Detruncation of Attenuation Maps using Neural Networks

A Major Qualifying Project Report

submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Daniel Beckwith

Aditya Nivarthi

Akshay Thejaswi

Date: April 20, 2017

Approved:

Dr. Emmanuel Agu, Major Advisor

Dr. Clifford Lindsay, University of Massachusetts Medical School Advisor

Abstract

Single-Photon Emission Computed Tomography (SPECT) and Positron Emission Tomography (PET) are techniques used to image the human body for medical evaluation in areas such as cardiology, oncology, and cancer localization.

Our project focuses on improving SPECT imaging used in cardiology for detecting heart diseases and predicting heart attacks. The accuracy of SPECT imaging can be improved with additional imaging information known as *attenuation maps*, which are derived from Computed Tomography (CT) scans of the patient.

Extremely obese or misaligned patients cannot be completely scanned by the CT machine, which creates incomplete scans. The partial results from these scans reduce the accuracy of derived attenuation maps for SPECT imaging, which results in inaccurate evaluations of the patients' heart conditions.

The goal of this project was to develop an autonomous method to reconstruct incomplete attenuation maps that are provided for SPECT imaging by accurately generating the missing information in the associated CT images using modern image processing and deep learning techniques. We used image processing to extract the contour, or bounding shape of the body, from each of the CT scans. We then used a Convolutional Neural Network (CNN) to estimate the shape of the truncated regions in the contour. The corrected contour is then used to guide a voxel filling algorithm to produce a corrected CT scan.

By reconstructing the CT scans without requiring additional scans, changes to the CT imaging hardware, or changes to the scanning methods, we are able to generate complete and accurate information required for the SPECT scans at minimal cost to medical professionals and facilitate better diagnoses and treatments of cardiac patients.

Our results show that we are able to accurately correct truncation in CT scans that we artificially truncate, from a dataset of 1675 CT scans. From a cardiac imaging perspective, we are able to produce photon counts in our reconstructed SPECT images with a standard deviation of 1.22% percentage error. While our results for truncated scans from obese patients are subjectively plausible, thorough evaluation of these results in clinical diagnoses is considered future work at this time.

Acknowledgments

We would like to thank:

Dr. Emmanuel Agu, for his support and expertise in image processing and machine learning techniques.

Dr. Clifford Lindsay, for his support and expertise in image processing and medical imaging.

Dr. Petrus H. Pretorius, for his expertise with the medical evaluation pipeline of SPECT images for cardiac diagnoses at University of Massachusetts Medical School.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

This project focuses on improving of medical imaging technology used for diagnosing conditions, diseases, and cancers. Single-Photon Emission Computed Tomography (SPECT) and Positron Emission Tomography (PET) are two of the techniques used to image the human body for medical evaluation. [1] This project focuses on the use of SPECT imaging on the human body.

1.1 Basics of SPECT Imaging

SPECT imaging constructs a 3D representation of the scanned body, which can then be used for diagnoses and evaluation of medical conditions. [39] The next sections describe some uses of SPECT images and the SPECT imaging process.

1.1.1 Uses for SPECT Imaging

SPECT images are used for thyroid cancer localization in order to diagnose and monitor tumors, in oncology for diagnosing bone cancer, and in cardiology for diagnosing diseases [1]. SPECT images are vital for treating patients who have had heart attacks, bypasses or lesions in their hearts. [27] This project focuses on the use of SPECT imaging specifically in cardiology.

1.1.2 SPECT Imaging Process

The first step of the SPECT imaging process requires patients to undergo preparation, which includes injections of radioactive tracer material. These injections pass a radiopharmaceutical chemical into the patient's bloodstream, which eventually circulates throughout the body. The patient is then required to lie on their back on the scanner bed, and allow the SPECT scanner to scan around their body. The scanner rotates around the patient in order to capture the 3D information of the body. [35] A schematic of this setup is shown in figure 1.1.

The SPECT system uses a *gamma camera* to measure photons emitted from the radionuclides in the tracer material, allowing it to image the radiation produced from the tracer material. [1]

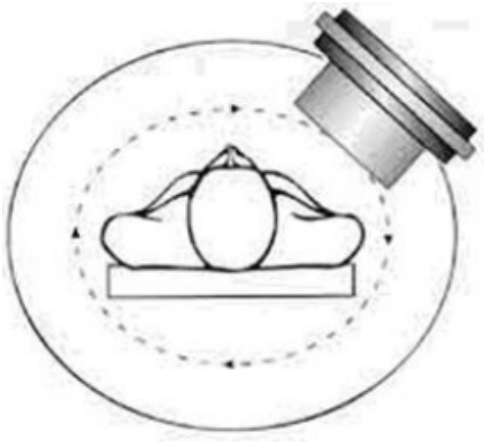


Figure 1.1: The setup of a SPECT imaging system. The SPECT camera rotates around the patient to capture the emitted gamma radiation. [39]

1.2 Computed Tomography Scans

Computed Tomography (CT) scans are used to reconstruct an image of the internal structure of the human body. CT images are created by projecting X-rays through the body to capture measurements from different directions. These measurements are combined using tomographic reconstruction to form a complete three dimensional volume of the body. [24] This 3D CT image of the body is comprised of voxel values that correspond to the linear attenuation coefficients of the internal body tissue. An example of a 3D CT image is shown in figure 1.2.

These coefficients also represent the density of the body across the image. The measurements of densities of different materials in the body are defined on the *Hounsfield* scale, some of which are shown in table 1.1. [17]

Material	**Hounsfield Units (HU)**
Bone	1,000
Liver	40 to 60
Grey matter	37 to 45
Blood	40
Kidney	30
White matter	20 to 30
Muscle	10 to 40
Cerebrospinal fluid	15
Water	0
Fat	-100 to -50
Air	-1,000

Table 1.1: Hounsfield values for various materials. [17, p. 3-4]

The voxel values in our dataset use a shifted version of the Hounsfield scale. In our dataset, material densities are shifted by +1,000 HU such that air starts at 0 HU instead of -1,000 HU. These values have been shifted to create a zero-based Hounsfield scale for the CT scans.

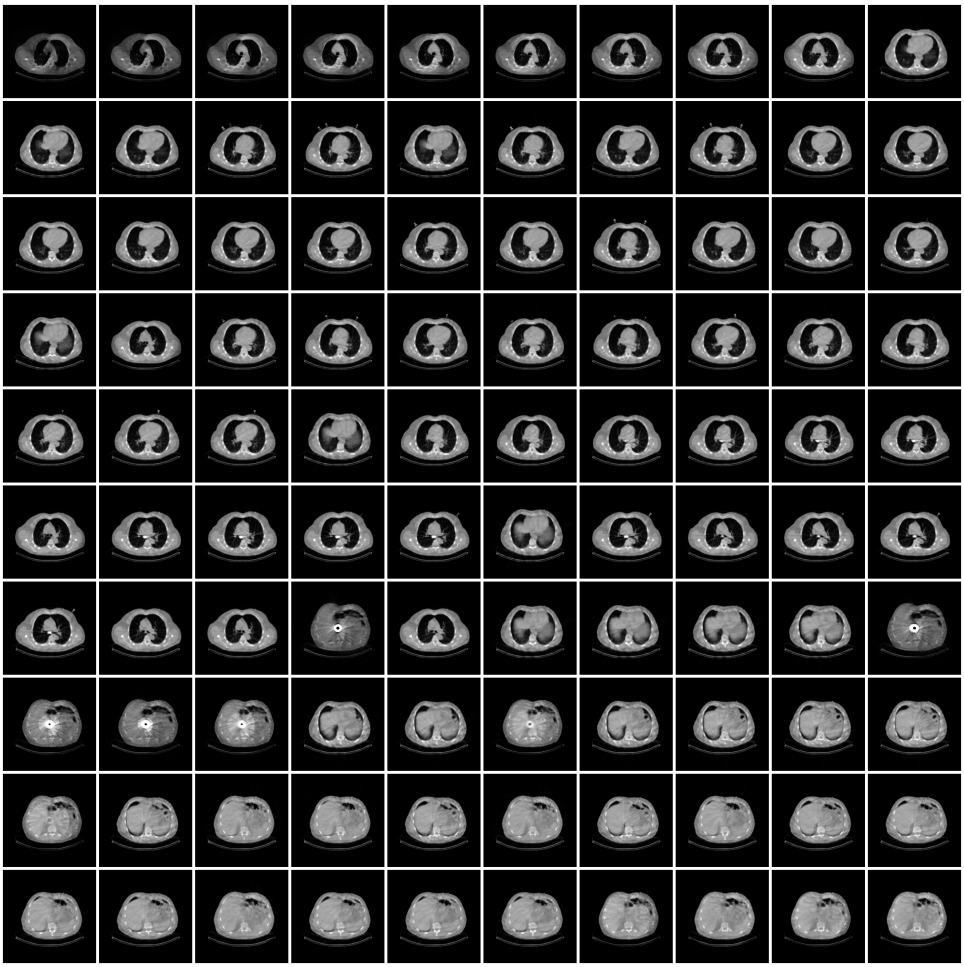


Figure 1.2: An example of a 3D CT scan, split into 100 slices. The slices are stacked vertically to form the three dimensional image of the body.

Using the known densities of various materials in the body, CT images encode information about the types of tissue that exist in the body to the SPECT system, which can then be used to improve the accuracy of imaging and further diagnoses. [1]

1.3 Attenuation Maps

When the SPECT imaging system scans the patient's body for gamma radiation, it detects the photons that are emitted. As these photons are emitted from different areas of the body, there is a probability that they will be absorbed or scattered by the tissue (called *attenuation*) within the patient's body, such as muscle, skin, or organs, prior to reaching the detector. [1] A diagram of photon scattering in a SPECT scan is shown in figure 1.3.

An *attenuation map* can be constructed to help the SPECT system correct for the attenuation of photons. A common method used for building attenuation maps involves converting the voxel values of the patient's CT scan, which are in Hounsfield units, to attenuation coefficients.

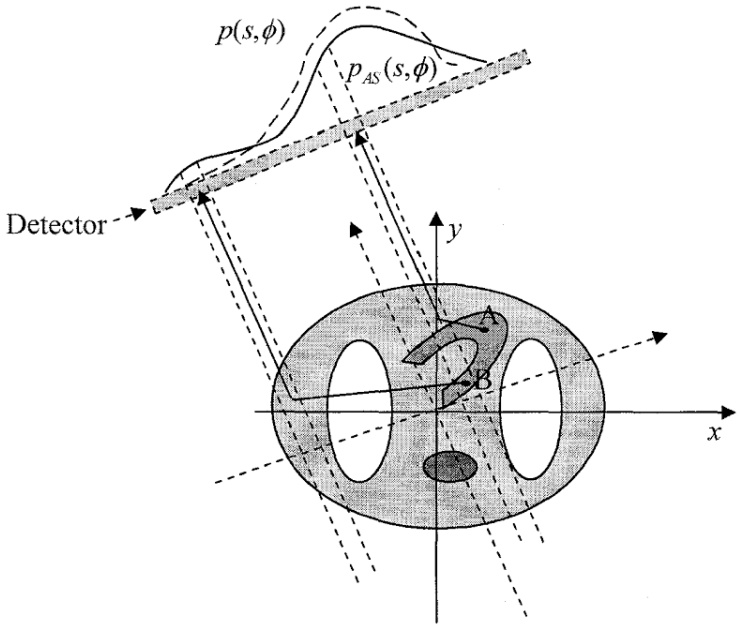


Figure 1.3: A and B depict photons that are scattered and detected along the incorrect line of response. $p_{AS}(s, \phi)$ is the projection data degraded by attenuation and scattering. $p(s, \phi)$ is the ideal projection. [56]

Equation (1.1) shows how to convert Hounsfield values to linear attenuation coefficients. [3]

$$\mu_x = \left(1 + \frac{\text{HU}_x}{1{,}000}\right) \times \mu_w \tag{1.1}$$

In eq. (1.1), HU_x is the Hounsfield units for the material x, μ_x is the attenuation coefficient of the material x in the CT image, and μ_w is the attenuation coefficient of water, which is $2.764 \times 10^{-2}\ \text{cm}^2 \cdot \text{g}^{-1}$. [42] These attenuation coefficients depend on the effective CT energy at which the scan was captured. [3] The CT scans in our dataset were taken at a CT energy of 140 kilo-Electron Volts (kEV).

Using the attenuation coefficients, an attenuation map can be constructed which describes how different parts of the body attenuate photons. At its core, an attenuation map is a representation of linear attenuation coefficients with their position in a 3D volume. The voxels within this volume correspond to the patient's anatomy, and the coefficients identify the density of regions throughout the patient's body. [57] Using these coefficients, the SPECT system can correct for absorption by compensating for photons that may have been emitted deeper within the patient's body but were absorbed before reaching the detectors, resulting in a more accurate image. [3, 48, 31]

1.3.1 Distortion in Attenuation Maps

While attenuation maps are used to improve the accuracy of SPECT systems, there are many cases where the attenuation maps are not completely accurate. There are a number of causes for distortion in attenuation maps, but some stem from distortions in the original CT scans, such as metal artifacts, motion, and truncation.

These CT scan distortions can lead to incomplete and inaccurate data, which introduces ambiguity into diagnoses and treatments for patients. This kind of uncertainty can lead to harmful or even fatal consequences for the patients. [27] Without the attenuation maps, the attenuation of photons will result in images of organs that are distorted, as shown by the example in figure 1.4. [31] In this example, the distortion leads to the mislocalization of lesions in the body.

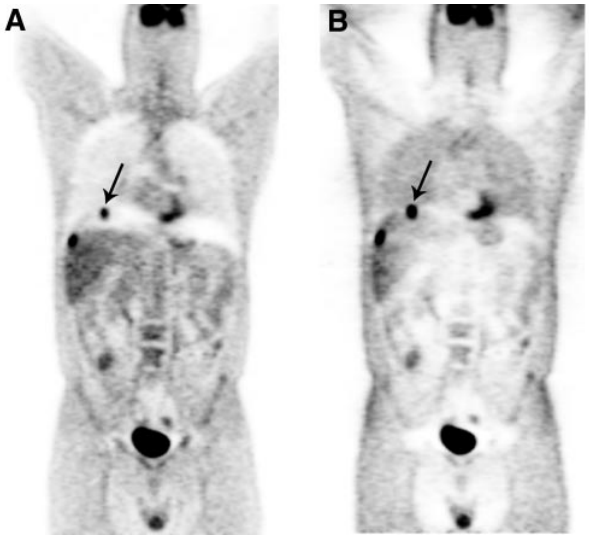


Figure 1.4: (A) 58-y-old man with colon cancer. Lesion at the dome of liver is mislocalized to right lung (arrow) because of respiratory motion. (B) Image without attenuation correction shows that all lesions are confined to the liver. [48]

Our project focuses on incomplete data in the attenuation maps caused by truncation, which itself is caused by truncation in the corresponding CT scans.

1.4 Inaccuracies in CT Scans

In order for the attenuation maps produced from the CT scans to be effective, the CT scans need to produce an accurate image of the patient. However, CT scans can exhibit several types of inaccuracies that need to be corrected before using them in further analysis.

1.4.1 Metal Artifacts

Some artifacts in CT scans are caused by metallic objects such as implants and medical devices, which may be present in or on the patient at the time of scanning. These artifacts usually manifest in CT images as areas with extremely high Hounsfield values due to their high photon absorption. This can result in a streaking pattern to propagate across the image and later become amplified during image reconstruction. The increased CT values lead to increased attenuation coefficients and result in an inaccurate correction of the attenuation in the SPECT scans. [48] An example of the effects of metal artifacts on scans is shown in figure 1.5.

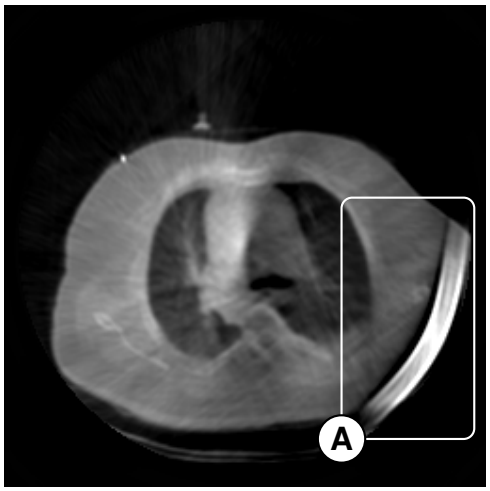


Figure 1.5: Example of a CT artifact due to metal. (A) The artifact.

1.4.2 Movement Artifacts

Significant artifacts can occur during the scanning process due to movements such as patient motion, respiratory motion, or heartbeats. These movements can result in cross-sections of the scan appearing blurry and can create inconsistencies across the scan as it proceeds across the patient's body. [48]

The main reason for movement artifacts lies in the way that CT scans are created. Movement artifacts occur because the time required to capture the image is longer than the time patients are able to hold a fixed position. [35] Patients can move externally with their limbs or internally with their organs or tissues and affect the CT image. [24]

Respiratory and cardiac movement inside the body can also cause artifacts in the resulting scan. The varying volume of air in the lungs can cause the CT system to image the body across respiratory cycles of the body. The lungs also cause larger movements in the rest of the torso of the body, due to their position and orientation around other internal organs. [35]

1.4.3 Truncation in CT Scans

CT scans can also be truncated such that parts of the patient's body do not get scanned. This is due to offset positioning within the scanner as well as limitations of the scanner's *Field of View* (FOV), especially with large or obese patients whose body extends beyond the FOV.

For example, an image captured with the patient's arms by their sides might result in the arms being cut off in the resulting scan, or a scan involving a large patient might produce a scan where the torso has not been fully imaged. [21] One example of a truncated scan is shown in figure 1.6.

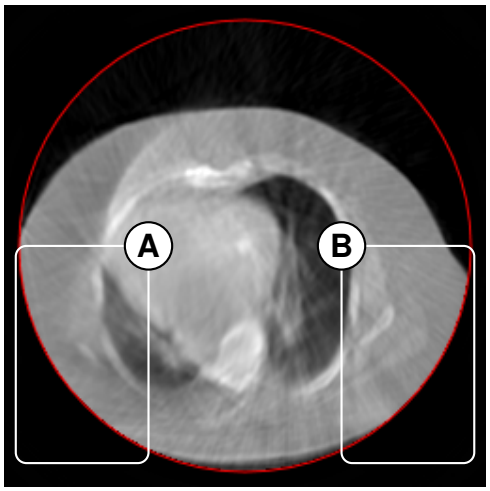


Figure 1.6: Example of truncation in a scan. The red outline is the FOV.
(A, B) are regions that extend outside the FOV.

The resulting scans are cut off at the edges of the FOV, thereby preventing attenuation values from being collected for the missing regions. This causes problems during the SPECT reconstruction phase, specifically during the attenuation compensation step.

A region of the body that is truncated in the CT image, and therefore in the attenuation map, will not have any attenuation data for that missing region. In turn, the SPECT system will not be able to compensate for photons that were attenuated in that truncated region. [55] The mismatch between the detected photons and the regions with attenuation values can cause an incorrect scaling factor to be applied to the final SPECT image, which can brighten or darken parts of the image. [21]

The varying intensity of regions in the image can change how parts of organs and different tissues are perceived by medical staff. For example, incorrect intensities can misrepresent the size and shape of the heart, from the outer wall to the apex. On the other hand, there can be cases where soft tissue located closer to the skin layer of the body is more enhanced than tissue located deeper in the body, distorting the perception of the inner tissues. These types of inaccuracies can cause incorrect diagnoses, ranging from perceiving that a treatment is working, to a scenario where an illness is not detected. [55]

Our project focuses on truncation caused by large or obese patients who may not fit within the scanner FOV, as well as minor misalignment of the patient on the scanner bed.

1.5 Methods for Correcting Truncation

Since the SPECT scans depend on the attenuation maps, which in turn depend on CT scans, this project is focused on creating a method to *detruncate* the CT scans. In our project, detruncation is the process of augmenting an image so as to extrapolate areas that were previously truncated.

There are two major categories of methods to detruncate CT scans: pre-reconstruction correction and post-reconstruction correction. Pre-reconstruction correction refers to correcting artifacts such as truncation before constructing volumetric CT data from the raw projection data. [21] Post-reconstruction correction refers to processing artifacts on the volumetric CT data.

Our goal is to detruncate the post-reconstruction CT images, and then derive the associated attenuation maps for evaluation. Working with post-reconstruction data allows our implementation to be more general and independent of reconstruction methods. We researched multiple methods to correct the truncation, which we describe in more detail in chapter 3.

1.5.1 Machine Learning

Machine learning is a popular method that has been shown to be effective for image correction and completion tasks similar to those in our project. Machine learning is the idea that we can teach a computer to *learn* from experience to do certain tasks and that its performance at doing those tasks improves as its experience increases. [41] The motivation for certain machine learning models, such as neural networks, comes from the internal architecture of the human brain, which these models attempt to emulate. [41] Machine learning has been used in areas related to our work such as object classification and image reconstruction. We will discuss machine learning in more detail in section 2.1.

Based on the nature of our problem, we have determined that machine learning is the best method for our problem because of its effectiveness and capabilities to complete tasks of similar nature to this project. We believe that a machine learning model can learn what a corrected CT image should look like based on our example data, and correct the incomplete scans.

1.6 The Goal of This MQP

Ultimately, the goal of our MQP is to improve the accuracy of SPECT scans in cases where the CT images used to generate attenuation maps are truncated. To accomplish our goal, we plan to achieve the following goals:

- Use modern image processing techniques to remove select artifacts from incorrect CT images and prepare the scans to be corrected for truncation.
- Use machine learning techniques to develop a detruncation pipeline that is able to learn from processed CT images in order to correct truncated CT scans.
- Generate and evaluate attenuation maps corrected by our system with an evaluation pipeline for SPECT imaging to measure the accuracy of our results.

2 Background

This chapter presents concepts and relevant background information about topics in this report to facilitate a deeper understanding of our project and provide a reference for the implementation of our project.

2.1 Machine Learning

Machine learning is defined by Mitchell [41] as follows:

Definition 2.1.1. Machine Learning: A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T measured by P, improves with experience E. [41]

The goal of machine learning is to develop computer programs that are able to mimic the methods that the brain uses to learn and understand. Similar to how humans learn, it is necessary to provide the program *examples* of the task or problem that it needs to learn to solve or complete. [8] These examples are another term for the previously mentioned *experience*. As the number of examples increases, the system gains a greater understanding of how to complete the tasks. [41] In our project, T is the task of detruncating CT scans that were truncated, P is the measure of accuracy of our detruncated scans based on ground truth image comparisons and medical evaluations, and experience E is the training process on a set of artificially truncated CT scans.

For both humans and a machine learning system, examples facilitate learning. Learning, in this case, is more than just acquiring knowledge, but also processing further information from that knowledge, called *insight*. [8] Humans use insight in order to solve problems or complete tasks of similar nature to those originally presented. In the same way, we can build machine learning systems to *generalize*, or apply knowledge and insight acquired through examples, and solve new and unseen examples or problems. [8] The ability to generalize knowledge and insight is vital to the success of a machine learning system.

2.1.1 Types of Machine Learning

Within the description of machine learning, there are a variety of approaches that can be used, depending on the goals or the problem to be learned.

Unsupervised learning is one method of machine learning where the system receives no feedback regarding the provided examples, and attempts to learn patterns in the input through

training. [45] This approach for machine learning is mainly used for clustering, grouping sets of inputs together based on common patterns or characteristics. [46]

Supervised learning provides the system with an expected result for a labeled input, also called the *ground truth*. [45] The system compares its output to the ground truth in order to update its machine learning model to reduce errors during model training. Supervised learning is the method that the majority of machine learning systems utilize. It is specifically used in applications where previous data is similar to or can predict future inputs. [46]

Reinforcement learning is a method of machine learning that utilizes rewards and penalties to train a system to achieve correct results. A metric is given to the system, and the goal of the system is to maximize its score against this metric. Good results or "behaviour" increases the score; bad results decrease this score. Based on rewards given for its output, the system can detect which of its actions are responsible for the corresponding consequences during model training. [45] This method is frequently used in robotics, where decision-based actions are more commonly used.

We plan to use supervised learning in our project because it provides feedback to the system on its output, which we are able to do since our input CT scans will be labeled as truncated or not truncated. Having ground truth data allows us to construct a system that can learn by reducing error between input data and ground truths.

2.2 Neural Networks

Machine learning stems from the concept of having a computer program learn in the same manner that human brains learn. In addition to this inspiration, there are also specific implementations of machine learning models that are built using mathematical models of how the brain functions. One of the most common implementations is a *neural network*. [45]

A neural network is a supervised learning model that emulates the brain's activity. It is composed of a number of *layers*, where the first layer is the input layer, the last layer is the output layer, and the layers in between are hidden layers. Each of these layers is comprised of *neurons*, or *nodes*, which are singular units of the network that process their inputs through functions and into outputs. The functions used to process the inputs are known as *activation* functions. These functions are used to introduce non-linearity into the network, which lets the network represent a nonlinear function. [45] An example of a neuron is shown in figure 2.1.

Neurons "fire" and send their outputs when the linear combination of their inputs is above some predefined threshold. The inputs to each neuron come from connections directed to that neuron, and the output from a neuron goes to another neuron as its input. The connections between neurons across layers are called *links*. Links are used to pass outputs to succeeding layers, and have their own associated *weight*. The weight is a numeric representation of the strength of that link between the connected neurons.

For a given neuron in the network, the network calculates a weighted sum of all its inputs,

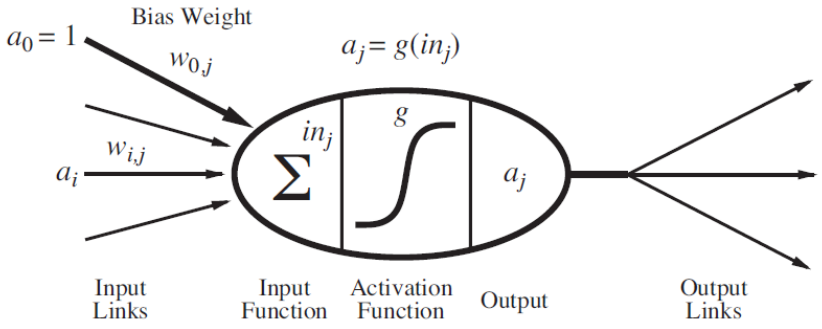


Figure 2.1: A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i, and $w_{i,j}$ is the weight from unit i to this unit. [45]

using the outputs from preceding nodes and the weights of the links to those nodes. The network then applies the activation function to the sum, which produces that node's output. [45] This process repeats for the activated neurons through the layers of the network, until the output layer is reached and the network produces a result for the original input.

While the basic neural networks can use the error between their output and the ground truth data to determine how well they are performing, that error does not completely explain what parts of the network need to be corrected to reduce that error, but rather only shows that the entire network is producing some error. One method to mitigate this issue is called *back propagation*. Back propagation allows the error from the output layer to be passed backwards through the hidden layers. At each node, a fraction of the error can be "blamed" on that node based on the weight of its connections to the output layer. [45]

We plan to use neural networks in our project. More specifically, we plan to use a derivative of machine learning known as *deep learning*, which introduces various types of neural networks and techniques that are appropriate for this project. We describe deep learning and its appropriateness for our project in section 2.3.

2.3 Deep Learning

One of the issues with conventional machine learning models, such as Artificial Neural Networks (ANN), Bayesian networks and Decision Trees, is that they are limited in their ability to process raw natural data. [41] Converting raw speech or image data into a representation that the network or model could process and extract features from requires a completely separate model that needs to be fine-tuned, almost to the same extent as the machine learning model itself. [36]

A newer form of machine learning, called *Representation Learning*, is a set of methods that enables machine learning models to take in raw data and translate them into internal representations that are better suited for pattern detection, feature extraction, and classification. [36]

Deep learning is an extension of representation learning and machine learning where multiple levels of representation are used on the same input data. The various forms of the raw data are also expressed in terms of other, simpler forms of the same data. [18]

Deep learning introduces computational models that are comprised of layers of modules, each capable of learning on one of the various representations of data that have been formulated. The benefit of deep learning models and their nested layers is that the layers of the models can identify key features using a general learning method, without the need for supervised input on important features. [36] An example of the connections between these layers is shown in figure 2.2.

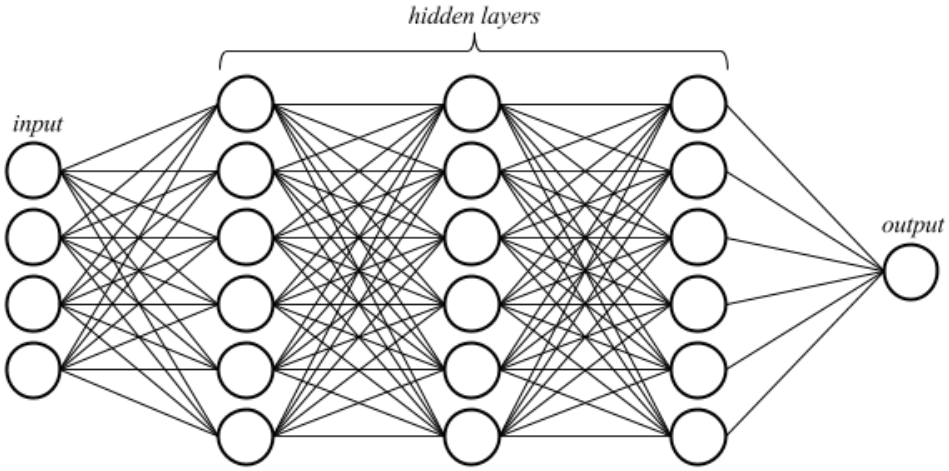


Figure 2.2: An example of the architecture of a deep learning model and the connections between its layers. [18]

We have determined that deep learning models are appropriate for this project based on the benefits that deep learning models provide for image completion problems over other models. Deep learning models have the ability to represent the input in multiple forms, each of which enables the model to extract a different level of features. Some models, such as Decision Trees, are unable to represent image inputs in a form that allows feature extraction. Decision Trees are more suited for problems that have inputs represented in simpler forms, such as attribute-value pairs. [41]

On the other hand, deep learning models have network layers used for feature extraction that are assigned weights automatically while the network is being trained, which results in a more generalized model that solves the problem. [22] These deep learning models have been used for work in image completion, object recognition, and image classification with much success. [18] We discuss some of these works in chapter 3.

2.3.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a neural network that utilizes learned convolution filters in order to extract a number of spatially-invariant features from its input. [18]

Each layer of a CNN takes its input and applies a set of convolution filters, which are locally connected and translation-invariant operations. This means that the convolution operates the same no matter where in the input it is applied. [53] For example, a CNN could accept an input tensor representing an image with dimensions of width, height, and three color channels red, green, and blue. One layer of the CNN could apply a parametrized averaging operator over an area of pixels, which would have the effect of blurring the image. See figure 2.3 for a diagram of the architecture of a CNN.

