm in [Cis+17, Alg. 1]. First, since [Cis+17] regards classification network, they must handle so-called *aggregation layers* as well. This is not relevant for our case of denoising networks, hence we have removed it all-together. Second, in [Cis+17] they propose sampling a subset S of rows of $\mathbf{W}$ in each iteration, and perform the retraction only on the submatrix $\mathbf{W}_S$ formed by the rows indexed by S. This will bring the time complexity of the retraction step to $O(|S|^2 d)$ (where d is the number of layers). Since dense layers often have a lot more parameters than convolutional layers, this is especially helpful for dense layers. However, since our denoising networks rarely comprises dense layers, this is not as useful for our case. The empirical tests shown in Section 6.3 did not show any noticeable speed-down when Parseval constraints were applied, so the sampling seems unnecessary for our application.

5.4 Parseval Networks

By training our networks with the Parseval update step in-between each iteration of our optimization algorithm of choice, illustrated in Algorithm 5.1 for Gradient Descent, we get Parseval networks. The authors of [Cis+17] only discusses using this technique for classification networks. We will briefly illustrate Parseval classification networks in Section 6.3.

Parseval Denoisers

Extending the techniques from [Cis+17] to denoising networks is almost completely straight forward. We will use Parseval training to train the network, but we also need to address the structure of the denoiser itself. Recall from Equation (3.14) on page 30 that a denoiser is the sum of a neural network and the identity. The identity is clearly a contraction, and by using Parseval training to train our neural network, the network is also a contraction. However, recall from the derivations on page 47 that the sum of two contractions is not necessarily a contraction. Hence, for Parseval denoisers we need to redefine Equation (3.14) to use convex combinations instead:

$$D(\mathbf{x}) = \lambda\mathbf{x} + (1-\lambda)\phi(\mathbf{x}) \qquad \lambda \in [0,1] \tag{5.12}$$

Thus, by Proposition 5.3, we can guarantee the whole denoiser to be a contraction.

We can set λ to a fixed value, for example $\lambda = 0.5$, or treat it as a trainable parameter. If we choose to let λ be trainable we must ensure that it never leaves the unit interval. We could do this by clipping the value at 0 and 1[2]. However, recall from Equation (3.3) that the sigmoid function is always in the unit interval. Thus, if we want to have the convex parameter trainable, we can implement that as

$$D(\mathbf{x}) = \sigma(p)\mathbf{x} + (1-\sigma(p))\phi(\mathbf{x}) \qquad p \in \mathbb{R} \tag{5.13}$$

and let p be a trainable parameter with no restrictions.

[2]That is, perform the optimization iteration as normal, and do $\lambda \leftarrow \min(\max(0, \lambda), 1)$ afterwards.

CHAPTER 6

Implementations and Experimental Results

We are now going to examine our implementations and put Parseval reconstruction networks to the test.

In Sections 6.1 to 6.3 we will reimplement and reproduce results from [Sch+18; Ant+19; Cis+17]. Later in Section 6.3 we will test the performance and stability of the denoiser proposed in Section 5.4. Finally, in Section 6.4 we will discuss the performance of Parseval denoisers with regards to reconstruction capabilities and noise vulnerability.

6.1 Our Setup

To perform empirical tests on the effectiveness of Parseval constraints on denoising networks, we opted to use the DeepMRINet [Sch+18] as our base architecture. Throughout we used Python 3.5 [Py3] and TensorFlow 1.8 [TF].

Reimplementation of DeepMRINet

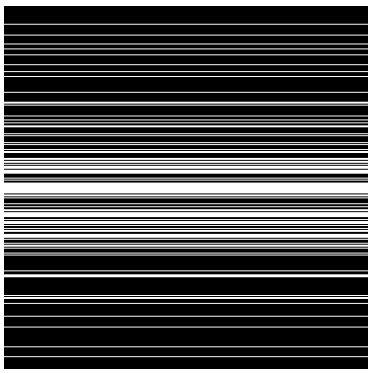
The original implementation of DeepMRINet used somewhat outdated libraries[1]. Because of this we opted to reimplement the network using a more modern framework.

In our reimplementation and retraining of DeepMRINet we made some important changes to the network. First off, we changed the sampling pattern from the line based one in [Sch+18] to Gaussian sampling. The original DeepMRINet were trained on two different sampling rates, namely 33% and 17%. Our sampling rate was set at 25%. A comparison of the two sampling patterns is found in Figure 6.1.

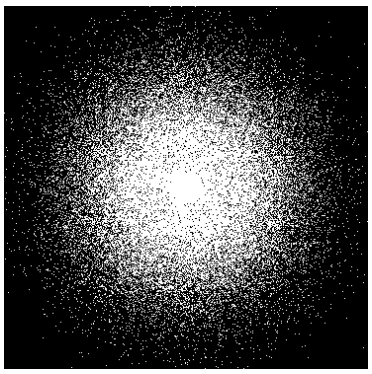
Because of limited computation resources we had to scale down the number of layers in the network slightly in order to fit it on a single GeForce GTX 1080 without reducing the batch size too much. Thus our version has 4 CNNs instead of the original 5, and each CNN consist of 4 32-channeled convolutional layers instead of 5 64-channeled layers.

All of our networks were trained on a subset of the dataset from [Ham+18] consisting of 2 679 images. Of these, 120 were set aside as a test set. We

[1] [Sch+18] uses Theano, which was announced to be discontinued after it's initial 1.0 launch in 2017. It has received some bug fixes and compatibility updates since then, but no feature updates.



(a) Sampling pattern in [Sch+18], adjusted to our sampling rate



(b) Our sampling pattern

Figure 6.1: Sampling patterns in the Fourier domain. Both subsample at 25%.

trained the network using the Adam optimizer for 40 epochs resulting in 10 236 iterations. During training we employed different data augmentation strategies, including flipping along both width and height axis, rotations and addition of Gaussian noise. These augmentations were done independently for each batch every time a new batch was drawn.

Changes to Algorithm 4.1

The implementation of Algorithm 4.1 given in [Ant+19] is very general, and meant to work on any denoising network written in any framework. Our situation is not as general, so we have opted to implement Algorithm 4.1 differently than [Ant+19].

We implemented Algorithm 4.1 as a Neural Network, see Figure 6.2 for a graph representation of the architecture. When searching for perturbations we view all the variables in the network as untrainable constants *except* for the perturbation $\boldsymbol{\delta}$. We then seek to maximize the value of the Q node in the graph given in Figure 6.2. This node computes Equation (4.4).

Note that if we use Stochastic Gradient Descent with Nesterov Momentum[2] to train this network, our approach is identical to Algorithm 4.1. However, we achieved faster convergence with the Adam optimizer, so we opted to use Adam instead.

Thus, our algorithm has the following hyper-parameters:

- N – The number of steps for the optimization algorithm
- η – The step length of the optimization algorithm
- α_1, α_2 – Decay rates for Adam. In our experiments, these are always set at the suggested default of $\alpha_1 = 0.9$ and $\alpha_2 = 0.999$.

[2]This method is implemented in TensorFlow as `tf.train.MomentumOptimizer` if `use_nesterov=True` is given as a keyword argument.

