Cognitive Impairment Anomaly Detection Using Metric Learning on Triplet Networks

by

Dania Maryam Waqar

**A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Engineering
(Robotics Engineering)
in the University of Michigan-Dearborn
2023**

Acknowledgments

Before all else, I would like to thank my Creator for blessing me with this opportunity, for guiding me in making ML my niche, and for granting me success in completing the project within the given timeframe.

I would like to express my immense gratitude towards my supervisor, Professor Yi Lu Murphey, for her endless support and encouragement throughout my project. I am glad to have taken two other classes with her at the University of Michigan-Dearborn, so I can really say that a significant proportion of my degree has been through her mentorship. A huge thanks also goes to my committee members for their endless support and feedback on my work. I would also like to thank my research groupmates, Dr. Song Wang and Tian Qin, for their words of guidance through the extra Zoom calls and the very many emails.

My family is a big reason I am who I am, and I am at where I am at. For this reason, I would like to dedicate my thesis to my parents: my *Ammi*, Mrs. Tabassum, and my *Abbu*, Dr. Waqar Asrar. Their endless love, support, and reassurance (from literally across the world) is what has motivated and given me the confidence to complete my thesis today, and for that (and so much more), I cannot thank them enough. After all the video calls, after all the vent sessions, after all the smiles, after all the tears: this is for you.

Words are not enough to thank my brother, Mohammad Ehaab, and my sister, Fanoos Qadri, both of whom constantly supported me by listening, by being there when I needed them, and by always having my back, not only when I was at my best, but also when I was at my worst. A special thanks to my sister, Saniya Heeba, for providing me with questionable (but surprisingly useful) therapy sessions, for listening to my rants through our Zoom “work” calls, and for lifting me up when I needed it most (*hsop* here we come!). And of course, I cannot go without the mention of my two little cheerleaders: Ayedin and Behraam, who do everything from writing “You can do it!” on my wall to giving me hugs to brighten up my day.

Thank you all for helping me achieve my dreams. Thank you all for keeping me sane. Thank you all for being my comfort blankets. I hope (insert music) *ki mene aap ka naam kiya hai roshan.*

Table of Contents

List of Tables

List of Figures

Abstract

Machine learning is used for many application purposes; some of the common ones being classification, regression, and anomaly detection. This project aims to deal with anomaly detection using metric learning of the data collected by NHATS. This data is usually used to predict cognitive impairment in individuals. This dataset is called the Clock Drawing Test (CDT) and contains drawings from people of different levels of cognitive impairment which have been classified into different classes by a human coder. Since these drawings have been judged by a coder, it is prone to variations from person to person, or even when performed by the same person and hence contain anomalies. We have developed a triplet network based deep metric learning system architecture for anomaly detection and have implemented it to find first find anomalies in a given class, and then find the correct class for the anomalous images. We have compared two different architectures (ResNet101 and EfficientNetB7) to find the best base for transfer learning, and two different metrics (Root Mean Square (RMS) and Euclidean distances) to find the optimal metric for our use case. We have also compared its performance with the more commonly used Siamese Networks to find the most effective metric learning solution for this dataset. Our results showed best performance with a combination of ResNet101 and the RMS metric, outperforming the current Siamese network on this dataset. This report is divided as follows; Chapter 1 introduces the problem followed by descriptions on the method we will use, Chapter 2 provides an extensive literature review, Chapter 3 describes the methodology that is to be followed, Chapter 4 describes the various results obtained, and Chapter 5 closes with conclusion and future work.

Chapter 1 Introduction

Section 1.1 Defining the Problem

The Clock Drawing Test (CDT) is a popular tool used to screen patients for the early onset of, or current presence of, dementia and other forms of cognitive impairment. It is a simple tool to implement, with patients being asked to draw a clock with hands pointing at a specific time. The quality of the drawing is judged based on the position of the numbers and the hands of the clock. This drawing is judged by a human coder and placed into different classes based on the following description.

0: Not recognizable as a clock

1: Severely distorted depiction

2: Moderately distorted depiction

3: Mildly distorted depiction

4: Reasonably accurate depiction

5: Accurate depiction of a clock

Since these descriptions are subjective, the classification of clock images in the dataset is prone to human errors, called coder effects. These are systematic differences in the way the coder understands and implements the description of the drawings. They can vary based on coder to coder, and even on the mood and time of day of the same coder. These variations can introduce inconsistencies in the scores, which can be a big concern in longitudinal studies. Further, descriptions of the in-between classes (classes 2, 3 and 4) are only slightly different in their descriptions and can easily be interpreted in different ways by a coder. In this thesis, we aim to minimize the effects of these variations in the CDT dataset provided to us by NHATS. By performing anomaly detection, we aim to remove the inconsistent and improperly placed clock drawings in a specific dataset and then reclassify them into their correct class. We aim to perform this anomaly detection using metric learning and triplet networks, which will be defined in the next few subchapters.

Section 1.2 Introduction to Metric Learning

Subconsciously, our human mind performs similarity computations when subject to tasks of classification or identification. Depending on the application, humans pay attention to different details at different times (for example, focusing on frown lines for emotion recognition, or facial features for person identification). For machines to perform this sort of task, metric learning in some form is used for similarity detection which can be seen in algorithms such as nearest neighbor classification and kernel methods.

Metric learning is the process of teaching a machine which pairs or triplets are similar or dissimilar so that it can make appropriate judgments of unseen input based on its known side information. This side information is usually of the must-link/ cannot-link constraint format (which constitute of similar and dissimilar pairs), or of relative constraints (consisting of triplets in which two are similar to each other and dissimilar from the third).

There are many options of metrics when it comes to metric learning [1]. These are briefly described below:

a. Minkowski distances: These are also known by L_p norms (Eq 1).

$$d_p(x, x') = \left\|x - x'\right\|_p = (\sum_{i=1}^{d}|x_i - x_i'|^p)^{\frac{1}{p}} \qquad \text{Equation 1}$$

i. When p = 1, we obtain the formula for Manhattan distance (Eq 2).

$$d_{man}(x, x') = ||x - x'||_1 = \sum_{i=1}^{d} |x_i - x_i'| \qquad \text{Equation 2}$$

ii. When p = 2, we obtain the famous Euclidean distance (Eq 3).

$$d_{euc}(x, x') = ||x - x'||_2 = (\sum_{i=1}^{d} |x_i - x'_i|^2)^{1/2} = \sqrt{(x - x')^T(x - x')} \qquad \text{Equation 3}$$

iii. When p → ∞, we obtain the Chebyshev distance (Eq 4).

$$d_{che}(x, x') = ||x - x'||_\infty = \max_i |x_i - x'_i| \qquad \text{Equation 4}$$

b. Mahalanobis distances: This refers to the quadratic distance (Eq 5).

$$d_M(x, x') = \sqrt{(x - x')^T M (x - x')} \qquad \text{Equation 5}$$

Here, **M** ε S_+^d, where S_+^d is a symmetric positive semi-definite (PSD) d x d matrix. When **M** is equivalent to the identity matrix, the Mahalanobis distance simplifies to the Euclidean distance. Alternatively, **M** is equivalent to $\mathbf{L}^\mathbf{T}\mathbf{L}$, where **L** ε $R^{k \times d}$ and k is the rank of **M**. This gives the distance equation to be as follows. If the rank of **M** is low, this means that dimensionality reduction will occur as points would be linearly projected into a lower dimension, leading to more concise depiction of information and more efficient computations of high dimensional data (Eq 6).

$$d_M(x, x') = \sqrt{(x - x')^T M (x - x')}$$
$$= \sqrt{(x - x')^T L^T L (x - x')} = \sqrt{(Lx - Lx')^T (Lx - Lx')} \qquad \text{Equation 6}$$

c. Cosine Similarity: Used frequently in text and image retrievals, this method calculates the cosine of the angle between two inputs (Eq 7).

$$S_{cos}(x, x') = \frac{x^T x'}{||x||_2 ||x'||_2} \qquad \text{Equation 7}$$

d. Bilinear Similarity: This method is similar to cosine similarities, except it does not normalize the similarity by the L_2-norms of the inputs and is instead multiplied by a factor of **M** (Eq 8). This method is highly efficient for methods which do have sparsely populated data, and it can be used for similarity calculations between heterogeneous dimensionalities of data.

$$S_M(x, x') = x^T M x' \qquad \text{Equation 8}$$

e. Linear Kernel: This simple equation refers to the dot product of each data point (Eq 9).

$$K_{lin}(x, x') = \langle x, x' \rangle = x^T x' \qquad \text{Equation 9}$$

f. Polynomial Kernels: This kernel can be defined up to various degrees, p. The corresponding output space consists of all monomials of a degree up to p (Eq 10).

$$K_{poly}(x, x') = (\langle x, x' \rangle + 1)^p \qquad \text{Equation 10}$$

g. Gaussian RBF Kernel: This kernel function is popularly used with the output space being infinite dimensional (Eq 11). Here, $\boldsymbol{\sigma}^2 > 0$ is the width parameter.

$$K_{rbf}(x, x') = \exp\left(-\frac{\left\|x-x'\right\|_2^2}{2\sigma^2}\right) \qquad \text{Equation 11}$$

h. Histogram Distances: This distance is based on the statistical chi-squared test and is used in computer vision and text processing (Eq 12).

$$\chi^2(x, x') = \frac{1}{2}\sum_{i=1}^{d}\frac{(x_i - x'_i)^2}{x_i + x'_i} \qquad \text{Equation 12}$$

Metrics are also used widely in the realm of machine learning to provide quantitative measures of performance. For example, the prediction produced during classification using k-Nearest Neighbors is a metric used to find the instance that belongs to the class with the highest majority. Another example is clustering in models such as K-Means and K-Medoids where the metric used is the distance between interclass and intraclass points. This is the distance that this paper will also aim to minimize.

Next, we delve into metric learning properties. There are five major metric learning properties which are explained in detail below.

a. Learning paradigm.

1. Fully supervised: In this method, each training instance is associated with its respective label which are used to form pairs/triplets constraints based on their location in the feature space.
2. Weakly supervised: In this method, training labels are not fed into the model. Instead, side information constraints are fed into the model, providing a computationally cheaper alternative.
3. Semi-supervised: In this method, the model is trained with either fully supervision or weak supervision while simultaneously using a large number of instances which are not associated with any side information constraints. This extra data is usually used to ensure that overfitting does not occur.

b. Form of metric.

1. Linear metrics: These metrics usually do not hold as much information, but they can easily converge to the global minimum with a low chance of overfitting occurring.

2. Nonlinear metrics: These metrics can be used to measure variations that are nonlinear in nature but have a higher chance of overfitting and can often converge to local minima instead.
3. Local metrics: A combination of metrics are used to model the feature space better to solve more convoluted problems. Unfortunately, these metrics can overfit easily due to the large number of parameters they learn.

c. Scalability: Due to the growing availability of information, the machine learning model needs to be able to adapt to Big Data by scaling it to d dimensions.

d. Optimality of the solution: Ideally, it should be ensured that the final answer reached by the model is the global minimum, and the parameters should be adjusted accordingly to achieve this goal.

e. Dimensionality reduction: Another result of Big Data is high-dimensional computations. Due to this, features are often transformed into another more condensed lower-dimension space which results in quicker computations of distance.

Section 1.3 Background on Siamese Networks

A popular network used with anomaly detection is the Siamese network. This is often used in situations where the amount of data is low. Siamese networks are known, as their name implies, for being constructed from identical subnetworks. Each subnetwork consists of the same number of parameters, and the same values of weights. The aim of the model is to learn a similarity function which detects the input pair's affinity [2].

Siamese networks are advantageous for many reasons. Firstly, an abundance of data is not needed to produce high quality predictions. The presence of just a couple of images in each class is sufficient for the network to identify images [2]. This also means that it can be used for situations where the number of output classes is very large but the number of samples per class is small [2,3]. Secondly, the model can efficiently be combined with other classifiers to produce better predictions on average. Lastly, Siamese networks are popular in learning the semantic similarity of classes that are close together in the feature space (where metric learning for anomaly detection can play an important role). This also means that they can easily be extended to add new classes when need be [3].

On the other hand, Siamese networks may take a longer time to train as they involve pairs of a quadratic nature to learn from. Further, the final output of Siamese networks is different from traditional neural networks; instead of probabilities, the distance between each class is produced instead.

There are two main types of loss functions associated with Siamese networks. Firstly, triplet loss is commonly used where three inputs are sent into the network, a positive label, a negative label, and an anchor label. The aim of this loss function is to minimize the loss function to ensure the positive label moves closer to the anchor, and the negative label moves further away. This is the transformation of a Siamese network into a triplet network. The second (and more popular) type of loss is the contrastive loss [4]. In this function, the loss learns embeddings between two inputs with a low Euclidean distance, and two inputs with a large one [2].

Section 1.4 Siamese Network Architecture

As stated earlier, the Siamese network is known for its name as it consists of two identical and symmetric networks which are linked together by an energy function. This energy function is used to calculate the difference between the highest-level features from each network output. Since identical parameters are used in both networks in a method called weight tying, it ensures that images from the same class are mapped onto similar positions in the feature space since the same energy function is applied to each one. This ensures that a very similar output is generated [4].

Section 1.4.1 Introduction to Neural Networks

Since Siamese networks are based off Convolutional Neural Networks (CNNs), we will have a brief introduction into neural networks first.

Neural networks are based on a biological background: our brain. Our brain is considered as a complex, yet systematically arranged, system which processes a range of information from our senses efficiently and rationally. The human brain is based on neurons and synapses, which have been translated into elements and connections in an artificial neural network (ANN) [5].

The simplest type of ANN is the perceptron. This is a network consisting of nodes v, connection weights w, and a bias, b. These three combine together to form the output, as seen in Eq 13, after passing through an activation function, f.

$$y = f(\sum_{i=1}^{D} v_i w_i + b) \qquad \text{Equation 13}$$

When extending this function to many layers, the resulting network is called a multilayer perceptron. This is done by including many hidden layers between the first and last layer where the units are fully connected amongst one another, but their weights are independently connected to be constrained within each layer.

As a network trains, it learns a set of weights that will give the desired output each time an input is set. To learn, the network tries to minimize a given error function. This function is usually set as a cross-entropy cost function (Eq 14) for classification of K-classes, where t_n is the label for every input x_n to obtain the output y_n.

$$E = -\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk} \qquad \text{Equation 14}$$

To minimize this error function, a gradient descent algorithm is utilized to ensure the parameters are updated. One way to do this is through error backpropagation. The main idea of backpropagation is to ensure that the errors are propagated from the output back to the input layer to estimate the error at each layer. This error is then subtracted from the original weights to allow the weights to converge to their optimal value. This gradient descent algorithm can be swapped with other, more optimized, versions such as stochastic gradient descent (which may be used for learning on larger scales) or mini-batch gradient descent (which calculates parameters based off a smaller batch of samples at a time) [5].

Moving on to Convolutional Neural Networks (CNNs), these networks consist of a sequence of convolutional and pooling layers, fully connected layers (also known as dense layers), and an output layer defined by a function (a popular function is the SoftMax function). A traditional setup is shown in Figure 1.

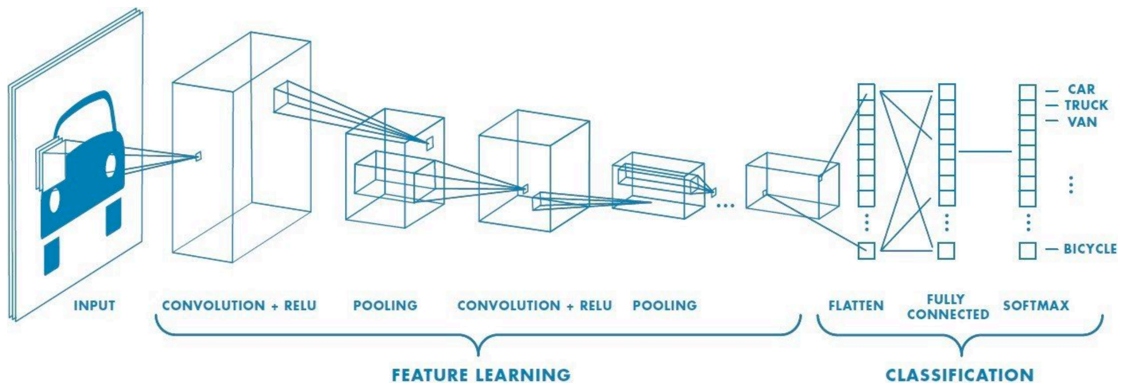


Figure 1: CNNs From [6]

Each of these layers are explained below [2, 3, 7].

a. Convolutional layer: This is the main computational component of the neural network. It is responsible for the dot products between the array of learnable parameters in the kernel matrix and the receptive field matrix. The kernel is known to be more in-depth, but spatially smaller, than an image. The kernel moves across the image at different spatial positions with its stride. This generates a two-dimensional depiction of the original image which is called its activation map (Figure 2).

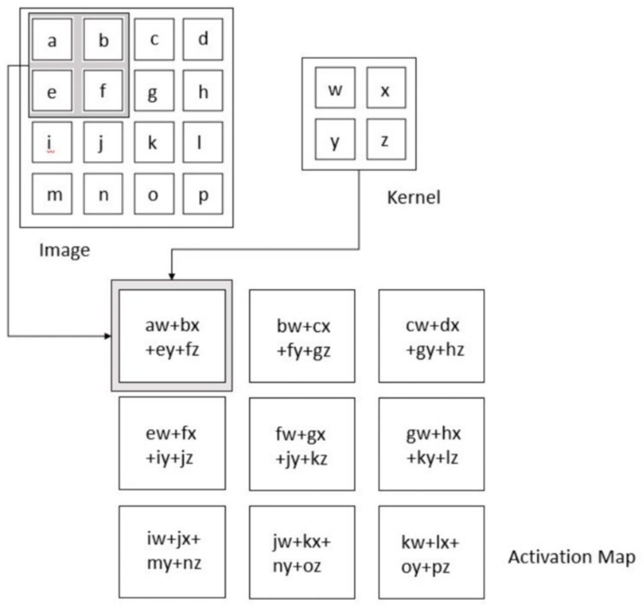


Figure 2: Activation Map From [7]

b. Pooling layer: This layer is used to reduce the representation's spatial size using a statistical method to summarize the nearby outputs. This helps to further reduce the number of parameters in the weight matrix, leading to a lower number of computations. The most popular type is the max pooling process, which picks out the maximum output from the neighborhood. This layer helps to provide some invariance in translation, i.e. the network becomes more robust in recognizing the object in the image irrespective of its position in the frame.

b. Fully Connected layer: In this layer, each neuron from the previous layer and succeeding layer are completely connected with one another. It is used for mapping the representation from the input to the output.

Convolutional neural networks are advantageous due to their sparse interaction, sharing of parameters, and equivariant representation. Traditionally, neural networks used to have each input unit performing matrix multiplication with each output unit. CNNs removed the need for this by using sparse interactions using its kernel, helping to reduce the dimensionality. This helps to improve the efficiency of the network, while ensuring lower memory is used. Further, trivial neural networks performed one-time use of parameters in their weight matrix. Mathematically, however, it is inferred that the function used to calculate the feature of one spatial point could also be beneficial for another point. This imposes a constraint on the neurons to have the same set of weights, i.e. the same parameters. Lastly, due to this sharing of parameters, if the input is changed in any way, the output would also change in the same way, which is known as equivariance to translation [8].

Siamese networks are very similar in their architecture to CNNs. They contain the same convolutional and pooling layers which perform the same function. However, the one main difference is that they do not contain a SoftMax layer. This means that the architecture halts at the fully connected layers (two layers, since it is a Siamese network). The maximum value from each layer is chosen, and the difference between them is calculated. This is output using a sigmoid function to obtain a value between 0 and 1. A simple representation of a Siamese network is shown in Figure 3 [3].

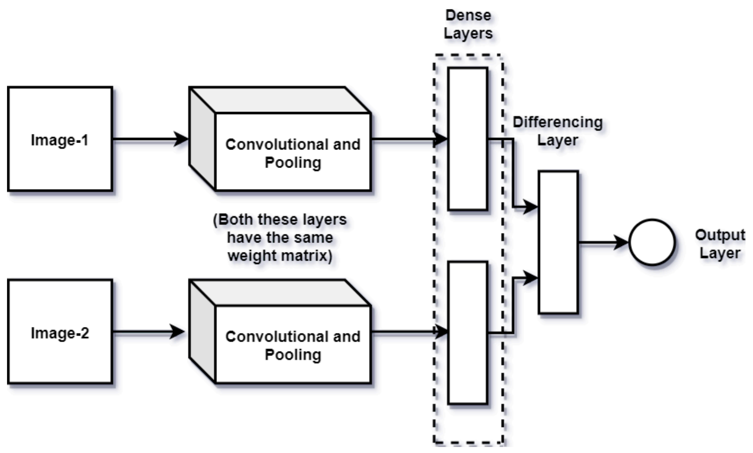

Dense Layers
Image-1
Convolutional and Pooling
Differencing Layer
(Both these layers have the same weight matrix)
Output Layer
Image-2
Convolutional and Pooling

Figure 3: Siamese Network Representation From [3]

Section 1.5 Siamese Networks as Triplet Networks

There are many similarities between triplet networks and Siamese networks. Triplet networks are considered as an advancement of the Siamese network, with three inputs needed instead of two to create a similarity metric. The anchor, positive input, and negative input are chosen such that the anchor and positive sample are from the same class, while the negative sample is from a different class. The triplet network can then learn both positive and negative distances concurrently while using the different sequences of training data to avoid overfitting.

As mentioned before, the network learns to minimize the distance between the anchor and positive sample and increase the distance between the anchor and the negative sample, such that the differences between these two distances should converge to a predefined hyperparameter called alpha. This is because once the values are equal to one another, the difference between the two would be zero, leading to no further modifications of the network's parameters. A difficulty of using this loss function is in the production of the labeled data, especially the anchor and positive sample as they are lower in number. To prevent overfitting from occurring, the ratio of anchor-positive samples and anchor-negative samples would have to be controlled [9].

Section 1.5.1 Triplet Loss Architecture

Since triplet loss is an extension of the Siamese network, it is correspondingly also an extension of CNNs. The architecture is again designed to learn similarity and dissimilarity notions through parallel networks. For this method however, triplets must be built from the training data. The format for these triplets is <anchor, positive, negative> where, as mentioned earlier, the anchor and positive images are from the same classes, while the anchor and negative images are from different classes. The loss function used here is of key importance (Eq 15) to ensure that the distance between the target and positive sample is lower than the distance between the target and negative sample.

$$L\,(a, p, n) = \max(0, D(a, p) - D(a, n) + \text{margin}) \qquad \text{Equation 15}$$

where D (x, y): distance between vector representations of x and y [10].

The architecture rests on having three identical networks which share the same network parameters (i.e. the same weights). The same types of layers (convolutional layers, fully connected layers, and pooling layers) are used to construct each network. The last layer should have n number of neurons, where n is the dimension of the final vector representation (Figure 4) [11].

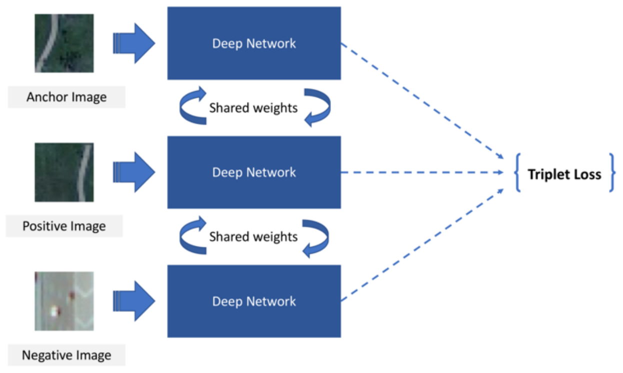


Figure 4: Triplet Network Representation From [11]

Chapter 2 Literature Review

Section 2.1 Introduction

In the next section, we will outline past work which has been done in outlier detection. Along with different approaches to anomaly detection, we find the different datasets that have been used for evaluation, the different metrics used, and the different applications for metric learning.

Section 2.2 Related Work

As mentioned earlier, metric learning is a growing area of interest. Our particular focus in the field of metric learning is in anomaly detection. As described by [12], deep neural networks are useful in object detection with great certainty but can often face difficulties in categorizing unknown objects. They may still classify these unknown objects into a known class with reasonably high accuracy. Due to this, a new field of study called out-of-distribution detection is used for anomaly detection; one which aims to ensure that any unknown objects produce no answer as output instead of being classified into one of the predefined categories.

The contributions of [12] are to be able to perform both novelty and anomaly detection to produce comparable results in the scientific community and apply this method onto the Tsinghua traffic sign dataset for quantitative results. In doing so, they first defined the terms novelty and anomaly. Novelty refers to any sample that is overall different from its dataset, but its appearance bears some resemblance to the existing samples. Anomalous data refers to those samples that are completely unrelated to the existing samples, which is also the definition we will use in our project.

Their method proposed a new loss function using contrastive loss (Eq 16) which was adapted to include out-of-distribution data. They introduced a new label z which is flagged 0 if both images are from out-of-distribution, and 1 if they are not.

$$L(x_1, x_2, y; W) = \frac{1}{2}(1-y)zD_w^2 + \frac{1}{2}yz(\max(0, m - D_w))^2 \qquad \text{Equation 16}$$

They used a minibatch technique to sample image pairs instead of Siamese networks as it is a more efficient technique. These feed into the loss layer, the output of which is used to backpropagate the gradient. Since a Siamese network is not used, the final network only consists of one single copy of weights. The results showed a comparison between CC-AG [13] and ODIN [14] with their novel approach (ODM) for SVHN and CIFAR-10 datasets. Both CC-AG and ODM used out-of-distribution samples (using data from other datasets or Gaussian noise) during training and have near perfect performance for detecting out-of-distribution data, while ODIN has lower results as it does not incorporate any out-of-distribution samples during training.

They tested their model on the Tsinghua traffic sign dataset [15] which consists of 100,000 Tencent Street view panoramas which contain 30,000 traffic sign samples in various lighting and weather conditions. They compared ODM and ODIN for anomaly detection, i.e., to detect which input is not a traffic sign. The results showed that both methods performed very well for in-distribution detection, achieving 99.00% and 98.29% accuracies respectively. However, ODM performed better for anomaly detection due to its introduction of anomalous samples (Gaussian noise and background patches from images from the same dataset) during training, producing 99.99% AUPR in out-of-distribution detection. The advantages of their method were in the use of a minibatch method to propose their new loss function. This function helped to produce very high accuracies for detecting out-of-distribution data; over 98% AUROC. However, the disadvantage of their method was the lower classification accuracy of their in-distribution data. Further, novelty detection in their method occurred by pushing novel items away from the in-distribution data, but damage can occur to the detection if the novel sample is pushed far enough away.

Next, in the work done by [16], we see anomaly detection being done using metric learning and an optimal clustering algorithm using a different loss function. They followed the method as used by the Patch-SVDD algorithm [17] which uses hierarchical encoding to capture input patch features, with the addition of a new loss function from clustering and metric learning (Figure 5).

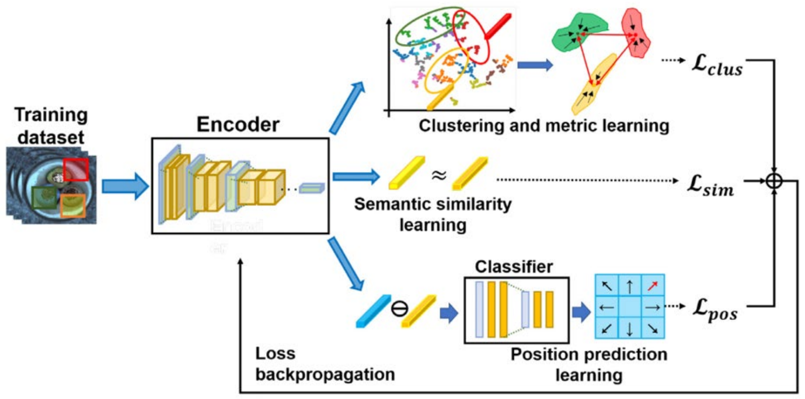


Figure 5: Metric Learning Flow By [16]

Semantic similarity learning occurs to attenuate the Euclidean distance between two randomly selected adjacent patches (L_{sim}). This gives the loss function shown in Eq 17.

$$L = \mathrm{Cross-entropy}(y, C(f(p1), f(p2))) \qquad \text{Equation 17}$$

Next, position prediction learning is done by the encoder to predict the comparative positions of randomly selected patches (L_{pos}). This gives the loss equation shown in Eq 18.

$$L = \mathrm{CrossEntropy}(c, \mathrm{Mk}\,(f(p1) - f(p2))) \qquad \text{Equation 18}$$

The novel loss function proposed uses clustering to ensure only the most relevant and information-filled features are used in anomaly detection. They used hard example mining, center loss (L_{met} focuses on creating compact clusters, and L_{var} tries to reduce any cluster overlaps), and anomaly synthesis (L_{syn}) to create their final new loss function which includes the semantic similarity loss and position prediction loss (Eq 19).

$$\mathcal{L} = \mathcal{L}_{\mathrm{pos}} + \lambda_1\mathcal{L}_{\mathrm{sim}} + (1 - \lambda_1)(\mathcal{L}_{\mathrm{met}} + \mathcal{L}_{\mathrm{var}}) + \mathcal{L}_{\mathrm{syn}} \qquad \text{Equation 19}$$

Clustering was done using the K-means clustering algorithm and anomalous samples were introduced during training through images containing motion, blur, stain, and crack. The final training process is as shown in Figure 6, where patch features are extracted from each training image before clustering is applied. Metric learning is performed to calculate the loss function, after which the clustering results are retained and used during inference. An anomaly map is built by searching for the nearest cluster for each feature using the k-NN algorithm, for which the distance obtained corresponds to the anomaly score.

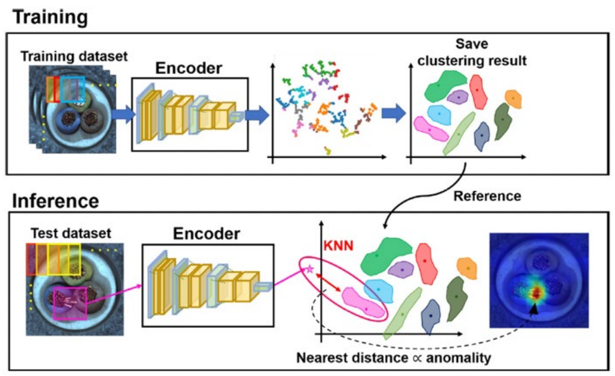


Figure 6: Training and Inference Process From [16]

The results obtained used the MVTec-AD dataset which holds 5000 high resolution industrial inspection images from 15 object and texture classes. The AUROC metric is used for comparison between different algorithms (Stud-Teach, Patch-SVDD and the Proposed algorithm) on the 15 classes to show that the proposed method surpasses the other methods in most classes, especially the object classes. The results also showed improved processing speed as well as enhanced anomaly detection and segmentation accuracy. The reason for their valuable results lay in extracting only relevant features using clustering and metric learning. Henceforth, the advantages of this method was in the boosted processing speed by 10-35% and the enhanced anomaly detection accuracy (average of 94.4%) and segmentation accuracy (average of 95.6%).

Next, in the work done by [18], a different application of anomaly detection is shown, namely intrusion detection systems (IDS). IDS consists of both anomaly and misuse detection, of which only anomaly detection can detect zero-day attacks. Traditionally, network anomaly detection systems (NADSs) use distance metrics between feature vectors of connections to diagnose attacks. However, these metrics are not as efficient in this application due to the heterogeneous nature of network connections. The main idea for this paper is to propose a new data-driven metric of distance for NADS [18] involving a new transformation matrix to map features into a different feature space while preserving local neighborhood information. The type of metric learning used is linear, which derives similarity/dissimilarity constraints from samples and uses feature weighting for feature space transformation. The overall inference process is outlined in Figure 7.

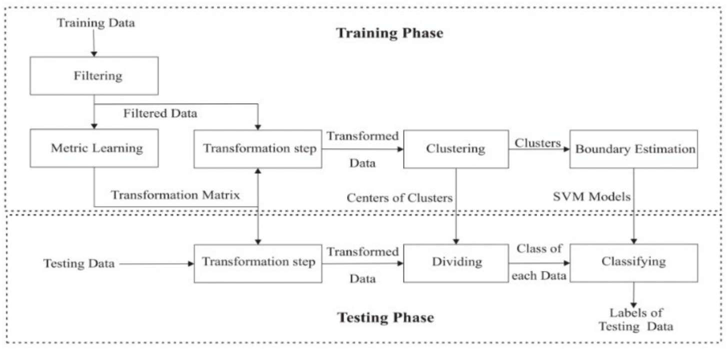


Figure 7: Training and Inference Process From [18]

In this pipeline, filtering occurs first to separate normal and compromised connections. Metric learning ensues to extract all similarity/dissimilarity constraints and to define the optimization problem. The transformation step is responsible for downsizing the filtered data into a smaller dimensional feature space. Clustering using the k-Nearest Neighbor algorithm is done to collect like-samples together. Lastly, boundary estimation occurs using a one-class support vector machine to define the border of each class to ensure that all samples within the boundary are from the same class of normal traffic behavior. In the testing phase, data is transformed using the same transformation matrix to map feature vectors into a lower-dimensional feature space. These are then assigned to a nearest cluster, which, if within the boundary, is considered normal traffic behavior.

To test their pipeline, the Kyoto 2006+ dataset was used as it is created from real-world network traffic data. There are three main classes defined: normal (no attack), attack (known attack), and unknown attacks. They chose 5000 randomly selected pairs of normal filtered data to compute the similarity/dissimilarity constraints. The results showed an improved NADS performance as it surpasses the efficiency of Euclidean distance based NADS in terms of accuracy of normal and false positive rate, but as par results in accuracy of attack and false negative rate.

An advantage of this method is the enhanced security of the system as shown in the better detection of anomalous network traffic data, leading to better protection of the system. Further, another advantage lies in the ability to compute with high dimensional datasets as the transformation matrix is able to convert n-dimensional data into smaller d-dimensional data. Moreover, the usage of an appropriate data-driven distance metric allows for the comparison of heterogeneous features, which is more apt when dealing with network traffic data due to its wide range of features in each network packet. However, a disadvantage lies in the fact that only default parameters had been used in the ML algorithms; no hyperparameter tuning had been done in an attempt to improve performance.

Next, in the work done by [19], we see a new application of anomaly detection, namely in industrial factories. Due to quality control, industrial applications of anomaly detection are of vital importance. This ensures that products are not only in excellent condition but would lead to minimal error in their area of operation. Examples are of the healthcare or automotive industry, where any minor error could result in loss of life or funds. Flaws in detecting faulty products could occur during inspection through human error, and automating this process could lead to better detection and removal of failures. Since effort has already been put in to reduce errors in product inspection, there exists little non-defective samples to create an efficient dataset for training networks. Instead, it is more effective to shift the objective from classifying defects to identifying anomalies as this would require no defective samples during training. Deep neural networks with the ability to learn similarity metrics were used in triplet learning to learn patterns between different image classes. From the three images fed into the network, two were from the same class, and one was from a different class. The Euclidean distance between the 3 feature points are calculated and learnt by the network to assign shorter distances between the two features from the same class. These two distances are then connected to form one vector which is normalized and trained against the target vector (which signifies which of the three inputs were from the same class and which were not). This is shown in Figure 8.

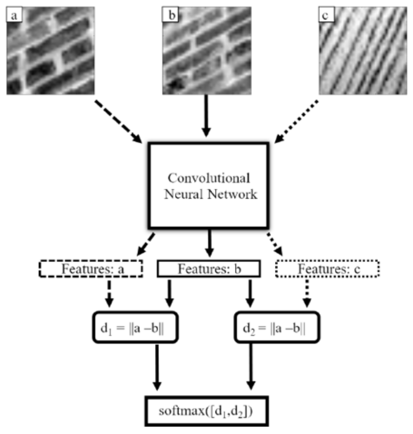


Figure 8: Triplet Network Structure From [19]

The work was based on that done by [20], which used an MSE loss function on one network which shares its weights. This enables all the weights to be updated simultaneously (Figure 9).

$$Loss(d_+, d_-) = \|(d_+, d_- - 1)\|_2^2 = const \cdot d_+^2$$

where

$$d_+ = \frac{e^{\|Net(x)-Net(x^+)\|_2}}{e^{\|Net(x)-Net(x^+)\|_2} + e^{\|Net(x)-Net(x^-)\|_2}}$$

and

$$d_- = \frac{e^{\|Net(x)-Net(x^-)\|_2}}{e^{\|Net(x)-Net(x^+)\|_2} + e^{\|Net(x)-Net(x^-)\|_2}} .$$

We note that $Loss(d_+, d_-) \to 0$ iff $\frac{\|Net(x)-Net(x^+)\|}{\|Net(x)-Net(x^-)\|} \to 0$, which is the required objective.