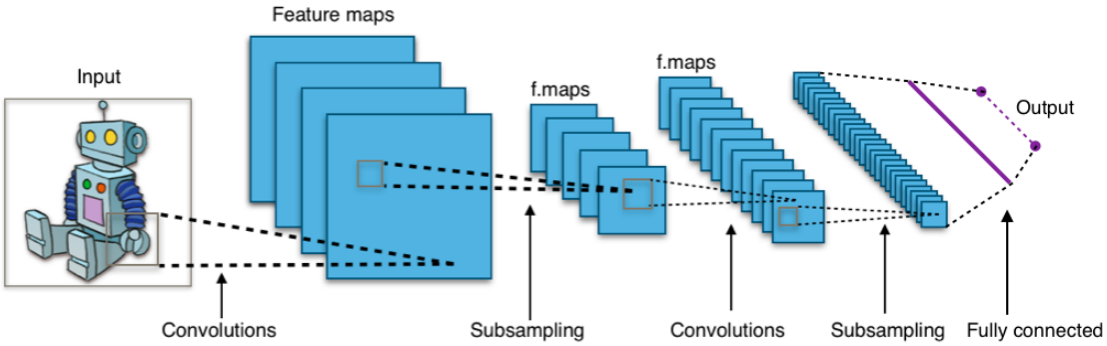


Figure 2.3: Typical CNN architecture. [2]

We concluded that CNNs would be appropriate for our project. The ability for a CNN to extract features from input images at different hierarchies allows it to be a more generalized model for solving the CT scan truncation problem at hand. [18]

2.3.2 Autoencoders

In machine learning, *encoders* are transformation functions used to convert a given input into an internal representation, or a *code*, that has reduced dimensions but retains the important features of the input. Encoders are useful in machine learning models where feature extraction is necessary, such as *autoencoders*. [18]

Autoencoders are a type of ANN used in unsupervised machine learning. The purpose of this type of network is to learn general features of the input in a way that can also be used to reconstruct the original input. The structure of a typical autoencoder consists of an input and output layer of the same size with a number of hidden layers between them. Typically the hidden layers are smaller than the input layer so that the network is forced to reduce the information in the input. The central hidden layer is called the code layer because it contains the network's lowest dimensionality encoding of the input. The network is trained by trying to minimize its reconstruction loss, which measures how closely the output of the network matches the input. In this way the network learns to reduce the dimensionality of its input and thus is learning high-level features. [4, 23] An example of the architecture of an autoencoder is shown in figure 2.4.

Autoencoders are especially useful for feature recognition on an input dataset. Given that

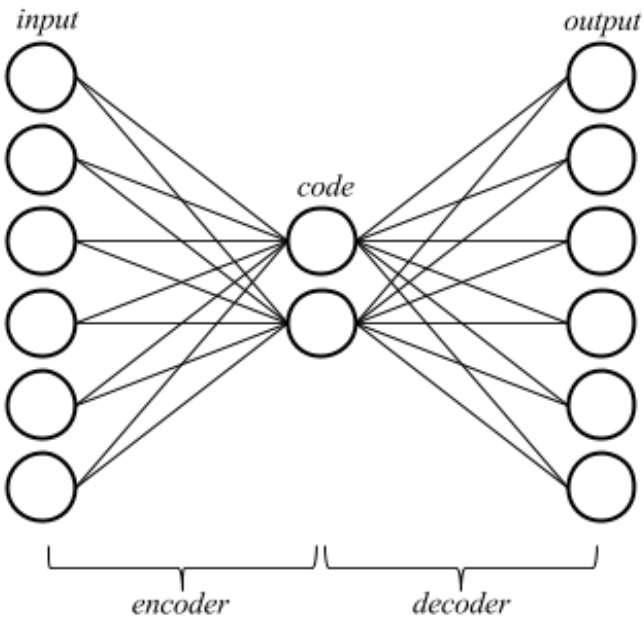


Figure 2.4: A diagram of the architecture of an autoencoder. [18]

the goal of an autoencoder is to rebuild its input, the dimension reduction it performs makes it powerful as a feature extractor. In addition, *denoising* autoencoders are a specific kind of autoencoder that have the purpose of recovering corrupt input and generating the original undistorted input. [18] Based on this definition, we would categorize our CT scan detruncation model as an denoising autoencoder because it has the effect of reconstructing truncated (corrupt) input to produce a detruncated (complete) output.

3 Related Work

This chapter presents relevant and existing work that attempts to solve similar problems to those in our project and may provide inspiration for the methods and implementation of our solution.

3.1 Other Considered Strategies for Correcting Truncation

This section details strategies that we considered applying in our project, but deemed insufficient to correct truncation.

3.1.1 Mirroring the Scan Region

One of the simplest methods for correcting truncation in CT images involves mirroring the image itself, which is done by replicating one half of the body onto the truncated half. The concept lies in the idea that the body in the scan is symmetrical on the outer edges, specifically where the layer of skin and bone exist. If the scan is truncated only on one side, the side that is missing can be filled in by using a reflected version of the side that is included in the scan.

The problem with mirroring is that one-sided truncation is not the only type of truncation that occurs in CT scans. In many cases, the patient is too large for the scanner and is truncated on both sides or in other areas. This method can also introduce inaccuracies, especially if the trivial vertical mirror axis is used. [38] An example of this is shown in figure 3.1 and figure 3.2.

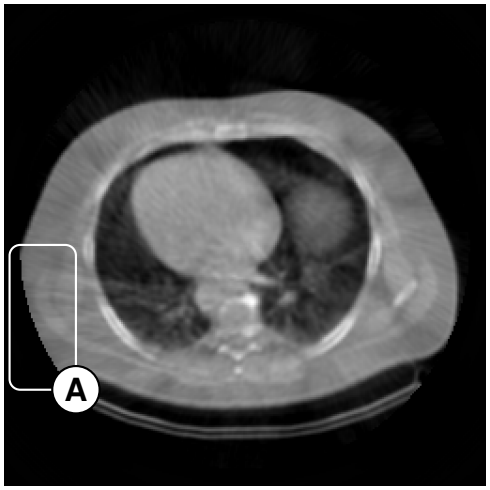


Figure 3.1: The truncated scan. (A) The truncated region.

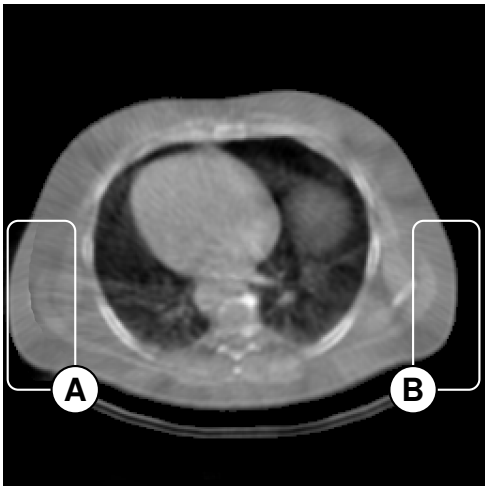


Figure 3.2: Corrected scan. (A) The corrected region. (B) The area that was mirrored onto (A).

In this example, sharp edges can be seen between the area mirrored and the original truncated edge. While additional processing could improve the region, the result would be anatomically inaccurate. In addition, given that the body is not perfectly symmetric, asymmetries are important to preserve in order to create an effective attenuation map.

3.1.2 Texture Synthesis

Texture synthesis is another method for filling missing regions of images. The idea behind texture synthesis is to use existing and accurate areas of an image to generate a texture for the areas that are missing. Texture synthesis can be used to generate a new image derived from an input texture, which would appear to be generated using the same process or function as the input texture. [54]

A texture synthesis model has two major components that are required to generate more texture data for an image. The first is a model that estimates the process or function used to create the original input. The other component is a sampling method that can efficiently produce new textures from the estimated process model. [54]

Texture synthesis has some drawbacks that make it infeasible for our project. One issue comes from the main difference between textures and *general images*. Textures have two important characteristics that make them different from general images: *locality* and *stationarity*. Locality refers to the idea that any given pixel in a texture is characterized by a small number of surrounding pixels. [51] Pixels in general images are not localized in this manner. A picture of a human face has distinct features such as eyes, mouth, and the nose that cannot be replicated with localized pixels. [54] Stationarity refers to the idea that given any window of the image, that window can be perceived as similar to any other window of the image. [51] General images are not perceived in this manner. [54] An example of this difference can be seen in figure 3.3, a general image, and figure 3.4, a texture image.

In our data, some of the regions that need to be corrected have hard edges or distinct layers of fat and muscle. Texture synthesis cannot model general features of human anatomy. For this reason, we decided that texture synthesis is not a sufficiently versatile method to correct for the truncation in our data.

However, we determined that texture synthesis can be used as a supplementary method for filling small areas of images of the body. After the initial correction for the truncation is complete, texture synthesis could be a viable method to generate voxel data in the corrected region, representing tissue, muscle, or fat, as required.

Figure 3.3: General image. (A) Region that cannot be synthesized or represented by texture function. [51]

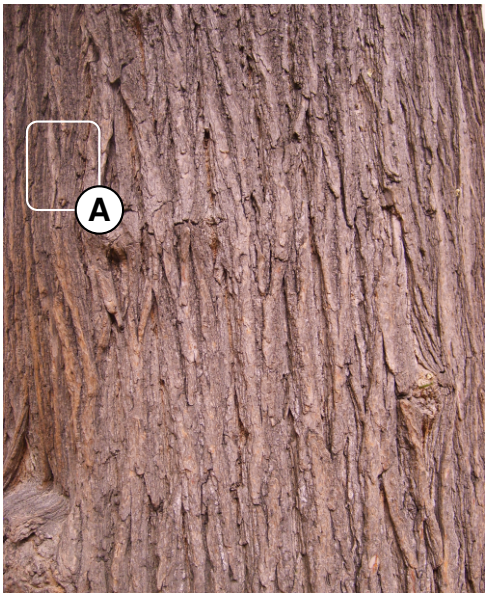


Figure 3.4: Texture image. (A) Region that could be synthesized due to its stationarity and locality. [14]

3.2 Adaptive Detruncation of CT Scans

A typical method to correct truncations in CT scans is by extrapolating the existing data outside the scan range. One such method, known as Adaptive Detruncation (ADT), presented by Sourbelle et al. [47], also corrects brightness artifacts at the truncation boundaries, as well as attempts to extend the image data beyond the camera's Field of View (FOV). This method depends on the assertion that the total attenuation of an object will remain the same at different projection angles, allowing the estimation of truncated regions based on non-truncated projections of the image. This type of algorithm performs very well with respect to correcting streaking artifacts, but it can only produce limited reconstructions of truncated data, and only when the object being scanned has a relatively low amount of contrast. [47]

3.2.1 Comparison of Adaptive Detruncation of CT Scans to Our Project

While the method created by Sourbelle et al. [47] attempts to solve the same problem, it is also different in that it takes raw projection data as input, not volumetric data. ADT analyzes each projection of the CT scan to figure out the location and total attenuation of the patient, which is then used to fill truncated projections and re-construct the final image. [47] By working on the projection data, the corrections for truncations become dependent on the type of reconstruction used for the CT scans. In contrast, our proposed method will be applied to the final CT image which is independent of the reconstruction method, thereby increasing its applicability.

3.3 Prior-Based Artifact Correction

Heußer et al. [21] use a second class of detruncation methods to fill in truncated regions using *prior data*. Prior data are previous scans of the same patient, or similar patients if none are available. To begin, the artifact regions in the incomplete scan are identified manually. Co-registration is then performed on the "donor" images, a process where the prior data is deformed in various ways to match the geometry of the target image as much as possible. This is done in order to minimize anatomical or positioning differences between the two scans, especially if the prior data was sourced from another patient. The prior data is then merged with the target data in the artifact areas or missing regions, and smoothing is applied to the boundaries in order to reduce any sharp edges between the two. This method was found to be more effective than extrapolation techniques in correcting all types of CT artifacts. We felt that this method would be particularly useful for attenuation correction, as it is able to estimate truncated anatomical features much more accurately than other methods. [21]

3.3.1 Comparison of Prior-Based Artifact Correction to Our Project

The method by Heußer et al. [21] is similar to our approach for this project in that it uses actual CT scan data to correct a truncated image, ensuring that the output will be anatomically plausible. However, it is limited to using a single example, whereas we attempt to learn general features from a large number of samples. It is also different in that a previous scan of the same patient can be selected for prior-based correction, so the corrected image may be more accurate for the selected patient in that case. Previous complete scans of patients with truncated scans are not available in our case, so this method cannot fully be applied.

3.4 Context Encoders for Inpainting

Inpainting is a method of learning features of an image in order to reconstruct missing regions of the image. Pathak et al. [43] used a type of autoencoder called a *context encoder* to learn features of various types of images and fill in artificially removed image regions based on surrounding regions. The context encoder was implemented using a CNN, with both the encoder and decoder also being CNNs.

The main result of the paper was that Pathak et al. [43] used adversarial loss to train the context encoder, in addition to the traditional reconstruction loss used with autoencoders. Pathak et al. [43] state that using only reconstruction loss can result in multiple plausible solutions for the filled in region of the image that fit the context, but the adversarial loss "tries to make the prediction look real, and has the effect of picking a particular mode from the distribution." [43, p. 4] Their results using adversarial loss were subjectively more convincing than those with only reconstruction loss.

For their reconstruction loss, Pathak et al. [43] use L2 loss, also known as Least Squares Error (LSE). LSE as a loss function attempts to reduce the sum of the squared differences between the ground truth and output. An example of one of the corrupt input images is shown in figure 3.5, an output using L2 loss in figure 3.6, and an output using L2 loss and adversarial loss is shown in figure 3.7.

Figure 3.5: Input context. [43]

Figure 3.6: Context encoder (L2 loss). [43]

Figure 3.7: Context encoder (L2 loss + adversarial loss) [43]

It is clear from figure 3.6 and figure 3.7 that the adversarial loss makes the inpainted region significantly sharper and more accurate than just using the L2 loss.

3.4.1 Comparison of Context Encoders for Inpainting to Our Project

Pathak et al. [43] attempt to solve a problem similar to the one in our project, but their work differs from ours in a few ways. First, the images they processed were two-dimensional and full-color, meaning two-dimensional input with three bounded component values (RGB color), whereas our input data is three-dimensional and has only one unbounded component value (Hounsfield values). Having two dimensional input with multiple channels can increase the size of the network, but the convolution filter is still two dimensional. On the other hand, our project uses volumetric data, which means the filter applied is across the three dimensions. This added dimension can cause increased memory consumption or longer training time, especially with a CNN that has a fully connected layer. [28]

The regions that were being filled by Pathak et al. [43] contained arbitrary objects at arbitrary locations in the image. Pathak et al. [43] had to create a model that could generalize to inpaint any type of content. On the other hand, filling in medical images is constrained to a domain of values based on the scale of possible values in the image, which in our case the Hounsfield scale. In addition, the truncation in our CT scans only occurs at the edges, which further restricts the domain of values to a select few types of tissues, specifically fat, skin, and some muscle. These differences will simplify our model and reduce our problem space significantly.

3.5 ImageNet Classification with DCNNs

Deep CNNs (DCNN) have previously been used for object recognition in images, specifically in the work by Ciregan et al. [10] and Cireşan et al. [11]. Krizhevsky et al. [33] make significant improvements to the implementation of a DCNN for object recognition.

The DCNN implemented by Krizhevsky et al. [33] consisted of 60 million parameters and 650,000 neurons, connected over 5 convolutional layers. Having a network of this size causes problems for training the network and *overfitting* the network to the training dataset. [33] Overfitting is the term that describes the idea that the machine learning model has been trained so specifically to the training data that it also describes the noise in the data, which results in strong overreactions to small fluctuations in future input data and is less generalizable to new inputs. [50]

Krizhevsky et al. [33] used a method of regularization known as *dropout* in order to reduce overfitting. In their case, dropout involved giving hidden neurons a probability of 0.5 of outputting a value of 0. In turn, these neurons do not contribute to feeding forward, nor do they contribute in back propagation. This enables the network neurons to remain independent, since neurons are not always present for every input. [33]

The network developed by Krizhevsky et al. [33] was trained using 1.2 million examples from the larger datasets on ImageNet, which is a large image database that is organized by descriptive keywords for the image content. [26] The network itself was too large to train on a single GPU, especially for a dataset of that size. To improve performance for training and reduce the network's error, the training for this network was performed on multiple GPUs with parallelization enhancements. Parallelization across GPUs was performed by storing the network partially on each GPU, and communicating inputs for neurons on certain layers across the GPUs. [33]

3.5.1 Comparison of ImageNet Classification with DCNNs to Our Project

The work done by Krizhevsky et al. [33] explores many of the concepts that we will need to incorporate into our project, but there are substantial differences from this paper and our project.

First, the paper mentions that the network was set up across GPUs to increase performance and the error on the outputs. The setup for allowing communication across GPUs for different layers required significant tuning in order to achieve acceptable results. Krizhevsky et al. [33] mentions using two NVIDIA GTX 580 GPUs, which are relatively outdated compared to some of the hardware that is currently accessible to us. We believe that our project can utilize the use of GPUs for training and improving network performance, but the parallelization that was used in this paper may not be necessary due to improvements in hardware at this time, such as the release of NVIDIA GTX 1070 GPUs.

A major difference from this paper and our project is in the use case of the network and the training dataset. Krizhevsky et al. [33] uses a large convolutional neural network for object classification, which involves training the network on a large and varied dataset such that it is able to distinguish a broad range of objects within sets of images. The larger scale was required to accommodate for a larger learning capacity, which was needed in order to meet the demands of object classification on such a broad spectrum of images.

On the other hand, our project involves a dataset which consists of solely CT scans, making our dataset much more restricted compared to the arbitrary images used for classification by Krizhevsky et al. [33]. In turn, this means our input dataset for training does not need to be as large as the one used by Krizhevsky et al. [33] The amount of data we estimate to be necessary for our project not only depends on the type of deep learning model, but also how medical images are used with deep learning, as described in section 3.6.

3.6 Medical Image Training Set Size Estimation Using Convolutional Neural Networks

There have been studies on using DCNNs specifically for medical images such as the work by Havaei et al. [20] and Hua et al. [25]. Another similar study, done by Cho et al. [9], explored a body part classification problem with medical images using deep learning. The main objective of this paper was to understand how much data is required in a medical image training set to classify medical images by the region of the body that they capture.

Cho et al. [9] aggregated CT images with body parts from 6 different classes: brain, neck, shoulder, chest, abdomen, and pelvis. They were able to collect approximately 6,000 CT images across these classes of body parts. They implemented the GoogLeNet network architecture for their classification problem, running each training set test for 200 epochs. For each of the body part types, they ran the training sets with sample sizes of 5, 10, 20, 50, 100, and 200 images.

The main result from the paper was the relationship between training set size and the classification accuracy of the network. Cho et al. [9] found a steep increase in the accuracy from 5 to 20 samples, and were able to fit a learning curve to the accuracies from each training set size. They estimated a 98% accuracy for 1,000 training samples, and 99.5% accuracy for 4,092 samples based on the fitted curve. The predicted results are shown in figure 3.8, and the actual results are in figure 3.9.

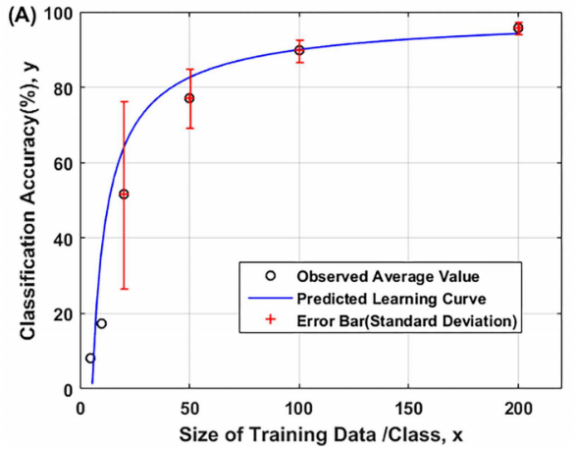


Figure 3.8: The predicted curve for accuracy across increasing training set sizes. [9]

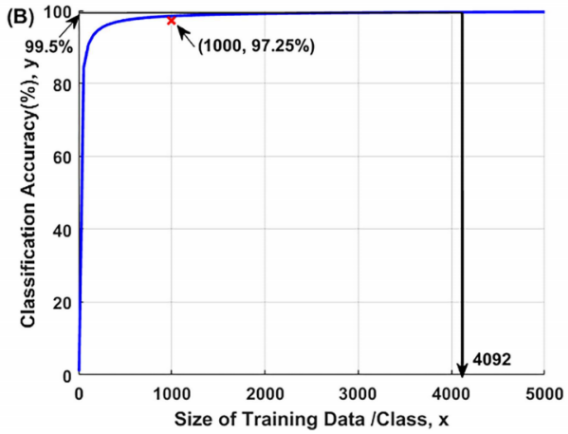


Figure 3.9: The resulting accuracy curve across increasing training set sizes. [9]

3.6.1 Comparison of Medical Image Training Set Size Estimation Using Convolutional Neural Networks to Our Project

The work done by Cho et al. [9] presents ideas that are relevant to our project, but there are some differences between their work and this project. Cho et al. [9] focuses on image classification, and the range of CT scans that they used is far larger than the dataset we use in our project. Our project is focused on image completion with CT scans, and the scans themselves are restricted to the torso of the human body, which reduces the values of inputs.

One major point we take away from the work by Cho et al. [9] is the estimation of required training set sample sizes, specifically for medical images. The estimation of 98% accuracy using 1,000 CT scans, and 99.5% with 4,092 scans is something we need to consider and provides reference for our project. Based on these estimates, we concluded that we need to have training samples in the range of 1,000 to 4,000 as well, and use subsets of that dataset for our training and validation sets, depending on the accuracy we are able to achieve.

We describe in section 4.1.1 how we determined scans that were usable for our training set. We determined that we had 1,169 non-truncated CT scans in our dataset. In section 4.2.2 we describe how we augmented the number of examples for our training set to 5,845, which we then divided into 4,092 samples for training and 1,753 samples for validation.

4 Methodology

This chapter presents the process we followed to reach the goals of our MQP. Figure 4.1 shows a block diagram of our methodology steps, which are described in detail in the following sections.

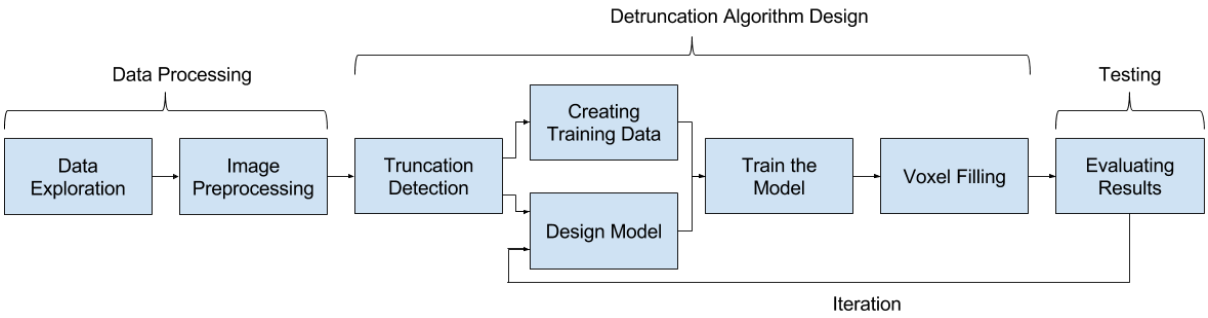


Figure 4.1: The main steps we followed to reach the goals of this project.

4.1 Data Processing

Initially, we explored our CT scan dataset in order to gain a clear understanding of the the kinds of scans and types of truncation we had to work with, and preprocessed the data prior to the detruncation phase. These topics are described in section 4.1.1 and section 4.1.2, respectively.

4.1.1 Data Exploration

Our data exploration step involved understanding how much data was in our CT scan dataset and how the CT scans vary. Variations in the CT scan data included patient body size, scanner and body alignment issues, CT artifacts that occur, and truncations that exist across patients. Our first exploration into the data involved gathering some simple statistics on the data prior to doing any processing or manipulation, shown in table 4.1.

Total data size	49.1 GiB
Number of patients	1,007
Total number of scans	1,674

Table 4.1: Initial patient data statistics.

In our dataset, each patient had two CT scans, a stress scan and a rest scan. An example of rest scan and stress scan are shown in figure 4.2 and figure 4.3, respectively.

Figure 4.2: An example of a rest CT scan for a patient, taken before exercise and activity.

Figure 4.3: An example of a stress CT scan for a patient, taken after exercise and activity.

The rest scan was taken prior to any exercise or activity, and the stress was taken shortly after the patient had either performed exercise or was injected with a substance that simulates cardiac stress. For our project, the distinction between the two types of scans was not considered a factor. (C. Lindsay, pers. comm.) There were many patients that only had one CT scan, in which case it was the rest scan. The statistics for these patients are shown in table 4.2.

Number of patients with rest & stress scans	667
Number of patients with only rest scans	340

Table 4.2: Statistics on the number of patients with both rest and stress scans vs. only rest scans.

Each scan was a volumetric model of the patient's body, divided along the Z-axis into *slices*, which are separate 2D planes of CT data that represent the patient's body when stacked together. Each slice was a 256×256 pixel image. The slices ranged from the upper torso near the arms down to the lower torso and gut. A diagram showing the CT scan dimension is shown in figure 4.4.

With the help of experts at University of Massachusetts (UMass) Medical School, we were able to compile a comprehensive list of patients and scans that would be representative of the truncation observed in clinical studies. This helped us categorize the CT scans into sets of non-truncated scans and truncated scans, without needing to label each image manually.

There were also some patients whose data we had to exclude from our dataset. These patients had CT scans that were distorted or had artifacts too prominent for us to be able to

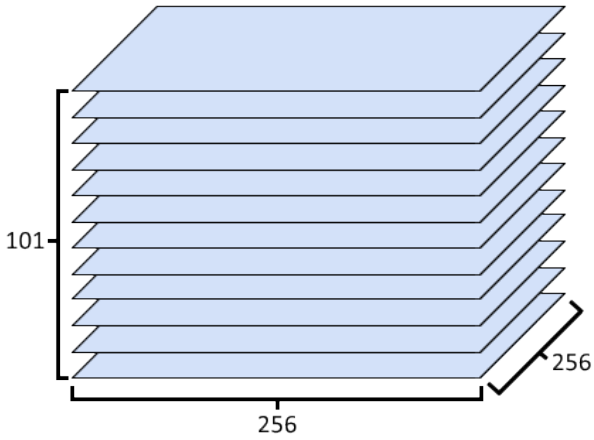


Figure 4.4: A diagram of the structure of our CT scan data. The data has 101 slices along the Z-axis, each of which is a 256×256 pixel image.

reasonably repair. The scans had issues such as distorted CT values across the scan, wing artifacts from raised arms remaining in the scan, missing cavities in the volumetric data, and metal artifacts that were not only difficult to correct, but also outside the scope of this project. One example of these scans is shown in figure 4.5.

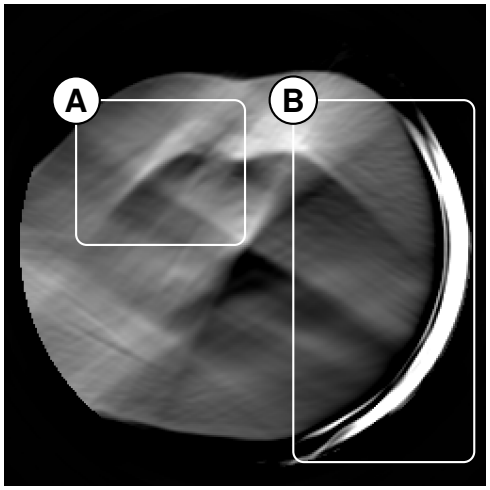


Figure 4.5: An example of a CT scan that had extreme artifacts that lead us to omitting it from our dataset. (A) Streaking artifacts. (B) FOV artifacts.

We took these scans to experts at UMass Medical School, who verified that those scans were reasonable to omit from our dataset. The result was that we omitted 9 scans from our dataset, which made up approximately 0.5% of all our data. This narrowed our final dataset to 1,665 CT scans, which were the aggregate of all the patients' stress and rest scans that we determined to be useable in our project.

4.1.2 Image Preprocessing

Once we gained a better understanding of our dataset, we preprocessed the data for the detruncation phase. This included removing artifacts from scans that caused nontrivial issues in our machine learning model, as well as identifying important characteristics that we could utilize during our learning process. We determined that the scanner bed in the CT scans and the CT cone beam artifacts needed to be removed before using the data in any subsequent steps. We describe our implementations for these steps in section 6.1.

4.2 Detruncation Algorithm Design

Our detruncation algorithm design phase involved creating the various components of our detruncation system. This included detecting truncation in CT scans, creating training data for our machine learning model, creating the model, and training our model to produce corrected CT scans.

4.2.1 Truncation Detection

As we mentioned in section 4.1.1, we were able to determine which scans were non-truncated and truncated based on information from experts at UMass Medical School. In addition to knowing if a scan was truncated, we needed to detect the regions and edges that were truncated in the scan to identify where to apply our detruncation algorithm. We describe our implementation of this step in section 6.2.

4.2.2 Creating Training Data

In order to train the machine learning model, we needed to create data that could be learned by the model in a way that the results could be evaluated automatically. The *ground truth*, or expected result, does not exist for scans that were already truncated. We decided that the best way to resolve this was to artificially truncate complete scans, which enabled us to have both a truncated scan and the ground truth to compare for error. We were then able to use the results and ground truth to quantitatively evaluate the efficacy of our model. There were some initial concerns with artificially truncated the complete scans, including:

1. The artificially truncated scans must be truncated in the same manner as the naturally truncated scans.

2. There could be some bias in training our model on only these artificial examples, since the body shapes and sizes could differ between non-truncated and truncated scans.

We created methods to artificially truncate the CT scans in a variety of ways, yielding a wider variety of training data to use, as described in section 6.4. Using these methods, we generated 5 artificially truncated samples per non-truncated CT scan in our dataset. This resulted in 5,845 examples in our training set, which during training we randomly divided into 70% for training and 30% for validation. This resulted in 4,092 training examples and 1,753 validation examples.

4.2.3 Deep Learning Model Design

As we created suitable training data for our machine learning model, we also developed the model itself. We developed a CNN for correcting the truncation, based on the Context Encoder by Pathak et al. [43]. The details of our context encoder model are described in section 6.5.

4.2.4 Training the Deep Learning Model

We used a supervised training approach to train our context encoder model with the training data created as described in section 4.2.2. During each epoch of training, the full training set of contours is divided into 81 separate batches, each with 50 contours. Each batch of training contours is then run through one training step on the GPU. The GPU runs each contour through the network to get an output, then calculates the Mean Squared Error (MSE) loss based on the output and the ground truth. Then the combined losses are used by an optimizer to update the weights in the network. The specific type of optimizer used is described in section 6.5. A full epoch passes when the network has trained on each contour. [5]

4.2.5 Voxel Filling

Using the output of our context encoder model, as described in section 6.5, we filled in the regions of the CT scan between the detruncated contours and truncated CT scans. We implemented a voxel filling algorithm to generate CT values for the missing regions, which we describe in section 6.6.

4.3 Testing

After we completed a working model for detruncation, we tested its output based on ground truth data, and continued to redesign and tune the model as needed. We also took the corrected scans of artificially truncated inputs and truncated inputs for further evaluation in order to better determine the model's accuracy.

4.3.1 Evaluating Results

We repeated the steps from section 4.2.3 and section 4.2.4 until our model's loss was below 1×10^{-4} or the model trained for 1,000 epochs. As detailed in section 6.5.1, we developed an early stopping feature in our training process that would terminate training when either the loss threshold or epoch count was reached, whichever came first. This was done to balance the accuracy of the model and the time required to train the model.

Based on the quality of the reconstructed scans, we were able to tweak various parts of our model, such as the convolution kernel size, convolution depth, or the number of encoder and decoder convolutions. Our adjustments to our model were based on the error of our model and our evaluation of the results from the model. Our experiments with these parameters are detailed in chapter 7.