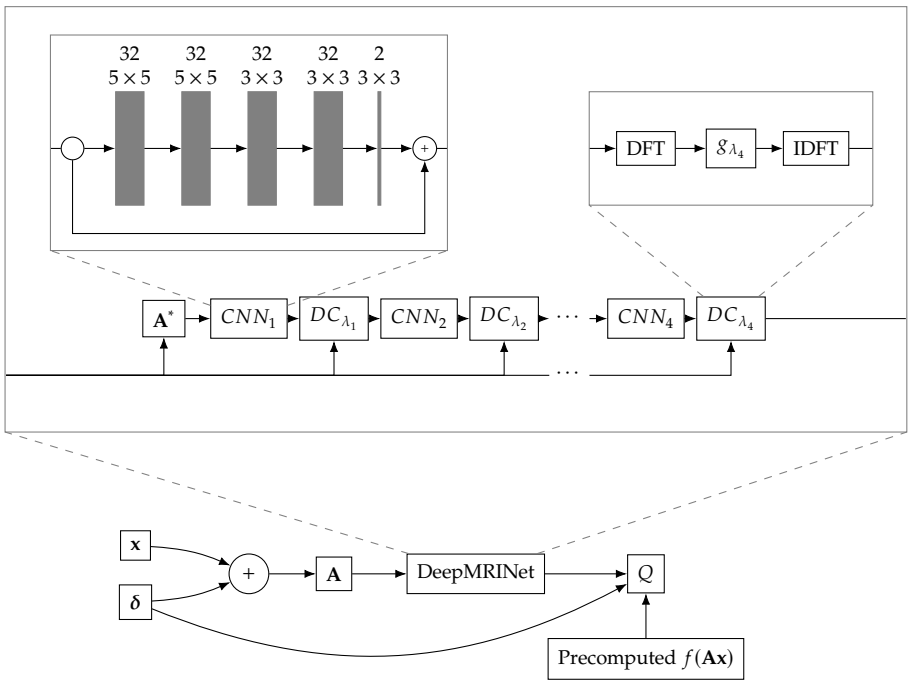


Figure 6.2: The architecture for the instability experiments

- λ – The trade-off between reconstruction error and perturbation size
- τ – The initial element-wise size of the perturbation. Ie, draw initial perturbation elements from Unif$[0, \tau]$

6.2 Current Instabilities

In this section we will recreate some of the results from [Ant+19] using our own version of Algorithm 4.1, outlined in Section 6.1.

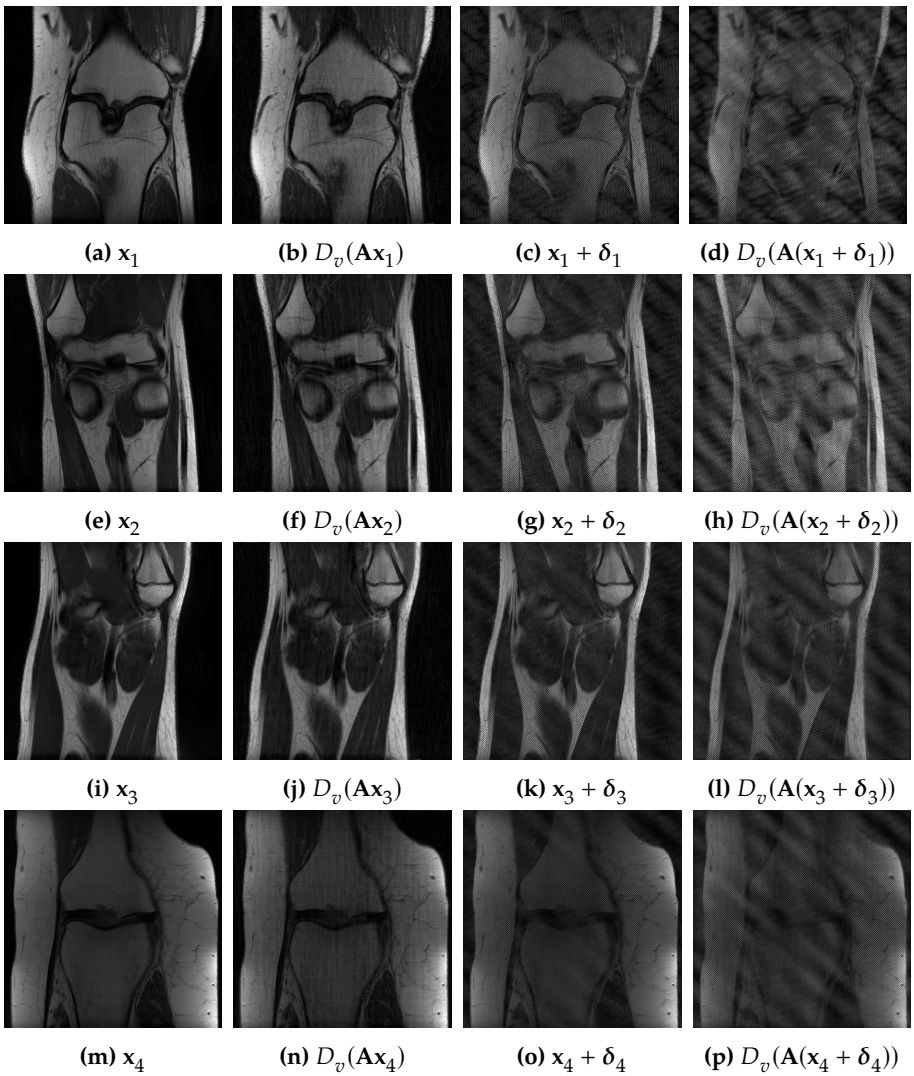
We found that the following hyper-parameters gave good results:

$$N = 50 \qquad \eta = 0.001 \qquad \lambda = 1 \qquad \tau = 10^{-5}$$

Example perturbations found with these hyper-parameters are found in Figure 6.3. In these examples we have drawn 4 images from an independent test set, never before seen by the network. For each sample, we have created synthetic measurements by using the measurement operator:

$$\mathbf{y}_i = \mathbf{A}\mathbf{x}_i \quad \text{for } i = 1, 2, 3, 4$$

We then used our own reimplementation of DeepMRINet to recover $\hat{\mathbf{x}}_i$ from $\mathbf{y}_i$. Note that since the Data Consistency layers depend on the measurements in the Fourier domain, the input to DeepMRINet is $\mathbf{y}_i$ directly, and not $\mathbf{A}^*\mathbf{y}_i$. Taking the adjoint of the measurement operator is built into the network. Throughout this chapter we will use D_v to denote our vanilla DeepMRINet implementation.



(a) $\mathbf{x}_1$ **(b)** $D_v(\mathbf{A}\mathbf{x}_1)$ **(c)** $\mathbf{x}_1 + \boldsymbol{\delta}_1$ **(d)** $D_v(\mathbf{A}(\mathbf{x}_1 + \boldsymbol{\delta}_1))$

(e) $\mathbf{x}_2$ **(f)** $D_v(\mathbf{A}\mathbf{x}_2)$ **(g)** $\mathbf{x}_2 + \boldsymbol{\delta}_2$ **(h)** $D_v(\mathbf{A}(\mathbf{x}_2 + \boldsymbol{\delta}_2))$

(i) $\mathbf{x}_3$ **(j)** $D_v(\mathbf{A}\mathbf{x}_3)$ **(k)** $\mathbf{x}_3 + \boldsymbol{\delta}_3$ **(l)** $D_v(\mathbf{A}(\mathbf{x}_3 + \boldsymbol{\delta}_3))$

(m) $\mathbf{x}_4$ **(n)** $D_v(\mathbf{A}\mathbf{x}_4)$ **(o)** $\mathbf{x}_4 + \boldsymbol{\delta}_4$ **(p)** $D_v(\mathbf{A}(\mathbf{x}_4 + \boldsymbol{\delta}_4))$

Figure 6.3: Adversarial attacks against the author's implementation of DeepMRINet

6.3 Applying Parseval Constraints

We will now study the effects of applying Parseval training. We will begin by recreating the results from [Cis+17] before applying Parseval constraints to MRI reconstruction networks.