Figure 9: MSE Loss Function From [20]

They tested their triplet network on three datasets; the Kylberg Texture Dataset (consisting of 28 feature classes with 160 images each), the Public DAGM surface defect class (consisting of 6 classes with 1000 non-defective and 150 defective samples each) and the CIFAR100 dataset (consisting of 6000 images of various everyday object classes distributed in 100 classes). This dataset was used to test the assumption that introducing more complex similarities between objects leads to better texture anomaly detection. They augmented the data through random resizing and by introducing patches of Gaussian noise to create out-of-class samples.

The model was tested on the six classes from the DAGM dataset with samples that were not used during training. The competition DAGM set (which consisted of four novel classes with 2000 non-defective and 300 defective samples each) were used to detect the model's performance on unknown data. The results showed high AUC (area under curve) results for known surface classes (0.99, 1.0, 1.0, 1.0 best results for classes 1, 3, 5, and 6) compared to the unknown classes. The introduction of CIFAR100 was proven to improve surface detection, especially for unknown classes. The use of Euclidean distance was questioned as it did not fit well with high-dimensional data, and for future work to investigate different measures such as Manhattan distances and alternative CNN architectures instead. The advantage of this method lies in the use of triplet networks with the MSE loss function which produced high AUC results (83% on average). However, a disadvantage was that the Euclidean distance may not have been the best fit in this scenario of high-dimensional data; and that not every class performs very well; for example class 2, class 4 and class 10 (with best performances of 48%, 66%, and 63% respectively).

Next, [21] proposed a new loss function for deep metric learning, one that uses angular relationships instead of distance relationships. The reason for this lies in the lack of sensitivity of the distance metric to changes in scale and the use of only two points in the computation which lead to substandard convergence. Instead, the angular metric is a more robust metric as it is rotation and scale invariant. Secondly, three points instead of two are needed to create the loss function, leading to additional local structure collected and better convergence overall.

The traditional triplet loss approach, which is based on the distance metric, is based on the hinge loss shown in Eq 20 that needs to be minimized. This helps to increase the distance between the anchor and negative sample as opposed to the positive sample.

$$\mathrm{l_{tri}}(\mathcal{T}) = [\| \mathrm{x_a} - \mathrm{x_p} \|^2 - \| \mathrm{x_a} - \mathrm{x_n} \|^2 + \mathrm{m}]_+ \qquad \text{Equation 20}$$

The loss function is optimized to get the rate of change for each triplet sample (Figure 10).

$$\frac{\partial l_{tri}(\mathcal{T})}{\partial \mathbf{x}_n} = 2(\mathbf{x}_a - \mathbf{x}_n),$$
$$\frac{\partial l_{tri}(\mathcal{T})}{\partial \mathbf{x}_p} = 2(\mathbf{x}_p - \mathbf{x}_a),$$
$$\frac{\partial l_{tri}(\mathcal{T})}{\partial \mathbf{x}_a} = 2(\mathbf{x}_n - \mathbf{x}_p),$$

Figure 10: Weight Change for Each Triplet Sample [21]

Diagrammatically, the flow of the triplet loss, and the new proposed angular loss, are shown in Figure 11 and 12.

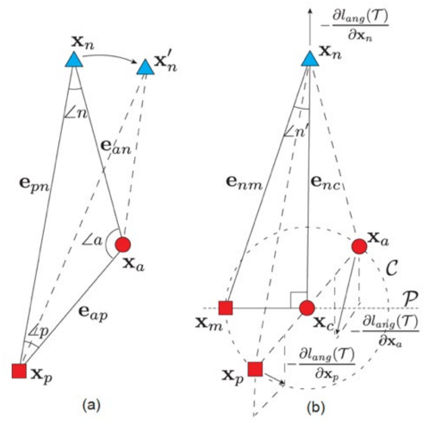


Figure 11: Proposed Angular Loss [21]

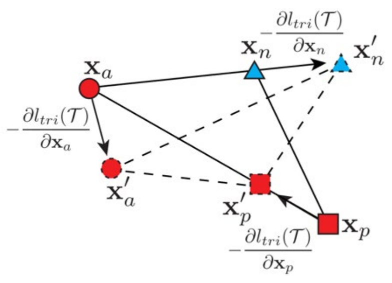

$-\frac{\partial l_{tri}(\mathcal{T})}{\partial \mathbf{x}_n}$
$\mathbf{x}_n$
$\mathbf{x}'_n$
$\mathbf{x}_a$
$-\frac{\partial l_{tri}(\mathcal{T})}{\partial \mathbf{x}_a}$
$\mathbf{x}'_a$
$\mathbf{x}'_p$
$\mathbf{x}_p$
$-\frac{\partial l_{tri}(\mathcal{T})}{\partial \mathbf{x}_p}$

Figure 12: Movement of Triplets From [21]

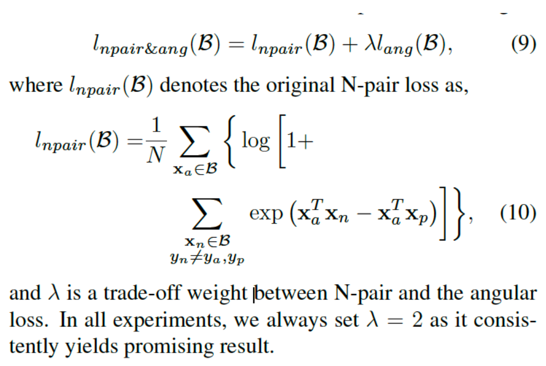
As seen in Figure 11, the proposed angular loss is also intended to increase the relative distance between the anchor and negative point. To find the angle that needs to be minimized, a new triangle needs to form. A new circle with circumference C forms at the center of line $\mathbf{e}_{ap}$. A hyperplane is created to intersect the circle at two nodes, one of which is called x_m. The angle to minimize is now ∠mcn, which is constrained with the parameter. This alpha value is found through experimentation to select the parameter that leads to the most optimal performance (Eq 21).

$$\tan \angle n' = \frac{\|x_m - x_c\|}{\|x_n - x_c\|} = \frac{\|x_a - x_p\|}{2\|x_n - x_c\|} \leq \tan \alpha \qquad \text{Equation 21}$$

This allows the new hinge loss to be as seen in Eq 22. However, this function can be replaced with its smooth upper bound (Eq 23). Due to the generalizability nature of this loss, it can be combined with the traditional distance metric loss in order to improve the performance. The final angular loss proposed is shown in Figure 13.

$$l_{ang}(\mathcal{T}) = [\| x_a - x_p \|^2 - 4\tan^2\alpha \| x_n - x_c \|^2]_+ \qquad \text{Equation 22}$$

$$\log(\exp(y1)) + \exp(y2) \geq \max(y1, y2) \qquad \text{Equation 23}$$



$$l_{npair\&ang}(\mathcal{B}) = l_{npair}(\mathcal{B}) + \lambda l_{ang}(\mathcal{B}), \qquad (9)$$

where $l_{npair}(\mathcal{B})$ denotes the original N-pair loss as,

$$l_{npair}(\mathcal{B}) = \frac{1}{N} \sum_{\mathbf{x}_a \in \mathcal{B}} \left\{ \log \left[1 + \sum_{\substack{\mathbf{x}_n \in \mathcal{B} \\ y_n \neq y_a, y_p}} \exp\left(\mathbf{x}_a^T \mathbf{x}_n - \mathbf{x}_a^T \mathbf{x}_p\right) \right] \right\}, \qquad (10)$$

and λ is a trade-off weight between N-pair and the angular loss. In all experiments, we always set $\lambda = 2$ as it consistently yields promising result.

Figure 13: Angular Loss Function From [21]

To evaluate their new loss function, they performed two tasks: image retrieval and image clustering. This is done on three public datasets. The first one is CUB-200-2011, which consists of 11,788 images of 200 bird species, of which approximately half were used for training, and half for testing. The next dataset used was the Stanford Car dataset, consisting of 16,185 automobile images from 196 classes, of which 98 classes were used for training and the remaining for testing. The third dataset used was the Stanford Online Products dataset, which consists of 120,053 images of 22,634 classes, of which the first 11,318 classes were used for training and the remainder was used for testing.

The results were measured using F1 and NMI (normalized mutual information) metrics. They compared both triplet losses; the traditional loss (T-I) and the optimized version that uses N-pair sampling (T-II). They experimented with the addition of Lifted Structure (LS: referring to when pairwise distances in the batch are lifted to the pairwise distance matrix) and N-pair Loss. The results showed a substantial improvement with NL, with the best combination of NL & AL (angular and N-pair loss combined) of 29.4% (second-best), 32.2%, and 29.9% F1 score in the three datasets (CUB-200-2011, Stanford Car, and Stanford Online Products respectively). NL & AL were able to find more discriminative features to classify the correct image from the correct class in image retrieval, with four successful same class matches compared to 1 to 3 correct matches by NL. In the clustering task, t-SNE was used along with the k-means clustering algorithm to generate the feature embeddings. The novel method was correctly able to establish compact feature mappings which conserve the semantic similarities. Final work included experimenting on the image retrieval task to find the best value, which was found to be between 36and 55. Although a disadvantage of the method lies in the more complex encoding method used, the final metrics are more robust to rotations and scale variations, leading to better overall convergence.

In the work done by [22], we can see the applications of metric learning on face identification. The aim of facial identification is to understand if two face images are of the same person or not. This can be difficult due to variations in posture, accessories, lighting, and background, amongst others. As seen in the work done by [22] a comparison is done between a logistic discriminatory method using metric learning (LDML) and a nearest neighbor approach called MkNN. These are performed on the Labeled Faces in the Wild dataset for both restricted and unrestricted (a novel contribution) settings, achieving results of 79.3% and 87.5% accuracy respectively. The advantages of their method lies in the fact that they were the first ones to publish results in an unrestricted setting of image pairs, and that they obtained higher than the state-of-the-art accuracies of 79.3% and 87.5%. On the other hand, the MkNN classifier was computationally expensive as it uses training data that has been labeled and uses all pairs implicitly. This means that the nearest neighbors need to be found for a very large amount of data that has been labeled, leading to higher computational complexity.

The work done by [13] deals with the deep neural network being overconfident in its guesses of whether a particular test sample is from in-distribution (i.e. from within the training dataset) or out-of-distribution (i.e. a sample that is significantly different from the data the classifier has dealt with). To solve this problem, they have proposed a new method for training which introduces two new terms added to the loss function; the first encourages the classifier to be less confident on its out-of-distribution guesses, while the second allows for more efficient training samples to be generated. The original method to improve the classifier lies in the addition of out-of-distribution samples during training by minimizing the Kullback-Leibler (KL) divergence. But these samples may be difficult to obtain prior to training; instead, they have solved this issue by using a generative adversarial network (GAN) to create new samples based on the output. This new GAN helps to create borderline training samples. They then constructed a scheme that minimizes both the classifier's loss and the new additional GAN's loss one by one; each improving the next as training proceeds. They tested their method using a backbone of AlexNet and VGGNet on datasets such as CIFAR, SVHN, and ImageNet for image classification. The following metrics were used as their threshold-based detectors: accuracy, the true negative at the 95th percentile of the true positive rate, AUROC, and AUPR. They observed that out-of-distribution images that resemble closer to in-distribution aided in the network's performance in detecting out-of-distribution samples. Their results showed that when SVHN is used as the in-distribution data, the classifier has a dramatic increase in classification performance. But, when CIFAR-10 is used instead, the confidence loss does not improve. This problem can be solved by using a proposed joint confidence loss. Their results also showed better performance of the GAN in out-of-distribution detection for image classification tasks compared to when using cross entropy loss when testing it on the MNIST dataset. They also found that using the original GAN helps to improve detection performance, but their proposed GAN outperforms the original. The advantages of their work were in the manipulation of the original loss function to allow the classifier to be less confident in its predictions, and the introduction of the GAN to create more samples while training goes on. They were also able to use different base deep convolutional neural networks to find the best architecture and evaluated their performance on multiple datasets to achieve better out-of-distribution detection without compromising the original accuracy for classification. However, a disadvantage of their method is that they only tested their method on image classification tasks, and only using neural networks, even though it can be applied to other tasks using regression or Bayesian algorithms.

In the work by [23], we see a new approach to the out-of-distribution detection problem. They argued that SoftMax probabilities are likely to be higher for instances that are correctly classified, compared to those that are from out-of-distribution. Henceforth, analyzing the probabilities of predictions of in-distribution samples should aid in classifying a sample as erroneous or normal. Besides this, their work proposes evaluation metrics for anomaly detection by testing using known images from datasets and naturally distorting it. This would lead to known out-of-distribution examples. They have proposed using the AUROC (Area Under the Receiver Operating Characteristic curve) as the performance metric for out-of-distribution detection. This is further explained in Section 3.5. In analyzing their model, they have used three datasets: MNIST, CIFAR-10, and CIFAR-100. When testing, they set all the testing examples as in-distribution examples, while out-of-distribution examples are set to realistic looking images and noise. For example, when using CIFAR-10 and CIFAR-100, the out-of-distribution samples are taken from the Scene UNderstanding dataset (SUN), consisting of 397 different scenes. An advantage of their approach is that it allows for a simple and effective method in classifying in- and out-of-distribution samples while only dealing with probabilities from the softmax layer. Further, they demonstrated their detection system on numerous architectures and datasets. The disadvantages lie in the fact that the probability may, in isolation, be misleading. There are times when they are not as effective and can misclassify samples.

Finally, in the work done by [24], we see that they have proposed a novel version of the triplet loss function for facial recognition and clustering. The final model proved to not only be highly efficient (taking only 128 bytes per face), but also produced considerably high accuracy with 99.63% accuracy on the Labeled Faces in the Wild dataset, and 95.12% accuracy on the YouTube Faces DB dataset (a dataset of videos). Their work was able to cut down the error rate by 30% compared to earlier publications. Further, they were the first to propose the idea of harmonic embeddings and a harmonic version of the triplet loss function; this used a new online negative mining approach to produce increasingly difficult triplets to train the model, as they noticed that the training triplets produced were not all very difficult for the model to learn. They used a stochastic gradient descent algorithm with an initial learning rate of 0.05, and an alpha value of 0.2. They compared the use of two core architectures: the Zeilar&Fergus network (22 layers deep and 140 million parameters), and the Inception network (a much more lightweight model consisting of between 6.6 – 7.5 million parameters, depending on the version used). Their metric in the triplet loss function was the Euclidean distance, which will also be used as one of our metrics in this project, as explained in Section 3.

Section 2.3 Summary of Related Work

Table 1: Summary of Related Work

Paper	Idea/Proposed Method	Advantages	Disadvantages
[12]	Novelty and Anomaly Detection	Proposed a new loss function achieving very high accuracies (over 98% AUROC for out-distribution data)	Metric learning has lower classification accuracy of in-distribution data Can damage novelty detection by pushing it farther away from embedding space of in-distribution data.
[16]	Patch-based anomaly detection	Improved processing speed by 10-35% Enhanced anomaly detection (avg: 94.4%) and segmentation accuracy (avg: 95.6%)	Relatively small dataset MVTec-AD dataset used Does not perform as well in texture classes by itself; needs addition of center loss with hard mining and anomaly synthesis
[18]	Anomaly detection of IDS	New data-driven metric of distance that deals with heterogeneous features Better security of system Conversion into lower dimensional features	Only used default parameters for ML algorithms: no hyperparameter tuning

[19]	Anomaly detection in industrial factories	Used triplet network with MSE loss function High AUC results (83% on average)	Euclidean distance may not fit this use case with high-dimensional data Not all classes perform equally well
[21]	Deep metric learning using angular relationships	Metrics are most robust since they are rotation and scale invariant Better convergence	More complex encoding method
[22]	Face identification using metric learning	First to publish results in unrestricted setting of image pairs Higher accuracy than state-of-the-art (79.3% for restricted and 87.5% for unrestricted settings)	One of the methods (MkNN) is computationally expensive Difficult to detect faces with changes in pose.
[13]	Confidence-calibration for out-of-distribution classifiers	Manipulation of the original loss function to allow the classifier to be less confident in its predictions Introduction of the GAN to create more samples while training goes on. Using different base deep CNNs to find the best architecture Evaluated performance on multiple datasets (TNR improved by 25.1% on	Only tested their method on image classification tasks, and only using neural networks, even though it can be applied to other tasks using regression or Bayesian algorithms.

		CIFAR-10 and 52.6% on SVHN)	
[23]	Detecting Misclassified out-of-distribution samples using SoftMax distributions	Simple and effective method in classifying in- and out-of-distribution samples while only dealing with probabilities from the SoftMax layer. Demonstrated detection system on numerous architectures and datasets.	Probability may, in isolation, be misleading. because there are times when they are not as effective and can misclassify samples.
[24]	FaceNet	Proposed a new method for online triplet mining Proposed harmonic embeddings Proposed harmonig triplet loss Cut error rate by 30% Efficient final model	Did not compare to ResNet101 Did not compare with any other metric

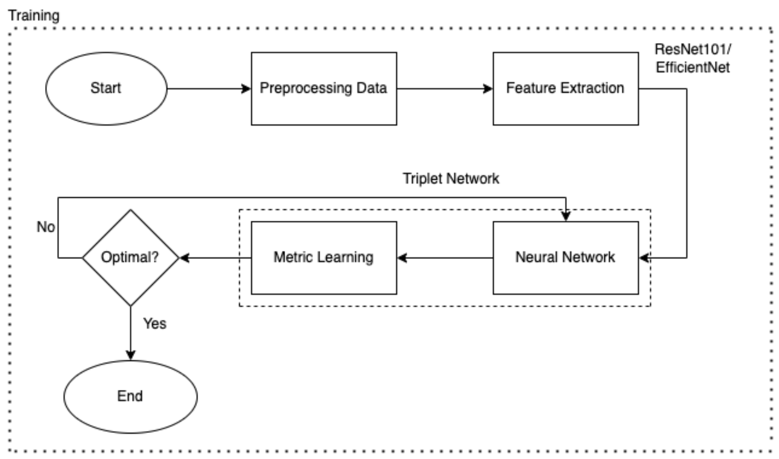
Chapter 3 Methodology

Section 3.1 Introduction

In this chapter, we will outline the main approach that will be undertaken in the second portion of the directed study course. To perform effective anomaly detection, we will follow the pipeline outlined in Section 3.1. A detailed explanation of each part of the pipeline are in subsequent subsections from Section 3.1.1 to 3.2. The remaining sections describe the dataset used (Section 3.2), and the performance metrics that will be used to evaluate the results (Section 3.3).

Section 3.2 Project Pipeline

The training and testing processes are shown in Fig 14. Each step will be explained in the subsequent sections.



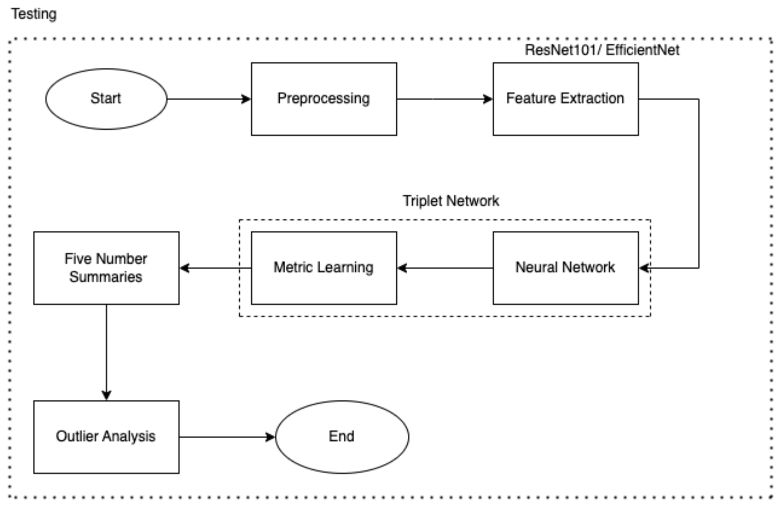


Figure 14: Training and Testing Processes for Our Project

Section 3.2.1 Preprocessing

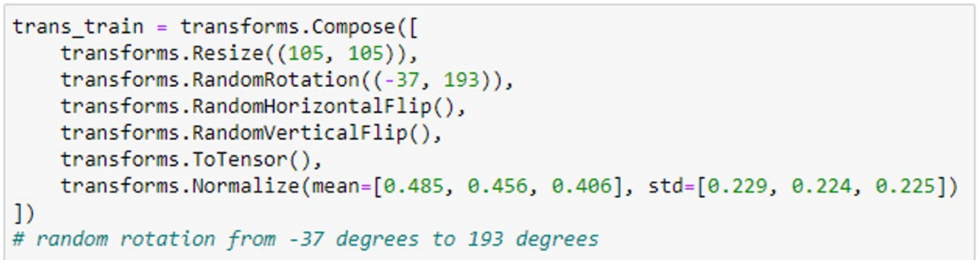
Preprocessing data refers to the techniques required to clean and process information into a form that can be used for the desired task to achieve the requested result. This is because image data may be associated with issues related to inaccuracies, imperfections, and complexity. Thus, preprocessing data allows us to reduce the inadequacies related to the data, reduce its complexity, and improve the overall quality of results. The preprocessing steps undertaken in this project's anomaly detection are in removing the "clock" image from the CDT report. The page consists of the clock image which may or may not contain a series of lines. If these lines exist, they are removed. If they do not exist, they go straight into the process for clock extraction. The next step consists of extracting the clock image from the report.

To extract the clock image, every black pixel on the image is detected and clustered. To accumulate the black pixels into a cluster, the distance between each pixel is calculated with one another. According to a predetermined threshold, pixels are classified into clusters, i.e. if they exist close to one another, they are classified as within the same cluster. This threshold value is chosen to be 0.08 as it provides a 91% accuracy in clustering the clock correctly (found through experimentation). As there may be multiple clusters in the report, the density and ratio of each cluster needs to be considered to determine which cluster contains the clock.

i) Density is defined as the number of pixels in an area. Any cluster densities larger than 0.6 are removed.

ii) The ratio is defined as the width/height of the cluster. A variable alpha is defined as the magnitude of |ratio – 1|. If this number is larger than 0.4, it is removed. This is because if the cluster contains a clock, it's ratio will be close to 1, and thus, alpha will be small. The clock cluster is then defined as the one with the smallest alpha value.

Further, the images are preprocessed before being loaded into the model. As seen in Fig 15, the images are first resized, then randomly rotated and flipped, converted into a tensor, and normalized. The reason they are resized is so that they all remain the same size as they are being trained, a size that would define the input shape of the network as well. The images are randomly rotated between -37 degrees and 193 degrees, and are randomly flipped both horizontally and vertically, to allow for some variability in the images to make the model more robust to different types of images, and to reduce the chances of overfitting. They are converted into a tensor as this is the format accepted by the model and normalized with a specific mean and standard deviation to match the images in ImageNet, as this is what our ResNet101 model is trained upon.



```python
trans_train = transforms.Compose([
    transforms.Resize((105, 105)),
    transforms.RandomRotation((-37, 193)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
# random rotation from -37 degrees to 193 degrees
```

Figure 15: Compose Function Used

Section 3.2.2 Coder

Since Siamese networks require two inputs, a coder must be used to create image pairs. These image pairs consist of one image from the true class, and one image from a different class. Since our focus is to detect anomalous images of class four, we will be pairing every class four image with every other image in the other five classes. This is a human coder that classifies the images into different classes. Since there is variation from coder to coder, and from day to day in a single coder, there is room for human error in the classifications of each class. This error is what we aim to find out in class 3 to produce a higher quality dataset. We have chosen to work with the images coded by coder 103, as this is the set with the largest number of images, and so will generate the most training and testing data (Table 2).

Table 2: Dataset Sizes of Different Coders

Coder	**103**	**104**	**105**	**106**	**107**	**108**	**109**	**111**
Total Rounds	6	1	8	1	2	3	2	3
CDT Number	8,608	808	8,315	1,276	1,379	2,263	3,344	1,769

Section 3.2.3 Feature Extraction

As mentioned by [25], image processing relies heavily on feature extraction. In the project pipeline, feature extraction occurs after image preprocessing. The aim of this step is to extract features which contain distinguishing information to discern the different classes. The features should also be unresponsive to changes in input variability and be finite in number to ensure effective computation time is taken and limited training data is used. The process revolves around extracting information from objects (or in our case images) to form feature vectors. These vectors are then used, in conjunction with target vectors, to train the classifier to identify the input. The fact that these features consist of only discerning information helps in training the classifier to discriminate between the classes based only on key patterns. This feature vector becomes the unique and precise identity of the object.

Feature extraction is important because it is responsible for extracting key distinguishing information from data for classification reasons, while simultaneously reducing the intra-class (and increasing the inter-class) variability of patterns. For this project, we will place our benchmark on the ResNet101 network, and try to improve results further using EfficientNet.

Section 3.2.4 ResNet101

ResNet101 can be used for this feature extraction phase as demonstrated in Fig 16. It is used as a backbone for many neural networks in computer vision exercises. The reason why ResNet is a great model for feature extraction lies in its use of the skip connection.

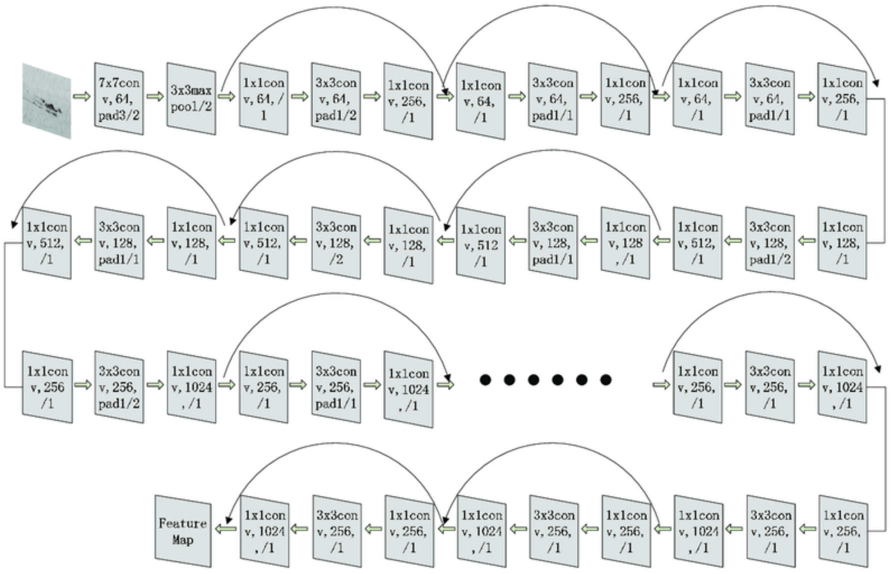


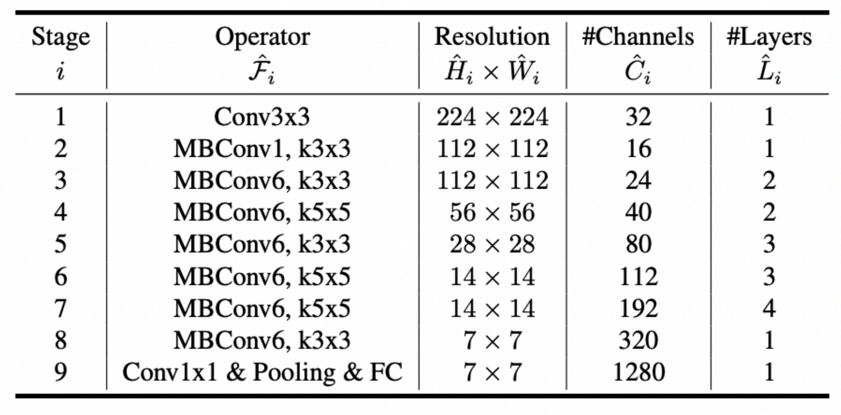
Figure 16: ResNet101

The skip connection occurs when the input, as well as the previously stacked convolutional layers, are connected to the output layer. The presence of this added component helps to minimize the vanishing gradient effect by permitting the gradient to flow through this alternate shortcut path. Through this method, the model also learns an identity function which helps ensure that the top layers do not perform worse than the lower layers. By using the ResNet101 through transfer learning, we will use the pretrained weights but replace the classifier portion with our own Triplet network [26].

Section 3.2.5 EfficientNet

In a similar fashion to ResNets, EfficientNets are also built upon the basics of ConvNets (Convolutional Neural Networks). EfficientNets, however, are built for a smaller baseline:

mobiles. By performing a neural architecture search intended for accuracy and FLOPS optimization, [27] produced the EfficientNet-B0. It consists of 9 main convolutional blocks of different sizes (Fig 17) which are scaled up using compound scaling.



Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Figure 17: EfficientNet Layers

To form an effective comparison, [27] performed a comparison of compound scaling between EfficientNet models and the older MobileNets and ResNet. Scaling was seen to effectively improve the accuracy of all models, with the EfficientNet-B7 achieving 84.3% accuracy on ImageNet with an order of magnitude lower parameter number and FLOPS. This means that using EfficientNet models leads to a smaller model that is computationally less costly. This makes EfficientNets an area that should be explored in comparison to ResNets in our project, as it has the potential for more optimal performance.

Section 3.3 Triplet Network

Section 3.3.1 Neural Network

As mentioned earlier, the triplet network is an extension of the Siamese network. Three images would be input into the network instead of two; an anchor, a positive image, and a negative image. Due to the use of three inputs instead of two, the network has an advantage over Siamese networks as it is able to learn two distances simultaneously and the large number of triplet permutations helps to reduce overfitting. The production of the labeling may be tricky as it is composed of two parts: the anchor-positive pair, and the anchor-negative pair. The anchor-negative pair is easier to obtain from the training data, leading to a higher count in the overall pool of pairs. This may lead to overfitting on the anchor-positive pairs. To counter this, we will configure a ratio between the two types of pairs: 1:3 (one anchor-negative pair for every three anchor-positive pairs) [28].

Section 3.3.2 Metric Learning (Loss function)

The loss that is to be minimized is the triplet loss. This is calculated using the formula in Eq 24 and is based on [24].

$$\sum_i^N [\parallel f(x_i^a) - f(x_i^p) \parallel_2^2 - \parallel f(x_i^a) - f(x_i^n) \parallel_2^2 + m]_+ \qquad \text{Equation 24}$$

This formula is used to calculate the estimation loss for the three inputs. This loss is minimized to ensure the distance between the anchor and the positive sample is low and the distance between the anchor and the negative sample is high. The difference between the two distances should reach alpha (a hyper-parameter defined by the user). We will set this hyperparameter to 5 but will modify it after running multiple iterations to find the optimal value for our use case. Once the difference between the two distances is smaller than alpha, the loss would become zero and the weights of the network would stop updating [28].

Section 3.4 Anomaly Detection

Finding the anomalous pairs of images will be dependent on the output of the model. Since the model is trained to minimize the loss function, it will try to ensure that same-class samples have a smaller distance between them compared to different-class samples. During testing, we input two new images into the model and plot the output values. Since our focus is finding anomalous images for class 4, we will pair each class 4 image from the Golden Standard Dataset (GSD) (known to be accurate without containing anomalous images within) with an image from every other test class. For each pair, the box-and-whiskers plot will be shown, showing the lower and upper quartiles and the median value. The interquartile range is then calculated using Eq 25.

$$\text{Interquartile Range, IQR = Upper Quartile - Lower Quartile} \qquad \text{Equation 25}$$

Anomalous pairs are then defined as shown in Equation 26 and 27.

$$\text{Anomalous pair distance} > \Gamma \qquad \text{Equation 26}$$

$$\Gamma = 1.5 * \text{IQR} \qquad \text{Equation 27}$$

Section 3.5 Dataset

The Clock Drawing Test (CDT) dataset will be used in this project. This test is commonly used in clinical practice to detect cognitive impairment in patients. It is rapid to execute and score and correlates well with other tests such as Mini-Mental State Examination (MMSE) which takes around 20 to 30 minutes to execute and can misdiagnose patients in an earlier stage of cognitive impairment. Additionally, since the MMSE is a verbally administered test, social background, race, level of education, gender, and age may affect the outcome of the screening [29]. The data consists of drawings of clock images (samples seen in Fig 18) from patients aged 60-80 years who have obtained an education for more than 6 years [30].

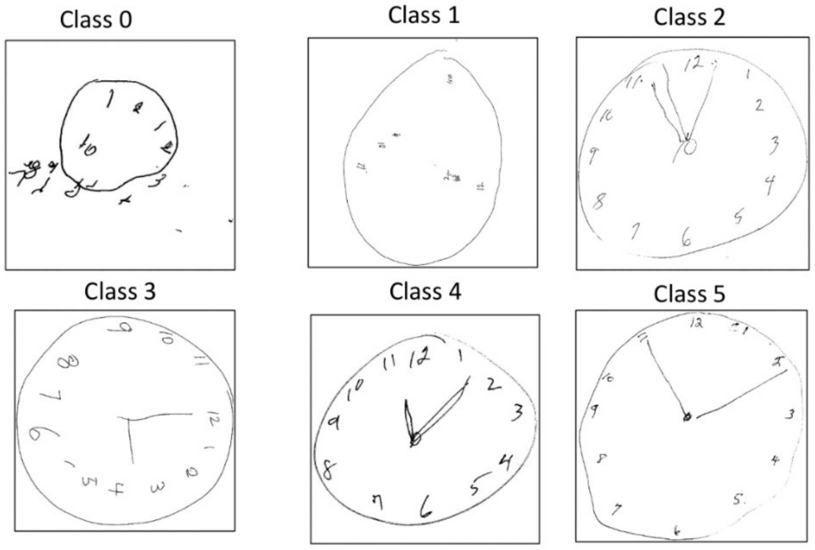


Figure 18: Samples of Clock Images Classes 0 ~ 5

Additionally, the CDT can be considered as an unbiased metric for cognitive impairment detection as it is independent of cultural backgrounds. The data consists of a compilation of images drawn by subjects. In these drawings, subjects are asked to draw a clock with its hands pointing to the time 11:10. This drawing is then scored by the physician in charge who gives it a score depending on the method used, i.e., the three-point method, the five-point method, or the seven-point method. If the final score is less than the predetermined threshold, the patient has a higher chance of developing cognitive impairment [31]. These scores, however, remain highly subjective based on the physician and can thus introduce anomalies based on human error. By introducing errors into the dataset, this can skew the final neural network model trained by producing lower performance. This in turn leads to a slightly higher chance of misdiagnosed cognitive impairment in patients relying on the software for classification. For these reasons, it becomes imperative to perform outlier analysis on the dataset to find and remove those images that have been misclassified in the dataset.

For our project, we will use 7,253 images, of which 7,093 are used for training and validation, and 160 are used as our test set in which we will find anomalies. The images are divided into 6 classes. The images are scored by a coder based on the following description:

– 0: Not recognizable as a clock

– 1: Severely distorted depiction

– 2: Moderately distorted depiction

– 3: Mildly distorted depiction

– 4: Reasonably accurate depiction

– 5: Accurate depiction of a clock

In this method, the normal image is considered as one in which every number from 1-12 appears in the correct order with the time of 11:10 being correctly sketched. Any score less than 3 is considered abnormal [31]. Classes 0, 1, and 2 are henceforth considered as abnormal and prone to cognitive disorders, while classes 3, 4, and 5 are considered normal. The final division of the entire training and testing datasets was as seen in Table 2.

The next vital dataset that is used is the Golden Standard Dataset (GSD). This dataset is usually used to evaluate coders and contains 221 images coded into the 6 classes by neuropsychology fellows and is shown in Table 3. Because of their high level of expertise in the field, these images are considered as ground truth, i.e., they contain no miscoding and will be used to represent the "anchor" in all of our training iterations.

Table 3: GSD Size

Class	0	1	2	3	4	5	Total
Count	2	6	29	48	102	34	221

Section 3.6 Defining the Anchor

As mentioned earlier, the anchor is defined as images in our GSD dataset. As seen in [11], [19], [20], and [24], the anchor is defined as an image that belongs to the same class as the positive sample, and a completely different class from our negative sample. According to [11], one way of defining similar and dissimilar images is that images belonging to the same class are considered similar, and images across every other class is considered as dissimilar. In [19], the anchor is considered as the input sample, with the positive input being defined as a same-class sample and the negative input defined as the out-of-class sample. The Euclidean distance between the anchor-positive and anchor-negative input is then calculated. Their work is based on [20], in which the anchor is defined as the reference value. The reference and positive inputs are classified as being from the same class, whereas the negative input is defined as being from a different class. The objective of the network is to correctly find out which image shares the same class as the reference. In the more generic sense, the objective is to learn which image is closer to the reference based on the metric embedding (which is what we will do). Finally, in [24], they mention how the triplet loss function aims to minimize the relative distance between the anchor and a positive instance, and the anchor and a negative instance. In this case, they define the anchor and the positive image to have identical identities, whereas the anchor and the negative image are defined to have non-identical identities. As their use case was in facial identification, the anchor and positive image were images of the same person.