In order to formally test the accuracy of the corrected images produced from our model, we used the images in further assessments. Using a SPECT reconstruction pipeline at UMass Medical School, we were able to use the corrected CT scan and reconstruct the original SPECT image, corrected with the derived attenuation map. We were then able to evaluate the attenuation-corrected SPECT image in a medical context, as described in section 8.1.2. Our results from this pipeline produced percent changes of photon counts between the original CT scan and our detruncated output. Overall, we anticipated producing increases in photon counts in the range of 5% to 10%.

5 Pilot Studies

This chapter describes various algorithms and methods that we developed to use as part of our detruncation pipeline, but abandoned due to feasibility or performance concerns.

5.1 Truncation Detection Using Intersection Arcs

As a part of our truncation detection step, described in section 4.2.1, we initially wanted to identify regions where the patient's body extended outside the scanner FOV. To detect these truncation regions, we developed a method to find the common regions between the patient's body and the scanner FOV.

We created a mask of the scanner FOV that could be positioned over the original CT scan to identify the areas where they intersect. This resulted in intersection arcs, which show where the truncation began to extend outside the scanner FOV. An example of this is shown in figure 5.2. The original scan with the truncation is shown in figure 5.1.

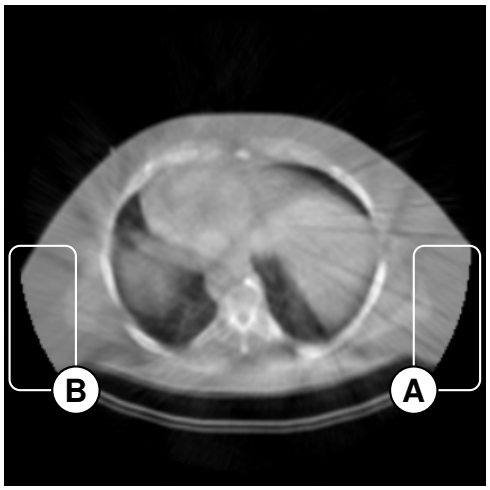


Figure 5.1: (A, B) The regions where the scan is truncated.

Figure 5.2: The isolated arcs that identify regions where truncation occurred.

This method was sufficient for detecting truncation, but we later created an improved method to detect truncation which is detailed in section 6.2.1.

5.2 Extending the Layer of Fat to Create Training Data

We described in section 4.2.2 that we needed to create training data for our machine learning model from non-truncated CT scans. In order to create this training data, we realized that we had to make the non-truncated scans appear similar to the truncated scans in order to cause similar truncations. One of the biggest differences between non-truncated scans and truncated scans was that the patient's body was considerably larger in the truncated scans. Figure 5.3 and figure 5.4 show that non-truncated scans are smaller, and truncated scans are larger, respectively.

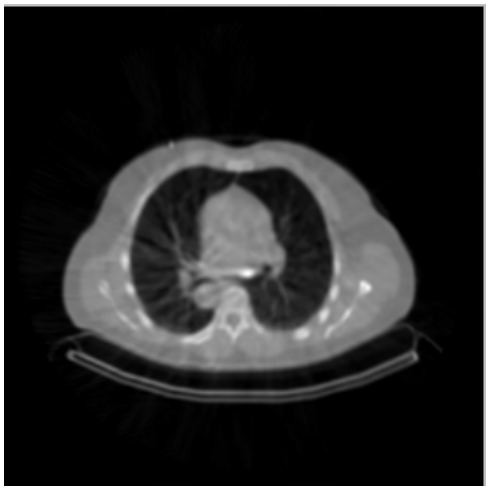


Figure 5.3: Non-truncated scans had smaller bodies.

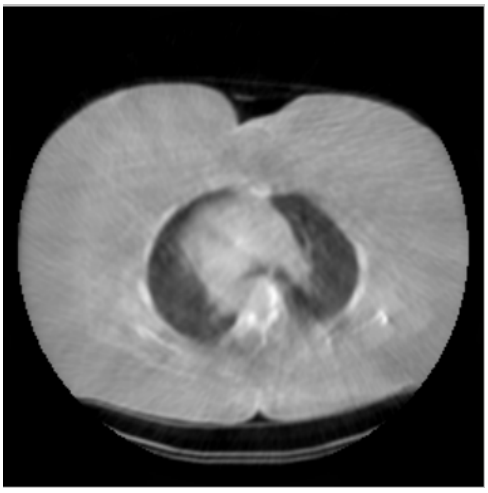


Figure 5.4: Truncated scans contained larger bodies.

The cause for the size variation lies in the fact that the truncated patients were far more obese than those who were not truncated. Some exceptions can be made for scans where the patient was not positioned properly.

Creating training data that best represented the the truncated scans in our dataset involved compensating for the difference in body sizes between non-truncated scans and truncated scans. To compensate for the body size in the non-truncated scans, we attempted to increase the fat layer on the smaller patients to a size large enough to cause truncation. To do this, we used our dataset truncation statistics to create a *truncated distribution* of the percent truncation across all the scans. We sampled this distribution and retrieved a percent truncation value that we would attempt to replicate on the scan that we add the layer of fat. By masking the scan and the fat with the scanner FOV, we were able simulate truncation and determine the percentage truncation on the training scan. We used a *binary search* to better approximate the truncation percentage as close as possible to the sampled value, modifying the fat layer thickness on each iteration.

To actually add the layer of fat, we used a contour-finding algorithm to find the contour of the patient, which is the outside edge of the patient's body. We reused this algorithm in

our final implementation, described in section 6.2.1, using OpenCV's **`findContours`** function. From the contour of the patient, we were able to calculate the horizontal width of the body, which we then used as a basis to extend the body fat. We calculated the thickness of the extra layer such that it would be slightly larger than the difference from the scanner FOV to the outer edge of the patient's body. This would allow some regions of the added fat layer to be truncated in a similar manner to those that were already truncated. We used a flat value for the CT values in the added layers, as described in section 5.4. An example of the result is shown in figure 5.6.

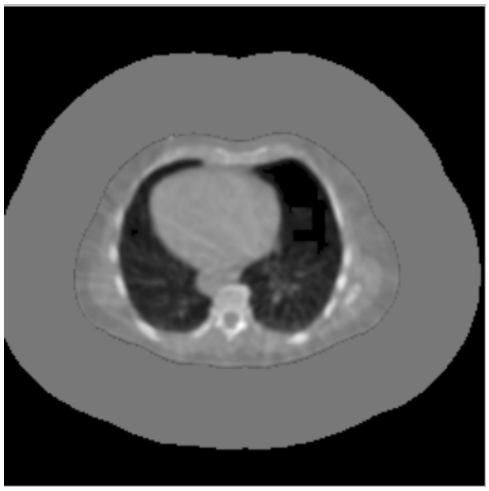


Figure 5.5: Non-truncated CT scan with added layer of fat.

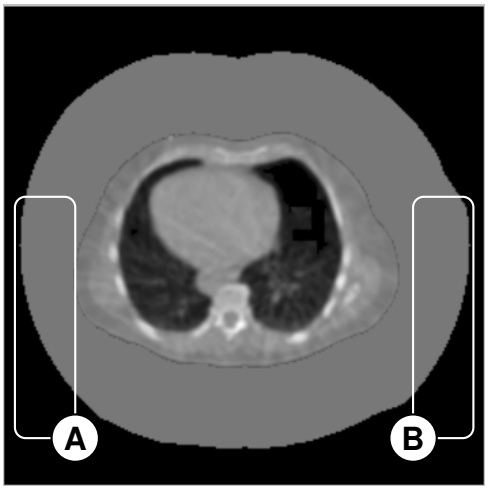


Figure 5.6: Non-truncated CT scan with layer of fat after truncation. (A, B) Truncated areas of added fat layer.

After some initial results with this method of extending the fat layer, we realized that it was not a realistic way to create good training data. The fat extension method created significant inaccuracies on patients who were significantly smaller and required much more surface area to cause truncation. The surface area did not scale reasonably in a way that would simulate a larger patient. We created a new method to obtain training data which is discussed in section 6.4.

5.3 Spline Curve Fitting for CT Scan Detruncation

One of our initial ideas for our detruncation model, outlined in section 4.2.3, involved using splines to estimate the missing regions of the CT scan. *Splines* are mathematical functions that define smooth curves or surfaces with precise control. [29] Our use of spline curves relies on the fact that the human body is naturally round and smoothly varying, especially on the sides where most of the truncation in our data is located.

The first step of fitting the splines involves finding the intersection arcs of truncation between the scanner FOV and the body, which we described in section 5.1. The endpoints of these truncation regions become the endpoints of the spline.

With these points, we estimated the angle at which each contour extended outside the scanner FOV. We used a set of points along the contour leading to the end point to determine the angle at which the contour exits the scanner FOV, shown in figure 5.7. This angle is the average gradient of the contour as it leads up to the spline endpoint.

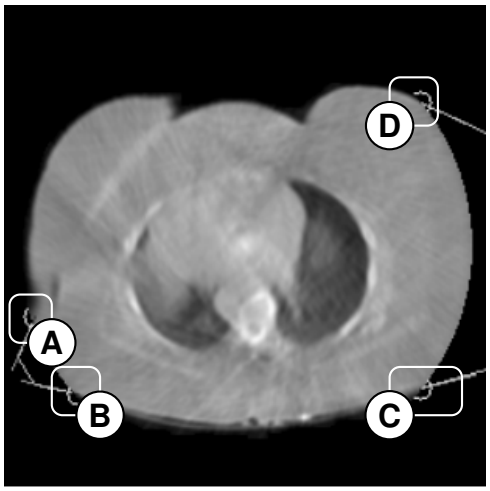


Figure 5.7: Spline control points for detruncation. (A, B, C, D) Control points for the splines.

Figure 5.8: Example of truncated areas estimated by splines.

We then defined control points for the spline. For our initial test, we simply used two fixed control points based on the endpoints' positions and angles. We planned to extend this to use a variable number of control points, whose positions would be determined by a neural network.

$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i \tag{5.1}$$

After the control points are defined, the curve needs to be filled. The outline of the curve is defined by the parametric equation shown in eq. (5.1), where $\mathbf{P}_i$ are the control points, $\mathbf{P}_0$ being the first endpoint and $\mathbf{P}_n$ being the last endpoint. [30] The result is shown in figure 5.8.

From the estimated regions, we then sample the values from the original body scan to produce a similar color and density for the fat. The result is an approximation of the shape of the original body. An example is shown in figure 5.10, in comparison to the original scan in figure 5.9.

The spline curve filling method proved useful as a means to correct the missing regions of the patient's body given pre-calculated curve parameters, but the variability of the missing regions' size and location, and the varying number of truncated regions in each scan made it unclear how we could design a neural network structure to model detruncation in this manner. We were unable to find a way to design a network that could output a single spline curve for any part of the body and have it be sufficiently accurate.

See listing A.2 for our Python implementation of this method.

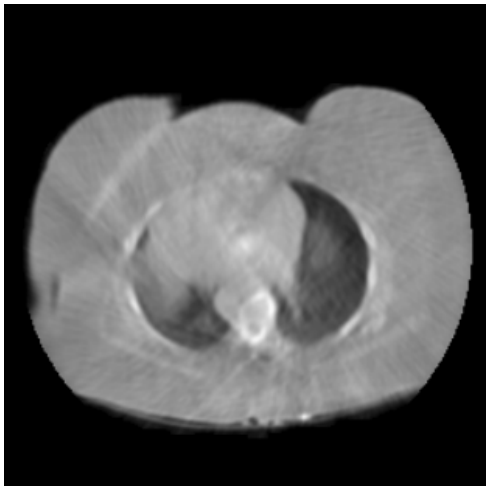


Figure 5.9: The original truncated scan, prior to being processed by spline filling.

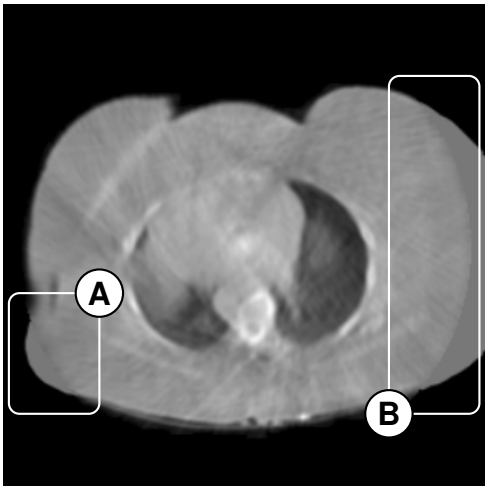


Figure 5.10: The spline curves estimate the body shape. (A, B) Regions filled by spline curves.

While we did not use spline curves in our final implementation, the concept of using the outer curvature of the body motivated our contour-based body detection and detruncation model described in section 6.2.1 and section 6.5, respectively.

5.4 Voxel Filling Using Flat Values

As described in section 6.2.1, we began using the contours of the bodies in the CT scans to detruncate the scans, and then use image processing to fill in the missing CT data between the original scan and the detruncated contour.

Our initial approach for filling the fixed contour data was to fill the missing regions of the scan with a constant value. We selected this constant value based on an average of the skin, fat, and muscle values in the CT scans being processed. We used this value across the entire region that was being filled, between the detruncated contour and the truncated CT scan.

This method was successful at providing an initial estimate of what the filled regions would look like after detruncation, and allowed us to better evaluate the quality of our detruncated contours. Eventually, we realized that this method was not sufficient for producing realistic CT values for the truncated regions. Our final voxel filling implementation is described in section 6.6.

6 Detruncation Pipeline Implementation

This chapter describes the specific details of our proposed pipeline to detruncate CT scans. We describe implementations of methods that we integrated into our final solution, which we developed through the steps in figure 4.1.

For our implementation, we determined that in order to go from a raw CT scan to a detruncated output, we would need a pipeline with multiple steps, each of which would handle a specific aspect of rebuilding the truncated scan. We found that it was more feasible to have each step focused on a specific sub-task of correcting the truncation in the data. The major steps in our pipeline implementation are as follows:

1. Image Preprocessing: Preparing the images for the detruncation phase by removing non-trivial artifacts, such as the scanner bed and the CT cone beam artifacts
2. Truncation Detection: Detecting truncated regions in the CT scans using contours
3. Data Visualization: An analysis step to allow us to view and understand the data in the context of medical imaging
4. Creating Training Data: Generating contour training data for our detruncation model
5. Contour-Based Detruncation: Reconstructing the contour of the patient body
6. Voxel Filling: Generating voxel values for the regions that were added in the contour detruncation

The truncated CT scans are processed through each of these steps in our detruncation pipeline, which produces the detruncated scan. An high-level overview of these steps is shown in figure 6.1.

Figure 6.1: Overview of the CT scan through our pipeline. (1) Original truncated CT scan. (2) Detected contour from CT scan. (3) Contour is detruncated using a CNN (4) Regions to be filled are identified using the detruncated contour (5) Voxel filling is applied to produce final result.

6.1 Image Preprocessing

As mentioned in section 4.1.2, our first major task was preprocessing the images before using them in the machine learning phase of our project. We described in section 1.4 that CT scans can contain various defects and irregularities that are obstructive to processing and image analysis. The next sections describe our implementations for correcting the inaccuracies we determined were obstructive to the processing and machine learning we required later in our pipeline.

6.1.1 Removing the Scanner Bed

The first major task for preprocessing involved eliminating the scanner bed from the images. The scanner bed is the flat area on which the patient is lying when the CT scan is acquired. The bed would always appear in the same location in the CT scans, and there were remnants of the bed within the attenuation map as well. An example of the artifacts of the bed in the CT scan and associated attenuation map are shown in figure 6.2 and figure 6.3, respectively.

In order to remove the bed, we first create a mask of the CT scan that captures the regions that contain Hounsfield values higher than air. To do this, we apply a threshold operation to the scan with the threshold value set to the maximum Hounsfield value of air, which results in a binary mask of the body with the bed. This mask includes the patient body and the scanner bed, as shown by figure 6.4.

We perform *erode* and *dilate* morphological operations to create a mask that contains only the larger body region, and not the bed. We specifically use the *erode* operation to degrade the smaller and thinner bed artifacts until they are no longer present, and the *dilate* operation for restoring the original size of the eroded mask back to the size of the patient's body. The bed does not reappear in the image after dilation because it was eliminated from the mask after the erosion step.

Internally, the *erode* operation iterates over each of the pixels in the image, and uses a

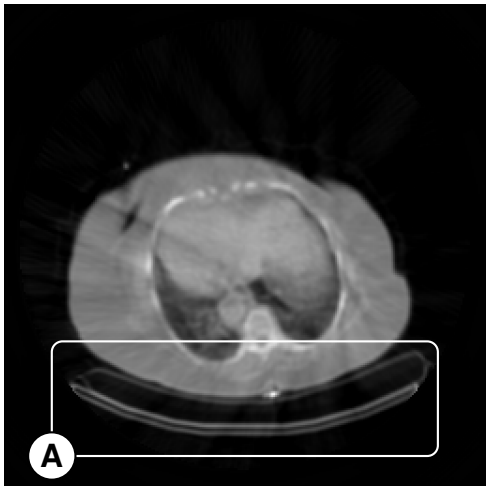


Figure 6.2: CT scan with bed artifacts. (A) The bed artifact.

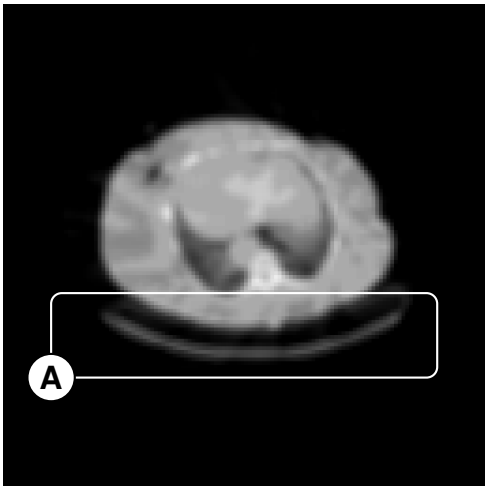


Figure 6.3: Attenuation map with bed artifacts. (A) The bed artifact.

Figure 6.4: Masked area containing patient body and scanner bed.

kernel to determine which pixels need to be eroded. The kernel is a matrix of size 3 $\times$ 3 on the image, centered on a given pixel. It takes the minimum value of all the pixels around the pixel in consideration and set the pixel to that value. This value can be 0, meaning it will be removed, or 1, meaning it will persist. This process is repeated across the entire image for all 3 $\times$ 3 windows . [6] The results of this can be seen in figure 6.5.

The *dilate* function rebuilds the edges of the body that were eroded, but maintains the loss of the bed artifacts. Internally, it is the opposite of the *erode* function. For each pixel considered in the kernel, we take the maximum value of all pixels around the pixel in consideration. [6]

The result is a mask with only the body, as shown in figure 6.6. We can take this mask and combine it with the original image to remove the scanner bed from the CT scan. This results in a image such as figure 6.7, where the bed is eliminated.

We used OpenCV's implementations of **`erode`** and **`dilate`** to perform the *erode* and *dilate* operations.

Figure 6.5: The result of eroding the bed from the CT image mask.

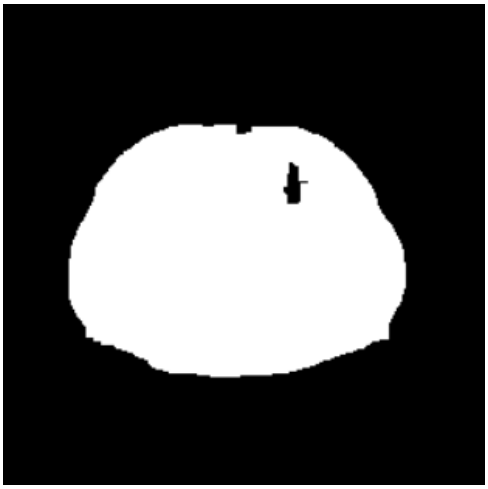


Figure 6.6: The result of dilating the mask after eroding the bed, in order to preserve the body in the mask.

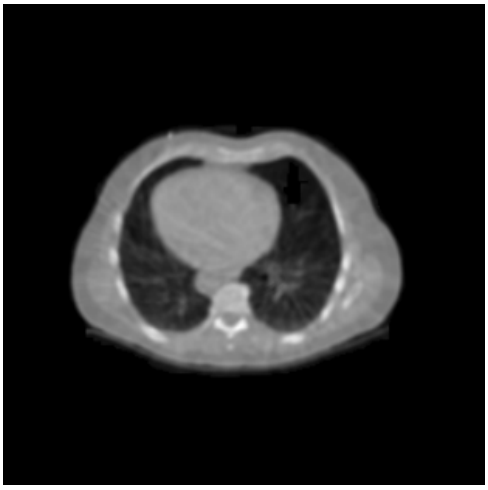


Figure 6.7: Final result after removing the scanner bed.

6.1.2 Eliminating CT Cone Beam Artifacts

One of the minor issues with CT scanners is that during the scan process, artifacts caused by the cone shape of the scanner can manifest in the beginning and final slices. This cone shape causes artifacts at the ends of the scan image, towards the top of the body and at the lower end of the body. An example of the cone artifacts is shown in figure 6.8, where the bright circle in the center is an artifact.

We eliminated the cone artifacts from the CT scans by removing a number of slices in the scan, as shown by figure 6.9. We removed the top 13 and the bottom 16 slices across all scans, which eliminated all cone-beam-related artifacts. Eliminating this number of slices brought all the scans from 101 slices to a total 72 slices. This number became advantageous later in our neural network because it is divisible by 8, a high power of 2, allowing that dimension to be

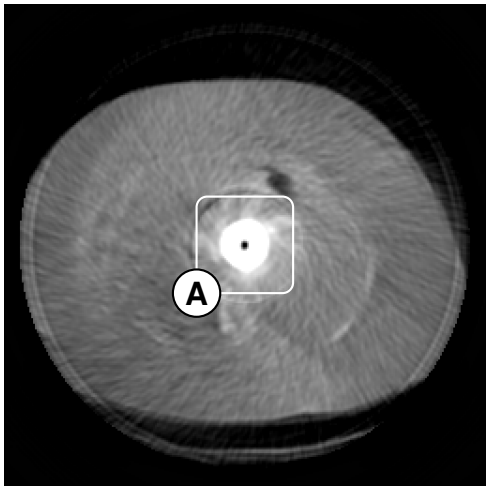


Figure 6.8: An example of the adverse affects of the CT scanner on the final image. (A) The cone artifact.

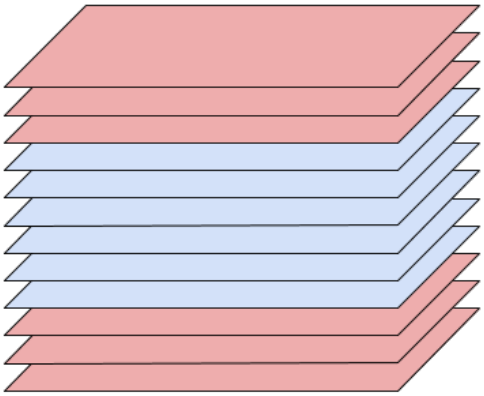


Figure 6.9: Diagram showing slices removed from the CT scan due to cone beam artifacts. Red slices are those being removed.

easily scaled down by a factor of 8.

6.2 Truncation Detection

As described in section 4.2.1, the next step in our pipeline process was to detect the regions of the scan that were truncated by the scanner FOV.

6.2.1 Contour Detection

As we started building a machine learning model for our project, we realized that we needed a new representation of the CT scans that could be easily understood by the machine learning model. Extracting features from a 3D representation of voxel values would be much more difficult because of the complexity that three dimensions adds to the machine learning model.

Many works in the past have utilized the similarity of the shape of the human body to cylinders as a means to coarsely represent the human body. [13] We started treating the images as polar coordinate functions, with the slices stacked vertically along the Z-axis. The slice was any layer of the data, along the vertical axis parallel to the patient's spinal cord.

By treating the scan as a cylindrical shape, we focused on detecting the entire outer *contour* of the body. The contour in our case is the outer boundary of the body, essentially the representation of the patient's outer skin layer in 3D space. In the case of truncation, the outer boundary is a delineation of the missing tissue.

We created a method to detect the contour of the patient's body, using a contour-finding operation. The contour-finding operation works by scanning the image for points that represent

outer border edges or internal edges from holes. When a point that meets these requirements is found, it is tracked in a table where all the discovered contours are stored. The current point is also matched with the parent contour to prevent overlapping or repeating contours. [49]

From the current point, the border is followed around the entire contour, and each pixel on the border is marked with a unique number for that contour. Subsequent contours that may overlap do not update the values, but only mark pixels that are not already marked. Once the border is completely followed, the scan is resumed for new points on borders. This process is repeated until the image is completely scanned. [49] We used OpenCV's **`findContours`** function to perform the contour finding operation. An example of a body contour for one slices of the CT scan is shown in figure 6.10.

Figure 6.10: An example of the contour around the patient body for a single slice.

The contour produced using **`findContours`** contained a variable number of points, averaging at 750 points. First, we converted this contour into a polar function for each slice, which lists the radii at regular angles around the body. We did this by sampling OpenCV's contour at each angle and interpolating between points. The radii around the body extend from the center of the FOV window in the image outwards to the skin layer.

The resolution of this contour is however, still too high to be used in our neural network, so we downsampled the contour to be 96 points per slice. We used 96 points because it was evenly divisible by 8, a high power of 2, which was useful later in our pipeline for further downsampling. The polar representation of the contour after downsampling is shown in figure 6.11.

Using these plots across the CT scans in our dataset, we were able to identify the scanner FOV radius R_S at 120 pixels. We found this radius based on the correspondence between plateaus in radii in the polar plots and regions of truncation in the CT scans.

The polar representation in figure 6.11 is for a single slice of the scan. From this representation, we are able to stack all the slices and produce the two dimensional representation of the contour. The 2D representation of the high resolution contour from **`findContours`** is shown in

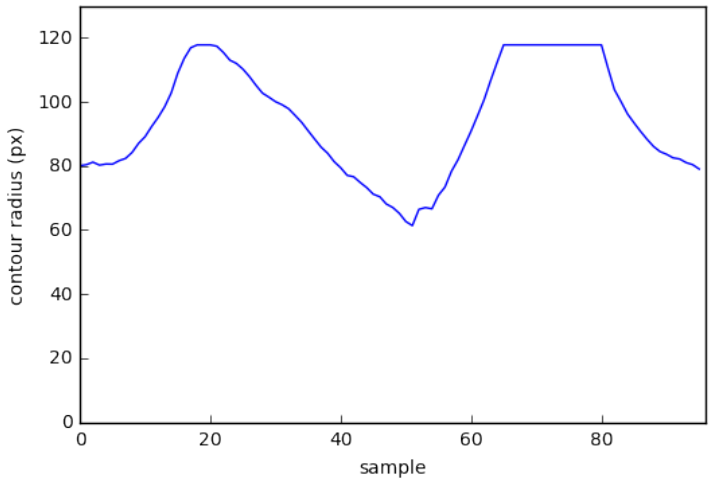


Figure 6.11: A plot of the contour of a single slice as a polar function. The X-axis shows the sampled point around the contour, and the Y-axis shows the radius in pixels. The plateaus at 120 px radius indicate truncation.

figure 6.12.

Figure 6.12: High resolution body contour with 377 contours points per slice, across 72 slices, plotted in two dimensions.

After downsampling, we get the 2D contour sampled at 96 points. This 2D representation is shown in figure 6.13.

For training our neural network, we also needed to categorize the scans as either truncated or not truncated. In order to determine if a CT scan as a whole was considered truncated, we first used the high-resolution contours. We looked at each point in the contour and marked points beyond the scanner FOV radius (R_S) as truncated points. We represented these contour points as a graph, where points in adjacent slices and at adjacent angles, including the angle extrema, are considered connected. Then, using a connected-components algorithm [12], we found all of the connected regions of these points, giving us the truncation regions of the contour. The body as a whole was considered truncated if the size of any region in both the slice-dimension and the angle-dimension is greater than w, the number of pixels between points in the training contour. This was done to preserve regions of truncation that appeared as noise in the downsampled contour, but were nontrivial in the high-resolution contour. Equation (6.1) shows the formula for this value and the value we used with our 120-pixel-radius scanner FOV and 96-point contour.

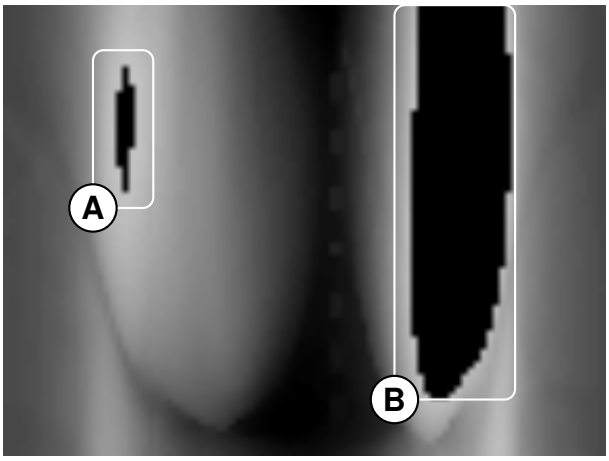


Figure 6.13: A 2D representation of the body contour, across all slices. (A, B) Regions that are truncated in the contour.

$$w = \left\lceil \frac{2\pi \cdot R_S}{N} \right\rceil = \left\lceil \frac{2\pi \cdot 120}{96} \right\rceil = 8 \text{ pixels} \tag{6.1}$$

We then downsampled the map of individual truncated points in the same way that we downsampled the contour, so that later in our pipeline we could still easily determine if individual points in the 96-point contour were truncated or not. Figure 6.14 and figure 6.15 show the results of this process.

Figure 6.14: High resolution truncation points. Artifacts in truncation detection present as vertical black lines at far left.

Figure 6.15: Downsampled truncation points. Artifacts are removed in the process of downsampling.

See listing A.1 for our Python implementation of this process.

6.3 Data Visualization

Visualizing our data, specifically the contours and the associated truncation regions, became an important step in our pipeline which initially helped us understand our input data, and later the accuracy of our outputs. Visualizing our data was mainly an analysis step, as opposed to a step along our implemented pipeline.

6.3.1 Truncation Percentage Statistics

Once we were able to detect the regions where the truncation occurred in a scan, we were able to calculate what percentage of the scan had been truncated. The distribution of the amount

of truncation across the truncated scans is shown in figure 6.16.