Recreating the Results From [Cis+17]

To confirm that our implementation is correct, we will reproduce the results in [Cis+17]. We will do this by implementing a simple classification network and train it with and without the Parseval retraction step. Since classification networks are not the main focus of this thesis, we will only give a broad summary

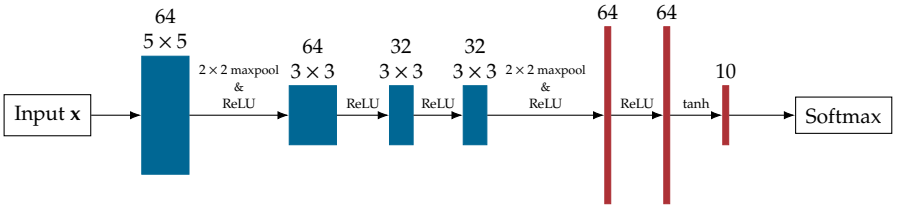


Figure 6.4: Architecture of the classifactor used to verify our implementation of Parseval constraints. Blue blocks depict convolutional layers, while red blocks depict dense layers.

of our network, and refer to Appendix A.1 for details. Briefly speaking, the main difference from before is that the output of the network is now categorical (ie, the predicted label of the input).

The authors of [Cis+17] did not report on any specifics of their architecture. Our network will be fairly small in order to make the network train fast, and since we are more concerned with the *change* in stability rather than the performance of the network. Our network will consist of 4 convolutional layers, followed by 3 dense layers. We use ReLU activation, except for the two last layers where the second last has tanh activation and the last layer has no activation. The output of the last layer is sent to a softmax function (see Equation (A.1)) to turn the output into a valid discrete probability distribution.

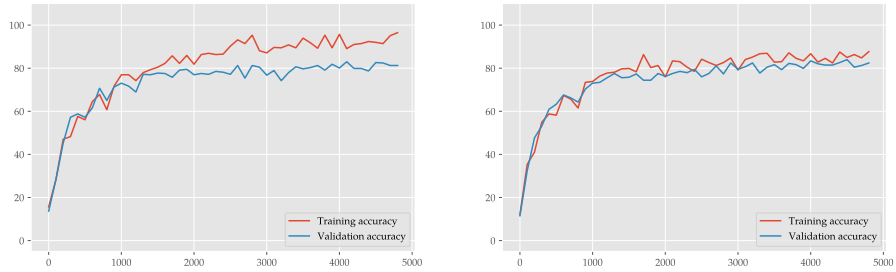
Between layers 1 and 2 and layers 4 and 5 we employed max pooling (see Equation (A.2)) to reduce the dimensionality. At the end of the network, we let the arg max of output from the softmax function be the predicted label of the input. A graph representation of our classificator architecture is found in Figure 6.4.

We trained our networks to classify images from the CIFAR-10 database [KH09]. It consists of 50 000 labeled $32 \times 32 \times 3$ images from 10 different classes, as well a dedicated test set of 10 000 additional images. We trained the networks using the Adam optimizer with cross-entropy loss (see Equation (A.3)), and trained for 50 epochs with a batch size of 512 as the networks seemed to converge quickly. We also employed *dropout* to limit overfitting [Sri+14]. Both the vanilla and Parseval classifier were initialized with orthogonal rows. The training progressions are found in Figure 6.5. The vanilla classifier converged with a validation accuracy of 81.2% and the Parseval classifier converged with a validation accuracy of 82.4%.

In [Cis+17] the authors report several benefits of Parseval classifiers, among them:

1. Near-orthogonal weight matrices
2. Faster convergence
3. Higher robustness to adversarial noise

The near-orthogonal "benefit" is merely a result of the Parseval retraction step, but is nevertheless a good indicator that our implementation is working as expected. Since this will also be true for our denoising networks, we will not



(a) Vanilla classifier **(b)** Parseval classifier

Figure 6.5: Training progression of the classifier depicted in Figure 6.4.

investigate this further in this section but rather revisit it when discussing the Parseval denoiser (see Figure 6.8).

We did not see a massive increase in convergence speed, but we did observe a more stable convergence. That is, the loss function and accuracy did not fluctuate as much while training the Parseval classifier. However, one single observation of Parseval training on classifiers is not sufficient to either disprove the reported increase in convergence in [Cis+17] or prove increased stability during training. To make such a conclusion, we would need to train several networks of different architectures with and without the Parseval retraction step and do proper statistical inference.

To test the robustness of these models we will take 100 images from the test set that the classifiers classify correctly, and run the DeepFool algorithm for adversarial attacks [MFF16] as implemented in [RBB17] until a misclassification happens. We then compute the *mean Signal-to-Noise Ratio* (*mean SNR* or *MSNR*) for each of these batches of 100 attacks. For images $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ with corresponding perturbations $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \dots, \boldsymbol{\delta}_n$ the mean SNR is given as

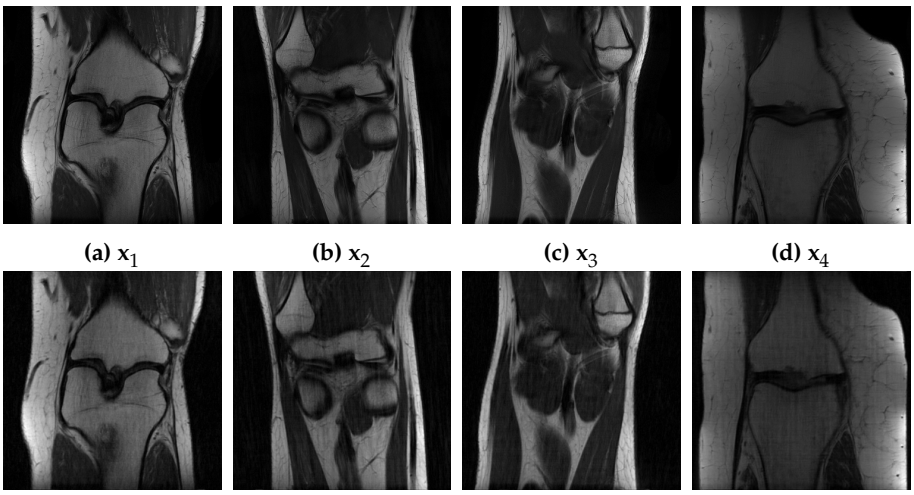
$$\mathrm{MSNR}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n, \boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \dots, \boldsymbol{\delta}_n) = \frac{1}{n} \sum_{i=1}^{n} 10 \log_{10} \frac{\|\mathbf{x}_i\|_2}{\|\boldsymbol{\delta}_i\|_2}$$

and will be a measure on *how much* we must typically perturb an image before a misclassification happens, where a lower mean SNR means that a larger perturbation is necessary. Doing this computation, we get that for the vanilla classifier, the mean SNR for our batch of images is 7.8, while for the Parseval classifier, the mean SNR for the batch is 4.2, suggesting that a larger perturbation is necessary to fool the Parseval classifier[3].

Parseval Constraints on DeepMRINet

To test the effectiveness of Parseval constraints on MRI reconstruction networks, we trained an alternative version of our reimplementation of DeepMRINet with

[3] These signal-to-noise ratios are small, suggesting that both of these models are rather stable. This is not very surprising as the performance of 80% is not very high. Instability usually increases with performance.



(a) $\mathbf{x}_1$ **(b)** $\mathbf{x}_2$ **(c)** $\mathbf{x}_3$ **(d)** $\mathbf{x}_4$

(e) $D_p(\mathbf{A}\mathbf{x}_1)$ **(f)** $D_p(\mathbf{A}\mathbf{x}_2)$ **(g)** $D_p(\mathbf{A}\mathbf{x}_3)$ **(h)** $D_p(\mathbf{A}\mathbf{x}_4)$

Figure 6.6: Reconstruction with the Parseval DeepMRINet, using the same measurement operator as before.

Parseval constraints applied. The network architecture, data set and choice data augmentation and optimizer was kept identical to the vanilla case defined in Section 6.1, except for the definition of the denoiser which where changed to the one in Equation (5.13). We will use D_p to depict the Parseval version of DeepMRINet.

