Quantifying Trust in Deep Learning Ultrasound Models by Investigating Hardware and Operator Variance

QUANTIFYING TRUST IN DEEP LEARNING ULTRASOUND MODELS BY INVESTIGATING HARDWARE AND OPERATOR VARIANCE

BY

CALVIN ZHU, B.Eng.

A THESIS

SUBMITTED TO THE SCHOOL OF BIOMEDICAL ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2021) McMaster University

(School of Biomedical Engineering) Hamilton, Ontario, Canada

TITLE:	Quantifying Trust in Deep Learning Ultrasound Models by Investigating Hardware and Operator Variance
AUTHOR:	Calvin Zhu B.Eng., (Electrical and Biomedical Engineering) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Michael Noseworthy
NUMBER OF PAGES:	xv, 118

To Bella.

My inspiration for pursuing graduate school.

Abstract

Ultrasound (US) is the most widely used medical imaging modality due to its low cost, portability, real time imaging ability and use of non-ionizing radiation. However, unlike other imaging modalities such as CT or MRI, it is a heavily operator dependent, requiring trained expertise to leverage these benefits.

Recently there has been an explosion of interest in artificial intelligence (AI) across the medical community and many are turning to the growing trend of deep learning (DL) models to assist in diagnosis. However, deep learning models do not perform as well when training data is not fully representative of the problem. Due to this difference in training and deployment, model performance suffers which can lead to misdiagnosis. This issue is known as dataset shift. Two aims to address dataset shift were proposed. The first was to quantify how US operator skill and hardware affects acquired images. The second was to use this skill quantification method to screen and match data to deep learning models to improve performance.

A BLUE phantom from CAE Healthcare (Sarasota, FL) with various mock lesions was scanned by three operators using three different US systems (Siemens S3000, Clarius L15, and Ultrasonix SonixTouch) producing 39013 images. DL models were trained on a specific set to classify the presence of a simulated tumour and tested with data from differing sets. The Xception, VGG19, and ResNet50 architectures

were used to test the effects with varying frameworks. K-Means clustering was used to separate images generated by operator and hardware into clusters. This clustering algorithm was then used to screen incoming images during deployment to best match input to an appropriate DL model which is trained specifically to classify that type of operator or hardware.

Results showed a noticeable difference when models were given data from differing datasets with the largest accuracy drop being 81.26% to 31.26%. Overall, operator differences more significantly affected DL model performance. Clustering models had much higher success separating hardware data compared to operator data. The proposed method reflects this result with a much higher accuracy across the hardware test set compared to the operator data.

Acknowledgements

Before anyone else, I want to thank Dr. Noseworthy for his support, guidance, and understanding. Without him, this thesis would never have been finished and I would have no idea where I'd be. I am, and will always be grateful for all that I've learned from you ever since I stumbled into that 4BC3 class. I hope to keep learning more in the future too. I would also like to thank Dr. Doyle for his insights, ideas, and hardware which all helped tremendously. My laptop let out a huge sigh of relief not needing to train my models for hours on end.

Unfortunately, the majority of this work was done during a strange, COVID year so I didn't quite get the chance to really work with all my wonderful lab mates as much as I would've liked. Nonetheless, I was amazed at all the support that I got, especially for my GradFlix video win. Special shout outs go to Nick for reassuring me in my GradFlix submission when I thought it wasn't good enough, Brian for giving me a friendly rivalry while being in the trenches with me all the way through it, and Lauren as well for all the supportive words and votes of confidence.

Of course, I have to give huge thanks to "my expert," Kyla. Somehow, the stars aligned so that you'd be there to help me out with scanning, to teach me the small little things, and to be a great friend that helped me keep my sanity through these strange, lock-down months. Thanks everyone!

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Statement

Ultrasound (US) is the most widely used medical imaging modality for its low cost, portability, real time image presentation and use of non-ionizing radiation. The underlying concept is also simple, send sound waves into tissue and measure reflections from tissue and reconstruct an image based on intensity and travel time. However, unlike other imaging modalities such as CT or MRI, it is a heavily operator dependent modality, requiring trained expertise to leverage any of the benefits. This leads to subjectivity in assessments and possible variability in diagnostics. This can be especially true in emergency scenarios where trained personnel may not be readily available. While errors in emergency ultrasound are due to multiple factors, many include operator or hardware related errors. These include lack of knowledge of technical equipment, inappropriate choice of ultrasound probe, inadequate image optimization, failure of perception, and overestimation of one's skill.(Pinto *et al.* (2013))

Due to the data driven nature of medical imaging, many theorize the growing field

of machine learning, specifically deep learning, may add objectivity to the subjective nature of ultrasound imaging and potentially address this operator dependency. (Liu *et al.* (2019))(Akkus *et al.* (2019))(Park (2021))

Machine learning is a broad term that encompasses a wide array of techniques. At its core, machine learning uses mathematics to learn from a large amount of representative data of an event or outcome in order to determine or predict the event or outcome using similar data. Many studies have already been done to train a Computer Aided Diagnosis (CAD) tool for ultrasound imaging. One example is Park et al., where they trained a deep learning and other machine learning based CAD tools which matched the performance of human radiologists in identifying thyroid nodules in ultrasound images. (Park *et al.* (2019))

The operator dependent nature of the imaging modality is a unique hurdle to successful implementation. This is largely due to the idea of dataset shift. Dataset shift refers to differences in training data and deployment. Medical AI typically suffers in performance due to changing subjects. Compounding this issue further, ultrasound has high variability across institutions and hardware manufacturers (Liu *et al.* (2019)) leading to uncertainty that a trained model would be fully generalize. Studies are typically done at only one institution with one ultrasound machine (Akkus *et al.* (2019)) suggesting that models may not account for incoming data from differing hardware and operators found at other institutions. The results of this on model performance is therefore not clearly defined.

In this thesis a method is proposed that aims to accomplish two goals; first to quantify US operator skill level and how specific hardware affects the acquired ultrasound images acquired. The second aim was to use this skill quantification method

to screen and match data to deep learning models to improve performance. This was done by using unsupervised machine learning to separate images generated by various operators and hardware set ups into clusters and computing the distance between these separated clusters. This clustering algorithm can then be used to screen incoming images during deployment to best match the input to an appropriate deep learning model which is training specifically to classify that type of operator or hardware. Specifically, it was hypothesized that a clustering algorithm would be able to identify differences in hardware and operators and the distances between clusters would be a metric to define them and match data to an appropriate deep learning model.

The rest of this thesis is summarized as follows;

Chapter 2 provides background knowledge on the various topics covered and highlights key ideas, current work, and potential gaps in regards to ultrasound, machine learning, deep learning, and trust quantification in machine learning.

Chapter 3 covers the experimental protocol including the data collection process, hardware, software, algorithms used, experimental set ups, performance metrics, and implementation.

Chapter 4 presents the results of the proposed methodology covering the specific performances of the various models and algorithms used.

Chapter 5 focuses on key findings, discussing model performances and potential implications of the study.

Chapter 6 concludes the thesis and summarizes findings, identifies improvements, and proposes potential future work.

Chapter 2

Background

This thesis includes elements from varying fields of expertise. These fields range from medical imaging and wave physics to machine learning, mathematical modelling, and trust. In order to bridge potential gaps in knowledge this chapter aims to give an overview of relevant topics, highlight some current applications, as well as discuss some of the core issues this thesis aims to address.

2.1 Ultrasound

2.1.1 Ultrasonic Physics

The fundamental principle behind ultrasound imaging is the transmission of ultrasonic waves. These are typically transmitted as pulses and this is known as pulsed ultrasound. Pulse repetition frequency (PRF) describes the number of pulses per unit time. (Chan and Perlas (2011)) Typically, these pulses are 1ms in length and multiple pulses are emitted per second. (Aldrich (2007)) Since these pulses travel in

straight lines, they are often referred to as beams. The direction of US propagation along the beam is called the axial direction, and the direction perpendicular to the axial direction is called lateral.

To describe these waves frequency and wavelength are commonly described. Clinical ultrasonic waves are high frequency sound waves, typically 1-20 MHz(Chan and Perlas (2011)), that are inaudible as the threshold for human hearing is about 20 kHz. The wavelength is inversely proportional to frequency, this means that high frequency waves have correspondingly low wave length and vice versa. Higher frequency waves (10-15 MHz) result in a higher number of waves of compression for a given distance and more accurately differentiates between two structures along the axial direction. Thus, high frequency waves have higher axial resolution. (Chan and Perlas (2011)) However, higher frequency waves are also prone to higher attenuation and therefore have less ability to penetrate deeper into tissue. This means that higher frequency waves are more suitable for superficial tissue imaging. In contrast, lower frequency waves (2-5 MHz) have lower attenuation and can be used to image deeper structures albeit with lower axial resolution.(Chan and Perlas (2011))

These waves are transmitted into the subject and reflections at the source are recorded. When ultrasonic waves travel through tissue, the wave can either reflect, refract, transmit to deeper tissue, or transform into heat. This is due to the fact that sound waves travel at different speed through differing media. The behaviour of the waves is determined by propagation speed. Propagation speed refers to the speed at which sound can travel through a medium and is typically considered to be 1540m/s in soft tissue. (Aldrich (2007)) This speed is determined solely by an intrinsic, physical properties of the medium known as acoustic impedance and is defined as the density

multiplied by the velocity of the ultrasound wave propagation in the medium.(Chan and Perlas (2011)) Air containing tissues such as the lungs have the lowest impedance and dense tissues such as bone have the highest impedance.

Reflections of ultrasound beams form the core of this imaging technique and these reflection beams are commonly referred to as echoes. Reflection occurs at boundaries between two materials provided the acoustic impedance is sufficiently different. Of note is that the only requirement is a difference in acoustic impedance. This means that regardless if the beams are travelling from a higher to lower impedance or a lower to higher impedance medium, an echo will still occur. If the difference between the materials is small, a weak echo is reflected and most of the energy is transmitted deeper into the tissue. If the difference is large, a strong echo is reflected and little energy is transmitted deeper. If the difference is large enough, such as an interface with air, all of the US beam is reflected and no energy is transmitted deeper. Typically in soft tissue, only a small percentage of the beam is reflected. After reconstruction, strong echoes appear as white and weaker echoes are shown as gray.

Refraction occurs when the ultrasound beam hits a surface at angle rather than at 90 degrees. In this scenario, an echo returns at an angle equal to the incident while the rest of the beam transmits deeper at an angle that deviates from the incident angle determined by Snell's Law (Equation 2.1). Figure 2.1 shows a visual example of Snell's Law.

$$n_1 sin(\theta_1) = n_2 sin(\theta_2) \tag{2.1}$$

Refraction can commonly cause confusion since the reconstruction assumes that the US beams travel in straight lines without deviation. However, this is only in

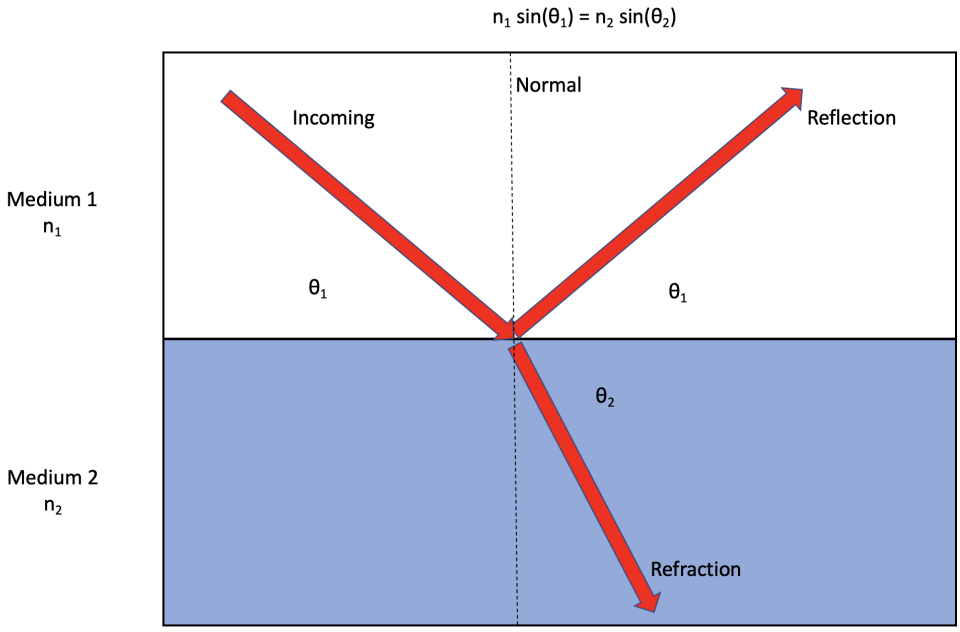

Figure 2.1: Visual representation of Snell's law. Snell's law states that an incoming ultrasound wave will deviate at a medium boundary defined by the given equation where n_1 and n_2 are the acoustic impedances of the media, θ_1 is the angle of the incoming wave and θ_2 is the angle of the refracted wave.

the idealized case and refraction is a useful property for imaging irregularly shaped objects. (Aldrich (2007)) Scattering occurs when the beam hits an uneven or rough surface and the echoes reflect in various directions. This allows for some of the echo to reach the transducer, allowing for irregular objects to be imaged.

Transmission occurs through uniform tissue or at tissue boundaries of identical acoustic impedance. The wave simply continues into deeper regions until it meets another boundary. However, the wave may lose small amounts of energy as it may transform into heat as the kinetic energy is absorbed by particles.

All of this allows for the reconstruction of an image based on the differing arrival times of the echoes. An example image created through ultrasound imaging is seen in Figure 2.2.

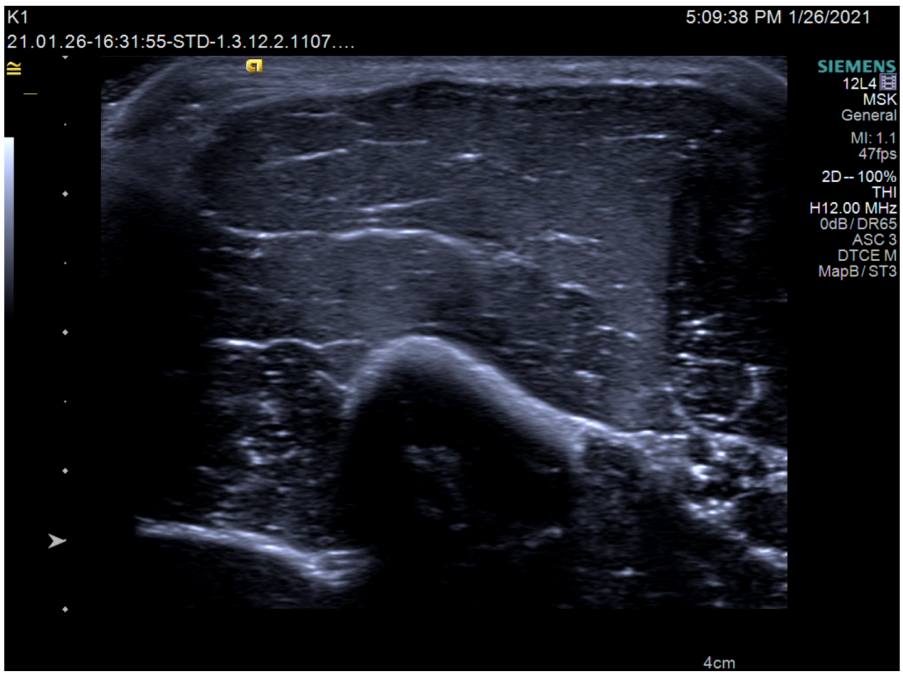

Figure 2.2: Sample ultrasound image. Image depicts biceps muscle of a healthy male volunteer scanned with the Siemens S3000 12L4 transducer at 12 MHz.

2.1.2 Hardware

Typically, there are two core hardware components in an ultrasound system. The first component is the probe which both transmits the ultrasonic waves and detects the reflecting echoes. The second component is a computing device to reconstruct an image based on the travel times and energies of the echoes.

Probes vary in type and frequency. They are made of piezoelectric elements which vibrate when an electric field is present and conversely also create an electric field when it vibrates. Probes come in various array arrangements and produce variable frequency waves for differing imaging requirements. Figure 2.3 shows examples of probes.

Most ultrasound systems are pulse echo systems. This means that the device

Figure 2.3: Ultrasound Probes. From left to right, Siemens S3000 12L4, Clarius L15, SonixTouch L14-5. These particular examples are all linear phased array that produce 2D images.

detects three things, echo strength, echo direction, and time of arrival of echos from the tissue boundaries.

Due to the many sources of attenuation, relative intensity levels are more important to measure than absolute intensities and differing signal amplification is commonly applied during reconstruction.(Aldrich (2007))

2.1.3 Image Interpretation and Artifacts

There are four types of ultrasound scans commonly used for diagnosis, A-Mode, B-Mode, M-Mode, and Doppler.(Carovac *et al.* (2011)) In A-Mode, a transducer scans a line and plots echoes as a function of depth. In B-Mode, a linear array is used to image a plane through the body which is reconstructed as a 2D image. M-mode refers to motion and is used to capture a range of motion. Doppler ultrasound takes advantage of the Doppler effect to image blood vessels. Most ultrasound imaging is done in real time B-mode, also referred to as brightness mode. During the reconstruction

of an ultrasound image, brighter regions are referred to as hyperechoic and darker regions are referred to as hypoechoic. This corresponds to weaker and stronger echos respectively. Isoechoic regions have echoes equivalent to neighbouring tissue and anechoic regions appear as black regions without echoes.