From this, we can see that the anchor is always defined as an image from the same class as the positive sample. In [11], two images from the same class are chosen at random: one is defined as the anchor, and the other as the positive sample. This means that the anchor and positive image should be defined from the same set of samples. Since we have the benefit of a Golden Standard Dataset (GSD), we have used the same class samples in our GSD as we are using as our target class, i.e., if we are training a class 3 network (class 3 image folder is where our positive images are randomly sampled from), our anchor would be the class 3 GSD images. Not only does this provide variation in images (as the GSD and target class images are different from one another), it ensures that there is never overlap in the pickings of the anchor and positive images, as the folders are completely separated. However, disadvantages of this method lie in the fact that the

GSD set is small so many anchor-positive pairs would have the same anchor, and that the positive set itself has not been screened for anomalies and may itself have out-of-class images in it.

Section 3.7 Performance Metrics

To compare outlier detectors, simply using accuracy would not be sufficient. Accuracy would not be able to encapsulate the detection of in- and out-of-distribution detection. It is instead used for classification problems; to determine the number of correctly classified samples in the test set.

A box and whisker plot will be used to evaluate the anomalies in each set of test images. A boxplot consists of five numbers: the minimum value of the lowest whisker, the lower quartile, the median, the upper quartile, and the highest value of the upper whisker. This highest value is called Γ, and is defined as in Equation 27. Any value above Γ is considered as an outlier and can be visualized on the plot.

Instead, we have devised two tests to detect the significance and performance of our model. The first test is called the GSD test. Here, the GSD set of the network is divided into two folders: the base and test folders. The base folder is used as the positive set, and the test folder is used as the negative set. The two sets are paired with one another and passed through the model. If our model is working appropriately, the final boxplot should show zero outliers, as the GSD set contains no miscoding.

The second test consists of taking the original GSD test set and adding three known outliers into it from class 1. As these are significantly different from the training set, they should be detected by the model as clear outliers from the remaining data. If the model is able to detect most of these images, it will be considered successful and working to some degree. The best-case scenario would be if the model is able to detect each and every outlier in this dataset.

Chapter 4 Results

Section 4.1 Introduction

In this chapter, we document all the results that were achieved as part of the thesis requirements. First, we describe the implementation of the triplet network coded using Keras. Then, we perform a range of experiments to find the best triplet network model for our CDT dataset. Once the model is found and saved, it is used to perform anomaly detection on our test dataset. These results will be compared with the results of anomaly detection of Class 3 using the Siamese network. The anomalies obtained from our network will be fed into the Class 2 and Class 4 trained models of the triplet network to classify them into the correct classes.

Further, in order to prove the reliability of our model, we have run it on a sample test set with known anomalies. By ensuring the model detects these known anomalies, we can verify that our model is indeed successful or not.

Section 4.2 Implementation

For this thesis, we have written our own custom code from various references in order to fit our use case. Firstly, we have imported various packages that will be used in our code. These packages are listed and explained below.

i. Matplotlib: This is a popular Python library used to create static, interactive, and animated visualizations in 2D and 3D. Through this library, we are able to create plots such as line, scatter, bar, histogram, and boxplots.

ii. Torchvision: This library is a powerful computer vision tool built on PyTorch. It provides a set of utilities and datasets to work with image and video data, as well as enabling the build and training of deep neural networks from scratch. It can also be used for to import pre-trained models for image classification and other computer vision tasks.

iii. Torch: This package is a popular scientific computer framework to build and train deep neural networks. It consists of many sub-packages including torch (core package for arithmetic, tensor, and data manipulation operations), nn (to build neural networks through pre-built layers, loss functions, and optimization algorithms), optim (an optimization package), and image (an image processing package).

iv. NumPy: This library is popular for many numerical computing tasks involving large, multi-dimensional arrays and matrices by providing an N-dimensional array object that represents and manipulates data in a well-suited manner for scientific computations. It can also provide tools for integration with other libraries such as Pandas, Matplotlib, SciPy, scikit-learn, and TensorFlow.

v. Random: This package in Python is commonly used to generate pseudo-random numbers like integers, floats and sequences, as well as functions for obtaining a seed value for a random number generator. It is based (by default) on the Mersenne Twister algorithm, which is a popular algorithm known for its speed and statistical properties.

vi. PIL: The Python Imaging Library (or PIL) is used to open, manipulate, and save image files in a range of formats such as JPEG, PNG, BMP, TIFF. The package allows us to process images in many different ways such as cropping, resizing, rotating, blurring, adjusting contrast and brightness, while also supporting more advanced image processing tasks such as image segmentation and object detection (through its joint use with other libraries such as scikit-image).

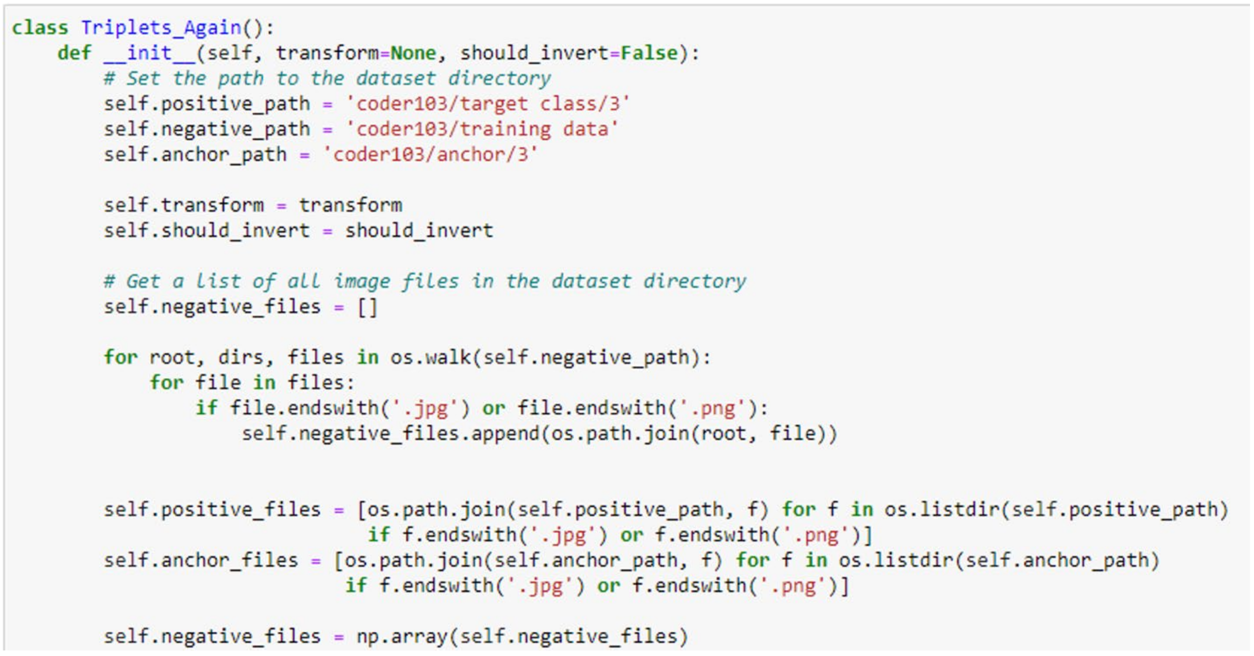
vii. OS: This library is used to allow us to interact with the operating system on which Python is running by providing a range of functions to create, delete, move, and copy files and directories. Besides this, it can also execute external programs and manage processes, while also providing functional abilities to work with file metadata.

viii. Keras: This is a neural network API that makes it easy to build and experiment with different architectures and model types. Keras allows users to create neural networks with very few lines of code, making it easier to do so using pre-build layers such as convolutional and dense layers. It also provides tools for data preprocessing, such as normalization, feature scaling, and data augmentation.

ix. TensorFlow: This is a machine learning framework developed by Google that provides many tools and resources to build and train deep learning models by using Keras. It is used for a range of applications such as computer vision, natural language processing, and speech recognition.

x. Sklearn: Known as scikit-learn, this popular Python library is commonly used for machine learning because of its range of tools for data mining, data analysis, and predictive modeling. It contains many machine learning algorithms such as linear regression, logistic regression, support vector machines, and k-nearest neighbors. It also supports statistical performance evaluation metrics such as accuracy, precision and recall, as well as tools for model tuning and hyperparameter optimization.

After the packages have been imported, we set up a class called Triplets which has one main function. As seen in Fig 19, the positive, negative, and anchor files are read and stored into their respective arrays. These arrays now hold every image that will be used for training and validation. The number of files in the positive set is 1592, in the negative set is 5453, and in the anchor set is 48.



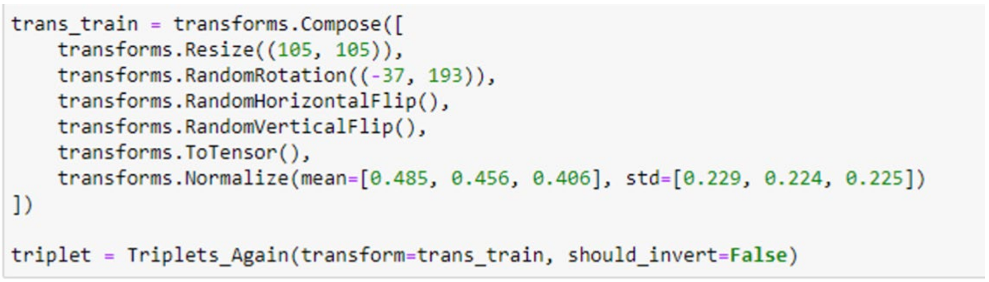
```
class Triplets_Again():
    def __init__(self, transform=None, should_invert=False):
        # Set the path to the dataset directory
        self.positive_path = 'coder103/target class/3'
        self.negative_path = 'coder103/training data'
        self.anchor_path = 'coder103/anchor/3'

        self.transform = transform
        self.should_invert = should_invert

        # Get a list of all image files in the dataset directory
        self.negative_files = []

        for root, dirs, files in os.walk(self.negative_path):
            for file in files:
                if file.endswith('.jpg') or file.endswith('.png'):
                    self.negative_files.append(os.path.join(root, file))


        self.positive_files = [os.path.join(self.positive_path, f) for f in os.listdir(self.positive_path)
                               if f.endswith('.jpg') or f.endswith('.png')]
        self.anchor_files = [os.path.join(self.anchor_path, f) for f in os.listdir(self.anchor_path)
                             if f.endswith('.jpg') or f.endswith('.png')]

        self.negative_files = np.array(self.negative_files)
```

Figure 19: Triplet Class (Extracting the Images)

The positive and negative sets are obtained from the CDT dataset as described in Chapter 3. The anchor class is taken from the Golden Standard Dataset (GSD). This dataset is considered as one with no anomalous values; it has been coded by a professional neuropsychologist and is not considered to be false. For this reason, we have chosen it to represent our anchor image.

The files are split into training and validation using the train_test_split function from sklearn, with 60% being in training and 40% in validation. The test set is not loaded here, it will be loaded during evaluation of the code to detect anomalies.

The function get_triplets is used to extract and push an array of 3 images: the anchor, the positive, and the negative image. A random image from the training or validation set is chosen, opened using the Image package, and converted into RGB format. It has been converted because our base architecture, ResNet101, only accepts RGB images. The images are then transformed using the Compose function (Fig 20), which resizes all the images to a 105 x 105, rotates them randomly between a range of -37 degrees to 193 degrees (random values chosen to prevent overfitting), randomly flipped both horizontally and vertically, and converted into a tensor which is normalized. The tensor has to be normalized with a mean and standard deviation of [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225] because of the input requirements of ResNet101.



```
trans_train = transforms.Compose([
    transforms.Resize((105, 105)),
    transforms.RandomRotation((-37, 193)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

triplet = Triplets_Again(transform=trans_train, should_invert=False)
```

Figure 20: Compose Function

Next, we define the generator function for our triplet network (Fig 21). First, an array of zeros is created for each type of image: anchors, positives, and negatives. The shape is defined as (16, 105, 105, 3), which corresponds to (batch_size, image_shape), the correct shape of an input to a ResNet101 architecture.

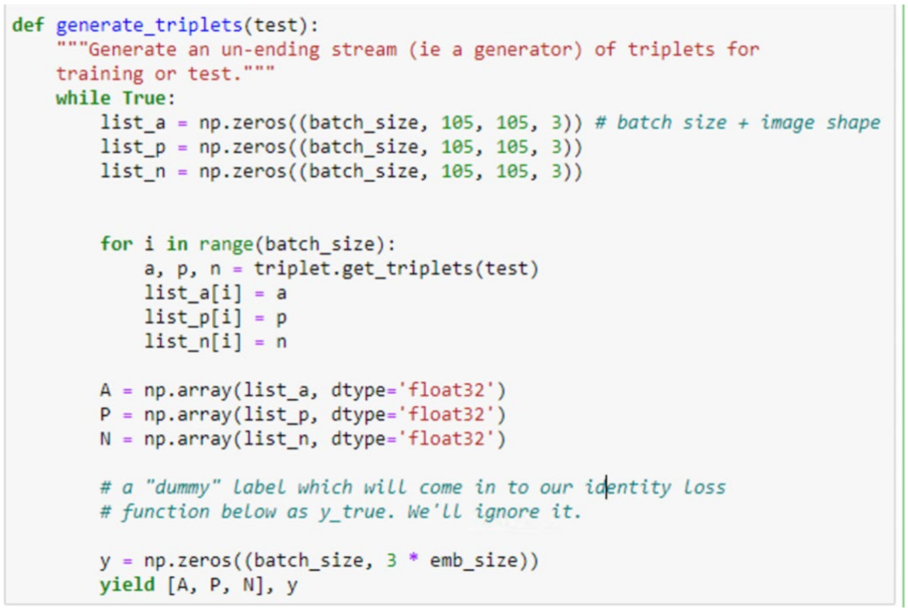

```
def generate_triplets(test):
    """Generate an un-ending stream (ie a generator) of triplets for
    training or test."""
    while True:
        list_a = np.zeros((batch_size, 105, 105, 3)) # batch size + image shape
        list_p = np.zeros((batch_size, 105, 105, 3))
        list_n = np.zeros((batch_size, 105, 105, 3))


        for i in range(batch_size):
            a, p, n = triplet.get_triplets(test)
            list_a[i] = a
            list_p[i] = p
            list_n[i] = n

        A = np.array(list_a, dtype='float32')
        P = np.array(list_p, dtype='float32')
        N = np.array(list_n, dtype='float32')

        # a "dummy" label which will come in to our identity loss
        # function below as y_true. We'll ignore it.

        y = np.zeros((batch_size, 3 * emb_size))
        yield [A, P, N], y
```

Figure 21: Triplet Generator

This is followed by a loop to extract random images from the get_triplets function for each batch, and to store them in their corresponding arrays. This array is converted into an NumPy array with a float32 format and returned along with a dummy label of zeroes (which will be used in the triplet loss function).

Next, we define our model. Since we are using Transfer Learning, we load the ResNet101 model with the pretrained ImageNet weights, but without the first input layer. This is because we want to create our own custom input layer that takes in 3 inputs simultaneously. The input shape is defined as our image shape of (105, 105, 3). Each layer is set to being trainable in order to achieve trainable parameters for each layer in the ResNet101. The three inputs are defined as the anchor, positive, and negative inputs. Each input is passed through the ResNet101 to generate its feature embeddings. A GlobalAveragePooling2D layer is added which takes the feature map of each input and averages it as a resulting vector. This is advantageous over the usual fully connected layer as it has a built-in regularizer which aids in the prevention of overfitting [32].

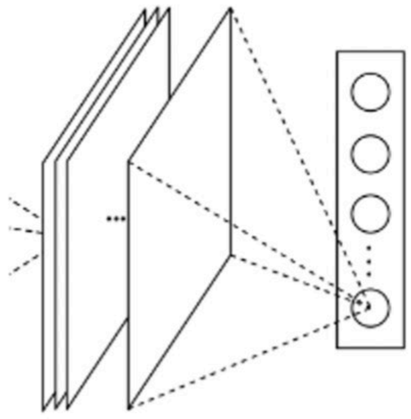


Figure 22: Global Average Pooling 2D Layer From [32]

This output vector is minimized using the triplet loss function. We then use the Dropout layer to help prevent overfitting as well, as it randomly assigns 50% of its inputs as zero with each training step. This ensures that all the weights are not optimized simultaneously, and hence stops them from meeting the same goal [33, 34]. Lastly, a Dense layer is applied which performs the operation as seen in Eq 28.

$$\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias}) \qquad \text{Equation 28}$$

This function performs the NumPy dot operation on the input data with the weight data and adds it to the bias value that helps to optimize the model [35]. This is then passed through an activation function, which in our case is not defined and is hence set to Linear, i.e. a(x) = x [36]. An L2 kernel regularizer is added to prevent overfitting on our small dataset. This type of kernel regularization is called the Ridge, with a strength of 0.01 [37]. This helps to provide a weight penalty with the equation shown in Eq. 29 [38].

$$\text{loss} = \text{L2} * \text{reduce\_sum}(\text{square}(\text{x})) \qquad \text{Equation 29}$$

This type of regularization penalizes weights that are too small or too large by converting it into a number that is close to zero [39]. We have used a regularizer value of 0.01, which, after its addition, immediately allowed the model to stop overfitting and converge slowly to a minimum.

Section 4.3 Triplet Loss Function

The most important part of our code is the loss function. The triplet loss is based on the work done by [24] and [40]. This function is the main reason for the difference in performance between Siamese and Triplet Networks. The function receives the feature embeddings of the anchor, positive, and negative images as input, and attempts to bring the anchor and positive-class images closer together in the feature embedding space, while pushing the anchor and negative-class images farther apart. We are comparing two different metrics in this project: the Euclidean distance, and the RMS (root mean square) value. As described in Section 3, Eq 24 is used to calculate the Euclidean distance. The RMS is calculated using Eq 30 and 31: this calculates the distance between the anchor-positive and anchor-negative embeddings respectively.

$$d_p = \sqrt{\frac{\sum_{i=0}^{N-1}(f(a_i)-f(p_i))^2}{N}} \qquad \text{Equation 30}$$

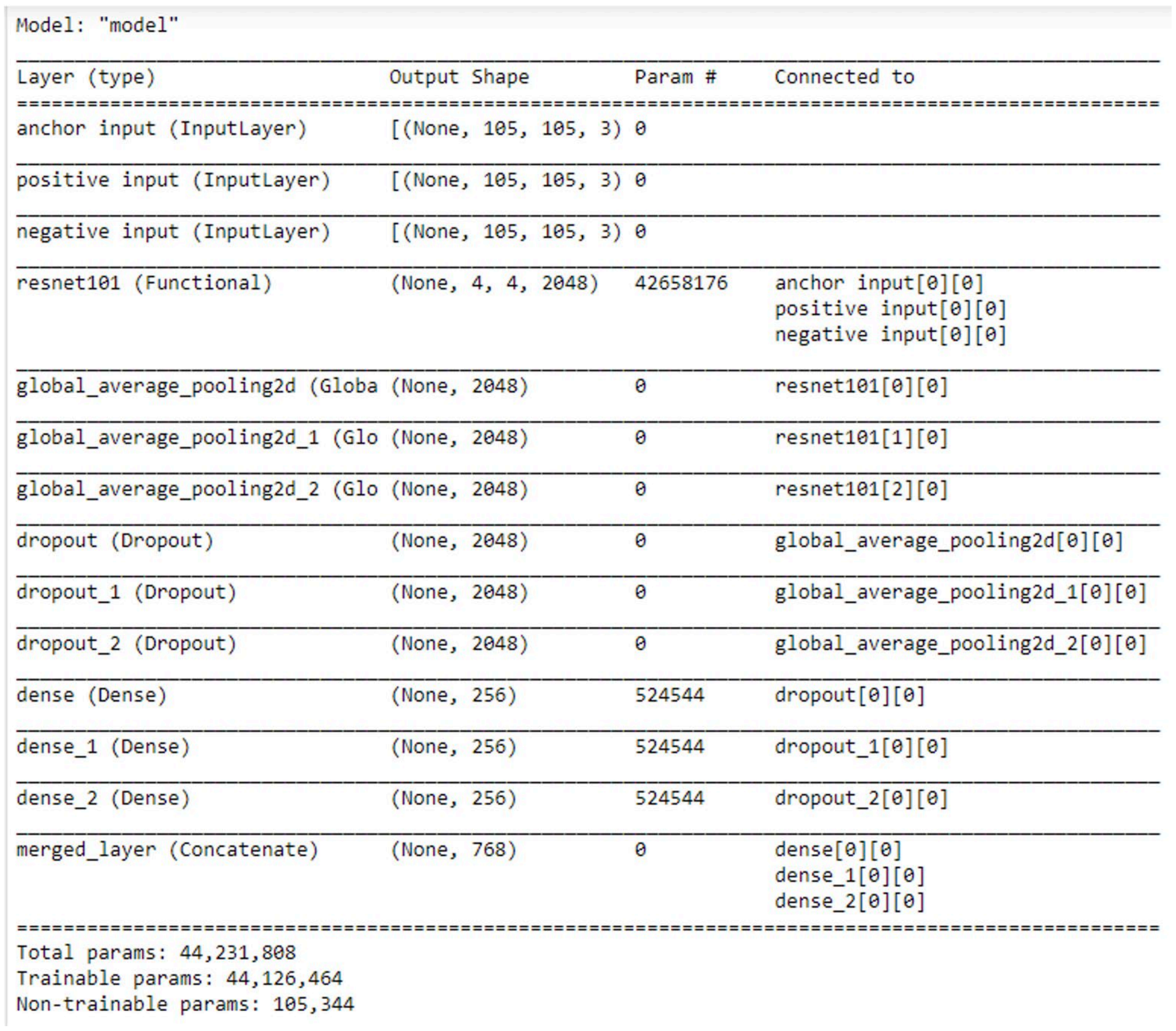
$$d_n = \sqrt{\frac{\sum_{i=0}^{N-1}(f(a_i)-f(n_i))^2}{N}} \qquad \text{Equation 31}$$

We must ensure that $d_p \leq d_n$, which means $d_p - d_n \leq 0$. To ensure that the equation is not trivial, we add a margin hyperparameter α to get $d_p - d_n + \alpha \leq 0$. The triplet loss, L, is then defined in Eq 32.

$$L = \max(d_p - d_n + \alpha, 0) \qquad \text{Equation 32}$$

As seen in Fig 21, an array of the anchor, positive, and negative embeddings is passed as y_pred. The values are separated according to the embedding size (the size of the last Dense layer output vector) and used to calculate d_p and d_n as in Eq 30 and 31. Then, the loss which is to be minimized is returned, based on Eq 32. The value is minimized as it is compared with y_true, which is an array of zeroes of size (batch_size, 3 * embedding_size). The model is set up with three inputs into ResNet101, with the output being the merged embeddings of the anchor, positive, and negative images. The model is compiled with the triplet loss function and the Adam optimizer. The learning rate and type of optimizer will be changed as part of the experimentation to find the optimal model in the next section.

Finally, the model summary is shown in Fig 23, with the model layers in Fig 24. This shows how the three input layers are connected to the ResNet101 model before the embeddings are extracted and concatenated into a single output from which the triplet loss is calculated.



```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
anchor input (InputLayer)       [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
positive input (InputLayer)     [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
negative input (InputLayer)     [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
resnet101 (Functional)          (None, 4, 4, 2048)   42658176    anchor input[0][0]
                                                                 positive input[0][0]
                                                                 negative input[0][0]
__________________________________________________________________________________________________
global_average_pooling2d (Globa (None, 2048)         0           resnet101[0][0]
__________________________________________________________________________________________________
global_average_pooling2d_1 (Glo (None, 2048)         0           resnet101[1][0]
__________________________________________________________________________________________________
global_average_pooling2d_2 (Glo (None, 2048)         0           resnet101[2][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 2048)         0           global_average_pooling2d[0][0]
__________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 2048)         0           global_average_pooling2d_1[0][0]
__________________________________________________________________________________________________
dropout_2 (Dropout)             (None, 2048)         0           global_average_pooling2d_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 256)          524544      dropout[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          524544      dropout_1[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 256)          524544      dropout_2[0][0]
__________________________________________________________________________________________________
merged_layer (Concatenate)      (None, 768)          0           dense[0][0]
                                                                 dense_1[0][0]
                                                                 dense_2[0][0]
==================================================================================================
Total params: 44,231,808
Trainable params: 44,126,464
Non-trainable params: 105,344
```

Figure 23: ResNet101 Model Summary

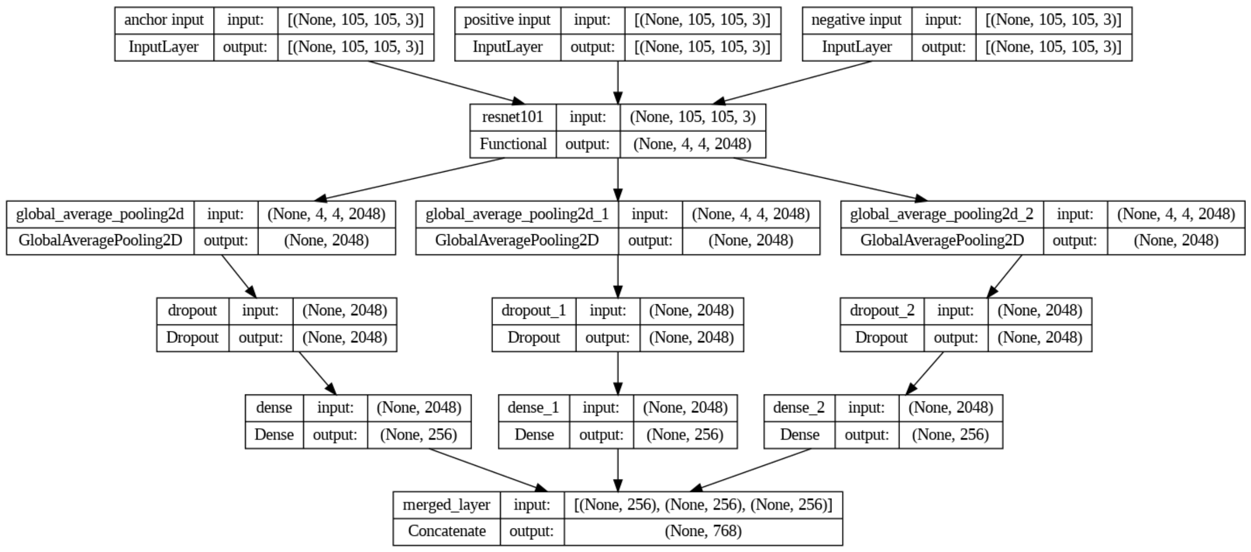

anchor input | input: | [(None, 105, 105, 3)]
InputLayer | output: | [(None, 105, 105, 3)]
positive input | input: | [(None, 105, 105, 3)]
InputLayer | output: | [(None, 105, 105, 3)]
negative input | input: | [(None, 105, 105, 3)]
InputLayer | output: | [(None, 105, 105, 3)]
resnet101 | input: | (None, 105, 105, 3)
Functional | output: | (None, 4, 4, 2048)
global_average_pooling2d | input: | (None, 4, 4, 2048)
GlobalAveragePooling2D | output: | (None, 2048)
global_average_pooling2d_1 | input: | (None, 4, 4, 2048)
GlobalAveragePooling2D | output: | (None, 2048)
global_average_pooling2d_2 | input: | (None, 4, 4, 2048)
GlobalAveragePooling2D | output: | (None, 2048)
dropout | input: | (None, 2048)
Dropout | output: | (None, 2048)
dropout_1 | input: | (None, 2048)
Dropout | output: | (None, 2048)
dropout_2 | input: | (None, 2048)
Dropout | output: | (None, 2048)
dense | input: | (None, 2048)
Dense | output: | (None, 256)
dense_1 | input: | (None, 2048)
Dense | output: | (None, 256)
dense_2 | input: | (None, 2048)
Dense | output: | (None, 256)
merged_layer | input: | [(None, 256), (None, 256), (None, 256)]
Concatenate | output: | (None, 768)

Figure 24: ResNet101 Model Layer Flow

Section 4.4 Results

In this subchapter, we perform multiple iterations to find the optimal model for CDT anomaly detection using triplet networks. We will be modifying six parameters to find the best model. These six parameters are:

i. Learning Rate: The learning rate refers to how quickly the model converges to the solution. A large value leads to faster convergence with a lower number of epochs, whereas a smaller learning rate corresponds to smaller weight changes. However, a large learning rate can lead to a suboptimal solution, while a learning rate that is too small can cause training to become stuck [41].

ii. Embedding Size: The embedding size is used as the size of the output vector of the Dense layer after Eq xx is performed [35].

iii. Batch Size: The batch size holds the number of samples that will be processed by the model before the first update of weights. In experiments done by [42], we see that increasing the batch size decreases performance, if the learning rate is not altered. In order to compensate for the larger size, a range of learning rates need to be tried to find the best one. If both factors are changed, the net affect will not largely impact the model [42].

iv. Alpha: Also known as the margin value, this number is important in keeping the positive and negative samples apart for classification by representing the smallest ratio between their L2 distances. Through using this margin, the model attempts to push negative samples farther away from the positive and anchor embeddings [43].

v. Type of Optimizer: The type of optimizer used to train the model is very important as it controls how the weights are updated. There are two main kinds of problems that are used to distinguish optimizers: convex- and non-convex optimization problems. The most important difference between the two lies in the number of minima; convex problems usually have a single global minimum, whereas non-convex problems may have more than one local minima. The optimizer is hence used to minimize the cost function to converge to the global minimum.

The following subchapters depict the 16 experiments that were run to find the best model parameters for the CDT dataset.

Section 4.4.1 Finding Optimal Learning Rate

In the first few experiments, we decided to vary the learning rate. We noticed that this was the parameter leading to the largest changes in loss and decided to find the optimal learning rate first.

Iteration 1

In the first experiment, a significantly small learning rate of 0.0001 is chosen with the Adam optimizer. The remaining parameters are set as listed below.

Embedding size = 512

Batch size = 1,000

Epochs = 50

Alpha = 5

Learning Rate = 0.0001

This led to the train-validation loss graph shown in Fig 25. As we can see, the training loss decreases from a starting value of 29.45 to 19.45, while the validation loss decreases up until about 15 epochs, after which it starts oscillating. This oscillation may be due to the small learning rate used, as the final validation loss is significantly higher than its start (29.57 while starting, and 41.69 towards the end).

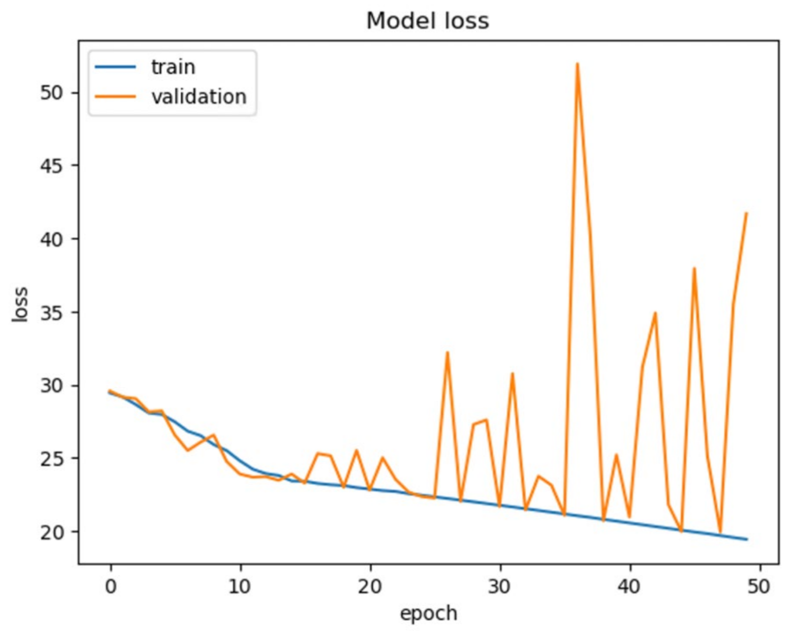


Figure 25: Iteration 1, LR = 0.0001

Iteration 2

Next, we decided to check if decreasing the learning rate even further would lead to any improvement in the losses. The parameters listed below were used, with the final train-validation loss as shown in Fig 26. From here, we can see that training loss slightly decreased from a starting value of 29.61 to 27.43, while validation loss mainly increased from 29.20 to 370.31, a 1168.18% increase in loss. This means that the learning rates we have chosen are too small and that we would need to increase them slightly for an improvement in results.

Embedding size = 512

Batch size = 1,000

Epochs = 50

Alpha = 5

Learning Rate = 0.00001

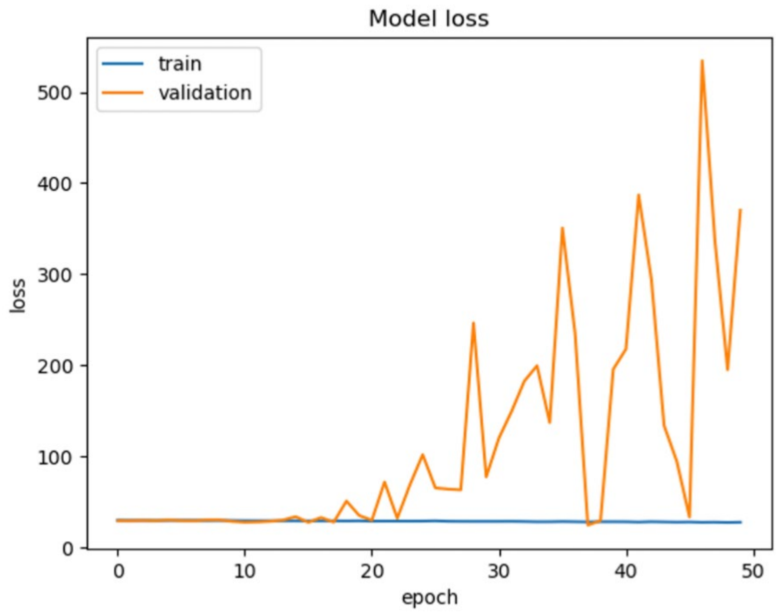


Figure 26: Iteration 2, LR = 0.00001

Iteration 3

In the next iteration, we increased the learning rate to 0.001. This produced a smooth curve (Fig 27) starting at about 28.64 to 0.3047 for training loss, and 26.10 to 1.18 for validation loss. A small bump in validation loss occurred around 40 epochs, but eventually came down before just starting to increase again around 50 epochs which may be because the model was starting to overfit.

Embedding size = 512

Batch size = 1,000

Epochs = 50

Alpha = 5

Learning Rate = 0.001

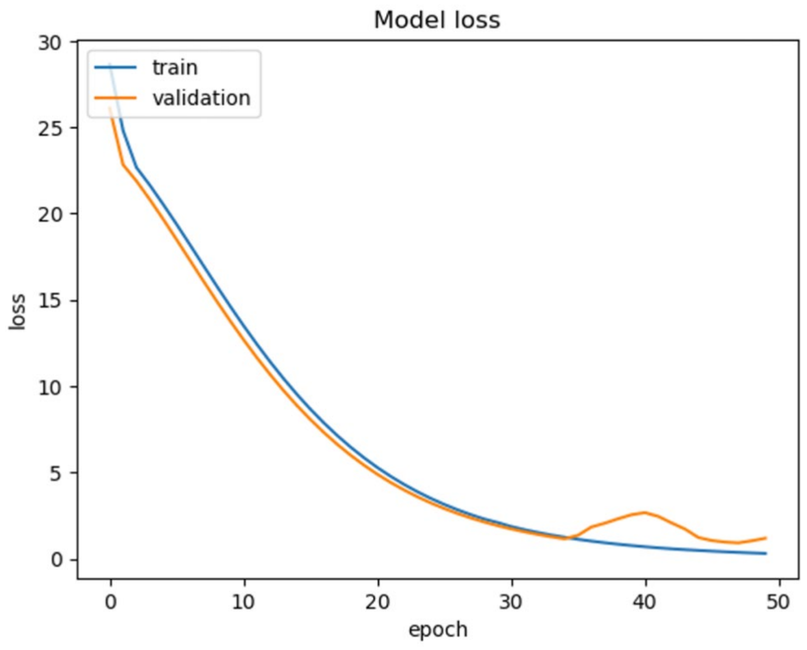


Figure 27: Iteration 3, LR = 0.001

Iteration 4

Next, we decided to increase the learning rate slightly again to 0.005. Interestingly, this led to a drastic change in the validation loss, which although started around 20.66, it peaked at a loss of the order of around 1e20 at 15 epochs, before eventually coming down to 0.0925 (while training loss decreased from 27.80 to 0.11), as seen in Fig 28. This meant that the optimal learning rate for our model is 0.001.