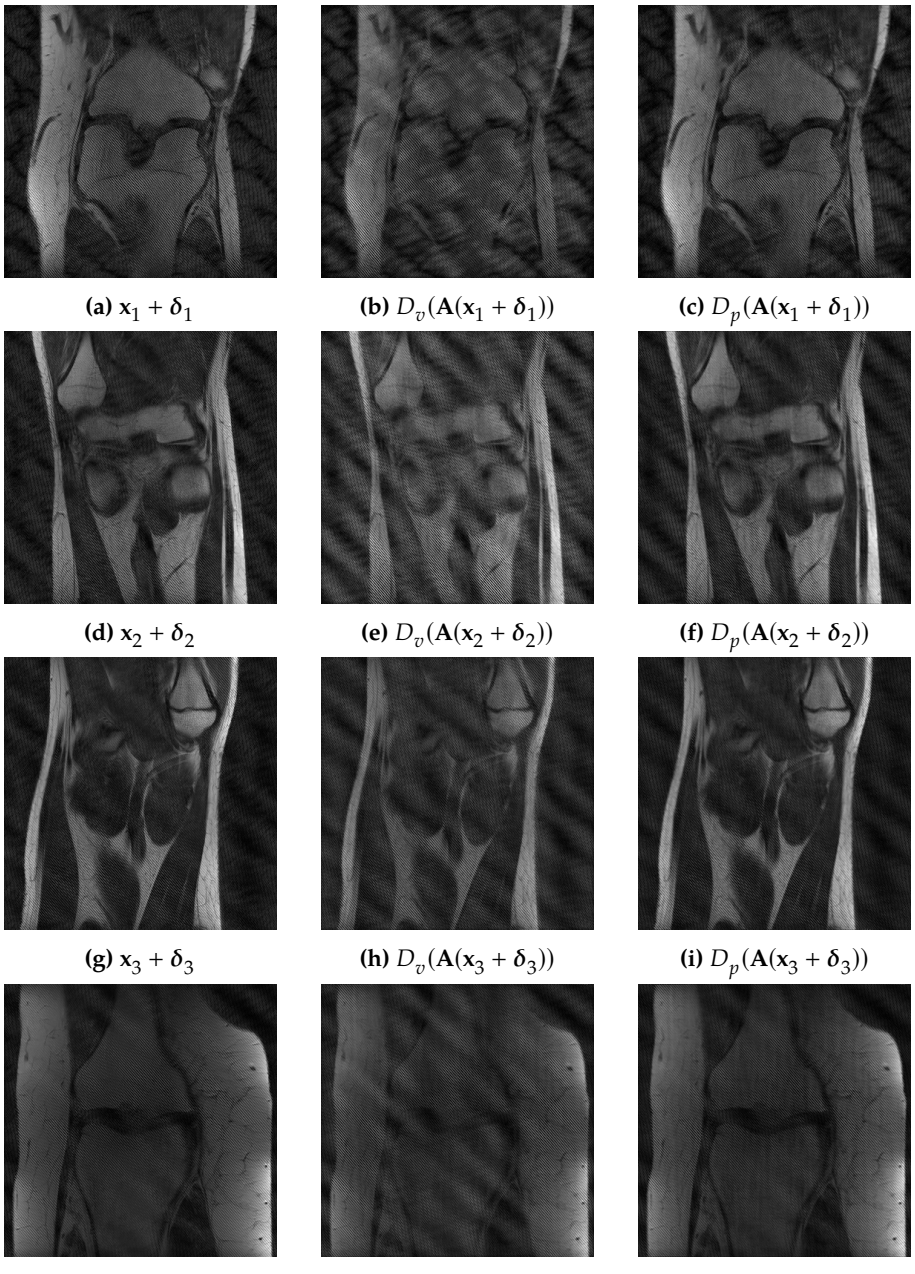
In Figure 6.6 we display the reconstruction capabilities of the Parseval DeepMRINet, and in Figure 6.7 we show the Parseval DeepMRINet and the vanilla DeepMRINet reconstructions of the same perturbed inputs as in Figure 6.3. We see that the reconstruction error is much smaller for the Parseval network. We will discuss changed performance in more detail in Section 6.4.

To ensure that the Parseval retraction step does what we expect it to do, we can calculate the singular values of the trained weight matrices. In the Parseval case, we would expect all the singular values to be approximately 1, while in the vanilla case we would expect them to distribute wider. If we do this, we find that this is in fact the case, as shown in Figure 6.8.

6.4 Perceived Changes

We will now discuss some of the changes in performance we observe when applying Parseval constraints to our denoising networks. Specifically, we are going to examine three perceived changes:

- Networks with Parseval constraints seem to be less vulnerable to overfitting
- Denoisers with Parseval constraints does not seem to amplify errors on adversarial attacks



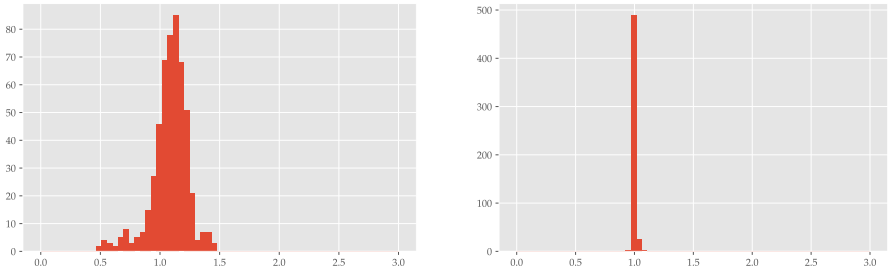
(a) $\mathbf{x}_1 + \boldsymbol{\delta}_1$ **(b)** $D_v(\mathbf{A}(\mathbf{x}_1 + \boldsymbol{\delta}_1))$ **(c)** $D_p(\mathbf{A}(\mathbf{x}_1 + \boldsymbol{\delta}_1))$

(d) $\mathbf{x}_2 + \boldsymbol{\delta}_2$ **(e)** $D_v(\mathbf{A}(\mathbf{x}_2 + \boldsymbol{\delta}_2))$ **(f)** $D_p(\mathbf{A}(\mathbf{x}_2 + \boldsymbol{\delta}_2))$

(g) $\mathbf{x}_3 + \boldsymbol{\delta}_3$ **(h)** $D_v(\mathbf{A}(\mathbf{x}_3 + \boldsymbol{\delta}_3))$ **(i)** $D_p(\mathbf{A}(\mathbf{x}_3 + \boldsymbol{\delta}_3))$

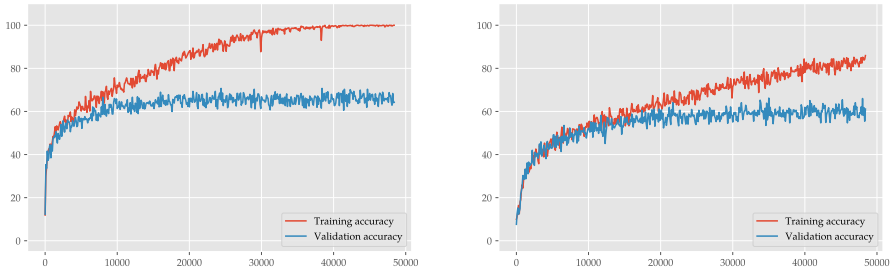
(j) $\mathbf{x}_4 + \boldsymbol{\delta}_4$ **(k)** $D_v(\mathbf{A}(\mathbf{x}_4 + \boldsymbol{\delta}_4))$ **(l)** $D_p(\mathbf{A}(\mathbf{x}_4 + \boldsymbol{\delta}_4))$

Figure 6.7: Comparison of adversarial attacks against Parseval and vanilla DeepMRINet.



(a) Vanilla DeepMRINet **(b)** Parseval DeepMRINet

Figure 6.8: Histograms of singular values for the layers of our recreation of DeepMRINet, compared to the Parseval version



(a) Vanilla classifier **(b)** Parseval classifier

Figure 6.9: Overfitting the classifier depicted in Figure 6.4.

- Denoisers with Parseval constraints seem to be less able to reliably recover fine details in images

Risk of Overfitting

Discussing overfitting on denoising networks are very difficult, as we do not have any clear measure on *successful recover*[4]. Hence, when discussing overfitting, we will consider the classification network we designed in Section 6.3.

