KU LEUVEN

FACULTY OF SCIENCE

Super-Resolution of Solar Images using Generative Adversarial Networks

Adam Shamash

Supervisor: Prof. G. Lapenta
Centre for mathematical Plasma-Astrophysics
KU Leuven

Co-supervisor: Dr. J. Amaya
Centre for mathematical Plasma-Astrophysics
KU Leuven

Co-supervisor: Dr. D. González
Centre for mathematical Plasma-Astrophysics
KU Leuven

Thesis presented in
fulfillment of the requirements
for the degree of Master of Science
in Mathematics

Academic year 2017-2018

Super-Resolution of Solar Images using Generative Adversarial Networks

By

ADAM SHAMASH

Department of Mathematics
KATHOLIEKE UNIVERSITEIT LEUVEN

Thesis supervised by:

Prof. Giovanni Lapenta	KU Leuven
Dr. Jorge Amaya	KU Leuven
Dr. Diego González	KU Leuven

AUGUST 2018

Preface

Super resolution, the process of increasing image quality, is currently a promising field of research with a constant stream of new developments and many applications to scientific work. This thesis aims at preforming super resolution of solar images taken by the SOHO space mission through the use of wavelets and generative adversarial networks.

For helping to create this thesis, I would like to thank my supervisors, Prof. Giovanni Lapenta, Dr. Jorge Amaya, and Dr. Diego González, for providing the opportunity to explore and learn about such a new and interesting field and thank you for all your feedback and guidance. Thank you to Savvas and Gilles who have helped me obtain a deeper understanding of machine learning and its many applications. I wish you all the best on your future paths.

Most importantly, to my parents: all my life, you have always been there for me. I could not have been able to make it this far without your love and support. I cannot thank you enough.

And to all those that helped and guided me not mentioned: Thank you!

Summary

This thesis aims to create a mapping from low-resolution images of the Sun to the same image at a higher resolution, a process known as super resolution. This is done through the use of neural networks, a machine learning framework based on biological neural networks, and wavelets, special functions used to extract localized information about a signal.

More specifically, a two-stage neural network is proposed. The first stage, called the decomposer, attempts to learn the wavelet coefficients of our high-dimensional image from the low-dimensional input image. The second stage, called the recreater, takes the recreated wavelet coefficients and low-resolution input and combines them to recreate the final super-resolution image. It does so by mimicking the inverse wavelet transform for obtaining a high resolution image from a low-resolution image and a set of wavelet coefficients. Due to this structure, it becomes possible to directly compare the recreater with the inverse wavelet transform and allows the creation of a performance metric based on the network itself, rather than the super resolution images obtained. Both the decomposer and the recreater are effectively generative adversarial networks, making use of adversarial loss functions to judge the quality of the recreated wavelet coefficients and the final output. Data is taken from the SOHO space mission.

Once trained, the final network is able to recreate the overall structures found in the original high-resolution images from a low-resolution input, preforming particularly well in coronal hole regions of the Sun. The network does have some difficulty with dealing with static as well as some intricate details, especially in active regions.

List of Abbreviations

AD	Automatic Differentiation
AIA	Atmospheric Imaging Assembly
ConvNet	Convolutional Neural Network
EIT	Extreme ultraviolet Imaging Telescope
HR	High Resolution
LR	Low Resolution
GAN	Generative Adversarial Network
MLP	MultiLayer Perceptron
MPM	Model-Based Performance Metric
MSE	Mean Square Error
PSNR	Peak Signal-to-Noise Ratio
ReLU	Rectified Linear Unit
SDO	Solar Dynamics Observatory
SGD	Stochastic Gradient Descent
SOHO	Solar and Heliospheric Observatory
SR	Super Resolution
WC	Wavelet Coefficient

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Super resolution is the process of taking a low quality image and transforming it into a higher quality one. From crime dramas identifying the suspect on a cheap CCTV camera to science fiction trying to determine some unknown object, super resolution appears in our imaginations as a basic and reliable tool found in any arsenal of technologically advanced gadgets. Today, super resolution is an active topic field with promising new results. Much of this advancement has come through the use of machine learning techniques, particularly neural networks.

For solar researchers and scientists, super resolution provides an opportunity for high quality data of the Sun, both in the present as well as in the past when instrumentation was not up to modern standards. This will allow better models and a deeper understanding of the solar events and phenomenon. Additionally, since super resolution can be preformed remotely and cheaply, there will be less need for expensive and weighty hardware for observational missions.

This thesis aims to develop a super resolution method for solar images through the use of neural networks, a machine learning framework based on biological neurons. It does so by first estimating the image's wavelet coefficients, values which contain high frequency information about the image, and then provides a final high quality image based on these coefficients and the original low quality image. Convolutional layers are

used extensively and two adversarial networks are created to judge the quality of estimated wavelet coefficients and the final super resolution image. Training data is taken from SOHO, a NASA/ESA mission launch to study solar activity. All theory necessary to understand this methodology, as well as a brief discussion on super resolution, is presented in chapter 2.

Chapter 3 lays out the methodology used. Here, the reader will find information on how the data was preprocessed to focus on particular regions of the Sun. The architecture used as well as how a network is trained is explained. Once created, a performance metric is needed to judge the results of a network. Unfortunately, due to the ill-defined nature of super resolution, no good metric exists. To address this, a performance metric based on the network itself rather than the results is created and explained. The chapter closes with how to combine the results for different solar regions.

Final results can be found chapter 4. This includes hyper-parameters and network modifications tested, the network's performance on physically interesting features of the Sun, as well as implementation of a super resolution implementation for all regions of the Sun.

Finally, chapter 5 closes the thesis with a brief summary of the results as well as future improvements that can be done. For interested readers, the neural network code is found in Appendix A.

This thesis work is part of a larger project with the H2020 Project AIDA (www.aida-space.eu) funded by the European Community for predicting solar weather events through the use of neural networks [3][26].

Chapter 2

Theory

Here, the reader finds the necessary theory for understanding this thesis. Sec. 2.1 gives a very brief overview of the SOHO space mission, whose data this thesis relies on. The reader is then introduced to the two main mathematical tools, wavelets and neural networks, that this work uses to accomplish super resolution of SOHO images in Sec. 2.2 and 2.3. These three sections can be read independently of each other. The chapter closes with an overview about super resolution in Sec. 2.4. A shallow knowledge of neural networks is expected for this section.

2.1 SOHO Mission

The Solar and Heliospheric Observatory (SOHO) is a collaborative project between NASA and the European Space Agency aimed at providing a clearer understanding of the Sun and solar events (see Fig. 2.1). Stationed 1.5 million km away from Earth at the Earth and Sun's first Lagrangian point, the craft provides a constant view of the Sun for its twelve on-board instruments. These include CDS which measures coronal temperature and density, ERNE which measures composition of the solar wind, and MDI which measures magnetic and velocity field of the photosphere. SOHO is also equipped with the Extreme ultraviolet Imaging Telescope (EIT) operated by NASA (see Fig. 2.2). EIT looks

Figure 2.1: SOHO spacecraft (Credit: NASA/ESA)

extreme ultraviolet range around wavelengths 171 Å, 195 Å, 284 Å, and 304 Å to provide an invaluable view of the Sun's corona [24]. Until the launch of the Solar Dynamics Observatory (SDO) and its Atmospheric Imaging Assembly (AIA) in 2010, EIT provided researchers with the highest quality of JPEG coronal images [1].

Launched on December 2, 1995, the SOHO mission was scheduled to run only three years. Today, SOHO continues to provide scientists and researchers with information and images of the Sun.

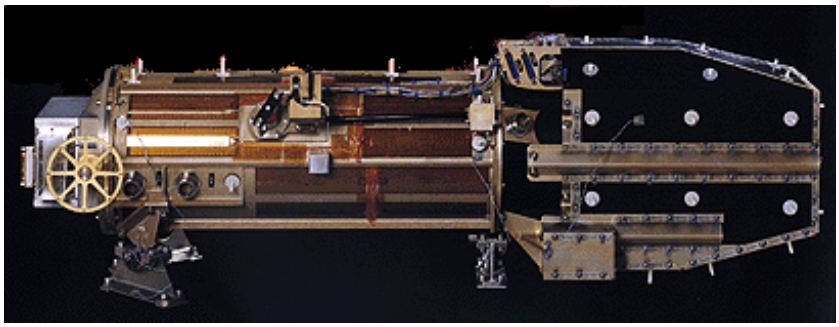

Figure 2.2: EIT (Credit: NASA/ESA)

2.2 Wavelets

When analyzing a signal, it is often beneficial to break it into meaningful components first. Traditionally, this has been done with Fourier analysis, which uses the fact that any signal $f(t)$ can be related to frequencies of sinusoidal functions. If f is periodic with period T, then f can be represented as a sum of sinusoids, i.e.

$$f(t) = \sum_{i=0}^{\infty} \left(a_i \sin\left(\frac{2\pi i t}{T}\right) + b_i \cos\left(\frac{2\pi i t}{T}\right) \right) \tag{2.1}$$

where $a_i, b_i \in \mathbb{R}$ are constants that encode how important a certain frequency is. In this way, frequency information for the signal as a whole can be obtained.

When looking at many real signals, such as sound recordings, the underlying frequencies may change with time and global frequency information is not as valuable. Instead data analysts may desire local frequency information, which can be found through the use of time-frequency atoms. Time-frequency atoms are special functions that are concentrated about a small neighborhoods. Similar to the Fourier transform with sinusoids, the idea is to represent the signal as a linear combination of various atoms. This representation would then hold the local information based on the neighborhoods of where the atoms are centered, rather then across all time. Various atoms exist, with a type called wavelets being one of the most common and useful.

2.2.1 Wavelet Decomposition

Wavelets are small oscillatory functions centered in space. To decompose a signal into wavelets, one first needs a basis of orthogonal wavelets $\{\psi_{s,n}\}_{s,n\in\mathbb{Z}}$. The functions $\psi_{s,n}$ are defined through some mother wavelet $\psi(t) \in \mathbb{L}^2(\mathbb{R})$ as

$$\psi_{s,n} = \frac{1}{\sqrt{2^s}}\psi\left(\frac{t-2^s n}{2^s}\right) \tag{2.2}$$

A mother wavelet should be such that $\int \psi(t)dt = 0$, $||\psi(t)|| = 1$, and ψ is centered near $t = 0$. Some examples of mother wavelets are shown in Fig. 2.3. The Haar mother wavelet, of particular interest for this thesis (see Chapter 3), is given by

$$\psi(x) = \begin{cases} 1 & : x \in [0, 1/2) \\ -1 & : x \in [1/2, 1] \\ 0 & : \text{otherwise} \end{cases}$$

Once the basis $\{\psi_{s,n}\}_{s,n\in\mathbb{Z}}$ is found, any signal $f(t) \in \mathbb{L}^2(\mathbb{R})$ can be decomposed into wavelets by

$$f(t) = \sum_{s,n\in\mathbb{Z}} c_{s,n}\psi_{s,n} \tag{2.3}$$

where the coefficients $c_{s,n} = \langle f, \psi_{s,n}\rangle$ are given by the inner product

$$\langle h, g\rangle = \int_{-\infty}^{\infty} h(t)g(t)dt \tag{2.4}$$

These coefficients are called wavelet coefficients (WCs). Eq. 2.3 is called the wavelet decomposition of $f(x)$. Note that the index s is related to the resolution of the wavelet, i.e. how large a neighborhood in time t the coefficient $c_{s,n}$ draws information from, while the index n is related to the position of the wavelet, i.e. where is the information contained in $c_{s,n}$ localized.

2.2.2 Multi-resolution Analysis and Father Wavelets

When working with real data, it is usually impossible computationally to work with all WCs. Additionally, we often only care about describing

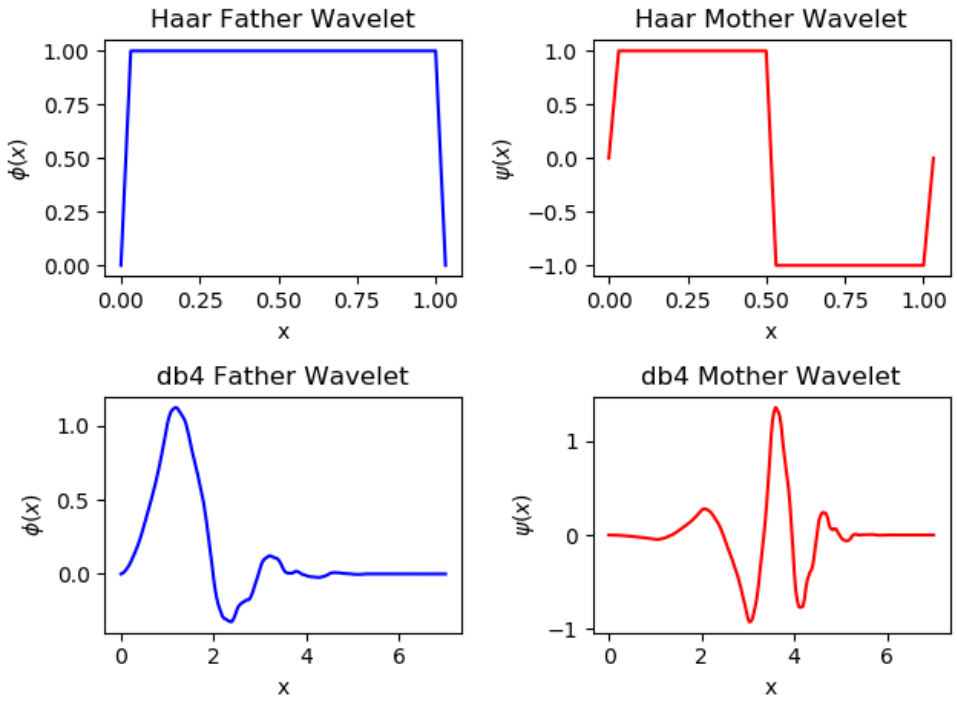

Figure 2.3: Examples of Wavelets: Haar and Daubechies 4

a particular degree of resolution, e.g. photographic images of certain pixel resolution. For working with different resolutions, it is useful to break up $\mathbb{L}^2(\mathbb{R})$ by resolution levels, called a multi-resolution analysis.

Definition 1. *A multi-resolution analysis is a sequence of subspaces of* $\mathbb{L}^2(\mathbb{R})$, $\{V_s\}_{s\in\mathbb{Z}}$, *such that*

1. $V_{s+1} \subset V_s$ *for all* $s \in \mathbb{Z}$
2. *There exists a function* $\phi \in \mathbb{L}^2(\mathbb{R})$ *such that* $\{\phi_{0,n}\}_{n\in\mathbb{Z}}$ *forms an orthonormal basis of* V_0, *where* $\phi_{s,n} = \frac{1}{\sqrt{2^s}}\phi\left(\frac{t-2^s n}{2^s}\right)$
3. *For all* $s \in \mathbb{Z}$, $f(t) \in V_s$ *if and only if* $f\left(\frac{t}{2}\right) \in V_{s+1}$
4. $\bigcap_{j\in\mathbb{Z}} V_j = \{0\}$
5. $Closure\left(\bigcup_{j\in\mathbb{Z}} V_j\right) = \mathbb{L}^2(\mathbb{R})$

The function $\phi(t)$ is called the father wavelet or scaling function.

By properties (1)-(3), it is trivial to show that $\{\phi_{s,n}\}_{s,n\in\mathbb{Z}}$ forms a basis for V_s [6]. In other words, the basis functions that make up V_s affect a domain twice as large i.e. the basis functions become half as precise, losing the ability to represent changes in small domains. V_s is said to have a scale of 2^s where V_0 is defined to have a scale of 1. The inverse of scale is called resolution so that V_s is said to have a resolution of 2^{-s}.

Once a multi-resolution analysis is created with defining father wavelet, any signal $f(t)$ can be approximated as f^s with a resolution 2^{-s} by projection of f onto V_s, i.e.

$$f^s(t) = \sum_{n\in\mathbb{Z}} d_{s,n}\phi_{s,n} \tag{2.5}$$

where the coefficients are given by $d_{s,n} = \langle f, \phi_{s,n}\rangle$. These coefficients represent a discrete low resolution (LR) version of the signal with the desired resolution [21]. In this way, the father wavelets act a low-pass filter bringing the signal to the desired resolution level.

A simple example of a father wavelet is the Haar father wavelet, given by

$$\phi(x) = \begin{cases} 1 & : x \in [0,1] \\ 0 & : \text{otherwise} \end{cases}$$

For going between resolution levels, the orthogonal complements of the subspaces V_s are also needed. Let W_{j+1} be the orthogonal complement of V_{s+1} in V_s, i.e. $V_s = V_{s+1} \oplus W_{s+1}$. By using the father wavelet, it is possible to construct an orthonormal basis for W_{s+1}, $\{\psi_{s+1,n}\}_{n\in\mathbb{Z}}$, where the functions $\psi_{s,n}$ are the same wavelets described above [21]. In the multi-resolution analysis associated with the Haar father wavelet, the wavelet basis $\{\psi_{s_n}\}$ is generated by the Haar mother wavelet.

Through the use of wavelets, any signal with a resolution 2^{-s}, $f(x) \in V_s$, can have its resolution doubled by adding the appropriate wavelets to Eq. 2.5, i.e.

$$f^{(s-1)}(t) = f^s(t) + \sum_{n\in\mathbb{Z}} c_{s,n}\psi_{s,n}(t) \tag{2.6}$$

where $f^{(s-1)}(t)$ is an approximation of f at resolution $2^{-(s-1)}$. The WCs thus holds information about high-frequency details of a signal.

2.2.3 Discrete Signals

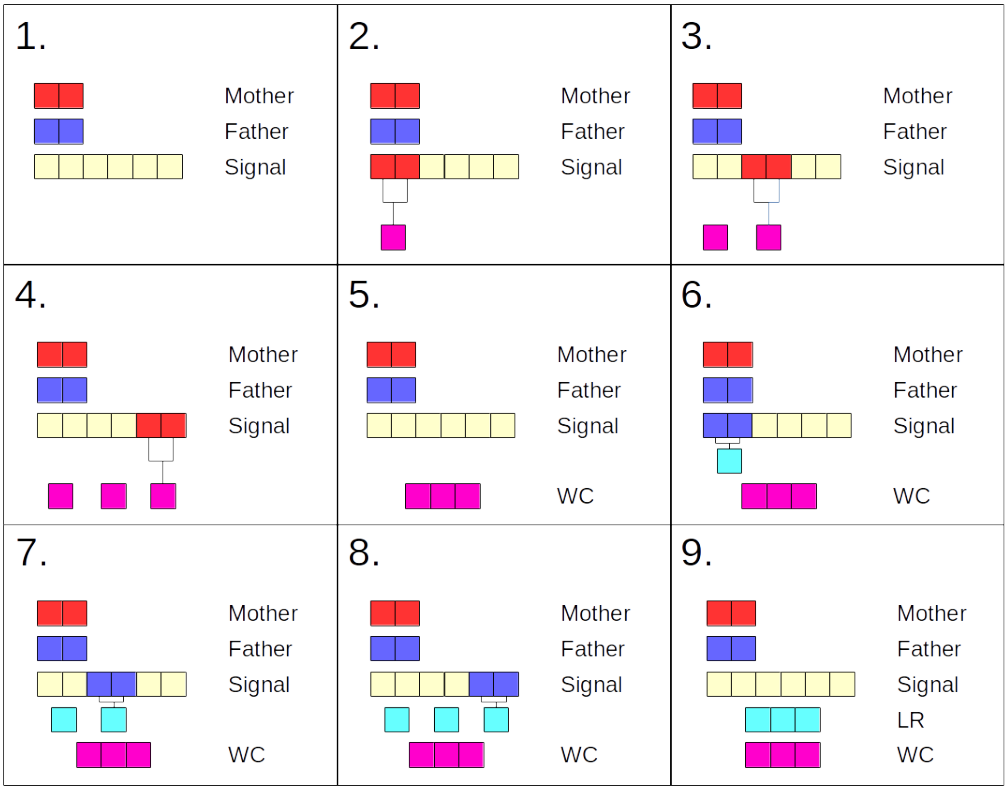

Figure 2.4: Discrete 1-D Wavelet Decomposition on discrete signal with length 6 done with the Haar Wavelet

In practice, signals are given by some discrete function $f^0[t] \in V_0$ where $t \in \mathbb{Z}$ and $f^0[t]$ is known only for $0 \leq t \leq T$ for some $T \in \mathbb{N}$. In such cases, the theory outlined above still holds, but discrete wavelets must be used and summations replace integrals. The discrete Haar father wavelet is given by

$$\phi(x) = \begin{cases} \frac{1}{\sqrt{2}} & : x = 0 \text{ or } x = 1 \\ 0 & : \text{otherwise} \end{cases}$$

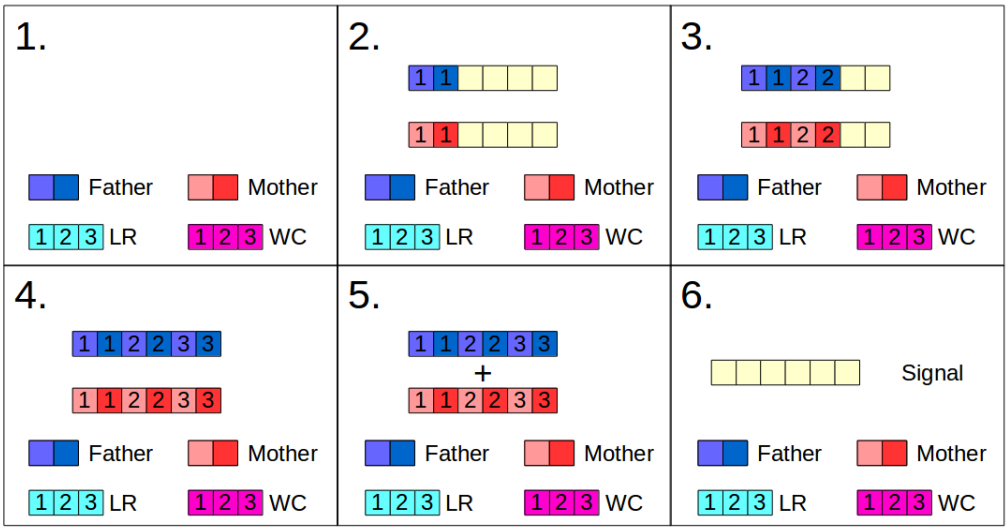

Figure 2.5: Discrete 1-D Wavelet Inverse Decomposition recreating discrete signal with length 6 done with the Haar Wavelet

and the discrete Haar mother wavelet is given by

$$\psi(x) = \begin{cases} \frac{1}{\sqrt{2}} & : x = 0 \\ -\frac{1}{\sqrt{2}} & : x = 1 \\ 0 & : \text{otherwise} \end{cases}$$

Calculation of the LR and WC values is done with the discrete convolution. The discrete convolution between discrete functions $f[t]$ and $h[t]$ is given by

$$f \star h[n] = \sum_{i=-\infty}^{\infty} f[i]h[n - \delta i] \tag{2.7}$$

where δ is called the stride of the convolution. Then, the WC values can be calculated by $c_{s,n} = f^{(s)} \star \psi[n]$ and LR values by $d_{s,n} = f^{(s)} \star \phi[n]$ where both convolutions use a stride of $\delta = 2$. In other words, a discrete approximation of resolution 2^{-s} can be decomposed into

$$f^{(s)}[t] = f^{(s)} \star \phi[t] + f^{(s)} \star \psi[t] \tag{2.8}$$

where $f^{(s+1)}[t] = f^{(s)} \star \phi[n]$ is the discrete LR approximation of resolution $2^{-(s+1)}$ and $f^{(s)} \star \psi[t]$ are the wavelet coefficients. See Fig. 2.4 for visualization of this process.