Embedding size = 512

Batch size = 1,000

Epochs = 50

Alpha = 5

Learning Rate = 0.005

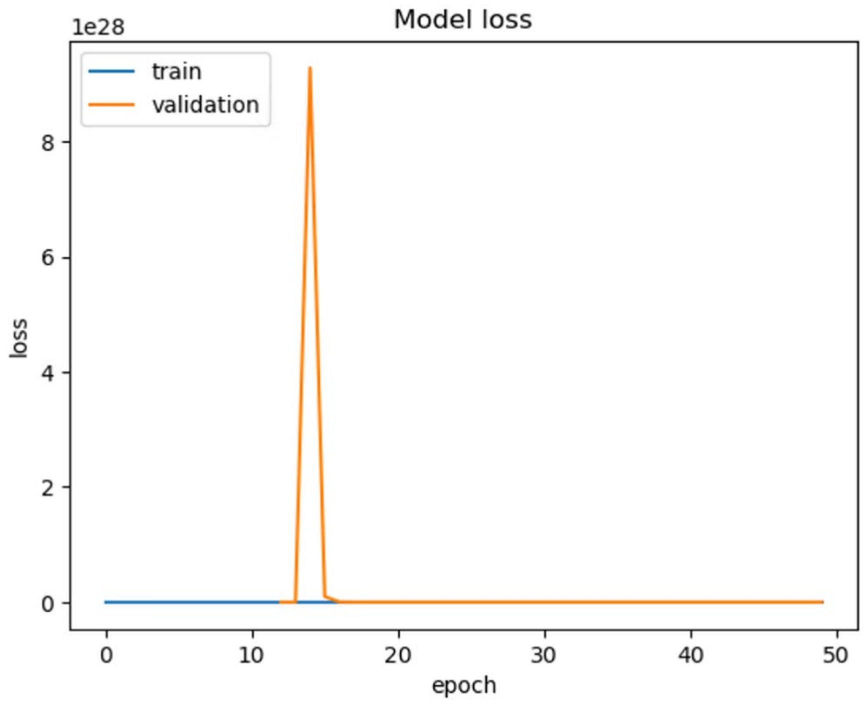


Figure 28: Iteration 4, LR = 0.005

From these iterations, we can see that our optimal learning rate is 0.001, which is set and chosen for future experiments to follow.

Section 4.4.2 Finding Optimal Embedding Size

Next, we modify the embedding size. In the previous experiments, this was set to 512. We try to see if a decrease in embedding size leads to an improvement in results. An increase in embedding size is not done, as this would lead to a more complex model with large computations between 1024 sized vectors.

Iteration 1

In the first iteration, we set the embedding size to 128. We can see that the training and validation losses start from lower numbers than when the embedding size was 256 (11.83 and 8.75 respectively). However, the final values of the losses were not as low as before, reaching its minimum at around 1.18 and 1.14 respectively, as seen in Fig 29. The value in bold signifies that it has been set from previous iterations.

Embedding size = 128

Batch size = 1,000

Epochs = 50

Alpha = 5

Learning Rate = 0.001

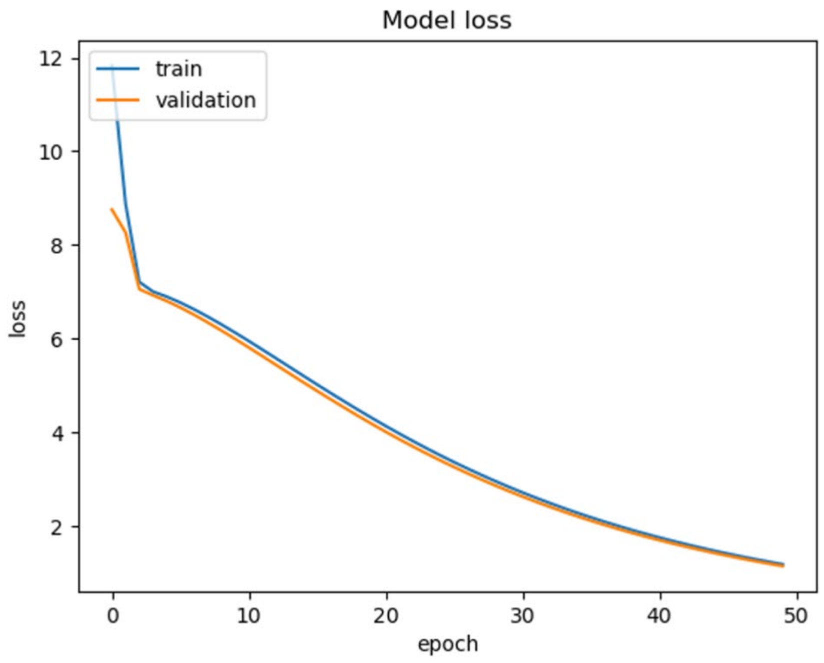


Figure 29: Iteration 1, Embedding Size = 128

Iteration 2

In the second iteration, we increase the embedding size to 256. Here, we see that the starting training and validation losses are slightly higher than at 128, of around 18.02 and 16.32 respectively. However, the smooth curve lands at a minimal loss of around 0.68 and 0.65 for training and validation, the lowest it has been so far, as seen in Fig 30.

Embedding size = 256

Batch size = 1,000

Epochs = 50

Alpha = 5

LR = 0.001

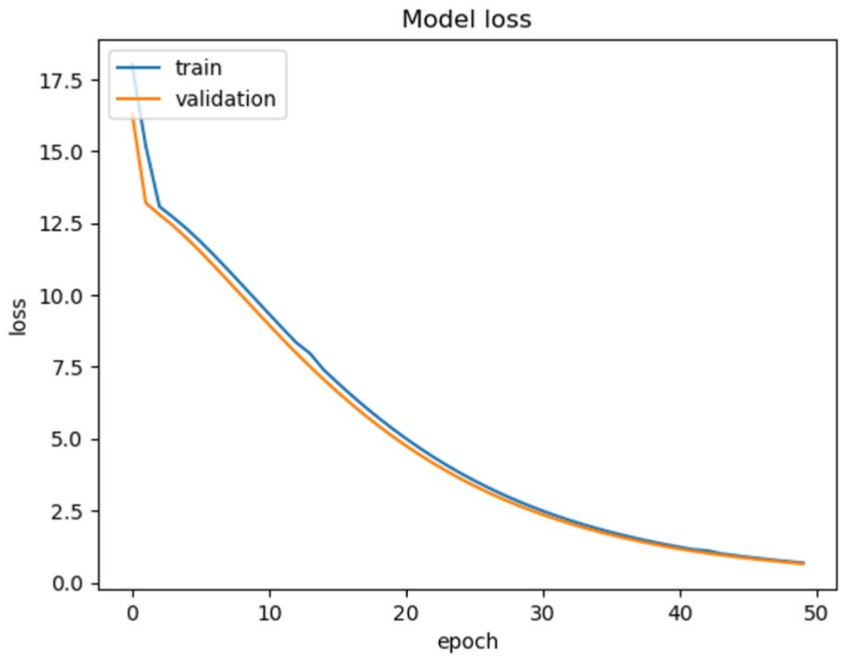


Figure 30: Iteration 2, Embedding Size = 256

Iteration 3

The third iteration is identical to iteration 3 when modifying the learning rate and has been presented here for effective comparison between the different embedding sizes. We can see that the losses start at a much higher value (almost 10 units higher), and converge at about 0.30 for training loss, and 1.18 for validation loss. Although the training loss in this case is lower, the validation loss in iteration 2 is about 45.11% lower (Fig 31).

Embedding size = 512

Batch size = 1,000

Epochs = 50

Alpha = 5

LR = 0.001

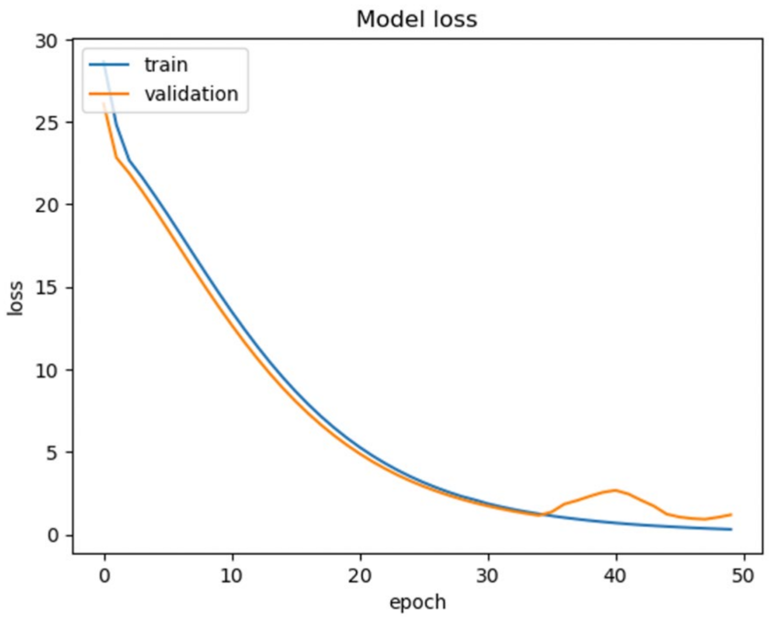


Figure 31: Iteration 3, Embedding Size = 512

Henceforth, the optimal embedding size is switched from 512 to 256 for all future experiments.

Section 4.4.3 Finding Optimal Batch Size

Now that we have set the embedding size to 256 and the learning rate to 0.001, we move on to modifying the batch size.

Iteration 1

The first iteration is the duplicate of iteration 2 from Section 4.2.1 and is shown here for completion and effective comparison purposes. The training loss decreases by 96.23%, while the validation loss decreases by 96.02% over the course of 50 epochs.

Embedding size = 256

Batch size = 1,000

Epochs = 50

Alpha = 5

Learning Rate = 0.001

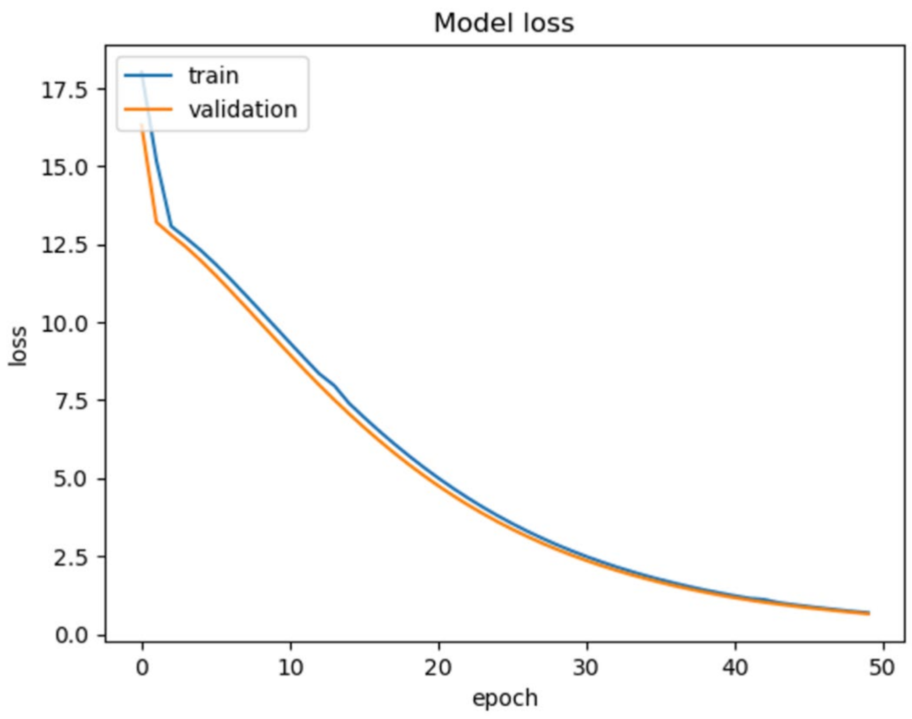


Figure 32: Iteration 1, Batch Size = 1000

Iteration 2

Next, we try decreasing the batch size to 500 to see if there are any effects in performance. Interestingly, we see that although the training loss decreases smoothly from 17.96 to 0.75, the validation loss decreases only up until about 39 epochs, after which it starts rapidly oscillating. The loss hits peaks of about 110, before slowly settling down to 0.69 at 50 epochs.

Embedding size = 256

Batch size = 500

Epochs = 50

Alpha = 5

Learning Rate = 0.001

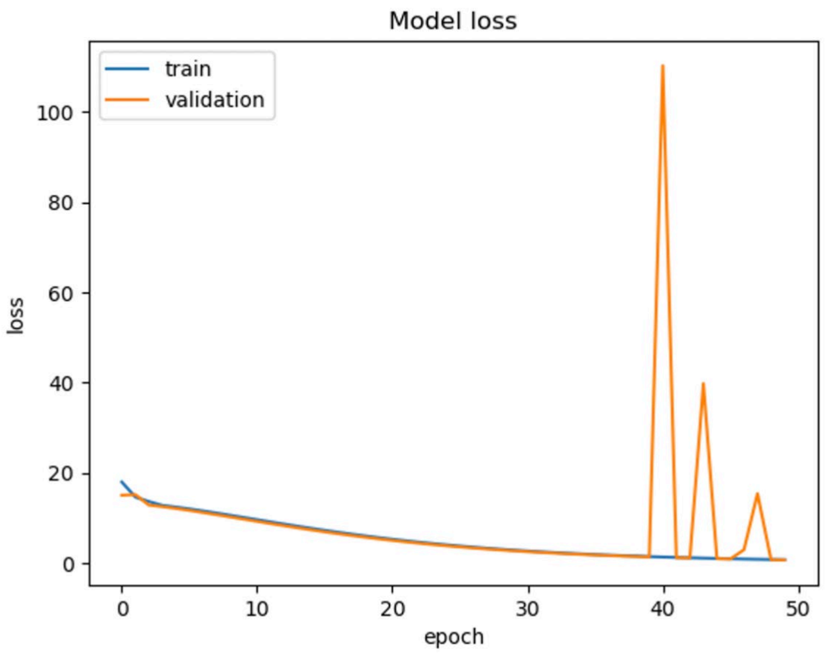


Figure 33: Iteration 2, Batch Size = 500

Iteration 3

Next, we try increasing the batch size to 1,500. We can see that both the training and validation losses have a smooth decrease from 17.73 to 0.99 and 14.36 to 1.06 respectively. This time, there are no spikes in the validation loss, which owes to the fact that triplet networks normally need higher batch sizes.

Embedding size = 256

Batch size = 1,500

Epochs = 50

Alpha = 5

Learning Rate = 0.001

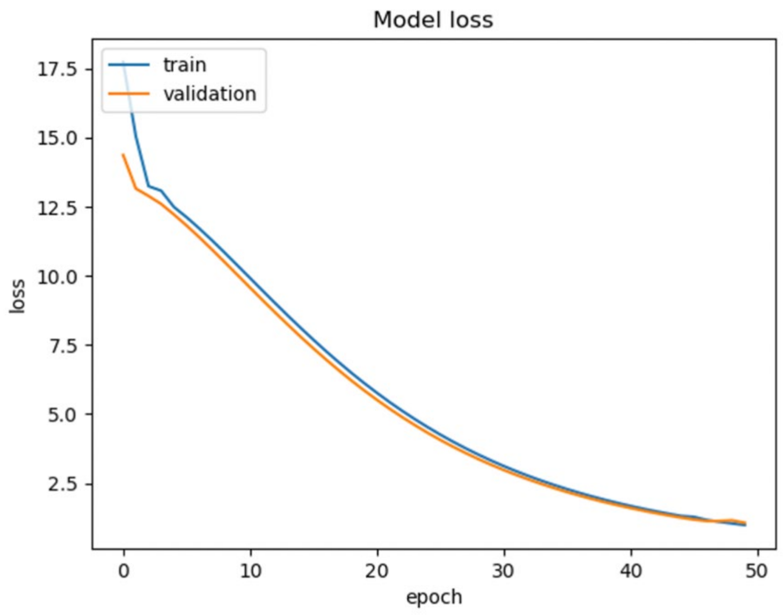


Figure 34: Iteration 3, Batch Size = 1500

The optimal batch size is chosen to be 1,000 as this results in the lowest errors of 0.68 and 0.65 for training and validation losses. Future experiments would be set with a batch size of 1000, an embedding size of 256, and a learning rate of 0.001.

Section 4.4.4 Finding Optimal Alpha

Choosing the appropriate margin value is very important when it comes to anomaly detection using triplet networks. If the margin is too high, it can cause misclassified outliers to occur, while if it is too low, it can cause inefficient separation of positive and negative classes (and hence not allowing all the anomalies to be detected).

Iteration 1

In the first iteration, we set alpha to 5, as this was the same value we used in our Siamese network. This gave us a smooth run over 50 epochs, with a loss decreasing from 18.02 to 0.68, and validation loss decreasing from 16.32 to 0.65. The remainder of the parameters have been set to the ones found earlier, as seen below.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 5

Learning Rate = 0.001

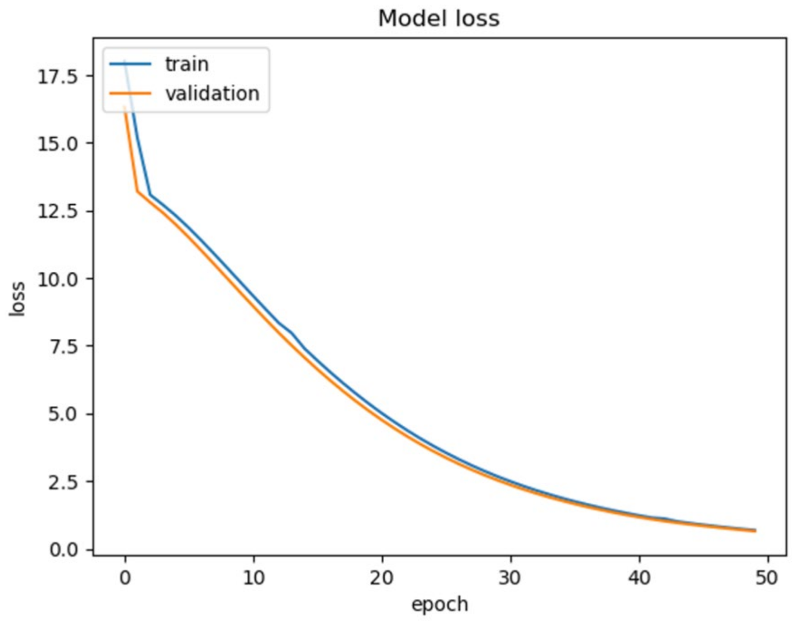


Figure 35: Iteration 1, Alpha = 5

Iteration 2

In the next iteration, we try to improve the result by decreasing the value of alpha. We note that the optimal value of alpha found by [24] is 0.2, which is a relatively small value compared to 5. After decreasing alpha to 4, we notice that the losses have decreased further, both in its initial start (17.25 and 15.35) and its final value at 50 epochs (0.64 and 0.62) for training and validation respectively. This helps prove that lowering alpha has an improvement in performance.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 4

Learning Rate = 0.001

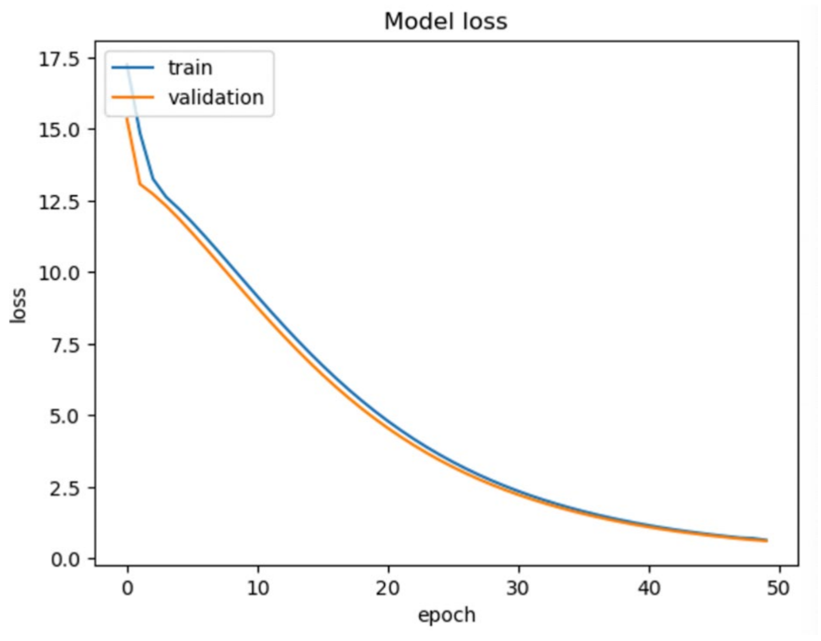


Figure 36: Iteration 2, Alpha = 4

Iteration 3

Next, we reduce alpha even further to 3. This gives a similar loss to when alpha is 5, but with lower initial losses of 16.05 and 13.56 for training and validation. The final values end at about 0.68 and 0.65, very similar to when alpha is 5. The loss curve generated is still smooth, with a small blip at around 35 epochs for the validation loss, which is when it surpasses the training loss in value.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 3

Learning Rate = 0.001

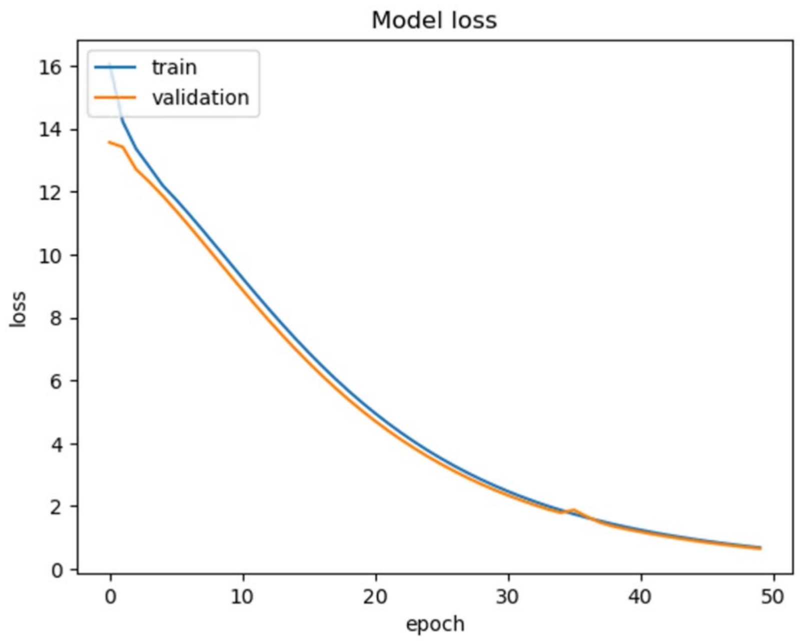


Figure 37: Iteration 3, Alpha = 3

Iteration 4

Next, we decreased alpha further to 2. This showed a dramatic change in performance, with validation loss peaking at around 80,000 units at about 18 epochs. Although the initial and final losses were smaller (15.34 to 0.35 and 14.42 to 0.33 for training and validation losses respectively), the presence of the two large variations in loss let us know that alpha should not be equal to 2.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 2

Learning Rate = 0.001

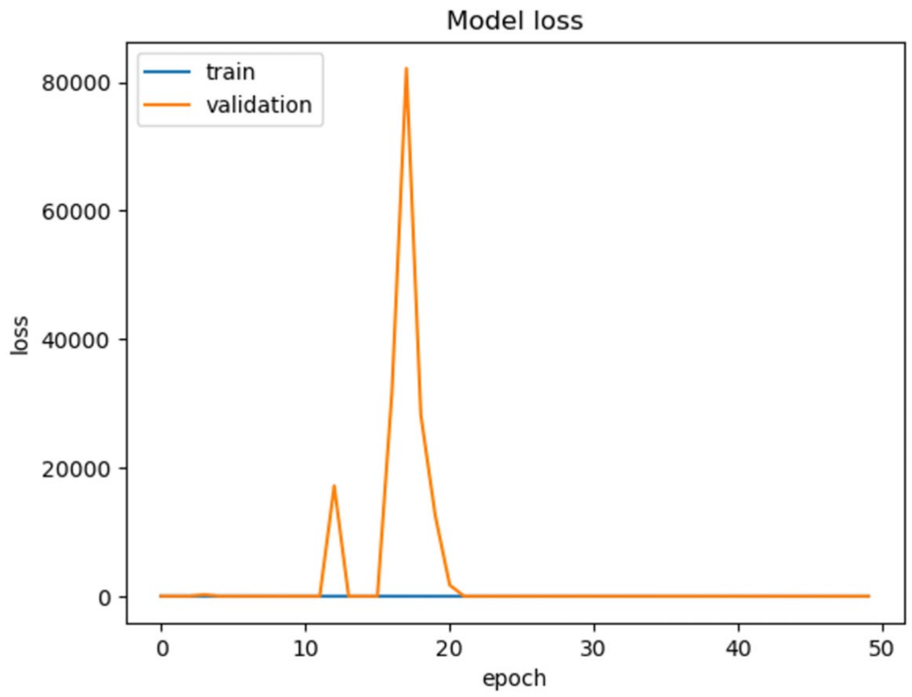


Figure 38: Iteration 4, Alpha = 2

Iteration 5

In the next iteration, we decreased alpha further to a value of 1. From the graph, we can see a spike in validation loss earlier on during the training process, hitting a value of over 40 units at around 4 epochs. The starting losses (14.44 and 13.50 for training and validation) were smaller compared to when alpha was any number larger than one, with the end losses of 0.29 and 0.38 being one of the smallest we have obtained so far.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 1

Learning Rate = 0.001

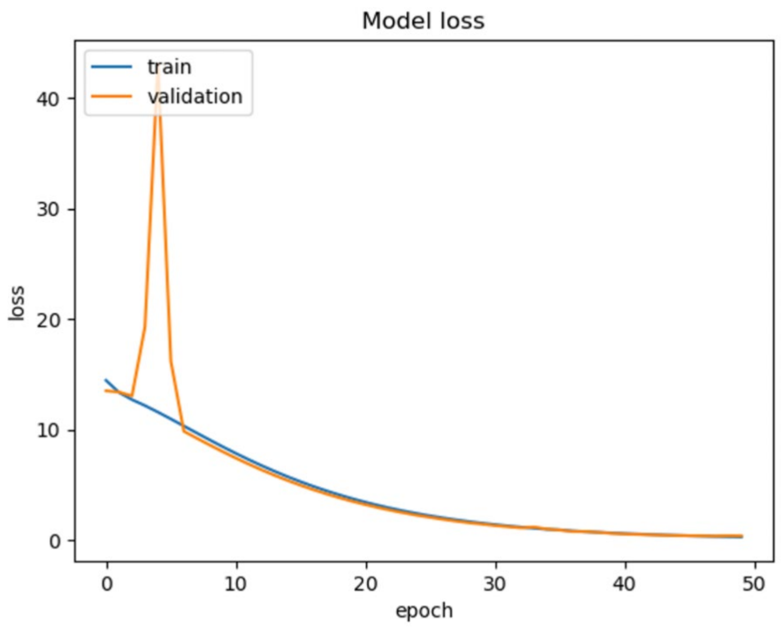


Figure 39: Iteration 5, Alpha = 1

Iteration 6

In the next iteration, we start experimenting with values lower than one to see if we can find an optimal alpha value, as the value found by [24] was 0.2. By setting alpha to 0.5, we see a smooth curve decreasing from the initial loss of 14.08 and 13.14 for training and validation loss, ending at 0.22 and 0.21 respectively. This is our lowest loss so far and, as we will see, is the alpha value chosen as it produces the lowest loss curve.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 0.5

Learning Rate = 0.001

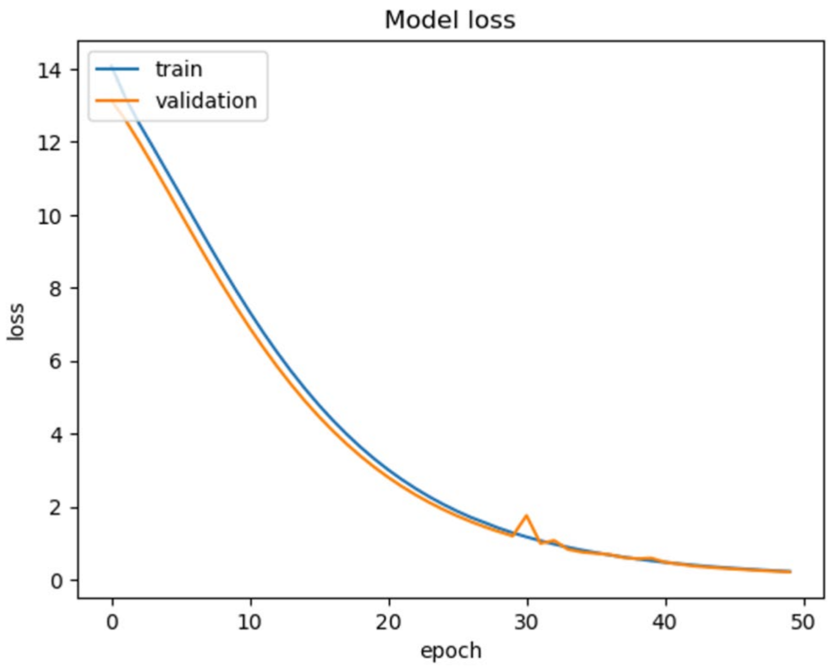


Figure 40: Iteration 6, Alpha = 0.5

Iteration 7

In the last iteration, we test the same alpha value as [24]. Interestingly, although it starts off at a similar initial loss of around 14.29 and 13.10 for training and validation, the final losses (0.28 and 0.27) are higher than what we have obtained for alpha = 5. This also shows us that decreasing alpha any further may result in higher final loss values, and that the optimal value of alpha for our use case is 0.5.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 0.2

Learning Rate = 0.001

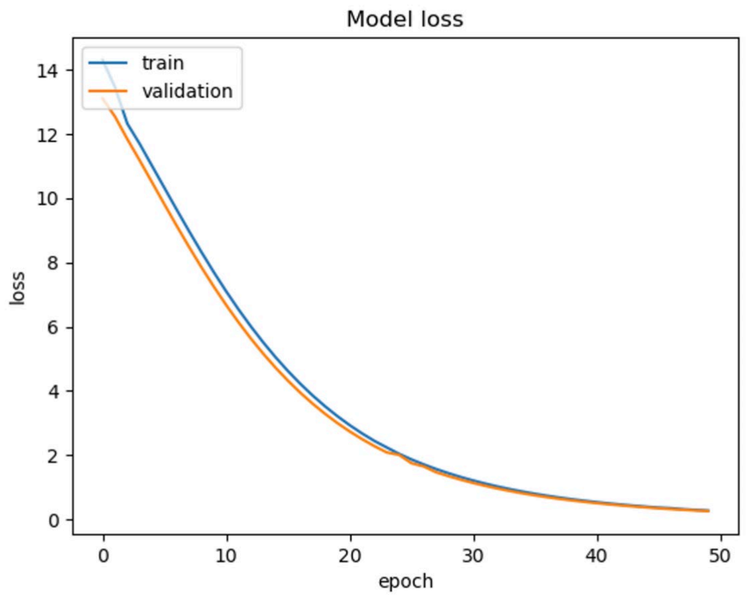


Figure 41: Iteration 7, Alpha = 0.2

Section 4.4.5 Changing the Number of Epochs

So far, we have trained for a total of 50 epochs. In an attempt to decrease the loss even further, we try to increase the number of epochs so that training runs for a longer period of time and goes through the dataset multiple times to learn more patterns.

Iteration 1

The first iteration is a replica of iteration 6 from Section 4.2.3, showcasing the results of 50 epochs when alpha is 0.5. The full list of parameters set for this run are shown below. The results are shown again here for completion and comparison purposes.

Embedding size = 256

Batch size = 1000

Epochs = 50

Alpha = 0.5

Learning Rate = 0.001

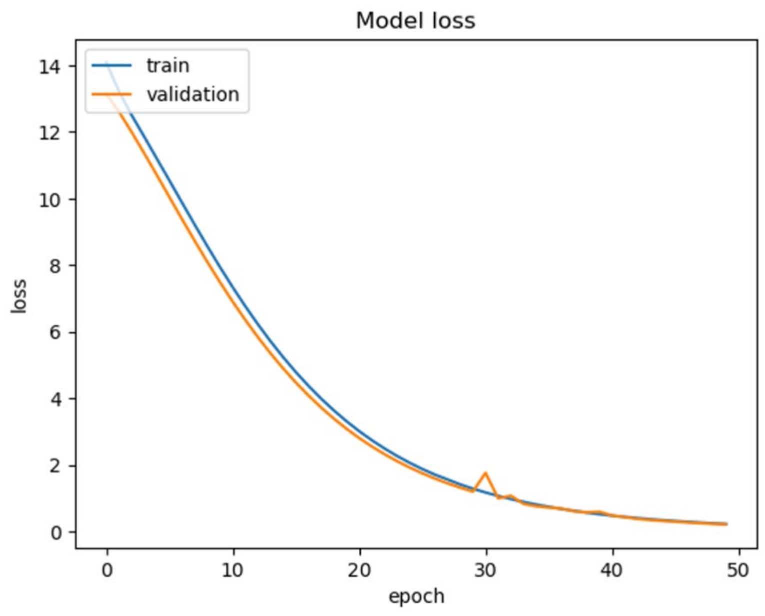


Figure 42: Iteration 1, NumEpochs = 50

Iteration 2

In the next iteration, we increase the number of epochs to 75. This produces the loss curve shown in Fig 43, with training and validation losses initializing from 13.96 and 15.29, to ending at 0.096 and 0.092 respectively. We can see that the training loss is consistently above validation loss, except at around 40 epochs. The two losses then converge to around the same value of 0.09.

Embedding size = 256

Batch size = 1000

Epochs = 75

Alpha = 0.5

Learning Rate = 0.001

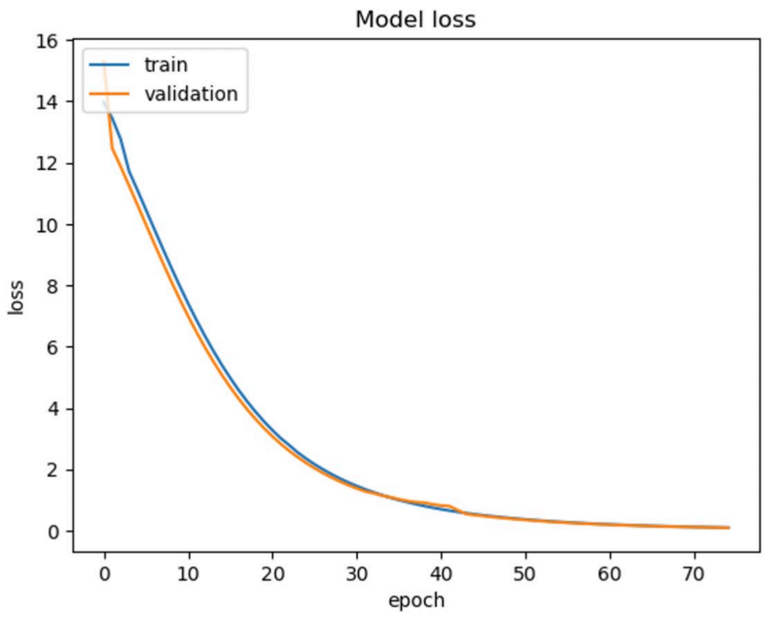


Figure 43: Iteration 2, NumEpochs = 75

Iteration 3

Next, we increase the number of epochs even further to 100. This gives us the loss graph shown in Fig 44, with the initial training and validation losses being 14.34 and 13.27, ending at 0.014 and 0.0136 respectively. This can be seen as an extension of the previous run of 75 epochs, leveling out at a loss of around 0.01 each. We decided to not train it further to reduce the chances of the model overfitting and memorizing the data.