In Figure 6.5 we can see some tendencies that the vanilla network is overfitting more than its Parseval counterpart. We will however redo the training. In Section 6.3 we trained our network with several techniques that limit overfitting, such as dropout and data augmentation. In order to test the effectiveness of the Parseval regularizer as a means of limiting overfitting, we will remove all of these techniques and retrain the model with nothing but simple Stochastic Gradient Descent (Algorithm 3.1) and a dedicated validation set to measure overfitting.

[4] other than the eyeball metric (i.e. looking at the image and deciding manually) which is not feasible to implement when training.

i	$\|\boldsymbol{\delta}_i\|_2$	$\|D_v(\mathbf{A}\mathbf{x}_i) - D_v(\mathbf{A}(\mathbf{x}_i + \boldsymbol{\delta}_i))\|_2$	$\|D_p(\mathbf{A}\mathbf{x}_i) - D_p(\mathbf{A}(\mathbf{x}_i + \boldsymbol{\delta}_i))\|_2$
1	41.87	61.63	43.86
2	33.72	61.34	34.99
3	29.84	46.19	30.38
4	26.17	41.21	27.98

Table 6.1: Amplification of adversarial noise

We trained two versions of the classifier on CIFAR-10 for 500 epochs with a batch size of 512 and a step size of 0.001. The resulting training progressions are depicted in Figure 6.9. We can see that both models begins to overfit after 10 000-20 000 iterations, however the overfitting in the vanilla case is far more severe, even approaching a training accuracy of 100%.

Robustness to Adversarial Attacks

It is clear from Figure 6.7 that our Parseval DeepMRINet does not amplify the noise in our adversarial attacks in the same way that the vanilla DeepMRINet does.

To get a more concrete measure on how our denoisers amplify noise from adversarial attacks, we can compare the difference between the reconstructed image and the reconstructed perturbed image (the first term of Equation (4.4)) with the size of the perturbation (second term of Equation (4.4)). If the difference between the two reconstructions are larger than the norm of the perturbation we can say that the error has been amplified. A table of these norms and differences are found in Table 6.1.

The computations in Table 6.1 confirms what we see in Figure 6.7. Theoretically, we should not see any noise amplification for the Parseval DeepMRINet since the entire denoiser is a contraction. Recall Equation (4.1) on page 37, or specifically for a Parseval denoiser D_p and a measurement operator $\mathbf{A} = \mathbf{P}_\Omega \mathbf{F}_n$, we have that:

$$\begin{aligned} \|D_p(\mathbf{A}\mathbf{x}) - D_p(\mathbf{A}(\mathbf{x}+\boldsymbol{\delta}))\|_2 &< \|\mathbf{A}\mathbf{x} - \mathbf{A}(\mathbf{x}+\boldsymbol{\delta})\| \\ &\leq \|\mathbf{A}\|_2 \|\boldsymbol{\delta}\|_2 \\ &= \|\mathbf{P}_\Omega \mathbf{F}_n\|_2 \|\boldsymbol{\delta}\|_2 \\ &= \|\mathbf{P}_\Omega\|_2 \|\boldsymbol{\delta}\|_2 \\ &= \|\boldsymbol{\delta}\|_2 \end{aligned}$$

However, we see slight increases in error for the Parseval DeepMRINet in Table 6.1. This is most likely be due to a combination of two contributors.

First, recall from Section 5.3 that we are not guaranteeing the weight matrix of a layer to be a Parseval frame, but merely approximating. As illustrated in Figure 6.8b this leaves the possibility for some singular values to be slightly larger than 1. Second, we have that numerical errors could be affecting our computations, as the network were trained on a GPU only supporting singe-precision (32-bit) floating-point numbers. Thus making round-off errors

(a) $\mathbf{x}_1$ zoomed **(b)** $D_v(\mathbf{A}\mathbf{x}_1)$ zoomed **(c)** $D_p(\mathbf{A}\mathbf{x}_1)$ zoomed

Figure 6.10: Reconstruction of fine details with vanilla and Parseval DeepMRINet

more severe what we are used to with double-precision (64-bit) floating-point numbers[5].

Reconstruction Capabilities

Finally, we will compare the reconstruction performance of Parseval and vanilla denoising networks. Making a stable denoiser is not impressive if we loose too much of the performance. For example, using the identity as a denoiser will be extremely stable to adversarial noise, but that does not mean that it will work well as a denoiser.

From Figure 6.6 we see that the Parseval DeepMRINet is at least capable of reconstructing large-scale details. However, when zooming in at certain areas of the resulting image, some weaknesses appear.

In Figure 6.10 we have taken $\mathbf{x}_1$ and zoomed in roughly 350%, and we see that many fine details are lost or blurred in both the vanilla and Parseval reconstructions, but even more so in the Parseval case. These weaknesses were not present in [Sch+18]. This observation raises two main questions:

- Why is our reimplementation of DeepMRINet less capable of recovering small details than the original implementation?
- Why is our Parseval denoiser less capable of recovering small details than its vanilla counterpart?

We will further provide some thoughts of what the source(s) for the reduction in performance can be.

Performance reduction for vanilla DeepMRINet

We did a number of changes to our version of DeepMRINet compared to the original in [Sch+18]. Most notably, we did the following changes on our version:

[5]Since most commercial GPUs only have 32-bit arithmetic engines, using single-precision numbers is still quite common in DL

- We scaled the network down from 5 CNNs with 5 layers each consisting of 64 3×3 convolutions, to 4 CNNs with 4 layers each consisting of 32 convolutions with 5×5 filters in the first layer and 3×3 in the last 3.
- We changed both the sampling pattern and sampling rate.
- We trained on a different dataset, and with several times more data.
- We trained for fewer epochs.

Any of these changes could have contributed to the reduction in performance. However, we suspect that the change in architecture, sampling rate and dataset are the most likely to affect the performance negatively.

The reduction in network size is an obvious candidate for a reduction in performance. By reducing the network, we potentially reduce the generality and approximation capabilities of the network.

Lowering the sampling rate gives us less data to work on. In [Sch+18], the authors train two versions of DeepMRINet, one with 33% sampling rate and one with 17% sampling rate. By comparing the performance of these two models, we see that the 17% versions exhibits more of the blurring artifacts present in Figure 6.10 than the 33% version [Sch+18, Figures 8 and 9]. Thus, lowering the sampling rate to 25% could be a contributor to the reduction in small-detail reconstruction performance.

Finally, our dataset is an order of magnitude larger than the dataset in [Sch+18], and contains larger variations as we trained on data from different cordial axes and fat-suppression. Hence, we tried to learn to reconstruct a larger domain of images. This could also have led to more overfitting of the original than ours. In addition, we used a completely separate test set to evaluate our model performance, while [Sch+18] used a 2-fold cross-validation approach. Thus, the recoveries reported in [Sch+18] might have been overly optimistic compared to our tests.

We do not suspect the change in number of epochs or sampling pattern to negatively impact the performance. Even though we trained for far fewer epochs, each epoch consist of more iterations, bringing the total number of iterations up[6]. We stopped the training after 40 epochs as we no longer saw meaningful changes to the validations loss, suggesting that we were about to enter an overfitting phase. The change of sampling pattern was a conscious decision as a Gaussian sampling pattern preserves more details and produces less artifacts when recovering with the adjoint compared to the original pattern in [Sch+18].

Performance reduction for Parseval DeepMRINet

The noticeable reduction in performance for the Parseval DeepMRINet might indicate that the Parseval constraint is a very strict regularization scheme. When we limit how much an error can propagate, we also limit the expressiveness of

[6] As mentioned on page 56, we trained for a total of 10 236 iterations, while [Sch+18] was trained with a batch size of 1, using 150 images for 200 epochs resulting in 30 000 iterations – which is still more, but not in terms of an order of magnitude as it might initially seem when only comparing epoch numbers.

the model, not unlike the previously mentioned bias-variance trade-off (page 41).