Artifacts are image results that are not true to the object being imaged thus leading to some form of inaccurate representation in image space. Artifacts often arise due to assumptions made during collection and reconstruction. Some of the key assumptions are that sound waves travel strictly in a straight line, that reflections occur from structures along the central axis of the beam, that the intensity of the reflection corresponds to the reflector scattering strength, that the speed of sound in tissue is exactly 1540 m/s, and that the sound will travel directly to the reflector and back. (Aldrich (2007)) In practice these assumptions are made false. Artifacts are helpful sometimes and detrimental other times. It is up to the operator to discern when they occur and how to interpret them (i.e. differentiate them from truth). It is in part why the imaging modality is so operator dependent as it is difficult to determine what the image is properly depicting without context.

2.2 Machine Learning

2.2.1 Overview

Datasets are typically divided into training, validation, and testing sets. Training sets are used to first train an algorithm and constitute the largest portion of the data. Validation sets are used to adjust training and search for hyperparameters, adjustable values used to tune models, and test sets are used to determine how well

the algorithm performs on similar, but unseen data. Both the validation sets and test sets are typically similar in size.

Using this collection of data, algorithms are developed to model the data. This modelling is typically done in a supervised, unsupervised, or reinforcement based approach.

Supervised learning is one of the most common forms of machine learning and involves the use of labelled data. In supervised learning problems, an algorithm is given data and outputs a vector of scores. The goal is to have the algorithm output scores that align with the proper labels of the given data. This is usually done with an objective function, also known as a cost function, that measures the error between the outputted scores and the desired scores. Given examples, the algorithm modifies its internal weights in order to reduce the error and thus minimize the objective function.

Unsupervised learning is growing in interest and involves the use of unlabelled data. In unsupervised learning problems, an algorithm is given data without labels. Unlike supervised learning problems, there is no given result that the algorithm tries to match. Instead, the algorithm tries to identify patterns, structure, or trends in the data commonly referred to as clusters.

Although not used in this thesis, reinforcement learning is briefly described for sake of completion. Reinforcement learning is the least common form of machine learning, but is also seeing growing interest. It involves training "agents" to navigate a state space by learning "policies" and typically requires specific learning environments rather than raw data samples. Typically agents learn by trial-and-error where they interact with the environment, receive feedback, and update their policies on how to handle certain situations in the future. One of the most famous examples of

reinforcement learning in recent times is the program AlphaGo by Silver et al., the first computer program to beat a world champion at the game of Go. (Silver *et al.* (2017)) AlphaGo plays games against itself over and over, iterating through games in order to improve itself . Remarkably, AlphaGo achieved its superhuman performance without human guidance outside of the rules of the game, highlighting the potential of reinforcement learning for areas with limited or unreliable sources of data.

2.2.2 Conventional Training Methods

Conventional training methods typically involve mathematical modelling techniques with a statistical basis. In these methods, heavy domain expertise is usually required for feature engineering (LeCun *et al.* (2015)) as raw data can take on many forms in differing context. For example, voltage readings from sensors may require different expertise than pixel values from images. The specialized knowledge needed to transform differing formats of data in differing contexts into features suitable for use in internal representations or feature vectors leads to a very narrow scope and places limits on the ability to work with certain inputs. However, because the input features are hand crafted, results are typically very easily explained.

K-Means Clustering

K-Means clustering is a conventional training method for unsupervised learning. It was first proposed by Stuart Lloyd at Bell labs in 1957(Géron (2019)) and aims to separate data into clusters based on minimizing distances to those clusters. The algorithm works as follows:

1. Centroids are placed randomly within an m-dimensional data space

2. The distance from each instance to each centroid is calculated

3. Centroids are moved

4. Repeat until convergence

To avoid convergence to local minima, this process is typically repeated n times to produce the best result. A typical n value would be 10 times and the best result is decided with a value known as inertia. Inertia is calculated as the mean squared distance between each instance and its closest centroid.

2.2.3 Neural Networks and Deep Learning

Representation learning is a set of methods that allow a machine to be fed raw data and to automatically discover the trends and representations required to model the data. Deep learning is a form of representation learning with multiple levels. Each level learns more abstract representations of the previous level. (LeCun *et al.* (2015)) Through this method, deep learning aims to discover structure in large datasets typically using a back-propagation algorithm to indicate how internal parameters called weights should be adjusted to best compute the representations in each layer. Each layer is composed of processing units which form a network.

These processing units are commonly referred to as artificial neurons and derive their name from biological neurons. Likewise, combinations of them are named neural networks. Biological neurons produce small electrical signals, called action potentials that trigger the release of chemical signals, called neurotransmitters in other neurons. Similarly, artificial neurons process inputs, and if the inputs reach a designated threshold, they produce an output to other artificial neurons.

A neural network is referred to as "deep" when it has many layers, the exact number of which is sometimes arbitrary. Figure 2.4 shows a visual representation of a neural network.

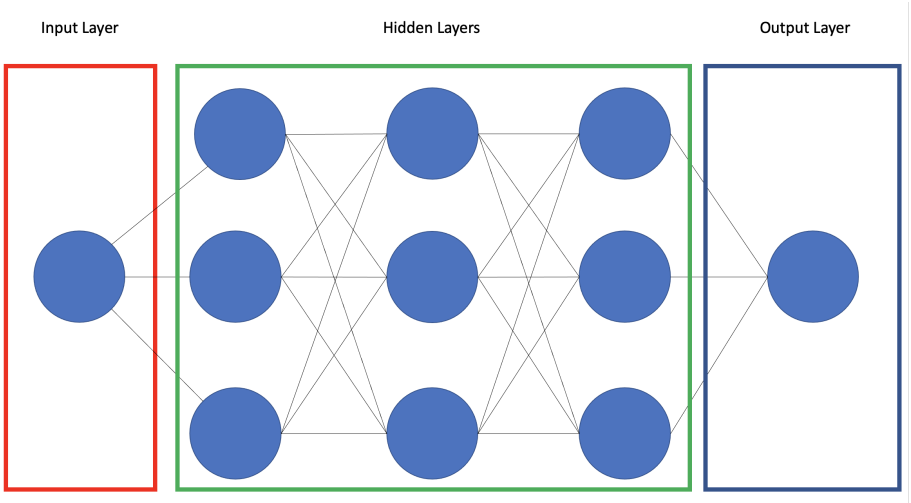

Figure 2.4: Visual representation of a neural network. Individual processing units are connected in layers for representation learning. Input layer refers to the neurons that first process the input vectors. Hidden layers refer to the middle portion where the weights are adjusted to learn abstract representations. Output layer refers to the output from the network.

2.3 Trust in Deep Learning

2.3.1 Quantification of Trust

Adding objectivity and quantification for trust in machine learning is difficult because there is no common definition or agreed upon standard. Schmidt and Biessmann (Schmidt and Biessmann (2019)) bring up a comparison to the Turing test used to define machine intelligence. Before the advent of the Turing test, it was difficult to measure if a machine had intelligence because there was no true definition or test

for machine intelligence. Thus, it is difficult to determine trust and trustworthiness of machine learning algorithms and many are either proxy measurements or qualitative.(Schmidt and Biessmann (2019))

On the other hand, Toreini et al. (Toreini *et al.* (2020)) introduced the idea of FEAS technologies (i.e. fairness, explainability, auditability and safety) as qualities machine learning algorithms must have in order to be trusted. Fairness refers to technology that does not discriminate or have bias against certain demographics and have means to prevent or protect against it. Explainability refers to the ability to explain and interpret results in a humane manner to stakeholders and end users. Auditability refers to the ability to have third-parties examine, regulate, monitor, or challenge the operation of deployed models. Safety refers to the ability for a model to continue to perform as intended in response to passive or active malicious attacks.

2.3.2 Dataset Shift

Dataset shift is a concept that analyzes data quality. Dataset shift occurs when unseen testing data experiences events that lead to a change in the distribution of a feature, a combination of features, or class boundaries and as a result, the assumption that the training data and testing data are representative of the same distribution is violated.(Moreno-Torres *et al.* (2012))

If a classification problem is defined as a set of X features used to produce a target variable Y, then dataset shift is a situation in which the distribution of X_{train} is not equal to the distribution of X_{test}, therefore leading to an invalid relationship between X and Y.

Dataset shift is a classic problem that can occur in medical AI. Medical AI typically

suffers in performance due to changing subjects as populations are not static. Ultrasound introduces many other avenues for dataset shift due to its operator dependency. Ultrasound has high variability across institutions and hardware manufacturers. (Liu *et al.* (2019)) Furthermore, many studies are typically done at only one institution with one ultrasound machine. (Akkus *et al.* (2019)) Thus, US model performance can vary greatly and there is no certainty that a trained model would be fully generalize regardless of the differing operators and hardware.

2.4 Thesis Goals

Core issues in regards to deep learning medical advisory systems for ultrasound revolve around the notion of its operator dependent nature, the lack of explainability in most deep learning models, and the overall subjectivity of ultrasound and trust in deep learning. There is no means to quantify how much operator skill or hardware variance affect the quality of an ultrasound image and no objective metric in which to quantify the trustworthiness of an algorithm's decision.

This thesis aims to address these issues by accomplishing the following goals; to verify and observe differences in deep learning model performance due to hardware and operator differences, to quantify the differences in hardware and operators, and to create a metric to quantify how much the input data varies from the training data.

Chapter 3

Methods

3.1 Data Collection and Hardware

3.1.1 Ultrasound Machines

Siemens S3000

The first machine was the Siemens S3000 Ultrasound (Siemens Healthcare, Erlangen Germany) using the 12L4 Transducer. This machine saved files in an .avi format with a resolution of 1024 x 768. Figure 3.1 shows the Siemens S3000 machine used.

Clarius Ultrasound

The next machine was the Clarius Ultrasound (Clarius Mobile Health, Vancouver Canada) with the L15 Transducer. This machine also saved files in the .avi format and had options for resolution. A resolution of 1024 x 768 was selected for consistency. Figure 3.1 shows the Clarius Ultrasound machine used.

SonixTouch

Lastly, the BK Medical Ultrasonix SonixTouch (BK Medical Ltd., Herlev Denmark) ultrasound (SXTCH3.1-1012.0.11, software Version 6.07) with the L14-5 Transducer was used. This machine saved files as MPEG also with a 1024 x 768 resolution. Figure 3.1 shows the SonixTouch ultrasound machine used.

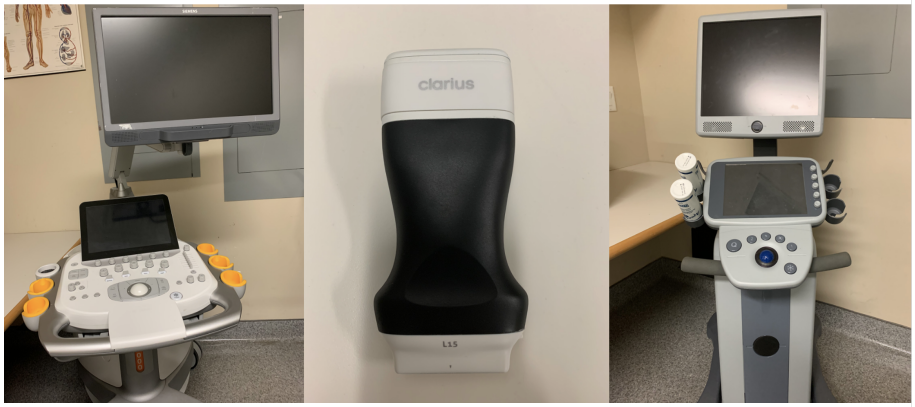

Figure 3.1: Ultrasound machines used. From left to right, Siemens S3000, Clarius Ultrasound, SonixTouch Ultrasound.

3.1.2 Phantom

CAE Blue Phantom

The primary subject for this thesis was a soft tissue mimicking phantom from CAE Healthcare. (Sarasota, FL) Specifically, the 'Soft Tissue Biopsy Ultrasound Training Block Model' was selected. The phantom contains 16 masses of varying sizes (4-11 mm in diameter) that are hypoechoic or hyperechoic in nature, relative to the gel they are embedded in. Figure 3.2 shows the phantom used.

Figure 3.2: Soft tissue mimicking BLUE phantom from CAE Healthcare (Sarasota, FL).

3.1.3 Operators

Operator 1

Operator 1 was a graduate student with no formal training in ultrasound operation and minimal medical imaging experience. The purpose of this operator was to closely represent an operator with minimal technical skill with ultrasound imaging. Figure 3.3 shows sample images collected from Operator 1 using the various machines.

Operator 2

Operator 2 was a medical imaging expert with no formal training in ultrasound operation. The purpose of this operator was to represent an intermediate skill level. Figure 3.4 shows sample images collected from Operator 2 using the various machines.

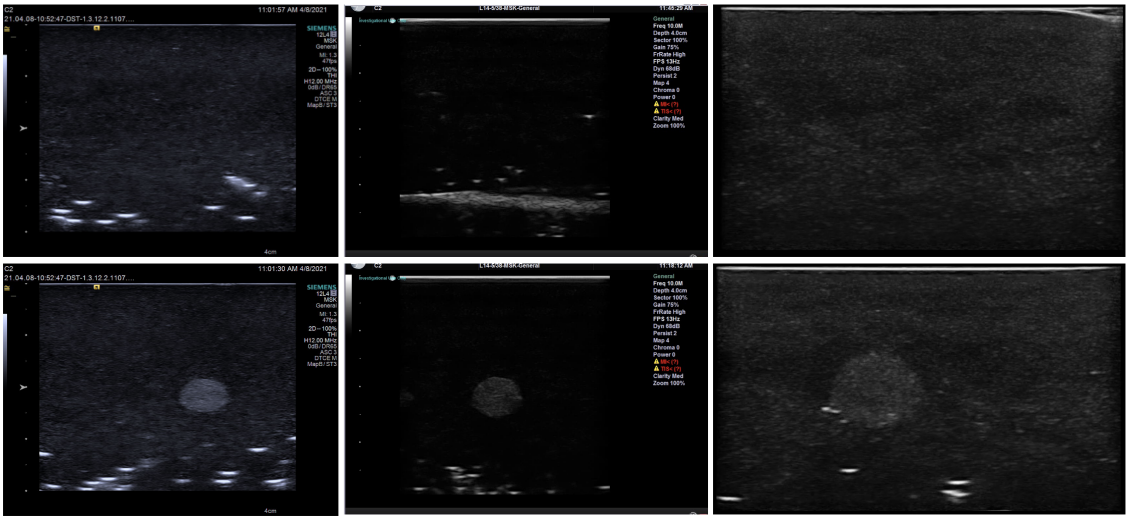

Figure 3.3: Sample images collected from Operator 1. From left to right, images from the Siemens S3000, SonixTouch Ultrasound, and Clarius Ultrasound. Top row shows samples of normal, uniform tissue. Bottom row shows samples of masses.

Operator 3

Operator 3 was an ultrasound technologist with 3 years of formal training at the time of the study and 8 months of clinical experience. The purpose of this operator was to represent trained personnel with a higher skill level. Figure 3.5 shows sample images collected from Operator 3 using the various machines.

3.2 Software

3.2.1 Python 3

All computer code was implemented using Python 3.8 (Python Software Foundation, https://www.python.org) and various supporting open source libraries. Python is an interpreted, high level scripting language. It is open source and runs on various operating systems. Python was selected as the implementation language for the wide

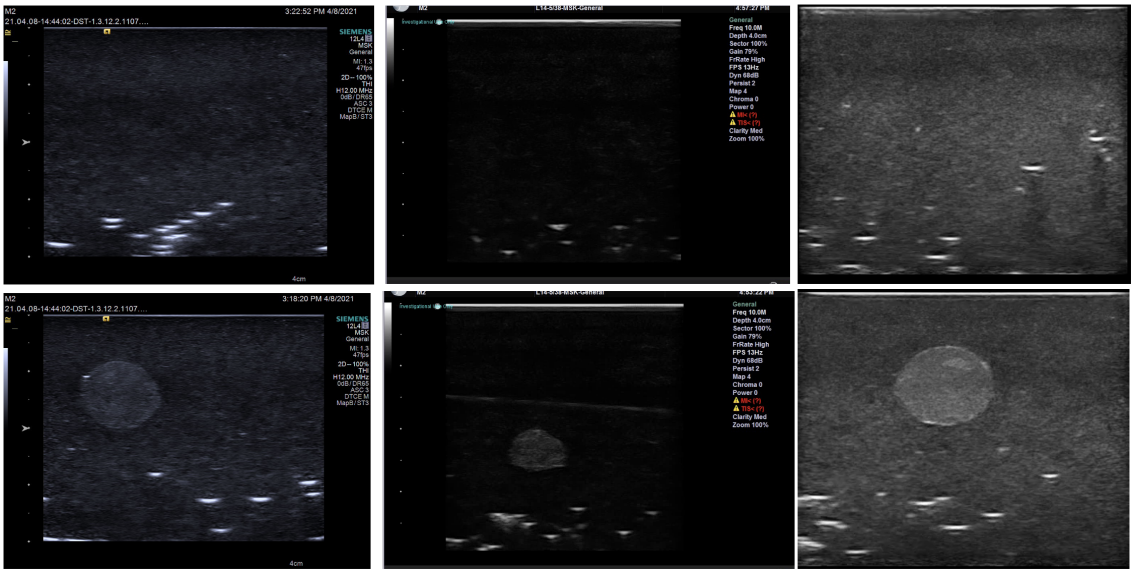

Figure 3.4: Sample images collected from Operator 2. From left to right, images from the Siemens S3000, SonixTouch Ultrasound, and Clarius Ultrasound. Top row shows samples of normal, uniform tissue. Bottom row shows samples of masses.

selection of well supported libraries.

3.2.2 OpenCV

OpenCV (Bradski (2000)) is an open source computer vision library. It was built primarily to support computer vision and machine learning applications and supports a wide variety of algorithms for use. However, in this case, OpenCV was primarily used for its simple video editing and frame extraction functions.

OpenCV for Python was used for the first pre-processing step. Deep learning leans heavily on the ability to automatically extract features. Thus, the preprocessing consisted mostly of converting video files into individual images and cropping UI elements from each frame. Each video was converted to individual frames stored as .PNG at 10 frames per second (FPS) using OpenCV.

