4-18-2023 10:30 AM

Anomaly Detection on Partial Point Clouds for the Purpose of Identifying Damage on the Exterior of Spacecrafts

Kaitlin T. Hutton, *Western University*

Supervisor: McIsaac, Kenneth, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Electrical and Computer Engineering

Recommended Citation

Hutton, Kaitlin T., "Anomaly Detection on Partial Point Clouds for the Purpose of Identifying Damage on the Exterior of Spacecrafts" (2023). *Electronic Thesis and Dissertation Repository*. 9256.
https://ir.lib.uwo.ca/etd/9256

Abstract

The Canadarm3 is going to operate autonomously aboard the Lunar Gateway space station for the purpose of inspections and repairs. To make the repairs, damage to the spacecraft needs to be detected accurately and automatically. This research investigates methods for training Machine Learning (ML) models on 3D point clouds to identify anomalous structural damage. The PointNet algorithm was used to train models on point clouds without affecting their structure. The optimal training data style was found by comparing how well the different styles of data performed at classifying the point cloud testing data. Two different methods of anomaly detection were tested and compared; statistical anomaly detection based on classification scores and anomaly detection using an autoencoder. The autoencoder method proved superior and achieved a recall score of 90.42% with a specificity of 79.31% and a classification score of 97.93%. This showed the potential to use an autoencoder on 3D point clouds for anomalous damage detection on the exterior of spacecrafts.

Keywords

Machine Learning, Point Clouds, Anomaly Detection, Object Classification, PointNet.

Summary for Lay Audience

The robotic arm Canadarm3 is being built by Canadian company MDA to perform repairs while operating on the Lunar Gateway space station. Since the arm is being built for both remote and autonomous operation, there needs to be a way to automatically identify damaged parts of the spacecraft exterior so that it will know what parts need to be repaired. Scans of the exterior of the Gateway will be taken using a sensor on the end of the Canadarm3 that produces point clouds, which are 3D representations of an object. This research investigates potential Machine Learning models and methods for training them on 3D point clouds for the purpose of automatically identifying anomalous structural damage.

Various methods for applying deep learning models on point cloud data were researched, and the most promising model architectures were identified and selected for further investigation. To perform the investigation, a procedure for generating training and testing data, both ‘normal’ and ‘damaged’, was developed. The data were generated for simple geometric objects as representations of components which make up the Gateway, with the conjecture that if the models and training would not work on simple objects, they would not work on the more complex spacecraft structures. Multiple styles of training data were tested and compared to each other to determine which style of data provided the best model results.

With the optimal training data style determined, two models were then tested and compared: a classification only model and a multi-output model. The classification only model classified and labeled each point cloud of a simple object, representing a spacecraft component. To evaluate its performance, a statistical method was used to predict if the point cloud contained damage based upon how confident the model was in the label it assigned to the object. The multi-output model had one output which classified the object and another output that reconstructed the point cloud. The reconstructed point cloud was compared against the original point cloud and the difference between them was the reconstruction error. Point clouds with larger reconstruction errors contained damage. The results from each model were compared and the model that achieved the best results was then tuned using hyperparameter tuning to create the best model possible. The results from this optimal model were then analyzed to assess it as a potential solution for automatically detecting real spacecraft damage.

Acknowledgments

Thank you to my supervisor Dr. Ken McIsaac for helping me navigate through this thesis and for providing his support along the way.

Thank you to MDA for bringing this problem to our lab and for allowing me to perform research on something so interesting.

And finally, thank you to my parents Joe and Geni and my sister Jackie. Without you this thesis would not exist.

Table of Contents

List of Tables

List of Figures

List of Appendices

1 Introduction

1.1 Lunar Gateway and the Canadarm

The Lunar Gateway (Figure 1-1) is a space station with a planned lunar orbit that is being developed as the successor to the International Space Station (ISS). The United-States, Canada, and other ISS partners are creating the Gateway as a part of the National Aeronautics and Space Administration's (NASA) Artemis campaign which is focused on scientific exploration and discovery on and around the Moon. The station will include a scientific laboratory and crew quarters for astronauts who will make crewed trips to the lunar surface. Unlike the ISS, the Gateway will not be crewed continuously and will have periods where it is being operated remotely [1]. As part of that remote operation, the robotic arm Canadarm3 (Figure 1-2) is being developed by Canadian company MDA to facilitate any repairs needed on the station.

Figure 1-1: Artist rendering of the Lunar Gateway Space Station [1]

Canadarm3 is the third robotic arm in the Canadarm series. The original Canadarm was built for the NASA Space Shuttle program while Canadarm2 has been in operation aboard the ISS since 2001. Canadarm2 is primarily used for repairing the ISS and for guiding vehicles into the station [2]. Like Canadarm2, Canadarm3 will have seven degrees of freedom and will move end-over-end to access all areas of the Gateway space station [3]. One of the main developments of Canadarm3 over its predecessors is the incorporation of Artificial Intelligence (AI)-based robotic systems to allow for the arm to operate autonomously. This will allow for the arm to continue to operate while there are no astronauts on the Gateway, and when the station has orbited to the far side of the Moon and is out of contact with the Earth.

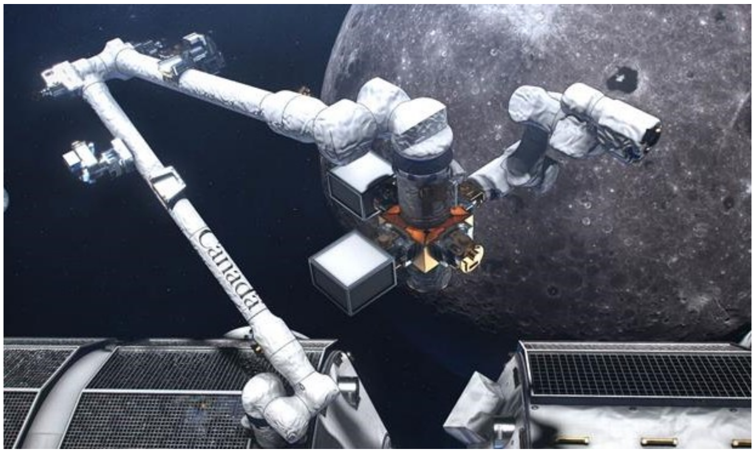


Figure 1-2: Artist rendering of the Canadarm3 [2]

One of the objectives of the operations of the Canadarm3 is for it to be able to autonomously identify structural damage on the exterior of the station for repairs. A LiDAR (Light Detecting and Ranging) sensor will be one of the tools that can be mounted to the end of the arm and used to periodically take scans of the Gateway**.** The scans, which produce 3D point clouds, will need to be processed in such a way as to detect damage automatically, and with a high degree of confidence.

1.2 Problem Definition and Assumptions

This research takes the premise outlined in the previous section and investigates the possibility of using Machine Learning (ML) as a method of automatically detecting damage using the point clouds. To be successful, the model must be able to identify what part or piece of the space station it is looking at and if that section or part is damaged. Key assumptions for this research are that there will never be an unknown object on the space station, and each part of the space station can be dissected into simpler individual components (such as antennas, solar arrays, panels, hatches, and other such objects). This means that anything anomalous can immediately be considered to be a result of damage instead of it being something that is intact but unknown.

Another assumption for this project is that the 3D scans will only contain partial views of the spacecraft components. The reasons for this are that the scanner on the Gateway will not be scanning a component from all angles, a scan may only contain a fraction of a component in its field of view (FOV), and parts of the scan may be occluded (shadowed) by other objects. Hence, any solution needs to be able to identify the component of the spacecraft and detect damage from only partial, potentially occluded scans. Additionally, it is assumed that the scans contain only one object at a time, and hence, the challenge of scene segmentation is out of the scope of this project. This is a fair assumption given the size of the space station its major components, and the fact that the FOV of the LiDAR scans will be quite narrow due to the limited distance the arm can get from the surface (max 8.5 m [3]) coupled with the need to keep the point density high enough.

1.3 Contributions

This work makes the following contributions:

1. A method to develop training datasets for partial 3D point clouds of damaged objects
2. A method to develop damaged object models for the purpose of creating damaged 3D point clouds for anomaly detection
3. An object classifier for partial point clouds of damaged objects
4. An evaluation of multiple methods of anomaly detection for partial point clouds
5. A proposed technique for anomaly threshold detection

1.4 Thesis Outline

This thesis contains the following content. The necessary background content and literature review on ML is covered in Chapter 2. In Chapter 3, the methodology for the research and the procedure for data generation is explained. An optimal training data experiment is presented in Chapter 4, where the best style of training data for the rest of the ML experiments is determined. Chapter 5 compares ML experiments to see which approach is best for performing anomaly detection on point cloud data. The best method then undergoes hyperparameter tuning in Chapter 6 and the results are analyzed. Chapter 7 contains the summary and conclusion, and Chapter 8 explores future considerations.

2 Background and Literature Review

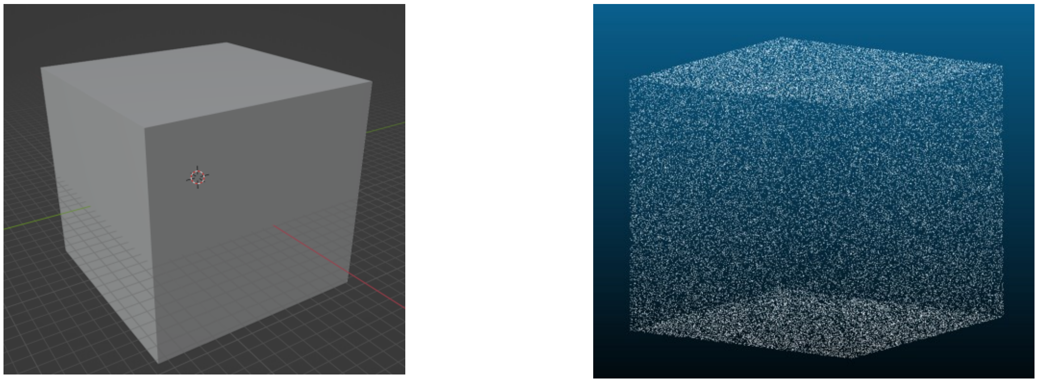
2.1 Point Clouds

2.1.1 Introduction to Point Clouds

A point cloud is a form of data representation consisting of a discrete set of points, where each point consists of an unstructured vector [4]. At its most basic, a point cloud consists only of Cartesian XYZ coordinates. However, additional information like surface normals, intensity, and RGB values can be included in point clouds depending on the point sampling method [5]. Point clouds have three main characteristics: the points are unordered, there is interaction amongst the points, and the point clouds are invariant under transformations [6]. Point clouds can be used to represent a 3D shape by sampling points along the surface of the object. This lends itself to many applications, such as 3D robotic perception, computer-aided design (CAD) modeling, geographic surveying/ mapping, and other areas where having depth information is desired. Compared to other 3D data representation methods, point clouds preserve the original geometry of the scan without discretization, thus preventing discretization error and data loss [7].

2.1.2 Generating and Simulating Point Clouds

The focus of this research is on anomaly detection and shape classification. For shape classification using point clouds, either synthetic or real-world data can be used [7]. Synthetic data consists of data generated without using a real sensor, either by simulating a sensor detecting a simulated object or by sampling points on the simulated object itself to generate the point cloud (Figure 2-1).



a) Simulated model of cube

b) Synthetic point cloud from simulated cube

Figure 2-1: Example of a synthetic point cloud from a simulated cube

The regulated environment of the synthetic data allows for more control over the resulting point clouds. For example, point clouds can be generated for the whole object without the occlusions and gaps that may occur in real-world data. This makes the data easier to work with and makes it easier to see and compare the results of various models and algorithms used to process the data. One method of synthesizing point cloud data is to use 3D mesh models. Mesh models consist of a 3D space of interconnected points which make up a shape volume [8]. Point clouds can be generated by using the geometric shapes that make up the mesh and sampling points from the surface. This can be done randomly, or by following a specific sampling algorithm, such as Poisson-disk sampling. Poisson-disk sampling involves randomly sampling points on a mesh with a uniform distribution such that the points are a user-defined minimum distance apart from one another [9]. This guarantees that two points will not be sampled in the same place.

Real-world point clouds are made by using various types of active or passive imaging sensors. These could be either LiDAR scanners, RGB-depth cameras, stereo cameras, or other 3D scanners [7]. LiDAR scanners are sensors that are commonly used in the Space sector [10]. A LiDAR creates a point cloud by sending out laser light pulses from a transmitter. The light pulses reflect upon hitting an object, and the returning light is detected by the LiDAR's receiver. The time elapsed between pulse emission and detection times the speed of light determines the range from the LiDAR to the object.

This plus information such as scan angle, intensity, and other features enable a point cloud of the LiDAR's surroundings to be constructed [11].

Real-world data (Figure 2-2) can have occlusions from objects not being fully detected, as well as background noise and other irregularities that make working with the data difficult. However, these complications reflect the conditions that are to be expected when designing something to be used in the real-world beyond theoretical research.

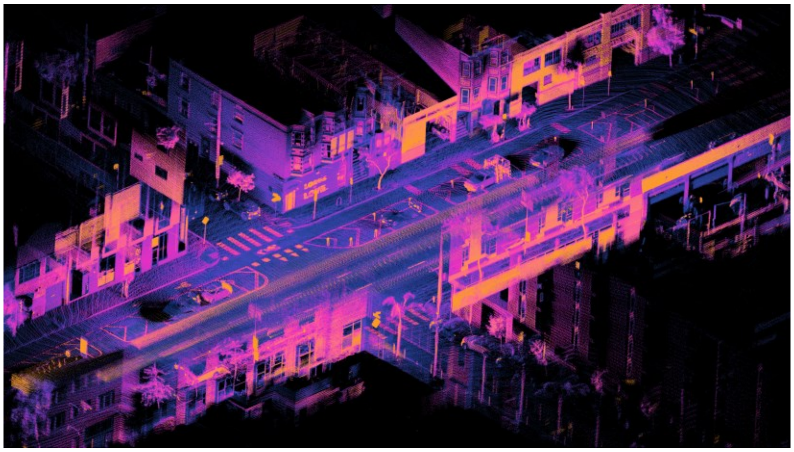


Figure 2-2: Example of a point cloud of a highway from a LiDAR scanner [12]

2.2 Deep Learning Background

2.2.1 Introduction to Machine Learning

Artificial intelligence (AI) is a thriving field in which techniques are used to try and emulate on machines the learning mechanisms that occur in the human brain and other biological organisms. There are many different problems that can be approached by artificial intelligence, the most difficult being those which are hard to formally define and which humans process intuitively [13, p. 1]. This can include things such as processing language and transcribing speech into text, identifying relevant search results, and identifying objects in images. Machine learning (ML) is the subset of AI that is used to tackle these challenges [14].

In ML, problems can be split into supervised, unsupervised, and reinforcement learning problems. Supervised learning is when the ML model is trained on input data that has a corresponding output target. This means that the target groups for the predicted outputs are known, and the model can directly assign its predictions to them. Supervised learning problems can be further split up by whether they are a regression or classification problem. Regression problems involve predicting an output that is continuous or quantitative. Predicting chemical yields or the outside temperature would be regression problems. When the output is not continuous, and instead is comprised of predicting the input as belonging to a discrete category, it is a classification problem [15, p. 3]. Recognizing handwritten digits or detecting objects in an image would be classification problems.

For unsupervised learning problems, the output targets for the model are not provided during training. The model predicts and creates output targets based upon the features it extracts from the inputs. Depending on the task of the model, the model could either identify patterns within the input data and create output clusters, or it could learn and predict the distribution of the data as a density estimation. Similarly to unsupervised learning, there are no provided output targets for reinforcement learning. Instead, reinforcement learning involves identifying the optimal targets through trial and error [15, p. 3]. The model is trying to perform the best actions which maximize a reward target, and the reinforcement comes from learning what the reward was based upon its previous actions.

Deep learning is a further subset of ML that can be supervised or unsupervised, where the model is often made up of artificial neural networks. These artificial neural networks comprise layers of "neurons" with the goal of making a model that recreates the synapses that occur between neurons in the brain. The inputs are scaled with weights which affect the computations at each neuron [16, pp. 1–2]. The depth of the deep learning model refers to how many hidden layers there are between the input and predicted output. Deep learning models are increasingly being used to tackle the ML problems discussed earlier where the goal is to have the model learn a concept that is more intuitive to a human. The recent advances in deep learning have made it the premier method for applications such as speech recognition, image recognition, predicting drug molecule activity, and predicting the effects of mutation on DNA gene expressions [14]. Deep learning is the focus of this research and will be discussed further in the following sections.

2.2.2 Training, Validation, and Testing Sets

When developing a ML algorithm, it is important to split the data appropriately. Datasets are often split into training, validation, and testing sets. The training set is what the model learns from and what it develops its patterns from. The validation and testing sets are both used to evaluate the model's performance. The validation set is a subset of the training set, and it is used to estimate the generalization error while training the model for the purpose of optimizing hyperparameters [13, p. 119]. Hyperparameters are parameters of the model that cannot be estimated and must be set by the user. Hyperparameter optimization is further discussed in Section 2.2.8. The testing set is used to evaluate the performance of the model after the training is done. Each of the datasets are independent from each other to reduce bias.

When training on the training set, the goal of the model is to reduce the prediction error of the training set called the training error. A main theory of machine learning is that the training and testing sets are drawn from the same probability distribution. There exist fixed weights, w, which result in a minimization of the training error. Since the testing and training sets come from the same distribution, w would also result in a minimization of the testing error [13, pp. 107–109]. In an idyllic scenario, a machine learning model samples from the training set, learns the weights that minimize the training set error, and uses those weights when sampling from the testing set to make accurate predictions. If there are variances in the type of data from the training to the testing set, which is common in real-world applications, the model may not perform as expected and the testing error would be high.

An important part of the machine learning process is making sure to split the datasets appropriately. Often, data is split at random from a large, collected pool. However, this can lead to datasets that don't represent the real-life problem that the model is trying to solve. Without creating the datasets intentionally, the data collected may be different than the data used in the real-life application [17]. Depending on the objective of the model, the method of creating the train and test split will vary.

2.2.3 Single-layer and Multilayer Perceptron

The basis for deep learning is the feed-forward neural network, which refers to the structure of the network. In a feed-forward network, information only flows in the forward direction and no feedback is provided [13, p. 164]. At a minimum, a neural network consists of an input layer and an output node, which is called a single-layer perceptron (Figure 2-3).

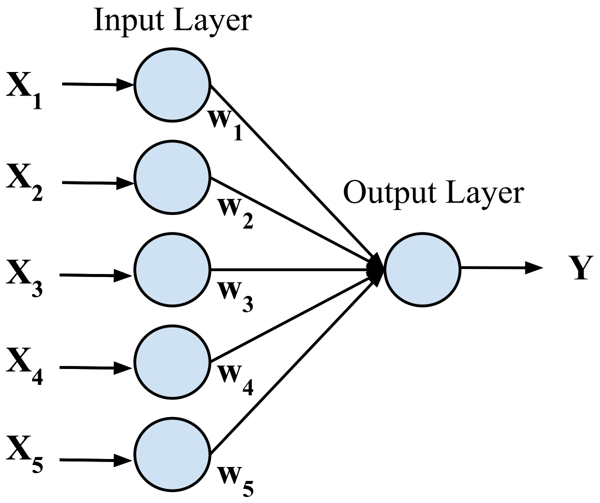


Figure 2-3: Single-layer perceptron

The input vector of form $X = [x_1, \cdots, x_d]$ transmits d features with weights $W = [w_1, \cdots, w_d]$ to the output node. At the output node, the weights from all the features are aggregated and an activation function is applied to predict the output Y (1) [16, pp. 5–6]. An activation function is the chosen function to model the behavior of the network depending on the desired output. For a simple single-layer perceptron, that function is often the sign function$\#(sign\, function)(2)$ which maps the aggregated features to either +1 or -1.

$$\hat{y} = \phi\left(\sum_{j=1}^{d} w_j x_j + w_{j0}\right) \tag{1}$$

$$\phi(v) = \begin{cases} -1 & for\, v < 0 \\ 1 & for\, v > 0 \end{cases} \qquad (sign\, function)(2)$$

In the function, w_{j0} is an invariant bias that the input data may have. This model can be used for binary classification as the outputs are predicted as belonging to one of two classes.

There are many activation functions that are used in machine learning to model the output of a layer. In addition to the sign function, three of the most common ones are the identity, sigmoid, and tanh functions. The identity function is a linear activation function that is used on output nodes to map the result to a real target value. The sigmoid function is a logistic function that outputs values between 0 and 1, making it useful for probabilistic outputs. The tanh function is like the sigmoid function, except the values are outputted between -1 and +1. Depending on the use of the desired output, either the tanh or sigmoid function may be preferred [16, pp. 12–13].

$$\phi(v) = v \qquad (identity\, function)(3)$$

$$\phi(v) = \frac{1}{1 + e^{-v}} \qquad (sigmoid\, function)(4)$$

$$\phi(v) = \frac{e^{2v} - 1}{e^{2v} + 1} \qquad (tanh\, function)(5)$$

As mentioned, the sigmoid function is useful for determining probabilistic outputs. It is mostly used for binary classification problems where given an input, it would return the probability of it belonging to one of the two classes. However, when the problem is multiclass and there are multiple output nodes, a variation of the sigmoid function is used. This is called the softmax function. For each of the i inputs, a probability is calculated for it belonging to each of the k classes. The class with the highest probability is then assigned to the output [18, p. 410].

$$\phi(\bar{v})_i = \frac{e^{v_i}}{\sum_{j=1}^{k} e^{v_j}} \qquad (SoftMax)(6)$$

While the sigmoid and tanh functions are the traditional functions used for nonlinear outputs, additional piecewise functions have been developed. An exceedingly popular one is the Rectified Linear Unite (ReLU) function.

$$\phi(v) = \begin{cases} 0 & for\ v\ < \ 0 \\ v & otherwise \end{cases} \qquad (ReLU)(7)$$

One of the reasons for its popularity is for how it makes training multilayer neural networks easier and can create more powerful nonlinear models [16, p. 13].

A multilayer neural network (Figure 2-4), otherwise known as a multilayer perceptron (MLP), is when additional computational layers are used in the network between the input and output. The computations performed in the intermediate layers are not visible from the outside, which is why these layers are referred to as hidden layers. For each hidden layer, every node from the previous layer is connected to each node in the current layer. The connections between each layer are therefore matrices of weights calculated by the activation function chosen for that layer. MLPs are more computationally powerful compared to the perceptron, since the computations performed in each hidden layer are continuous sigmoidal nonlinearities [15, p. 229]. This allows for more complex algorithms and for more intricate applications. A common notation for MLPs and other networks is to specify the number of nodes in each layer in brackets beside the network type. For example, MLP(16, 32, 64) would refer to an MLP of 3 layers consisting of 16, 32, and 64 nodes respectively.

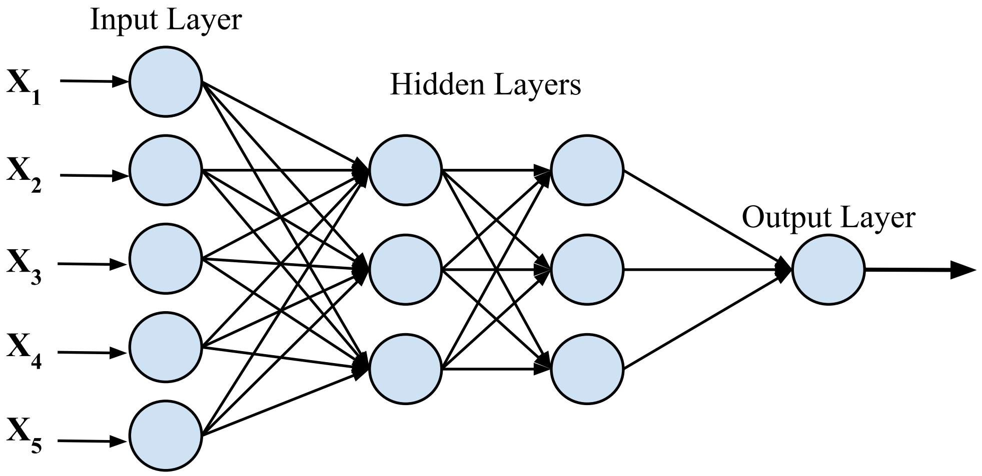


Figure 2-4: Multilayer perceptron

2.2.4 Convolutional Neural Networks

A convolutional neural network (CNN) is a popular type of neural network that is often used for object detection and image classification. One of the reasons for CNNs popularity is that invariance properties can be built into the network structure [15, p. 268]. CNNs operate on structured datapoints with feature channels. In traditional 2D images, the pixels are ordered in a 2×2 array, with the RGB values for each pixel becoming feature channels giving the 3D input array a depth of three. CNNs work by identifying patterns in the data at a local and global level that assign the subject of the data to a particular class. They first start at a local level, identifying low-level features that make up distinguishing relationships between neighbouring datapoints. These local features are then combined to form global features to gather an understanding of how all the local features interact with each other [18, p. 412]. Because of this, variances in data input such as translation and rotation of objects can be dealt with since the identifying features from object to object will still be the same. CNNs do this by using convolution layers to identify patterns and subsampling layers to down sample and aggregate them.

2.2.4.1 Convolution Layers

Convolution layers are where convolution operations are performed to extract features from the data. A convolution is an operation that involves multiplying matrix elements and summating them [18, p. 412]. The specific convolution operation can be defined on a network-to-network basis. Each layer has many convolution filters with different parameter values that apply this operation to the input data of the layer. The number of distinct filters determines the depth of the next layer.

$$Original\ Data\ = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$Convolution\ Filter\ = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

Each filter is applied recurrently to a submatrix of the input data, until the entire input data is sampled. This detects the same pattern across the input [15, p. 268].

$$Covolved\ Output = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \end{bmatrix}$$

By choosing different values for the filter, different patterns and their locations are extracted from the input. The output of the convolution layer is then fed into a subsampling layer.