A danger of limiting all the layers in the network to be contractions, is that we risk the output of each layer to get exponentially smaller. Thus, the scale of the output of the neural network in a denoiser could be small compared to the scale of the input image, so when summing them, the contribution from the network is largely neglected (recall Equation (3.14) on page 30 and the refined version in Equation (5.12) on page 53).

However, we have not adjusted the architecture or hyper-parameters when applying the Parseval constraints (other than the ones necessary to apply the constraints). It is not unreasonable to think that we should adjust some of the hyper-parameters when we add the Parseval update steps and scalings, in addition to changing the initialization scheme. There could be an architecture and a set of hyper-parameters where a Parseval denoiser would perform as well as non-Parseval denoiser. We will however not investigate this further in this thesis.

CHAPTER 7

Conclusions

In this thesis we have reviewed some of the theory and practices in Deep Learning for MRI reconstruction (Chapter 3), as well as some preliminary theory on Compressive Sensing techniques (Section 2.1). We have reviewed recent research regarding the stability of such methods (Section 4.2), and reproduced some of their results (Section 6.2).

In addition, we have reviewed a recently proposed regularization technique for classification networks meant to increase stability towards adversarial attacks (Chapter 5). We have extended the theory behind this regularizer to cover convolutional layers as they appear in modern Deep Learning (page 44), and provided an implementation of Parseval training as a free and open source software library for Python with TensorFlow[1]. We have used this to introduce *Parseval denoisers* as a recovery scheme for undersampled MR images.

We have seen that by applying the Parseval retraction step during training, we achieve a more stable recovery scheme (Figure 6.7) that does not significantly amplify noise (Table 6.1), while we seem to sacrifice *some* reconstruction capability (Figure 6.10). However, as discussed on page 67 of Section 6.4, we have not performed an extensive test of architectures and hyper-parameters. Hence there could exist an architecture and a set of hyper-parameters where Parseval denoisers would perform on-par with state-of-the-art Deep Learning based approaches. We leave the search for such parameters for future work.

[1] Available at GitHub: `https://github.com/mathialo/parsnet`, as well as on the Python Package Index. See Appendix B.1 for more details.

Bibliography

[Adc+17] B. Adcock, A. C. Hansen, C. Poon, and B. Roman. "Breaking the coherence barrier: A new theory for compressed sensing." In: *Forum of Mathematics, Sigma*. Vol. 5. Cambridge University Press. 2017.

[Ant+19] V. Antun, F. Renna, C. Poon, B. Adcock, and A. C. Hansen. "On Instabilities of Deep Learning in Image Reconstruction – Does AI Come at a Cost?" In: *arXiv preprint arXiv:1902.05300* (2019).

[Big+13] B. Biggio et al. "Evasion attacks against machine learning at test time." In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer. 2013, pp. 387–402.

[BT09] A. Beck and M. Teboulle. "A fast iterative shrinkage-thresholding algorithm for linear inverse problems." In: *SIAM journal on imaging sciences* 2.1 (2009), pp. 183–202.

[Cab+14] J. Caballero, A. N. Price, D. Rueckert, and J. V. Hajnal. "Dictionary learning and time sparsity for dynamic MR data reconstruction." In: *IEEE transactions on medical imaging* 33.4 (2014), pp. 979–994.

[Cis+17] M. Cisse, P. Bojanowski, E. Grave, Y. Dauphin, and N. Usunier. "Parseval Networks: Improving Robustness to Adversarial Examples." In: *International Conference on Machine Learning* (2017), pp. 854–863.

[CRT06] E. J. Candès, J. Romberg, and T. Tao. "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information." In: *IEEE Transactions on information theory* 52.2 (2006), pp. 489–509.

[CT06] E. J. Candes and T. Tao. "Near-optimal signal recovery from random projections: Universal encoding strategies?" In: *IEEE transactions on information theory* 52.12 (2006), pp. 5406–5425.

[Cyb89] G. Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[Don06] D. L. Donoho. "Compressed sensing." In: *IEEE Transactions on information theory* 52.4 (2006), pp. 1289–1306.

[FDA17] United States Food and Drug Administration. *510k Premarket notification of Compressed Sensing Cardiac Cine*. Jan. 27, 2017. URL: `https://www.accessdata.fda.gov/cdrh_docs/pdf16/K163312.pdf` (visited on 04/30/2019).

[Fer+10] R. Fergus, M. D. Zeiler, G. W. Taylor, and D. Krishnan. "Deconvolutional networks." In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition(CVPR)*. 2010.

[Flo12] M. A. Flower. *Webb's physics of medical imaging*. CRC Press, 2012.

[FMF17] A. Fawzi, S. M. Moosavi Dezfooli, and P. Frossard. "The Robustness of Deep Networks-A geometric perspective." In: *IEEE Signal Processing Magazine* 34 (2017).

[FR13] S. Foucart and H. Rauhut. *A Mathematical Introduction to Compressive Sensing*. Birkhauser, 2013.

[GB10] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.

[GR14] S. Gu and L. Rigazio. "Towards deep neural network architectures robust to adversarial examples." In: *arXiv preprint arXiv:1412.5068* (2014).

[Ham+18] K. Hammernik et al. "Learning a variational network for reconstruction of accelerated MRI data." In: *Magnetic Resonance in Medicine* 79.6 (2018), pp. 3055–3071.

[Hau19] K. M. Haug. "TITTEL." MA thesis. University of Oslo, 2019.

[He+15] K. He, X. Zhang, S. Ren, and J. Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[Hor91] K. Hornik. "Approximation capabilities of multilayer feedforward networks." In: *Neural networks* 4.2 (1991), pp. 251–257.

[IS15] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *arXiv preprint arXiv:1502.03167* (2015).

[ISL] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer, 2013.

[KB14] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[KH09] A. Krizhevsky and G. Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. University of Toronto, 2009.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LeC+98] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[LS18] Z. C. Lipton and J. Steinhardt. "Troubling trends in machine learning scholarship." In: *arXiv preprint arXiv:1807.03341* (2018).

[MFF16] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. "Deepfool: a simple and accurate method to fool deep neural networks." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2574–2582.

[MJU17] M. T. McCann, K. H. Jin, and M. Unser. "Convolutional neural networks for inverse problems in imaging: A review." In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 85–95.

[MM16] J. Ma and M. März. "A multilevel based reweighting algorithm with joint regularizers for sparse recovery." In: *arXiv preprint arXiv:1604.06941* (2016).

[MW13] J. N. McDonald and N. A. Weiss. *A Course in Real Analysis*. 2nd ed. Academic Press, 2013.

[Pin99] A. Pinkus. "Approximation theory of the MLP model in neural networks." In: *Acta Numerica* 8 (1999), pp. 143–195.

[Py3] G. van Rossum et al. *Python: A dynamic, open source programming language*. Available at `https://www.python.org`.

[RBB17] J. Rauber, W. Brendel, and M. Bethge. "Foolbox: A Python toolbox to benchmark the robustness of machine learning models." In: *arXiv preprint arXiv:1707.04131* (2017).

[RFB15] O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." In: *Medical Image Computing and Computer-Assisted Intervention*. Springer International Publishing, 2015, pp. 234–241.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–536.

[RKK18] S. J. Reddi, S. Kale, and S. Kumar. "On the Convergence of Adam and Beyond." In: *International Conference on Learning Representations*. 2018.

[Rus+15] O. Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge." In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252.

[Rya19] Ø. Ryan. *Linear Algebra, Signal Processing and Wavelets – A Unified Approach. Python version*. Springer, 2019.

[Sch+18] J. Schlemper, J. Caballero, J. V. Hajnal, A. N. Price, and D. Rueckert. "A Deep Cascade of Convolutional Neural Networks for Dynamic MR Image Reconstruction." In: *IEEE transactions on Medical Imaging* 37.2 (2018), pp. 491–503.