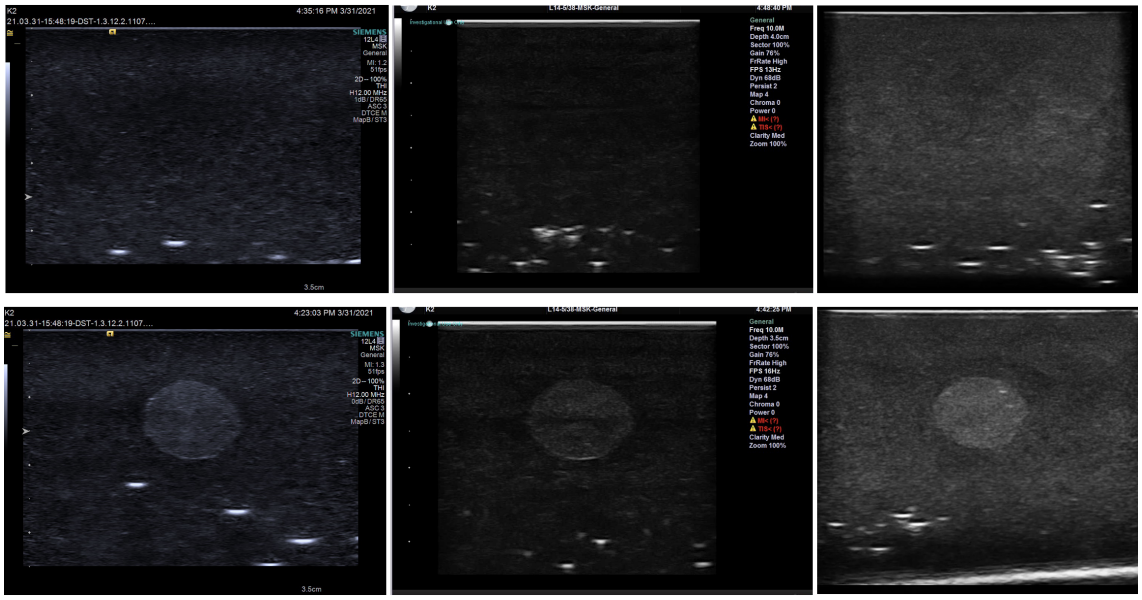

Figure 3.5: Sample images collected from Operator 3. From left to right, images from the Siemens S3000, SonixTouch Ultrasound, and Clarius Ultrasound. Top row shows samples of normal, uniform tissue. Bottom row shows samples of masses.

3.2.3 Python Imaging Library

Python Imaging Library (PIL) (Clark (2015)) is a Python library used for general image processing. PIL was chosen for its ability to support a wide variety of image types and sizes. The individual frames extracted from the video clips were then passed into a PIL based program for further pre-processing. Specifically, cropping of the UI was done using PIL to create 320x720 images with the exception of the Clarius data. The UI of the Clarius app relative to the image is large and thus were cropped to be 250x400 in order to remove the entire UI.

3.2.4 TensorFlow

TensorFlow(Abadi *et al.* (2015)) is an open-source Python package that implements machine learning algorithms that can be used across a variety of systems. It has seen use in many areas of computer science and other fields including speech recognition, computer vision, robotics and more. TensorFlow was used for re-sizing images to be 150x150 in order to speed up training and unify their size for giving to each model and for implementing the deep learning models.

3.2.5 SciKit-Learn

SciKit-Learn(Pedregosa *et al.* (2011)) is an open source, Python-based, machine learning library built for medium scale supervised and unsupervised learning problems. SciKit-Learn was used for implementing Principle Component Analysis and the K-Means Clustering algorithm.

3.3 Model Details

3.3.1 Principle Component Analysis (PCA)

Conventional machine learning methods typically do not handle raw data very well and as such a feature extractor was required. Principle component analysis (PCA) is a common feature extraction and dimensionality reduction tool used for many machine learning applications. PCA works by selecting a hyperplane closest to the data and projecting the data onto an axis along it. By projecting the data onto a hyperplane that minimizes the mean squared distance between the original data and its projection onto the axis, most of the variance in the data is preserved. The data

projected onto this hyperplane is referred to as the first principle component. The data is then projected onto axes on hyperplanes orthogonal to this plane forming additional principle components and preserving the rest of the variance in the data. (Géron (2019))

This whole method is done by singular value decomposition (SVD) described by Equation 3.1.

$$X = U\Sigma V^T \tag{3.1}$$

where X is the training set matrix and V^T contains unit vectors that define all principle components.

3.3.2 K-Means Clustering

The implementation of K-Means clustering through the SciKit Learn library uses a K-Means++ initialization which tends to select centroids further apart. The algorithm is as follows:

1. Take one centroid, c_1, chosen uniformly at random from the dataset
2. Take a new centroid, c_i, choosing an instance x_i with probability

$$\frac{D(x_i)^2}{\Sigma_{j=1}^{m} D(x_j)^2} \tag{3.2}$$

   where $D(x_i)$ is the distance between xi and the closest centroid that was already chosen

3. Repeat for k clusters

Furthermore, to reduce the likelihood of convergence to local minima, the K-Means algorithm is repeated by default 10 times and the result with the lowest inertia is returned. Inertia is defined as the mean squared distance between each instance and its closest centroid.

The performance of these clustering models are evaluated using the silhouette score. Silhouette score is a method for cluster analysis that shows which instances are well placed in a cluster and those instances which are somewhere between clusters.(Rousseeuw (1987)) Clusters are represented with silhouettes where average silhouette width is a means for evaluating cluster validity and can be used to select an appropriate amount of clusters. Silhouettes are constructed from cluster labels and the distances between instances computed by Equation 3.3.

$$s(i) = \frac{b(i) - a(i)}{max(a(i), b(i))} \tag{3.3}$$

where i is an instance in a cluster, a(i) is the average dissimilarity (distance) of i and all other objects in the same cluster, and b(i) is given by Equation 3.4.

$$b(i) = minimum(d_i, C) \tag{3.4}$$

where $C \neq A$ and (d_i, C) is the average distance from the instance to all values in another cluster.

The silhouette score is bounded between $-1 \leq s(i) \leq 1$ and scores close to -1 indicate a likely mis-classification, scores close to 0 indicate a decision boundary, and scores close to 1 indicate that the instance is close to the cluster center.

3.3.3 Deep Learning

Architectures

The following three architectures were chosen for the deep learning models for their high performance in image classification tasks.

Xception (Chollet (2017)) is a convolutional neural network based on depthwise separable convolutional layers. It has 36 convolutional layers structured into 14 modules to act as a feature extracter followed by a logistic regression layer.

ResNet50 (He *et al.* (2015)) uses a deep residual learning framework to allow for better convergence on a deeper network at 34 layers. ResNet50 was the architecture that won 1st place in the ILSVRC 2015 classification competition.

VGG19 (Simonyan and Zisserman (2015)) is a convolutional neural network architecture that won 2nd place in the ImageNet Challenge 2014 for its classification performance. It is a network that uses small, 3x3 convolutional filters. This version uses 19 weight layers.

A normalization layer is included at the beginning of each network in order to accommodate the architectures being used. Also, a drop out layer is included before the output layer to regularize and help with overfitting.

The overall architecture is shown in Figure 3.6.

Training

After preprocessing a total of 39013 images were collected. Each set was divided into approximately 80% training and validation and 20% testing. Care was taken to ensure that data from one scan was not included in both the training and test sets in order to avoid overfitting and inflating the test accuracy.

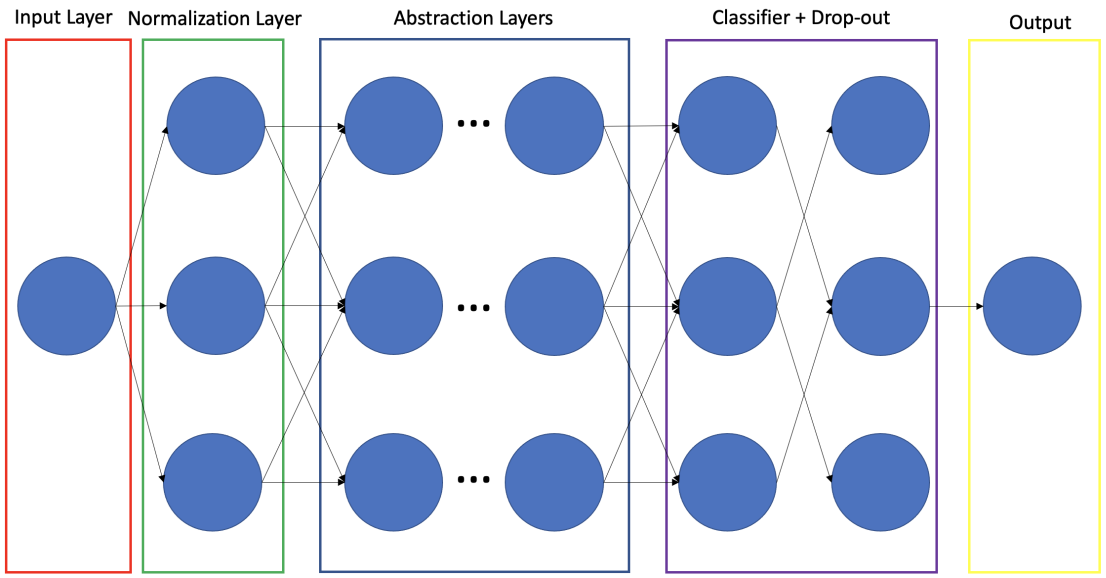

Figure 3.6: Visual depiction of the architecture used shown for clarity. Abstraction layers refer to the specific architecture Xception, ResNet50, or VGG19.

Optimizer

Adam (Kingma and Ba (2017)) is an algorithm for gradient-based optimization of stochoastic objective functions. It requires a learning rate, α and exponential decay rates, $\beta 1$ and $\beta 2$. The Adam implementation in TensorFlow uses default values as follows, learningrate=0.001, $\beta 1$=0.9, $\beta 2$=0.999.

3.4 Experimental Set-ups

3.4.1 Investigating Hardware and Operator Variability

The purpose of this portion was to verify and observe if any differences in performance existed when deep learning models are given ultrasound images from varying hardware

and operator setups despite the content of the image being the same.

Experimental Set-up

The data was grouped into various sets in order to facilitate ease of data flow. Table 3.1 shows how the data was grouped.

Table 3.1: Data Sets from Operator 3

Dataset	Train Set	Testing Set
Sonix	4836	1193
Siemens	5462	1572
Clarius	5772	1567
Total	16070	4332

Table 3.2: Data Sets from Siemens S3000

Dataset	Train Set	Testing Set
Operator 1	5041	1412
Operator 2	3954	1170
Operator 3	5462	1572
Total	14457	4154

Models were trained given data acquired from one machine only. In this first step, the operator was kept consistent to ensure that differences in performance would primarily come from the hardware differences. Data from Operator 3 was used. Once training was complete, the models were tested on data from the other machines to observe performance.

Next, models were trained given data acquired from one operator only. In this second step, the machine was kept consistent to ensure that differences in performance would primarily come from the operator differences. Data from the Siemens S3000

was used. Once training was complete, the models were tested on data from the other operators to observe performance.

Figure 3.7 shows a visual representation of the training and testing process.

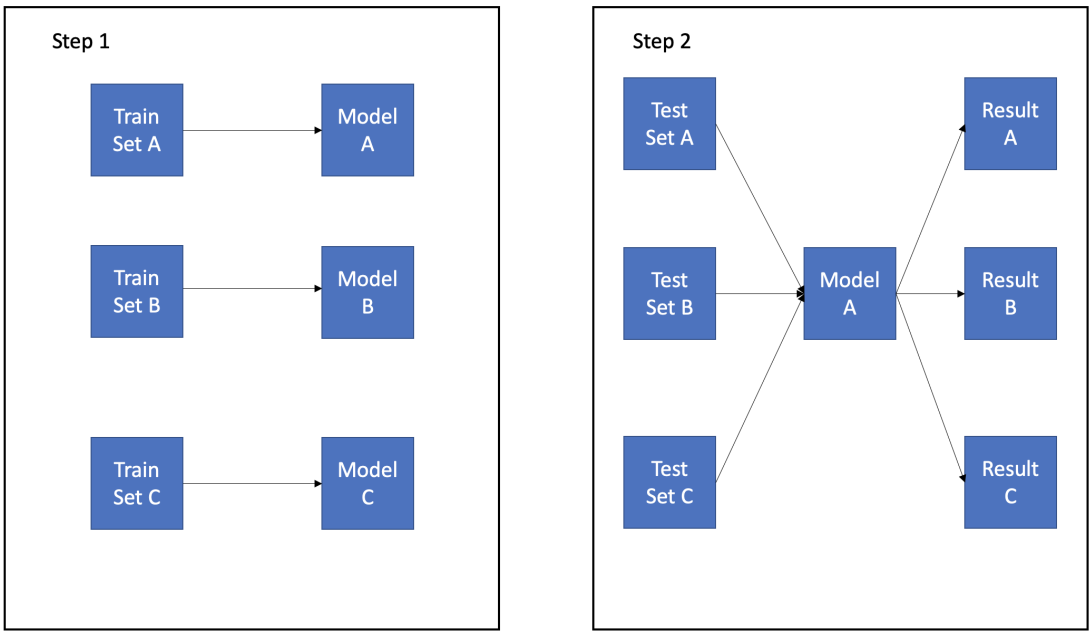

Figure 3.7: Training and testing method for the Deep learning models. Step 1: Train individual models with data from only one designated set. Step 2: Test model performances from the varying sets.

3.4.2 Quantifying Hardware and Operator Variability

The purpose of this portion was to quantify the differences in hardware and operator setups of ultrasound image acquisition. It was proposed that a clustering algorithm would be able to identify differences in hardware and operators due to the differences in image acquisition and the distances between clusters would be a metric to define the differences in hardware types and operator skill.

Experimental Set-up

PCA is first applied to the datasets for dimension reduction and feature extraction. The K-Means algorithm was then ran on the data from the three machines. The operator was kept consistent and data from Operator 3 was used. The number of clusters generated ranged from 2 to 6. Then this was repeated on data from the three operators where the machine was kept consistent and data from the Siemens S3000 was used. The number of clusters generated also ranged from 2 to 6. Figure 3.8 shows a visual representation of the cluster generating process.

The same data in Tables 3.1 and 3.2 were given to the clustering algorithm to create clusters.

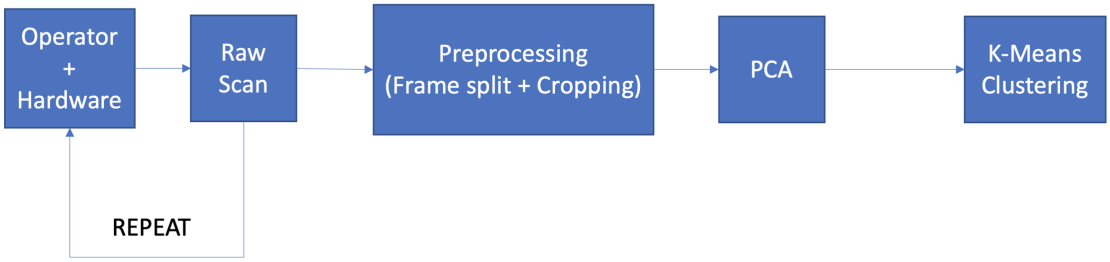

Figure 3.8: Visual representation of the clustering process. This process is repeated with differing combinations of hardware and operators to generate from 2 to 6 clusters.

3.4.3 Proposed Method

By using the clustering model as a screening tool, an end user is given more information about the model's decisions by which to trust the results or not. Furthermore,

the models themselves will in theory be more resistant to dataset shift as there will be a means to ensure that the models are only given data similar to what it was trained on.

Experimental Set-up

To test this, incoming test data was first given to the K-Means cluster model and the Euclidean distance from each cluster center is calculated. The data was then given to the model of the closest center. See Figure 3.9 for a visual representation.

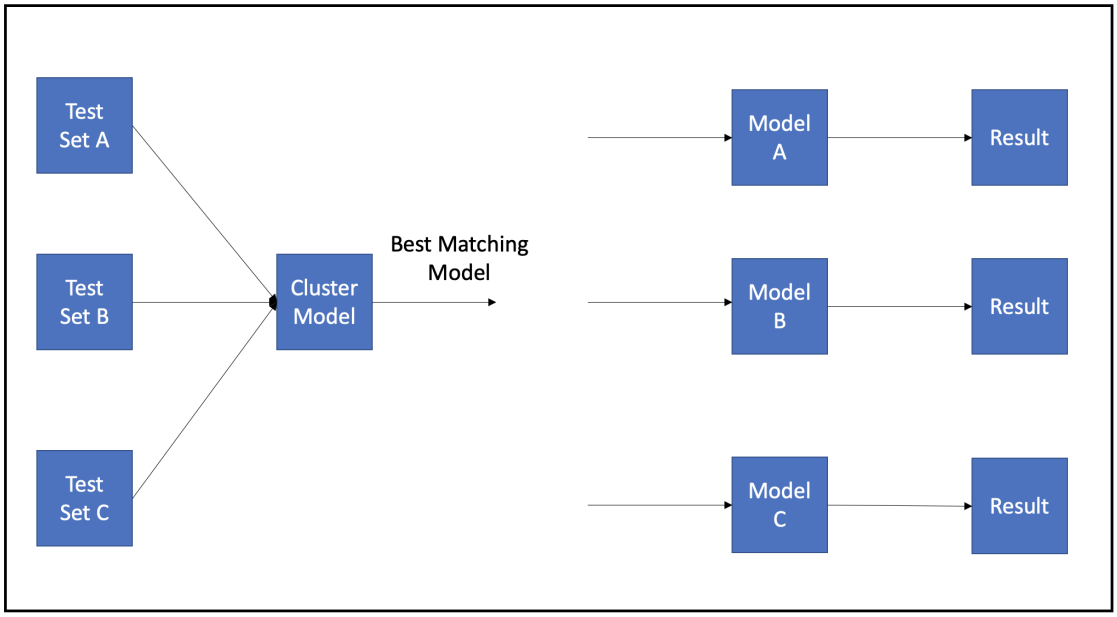

Figure 3.9: Proposed method to create more consistent model performance. Test data was first given to a clustering model to determine which model is most appropriate before classification.