2.2.4.2 Subsampling Layers

Subsampling layers, otherwise known as pooling layers, combine the features across the data extracted by each convolutional filter used in the previous layer. The most common type of pooling layer is called max-pooling. Like convolutional operators, pooling operators are applied to one small submatrix of data at a time. The max-pooling operator finds and returns the largest value of all the datapoints in the submatrix. Doing this reduces the dimensionality of the data and also introduces invariance [18, p. 415]. The largest feature value will be transferred to the next layer, no matter where within the submatrix it occurs. Other types of pooling such as average-pooling are sometimes used, but max-pooling remains the most popular [16, p. 327].

2.2.4.3 Data Augmentation

When training a neural network for classification or object detection, it is a good idea to augment the data to create new training examples. Providing more varied training examples can help avoid overfitting, which is discussed more in Section 2.2.5. By taking existing training data, modifying it slightly, and then using it to additionally train the network the model can get better at recognizing different versions of the object [16, p. 337]. With 2D images, some common data augments include flipping the image, translating, and rotating the object, and changing the colour intensities of the image. For 3D point clouds, some common methods of augmentation include rotating the object around the up-axis and jittering the points with Gaussian noise [6].

2.2.5 Loss Functions and Regularization

As discussed in Section 2.2.2, a machine learning model samples from the training set, learns the weights that minimize the training set error, and uses those weights when sampling from the testing set to make accurate predictions. Therefore, the goal of the model is to both minimize the training error and minimize the gap between the training and testing errors [13, p. 109]. To do this, the model's error needs to be known. The error of how well a model's predictions match the ground-truth data is calculated using a loss function [18, p. 29]. Loss functions differ from model to model depending on the nature of the machine learning problem. For regression models, the most common function is the mean squared error (MSE) algorithm. In this equation $\hat{f}(x_i)$ is the prediction for the ith observation and *n* is the total number of observations.

$$Mean\ Squared\ Error = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(x_i)\right)^2 \tag{8}$$

For a classification deep learning neural network with categorical targets, the function to calculate loss is taken as the negative multinomial log-likelihood, otherwise known as cross-entropy loss (9) [18, p. 410].

$$cross - entropy\ loss = -\sum_{i=1}^{n}\sum_{k=0}^{K} y_{ik} log[f_k(x_i)] \tag{9}$$

Cross-entropy loss is the sum of the loss calculated at each ith instance, where $f_k(x_i)$ is the probability of the class k as calculated using the SoftMax function (6).

When assessing the accuracy of the model, if the gap between the training and testing errors is too large the model may be overfitting to the training data. The model may be able to predict the classes on the training data perfectly, but if it is overfitted it will have low performance on the test data [16, p. 25]. Overfitting can occur when models are complex, and the number of weight parameters is greater than the number of training data instances. This leads to a potentially infinite number of solutions, such that the function will result in zero training error but when presented with new data there will be large testing errors. A model needs to be complex enough to capture the relationship of the training data, but not so overly complex that it cannot generalize on test data. One of the ways to address overfitting is to augment the existing training data to create new data, as discussed in the above Section. Another way to address overfitting is to introduce regularization into the loss function. Regularization introduces a penalty term to the loss function which prevents the parameters of the loss function from growing exceedingly large [15, p. 10]. The forms of regularization common with cross-entropy loss include ridge regression and dropout regularization.

Ridge regression, otherwise known as L2 regularization or Tikhonov regularization, adds a penalty to the loss function that is defined by the sum of the squares of the function weights [16, p. 182].

$$Ridge\ Regression = Loss + \lambda \sum_{i=0}^{d} w_i^2 \tag{10}$$

The regularization parameter λ determines how aggressive the penalty is and *d* is the total number of parameters in the function. By testing different λ terms, the optimal penalty term for the function can be found. The tuning of λ is performed on the validation set along with other hyperparameters, and the method of which can be found in Section 2.2.8.

Dropout regularization is when nodes are ‘dropped’ from a layer of a designed model to prevent nodes from becoming too specific [18, p. 438]. To preserve the model architecture the nodes are not actually dropped, instead their activation functions are set to zero to prevent their influence on the model. The nodes are chosen at random for each training datapoint so that the same nodes are not dropped constantly. The number of dropped nodes in a run is a specified fraction θ of the total number of nodes. The weights of the remaining nodes increase by $1/(1-\theta)$, the effect of which acts as regularization within the model.

2.2.6 Optimization Functions

Having chosen a loss function to calculate the model’s error, the next step is to choose the method of finding the parameters that minimize the loss. Assuming regularization has occurred, these would be the optimal parameters. The method of doing this is the optimization function, which is often a gradient-descent method. By taking the gradient of the loss function, the parameters of the model can be updated to move towards a global minimum [16, p. 134]. The function works iteratively until the function no longer decreases, implying convergence.

A popular optimization function is that of stochastic gradient descent (SGD). The parameters of the function are updated after each run, where w is the vector of parameters, τ is the iteration number, and ∇L is the gradient of the loss function [15, p. 144].

$$w^{(\tau+1)} \leftarrow w^{\tau} - \epsilon \nabla L \qquad (SGD)(11)$$

Here, ϵ is the learning rate of the gradient descent. The learning rate determines the size of the step that the gradient descent takes while updating the next iterations parameters. It is another hyperparameter which can be optimized. If a learning rate is too low at the beginning of the optimization, the steps taken will be small and the algorithm will take a very long time to come to convergence. If the learning rate is large the optimal solution may be approximated early, but it may never be reached as the algorithm oscillates around it [16, p. 135]. The solution to this is to have the learning rate start large and then decay over time. The two most popular decay functions are exponential and inverse decay, where k controls the rate of decay.

$$\epsilon_\tau = \epsilon_0 exp(-k \cdot \tau) \qquad (Exponential\ Decay)(12)$$

$$\epsilon_\tau = \frac{\epsilon_0}{1 + k \cdot \tau} \qquad (Inverse\ Decay)(13)$$

Even with a decaying learning rate, it can be desirable to minimize the oscillation of the gradient descent steps. This is done by adding the momentum parameter α and the variable v which represents velocity. The velocity is calculated from the average of the decaying gradient [13, p. 293].

$$v \leftarrow \alpha v - \epsilon \nabla L \qquad (14)$$

$$w^{(\tau+1)} \leftarrow w^{\tau} + v \qquad (15)$$

The hyperparameter α determines the rate of decay of the previous gradients. The equation remembers the previous direction the gradient descent was going and uses this to cut out unnecessary movements. This allows for an optimization that generally knows the direction of the optimum, making the gradient descent function perform better over flat sections and local minima [16, p. 136]. The affect momentum has on optimization is a smoothing of the zig-zag oscillations, allowing for the optimum to be reached quicker.

An example of another optimization function that uses momentum is the adaptive moment (Adam) algorithm. Adam exponentially smooths the first-order gradient as a variant of the above momentum method to include the momentum in the parameter update [16, pp. 140–141]. Adam also corrects the bias of the estimates of the first- and second-order moments [13, p. 305]. Given a decay parameter $\alpha \in (0,1)$, A_i is the exponentially averaged value of the ith parameter w_i.

$$A_i \leftarrow \alpha A_i + (1 - \alpha)(\nabla L)^2 \ \forall i \qquad (16)$$

The exponentially smoothed value is then calculated using the decay parameter α_f:

$$F_i \leftarrow \alpha_f F_i + (1 - \alpha_f)\nabla L \ \ \forall i \tag{17}$$

This then makes the updating of the parameters look like the following function:

$$w^{(\tau+1)} \leftarrow w^{\tau} - \frac{\epsilon_\tau}{\sqrt{A_i}} F_i \ \ \forall i \tag{18}$$

$$\epsilon_\tau = \epsilon \left(\frac{\sqrt{1 - \alpha_\tau}}{1 - \alpha_f^{\tau}} \right) \tag{19}$$

ϵ_τ now makes the learning rate a bias correction factor to counteract the bias introduced in the initialization of the algorithm.

2.2.7 Anomaly Detection

Anomaly detection, otherwise known as outlier detection, is a category of machine learning where the goal is to identify datapoints that deviate from the expected behavior of the data. This can be applied to anything from credit card fraud detection to tumor identification in MRI imaging [19, pp. 2–3]. If the defined pattern deviates from the norm it is considered ‘of interest’ and becomes desirable for analysis. There are many methods for approaching the task of anomaly detection and the most suitable one varies depending on the task at hand and the available data.

There are several categories of anomaly detection problem types, and one that is relevant to this thesis is Industrial Damage Detection. This refers to the physical damage that occurs to industrial units, either mechanical components or physical structures. The data is collected by sensors and analyzed for anomaly detection to allow for the repair and replacement of the physical components before issues arise [19, pp. 17–18]. Two of the common methods for detecting anomalies for this type of damage detection include Parametric Statistical Modeling and using NN autoencoders.

Parametric Statistical Modeling involves using statistical techniques to analyze data that follows a normal distribution, assumed to be Gaussian. With a Gaussian distribution of the data, the mean, μ, and standard deviation, σ, for the dataset can be calculated. In the simplest of calculations, the anomaly score for a datapoint is said to be the distance of the datapoint from the estimated mean. The threshold for determining how far away from the mean is considered anomalous can be set using the standard deviation. Datapoints with anomaly scores outside of $\mu \pm x\sigma$ will be considered anomalous where x is a chosen multiplier between 1 to 3. A value of 3 would result in a range consisting of 99.7% of the dataset [19, p. 34].

A variation of the anomaly score is the Z-Score, which takes the anomaly scores from above and normalizes them by taking the absolute values and dividing by the standard deviation [20, p. 1014].

$$Z - Score = \frac{|x - \bar{x}|}{s} \tag{20}$$

The value x is a sample point, $\bar{x}$ is the mean across all samples, and s is the standard deviation. The Z-Score provides the number of standard deviations from the mean for all datapoints, which can once again be compared to a set threshold to determine if the data is anomalous.

A different methodology for structural damage anomaly detection is to use NN autoencoders. Autoencoders take an approach that utilizes unsupervised deep learning architecture to perform anomaly detection. The goal is to have a model that learns the low-level features of the data using MLPs for the purpose of reconstructing the data as accurately as possible. By learning the features which make up the data, the data is encoded. The architecture of an autoencoder has a constricted middle, such that the interior layers have fewer units than the input and output layers (Figure 2-5). This makes the model learn a reduced representation of the data which prevents a direct reconstruction [16, p. 71]. Normal data will be reconstructed well by the model while anomalous data will have larger reconstruction errors since the features will not be as present [21].

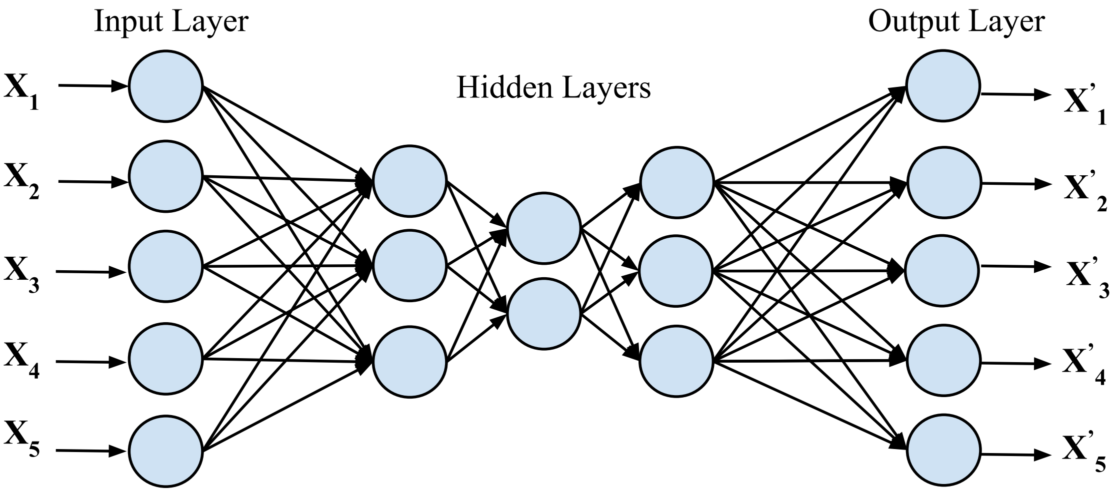


Figure 2-5: Autoencoder neural network

The loss function to get the error is taken by calculating the distance between the points in the original and reconstructed point clouds. In this context, distance is a non-negative function which measures the dissimilarity between the two point clouds [22]. By looking at the differences between the input and output data, the accuracy of the reconstruction can be evaluated. A common metric used for calculating the differences between point clouds is the Chamfer Distance (21).

When using a loss function for comparing point clouds, certain conditions must be met. The loss function must be differentiable across all point locations, efficient to compute, and robust against point outliers [23]. The Chamfer Distance calculation meets these requirements while also not requiring point pairings from each point cloud to be explicitly defined. For each point in point cloud A, the CD algorithm [22] finds the nearest neighbor in point cloud B. The point-level pair-wise distances for all points are averaged together. This is done symmetrically for points from A → B and B → A to get the average distance between points.

$$d_{CD}(S_1,S_2) = \frac{1}{|S_1|}\sum_{x\in S_1} \min_{y\in S_2} \|x - y\|_2 + \frac{1}{|S_2|}\sum_{y\in S_2} \min_{x\in S_1} \|y - x\|_2 \tag{21}$$

2.2.8 Hyperparameter Optimization

As mentioned in previous sections, when building a model there are some parameters that must be set by the user. These parameters are called hyperparameters since they cannot be estimated from the data[13, p. 119]. The majority of the model's parameters will be optimized using the loss and optimization functions during training, but hyperparameter tuning must be implemented separately.

The number of hyperparameters that can be turned for each model depends on the type of model that is being used. Deep learning models have many hyperparameters which can be optimized. For the model itself, the number of hidden layers, the number of neurons on each layer, and the activation function for each layer are all hyperparameters. If the model architecture doesn't require a specific value for each of those hyperparameters then they can all be tuned. For model compilation, hyperparameters include the optimization function, the learning rate and momentum of the optimization function, and the method of regularization. For model fitting, some hyperparameters include batch size and the number of epochs that the model is trained for.

To tune these parameters, the validation set discussed in Section 2.2.2 is used. This data set is a subset of the training data that the model did not see during training. The generalization error is calculated at the end of each run using the validation set, and this is used to check the performance of the model with various hyperparameters [13, p. 119]. The hyperparameters can then be updated according to the hyperparameter optimization method (HOM) to improve performance. This means that given a hyperparameter configuration space $\Lambda = \Lambda_1 \times \Lambda_2 . \times . . \Lambda_2$ with vectors of hyperparameters $\lambda \in \Lambda$, the formula for optimizing hyperparameters is [24, p. 5],

$$\lambda^* = argmin_{\lambda\in\Lambda}\mathbb{E}_{(D_{train},D_{valid})\sim\mathcal{D}}V(\mathcal{L},\mathcal{A}_\lambda,D_{train},D_{valid}) \tag{22}$$

where $V(\mathcal{L}, \mathcal{A}_\lambda, D_{train}, D_{valid})$ is the loss of the model for the applied hyperparameters λ on the algorithm $\mathcal{A}$. Additionally, the maximum may be searched for instead of the minimum.

The most basic HOM is the grid search. The grid search takes the cross product of the sets of values for each hyperparameter and looks for the combination that minimizes the error [24, p. 7]. While this method may be effective for simple models with few hyperparameter options, large configuration spaces are impractical with this method due to how the number of computations grows with the number of hyperparameters. Instead of grid search, the parameters can be chosen using the random search method. Marginal distributions are chosen for each hyperparameter instead of finite values and the values are chosen at random while the search is being conducted. This method parallelizes better than grid search and was found to be more efficient than grid search at determining the optimum parameters [13, p. 429]. However, there are still guided search options that may perform better than random search.

One of the popular guided methods for HPO is Bayesian optimization (BO). Bayesian optimization consists of an acquisition function to pick the hyperparameters for evaluation and a probabilistic surrogate model based on a prior distribution. In pseudo code the BO algorithm is as follows [25]:

```
for $n$ = 1,2,…., do
    select new $x_{n+1}$ by optimizing acquisition function $\alpha$
    $x_{n+1} = \arg\max_x \alpha(x; \mathcal{D}_n)$
    query objective function to obtain $y_{n+1}$
    augment data $\mathcal{D}_{n+1} = \mathcal{D}_n, (x_{n+1}, y_{n+1})$
    update statistical model
end for
```

This method uses the previous observations to improve the model using posterior updating. This allows for a more refined search procedure when compared to the random search since the algorithm is learning from previous configurations.

Another strategy is to use the Hyperband (HB) algorithm. This method is an evolution of the random search method. HB compares the performances of several random configurations of parameters and stops the worst performing ones using successive halving [24, p. 17]. It removes half of the configurations that performed the worst and re-evaluates the remaining ones until only one configuration remains. The HB method performs this in sections by dividing up the computational budget that has been allocated for HPO.

2.2.9 Performance Metrics

After the model has been developed the quality of its prediction needs to be assessed. This can be done beyond the error that is calculated using the loss function of the model. For classification problems, performance metrics are used to compare the model's predictions to the ground truth from the test dataset. The model's predictions are identified as either a True Positive (TP), False Positive (FP), True Negative (TN), or False Negative (FN) depending on how they compare to the actual label of the data (Figure 2-6). A common way to display this data is by using a confusion matrix. Based upon those categories, many performance metrics can be calculated [26]. Common ones for binary classification can be seen below in Table 2-1.

		Predicted Value	
		Negative	**Positive**
Actual Value	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

Figure 2-6:Binary performance matrix

Table 2-1: Binary performance metrics

Metric	Equation	Purpose
Accuracy	$\frac{tp + tn}{tp + tn + fp + fn}$	How well the classifier identifies true labels
Precision	$\frac{tp}{tp + fp}$	How well the classifier predicts the positive class
Recall	$\frac{tp}{tp + fn}$	The classifier's ability to properly identify positive labels
Specificity	$\frac{tn}{fp + tn}$	The classifier's ability to properly identify negative labels
Area Under Curve (AUC)	$\frac{1}{2}(Recall + Specificity)$	The classifier's ability to avoid false classification

2.3 Deep Learning on Point Clouds

2.3.1 Introduction to Deep Learning on Point Clouds

Applying deep learning techniques onto point clouds has been a focus in recent years as researchers have explored the potential uses of ML algorithms trained on point cloud data. Challenges arise from the fact that point clouds are unordered and have high dimensionality [7]. CNNs and other neural networks require a structured input to generate appropriate weights for each layer [27].

There are three main objectives when it comes to deep learning on 3D point clouds: point cloud segmentation, object detection and tracking, and shape classification. Point cloud segmentation involves separating the point cloud into various subsets based upon the semantics of each point [7]. Object detection and tracking involves identifying 3D objects within their scene and tracking them as the scene changes. The focus of this research is on shape classification, which involves identifying a 3D object and classifying it as belonging to a category. These objectives have all been achieved in recent years using 2D images, with applications including text processing and facial recognition [14]. Images consist of structured arrays of easily labelled and processed pixels, making them ideal data formats for deep learning applications. Successful 2D deep learning algorithms include RCNN [28] and YOLO [29]. To achieve the same success using point clouds, various methods have been proposed to address the challenges of the data format. These methods can be categorized into view-based methods, volumetric-based methods, and point-based methods.

2.3.2 View-Based Methods

View-based methods involve translating the 3D shape into multiple 2D images. It circumvents the challenges of the 3D data format by converting the data into a 2D format. A perspective camera is moved around the 3D shape and collects images from all different angles [30]. Features are then learned from the 2D images, using the successful image-based convolution methods mentioned above. The view-wise features that are extracted then need to be aggregated from all the images in order to get an accurate shape classification model [7].

Multi-view CNN (MVCNN) [31] is the pioneering work for view-based methods. The model pre-trains on large 2D datasets, such as ImageNet, and is then fine-tuned using the views of the 3D object. It uses element-wise maximum operating view-pooling layers to aggregate the results from each view that is passed through the first CNN. That means that only the maximum element from each view is included for the final CNN and soft max classification. This results in information loss [7] as other features are not included in the classification decision. Additionally, there are some challenges when it comes to determining the multiple views used in the approach. There is no method provided to decide the number of views of the object and where those views should be [32]. Too many views would be inefficient, but too few could result in self-occlusions. The optimal camera field of view and distance for each object must also be considered to capture shape data in the image.

2.3.3 Volumetric-Based Methods

Volumetric-based methods approach the point cloud problem by converting the point cloud to a 3D voxel representation and then use a CNN on the ordered grid to classify the shapes [7]. The voxelization process creates a volumetric occupancy grid to estimate free and occupied space based on the point cloud [33]. Each coordinate (x, y, z) is converted to a voxel (i, j, k) by a uniformly discretized mapping process. The resulting grid is also dependent on the chosen resolution. If the resolution is too large, aliasing can occur, and data can be lost. If the resolution is too small, shape information from the object may be lost [33]. Additionally, the computational and memory requirements for the occupancy gird scale cubically with the resolution [34]. This puts a limitation on most volumetric-based methods as it would not be practical to work with a high voxel grid resolution.

An approach to address this is to use an octree to reduce the computational memory requirements. An octree recursively subdivides a 3D space into octants, which means the space is split up into eight equal parts. The approach by Riegler et al. [34] applies subdivisions only to cells containing relevant information. That way, sparse areas will be allocated to large cells while dense areas will have voxels of a significant resolution to extract meaningful features. There are limitations to the efficiency of this method since the CNN is not trained on all the voxels of the object. Specifically, it was found that this method was less efficient than full-voxel solutions when the voxel resolution was lower than 64^3 [32].

2.3.4 Point-Based Methods

Point based methods were introduced in 2017 with PointNet [6], and have become popular because they work directly with the raw point cloud data. This prevents information loss that is inherent in the view-based and volumetric-based methods through data conversion [7]. The different point-based methods can be split into pointwise MLP networks, convolution-based networks, graph-based networks, and hierarchical data structure-based networks.

2.3.4.1 Pointwise MLP Networks and PointNet

Pointwise MLP networks involve using fully connected MLPs to extract local features from each point and aggregate them using max pooling layers[4]. PointNet (Figure 2-7) is the pioneering work for this type of method and will be the focus of the experiments in this research for reasons described below. The PointNet architecture works directly on the point cloud represented as a set of 3D points $\{P_i | i = 1, \cdots, n\}$, where each point is a vector containing at minimum Cartesian coordinate values [6]. As mentioned above, point clouds are unordered. To process a point cloud of N points, the network must be invariant to order and be able to work on any of the N! combinations of the point cloud. PointNet addresses this by using a symmetrical function on the input that will result in the same vector output regardless of the order of the input. A MLP network learns local features for each point, and then the symmetric max pooling function aggregates the results to extract all the global features [4]. PointNet can be represented by the following function f [27]:

$$f(P_1, P_2, \cdots, P_n) = \gamma\left(\underset{i=1,\cdots,n}{\mathrm{MAX}}\ h(P_i)\right) \tag{23}$$

where γ and h are features learned from MLPs and MAX is the symmetric function.

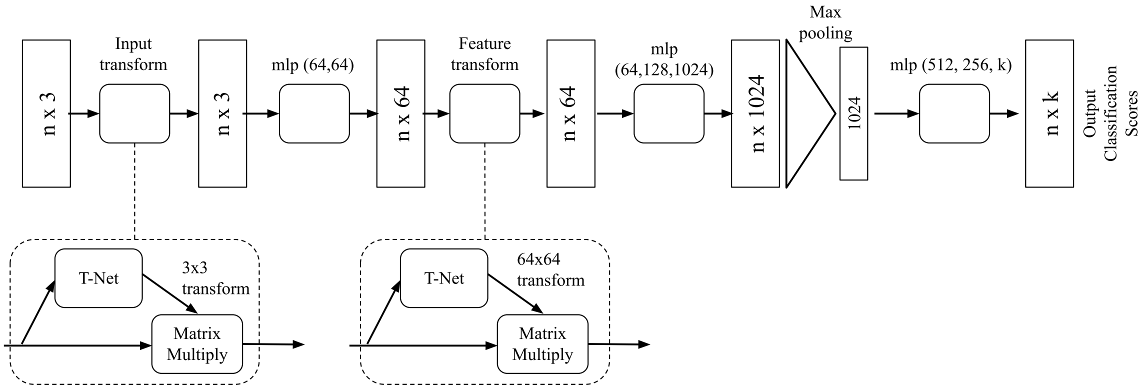


Figure 2-7: PointNet architecture[6]

The PointNet algorithm also is invariant to geometric transformations. If a point cloud of an object undergoes a rigid transformation of rotation or translation, the algorithm still classifies the point cloud under the same global category as before the transformation. To achieve this, PointNet implements a mini transformation network called T-net which develops an affine transformation matrix to align the input coordinates to a canonical space [35]. The T-net block is a regression network that predicts a 3×3 matrix to multiply with the $n \times 3$ input to achieve pose normalization. It consists of shared MLP(64, 128, 1024) layers, a max pooling layer, and fully connected Dense(512, 256) layers [6]. The weights and biases are then reshaped into a 3 × 3 matrix for the matrix multiplication with the input.