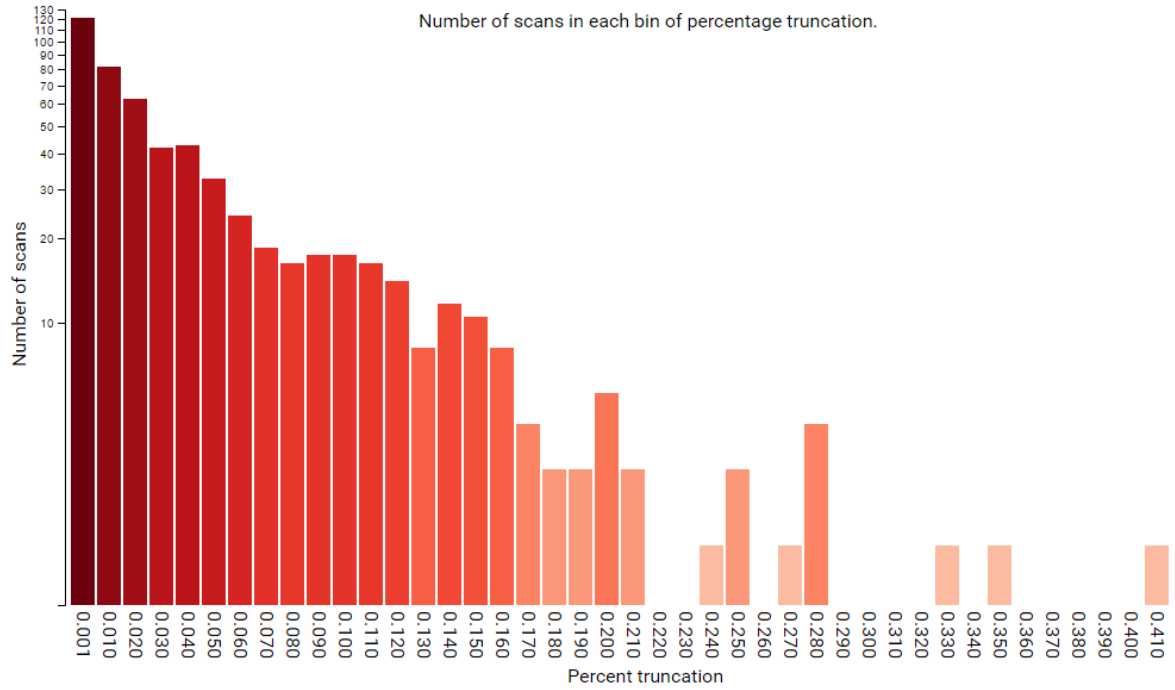


Figure 6.16: The distribution of percentage truncation values.

In this graph, the X-axis is the percentage bin, and the Y-axis is the number of scans in that bin. Based on our truncation threshold, we found that there were 1,169 scans that were considered not truncated, not shown above. The remaining 496 scans were truncated between 0.1% to 40.0%.

6.3.2 Visualizing 3D Contours

We created a way to visualize these contours that we had created, mostly as a way to understand how the truncated appears on the bodies and to verify we were generating the contours correctly. We used *three.js*, a JavaScript library made for rendering 3D objects [52].

With this visualization, we were able to identify the areas truncated in the contour, in a 3D view. This enabled us to really understand how the body was truncated. In the visualization, gray represents the area of the body not truncated, yellow represents the areas close to the scan limits that were not truncated, and red represents areas that were at or outside the scan limits, indicating truncation. An example contour view is shown in figure 6.17.

The visualization also lets us see across all the patients. We are able to generate a histogram of where on the body the truncation occurred the most. An example of this heatmap is shown in figure 6.18. It is clear from this heatmap that much of the truncation occurs on the upper left and right corners, which are the back left and right sides of the patient's body.

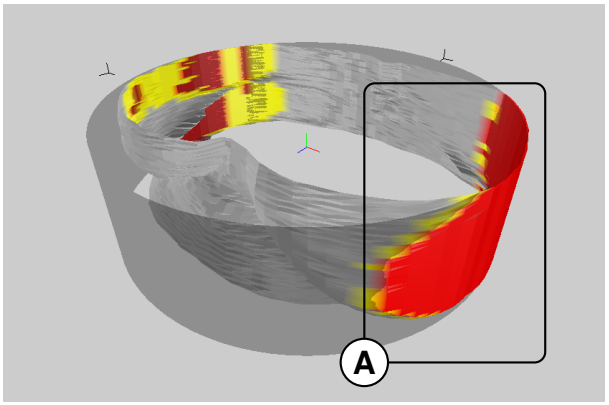


Figure 6.17: The 3D view of the contour shows us how the body was truncated on the actual patient. (A) Red represents truncated regions.

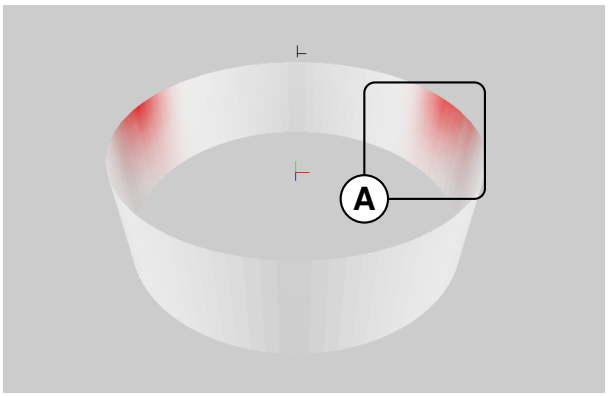


Figure 6.18: The 3D view of the histogram shows a heatmap of where truncation occurs the most. (A) Red represents regions with higher frequencies of truncation.

6.4 Creating Training Data

We determined that the best data for training existed in scans that were not truncated in any manner. From our dataset statistics, we were able to find scans that did not have truncation or had very minimal truncation that would serve as training data.

We devised a plan for creating training data, based on the contours of the patient bodies. We first grouped truncated scans and non-truncated scans into separate sets. From these sets, we then took scans from the set of non-truncated scans and simulated truncation on those scans based on truncations from the truncated scans. We simulated the truncation on non-truncated scans by making artificial truncation for radii in the contour larger than a certain percentage. This percentage is determined by randomly selecting a percentage from a threshold range. Our range is 85% to 95% of the maximum radius of the specified contour for the training set.

Once we determine the threshold for the radii to be truncated, we apply this threshold to the non-truncated scan. This results in a radius of 0 at each contour point that is above the threshold percentage.

In order to perform data augmentation, we repeat the process of generating a random percentage for the truncation threshold and artificially creating truncation multiple times for each non-truncated scan. This gives us a larger set of training data. A diagram of this process is shown in figure 6.19.

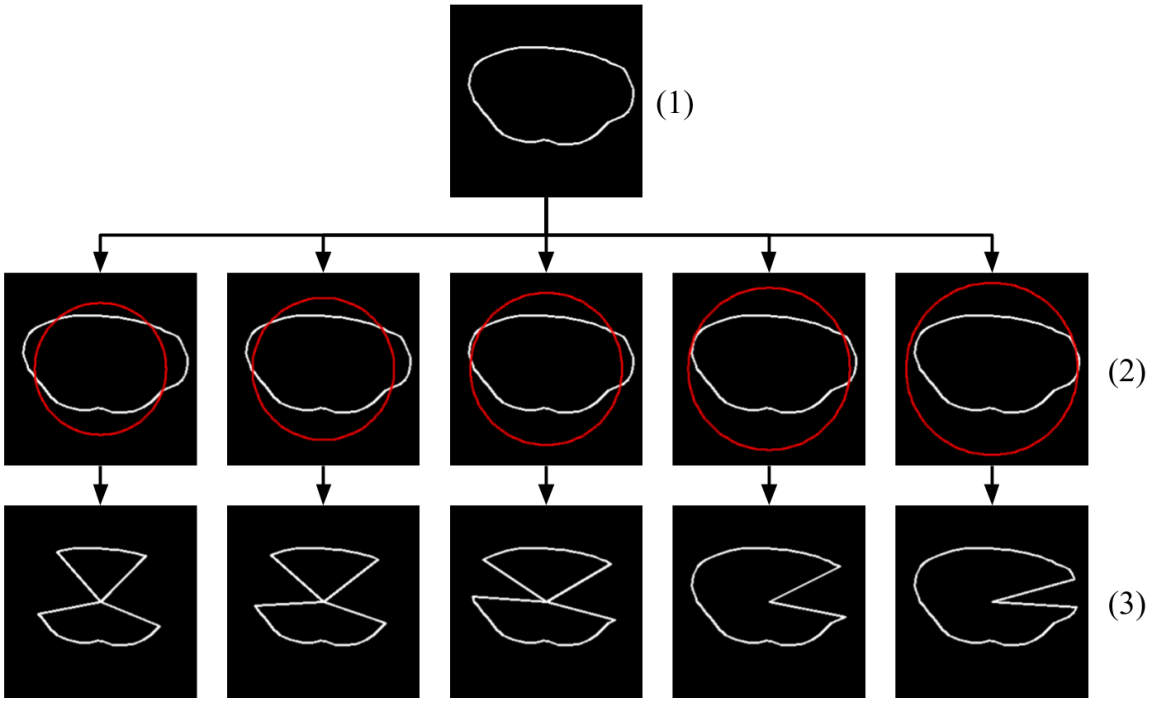


Figure 6.19: The process diagram of how we generate multiple contour training examples from a single contour. (1) The original, non-truncated contour. (2) A set of truncation thresholds are chosen. (3) The thresholds are applied to the contour.

We decided that sampling the existing truncation of other scans was better than our previous methods because it maintained the realistic characteristics of the body even when adding truncation. This method also does not require any additional logic to create material or truncation in such a way that mimics the physics and composition of human skin and fat, especially for a patient lying on the scanner bed.

6.5 Contour-Based Detruncation

Our CNN model for the contour reconstruction is built to process the contour data as single channel image, where the slices are on the X axis and the Y axis contains sampled contour points around each slice. Constructing the network to process all of the slices in a contour at once in three dimensions gives the network context about the contours across slices, which is important to preserve the overall coherence across the body.

We use a fully convolutional encoder and decoder, which are connected by a single fully connected layer. We connect the encoder and decoder with the fully-connected layer in this way because this "allows each unit in the decoder to reason about the entire image content". [43]

While our network is mainly modeled on the Context Encoder network presented in Pathak et al. [43], we made some modifications to our model to better suit it to filling in truncated contours. The encoder and decoder in our model both have 4 convolutional layers with pooling layers. The architecture of our network is shown in figure 6.20.

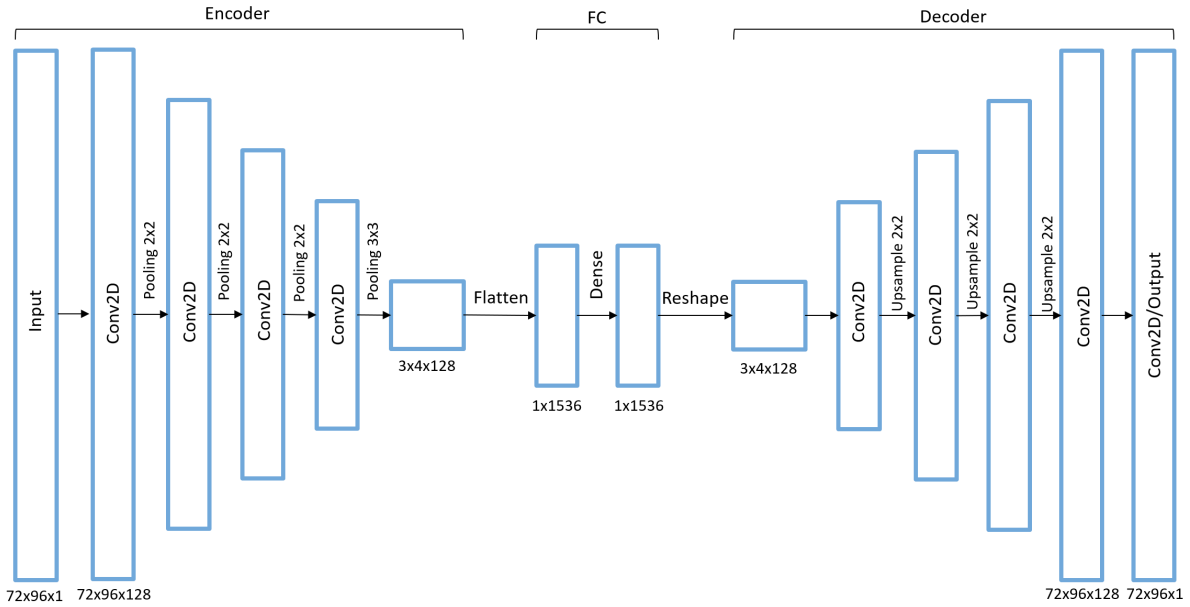


Figure 6.20: The architecture of our CNN.

In this diagram, the starting input is the entire scan contour, which has *nSlices* slices, each of which has *nSamples* contour points around the body. In our case, *nSlices* = 72 and *nSamples* = 96. An example of this 2D contour is shown in figure 6.21. For examples of intermediate feature map outputs from each layer in figure 6.20, see appendix B.

The input contour is passed to encoder, which applies a number of learned convolution filters to the image. The convolution filters used are 5×5, which allows each sample point in the input to map features from its immediate neighborhood and the slices above and below. Between each convolutional layer is a 2×2 pooling layer, which reduces the width and height of the image. For the last encoder layer, a 3×3 pooling is used in order to further reduce the size of the feature maps so as to restrict the number of parameters required for the following fully-connected layer.

After the fully-connected layer, the intermediate representation is passed to the decoder, which has a similar architecture to the encoder, except the up-sampling layers are used correspondingly to the pooling layers from the encoder such that the final output is the same size as

Figure 6.21: An example of the 2D input contour to our context encoder model.

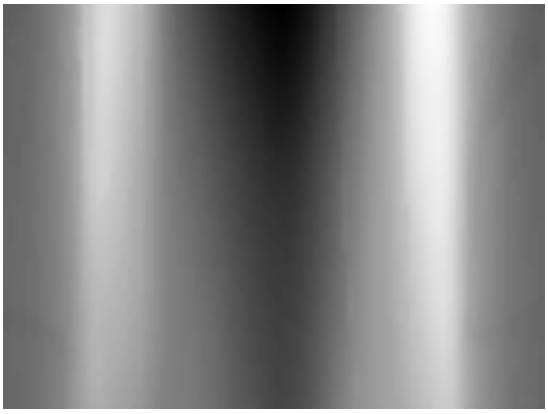


Figure 6.22: An example of the 2D output contour of our context encoder model.

the input. Since our network takes in truncated contours as input and attempts to reconstruct the detruncated output contour, we can say our network acts like a denoising autoencoder. [40]

The final output of our network forms a highly accurate approximation of the entire contour. This final output has approximations for the truncated regions in the contour, but it also contains approximations of the non-truncated regions as well. Since we already have information about the non-truncated regions of the contour, based on the original input, the final step in our model combines the input contour and the output of the network. We combine the two contours such that we preserve contour points from the original input where there was no truncation, and substitute the points from the model output where truncation existed. An example of this output is shown in figure 6.22.

6.5.1 Training and Optimization

We use Mean Squared Error (MSE) over all of the pixels in the contour as our loss function while optimizing our model, where the expected value is a ground-truth contour, and the predicted value is the output of the neural network given an artificially truncated contour. For optimization, we used the *Adam* optimizer, which has been shown to be successful in computer vision tasks using convolutions networks when compared against Stochastic Gradient Descent (SGD) and other algorithms. [32]

Duing the optimization process, we implement mini-batch training and an early stopping feature with a target error value and a maximum number of epochs. The optimization continues until the error has reached the target or the maximum number of training epochs have been performed, whichever comes first.

6.6 Voxel Filling

After generating the detruncated contour, we used them as a guide for the final shape of the body. The areas between the original truncated body and the new contours needed to be filled with CT values representing skin, fat, and muscle in order to fully reconstruct the scan. An example of this region is shown in figure 6.23.

Figure 6.23: A display of the difference between the detruncated contour and the truncated CT scan. (A) The region needed to be filled with voxel values.

In order to preserve existing CT data from the originally truncated scan, we used image masks to mask over our voxel filled regions with the original scan. This resulted in the original truncated scan to be overlaid on top of our voxel filling.

6.6.1 Material Profile Sampling

Our final method for voxel filling involved creating a profile of representative values seen along the contours of non-truncated scans. This profile was called a *material profile*. We used this material profile as a guide to fill in the detruncated contours.

To create this material profile, we marched along the contours of non-truncated scans and sampled the voxel values. For each slice, at each point of the contour, we marched inward along the normal of the contour. Through this process, we were able to extract a layer of material and tissue values from the body. The values we maintained in the material profile were the averages of sampled values that we obtained by marching along the normals of the contour, over all the non-truncated patients in our dataset. An image of the material profile is shown in figure 6.24.

With this profile, we could voxel fill other detruncated contours by sampling the profile in the regions that we needed to fill. To do this, we marched along the detruncated contour and filled in a quadrilateral mesh in each truncated region with values from the material profile. This process allowed us to capture the significant features along body contours, such as skin

Figure 6.24: One slice of the material profile we used for estimating voxel values. The vertical axis is the distance from the outer edge of the contour and the horizontal axis is the angle along the contour. There is one of these profiles for each slice of the scan.

thickness, skin density, fat values, and muscle values. An example of a filled-in region using this method is shown in figure 6.26.

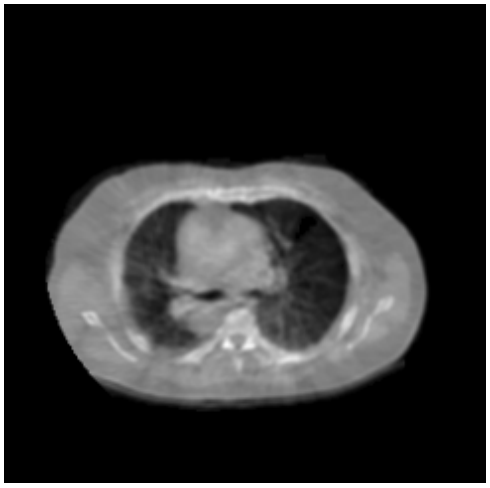


Figure 6.25: The truncated scan being filled with the region shown in figure 6.26.

Figure 6.26: The filled region sampled from the material profile.

7 Experiments to Optimize Our Pipeline

This chapter describes the various experiments we tried in our project, parameters used, and the hardware and software we used for development and our experimentation.

7.1 Software Specifications

Table 7.1 describes the various software platforms we used for our project. Listed are the names, version numbers and platforms that we used.

Name	Version	Operating System(s)
Anaconda	4.2.0 (x64)	Linux x64, Windows x64
Keras	1.2.1	Linux x64
NVIDIA CUDA Toolkit	8.0	Linux x64
NVIDIA cuDNN	v5	Linux x64
OpenCV	2.4.15	Linux x64, Windows x64
Python	2.7.12	Linux x64, Windows x64
SciKit Image	0.12.3	Linux x64, Windows x64
TensorFlow (GPU)	0.12.1	Linux x64

Table 7.1: Software distributions and platforms used in our development.

The image processing tasks we completed were developed and tested cross-platform on Windows and Linux, but our Keras and TensorFlow machine learning was done exclusively on Linux due to lack of support for TensorFlow with Python 2.7 on Windows systems at the time.

7.2 Hardware Specifications

Table 7.2 describes the various machine configurations we used for our project.

CPU	RAM	GPU
Intel i5 4670k	8GB	NVIDIA GeForce GTX 1070
Intel i7 4720HQ	8GB	NVIDIA GeForce GTX 970m
AMD FX-6350	16GB	AMD Radeon RX 480

Table 7.2: Machine configurations used in our development.

Our machine learning model training was done on an NVIDIA GeForce GTX 1070 and NVIDIA GeForce GTX 970m. Our image processing was all done on the systems listed in table 7.2, and is operating system independent.

For machine learning, our benchmarks and performance are based on the capabilities of the GPUs listed in table 7.3.

GPU Name	RAM	VRAM	CUDA Cores
NVIDIA GeForce GTX 1070	8GB	GDDR5	1,920
NVIDIA GeForce GTX 970m	3GB	GDDR5	1,280

Table 7.3: GPU specifications for the machines used in our development.

7.3 Model Configurations & Hyperparameter Search

Once we implemented our context encoder model, as described in section 6.5, we were able to train it under various configurations of parameters. We analyzed the results of each of these configurations to determine which set of parameters produced the best results with regard to accuracy and training time. Our goal with these experiments was to find the parameters that minimized our errors for training and validation sets while preventing overfitting. All of the experiments in this section were performed using an NVIDIA GeForce GTX 1070, unless otherwise specified.

7.3.1 Kernel Size & Depth Search

Our initial experiment focused on testing combinations of convolution kernel size and convolutional filters. In the following figures and tables in this section, D is the depth, or number of convolutional filters, and K is the convolution's kernel size. *Training Cost* is the final error of our context encoder model on the training set, and *Validation Cost* is the final error on the validation set. A list of our configurations is shown in table 7.4.

D	K	Training Epochs	Training Samples
32	3	250	1,000
32	5	250	1,000
64	3	250	1,000
64	5	250	1,000
128	3	250	1,000
128	5	250	1,000
256	3	250	1,000
256	5	250	1,000

Table 7.4: Configurations of our context encoder model with variations on kernel size and depth for our experiments.

For these configurations, our control parameters were the number of training epochs and the number of training samples. We used 250 epochs for all our experiments, and divided 2,000 training samples evenly between the training set and validation set.

For each of the configurations in table 7.4, we measured the memory usage, training time, final training set error (Training Cost), and final validation set error (Validation Cost) of the model, shown in table 7.5.

D	K	Parameters	Memory (MB)	Time	Training Cost	Validation Cost
32	3	222,433	0.890	07m 09s	0.0012	0.0014
32	5	354,529	1.418	10m 08s	0.0010	0.0011
64	3	887,233	3.549	13m 16s	0.0008	0.0008
64	5	1,413,569	5.654	19m 22s	0.0008	0.0010
128	3	3,543,937	14.176	28m 00s	0.0006	0.0007
128	5	5,645,185	22.581	42m 10s	0.0005	0.0006
256	3	14,165,761	56.663	01h 09m 48s	0.0006	0.0007
256	5	22,562,561	90.250	01h 46m 49s	0.0004	0.0005

Table 7.5: Execution statistics for context encoder model configurations with variations on kernel size and depth for our experiments.

We found that various combinations of kernel size and depth dramatically increased the size of the network, as shown by the *Parameters* and *Memory* columns in table 7.5. As these parameters increased, they also increased the time required to train the model.

Figure 7.1 shows a graphical representation of the final training and validation errors for each of our tested configurations.

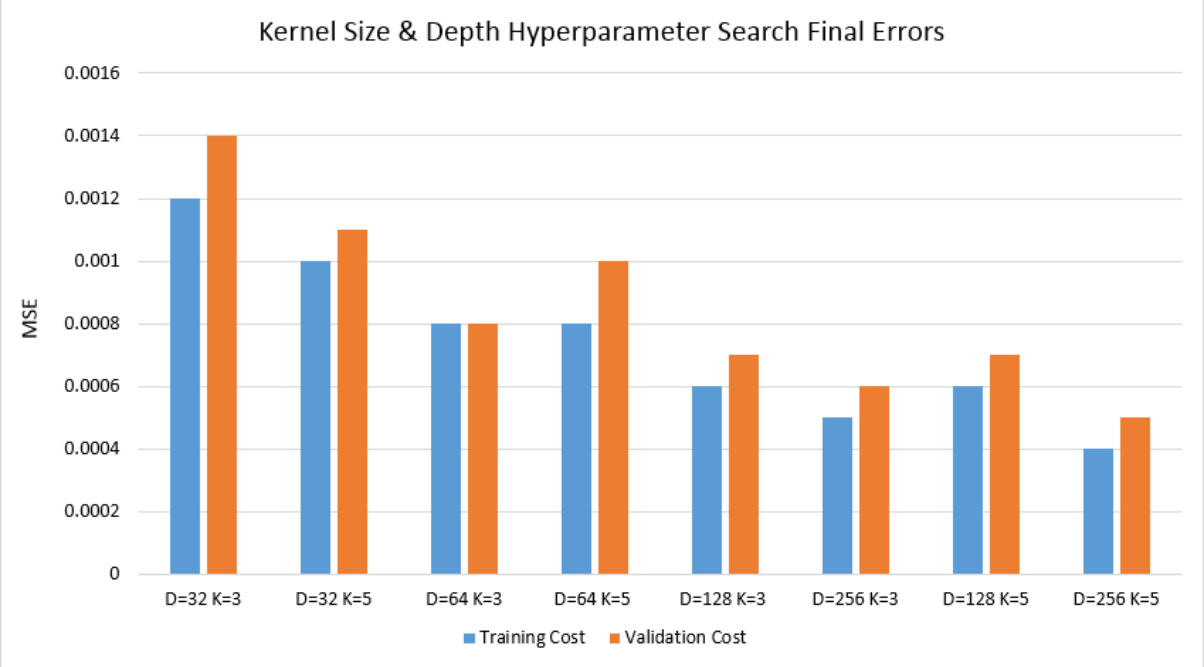


Figure 7.1: Final training and validation errors for context encoder model configurations with variations on kernel size and depth for our experiments.

From these results, we concluded that having a larger kernel size and more filters (depth) was directly related to reducing the error of the model. Our best results came from $K = 5$, $D =$

128 and $K = 5$, $D = 256$.

In addition to final errors, we also aggregated the training error of each model during the training period. These errors for each model are shown in figure 7.2.

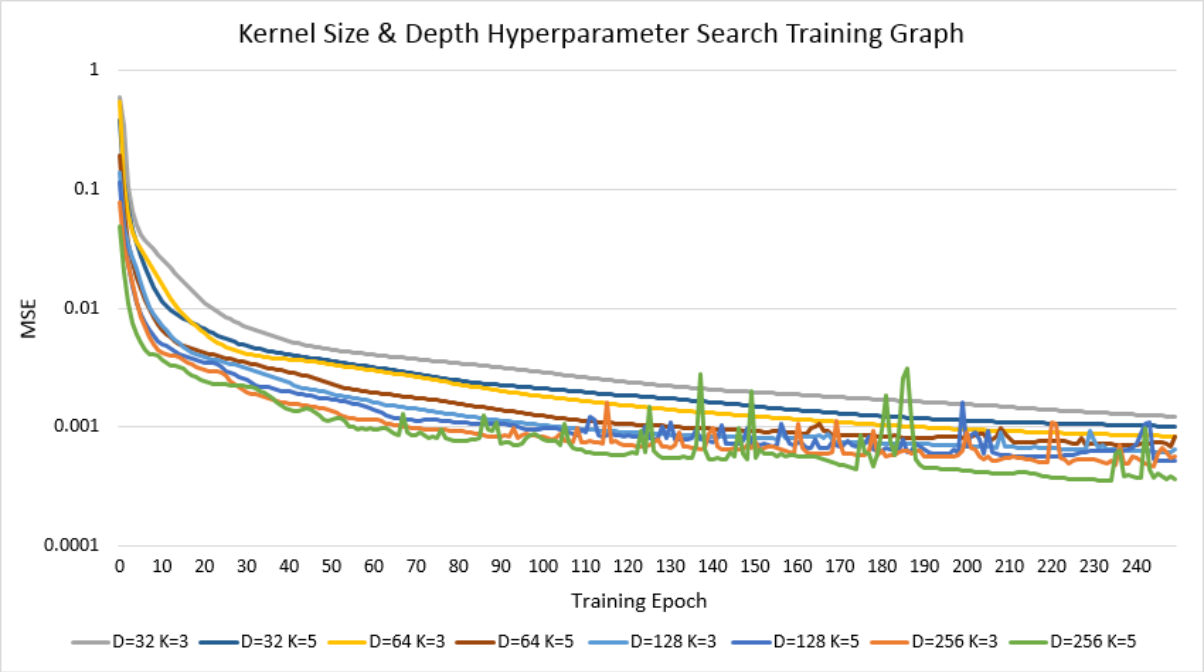


Figure 7.2: Graphs of training errors for context encoder model configurations with variations of kernel size and depth for our experiments.

In this graph, we see that each configuration approaches an asymptotic error value. As K and D increase, we see the overall errors decrease. However, one important trend is that the error also becomes unstable at higher values of K and D, specifically where $D = 256$. This becomes more clear in the later epochs, as shown by figure 7.3.

It is not clear the reason for the instability of the model, but based on this trend we concluded that the configuration that produced the best results was $K = 5$, $D = 128$, which we used for our final model configuration.

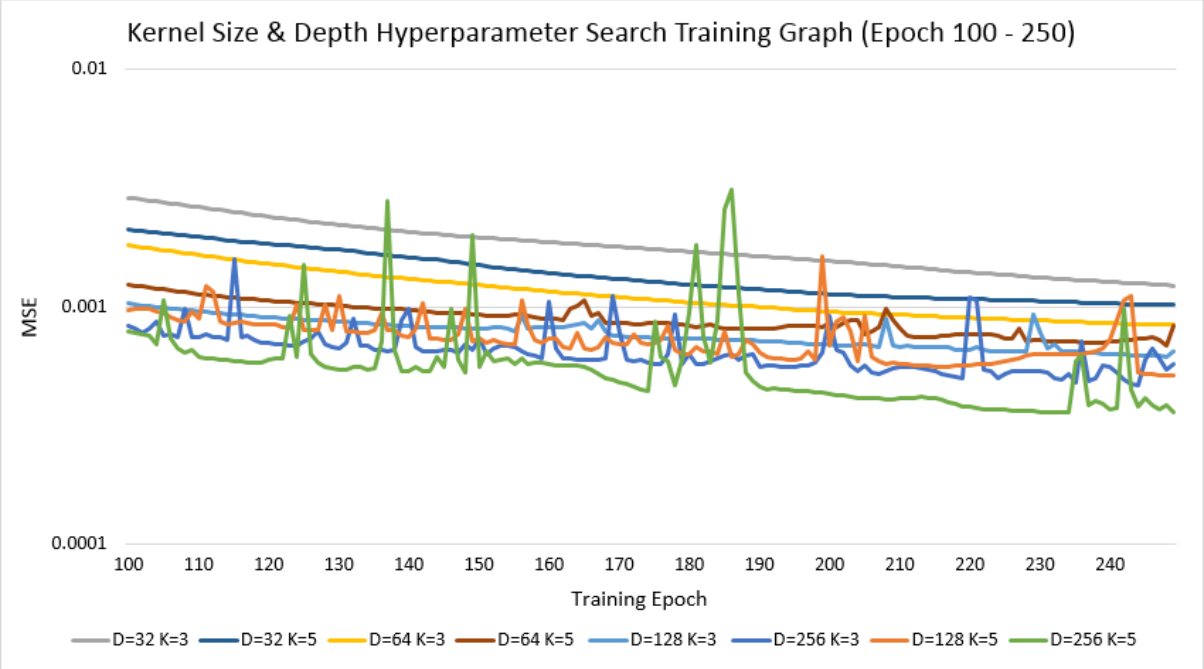


Figure 7.3: Graphs of training errors for context encoder model configurations with variations of kernel size and depth from epoch 100 - 250 for our experiments.

7.3.2 Training Set Size Search

Another experiment we performed on our model was varying the total number of samples used for training and testing the model. A list of our configurations for this experiment is shown in table 7.6. In this section, we note T_S as the number of training samples.

D	K	Training Epochs	T_S
128	5	250	100
128	5	250	500
128	5	250	1,000
128	5	250	2,922

Table 7.6: Configurations of our model with variations in training set size for our experiments.

For these configurations, we used our best model parameters for kernel size and depth, as described in section 7.3.1 for K and D. Our control parameters were the number of training epochs, K = 5, and D = 128. We attempted to evenly divide the number of training samples between training and validation sets.

We measured the memory usage, training time, final training set error (Training Cost), and final validation set error (Validation Cost) of the model of each of the configurations in table 7.6, shown in table 7.7.

Based on these results, we found that larger amounts of training samples directly correlated to the reduction of our model's error. Our best results came from T_S = 2,922, which was evenly

T_S	Time	Training Cost	Validation Cost
100	04m 16s	0.0025	0.0027
500	20m 50s	0.0011	0.0012
1,000	42m 10s	0.0005	0.0006
2,922	01h 54m 25s	0.0003	0.0003

Table 7.7: Execution statistics for context encoder model configurations with variations in training set size for our experiments.

dividing the entire set of training data we generated, described in section 4.2.2.