Chapter 4

Results

4.1 Observing Machine and Operator Variability

4.1.1 Hardware Variability

Initial Training and Testing Results

Models were trained and tested within distribution, i.e. the training and test data were from the same operator and machine. These results were used as a base to compare performance when tested out of distribution, i.e. the test data is from a different machine. Figure 4.1 shows training results for the Xception models. Figure 4.2 shows training results for the ResNet50 models. Figure 4.3 shows training results for the VGG19 models. Table 4.1 shows the testing results.

Out of Distribution Results

Each Model was tested with the theorized out of distribution test set from each different machine while maintaining the same operator, operator 3. The expert level

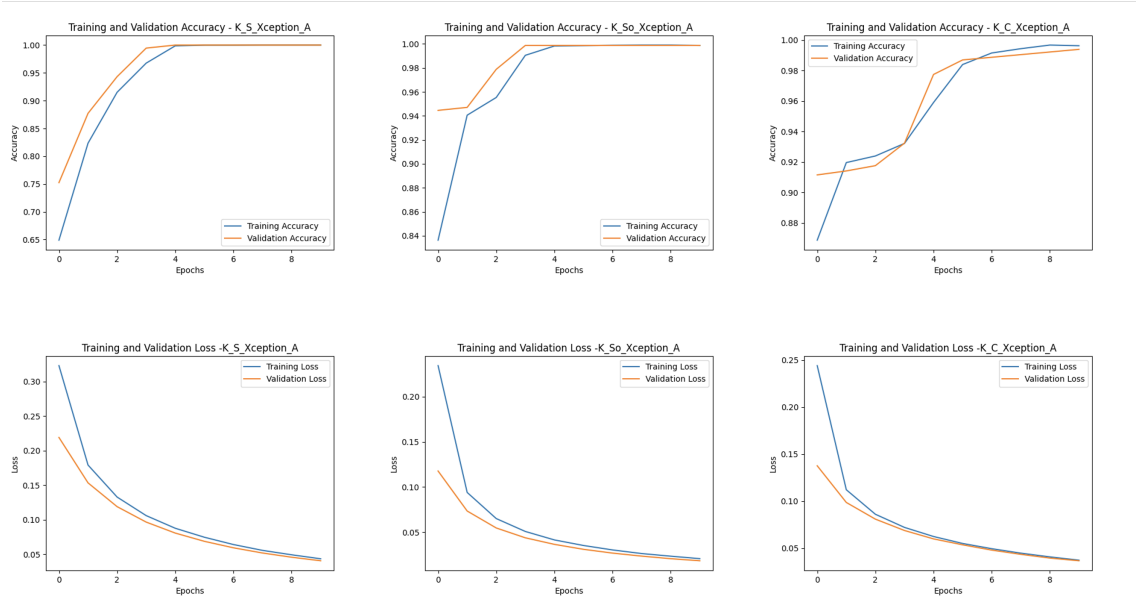

Figure 4.1: Training results from the hardware Xception specific models. From left to right, Siemens S3000, SonixTouch Ultrasound, Clarius Ultrasound. Top row shows training and validation accuracy. Bottom row shows training and validation loss.

operator selected to minimize errors in scanning and ensure differences were due to hardware differences. Tables 4.2, 4,3, 4.4, show the test results for the Xception, ResNet50, and VGG19 models respectively.

4.1.2 Operator Variability

Initial Training and Testing Results

Models were trained and tested within distribution, i.e. the training and test data were from the same operator and machine. These results were used as a base to compare performance when tested out of distribution, i.e. the test data is from a different operator. Figure 4.4 shows training results for the Xception models. Figure 4.5 shows training results for the ResNet50 models. Figure 4.6 shows training results

Table 4.1: Training and testing model results for the hardware specific models. Models trained on specific hardware are tested with the matching data sets as a baseline for performance.

Model	Architecture	Training Accuracy	Testing Accuracy	Sensitivity	Specificity
Sonix	Xception	0.9988	1.0	1.0	1.0
	ResNet50	0.7330	0.8878	0.9736	0.6939
	VGG19	0.9376	0.9996	1.0	0.9988
Siemens	Xception	1.00	1.0	1.0	1.0
	ResNet50	0.7995	0.8126	0.6242	0.9410
	VGG19	0.9332	0.8483	0.8097	0.8745
Clarius	Xception	0.9963	0.9948	1.0	0.9894
	ResNet50	0.6831	0.7634	0.5415	1.0
	VGG19	0.9335	0.9616	0.9293	0.9960

Table 4.2: Out of Distribution Hardware Test Results - Xception Models. Models trained on specific hardware are tested with the differing data sets to look for differences in performance.

Model	Dataset	Testing Accuracy	Sensitivity	Specificity
Sonix	Sonix	1.0	1.0	1.0
	Siemens	0.9783	1.0	0.9635
	Clarius	0.9475	1.0	0.8916
Siemens	Sonix	0.9659	1.0	0.8890
	Siemens	1.0	1.0	1.0
	Clarius	0.7653	1.0	0.5151
Clarius	Sonix	0.9770	0.9669	1.0
	Siemens	0.7845	1.0	0.6377
	Clarius	0.9948	1.0	0.9894

Table 4.3: Out of Distribution Hardware Test Results - ResNet50 Models. Models trained on specific hardware are tested with the differing data sets to look for differences in performance.

Model	Dataset	Testing Accuracy	Sensitivity	Specificity
Sonix	Sonix	0.8878	0.9736	0.6939
	Siemens	0.8068	0.7437	0.8499
	Clarius	0.5134	0.9467	0.0515
Siemens	Sonix	0.7870	0.7487	0.8738
	Siemens	0.8126	0.6242	0.9410
	Clarius	0.3126	0.5563	0.0528
Clarius	Sonix	0.3752	0.0987	1.0
	Siemens	0.5073	0.5393	0.4855
	Clarius	0.7634	0.5415	1.0

Table 4.4: Out of Distribution Hardware Test Results - VGG19 Models. Models trained on specific hardware are tested with the differing data sets to look for differences in performance.

Model	Dataset	Testing Accuracy	Sensitivity	Specificity
Sonix	Sonix	0.9996	1.0	0.9988
	Siemens	0.8833	0.9323	0.8499
	Clarius	0.5345	1.0	0.0383
Siemens	Sonix	0.8487	1.0	0.5070
	Siemens	0.8483	0.8097	0.8745
	Clarius	0.5319	1.0	0.0330
Clarius	Sonix	0.3860	0.1142	1.0
	Siemens	0.7558	0.4025	0.9967
	Clarius	0.9616	0.9293	0.9960

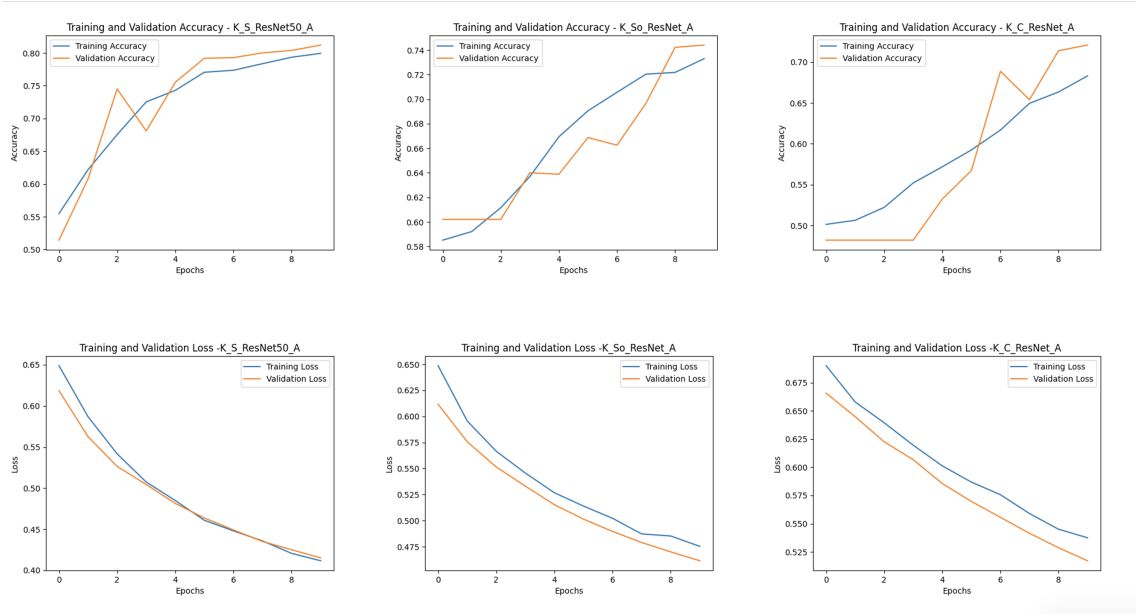

Figure 4.2: Training results from the hardware specific ResNet50 models. From left to right, Siemens S3000, SonixTouch Ultrasound, Clarius Ultrasound. Top row shows training and validation accuracy. Bottom row shows training and validation loss.

for the VGG19 models. Table 4.5 shows the testing results.

Out of Distribution Results

Each Model was tested with the theorized out of distribution test set from each operator while maintaining the same hardware source, the Siemens S3000. The Siemens S3000 was selected as it was the machine with the best resolution. Tables 4.6, 4.7, 4.8, show the test results for the Xception, ResNet50, and VGG19 models respectively.

Table 4.5: Training and testing model results for the operator specific models. Models trained on specific operators are tested with the matching data sets as a baseline for performance.

Model	Architecture	Training Accuracy	Testing Accuracy	Sensitivity	Specificity
Operator 1	Xception	0.9975	0.8665	0.9984	0.7496
	ResNet50	0.6296	0.4989	0.4350	0.5555
	VGG19	0.9239	0.8388	0.9848	0.7095
Operator 2	Xception	0.9937	0.9700	0.9637	1.0
	ResNet50	0.7112	0.7968	0.8582	0.5037
	VGG19	0.8943	0.9365	0.8834	1.0
Operator 3	Xception	1.0	1.0	1.0	1.0
	ResNet50	0.7995	0.8126	0.6242	0.9410
	VGG19	0.9332	0.8483	0.8097	0.8745

Table 4.6: Out of Distribution Operator Test Results - Xception Models. Models trained on specific operators are tested with the differing data sets to look for differences in performance.

Model	Dataset	Testing Accuracy	Sensitivity	Specificity
Operator 1	Operator 1	0.8665	0.9984	0.7496
	Operator 2	0.9947	0.9937	1.0
	Operator 3	0.5015	1.0	0.1618
Operator 2	Operator 1	0.9609	0.9516	0.9692
	Operator 2	0.9700	0.9637	1.0
	Operator 3	0.9987	1.0	0.9978
Operator 3	Operator 1	0.8112	0.9592	0.6800
	Operator 2	0.7773	0.8251	0.5488
	Operator 3	1.0	1.0	1.0

Table 4.7: Out of Distribution Operator Test Results - ResNet50 Models. Models trained on specific operators are tested with the differing data sets to look for differences in performance.

Model	Dataset	Testing Accuracy	Sensitivity	Specificity
Operator 1	Operator 1	0.4989	0.4350	0.5555
	Operator 2	0.3919	0.4283	0.2180
	Operator 3	0.6908	0.2374	1.0
Operator 2	Operator 1	0.5344	0.7859	0.3119
	Operator 2	0.7968	0.8582	0.5037
	Operator 3	0.6934	0.4292	0.8735
Operator 3	Operator 1	0.4705	0.9486	0.0468
	Operator 2	0.7656	0.9259	0
	Operator 3	0.8126	0.6264	0.9410

Table 4.8: Out of Distribution Operator Test Results - VGG19 Models. Models trained on specific operators are tested with the differing data sets to look for differences in performance.

Model	Dataset	Testing Accuracy	Sensitivity	Specificity
Operator 1	Operator 1	0.8388	0.9848	0.7095
	Operator 2	0.9425	0.8944	1.0
	Operator 3	0.4149	0.9842	0.0267
Operator 2	Operator 1	0.9041	0.9138	0.8955
	Operator 2	0.9365	0.8834	1.0
	Operator 3	0.8355	0.7783	0.8745
Operator 3	Operator 1	0.5805	1.0	0.2088
	Operator 2	0.5441	1.0	0
	Operator 3	0.8483	0.8097	0.8745

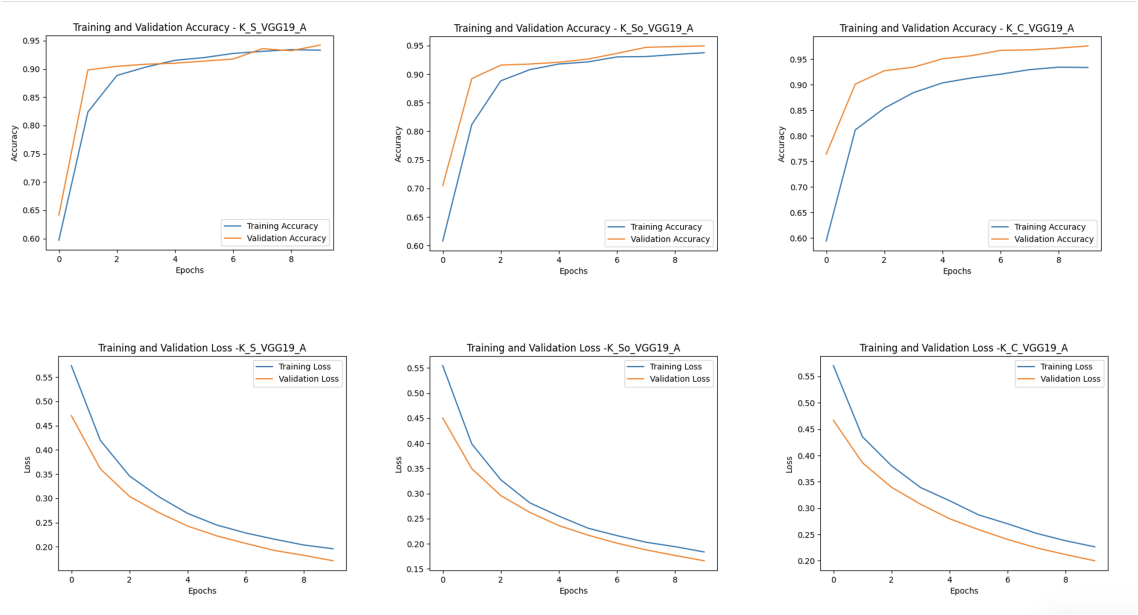

Figure 4.3: Training results from the hardware specific VGG19 models. From left to right, Siemens S3000, SonixTouch Ultrasound, Clarius Ultrasound. Top row shows training and validation accuracy. Bottom row shows training and validation loss.

4.2 Quantifying Machine and Operator Variability

4.2.1 Principle Component Analysis (PCA) Results

Using PCA the dimensionality of the hardware data was able to be reduced to 175 components while preserving 95% of the variance. In terms of the operator data PCA was used to successfully reduce the dimensionality of that data to 482 components while preserving 95% of the variance.

4.2.2 Hardware Clusters

The K-Means algorithm was run repeatedly to generate n = 2 to n = 6 clusters with data from Operator 3 using all three machines. The expert level operator selected

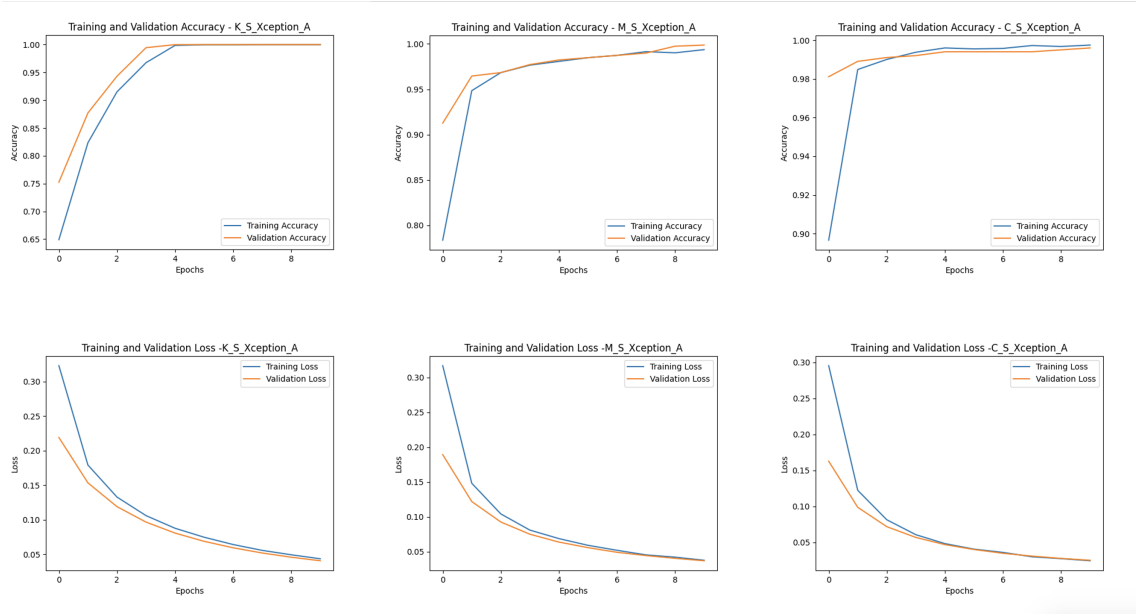

Figure 4.4: Training results from the operator specific Xception models. From left to right, Siemens S3000, SonixTouch Ultrasound, Clarius Ultrasound. Top row shows training and validation accuracy. Bottom row shows training and validation loss.

to minimize errors in scanning. The number of clusters was selected to range from the minimum number of potential expected clusters, 2, to the maximum number of potential expected clusters, 6. n = 2 clusters corresponds to a split between normal and abnormal tissue scans. n = 6 clusters corresponds to a split between 3 machines, each with a split of normal and abnormal tissue scans. Figure 4.7 shows the results for the expected n = 3 clusters. Refer to Figures A.1 to A.4 in Appendix A.01 for the other cluster results. The mis-classification rate for the n = 3 clusters model was 0.597%. For n = 2 through 6, the average silhouette scores were 0.3446, 0.2683, 0.2638, 0.2885 and 0.2923, respectively. Meanwhile, the average distance between clusters was 45.3892.