After alignment, the point cloud is then passed through an MLP(64, 64) network to extract features from each point [36]. The T-net block is applied with higher dimensionality a second time in the architecture, after the 64-dimensional feature extraction. This implementation predicts a matrix to align the extracted features of the point cloud, where the first one aligns the point cloud itself within the canonical space [4][6]. After the second T-net, a second MLP(64, 128, 1024) is applied for additional feature extraction on the aligned features. Next, the previously discussed max pooling layer is used as the symmetric function to extract the global features of the point cloud. With the global features known, a final MLP(513, 256, *k*) is used to generate classification scores for each category [36]. Here, *k* is the number of categories to choose from for classification. PointNet then uses a SoftMax cross-entropy function as the loss function in the activation layer to predict the classification probability for each category. The loss function is defined below [36]:

$$Loss = -\frac{1}{k+1}\sum_{\varsigma}\sum_{l=0}^{k}\left(indic_{\varsigma,l}\cdot\log\left(\hat{y}_l^j\right)\right) + weight_{regre}L_{reg} \tag{24}$$

$$indic_{\varsigma,l} = \left\{\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 \end{bmatrix}\begin{matrix} First\ category, \zeta = 0 \\ Second\ category, \zeta = 1 \\ \vdots \\ n^{th}\ category, \zeta = n-1 \end{matrix}\right. \tag{25}$$

$$\hat{y}_l^j = softmax\left(Z_l^j\right) = \frac{e^{Z_l^j}}{\sum_{l=0}^{n} e^{Z_l^j}} \tag{26}$$

$$Z_l^j = \omega * p^j \tag{27}$$

The $weight_{regre}$ value is chosen to be 0.001, while the indicator matrix holds the one hot encoded (OHE) values of the categorical label. This is what translates the qualitative categories into values that the architecture can work with. Z_l^j uses the weights (ω) of each layer to calculate the probability of each categorical output *l* for each point cloud p^j.

$softmax(Z_l^j)$ then takes those values and predicts what is the most likely category that the point cloud belongs to.

Optimization gets more difficult because of the high dimensionality, so a regularization term is applied to the SoftMax optimization to help with stabilization. PointNet uses an approximated orthogonal matrix to constrain the feature transformation matrix,

$$L_{reg} = \|I - AA^T\|_F^2 \tag{28}$$

where A is the predicted output matrix from the second implementation of the T-net block [6].

PointNet is also shown to be robust against various input corruptions. In testing accuracy, Qi et al. [6] found it took more than 70% of the points to be missing from the point cloud to see a significant drop in the metric. PointNet detects critical points for each shape and has upper bounds for point clouds that would result in the same classification. If a point cloud provides points between the critical and upper-bounds limits, it will result in the same classification as another point cloud with points within that range. This is a good basis for extending the research to classification accuracy while using point clouds that only show part of the object. Theoretically, if the critical points for each object are still identifiable, the object can be accurately classified. It is also robust against outliers, as tests showed that when 20% of the points in the cloud were outliers the accuracy of the model was still over 80% [6]. This robustness makes this method a viable choice when working with LiDAR data, as it can withstand the natural variance that may occur within LiDAR scans.

In terms of computational efficiency, PointNet is more efficient than methods previously discussed. When compared to MVCNN, it has a floating-point operations/sample (FLOPs/sample) rate that is $141x$ more effective [6]. Compared to the squared and cubic complexities found in other models, PointNet operates in a linear manner, making it more efficient to operate for real-time applications. PointNet has limitations when it comes to scene segmentation, since it does not consider the local features of neighbouring points [4]. It that scenario, other architectures may be more appropriate. Since it is the first architecture of its kind, PointNet is very well documented and has been a launching point for many other networks. Because of this and the abilities described above, PointNet was chosen as the primary architecture for this research.

2.3.4.2 Convolution-Based Networks

Convolution-based networks involve applying pointwise convolutional operations on each point of the point cloud either through continuous or discrete convolution. These operations extract the relationships between local features of neighbouring points. For continuous convolution, convolution is applied on a continuous space across all points surrounding a center point. The spatial distribution between the neighbouring points and the center point is then used to modify the developed weights from other learned features [7]. An example of this type of convolutional network is Relation-Shape CNN (RS-CNN) which implements the developed relation-shape convolution (RS-Conv) operator [37]. RS-Conv uses an MLP to apply convolution to a neighbourhood of points around the center point. It learns features by relation, such as the Euclidian distance and relative position for each point to the center point [7]. By convoluting in such a way, the local relationships between points are known, which translates information to the underlying global 3D shape made by the point cloud. This creates a model that is shape-aware [37].

Discrete convolution involves assigning a uniform grid over the neighbourhood points before convolution. Instead of extracting the spatial distribution surrounding the center point, the convolution extracts the offsets of the neighbouring points to the center one. This is what is used to modify the weights learned by the layers [7]. An example architecture is Pointwise CNN [38], which uses this principal in their pointwise convolution operators. There is a $3 \times 3 \times 3$ convolutional kernel centered at each point of the point cloud, and it assigns the same weights to all points within the kernel. Unlike in PointNet, no pooling layers are used. Instead, the mean features of neighbouring points around a center point are calculated using previous layers [7]. This also means that there is no up-sampling or down-sampling since all the convolutions are pointwise [4]. Another aspect of Pointwise CNN is that it requires points to be ordered for processing, which is an extra step that PointNet does not need.

2.3.4.3 Graph-based Networks

Another way to process point clouds is by using graph-based methods. Each point in the point cloud is used as a vertex of a graph, assigning direction between each of the points [7]. Complex geometric structures can be encoded within the graph, making this type of network desirable when dealing with intricate data [39]. There are two ways to perform feature learning on the graphs, either using a spectral or spatial filtering method. For spectral methods, the convolution is done in the spectral domain using eigen-vectors of the graph Laplacian matrices. The signal on the graph is multiplied with the eigen-vector matrix. An example of this type of network is the Regularized Graph Convolution Network (RGCNN) [40]. The geometry of the point cloud is taken as an input through the coordinates and normal of each point. The graph Laplacian matrix is updated after each layer to learn the dynamic features of the point cloud. Without any approximations, spectral methods have high computational complexity due to the eigen-decomposition of the graph Laplacian matrices [40].

When spatial filtering is used, the convolution operations are defined in the spatial domain where points are assigned to spatial neighbourhoods [7]. An MLP can be implemented over each neighbourhood with pooling used to aggregate features and down-sample the data into a coarser graph. Edge-Conditioned Convolution (ECC) [41] is the pioneering method for this method. The vertices of the graph are connected to nearby points with edges, and a convolution operation is performed where filter weights are conditioned on each edge label. In this way, the features of each neighbourhood are extracted.

2.3.4.4 Hierarchical Data Structure-based Networks

Hierarchical data structure-based networks use various structures to extract features hierarchically from the point cloud [7]. These structures, such as octrees described in Section 2.3.3, divide the 3D space into tiered groups. A primary example, Kd-Net [42] uses k-dimensional trees (kd-trees) to organize the point clouds. A kd-tree splits the data along a coordinate axis into two sections containing an equal number of data points. The division starts at the originating root node and then happens recursively for a set depth D, ending with the tree's leaf nodes. By identifying the direction and location of the splits within a kd-tree, the geometric information of the data can be encoded. The networks are then processed from the leaves to the roots to learn local and global features. For each level, an MLP is used to extract the features of a node. The features are then combined with all other features of the child nodes to create the representation of the parent node [7]. This continues until the network culminates in the root node, upon which classification scores are predicted.

2.4 Related Works

There are multiple related works to this research which focuses on anomaly detection using point cloud data from partial scans of fixed objects and Deep Learning. Griebel et al. [43] looked at using the PointNet algorithm for anomaly detection using Radar data. Their application was identifying anomalies for autonomous vehicles during adverse weather conditions. In contrast to this research, their anomalies are dynamic bodies such as pedestrians or bicyclists. By modifying the PointNet and other algorithms they were able to achieve an F1 score and inference time that showed the ability for this method to be used for anomaly detection.

In [44] the researchers Masuda et al. developed a variational autoencoder (VAE) to perform anomaly detection on 3D point clouds. They believe their paper to be the first to attempt anomaly detection on 3D point clouds of general objects. They looked at reconstruction errors between the original and reconstructed point clouds to determine if the point cloud contained anomalies. However, in contrast to this research, their anomaly detection was performed on whole objects from the ShapeNet database instead of partial scans. For the encoder they didn't define a specific method but used skip-connection and max-pooling layers to extract the features from the point clouds. For their decoder, they used an architecture called FoldingNet. For their reconstruction loss metric, they used the Chamfer Distance calculation (21) to obtain the distance between the reconstructed and original point clouds.

Another relevant paper is that by Wang et al. [45]. Their research is looked at using PointNet and other encoders to reconstruct and 'repair' point clouds that have occlusions.

The method for generating occlusions involves taking complete point clouds and viewing them from a 3D camera reference frame. They used the ModelNet40 dataset in the research, which contains complete point clouds of singular objects. The occluded point clouds were generated by projecting the view from a virtual pinhole camera and removing all the points that were occluded from that view. The encoder mapped the occluded point clouds, and the decoder reconstructed the point cloud and attempted to recreate it without the occlusions. The Chamfer Distance (21) was also used as a loss metric to calculate the distance between the predicted reconstruction and the ground truth point clouds.

3 Methodology

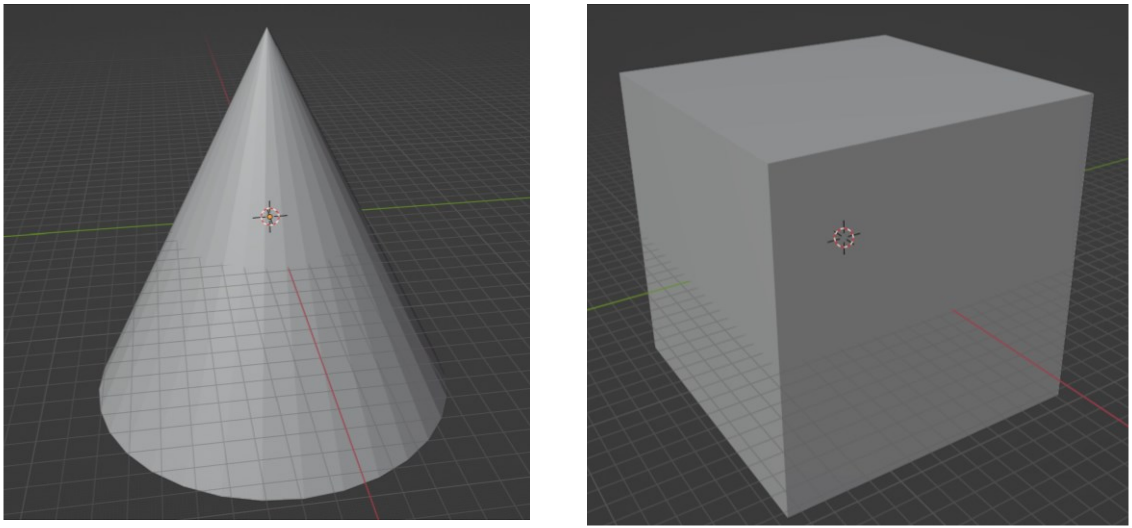
As discussed in the previous section, the PointNet ML architecture was chosen for processing the point clouds and the Parametric Statistical Modeling and Neural Networks autoencoder methods were chosen to conduct the research into using ML on 3D point clouds for damage detection.

The following approach was used:

1. Synthetic datasets comprised of several types of 3D scans were created and used for training, testing and validation. The types of 3D scans were categorized as full scans, targeted partial scans, and random partial scans. The random partial scans were used for both training research and the final testing data since they best represent the type of 3D point cloud data that would be collected on the space station.
2. A PointNet-based object classification model was built, and an Optimal Training Data experiment was conducted to determine which type of scans were most effective in classification given that the random partial scans would be used for the testing data.
3. Multiple models that perform both classification and anomaly detection were created and trained using the optimal training data type determined in the previous step.
4. The model that worked the best for anomaly detection was then chosen and hyperparameter tuning was done to optimize its performance.
5. The tuned model was then tested and validated, and its results analyzed.

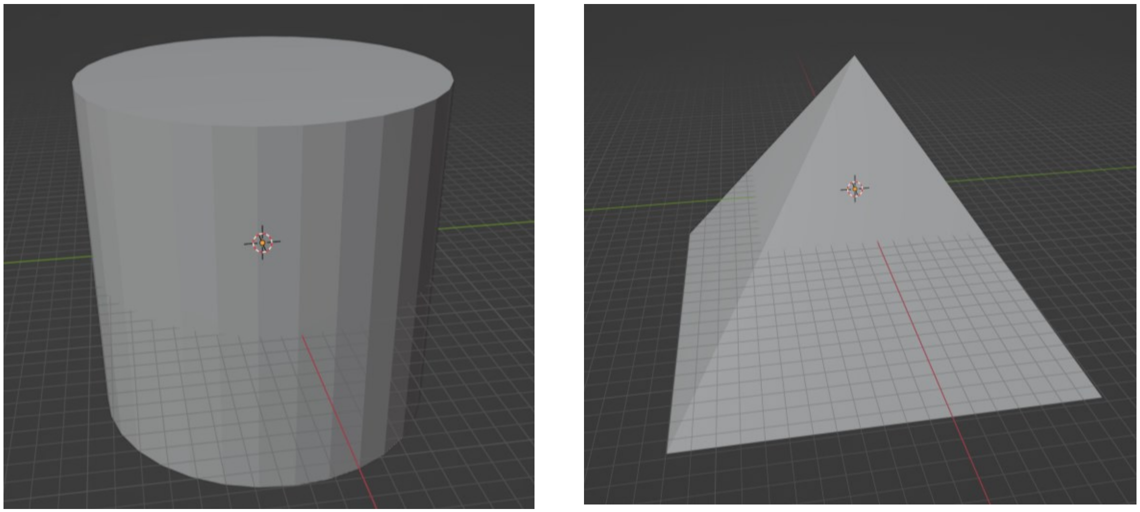
3.1 Data Generation and Preprocessing

The synthetic dataset used for this research is referred to as the Simple Objects dataset. It contains 3D object models of four basic geometric shapes: a cube, cylinder, cone, and pyramid (Figure 3-1), which were chosen as representatives of components that would make up a spacecraft. The 3D object models were used to generate full and partial 3D scans (point clouds) of each object, which were used then to test and create a baseline for the ML models being evaluated. The partial scans were created with the goal of representing the real-world scenario where occlusions are a common occurrence and, in many cases, only a part of an object will be captured in the field of view of a LiDAR scan.



a) Cone Object Model

b) Cube Object Model



c) Cylinder Object Model

d) Pyramid Object Model

Figure 3-1: Simple Object CAD Models

The Simple Objects data set was chosen to perform the primary research and create a baseline since: a) most complex objects can be made up from basic shapes, b) if the model could not be used to detect simulated damage on basic shapes, then it would most likely not succeed on more complex shapes. To frame this research, each simple object is representative of a component that would be found on a space station, as seen in Figure 3-2.

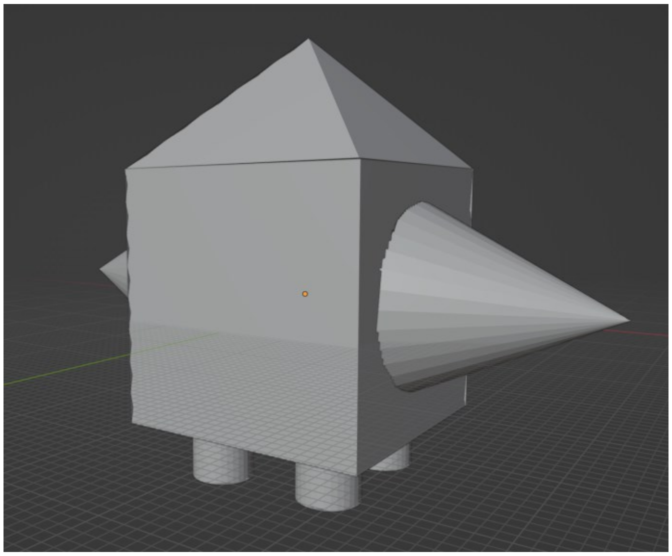


Figure 3-2: Theoretical spacecraft comprised of Simple Objects

Full and partial scans were first used to investigate which type of scan would be more effective in training the model using an Optimal Training Data experiment. Once this was determined, the less effective data styles were discarded and only the optimal training data style was used. The dataset was then used to research and evaluate an optimal anomaly detection method.

3.1.1 Simple Objects Full Scans

Full scans are defined as point clouds where an object is completely represented in the scan, in that there are no occlusions, and the point density is sufficient to fully render the object. Full scans of the four simple objects were generated for the purpose of testing the effectiveness of the scan type as training data. The scans were made by starting with the solid models of the objects in the dataset, representing them with triangular surface meshes (Figure 3-3), and then generating point clouds from the meshes. Triangular meshes describe the surface geometry of a 3D object using triangles that are connected by their common vertices and are a simple and effective way of digitizing an object.

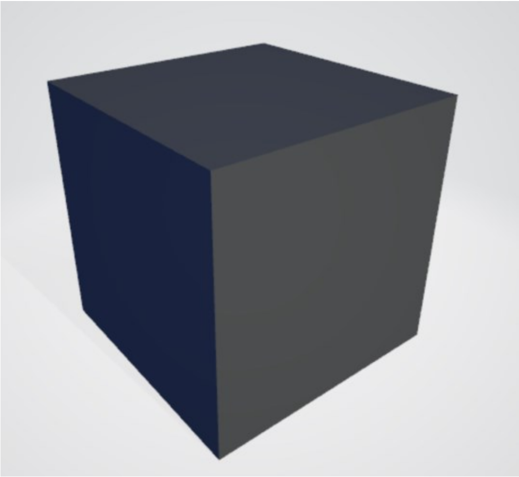


Figure 3-3: STL triangular mesh of the cube object model, used for sampling point clouds

The 3D models of the objects were made using Blender [46], a free CAD modeling software. The models for the cube, cone, and cylinder were made using existing Blender 'Primitives', which are premade meshes in simple 3D shapes. The pyramid model was constructed from a Primitive cone model by setting the number of vertices around the base of the cone to four and transforming it into a square-based pyramid. After this, the models were exported as triangular meshes to STL files, which is a common file format used for prototyping and 3D printing. The file type makes it very easy to transfer models between software.

The software used for the point cloud generation was CloudCompare (CC). CC is a free, open-source software that was developed for processing 3D point clouds and triangular meshes. One of CC's functions is to generate a point cloud by sampling points on a mesh. A benefit to using CC is that it has a Python API called CloudComPy that provides the ability to use some of its functions programmatically. This allowed for a Python script called FullScan.py (Figure 9-1: FullScan.py preambleFigure 9-1 and Figure 9-2) to be written that automated the sampling of points on the mesh so many point clouds could be generated.

For the automated point cloud generation, an input text file was made that specified the file location and label for each of the meshes. This allowed for the number of meshes to be variable so the code could be reused in the future. It was decided that 2,000 point clouds would be generated for each object to create an appropriately sized, balanced dataset that provided enough point clouds for training. With four different objects, this would lead to a training dataset of 8,000 point clouds.

Each point cloud was generated using the *samplePoints*() function from CloudComPy, where the sampling parameter for number of points was set to 50,000. This number of points was selected because it was found through trial and error to provide an appropriate point density that allowed all the features of each object to be visible. It also allowed for various levels of down sampling when using the clouds as training data without losing too much resolution. The *samplePoints*() function divides the sampling parameter by the number of triangles in the mesh and randomly samples that number of points in each triangle. The function sampled points on the entire mesh, creating a point cloud that represented the whole object. The point clouds (Figure 3-4) were saved to a folder, and a .csv file was made as a directory containing the location and object label for each of the point clouds.

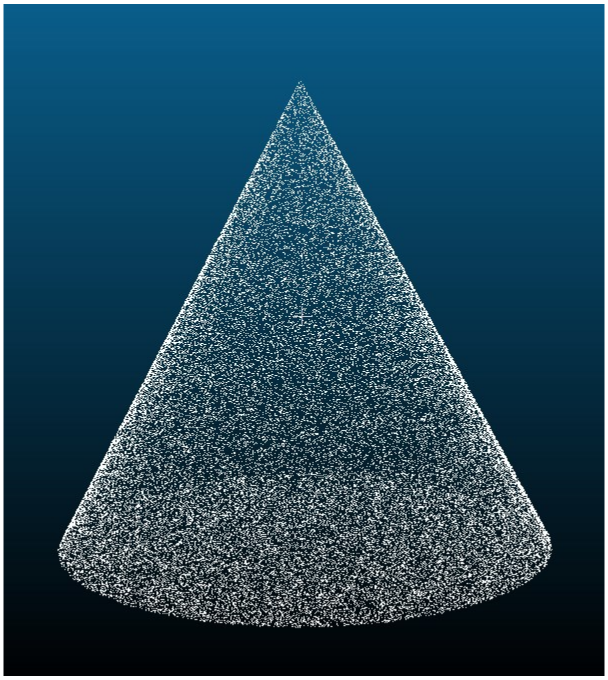


Figure 3-4: Example Full Scan PC

3.1.2 Simple Objects Partial Scans

Partial scans are defined to be point clouds where the full object is not represented in the scan. Three types of partial scans were generated, “targeted”, “random”, and “random damaged”. The random partial scans were generated with views taken at random to simulate real-world conditions. These scans were used in the training dataset and as the normal data in the test dataset. In contrast, the targeted partial scans were generated to ‘target’ the features that make up an object being classified, such as its edges and vertices, and were only used in the training dataset. Here the theory is that since PointNet works by identifying the critical features that make up the objects, it may be possible to improve the overall test results by pre-isolating these features in the training dataset when comparing the data style to random or full scans. The random damaged scans were random partial scans that included simulated damage to test the model in anomaly detection. These scans were not used for training and were used in the test dataset as anomalous data.

3.1.3 Simple Objects Targeted Partial Scans

To create the targeted partial scans, the STL files created in 3.1.1 for the Full Scans of each object were used. First the unique features of each object were identified, such as face top, edge, and vertex. A virtual camera was then simulated to view the point cloud of the object from different perspectives that isolated each feature. To obtain the different perspectives, the point clouds were rotated around each axis such that a targeted feature was within the camera’s field of view. Only the points within the field of view were then kept, all other points were discarded using a function that removes occluded points. This was repeated for each unique feature of the object. The unique features and the axis rotations to achieve them are summarized in Table 3-1 below. Here, ‘R’ stands for “Rotation around the axis”. The degrees were found through estimation and trial and error.

Table 3-1: Object model transformations for targeted scan generation

Label	*View*	*RX (°)*	*RY (°)*	*RZ (°)*
Cube	Face	0	0	0
	Edge	15	0	0
	Vertex	15	15	0
Cone	Top	0	0	0
	Side	30	0	0
Cylinder	Face	0	0	0
	Side	30	0	0
	Edge	0	15	0
Pyramid	Top	0	0	0
	Edge	15	90	0
	Face	300	0	0

To implement the method described above, the software Open3D [47] was used. Open3D is an open-source library for 3D data processing that has well-defined point cloud classes, and a Python API which made it desirable for use in this application. Similar to CloudCompare, Open3D has functions for sampling points off of a mesh. The sampling algorithms are either to uniformly sample points over the mesh in an assumed grid, or to sample them using a Poisson disk sampling algorithm. As explained in 2.1.2, the Poisson disk sampling algorithm uses a minimum distance parameter to guarantee an even distribution of randomly sampled points. This algorithm is closer to the algorithm used in CC, which is random sampling over a defined area, than the uniform grid sampling, so the *sample_points_poisson_disk*() function was chosen. The sampling parameter of 50,000 was kept consistent between generating methods to guarantee the same point density for all point clouds.

The Open3D function *hidden_point_removal*() is based off of Katz et al. [48] and takes in a defined camera location and camera view radius as parameters. It uses a very simplistic model of a pinhole camera, which projects a view onto the point cloud based upon the location and aperture radius of the virtual camera. After experimenting with the function, it was decided that for each targeted feature two different levels of 'zoom' should be simulated. One would be far away, capturing all points visible from that perspective. The other would be zoomed in to provide a view where the data isn't as clear and clean, in the hopes of creating a more robust training set less prone to overfitting. Given the eleven views in the table above and the two zoom levels, it was decided that each mesh should be sampled 1,000 times to have sufficient training data for each view. This would give a total of 22,000 point clouds for the targeted partial scans training dataset.

In a similar manner to the full scan generation, the mesh locations were read in from an input text file (Figure 9-4). The camera views were also read in from a text file. The location of the virtual camera was chosen by setting the X and Y coordinates to 0 and setting the Z coordinate to the normal of the difference between the maximum and minimum point cloud bounds. This would guarantee that for the far zoom level the entire feature would be visible.

For each mesh sample, the point cloud was rotated the required amount by converting the view parameters to a transformation matrix and applying it to the point cloud (**Error! Reference source not found.**). The level of zoom for the camera was then applied by adjusting the simulated cameras radius. For the far zoom level, the camera radius was set equal to the Z distance of the location of the camera. For the close zoom level, the radius was set to half that value. This narrowed the field of view as if the camera had zoomed in. After this the hidden point removal function was applied so that the points not within the camera's field of view were removed. Each point cloud (Figure 3-5) was saved to a folder and their object labels and file locations were exported to a directory.