Embedding size = 256

Batch size = 1000

Epochs = 100

Alpha = 0.5

Learning Rate = 0.001

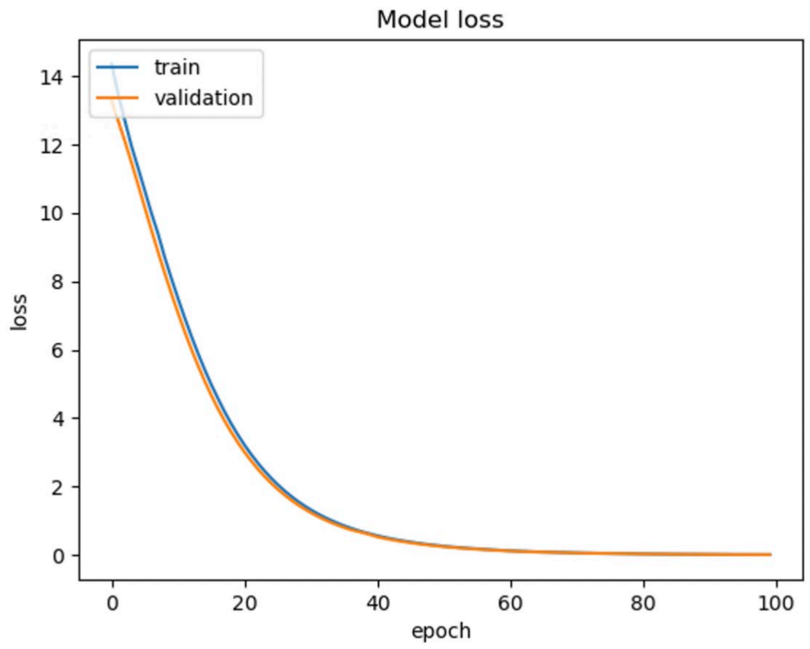


Figure 44: Iteration 3, NumEpochs = 100

Section 4.4.6 Modifying the Type of Optimizer

Next, we decided to change the type of optimizers used, based on the choices of [44]. These choices are listed below.

- Adam: This optimizer uses the Adam algorithm which is based on the SGD (or stochastic gradient descent) algorithm. It utilizes the adaptive estimation of first and second-order moments [45]. As mentioned by [46], this optimization technique not only uses low memory, it is also unaffected by gradient rescaling, is computationally efficient, and works best for problems with a large number of parameters or data.
- SGD: Improving on the gradient descent algorithm, SGD randomly selects points during training to update the weights in the opposite direction of the gradients [47, 48].
  - SGD with momentum: Adding a momentum value to the standard SGD algorithm helps to ensure gradients are accelerated in the correct direction, which helps the model converge to the optimal value in a quicker fashion [49].

- Adagrad: This type of optimizer can modify the learning rate depending on the parameters and previous knowledge. If features are present recurrently, a lower learning rate is assigned. If they are less frequent, a higher learning rate is assigned. This helps the model perform parameter updates that are smaller in size [50, 51].
- Nadam: This optimizer is a combination of Adam and NAG (Nesterov Accelerated Gradient), which adjusts the momentum of the Adam optimizer. This allows the optimizer to produce a more accurate step during gradient descent by ensuring that the parameters are updated before the gradient is computed, thus allowing for more optimized convergence of functions convex in nature [50, 51].
- RMSprop: This optimizer aims to keep a running (discounted) average of the gradient's squared values, while dividing the gradient by the square root of this average [52]. It helps to ensure oscillations only occur in one (vertical) direction, which means that if we were to increase the learning rate, the algorithm could converge faster in the horizontal direction [53].

Section 4.4.6.1 Adam

Iteration 1

This first iteration is identical to iteration 2 from Section 4.2.4 and is shown here for completion and comparison purposes. The list of hyperparameters chosen for this run are shown below. We again note that the training and validation losses have a smooth decrease over 100 epochs (from 14.34 and 13.27 to 0.014 and 0.0136 respectively).

Embedding size = 256

Batch size = 1000

Epochs = 100

Alpha = 0.5

Learning Rate = 0.001

Optimizer = Adam

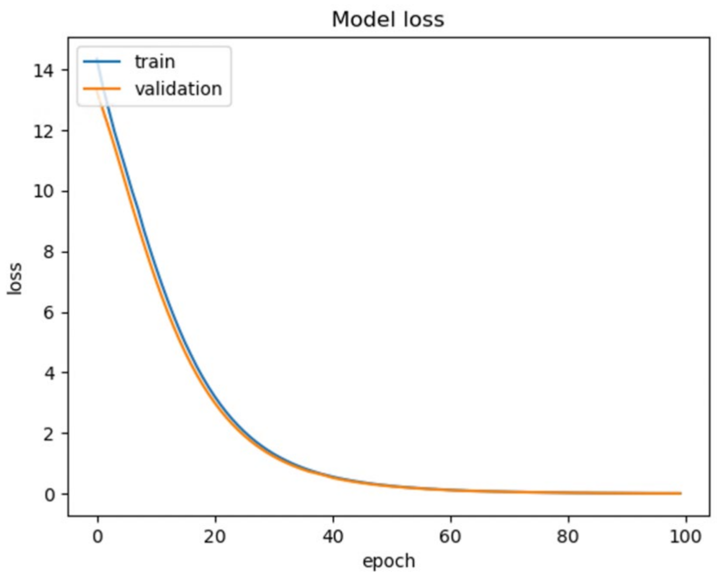


Figure 45: Iteration 1, Optimizer = Adam

Section 4.4.6.2 SGD

Iteration 2

Next, we change the optimizer to SGD without momentum. From Fig 46, we can see that the validation loss becomes incredibly unstable, peaking at over 40,000 units at almost 50 epochs. The training loss seems to decrease, but also insignificantly, as it moves decreases from 14.36 to 13.74 after 100 epochs, while validation loss significantly increases by about 384% from 14.28 to 69.11 units. This helps us show that SGD would not be the best optimizer for this use case.

Embedding size = 256

Batch size = 1000

Epochs = 100

Alpha = 0.5

Learning Rate = 0.001

Optimizer = SGD

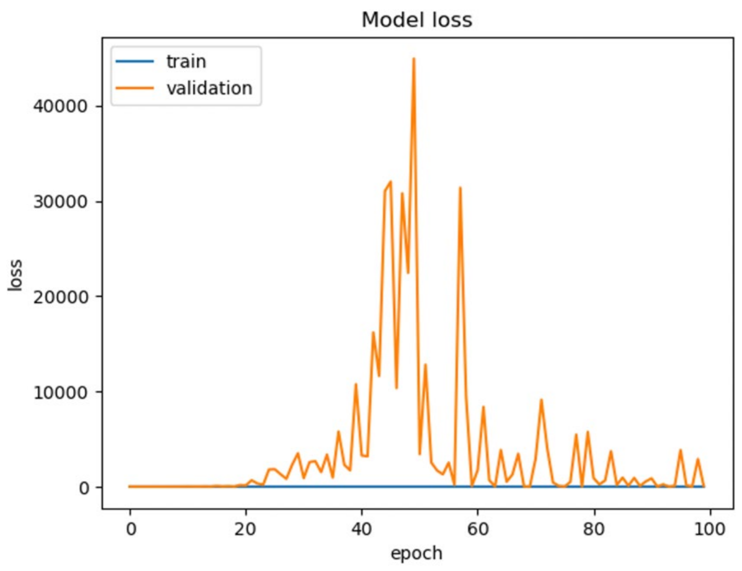


Figure 46: Iteration 2, Optimizer = SGD

Section 4.4.6.3 SGD with Momentum

Iteration 3

Next, we try adding momentum to the SGD optimizer. The value of momentum is chosen to be 0.9, as suggested by [54]. While the validation loss spike has decreased from when we used SGD without momentum, we can still see the unstable performance of the validation loss, which increases at around 10 epochs before starting to decrease around 40 epochs. The two losses also did not decrease significantly overall: with training loss going from 14.40 to 12.65, and validation loss decreasing from 14.67 to 12.64 after 100 epochs. This helps us know that even if we were to add momentum to the SGD optimizer, this optimizer does not perform very well in this task as the final losses are too high and close to the initial loss, and the training is very turbulent in terms of performance.

Embedding size = 256

Batch size = 1000

Epochs = 100

Alpha = 0.5

Learning Rate = 0.001

Optimizer = SGD with momentum=0.9

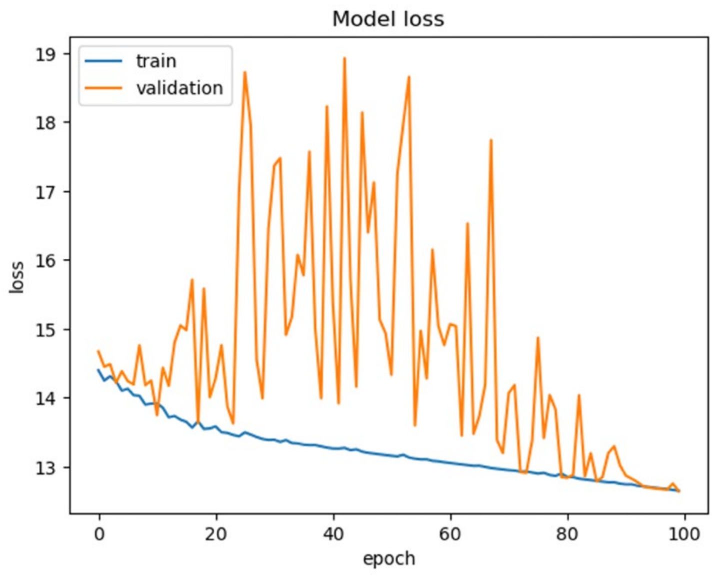


Figure 47, Iteration 3, Optimizer = SGD with Momentum = 0.9

Section 4.4.6.4 Adagrad

Iteration 4

Next, we switched to the Adagrad optimizer. This also showed very turbulent training performance, mainly in terms of validation loss, which increased from its initial loss (14.14 to 15.29) while peaking at almost 70 units at around 60 epochs. The training loss, although it decreased, only did so by a small amount; 8.46% (14.58 to 13.35). This shows us that the Adagrad optimizer would also not be a great fit for training as it gives us very unstable results.

Embedding size = 256

Batch size = 1000

Epochs = 100

Alpha = 0.5

Learning Rate = 0.001

Optimizer = Adagrad

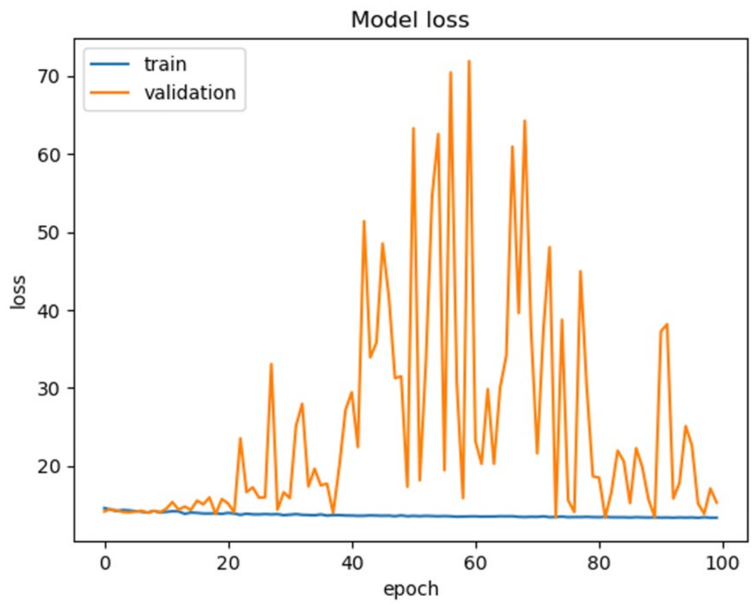


Figure 48: Iteration 4, Optimizer = Adagrad

Section 4.4.6.5 Nadam

Iteration 5

Next, we decided to switch to the Nadam optimizer. Although this time the losses were not as turbulent as earlier, the huge spike in validation loss at around 10 epochs (reaching over 175 units) makes this an unwise choice to use. Besides this, however, the losses seem to decrease from 14.25 to 0.021 and 13.17 to 0.020 for training and validation losses respectively; a decrease not as large as when we used the Adam optimizer.

Embedding size = 256

Batch size = 1000

Epochs = 100

Alpha = 0.5

Learning Rate = 0.001

Optimizer = Nadam

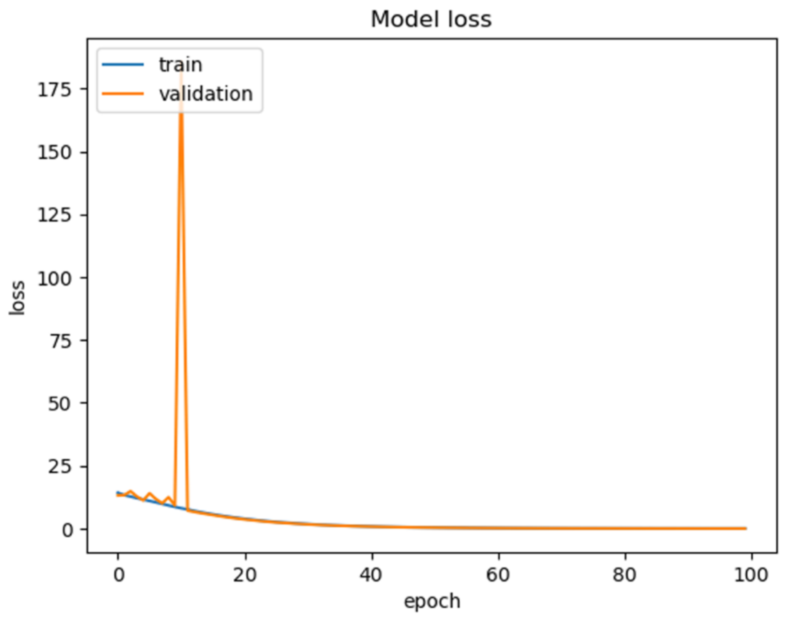


Figure 49: Iteration 5, Optimizer = Nadam

Section 4.4.6.6 RMSprop

Iteration 6

The last optimizer we tested was the RMSprop. As seen in Fig 50, the results look choppy with random spikes in losses, the worst being at around 3 epochs for validation loss (spiking at over 40 units) and around 75 epochs for training loss (reaching around 5 units). The final losses were at 0.076 and 0.072 from 13.74 and 12.25 for training and validation losses respectively, a value not as low as when we used the Adam optimizer.

Embedding size = 256

Batch size = 1000

Epochs = 100

Alpha = 0.5

Learning Rate = 0.001

Optimizer = RMSprop

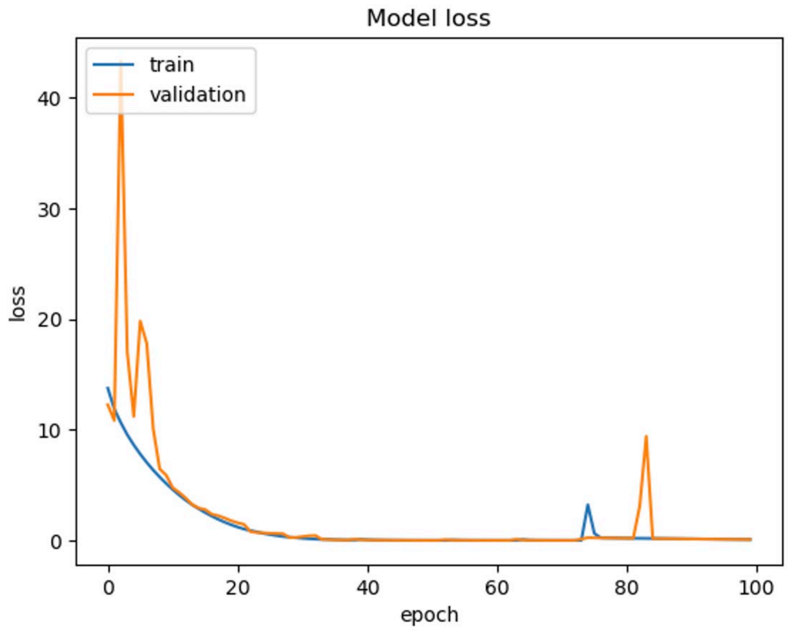
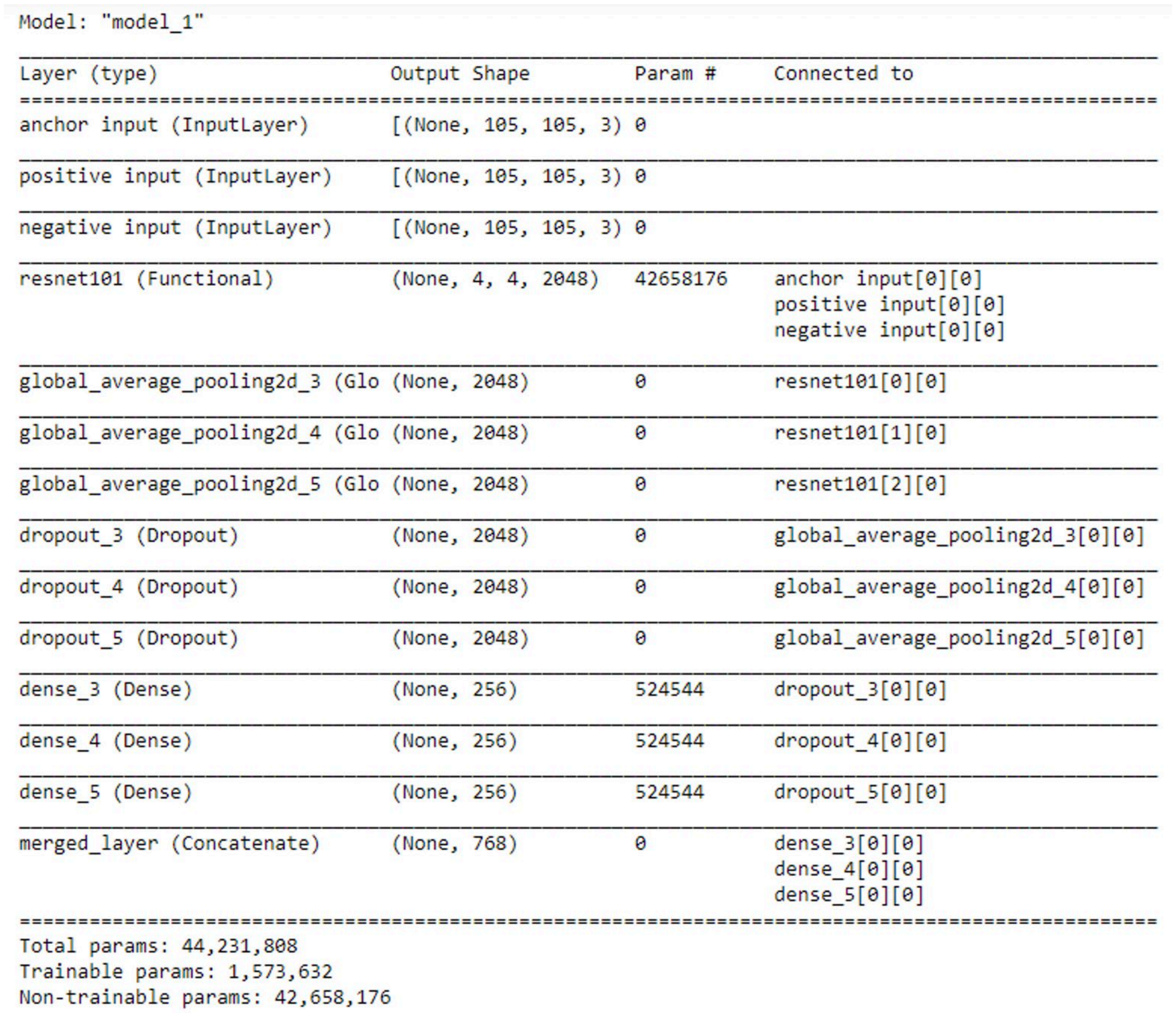


Figure 50: Iteration 6, Optimizer = RMSprop

After training all the iterations, we realized two key mistakes; one, the model was setting all layers in the ResNet101 to be trainable, and second, the batch size when picking the trio of inputs did not change when we modified the variable. This was because the loop for creating the batches of images was fixed to run 16 times, corresponding to a batch size of 16 every time. The first mistake was solved by setting the ResNet101 layers to not be trainable and using the frozen weights that had already been found through using the ImageNet training set. This gives the new model summary as seen in Fig 51, with a much smaller number of trainable parameters and a much better performance rate.



```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
anchor input (InputLayer)       [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
positive input (InputLayer)     [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
negative input (InputLayer)     [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
resnet101 (Functional)          (None, 4, 4, 2048)   42658176    anchor input[0][0]
                                                                 positive input[0][0]
                                                                 negative input[0][0]
__________________________________________________________________________________________________
global_average_pooling2d_3 (Glo (None, 2048)         0           resnet101[0][0]
__________________________________________________________________________________________________
global_average_pooling2d_4 (Glo (None, 2048)         0           resnet101[1][0]
__________________________________________________________________________________________________
global_average_pooling2d_5 (Glo (None, 2048)         0           resnet101[2][0]
__________________________________________________________________________________________________
dropout_3 (Dropout)             (None, 2048)         0           global_average_pooling2d_3[0][0]
__________________________________________________________________________________________________
dropout_4 (Dropout)             (None, 2048)         0           global_average_pooling2d_4[0][0]
__________________________________________________________________________________________________
dropout_5 (Dropout)             (None, 2048)         0           global_average_pooling2d_5[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                 (None, 256)          524544      dropout_3[0][0]
__________________________________________________________________________________________________
dense_4 (Dense)                 (None, 256)          524544      dropout_4[0][0]
__________________________________________________________________________________________________
dense_5 (Dense)                 (None, 256)          524544      dropout_5[0][0]
__________________________________________________________________________________________________
merged_layer (Concatenate)      (None, 768)          0           dense_3[0][0]
                                                                 dense_4[0][0]
                                                                 dense_5[0][0]
==================================================================================================
Total params: 44,231,808
Trainable params: 1,573,632
Non-trainable params: 42,658,176
```

Figure 51: Model Summary (without non-trainable params)

After finding these mistakes, we decided to rerun the model using the following parameters (based on [55] and [24]).

Embedding size = 64

Batch size = 256

Epochs = 250

Alpha = 5

Learning Rate = 0.0001

Optimizer = Adam

This was done for testing purposes, as we found out that when we tested the model with known anomalies, it was unable to find them using the previous models. This iteration trained smoothly over 250 epochs, which took around 5 hours, a considerable increase in training time compared to earlier iterations as the batch size was significantly larger. This resulted in the loss curve shown in Fig 52, with an initial loss of 4.58 and 3.80, ending around 0.44 and 0.44 for training and validation losses respectively.

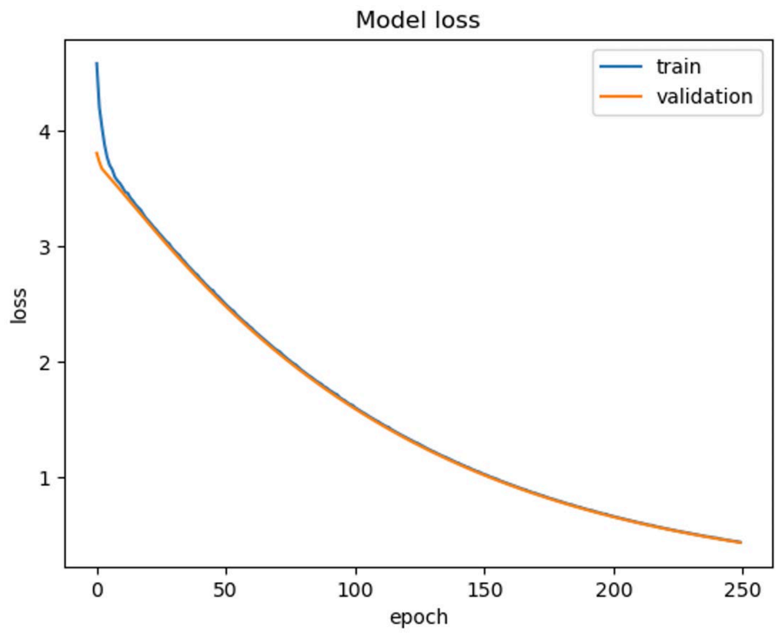


Figure 52: Updated Model Loss Function

As we will see in subsequent chapters, this new updated model is able to detect anomalies in the dataset. Henceforth, we tried to modify these parameters further to find the optimal model. We modified the hyperparameters to instead have a batch size of 512, and an alpha value of 0.5 (closer to what we imagined the optimal would be). This resulted in the graph shown in Fig 53, with losses at 4.55 and 3.76, ending at around 0.30 and 0.299 for training and validation losses respectively.

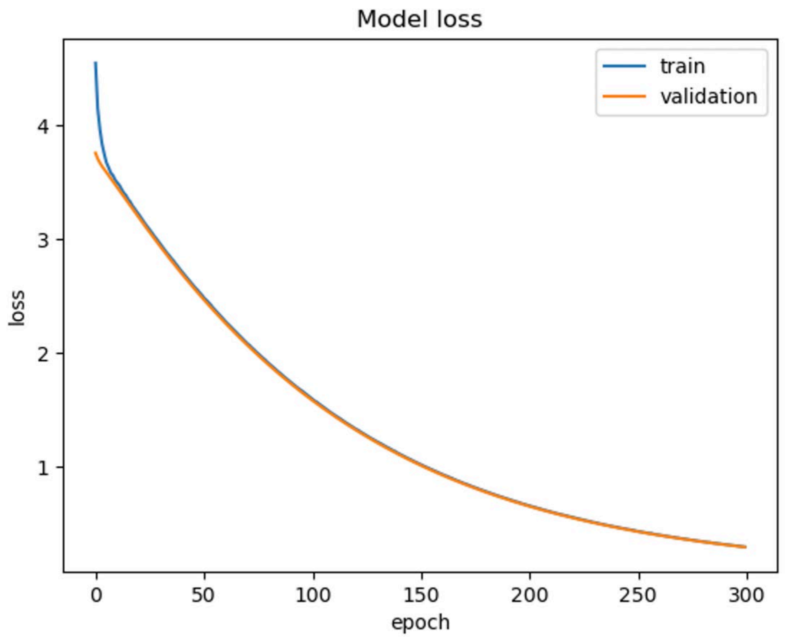


Figure 53: Updated Model Loss Function Iteration 2

Due to the time constraint, we decided to run one more iteration with the optimal parameters found in Section 4.2.5. This gave the result shown in Fig 54, with the initial training and validation losses of 13.97 and 12.57, ending at 0.028 and 0.027 respectively. Since this combination provided the lowest possible loss, this is the optimal model we have chosen.

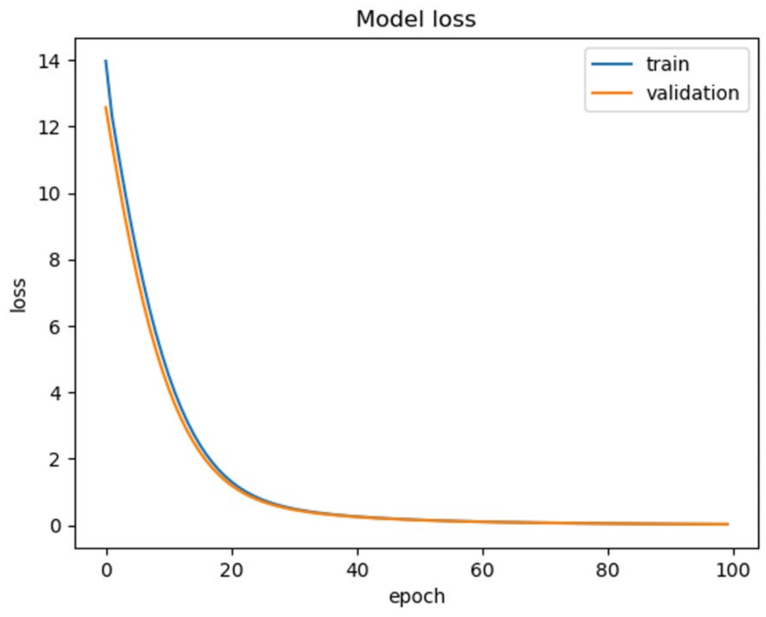


Figure 54: Optimal Model Loss Graph

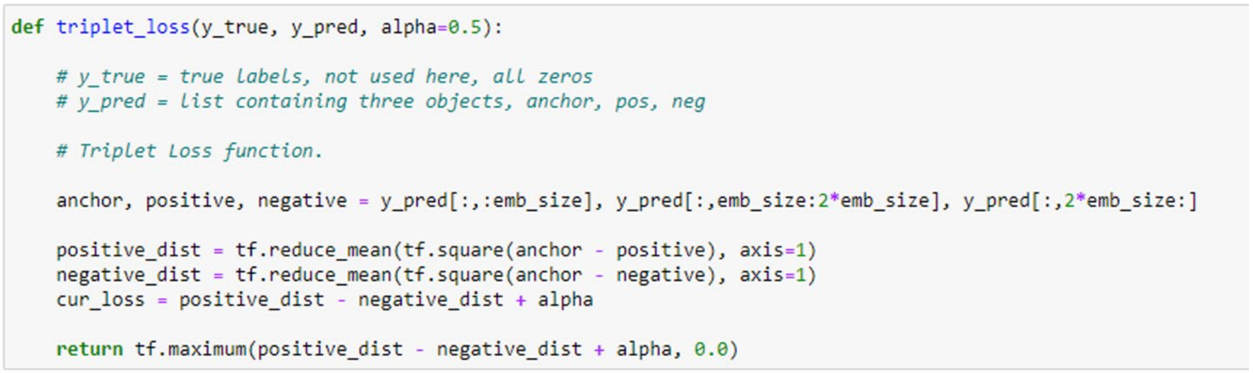
Section 4.5 The Final Model Hyperparameters

After multiple iterations, we were able to find the best parameters for our model. We used a learning rate of 0.001 as this gave the most stable training loss curve, an embedding size of 256 as this gave us the lowest loss, a batch size of 1000 as this gave the lowest error and most stable loss curve, an alpha margin value of 0.5 for smallest error, 100 epochs as this converged at the smallest loss value while preventing overfitting, and an Adam optimizer as this gave us the most stable and highest performing results.

Section 4.6 Test Results: Root Mean Square

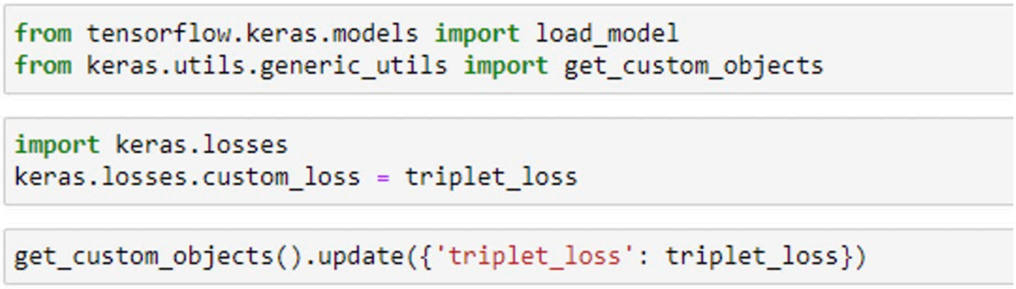
Once the optimal model was found, the next step was to run the model on the test set to perform anomaly detection. The first step to do this was in loading the saved model. In order to load it, the triplet loss function had to be redefined. This was because the Keras model would not load unless the loss function is defined in the code, and since we are using a custom loss function, we would have to define it as a custom loss function in keras.losses.custom_loss. This is shown in Fig 55 and 56.

$$\text{root\_mean\_square} = \sqrt{\sum_{i}^{N}(f(p) - f(n))^2} \qquad \text{Equation 33}$$



```
def triplet_loss(y_true, y_pred, alpha=0.5):

    # y_true = true labels, not used here, all zeros
    # y_pred = list containing three objects, anchor, pos, neg

    # Triplet Loss function.

    anchor, positive, negative = y_pred[:,:emb_size], y_pred[:,emb_size:2*emb_size], y_pred[:,2*emb_size:]

    positive_dist = tf.reduce_mean(tf.square(anchor - positive), axis=1)
    negative_dist = tf.reduce_mean(tf.square(anchor - negative), axis=1)
    cur_loss = positive_dist - negative_dist + alpha

    return tf.maximum(positive_dist - negative_dist + alpha, 0.0)
```

Figure 55: Triplet Loss Function (Code)



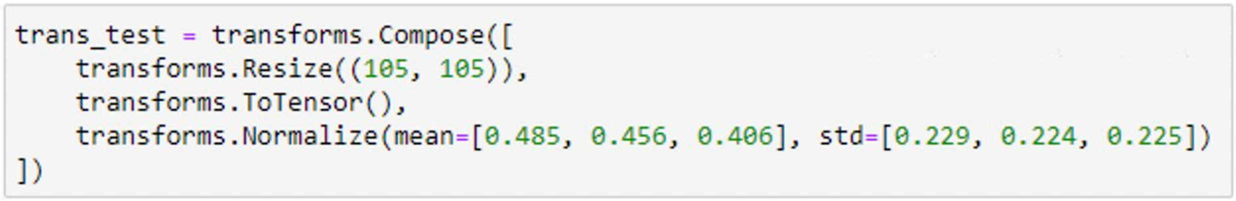
```
from tensorflow.keras.models import load_model
from keras.utils.generic_utils import get_custom_objects
```

```
import keras.losses
keras.losses.custom_loss = triplet_loss
```

```
get_custom_objects().update({'triplet_loss': triplet_loss})
```

Figure 56: Setting Custom Objects (Code)

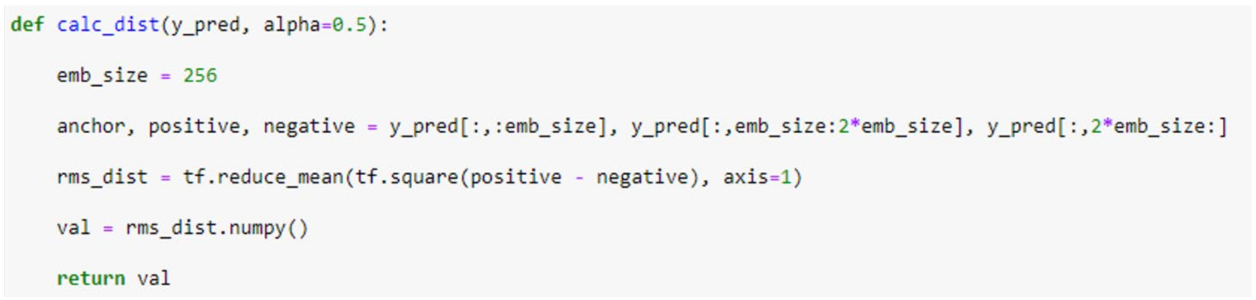
Once the model is loaded, we define a few more variables and functions. First, we define the trans_train function (Fig 57), in order to ensure all the input images undergoing testing are of the same type, format, and size as what is expected by the model.



```
trans_test = transforms.Compose([
    transforms.Resize((105, 105)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

Figure 57: Testing Compose Function (Code)

Next, we define the function used to calculate the RMS distance between the images. Since we are interested in anomaly detection, we will use the distance between the positive and negative images. This is because the model has learnt an embedding between the anchor-positive images and the anchor-negative images. The weights are set such that once any trio of images are input, the anchor-positive distance is small, and the anchor-negative distance is large. The only variable and untouched distance is between the positive and negative image, which is hence the distance we will use to find the anomalies (Fig 58). The same embedding size of 256 is defined again, identical to the one seen during training (as this is the size of the last Dense layer in the model, so the output vector would be embedded with this size).