Figure 7.4 shows a graphical representation of the final training and validation errors for each of the tested training set sizes.

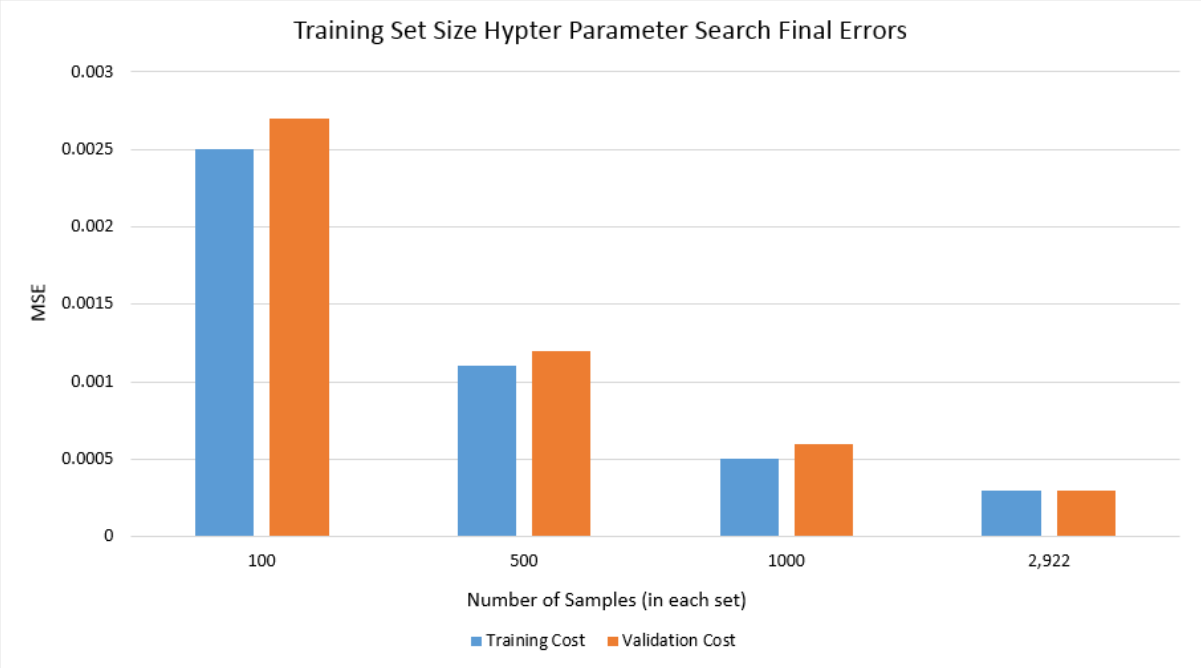


Figure 7.4: Final training and validation errors for context encoder model configurations with variations in training set size for our experiments.

From these results, it was clear that having more training samples reduced the error of our model significantly. As mentioned before, our best results came from T_S = 2,922. We also aggregated the training error of each configuration during the duration of training, which is shown in figure 7.5.

In this graph, we see again that each configuration approaches an asympotic error value. As the number of examples for training increases, the overall errors decrease. It is important to note that there is some instability in the training as the number of samples increases, but this is significantly less than instability seen in other experiments, such as those in section 7.3.1. We determined that our best results came from using all of our training data, which we used for our final model.

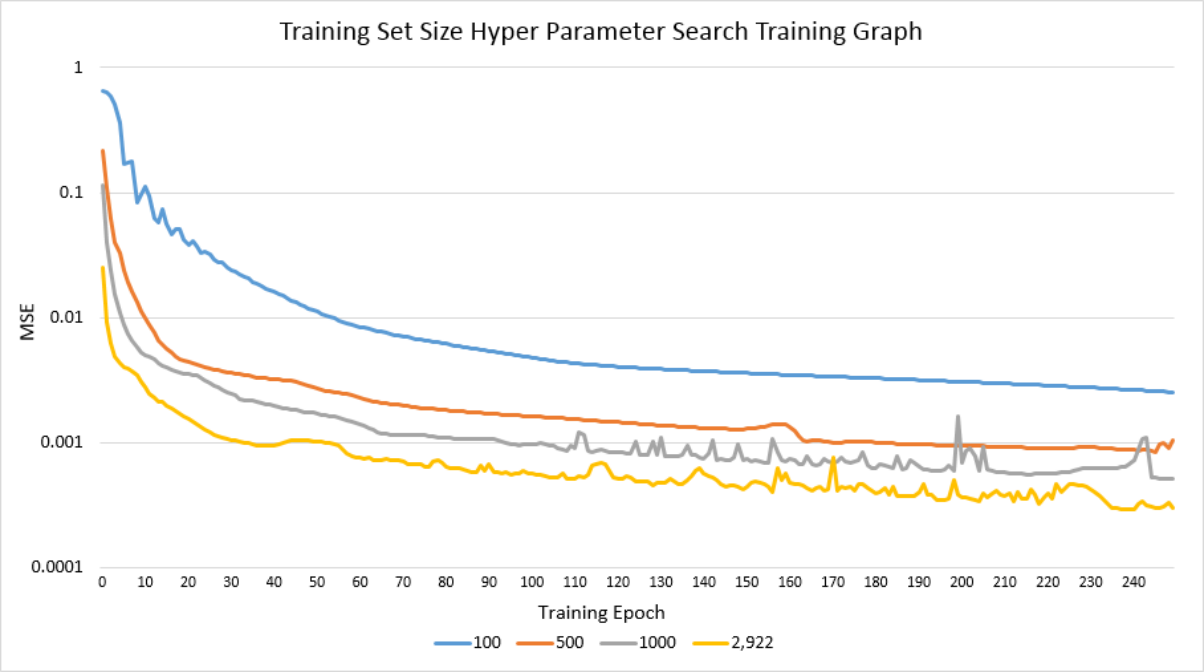


Figure 7.5: Graphs of training errors for context encoder model configurations with variations in training set size for our experiments.

8 Results

This chapter details the results of our project and the metrics and evaluations we performed to measure the accuracy and success of our implementation.

8.1 Evaluation Methods

This section describes the various methods we used to evaluate our results along the pipeline we developed.

8.1.1 Structural Similarity Index

Structural similarity index (SSIM) is a measurement used to quantify the similarity between two images. SSIM is a metric that was formed from the idea that humans view images by identifying landmarks and structures in the image. [34] Human perception to distortions in these structures is very sensitive, and so SSIM was made to measure similarity between images by measuring characteristics of these structures in images. The main components of a SSIM calculation are luminance l, contrast c, and structure s. [37] Each of these terms is defined in eq. (8.1).

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \quad c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \quad s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{8.1}$$

In these equations, μ_x and μ_y are local sample means of x and y, respectively, σ_x and σ_y are local sample standard deviations, respectively, and σ_{xy} is the local sample correlation coefficient between x and y. The constants C_1, C_2, and C_3 constants stabilize the denominator of each component when the values reach smaller values. [37] Calculating the SSIM involves a weighted combination of all three terms, as shown in eq. (8.2).

$$\mathrm{SSIM}(x,y) = [l(x,y)]^\alpha \cdot [c(x,y)]^\beta \cdot [s(x,y)]^\gamma \tag{8.2}$$

In this equation, α, β, and γ are weights that determine the relative weight of each component. For our project, $\alpha = \beta = \gamma = 1$ since we weighted each component equally. The result of this calculation produces the SSIM index, which is a value from 0 to 1, where 1 represents perfect similarity.

For performing the calculation of SSIM, we used the implementation from **`scikit-image`**.

The **scikit-image** library has a collection of algorithms used specifically for image processing, one of which is the **measure.compare_ssim** function we used.

We used SSIM as a means to measure the similarity between the ground truth CT scans and our detruncated CT scans. We determined that SSIM was a more comprehensive metric for image similarity than a metric such as mean squared error because it accounted for image components such as luminance, contrast and structure.

8.1.2 Medical Evaluation Using Polar Maps

Our primary means for evaluating detruncated CT scans involved a series of steps in a quantification pipeline used by UMass Medical School for medical evaluation of cardiac scans, which was made available for our use. We ran our detruncated scans through the pipeline and rebuilt the corrected SPECT images for the patients who had truncated scans. Part of this process involved motion correction for movements the patient makes during the scan, which is described in Feng et al. [16]. Then, using these new SPECT images, the system is able to perform a cardiac evaluation. Pretorius et al. [44] describes in detail how the cardiac evaluation is performed. The process uses *polar maps* to visualize walls of the heart before and after physically induced or chemically induced stress on the patient, which are then used for further analysis and evaluation.

The distribution of radioactive tracer throughout the heart exists in three dimensions to correspond to the heart's volume in space. Polar maps are a means to visualize this distribution in two dimensions. [15] Projecting the heart's volume in three dimensions to the two dimensional polar map involves mapping the 3D volume regions of the heart's walls to a standard stratified map with 17 regions. [7] This is a standard mapping used in cardiac assessment in radiology to gauge the health of the cardiac muscle tissue by quantifying radioactive uptake using heatmaps. [44] An example of this heatmap is shown in figure 8.1.

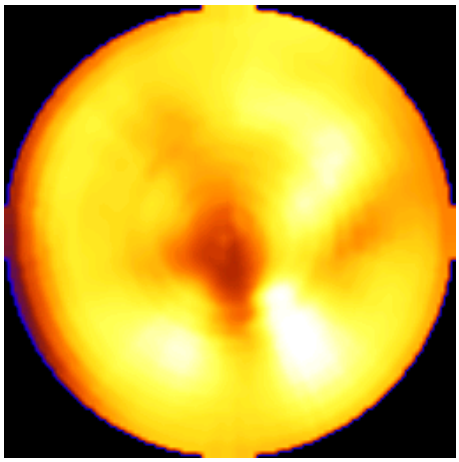


Figure 8.1: Example of radioactive uptake heatmap.

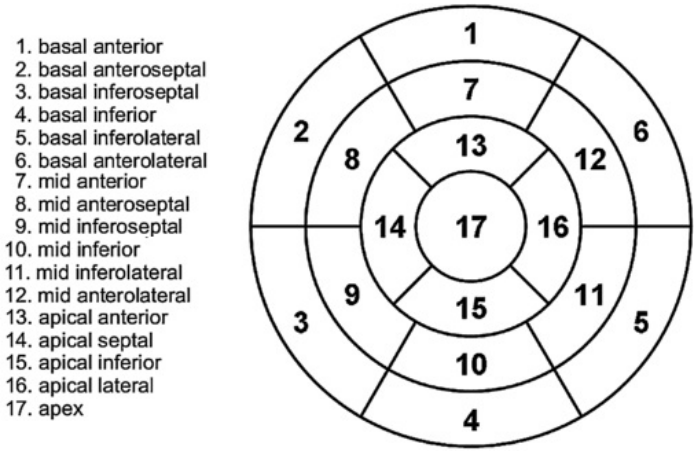


Figure 8.2: Mapping of heatmap from figure 8.1 to polar map regions. [7]

The center of the polar map is the *apex*, which is the lower peak of the heart. Moving outward from the center, the values correspond to areas closer to the base of the heart. Figure 8.2 shows a labeled diagram of a polar map, identifying the various regions of the heart.

The values displayed on the polar map are normalized with respect to the region that has the highest photon counts. These normalized values can then be compared to a database of other normalized polar maps, from which statistical analysis can be performed. [15]

Our primary goal with our detruncated CT scans was to derive attenuation maps and process them using the UMass Medical School pipeline and produce improved photon counts in the polar maps. We were able to execute the pipeline for both the original CT scan, and our detruncated CT scan. After we generated the polar maps for each, we calculated a percentage difference between the photon counts. [44] Producing higher photon counts with detruncated CT scans was the main indicator that our implementation made a positive difference on the truncated CT scans.

We focused on improving photon counts in the lateral regions of the heart, specifically regions 1, 4, 5, 6, 11, 12, and 16. These regions are all related to the lateral wall of the heart, and having higher photon counts in these regions would be more beneficial to patient cardiac studies. (P.H. Pretorius, pers. comm.)

8.2 Contour-Based Detruncation Results

This section presents the results of detruncating the contours with our context encoder model.

8.2.1 Contour Outputs

We list the parameters for the final training configuration of our model in table 8.1.

D	K	Training Epochs	T_S	V_S
128	5	583	4092	1753

Table 8.1: The final configuration we used to train our model.

These parameters were selected based on our experiments and testing, which helped us determine the parameters for our model that produced the best results. We describe these experiments in section 7.3. The number of epochs is dependent on when our early stopping mechanism determines that training should end. This is described in detail in section 6.5.1.

Figure 8.3 shows an example of an input contour to our context encoder model. As we described in section 6.4, we set the radii to zero where the contour has truncation to indicate to our model that those regions need to be corrected. Figure 8.4 shows an example of the output from our model for the truncated contour in figure 8.3.

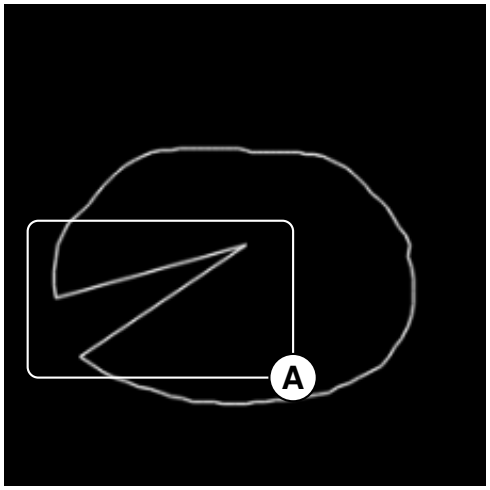


Figure 8.3: An example of a contour slice that was truncated. (A) The truncated region set to 0 radius.

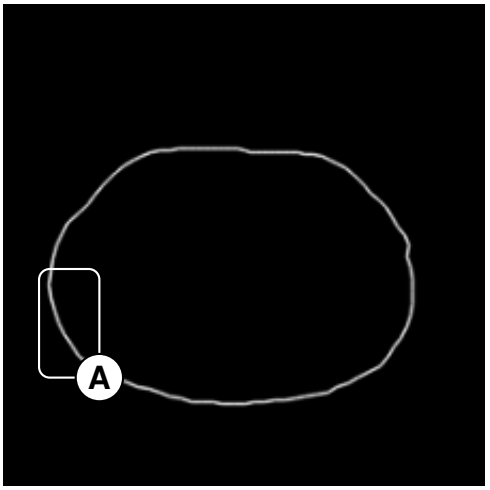


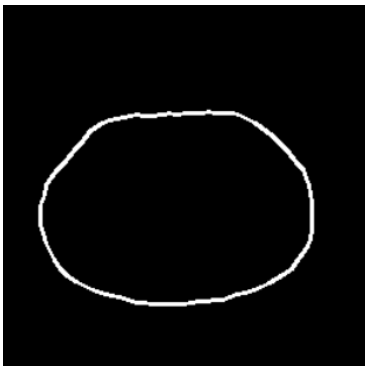
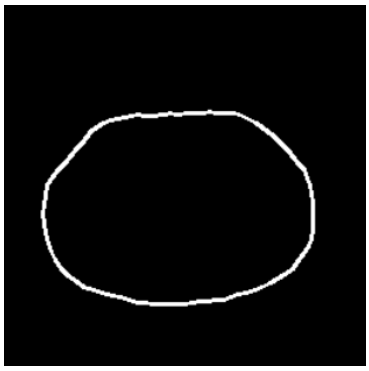
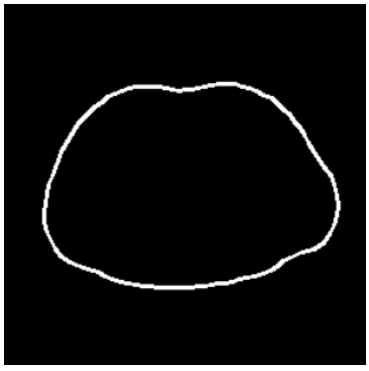
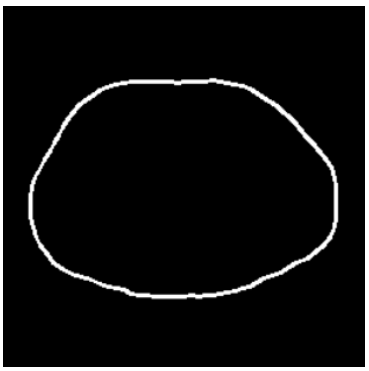
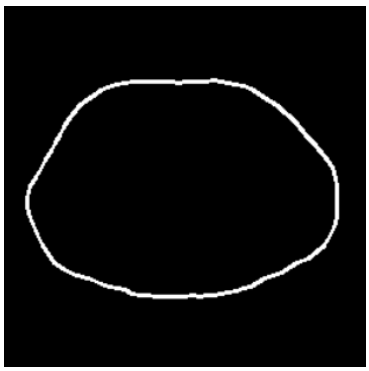
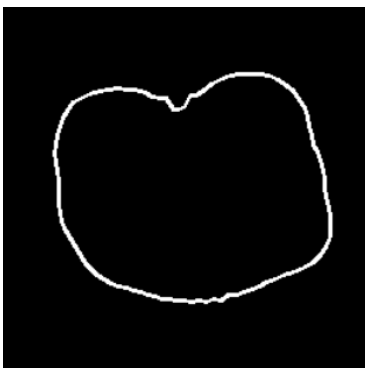
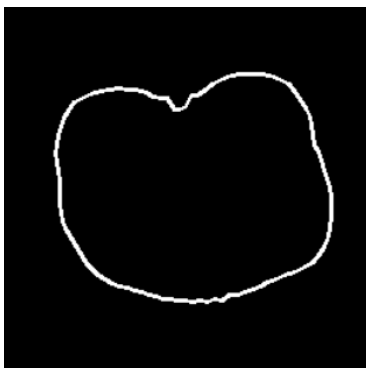
Figure 8.4: An example of a contour slice that was detruncated by our model. (A) The detruncated region.

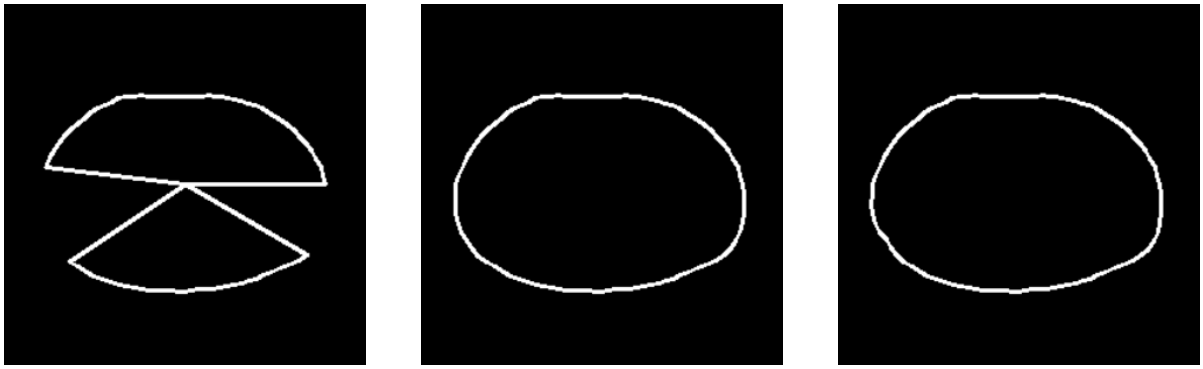
The output in figure 8.4 is the final output of the model that has been masked with the original input where no truncation occurred. Annotation (A) in figure 8.4 shows the corrected region, which accurately represents the patient's original contour.

The next sections present additional examples of outputs from our context encoder model. Section 8.2.2 has examples of detruncated contours from artificially truncated non-truncated CT scans. Section 8.2.3 has examples of 2D detruncated contours from artificially truncated non-truncated CT scans. Section 8.2.4 has examples of detruncated contours from originally truncated CT scans. Section 8.2.5 has examples of 2D detruncated contours from originally truncated CT scans.

8.2.2 Detruncated Contour Results Based On Non-Truncated Scans

This section presents single slice examples of detruncated contours based on non-truncated scans. The images presented in each row are the artificially truncated input to the model, the detruncated output, and the ground truth contour, respectively. The patients in each row are patient 10, 20, 29, 37, and 48, respectively.

Truncated Input	Detruncated Output	Ground Truth
		
		
		
		

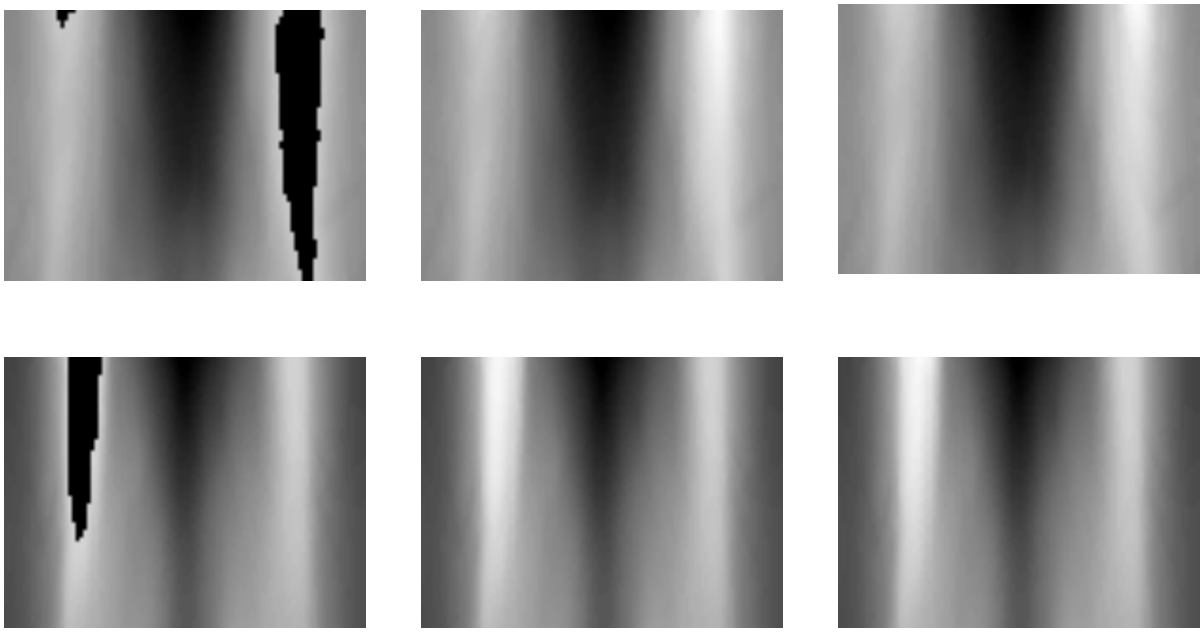


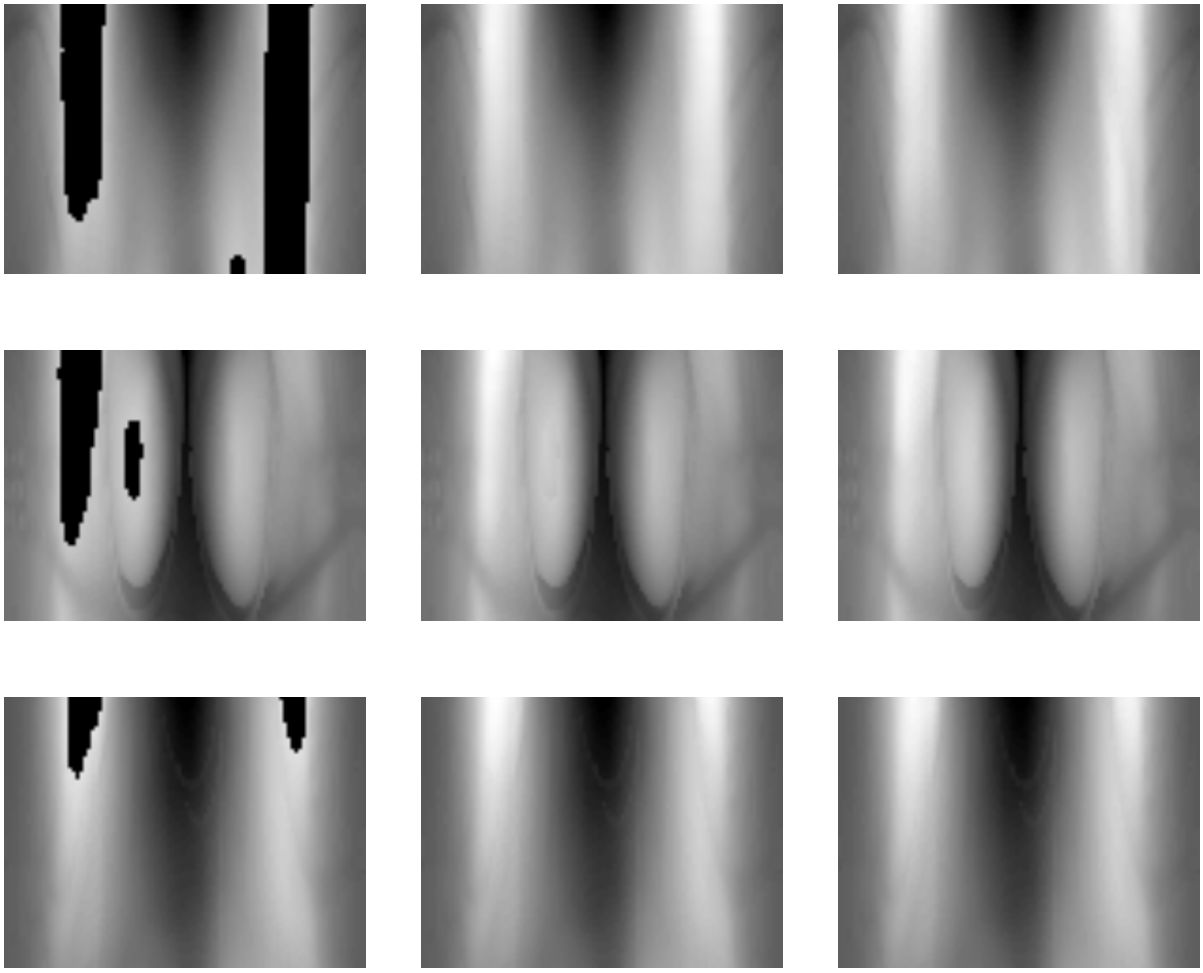
From these examples, it is very clear that we are able to fully reconstruct the contours of the CT scans, as there is very little difference between our context encoder's output and the ground truth.

8.2.3 Detruncated 2D Contour Results Based On Non-Truncated Scans

This section presents 2D contour examples of detruncated contours based on non-truncated scans. The images presented in each row are the artificially truncated input to the model, the detruncated output, and the ground truth contour, respectively. The patients in each row are patient 10, 20, 29, 37, and 48, respectively.

Truncated Input | Detruncated Output | Ground Truth





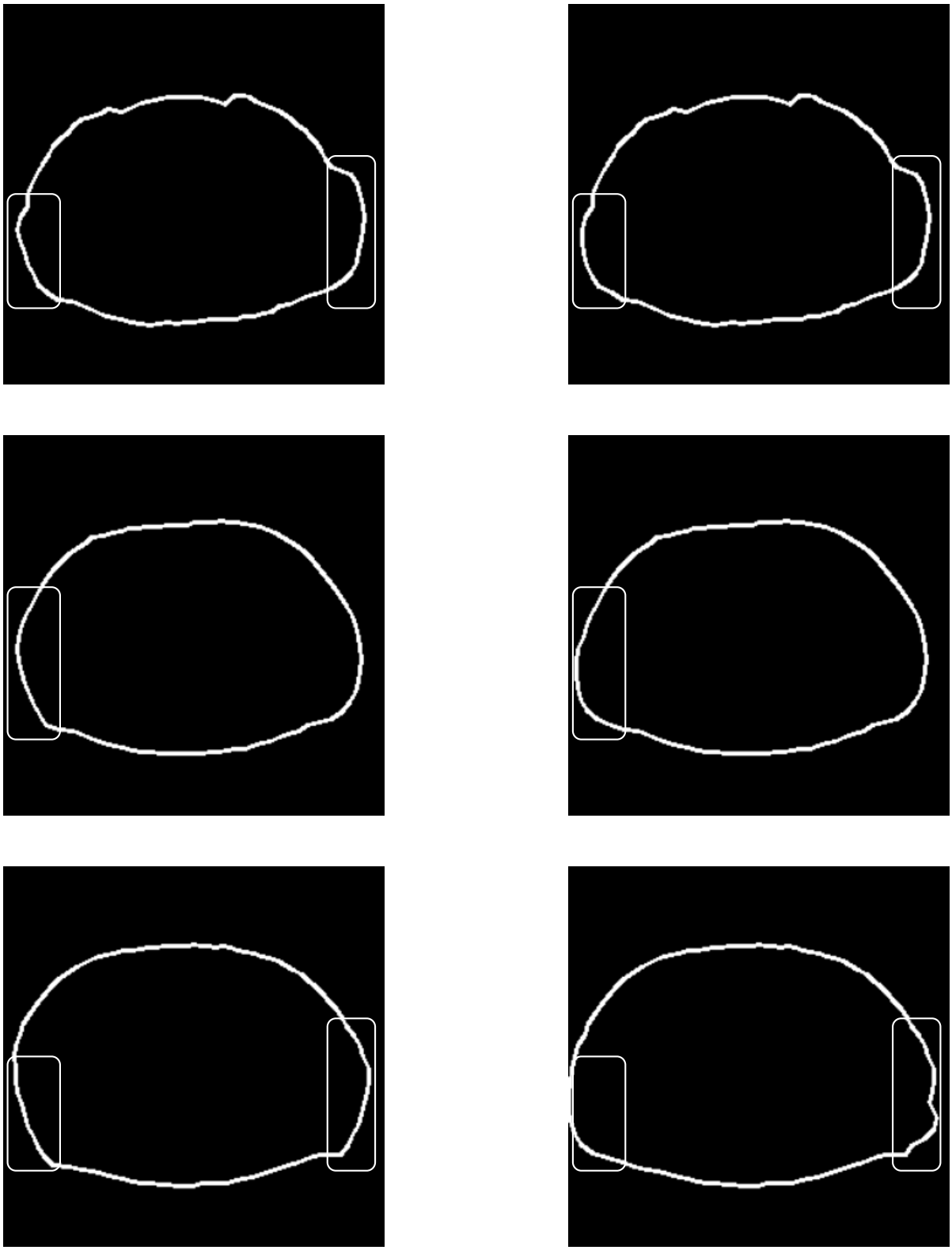
From these examples, it is very clear that we are able to fully reconstruct the contours of the CT scans, as there is very little difference between our context encoder's output and the ground truth. The reconstructed contour also maintains context between slices, which preserves anatomical features.

8.2.4 Detruncated Contour Results Based On Truncated Scans

This section presents single slice examples of detruncated contours based on originally truncated scans. The images presented in each row are the original contour and detruncated contour, respectively. The patients in each row are patient 44, 143, and 978 respectively.