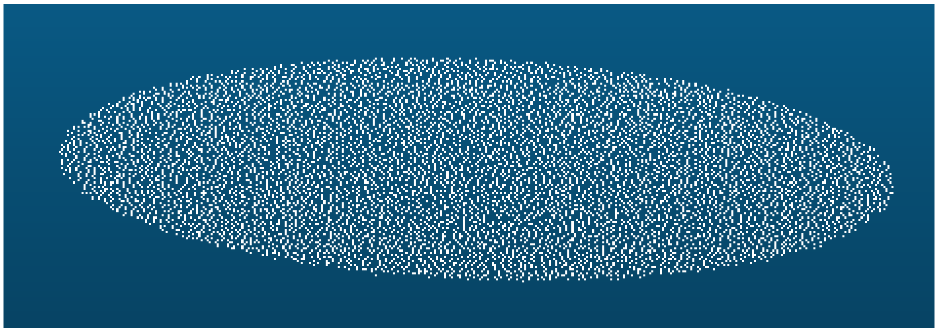


Figure 3-5:Targeted partial scan example of a cylinder 'face'

3.1.4 Simple Objects Random Partial Scans

The random partial scans were generated with CloudCompare by creating random slices of the Full Scan sampled point cloud. CC was used since it has easier to implement functions for creating random slices than Open3D. For each generated point cloud, the bounding box was retrieved. The bounding box is a property of the point cloud which contains the minimum and maximum point locations in each of the XYZ directions which "bound" the point cloud (Figure 3-6). A random transformation was then applied to the bounding box so that a new bounding box was generated that had different maximum and minimum point locations. This made the new bounding box contain only part of the original point cloud (Figure 3-7).

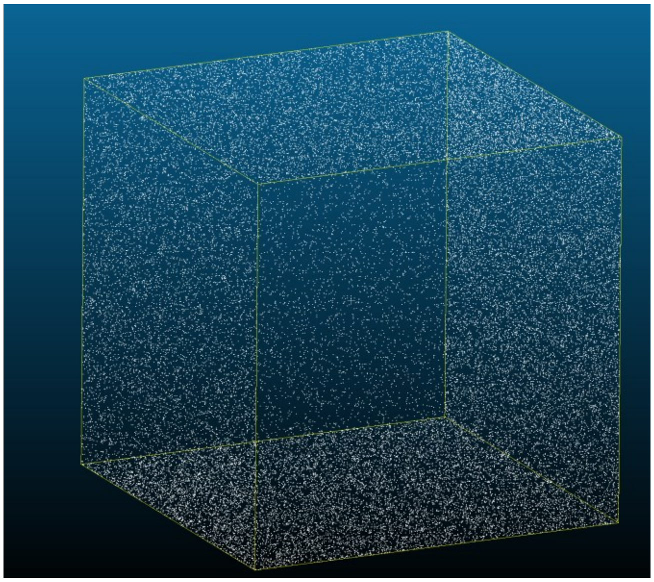


Figure 3-6: Complete point cloud with original bounding box in yellow

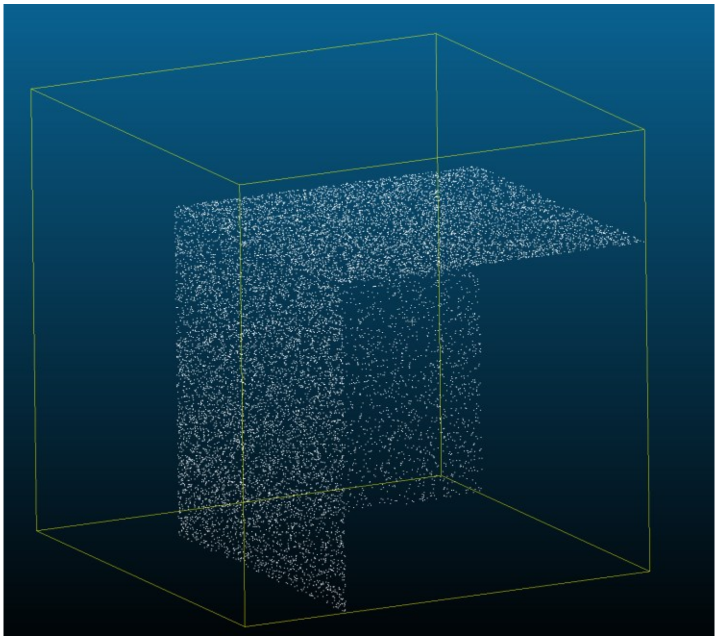


Figure 3-7: New partial point cloud from translated bounding box (yellow)

For each of the roll, pitch, and yaw in the random transformation matrix a number was randomly chosen between 0 and 180 to represent the degree of rotation. The length of the new bounding box in each direction was calculated by subtracting the location of the minimum corner from the maximum corner (Figure 9-5). A random translation was then chosen for each direction by multiplying the length by a random value between 0.1 and 0.9. This range guaranteed that there would be points in each point cloud. The direction of the translation was also randomized to avoid translating to the same corner of the point cloud each time. The CloudComPy class ccGLMatrix has a function *initFromParameters()* that can create a new bounding box based upon the calculated rotation and translation parameters.

The CloudComPy function *ExtractSlicesAndContours()* allowed for a 'slice' to be extracted from an existing point cloud by providing the existing bounding box and the transformed bounding box of the desired slice. This method was used to generate both testing and training datasets. They were generated at different times to keep the datasets separate. For the training data, 2000 samples for each mesh were taken, creating a total of 8000 point clouds. For the testing dataset, 1000 samples were taken for each mesh, creating a total of 4000 point clouds.

The meshes used were the same ones used for the other data generation. The location and label for each mesh was read in through an input file and the output directory DataFrame was pre-allocated for the number of point clouds being made. Each mesh was then sampled, and for each sample a random slice was generated using the method above (Figure 9-6**Error! Reference source not found.**). The new sliced point cloud was retrieved and saved to the output folder location. The location, label, and id of the point cloud were then saved to the directory. The id of each point cloud (Figure 3-8) was the label of the object combined with the creation number of the cloud.

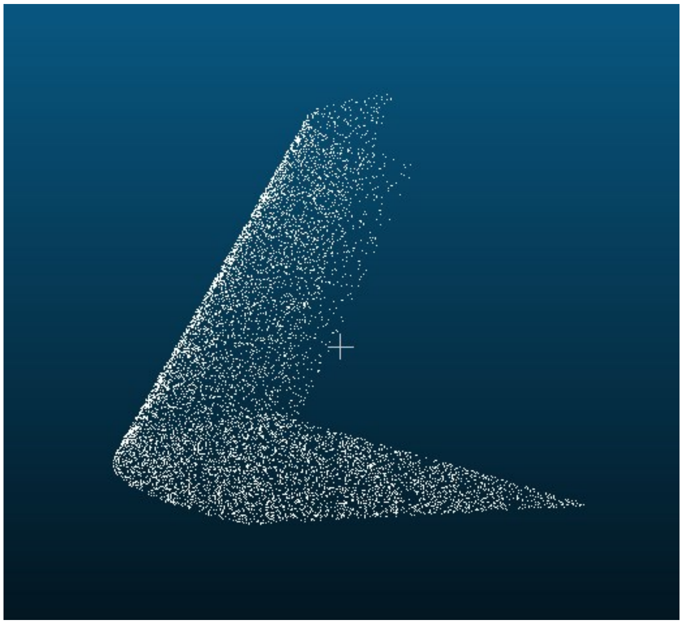


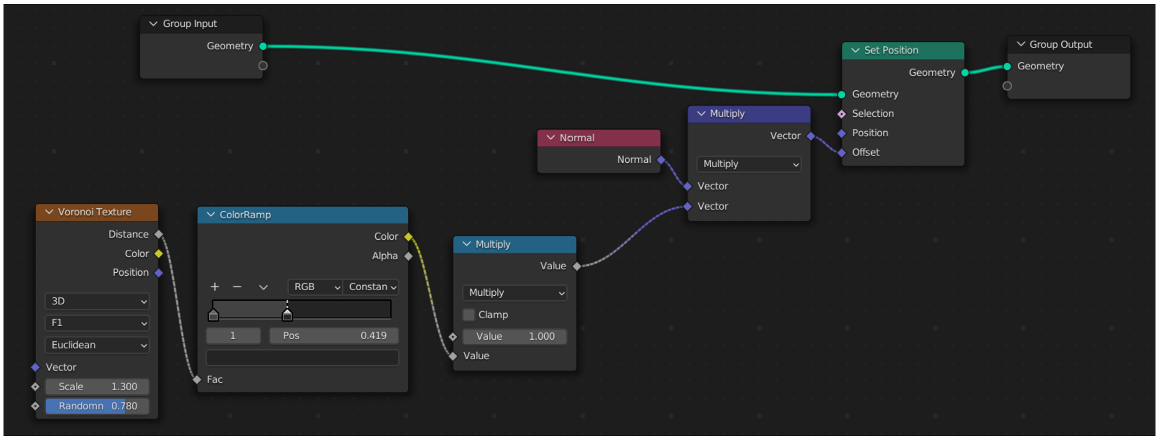
Figure 3-8: Example random partial point cloud of a cone

3.1.5 Simple Objects Damaged Scans

The damaged scans were additional random partial scans in the test dataset containing physical abnormalities that the model would try to identify as anomalous compared to the rest of the data. To make the damaged scans, each object model was randomly modified to include different types of damage. After the object models were modified with damage, the same procedure used to generate the normal random partial scans was followed.

The damage was generated and applied to each model using the Blender Python API and Blender nodes. Blender provides the functionality to create nodes, which are nodal blocks that allow for the modification of the active scene. There are many different types of nodes, and they can take the active model as the input, apply a specified modification, and return the output to the active scene. These modifications are precise, repeatable, and consistent. When nodes are combined into nodal groups, modifications can be performed on a large scale.

To develop a nodal group to randomly generate damage, the scale and type of the damage first needed to be defined. For this research the type of damage to the spacecraft to be identified is classified as "microscopic". This would include smaller scale damage such as craters, cracks, or divots. So-called "macroscopic" damage such as parts broken into pieces or large chunks being missing is out of scope and hence was not simulated. To simulate the microscopic damage, a YouTube tutorial [49] was followed that showed how to build a node group that would procedurally generate crater damage. This node group served as the basis for the damage that was generated on the Simple Objects (Figure 3-9).

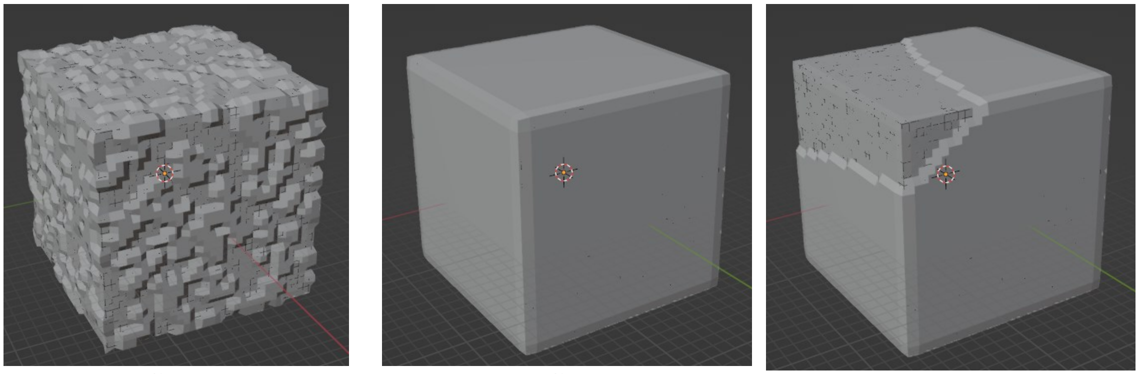


Scale 1.300
Randomn 0.780
1 Pos 0.419
Value 1.000

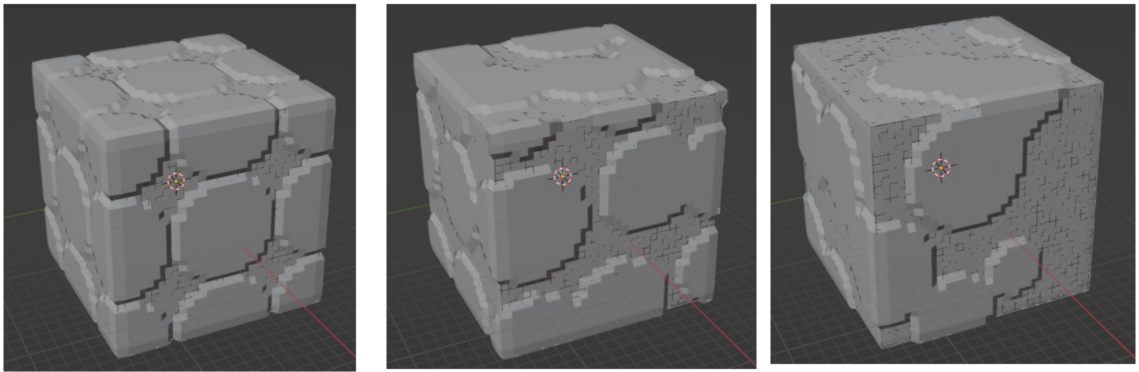
Figure 3-9: Blender damage node group

The geometry of the shape was first read in using the Group Input node. The damage was applied to the geometry as an offset using a Set Position node, was then transferred to the Group Output node. The offset which creates the damage is comprised of multiple node blocks which feed into each other. The starting point for the damage is the Voronoi

Texture block, which implements a Worley noise function. Worley noise, introduced by Steven Worley in 1996, is a texture function based off the distribution of irregularly scattered feature points [50]. This creates a pattern that mimics biological cell walls and can be used as a surface texture for generated objects such as stone pavements, reptilian skin, or hammered metal. For this implementation, the F1 feature was used which means to compute the distance to the nearest neighbour of each point. This was done using Euclidean distance as the metric calculated between points. The scale of the Voronoi texture node is a multiplier which determines the scale of the pattern, and the randomness determines if the pattern will appear in an orderly repeating fashion or not (Figure 3-10).



a) Scale = -5.0 **b) Scale = 0** **c) Scale = 0.35**



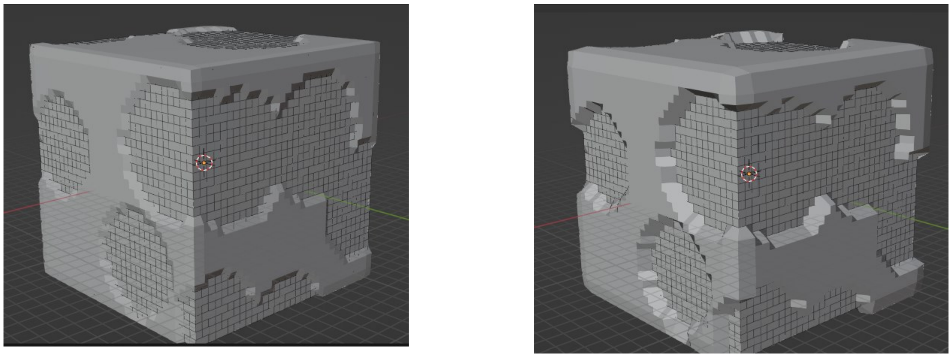
d) Randomness = 0 **e) Randomness = 0.35** **f) Randomness = 1**

Figure 3-10: Effect of varying scale and randomness values of Voronoi Texture block

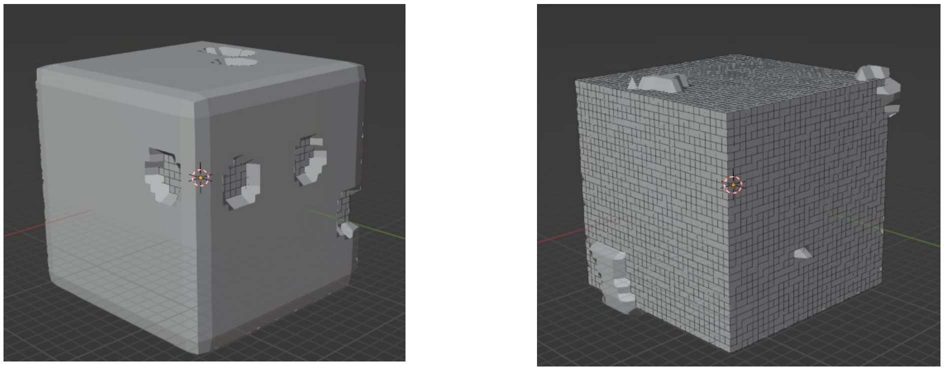
The Voronoi Texture node was then fed into a Color Ramp node. The traditional purpose of a Color Ramp node is to map values to colours using a gradient. In this method, it was used to set the intensity of the pattern generated by the Voronoi Texture block. The hue and saturation of the HSV colour wheel were both set permanently to 0 while the value slider was used to change the depth of the pattern (Figure 3-11). The position of the first colour block was set to 0, while the second one was used to add variability and determine the direction of the pattern (Figure 3-12). The output of the Colour Ramp was fed into a ‘Multiply’ block which converted the value into a vector. The vector was then multiplied with a ‘Normal’ node, which is the normal to the surface of the active object. This is what is applied to the objects surface as an offset using the Set Position node. When combined in this way, these nodes created an effect that looks like surface level damage that might occur on the exterior of a space craft.

H 0.000
S 0.000
V 0.220
A 1

Figure 3-11: HSV Colour wheel from Colour Ramp node



a) Value = 0.035 **b) Value = 0.1**



c) Position = 0.2 **d) Position = 0.8**

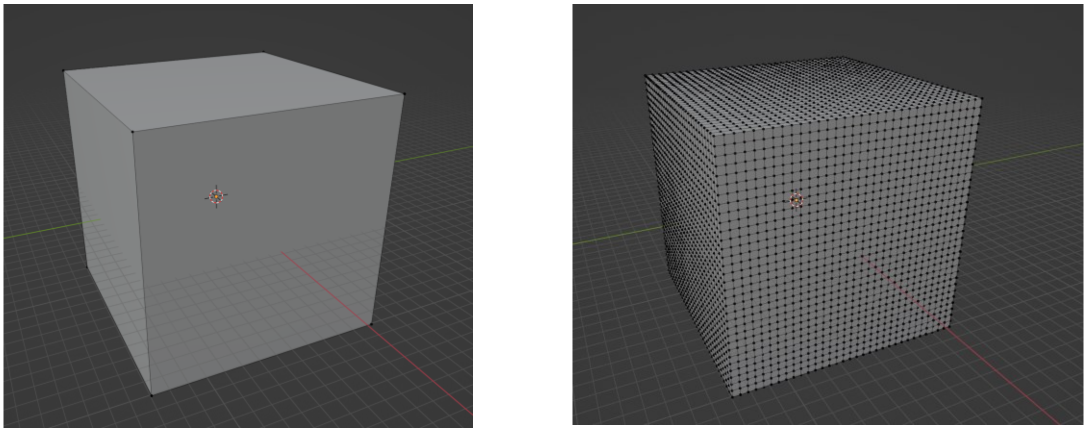
Figure 3-12: Effect of varying Value and Position numbers in Colour Ramp Node

To create random damage, ranges were determined for the scale, randomness, value, and position variables (Table 3-2). These ranges were found through experimentation and deciding what would be reasonable damage for the exterior of a spacecraft. The damage needs to be surface level damage that is distinct enough to be identifiable as anomalous when compared to the normal object. Variables that created patterns that were too orderly, or damage that was too deep or tall, were excluded from the acceptable range.

Table 3-2: Ranges for damage generation variables

Range	***Min value***	***Max value***
Scale	-5	2
Randomness	0.35	1
Value	0.1	0.95
Position	0.2	0.8

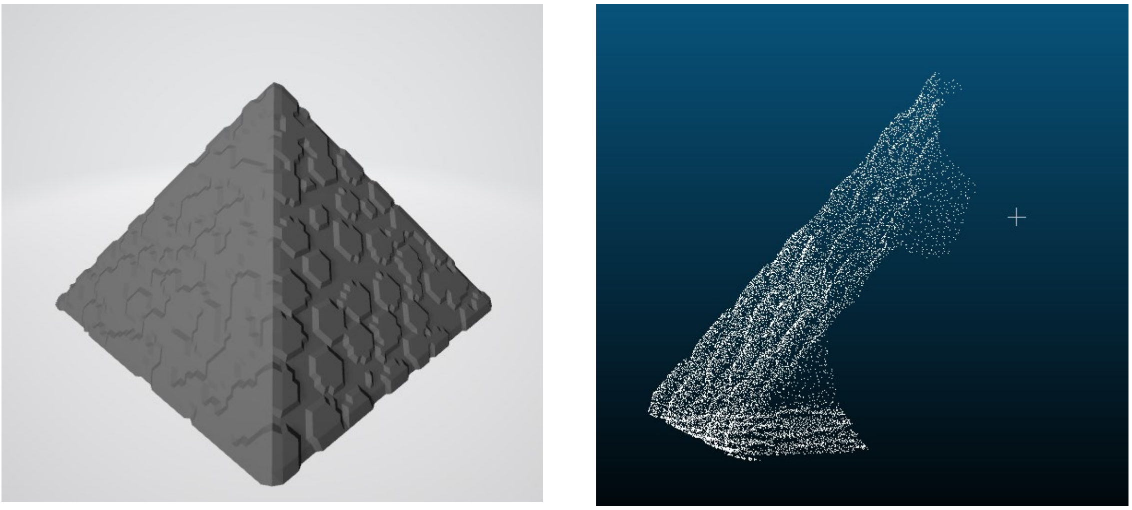
Random number generators were then used to select numbers in those ranges for each variable. The Python API allowed for the node group to be applied programmatically to the simple object dataset. Since the damage is applied as normal to each face of the base object, the object's faces were subdivided until a sufficient granularity was achieved so that the damage was noticeable (Figure 3-13). The node group was added to each object's Blender file and a new STL file was saved for each damage variant (Figure 3-14a)). For each simple object, 25 damaged STL files were created to have a large variation of damage in the test dataset.



a) Undivided object model

b) Subdivided object model

Figure 3-13: Cube model before and after face subdivision



a) STL of damaged pyramid

b) Partial point cloud of the damaged STL file

Figure 3-14: Example of damaged pyramid STL and damaged partial scan

The same code used to create the random partial scans test dataset in 3.1.4 was used on each of the damaged STL files to create the random test point clouds (Figure 3-14b)). Ten point clouds were created from each STL file, creating 250 per object and 1000 total anomalous point clouds. To designate if a test data point cloud contained damage, a new label was created in the directory which assigned a zero to all normal data and a one for all anomalous data. This had to be done manually for the damaged scans due to the nature of their creation. The partial scans were created by randomly generating a bounding box. Since the damage is randomly generated on the surface of the object and the slicing of the point cloud is also random, there is a chance that the damage could be missed, and the final point cloud would not contain anything identifiable as anomalous. Each point cloud had to be assessed visually and manually approved as containing anomalous damage. This reduced the overall number of anomalous point clouds to 616. A breakdown of the per-object anomalous point clouds can be seen below in Figure 3-15.

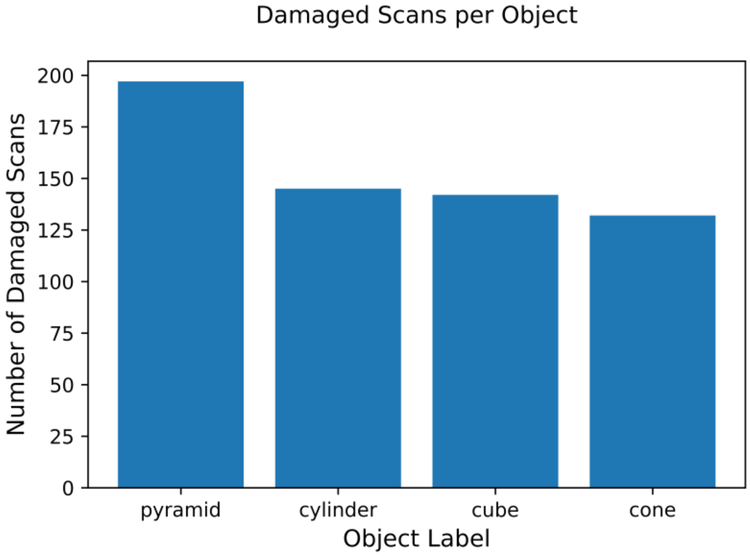
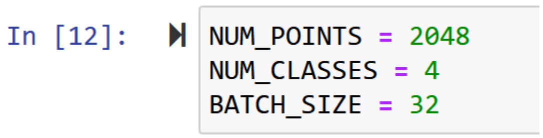


Figure 3-15: Breakdown of damaged scans by object label

4 Optimal Training Data Model Experiment

The Optimal Training Data Model was used to determine what type of training data would be the most effective in classifying the object. Classifying the objects is one of the constraints of this research, and it is a more straight forward and definable task compared to anomaly detection. The options for the type of training data were full scans, targeted partial scans, random partial scans, or a mixture of the full scans and random partial scans. A jupyter notebook was created to train the PointNet algorithm on each of the different types of training data to compare the loss, accuracy, area under the curve, and recall for each type. Making the model in a jupyter notebook allowed for greater control while debugging since code cells could be executed individually in comparison to a normal python script which executes everything all at one. This model was heavily based off the Keras point cloud classification tutorial authored by David Griffiths [35]. Keras is a Python deep learning API that implements the ML platform TensorFlow [51].

The roots for all the file locations were set appropriately depending on the type of training data being assessed. The directories for the test and training data were then read into pandas DataFrames (Figure 9-7). Due to an oversight in data generation, the test partial scans were not saved with the right label in the directory. This was rectified in the next step, shown in Figure 9-8. The labels were then mapped to numerical values. This was done in alphabetical order to keep the numerical labels consistent between trials (Figure 9-9). A function was then defined to access and down sample the individual point clouds. The function took in the number of points that were being sampled from each point cloud and returned multiple arrays containing the points and their label (Figure 9-10). The default hyperparameters were specified and the points and labels for each point cloud were extracted using the dataset function. For this model, the hyperparameters of the number of sampled points and the batch size of data being processed were kept consistent between all training models (Figure 4-1).



```
NUM_POINTS = 2048
NUM_CLASSES = 4
BATCH_SIZE = 32
```

Figure 4-1: Predefined parameters

The arrays containing the points and labels were then converted into TensorFlow Dataset objects. TensorFlow Dataset objects are used to streamline the processing of large amounts of data while using the TensorFlow library. Transformations and augmentation can be performed over large amounts of data and functionality is provided for iterating over elements. Converting the data to Dataset objects also directly links the points and labels together beyond the order in the array. The training data is augmented here to introduce more variability to avoid overfitting. The data is augmented by jittering the points in a uniform manner between -0.01 and 0.01 and by shuffling the order of the points in each point cloud (Figure 9-11).