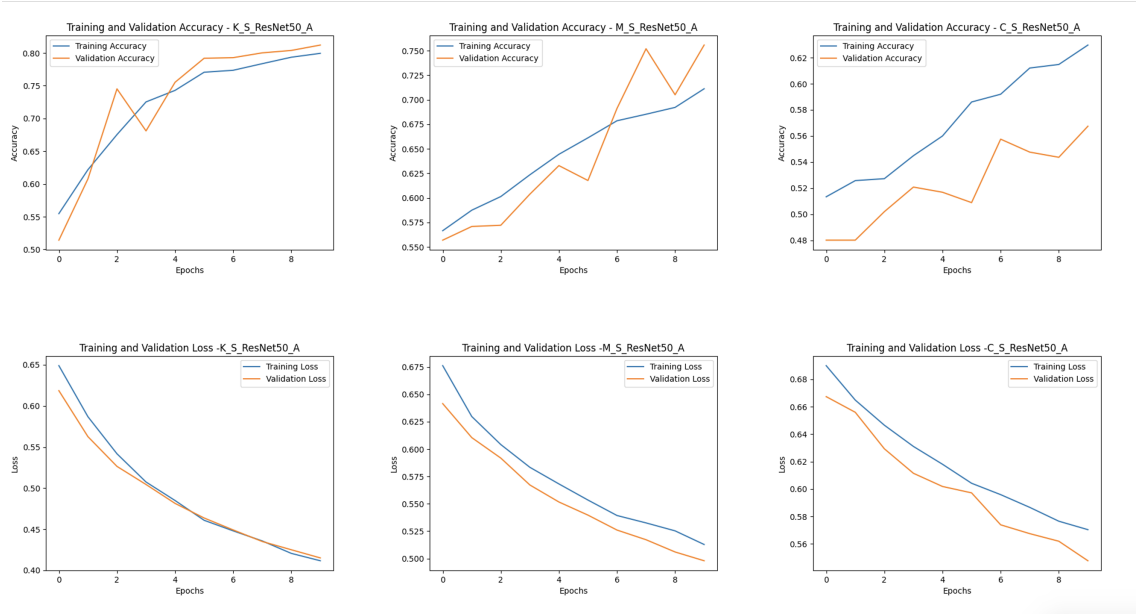

Figure 4.5: Training results from the operator specific ResNet50 models. From left to right, Siemens S3000, SonixTouch Ultrasound, Clarius Ultrasound. Top row shows training and validation accuracy. Bottom row shows training and validation loss.

4.2.3 Operator Clusters

The K-Means algorithm was once again ran repeatedly to generate n = 2 to n = 6 clusters but with data from all operators using the Siemens S3000 machine. The Siemens S3000 was selected as it was the machine with the best resolution. The number of clusters was selected to range from the minimum number of potential expected clusters, 2, to the maximum number of potential expected clusters, 6. n = 2 clusters corresponds to a split between normal and abnormal tissue scans. n = 6 clusters corresponds to a split between 3 operators, each with a split of normal and abnormal tissue scans. Figure 4.8 shows the results for the expected n = 3 clusters. Refer to Figures A.5 to A.8 in Appendix section A.01 for the other cluster results. The mis-classification rate for the n = 3 clusters model was 71.4%. For n = 2 to 6

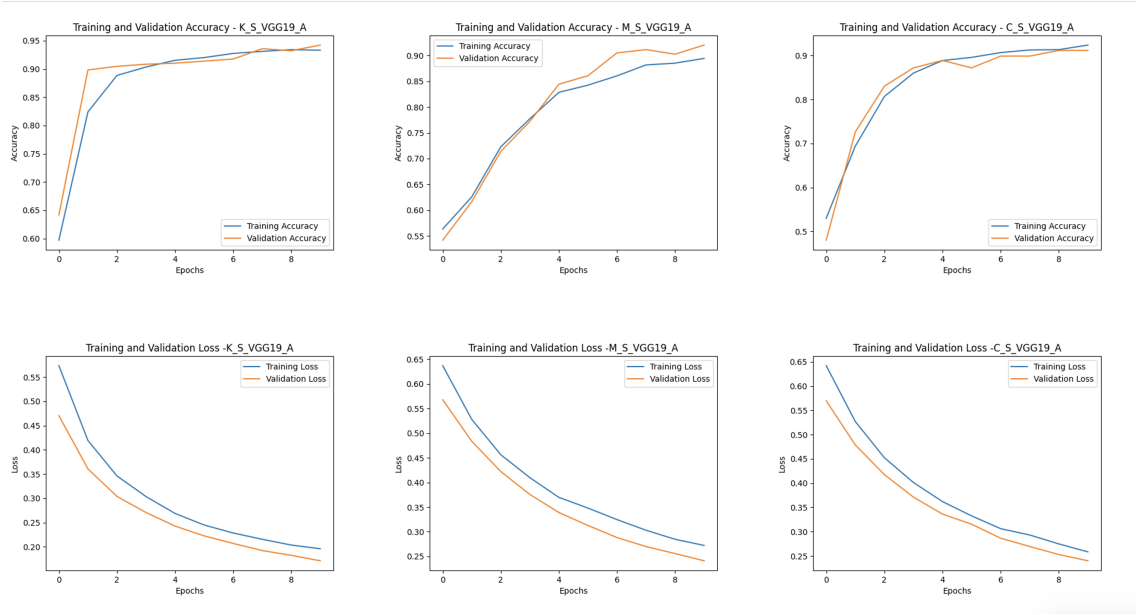

Figure 4.6: Training results from the operator specific VGG19 models. From left to right, Siemens S3000, SonixTouch Ultrasound, Clarius Ultrasound. Top row shows training and validation accuracy. Bottom row shows training and validation loss.

clusters, the average silhouette scores were 0.1204, 0.1290, 0.1241, 0.1268 and 0.1381, respectively. Here, the average distance between clusters was 37.0233.

4.3 Proposed Method

Each test sample was first passed through the clustering algorithm to identify which cluster it belonged to. From there, the designated deep learning model was used for classification. This process was repeated for each type of architecture, the Xception, ResNet50, and VGG19. Table 4.9 shows the testing accuracy.

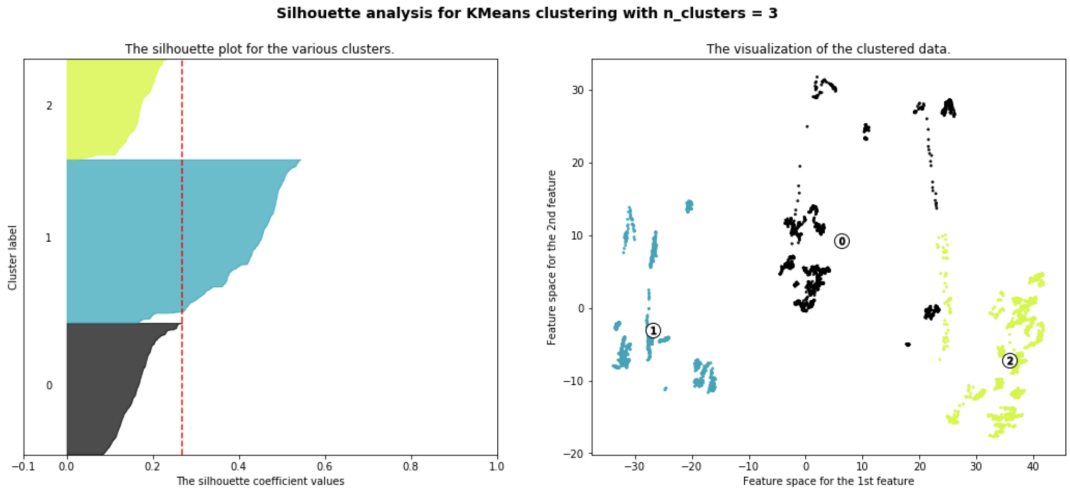

Figure 4.7: Silhouette analysis and visual representation of the clusters running K-Means to generate 3 clusters of hardware. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted. Cluster 0 represents the Siemens S3000. Cluster 1 represents the SonixTouch Ultrasound. Cluster 2 represents the Clarius ultrasound.

Table 4.9: Model Results - Proposed Method. Testing accuracy is determined from the entire test set, including data from each hardware dataset in the hardware cluster models and data from each operator in the operator cluster models.

Cluster Type	Model	Testing Accuracy
Hardware	Xception	0.9979
	ResNet50	0.8266
	VGG19	0.9311
Operator	Xception	0.8854
	ResNet50	0.6026
	VGG19	0.8270

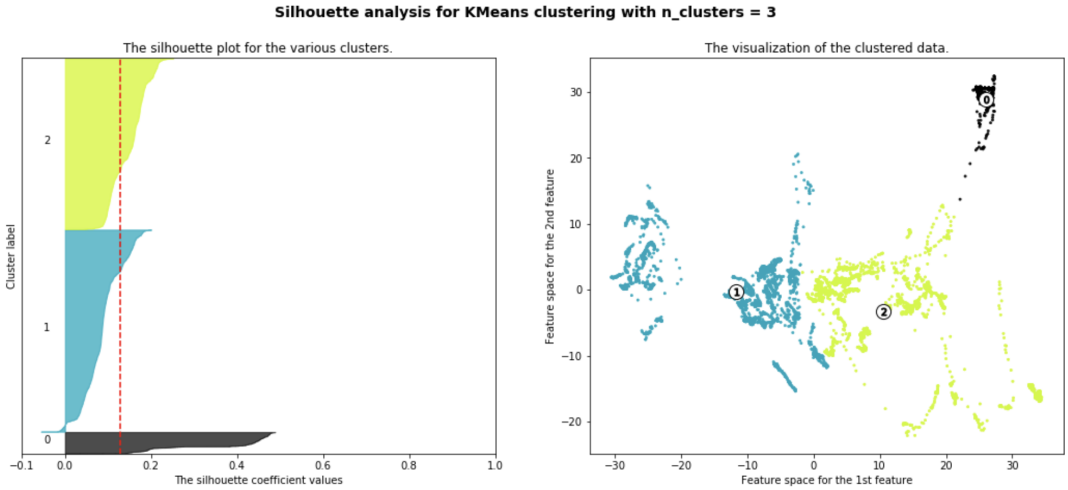

Figure 4.8: Silhouette analysis and visual representation of the clusters running K-Means to generate 3 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted. Cluster 0 represents operator 2, the intermediate operator skill level. Cluster 1 represents operator 1, the novice operator skill level, and Cluster 2 represents operator 3, the expert operator skill level.

Chapter 5

Discussion

5.1 Hardware and Operator Variability

Overall, the models had good in-distribution testing results. The Xception models performed the best with all models having an in distribution testing accuracy of over 86.65%, with the highest being 100%. As expected, models evaluated on 'out of distribution' data had inconsistent performance. Performance varied widely when the hardware or operator was different than the training data.

The largest changes in testing accuracy of the hardware based models was found when the Clarius data was tested on differing models such as in the Siemens ResNet50 model which dropped from 81.26% accuracy to 31.26%, the Siemens Xception model which dropped from 100% to 76.53%, the SonixTouch ResNet50 model which dropped from 88.78% to 51.34%, and the SonixTouch VGG19 model which dropped from 99.96% to 53.45%. Of note is that the Clarius ultrasound is the only portable ultrasound machine used in this thesis and thus differs in hardware inherently. The smallest drop in accuracy of the hardware models was the SonixTouch Xception model which

only dropped between 2-5% in testing accuracy going from the SonixTouch test set to the others.

In comparison to the hardware models, the largest drop in test accuracy of the operator models was found when the models trained with operator 3, the expert level operator, was tested with the other data. When operator 3 models were given data from the other operators, the Xception model dropped in testing accuracy by over 20%, the ResNet50 model dropped in testing accuracy by up to 34%, and the VGG19 model dropped up to 40%. Counter intuitively, some models had increased testing accuracy when evaluated out of distribution which was the case primarily with operator 1, the novice skill level. In the Xception model, the testing accuracy increased from 86.65% to 99.47% when tested with data from operator 2, but sharply decreased to 50.15% when tested with data from operator 3. This result occured again in the VGG19 model where the testing accuracy increased from 83.88% to 94.25% when tested with data from operator 2, but sharply decreased to 41.49% when tested with data from operator 3.

Interestingly, models trained using the data from operator 2, the intermediate skill level, seemed less variable to changes in operators. With operator 2, the Xception model had performances within 3% performance differences, and the ResNet50 and VGG19 models had performances within 10%. This may be due to an intermediate level operator scanning with elements from both a novice and expert. In this way, the model would theoretically be exposed to tendencies from a wider range of skill levels.

Overall, the out of distribution tests tended to have skewed sensitivity or specificity, where in some models the specificity dropped to almost 0. The larger numbers

of false positives may be due to the models interpreting the differences in the distribution of the test data as an 'abnormality' akin to the tumours the models are trying to classify, and follow up work may be done to verify this. In terms of architecture, the Xception models as a whole seemed the least affected by changing the testing data and overall had the best performance. Based on these findings it appears that operator variance affected performance more.

5.2 Quantifying Differences

The hardware cluster model with 3 clusters had a mis-classification rate of 0.597%. The highest silhouette scores correspond to 2 and 6 clusters with 0.3446 and 0.2923 respectively. The operator cluster model with 3 clusters had a mis-classification rate of 71.4%. Similarly the highest silhouette scores corresponded to 3 and 6 clusters with 0.1290 and 0.1381, respectively. These higher scores imply that the number of clusters in the data align with the designated clusters when collecting the data. 2 clusters imply that there may be clearer separation between abnormal tissue and normal tissue. 3 clusters imply that there is a separation between the three types of hardware or operators used. 6 clusters imply separation in both abnormal and normal tissue for each with the three types of hardware or operators.

Overall, the operator cluster models had lower average silhouette scores and higher average distance between clusters implying that the clusters in the operator cluster model are less separable. This is consistent with the extremely high mis-classification rate. This suggests that differences in hardware are more easily found than differences in operators which is against the common notion that ultrasound is a heavily operator dependent imaging modality. This finding however, contradicts the previous section

where operator variance had a larger impact on model performance. The consistency of model performance and cluster centre distances should in theory have an inverse relationship. The further clusters are, the more separable the data is, and thus there are larger differences in the data. Larger differences in the data should increase the effects of data set shift and thus lead to lower model consistency.

Of note, is that the silhouette scores of the cluster models with 4 or 5 clusters are comparable to the scores of the other models. The hardware cluster model with 5 clusters in fact had a higher score of 0.2885 compared to the model with 3 clusters of 0.2683. This suggests that there may be overlap in clusters instead of simply having the expected clusters of 2, 3, and 6. There may be be 4 or 5 clusters instead where some of the data overlap to form one or two larger clusters due to overlap in operator tendencies or machine characteristics.

5.3 Proposed Method

Using the proposed screening method with the hardware cluster model resulted in more consistent model performance when given out of distribution data. The Xception models in particular saw very good test results with 99.79% accuracy when given the test sets of the three machines. The VGG19 models had similar performance at 93.11%. The ResNet50 models had 82.66%.

Using the proposed screening method with the operator cluster model resulted in a lower performance compared to the hardware cluster model. The Xception models still performed the best at 88.54% accuracy across the test sets. The VGG19 models had an accuracy of 82.70% and the ResNet50 Models had an accuracy of 60.26%.

The poor performance of the operator models with the proposed method is likely

due to the poor performance of the corresponding cluster model. In theory, the cluster model would act as a proper screening method to ensure that the deep learning model used for classification would see test samples within distribution. However, this is highly dependent on the performance of the cluster model.

Chapter 6

Conclusion

In this thesis, hardware and operator variability was studied for investigating its effects on deep learning model performance and to create an objective metric by which to help users trust deep learning medical advisory systems for ultrasound. This thesis proposed using K-Means clustering to screen incoming data and distances between test samples and the generated cluster centres as a means to produce a quantifiable metric to improve trust in deep learning models.

This was primarily motivated by the subjectivity in ultrasonography and the ability for deep learning to provide objectivity in a medical advisory system. However, deep learning is 'black box' in nature and it is difficult to trust results when much of the information used to make the decision is hidden. Ultrasonography is extra susceptible to error due to dataset shift due to operator and hardware variance leading to even more difficulty trusting results. Therefore, a method to investigate the variance and provide more information to users was proposed.

This was done by first gathering B-mode ultrasound scans from differing hardware and operator combinations and training various deep learning models based on this

data. These models were tested on out of distribution data to investigate the effects of hardware and operator variance. In terms of hardware variance, it was found that the Clarius ultrasound created the largest discrepancies in testing results when models were tested out of distribution. This hardware mismatch created drops in accuracy of upwards of 50%. In terms of operator variance, models trained with data collected from expert operator had the largest drops in performance when tested out of distribution. The operator mismatch created drops in accuracy of upwards of 40%. Xception models as a whole performed the best, and it appears that operator variance creates larger discrepancies in testing accuracy.