Original Scan

Detruncated Output

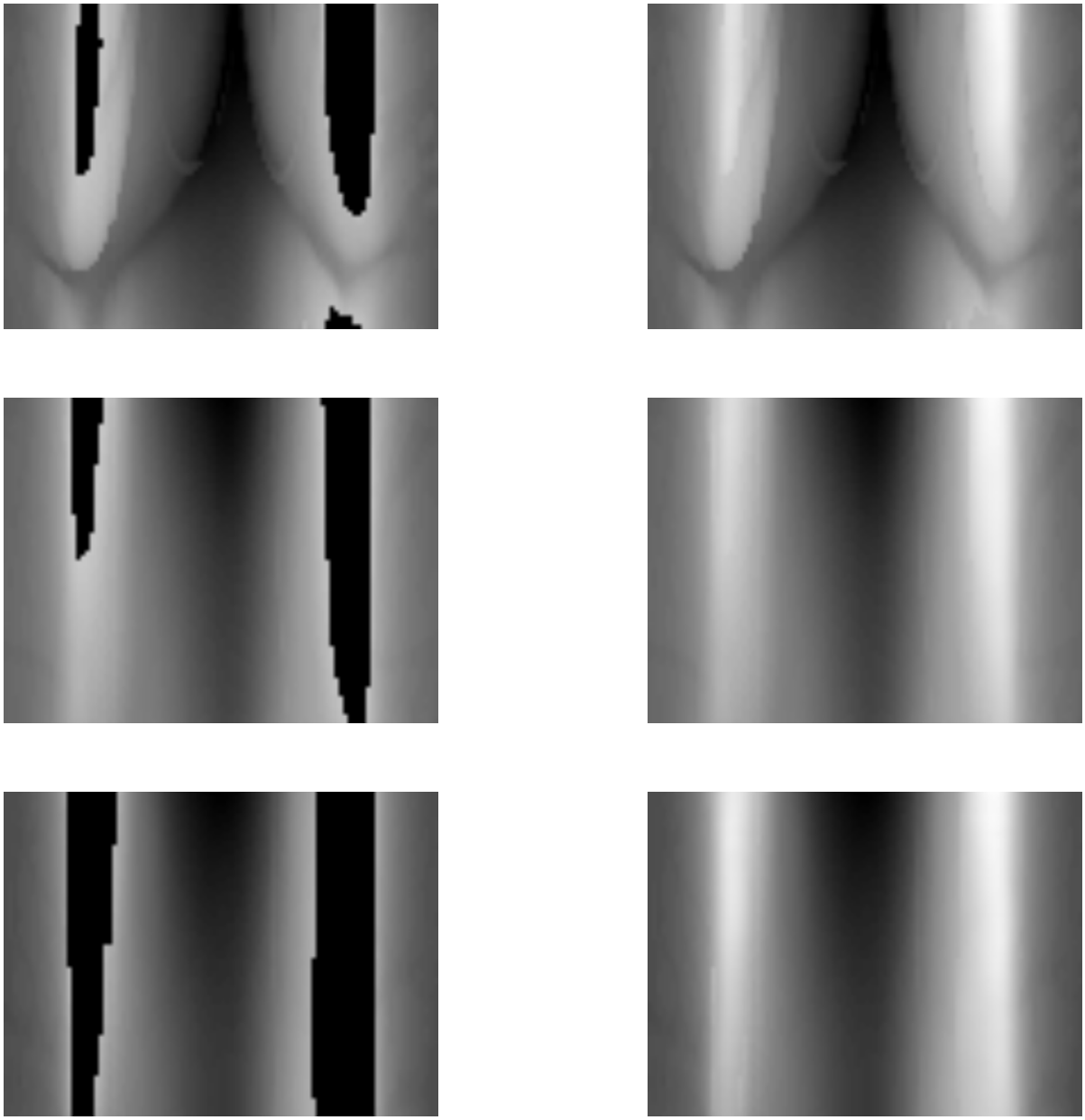


From these examples, we are able to show that we can produce reasonable estimations of the contours for these originally truncated CT scans. There is no ground truth for us to measure accuracy against, but we can see the outputs do make reasonable adjustments to the truncated regions.

8.2.5 Detruncated 2D Contour Results Based On Truncated Scans

This section presents 2D contour examples of detruncated contours based on originally truncated scans. The images presented in each row are the original contour and detruncated contour, respectively. The patients in each row are patient 44, 143, and 978 respectively.

Original Scan Detruncated Output



From these examples, we are able to show that we can produce reasonable estimations of the contours for these originally truncated CT scans. There is no ground truth for us to measure accuracy against, but we can see the outputs do make reasonable adjustments to the truncated regions. The reconstructed contour also maintains context between slices, which preserves anatomical features.

8.2.6 Training Set & Validation Set Errors

Over the course of training, we were able to measure the error of our model on the training set and validation set. Figure 8.5 shows the error for both these datasets over all the epochs.

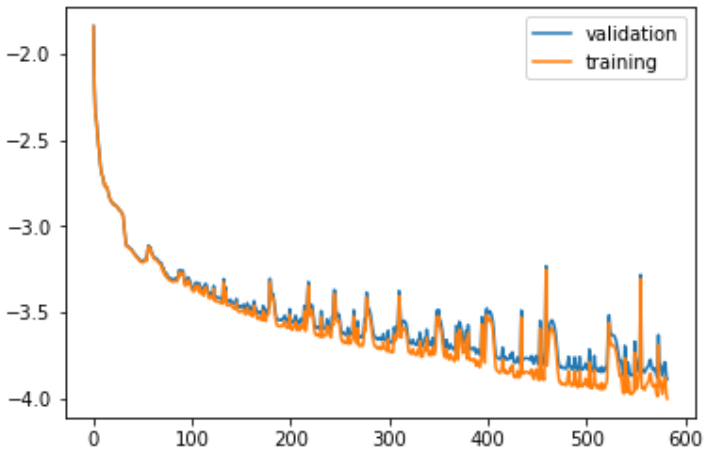


Figure 8.5: Graph of training set and validation set error over the course of training.

One of the main concerns was that our model has the tendency to overfit. We can see indications of overfitting in the later epochs, specifically where the validation error and training error begin to diverge. The diverging error shows that the model continued to lower its error on the training set, but its error on the validation set did not improve.

We were also able to plot the distributions of errors for the training set, validation set, masked training set, and masked validation set. The masked training set and masked validation set represent the final output of our context encoder model that was merged with non-truncated regions of the original input to preserve all data that was not originally missing, as described in section 6.5. These distributions are shown in figure 8.6 as *training*, *validation*. $x|y$ *training*, and $x|y$ *validation*, respectively.

We plotted the normal distributions of our error for the training and validation sets. We can see that there is more variance across the results of the validation set than the training set, which we attribute to the lower number of samples for validation than training.

Figure 8.7 shows a histogram of the Mean Absolute Percentage Error (MAPE) results for all of the contours we generated from our network.

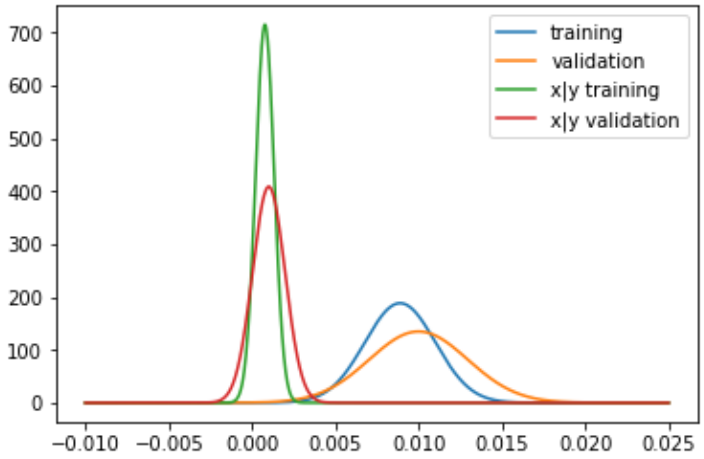


Figure 8.6: The estimated normal distributions of errors for the training

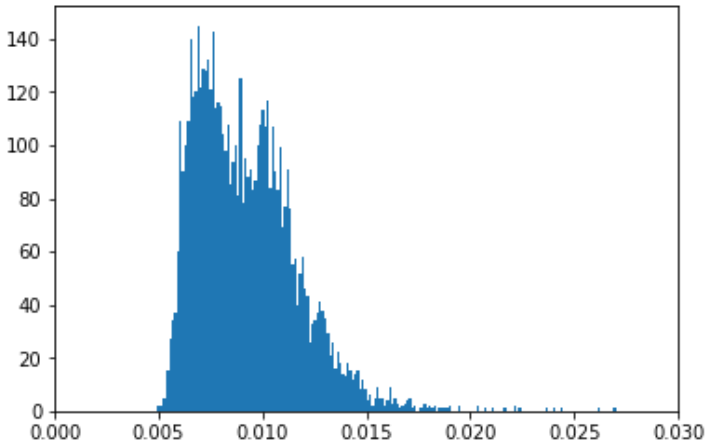


Figure 8.7: Histogram of MAPE values for validation set contours.

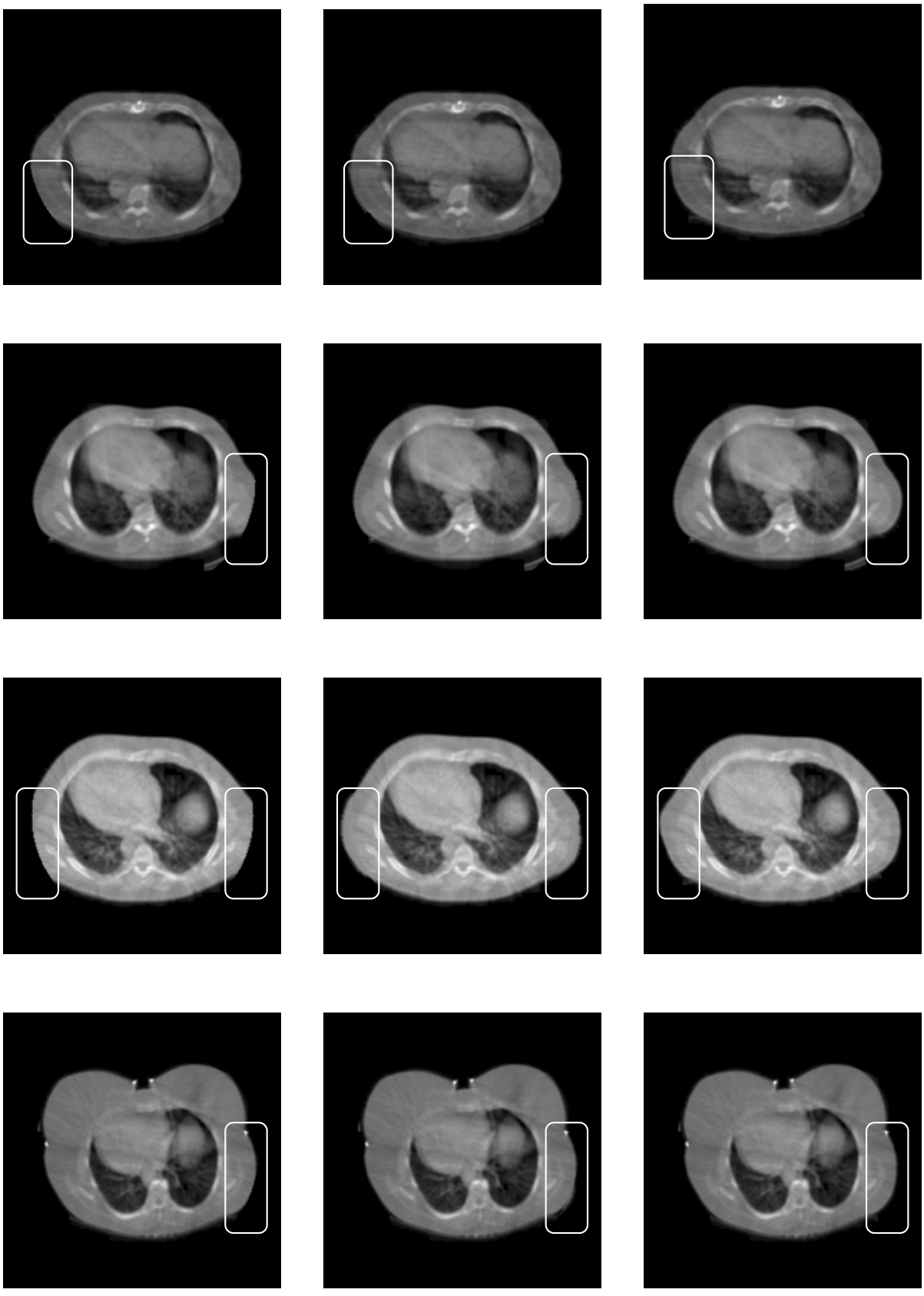
8.3 CT Scan Voxel Filling Results

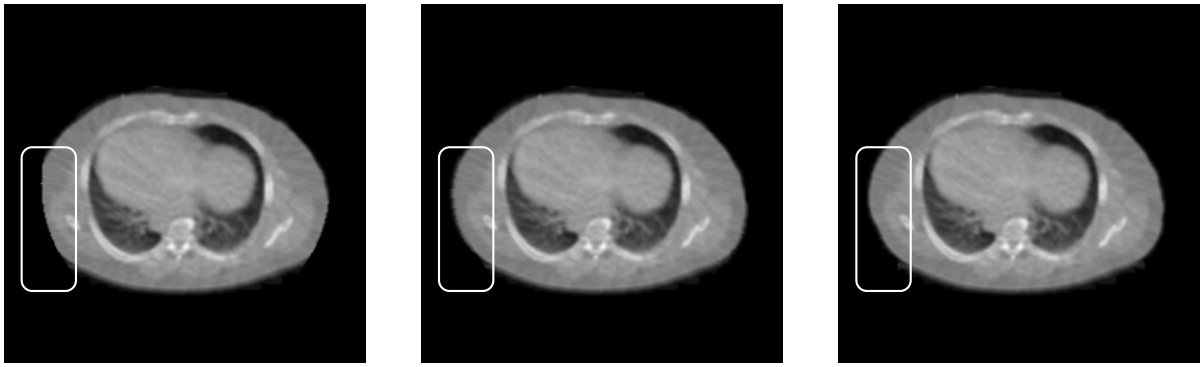
After generating the detruncated contours from our network model, we were able to apply our voxel filling methods on the contours and generate the detruncated CT scans. This section presents our final detruncated results of our CT scans. Section 8.3.1 presents the resulting CT images after our detruncation processing on artificially truncated versions of non-truncated scans, and section 8.3.2 presents our CT scan results for originally truncated scans.

8.3.1 Detruncated CT Scan Results Based On Non-Truncated Scans

This section presents single slice examples of detruncated CT scans based on non-truncated scans. The scans in this section were processed using our material profile voxel filling. The images presented in each row are the truncated input, detruncated output, and the ground truth contour, respectively. The patients in each row are patient 10, 20, 29, 37, 48 respectively.

Truncated Input | Detruncated Output | Ground Truth

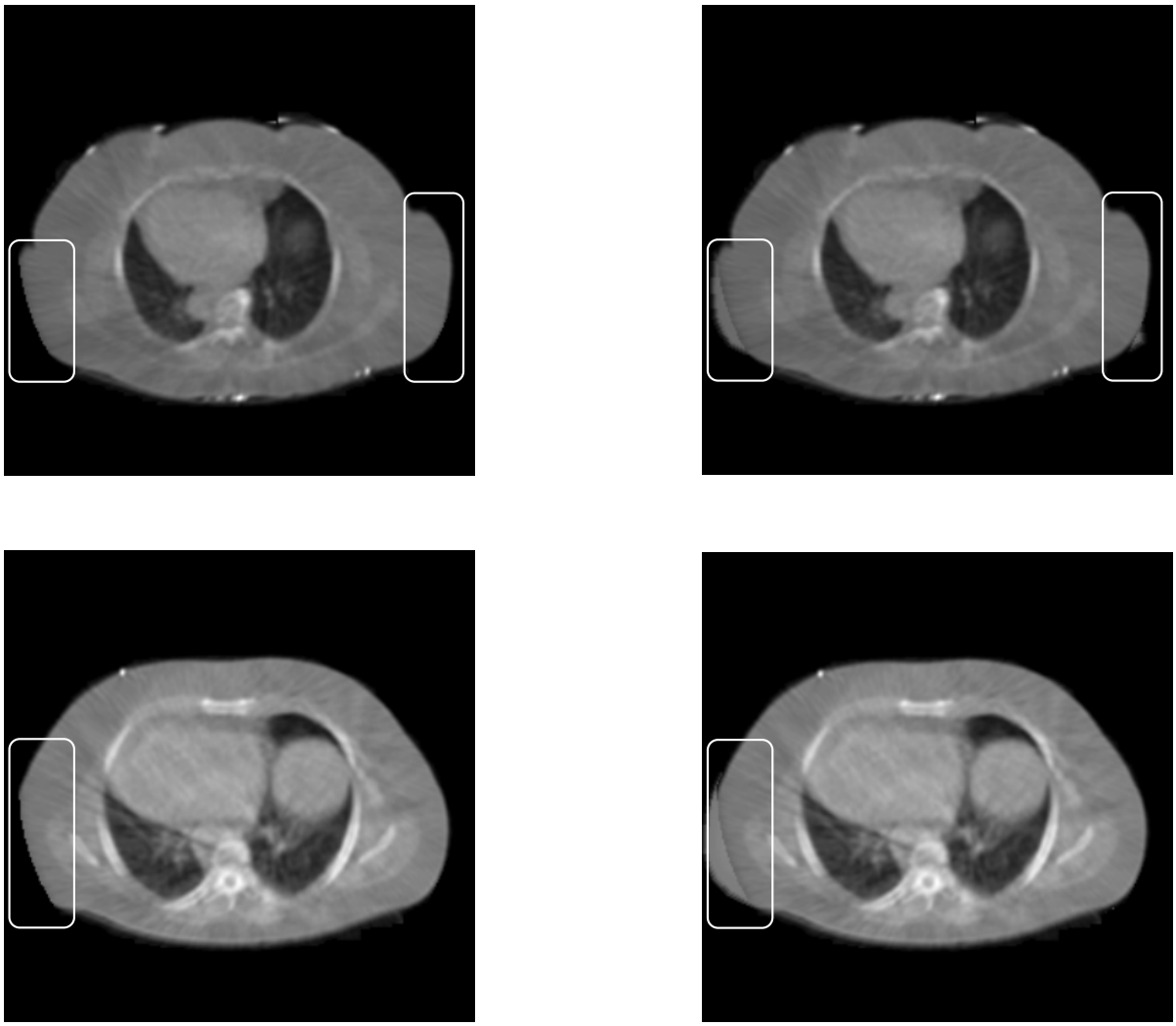


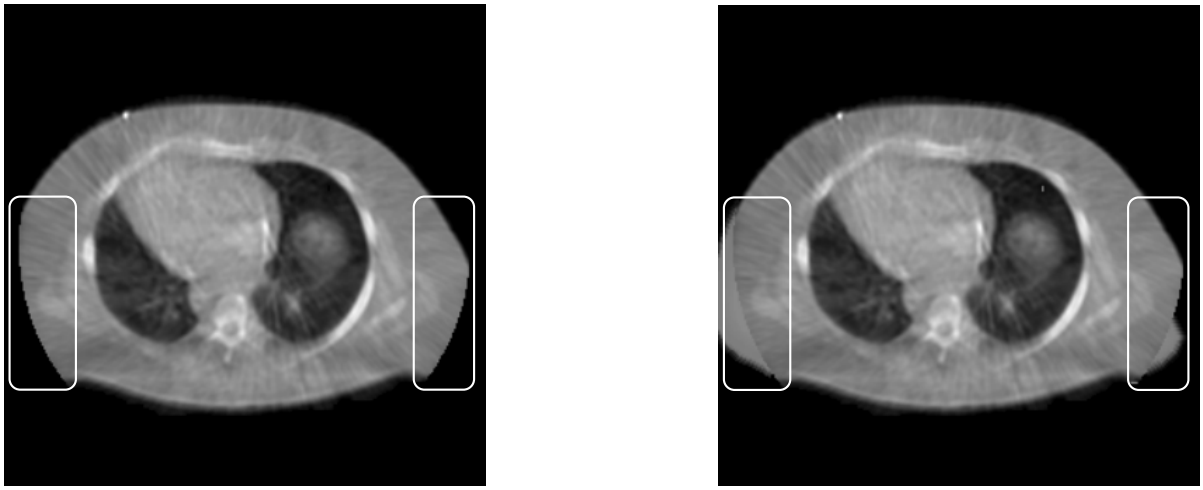


8.3.2 Detruncated CT Scan Results Based On Truncated Scans

This section presents single slice examples of detruncated CT scans based on originally truncated scans. The scans in this section were processed using our material profile voxel filling. The images presented in each row are the original input and the detruncated output, respectively. The patients in each row are patient 44, 143, and 978, respectively.

Original Scan Detruncated Output





8.4 CT Scan Image Comparison Results

After applying our material profile-based voxel filling to the detruncated contours and producing corrected CT scans, we were able to evaluate our detruncated CT scans using various image metrics to compare ground truths and detruncated outputs.

For each of the non-truncated scans, we measured the SSIM between the ground truth and the detruncated output, but only in the regions where the filling occurred. This is the *Masked SSIM* measurement in figure 8.8.

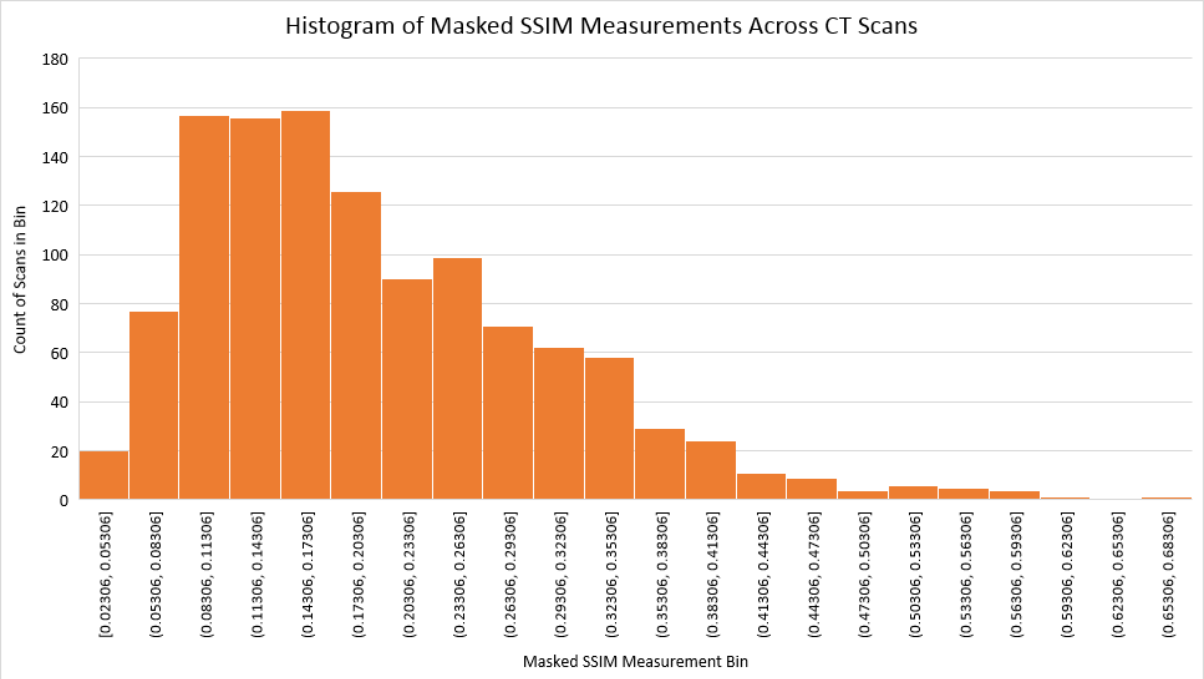


Figure 8.8: The histogram of masked SSIM measurements on detruncated CT scans.

We see from these results that the majority of SSIM values range between 0.05 and 0.2. We believe the lower values of SSIM similarity are due to the material profile's inability to recreate

distinct features in the filled regions, therefore lowering the structural component of the SSIM comparison. Additional statistics on the values in this histogram can be seen in table 8.2.

Statistic	Masked SSIM
Minimum	0.0230
Maximum	0.6631
Average	0.1995
Standard Deviation	0.1035

Table 8.2: Additional statistics on the masked SSIM measurements for detruncated CT scans.

The second measurement we performed is the percentage improvement of the SSIM between comparing the ground truth to the artificially truncated CT scan and comparing the ground truth to the detruncated scan. This is the *SSIM % Improvement* measurement in figure 8.9.

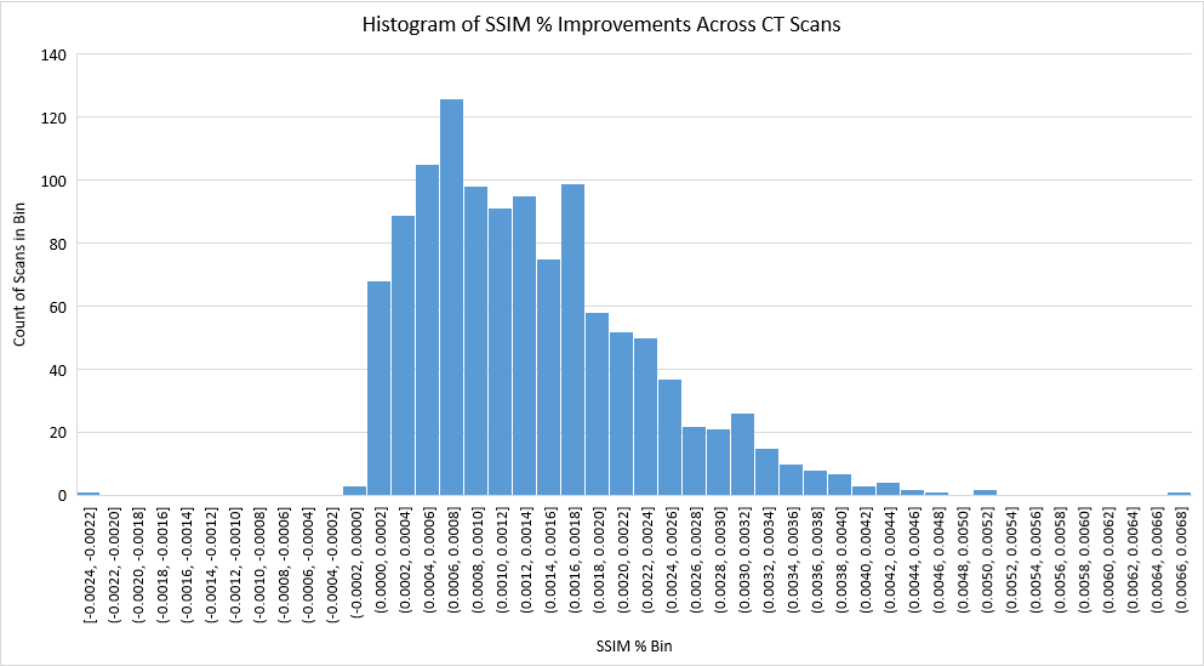


Figure 8.9: The histogram of SSIM % improvement measurements on detruncated CT scans. Values are in %.

From this histogram, we see that the majority of our measurements are positive, indicating that our detruncated CT scans were more similar to the ground truth than the artificially truncated CT scan. This signifies that our detruncation and voxel filling made improvements to the CT scans and made the scans more similar to the original ground truths. Additional statistics on the measurements in this histogram are shown in table 8.3.

Based on these results, we see that our detruncation and voxel filling made a positive improvement on the truncated CT scans. However, we note that there is room for improvement,

Statistic	SSIM % Improvement
Minimum	-0.0023%
Maximum	0.0066%
Average	0.0014%
Standard Deviation	0.0009%

Table 8.3: Additional statistics on the SSIM % improvement measurements for detruncated CT scans.

specifically on the material profile filling method.

8.5 Polar Map Medical Evaluation

As described in section 8.1.2, we used polar maps to quantify our results from the UMass Medical School pipeline. We generated polar map results for 5 non-truncated patients and 3 truncated patients. We were only able to process these 8 patients for our model through the pipeline due to the time required (20 min) per patient to generate the polar map results.

For non-truncated patients, we performed the comparison of photon counts between the ground truth CT scan and our detruncated CT scan. Our goal was to produce minimal percentage differences between the two scans, signifying that our detruncation was able to produce similar photon counts to the original non-truncated scan. These results are shown in table 8.4.

These results show overall that the percentage change in photon counts between the non-truncated ground truth and the detruncated output from our model is very small. Across these scans, the mean change was 0.23%, and the standard deviation was 1.22%. We concluded that our detruncation and voxel filling is able to produce similar photon counts across the cardiac area within a small threshold.

For truncated patients, we did not have a ground truth for comparison, and could not perform a comparison of photon counts to the ground truth in the same fashion. We were able to use the results compared to the truncated ground truth as an indicator of improvement in photon counts overall, but we are unable to relate these results to improvement in the context of medical evaluation. These results are shown in table 8.5.

Looking at the percentage changes, there is an overall trend for positive changes compared to the original scan. With truncated CT scans, this indicated an increase in photon counts in the cardiac regions. Specifically for patient 978, there is a wider range of values for percentage changes. It is not clear the cause for this wide range of values due to a lack of a ground truth and more information about the patient.

As mentioned in section 8.1.2, it is difficult to make a contextual evaluation of these truncated patients because there is no ground truth. Some of the decreases in photon counts may not be detrimental to the CT scan, but rather a better representation of the cardiac region, so it is not clear if our detruncation makes a significant improvement. We conclude that there is

		Patient				
		10	20	29	37	48
Region	1	0.46	2.66	-0.00	1.53	1.54
	2	-0.12	0.37	0.03	0.40	2.30
	3	-0.16	-3.78	-0.00	0.19	1.78
	4	-0.83	-3.91	0.00	0.18	0.94
	5	-1.35	-0.78	-0.02	-0.17	0.01
	6	-0.47	0.64	0.02	1.08	0.08
	7	0.07	-1.25	0.16	1.18	1.91
	8	-0.32	-0.99	0.08	0.32	2.51
	9	-0.06	-0.25	0.00	0.17	2.24
	10	0.70	-0.43	-0.00	0.10	1.43
	11	-0.10	-0.40	0.00	-0.11	0.66
	12	0.02	-1.59	0.17	0.42	1.21
	13	-0.26	-1.25	0.11	0.01	2.23
	14	0.02	-1.14	0.04	-0.12	3.05
	15	1.26	1.75	0.01	-0.07	2.45
	16	0.34	-1.47	0.07	-0.21	1.80
	17	0.42	-2.36	0.02	-0.21	2.91

Table 8.4: The relative changes for photon counts in detruncated outputs of non-truncated CT scans from our context encoder model processed in the SPECT pipeline, using material profile voxel filling. The units are in percentage change.

change in the photon counts based on our detruncation, but further analysis would need to be performed to show positive change, which we note as future work.

		Patient		
		44	143	978
Region	1	0.73	1.72	-3.61
	2	0.71	0.77	-25.03
	3	0.22	0.20	7.89
	4	0.03	0.41	6.06
	5	0.43	0.10	5.21
	6	0.88	0.77	4.95
	7	0.71	2.53	34.16
	8	0.48	0.88	-10.24
	9	0.28	0.11	-7.59
	10	0.19	0.82	-1.02
	11	0.51	0.65	1.44
	12	0.90	1.64	-0.45
	13	0.21	2.69	13.63
	14	0.04	0.85	17.55
	15	0.07	1.16	-10.37
	16	0.31	1.43	-20.10
	17	0.02	1.37	0.53

Table 8.5: The relative changes for photon counts in detruncated outputs of truncated CT scans from our context encoder model processed in the SPECT pipeline, using material profile voxel filling. The units are in percentage change.