```
def calc_dist(y_pred, alpha=0.5):

    emb_size = 256

    anchor, positive, negative = y_pred[:,:emb_size], y_pred[:,emb_size:2*emb_size], y_pred[:,2*emb_size:]

    rms_dist = tf.reduce_mean(tf.square(positive - negative), axis=1)

    val = rms_dist.numpy()

    return val
```

Figure 58: Calculating RMS Distance Function (Code)

Next, we define the paths to the image folders, and these paths would depend on which test we are doing. There are a few tests we will run for our anomaly detection project, which are based on our OCAD algorithm presented for our Siamese network. There are two main segments for this algorithm. The first one focuses on calculating the class distribution for a class C*, C*-1 and C*+1 using the Triplet networks to obtain models Tnet_C*, Tnet_(C*-1) and Tnet_(C*+1). This is done by undergoing the following steps.

(1) For each image triplet, $\bar{x}_i = (x_1^i, x_2^i, x_3^i)$ in $\Phi_{k,k}$, calculate their distance, d_i^{k*} = Tnet_k*($\bar{x}_i$), for i = 1, …, N_{k*}.

(2) Calculate the five-number summary of distribution in $D^k = \{d_1^{k*}, d_2^{k*}, \ldots, d_{Nk}^{k*}\}$. The five number summary is defined as FNS_ D^{k*} = (Min^{k*}, $Q1^{k*}$, Q^{k*}, $Q3^{k*}$, Max^{k*}}, where min^{k*} and max^{k*} are the minimum and maximum distances in D^{k*} respectively, Q_1^{k*}, $Q^{k*}$and Q_3^{k*} are the 25th percentile, 50th percentile and 75th percentile in D^{k*} respectively.

(3) If (k*-1) ≥ 0, then repeat step (1) and (2) by replacing k* with k*-1. The result should be FNS_ D^{k*-1} = (Min^{k*-1}, $Q1^{k*-1}$, Q^{k*-1}, $Q3^{k*-1}$, Max^{k*-1}),

(4) If (k*+1) ≤ K, then repeat step (1) and (2) by replacing k* with k*+1. The result should be FNS_ D^{k*+1} = (Min^{k*+1}, $Q1^{k*+1}$, Q^{k*+1}, $Q3^{k*+1}$, Max^{k*+1}}.

(5) For each class k, we derive the parameter Γ^k, for detecting out of class k distribution instances: $\Gamma^k = Q3 + \gamma^k(Q3 - Q1)$, where γ^k is typically set to 1.5 because the region between $Q1 - \gamma^k IQR$ and $Q3 + \gamma^k IQR$ contains 99.3% of data points in D^k [56].

Section 4.6.1 ResNet101 Results

Section 4.6.1.1 Class 3

The focus of our thesis is to find the anomalies in class 3. However, in order to do this, we must put our model to the test first, to ensure that the anomalies detected are true anomalies. First, we generate the five number summary (FNS) of the GSD class 3 dataset (Fig 59). The GSD dataset, which consists of 48 images (as seen in Table 3), is divided into two folders of 24 images each. One of these folders used for the positive image inputs, and the other is used for the negative image inputs. As expected, this shows that there are no anomalies detected, which should be the case of the golden standard dataset. From the boxplot, we can generate the five number summary, as seen in Table 4.

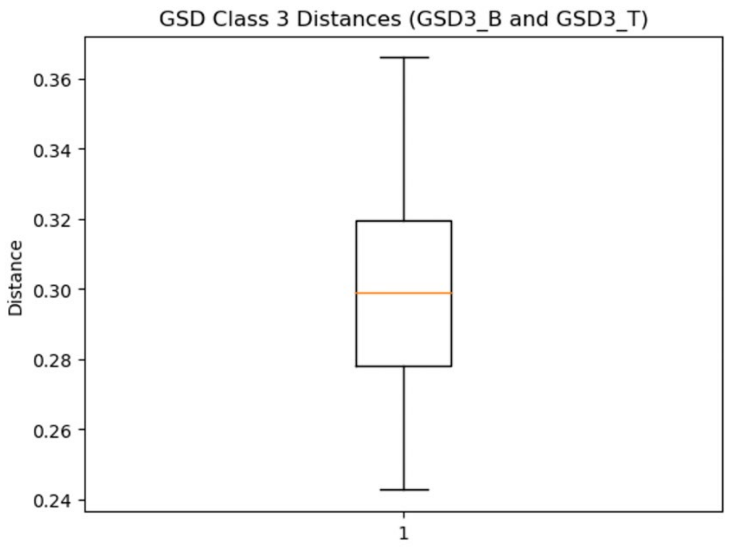


Figure 59: ResNet101, RMS, Class 3, GSD Test

In the second experiment, we use a folder with known anomalies in it. Similar to the first test, we divide the GSD folder into two; with 24 images in the positive folder, and 27 images in the negative folder (24 images from the class 3 GSD dataset and 3 images from class 1, which are taken to be anomalies). The ideal model would be able to detect all 3 images as anomalies. Our model produces the output shown in Fig 60, with two out of three anomalies being detected successfully.

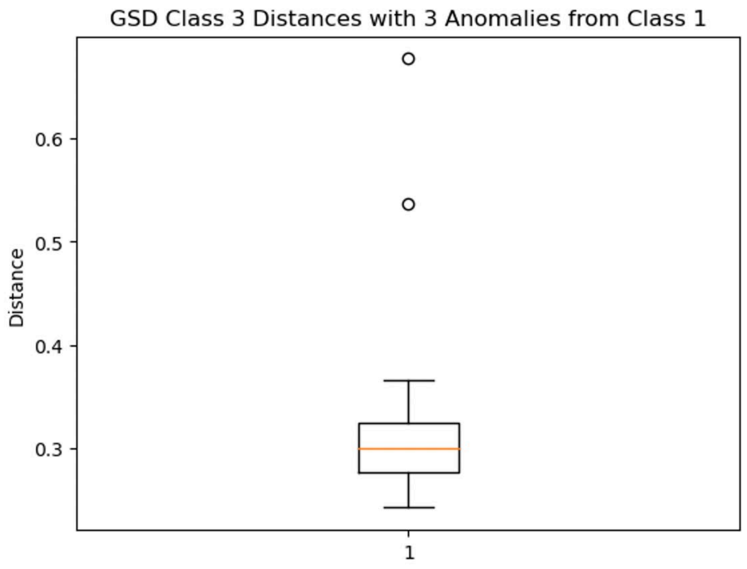


Figure 60: ResNet101, RMS, Class 3, GSD + Anomalies

Although we have not achieved detection of all 3 anomalies, we have detected the majority and, after multiple iterations in finding the optimal model, we have continued with the current model due to the time constraint. These 3 images are shown in Fig 61, with the first two being correctly detected, and the third being mis-detected.

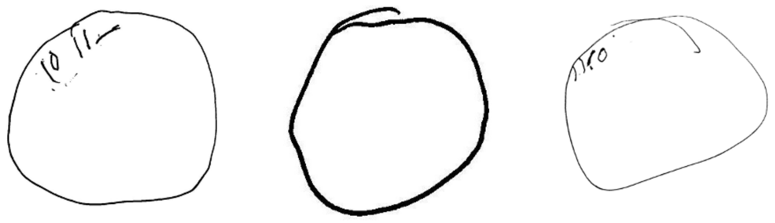


Figure 61: 3 Anomalous Images Added

Finally, we run our test class 3 set through the model. This test set consists of 160 images, of which 10 have been detected as anomalies (Fig 62). These anomalous images are shown in Fig 63.

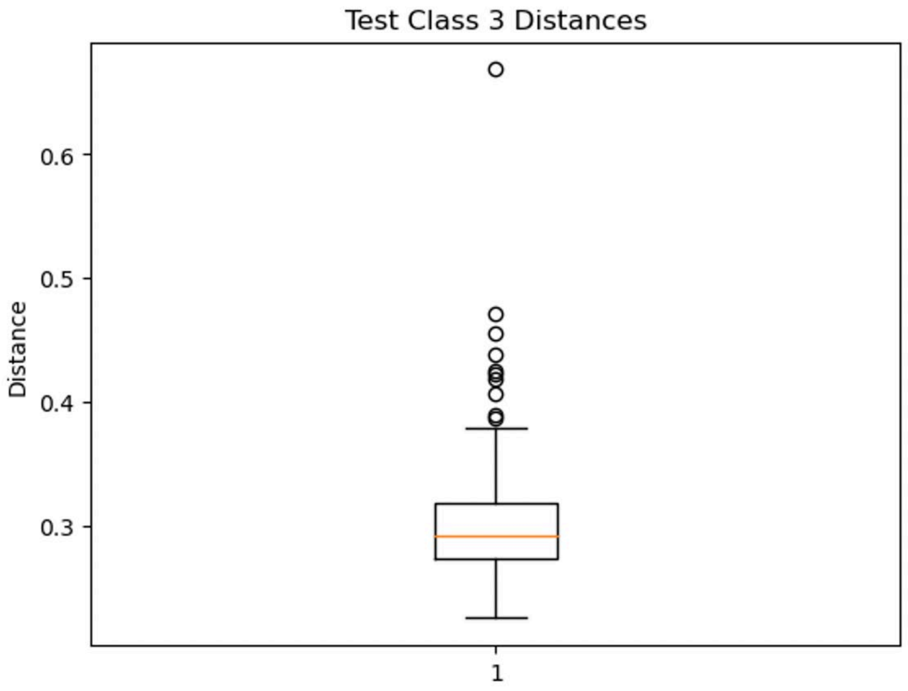


Figure 62: ResNet101, RMS, Class 3 Test Set

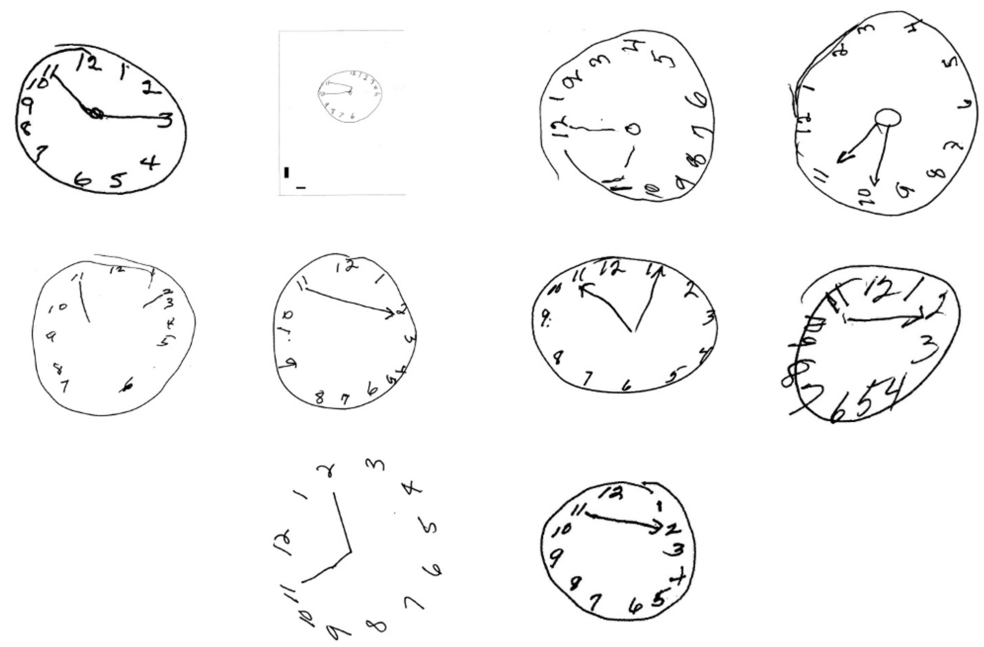


Figure 63: ResNet101, RMS, Anomalous Images Class 3

For comparison purposes, we have shown 5 images from the GSD class 3 dataset in Fig 63. As we can see, these images clearly do not belong to class 3 and have been misclassified by coder103.

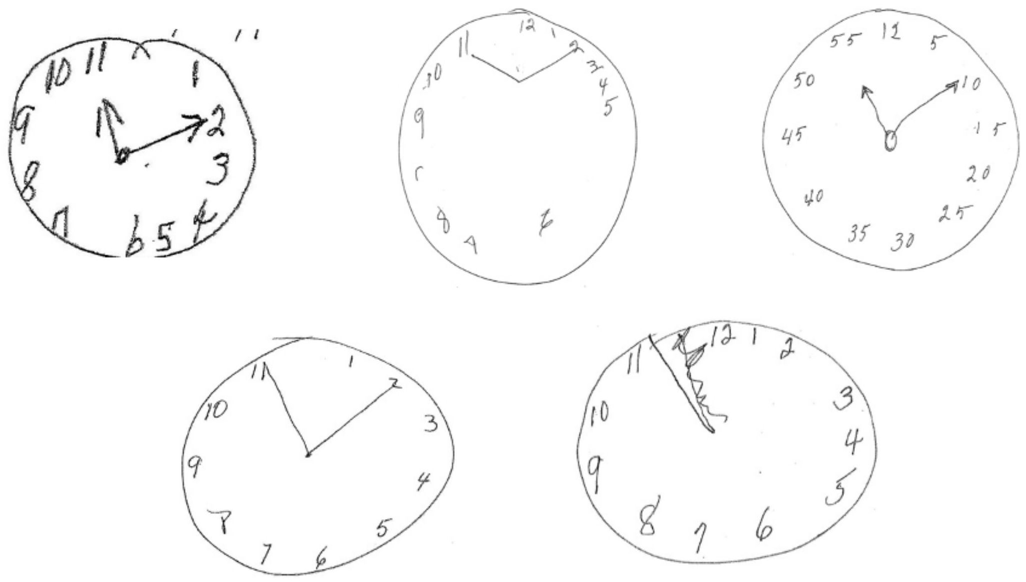


Figure 64: Sample GSD Images

The next step in the algorithm is to find out the correct classification of these anomalous images. Hence, we pass them through two other trained networks: Tnet_4 and Tnet_2.

Section 4.6.1.2 Class 4

The Tnet_4 model is trained with identical hyperparameters as Tnet_3. The resultant loss curve is shown in Fig 64, with losses decreasing from 13.996 and 12.596 to 0.0296 and 0.0287 for training and validation losses respectively.

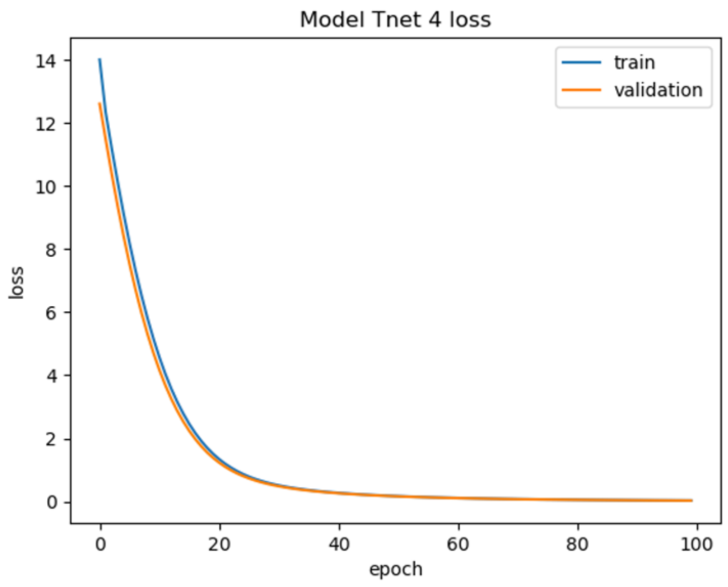


Figure 65: ResNet101, RMS, Class 4 Loss Graph

Next, the FNS of Tnet_4 is calculated in the same as Tnet_3. The GSD 4 class is divided into two: GSD4_B (the base set) and GSD4_T (the test set). Each image from the test set is paired with every other image in the base set in each iteration to find the average distance from it to an image in the base set. This is repeated for every image in the test set, obtaining the final boxplot as seen in Fig 65. The FNS is reported in Table 4.

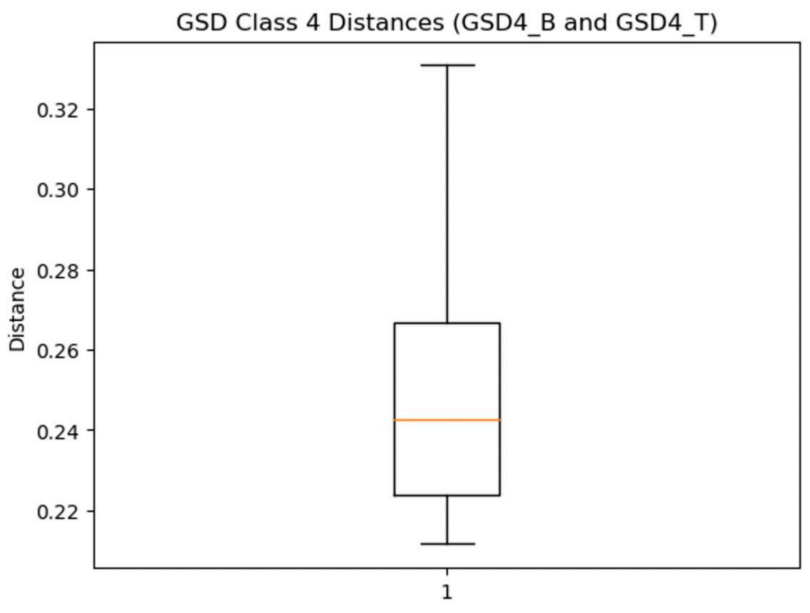


Figure 66: ResNet101, RMS, Class 4, GSD Test

Next, we pair each of the 10 anomalous images with every image in the GSD 4 dataset, obtain the average for each pair, and overlay the points on Fig 65. This is shown in Fig 66.

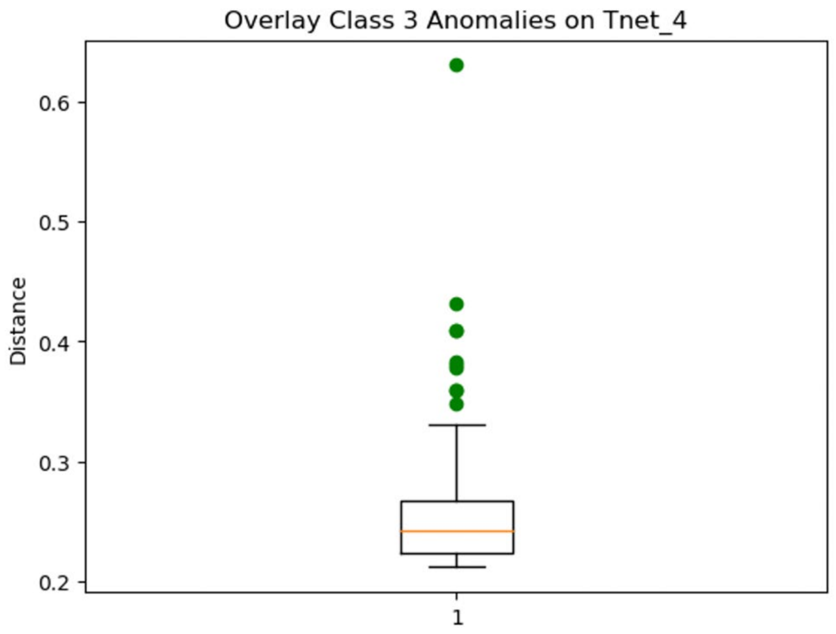


Figure 67: ResNet101, RMS, Class 4 Overlay of Anomalous Images

As we can see, this shows us that none of the 10 anomalous images are from Class 4, as they all lie above the gamma value of 0.331. However, this makes sense. Since our dataset follows an ordinal classification from 0 to 5, with 0 being the lowest quality and 5 being the highest quality, we can hypothesize that since the anomalous images from class 3 were of lower quality than class 3 itself, they should be anomalous to Tnet_4 as well.

Section 4.6.1.3 Class 2

Next, the Tnet_2 model is trained with identical hyperparameters as Tnet_3 and Tnet_4. The resultant loss curve is shown in Fig 67, with losses decreasing from 13.95 and 12.57 to 0.0247 and 0.0237 for training and validation losses respectively.

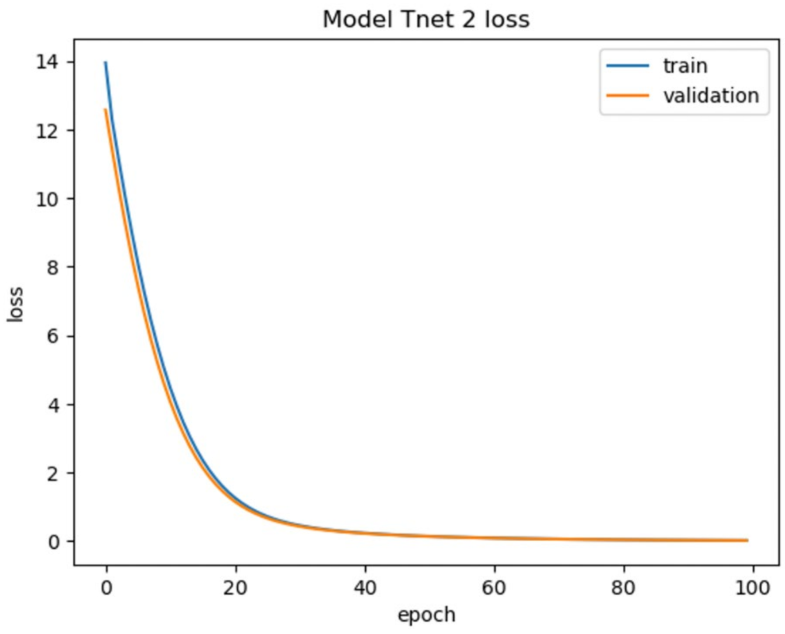


Figure 68: ResNet101, RMS, Class 2 Loss Graph

Next, the FNS of Tnet_2 is generated using the same method as earlier; by splitting the GSD 2 dataset into two, one holding about half the images in a folder titled GSD2_B (the base set), and another holding the second half of the images in a folder titled GSD2_T (the test set). The RMS distance between the positive and negative images were calculated in the same fashion as with Tnet_4, with the resulting boxplot shown in Fig 68.

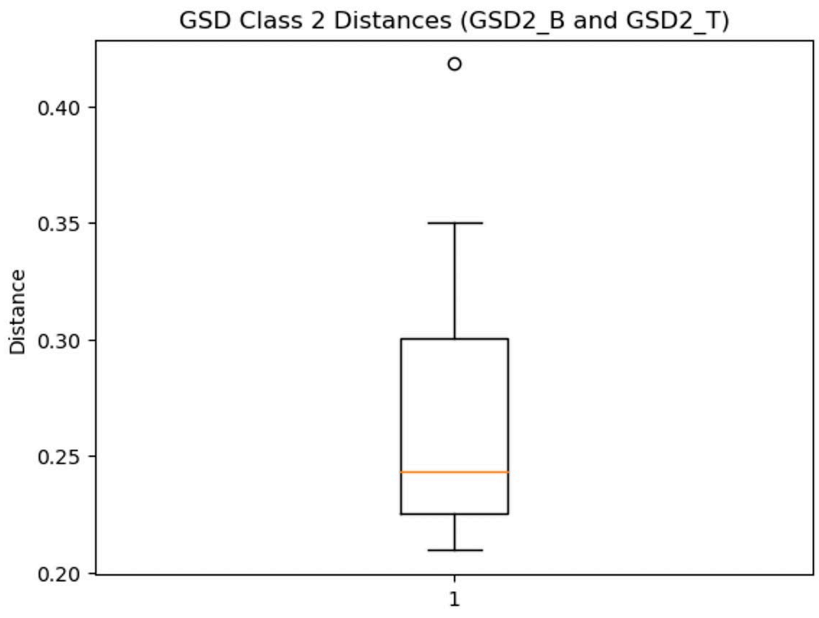


Figure 69: ResNet101, RMS, Class 2, GSD Test

Interestingly, this model has classified one instance as an anomaly. Upon closer inspection of this image, we see that it is significantly different from other images in the training set (Fig 69), which is the reason the model classifies it as anomalous.

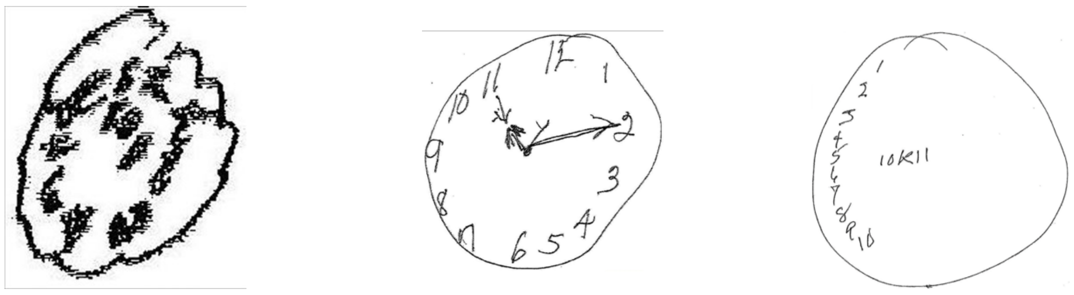


Figure 70: Class 2 Anomalous Image w/ GSD Comparison

Next, we overlay the 10 anomalous images from class 3 onto Fig 68 to produce Fig 69. Now, we can see that 6/10 of the anomalous images fall under Q3 of Tnet_2, showing that they should be classified as Class 2 images and have been miscoded as Class 3 images. There are 4 images that still fall outside the limit of Tnet_2 and are still considered anomalies of Class 2. This may be because they are of worse quality than Class 2 and would be considered Class 0 or Class 1 images instead.

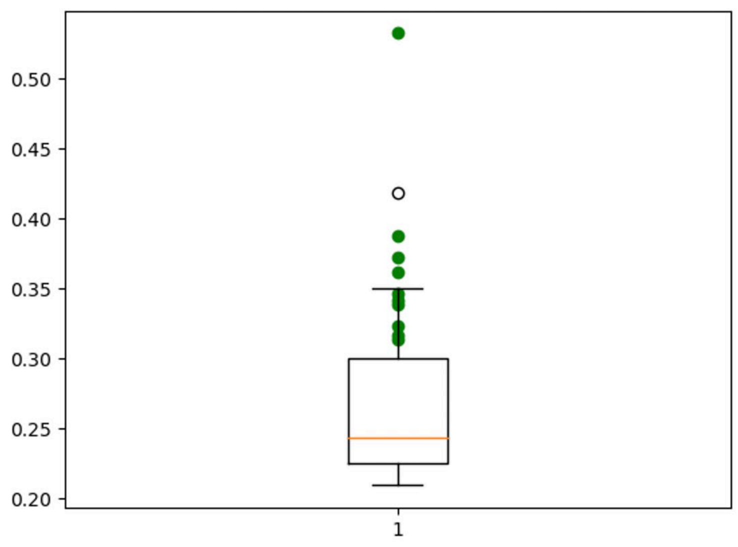


Figure 71: ResNet101, RMS, Class 2 Overlay of Anomalous Images

The four images which are considered anomalies of class 2 are shown in Fig 72.

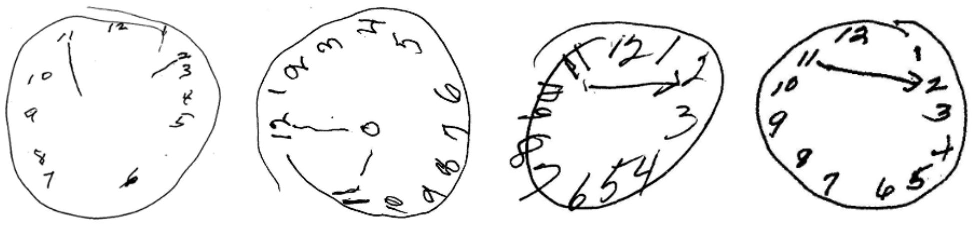


Figure 72: 4 Remaining Images Mis-detected

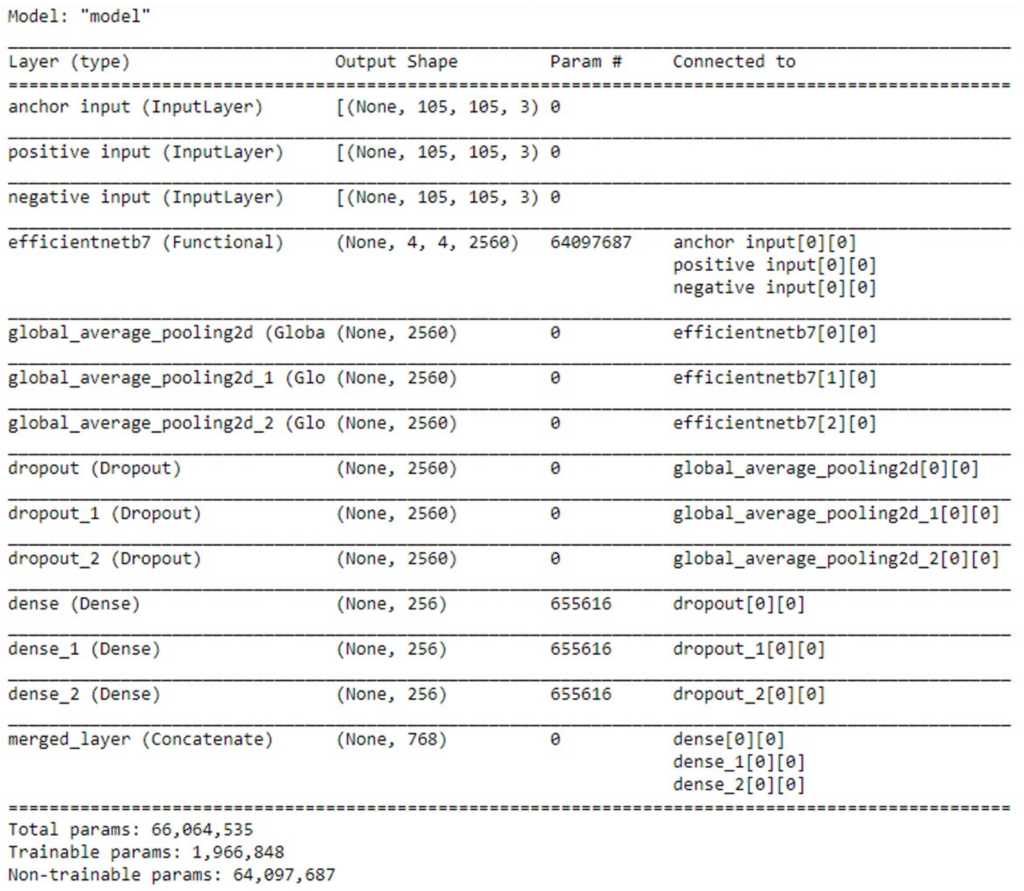
Section 4.6.1.4 Five Number Summaries ResNet101

Table 4: FNS ResNet101, RMS

Class	Q1	Q2	Q3	Max	Gamma
2	0.225	0.243	0.301	0.350	0.415
3	0.278	0.299	0.320	0.366	0.383
4	0.224	0.243	0.267	0.331	0.331

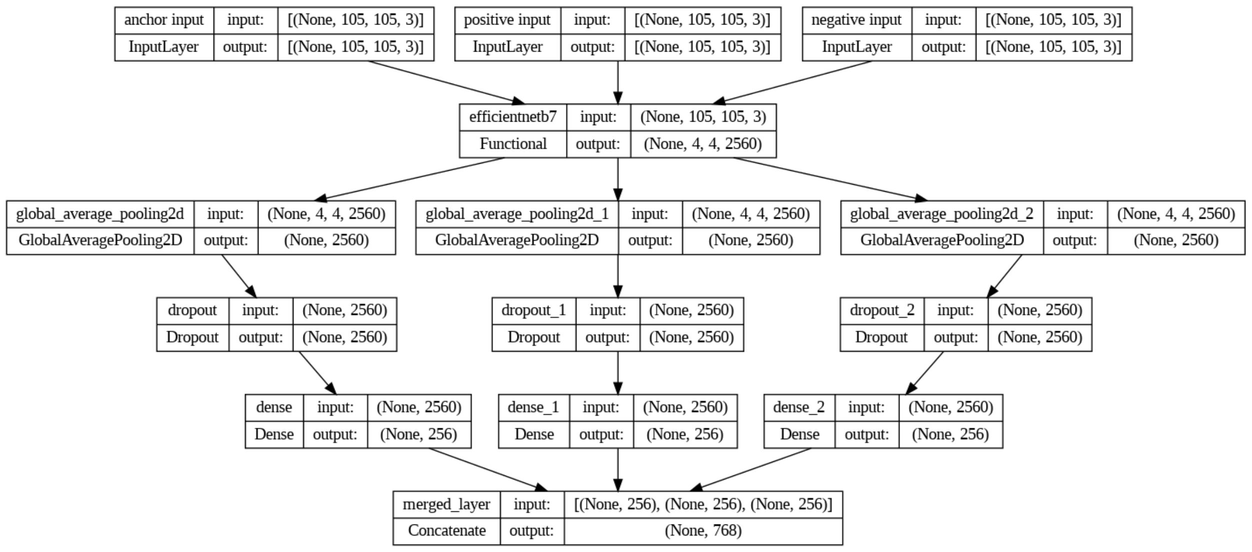
Section 4.6.2 EfficientNetB7 Results

Next, we decide to train the model using EfficientNetB7 instead of ResNet101 to see if there is any difference in performance and model size. For the purpose of this project, we decide to use EfficientNetB7, the latest and highest performing version of EfficientNetB7. An identical version of the code is used, with identical parameters, the only difference being the base architecture imported. The final model summary for Tefficientnet_3 is shown in Fig 73, with the model pipeline flow in Fig 74.