Next, these scans were clustered using PCA and the K-Means algorithm to create a measurement by which to quantify the differences caused by operators and hardware. These measurements were used as both a screening tool and means to provide users with more information about the deep learning models decisions to foster trust. The hardware cluster model had a mis-classification rate of 0.597%. The operator cluster model had a mis-classification of 71.4%. The silhouette scores and average distance between cluster centres was higher in the hardware model, implying that the hardware data is more separable which contradicts with the previous results.

Model performances on out of distribution data were compared when screened by the clustering algorithm and when they were run independently. The hardware cluster models had good performance with higher accuracies than the individual models. The operator cluster model had comparatively very poor performance. In theory, the cluster model would act as a barrier to prevent models from being deployed on out of distribution data leading to more reliable results. This however, depends heavily on the performance of the cluster model. In practice, the cluster model did not have

high enough performance to achieve this task. Therefore, it may be better to have a human operator select which model best matches the data instead.

Overall, it is difficult to conclude whether the proposed methods create a proper means to provide an objective measurement of trust and further work may shed more light on this. Two possible approaches are provided. Firstly, the proposed method should be tested with human users to see if the extra information provided allows for more trustworthy models. For example, a study that has a control group using the deep learning models by themselves and a separate group using the proposed methods with clustering information can be conducted to determine if the proposed methods create more trustworthiness. This would give more insight if the proposed methods achieve their goal of creating more objective metrics by which users can choose to trust the results or not.

Secondly, the models themselves can be improved in terms of data and feature engineering. The ultrasound machines used in this thesis vary wildly in age and quality. This will naturally create large differences in the data. Furthermore, only three machines were used. Future work could look into using a larger number of machines that are better quality matched in order to properly validate that hardware differences exist and to better measure to what extent these differences are. On the other hand the operators for this thesis do not have as wide a range of variance in skill. Follow up work should include more operators at even more varied skill levels. Also, the proposed methods could potentially be repeated on patients instead of a phantom such that skill can be better expressed. This is key in separating operator skill as a large part of ultrasonography is using landmarks to identify anatomy and interpreting artifacts. These elements were not present when scanning a phantom as

it is uniform and unlikely to produce artifacts.

The need to expand the data set extends into the idea of combined, or hidden clusters as well. This is based on the findings of the clustering models with an unexpected number of clusters. The silhouette scores of the cluster models with 4 or 5 clusters were comparable to the scores of the expected 2, 3, and 6 cluster models. This could imply that there are potentially other means to cluster the data than the expected clusters. For example, a repeated study with 6 machines may find that there are 3 clusters denoting three types of machines rather than 6 separate machines. This may add increased generalizability of the models and screening method as instead of matching models to specific machines, they can be matched to a type instead. On the other hand, this may also be due to small sample size of similar data. If in that same 6 machine example, 6 clusters have a notably higher silhouette score than the others, it further reinforces differences in specific machines.

Ultrasonography has always been known as an operator dependent modality and thus there is great need for adding objectivity. Deep learning may be an avenue to provide this objectivity, but before deep learning models can be properly adopted, users must be able trust the models. There are many barriers to properly trusting deep learning models such as dataset shift which can occur due to hardware and operator variance that must be addressed first. The challenge of quantifying trust in deep learning is a multifaceted problem with a large body of work ahead. With the continued growth in the use of deep learning algorithms in everyday life, the need for regulation on deep learning is growing as well. Before proper regulations can be put in place, more quantitative metrics must be established to measure deep learning algorithms.