[Sie17] Siemens AG. *FDA Clears Compressed Sensing MRI Acceleration Technology From Siemens Healthineers*. Feb. 21, 2017. URL: https://usa.healthcare.siemens.com/news/fda-clears-mri-technology-02-21-2017.html (visited on 11/30/2018).

[SM17] S. Sonoda and N. Murata. "Neural network with unbounded activation functions is universal approximator." In: *Applied and Computational Harmonic Analysis* 43.2 (2017), pp. 233–268.

[SMG13] A. M. Saxe, J. L. McClelland, and S. Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." In: *arXiv preprint arXiv:1312.6120* (2013).

[Sri+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[SZ14] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition." In: *arXiv preprint arXiv:1409.1556* (2014).

[Sze+13] C. Szegedy et al. "Intriguing properties of neural networks." In: *arXiv preprint arXiv:1312.6199* (2013).

[Sze+17] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. "Inception-v4, inception-resnet and the impact of residual connections on learning." In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[TF] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Available at https://www.tensorflow.org.

[Yan+18] G. Yang et al. "Dagan: Deep de-aliasing generative adversarial networks for fast compressed sensing mri reconstruction." In: *IEEE transactions on medical imaging* 37.6 (2018), pp. 1310–1321.

[Zhu+18] B. Zhu, J. Z. Liu, S. F. Cauley, B. R. Rosen, and M. S. Rosen. "Image reconstruction by domain-transform manifold learning." In: *Nature* 555.7697 (2018), pp. 487–492.

APPENDIX A

Supplementary Material

A.1 Neural Networks for Classification

In this section, we will show how neural networks can be applied as classificators. Classification is a supervised learning problem where the output is a categorical variable, often called the *class* or *label* of the input. We always know the set of possible classes a priori.

When using neural networks for classification, we let the last layer have dimensionality identical to the number of classes. We want the last layer to be a discrete probability distribution depicting how likely the input is to be of the different classes. The prediction made is then the arg max of this output vector.

A vector is a discrete probability distribution if all elements are non-negative and sum to 1. To achieve this, it is very common to use the softmax function, defined as

$$s(\mathbf{a})_j = \frac{e^{a_j}}{\sum_i e^{a_i}} \tag{A.1}$$

as the activation in the last layer.

Classification networks often employ additional layers not typical for denoising networks. One example is *pooling layers*. Pooling can be viewed as a type of non-linear sampling, reducing spatial dimensionality in some way. The most common pooling types are max pooling and average pooling. In 1D, max pooling with stride 2 is defined as

$$\operatorname{maxpool}_2 \mathbf{x} = \begin{bmatrix} \max\{x_1, x_2\} \\ \max\{x_3, x_4\} \\ \vdots \\ \max\{x_{n-1}, x_n\} \end{bmatrix} \tag{A.2}$$

Average pooling works the same, but with means instead of maximums. This extends to higher dimensions in the obvious way. For example, 2D pooling will work on $m \times n$ patches. Pooling are typically applied before the activation function. Note that since all the typical activation functions are increasing (recall Equation (3.3) on page 19), we have that

$$\rho(\operatorname{maxpool}_2 \mathbf{x}) = \operatorname{maxpool}_2 \rho(\mathbf{x})$$

So it does not make a difference if we apply max pooling before or after the activation. However, the same is *not* true for average pooling.

Since the output is categorical, we often interpret it as a whole number between 1 and the number of classes C. However, during training it is often convenient to use the *one-hot encoding* of the output instead. The one-hot encoded version of the class y is a vector of length C consisting of 0s, except for a 1 on index y. That is, the one-hot version $\mathbf{y}'$ of a class y is given as

$$\mathbf{y}' = \left[\mathcal{I}(y = i)\right]_{i=1}^{i=C}$$

where $\mathcal{I}(\cdot)$ depicts the indicator function.

One immediate loss function using the one-hot encoding is to take the MSE between the one-hot encoded class and the output probability distribution from the network:

$$\mathcal{L}(\theta \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, y_1, \ldots, y_n) = \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{y}'_i - \Phi_\theta(\mathbf{x})\|_2^2$$

Another commonly used loss function for classification is the *cross-entropy loss*, defined as

$$\mathcal{L}(\theta \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, y_1, \ldots, y_n) = \frac{1}{n}\sum_{i=1}^{n}\left(-\sum_{j=1}^{C}(\mathbf{y}'_i)_j \log(\Phi_\theta(\mathbf{x}_i)_j)\right) \tag{A.3}$$

Both assume that the softmax have been applied to the last layer.

APPENDIX **B**

Implementation Details

In this chapter we will give some details on our implementations for the proposed method, as well as how selected figures are generated. Throughout we will be using Python 3.5, with the following external libraries:

- TensorFlow version 1.8
- NumPy version 1.16.2
- matplotlib version 3.0.3

B.1 Parseval Denoisers

To implement general Parseval Denoisers in TensorFlow we need two components, namely Parseval training (Algorithm 5.1, in particular Equation (5.11)) and optionally a trainable convex combination (Equation (5.13)).

All of the classes and functions described in this section is available as a library for Python 3 called `parsnet`. It is available from the Python Package Index (PyPI), and is thus installable with `pip`:[1]

```
$ pip install parsnet
```

The source code is available at the author's GitHub page[2] and is licensed under the free *Lesser GNU Public License version 3 (LGPLv3)*.

Equation (5.11)

Since version 1.5, TensorFlow has included a `Constraint` class and the associated `kernel_constraint` keyword argument for most layer constructors. We will use this framework for our implementation of Parseval training.

```
1  from tensorflow.python.keras.constraints import Constraint
2  from tensorflow.python.ops import math_ops, array_ops
3
4
5  class TightFrame(Constraint):
6      """
```

[1] Depending on your Python installation, you might need to run this command as root or with the `--user` flag. You will also have to specify `pip3` if Python 2 is your standard system version.

[2] `https://github.com/mathialo/parsnet`

```
    Parseval (tight) frame contstraint, as introduced in
    https://arxiv.org/abs/1704.08847

    Constraints the weight matrix to be a tight frame, so that the Lipschitz
    constant of the layer is <= 1. This increases the robustness of the network
    to adversarial noise.

    Warning: This constraint simply performs the update step on the weight matrix
    (or the unfolded weight matrix for convolutional layers). Thus, it does not
    handle the necessary scalings for convolutional layers.

    Args:
        scale (float):    Retraction parameter (length of retraction step).
        num_passes (int): Number of retraction steps.

    Returns:
        Weight matrix after applying regularizer.

    Raises:
        ValueError:       If input numbers are illegal
    """


    def __init__(self, scale, num_passes=1):
        self.scale = scale

        if num_passes < 1:
            raise ValueError(
                "Number of passes cannot be non-positive! (got {})".format(num_passes)
            )
        self.num_passes = num_passes


    def __call__(self, w):
        # CNN layers have 4D weight tensors, dense layers have 2D
        need_unfolding = (len(w.shape) == 4)

        # Do unfolding operator
        if need_unfolding:
            w_reordered = array_ops.reshape(w, (-1, w.shape[3].value))
        else:
            w_reordered = w

        # Do update step on (unfolded) weight matrix
        last = w_reordered
        for i in range(self.num_passes):
            temp1 = math_ops.matmul(last, last, transpose_a=True)
            temp2 = (1 + self.scale) * w_reordered
            temp3 = temp2 - self.scale * math_ops.matmul(w_reordered, temp1)

            last = temp3

        # Undo the unfolding to recast the output back to a 4D tensor if necessary
        if need_unfolding:
            return array_ops.reshape(last, w.shape)
        else:
            return last


    def get_config(self):
        return {"scale": self.scale, "num_passes": self.num_passes}
```

We can then make TensorFlow apply the the Parseval retraction step after each iteration during the training by passing an instance of the `TightFrame`[3] class to the `kernel_constraint` keyword argument of the layer constructor as such:

```
last_layer = tf.layers.conv2d(
    inputs=last_layer,
    kernel_size=(3, 3),
    filters=64,
    strides=(1, 1),
    padding="SAME",
    activation=tf.nn.relu,
    kernel_initializer=tf.initializers.orthogonal(),
    kernel_constraint=parsnet.constraints.tight_frame(0.001),
) / 3
```

[3] Aliased as `tight_frame` in the `parsnet` library for consistency with the rest of the TensorFlow API.