8.6 Performance Testing

Our performance testing for our implementation was focused on acquiring statistics regarding run times and memory usage. We measured appropriate performance metrics for the major components of our pipeline.

8.6.1 Contour Generation

Table 8.6 shows timing results for the contour generation step of our pipeline.

Device	Scans	Total Time	Time/Scan
AMD FX-6350	1,665	03h 57m 00s	8.54s
Intel i5 4670k	1,665	02h 09m 00s	4.65s

Table 8.6: Contour generation timing

For contour generation, the system processor being used heavily affects the time to generate a single contour. We measured the time to perform contour generation over our entire dataset, and extrapolated the times for single contours for each processor in our systems.

8.6.2 Model Training

The results of our performance metrics for training our model are shown in table 8.7. The training for these measurements was performed with our configuration for optimal network results.

Training Device	Epochs	Training Time
NVIDIA GeForce GTX 1070	250	2h 29m 13s
Intel i5 4670k	250	194h 26m 40s*
NVIDIA GeForce GTX 970m	250	6h 10m 35s
Intel i7 4720HQ	250	195h 30m 14s*

Table 8.7: Model training times for each of the systems we used in our development (* represents estimated values).

For these measurements, we trained our model for 250 epochs on each system we had available in order to get statistics at a constant number of epochs. We primarily trained our model on our NVIDIA GeForce GTX 1070, which gave us our shortest training time for our model. We attempted to train the same model on our CPUs, but it was infeasible due to time constraints, and so we estimated the total training time for these configurations.

9 Discussion

This chapter details the implications of our results with regard to our original problem and goals, as well as limitations and bugs of our implementation.

9.1 Lessons Learned from Our Project

This section describes important lessons that we learned through our work on this project that should be considered and applied to future work related to this project.

9.1.1 Develop Assessment Metrics Before Implementing Solutions

A major lesson we learned from our project was that it is crucial to fully identify and develop the methods to quantify results prior to experimenting with the implementation of the solution. Throughout our project, our methods for detruncating CT scans evolved and improved, but at the same time our methods to validate our results also evolved. This resulted with us having different ways of quantifying our results for our different methods, which made it difficult to actually compare across versions of our implementations.

It would have been more efficient to fully design our evaluation methods prior to beginning our implementations and experiments, as it would provide us with a set of methods that could consistently compare the results of any implementations we develop throughout the project.

9.1.2 Visualize the Dataset in Multiple Ways

A lesson we learned and were able to apply to our project is to visualize our data in multiple ways. As described in section 6.3, we were able to create a visualization tool to view our CT scans and contour data in 3D space. Most of the methods to view the dataset that we were initially presented with focused on showing CT scans by slice. With three dimensions, we were able to analyze the data and produce meaningful results such as our truncation histograms. We see much potential and benefit to visualizing the data in our project in other forms that can show new perspectives on the data.

9.1.3 Utilize Languages & Libraries that are Accessible

A lesson we learned from our project was that it is important to consider the accessibility and ease of use of the language and libraries that are used in the final implementation. We found Python to be a language that was very accessible and easy to use. The libraries we used, such

as NumPy, SciPy, TensorFlow, and OpenCV were effective and easy to use as well. NumPy especially made working with image and vector data very intuitive. Jupyter Notebook was also an excellent tool for prototyping and debugging code.

The only other language that was a serious alternative we considered for this project was C++, because of the numerous image processing and machine learning libraries that exist for it. However, we felt that C++ was not well-suited to our needs in terms of being able to get usable code running as quickly as possible, which was a factor due to the time constraints on our project.

9.2 Implications of Our Project

The results of our detruncation pipeline show that we are able to detruncate CT scans with positive improvements. This was demonstrated by applying artificial truncation to non-truncated CT scans, and comparing our detruncated output of those scans to the original ground truths.

With the results of our system, we are able to show that the methods we use for detruncation and voxel filling are a step towards reasonably and accurately estimating the truncated regions of truncated CT scans. Our system shows initial promise that there are methods to generate the data about patients without necessarily having previous data about each patient. Our project shows that there is a way to get more complete data about patients, without having to send them to other facilities to be scanned on more versatile equipment, which may not always be available or within monetary reach.

The major caveat to this implication is that our results have not been fully evaluated in a medical context at this time. It is not clear at this time whether or not the outputs of our detruncation system actually provide additional insight into patients' conditions. We were able to show that we can detruncate scans of non-truncated patients and provide similar photon counts to their ground truths, but we are unable to verify if our reconstruction is accurate with regard to the patients' cardiac diagnoses. We describe this issue in more detail in section 10.3.5.

9.3 Limitations & Bugs

This section describes important limitations in our project that should be considered when using or extending this project in the future.

9.3.1 Scanner Bed Removal & Contour Detection

One issue that we encountered with our method of removing the bed is that OpenCV's `erode` and `dilate` methods would actually deteriorate the layer of skin and fat outside the body to the point where the result would have holes in the outer contour. This problem was

found specifically on scans that were very small and had an uncharacteristically thin skin layer. An example of this issue is shown in figure 9.1, where the bottom right of the scan has a missing section of the contour.

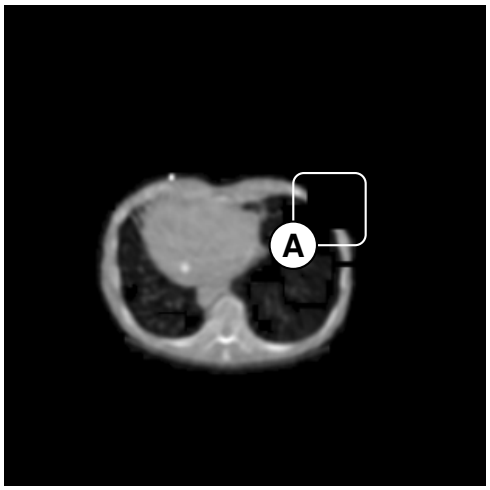


Figure 9.1: An example of the effects of removing the bed on patients with thin skin. (A) A missing part of the patient body contour.

With this method of removing the scanner bed, we see that 1.2% of our processed scans are affected by this issue in any manner, but only 0.3% are affected in a nontrivial manner.

9.3.2 Bias In Training Data

The training data used to train our model consist solely of CT scans that do not have any truncation. As we mentioned in section 5.2, there is a significant difference in the size of the non-truncated patients and the truncated patients. In most cases, the truncated patients were more obese, except for the few cases where the patient was misaligned in the scanner.

Due to the substantial difference in size and shape between non-truncated and truncated bodies, we note that there is a bias in our training data, which in turn limits the ability of our model to accurately detruncate originally truncated CT scans. The main issue stems from the fact that we do not have any ground truth on truncated bodies, and therefore are unable to train with data that is similar. Being able to train with ground truths for these larger patients would assist our models understanding of how to correct truncated patients.

9.3.3 Naïve Voxel Filling

As described in section 6.6.1, our main implementation of voxel filling to fill in detruncated contours is using our material profile method. A major issue with this method is that it does not account for anatomic structures inside the human body.

The profile attempts to use realistic values from similar regions across other patients, but the problem is that the profile is generated from non-truncated patients. As we mentioned in sec-

tion 5.2, there is a noticeable difference in size between non-truncated and truncated patients, which indicates that the tissue makeup of non-truncated and truncated patients at regions on the outer edge of the body are different as well. Truncated patients have a significantly larger fat layer due to obesity, which cannot be captured in a material profile from only non-truncated scans. Our filling methods using the material profile are limited in their accuracy due to the lack of data from truncated patients.

10 Conclusion & Future Work

This chapter presents the main achievements of this project, as well as potential avenues for future work that can expand on this project.

10.1 Our Contributions

This project attempts to detruncate attenuation maps that are used to correct SPECT images by detruncating the associated CT scans using machine learning techniques. Existing work in the field of detruncating CT scans has also attempted to solve the same problem, but the applicability of this work is limited in many aspects. Some of this work is dependent on the methods used to construct the CT scans from the raw projection data, while others rely on having additional patient scans that can be used for reference. However, these dependencies limit the cases for which these methods can be used. On the other hand, machine learning has been applied to image completion, but has been mainly focused on correcting optical images with content from a vast array of categories. This has led to the models and training datasets being extremely large, which in turn can lead to increases in the computing and time requirements for these models.

We propose a new method that expands on the existing detruncation methods by applying machine learning and image completion to a dataset with a restricted domain of only CT images. We present a system that detruncates CT scans independent of the original reconstruction methods used to build the scans, and therefore is applicable to a wider variety of different medical image modalities, such as MRI, PET, and SPECT.

This paper demonstrates the use of deep learning models such as CNNs as a means to detruncate the contours of CT scans. We also present implementations of multiple voxel filling methods to fill in the truncated regions of the CT scans that have been detruncated.

10.2 Conclusion

As stated in section 1.6, our goal for this project was to improve the accuracy of SPECT images by detruncating CT scans that were used to derive attenuation maps for the SPECT images. We planned to reach this goal by using modern image processing techniques to remove artifacts from the CT scans before using them in our machine learning model, which would process truncated scans and produce a detruncated CT scan. We were able to evaluate our detruncated results in a validation pipeline for SPECT imaging in order to get accuracy measurements in a medical context.

To reach these project goals, we used image processing libraries such as OpenCV to remove artifacts from the CT scans, such as the scanner bed and the cone beam remnants. We extracted the contour of the patient's body from each CT scan for detruncation.

We used a context encoder model built from a CNN for detruncating contours. We were able to train our model to detruncate our artificially truncated contours. We then used the model to generate corrected contours for originally truncated patients and artificially truncated patients.

We found from our own metrics, such as SSIM, and the validation pipeline at UMass Medical School, that our results from artificially truncated scans were very close to the ground truths. The validation pipeline allowed us to measure photon counts in both the ground truth and the detruncated CT scan, and we found the photon counts to be very close to each other. With these results, we were able to demonstrate that our detruncation pipeline could rebuild truncated scans. We also generated detruncated outputs for originally truncated scans, but without a ground truth we could not verify the accuracy of these scans, even though we saw positive improvements in photon counts.

In the end, our project illustrates that deep learning models such as CNNs are capable of detruncating contours of CT scans, which are then able to be used to generate detruncated CT scans. We present an implementation that is a basis for future work in solving the problem of detruncating CT scans.

10.3 Future Work

This section outlines various aspects of our project that could be modified in the future to improve the results of our implementation.

10.3.1 Correction of Additional Artifacts in CT Scans

As described in section 1.4, there are a variety of artifacts that can manifest in CT scans, which can be detrimental to the derived attenuation maps. There were a number of these artifacts that we encountered in our dataset, some of which led to the exclusion of certain CT scans due to their extensive effects on the scans and attenuation maps, as mentioned in section 4.1.1.

We note that the system we have developed can be extended by including methods to remove or reduce the effects of other CT artifacts that we did not explicitly handle. Even though we omitted 0.5% of our dataset due to these artifacts, which is a relatively small percentage, some of these scans are truncated, meaning that correcting the CT artifacts would enable our pipeline to detruncate those scans, producing a more complete result than what already exists.

By enhancing our system's ability to perform more extensive preprocessing of CT scans, a wider range of CT scans can be processed and detruncated, leading to fewer extraneous scans that are considered special cases and therefore not able to be corrected.

10.3.2 Improved Scanner Bed Removal

One of the image preprocessing steps we determined was necessary was to remove the scanner bed in the CT image. We described the methods we used to accomplish this in section 6.1.1. One of the recurring issues with our method was its effects on our contour detection. There were some cases where the detected contour would contain a hole due to part of the skin layer being deteriorated.

While the effects were prominent on a small number of scans, future work could focus on improving the scanner bed removal in order to improve the contours that we create for each patient body. With this improvement, there would be fewer CT scans that produce inaccurate detruncation outputs or cannot be processed at all.

10.3.3 Detruncation of PET Scans

Our project focused on the use of attenuation maps for correcting SPECT images, which are later used in cardiac diagnoses. We realize that SPECT imaging is not the only system used for these types of studies and analyses. There is also work to be done that can extend our system to also be used for attenuation maps that are used to correct PET images. While the general pipeline we have developed is still applicable to PET systems, there are fundamental differences between SPECT and PET images which need to be researched prior to extending our implementation for additional cardiac imaging systems.

10.3.4 Improvements to Voxel Filling Using Texture Synthesis

Section 6.6 describes the methods we implemented to fill in the voxel information for the detruncated regions of the CT scan. We realize that the methods we used provided adequate results for improving the CT scans, but methods such as texture synthesis could provide improvements to the final output.

We noted in section 3.1.2 that texture synthesis could be a supplementary method specifically for filling in voxel information. Given the similarity of voxel values across detruncated regions, it would be an appropriate method to use for voxel correction. We believe that future work using texture synthesis can provide better voxel results in comparison to the methods we have implemented.

10.3.5 Comprehensive Evaluation of Our Work In a Medical Context

While the medical evaluation using polar map quantitative statistics derived from the UMass Medical School pipeline, described in section 8.1.2, provided us with a contextual metric for our output and how it relates to cardiac diagnoses, it was not a sufficiently comprehensive metric. One of the main issues with this and other metrics we used is that we could only

compare CT scans that we detruncated based on artificial truncation that we applied to scans that originally did not have truncation. With these scans, we had a ground truth for comparison, so our results could be measured for accuracy against a truth.

On the other hand, generating results for previously truncated scans lead to incomplete and inconclusive results. Based on the polar map evaluation, we could say our results provided improved photon counts in various regions of the heart. However, the measurement fails to determine if the results actually provide insight into the patient's heart conditions, such as if our output image showed characteristics that were expected based on the patient's history, or new information due to the detruncation improving the view of the heart. These shortcomings stem from the root problem of not having ground truths for truncated scans.

Truncated scans are the inputs to our pipeline that we are attempting to correct and produce the largest improvements, but we are unable to contextually evaluate these scans without a ground truth. For future work, we believe some form of a ground truth needs to be present for these truncated CT scans. We also believe meaningful evaluations of the detruncated outputs from knowledgeable experts, such as cardiologists would be a reasonable way to get contextual measurements. This could be done through a Receiver Operating Characteristics (ROC) study on the original CT scans and the detruncated CT scans. [19] Having ground truth or a contextual evaluation would allow our system to be trained with a wider variety of CT scans, and prevent the model from being biased towards artificial truncation and smaller-bodied patients.

10.3.6 Comprehensive Comparison of Our Results to Other Work

As mentioned in chapter 3, we reviewed existing work that is related to our project. We found existing work that also attempted to detruncate CT scans such as the work by Heußer et al. [21] and Sourbelle et al. [47]. We wanted to compare our results to these methods, but a comprehensive comparison was not feasible in our project's time frame.

In addition to our own implementations, we would have had to implement the related work and compare results between them and our implementation. It would also be necessary that we adapt these other methods to the dataset in our project and the validation pipeline from UMass Medical School, which would also require additional time.

References

[1] Hojjat Ahmadzadehfar and Hans-Jürgen Biersack. *Clinical Applications of SPECT-CT*. Springer Science & Business Media, 2013. 1, 1.1.1, 1.1.2, 1.2, 1.3

[2] Aphex34 via Wikimedia Commons. Typical CNN architecture, 2015. URL `https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png`. 2.3

[3] Chuanyong Bai, Ling Shao, Angela J. Da Silva, and Zuo Zhao. A generalized model for the conversion from CT numbers to linear attenuation coefficients. *IEEE Transactions on Nuclear Science*, 50(5):1510–1515, 2003. 1.3, 1.3

[4] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009. 2.3.2

[5] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010. 4.2.4

[6] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. " O'Reilly Media, Inc.", 2008. 6.1.1, 6.1.1

[7] Woutjan Branderhorst, Frans van der Have, Brendan Vastenhouw, Max A. Viergever, and Freek J. Beekman. Murine cardiac images obtained with focusing pinhole SPECT are barely influenced by extra-cardiac activity. *Physics in medicine and biology*, 57(3):717, 2012. 8.1.2, 8.2

[8] Henrik Brink, J. Richards, and Mark Fetherolf. *Real-World Machine Learning*. Manning, 2014. 2.1

[9] Junghwan Cho, Kyewook Lee, Ellie Shin, Garry Choy, and Synho Do. How much data is needed to train a medical image deep learning system to achieve necessary high accuracy? *arXiv preprint arXiv:1511.06348*, 2015. 3.6, 3.8, 3.9, 3.6.1

[10] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012. 3.5

[11] Dan C Cireşan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*, 2011. 3.5

[12] Thomas H. Cormen. *Introduction to Algorithms*. MIT press, 2009. 6.2.1

[13] Jon deMartin. *Drawing Atelier - The Figure*. F+W Media Inc., 2013. 6.2.1

[14] Doudoulolita via Wikimedia Commons. Texture arbre, 2010. URL `https://upload.wikimedia.org/wikipedia/commons/9/90/Texture_arbre.jpg`. 3.4

[15] Ryan A. Dvorak, Richard K.J. Brown, and James R. Corbett. Interpretation of SPECT/CT myocardial perfusion images: Common artifacts and quality control techniques. *Radiographics*, 31(7):2041–2057, 2011. 8.1.2, 8.1.2

[16] Bing Feng, Howard C. Gifford, Richard D. Beach, Guido Boening, Michael A. Gennert, and Michael A. King. Use of three-dimensional gaussian interpolation in the projector/backprojector pair of iterative reconstruction for compensation of known rigid-body motion in SPECT. *IEEE Transactions on Medical Imaging*, 25(7):838–844, 2006. 8.1.2

[17] Timothy G. Freeman. *The Mathematics of Medical Imaging*. Springer Undergraduate Texts in Mathematics and Technology. Springer Science and Business Media, LLC, Springer, NY, 2010. 1.2, 1.1

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning Book*. MIT Press, 2016. `http://www.deeplearningbook.org`. 2.3, 2.2, 2.3, 2.3.1, 2.3.1, 2.3.2, 2.4, 2.3.2

[19] K Hajian-Tilaki. Receiver operating characteristic (roc) curve analysis for medical diagnostic test evaluation. *Caspian journal of internal medicine*, 4(2):627, 2013. 10.3.5

[20] M Havaei, A Davy, D Warde-Farley, A Biard, A Courville, Y Bengio, C Pal, PM Jodoin, and H Larochelle. Brain tumor segmentation with deep neural networks. *Medical image analysis*, 35:18, 2016. 3.6

[21] Thorsten Heußer, Marcus Brehm, Ludwig Ritschl, Stefan Sawall, and Marc Kachelrieß. Prior-based artifact correction (PBAC) in computed tomography. *Medical Physics*, 41(2): 021906, 2014. 1.4.3, 1.4.3, 1.5, 3.3, 3.3.1, 10.3.6

[22] Samer Hijazi, Rishi Kumar, and Chris Rowen. Using convolutional neural networks for image recognition, 2015. 2.3

[23] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. 2.3.2

[24] Jiang Hsieh. *Computed Tomography: Principles, Design, Artifacts, and Recent Advances*. Wiley, 2009. 1.2, 1.4.2

[25] Kai-Lung Hua, Che-Hao Hsu, Shintami Chusnul Hidayati, Wen-Huang Cheng, and Yu-Jen Chen. Computer-aided classification of lung nodules on computed tomography images via deep learning technique. *OncoTargets and therapy*, 8:2015–2022, 2014. 3.6

[26] ImageNet. ImageNet Home. http://image-net.org/, 2015. Accessed: 2017-01-12. 3.5

[27] Ami E. Iskandrian and Ernest V. Garcia. *Nuclear Cardiac Imaging: Principles and Applications*. Oxford University Press, 2008. 1.1.1, 1.3.1

[28] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013. 3.4.1

[29] Kenneth L. Judd. *Numerical Methods in Economics*. MIT Press, 1998. 5.3

[30] Michael Kamermans. A primer on Bézier curves, 2016. URL https://pomax.github.io/bezierinfo/. [Online; accessed 19-November-2016]. 5.3

[31] Paul E. Kinahan, Bruce H. Hasegawa, and Thomas Beyer. X-ray-based attenuation correction for positron emission tomography/computed tomography scanners. In *Seminars in Nuclear Medicine*, volume 33, pages 166–179. Elsevier, 2003. 1.3, 1.3.1

[32] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980. 6.5.1

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 3.5, 3.5.1

[34] Kieran Gerard Larkin. Structural similarity index ssimplified: Is there really a simpler concept at the heart of image quality measurement? *arXiv preprint arXiv:1503.06680*, 2015. 8.1.1

[35] Ludovic Le Meunier, Roberto Maass-Moreno, Jorge A. Carrasquillo, William Dieckmann, and Stephen L. Bacharach. PET/CT imaging: Effect of respiratory motion on apparent myocardial uptake. *Journal of Nuclear Cardiology*, 13(6):821–830, 2006. 1.1.2, 1.4.2

[36] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *NATURE*, 521, 2015. 2.3

[37] Chaofeng Li and Alan Conrad Bovik. Content-weighted video quality assessment using a three-component image model. *Journal of Electronic Imaging*, 19(1):011003–011003, 2010. 8.1.1, 8.1.1

[38] Jonathan S. Maltz, Supratik Bose, Himanshu P. Shukla, and Ali R. Bani-Hashemi. CT truncation artifact removal using water-equivalent thicknesses derived from truncated projection data. In *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 2907–2911. IEEE, 2007. 3.1.1

[39] Luigi Mansi, Vincenzo Cuccurullo, Sean Kitson, and Andrea Ciarmiello. Basic pemises to molecular imaging and radionuclide therapy. *Journal of Diagnostic Imaging and Therapy*, 1(1):137–156, 2014. 1.1, 1.1

[40] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional autoencoders for hierarchical feature extraction. In *International Conference on Artificial Neural Networks*, pages 52–59. Springer, 2011. 6.5

[41] Tom Mitchell. Machine learning. WCB, 1997. 1.5.1, 2.1, 2.1.1, 2.1, 2.3, 2.3

[42] NIST. X-ray mass attenuation coefficients, 1989. URL `http://physics.nist.gov/PhysRefData/XrayMassCoef/ComTab/water.html`. 1.3

[43] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting. *arXiv preprint arXiv:1604.07379*, 2016. 3.4, 3.5, 3.6, 3.7, 3.4.1, 4.2.3, 6.5

[44] P. Hendrik Pretorius, Karen L. Johnson, and Michael A. King. Evaluation of rigid-body motion compensation in cardiac perfusion SPECT employing polar-map quantification. *IEEE Transactions on Nuclear Science*, 63(3):1419–1425, 2016. 8.1.2, 8.1.2

[45] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. {Pearson US Imports & PHIPEs}, 2002. 2.1.1, 2.2, 2.1, 2.2

[46] SAS. Machine learning: What it is & why it matters, 2016. URL `http://www.sas.com/it_it/insights/analytics/machine-learning.html`. 2.1.1

[47] Katia Sourbelle, Marc Kachelrieß, and Willi A. Kalender. Reconstruction from truncated projections in CT using adaptive detruncation. *European Radiology*, 15(5):1008–1014, 2005. 3.2, 3.2.1, 10.3.6

[48] Waheeda Sureshbabu and Osama Mawlawi. PET/CT imaging artifacts. *Journal of Nuclear Medicine Technology*, 33(3):156–161, 2005. 1.3, 1.4, 1.4.1, 1.4.2

[49] Satoshi Suzuki and Keiichia Be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985. 6.2.1

[50] Igor V. Tetko, David J. Livingstone, and Alexander I. Luik. Neural network studies. 1. comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences*, 35(5):826–833, 1995. 3.5

[51] Isabella Thomm. Optimized appearance-spaces for texture synthesis. *ResearchGate*, 2007. 3.1.2, 3.3

[52] three.js. three.js - JavaScript 3D Library. `https://threejs.org/`, 2010. Accessed: 2016-12-07. 6.3.2

[53] Andrea Vedaldi and Karel Lenc. Matconvnet: Convolutional neural networks for Matlab. In *Proceedings of the 23rd ACM International Conference on Multimedia*, pages 689–692. ACM, 2015. 2.3.1

[54] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 479–488. ACM Press/Addison-Wesley Publishing Co., 2000. 3.1.2

[55] Miles N. Wernick and John N. Aarsvold. *Emission Tomography: The Fundamentals of PET and SPECT*. Academic Press, 2004. 1.4.3

[56] Yan Yan. *A Postprocessing Method for Scatter Compensation in Single Photon Emission Computed Tomography*. ProQuest, 2008. 1.3

[57] Habib Zaidi and Bruce Hasegawa. Determination of the attenuation map in emission tomography. *Journal of Nuclear Medicine*, 44(2):291–315, 2003. 1.3

A Code Samples

Listing A.1: Body contour generation

```
"""
This script generates body contours from the CT scans.
"""

import os.path
import math
import json
import numpy as np
import cv2 as cv

from damnnet.lib.common import *
from damnnet.lib.point import *
from damnnet.lib.load_scans import *
from damnnet.lib.ct_filter import *
from damnnet.lib.contours import *

# Debugging flag to output slice images
DEBUG = False


def get_patient_contour(scan, pts_per_slice):
    """
    Generates the body contour of the scan.

    Arguments:
        scan: The CTScan object for the data.
        config.pts_per_slice: The number of points to use along the
            circumference for the body contour.
    Returns:
        A numpy float32 array with dimensions: arr[slices][points around
            circumference].
    """
    contour_pts = np.zeros((scan.slices, pts_per_slice), dtype=np.float32)
    scan_center = Point(scan.size / 2.0, scan.size / 2.0)

    def angle(pt):
        """
        Gets the wrapped angle of a certain point.

```

```
        Arguments:
            pt: The point on the circumference.
        Returns:
            An angle in radians.
        """
        ang = pt.ang
        if ang < 0.0:
            ang += 2.0 * math.pi
        return ang

    # Run through slices
    for sl in range(scan.slices):
        # Convert contour points to polar function from center of CT scan
        sl_contour_pts = [p - scan_center for p in get_body_contour(scan,
            sl)]

        # Number of points in slice from raw contour
        num_sl_contour_pts = len(sl_contour_pts)

        # Get angle for each point in high resolution contour
        sl_contour_angles = [angle(p) for p in sl_contour_pts]

        # Convert the contour into a contour with circumference made up by
            config.pts_per_slice number of points
        for contour_pt_idx in range(pts_per_slice):
            # Find angle based on number of points
            target_angle = contour_pt_idx * 2 * math.pi / pts_per_slice

            best_valid_radius = None

            # Run through slices in high resolution contour
            for sl_contour_idx in range(num_sl_contour_pts):
                # Get current point and angle
                pt1 = sl_contour_pts[sl_contour_idx]
                ang1 = sl_contour_angles[sl_contour_idx]

                # Get next consecutive point and angle
                pt2 = sl_contour_pts[(sl_contour_idx + 1) %
                    num_sl_contour_pts]
                ang2 = sl_contour_angles[(sl_contour_idx + 1) %
                    num_sl_contour_pts]

                # If points wrapped, swap
                if ang2 < ang1:
                    pt1, pt2 = pt2, pt1
                    ang1, ang2 = ang2, ang1

```

```
81                  # Angle difference is more than 180 degrees
82                  if ang2 - ang1 >= math.pi:
83                      ang1, ang2 = ang2, ang1 + 2 * math.pi
84
85                  # Target angle is in between angles
86                  if ang1 <= target_angle <= ang2:
87                      # Find closer angle
88                      if target_angle - ang1 <= ang2 - target_angle:
89                          rad = pt1.mag
90                      else:
91                          rad = pt2.mag
92                      if best_valid_radius is None or rad > best_valid_radius
                            :
93                          best_valid_radius = rad
94
95              # Assert best_valid_radius is not None
96              if best_valid_radius is None:
97                  best_valid_radius = 0.0
98
99              contour_pts[sl, contour_pt_idx] = best_valid_radius
100     return contour_pts
101
102
103 def determine_if_truncated(px_contour, px_contour_truncated):
104     """
105     Finds the regions where truncation has occurred.
106
107     Arguments:
108         px_contour: The pixel resolution contour of the body.
109         px_contour_truncated: The pixel resolution contour marked for
                truncated points.
110     Returns:
111         True if truncated, False otherwise.
112     """
113     visited = set()
114
115     # Run through slices
116     for sl in range(px_contour.shape[0]):
117         # Traverse around slice
118         for ang in range(px_contour.shape[1]):
119             # Point is outside scan window radius and has not already been
                    traversed
120             if px_contour_truncated[sl, ang] and (sl, ang) not in visited:
121                 region = list()
122                 unvisited = list()
123                 unvisited.append((sl, ang))
124
```

```
125                 # Track range of slices and angle size of region
126                 min_sl = float('+inf')
127                 max_sl = float('-inf')
128                 min_ang = float('+inf')
129                 max_ang = float('-inf')
130
131                 # Traverse outwards from the current slice/angle point
                        until the region stops
132                 while unvisited:
133                     sl, ang = unvisited.pop()
134                     region.append((sl, ang))
135
136                     # Update size of region
137                     min_sl = min(min_sl, sl)
138                     max_sl = max(max_sl, sl)
139                     min_ang = min(min_ang, ang)
140                     max_ang = max(max_ang, ang)
141
142                     # Point now visited
143                     visited.add((sl, ang))
144                     for dsl in (-1, 0, 1):
145                         for dang in (-1, 0, 1):
146                             # Same slice and angle
147                             if dsl == 0 and dang == 0:
148                                 continue
149                             sl2 = sl + dsl
150                             ang2 = ang + dang
151
152                             # Out of range
153                             if sl2 < 0 or sl2 >= px_contour.shape[0]:
154                                 continue
155                             ang2 %= px_contour.shape[1]
156
157                             # If surrounding point also truncated but not
                                    visited yet, track it for part of this
                                    region
158                             if px_contour_truncated[sl2, ang2] and (sl2,
                                    ang2) not in visited:
159                                 unvisited.append((sl2, ang2))
160
161                 # Need to recalculate if angle wrapped around
162                 if min_ang == 0 and max_ang == px_contour.shape[1] - 1:
163                     min_ang = max(coord[1] for coord in region if coord[1]
                            < px_contour.shape[1] / 2)
164                     max_ang = min(coord[1] for coord in region if coord[1]
                            >= px_contour.shape[1] / 2)
165
```

```
166                 # Height of region in slices
167                 sl_width = max_sl - min_sl
168
169                 # Width of region in angle
170                 ang_width = max_ang - min_ang
171
172                 # Skip really small truncations that are actually noise
173                 if sl_width <= 1 or ang_width <= 1:
174                     continue
175
176                 return True
177     return False
178
179
180 def get_truncation_pts(contour):
181     """
182     Returns the points that are considered truncated.
183
184     Arguments:
185         contour: The contour to get the truncation points for.
186     Returns:
187         A boolean Numpy array of truncated points marked as True.
188     """
189     # Get contour
190     contour_pts = get_contour_pts(contour)
191
192     # Shift contour since scan window is not centered
193     shift = np.array([-1, -1], dtype=contour_pts.dtype)
194     contour_pts[:, :] += shift
195     shifted_contour = np.sqrt(np.sum(np.square(contour_pts), axis=2))
196
197     # Set points to True if outside scan window threshold
198     return shifted_contour >= SCAN_LIMIT_RADIUS - 3.0
199
200
201 def create_contour(data):
202     """
203     Creates the contour output file.
204
205     Arguments:
206         data: The raw scan data.
207     Returns:
208         A tuple of the contour, truncated points contour, and if the
                contour is truncated.
209     """
210     scan = CTScan(data)
211
```

```
212     # Remove scanner bed
213     remove_bed(scan)
214
215     # Ratio of contour points to pixels
216     num_px_contour_pts = int(np.ceil(np.pi * 2.0 * SCAN_LIMIT_RADIUS / 2.0)
            )
217     px_contour = get_patient_contour(scan, num_px_contour_pts)
218
219     # Get points that are truncated
220     px_contour_truncated = get_truncation_pts(px_contour)
221
222     is_truncated = determine_if_truncated(px_contour, px_contour_truncated)
223
224     contour = np.empty((scan.slices, config.pts_per_slice), px_contour.
            dtype)
225     contour_truncated = np.empty((scan.slices, config.pts_per_slice), np.
            bool_)
226
227     # Interpolate points on each slice to downsample from raw contour
228     for sl in range(scan.slices):
229         contour[sl] = np.interp(np.linspace(0, 1, config.pts_per_slice), np
                .linspace(0, 1, num_px_contour_pts), px_contour[sl])
230         contour_truncated[sl] = np.interp(np.linspace(0, 1, config.
                pts_per_slice), np.linspace(0, 1, num_px_contour_pts),
                px_contour_truncated[sl]) >= 0.5
231
232     # Convert truncated contour points and cleanup for noise
233     contour_truncated = contour_truncated.astype(np.uint8)
234     kernel = np.ones((3, 3), np.uint8)
235     contour_truncated = cv.dilate(contour_truncated, kernel)
236     contour_truncated = cv.erode(contour_truncated, kernel)
237     contour_truncated = contour_truncated.astype(np.bool_)
238
239     # TODO Remove debug output
240     if DEBUG:
241         import matplotlib.pyplot as plt
242         debug_sl = 55
243         plt.subplot(1, 4, 1)
244         plt.imshow(px_contour, vmin=0.0, cmap='gray', aspect=1)
245         plt.axis('off')
246         plt.subplot(1, 4, 2)
247         plt.imshow(px_contour_truncated, cmap='gray', aspect=1)
248         plt.axis('off')
249         plt.subplot(1, 4, 3)
250         plt.imshow(contour, vmin=0.0, cmap='gray', aspect=1, interpolation=
                'none')
251         plt.axis('off')
```

```
252         plt.subplot(1, 4, 4)
253         plt.imshow(contour_truncated, cmap='gray', aspect=1, interpolation=
                'none')
254         plt.axis('off')
255         plt.show()
256
257     return contour, contour_truncated, is_truncated
258
259
260 def create_contours():
261     """
262     Creates contours for the data files.
263     """
264
265     # TODO Remove debug output
266     if DEBUG:
267         create_contour(load_scan('0008_1')[0])
268         exit()
269
270     # Create contour for each CT scan
271     for data, scan_name in load_scans():
272         contour, contour_truncated, is_truncated = create_contour(data)
273
274         np.save(get_scan_data_path(config.truncated_contours_folder if
                is_truncated else config.non_truncated_contours_folder,
                scan_name), contour)
275         np.save(get_scan_data_path(config.contour_truncated_pts_folder,
                scan_name), contour_truncated)
276
277
278 if __name__ == '__main__':
279     from argparse import ArgumentParser
280
281     parser = ArgumentParser(description='Gather␣body␣contour␣data.')
282     parser.add_argument('config_file', type=file)
283     args = parser.parse_args()
284
285     # Load configuration
286     load_config(args.config_file)
287
288     create_contours()
```