The labels for each of the point clouds were then one hot ended (OHE) (Figure 4-2). One hot encoding means to create a legend for the labels using binary data. Each object represents a column and a “1” is used to signify when that label is “True”. Every other object label would be equal to zero. OHE is often used for categorical data when there is no numerical link between the categories. This makes the categories easier for the model to process. The integer encoding done earlier may be sufficient, but OHE is considered more foolproof as no relationship or numerical order between categories is presumed.

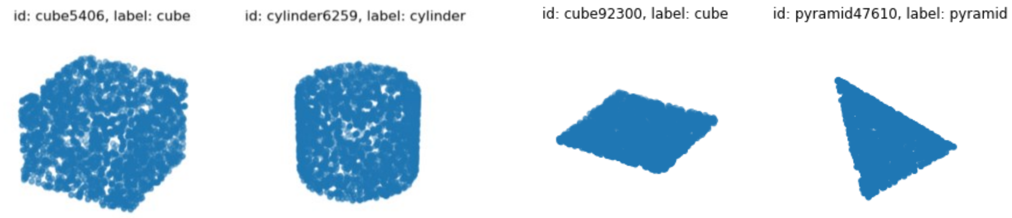
Cone	*Cube*	*Cylinder*	*Pyramid*
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Figure 4-2: OHE object labels

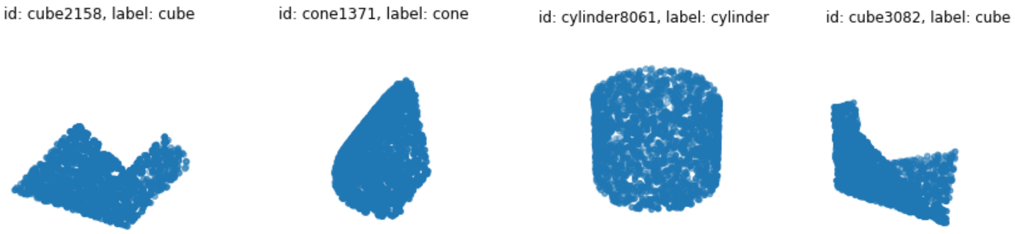
The machine learning model was built next. For the Training Data Test, the model was kept as the basic PointNet algorithm using a Keras implementation (Figure 9-12). The convolution and fully connected dense layers were implemented as functions that included the batch normalization layer and the ReLu activation function. The tnet block was implemented as a function that could be used for both applications in the model. The model was then created following the architecture outlined in Figure 2-7. The model was compiled using a Categorical Crossentropy loss. The optimizer was set to Adam with a learning rate of 0.001. The metrics being measured were accuracy, areas under the curve, and recall. The model was fit on the training dataset for 10 epochs before evaluation. Predictions were then made by applying the trained model to the test dataset. The function MakePredictions() was created to access the predictions for each point cloud, as well as the probabilities calculated for each class label. The predictions were displayed against the ground truth data in a confusion matrix to assess the accuracy of the model.

4.1 Results

The model was run with four different training data styles to find which was the most effective (Figure 4-3). The different styles of training data were Full Scans, Targeted Partial Scans (TPS), Random Partial Scans (RPS), and an equal mix of Full and Random Partial Scans called Mixed Scans. The initial hypothesis was that the best performance would be found from either TPS or RPS based upon the nature of the testing data and the model. Examples of each of the training data types can be found below.



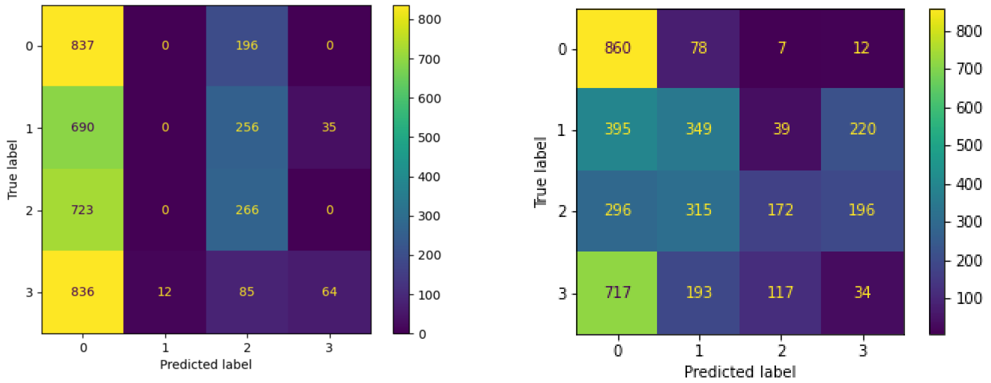
a) Full scans **b) Targeted partial scans**



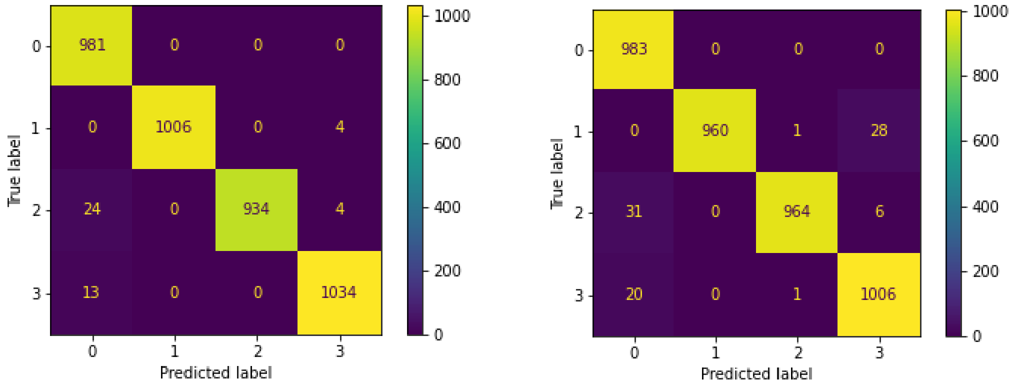
c) Random partial scans **d) Mixed scans**

Figure 4-3: Training data types being tested

For each training data type, the point clouds were down sampled to 2048 points for the purpose of fitting within computational memory restrictions. All experiments were run on a 24GB Nvidia GeForce RTX 3090 GPU with a compute capability of 8.6. The same testing data consisting of only normal test Random Partial Scans was used for all runs. Confusion matrices (Figure 4-4) were used as the primary method of comparing the results, with the accuracy, area under curve (AUC), and recall also being considered (Figure 4-5).

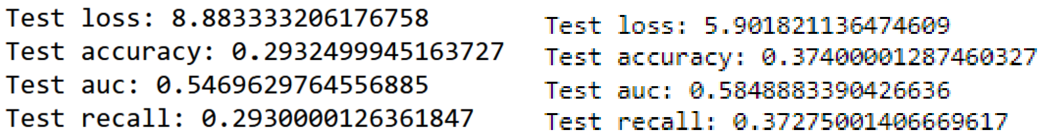


a) Full scans **b) Targeted partial scans**



c) Random partial scans **d) Mixed scans**

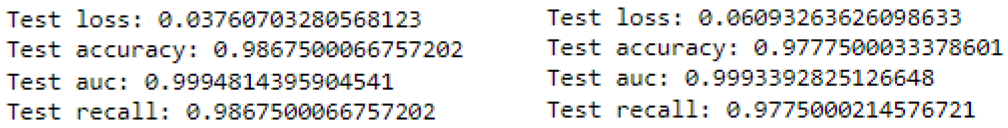
Figure 4-4: Optimal Training Data confusion matrices



```
Test loss: 8.883333206176758
Test accuracy: 0.2932499945163727
Test auc: 0.5469629764556885
Test recall: 0.2930000126361847
```

```
Test loss: 5.901821136474609
Test accuracy: 0.37400001287460327
Test auc: 0.5848883390426636
Test recall: 0.37275001406669617
```

a) Full scans **b) Targeted partial scans**



```
Test loss: 0.03760703280568123
Test accuracy: 0.9867500066757202
Test auc: 0.9994814395904541
Test recall: 0.9867500066757202
```

```
Test loss: 0.06093263626098633
Test accuracy: 0.9777500033378601
Test auc: 0.9993392825126648
Test recall: 0.9775000214576721
```

c) Random partial scans **d) Mixed scans**

Figure 4-5: Optimal Training Data performance metrics

Based upon the confusion matrices, the training data that performed the best with the testing data was the Random Partial Scans (RPS) with a classification accuracy of 98.68%. The Mixed Scans performed comparably with an accuracy of 97.78% but given the poor performance of the Full Scans at 29.32% it can be assumed that mixing the training data types did not provide any additional benefit. The good performance of the RPS makes sense for a couple of reasons. The first is that the style of training data directly matches with the style of the testing data. The model is training on partial scans and learning the expected features from this style of data. Presenting similar data for the testing dataset is more likely to have good results compared to a new style of data that has not been seen before. This is why it was expected Full Scans would perform the worst.

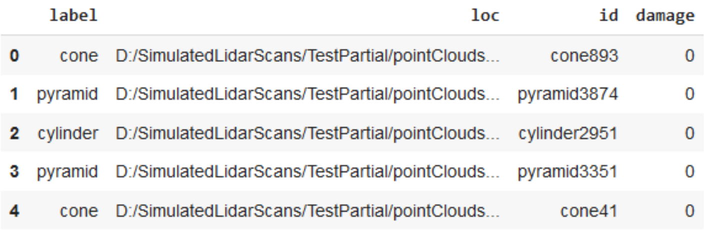
Another reason for the good performance of RPS is that the point clouds include context for the model to learn the objects features. While the intention for the TPS dataset was to highlight the critical features for the PointNet model for easier identification, this may not have had the intended effect. PointNet learns the critical features of the point cloud by learning where changes such as corners happen within the point cloud. By isolating the features, this may inhibit the model's ability to detect where the changes occur leading to a weaker model. With RPS, the scans provide enough context for the critical features to be properly identified. RPS was determined to be the optimal training data style and was used for all models moving forward.

5 Anomaly Detection Models Evaluation

5.1 Single Output Model with Statistical Anomaly Detection

The first model evaluated for anomaly detection was a single output model that used statistical methods for anomaly detection. It used the PointNet algorithm to classify the point clouds as objects and performed anomaly detection based upon the confidence level of the classification. The Random Partial Scans (RPS) were used as the training data since this was found to perform the best, based upon the results from the previous section.

This model followed the same structure as the Optimal Training Data Model. A jupyter notebook was used to facilitate the implementation of the Keras model. The location of the training and testing point clouds were read into training and testing pandas DataFrames. Since anomaly detection was being performed, damaged scans were also used as testing data. The locations of the damaged scans were read in and intermixed with the 'normal' testing data. Before doing this, a 'damage' label was added to the existing testing data where every 'normal' scan was set to 0 (Figure 5-1).



	label	loc	id	damage
0	cone	D:/SimulatedLidarScans/TestPartial/pointClouds...	cone893	0
1	pyramid	D:/SimulatedLidarScans/TestPartial/pointClouds...	pyramid3874	0
2	cylinder	D:/SimulatedLidarScans/TestPartial/pointClouds...	cylinder2951	0
3	pyramid	D:/SimulatedLidarScans/TestPartial/pointClouds...	pyramid3351	0
4	cone	D:/SimulatedLidarScans/TestPartial/pointClouds...	cone41	0

Figure 5-1: Loaded normal testing data

After combining the damaged and normal testing data there were a total of 4616 point clouds in the test dataset. The training data was then split into a training and a validation set. This split was set to an 80-20 training-validation split (Figure 9-13). This left 6400 point clouds in the training set with 1600 being allocated to the validation set.

The label map was created the same way as the previous model (Figure 9-9). The base parameters for the number of points in the point clouds, the number of classes, and the batch size were set the same as in the Optimal Data Model (Figure 4-1). These parameters were not able to be tuned with the other hyperparameters since they were used to create the Dataset objects and had to be set ahead of time. They were manually tuned by comparing various runs to find the optimal conditions, as discussed in the Results and Analysis section.

The Dataset objects were then created from the point clouds using the same method as the previous model. A small variation was a validation Dataset was created at the same time as the training and testing dataset objects. The other change from the previous model's Dataset generation function was the addition of the damage labels as a target for the testing Dataset (Figure 9-14). This allowed for the multiple objectives of this model to be achieved. The same data augmentation was applied to the training dataset.

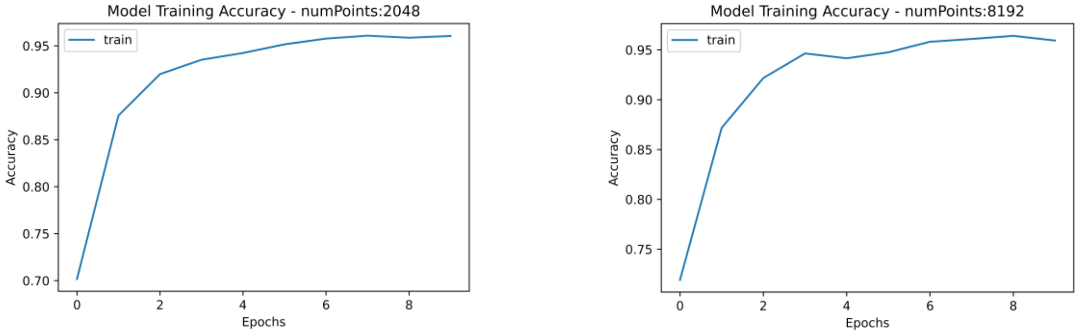
The PointNet classification model was then built in the same way as in the Optimal Training Data Model with the same parameters for the layers and activation functions. Predictions were then made using the trained model on the test dataset. These predictions returned the predicted class label for each point cloud as well as the probability of the point cloud belonging to each class label. The predicted classification labels were displayed in a confusion matrix against the ground truth data to evaluate the classification performance. A Z-Score was then calculated from the probabilities of each of the predictions using the getZScore() function (Figure 9-15). For each point clouds top predicted label, the percentage of that prediction was collected. This represents how confident the SoftMax output is that it predicted the correct label. The mean and standard deviation of these top predictions were then calculated. The threshold for what was considered anomalous data was created by saying $threshold = mean + std$ of the Z-Scores.

The anomalous data was then predicted by comparing the calculated Z-Score for each test point cloud with the calculated threshold. The point clouds with Z-Scores greater than the threshold would be labeled anomalous. These predictions were plotted in a confusion matrix using against the ground truth damage labels from the test dataset.

5.1.1 Results

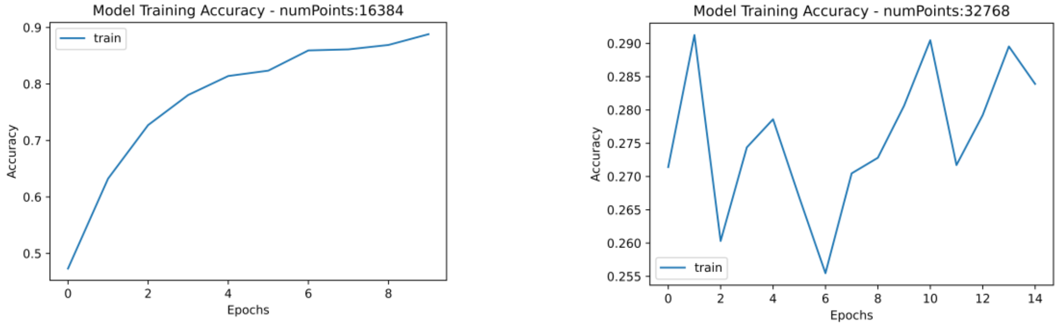
The single output model was the first method used for anomaly detection. The same PointNet model from the Optimal Training Data Model was trained over a minimum of 10 epochs and used to predict the class labels of the test dataset. The predictions were returned as a percent probability of a point cloud belonging to each potential class. The class with the highest percentage became the predicted label for that point cloud. For anomaly detection, the percentage predictions were used to determine which point clouds had damage. The theory was that for a normal and a damaged point cloud belonging to the same class, the percent confidence would be different for their class predictions. Since the identifiable features in the normal point cloud would more closely match the training dataset, the confidence in the prediction would be greater than for the damaged dataset. It was hypothesized that damaged point clouds would be more difficult to classify correctly, and statistical methods could be used to differentiate them from the normal point clouds based on their classification scores. This method was first tested with point clouds containing 2048 points in batches of 32 clouds.

The results from the first run performed poorly, as seen in Figure 5-4 a), b). While the classification accuracy was a very high 97.89%, the anomaly detection went very poorly. Overall, only 6.15% of the damaged point clouds were identified correctly, with ‘Cones’ having the highest damage identification percentage at 14.66%. Evidently the model was doing a very good job at classifying the object type for each scan and there was not a noticeable difference in classification confidence between normal and damaged scans. One of the reasons for this may be the resolution of the point clouds. The point clouds were down sampled from their original amount of 50,000 points to the computationally efficient number of 2048 points. Too much down sampling will result in the damage not being noticeable since the space between “point” could cause the damage to be missed. To test this hypothesis, the same model was run on point clouds of increasing size to establish a relationship between resolution and the ability to identify damage. The model was run with point clouds containing 8192, 16384, and 32768 points. The batch sizes had to be decreased to train models with data of this magnitude, meaning fewer point clouds could be processed at a time and the models took longer to run. The different batch sizes meant some point clouds had to be dropped to have complete batches, resulting in slight differences in the total number of clouds in each testing dataset.



a) 2048 points

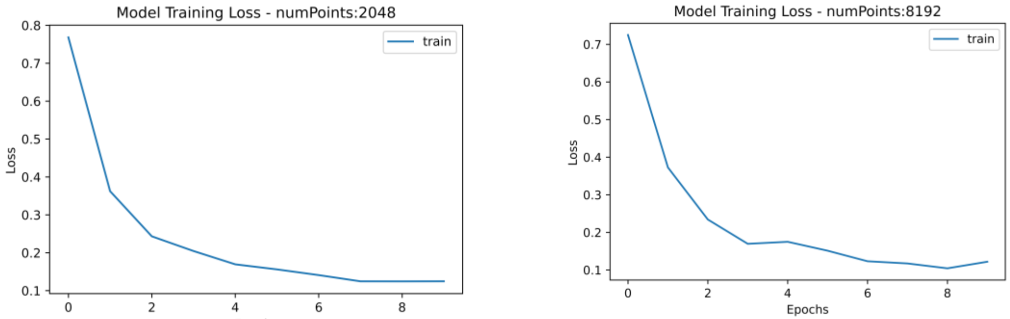
b) 8192 points



c) 16384 points

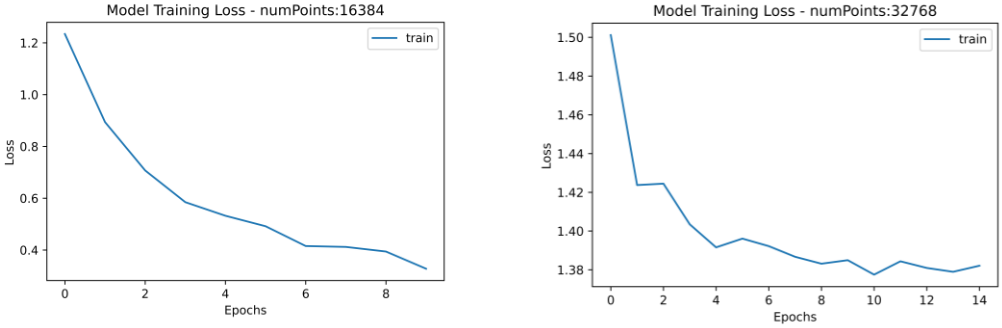
d) 32768 points

Figure 5-2: Model training accuracy with increasing point cloud resolution



a) 2048 points

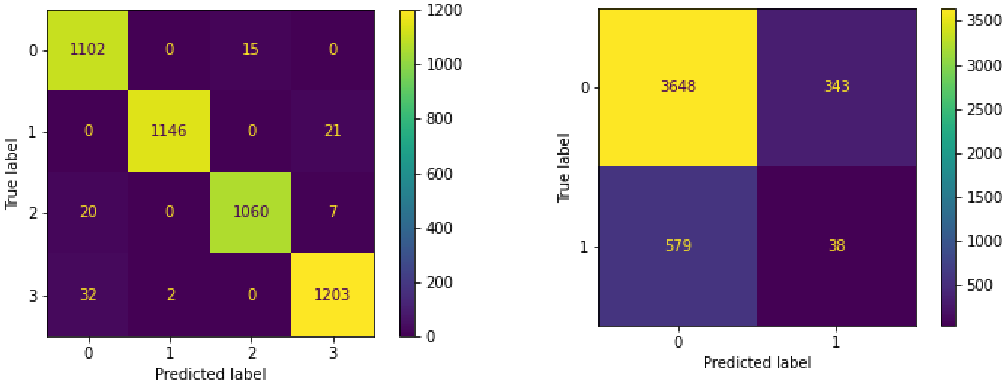
b) 8192 points



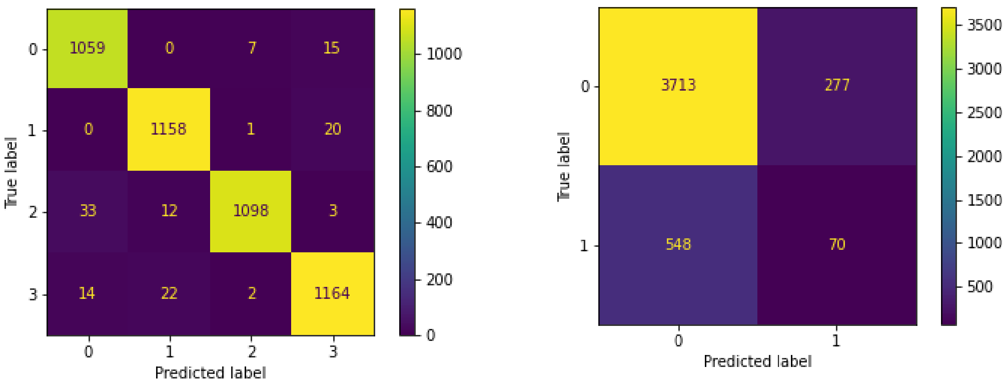
c) 16384 points

d) 32768 points

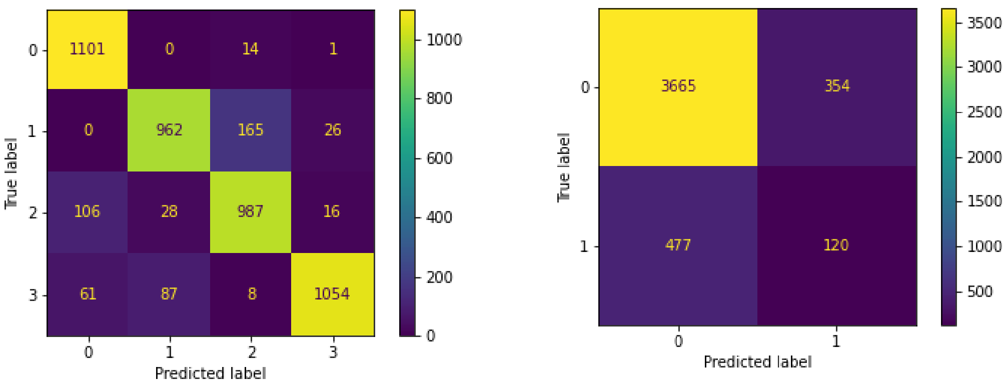
Figure 5-3: Model training loss with increasing point cloud resolution



a) **2048-Point Class CM** b) **2048-Point Anomaly CM**



c) **8192-Point Class CM** d) **8192-Point Anomaly CM**



e) **16384-Point Class CM** f) **16384-Point Anomaly CM**

Figure 5-4: Confusion matrices for classification (left) and anomaly detection (right) with increasing point cloud resolution

As seen in Figure 5-2 and Figure 5-3, the 8192 and 16384 models trained well over the 10 epochs, while the 32768 was never able to train even with additional epochs used.

This is potentially because of computational limitations, or because the nature of the model meant there were too few parameters for the number of points and no patterns could be determined. Given that, no predictions were made using the 32768 model. It was found that overall, the anomaly detection did improve with increased point cloud resolution. The best performing model was found using point clouds containing 16384 points and it was able to detect damaged point clouds with a recall of 20.10%. While still not desirable, this is a dramatic increase from the recall of the 2048 model at 6.15%.

However, it can also be seen that while the damage detection increased, the classification accuracy dropped to 89.06% from the high of 97.89% with the 2048 model. Given the anomaly detection performance increased, it can be assumed that many of the improperly classified point clouds were damaged and that is why they were not classified properly. This indicates that as the resolution of the point cloud increased, the damage became more easily detectable, and the model had more difficulty classifying the damaged point clouds as belonging to a specific class. It also could be that the number of points in comparison to the parameters of the model did not allow for proper training and for features to be completely identified, like with the 32768 model. To test this, a new model would need to be built with more layers to see if training improved. Either way, the limit of the method with the current model was shown.

Unfortunately, the need to increase point cloud resolution and the size of the model required to make this method feasible would be impractical to implement due to the expected computational limitations available on the space station. Additionally, these results show that when using this method, the two objectives of accurate object classification and damage detection are reciprocals making this method sub-optimal. Even with hyperparameter tuning this would not be a practical method since both classification and anomaly detection need to be performed to a high degree.