Note: Since TensorFlow version 1.13, this way of constructing neural network layers have been deprecated[4]. However, the `TightFrame` class is fully compatible with the new object-oriented API[5] as well.

Equation (5.13)

To implement a convex combination with a trainable convex parameter we will follow the idea from page 53:

```python
from tensorflow.python.ops import math_ops
from tensorflow.python.ops import variables
from tensorflow.python.framework import dtypes
import numpy as _np


def convex_add(input1, input2, initial_convex_par=0.5, trainable=False):
    """
    Do a convex combination of input1 and input2. That is, return the output of

        lam * input1 + (1 - lam) * input2

    Where lam is a number in the unit interval.

    Args:
        input1 (tf.Tensor):          Input to take convex combinatio of
        input2 (tf.Tensor):          Input to take convex combinatio of
        initial_convex_par (float): Initial value for convex parameter. Must be
                                     in [0, 1].
        trainable (bool):            Whether convex parameter should be trainable
                                     or not.

    Returns:
        tf.Tensor: Result of convex combination

    Raises:
        ValueError:     If initial_convex_par is outside of legal limit.
        TypeError:      If types are incorrect
    """
    # Find value for p, also check for legal initial_convex_par
    if initial_convex_par < 0:
        raise ValueError("Convex parameter must be >=0")

    elif initial_convex_par == 0:
        # sigmoid(-16) is approximately a 32bit roundoff error, practically 0
        initial_p_value = -16

    elif initial_convex_par < 1:
        # Compute inverse of sigmoid to find initial p value
        initial_p_value = -_np.log(1/initial_convex_par - 1)

    elif initial_convex_par == 1:
        # Same argument as for 0
        initial_p_value = 16

    else:
        raise ValueError("Convex parameter must be <=1")

    p = variables.Variable(
        initial_value = initial_p_value,
        dtype=dtypes.float32,
        trainable=trainable
    )

    lam = math_ops.sigmoid(p)
    return input1 * lam + (1 - lam)*input2
```

[4] https://www.tensorflow.org/versions/r1.13/api_docs/python/tf/layers#functions

[5] https://www.tensorflow.org/versions/r1.13/api_docs/python/tf/keras/layers#classes

B.2 Figures

We will now give short descriptions of how some of the figures in the thesis are generated, as well as provide example Python code. Throughout we will use the NumPy format for storing arrays (NPY)[6] for loading and saving matrices and images. The conversion from NPY to an image format, such as PNG, will be omitted.

Figures 2.1 and 2.3

Given an image $\mathbf{x}$, a sampling pattern Ω and a measurement operator $\mathbf{A} = \mathbf{P}_\Omega \mathbf{F}_n$, we compute the adjoint-recovered $\tilde{\mathbf{x}}$ as

$$\tilde{\mathbf{x}} = \mathbf{A}^* \mathbf{A}\mathbf{x} = \mathbf{F}_n^{-1} \mathbf{P}_\Omega^T \mathbf{P}_\Omega \mathbf{F}_n \mathbf{x}$$

In practice, we implement this as an FFT, zeroing our all the coefficients that are not in Ω, followed by an IFFT.

If `pattern.npy` contains a boolean matrix depicting Ω and `phantom.npy` depicts an image, the following code will compute the above equation:

```
import numpy as np

shepp_logan = np.load("phantom.npy")
sampling_pattern = np.fft.fftshift(np.load("pattern.npy"))

# F_n
samples = np.fft.fft2(shepp_logan)

# P^T P
samples[np.logical_not(sampling_pattern)] = 0

# F^{-1}
recon_adj = np.fft.ifft2(samples)

np.save("phantom_adjoint_recovery.npy", recon_adj)
```

Figure 2.2

To compute the DWT of images we will use the `tfwavelets` package for Python[7]. Given an image `lily.npy`, we can compute the DWT of the image using the functions in the `wrappers` module of `tfwavelets`:

```
import tfwavelets as tfw
import numpy as np

image = np.load("lily.npy")

level1 = tfw.wrappers.dwt2d(image, "haar", 1)
level2 = tfw.wrappers.dwt2d(image, "haar", 2)

np.save("lily_dwt_1.npy", level1)
np.save("lily_dwt_2.npy", level2)
```

[6] More information about NPY can be found at `https://www.numpy.org/devdocs/reference/generated/numpy.lib.format.html`

[7] A joint project between the author and Kristian Monsen Haug. Available at `https://github.com/UiO-CS/tf-wavelets`

Figure 3.4

A spline is essentially a piece-wise polynomial, the joint between consecutive polynomial pieces is often called a knot. To overfit data using splines we create a very dense knot vector, meaning the interval of each polynomial is very small. We are using cubic polynomials in each interval, and constraining the joints to have continuous 1st and 2nd derivatives to achieve a smooth spline. We are using the author's own library for spline computations[8]. The following code produces a plot similar to Figure 3.4b:

```
import numpy as np
import matplotlib.pyplot as plt
import splinelib as spl

# Draw points from a quadratic model, and add gaussian noise
xs = np.linspace(0, 4, 25)
ys = -xs**2 + 4*xs + np.random.normal(loc=0, scale=.4, size=xs.size)

# Create data matrix with data series as columns as splinelib demands
data = np.vstack([xs, ys]).T

# Create an overly general spline space using way too many knots
degree = 3
knot_number = 83
knots = spl.fit.generate_uniform_knots(
    spl.fit.cord_length(data),
    degree,
    knot_number
)

# Find the least squares fit using the above spline space
spline = spl.fit.least_squares(
    data,
    knots,
    degree
)

# New figure, add points and plot
plt.figure(figsize=[4, 3])
plt.scatter(xs, ys)
plt.plot(plotx, np.squeeze(spline(plotx)))
plt.tight_layout()
plt.savefig("overfit_fitted.pdf")
```

Figure 4.1

In this example we are using a pre-trained version of the ResNet-50 network in [He+16], trained on the ImageNet database [Rus+15]. To generate the adversarial noise we use the Foolbox package for Python [RBB17], which is a collection of many popular adversarial attack algorithms for classification networks. We also used Keras instead of TensorFlow since Keras comes with built-in pre-trained models. In order to get a proper misclassification we set a target class manually, and find a perturbation that will lead us there. The following code produces a perturbation similar to the one found in Figure 4.1:

```
import foolbox
import keras
import numpy as np
from labeldict import labels

# Get pretrained resnet model from TF
model = keras.applications.ResNet50()

# ResNet requires this weird preprocessing:
subtract = np.array([104, 116, 123])

# Create fooling object
```

[8] Available at `https://github.com/mathialo/splinelib`

```
13  foolmodel = foolbox.models.KerasModel(model, bounds=(0, 255), preprocessing=(subtract, 1)
        )
14
15  # Get sample image
16  image, label = foolbox.utils.imagenet_example()
17
18  # Apply attack (-1 is to reverse channel numbering to BGR as demanded by ResNet)
19  attack = foolbox.attacks.LBFGSAttack(foolmodel, criterion=foolbox.criteria.TargetClass
        (22))
20  adversarial = attack(image[:, :, ::-1], label)
21  new_label = np.argmax(model.predict(np.expand_dims(adversarial-subtract, 0)))
22
23  # Print labels
24  print("Original label:  {}".format(labels[label]))
25  print("Perturbed label: {}".format(labels[new_label]))
26
27  # Save results
28  np.save("deepfool_original.npy", image)
29  np.save("deepfool_perturbed.npy", adversarial[:,:,::-1])
```