Note that each time the resolution is the signal is halved, the known indexes are also halved. In other words, there is an implicant rescaling in the indexes of the returned coefficients. For wavelets whose discrete representation has a length of more than 2 or for signals with odd T, some assumptions about the unknown portions signal must be made. Common assumptions include all unknown values are zero and $f[n] = f[0]$ for $n < 0$ and $f[n] = f[T]$ for $n > 0$.

The inverse of this decomposition is to apply Eq. 2.3 and is shown visually in Fig. 2.5.

The above procedure describes the process for 1-dimensional signals. For signals of more than a single dimension, such as images or videos, the 1-D wavelet transform is preformed along each dimension sequentially. For example, when working with 2-D images, a 1-D wavelet transform is first preformed along the rows of the image and then along the columns, as shown in 2.6 This gives three sets of WCs and one set of LR values. The WC sets are labeled LH, HL, and HH for applying father wavelet followed by mother wavelet, mother wavelet followed by father wavelet, and mother wavelet followed by mother wavelet respectively. The inverse process is then the same as the 1-D inverse wavelet transform applied to the different axes in the revers order as the 1-D wavelet transforms were preformed.

2.3 Neural Networks

Recent years have seen remarkable growth in the development and deployment of various AI techniques. Of these, neural networks in particular have shone brightly, completing previously unimaginable tasks such as translation between languages [40], playing games [22], and even projecting human thoughts [31].

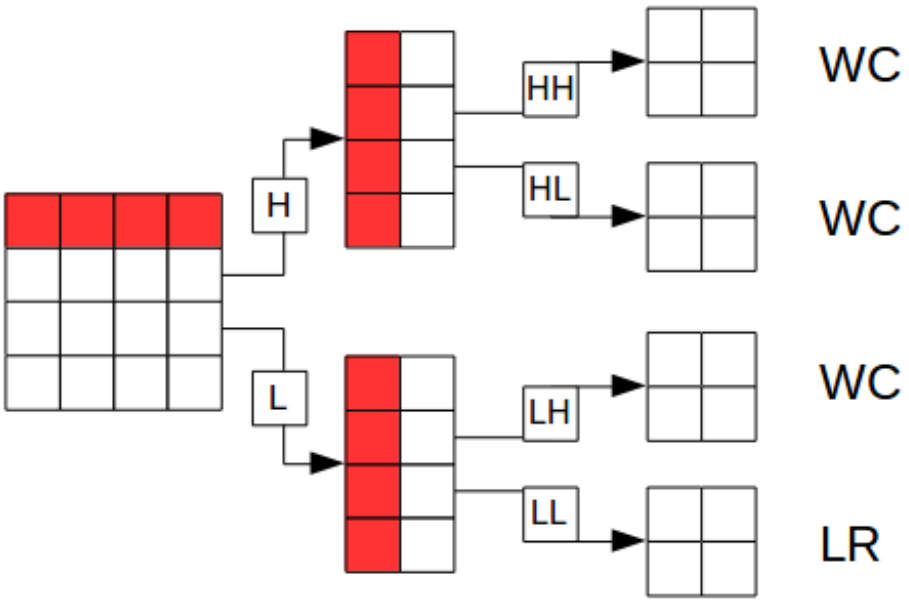

Figure 2.6: 2-D wavelet transform

2.3.1 Neurons and Neural Networks

Put simply, a neural network is some function $\mathcal{N}$ which maps an input x to some z which seeks approximate some true function $\mathcal{N}^*$ through the use of artificial neurons [8]. An artificial neuron, usually shortened to neuron and also called a unit, takes inspiration in biological neurons which take in multiple electric signals as inputs, determine how important the combination these inputs are, and produces a single output based on the strength of its inputs.

An artificial neuron is a simple vector to scalar function that operates by taking a linear combination of the input vector and applying an activation function to it, i.e. the neuron maps input $\vec{x} \in \mathbb{R}^n$ to output $a = g(\vec{w}^T\vec{x} + b)$, where $\vec{w} \in \mathbb{R}^n$ is a vector of weights, $b \in \mathbb{R}$ is some bias term, and g is some activation function (see Fig. 2.7). The inputs to a neuron are called features and the output of a neuron, a, is also called an activation. Common activation functions include the sigmoid, softmax, and hyperbolic tangent. Of particular importance is

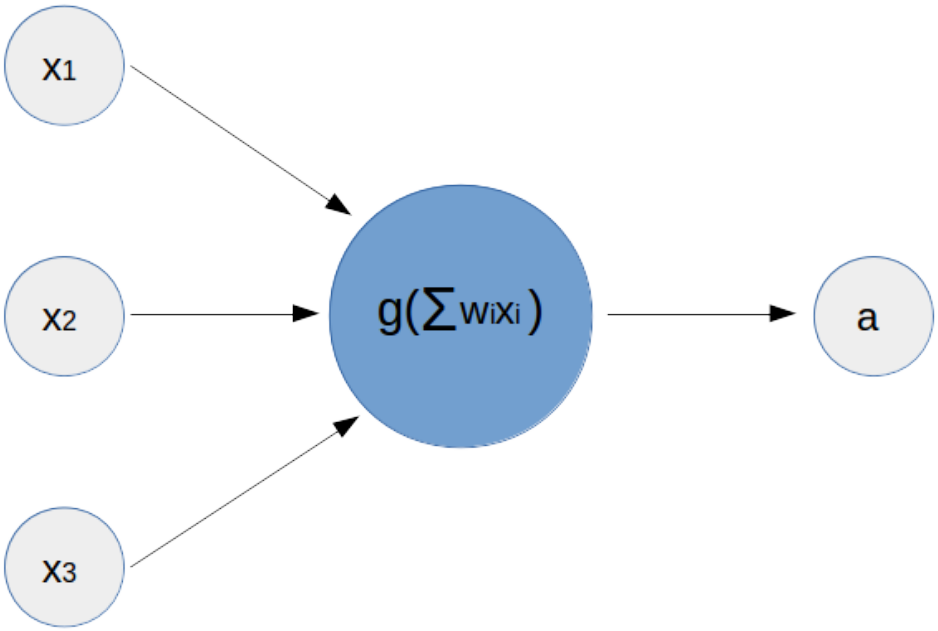

Figure 2.7: Artifical Neuron

the rectifier function, which maps x to $\max(0, x)$. A neurons using the rectifier is called a Rectified Linear Unit or ReLU (see Fig. 2.8), with ReLU also being used to describe the rectifier function itself depending on the context. ReLUs have been shown to improve training of networks [23]. Variations of ReLUs are also commonly used, such as leaky ReLUs which maps x to $\max(0.01 * x, x)$, allowing some negative inputs to "leak" into the output.

Neurons can be combined in various forms to form neural networks. A simple example of a neural network is the multilayer perceptron or MLP. In MLPs, neurons are placed into groups called layers and the layers are placed in a sequential order such that neurons of the same layer take as inputs only outputs of the preceding layer. In practice, neurons of the same layer also use the same activation function. Thus, each layer is simply a vector to vector function and the network is the composition of layer functions. As an example, consider the MLP $\mathcal{N}$ mapping $\mathbb{R}^2$ to $\mathbb{R}$ shown in Fig. 2.9. The first layer, labeled 'I' is the input layer and contains the inputs $\vec{x} \in \mathbb{R}^2$ of the network. The second layer, 'H', contains three neurons, each taking the values from 'I' as

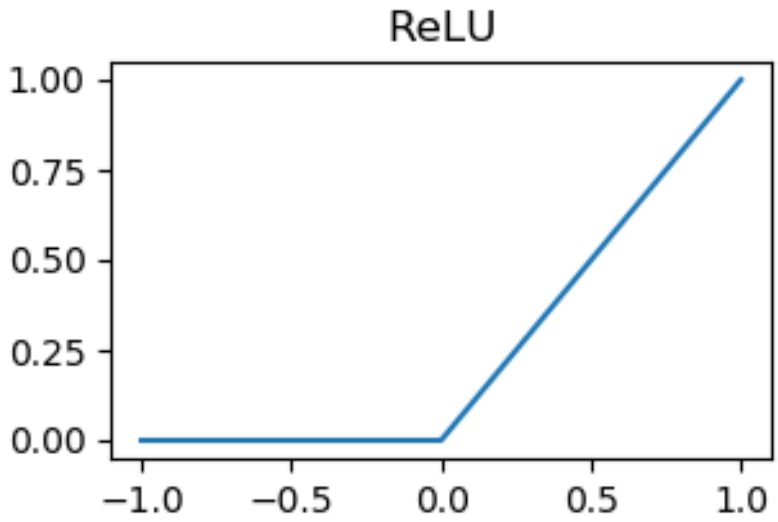

Figure 2.8: ReLU Function Plot

input and producing a scaler output. Thus, the layer 'H' is simply a function h mapping $\mathbb{R}^2$ to $\mathbb{R}^3$. Note that each neuron in 'H' contains 2 weight parameters and a possible bias term, giving 'H' a total of 9 possible parameters. Since the output of this layer is not seen by the user, this is called a hidden layer. The third and final layer 'O' is the output layer and contains only a single unit, whose output is the output of network. Similar to 'H', the entire layer 'O' can be thought of as a function q mapping $\mathbb{R}^3$ to $\mathbb{R}$. A total of 4 parameters are contained in layer 'O'. Thus, the figure shows a network $\mathcal{N}$ with only a single hidden layer and a depth of three, i.e containing a total of three layers, with $\mathcal{N}(\vec{x}) = q(h(\vec{x}))$. The number of layers, the number of units in each layer, and the activation function describe the architecture of $\mathcal{N}$ and training of $\mathcal{N}$, also called learning, is the process of finding the 'best' values for the 9+4=13 parameters to describe some data. If 'best' can be judged by comparing known inputs with known outputs, the learning process is called supervised learning, as opposed to unsupervised learning which only judges based on input data and reinforcement learning which has a special focus on output data. In this work, only supervised learning is considered and discussed.

MLPs are specific example of the more general feed-forward networks, which take some input and run it through a series of layers to produce the

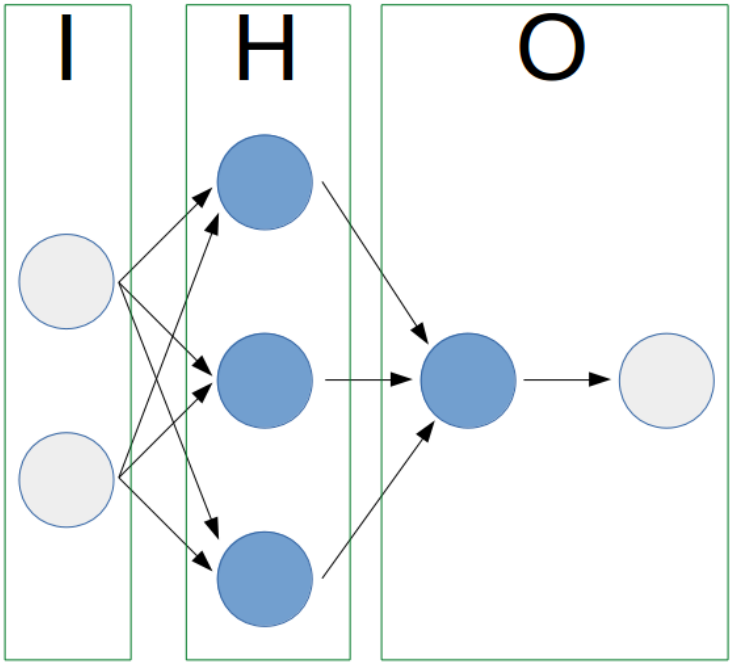

Figure 2.9: A simple MLP

desired output. The layers used in MLPs are called fully connected layers due to every neuron using all the available inputs. Other feed-forward network architectures, may use layers that only connect portions of the network or may introduce different types of connections. Some of these are discussed below (see Sec. 2.3.3).

The description of a neural network is often surprisingly simple, even why they work so well is not fully understood. In this work, $\mathcal{N}$ will always denote the network under consideration, which takes in an input $x \in \mathbb{R}^I$ and produces an output $z = \mathcal{N}(x) \in \mathbb{R}^O$. The $\mathcal{N}$ network is composed of n layers denoted $h_1 \dots h_n$, i.e. $\mathcal{N} = h_n \circ h_{n-1} \circ \cdots \circ h_1$. The set of parameters describing $\mathcal{N}$ is given by $\Theta = \Theta_1 \cup \cdots \cup \Theta_n$, where Θ_i is the set of parameters associated with layer h_i. When working with $\mathcal{N}$, it is often beneficial to look at the outputs of the individual layers, which shall be denoted $a_i = \{a_i^{(k)}\}$ for the output of the i^{th} layer. The function which produces a_i from x is $\mathcal{N}_i = h_i \circ h_{i-1} \circ \cdots \circ h_1$. Note that $a_0 = x$ and $a_n = z$.

2.3.2 Training

Simply put, neural networks represent a statistical regression problem, with the architecture of the network describing the statistical model used and the corresponding parameters, the weights and bias terms in the individual neurons, calculated to best fit the data.

To do this, the data is usually first divided into three groups: a training set, a validation set, and a test set. Actual training of a neural network is done with the training set (see below). Normally, it is not known beforehand which network architecture and hyper-parameters best model the data and so multiple networks are trained. Each network is then tested on the validation set using some performance metric, such as accuracy, and the network which obtains the best value on this performance metric is chosen. Final performance of a network is judged by how well it preforms on the test set. Although the validation and test sets appear to have identical roles i.e. judging the performance of the network, it is important to keep the two separate. The network chosen by the validation set is the best for the validation data and so is biased for the validation data. A separate test set is needed to provide an unbiased assessment of the network. How to divide the data into these three sets depends on multiple factors but the training set is generally the largest of the three, accounting for around 80% of the data, while validation and test sets are often of equal size.

As with any regression problem, a first step is to describe some loss function $\mathbb{L}$ which calculates how closely the outputs $Z = \{z_1, \ldots, z_n\}$ produced from inputs $X = \{x_1, \ldots, x_n\}$ of the network $\mathcal{N}$ fit the true data $Z^* = \{z_1^*, \ldots, z_n^*\}$. Which loss function is best ultimately depends on what the network is expected to learn, but common loss functions include the mean square error and cross entropy loss, as well as other neural networks (see Sec. 2.3.4).

Optimization

Once a loss function is chosen, the parameters Θ associated with $\mathcal{N}$ need to be calculated to minimize $\mathcal{L}$ through some optimization scheme. The most basic scheme is gradient decent, an iterative scheme that seeks to

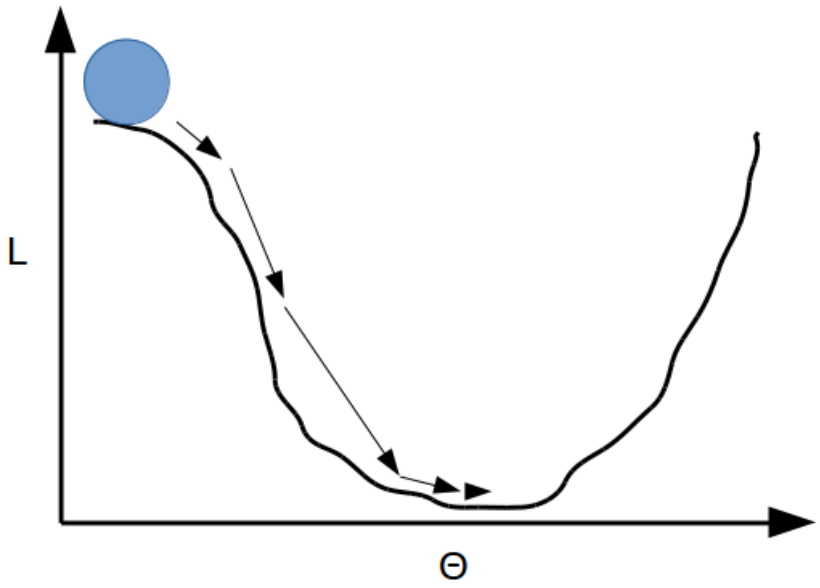

Figure 2.10: Gradient decent acts like rolling a ball down a hill

minimize $\mathcal{L}$ by following the path of greatest decrease (see Fig. 2.10). To apply, a user must first choose some initial set of parameters $\Theta^{(0)}$. Next, one calculates the negative gradient $-\nabla_\Theta \mathcal{L}(\Theta^{(0)})$, which gives the direction in Θ-space that $\mathcal{L}$ decreases fastest. One then steps in this direction to obtain a new parameter vector $\Theta^{(1)} = \Theta^{(0)} - \lambda \nabla_\Theta \mathcal{L}(\Theta^{(0)})$, where the learning rate $\lambda > 0$ is a hyper-parameter chosen by the user which describes how large of a step is taken each iteration. Learning rates which change with each iteration may also be used. Gradient decent is summarized with the iterative rule

$$\Theta^{(i+1)} = \Theta^{(0)} - \lambda \nabla_\Theta \mathcal{L}(\Theta^{(i)}) \tag{2.9}$$

Iteration stops when either $|\Theta^{(i+1)} - \Theta^{(i)}| < \epsilon$ for some $\epsilon > 0$ or more commonly after $s > 0$ iteration steps.

The loss and its gradient depend on the data and, given the enormous size of modern data sets and the complexity of neural networks, calculating the loss and gradient for the whole dataset is often computationally infeasible. To get around this, methods that approximations the loss

and gradient are used. The simplest is Stochastic Gradient Decent, or SGD. Rather than use the entire dataset, SGD chooses one random data point $z_i \in Z$ and, with its corresponding $z_i^* \in Z^*$, calculates $\mathcal{L}$ and $\nabla_\Theta \mathcal{L}$ as if this point was all the data available. Doing so, each iterative step is unlikely to descend in the direction of greatest decent but descend in the direction of greatest decent on average. The method is also computationally inexpensive, with the computational cost of calculation independent of the size of the full dataset once a data point is chosen.

A middle ground between the ideal gradient decent and the more feasible SGD is mini-batch gradient decent. Rather than picking a single point, a random subset $Z_i \subset Z$, called a mini-batch, of the dataset is chosen each iteration step and $\mathcal{L}$ and $\nabla_\Theta \mathcal{L}$ are calculated as if Z_i and its corresponding $Z_i^* \subset Z^*$ consisted of all the data. The cardinality Z_i is called the batch size. Common batch sizes include 16 and 32. Although the points in SGD and mini-batches in mini-batch gradient decent are chosen randomly, they are often chosen so that all the data is cycled through. A single cycle is called an epoch. Note that, in the literature, SGD often refers to mini-batch gradient decent and for the rest of this work, SGD will refer to mini-batch gradient decent.

Other more complex optimization schemes are used in training neural networks. These methods can involve a more complicated use of the gradient (Adam [16]), use previous gradient values (SGD with momentum [5]) or treat parameters in a more individual fashion (AdaGrad [13]). Similar to SGD, these methods can use mini-batches and require knowledge of the gradient of the loss function. The choice of an optimizer and its hyper-parameters like learning rate affect how quickly a local minimum for the loss is reached as well as which local minimum the the network tends towards [37].

Backward Automatic Differentiation

In the optimization schemes mentioned above, knowledge of the gradient of loss function is necessary. When using neural networks, this gradient is usually calculated with backward Automatic Differentiation (AD). The general idea behind AD is to calculate derivative values for parameters through the use of the chain rule. Backward AD starts

in the last layer of the network and use these results to calculate the derivatives for the parameters in the second to last layer. This process continues through the layers until the derivatives for the final weights associated with the front of the network are obtained and the entire gradient is calculated.