5.2 Multi-Output Model with Autoencoder Anomaly Detection

The next model evaluated for anomaly detection was an autoencoder. However, classification of the point cloud still needed to occur which could not be easily done using only an autoencoder reconstruction model. To address this, a multi-output model was made to perform both classification and anomaly detection using an autoencoder (Figure 5-5). The datasets were created using the same methods from the Single Output model presented above. The same PointNet architecture was used to encode the point clouds and extract their features. The split into multiple outputs occurs after the Max Pooling layer congregates the features. The first output was the classification part of the model which continued the same way as the previous models. Dense layers were used to reduce the number of features and a SoftMax activation function was used to predict the class of the point cloud. The second output was a fully connected decoder. It takes the features from the encoder and learns them for the purpose of reconstructing the point cloud as the output. The output is a list of points that is a reconstruction of the original point cloud. Both outputs were used by passing them as a list as the model's output (Figure 9-16).

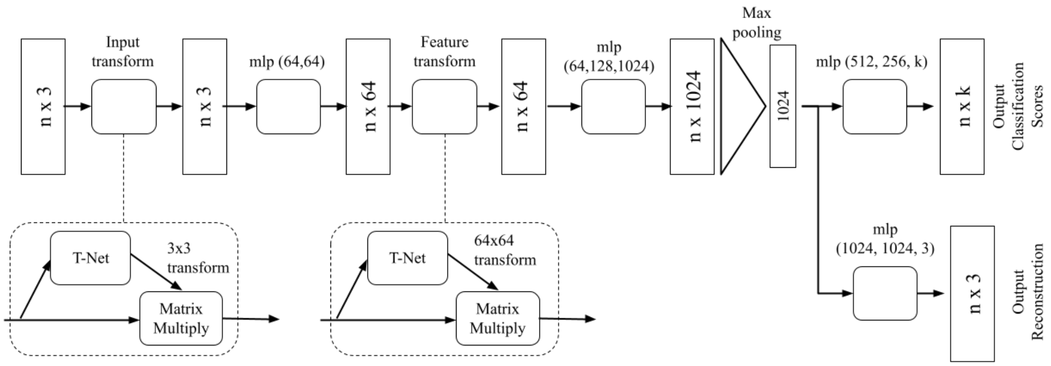


Figure 5-5: Multi-output model architecture, with a PointNet encoder and classification and reconstruction outputs

The parameters for the PointNet encoder and the classification output were set using the same values from the original PointNet paper. For the reconstructing decoder, the architecture and initial parameters were chosen based off a GitHub repository created by PointNet author Charles Qi [52]. In the repository, multiple decoder options are presented. This architecture was based off the default model seen in *model.py* in the repository and can be seen in the above Figure 5-5**.**

When the model was compiled, different loss functions were used for each output due to the different objectives to be achieved. The classification output used the Keras provided Categorical Crossentropy loss that has been used for all previous models. The autoencoder output used Chamfer Distance (21) as the loss function which did not come implemented in the Keras library and had to be implemented as a custom metric. Research was done into how to implement this in an efficient way. Since the datasets were generated as Tensorflow Dataset objects, methods that used Numpy arrays could not be easily implemented. The method that ended up being chosen was presented by StackOverflow User keineahnung2345 [53]. This method was chosen because the Tensorflow implementation was easily scalable to use on large, batched, Datasets (Figure 9-17). Additionally, the User provided other non-Tensorflow methods to calculate CD that were used to validate results using fake data. This provided the level of confidence needed to move forward with this metric.

To train the model, the data were read into the Jupyter Notebook using the same procedure as the previous models. However, since this model contained multiple outputs, the Dataset objects needed to be generated differently. For each input point cloud, there were two target outputs. One was the object label for the classification, and the other was the desired point cloud formation, which in the case of reconstruction is the ground truth input data. The autoencoder is technically an unsupervised model since it is not being told exactly what the point cloud should look like, simply that the difference between the output and input point clouds should be minimized. When using the Keras function API for implementation, the ground truth data still needed to be provided as a target so that it could be used as a part of the loss calculation (Figure 9-18). Because of this, the amount of memory required to load a point cloud into a Dataset object doubled as the input points were also being allocated as output points. This limited the resolution of the point clouds to 4096 points, as any larger would cause the GPU to run out of memory even with minimal batch sizes.

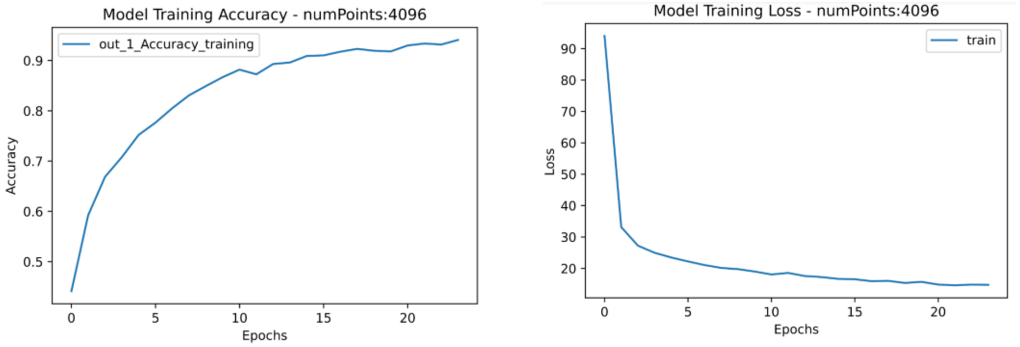
After training the model, predictions were made for the testing data. This meant for each test point cloud the output generated would be a class label and a reconstructed point cloud. The predicted class labels were compared to the ground truth labels by using a confusion matrix like in previous models. For determining the anomalous point clouds, a multi-step process was used.

First, the trained model was used to create predictions on the training data. This was done to create best-case-scenario point cloud reconstructions. Since the data had been used during training, the model was expected to perform better on this data than for the testing data. The chamfer distances were then calculated between the training and reconstructed point clouds. The mean and standard deviation of the chamfer distances were made to create a threshold for determining what is considered anomalous. Next, the chamfer distances were calculated between the testing point clouds and their reconstructed equivalents. These distances were compared against the threshold and anything higher than it was considered a damaged point cloud. These predictions were then compared to the ground truth 'damage' label that the testing dataset has.

5.2.1 Results

The Multi-Output Model was used to perform both classification and reconstruction on each point cloud. Output 1 of the model returned the percent probability of the object belonging to each of the class labels, the same as in the previous models. Output 2 was a new output consisting of a point cloud reconstruction of the original point cloud and was used for anomaly detection. The reconstructed point cloud was compared to the original input point cloud and the difference between the clouds would be the loss of the model. The theory is that point clouds containing damage will have a greater reconstruction error since the features of the point cloud don't match the features learned by the model during training. By getting the best-case reconstruction errors from the model on the training data, a threshold was created that was used to predict damage for any point cloud with a reconstruction error greater than the threshold.

With the Datasets loaded, the model was trained. For each output, the loss and accuracy of the model's performance were logged. To maximize performance, the model was set to train for a maximum of 50 epochs with an early stopping condition. If over the course of three epochs a change of less than 0.5 was detected in the loss of Output 2 the training would stop. The loss of Output 2 was the CD calculation and if no change occurred then training has reached a steady state. As seen in Figure 5-6 the model trained for 23 epochs.

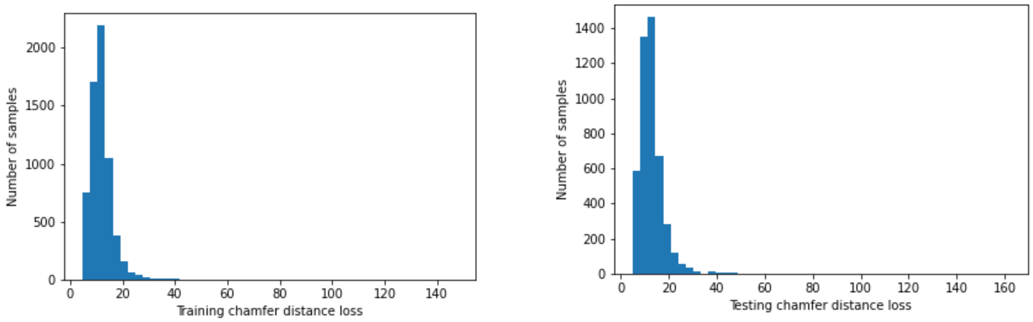


a) Model Accuracy **b) Model Loss**

Figure 5-6: Multi-output model loss and accuracy

After training, predictions were first made on the training data. The classification predictions were discarded, while the reconstructed point clouds were used to create the anomaly threshold. The CD was calculated between each of the reconstructed and original point clouds (Figure 5-7 a)). It should be noted that since CD is a distance metric, the magnitude of the loss will be correlated to the scale of the point clouds. For this experiment, the mean and standard deviation of the training CD were used to get the reconstruction error threshold seen in **Error! Reference source not found.** by saying that $threshold = mean + (0.5 * std)$. This choice was made based upon the same statistical methods that were chosen to calculate the threshold in the previous Single Output model. However, the datapoints from the Single Ouput model were normalized by calculating the ZScore, and the errors from the Multi-Output CD metric were not. Therefore, using the mean and standard deviation to calculate the threshold was not appropriate for this data since the distribution was not normalized. In future experiments this was realized, and the threshold calculation metric was changed.

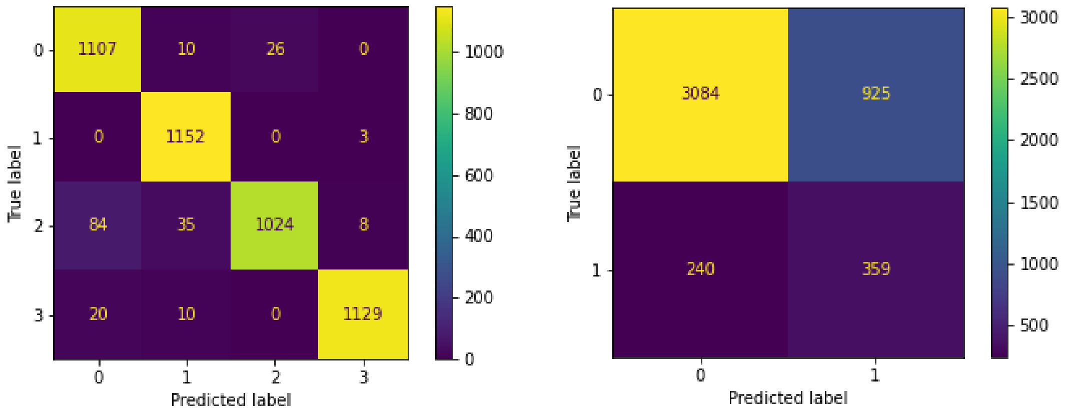
The intended purpose of the current threshold calculation was to use the mean and standard deviation to calculate a threshold that landed in the theoretical transitional range between normal and damaged point clouds. There is some overlap between the errors of the normal and damaged point clouds, and it would be impossible to completely split them cleanly into separate groups. For the research problem of identifying damage on a spacecraft, it is more important to avoid missing damaged point clouds than it is to avoid falsely labeling normal point clouds as having damage. Half of the standard deviation was added to the mean as an intended conservative estimate that would allow for most of the damaged scans and some of the normal scans to be classified as anomalies. On this dataset this resulted in a reconstruction error threshold of 14.199.



a) Training CD distribution **b) Testing CD distribution**

Figure 5-7: Chamfer Distance distributions for training and testing datasets

After getting the reconstruction error threshold from the training data, predictions were then made on the testing data using the custom prediction function. A classification confusion matrix was made from the results to evaluate the performance of the classification side of the model (Figure 5-8 a)). The results were admirable, with 95.7% of the point clouds being accurately labeled as the correct object. The reconstructed point clouds of the testing dataset were used to calculate the CD for each point cloud, with the CD loss distribution being shown in Figure 5-7 b). After using the reconstruction error threshold to predict anomalies, the results were compared in a confusion matrix to the ground truth damage labels with '1' representing damaged point clouds (Figure 5-8 b)).



g) Classification CM **h) Anomaly CM**

Figure 5-8: Confusion matrices for classification and anomaly detection of the multi-output model

Even with the incorrect threshold calculation method, the results showed a great improvement over the Single Output Statistical anomaly detection method. This method resulted in an anomaly detection recall of 59.9%, an increase from the previous best of 20.10%. While not incredible, there is potential for improvement in the model to be able to better differentiate between the normal and damaged point clouds. This model also shows promise since the recall was achieved with a False Positive (FP) rate of 23.07%. Given that it is very important to identify the anomalous damage and the distribution overlap of the normal and damaged testing data is unknown, a FP of 25% or below is considered appropriate for this research. These results also show that when using this method optimizing one objective did not worsen the other. Both object classification and damage detection could be optimized. Given this, the Multi-Output method was chosen for moving forward with hyperparameter tuning.

6 Hyperparameter Tuning and Final Test Results

The results from section 5 showed that the Multi-Output model performed the best for anomaly detection, so this was the model chosen for the final hyperparameter tuning to optimize the performance. Hyperparameter tuning was done by modifying the existing multi-output model so that the performance with various hyperparameter values could be evaluated. This was done using the KerasTuner framework API. KerasTuner allows for the easy implementation of hyperparameter tuning algorithms on Keras models. Random Search and Bayesian Optimization algorithms are built-in and can be implemented as KerasTuner objects.

To modify the existing model for optimization, it was recreated using the base HyperModel class. The class allows for the search space of the hyperparameters to be defined within a model by overriding the *build()* and *fit()* methods of the class. The previously created model was implemented in the overridden *build()* method with the value options for the various hyperparameters provided for the tuner. The specific hyperparameters were identified in the model by using the KerasTuner argument *hp* which is used to pass in a Keras HyperParameter class object.

The HyperParameter class allows for the definition of hyperparameters with a variety of different search spaces. The first hyperparameter in the model is the number of nodes used in each layer of the model. The PointNet algorithm follows a specific architecture of the number of nodes in the layers used to encode the point clouds. These variations in layer size are all multiples of the base number of nodes. To preserve this relationship, only the base number of nodes was implemented as a hyperparameter while the sizes of the other layers were set using the existing ratios. The layers and their ratios can be seen in Figure 9-19. For this hyperparameter, the HyperParameter *Int* method was used to create the search space. The space was defined so the base node number could be an integer between the values of 32 to 256 using a step size of 32.

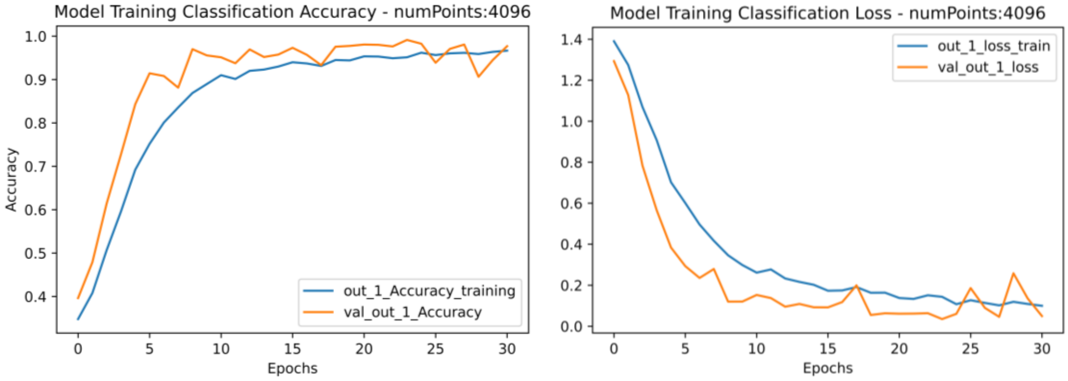
The other hyperparameters used to build the model were momentum values in the batch normalization layers and the activation functions used in the convolution and dense layers. The momentum was defined using a HyperParameter *Float* method with the range being between 0.0 to 0.8 with a step size of 0.1. The activation functions were decided to either be ReLU or tanh by using the HyperParameter *Choice* method, which makes a choice between a predetermined list of values. In this case, the values were Strings which identified the function to be used. The final hyperparameter was created during the compilation of the model. The Adam function was being used as the optimization function, and a hyperparameter was set to tune the value for the learning rate. It was also chosen using the *Float* method, with the range being from 0.001 to 1 using a base-10 log for the step size.

After defining the model through the *build()* and *fit()* methods, the HyperModel was passed into a Tuner object of the desired search method. The objectives of the Tuner were then set. The Tuner was told to minimize the loss from the autoencoder output and maximize the accuracy from the classification output on the validation dataset. The number of trials for the Tuner to run with different hyperparameter values from the search space was set to 15. Each hyperparameter configuration was additionally set to run twice to minimize the risk of bad results from poor initialization. The best results from the Tuner were then used to train the model completely and the results were analyzed using the same methods as in the multi-output model. This was done using both the RandomSearch and Bayesian Optimization tuners to compare the results of the multi-output model.

6.1 Results

Random Search was chosen for the hyperparameter tuner because of the size of the search space and the speed of the tuning algorithms. Doing a Grid Search would take an excessive amount of time to cover the search space and Random Search allows for a large variety of combinations to be seen. While Bayesian Optimization may converge on the optimal hyperparameters with fewer iterations, it takes longer to run. When working on a shared resource such as the lab computer, time is limited so this tuner was not chosen. The best hyperparameters found by the tuner can be seen in Appendix B and the summary of the model using the hyperparameters is in Appendix C.

With these hyperparameters, the model was trained with the same conditions as the previous multi-output model. The model was allowed to run for a maximum of 100 epochs with an early stopping condition in place that monitored the Output 2 loss for lack of change. The model trained for a total of 30 epochs, and the accuracy and loss can be seen in Figure 6-1 and Figure 6-2.



a) Model Accuracy **b) Model Loss**

Figure 6-1: Hyperparameter tuned model classification accuracy and loss

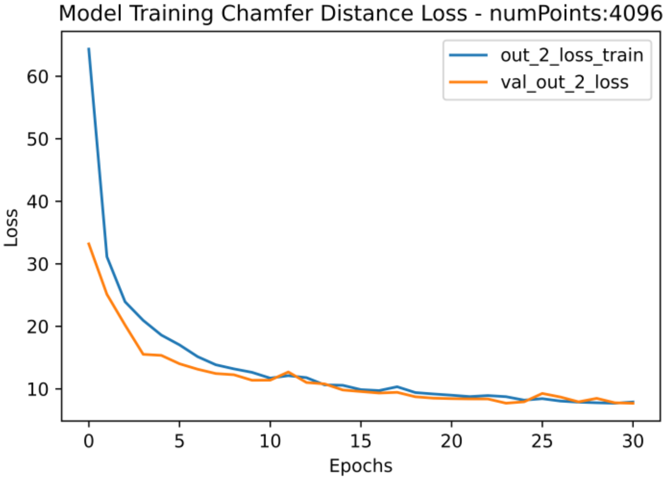
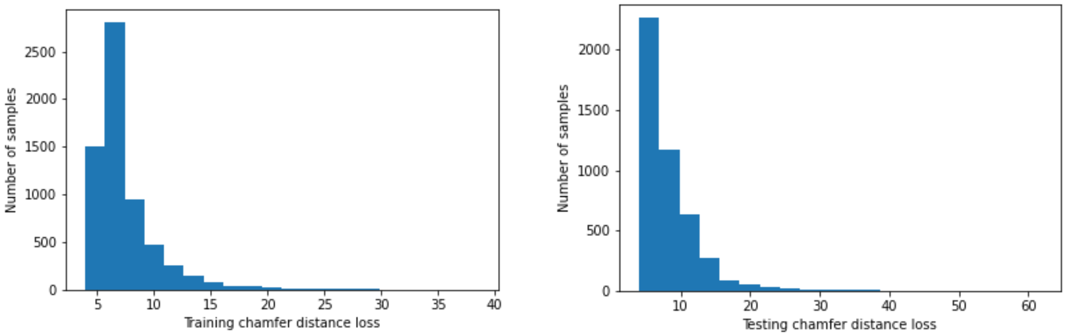


Figure 6-2: Hyperparameter tuned model Chamfer Distance loss

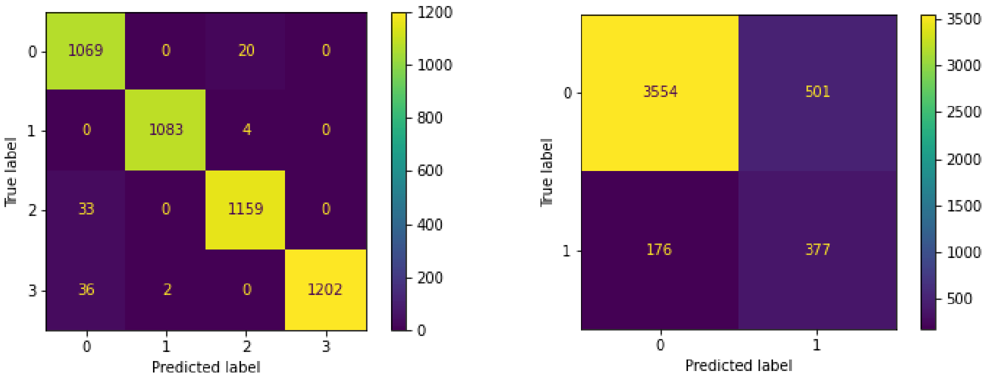
In comparison to the original multi-output model, the tuned model reached a lower loss before reaching steady state. This is also reflected in the CD loss distribution plots (Figure 6-3) for both the training and testing data. Both distributions have peaks closer to zero than with the untuned model. Using the Training CD losses, the reconstruction error threshold was calculated. At this time, the incorrect calculation was still being used which resulted in a threshold value of 10.77 (Figure 6-4).



a) Training CD Loss **b) Testing CD Loss**

Figure 6-3:HP model Chamfer Distance distributions

In terms of the classification and anomaly detection, the results are like the untuned model, if slightly improved (Figure 6-4). The classification output improved to 97.93% of the point clouds being predicted correctly. For the anomaly detection output, with the incorrect threshold the recall improved to 68.17%. Additionally, the level of False Positives decreased by a large amount, from 23.07% being incorrectly labeled in the previous model to only 12.35%, which gives a specificity of 87.65%.



a) Classification CM **b) Anomaly CM**

Figure 6-4: HP model classification and anomaly detection confusion matrices, old threshold

The lower FP rate with an improved TP rate shows that there is a clearer distinction between the errors of the damaged clouds and the normal clouds compared to the untuned model. When looking at the Testing CD losses compared to the Training CD losses, there are larger numerical loss values in the Test losses. This means that the algorithm is likely working as expected and the damaged points clouds are being reconstructed worse than the normal point clouds. Optimizing the threshold would improve the anomaly detection results.

It was at this point that the understanding of the distribution of CD Loss and the inappropriateness of the current threshold calculation came about. Instead of using the mean and standard deviation to calculate the threshold, the threshold was calculated by finding the value of the 75th percentile of the training loss distribution. This would provide a threshold value where at a maximum 25% of the training losses would be above the threshold and labeled anomalous. This 25% FP rate is considered acceptable given the expected overlap of the normal and anomalous error values and that the distribution of the anomalous CD errors is not known. It is more important to detect the anomalous data than it is to not falsely label the normal data. A low threshold would provide a better recall, but the FP rate would be too high and would lower the specificity of the results. In the other direction, a threshold that is too high could misclassify many anomalies. Without tuning, it was decided that 25% would be the maximum FP considered acceptable for these reasons. This provided a reconstruction error threshold of 8.131. Predicting the ideal threshold to optimize the anomaly detection is discussed more in Future Considerations.

When the new threshold calculation was used, the anomaly detection confusion matrix showed a drastic improvement (Figure 6-5). The recall improved to 90.42% with an FP rate of 20.69%. The specificity shows 79.31% of the 'normal' test point clouds have CD error values below the threshold, which is a similar but improved distribution from the training data.

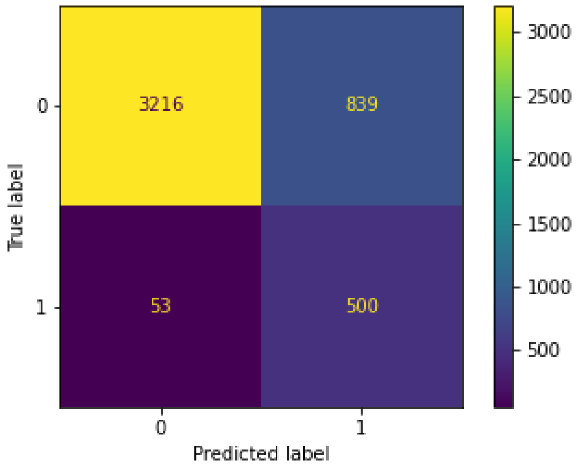


Figure 6-5: Anomaly detection confusion matrix, new threshold

Given these results on the Simple Object dataset, there is merit to using an autoencoder to detect anomalous point clouds.

7 Summary and Conclusion

The results of this research show that the PointNet ML architecture has the potential to allow classifier and autoencoder models to be trained on 3D point cloud data for the successful identification of structural anomaly detection on spacecrafts. Synthetic data comprised of Full Scans, Targeted Partial Scans, and Random Partial Scans with and without damage were generated for basic geometric shapes representing components of a space station. These datasets were used to train and evaluate the model's performance in classification and anomaly detection. Random Partial Scans were determined to be the best style of training data by running the Optimal Training Data experiment. These scans were then used to train a Single Object model for anomaly detection using classification probabilities. The results from this experiment were compared to the results from a Multi-Output model which used an autoencoder for anomaly detection. The Multi-Output model was found to work better and hyperparameter tuning was then performed on the model. The final tuned model was able to classify the point clouds with an accuracy of 97.93% while identifying point clouds that contained damage with a recall of 90.42% and a specificity of 79.31%. This level of anomaly detection accuracy is promising and indicates that the overall approach should be further explored to confirm its feasibility for identifying damaged point clouds on more complex spacecraft components. Recommended next steps for the research are discussed in Future Considerations.