Appendix A

Appendix

A.0.1 Code Listings

```
'''
frames.py
'''
import cv2
import os
#print(os.getcwd())

FILE_PATH = "DATA/RAW/SIEMENS"
videos = os.listdir(os.getcwd() + "/"+FILE_PATH)
print(videos)

def getFrame(NP, sec):
```

```
    vidcap.set(cv2.CAP_PROP_POS_MSEC,sec*1000)
    hasFrames,image = vidcap.read()
    if hasFrames:
        cv2.imwrite(str(NP[0:9])+"_IMAGE_"+str(count)+".png", image, [cv
# save frame as compressionless PNG
    return hasFrames

for video in videos:
        if "DS_Store" in video:
                continue
        vidcap = cv2.VideoCapture(FILE_PATH + video)
        sec = 0
        frameRate = 0.1 #//it will capture image in each 0.1 second
        count=1
        success = getFrame(video,sec)
        while success:
                count = count + 1
                sec = sec + frameRate
                sec = round(sec, 2)
                success = getFrame(video,sec)
```

```
'''
crop.py
'''
from PIL import Image
import os
FILE_PATH = "DATA/RAW/CLARIUS"

frames = os.listdir(os.getcwd() + "/" + FILE_PATH)

for frame in frames:
        if "DS_Store" in frame:
                print(1)
                continue
        # Opens a image in RGB mode
        im = Image.open(FILE_PATH + frame)

        # Setting the points for cropped image
        left = 175
        top = 275
        right = 575
        bottom = 525

        # Cropped image of above dimension
        im1 = im.crop((left, top, right, bottom))
```

```
'''
Load_data.py
'''
Import Libraries
'''
import tensorflow as tf
import pathlib
import os
import numpy as np
import sys
import matplotlib.pyplot as plt

'''
Image Parameters/Batch Settings
'''
# Setting random seeds
RANDOM_SEED = 1234
tf.random.set_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

IMG_HEIGHT = 150
IMG_WIDTH = 150

# Choose Batch Size
```

```
# TF Defaults to 32

BATCH_SIZE = 32

'''
Functions
'''

def load_data_trv(train_dir):
        '''
        Loads training and validation sets.
        Organize the file directory such that folder names are labels.
        Each folder contains all the data for that label.
        Function will create resized, labelled pairs in batches.

        Arguments:
        train_dir : string which is the directory containing the folders

        Returns:
        train_set : tf.data object containing the training set
        val_set : tf.data object containing the validation set

        '''
```

```
    train_dir = pathlib.Path(train_dir)

    train_set = tf.keras.preprocessing.image_dataset_from_directory(
            train_dir,
            #label_mode='binary',
            validation_split=0.2,
            subset = "training",
            seed = 123,
            image_size = (IMG_HEIGHT, IMG_WIDTH),
            batch_size = BATCH_SIZE,
            )

    val_set = tf.keras.preprocessing.image_dataset_from_directory(
            train_dir,
            #label_mode='binary',
            validation_split=0.2,
            subset = "validation",
            seed = 123,
            image_size = (IMG_HEIGHT, IMG_WIDTH),
            batch_size = BATCH_SIZE,
            )

    return train_set, val_set
```

```
def load_data_beastie(data_dir):
    '''
    Loads training and Validation Sets using TF 2.2 for Beastie
    Organize the file directory such that folder names are labels.
    Each folder contains all the data for that label.
    Function will create resized, labelled pairs in batches.

    Arguments:
    train_dir : string which is the directory containing the folders

    Returns:
    train_set : tf.data object containing the training set
    val_set : tf.data object containing the validation set
    '''

    data_dir = pathlib.Path(data_dir)
    image_count = len(list(data_dir.glob('*/*.png')))

    CLASS_NAMES = np.array([item.name for item in data_dir.glob('*')

    list_ds = tf.data.Dataset.list_files(str(data_dir/'*/*.png'))


    val_size = int(image_count * 0.2)
```

```
        train_ds = list_ds.skip(val_size)
        val_ds = list_ds.take(val_size)

        #print(len(list(list_ds.as_numpy_iterator())))

        train_set = train_ds.map(process_path, num_parallel_calls=tf.dat
        val_set = val_ds.map(process_path, num_parallel_calls=tf.data.ex

        train_set = train_set.shuffle(buffer_size=1000).batch(BATCH_SIZE
        val_set = val_set.shuffle(buffer_size=1000).batch(BATCH_SIZE)

        return train_set, val_set

'''
Optional Helper Functions
'''

def get_label(file_path):
        CLASS_NAMES = ["Abnormal","Normal"]
        parts = tf.strings.split(file_path, "/")
        #tf.print(file_path, output_stream=sys.stderr)
        #tf.print(parts, output_stream=sys.stderr)
        #tf.print(parts[-2], output_stream=sys.stderr)
        if parts[-2] == "Abnormal":
```

```
            return(0)
        else:
            return(1)
        #return  tf.argmax(one_hot)

def decode_img(img):
        img = tf.io.decode_png(img, channels=0) #color images
        #img = tf.image.convert_image_dtype(img, tf.float32)
        img = tf.image.resize(img, [IMG_WIDTH, IMG_HEIGHT])
        #convert unit8 tensor to floats in the [0,1]range
        return img

def process_path(file_path):
        #print(file_path)
        label = get_label(file_path)
        img = tf.io.read_file(file_path)
        img = decode_img(img)
        return img, label

def cache_data(train_set, val_set):
        '''
        Caches the training and validation set.
        Run for performance if loading the data multiple times.
```

```
    Arguments:
    train_set : tf.data object containing the training set
    val_set : tf.data object containing the validation set

    Returns:
    train_set : tf.data object containing the training set
    val_set : tf.data object containing the validation set

    '''
    train_set = train_set.cache().prefetch(buffer_size=10)
    val_set = val_set.cache().prefetch(buffer_size=10)

    return train_set, val_set
```

```
'''
create_models.py
'''
'''
Import Libraries
'''

import tensorflow as tf
from tensorflow import keras

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

'''
Constants and Parameters
'''

# Setting random seeds
RANDOM_SEED = 1234
tf.random.set_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

'''
```

```
Create Models
'''

def create_xception_model_A():
        '''
        Creates an Xception based model.

        Arguments:
        N/A

        Returns:
        model : TF model to be compiled and trained.

        '''
        base_model = keras.applications.Xception(
        weights="imagenet", # Load weights pre-trained on ImageNet.
        input_shape=(150, 150, 3), # Adjust input sizes accordingly.
        include_top=False, # Do not include the ImageNet classifier at t
        )

        # Freeze the base_model
        base_model.trainable = False

        # Create new model on top
```

```
inputs = keras.Input(shape=(150, 150, 3)) # Adjust input sizes a

# Normalization Layer
norm_layer = keras.layers.experimental.preprocessing.Normalizati
mean = np.array([127.5] * 3)
var = mean ** 2
# Scale inputs to [-1, +1]
x = norm_layer(inputs)
norm_layer.set_weights([mean, var])

# The base model contains batchnorm layers. We want to keep them
# when we unfreeze the base model for fine-tuning, so we make su
# base_model is running in inference mode here.
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)

# Drop-out Layer
x = keras.layers.Dropout(0.2)(x)

# Binary outputs
outputs = keras.layers.Dense(1)(x)

# Put model together
model = keras.Model(inputs, outputs)
```

```
        # Print summary
        model.summary()

        return model

def create_VGG19_model_A():
        '''
        Creates a VGG19 based model.

        WARNING! TAKES A VERY LONG TIME TO TRAIN.

        Arguments:
        N/A

        Returns:
        model : TF model to be compiled and trained.

        '''
        base_model = keras.applications.VGG19(
        weights="imagenet", # Load weights pre-trained on ImageNet.
        input_shape=(150, 150, 3), # Adjust input sizes accordingly.
        include_top=False, # Do not include the ImageNet classifier at t
        )
```

```
# Freeze the base_model
base_model.trainable = False

# Create new model on top
inputs = keras.Input(shape=(150, 150, 3)) # Adjust input sizes a

# Normalization layer
norm_layer = keras.layers.experimental.preprocessing.Normalizati
mean = np.array([127.5] * 3)
var = mean ** 2
# Scale inputs to [-1, +1]
x = norm_layer(inputs)
norm_layer.set_weights([mean, var])

# The base model contains batchnorm layers. We want to keep them
# when we unfreeze the base model for fine-tuning, so we make su
# base_model is running in inference mode here.
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)

# Drop-out Layer
x = keras.layers.Dropout(0.2)(x)
```

```
        # Binary output
        outputs = keras.layers.Dense(1)(x)

        # Put model together
        model = keras.Model(inputs, outputs)

        # Print summary
        model.summary()

        return model

def create_ResNet50_model_A():
        '''
        Creates a ResNet50 based model.

        WARNING! TAKES A LONG TIME TO TRAIN.

        Arguments:
        N/A

        Returns:
        model : TF model to be compiled and trained.

        '''
```

```
base_model = keras.applications.ResNet50(
weights="imagenet",  # Load weights pre-trained on ImageNet.
input_shape=(150, 150, 3), # Adjust input sizes accordingly.
include_top=False, # Do not include the ImageNet classifier at t
)

# Freeze the base_model
base_model.trainable = False

# Create new model on top
inputs = keras.Input(shape=(150, 150, 3))

# Normalization Layer
norm_layer = keras.layers.experimental.preprocessing.Normalizati
mean = np.array([127.5] * 3)
var = mean ** 2
# Scale inputs to [-1, +1]
x = norm_layer(inputs)
norm_layer.set_weights([mean, var])

# The base model contains batchnorm layers. We want to keep them
# when we unfreeze the base model for fine-tuning, so we make su
# base_model is running in inference mode here.
x = base_model(x, training=False)
```

```
        x = keras.layers.GlobalAveragePooling2D()(x)

        # Drop-out Layer
        x = keras.layers.Dropout(0.2)(x)

        # Binary output
        outputs = keras.layers.Dense(1)(x)

        # Put model together
        model = keras.Model(inputs, outputs)

        # Print summary
        model.summary()

        return model

'''
Model Compiler
'''

def set_metrics_and_compile(model, loss_func, result_metric):
        '''
        Compile model for training according to specified loss function
```

```
        Default is BinaryCrossentropy and accuracy. Uses ADAM optimizer.
                - Will probably remove defaults later. It's included for

        Arguments:
        model : tf.model created by previous functions.
        loss_func : loss function
        result_metric : optimizing metric

        Returns:
        model : tf.model ready to be trained
        '''

        model.compile(
            optimizer=keras.optimizers.Adam(),
            loss=loss_func,
            metrics=result_metric,
        )
        return model

'''
Train Models
'''
def train_model(model, train_dataset, val_dataset, epochs = 10, model_na
        '''
```

```
    Trains model including validation steps
        - Default of 10 epochs
        - "Saved_Models" is directory to store output files in
            - will create directory if not present
        - Generates plot of training history and loss for analys

    Arguments:
    model : model architecture. Generate through previous functions.
    train_dataset : training dataset, generate with load_data.py
    val_dataset : test dataset, generate with load_data.py
    epochs : int value of epochs to train for, default is 10. (Optio
    model_name : name of the model, default is "model" (Optional)

    Returns:
    N/A

    '''

    # Train Model

    fit = model.fit(train_dataset, epochs=epochs, validation_data=va

    # Create directory and store model for future use
    try:
```

```
            os.mkdir("Saved_Models")
            model.save("Saved_Models/"+model_name)
            print("Created directory 'Saved_Models' to store model."
    except FileExistsError:
            print("Directory 'Saved_Models' exists. Model saved to d
            model.save("Saved_Models/"+model_name)

    # Extacting Training information
    # Adjust before running
    accuracy=fit.history['binary_accuracy']
    loss = fit.history['loss']
    val_accuracy = fit.history['val_binary_accuracy']
    val_loss = fit.history['val_loss']
    epochs = range(epochs)

    # Plot training statistics
    plt.plot(epochs, accuracy, label="Training Accuracy")
    plt.plot(epochs, val_accuracy, label="Validation Accuracy")
    plt.title("Training and Validation Accuracy - " + model_name)
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.savefig("accuracy_plots_" + model_name + ".png")
```

```
plt.clf()
plt.plot(epochs,loss, label="Training Loss")
plt.plot(epochs,val_loss, label="Validation Loss")
plt.title("Training and Validation Loss -" + model_name)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.savefig("loss_plots_" + model_name + ".png")
```

```
'''
train_model.py
'''
Import Libraries
'''
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pathlib

# Load previously written libraries
import load_data as ltv
import create_models as cm

'''
Constants and Parameters
'''
# Setting random seeds
RANDOM_SEED = 1234
tf.random.set_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

'''
File Directories
```

```
'''
K_Siemens_DIR = "/Users/calvinzhu/desktop/Final_Models/DATA/Kyla/Siemens
K_Clarius_DIR = "/Users/calvinzhu/desktop/Final_Models/DATA/Kyla/Clarius
K_Sonix_DIR = "/Users/calvinzhu/desktop/Final_Models/DATA/Kyla/Sonix/Tra

C_Siemens_DIR = "/Users/calvinzhu/desktop/Final_Models/DATA/Calvin/Sieme
M_Siemens_DIR = "/Users/calvinzhu/desktop/Final_Models/DATA/Mike/Siemens

'''
Loading Data
'''
train_set, val_set = ltv.load_data_trv(K_Clarius_DIR)
'''
Creating and compiling models
'''
model = cm.create_ResNet50_model_A()
model = cm.set_metrics_and_compile(model, keras.losses.BinaryCrossentropy

'''
Training Model
'''
model_title = "K_C_ResNet_A"

cm.train_model(model, train_set, val_set, 10, model_name=model_title)
```

```
print(model_title)
```

```
'''
load_test_model.py
'''
'''
Import Libraries
'''
import tensorflow as tf
import pathlib
import matplotlib.pyplot as plt
import os
import numpy as np

np.set_printoptions(precision=4)
np.set_printoptions(suppress=True)



'''
Constants, Parameters, Image Settings
'''


# Setting random seeds
RANDOM_SEED = 1234
tf.random.set_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
```

```
'''
Helper Functions
'''
def load_test_set(data_dir,BATCH_SIZE):
        '''
        Loads test set from directory
        Organize the file directory such that folder names are labels.
        Each folder contains all the data for that label.
        Function will create resized, labelled pairs in batches.

        For ease of interpretation, loads one large batch, unshuffled.
        Batch size must be entered according to the number of test samp

        Currently written for interpretability, not performance.
                - Slow execution with large batch sizes
                - Requires argument of number of samples

        Arguments:
        data_dir : string which is the directory containing the folders.
        BATCH_SIZE: number format of total number of test samples

        Returns:
        test_set : tf.data object containing the test set.
```

```
    '''

    test_set = tf.keras.preprocessing.image_dataset_from_directory(
    pathlib.Path(data_dir),
    image_size=(150, 150), # Adjust input sizes accordingly.
    shuffle=False,
    batch_size = BATCH_SIZE
    )

    return test_set

def load_test_beastie(data_dir,BATCH_SIZE):
    '''
    Loads test set from directory for use with TF 2.2
    Organize the file directory such that folder names are labels.
    Each folder contains all the data for that label.
    Function will create resized, labelled pairs in batches.

    For ease of interpretation, loads one large batch, unshuffled.
    Batch size must be entered according to the number of test samp

    Currently written for interpretability, not performance.
            - Slow execution with large batch sizes
```

```
        - Requires argument of number of samples

    Arguments:
    data_dir : string which is the directory containing the folders.
    BATCH_SIZE: number format of total number of test samples

    Returns:
    test_set : tf.data object containing the test set.

    '''

    data_dir = pathlib.Path(data_dir)
    image_count = len(list(data_dir.glob('*/*.png')))

    CLASS_NAMES = np.array([item.name for item in data_dir.glob('*')

    list_ds = tf.data.Dataset.list_files(str(data_dir/'*/*.png'))

    test_set = list_ds.map(process_path, num_parallel_calls=tf.data.

    test_set = test_set.shuffle(buffer_size=1000).batch(BATCH_SIZE)

    return test_set
```

```
def load_model(model):
        '''
        Loads previously saved model created by train_model.py

        Arguments:
        model : string containing name of the saved model.

        Returns:
        loaded_model : tf.model object used to make predictions.
        '''

        model = pathlib.Path(model)
        loaded_model = tf.keras.models.load_model(model)

        # Previously, tf.model.evaluate() was called here to check
        # performance. However, due to output of logits and no
        # ability to adjust threshold, it no longer does this.
        # The accuracy and metrics reported made no sense.
        # May come back to fix this at some point though.

        #loaded_model.evaluate(test_set,verbose=1,batch_size=210)
```

```
    return loaded_model


def extract_labels(test_set):
    '''
    Creates a list of correct labels from a tf.data object

    Arguments:
    test_set : tf.data object from which to grab the labels

    Returns:
    labels : list of labels
    '''
    labels = list(test_set.as_numpy_iterator())[0][1]

    return labels

def make_predictions(model, test_set):
    '''
    Creates a list of predictions
    Runs the test set through the model.

    Arguments:
    model : tf.model loaded by load_model()
```

```
    test_set : tf.data loaded by load_test_set()

    Returns:
    predict_list : list of predictions
    '''

    # Run test set through model
    predictions = model.predict(test_set)
    #print("RAW PREDICTIONS")
    #print(predictions)

    # Model returns logits
    # Use sigmoid function to convert into binary labels
    predictions = tf.nn.sigmoid(predictions)
    predictions = tf.where(predictions < 0.5, 0, 1) # Choose thresho

    # Convert to numpy array for ease of manipulation
    predictions = predictions.numpy().astype(int)

    # Extract values into a list
    predict_list = []
    for value in predictions:
            predict_list.append(value[0])
```

```
        # convert back to array for "prettier" formatting
        predict_list = np.asarray(predict_list)

        #print((predict_list))
        return predict_list

def calc_accuracy(cf):
        '''
        Calculates accuracy from the confusion matrix.

        Accuracy = ( TP + TN ) / ( TP + FP + TN + FN )

        Arguments:
        cf : tf.tensor created by tf.math.confusion_matrix()

        Returns:
        accuracy : float value of accuracy
        '''
        cf = cf.numpy()
        accuracy = ((cf[0][0] + cf[1][1])) / np.sum(cf)
        return accuracy

def calc_precision(cf):
        '''
```

```
        Calculates precision from the confusion matrix.

        Precision = ( TP ) / ( TP + FP )

        Arguments:
        cf : tf.tensor created by tf.math.confusion_matrix()

        Returns:
        accuracy : float value of Precision
        '''
        cf = cf.numpy()
        precision = (cf[0][0]) / (cf[0][0] + cf[1][0])
        return precision

def calc_sensitivity(cf):
        '''
        Calculates sensitivity from the confusion matrix.

        Sensitivity = ( TP ) / ( TP + FN )

        Arguments:
        cf : tf.tensor created by tf.math.confusion_matrix()

        Returns:
```

```
        accuracy : float value of sensitivity
        '''
        cf = cf.numpy()
        sensitivity = (cf[0][0]) / (cf[0][0] + cf[0][1])
        return sensitivity

def calc_specificity(cf):
        '''
        Calculates specificity from the confusion matrix.

        Specificity = ( TN ) / ( FP + TN )

        Arguments:
        cf : tf.tensor created by tf.math.confusion_matrix()

        Returns:
        accuracy : float value of specificity
        '''
        cf = cf.numpy()
        specificity = (cf[1][1]) / (cf[1][0] + cf[1][1])
        return specificity

'''
Helper Functions
```

```
'''

def get_label(file_path):
        CLASS_NAMES = ["Abnormal","Normal"]
        parts = tf.strings.split(file_path, "/")
        tf.print(parts[-2],output_stream=sys.stderr)
        if parts[-2] == "Normal":
                return(1)
        else:
                return(0)
        #return tf.argmax(one_hot)

def decode_img(img):
        img = tf.io.decode_png(img, channels=0)
        #convert unit8 tensor to floats in the [0,1]range
        img = tf.image.resize(img, [150, 150])
        return img

def process_path(file_path):
        label = get_label(file_path)
        img = tf.io.read_file(file_path)
        img = decode_img(img)
        return img, label
```

```
'''
Main
'''

def main():
        BATCH_SIZE = 854+723

        #test_set = load_test_set("/Users/calvinzhu/desktop/Final_Models
        #test_set = load_test_set("/Users/calvinzhu/desktop/Final_Models
        #test_set = load_test_set("/Users/calvinzhu/desktop/Final_Models
        #test_set = load_test_set("/Users/calvinzhu/desktop/Final_Models
        #test_set = load_test_set("/Users/calvinzhu/desktop/Final_Models
        test_set = load_test_set("/Users/calvinzhu/desktop/Final_Models/
        labels = extract_labels(test_set)

        loaded_model = load_model("/Users/calvinzhu/desktop/Final_Models

        predictions = make_predictions(loaded_model, test_set)

        print(labels)
        print(predictions)

        # Create confusion matrix
```

```
cf = tf.math.confusion_matrix(labels,predictions)

print("Confusion Matrix:")
print(cf)

accuracy = calc_accuracy(cf)
precision = calc_precision(cf)
sensitivity = calc_sensitivity(cf)
specificity = calc_specificity(cf)

print()
print("Accuracy:",accuracy)
print("Precision:",precision)
print("Sensitivity:",sensitivity)
print("Specificity:",specificity)
```

```
'''
PM.py
'''
# Import ML Libraries
import tensorflow as tf
import sklearn
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_samples, silhouette_score
from joblib import dump, load

# Import Supporting Libraries
import pathlib
import os
import sys
import shutil

# Import Plotting Libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

'''
```

```
Functions
'''
def load_Sample(img_path):
    '''
    load_Sample()

    ##
    Loads test sample using Tensorflow into a Python Imaging Library(PIL
        - Resizes images into 150x150 pixels for consistency

    ##
    Input Arguements: Directory for image to be loaded
        - Expects string.

    ##
    Returns: PIL object of image.
        - Image is unprocessed.
        - Use show_Sample() on output of this function to display image.
        - Use process_Sample() on output of this function to convert to

    '''
    img=tf.keras.preprocessing.image.load_img(img_path, target_size=(150
    return(img)
```

```
def process_Sample(img,PCA_MODEL_DIR):
    '''
    process_Sample()

    ##
    Extracts features from sample for use in clustering model.
        - Loads PIL object, converts to np.array, then passed into PCA m
    ##
    Input Arguements: Image Sample.
        - Expects PIL Object
        - Generate PIL Object from load_Sample()

    ##
    Returns: Extracted features
        - np.array of features
        - Model was originally set to do predictions on bulk samples, so

    '''

    img_data = tf.keras.preprocessing.image.img_to_array(img)
    img_data = tf.keras.applications.xception.preprocess_input(img_data)
    features = []
    features.append((img_data).flatten())
    pca = load(PCA_MODEL_DIR)
```

```
    pca_features = pca.transform(features)
    return np.array(pca_features)

def pass_Sample_1(pca_features,CLUSTER_MODEL_DIR):
    '''
    pass_Sample_1()

    ##
    Runs extracted features through the clustering model, giving cluster

    ##
    Input Arguements: Extracted features
        - Expects array of arrays of features
        - Generate feature array with process_Sample()

    '''

    cluster = load(CLUSTER_MODEL_DIR)
    skill_level = cluster.predict(pca_features)

    return skill_level


def main():
```

```
PCA_MODEL_DIR = "/Users/calvinzhu/Desktop/Final_Models/SKModels/
CLUSTER_MODEL_DIR = "/Users/calvinzhu/Desktop/Final_Models/SKMo

TEST_SAMPLE_DIR_1 = "/Users/calvinzhu/Desktop/Final_Models/Data/
TEST_SAMPLE_DIR_2 = "/Users/calvinzhu/Desktop/Final_Models/Data/

TEST_SAMPLE_DIR_3 = "/Users/calvinzhu/Desktop/Final_Models/Data/
TEST_SAMPLE_DIR_4 = "/Users/calvinzhu/Desktop/Final_Models/Data/

TEST_SAMPLE_DIR_5 = "/Users/calvinzhu/Desktop/Final_Models/Data/
TEST_SAMPLE_DIR_6 = "/Users/calvinzhu/Desktop/Final_Models/Data/

DIR_1 = pathlib.Path(TEST_SAMPLE_DIR_1)
DIR_2 = pathlib.Path(TEST_SAMPLE_DIR_2)

DIR_3 = pathlib.Path(TEST_SAMPLE_DIR_3)
DIR_4 = pathlib.Path(TEST_SAMPLE_DIR_4)

DIR_5 = pathlib.Path(TEST_SAMPLE_DIR_5)
DIR_6 = pathlib.Path(TEST_SAMPLE_DIR_6)

DIR_1 = list(DIR_1.glob("*.png"))
DIR_2 = list(DIR_2.glob("*.png"))
```

```
DIR_3 = list(DIR_3.glob("*.png"))
DIR_4 = list(DIR_4.glob("*.png"))

DIR_5 = list(DIR_5.glob("*.png"))
DIR_6 = list(DIR_6.glob("*.png"))


#print(DIR_1[0][-1])


for file in DIR_1:
        sample = load_Sample(file)
        test = process_Sample(sample,PCA_MODEL_DIR)
        cluster = pass_Sample_1(test,CLUSTER_MODEL_DIR)

        shutil.copy(file,"/Users/calvinzhu/Desktop/Final_Models/

for file in DIR_2:
        sample = load_Sample(file)
        test = process_Sample(sample,PCA_MODEL_DIR)
        cluster = pass_Sample_1(test,CLUSTER_MODEL_DIR)

        shutil.copy(file,"/Users/calvinzhu/Desktop/Final_Models/
```

```
for file in DIR_3:
        sample = load_Sample(file)
        test = process_Sample(sample,PCA_MODEL_DIR)
        cluster = pass_Sample_1(test,CLUSTER_MODEL_DIR)

        shutil.copy(file,"/Users/calvinzhu/Desktop/Final_Models/

for file in DIR_4:
        sample = load_Sample(file)
        test = process_Sample(sample,PCA_MODEL_DIR)
        cluster = pass_Sample_1(test,CLUSTER_MODEL_DIR)

        shutil.copy(file,"/Users/calvinzhu/Desktop/Final_Models/

for file in DIR_5:
        sample = load_Sample(file)
        test = process_Sample(sample,PCA_MODEL_DIR)
        cluster = pass_Sample_1(test,CLUSTER_MODEL_DIR)

        shutil.copy(file,"/Users/calvinzhu/Desktop/Final_Models/

for file in DIR_6:
        sample = load_Sample(file)
```

```
test = process_Sample(sample,PCA_MODEL_DIR)
cluster = pass_Sample_1(test,CLUSTER_MODEL_DIR)

shutil.copy(file,"/Users/calvinzhu/Desktop/Final_Models/
```

```
'''
cluster.py
'''
# Import ML Libraries
import tensorflow as tf
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_samples, silhouette_score
from joblib import dump, load

# Import Supporting Libraries
import pathlib
import os

# Import Plotting Libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Setting Constants/Seeds/GPUS
RANDOM_SEED = 1234
tf.random.set_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
```

```
os.environ["CUDA_VISIBLE_DEVICES"]="2"

# Version Checks
print("TF")
print(tf.__version__)
print()

print(tf.config.experimental.list_physical_devices())
print()

#print(os.environ)

print(1)

def convert_img_to_array(img_path):
    '''
    Converts raw images to numpy arrays

    Inputs

    Outputs
    '''
    img = tf.keras.preprocessing.image.load_img(img_path, target_size=(1
```

```
    img_data = tf.keras.preprocessing.image.img_to_array(img)
    img_data = tf.keras.applications.xception.preprocess_input(img_data)
    return img_data

dir_1 = "data/Kyla/Siemens/Train/Normal"
dir_1 = pathlib.Path(dir_1)
dir_2 = "data/Kyla/Sonix/Train/Normal"
dir_2 = pathlib.Path(dir_2)
dir_3 = "data/Kyla/Clarius/Train/Normal"
dir_3 = pathlib.Path(dir_3)

dir_4 = "data/Kyla/Siemens/Train/Abnormal"
dir_4 = pathlib.Path(dir_4)
dir_5 = "data/Kyla/Sonix/Train/Abnormal"
dir_5 = pathlib.Path(dir_5)
dir_6 = "data/Kyla/Clarius/Train/Abnormal"
dir_6 = pathlib.Path(dir_6)

feature_list = []

list_1 = list(dir_1.glob('*.png'))
list_2 = list(dir_2.glob('*.png'))
list_3 = list(dir_3.glob('*png'))
```

```
list_4 = list(dir_1.glob('*.png'))
list_5 = list(dir_2.glob('*.png'))
list_6 = list(dir_3.glob('*png'))

list_of_files = list_1 + list_2 + list_3 + list_4 + list_5 + list_6

# Run PCA

for name in range(len(list_of_files)):
    features = convert_img_to_array(list_of_files[name])
    np_features = np.array(features)
    feature_list.append(np_features.flatten())

X = np.array(feature_list)

pca_data = PCA(0.95)
X = pca_data.fit_transform(X)

dump(pca_data, 'pca_HDWR.joblib')

print(pca_data.explained_variance_ratio_)
#print(pca_data.singular_values_)
print(sum(pca_data.explained_variance_ratio_))
print(len(pca_data.explained_variance_ratio_))
```

```
range_n_clusters = [2,3,4,5,6]

for n_clusters in range_n_clusters:
    # Create a subplot with 1 row and 2 columns
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)

    # The 1st subplot is the silhouette plot
    ax1.set_xlim([-0.1, 1])
    ax1.set_ylim([0, len(X) + (n_clusters + 1) * 10])

    clusterer = KMeans(n_clusters=n_clusters, random_state=0)
    cluster_labels = clusterer.fit_predict(X)

    # The silhouette_score gives the average value for all the samples.
    # This gives a perspective into the density and separation of the fo
    # clusters
    silhouette_avg = silhouette_score(X, cluster_labels)
    print("For n_clusters =", n_clusters,
          "The average silhouette_score is :", silhouette_avg)

    # Compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X, cluster_labels)
```

```
y_lower = 10
for i in range(n_clusters):
    # Aggregate the silhouette scores for samples belonging to
    # cluster i, and sort them
    ith_cluster_silhouette_values = \
        sample_silhouette_values[cluster_labels == i]

    ith_cluster_silhouette_values.sort()

    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i

    color = cm.nipy_spectral(float(i) / n_clusters)
    ax1.fill_betweenx(np.arange(y_lower, y_upper),
                      0, ith_cluster_silhouette_values,
                      facecolor=color, edgecolor=color, alpha=0.7)

    # Label the silhouette plots with their cluster numbers at the m
    ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

    # Compute the new y_lower for next plot
    y_lower = y_upper + 10  # 10 for the 0 samples
```

```
ax1.set_title("The silhouette plot for the various clusters.")
ax1.set_xlabel("The silhouette coefficient values")
ax1.set_ylabel("Cluster label")

# The vertical line for average silhouette score of all the values
ax1.axvline(x=silhouette_avg, color="red", linestyle="--")

ax1.set_yticks([])  # Clear the yaxis labels / ticks
ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

# 2nd Plot showing the actual clusters formed
colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
ax2.scatter(X[:, 0], X[:, 1], marker='.', s=30, lw=0, alpha=0.7,
            c=colors, edgecolor='k')

# Labeling the clusters
centers = clusterer.cluster_centers_
# Draw white circles at cluster centers
ax2.scatter(centers[:, 0], centers[:, 1], marker='o',
            c="white", alpha=1, s=200, edgecolor='k')

for i, c in enumerate(centers):
    ax2.scatter(c[0], c[1], marker='$%d$' % i, alpha=1,
                s=50, edgecolor='k')
```

```
    ax2.set_title("The visualization of the clustered data.")
    ax2.set_xlabel("Feature space for the 1st feature")
    ax2.set_ylabel("Feature space for the 2nd feature")

    plt.suptitle(("Silhouette analysis for KMeans clustering"
                  " with n_clusters = %d" % n_clusters),
                 fontsize=14, fontweight='bold')
    plt.savefig("PCA Reduced Normal Hardware - Silhouette Analysis for K

plt.show()
```