The goal is to calculate the gradient of $\mathcal{L}$ with respect to Θ at input x. To accomplish this, the output z and activations a_i's of the network are calculated by running x through $\mathcal{N}$. This process is called forward propagation. We then begin calculating the gradient through the chain rule by first calculating $\frac{\partial \mathcal{L}}{\partial z}$. Since $\mathcal{L}$ is just a simple analytic function in z, this calculation is trivial. Next, we seek $\frac{\partial z}{\partial \Theta} = \frac{\partial h_n(...h_1(x))}{\partial \Theta}$. Since only layer h_n depends on Θ_n,

$$\frac{\partial h_n(\dots(h_1(x))\dots)}{\partial \Theta_i} = \begin{cases} \frac{\partial h_n}{\partial \Theta_n} a_{n-1} & : i = n \\ \frac{\partial h_n}{\partial a_{n-1}} \frac{\partial h_{n-1}(\dots(h_1(x))\dots)}{\partial \Theta_i} & : i < n \end{cases}$$

Using the fact that the parameters Θ_i are only found in layer h_i, we can apply the above relationship recursively to obtain

$$\frac{\partial h_n(\dots(h_1(x))\dots)}{\partial \Theta_i} = \Delta_i \frac{\partial h_i}{\partial \Theta_i} a_{i-1} \tag{2.10}$$

where Δ_i is defined recurrently by

$$\Delta_n = 1$$

$$\Delta_i = \frac{\partial h_{i+1}}{\partial a_i} * \Delta_{i+1} \text{ for } i < n$$

Hence, multiplying Eq. 2.10 by $\frac{\partial \mathcal{L}}{\partial z}$ yields the desired gradient:

$$\frac{\partial \mathcal{L}}{\partial \Theta_i} = \frac{\partial \mathcal{L}}{\partial y} * \Delta_i \frac{\partial h_n}{\partial \Theta_i} a_{i-1} \tag{2.11}$$

The above expression require knowledge of $\frac{\partial h_i}{\partial \Theta_i}$ and $\frac{\partial h_i}{\partial a_{i-1}}$. Fortunately, since layers generally consist of analytic activation functions and simple

summations over weights, their calculation is generally trivial. See Sec. 2.3.5 for their calculation in fully connected layers.

The process of calculating the gradient with backward AD and preforming an optimization step is called back-propagation.

2.3.3 Layers

As mentioned above, the layers of a neural network can take on various different shapes. Of particular note are residual blocks [10], convolutional layers [18], upsampling layers, and batch normalization layers[12].

Residual Layers

In theory, neural networks of large depth should be able to learn a dataset better than shallower networks with fewer parameters. Yet, in practice, training deep networks for longer often leads to worst results. This has been attributed to deep networks having difficulty implicitly learning functions close to the identity mapping. Residual layers seek to overcome this challenge by allowing a network to focus on learning residual mappings.

To implement, let $h_i, \dots, h_{i+m}$ be layers in a neural network taking input x and producing output $x' = h_{i+m}(\dots h_i(x) \dots)$. The residual block is formed by simply combining this input and output, i.e. the output of the residual block is given by $y = x + x'$ (see Fig. 2.11). The connection between x and x' is an example of a shortcut connection. The residual layer is the addition of x and x'. Note that x and x' must be the same dimensions in this formulation, but it is possible to create residual blocks where the the dimensions do not match by taking a linear projection of x during the shortcut connection.

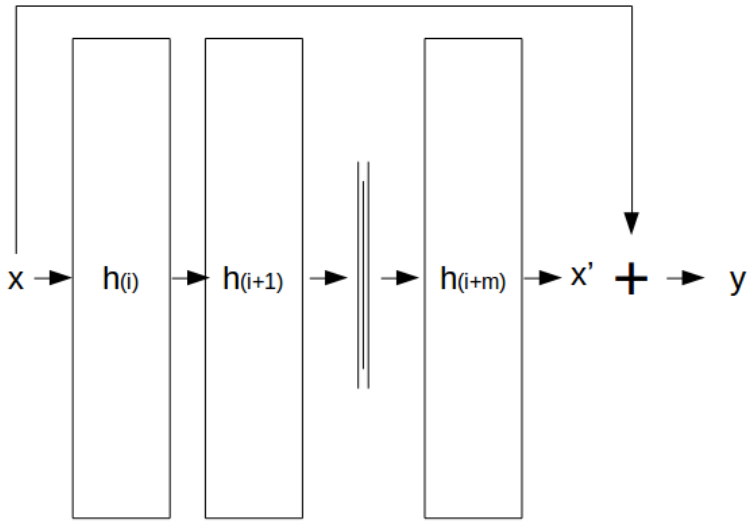

Figure 2.11: Residual Block

Convolutional Layers

Convolutional layers work by convolving some set of trainable parameters, called a kernel, across the inputs to produce the output. A simple example of this can be seen in Fig. 2.12. Here, we have a 1-D 5 element input vector $[a, b, c, d, e]$ and a 1-D 2 element kernel $[x, y]$. The kernel first match with the start of the input and the product of each of the matching elements is found. The summation of all these products forms the first output value $ax + by$. The kernel is then moved over by one and the process is repeated until the kernel reaches the end the the input. Since the kernel is moved one space at a time, the layer is said to have a stride of 1. Since the kernel only convolved on the given input, the convolution is said to be valid. In practice, it is often desirable to keep the output the same size as or a simple ratio to the size of the input. To accomplish this, the input may be enlarged with padding so that the output is the desired size when a valid convolution is applied. Common paddings include zero padding, where all the padding has value 0, and same padding, where the padded values match the value of the nearest input value.

The convolutional layer so far described is a 1-dimensional convolutional

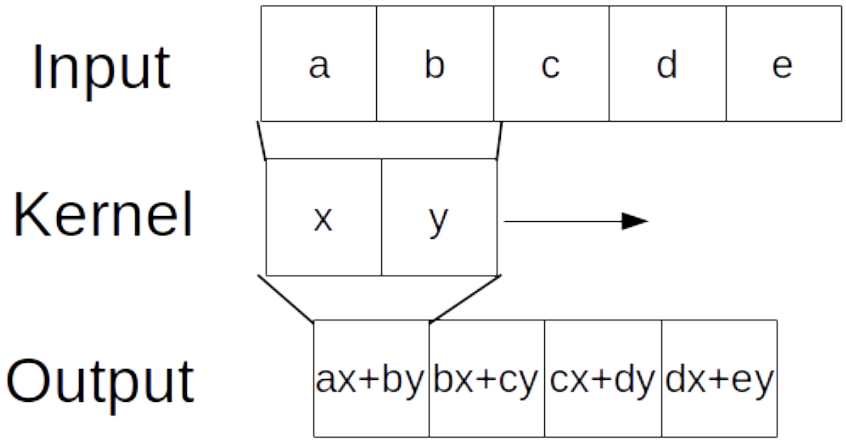

Figure 2.12: 1-D Convolutional Layer

layer. Convolutional layers of higher dimensions are also possible and more commonly used. In such layers, the input is the desired dimension, as are the kernel and stride. The output is produced in just the same way as for the 1-dimensional case, with the kernel convolving along all relevant axes of the input. This process produces an output of the same dimensionality as the input (see Fig. 2.13 for an example of a 2-D convolutional layer).

Input and output data often have more than a one channel, e.g. images often use three channels to hold color information. To allow for the possibility of multi-channel inputs, kernels are expanded so that each channel aligns in the same way with the kernel. The convolution than acts in the same way as for the single channel case. If multi-channel outputs are desired, multiple kernels of the same shape are applied onto the input until the desired number of channels are obtained.

Note that, unlike the convolutions described in Sec. 2.2, convolutional layers generally do not apply a 'true' convolution but instead apply a procedure called cross-correlation. The difference between convolution and cross-correlation is that the kernel is flipped in the former, e.g. Fig. 2.12 depicts cross-correlation whereas convolution would need a kernel given by $[y, x]$ to produce the same output. In the literature, as well as this text, the term convolution is used even if cross-convolution is the

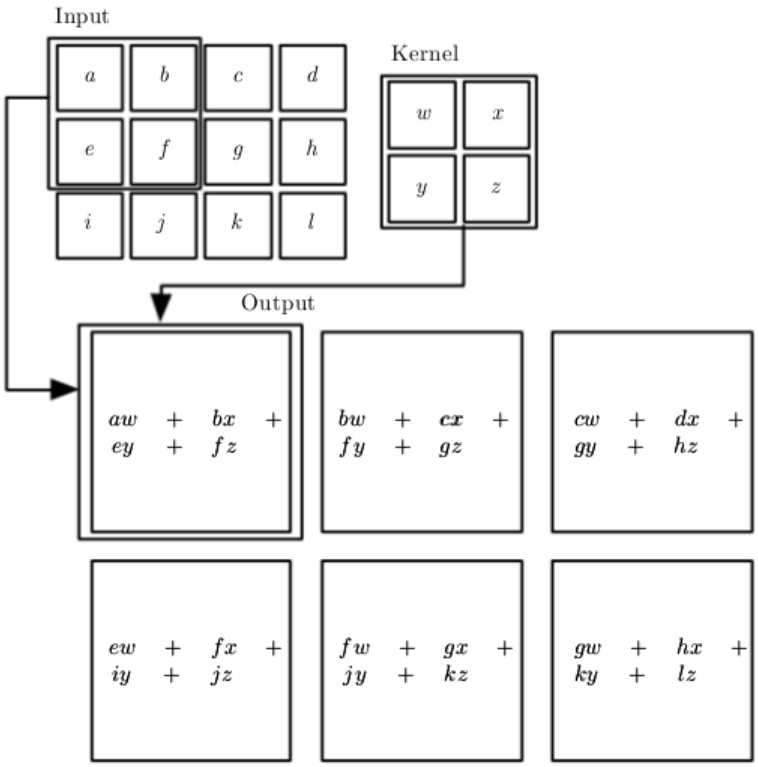

Figure 2.13: 2-D Convolutional Layer (Credit: *Deep Learning* by Goodfellow et al. (2016))

actual process.

Convolutional layers have numerous advantages compared to fully connected layers. The first is a smaller number of parameters to train, leading to faster training, less memory needed to store a network, and the possibility for deeper networks, which have been shown to produce better results [34]. The number of parameters in a fully connected layers is equal to the product of number of inputs into a layer and number of outputs from that layer and this can easily exceed values such as 10^5 parameters in the most simple of tasks. In contrast, the number of parameters of convolution layers depend only on the size of the kernel chosen. Kernels containing as little as 9 or 25 parameters are commonly used. The convolutions preformed by the kernel is also an efficient method of extracting information from an input. For instance, in object detection applications, the pixels that make up the desired

object are often concentrated in an image. Hence, rather than calculate output based on the entirety of an image, calculating values locally, as done in convolutional layers, would work well and be computationally more efficient. Calculation of features such as edges are many times local and commonly found with kernel-based methods [2][28]. Finally, all information needed for a convolutional layer is saved in the kernel, which is of a fixed shape independent of input and output size (excluding channels). Hence, networks that use only convolutional layers can take input of arbitrary size (although number of channels still needs to match that needed by the kernel).

A network which uses convolutional layers is called a convolutional neural network or ConvNet.

Upsampling

In designing neural networks, it is sometimes desirable to produce an output of greater size than the input. For fully-connected layers, this is a trivial issue, with number of neurons determining the size of the output independent of input size. ConvNets, however, while providing a natural method of decreasing lowering output size through higher stride sizes, lack a way of increasing the size of an input as it is fit through a network. One simple fix is to first upscale input, using methods such as linear interpolation, and feeding the upcaled input into the network [4]. While this leads to outputs of the desired size, training the network on a larger input increases training time without introducing new information. Recent networks avoid this by creating specific layers that upsample their inputs mid-network. By upsampling mid-networks using methods such as interpolation and nearest-neighbor, the network can produce outputs of the desired shape with a lower training time [32].

Batch Normalization

In neural networks, the ideal values for parameters in a layer h_i depend on the parameter values in the layers that come before h_i. As the network is trained, these parameters change and the data distribution for inputs to h_i also change. Called internal covariate shift, this process

leads to longer training times and a need for greater attention in tuning the network. Batch normalization layer seek to fix this by normalizing activation values during each optimization step.

Batch normalization seeks to address internal covariate shift in mini-batch training by normalizing the activation values in network. For example, let h_i be a batch normalization layer. Then, each h_i aims to take each individual input and normalizes it based on the mini-batch statistics, i.e.

$$h_i^* \left(a_{i-1}^{(k)} \right) = \frac{a_{i-1}^{(k)} - \mu^{(k)}}{\sigma^{(k)}} \tag{2.12}$$

where $\mu^{(k)}$ and $\sigma^{(k)}$ is the mean and variance of $a_{i-1}^{(k)}$ respectively found using the mini-batch values. Unfortunately, normalizing each activation separately causes the output of h_i to have a different meaning than its input. To address this, a scaling $\gamma^{(k)}$ and a shifting parameter $\beta^{(k)}$ are added to batch normalization layers for each input. These parameters are trainable. The full batch normalization layer is then given by

$$h_i \left(a_{i-1}^{(k)} \right) = \gamma^{(k)} h_i^* \left(a_{i-1}^{(k)} \right) + \beta^{(k)} \tag{2.13}$$

$$= \gamma^{(k)} \left(\frac{a_{i-1}^{(k)} - \mu^{(k)}}{\sigma^{(k)}} \right) + \beta^{(k)} \tag{2.14}$$

The parameters $\beta^{(k)}$ and $\gamma^{(k)}$ allow to batch normalization to act as the identity. Hence, the input and output have the same meaning if it is ideal.

Batch normalization has been shown to decrease training time and lead to better results predicted by models utilizing it [12]. Note that, since batch normalization introduces two parameters for each input, a batch normalization layer only works with inputs of a fixed size, unlike the convolution and residual layers described above.

2.3.4 Generative Adversarial Networks

One of the recent innovations in AI research has been the creation of Generative Adversarial Networks or GANs. Developed in 2014 by

Goodfellow et al, [9] GANs provide AI practitioners with a simple and powerful generative model and are used in a wide range of applications, including image generation from text descriptions, [27] denoising of astronomical images, [30], and image blending [38].

A GAN consists of two networks: a generative network and an adversarial network. The generative network, also called the generator, aims to create realistic generated objects that can fool the adversarial network into thinking they are real. At the same time, the adversarial network, also called the discriminator, seeks to tell true values from the fake output of the generator. In this way, the two networks are locked in constant competition.

More formally, the two networks are engaged in a two-player minimax game. Let G be the generator taking noise input x and let D take either generated values $G(x)$ or real values z and output the probability that the given input is real. The value function of the game, $V(G, D)$, can be described by

$$V(G, D) = \mathbb{E}_z[\log D(z)] + \mathbb{E}_x[\log(1 - D(G(x)))] \tag{2.15}$$

where the $\mathbb{E}_z$ and $\mathbb{E}_x$ refers to the expectation values with respect to the probability distributions of the true inputs z and noise x respectively. The generator aims to minimize this value while the discriminator aims to maximize it. The full game is then described by $\min_G \max_D V(G, D)$. Note that, while for our purposes G and D are neural networks, this game system allows G and D to be any type of generative model and differentiating model.

To implement, a user alternates between training the generator and training the discriminator. A basic algorithm for accomplishing this is shown below in Algorithm 1.

Algorithm 1 GAN Training: Single training step to update generator G and discriminator D using minibatches

Require: k: training steps for discriminator per iteration
Require: X: noise data
Require: Z: true data
Require: m minibatch size
Require: $\mathcal{L}_{\mathcal{G}}$: loss function for generator
Require: $\mathcal{L}_{\mathcal{D}}$: loss function for discriminator
Require: U_G: iterative optimization procedure for generator
Require: U_D: iterative optimization procedure for discriminator
1: sample m samples $\{x_1 \dots x_m\}$ from X
2: Update generator G by minimizing $\mathcal{L}_{\mathcal{G}}$ using U_G for one iteration
3: **for** k steps **do**
4: sample m samples $\{x_1 \dots x_m\}$ from X
5: generate $\{G(x_1) \dots G(x_m)\}$
6: sample m samples $\{z_1 \dots z_m\}$ from Z
7: Update generator D by minimizing $\mathcal{L}_{\mathcal{D}}$ using U_D for one iteration

For the loss functions $\mathcal{L}_{\mathcal{G}}$ and $\mathcal{L}_{\mathcal{D}}$ associated with G and D respectively, binary cross entropy can be used:

$$\mathcal{L}_{\mathcal{G}}(x) = -\log[D(G(x))] \tag{2.16}$$

$$\mathcal{L}_{\mathcal{D}}(y) = \begin{cases} \log[1 - D(y)] & : y = G(x) \\ \log[D(y)] & : y = z \end{cases} \tag{2.17}$$

Intuitively, the adversarial network acts as a loss function for the generator based on the 'realness' of the output. The generator loss may also incorporate other loss components not connected to the discriminator to account for different properties. The portion of the generator's loss associated with the discriminator is called the adversarial loss.

2.3.5 Software Implementation

To design neural network code, a programmer would first describe all values such as weights, inputs, and outputs as tensors. A tensor is

a generalization of a matrix that can take on any dimension. A 1-dimensional tensor is simply a vector while a 2-dimensional tensor is a matrix. As an example, consider a neuron. If a neuron takes inputs $a_1, \dots a_I$, these can be grouped into a vector $\mathbf{a} = [1, a_1, \dots, a_I]^T \in \mathbb{R}^{(I+1)}$. The initial 1 in $\mathbf{a}$ is to account for a bias term. The parameters of the neuron can then be described by $\mathbf{\Theta} = [\Theta_0, \Theta_1, \dots, \Theta_I] \in \mathbb{R}^{(I+1)}$ where Θ_0 is the bias term and the rest are weights. The output of the neuron is then just $g(\mathbf{\Theta}^T\mathbf{a})$ where g is the activation of the neuron. The notation to describe a fully connected layer is identical, but with $\mathbf{\Theta}$ a 2-dimensional tensor of size $K \times L$, $K = I + 1$, were the k^{th} row containing the weights connected to the k^{th} input and the l^{th} column containing the weights contained in the l^{th} neuron and connected to the l^{th} output. This makes it trivial to calculate and express the quantities $\frac{\partial h_i}{\partial \Theta_i}$ and $\frac{\partial h_i}{\partial a_{i-1}}$ discussed above in Sec. 2.3.2 when discussing backward AD. Taking simple derivatives, we find $\frac{\partial h_i}{\partial \mathbf{\Theta}_i} = \mathbf{a}_{i-1} g'(\mathbf{\Theta}_i^T \mathbf{a}_{i-1})^T$ and $\frac{\partial h_i}{\partial a_{i-1}} = \mathbf{\Theta}_i g'(\mathbf{\Theta}_i^T \mathbf{a}_{i-1})$. Methods to operate on and change tensors should also be developed by the programmer.

Once all relevant quantities are described clearly, a programmer creates a computational graph. A computational graph orders operations in the sequence they should be implemented. This allows not just running an input through the correct sequence of layers, but also determines the order for executing backward AD for training the network.

Finally, a user will need to implement all the necessary functions to build and train a network. This includes common activation functions and their derivatives, as well as various optimizers to train the network.

In creating neural networks, numerous packages exist to aid users. Popular python packages such PyTorch, Keras, Tensorflow, Theano, and many more can be divided into two groups: those that implement all the mathematics themselves (e.g. PyTorch) and those that seek to provide a simpler API by building upon the former (e.g. Keras). These packages often contain visualization tools to help users better understand the training procedure and see what can be improved. In this work, the package PyTorch is used [25].

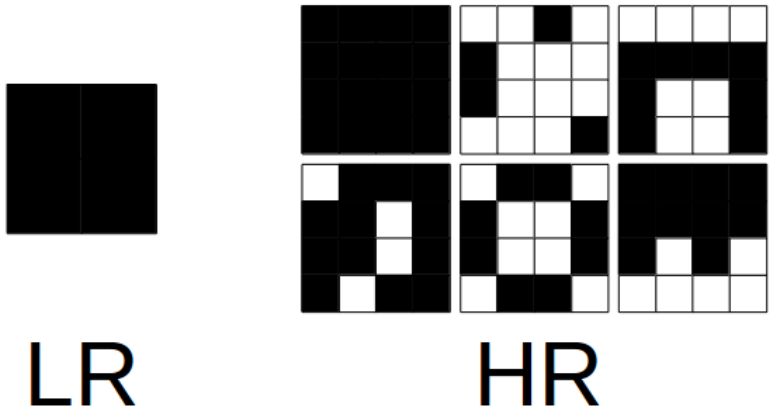

Figure 2.14: When all of the HR images on the right are downsampled using maxpooling, they all produce the same LR image found on the left.

2.4 Super Resolution

Super resolution or SR is the problem of finding a mapping from a low resolution (LR) image to its high resolution (HR) counterpart. Since a single LR image can correspond to multiple HR images, an exact, definitive solution is unobtainable (see Fig. 2.14). Despite this, immense progress has been made in recent years in creating SR mappings which produce realistic HR images from LR images. Here, only SR techniques which take a single image as input, called single-image super resolution, are considered.

Numerous methods of SR have been proposed, such as the use of nearest neighbor methods, [7] statistical regression methods, [15] and wavelets [33][36]. Of particle interest in recent years has been the use of ConvNets, which have been shown to obtain state of the art results [4]. Current methods aim on improving learning finer details such as textures.