8 Limitations and Future Considerations

While this research explores and presents a possible method for using machine learning to identify anomalous damage on space crafts, there are a few major limitations when it comes to extending this research to a practical application. One of the major ones is the quality and resolution of the data that is needed for this method to work. The synthetic data used in these experiments provided very clean, high resolution representations of each of the objects in the dataset. While noise was added through jittering, the scans were still ‘ideal’ point clouds in that there were no flaws that are introduced through using a LiDAR sensor such as skipped sections. A future step would be to test the model on a dataset containing LiDAR data to see how the ML model performs when introduced to an imperfect environment. Since the scans on the Gateway station will be generated using LiDAR it is important to test this method on real-world instead of synthetic data.

A dataset could be made by 3D printing a model of a spacecraft and then using a LiDAR to take scans of it. Specific damaged components could also be 3D printed so damaged scans could be generated. Scene segmentation would have to be done to isolate the scans into their pre-determined components, after which the method could proceed as defined. Testing on real LiDAR data would provide more confidence in this method’s ability to detect structural damage on the exterior of the Gateway space station. This would also explore if a LiDAR scanner would be able to provide the necessary resolution to perceive the damage and how much computer memory would be required for the LiDAR scans.

This leads to another limitation of this research, which is the memory requirements of the model and the scans. Memory limits were being hit while performing the multi-output experiment on a GPU that had 24GB of RAM. In order to train the model, the resolutions of the scans and the number of scans being trained on at once had to be reduced. In a space application, the computing power is likely to be lower and memory will be an even bigger issue. The model will be pre-trained, but it is still likely that there will be limitations on the size and resolution of the point clouds. Identifying these limitations is an important step for the practical implementation of this method.

A future step for further improvement of the results would be to build a model to predict what the optimal reconstruction error threshold would be for the Testing CD losses. This would be used to fine-tune the threshold and lower the False Positive rate while maintaining the high recall rate of the model. Datasets could be made using the Training CD losses as training data and the Testing CD losses as testing data. The datasets can be generated from a combination of training the MO-model numerous times with the optimal hyperparameters to get various models, and by creating many datasets to get various sets of CD calculations. A new model could be trained to predict a threshold value given a distribution of CD losses that would optimize the recall of the anomaly detection while minimizing the FP rate. This would improve the current procedure by helping to tune the MO-model to optimize the anomaly detection results.

Another next step for this research would be to recreate the multi-output autoencoder on a more complicated dataset. While the Simple Object dataset proves the validity of the method for point cloud anomaly detection, testing the model on a more complicated dataset would be necessary before moving forward to test the robustness of the model. A 3D model of a spacecraft that would normally be used for 3D printing could be used as the main model for the dataset. Components could be identified from the main model and separated to make classification groups. From here the procedure for dataset generation and model testing would proceed as it went for the Simple Object dataset.

References

[1] Canadian Space Agency, "The Lunar Gateway," *The Government of Canada*, Aug. 24, 2021. https://www.asc-csa.gc.ca/eng/astronomy/moon-exploration/lunar-gateway.asp (accessed Mar. 18, 2023).

[2] M. Picard, "An Overview of the Canadian Space Agency's Recent Space Robotics Activities," 2020.

[3] Canadian Space Agency, "Canadarm, Canadarm2, and Canadarm3 – A comparative table," *The Government of Canada*, May 20, 2019. https://www.asc-csa.gc.ca/eng/iss/canadarm2/canadarm-canadarm2-canadarm3-comparative-table.asp (accessed Mar. 18, 2023).

[4] W. Liu, J. Sun, W. Li, T. Hu, and P. Wang, "Deep learning on point clouds and its application: A survey," *Sensors (Switzerland)*, vol. 19, no. 19. 2019, doi: 10.3390/s19194188.

[5] S. A. Bello, S. Yu, C. Wang, J. M. Adam, and J. Li, "Review: Deep learning on 3D point clouds," *Remote Sensing*, vol. 12, no. 11. 2020, doi: 10.3390/rs12111729.

[6] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017, vol. 2017-January, doi: 10.1109/CVPR.2017.16.

[7] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep Learning for 3D Point Clouds: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 12. 2021, doi: 10.1109/TPAMI.2020.3005434.

[8] D. Lazzarotto and T. Ebrahimi, "Sampling color and geometry point clouds from ShapeNet dataset," Jan. 2022, [Online]. Available: http://arxiv.org/abs/2201.06935.

[9] B. Geng, H. J. Zhang, H. Wang, and G. P. Wang, "Approximate Poisson disk sampling on mesh," *Sci. China Inf. Sci.*, vol. 56, no. 9, 2013, doi: 10.1007/s11432-011-4322-8.

[10] J. A. Christian and S. Cryan, "A survey of LIDAR technology and its use in spacecraft relative navigation," 2013, doi: 10.2514/6.2013-4641.

[11] U. Wandinger, "Introduction to Lidar," in *Lidar. Springer Series in Optical Sciences*, vol. 102, New York: Springer-Verlag, pp. 1–18.

[12] D. L. Lu, "Ouster OS1-64 lidar point cloud of intersection of Folsom and Dore St, San Francisco," *Wikimedia Commons*, Dec. 03, 2019. https://commons.wikimedia.org/wiki/File:Ouster_OS1-64_lidar_point_cloud_of_intersection_of_Folsom_and_Dore_St,_San_Francisco.png (accessed Apr. 27, 2023).

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

[14] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, doi: 10.1038/nature14539.

[15] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[16] C. C. Aggarwal, *Neural Networks and Deep Learning*. Cham: Springer International Publishing, 2018.

[17] P. Riley, "Three pitfalls to avoid in machine learning," *Nature*, vol. 572, no. 7767. 2019, doi: 10.1038/d41586-019-02307-y.

[18] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*. Springer, 2021.

[19] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3. 2009, doi: 10.1145/1541880.1541882.

[20] E. Kreyszig, H. Kreyszig, and E. J. Norminton, *Advanced Engineering*

Mathematics, 10th ed. Wiley, 2011.

[21] G. Pang, C. Shen, L. Cao, and A. Van Den Hengel, “Deep Learning for Anomaly Detection: A Review,” *ACM Computing Surveys*, vol. 54, no. 2. 2021, doi: 10.1145/3439950.

[22] T. Wu, L. Pan, J. Zhang, T. WANG, Z. Liu, and D. Lin, “Balanced Chamfer Distance as a Comprehensive Metric for Point Cloud Completion,” in *Advances in Neural Information Processing Systems*, 2021, vol. 34, pp. 29088–29100.

[23] H. Fan, H. Su, and L. Guibas, “A point set generation network for 3D object reconstruction from a single image,” in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017, vol. 2017-January, doi: 10.1109/CVPR.2017.264.

[24] M. Feurer and F. Hutter, “Automated Machine Learning,” F. Hutter, L. Kotthoff, and J. Vanschoren, Eds. Cham: Springer International Publishing, 2019.

[25] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, “Taking the human out of the loop: A review of Bayesian optimization,” *Proceedings of the IEEE*, vol. 104, no. 1. 2016, doi: 10.1109/JPROC.2015.2494218.

[26] M. Sokolova and G. Lapalme, “A systematic analysis of performance measures for classification tasks,” *Inf. Process. Manag.*, vol. 45, no. 4, 2009, doi: 10.1016/j.ipm.2009.03.002.

[27] X. Yao, J. Guo, J. Hu, and Q. Cao, “Using deep learning in semantic classification for point cloud data,” *IEEE Access*, vol. 7, 2019, doi: 10.1109/ACCESS.2019.2905546.

[28] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” 2014, doi: 10.1109/CVPR.2014.81.

[29] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified,

real-time object detection," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, vol. 2016-December, doi: 10.1109/CVPR.2016.91.

[30] C. R. Qi, H. Su, M. Niebner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view CNNs for object classification on 3D data," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, vol. 2016-December, doi: 10.1109/CVPR.2016.609.

[31] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view Convolutional Neural Networks for 3D Shape Recognition," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015, pp. 945–953, doi: 10.1109/ICCV.2015.114.

[32] P. S. Wang, Y. Liu, Y. X. Guo, C. Y. Sun, and X. Tong, "O-CNN: Octree-based convolutional neural networks for 3D shape analysis," in *ACM Transactions on Graphics*, 2017, vol. 36, no. 4, doi: 10.1145/3072959.3073608.

[33] D. Maturana and S. Scherer, "VoxNet: A 3D Convolutional Neural Network for real-time object recognition," in *IEEE International Conference on Intelligent Robots and Systems*, 2015, vol. 2015-December, doi: 10.1109/IROS.2015.7353481.

[34] G. Riegler, A. O. Ulusoy, and A. Geiger, "OctNet: Learning deep 3D representations at high resolutions," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017, vol. 2017-January, doi: 10.1109/CVPR.2017.701.

[35] D. Griffiths, "Point cloud classification with PointNet," *Keras*, May 26, 2020. https://keras.io/examples/vision/pointnet/ (accessed Jan. 03, 2023).

[36] X. Chen, K. Jiang, Y. Zhu, X. Wang, and T. Yun, "Individual tree crown segmentation directly from uav-borne lidar data using the pointnet of deep learning," *Forests*, vol. 12, no. 2, 2021, doi: 10.3390/f12020131.

[37] Y. Liu, B. Fan, S. Xiang, and C. Pan, "Relation-shape convolutional neural network for point cloud analysis," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019, vol. 2019-June, doi: 10.1109/CVPR.2019.00910.

[38] B. S. Hua, M. K. Tran, and S. K. Yeung, "Pointwise Convolutional Neural Networks," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018, pp. 984–993, doi: 10.1109/CVPR.2018.00109.

[39] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering," *Adv. Neural Inf. Process. Syst.*, Jun. 2016, [Online]. Available: http://arxiv.org/abs/1606.09375.

[40] G. Te, W. Hu, A. Zheng, and Z. Guo, "RGCNN," in *Proceedings of the 26th ACM international conference on Multimedia*, Oct. 2018, pp. 746–754, doi: 10.1145/3240508.3240621.

[41] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017, vol. 2017-January, doi: 10.1109/CVPR.2017.11.

[42] R. Klokov and V. Lempitsky, "Escape from Cells: Deep Kd-Networks for the Recognition of 3D Point Cloud Models," in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017, vol. 2017-October, pp. 863–872, doi: 10.1109/ICCV.2017.99.

[43] T. Griebel, D. Authaler, M. Horn, M. Henning, M. Buchholz, and K. Dietmayer, "Anomaly Detection in Radar Data Using PointNets," in *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, 2021, vol. 2021-September, doi: 10.1109/ITSC48978.2021.9564730.

[44] M. Masuda, R. Hachiuma, R. Fujii, H. Saito, and Y. Sekikawa, "TOWARD

UNSUPERVISED 3D POINT CLOUD ANOMALY DETECTION USING VARIATIONAL AUTOENCODER," in *Proceedings - International Conference on Image Processing, ICIP*, 2021, vol. 2021-September, doi: 10.1109/ICIP42928.2021.9506795.

[45] H. Wang, Q. Liu, X. Yue, J. Lasenby, and M. J. Kusner, "Unsupervised Point Cloud Pre-training via Occlusion Completion," 2021, doi: 10.1109/ICCV48922.2021.00964.

[46] "Blender Python API." Blender Foundation, Dec. 20, 2022.

[47] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A Modern Library for 3D Data Processing," Jan. 2018.

[48] S. Katz, A. Tal, and R. Basri, "Direct visibility of point sets," *ACM Trans. Graph.*, vol. 26, no. 3, 2007, doi: 10.1145/1276377.1276407.

[49] CBaileyFilm, "Geometry Nodes Procedural Craters Blender Tutorial," Apr. 19, 2022. https://www.youtube.com/watch?v=RrAz5ccOp1c (accessed Jan. 15, 2023).

[50] S. Worley, "A cellular texture basis function," 1996, doi: 10.1145/237170.237267.

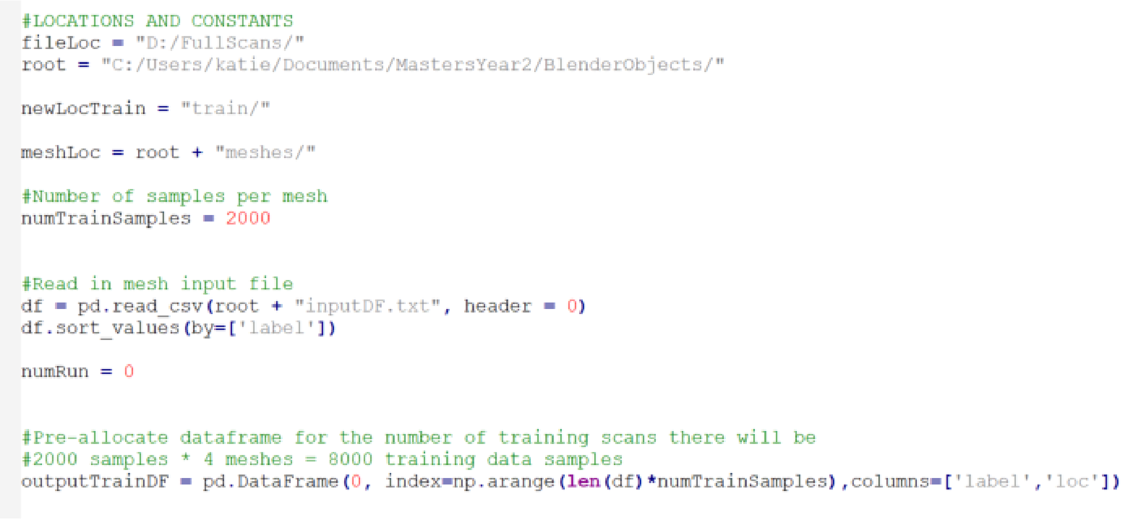
[51] F. Chollet, "Keras." GitHub, 2015, Accessed: Feb. 26, 2023. [Online]. Available: https://github.com/keras-team/keras.

[52] C. R. Qi, "pointnet-autoencoder." May 31, 2018, Accessed: Mar. 13, 2023. [Online]. Available: https://github.com/charlesq34/pointnet-autoencoder.

[53] keineahnung2345, "Chamfer distance between two point clouds in tensorflow," Feb. 19, 2019. https://stackoverflow.com/questions/47060685/chamfer-distance-between-two-point-clouds-in-tensorflow (accessed Mar. 13, 2023).

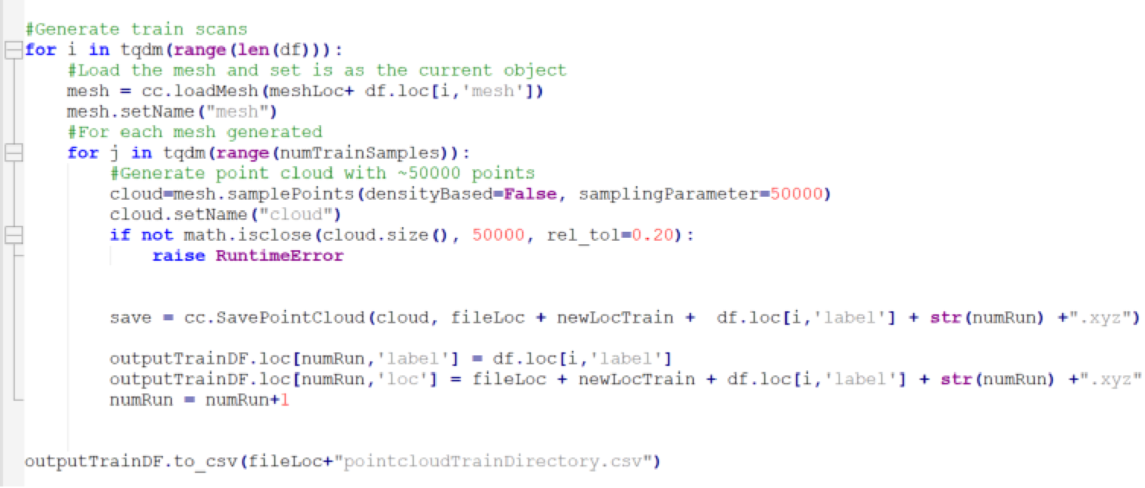
9 Appendices

Appendix A: Code Snippets



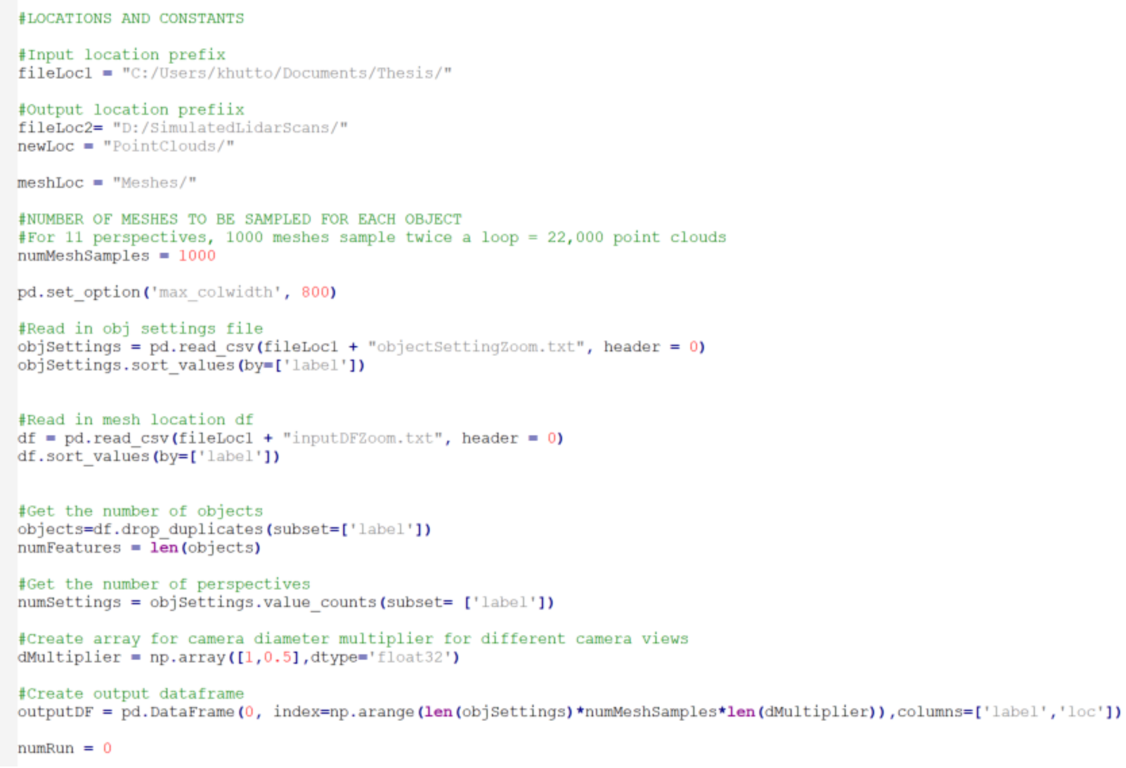
```python
#LOCATIONS AND CONSTANTS
fileLoc = "D:/FullScans/"
root = "C:/Users/katie/Documents/MastersYear2/BlenderObjects/"

newLocTrain = "train/"

meshLoc = root + "meshes/"

#Number of samples per mesh
numTrainSamples = 2000


#Read in mesh input file
df = pd.read_csv(root + "inputDF.txt", header = 0)
df.sort_values(by=['label'])

numRun = 0


#Pre-allocate dataframe for the number of training scans there will be
#2000 samples * 4 meshes = 8000 training data samples
outputTrainDF = pd.DataFrame(0, index=np.arange(len(df)*numTrainSamples),columns=['label','loc'])
```

Figure 9-1: FullScan.py preamble



```python
#Generate train scans
for i in tqdm(range(len(df))):
    #Load the mesh and set is as the current object
    mesh = cc.loadMesh(meshLoc+ df.loc[i,'mesh'])
    mesh.setName("mesh")
    #For each mesh generated
    for j in tqdm(range(numTrainSamples)):
        #Generate point cloud with ~50000 points
        cloud=mesh.samplePoints(densityBased=False, samplingParameter=50000)
        cloud.setName("cloud")
        if not math.isclose(cloud.size(), 50000, rel_tol=0.20):
            raise RuntimeError


        save = cc.SavePointCloud(cloud, fileLoc + newLocTrain +  df.loc[i,'label'] + str(numRun) +".xyz")

        outputTrainDF.loc[numRun,'label'] = df.loc[i,'label']
        outputTrainDF.loc[numRun,'loc'] = fileLoc + newLocTrain + df.loc[i,'label'] + str(numRun) +".xyz"
        numRun = numRun+1


outputTrainDF.to_csv(fileLoc+"pointcloudTrainDirectory.csv")
```

Figure 9-2: FullScan.py main loop

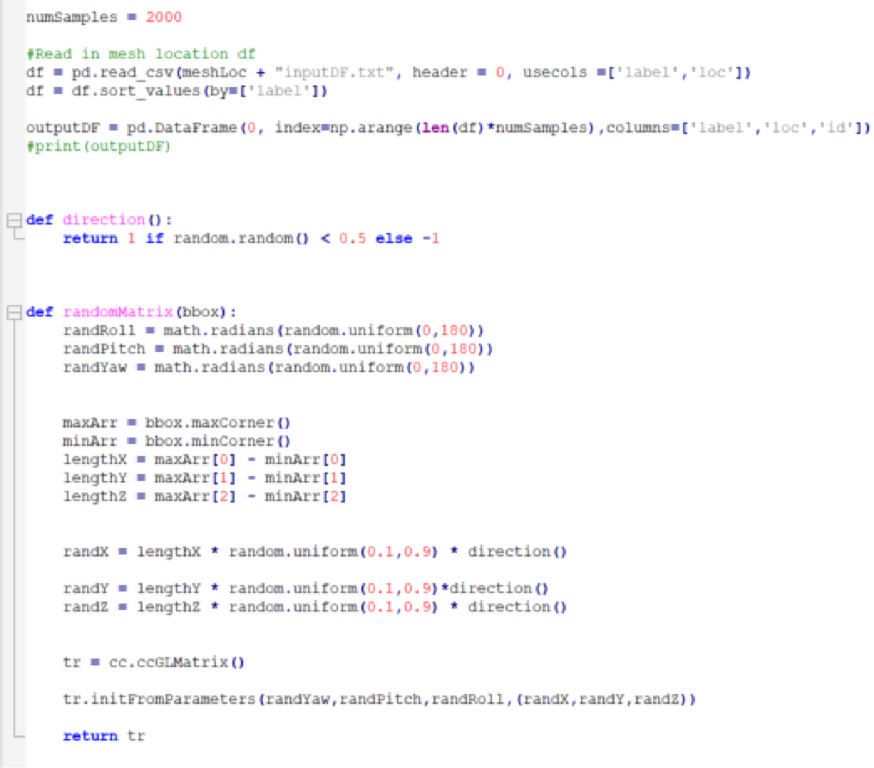


```
#LOCATIONS AND CONSTANTS

#Input location prefix
fileLoc1 = "C:/Users/khutto/Documents/Thesis/"

#Output location prefiix
fileLoc2= "D:/SimulatedLidarScans/"
newLoc = "PointClouds/"

meshLoc = "Meshes/"

#NUMBER OF MESHES TO BE SAMPLED FOR EACH OBJECT
#For 11 perspectives, 1000 meshes sample twice a loop = 22,000 point clouds
numMeshSamples = 1000

pd.set_option('max_colwidth', 800)

#Read in obj settings file
objSettings = pd.read_csv(fileLoc1 + "objectSettingZoom.txt", header = 0)
objSettings.sort_values(by=['label'])


#Read in mesh location df
df = pd.read_csv(fileLoc1 + "inputDFZoom.txt", header = 0)
df.sort_values(by=['label'])


#Get the number of objects
objects=df.drop_duplicates(subset=['label'])
numFeatures = len(objects)

#Get the number of perspectives
numSettings = objSettings.value_counts(subset= ['label'])

#Create array for camera diameter multiplier for different camera views
dMultiplier = np.array([1,0.5],dtype='float32')

#Create output dataframe
outputDF = pd.DataFrame(0, index=np.arange(len(objSettings)*numMeshSamples*len(dMultiplier)),columns=['label','loc'])

numRun = 0
```