Listing A.2: Spline curve-based detruncation

```
1 """
2 This script handles contour correction using splines to generate the new
      body curves.
3 """
```

```
 4 import numpy as np
 5
 6 from damnnet.lib.im_filter import *
 7 from damnnet.lib.ct_filter import *
 8
 9
10 _ANGLE_APPROX_MAX_DIST = 10
11 _ANGLE_APPROX_MAX_PTS = 10
12
13
14 def find_endpt_pairs(body_contour, size, bounds_radius=SCAN_LIMIT_RADIUS -
       2):
15     """
16     Find truncation endpoint pairs in the given body contour.
17
18     Arguments:
19         body_contour: A contour, an array of Points, usually from
               get_body_contour.
20         bounds_radius: A float, the radius outside of which points should
               be considered to be along the scan limit.
21     Returns:
22         An iterator of 2-tuples of 2-tuples of Points.
23         Each 2-tuple in the iterator is a pair of truncation endpoints in
               clockwise order.
24         Each truncation endpoint in the 2-tuple is a pair of the position
               and the direction of exit from the body.
25     """
26     assert bounds_radius > 0, 'bounds␣radius␣must␣be␣positive'
27
28     wrap_endpt = None
29     wrap_idx = None
30     in_endpt = None
31     in_idx = None
32
33     first_switch = True
34     prev_in_body = None
35
36     # TODO Docs here
37
38     def get_endpt_direction(endpt_idx, contour_direction):
39         """
40         Gets the direction of the specified endpoint.
41
42         Arguments:
43             endpt_idx: The endpoint index to check.
44             contour_direction: The direction around the contour to travel.
45         Returns:
```

```
46              A direction value.
47          """
48          direction = Point(0, 0)
49          endpt = body_contour[endpt_idx]
50          for i in range(_ANGLE_APPROX_MAX_PTS):
51              pt = body_contour[(endpt_idx + i * contour_direction) % len(
                    body_contour)]
52              direction += endpt - pt
53              if pt.distsq(endpt) > _ANGLE_APPROX_MAX_DIST *
                    _ANGLE_APPROX_MAX_DIST:
54                  break
55          direction.normalize()
56          return direction
57
58      for i in range(len(body_contour)):
59          in_body = body_contour[i].distsq(size / 2.0, size / 2.0) >=
                bounds_radius * bounds_radius
60
61          if prev_in_body is not None and in_body != prev_in_body:
62              if prev_in_body:  # from in to out
63                  if in_idx != i - 1:
64                      prev_i = (i - 1) % len(body_contour)
65                      endpt = (body_contour[prev_i], get_endpt_direction(
                            prev_i, 1))
66                      if first_switch:  # special case, need to wrap around
67                          wrap_endpt = endpt
68                          wrap_idx = i
69                      else:
70                          yield in_endpt, endpt
71                  else:
72                      # single vertex penetration, ignore
73                      pass
74                  in_idx = None
75              else:  # from out to in
76                  endpt = (body_contour[i], get_endpt_direction(i, -1))
77                  in_endpt = endpt
78                  in_idx = i
79
80              first_switch = False
81
82          prev_in_body = in_body
83
84      if wrap_endpt is not None and in_endpt is not None and in_idx != len(
            body_contour) - 1:
85          yield in_endpt, wrap_endpt
86
87  def get_splines(ct_scan):
```

```
88      """
89      Generates splines that can fill truncated regions of the given scan.
90
91      Arguments:
92          ct_scan: A CTScan object to get the splines from.
93      Returns:
94          An array of arrays of arrays of Points, where each array in the
                return value holds all the splines for that slice and each Point
                 in each spline is a control point for that spline.
95      """
96      splines = []
97
98      # Run through slices
99      for sl in range(ct_scan.slices):
100         body_contour = get_body_contour(ct_scan, sl)
101
102         slice_splines = []
103
104         for in_endpt, out_endpt in find_endpt_pairs(body_contour, ct_scan.
                size):
105             ctrl_pt_len = in_endpt[0].dist(out_endpt[0]) / 2.0
106             ctrl_pts = []
107             ctrl_pts.append(in_endpt[0])
108             ctrl_pts.append(in_endpt[0] + in_endpt[1] * ctrl_pt_len)
109             ctrl_pts.append(out_endpt[0] + out_endpt[1] * ctrl_pt_len)
110             ctrl_pts.append(out_endpt[0])
111             slice_splines.append(ctrl_pts)
112
113         splines.append(slice_splines)
114
115     return splines
116
117
118 def get_spline_regions(ct_scan, splines):
119     """
120     Generates a mask of the regions that would be filled in by the given
            splines in the given image.
121
122     Arguments:
123         ct_scan: A CTScan object to get the splines from.
124         splines: An array of arrays of Points, an array of splines for each
                 slice, see get_splines.
125     Returns:
126         A NumPy float32 (0.0 - 1.0) image masking the regions on each slice
                 where the splines should be filled in.
127     """
128
```

```
    size = ct_scan.size
    slices = ct_scan.slices
    spline_regions = np.zeros((slices, size, size), dtype=np.float32)

    for sl in range(slices):
        for spline in splines[sl]:
            fill_bezier(spline_regions[sl], 1.0, 5.0, *spline)

    spline_regions = spline_regions * (1 - ct_scan.body_mask)

    return spline_regions
```

B Context Encoder Model Example Feature Maps

This section presents examples of the feature map outputs of our context encoder model at each layer, as well as qualitative evaluation of the learned features in order to establish some understanding of how our model solves the task of detruncation. The architecture of our model is shown in figure 6.20 and described in section 6.5.

All of the images were generated by passing a single truncated contour into the model and observing the intermediate output of each convolution layer. The images are sorted by variance and displayed using the *plasma* color map the Matplotlib Python library. In these color maps, yellow and orange represent higher activations, while pink and purple represent lower values.

For reference, the original contour used for these feature maps is shown in figure B.1, and the corrected output is shown in figure B.2.

Figure B.1: This is the original 2D contour used to generate the feature map outputs in this section.

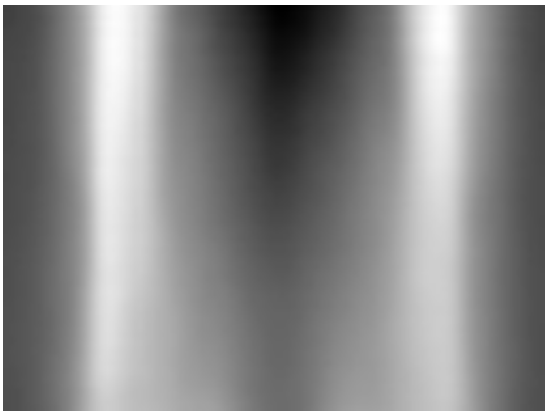


Figure B.2: This is the detruncated 2D contour used to generate the feature map outputs in this section.

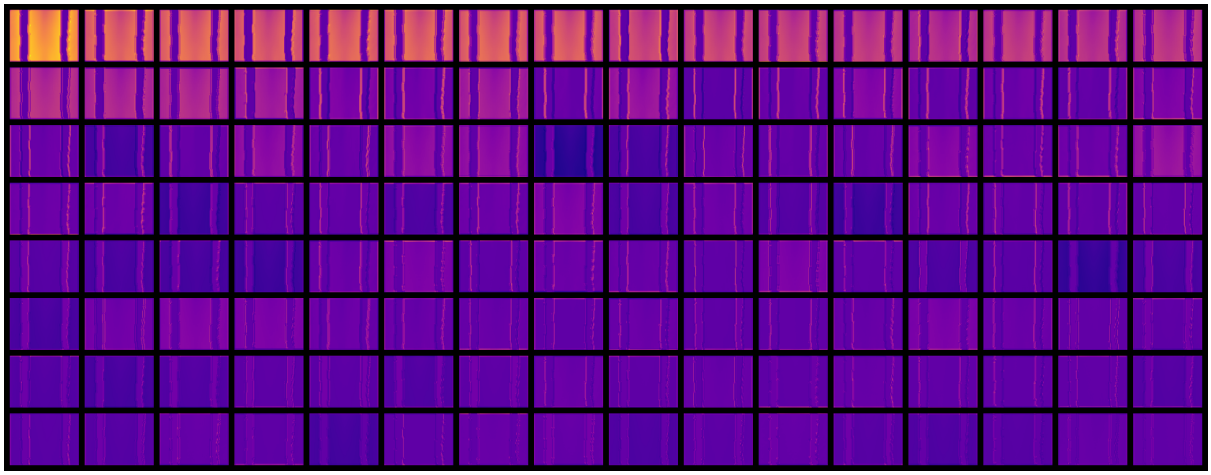


Figure B.3: Output of the encoder after convolution layer 1. The resolution of these feature maps are 72×96.

The feature maps in figure B.3, which are after convolution layer 1, appear to be encoding low-level features of the input images. The first feature map (row 1, column 1) has high values for the non-truncated regions, whereas the feature map in row 3, column 7 appears to have a higher value for truncated regions. There also seem to be some filters that are performing edge detection operations. An example of this can be seen in row 2, columns 5 and 6, which have highlighted the left and right edges of the truncation region, respectively.

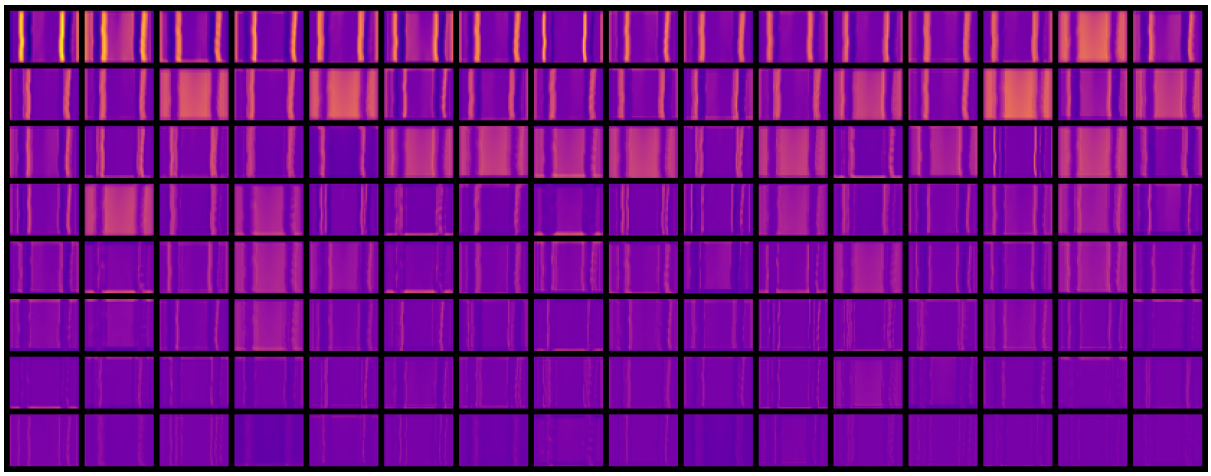


Figure B.4: Output of the encoder after convolution layer 2. The resolution of these feature maps are 36×48.

Feature maps resulting from convolution layers 2 to 4, shown in figure B.4, figure B.5, and figure B.6 have a similar structure to the input image (two vertical bars). It is possible that these layers are encoding low-level geometric features of the contours, such as edges, segmentation, and important locations in the image, such as in row 6, column 8 in figure B.4, which appears

to have higher values along the centers of where the truncated regions would be in the input image.

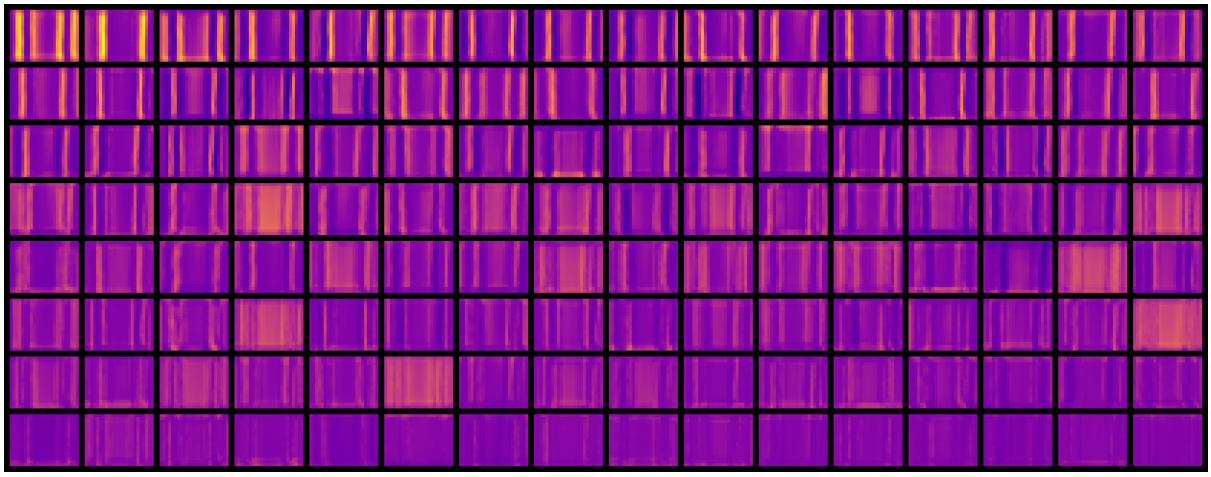


Figure B.5: Output of the encoder after convolution layer 3. The resolution of these feature maps are 18×24.

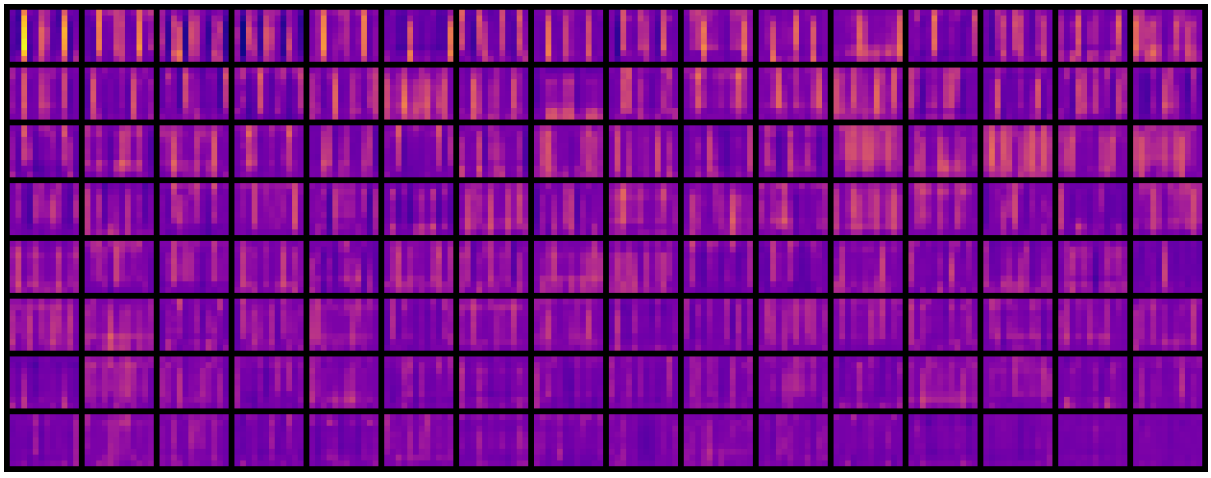


Figure B.6: Output of the encoder after convolution layer 4. The resolution of these feature maps are 9×12.

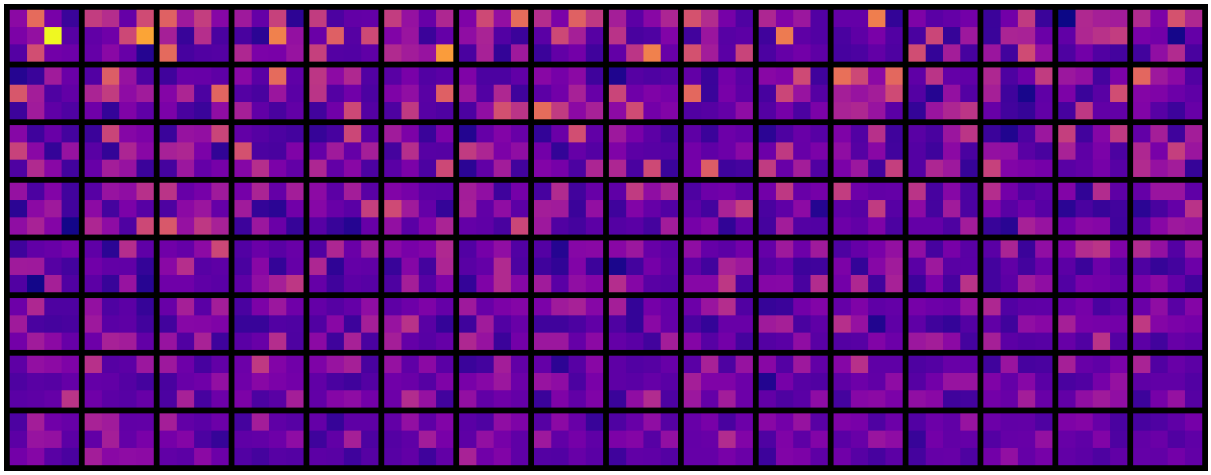


Figure B.7: Output of the fully-connected layer. This layer corresponds to the *code* layer of our autoencoder model. The resolution of these feature maps are 3×4.

As the resolution is reduced in the model, shown by figure B.5, figure B.6, and figure B.7, it is not as clear what features the activations of the fully-connected layer or the following convolution layer 5 represent (figure B.8).

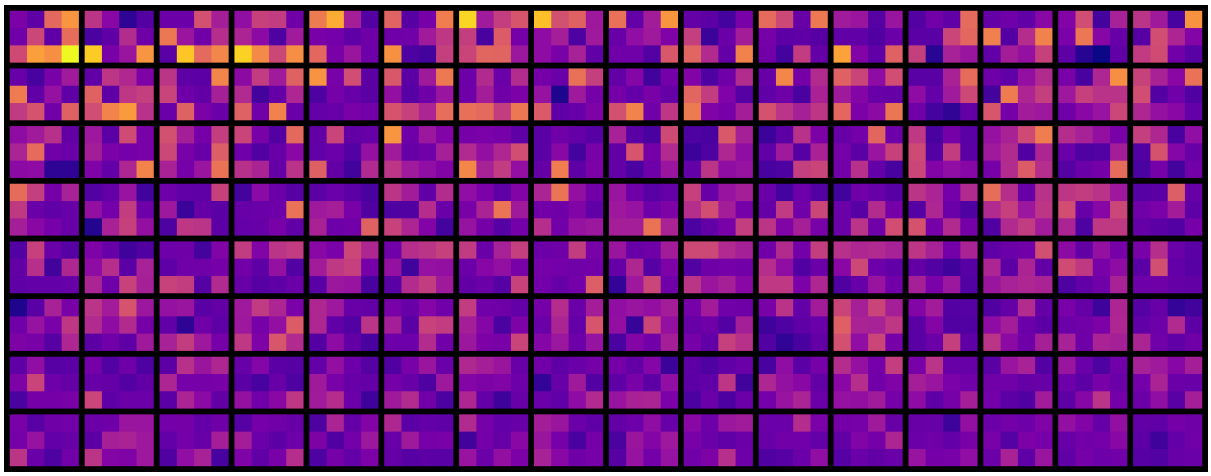


Figure B.8: Output of the decoder after convolution layer 5. The resolution of these feature maps are 9×12.

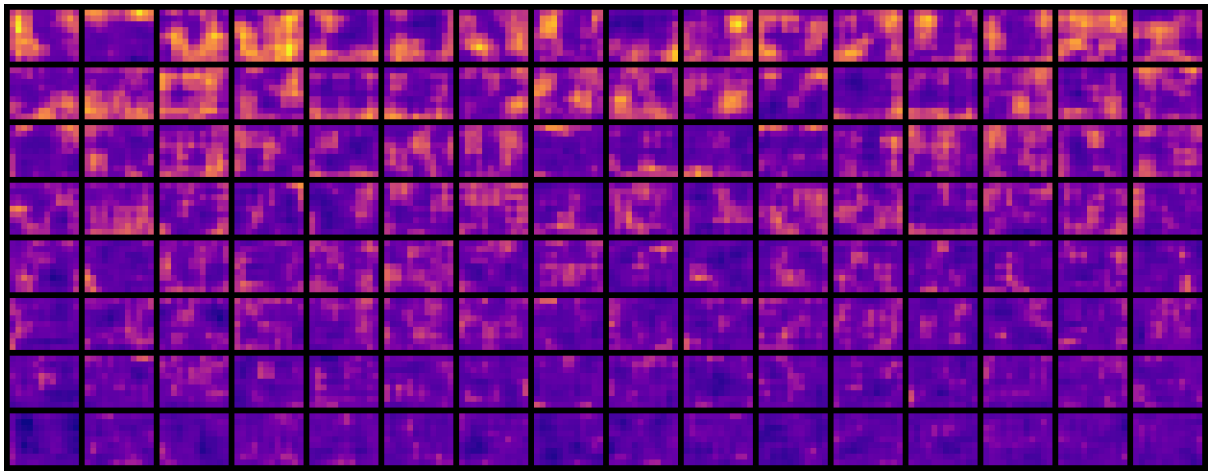


Figure B.9: Output of the decoder after convolution layer 6. The resolution of these feature maps are 18×24.

The feature maps resulting from convolution layer 6 (figure B.9) appear more structured than the previous layer. Note the blob-like features of feature maps in the first two rows. In the outputs of convolution layers 6 and 7 (figure B.9, figure B.10), we can observe faint 'checkerboard' or grid-like artifacts in many of the feature maps. This artifact is likely a result of the up-sampling and convolution process used in the decoder.

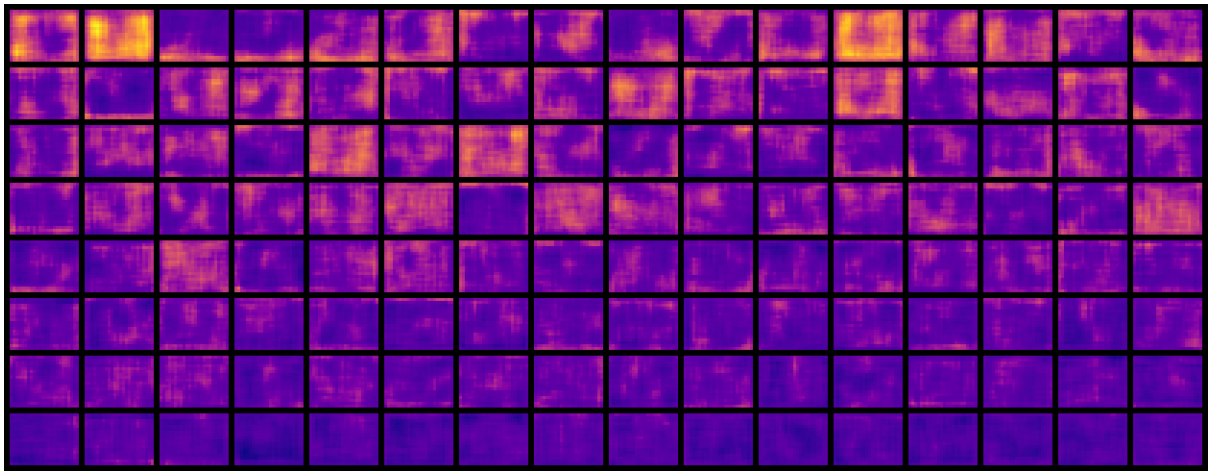


Figure B.10: Output of the decoder after convolution layer 7. The resolution of these feature maps are 36×48.

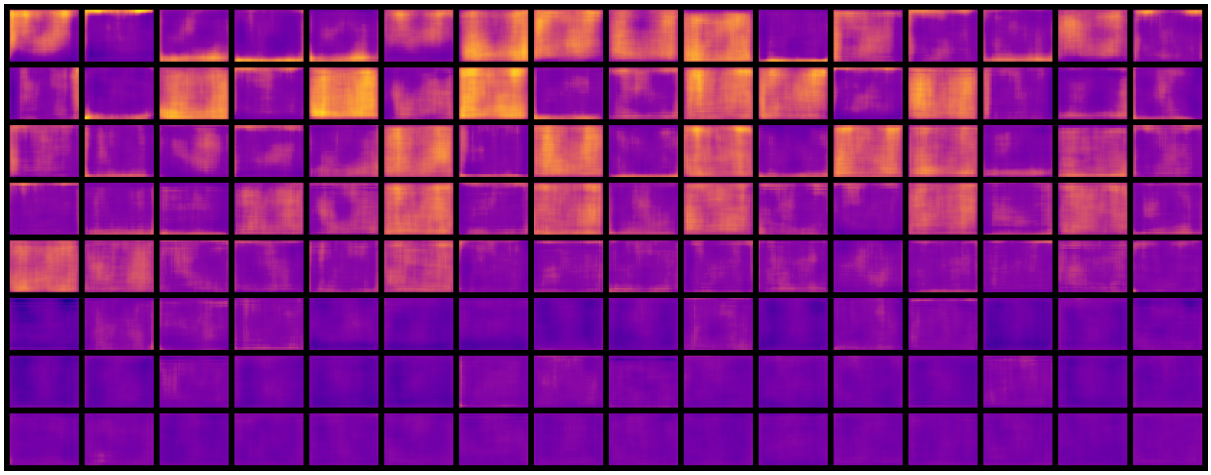


Figure B.11: Output of the decoder after convolution layer 8. The resolution of these feature maps are 72×96.

The feature maps resulting from convolution layer 7 (figure B.10) and 8 (figure B.11) appear smoother, although it is still unclear which features are being represented, as they do not resemble contours. Several feature maps from convolution layer 8 seem to be highlighting the edges of the images, such as row 1, columns 2 to 5.

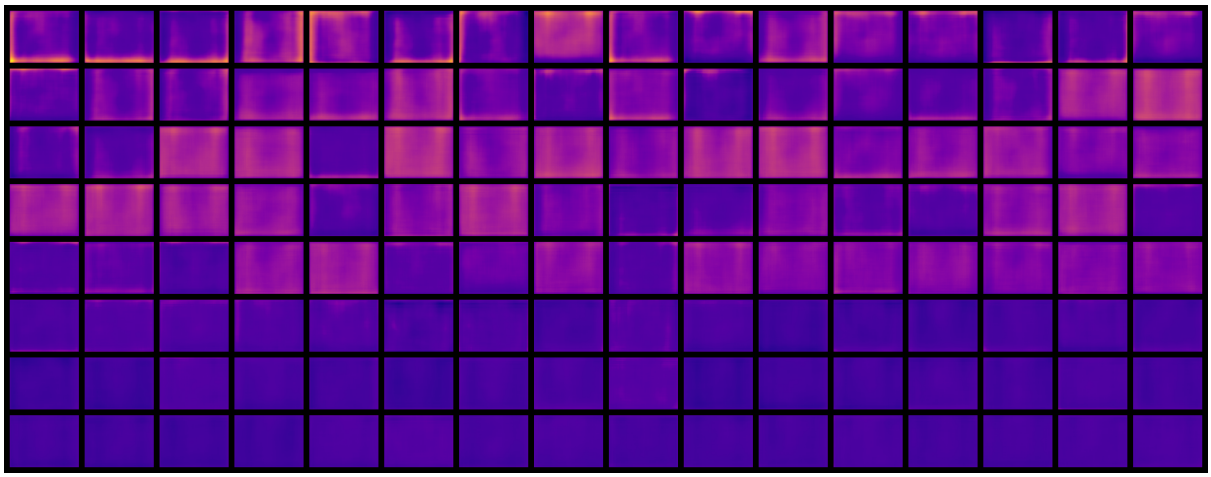


Figure B.12: Output of the decoder after convolution layer 9. The resolution of these feature maps are 72×96.

Several feature maps resulting from convolution layer 9 (figure B.12) bear a strong resemblance to a 2D contour, for example row 3, column 10. Some feature maps in the first row have higher values near the image borders. Since the final output is the result of a single convolution of these images, we would expect to find that many feature maps at this stage would exhibit the appearance of the output.