One approach is to have the network learn the wavelet coefficients associated with the HR image from the LR image and use this be recreate the HR image. Since wavelet coefficients contain high frequency information, these networks are effectively focused on learning these

fine features. SR ConvNets incorporating wavelets converge quickly and demonstrate remarkable performance [11][17].

Another approach to learn finer details is the creation of better loss functions. These loss functions tend to be summations of simpler loss functions. Like almost all SR ConvNets, a pixel-wise Mean Square Error (MSE) is included to help the network learn the average structure. The pixel-wise MSE is given by

$$\text{MSE} = \frac{1}{WH} \sum_{i,j} (I_{SR}[i,j] - I_{HR}[i,j])^2 \tag{2.18}$$

where $I_{SR}[i,j]$ and $I_{HR}[i,j]$ are the pixel intensity at pixel in the i^{th} row and j^{th} column for the SR and HR images respectively and the summation is over all possible pixel indexes. Perceptual losses are also commonly included. Perceptual losses seek to penalize the network if high-level features found in the HR and SR images do not match. To do so, these high-level features are extracted with some function, generally some pretrained neural network, and the loss is then constructed [14]. A feature-wise MSE is a commonly used for perceptual loss functions. Adversarial losses are also found in this loss summation. They represent a natural inclusion given realism is one of the main goals for SR techniques. Networks employing some combination of these losses and others have lead to extremely realistic SR results [19][29][39].

Chapter 3

Methodology

In this chapter, the results of this thesis work are presented to the reader. This thesis aims to preform super resolution of SOHO images through the use of a neural network. Special focus is paid to pixels in the center of the image. This chapter begins with an overview of what SOHO data is used in Sec. 3.1 followed by the preprocessing done to prepare it for use by the neural network in Sec. 3.2. The network architecture used is explained in Sec. 3.3. The performance metric used to judge between different models and hyper-parameters is introduced in Sec. 3.4 and the way models are trained is explained in Sec. 3.5. Finally, an explanation of how the methodology used can be expanded for a well-preforming super resolution of the entire Sun ends the chapter in Sec. 3.6.

The computational resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government - department EWI.

3.1 Data and Task

The aim of this thesis work is the super resolution of solar images. A method of doubling resolution is presented, accomplished through the use of a ConvNet network incorporating an adversarial loss. As input,

the network will take full images of the Sun of resolution 512×512. This work aims to output the same image at a resolution 1024×1024, where the HR image comes from the SOHO spacecraft at a wavelength of 171 Å, as shown in Fig. 3.1. All images are all single-channeled (i.e. gray-scale) and are described with a 2-dimensional tensor, I_R, where inputs $R = LR$ and WC sets $R = LH$, $R = HL$, and $R = HH$ have a height and width of $H \times W = 512 \times 512$ and the output $R = SR$ and true HR image $R = HR$ have a height and width of $2 * H \times 2 * W = 1024 \times 1024$. The i^{th} row, j^th column pixel is denoted (i, j) and its intensity is given by $I_R[i, j]$, with $I_R[0, 0]$ the intensity of the upper left-hand corner of an image. Pixel intensities range from 0 for pure black to 255. Any corrupted data is removed before preprocessing [3].

3.2 Preprocessing

Images of the Sun contain various structures that are of interest to solar researchers, from flares to prominences to sunspots. Visually, these structures are seen against a background of the solar surface when they occur near the center of an image or empty space when they occur near the edges of an image. This difference in background creates a fundamental difference in the distribution of pixel intensities these two regions.

I assume that the SOHO images can be visually divided into three regions: solar center, solar edge, and empty space. Let $\rho(i, j) = \sqrt{(i - H/2)^2 + (j - W/2)^2}$ the distance a pixel is from the center of the image. The solar center consists of portions of the Sun which the solar surface can be seen as background and is defined as such for all pixels (i, j) such that $\rho \leq \rho_c$ where $\rho_c = 0.825 * H$. In other words, the solar center consists of the all pixels found within a distance of 82.5% the height and width of the image from the center of the image. The majority of the Sun found in the image is contained in this region.

The solar edge consists of the portions of the image where solar activity is visible but where the solar surface may not be. This is defined as all pixels such that $\rho_{e1} \leq \rho \leq \rho_{e2}$ where $\rho_{e1} = 0.775 * H$ and $\rho_{e2} = 1.125 * H$. The solar edge is the outer edge of the Sun and shares some overlap with

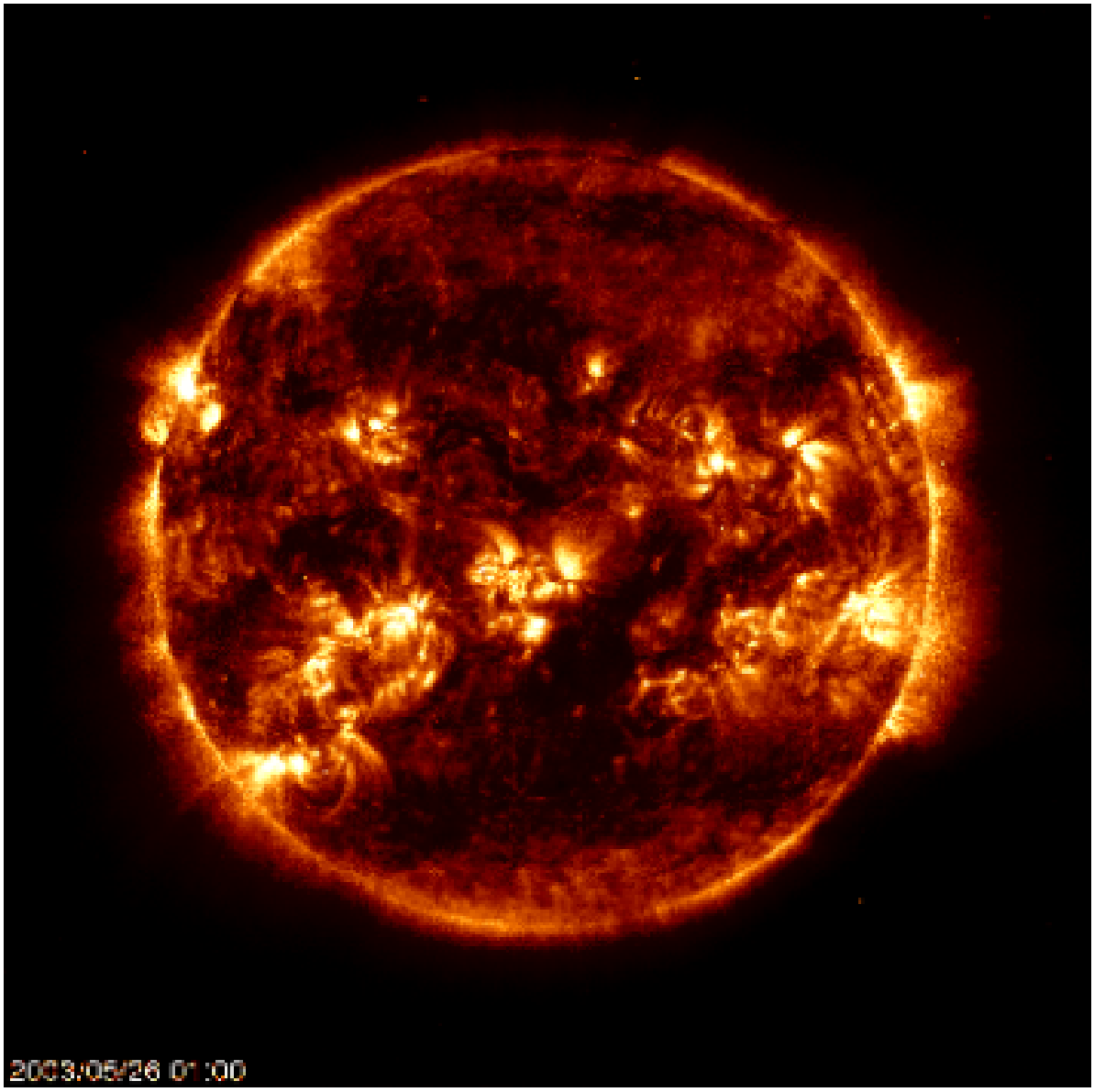

Figure 3.1: SOHO Image of Sun, resolution 1024×1024 (Credit: NASA)

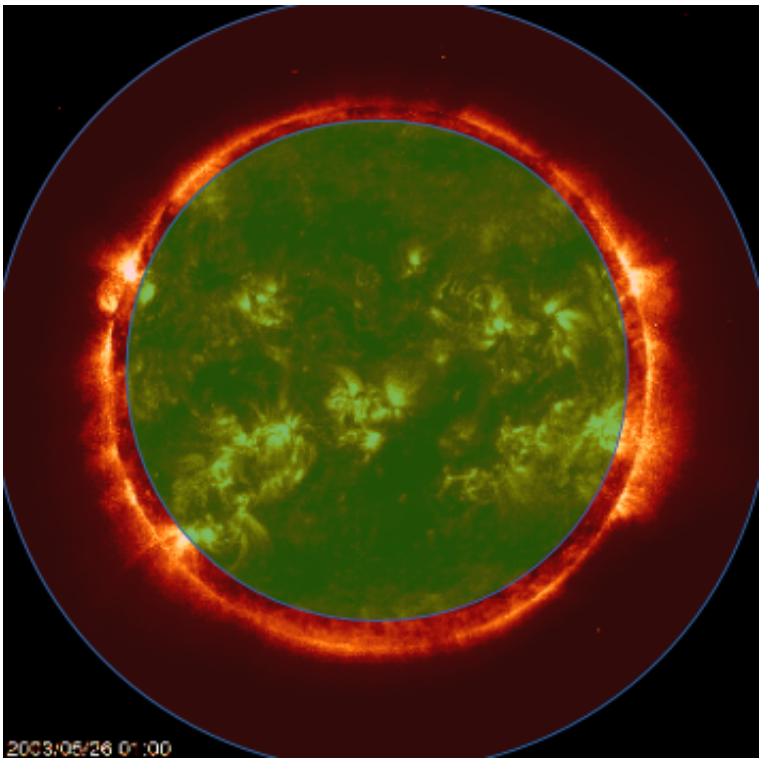

Figure 3.2: Solar Regions: Green shows the solar center and red shows the solar edge. The rest of the image is empty space and ignored.

the solar center. The remaining interesting portions of the image are found in this region.

The remaining pixels of the image almost always display the darkness of space. These pixels form the third region: empty space. Fig. 3.2 shows a visualization of these regions, with the green area showing the solar center, the red area showing the solar edge, and empty space the rest. This thesis work primarily looks at super resolution of the solar center with some focus on the solar edge. Since empty space is just primarily black pixels devoid of solar activity, it is ignored completely.

The final neural network architecture works by taking an LR image, producing the WCs associated with the HR image, and combining the LR image and WC sets to produce the HR image in a way analogous to an inverse wavelet transform (see Sec. 3.3). The network consists solely of layers which allow inputs of various resolution, such as convolutional

layers (see Sec. 3.3). This allows training to be done with the use of small patches taken from the full solar image and the network still being able to preform on the full 512×512 image. The use of patches may lead to a decrease in performance as the network is not trained to preform well on the entirety of the Sun or the entirety of one of its regions. Their use, however, allows focusing on a particular region of the Sun as well as a decrease in computational time. Additionally, since arbitrary sizes can be fed into the network, an image may be passed to the network multiple times to increase its resolution by more than a factor of two. Note that this has not been tested and should be done with care as network training only involved data of the starting resolution. Retraining the network to work with the higher resolution data first would be more appropriate.

In order to create the training data, two patches are selected from each 1024×1024 full sized HR image of the Sun (see Fig. 3.3). The central pixel of the first patch is chosen with $i \leq H/2$ and the central pixel of the second patch is chosen so that $i \geq H/2$. In other words, the first patch comes from the Sun's northern hemisphere while the second comes from the southern hemisphere. Both patches are chosen so that all their pixels are within the solar center. This allows training to focus on the solar center. The patch size was chosen to be 128. Patches of this size appear able to fully contain most major structures, such as sunspots or flares, found on the sun. This HR patch is then normalized by dividing all pixel intensities by 255, leading to intensities between 0 and 1.

Once a HR patch is created, a LR patch and three WC patches each of resolution 64×64 are produced by taking the 2-dimensional discrete wavelet transformation of the HR patch. This is preformed with the python package PyWavelets [20]. The Haar wavelet is used for this transformation due to the simplicity of its transformation. The Haar wavelet has also been shown to preform well in wavelet-based SR neural networks [11]. Since the pixel values in the HR patch range from 0 to 1, values in the LR patch range from 0 to 2. To keep the pixel intensities between 0 and 1, the LR patch is normalized by a factor of 2. See Fig. 3.4 for a summary of the preprocessing steps.

Training patches are only taken from the solar center, and so the completed network will focus on preforming well only in the solar center. Most interesting activity can be found in this region and using training

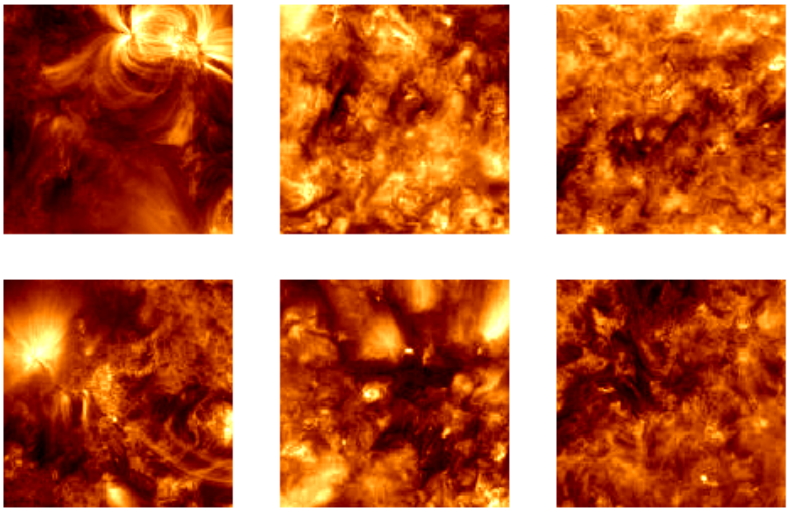

Figure 3.3: Examples of HR patches

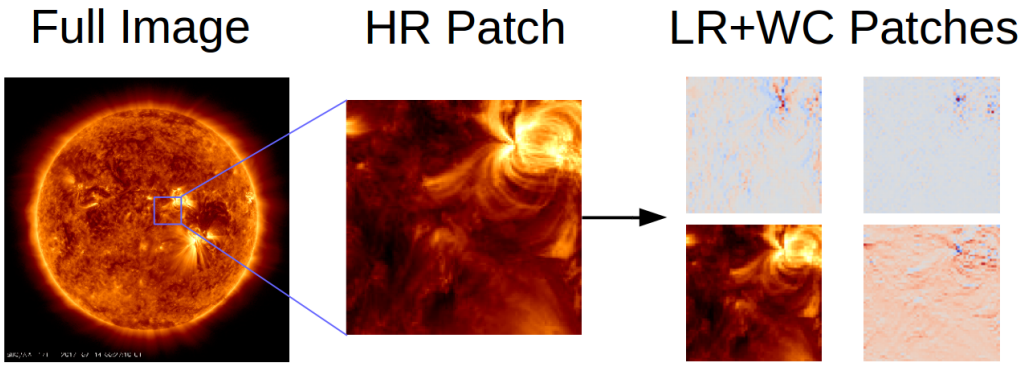

Figure 3.4: Summary of Preprocessing

patches only from this region should lead to better results than using patches from anywhere on the Sun. This comes at the cost of decreased performance for SR of the solar edge. For a well-trained SR method of the entire Sun, see Sec. 3.6

It should be noted that the LR patches thus produced depend on the wavelet used and may not match up with the LR images produced by real space instruments. Their use is necessary for the metric used to judge SR images (see Sec. 3.4). Additionally, since LR images produced by wavelets are weighted averages of pixels in the HR image near where the corresponding LR pixel will be located (e.g. the LR image produced by the Haar wavelet the set of averages of every 2×2 set of pixels in the HR image), the LR images used should not differ substantially from the true image.

3.3 Network Architecture

The neural network used consists of two sections: the decomposer and the recreater. The network starts with the decomposer, which takes as input the LR image and seeks to recreate the three corresponding WC sets. Once this is done, the LR image is united with the WC sets and run through the recreater to form the HR image. A loss function $\mathcal{L}^i$ is associated with both the decomposer ($i = W$) and the recreater ($i = H$), defined by

$$\mathcal{L}^i = \alpha^i \mathcal{L}^i_{MSE} + (1 - \alpha^i)\mathcal{L}^i_{GAN} \tag{3.1}$$

where $\mathcal{L}^i_{MSE}$ is a pixel-wise MSE loss, $\mathcal{L}^i_{GAN}$ is an adversarial loss given by Eq. 2.16, and $\alpha^i \in [0, 1]$ are hyper-parameters that determine the contribution of each loss component. The loss is thus designed so the network learns the overall structure of the WC sets and HR images with the MSE losses while producing realistic results for both through the use of adversarial losses. Note that WC intensities range from -1 to 1 and HR intensities range from 0 to 1, leading to $\mathcal{L}^W_{MSE}$ to have possible values from 0 to 4 and $\mathcal{L}^H_{MSE}$ to have values from 0 to 1. The adversarial losses can range from 0 to infinity in theory, but have an ideal value of

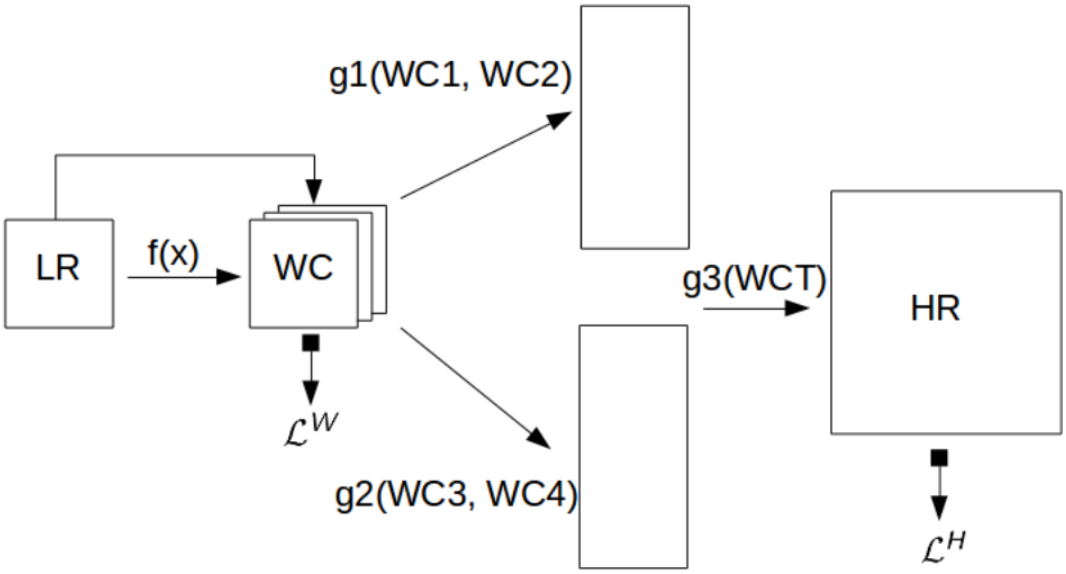

Figure 3.5: Overview of Network

about 0.69 [9]. Additionally, values where the sum $\mathcal{L}^W_{GAN} + \mathcal{L}^H_{GAN}$ is above 5 are heavily discouraged (see Sec. 3.5).

The full loss $\mathcal{L}$ for the neural network is a linear combination of $\mathcal{L}^W$ and $\mathcal{L}^H$ and is given by

$$\mathcal{L} = \alpha\mathcal{L}^W + (1-\alpha)\mathcal{L}^H \tag{3.2}$$

$$= \alpha(\beta\mathcal{L}^W_{MSE} + (1-\beta)\mathcal{L}^W_{GAN}) + (1-\alpha)(\gamma\mathcal{L}^H_{MSE} + (1-\gamma)\mathcal{L}^H_{GAN}) \tag{3.3}$$

where $\alpha, \beta, \gamma \in [0, 1]$ are hyper-parameters to be chosen in training. Note that since there is an adversarial loss for both the decomposer and the recreater, the entire training system contains a total of three neural networks. A visual overview of the network can be found in Fig. 3.5 and implementation was done through the use of PyTorch [25]. Complete code can be found in Appendix A.

Decomposer

The basic architecture for decomposer can be found in Table 3.1. In this table, the column 'Layer' displays the type of layer used by the network.

If a convolutional layer, 'Conv', is used, the kernal size is also listed. The column 'Input' lists the shape of the input if a convolutional layer is used (width$\times$ height $\times$ channels) or which layer's outputs are combined if a residual layer is used. The column 'Output' lists the shape of the output of the layer.

Inspired by the EnhanceNet network, the decomposer is composed solely of convolutional layers and residual layers so that, once trained, the network will be able to operate on inputs of various sizes, including complete images of the Sun [29]. In order to keep inputs and outputs the same width and height after a convolutional layer, zero padding is added to the layer's input. Each convolutional layer contains a bias term. A ReLU activation is applied after each convolutional layer except the last two (layer 9 and layer 10). Note that when an input is run through the network, by the time it reaches the end a single pixel sees all pixel intensities within a 19×19 box of which it sits at the center of. This means that about 52% of all pixels in the WC sets are generated without being influenced by padding and 71% are uninfluenced by padding after going through half the decomposer.