```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
anchor input (InputLayer)       [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
positive input (InputLayer)     [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
negative input (InputLayer)     [(None, 105, 105, 3) 0
__________________________________________________________________________________________________
efficientnetb7 (Functional)     (None, 4, 4, 2560)   64097687    anchor input[0][0]
                                                                 positive input[0][0]
                                                                 negative input[0][0]
__________________________________________________________________________________________________
global_average_pooling2d (Globa (None, 2560)         0           efficientnetb7[0][0]
__________________________________________________________________________________________________
global_average_pooling2d_1 (Glo (None, 2560)         0           efficientnetb7[1][0]
__________________________________________________________________________________________________
global_average_pooling2d_2 (Glo (None, 2560)         0           efficientnetb7[2][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 2560)         0           global_average_pooling2d[0][0]
__________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 2560)         0           global_average_pooling2d_1[0][0]
__________________________________________________________________________________________________
dropout_2 (Dropout)             (None, 2560)         0           global_average_pooling2d_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 256)          655616      dropout[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          655616      dropout_1[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 256)          655616      dropout_2[0][0]
__________________________________________________________________________________________________
merged_layer (Concatenate)      (None, 768)          0           dense[0][0]
                                                                 dense_1[0][0]
                                                                 dense_2[0][0]
==================================================================================================
Total params: 66,064,535
Trainable params: 1,966,848
Non-trainable params: 64,097,687
```

Figure 73: EfficientNetB7 Model Summary



anchor input | input: | [(None, 105, 105, 3)]
InputLayer | output: | [(None, 105, 105, 3)]
positive input | input: | [(None, 105, 105, 3)]
InputLayer | output: | [(None, 105, 105, 3)]
negative input | input: | [(None, 105, 105, 3)]
InputLayer | output: | [(None, 105, 105, 3)]
efficientnetb7 | input: | (None, 105, 105, 3)
Functional | output: | (None, 4, 4, 2560)
global_average_pooling2d | input: | (None, 4, 4, 2560)
GlobalAveragePooling2D | output: | (None, 2560)
global_average_pooling2d_1 | input: | (None, 4, 4, 2560)
GlobalAveragePooling2D | output: | (None, 2560)
global_average_pooling2d_2 | input: | (None, 4, 4, 2560)
GlobalAveragePooling2D | output: | (None, 2560)
dropout | input: | (None, 2560)
Dropout | output: | (None, 2560)
dropout_1 | input: | (None, 2560)
Dropout | output: | (None, 2560)
dropout_2 | input: | (None, 2560)
Dropout | output: | (None, 2560)
dense | input: | (None, 2560)
Dense | output: | (None, 256)
dense_1 | input: | (None, 2560)
Dense | output: | (None, 256)
dense_2 | input: | (None, 2560)
Dense | output: | (None, 256)
merged_layer | input: | [(None, 256), (None, 256), (None, 256)]
Concatenate | output: | (None, 768)

Figure 74: EfficientNetB7 Model Layer Flow

Section 4.6.2.1 Class 3

We started by retraining the model for Class 3 using the same training set. The model trained for 100 epochs, with losses decreasing from 14.01 and 12.89 to 0.0035 and 0.0034 for training and validation losses respectively (Fig 75).

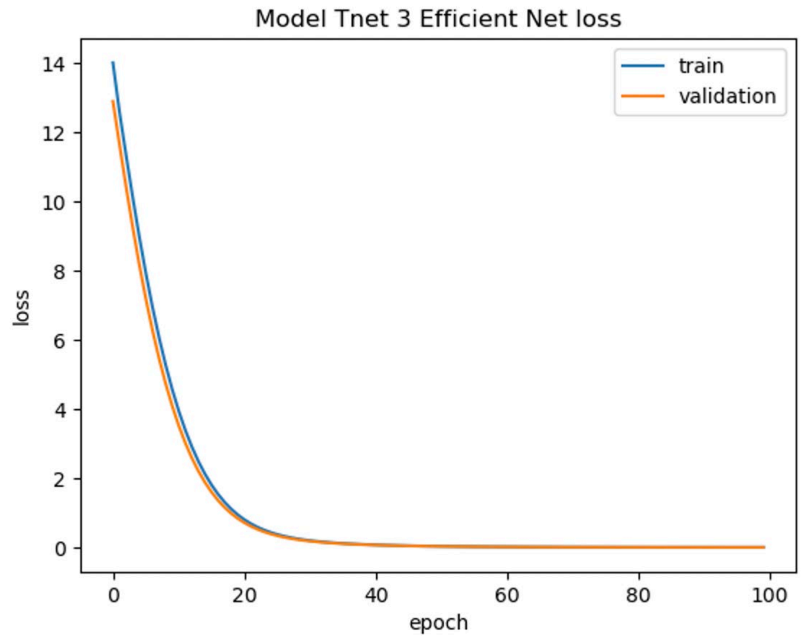


Figure 75: EfficientNetB7 Optimal Model Loss Graph

Next, we ran identical tests on the Tefficientnet_3 as we did in Tnet_3. The first test was to run the model on the GSD Class 3 set: with half of the images being from GSD3_B, and the other half being from GSD3_T. The output is shown in Fig 76.

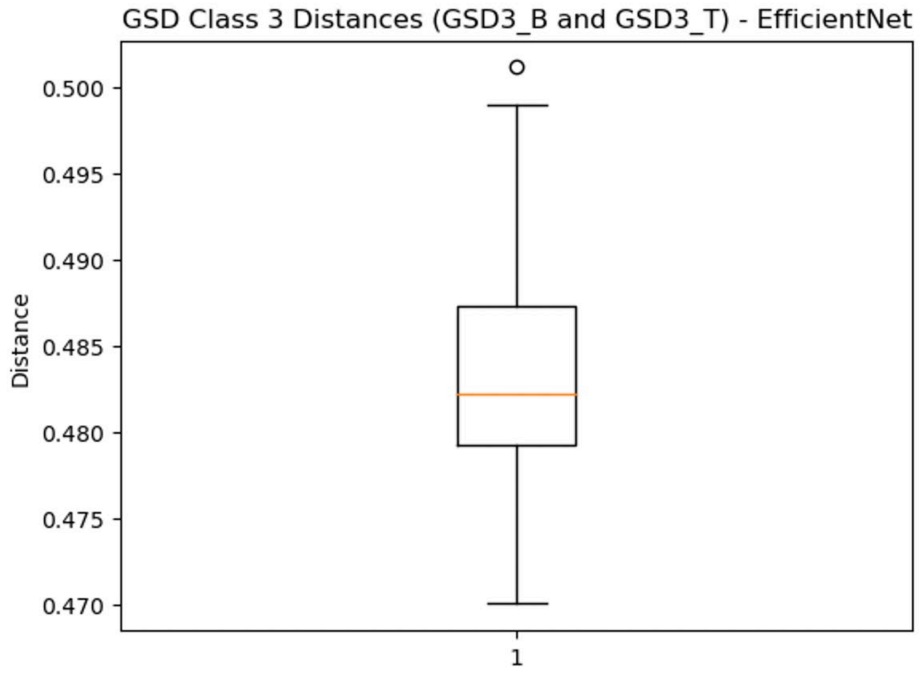


Figure 76: EfficientNetB7, RMS, Class 3, GSD Test

From the result, we can see that the EfficientNetB7 model does not perform as well as ResNet101 since it detects one of the images as an anomaly, although it is known that none of the images in the GSD set are anomalous (an effect correctly predicted by ResNet101). This image is shown in Fig 77.

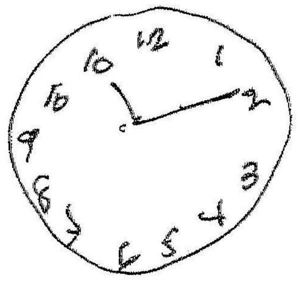


Figure 77: Anomalous Image Detected Incorrectly

After generating the FNS (reported in Table 5), we run the second test: finding known anomalies. The same folder of images (GSD3_T + 3 images from class 1) are run through the model, with their final result shown in Fig 78.

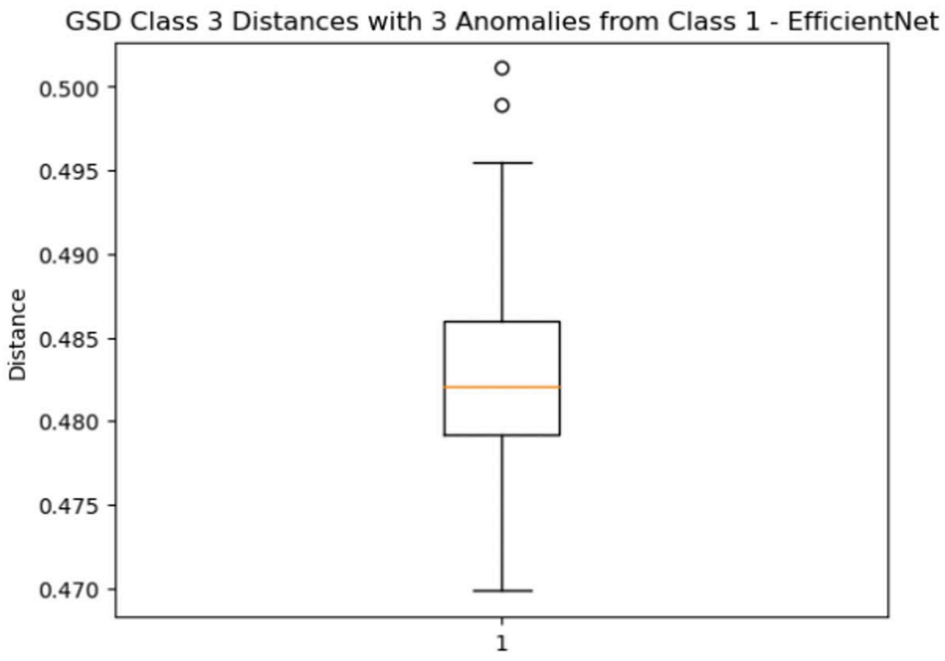


Figure 78: EfficientNetB7, RMS, Class 3, GSD + Anomalies

From this result, we can again see that two out of three anomalies have been detected. Upon closer inspection, however, we see that only one of these reported anomalies corresponds to the "correct" anomaly in the folder, with the other reported anomaly being the same incorrectly reported anomaly as in Fig 72. These two images are shown in Fig 79. From here, we can see that using EfficientNetB7 as our base architecture leads to incorrectly predicted results.

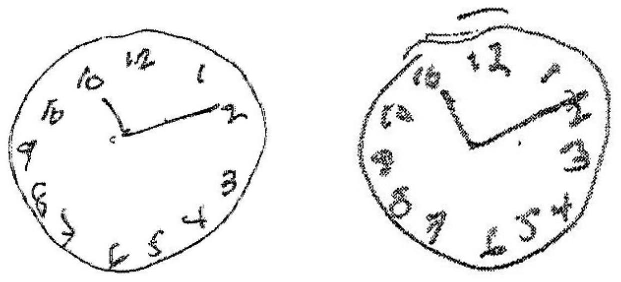


Figure 79: Anomalous Images Detected Incorrectly

Lastly, we run test 3 to see which anomalous values are predicted by this model. As we can see in Fig 80, the model detects 6 outliers, most of which are detected below the minimum value in the boxplot. Further, we know that images that are detected below the lowest whisker are not considered anomalous at all: rather, they are considered as one of the best results (most similar) as their RMS distance is very low (i.e., they are closer to each other in the vector space).

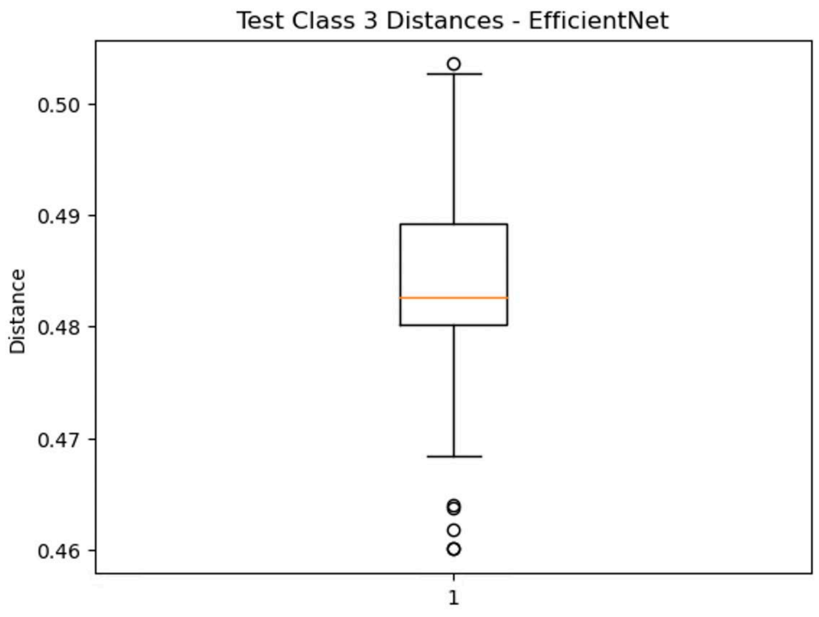


Figure 80: EfficientNetB7, RMS, Class 3 Test Set

Upon further inspection, these 6 anomalous values are shown below. We can see that none of these 6 images match the 10 images found by Tnet_3. Further, we can see that these 6 images are quite similar to those that should be classified as Class 3 images, and hence should not be considered anomalous in the first place. We can hence conclude that using EfficientNet as the backbone architecture is not accurate in detecting anomalies in class 3, and these anomalies need not be fed into Tefficientnet_4 or Tefficient_2.

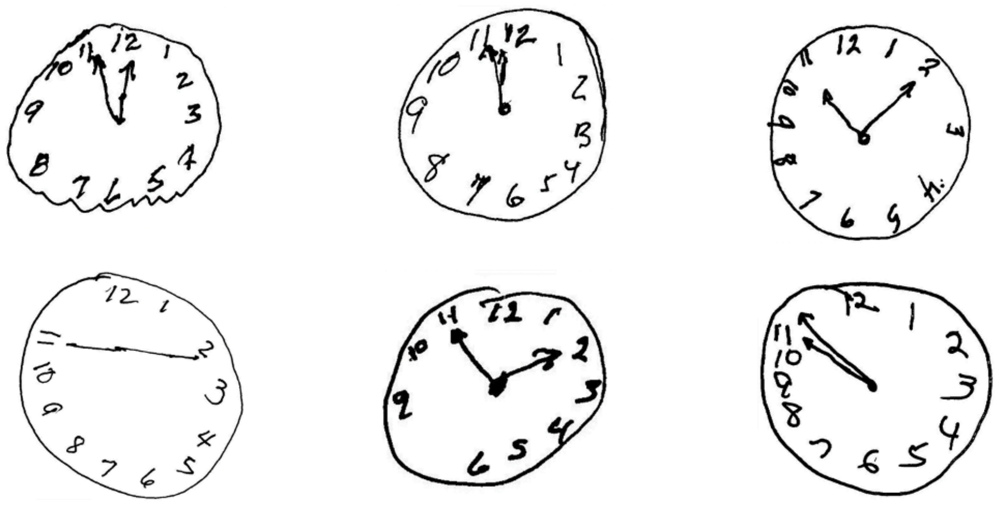


Figure 81: EfficientNetB7, Anomalous Images Class 3

Section 4.6.2.2 Class 4

In order to generate the FNS, we train the model using the same dataset as used in Tnet_4. The batch size had to be decreased to 16 as the kernel kept crashing with the high batch size. This may be because EfficientNetB7 has a higher number of parameters than ResNet101. We can see that this model trained for 100 epochs with losses decreasing from 13.89 and 12.88 to 0.0053 and 0.0061 (Fig 82).

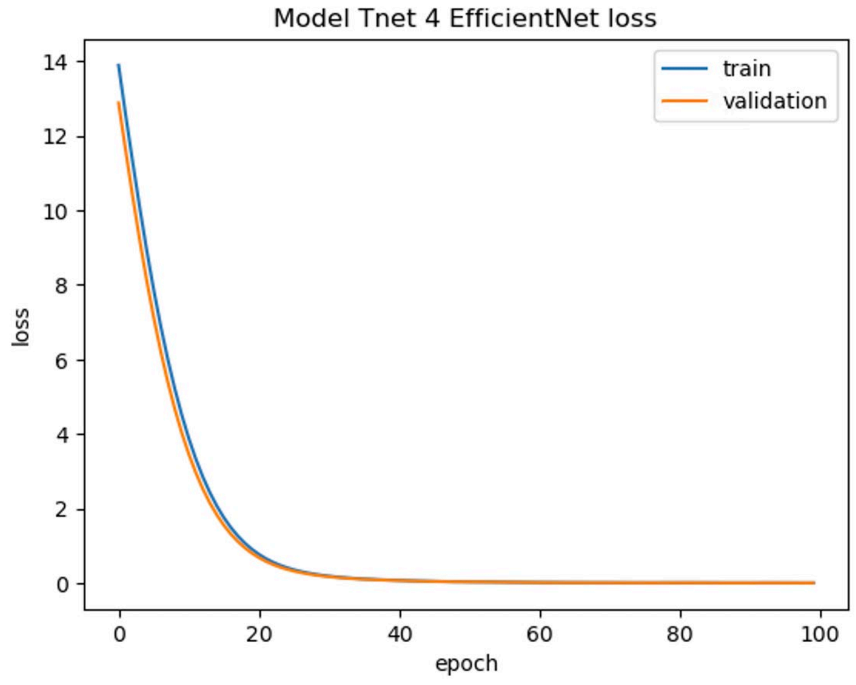


Figure 82: EfficientNetB7 Class 4 Loss Graph

Next, the divided GSD 4 dataset is run through the module, which should ideally produce zero anomalies (Fig 83). As we can see, the model incorrectly finds 7 anomalous images in the GSD dataset, a constant inconsistency in the EfficientNetB7 architecture (as it incorrectly detects anomalies in GSD 3 and GSD 2).

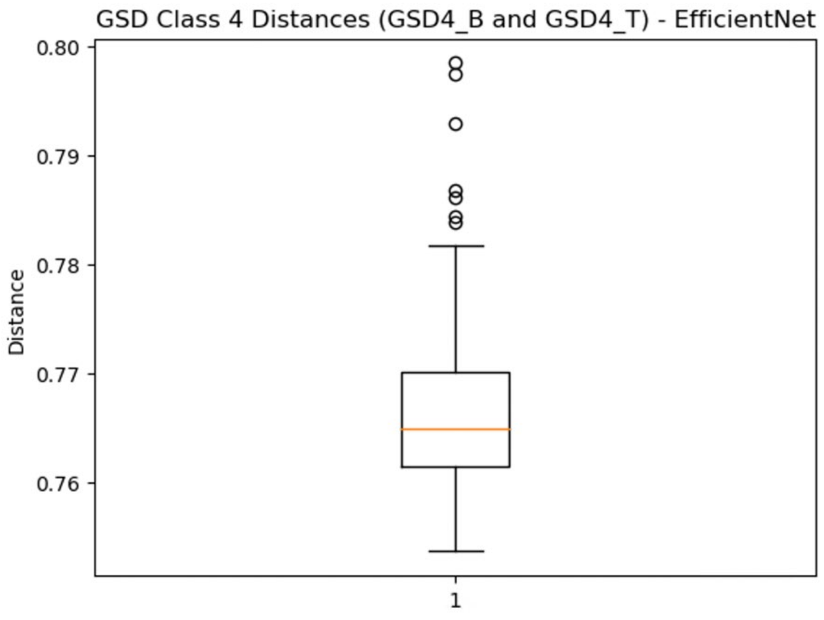


Figure 83: EfficientNetB7, RMS, Class 4, GSD Test

Section 4.6.2.3 Class 2

In order to generate the FNS, we train the model using the identical dataset used while training Tnet_2. The batch size was decreased to 100 as the kernel again kept crashing. The model then ran over 100 epochs with losses decreasing from 13.96 and 12.90 to 0.0046 and 0.0044 for training and validation losses respectively (Fig 84).

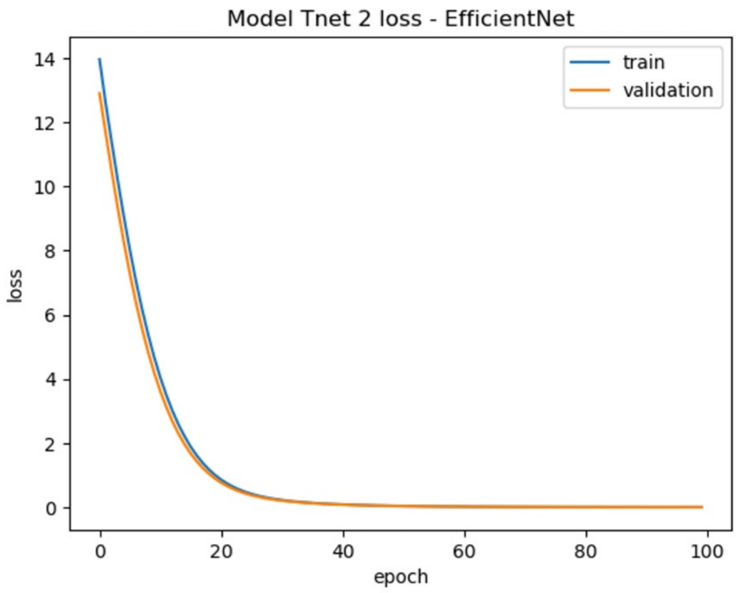


Figure 84: EfficientNetB7, Class 2 Loss Graph

This generated the boxplot seen in Fig 85, which takes half of the images from the GSD 2 set as GSD2_B (the base set) and GSD2_T (the test set). Ideally, this should produce zero anomalies, but as we can see, it produces a skewed graph with two anomalous values.

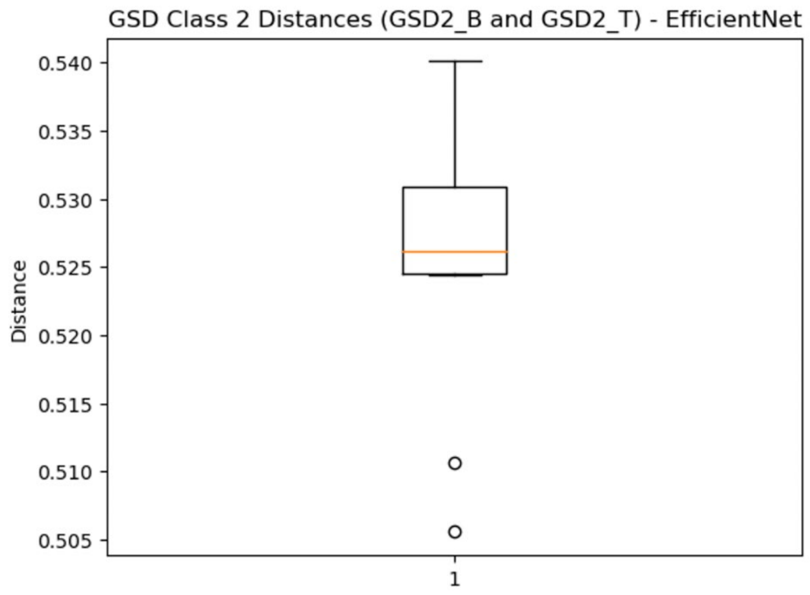


Figure 85: EfficientNetB7, RMS, Class 2, GSD Test

Section 4.6.2.4 Five Number Summaries (EfficientNetB7)

Table 5: FNS EfficientNetB7

Class	Q1	Q2	Q3	Max	Gamma
2	0.525	0.526	0.531	0.540	0.540
3	0.479	0.482	0.487	0.499	0.499
4	0.761	0.765	0.770	0.782	0.784

Section 4.7 Test Results (Euclidean Distance)

Section 4.7.1 ResNet101

Section 4.7.1.1 Class 3

Finally, the model was trained to compare the Euclidean distance metric in the triplet loss function. The final model loss is shown in Fig 86, with the losses decreasing from 153.14 and 13.45 to 1.48 and 1.48 for training and validation losses respectively.

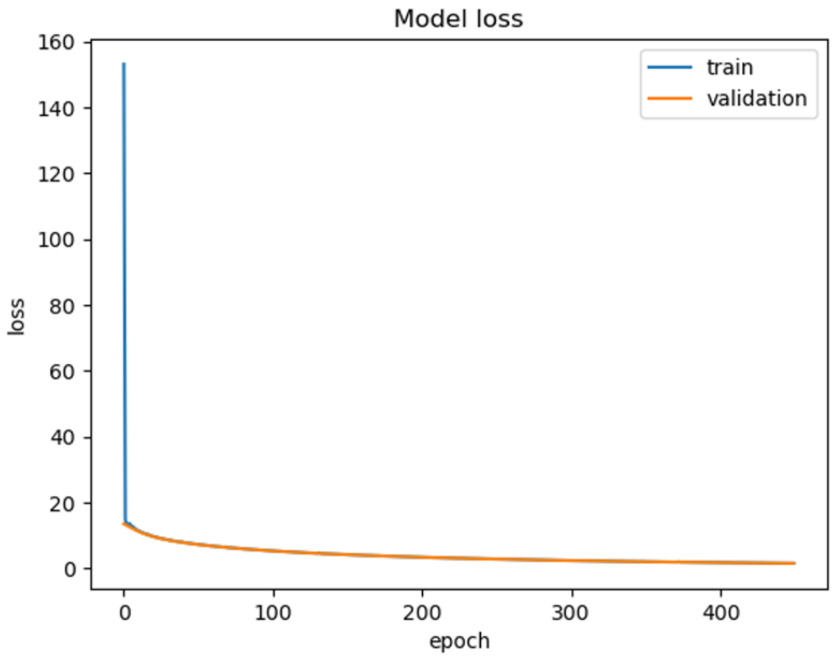
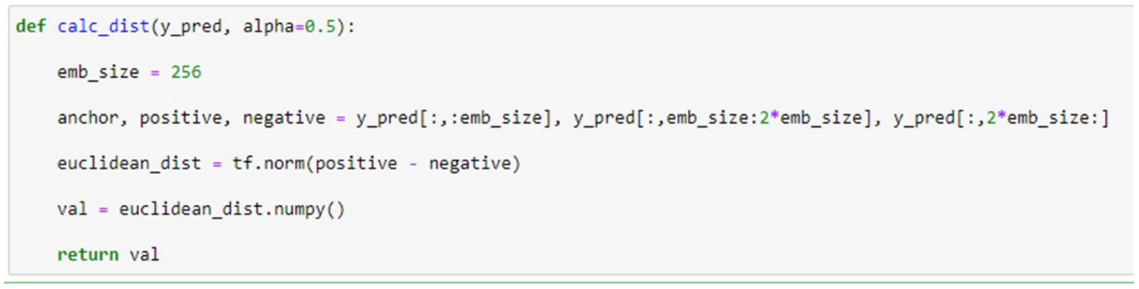


Figure 86: ResNet101, Euclidean Distance, Class 3 Loss Graph

Identical tests were run on this model. For reference, the Euclidean norm is shown in Eq 34.

$$\text{euclidean\_distance} = \sqrt{|f(p)^2 - f(n)^2|} \qquad \text{Equation 34}$$

This is calculated using the tf.norm function, from which the value is extracted using the .numpy() function.



```
def calc_dist(y_pred, alpha=0.5):

    emb_size = 256

    anchor, positive, negative = y_pred[:,:emb_size], y_pred[:,emb_size:2*emb_size], y_pred[:,2*emb_size:]

    euclidean_dist = tf.norm(positive - negative)

    val = euclidean_dist.numpy()