A.0.2 Other Cluster Results Figures

This section includes figures showing the silhouette analysis and clusters of running K-Means to generate other numbers of clusters rather than the 3 required for use in screening for the proposed method. These include Figures A.1 to A.8

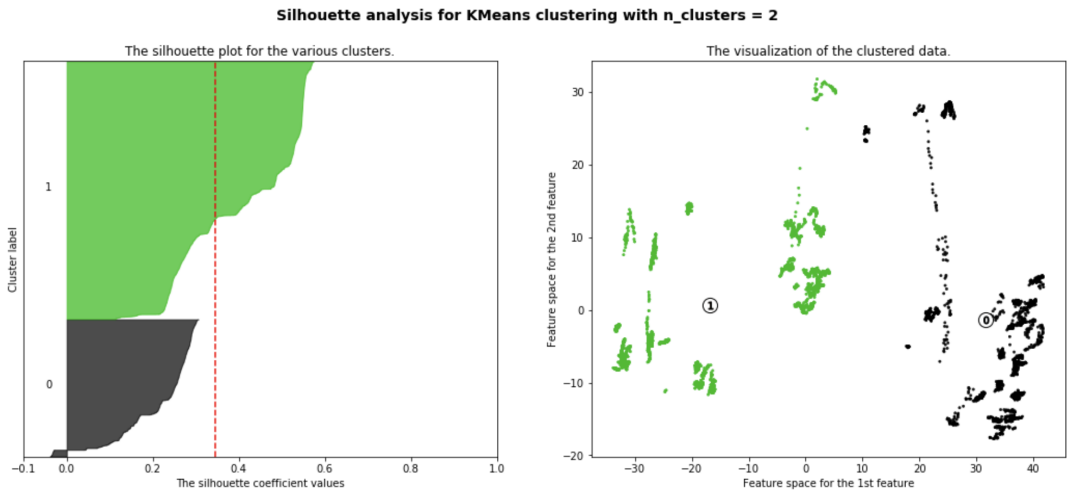

Figure A.1: Silhouette analysis and visual representation of the clusters running K-Means to generate 2 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

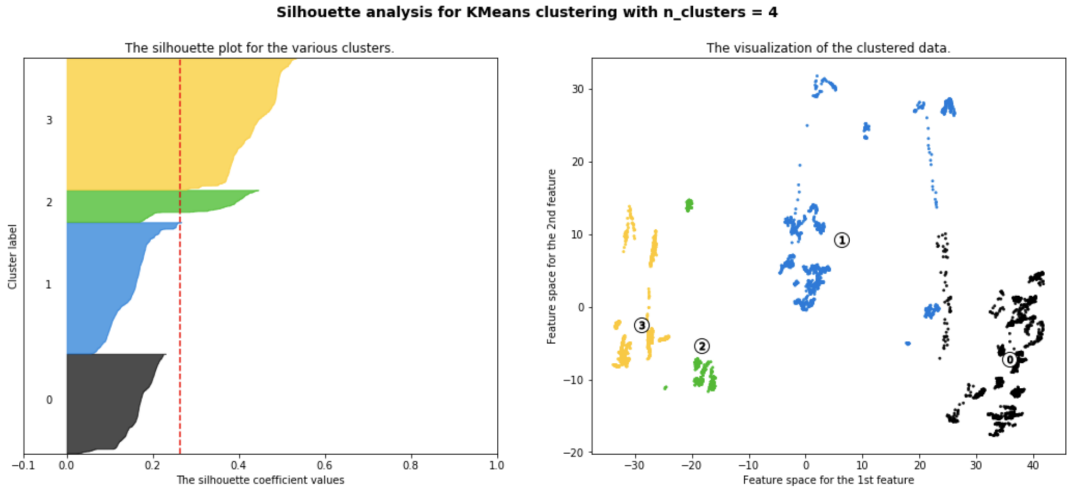

Figure A.2: Silhouette analysis and visual representation of the clusters running K-Means to generate 3 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

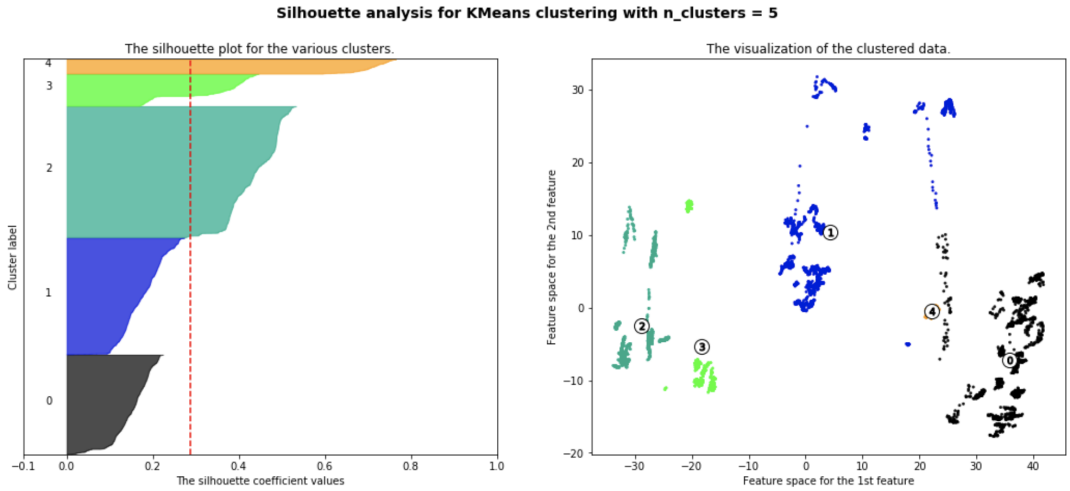

Figure A.3: Silhouette analysis and visual representation of the clusters running K-Means to generate 3 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

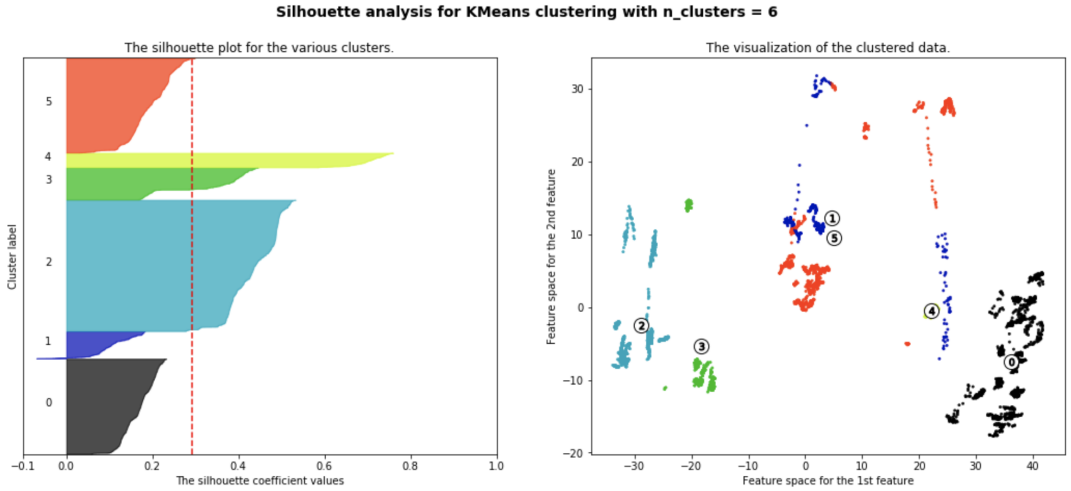

Figure A.4: Silhouette analysis and visual representation of the clusters running K-Means to generate 3 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

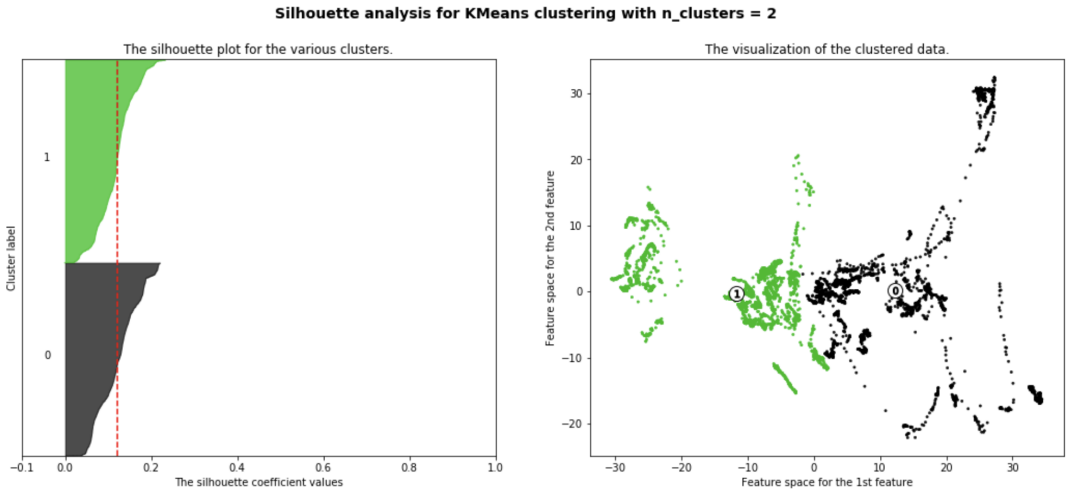

Figure A.5: Silhouette analysis and visual representation of the clusters running K-Means to generate 2 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

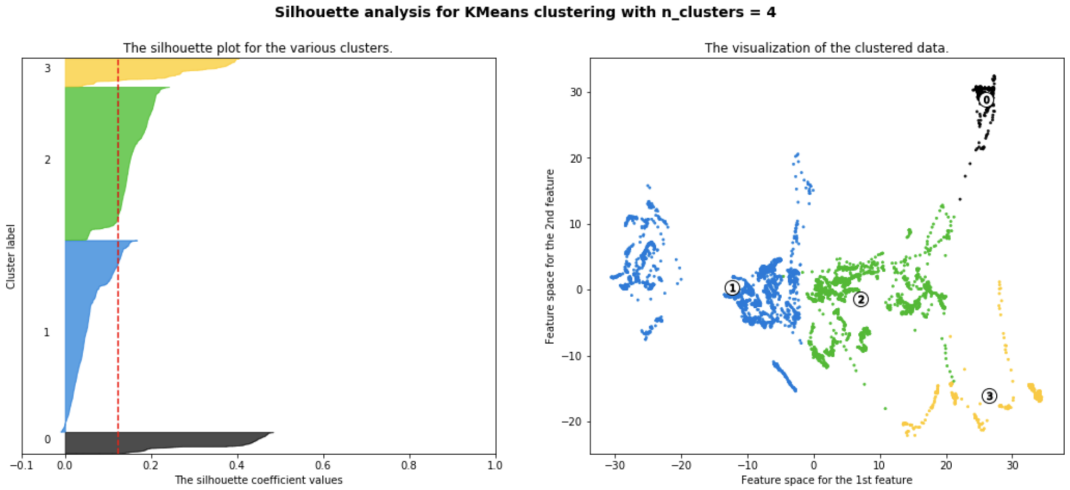

Figure A.6: Silhouette analysis and visual representation of the clusters running K-Means to generate 4 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

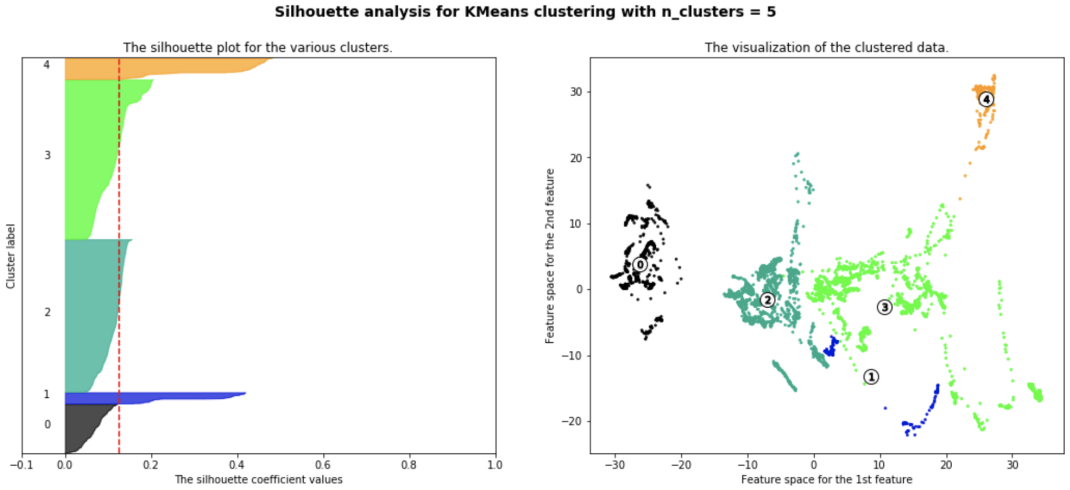

Figure A.7: Silhouette analysis and visual representation of the clusters running K-Means to generate 5 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

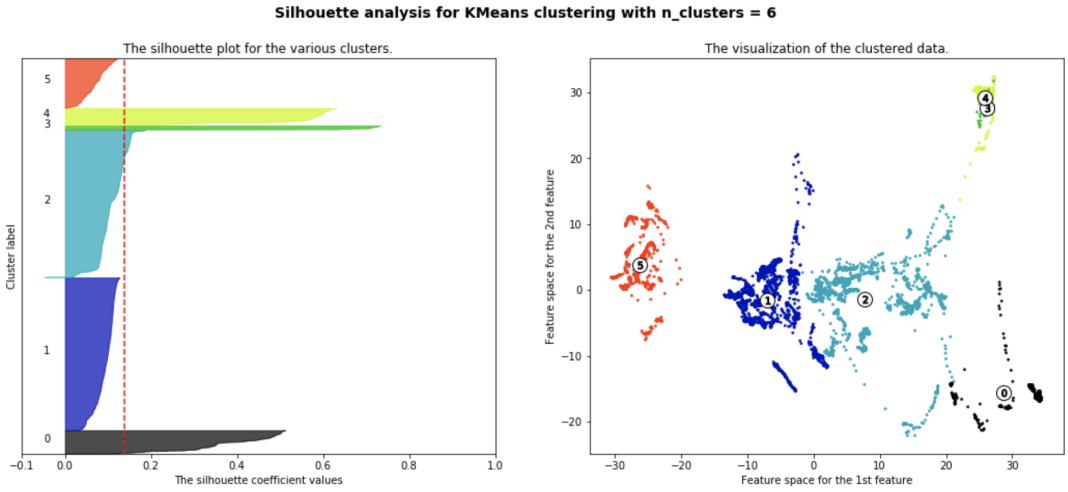

Figure A.8: Silhouette analysis and visual representation of the clusters running K-Means to generate 6 clusters of operators. On the left is a graphical representation of the silhouette score of each individual sample in each cluster. The red dotted line denotes the average silhouette score. On the right is a visual representation of the clustered data. Note that only the first two principle components are plotted.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Akkus, Z., Cai, J., Boonrod, A., Zeinoddini, A., Weston, A. D., Philbrick, K. A., and Erickson, B. J. (2019). A survey of deep-learning applications in ultrasound: Artificial intelligence–powered ultrasound for improving clinical workflow. *Journal of the American College of Radiology*, **16**(9, Part B), 1318–1328. Special Issue: Quality and Data Science.

Aldrich, J. E. (2007). Basic physics of ultrasound imaging. *Critical Care Medicine*, **35**(5).

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Carovac, A., Smajlovic, F., and Junuzovic, D. (2011). Application of ultrasound in medicine. *Acta informatica medica : AIM : journal of the Society for Medical Informatics of Bosnia & Herzegovina : casopis Drustva za medicinsku informatiku BiH*, **19**(3), 168–171. 23408755[pmid].

Chan, V. and Perlas, A. (2011). *Basics of Ultrasound Imaging*, pages 13–19. Springer New York, New York, NY.

Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions.

Clark, A. (2015). Pillow (pil fork) documentation.

Géron, A. (2019). *Hands-on machine learning with Scikit-Learn and TENSORFLOW: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, **521**(7553), 436–444.

Liu, S., Wang, Y., Yang, X., Lei, B., Liu, L., Li, S. X., Ni, D., and Wang, T. (2019). Deep learning in medical ultrasound analysis: A review. *Engineering*, **5**(2), 261–275.

Moreno-Torres, J. G., Raeder, T., Alaiz-Rodríguez, R., Chawla, N. V., and Herrera, F. (2012). A unifying view on dataset shift in classification. *Pattern Recognition*, **45**(1), 521–530.

Park, S. H. (2021). Artificial intelligence for ultrasonography: unique opportunities and challenges. *Ultrasonography (Seoul, Korea)*, **40**(1), 3–6. 33227844[pmid].

Park, V. Y., Han, K., Seong, Y. K., Park, M. H., Kim, E.-K., Moon, H. J., Yoon, J. H., and Kwak, J. Y. (2019). Diagnosis of thyroid nodules: Performance of a deep learning convolutional neural network model vs. radiologists. *Scientific Reports*, **9**(1), 17843.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12**, 2825–2830.

Pinto, A., Pinto, F., Faggian, A., Rubini, G., Caranci, F., Macarini, L., Genovese, E. A., and Brunese, L. (2013). Sources of error in emergency ultrasonography. *Critical Ultrasound Journal*, **5**(1), S1.

Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, **20**, 53–65.

Schmidt, P. and Biessmann, F. (2019). Quantifying interpretability and trust in machine learning systems.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre,

L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, **550**(7676), 354–359.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition.

Toreini, E., Aitken, M., Coopamootoo, K., Elliott, K., Zelaya, C. G., and van Moorsel, A. (2020). The relationship between trust in ai and trustworthy machine learning technologies. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, FAT* '20, page 272–283, New York, NY, USA. Association for Computing Machinery.