#	Layer	Input	Output
1	Conv 5x5	$64 \times 64 \times 1$	$64 \times 64 \times 64$
2	Conv 3x3	$64 \times 64 \times 64$	$64 \times 64 \times 64$
3	Conv 3x3	$64 \times 64 \times 64$	$64 \times 64 \times 64$
4	Residual	(1)+(3)	$64 \times 64 \times 64$
5	Conv 3x3	$64 \times 64 \times 64$	$64 \times 64 \times 64$
6	Conv 3x3	$64 \times 64 \times 64$	$64 \times 64 \times 64$
7	Residual	(4)+(6)	$64 \times 64 \times 64$
8	Conv 3x3	$64 \times 64 \times 64$	$64 \times 64 \times 32$
9	Conv 3x3	$64 \times 64 \times 32$	$64 \times 64 \times 3$
10	Conv 3x3	$64 \times 64 \times 3$	$64 \times 64 \times 3$

Table 3.1: Decomposer Structure

Recreater

The recreater takes the LR input and WC sets (LH, HL, HH) generated by the decomposer and creates the final SR output. Before put into the recreater, though, the LR input is scaled by a factor of 2 to undo the normalization preformed in preprocessing (see Sec. 3.2).

Convolutions are then preformed along the rows of the inputs to combine the LR and LH sets and the HL and HH sets producing two matrices of size $2 * H \times W$. The same kernel is used to combine both. Convolutions of the same dimension, but with kernels shaped to act along the columns rather than the rows, combine these two matrices into the final output: I_{SR} of shape $2 * H \times 2 * W$.

The convolutions are designed to mimic the Haar inverse wavelet transform, but with trainable kernels replacing the father and mother wavelets. This allows a one-to-one comparison between the kernel values and the values of the wavelet functions. Additionally, rather than preform convolutions along the LR and WC, convolutions are preformed across the two (see Fig. 3.6). If the kernels have the same values as the Haar father wavelet ψ and Haar mother wavelet ϕ, then applying kernel across the sets is identical to the true Haar inverse wavelet transform. This derives from $\psi[0] = \phi[1]$.

Discriminators

As mentioned above, both the decomposer and the recreater are associated with an adversarial loss, leading to two discriminator networks for the purpose of training. The architecture for these networks is summarized in Table 3.2 and Table 3.3 respectively. Both discriminators have an almost identical structure, with the main differences coming from initial input shape and the number of activations in Layer 5. Note after each layer except for the final layer, a leaky ReLU is applied. A sigmoid is applied to the final layer.

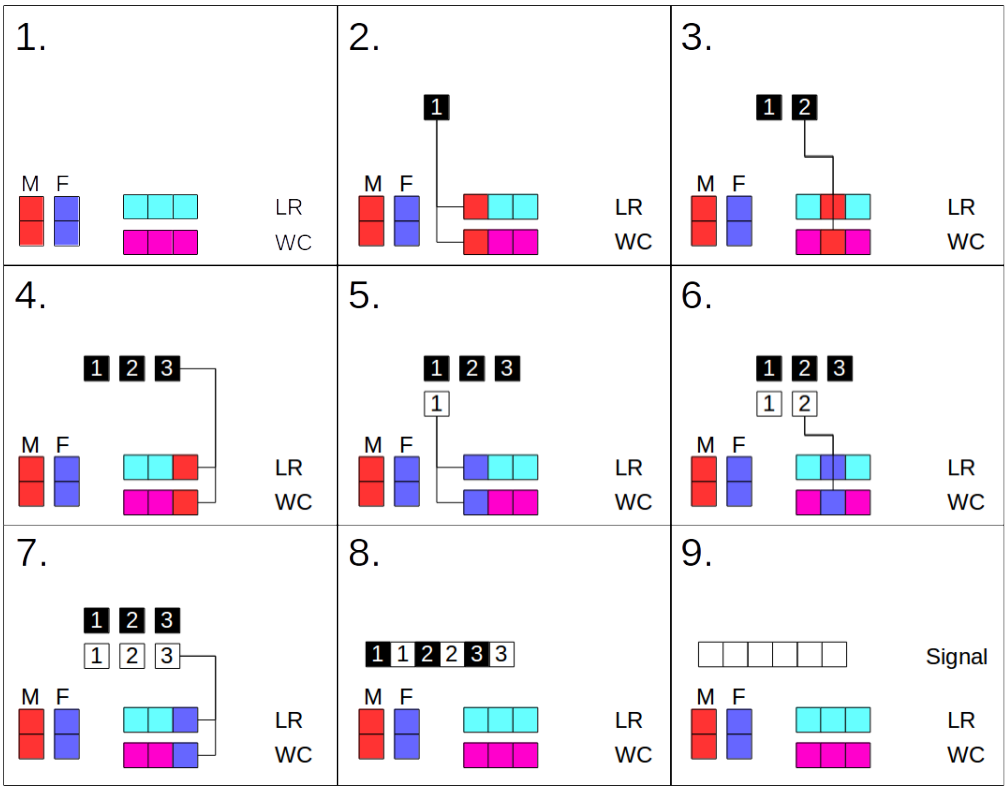

Figure 3.6: Alternative Discrete 1-D Wavelet Inverse Decomposition recreating discrete signal with length 6 done with the Haar Wavelets

#	Layer	Input	Output
1	Conv 3x3	$64 \times 64 \times 3$	$64 \times 64 \times 64$
2	Conv 3x3	$64 \times 64 \times 64$	$32 \times 32 \times 64$
3	Conv 3x3	$32 \times 32 \times 64$	$32 \times 32 \times 64$
4	Conv 3x3	$32 \times 32 \times 64$	$32 \times 32 \times 3$
5	Fully Connected	$32 \times 32 \times 3 = 3072$	64
6	Fully Connected	64	1

Table 3.2: Basic Dicriminator Network for calculating $\mathcal{L}_{GAN}^{W}$

#	Layer	Input	Output
1	Conv 3x3	$128 \times 128 \times 1$	$64 \times 64 \times 64$
2	Conv 3x3	$128 \times 128 \times 64$	$64 \times 64 \times 64$
3	Conv 3x3	$64 \times 64 \times 64$	$64 \times 64 \times 64$
4	Conv 3x3	$64 \times 64 \times 64$	$64 \times 64 \times 3$
5	Fully Connected	$64 \times 64 \times 3 = 12288$	128
6	Fully Connected	128	1

Table 3.3: Basic Dicriminator Network for calculating $\mathcal{L}^H_{GAN}$

3.4 Model-Based Performance Metric

One of the main challenges in creating a good super resolution model is defining what a good model looks like. In the literature, models are primarily judged by observing the average Peak Signal-to-Noise Ratio ($PSNR$) of images ran through the network. Defined as

$$PSNR = 10 \log_{10} \left(\frac{\max \left(I^2_{HR} \right)}{MSE} \right) \tag{3.4}$$

where MSE is the mean squared error between the HR and SR images, $PSNR$ allows comparison different SR methods without regard for scale of intensities. Note $PSRN$ ranges from 0 to infinity with infinity the SR image perfectly matches the original HR. In the case of SR where the images have been normalized, $\log 10(I_{HR}) = 0$ and Eq. 3.4 simplifies to

$$PSNR = -10 \log_{10}(MSE) \tag{3.5}$$

As a univariate injective function in MSE, $PSNR$ inherits MSE's flaw's as a performance metric, namely its preference for averaged values.

In this work, rather than using differences between HR and SR results to form a metric, the strength of the model judged by how well the network resembles the true function it is approximating. In particular, we judge

how well the recreater resembles the inverse wavelet transformation. Since it judges performance based on the model itself, the metric will be referred to as a Model-Based Performance Metric or MPM.

To design, let $p_1 \dots p_n$ be the values associated with the wavelets used in the 2-D Haar inverse wavelet transform and let $\hat{p}_1 \dots \hat{p}_n$ be kernel values used by the recreater, with $\hat{p}_i = p_i$ for all i impling the recreater is preforming the 2-D Haar inverse wavelet transform. Note that for the recreater used, $n = 8$. The proposed MPM P is then given by

$$P = \frac{\sum_{i=1}^{n} |p_i - \hat{p}_i|}{\sum_{i=1}^{n} |p_i|} \tag{3.6}$$

If the decomposer perfectly finds the WC, $P = 0$. The further the recreater's parameters deviate from $p_1 \dots p_n$, the larger P gets, indicating a worse model.

Note, an alternative MPM is given by

$$P^2 = \frac{\sum_{i=1}^{n} (p_i - \hat{p}_i)^2}{\sum_{i=1}^{n} (p_i)^2} \tag{3.7}$$

While P can indicate a perfectly working model, it is still not a perfect metric. Like $PSNR$, the P may favor models that preform worse in reality. This comes from the assumption that the decomposer perfectly constructs the wavelet coefficients. This is most certainly not the case and the parameters found by the recreater may be better than the 'ideal' parameters in creating the HR image. A well preforming decomposer should come close finding the true WC and so P provide a fair judgment of the model.

Another issue with MPMs is their limited scope. Since the inverse wavelet transformation is known, it is possible to judge how well of an approximation the model by just looking at the model itself. The workings of most neural networks and how they are related to the

functions they approximate are in most cases unknown and a MPM not possible.

3.5 Training Stages

To start, all parameters are randomly initialized with values chosen from a normal distribution centered at 0, with the exception of the parameters on recreater, which are initialized to their ideal values i.e. P begins at zero. A batch size of 16 is used for training.

Training of the network is preformed in three distinct stages: training only the decomposer while ignoring the discriminators and holding the recreater constant, training the decomposer while taking into account the discriminators and holding the recreater constant, and training the entire network. An Adam optimizer with a learning rate of 0.0001 is used to train the generator for all stages. Discriminators are trained a SGD with momentum optimizer with a learning rate of 0.001 and a momentum of 0.9.

The first stage aims to learn the global structure of the WC sets and the HR image, for which the MSE loss function is well-adapted for. In this stage, Eq. 3.3 is reduced to

$$\mathcal{L} = \alpha \mathcal{L}^{W}_{MSE} + (1 - \alpha)\mathcal{L}^{H}_{MSE} \tag{3.8}$$

After about 2 epochs, this loss saturates. The second stage begins after 2 epochs. Here, the network aims to learn the small details of the WC sets and the HR image through the use of the GAN losses. The network aims to reduce the full loss given in Eq. 3.3 with the recreater held constant. The discriminators were each trained for 3 steps after every training step preformed the generator. The second stage runs for 3 epochs.

Finally, the third stage begins, giving the full network the freedom to change. It is during this stage that P is allowed to deviate from zero. The network is trained in this stage for 3 epochs.

An alternative training sequence could be to start immediately from the second stage. This is avoided to decrease computational cost. Indeed, training the additional discriminators greatly increases training time.

Additionally, starting from the second stage may lead to decreased performance in the overall routine (see Chapter 4).

Another alternative is to allow the recreater to change from the beginning of training. This is not preformed in this work to allow the decomposer time to learn how to best act together with the real inverse wavelet transform.

The neural network is trained on two NVIDIA Tesla K20Xm cards and each network takes about 23 hours to train.

3.6 Super Resolution of the Entire Sun

As mentioned in Sec. 3.2, network training was only preformed on the solar center. To preform SR of the entire Sun, this thesis trains two networks, $\mathcal{N}_C$ and $\mathcal{N}_E$, that would specialize on learning SR for the solar center and solar edge respectively. The architectures of $\mathcal{N}_C$ and $\mathcal{N}_E$ need not be the same. Once created, a LR image I_{LR} is run through both networks to obtain $\mathcal{N}_C(I_{LR}) = I_{SR}^C$ and $\mathcal{N}_E(I_{LR}) = I_{SR}^E$. A new region, the merger region, is then added to the division of the sun discussed in Sec. 3.2 (see Fig. 3.7). This merger region is located on the boundary between solar center and solar edge and defined by

$$\rho_{mc} \leq \rho \leq \rho_{me} \tag{3.9}$$

where $\rho_{mc} = 0.7 * H$ and $\rho_{me} = 0.8 * H$.

To find the final SR, it is assumed that values deep within the solar center and solar edge are well approximated by I_{SR}^C and I_{SR}^E respectively, while those in the merger region are somewhere between these two approximations. The final SR image I_{SR} is given by

$$I_{SR}[i,j] = \begin{cases} I_{SR}^C[i,j] & : \quad \rho \leq \rho_{mc} \\ \left(1 - \frac{\rho - \rho_{mc}}{\rho_{me} - \rho_{mc}}\right) I_{SR}^C[i,j] + \left(\frac{\rho - \rho_{mc}}{\rho_{me} - \rho_{mc}}\right) I_{SR}^E[i,j] & : \rho_{mc} \leq \quad \rho \leq \rho_{me} \\ I_{SR}^E[i,j] & : \rho_{me} \leq \quad \rho \end{cases}$$

where $\rho(i,j) = \sqrt{(i - H/2)^2 + (j - W/2)^2}$.

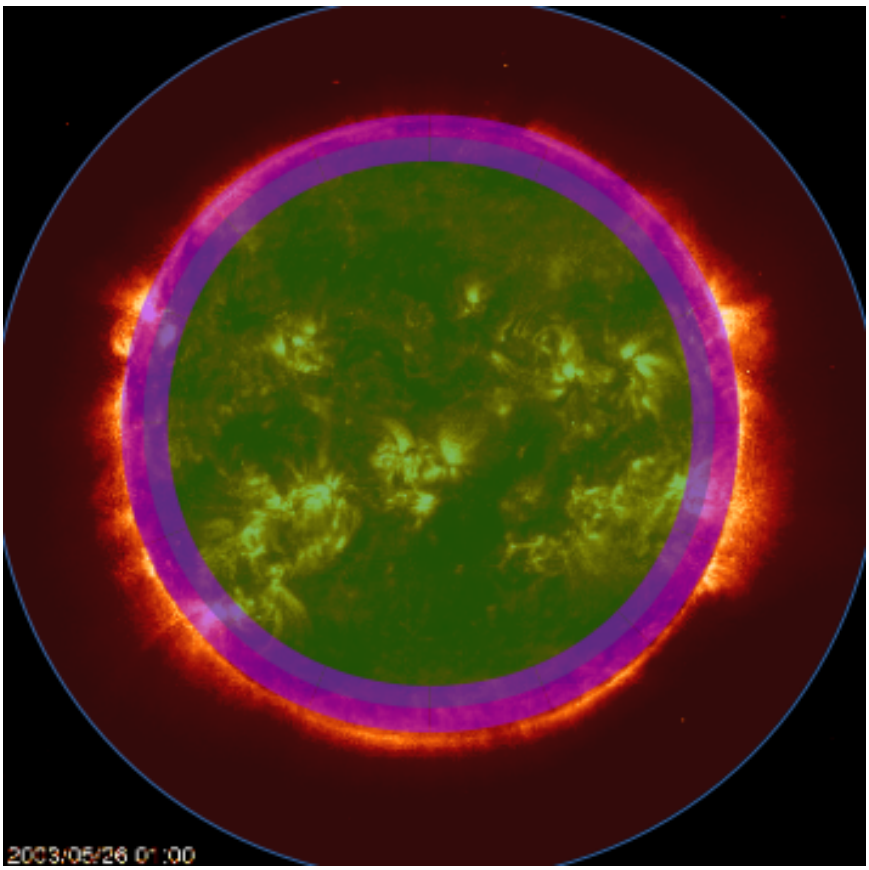

Figure 3.7: Solar regions with merger region in purple

Chapter 4

Results

Various models were tested to determine the best hyper-parameters for the final network. This mainly focused on selecting the correct loss parameters α, β, and γ found in Eq. 3.3. Testing can be divided into three main stages: α-selection, β-selection, and γ-selection. These stages and their results are summarized in Table 4.1.

The testing first stage, α-selection, keeps constant $\beta = \gamma = 0.5$ and varies α by 0.25. The ideal value for α appears to be around 0.5, scoring $P = 0.0220$ and $P^2 = 9.15 * 10^{-4}$. Interestingly, $\alpha = 0.5$ had the lowest $PSNR$ (34.25) while $\alpha = 0.75$, which both P and P^2 judged as second best out of three, had the highest (35.17). In this stage, two additional modifications to the network were tested, both at $\alpha = 0.5$. The first was to combine the first two training stages. Doing so lead to a significantly worse P and P^2, suggesting that separating training stages 1 and 2 may be a more effective training routine than combining the two. The second test was the removal of residual layers in the decomposer, i.e. the removal of layers 4 and 7 in Table. 3.1. This lead to a slightly worse P and P^2 as well as a grid-like pattern to appear in image results (see Fig. 4.1). The grid pattern initially forms for all networks as a results of the recreater stitching together the LR image and WC sets. As the network learns better parameters, the generator gets better at producing a more fluid final image. The appearance of a grid in the no residuals network suggests the network did not have enough time to learn the

Stage 1: α-selection

#	α	β	γ	Other	P	P^2	$PSNR$
1	0.50	0.5	0.5		0.0220	0.000915	34.25
2	0.25	0.5	0.5		0.0475	0.00426	34.64
3	0.75	0.5	0.5		0.0282	0.00229	35.17
4	0.50	0.5	0.5	Combine Training Stages	0.110	0.0263	35.09
5	0.50	0.5	0.5	No Residual	0.0297	0.00180	30.78

Stage 2: β-selection

#	α	β	γ	Other	P	P^2	$PSNR$
1	0.75	0.25	0.5		0.0252	0.00146	34.56
2	0.75	0.75	0.5		0.0237	0.00131	35.20
3	0.75	0.25	0.5	Batch Normalization	0.175	0.0423	28.81
4	0.50	0.25	0.5		0.0493	0.00436	33.11
5	0.50	0.75	0.5		0.0467	0.00546	35.01

Stage 3: γ-selection

#	α	β	γ	Other	P	P^2	$PSNR$
1	0.5	0.5	0.25		0.0880	0.0153	32.63
2	0.5	0.5	0.75		0.0522	0.00678	34.67

Table 4.1: Training Results

correct parameters.

The β-selection phase first tests with α and γ held constant at 0.75 and 0.5 respectively. The choice of $\alpha = 0.75$ rather than the seemingly more ideal $\alpha = 0.5$ came about not from reasoned choice but due to a typo in the code. With this α, the ideal value for β appears to be 0.5. Adding batch normalization layers to both discriminators after each convolutional layer (see Table 3.2 and Table 3.3) were also tested. Visually, networks with a batch normalized discriminators appear to preform better, producing less blurred images with static-like artifacts similar to those found in the HR image (see Fig. 4.2). This is most likely attributable to better discriminator performance, which expects such static-like artifacts for the image to appear 'real'. Yet, this modification lead to the worst results based on all three metrics. Since larger P

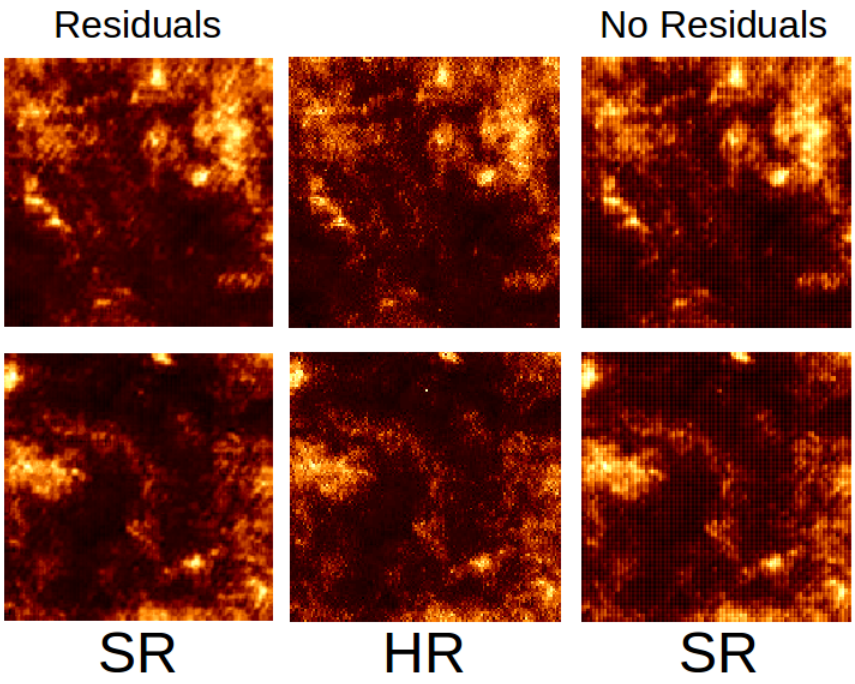

Figure 4.1: Comparison of networks with same α, β, and γ and with residual layers (left) versus without residual layers (right). True HR image is in center.

implies the WC sets, and so the higher frequency information, generated are further from truth, this modification is dropped.

Given that α-selection seems to suggest 0.5 as the best value for α, β-selection also tested β values while keeping both α and γ at 0.5. Doing so, the best value for α and β appears to be 0.5. Despite a network with $\alpha = 0.5$ having the best P value, in agreement with the α-selection stage, networks with $\beta = 0.25$ and $\beta = 0.75$ preformed better at $\alpha = 0.75$. This suggests the hyper-parameters α, β, and γ are linked and examining them independently may not lead to the best result. A more through study should be done to find the best values. Due to financial and time restraints, a better and more through search it not preformed.