    return val
```

Figure 87: Calculating Euclidean Distance Function (Code)

As before, the first test is run between the GSD3_B and GSD3_T images. As with the RMS metric, the boxplot correctly shows zero anomalous values detected (Fig 88).

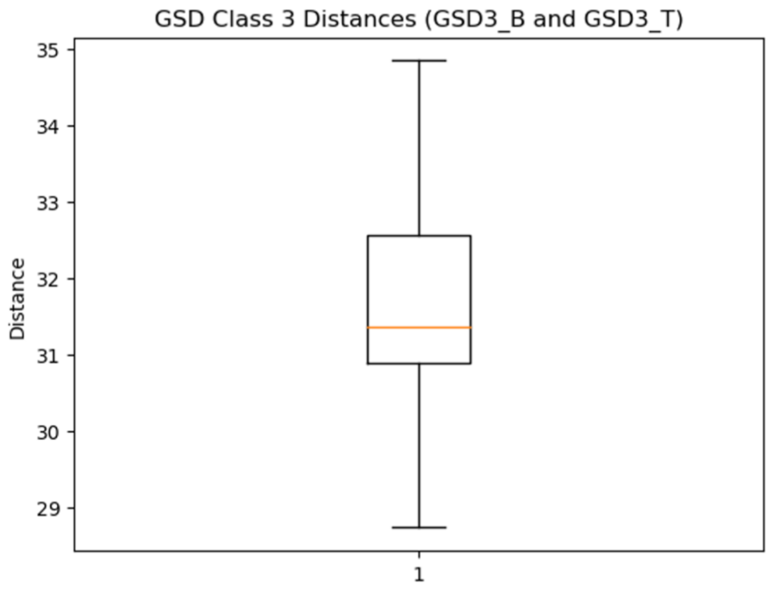


Figure 88: ResNet101, Euclidean, Class 3, GSD Test

In test 2, we run the model through GSD3_B and GSD3_T which contains the same GSD images as before with an additional 3 images from Class 1. Similar to RMS, the model was able to detect the same two out of three known anomalies.

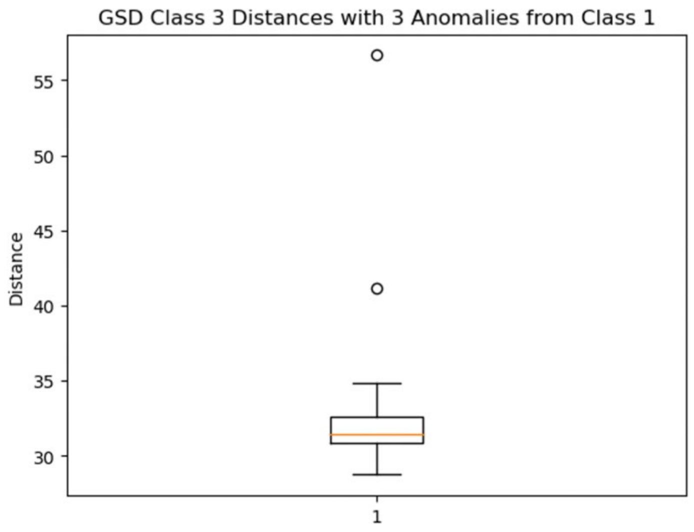


Figure 89: ResNet101, Euclidean, Class 3, GSD + Anomalies

Finally, we run the model on the class 3 test set. The output boxplot is shown in Fig 90, which has detected 19 anomalies. From these 19 anomalies, 10 are identical to the ones detected by our model when using RMS.

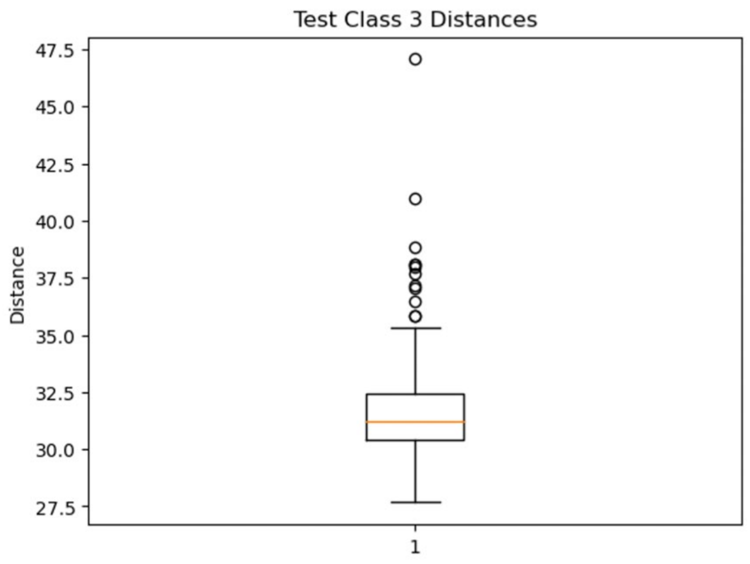


Figure 90: ResNet101, Euclidean, Class 3 Test Set

These 19 images are shown in Fig 91. As we can see, 10 of these images are repeated (in a red border), proving that the RMS value had indeed detected them as correctly being anomalous.

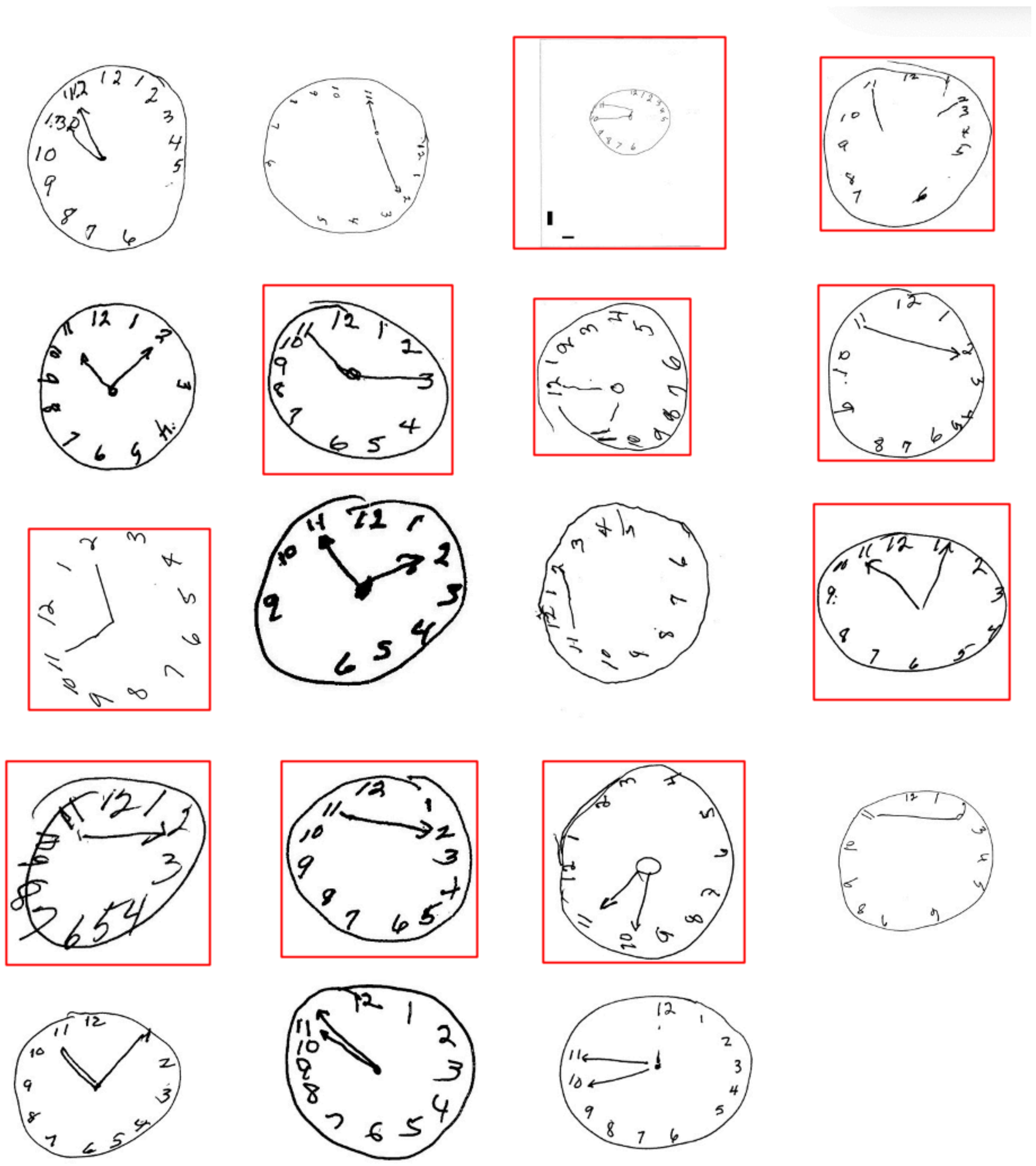


Figure 91: ResNet101, Euclidean, Anomalous Images Class 3

From the remaining 9 images, we can see that some of them are correctly classified as anomalous, whereas others may have been correctly coded. Since the loss at the end of training was around 1.5, we can conclude that further training would help generate a more accurate model in terms of anomaly detection.

Section 4.7.1.2 Class 4

Next, we generated the FNS for class 4 by splitting GSD 4 into two sets (base and test) and pairing them with one another. This generated the final loss curve and boxplot shown in Fig 92 and 93. The loss decreases from 183.71 and 13.46 to 1.67 and 1.67 for training and validation loss respectively.

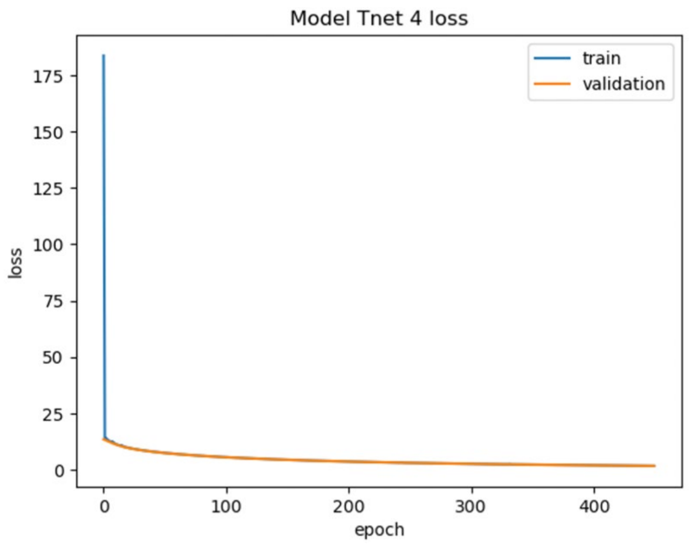


Figure 92: ResNet101, Euclidean, Class 4 Loss Graph

From the GSD plot, we can see that the model does not perform perfectly when using Euclidean distances as it detects two incorrect anomalous images (Fig 94). This may be because the model's final loss after 450 epochs is still a high value of around 1.67.

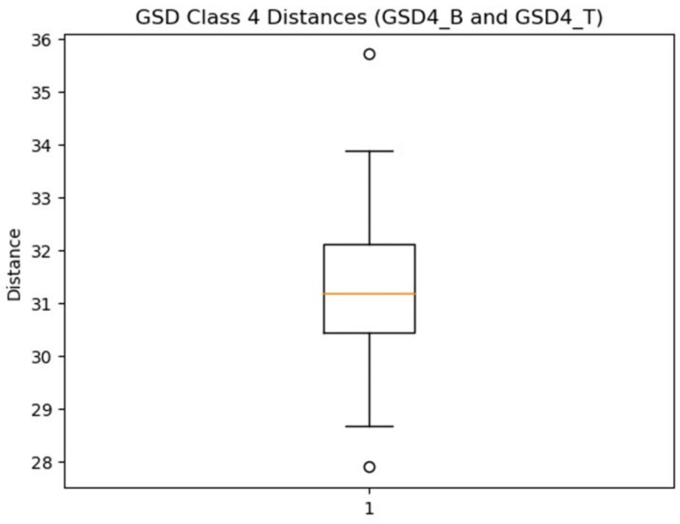


Figure 93: ResNet101, Euclidean, Class 4, GSD Test

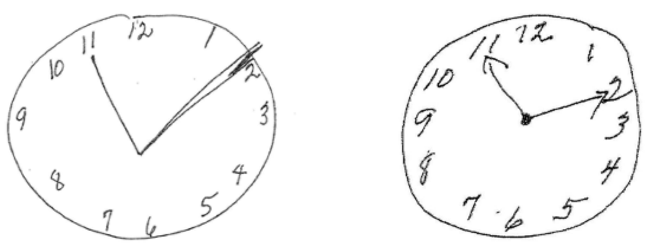


Figure 94: ResNet101, Euclidean, Class 4 Mis-detected Anomalies

Next, we fed these 19 images into the class 4 model to check whether or not they belong to class 4. The result shows that every image is correctly classified as anomalous to class 4, which, as earlier, would be correct as the CDT is classified in an ordinal fashion. Images anomalous to 3 which are of worst quality would not classify as class 4 images (Fig 95).

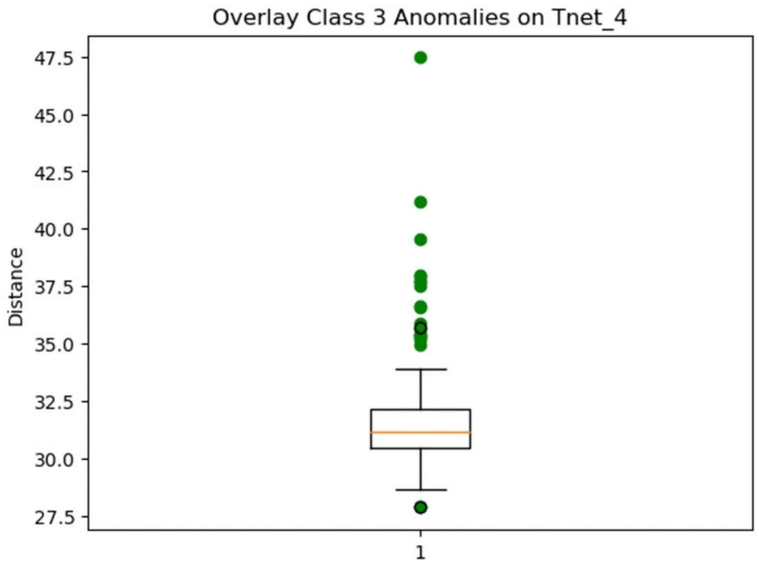


Figure 95: ResNet101, Euclidean, Class 4 Overlay of Anomalous Images

However, we notice an interest misclassification at around the 27.9 value. One of the incorrectly-labeled anomalous GSD images has a Euclidean distance of around 27.9 (Fig 97), which is around the same value given to one of the anomalous class 3 images (Fig 96). This tells us that our model can still be improved to become one that ensures all GSD 4 images are properly encoded (i.e., no anomalies).

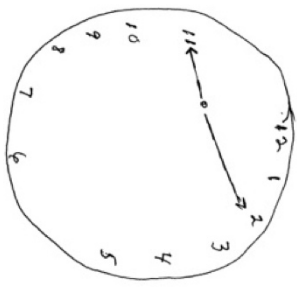


Figure 96: Class 3 Anomalous Image w/ d ≈ 27.9

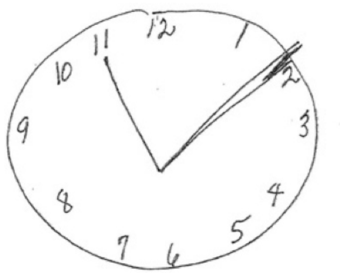


Figure 97: Class 4 GSD Anomalous-Labeled Image w/ d ≈ 27.9

Section 4.7.1.3 Class 2

Finally, we run the model for class 2 to generate the FNS. The final loss graph is shown in Fig 98, which decreases in training and validation losses from around 139.38 and 13.47 to 1.37 and 1.37 respectively. As we can see, the loss still does not decrease to a value below 1, which is room for detection error in our model. We perform the same steps as before: dividing the GSD class into two (GSD_B and GSD_T), making triplets and passing them into the model. This generates the boxplot shown in Fig 99, showing one anomaly.

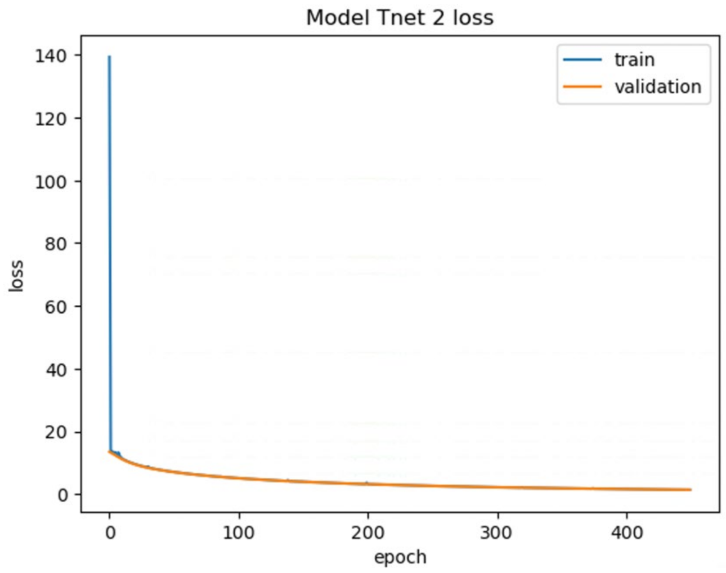


Figure 98: ResNet101, Euclidean, Class 2 Loss Graph

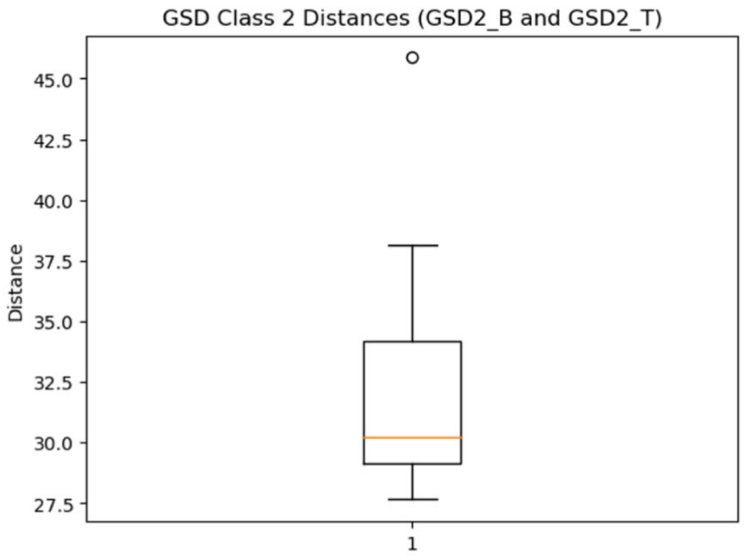


Figure 99: ResNet101, Euclidean, Class 2, GSD Test

Interestingly, this anomaly is identical to the anomaly detected by the RMS ResNet101 model (Fig 100). This may be because of the low quality of this image compared to the other images in the GSD dataset, and it can be improved by adding more images of this quality into our training dataset.

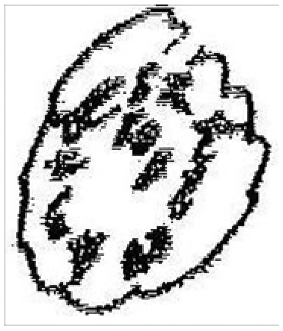


Figure 100: Class 2 Anomaly

Next, we ran the 19 anomalous images from Class 3 into the Class 2 model. This produced the boxplot shown in Fig 101. Here, we can see that there are four points which are located above or below the whiskers. The two images which have a distance smaller than the lower whisker are not considered anomalous as their Euclidean distance is actually closer together than the median Euclidean distances of the test images. This means that they are actually more likely to belong to Class 2. These two images are shown in Fig 102, where we can see that the model has correctly classified them as Class 2 images.

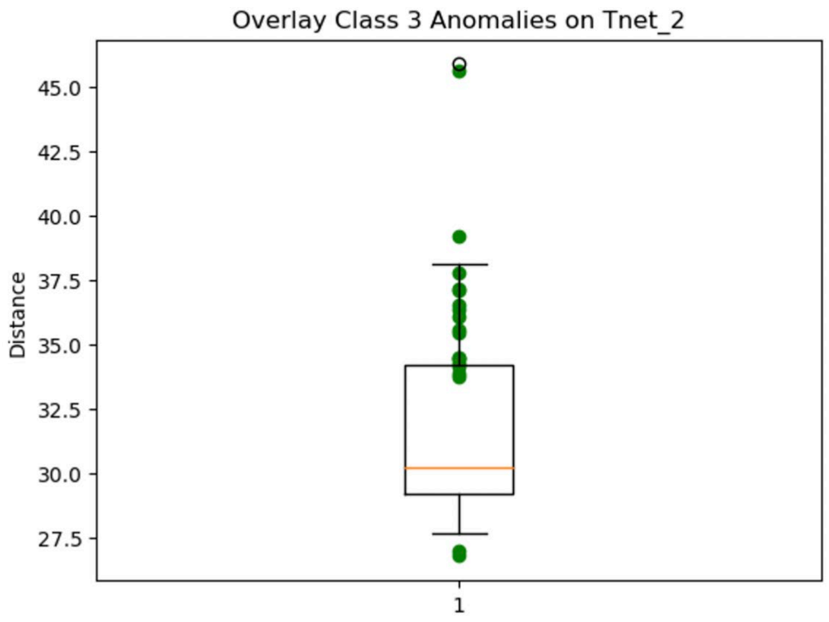


Figure 101: ResNet101, Euclidean, Class 2 Overlay of Anomalous Images

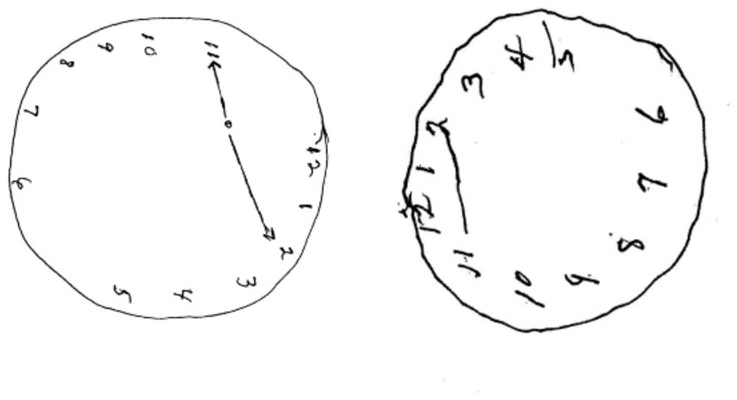


Figure 102: ResNet101, Euclidean, Remaining Images Correctly Classified

The next two anomalies are located above the upper whisker, also known as our Γ value. Anything located above this value is considered an anomaly. These two images are shown in Fig 102.

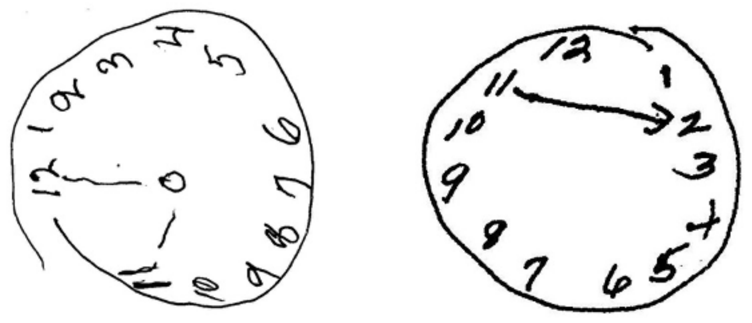


Figure 103: ResNet101, Euclidean, Remaining Images Mis-detected

While comparison these images to our GSD images, we can see that these are correctly not classified as Class 2 images as they are of higher quality. These images have most likely been mis-detected by our model as anomalies of class 3, when they have been correctly coded as class 3 by coder 103. Finally, the five number summaries of models using the Euclidean distance metric is shown in Section 4.6.

Section 4.7.1.4 Five Number Summaries (Euclidean Distance)

Table 6: FNS ResNet101, Euclidean Distance

Class	Q1	Q2	Q3	Max	Γ
2	29.16	30.23	34.16	45.86	38.13
3	30.89	31.36	32.57	47.07	34.85
4	30.45	31.19	32.13	47.45	33.88

Section 4.8 Siamese Network Class 3 Results

Finally, our last comparison would occur with our benchmark work done by our research group, which performs anomaly detection on the same dataset using Siamese networks. After running Snet_3 on the test class 3, we obtained the following 18 anomalies (Fig 103).

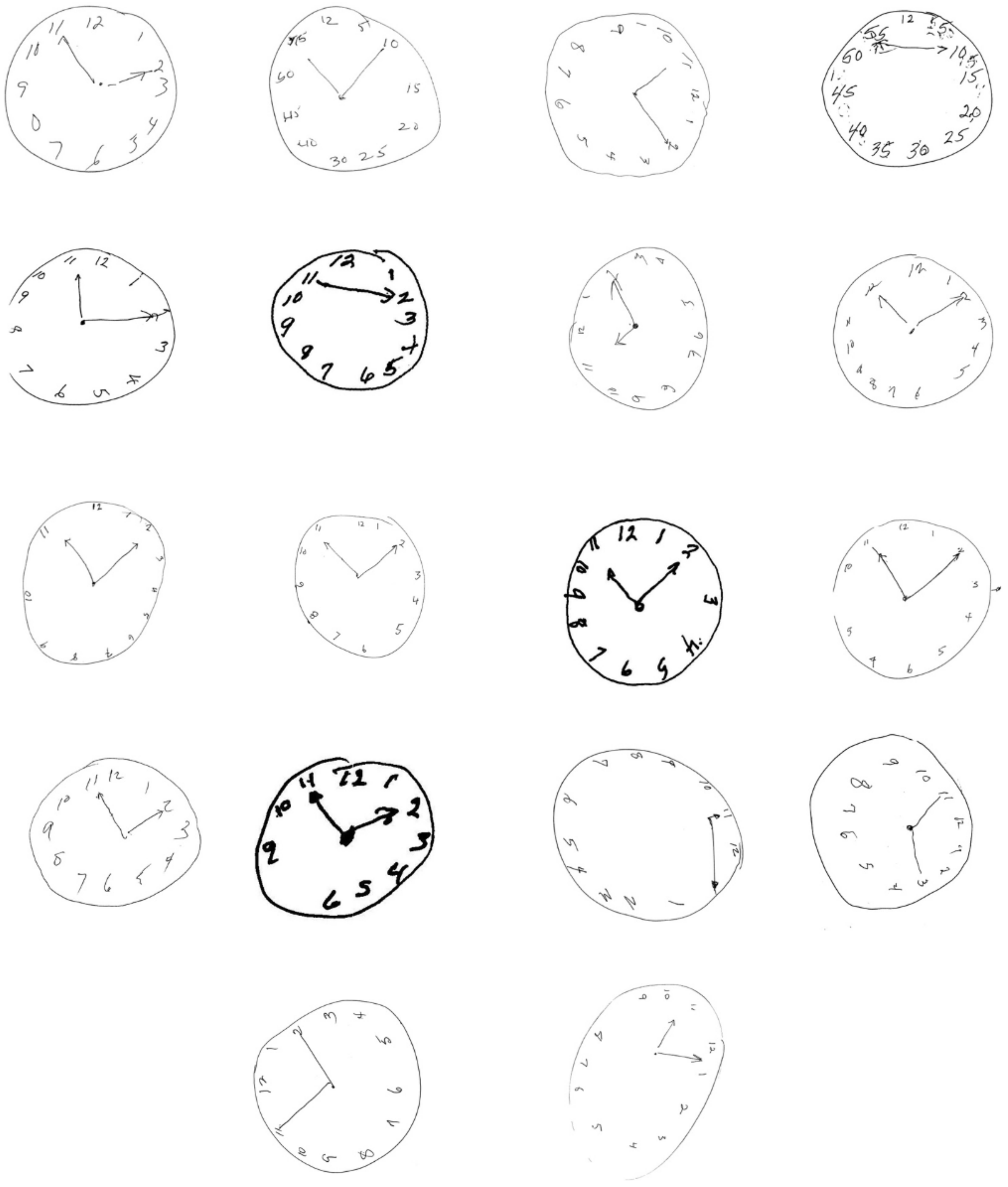


Figure 104: Siamese Network Class 3 Anomalies

Section 4.9 Comparisons

Section 4.9.1 Model Comparison

There are a couple of differences seen when using different architectures in training our anomaly detection model. Firstly, the number of trainable parameters in ResNet101 was 1,573,632 while in EfficientNetB7 it was 1,966,848 parameters. Further, the total number of parameters in EfficientNetB7 was of the order of 66 million, while ResNet101 was of the order of 44 million. Due to this larger number of parameters in EfficientNetB7, we can see that the final saved model is not smaller than the ResNet101 model; 190 MB versus 281 MB using EfficientNetB7. Because of this, we cannot say that the final model is more compact when using EfficientNetB7, as our ResNet101 model is significantly smaller.

Secondly, in terms of training time, our EfficientNetB7 model took a longer amount of time, with around 280 seconds per step per epoch, while it only took around 190 seconds per step per epoch when using ResNet101. This came up to around 15 hours and 33 minutes in total for EfficientNetB7, compared to only 10 hours and 33 minutes for the ResNet101 model. This was effectively due to the smaller number of total parameters in the ResNet101 model.

Finally, in terms of performance, we see that the EfficientNetB7 model is significantly worse than ResNet101: the GSD class 3 set successfully showed no anomalous values when using ResNet101 but displayed one anomaly when using EfficientNetB7. Further, the final distances between images using ResNet101 were closer together (had a much smaller value of around 0.3) than EfficientNetB7 (which were mostly of the order of around 0.5), which can be seen in the FNS of both architectures. The model was also unable to detect the intentional anomalies positioned, as it only detected one correct anomaly when using EfficientNetB7 (and one incorrect anomaly from the GSD set) but detected two correct anomalies when using ResNet101.

For these reasons, we can conclude that the best model architecture for our use case is ResNet101, as it is successfully able to distinguish between normal and abnormal data with reasonable accuracy, as is shown from our multiple tests with each class.

Section 4.9.2 Metric Comparison

Next, we will compare the two metrics used: RMS and Euclidean Distance. Although both metrics were able to detect anomalies in the test set, the number and quality of images were significantly different.

When using the RMS metric, we detected 10 anomalous images, of which 6 were classified as Class 2 images. The remaining 4 images were concluded to either be mis-detected by the model, or to belong to a lower class. The value of Γ was also much lower: 0.366.

In terms of time, the RMS ResNet101 took around 10 hours and 33 minutes to run (190 seconds per step per epoch), whereas Euclidean distance ResNet101 took around 390 seconds per step per epoch, adding up to 97 hours and 30 minutes. This is a huge difference in time owing to the more complicated calculation performed by the loss function, and due to the larger number of epochs taken to converge. Further, since the final loss metric was still larger than 1, we would need to train for a longer duration to have a comparable final metric value, which means that our final time taken to train would be a lot larger than 97 hours and 30 minutes.

When using the Euclidean distance metric, we detected 19 anomalous images, of which 10 were identical to the ones detected using RMS. From these 19, 17 were detected as Class 2 images, with two remaining images concluded as being mis-detected as anomalous by the model. The value of Γ was a higher number of 34.85.

Although the Euclidean distance metric was able to detect more anomalous images, the quality of these 19 images were not all lower than Class 3. A few of these images look mis-detected to the human eye. Further, the model was detected anomalous images in the GSD test of class 4. This gives us reason to believe that the Euclidean distance metric is not the optimal choice for this dataset in its current form. This may be because the final loss was 1.475 after 450 epochs, which is significantly higher than the 0.028 final loss value when using RMS.

For these reasons, we conclude that, with the current hyperparameters, the RMS metric outperforms the Euclidean distance metric. Further experimentation should be done to bring the Euclidean metric loss down to around 0.028 to compare the number and quality of anomalous images detected.

Section 4.9.3 Comparison with Siamese Network

Upon closer inspection of the anomalous images, we see that none of them are identical to the ones detected by our triplet network, even though they were comparing images from the same sets of data. Before highlighting a possible reason why the images are different, we will first compare the two networks and highlight their architectural differences.

First, the Siamese network compares two images during training which are randomly taken from the training dataset (which may contain anomalies). It passes these into a ResNet101 model with an embedding size of 256 (identical to our triplet network), after performing the same conversions (resizing, random flipping, and converting into a tensor). Once the embeddings are obtained, they are used to calculate the contrastive loss between them, which uses Euclidean distance instead of RMS. The model trained using a learning rate of 0.0005 (much lower than ours) for a period of 500 epochs (much longer than ours) to have a loss that decreased from around 7 to around 1.5 units in training loss, a value that is significantly higher than the triplet network with the RMS metric, but similar to the Euclidean distance metric (Fig 104).

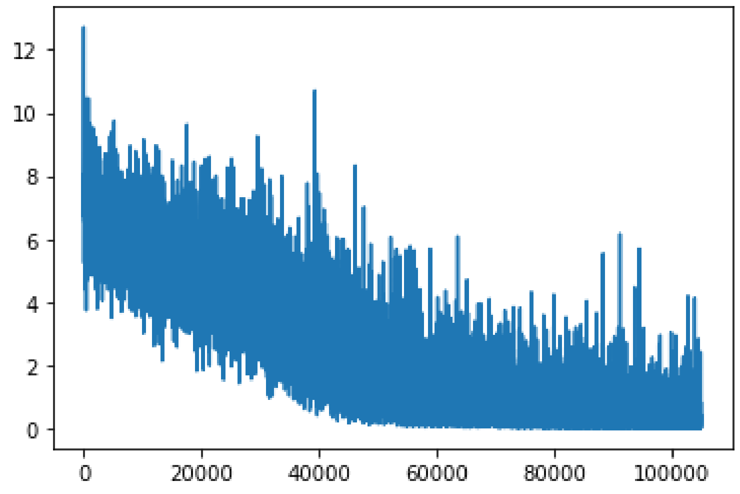


Figure 105: Siamese Network Loss Graph

The model parameters are then saved and reloaded for testing. Upon testing with the same test folder, we obtain the boxplot shown in Fig 105, corresponding to the anomalous images shown in Section 4.3.4.

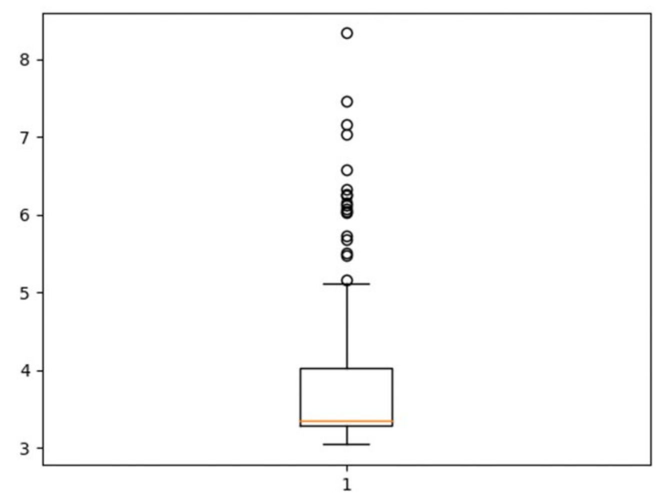


Figure 106: Siamese Network Class 3 Test Set

The reason that the anomalies may be inaccurate is because of the model run on Test 2 (known anomalies). In test 2, the model takes in two inputs: the GSD Class 3 (base folder) and the GSD Class 3 (test folder) + 3 anomalies from Class 1. The resulting boxplot is shown in Fig 106, which shows one anomaly detected by the network (Fig 107), which is not one of the class 1 images (it is one of the GSD images). Henceforth, we can conclude that our model outperforms the existing Siamese network on this dataset.

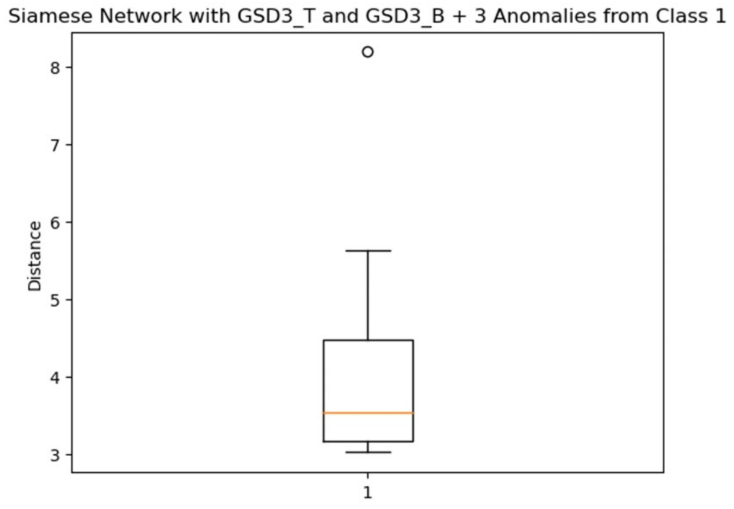


Figure 107: Siamese Network, Class 3, GSD + Anomalies

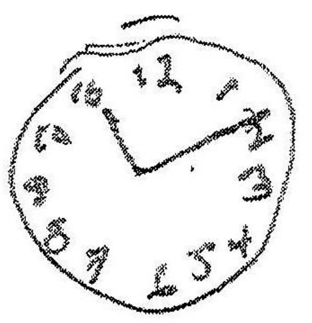


Figure 108: Siamese Network, Incorrectly Detected GSD Anomaly

Chapter 5 Conclusion and Future Work

In conclusion, in this project we aimed to perform anomaly detection for the NHATS Clock Drawing Test (CDT) dataset for Class 3 images. We modified several hyperparameters such as learning rate, type of optimizer, batch size, the margin (alpha) value, the embedding size and the number of epochs. We also compared our ResNet101 model with the popular EfficientNetB7 and concluded that our model outperforms one based on this architecture for this dataset. We also experimented in using the RMS and Euclidean distance metric and concluded that the RMS metric provides the best result for our use case. After thorough hyperparameter tuning and comparisons between different base architectures and metrics, we were able to successfully create this system by finding the optimal model. The final optimal model we have obtained consists of the ResNet101 architecture with the RMS metric. Once the anomalies were detected, they were fed into the adjacent-class networks in an attempt to find their true classes. We were successfully able to reclassify 6 out of 10 of the anomalies into their true class after multiple tests on the strength of our model. The remaining 4 anomalies could be considered as incorrectly identified anomalies by the model, which is a starting point for improvements.

Future work would entail improving the detection of anomalies: to increase the detection from 2 to all 3 anomalies. This could be done by changing the base architecture or by increasing the training set size through data augmentation. Further, the ResNet101 model which uses Euclidean distance should be retrained for a longer duration to minimize the loss even further to around 0.05. This new model should then be tested to check if anomaly detection improves. Another improvement would lie in using purely GSD sets as our anchor and positive image sets, i.e., to only pick anchor-positive pairs from GSD images. This would remove the potential of out-of-distribution images in the target class from being used in training. However, as mentioned in [12], introducing out-of-distribution images in the training set led to small improvements in anomaly detection. This would mean introducing images into the target class as known anomalies to help in the detection of similar kinds of anomalies. Finally, as we can see a significant difference in the number of anomalies found depending on the type of metric used, future work could involve experimenting with more metrics to find the optimal one for this use case.

References

[1] Bellet, A., Habrard, A., & Sebban, M. (2015). Metric learning. Synthesis lectures on artificial intelligence and machine learning, 9(1), 1-151.

[2] Benhur S. (2020. A friendly introduction to Siamese Networks. Towards Data Science. Retrieved from: https://towardsdatascience.com/a-friendly-introduction-to-siamese-networks-85ab17522942

[3] Prasad K. (2020). Siamese Networks. Towards Data Science. Retrieved from: https://towardsdatascience.com/siamese-networks-line-by-line-explanation-for-beginners-55b8be1d2fc6

[4] Koch, G., Zemel, R., & Salakhutdinov, R. (2015, July). Siamese neural networks for one-shot image recognition. In ICML deep learning workshop (Vol. 2, p. 0).

[5] Suk, H. I. (2017). An introduction to neural networks and deep learning. In Deep Learning for Medical Image Analysis (pp. 3-24). Academic Press.

[6] 2017, What Are Convolutional Networks? Introduction to Deep Learning [Webinar]. MATLAB. Retrieved from: https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html

[7] Heaton, J. Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning. Genet Program Evolvable Mach **19**, 305–307 (2018). https://doi.org/10.1007/s10710-017-9314-z

[8] Mishra M. (2020). Convolutional Neural Networks, Explained. Towards Data Science. Retrieved from: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

[9] I. Melekhov, J. Kannala and E. Rahtu, "Siamese network features for image matching," 2016 23rd International Conference on Pattern Recognition (ICPR), 2016, pp. 378-383, doi: 10.1109/ICPR.2016.7899663.

[10] Dong, X., & Shen, J. (2018). Triplet loss in siamese network for object tracking. In Proceedings of the European conference on computer vision (ECCV) (pp. 459-474).

[11] Das S. (2019). Image similarity using Triplet Loss. Towards Data Science. Retrieved from: https://towardsdatascience.com/image-similarity-using-triplet-loss-3744c0f67973

[12] Masana, M., Ruiz, I., Serrat, J., van de Weijer, J., & Lopez, A. M. (2018). Metric learning for novelty and anomaly detection. arXiv preprint arXiv:1808.05492.

[13] Kimin Lee, Honglak Lee, Kibok Lee, and Jinwoo Shin. Training confidence-calibrated classifiers for detecting out-of-distribution samples. In Int. Conference on Learning Representations (ICLR), 2018.

[14] Shiyu Liang, Yixuan Li, and R. Srikant. Enhancing the reliability of out-of-distribution image detection in neural networks. In Int. Conference on Learning Representations (ICLR), 2018.

[15] Zhe Zhu, Dun Liang, Songhai Zhang, Xiaolei Huang, Baoli Li, and Shimin Hu. Trafficsign detection and classification in the wild. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 2110–2118, 2016.

[16] Ahn, J. Y., & Kim, G. (2022). Application of optimal clustering and metric learning to patch-based anomaly detection. Pattern Recognition Letters, 154, 110-115.

[17] Yi, J., & Yoon, S. (2020). Patch svdd: Patch-level svdd for anomaly detection and segmentation. In Proceedings of the Asian Conference on Computer Vision.

[18] Aliakbarisani, R., Ghasemi, A., & Wu, S. F. (2019). A data-driven metric learning-based scheme for unsupervised network anomaly detection. Computers & Electrical Engineering, 73, 71-83.

[19] Staar, B., Lütjen, M., & Freitag, M. (2019). Anomaly detection with convolutional neural networks for industrial surface inspection. Procedia CIRP, 79, 484-489.

[20] Hoffer, E., & Ailon, N. (2015, October). Deep metric learning using triplet network. In International workshop on similarity-based pattern recognition (pp. 84-92). Springer, Cham.

[21] Wang, J., Zhou, F., Wen, S., Liu, X., & Lin, Y. (2017). Deep metric learning with angular loss. In Proceedings of the IEEE international conference on computer vision (pp. 2593-2601).

[22] Guillaumin, M., Verbeek, J., & Schmid, C. (2009, September). Is that you? Metric learning approaches for face identification. In 2009 IEEE 12th international conference on computer vision (pp. 498-505). IEEE.

[23] Hendrycks, D., & Gimpel, K. (2016). A baseline for detecting misclassified and out-of-distribution examples in neural networks. arXiv preprint arXiv:1610.02136.

[24] F. Schroff, D. Kalenichenko and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 2015, pp. 815-823, doi: 10.1109/CVPR.2015.7298682.

[25] G. Kumar and P. K. Bhatia, "A Detailed Review of Feature Extraction in Image Processing Systems," 2014 Fourth International Conference on Advanced Computing & Communication Technologies, 2014, pp. 5-12, doi: 10.1109/ACCT.2014.74.

[26] Image Recognition with Transfer Learning (98.5%), n.d.: https://thedatafrog.com/en/articles/image-recognition-transfer-learning/

[27] Tan, M., & Le, Q. (2019, May). Efficientnet: Rethinking model scaling for convolutional neural networks. In International conference on machine learning (pp. 6105-6114). PMLR.

[28] Du S. (2020). Metric Learning Using Siamese and Triplet Convolutional Neural Networks. Retrieved from: https://levelup.gitconnected.com/metric-learning-using-siamese-and-triplet-convolutional-neural-networks-ed5b01d83be3

[29] Pinto, E., & Peters, R. (2009). Literature review of the Clock Drawing Test as a tool for cognitive screening. Dementia and geriatric cognitive disorders, 27(3), 201-213.

[30] Youn, Y. C., Pyun, J. M., Ryu, N., Baek, M. J., Jang, J. W., Park, Y. H., ... & Kim, S. Y. (2021). Use of the Clock Drawing Test and the Rey–Osterrieth Complex Figure Test-copy with convolutional neural networks to predict cognitive impairment. Alzheimer's research & therapy, 13(1), 1-7.

[31] Feng, X., Zou, Q., Zhang, Y., Tang, Y., Ding, J., & Wang, X. (2020, December). Clock Drawing Test Evaluation via Object Detection for Automatic Cognitive Impairment Diagnosis. In 2020 IEEE 6th International Conference on Computer and Communications (ICCC) (pp. 1229-1234). IEEE.

[32] Lin, M., Chen, Q., & Yan, S. (2013). Network in network. arXiv preprint arXiv:1312.4400.

[33] Dropout layer (n.d.). Keras Documentation. Retrieved from: https://keras.io/api/layers/regularization_layers/dropout/

[34] Bonnin, R. (2017). Machine Learning for Developers: Uplift your regular applications with the power of statistics, analytics, and machine learning. Packt Publishing Ltd.

[35] Keras – Dense Layer (n.d.). Tutorialspoint. Retrieved from: https://www.tutorialspoint.com/keras/keras_dense_layer.htm

[36] Dense layer (n.d.). Keras Documentation. Retrieved from: https://keras.io/api/layers/core_layers/dense/

[37] Keras Regularization (n.d.). Educba. Retrieved from: https://www.educba.com/keras-regularization/

[38] Layer weight regularizers (n.d.). Keras Documentation. Retrieved from: https://keras.io/api/layers/regularizers/

[39] Alake R. (2020). Regularization Techniques And Their Implementation In TensorFlow (Keras). Retrieved from: https://towardsdatascience.com/regularization-techniques-and-their-implementation-in-tensorflow-keras-c06e7551e709

[40] Zhang R. (2022). Create a Siamese Network with Triplet Loss in Keras. Retrieved from: https://zhangruochi.com/Create-a-Siamese-Network-with-Triplet-Loss-in-Keras/2020/08/11/

[41] Brownlee J. (2019). Understand the Impact of Learning Rate on Neural Network Performance. Retrieved from: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

[42] Varikuti M. (2022). What Is the Effect of Batch Size on Model Learning? Retrieved from: https://pub.towardsai.net/what-is-the-effect-of-batch-size-on-model-learning-196414284add

[43] Zakharov, Sergey & Kehl, Wadim & Planche, Benjamin & Hutter, Andreas & Ilic, Slobodan. (2017). 3D Object Instance Recognition and Pose Estimation Using Triplet Loss with Dynamic Margin. 552-559. 10.1109/IROS.2017.8202207.

[44] Ikram,S.,Cherukuri,A.,Poorva,B.,Ushasree,P.,Zhang,Y.,Liu,X. & Li,G.(2021).Anomaly Detection Using XGBoost Ensemble of Deep Neural Network Models. Cybernetics and Information Technologies,21(3) 175-188. https://doi.org/10.2478/cait-2021-0037

[45] Adam (n.d.). Keras Documentation. Retrieved from: https://keras.io/api/optimizers/adam/

[46] Kingma, Diederik & Ba, Jimmy. (2014). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations.

[47] Srinivasan A. (2019). Stochastic Gradient Descent – Clearly Explained !! Retrieved from: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

[48] Raschka S. (n.d.). Machine Learning FAQ. Retrieved from: https://sebastianraschka.com/faq/docs/gradient-optimization.html

[49] Bushaev V. (2017). Stochastic Gradient Descent with momentum. Retrieved from: https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d

[50] Kousalya, K., Mohana, R. S., Jithendiran, E. K., Kanishk, R. C., & Logesh, T. (2022). Prediction of Best Optimizer for Facial Expression Detection using Convolutional Neural Network. https://doi.org/10.1109/iccci54379.2022.9740832

[51] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

[52] RMSprop (n.d.). Keras Documentation. Retrieved from: https://keras.io/api/optimizers/rmsprop/

[53] Gandhi R. (2018). A Look at Gradient Descent and RMSprop Optimizers. Retrieved from: https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b#:~:text=RMSprop%20Optimizer&text=The%20RMSprop%20optimizer%20restricts%20the,how%20the%20gradients%20are%20calculated.

[54] Wolfe C. (2021). Why 0.9? Towards Better Momentum Strategies in Deep Learning. Retrieved from: https://towardsdatascience.com/why-0-9-towards-better-momentum-strategies-in-deep-learning-827408503650#:~:text=In%20deep%20learning%2C%20most%20practitioners,many%20popular%20deep%20learning%20packages).

[55] Keany E. (2019). Siamese Network with Triplet Loss/MachinePart1.ipynb. Retrieved from: https://github.com/Ekeany/Siamese-Network-with-Triplet-Loss/blob/master/MachinePart1.ipynb

[56] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).