For the γ-selection phase, α and β are held constant at 0.5 and $\gamma = 0.25$ and $\gamma = 0.75$ are tested. Ultimately, the best network appears to be $\alpha = \beta = \gamma = 0.5$ having scored the best P-value (0.0220) and the best

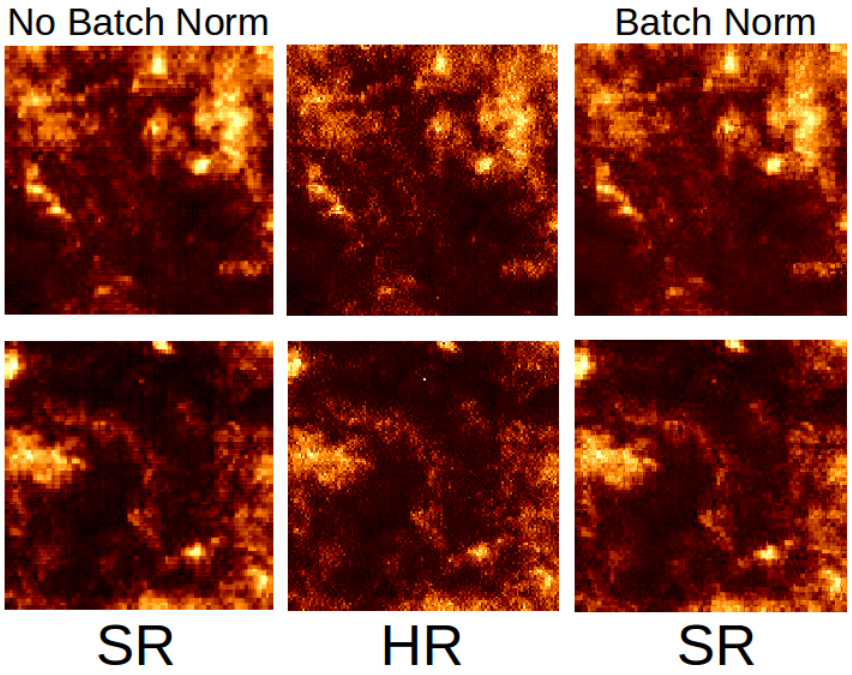

Figure 4.2: Comparison of networks with same α, β, and γ and with no batch normalized discriminators (left) versus with batch normalized discriminators (right). True HR image is in center.

P^2-value ($9.15 * 10^{-4}$). Final results are shown in Fig. 4.3.

Based on patch (1), the network preforms well visually when there is not much noise in the image. Once a patch experiences noise, as in patch (3) and patch (4), the network is unable to fully replicate the abrasive texture and instead creates a more blotchy texture similar to that found in a sponge painting. Fortunately, the network appears to only attempt to replicate this noise in areas where it is located in the original HR image, as seen in patch (2). As mentioned above, batch normalization layers in the discriminator seem to help when dealing with noise. Creating a measure for how much noise an image has and merging the batch normalization and non-batch normalization trained networks in a similar manner to that outlined in Sec. 3.6 based on the noise level rather than region location may lead to a better overall SR algorithm.

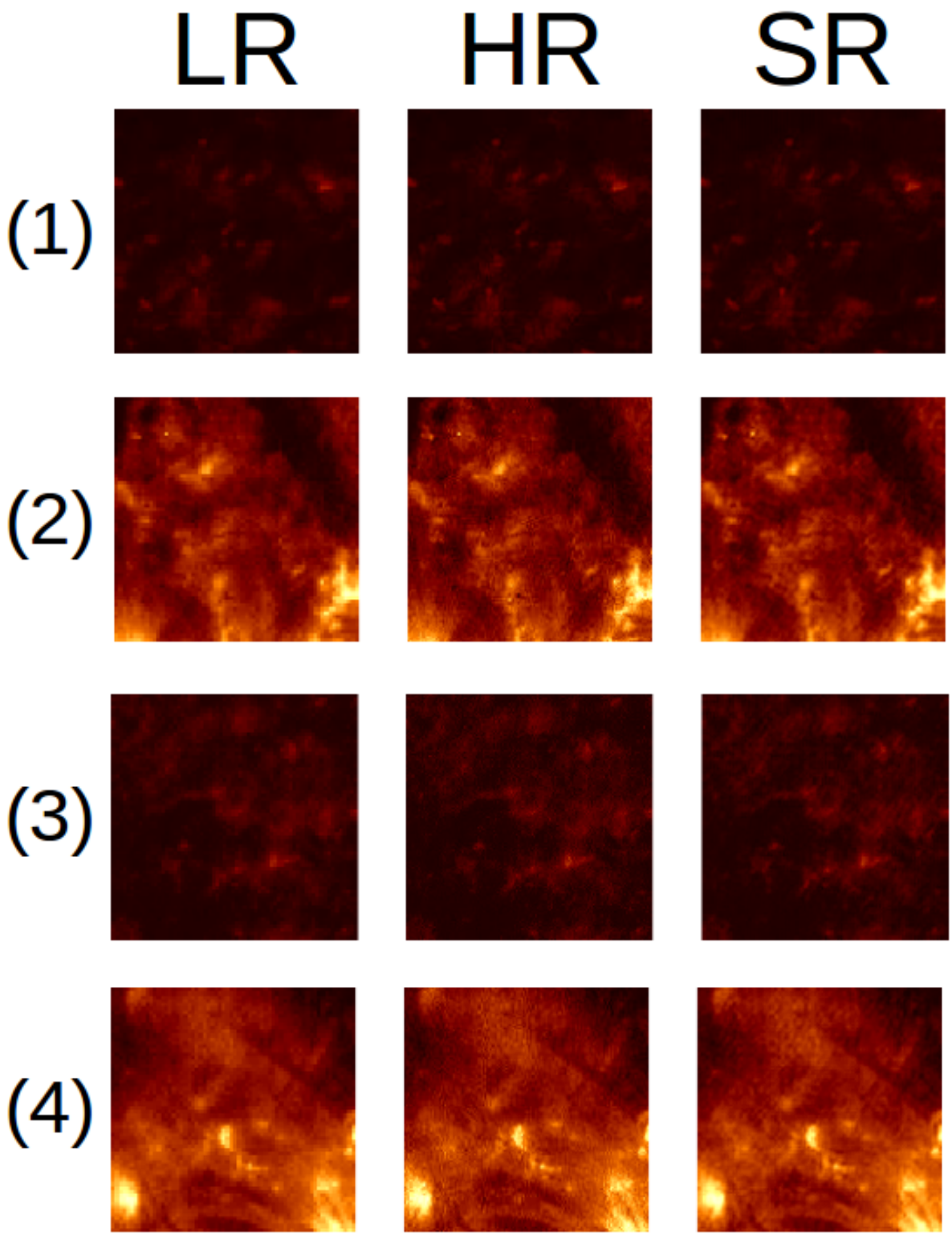

Figure 4.3: Final network results

Despite the network's difficulty in dealing with static, the SR of solar features appear to be done relatively well. Fig. 4.4 depicts sample results from areas of colder plasma called coronal holes. Focusing on patch (5), which has a lack of any structure over large portions of the image, the SR image has not created any major structure not found in the original image. This is true for all patches examined, thus supporting the applicability of the method for research purposes. In all the patches in the figure, the boundaries of the structures are defined and similar to those found in the HR image. This occurs despite the pixel intensities in coronal holes having values within about 0.15 of each other. As mentioned above, the SR network tries to mimic the static in the HR image. This is best seen in patch (8).

Samples of active regions, regions on the Sun with strong magnetic activity, are depicted in Fig. 4.5. Like for coronal holes, the SR method appears to do a good job at defining boundaries for major structures. This is especially clear in the diamond-shaped boundary just right of center in patch (11). Looking within, a small pill-shaped structure is found in the HR image that has been reproduced in the SR image but is ill-defined in the LR image. The talon-like structure near the center of patch (9) is another clear example of how boundaries are often faithfully reproduced. Within a boundary, however, the method may struggle fully capture the finer details of a structure. This is best seen in the upper right of patch (12). The SR image is an improvement over the LR image in terms of details, but only barely and the hot plasma comes off as blurry compared with the original HR image. This, and the models inability to deal with noise, suggests more work needs to be done with dealing with small details.

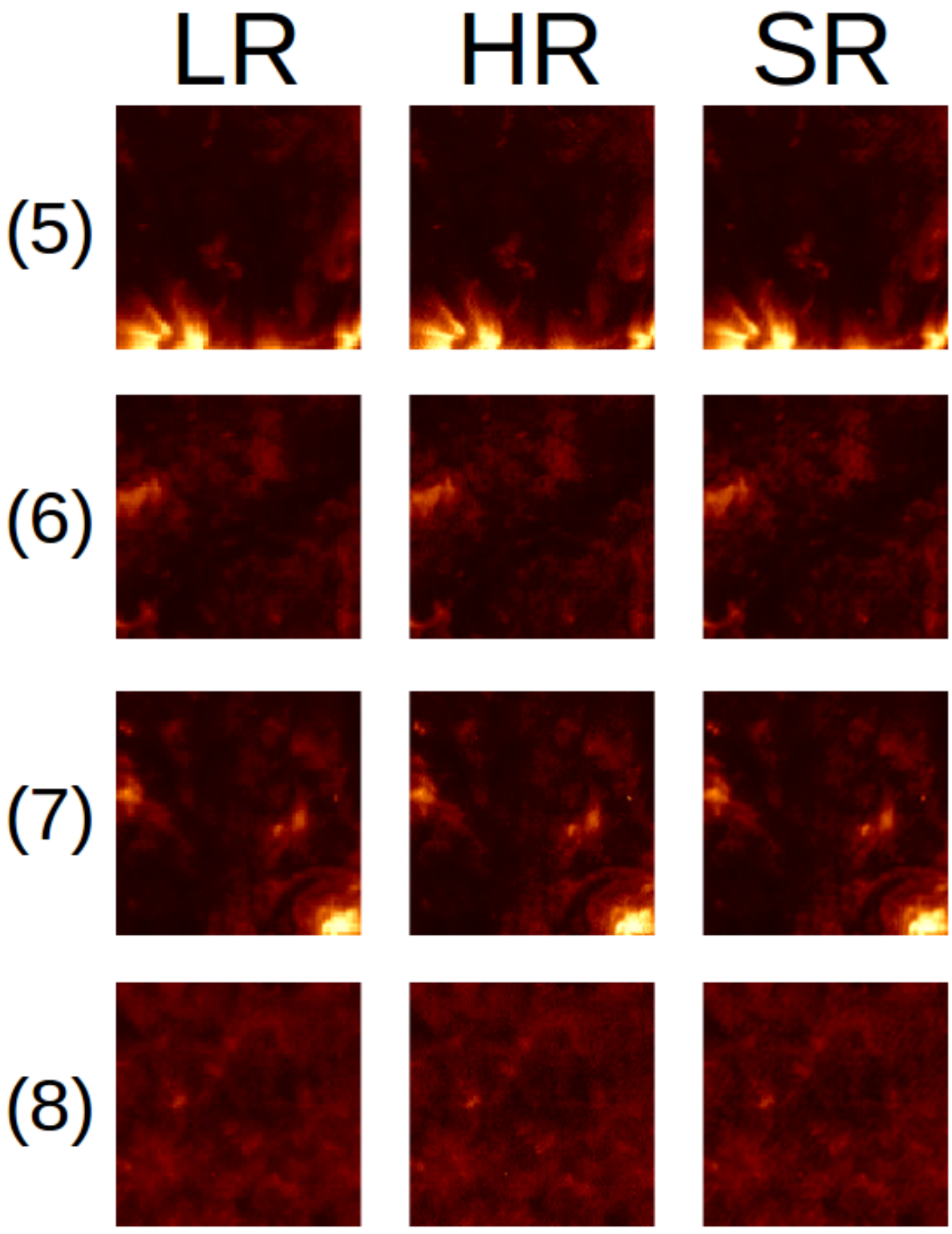

Figure 4.4: Coronal hole results

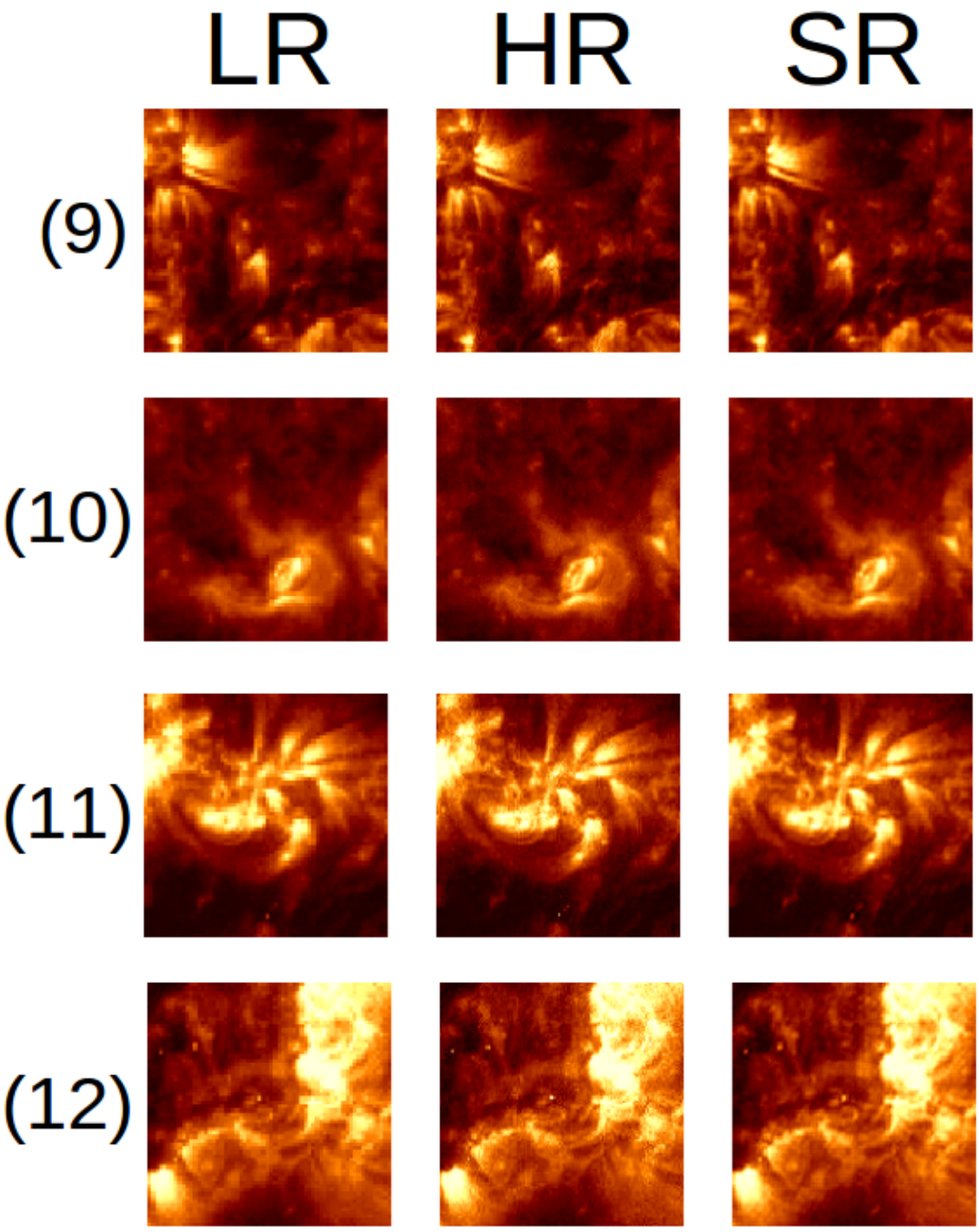

Figure 4.5: Active region results

Super Resolution of the Sun

To preform SR of the entire Sun, the method discussed in Sec. 3.6 is applied. The network for the solar center $\mathcal{N}_C$ has already been developed above. Due to time and financial constants, no time was spent developing the architecture for $\mathcal{N}_E$. It is assumed that the best network architecture for the solar edge is the same as for the solar center, so that $\mathcal{N}_E$ has the same structure as $\mathcal{N}_C$ but trained on patches from the solar edge rather than the solar center. The merger region is defined by $\rho_{mc} = 0.7$ and $\rho_{me} = 0.8$. Final SR images of the Sun are shown in Fig. 4.8 and Fig. 4.11 with the corresponding HR images in Fig. 4.7 and Fig. 4.10 and corresponding LR images in Fig. 4.6 and 4.9. Overall, the SR method suffers in the same way as described above when dealing with noise.

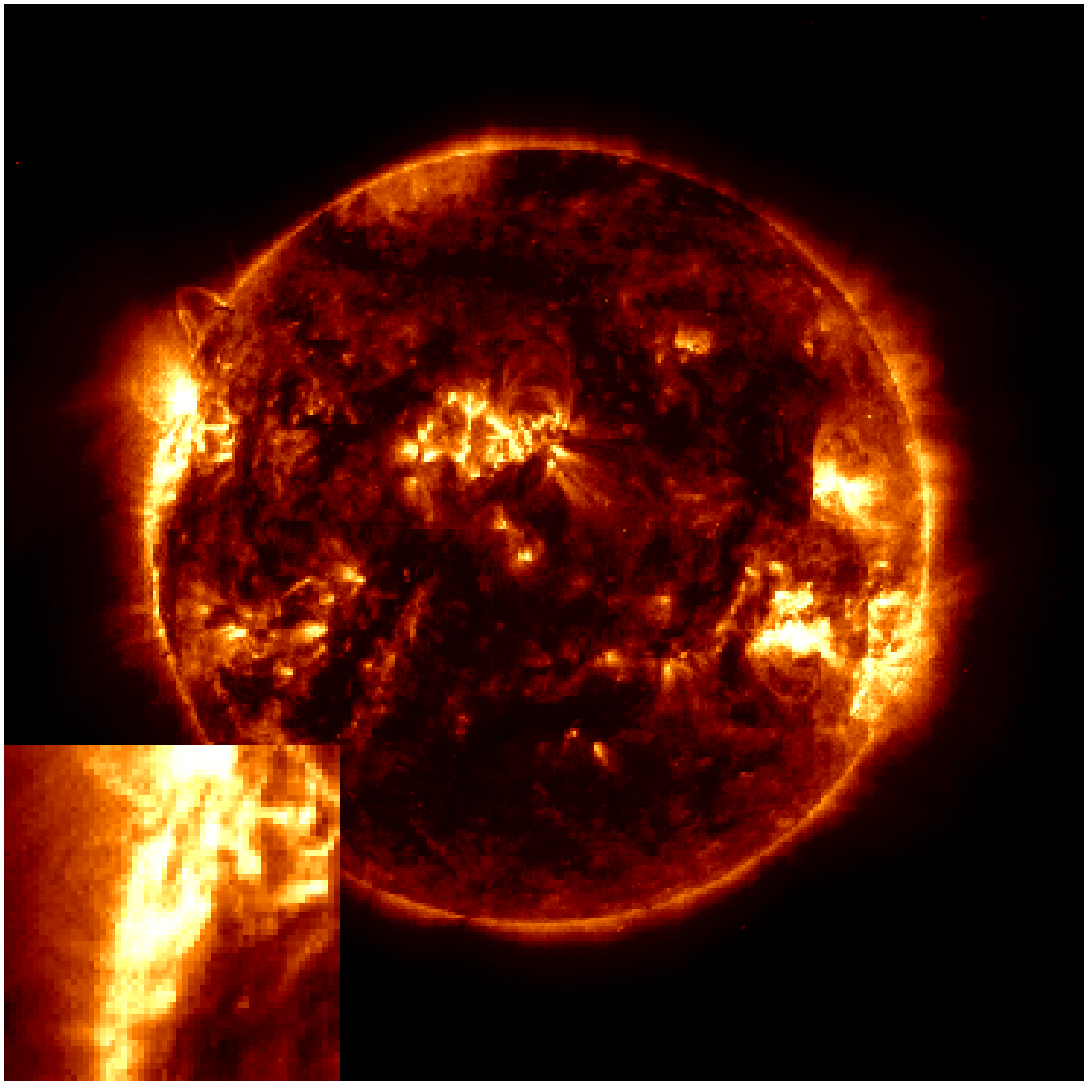

Figure 4.6: Low Resolution of the Sun: (1)

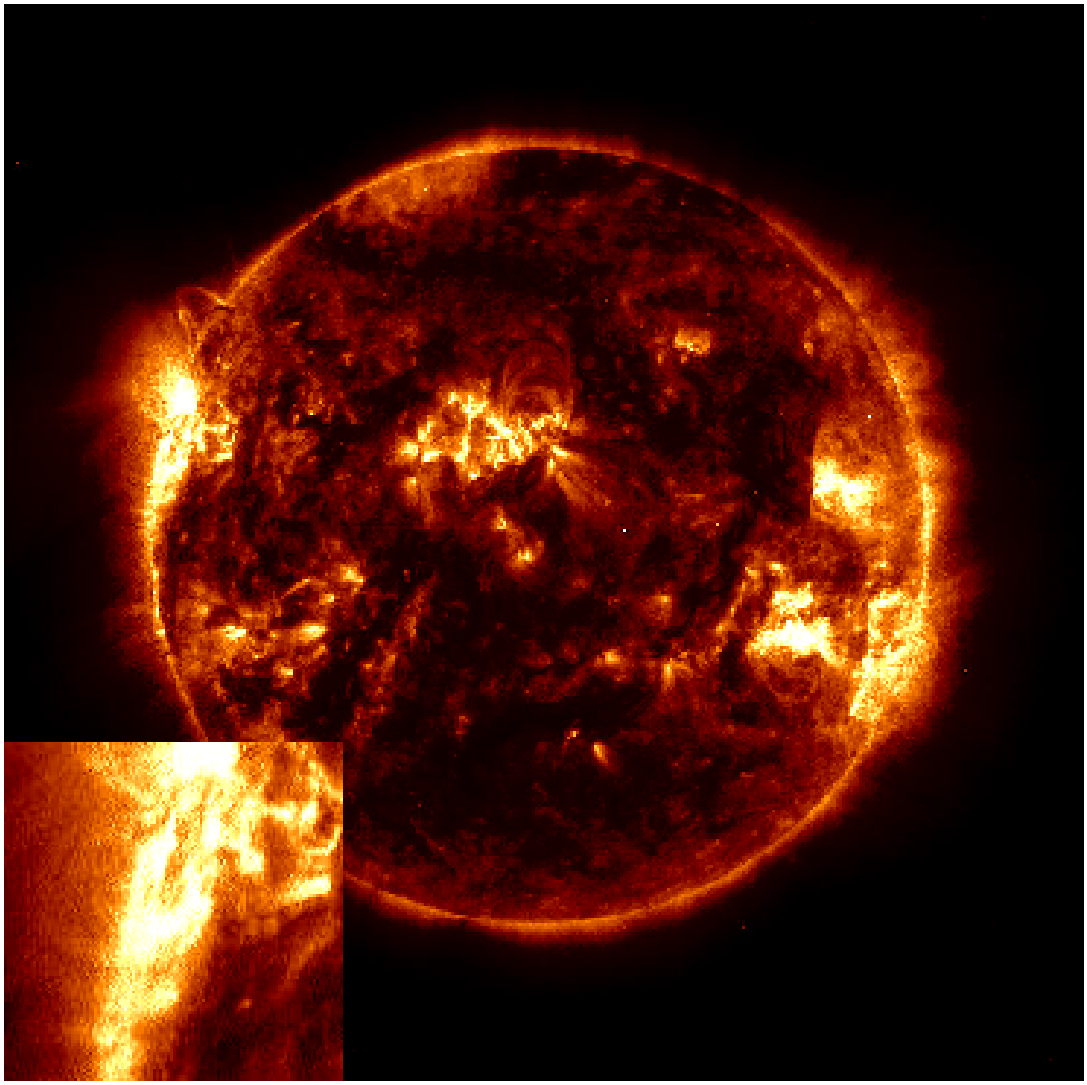

Figure 4.7: High Resolution of the Sun: (1)

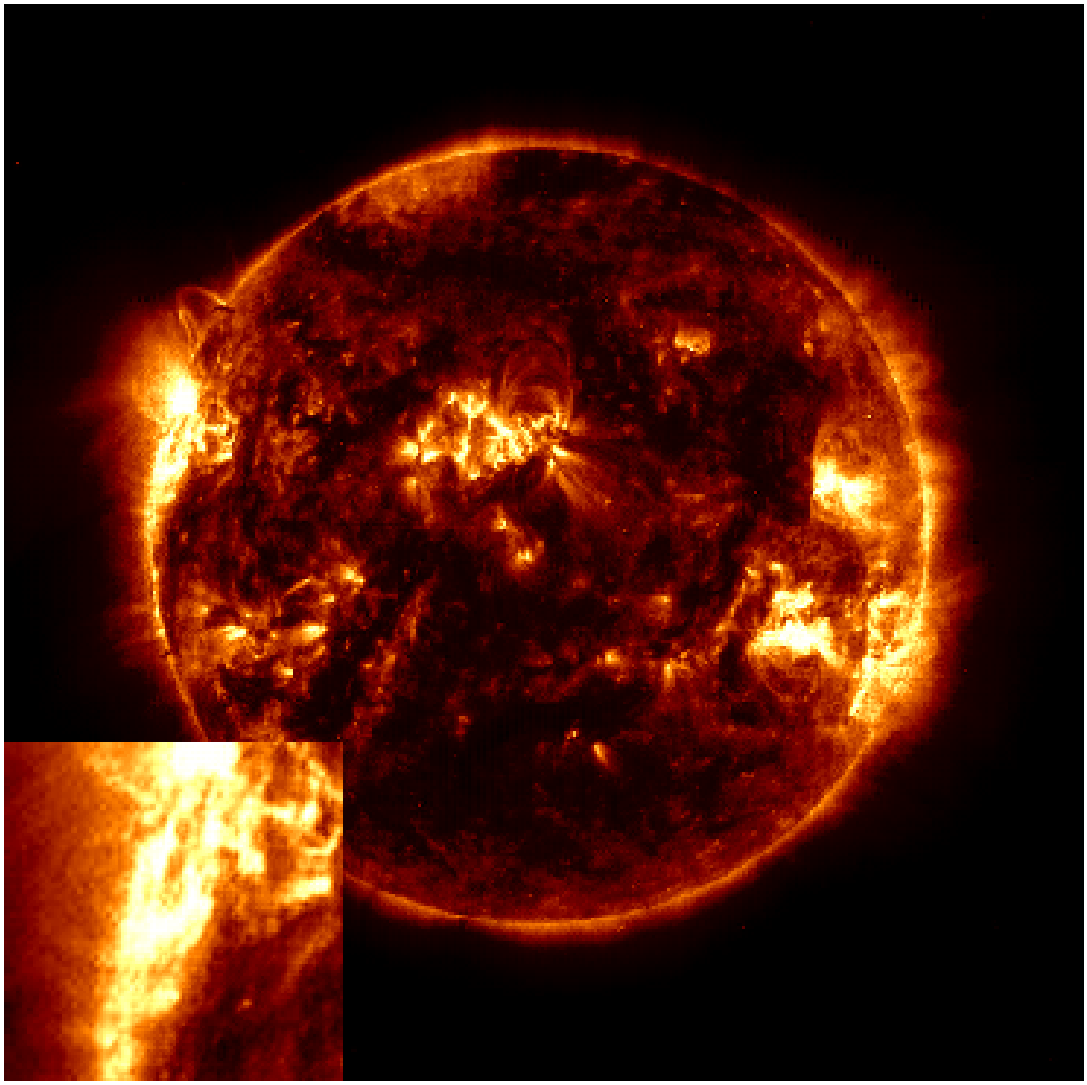

Figure 4.8: Super Resolution of the Sun: (1)

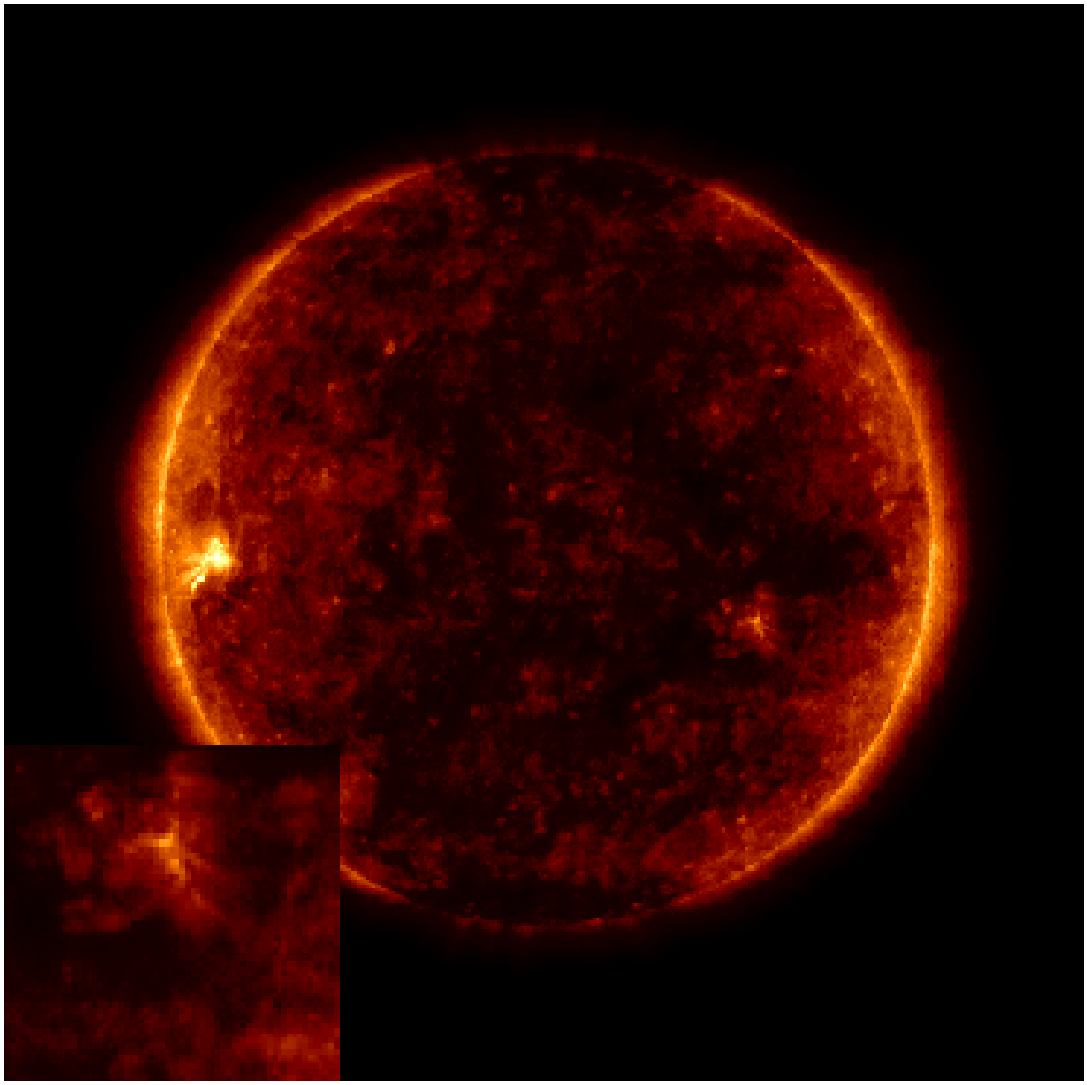

Figure 4.9: Low Resolution of the Sun: (2)

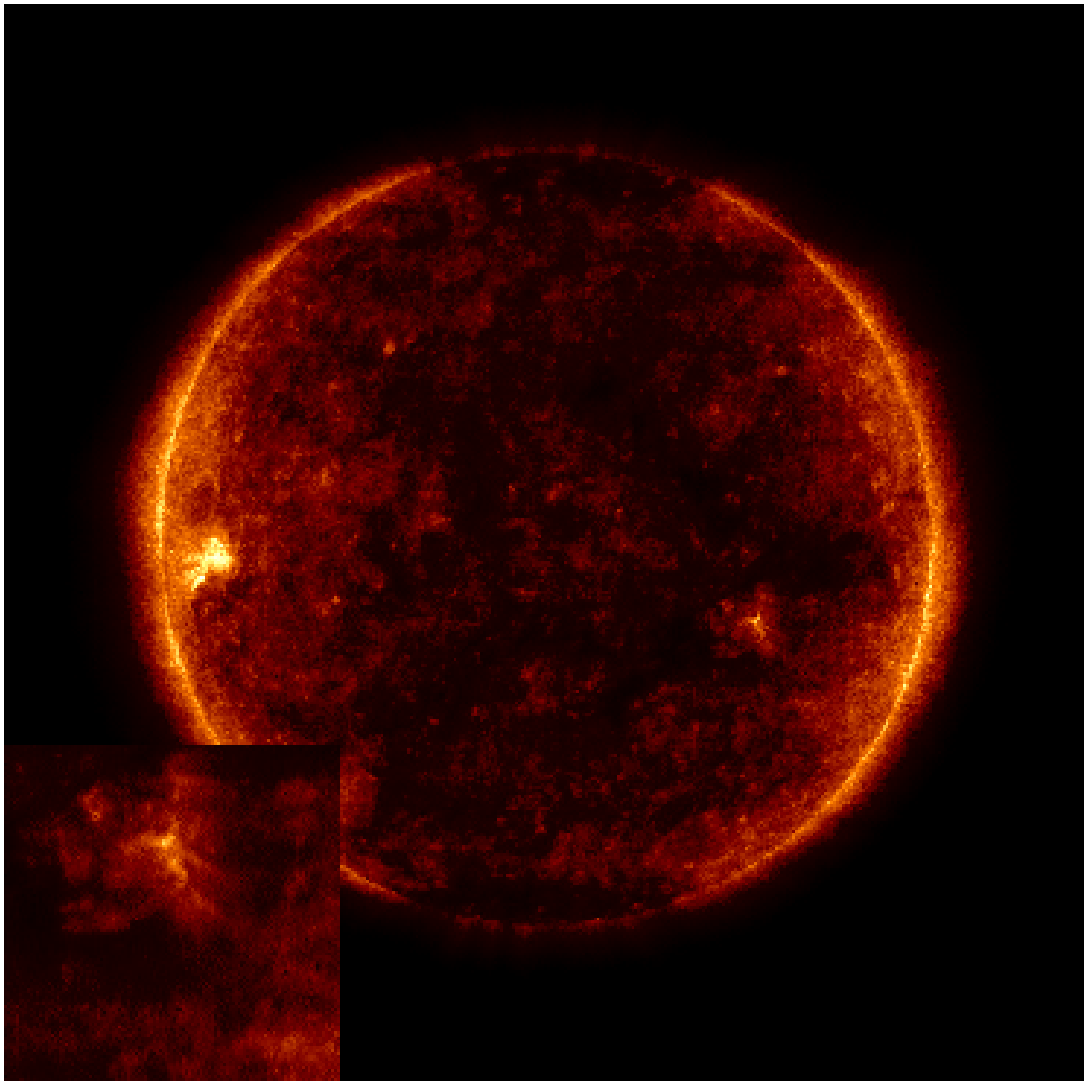

Figure 4.10: High Resolution of the Sun: (2)

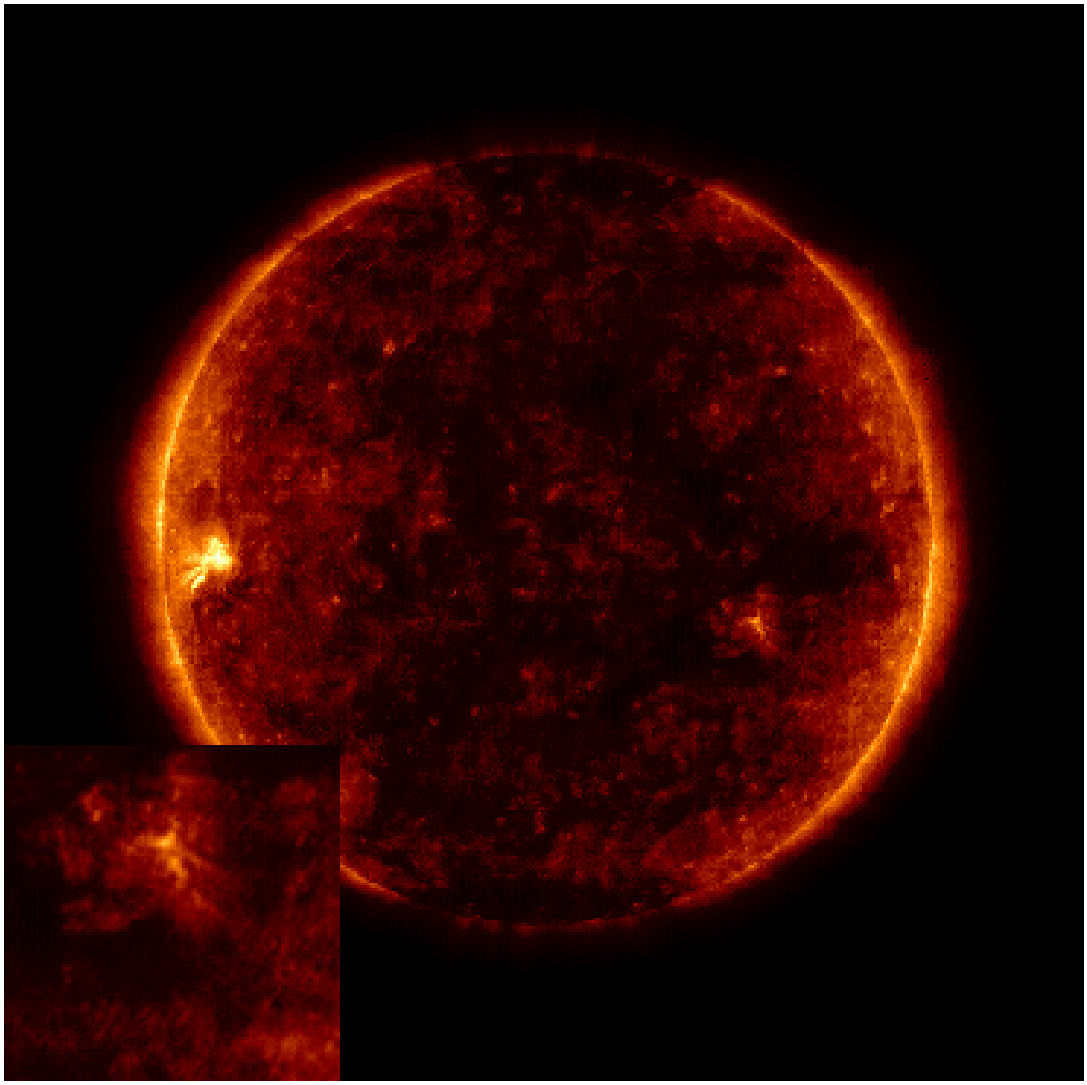

Figure 4.11: Super Resolution of the Sun: (2)

Chapter 5

Concusion

The created SR method created is able to take LR input and produce an image of higher resolution which faithfully captures the main structures of the aimed for HR image. Details previously unclear in the LR image become clear and no new structures appear. Thus, the SR method appears to be a valid tool for solar researchers hoping to improve their data and to understand solar phenomenon. Yet, the method is far from perfect. High frequency details, particularly in active regions, are often not caught by the network, producing a result little better than that found in the LR image.

In further developing this SR for solar data, numerous improvements can be preformed. As already mentioned, hyper-parameters should be checked in a more intelligent manner. Modifications to the network should also be explored as a better network architecture may exist. A better loss function should also be implemented. Although the combination of MSE and adversarial losses produced decent results, the use of perceptual loss functions should greatly improve SR capabilities. This is particularly true for scientific applications that can use scientifically important features to give a more physically accurate SR method. Theses features can be as simple as integrating over pixel intensities of the Sun [35], or more complex as looking at the activations of other neural networks attuned to measuring solar features or predicting solar events. Unfortunately, more development needs to

be done on these networks although some initial networks have been created by those associated with the AIDA project [3][26].

Dealing with noise also appears to be an issue for the network. Work should be done to see if this effects models which would potentially use the SR results to see how big of an issue this is.

More work should also be done on examining metrics based on network parameters as well as how these parameters can be used for machine learning applications. An understanding of network parameters can lead to a better understanding of neural networks as well as give inspiration on neural network structures. Their use in metrics can give researchers new ways of judging difficult to describe tasks such as super resolution.

MPMs may also help open the door to machine learning of physical laws. For example, suppose we have a network $\mathcal{N}$ trying to approximate the equation $\mathcal{N}^*$ that maps some input $X \in \mathbb{R}$ to some output $Z \in \mathbb{R}$. Let $p_1 \cdots p_n$ be the set of parameter values associated with $\mathcal{N}$ after the network is trained on X and Z. One can then create a set of basic functions, e.g. $\sin(x)$, e^x, e^{-x^2}, etc., and generate data for each of these functions, e.g. generate $f(X) = \{f(x) : x \in X\}$ for $f(x)$. A set of parameters $p_1^f \cdot p_n^f$ can then be generated for each basic function f by training $f(X)$ with $\mathcal{N}$. The parameter sets $\{p_1^f \cdots p_n^f\}$ can be compared with $\{p_1 \cdots p_n\}$ to select the basic functions that have the most underlying similarities to similarity to $\mathcal{N}$. Doing so, one may be able to reconstruct $\mathcal{N}^*$. Work should also be done on basis functions themselves to investigate how various scalings, e.g. $\sin(x)$ vs $\sin(2x)$ vs $2\sin(x)$, as well as basic arithmetic operations such as addition and multiplication affect the parameters of the network.

Appendix A

Network Code

```
#!/usr/bin/env python

#Generator Model in Pytorch

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
from numpy.random import seed, shuffle

##############################################################################
#Hyper Parameter List
########################################
#net_name: name of directory to save
#batch_size: batch size train for gen; batch size real+batch
    size fake for dis
#epoch_num: number of epochs
#epoch_on: start training dis on epoch_on
#dis_lim: if dis losses higher, dont train dis
#k_steps: dis training steps
#
#lr_gen: Generators Learning Rates
#lr_dis: Disriminators Learning Rates
```

```
#
#alpha: WC vs HR loss weight
#beta: MSE vs Dis on Dec loss weight
#gamma: MSE vs DIS on Rec loss weight
#
#initialize: initialize Rec or not
#
#rec_hold: start training rec on epoch rec_hold
#validate: statistics every 'validate' steps
#
#load: Load previously trained epoch
#load_epoch: Which epoch to load (if load=False, set load_epoch
    = -1)
################################################################################

net_name = 'Network01/'
batch_size = 16
epoch_num = 8
epoch_on = 2
dis_lim = 5.
k_steps = 3

lr_gen = 0.0001
lr_dis = 0.001

alpha = 0.75
beta = 0.25
gamma = 0.5

initialize = True
rec_hold = 5

validate = 100

load = False
load_epoch = -1


print('Parameter List for ' + net_name)
print('\tbatch_size: ', batch_size)
print('\tepoch_num: ', epoch_num)
print('\tepoch_on: ', epoch_on)
```

```
print('\tdis_lim: ', dis_lim)
print('\tk_steps: ', k_steps)
print('\n')
print('\tlr_gen: ', lr_gen)
print('\tlr_dis: ', lr_dis)
print('\n')
print('\talpha: ', alpha)
print('\tbeta: ', beta)
print('\tgamma: ', gamma)
print('\n')
print('\tinitialize: ', initialize)
print('\trec_hold: ', rec_hold)
print('-'*30)

DATA_PATH = '/staging/leuven/stg_00032/SR/Data/'
RESU_PATH = '/staging/leuven/stg_00032/SR/Results/'+net_name



#Data Loading
class ImageDataset(Dataset):
    def __init__(self, train_data_lr, train_data_wc,
        train_data_hr, train = 'train', div = (.8, .9)):
            """
            Args:
                train_data_lr (string): Path to the LR data
                train_data_wc (string): Path to the WC data
                train_data_hr (string): Path to the HR data
                train: training phase: 'train', 'cross', 'test',
                    'all'
                div: division of samples
            """
            self.train_data_lr = np.load(train_data_lr,
                mmap_mode= 'r')
            self.train_data_wc = np.load(train_data_wc,
                mmap_mode= 'r')
            self.train_data_hr = np.load(train_data_hr,
                mmap_mode= 'r')
            total = self.train_data_lr.shape[0]

            seed(100)
            index = np.arange(total)
```

```
            shuffle(index)

            if train == 'train':
                self.index = index[:int(div[0]*total)]
            elif train == 'val':
                self.index =
                    index[int(div[0]*total):int(div[1]*total)]
            elif train == 'test':
                self.index = index[int(div[1]*total):]
            else:
                self.index = index


    def __len__(self):
        return len(self.index)

    def __getitem__(self, idx):
        i = self.index[idx]
        sample = {'LR':
            torch.from_numpy(self.train_data_lr[i,:,:]),
                 'WC':
                    torch.from_numpy(self.train_data_wc[i,:,:,:]),
                 'HR':
                    torch.from_numpy(self.train_data_hr[i,:,:,:])}
        return sample



images = ImageDataset(DATA_PATH+'train_data_lr.npy',
                      DATA_PATH+'train_data_wc.npy',
                      DATA_PATH+'train_data_hr.npy')


print('Images: ', len(images))
dataloader = DataLoader(images, batch_size=batch_size,
                                        shuffle=True, num_workers=2)

discriminator_loader = DataLoader(images,
    batch_size=batch_size*2,
                                        shuffle=True, num_workers=2)
```

```
print('Data Loaded')
print('-'*30)


#Create models
#Decomposer
class Decomposition(nn.Module):
    def __init__(self, height, width, channels=1):
        super(Decomposition, self).__init__()
        self.conv1 = nn.Conv2d(channels,64, 5)

        #Residual
        self.conv2 = nn.Conv2d(64, 64, 3)
        self.conv3 = nn.Conv2d(64, 64, 3)

        #Residual
        self.conv4 = nn.Conv2d(64, 64, 3)
        self.conv5 = nn.Conv2d(64, 64, 3)

        self.conv6 = nn.Conv2d(64, 32, 3)
        self.conv7 = nn.Conv2d(32, 3, 3)
        self.conv8 = nn.Conv2d( 3, 3, 3)


    def forward(self, x):
        conv1 = nn.ReplicationPad2d(2)(x)
        conv1 = F.relu(self.conv1(conv1))

        conv2 = nn.ReplicationPad2d(1)(conv1)
        conv2 = F.relu(self.conv2(conv2))
        conv3 = nn.ReplicationPad2d(1)(conv2)
        conv3 = self.conv3(conv3)
        resid1 = conv1+conv3

        conv4 = nn.ReplicationPad2d(1)(resid1)
        conv4 = F.relu(self.conv4(conv4))
        conv5 = nn.ReplicationPad2d(1)(conv4)
        conv5 = self.conv5(conv5)
        resid2 = resid1+conv5

        conv6 = nn.ReplicationPad2d(1)(resid2)
        conv6 = F.relu(self.conv6(conv6))
```

```
        conv7 = nn.ReplicationPad2d(1)(conv6)
        conv7 = self.conv7(conv7)
        conv8 = nn.ReplicationPad2d(1)(conv7)
        conv8 = self.conv8(conv8)

        return conv8

class Discriminator(nn.Module):
    def __init__(self, height, width, channels=3):
        super(Discriminator, self).__init__()
        self.height = height
        self.width = width
        self.conv1 = nn.Conv2d(channels,64, 3)
        self.conv2 = nn.Conv2d(64, 64, 3, stride = 2)

        self.conv3 = nn.Conv2d(64, 64, 3)
        self.conv4 = nn.Conv2d(64, 3, 3)
        self.dens1 = nn.Linear(int(self.height*self.width*3/4),
            self.height)
        self.dens2 = nn.Linear(self.height, 1)

    def forward(self, x):
        conv1 = nn.ReplicationPad2d(1)(x)
        conv1 = F.leaky_relu(self.conv1(conv1))

        conv2 = nn.ReplicationPad2d(1)(conv1)
        conv2 = F.leaky_relu(self.conv2(conv2))

        conv3 = nn.ReplicationPad2d(1)(conv2)
        conv3 = self.conv3(conv3)

        conv4 = nn.ReplicationPad2d(1)(conv3)
        conv4 = F.leaky_relu(self.conv4(conv4))

        flat = conv4.view(-1, int(self.height*self.width*3/4))
        dens1 = F.leaky_relu(self.dens1(flat))
        logits = self.dens2(dens1)
        logits = torch.squeeze(logits)

        return logits
```

```
def combine(x, y, axis = 0):
    """combines x and y for reconstrution"""
    shape = x.shape[-1]
    N = x.shape[0]
    if axis == 0:
        z= torch.ones(N, 2*shape, shape)
        z=Variable(z).cuda()
        z[:, 0::2, :] = x
        z[:, 1::2, :] = y
    else:
        z= torch.ones(N, 2*shape, 2*shape)
        z=Variable(z).cuda()
        z[:, :, 0::2] = x
        z[:, :, 1::2] = y
    return z


#recreater
class Reconstruction(nn.Module):
    def __init__(self, height, width):
        super(Reconstruction, self).__init__()
        self.conv1 = nn.Conv3d(1, 2, (2,1,1), stride = (2,1,1),
            bias = False)
        self.conv2 = nn.Conv3d(1, 2, (2,1,1), stride = 1, bias =
            False)
        if initialize:
            self.init()

    def init(self):
        nn.init.constant(self.conv1.weight, 0.7071067811865476)
        self.conv1.weight.data[1,0,1,0,0] = -0.7071067811865476
        nn.init.constant(self.conv2.weight, 0.7071067811865476)
        self.conv2.weight.data[1,0,1,0,0] = -0.7071067811865476


    def forward(self, x):
        #x =(N, C= 4, H, W)
        x=x.unsqueeze(1)
        #x = (N,C =1, D=4, H, W)
        conv1 = self.conv1(x)
        #conv1 = (N, C=2, D=2, H, W)
```

```
        l = conv1[:,:, 0, :,:]
        h = conv1[:,:, 1, :,:]
        #l = (N, C=2, H, W)
        l = combine(l[:, 0, :, :], l[:, 1, :, :], 0)
        h = combine(h[:, 0, :, :], h[:, 1, :, :], 0)
        #l = (N, 2*H, W)
        l= l.unsqueeze(1)
        h= h.unsqueeze(1)

        l= l.unsqueeze(1)
        h= h.unsqueeze(1)

        mid = torch.cat((l, h), 2)
        #mid = (N,C= 1, D=2, 2*H, W)
        conv2 = self.conv2(mid)
        #conv2 = (N, C=2, D=1, 2*H, W)
        result = combine(conv2[:,0, 0, :, :], conv2[:,1, 0, :,
            :], 1)
        #result = (N, 2*H, 2*W)
        result = result.unsqueeze(1)
        #result = (N,C= 1, 2*H, 2*W)
        return result

def Preformance(Rec):
    params = list(Rec.parameters())
    low = np.array([0.7071067811865476,0.7071067811865476])
    high = np.array([0.7071067811865476,-0.7071067811865476])


    total = 0
    #rows vs columns wavelet recomposition
    for i in range(2):
        #l* vs h*
        total += abs(params[i][0, 0, 0, 0,
            0].data.cpu().numpy()-low[0])
        total += abs(params[i][0, 0, 1, 0,
            0].data.cpu().numpy()-low[1])
        total += abs(params[i][1, 0, 0, 0,
            0].data.cpu().numpy()-high[0])
        total += abs(params[i][1, 0, 1, 0,
            0].data.cpu().numpy()-high[1])
```

```
    denom = np.sum(np.abs(low)) + np.sum(np.abs(high))
    denom *= 2
    return float(total/denom)



def PreformanceSqrt(Rec):
    params = list(Rec.parameters())
    low = np.array([0.7071067811865476,0.7071067811865476])
    high = np.array([0.7071067811865476,-0.7071067811865476])


    total = 0
    #rows vs columns wavelet recomposition
    for i in range(2):
        #l* vs h*
        total += (params[i][0, 0, 0, 0,
            0].data.cpu().numpy()-low[0])**2
        total += (params[i][0, 0, 1, 0,
            0].data.cpu().numpy()-low[1])**2
        total += (params[i][1, 0, 0, 0,
            0].data.cpu().numpy()-high[0])**2
        total += (params[i][1, 0, 1, 0,
            0].data.cpu().numpy()-high[1])**2

    denom = np.sum((low)**2) + np.sum((high)**2)
    denom *= 2
    return float(total/denom)

device = torch.cuda.device("cuda")

Dec = Decomposition(64,64)
Rec = Reconstruction(64,64)

decDis = Discriminator(64,64)
recDis = Discriminator(64*2,64*2, channels = 1)

if torch.cuda.device_count() > 1:
    print('Number of GPUs ', torch.cuda.device_count())
    Dec = nn.DataParallel(Dec)
    Rec = nn.DataParallel(Rec)
    decDis = nn.DataParallel(decDis)
```

```
    recDis = nn.DataParallel(recDis)
Dec.cuda()
Rec.cuda()
decDis.cuda()
recDis.cuda()


total_loss_train = []
loss_comp_train = []
preformance = []
#load models
if load:
    Dec.load_state_dict(torch.load(RESU_PATH+'GAN_Dec'+str(load_epoch)))
    Rec.load_state_dict(torch.load(RESU_PATH+'GAN_Rec'+str(load_epoch)))
    decDis.load_state_dict(torch.load(RESU_PATH+'GAN_decDis'+str(load_epoch)
    recDis.load_state_dict(torch.load(RESU_PATH+'GAN_recDis'+str(load_epoch)

    total_loss_train = list(np.load(RESU_PATH
        +'total_loss_train.npy'))[:23*(1+load_epoch)]
    loss_comp_train = list(np.load(RESU_PATH
        +'loss_comp_train.npy'))[:23*(1+load_epoch)]
    preformance = list(np.load(RESU_PATH
        +'preformance.npy'))[:23*(1+load_epoch)]

#Set Loss
MSE = nn.MSELoss()
criter_discrim = nn.BCEWithLogitsLoss()

gen_group = [{'params': Dec.parameters()}, {'params':
    Rec.parameters()}]
#optimizer = optim.SGD(gen_group, lr=lr_gen, momentum=0.9)
#optimizer_hold = optim.SGD(Dec.parameters(), lr=lr_gen,
    momentum=0.9)


optimizer = optim.Adam(gen_group, lr = lr_gen)
optimizer_hold = optim.Adam(Dec.parameters(), lr = lr_gen)



BCE = nn.BCEWithLogitsLoss()
```

```
optimizer_decDis = optim.SGD(decDis.parameters(), lr=lr_dis,
    momentum=0.9)
optimizer_recDis = optim.SGD(recDis.parameters(), lr=lr_dis,
    momentum=0.9)


#labels for training discriminators
dis_labels = torch.ones(batch_size*2)
dis_labels[:batch_size] = 0
dis_labels = Variable(dis_labels.float()).cuda()

loss_labels = Variable(torch.ones(batch_size)).cuda()



print('Models Made')
print('-'*30)


for epoch in range(load_epoch+1, epoch_num):
    print('\tEpoch #', epoch)
    for i, data in enumerate(dataloader, 0):
        inputs = Variable(data['LR'].float()).cuda() #.to(device)
        labels_wc = Variable(data['WC'].float()).cuda()
            #.to(device)
        labels_hr = Variable(data['HR'].float()).cuda()
            #.to(device)
        # zero the parameter gradients
        optimizer.zero_grad()

        wavelets = Dec(inputs)

        loss_mse_wc = MSE(wavelets, labels_wc)*100

        logits_wc = decDis(wavelets)
        loss_bce_wc = BCE(logits_wc, loss_labels)

        wavelets_combine = torch.cat((inputs*2, wavelets), 1)
        outputs = Rec(wavelets_combine)
        loss_mse_hr = MSE(outputs, labels_hr)*100
        logits_hr = recDis(outputs)
        loss_bce_hr = BCE(logits_hr, loss_labels)
```

```
if epoch < epoch_on:
    loss_hr = loss_mse_hr
    loss_wc = loss_mse_wc
else:
    loss_hr = gamma*loss_mse_hr + (1-gamma)*loss_bce_hr
    loss_wc = beta*loss_mse_wc+(1-beta)*loss_bce_wc

loss = alpha*loss_wc+(1-alpha)*loss_hr
loss.backward()

if epoch < rec_hold:
    optimizer_hold.step()
else:
    optimizer.step()

#save statistics
if i%validate == 0:
    """
    print('Losses Step #', i)
    print('\tMSE_WC: ', float(loss_mse_wc))
    print('\tDis_WC: ', float(loss_bce_wc))
    print('\tMSE_HR: ', float(loss_mse_hr))
    print('\tDis_HR: ', float(loss_bce_hr))
    """
    #Training
    loss_comp_train.append([float(loss_mse_wc),
                            float(loss_bce_wc),
                            float(loss_mse_hr),
                            float(loss_bce_hr)])
    total_loss_train.append(float(loss))
    preformance.append(Preformance(Rec))

    np.save(RESU_PATH+'loss_comp_train.npy',
        np.array(loss_comp_train))
    np.save(RESU_PATH+'total_loss_train.npy',
        np.array(total_loss_train))
    np.save(RESU_PATH+'preformance.npy',
        np.array(preformance))
```

```
if float(loss_bce_wc+loss_bce_hr) < dis_lim and epoch >=
    epoch_on:
#train discriminators seperatly
    print('Dis Step #', i)
    for k, dis_data in enumerate(discriminator_loader,0):
        optimizer_decDis.zero_grad()
        optimizer_recDis.zero_grad()

        inputs_dis =
            Variable(dis_data['LR'].float()).cuda()
        labels_wc_dis =
            Variable(dis_data['WC'].float()).cuda()
        labels_hr_dis =
            Variable(dis_data['HR'].float()).cuda()

        #train decDis
        fake_start = inputs_dis[:batch_size, :, :, :]
        fake = Dec(fake_start)
        real = labels_wc_dis[batch_size:, :, :, :]
        wavelet_test = torch.cat((fake, real), 0)

        logit_decDis = decDis(wavelet_test)
        loss_decDis = BCE(logit_decDis, dis_labels)
        loss_decDis.backward(retain_graph=True)
        optimizer_decDis.step()

        #train recDis
        optimizer_decDis.zero_grad()
        optimizer_recDis.zero_grad()
        fake = torch.cat((fake_start, fake), 1) #add fake
            LR to WC
        fake = Rec(fake)
        real = labels_hr_dis[batch_size:, :, :, :]
        hr_test = torch.cat((fake, real), 0)

        logit_recDis = recDis(hr_test)
        loss_recDis = BCE(logit_recDis, dis_labels)

        loss_recDis.backward()
        optimizer_recDis.step()

        if k == k_steps:
```

```
                break

        if i == int(len(images)/batch_size)-1:
            break


    torch.save(Dec.state_dict(), RESU_PATH+'GAN_Dec'+str(epoch))
    torch.save(Rec.state_dict(), RESU_PATH+'GAN_Rec'+str(epoch))

    torch.save(decDis.state_dict(),
        RESU_PATH+'GAN_decDis'+str(epoch))
    torch.save(recDis.state_dict(),
        RESU_PATH+'GAN_recDis'+str(epoch))
```

Bibliography

[1] Addison, K. Sdo: Solar dynamics observatory, 2018.

[2] Canny, J. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8*, 6 (Nov 1986), 679–698.

[3] DEPYPERE, G. Forecasting of a solar wind classification using convolutional neural networks.

[4] Dong, C., Loy, C. C., He, K., and Tang, X. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence 38*, 2 (Feb 2016), 295–307.

[5] E. Rumelhart, D., E. Hinton, G., and J. Williams, R. Learning representations by back propagating errors. 533–536.

[6] Frazier, M. W. *An Introduction to Wavelets Through Linear Algebra*. Springer, 1991.

[7] Freeman, W. T., Jones, T. R., and Pasztor, E. C. Example-based super-resolution. *IEEE Computer Graphics and Applications 22*, 2 (Mar 2002), 56–65.

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

[9] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in Neural Information*

Processing Systems 27, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680.

[10] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016), pp. 770–778.

[11] Huang, H., He, R., Sun, Z., and Tan, T. Wavelet-srnet: A wavelet-based cnn for multi-scale face super resolution. In *The IEEE International Conference on Computer Vision (ICCV)* (Oct 2017).

[12] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR abs/1502.03167* (2015).

[13] John Duchi, E. H., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Rese 12* (2011), 2121–2159.

[14] Johnson, J., Alahi, A., and Fei-Fei, L. Perceptual losses for real-time style transfer and super-resolution. In *Computer Vision – ECCV 2016* (Cham, 2016), B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Springer International Publishing, pp. 694–711.

[15] Kim, K. I., and Kwon, Y. Single-image super-resolution using sparse regression and natural image prior. *IEEE Transactions on Pattern Analysis and Machine Intelligence 32*, 6 (June 2010), 1127–1133.

[16] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[17] Kumar, N., Verma, R., and Sethi, A. Convolutional neural networks for wavelet domain super resolution. *Pattern Recognition Letters 90* (2017), 65 – 71.

[18] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D.

Backpropagation applied to handwritten zip code recognition. *Neural Comput. 1*, 4 (Dec. 1989), 541–551.

[19] Ledig, C., Theis, L., Huszar, F., Caballero, J., Aitken, A. P., Tejani, A., Totz, J., Wang, Z., and Shi, W. Photo-realistic single image super-resolution using a generative adversarial network. *CoRR abs/1609.04802* (2016).

[20] Lee G., Wasilewski F., G. R. W. K. O. A. N. H., and Contributors. Pywavelets - wavelet transforms in python, 2006.

[21] Mallat, S. *a wavelet tour of signal processing.* Academic Press, 1998.

[22] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR abs/1312.5602* (2013).

[23] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (USA, 2010), ICML'10, Omnipress, pp. 807–814.

[24] NASA. Soho: Solar and heliospheric observatory, 2018.

[25] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch.

[26] Raptis, S. Processing solar images to forecast coronal mass ejections using artificial intelligence.

[27] Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., and Lee, H. Generative adversarial text to image synthesis. In *Proceedings of The 33rd International Conference on Machine Learning* (New York, New York, USA, 20–22 Jun 2016), M. F. Balcan and K. Q. Weinberger, Eds., vol. 48 of *Proceedings of Machine Learning Research*, PMLR, pp. 1060–1069.

[28] Roberts, L. Machine perception of three-dimensional solids. *In Optical and Electro-Optical Information Processing* (1965).

[29] Sajjadi, M. S. M., Schölkopf, B., and Hirsch, M. Enhancenet: Single image super-resolution through automated texture synthesis. *CoRR abs/1612.07919* (2016).

[30] Schawinski, K., Zhang, C., Zhang, H., Fowler, L., and Santhanam, G. K. Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. *Monthly Notices of the Royal Astronomical Society: Letters 467*, 1 (2017), L110–L114.

[31] Shen, G., Horikawa, T., Majima, K., and Kamitani, Y. Deep image reconstruction from human brain activity. *bioRxiv* (2017).

[32] Shi, W., Caballero, J., Huszar, F., Totz, J., Aitken, A. P., Bishop, R., Rueckert, D., and Wang, Z. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016).

[33] Shin, D. K., and Moon, Y. S. Super-resolution image reconstruction using wavelet based patch and discrete wavelet transform. *Journal of Signal Processing Systems 81*, 1 (Oct 2015), 71–81.

[34] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[35] Tai, M. M. A mathematical model for the determination of total area under glucose tolerance and other metabolic curves. *Diabetes Care 17*, 2 (1994), 152–154.

[36] Willett, R., Nowak, R., Jermyn, I., and Zerubia, J. Wavelet-based superresolution in astronomy. In *In Proc. Astronomical Data Analysis Software and Systems XIII* (2004), pp. 12–15.

[37] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 4148–4158.

[38] Wu, H., Zheng, S., Zhang, J., and Huang, K. GP-GAN: towards realistic high-resolution image blending. *CoRR abs/1703.07195* (2017).

[39] Yu, X., and Porikli, F. Ultra-resolving face images by discriminative generative networks. In *Computer Vision – ECCV 2016* (Cham, 2016), B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Springer International Publishing, pp. 318–333.

[40] Zhang, J., and Zong, C. Deep neural networks in machine translation: An overview. *IEEE Intelligent Systems 30*, 5 (Sept 2015), 16–25.

Department of Mathematics
Celestijnenlaan 200B
3001 Heverlee, België
tel.+32 16 327015
fax +32 16 327998
https://wis.kuleuven.be/