Figure 9-3:TargetedScan.py preamble

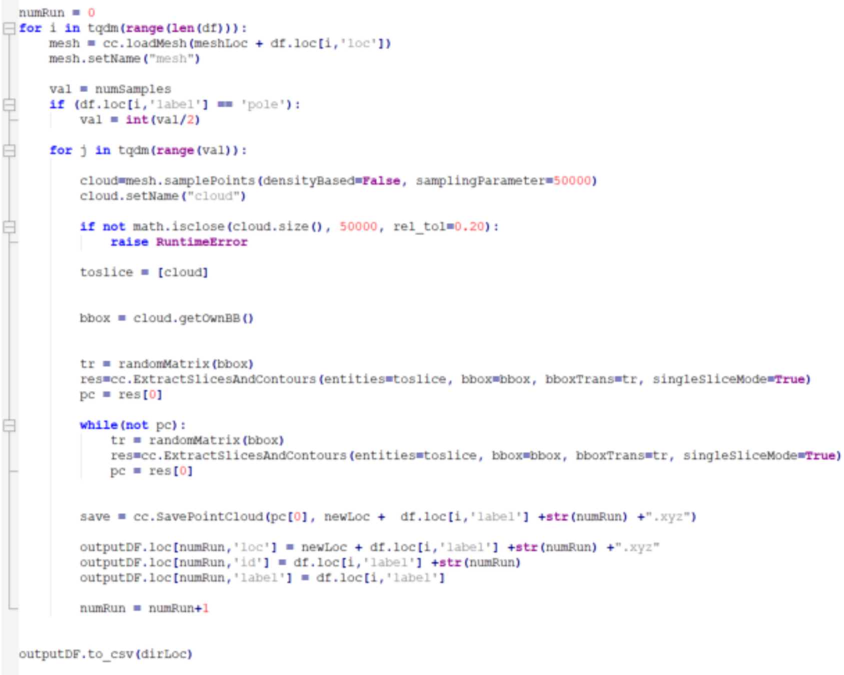
```
#For each object in list
for i in tqdm(range(numFeatures)):

    #Read in mesh to Open3D
    mesh = o3d.io.read_triangle_mesh(fileLoc2 +meshLoc+ df.loc[i,'mesh'])
    mesh.compute_vertex_normals()

    #Get the number of camera views for this object
    numObjSet = int(numSettings.loc[objects.loc[i,'label']].values)

    #For each mesh to be generated
    for j in tqdm(range(numMeshSamples)):

        #Sample 50,000 point on the mesh
        pcd=mesh.sample_points_poisson_disk(50000)

        #Get the camera distance from the range of points in the pointcloud
        distance = np.linalg.norm(
            np.asarray(pcd.get_max_bound()) - np.asarray(pcd.get_min_bound()))


        #Create a temporary object that contains only the camera views for this object
        tempObj = objSettings[objSettings.label == objects.loc[i,'label']].reset_index()
        #print(tempObj)

        #For each camera view
        for k in range(numObjSet):

            #Create a temprary pointcloud to not overwrite the original
            tempPC = o3d.geometry.PointCloud(pcd)


            T = np.eye(4)

            #Extract the camera view parameters as rotations and translations
            rotX = tempObj.loc[k,'RX']
            rotY = tempObj.loc[k,'RY']
            rotZ = tempObj.loc[k,'RZ']
            dX = tempObj.loc[k,'X']
            dY = tempObj.loc[k,'Y']
            dZ = tempObj.loc[k,'Z']

            #Create a transformation matrix based off of view paramters
            T[:3, :3] = tempPC.get_rotation_matrix_from_xyz((rotX, rotY, rotZ))
            T[0, 3] = dX
            T[1, 3] = dY
            T[2, 3] = dZ

            #Transform the pointcloud
            pcd_t = tempPC.transform(T)


            #Two views will be taken for each targeted transformation: A zoomed out global and a zoomed in local perspective.
            for numZoom in range(2):

                #Camera diameter and 'radius' for the current level of zoom
                camD = distance
                radius = distance*dMultiplier[numZoom]

                camera = [0, 0, camD]

                #Use the hidden_point_removal() function to remove all points not in the cameras view
                _, pt_map = pcd_t.hidden_point_removal(camera, radius)

                pcdNew = pcd_t.select_by_index(pt_map)

                #cloud, ind = pcdNew.remove_radius_outlier(nb_points=16,radius=6)

                #o3d.visualization.draw_geometries([cloud])


                o3d.io.write_point_cloud(fileLoc2 + newLoc + df.loc[i,'label'] + str(j) + str(k) + str(numZoom)+".xyz", cloud)

                outputDF.loc[numRun,'label'] = objects.loc[i,'label']
                outputDF.loc[numRun,'loc'] = fileLoc2 + newLoc + df.loc[i,'label'] + str(j) + str(k) + str(numZoom) +".xyz"


                numRun = numRun+1

outputDF.to_csv(fileLoc1+"pointcloudDirectoryTargeted.csv")
```

Figure 9-4:TargetedScan.py main loop



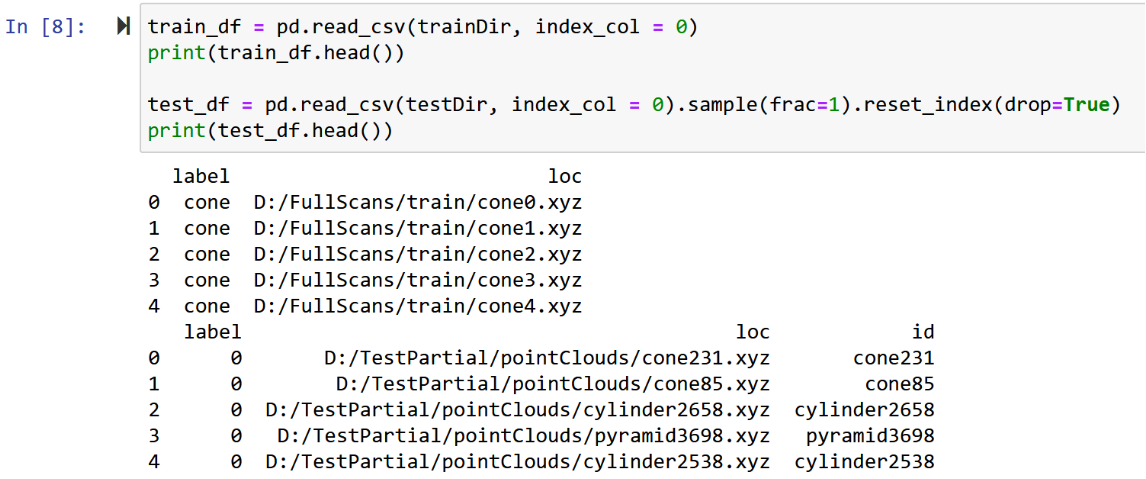
```python
numSamples = 2000

#Read in mesh location df
df = pd.read_csv(meshLoc + "inputDF.txt", header = 0, usecols =['label','loc'])
df = df.sort_values(by=['label'])

outputDF = pd.DataFrame(0, index=np.arange(len(df)*numSamples),columns=['label','loc','id'])
#print(outputDF)


def direction():
    return 1 if random.random() < 0.5 else -1


def randomMatrix(bbox):
    randRoll = math.radians(random.uniform(0,180))
    randPitch = math.radians(random.uniform(0,180))
    randYaw = math.radians(random.uniform(0,180))


    maxArr = bbox.maxCorner()
    minArr = bbox.minCorner()
    lengthX = maxArr[0] - minArr[0]
    lengthY = maxArr[1] - minArr[1]
    lengthZ = maxArr[2] - minArr[2]


    randX = lengthX * random.uniform(0.1,0.9) * direction()

    randY = lengthY * random.uniform(0.1,0.9)*direction()
    randZ = lengthZ * random.uniform(0.1,0.9) * direction()


    tr = cc.ccGLMatrix()

    tr.initFromParameters(randYaw,randPitch,randRoll,(randX,randY,randZ))

    return tr
```

Figure 9-5: scanPartial.py preamble



```python
numRun = 0
for i in tqdm(range(len(df))):
    mesh = cc.loadMesh(meshLoc + df.loc[i,'loc'])
    mesh.setName("mesh")

    val = numSamples
    if (df.loc[i,'label'] == 'pole'):
        val = int(val/2)

    for j in tqdm(range(val)):

        cloud=mesh.samplePoints(densityBased=False, samplingParameter=50000)
        cloud.setName("cloud")

        if not math.isclose(cloud.size(), 50000, rel_tol=0.20):
            raise RuntimeError

        toslice = [cloud]


        bbox = cloud.getOwnBB()


        tr = randomMatrix(bbox)
        res=cc.ExtractSlicesAndContours(entities=toslice, bbox=bbox, bboxTrans=tr, singleSliceMode=True)
        pc = res[0]

        while(not pc):
            tr = randomMatrix(bbox)
            res=cc.ExtractSlicesAndContours(entities=toslice, bbox=bbox, bboxTrans=tr, singleSliceMode=True)
            pc = res[0]


        save = cc.SavePointCloud(pc[0], newLoc +  df.loc[i,'label'] +str(numRun) +".xyz")

        outputDF.loc[numRun,'loc'] = newLoc + df.loc[i,'label'] +str(numRun) +".xyz"
        outputDF.loc[numRun,'id'] = df.loc[i,'label'] +str(numRun)
        outputDF.loc[numRun,'label'] = df.loc[i,'label']

        numRun = numRun+1


outputDF.to_csv(dirLoc)
```

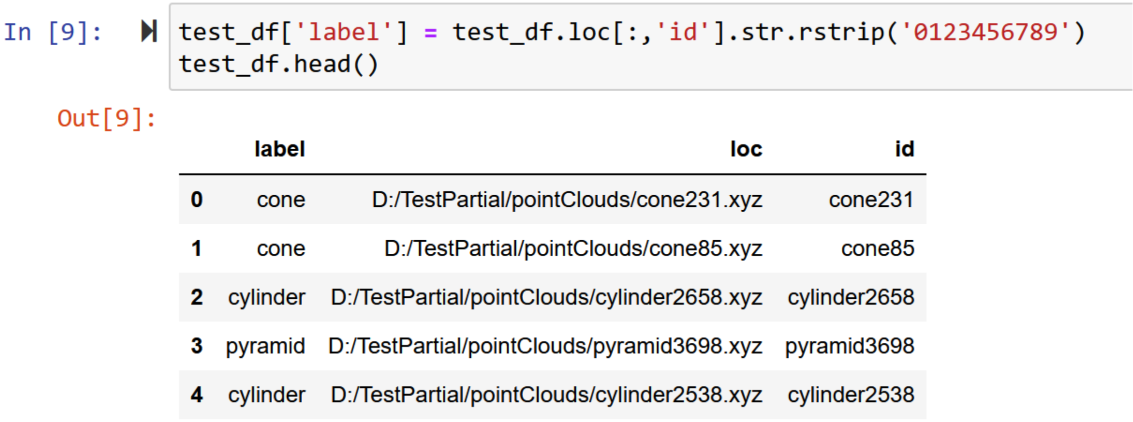
Figure 9-6: scanPartial.py main loop



```
In [8]:   train_df = pd.read_csv(trainDir, index_col = 0)
          print(train_df.head())

          test_df = pd.read_csv(testDir, index_col = 0).sample(frac=1).reset_index(drop=True)
          print(test_df.head())
```

```
  label                           loc
0  cone  D:/FullScans/train/cone0.xyz
1  cone  D:/FullScans/train/cone1.xyz
2  cone  D:/FullScans/train/cone2.xyz
3  cone  D:/FullScans/train/cone3.xyz
4  cone  D:/FullScans/train/cone4.xyz
   label                                       loc            id
0      0       D:/TestPartial/pointClouds/cone231.xyz       cone231
1      0        D:/TestPartial/pointClouds/cone85.xyz        cone85
2      0  D:/TestPartial/pointClouds/cylinder2658.xyz  cylinder2658
3      0   D:/TestPartial/pointClouds/pyramid3698.xyz   pyramid3698
4      0  D:/TestPartial/pointClouds/cylinder2538.xyz  cylinder2538
```

Figure 9-7: Optimal Training Data Model data input

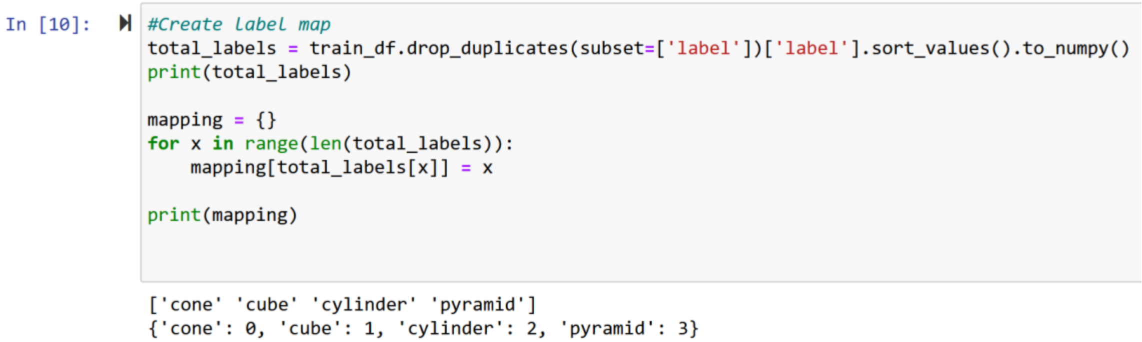


```
In [9]:   test_df['label'] = test_df.loc[:,'id'].str.rstrip('0123456789')
          test_df.head()
```

Out[9]:

	label	loc	id
0	cone	D:/TestPartial/pointClouds/cone231.xyz	cone231
1	cone	D:/TestPartial/pointClouds/cone85.xyz	cone85
2	cylinder	D:/TestPartial/pointClouds/cylinder2658.xyz	cylinder2658
3	pyramid	D:/TestPartial/pointClouds/pyramid3698.xyz	pyramid3698
4	cylinder	D:/TestPartial/pointClouds/cylinder2538.xyz	cylinder2538

Figure 9-8: Labeling point clouds



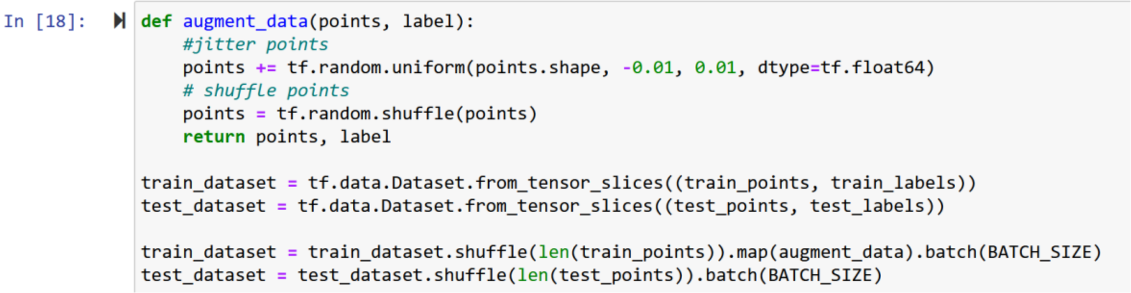
```
In [10]:  #Create label map
          total_labels = train_df.drop_duplicates(subset=['label'])['label'].sort_values().to_numpy()
          print(total_labels)

          mapping = {}
          for x in range(len(total_labels)):
              mapping[total_labels[x]] = x

          print(mapping)
```

```
['cone' 'cube' 'cylinder' 'pyramid']
{'cone': 0, 'cube': 1, 'cylinder': 2, 'pyramid': 3}
```

Figure 9-9: Mapping labels to values

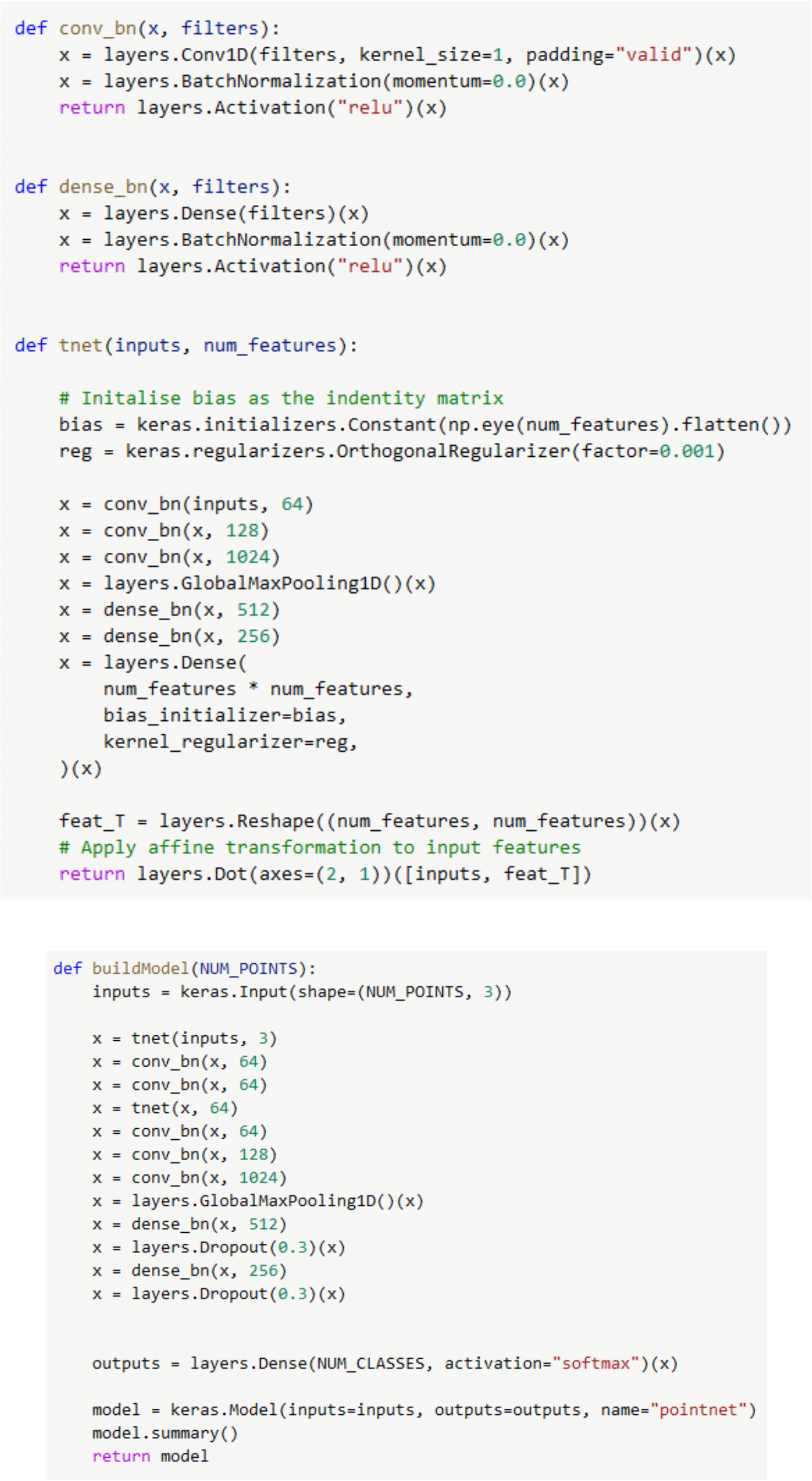
```
In [11]:  def dataset(num_points):
              train_points = []
              train_labels = []
              test_points = []
              test_labels = []


              #Training
              #For each point cloud in the training directory
              for i in tqdm(range(len(train_df))):
                  #Read in the pointcloud from the directory into a dataframe
                  df = pd.read_csv(train_df.loc[i,'loc'], header = None, sep = ' ', usecols=[0,1,2])

                  #Sample the number of specifed points
                  dfSampled = df.sample(n=num_points, replace=True)
                  array = dfSampled.to_numpy()
                  train_points.append(array)

                  #Append the appropriate label
                  label = train_df.loc[i,'label']
                  train_labels.append(mapping[label])



              #Testing
              #For each point cloud in the testing directory
              for i in tqdm(range(len(test_df))):
                  #Read in the pointcloud from the directory into a dataframe
                  df = pd.read_csv(test_df.loc[i,'loc'], header = None, sep = ' ', usecols=[0,1,2])

                  #Sample the number of specifed points
                  dfSampled = df.sample(n=num_points, replace=True)
                  array = dfSampled.to_numpy()
                  test_points.append(array)

                  #Append the appropriate label
                  label = test_df.loc[i,'label']
                  test_labels.append(mapping[label])

              return (
                  np.array(train_points),
                  np.array(train_labels),
                  np.array(test_points),
                  np.array(test_labels)
                     )
```

Figure 9-10: Dataset generation



```
In [18]:  def augment_data(points, label):
              #jitter points
              points += tf.random.uniform(points.shape, -0.01, 0.01, dtype=tf.float64)
              # shuffle points
              points = tf.random.shuffle(points)
              return points, label

          train_dataset = tf.data.Dataset.from_tensor_slices((train_points, train_labels))
          test_dataset = tf.data.Dataset.from_tensor_slices((test_points, test_labels))

          train_dataset = train_dataset.shuffle(len(train_points)).map(augment_data).batch(BATCH_SIZE)
          test_dataset = test_dataset.shuffle(len(test_points)).batch(BATCH_SIZE)
```

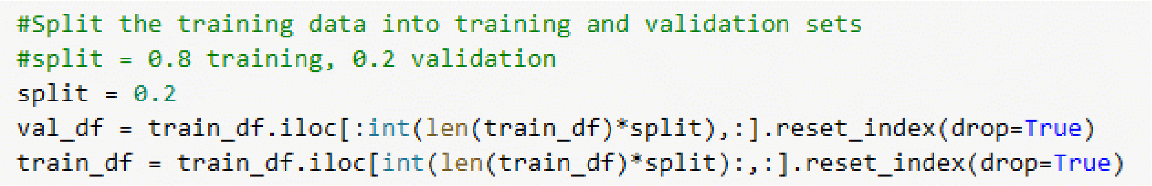
Figure 9-11: Data augmentation



```python
def conv_bn(x, filters):
    x = layers.Conv1D(filters, kernel_size=1, padding="valid")(x)
    x = layers.BatchNormalization(momentum=0.0)(x)
    return layers.Activation("relu")(x)


def dense_bn(x, filters):
    x = layers.Dense(filters)(x)
    x = layers.BatchNormalization(momentum=0.0)(x)
    return layers.Activation("relu")(x)


def tnet(inputs, num_features):

    # Initalise bias as the indentity matrix
    bias = keras.initializers.Constant(np.eye(num_features).flatten())
    reg = keras.regularizers.OrthogonalRegularizer(factor=0.001)

    x = conv_bn(inputs, 64)
    x = conv_bn(x, 128)
    x = conv_bn(x, 1024)
    x = layers.GlobalMaxPooling1D()(x)
    x = dense_bn(x, 512)
    x = dense_bn(x, 256)
    x = layers.Dense(
        num_features * num_features,
        bias_initializer=bias,
        kernel_regularizer=reg,
    )(x)

    feat_T = layers.Reshape((num_features, num_features))(x)
    # Apply affine transformation to input features
    return layers.Dot(axes=(2, 1))([inputs, feat_T])
```

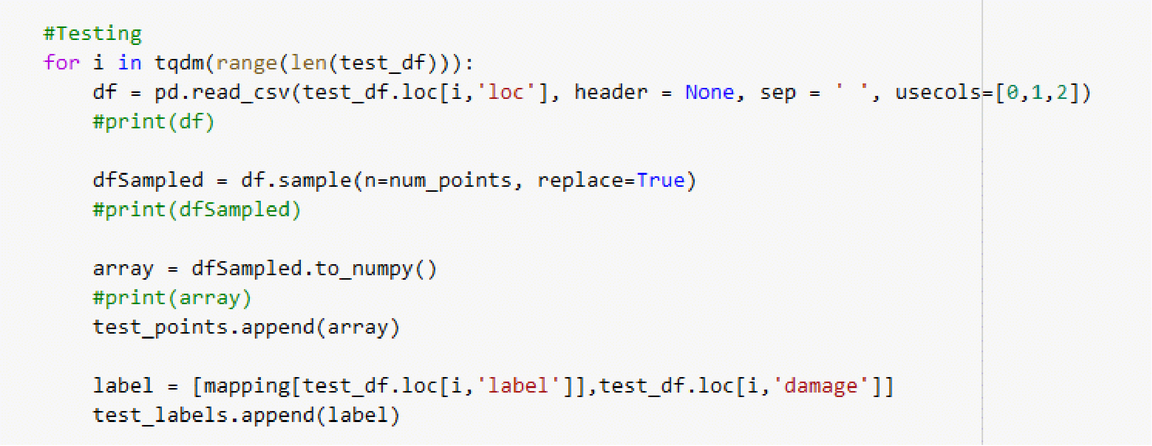
```python
def buildModel(NUM_POINTS):
    inputs = keras.Input(shape=(NUM_POINTS, 3))

    x = tnet(inputs, 3)
    x = conv_bn(x, 64)
    x = conv_bn(x, 64)
    x = tnet(x, 64)
    x = conv_bn(x, 64)
    x = conv_bn(x, 128)
    x = conv_bn(x, 1024)
    x = layers.GlobalMaxPooling1D()(x)
    x = dense_bn(x, 512)
    x = layers.Dropout(0.3)(x)
    x = dense_bn(x, 256)
    x = layers.Dropout(0.3)(x)


    outputs = layers.Dense(NUM_CLASSES, activation="softmax")(x)

    model = keras.Model(inputs=inputs, outputs=outputs, name="pointnet")
    model.summary()
    return model
```

Figure 9-12: Keras implementation of PointNet algorithm



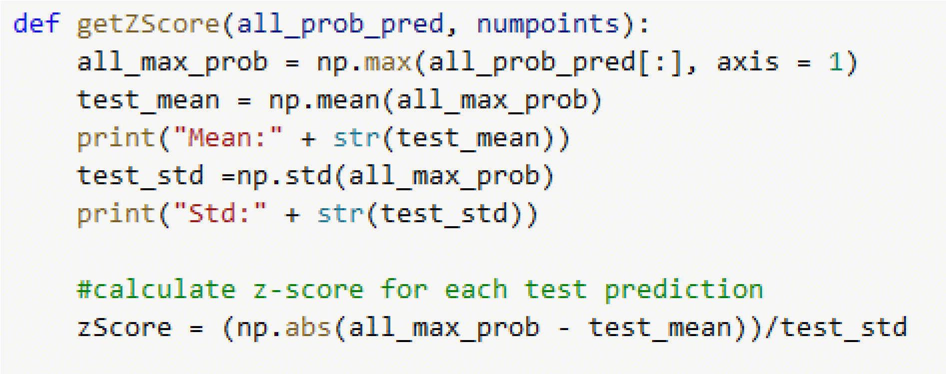
```
#Split the training data into training and validation sets
#split = 0.8 training, 0.2 validation
split = 0.2
val_df = train_df.iloc[:int(len(train_df)*split),:].reset_index(drop=True)
train_df = train_df.iloc[int(len(train_df)*split):,:].reset_index(drop=True)
```

Figure 9-13: Splitting the training and validation data



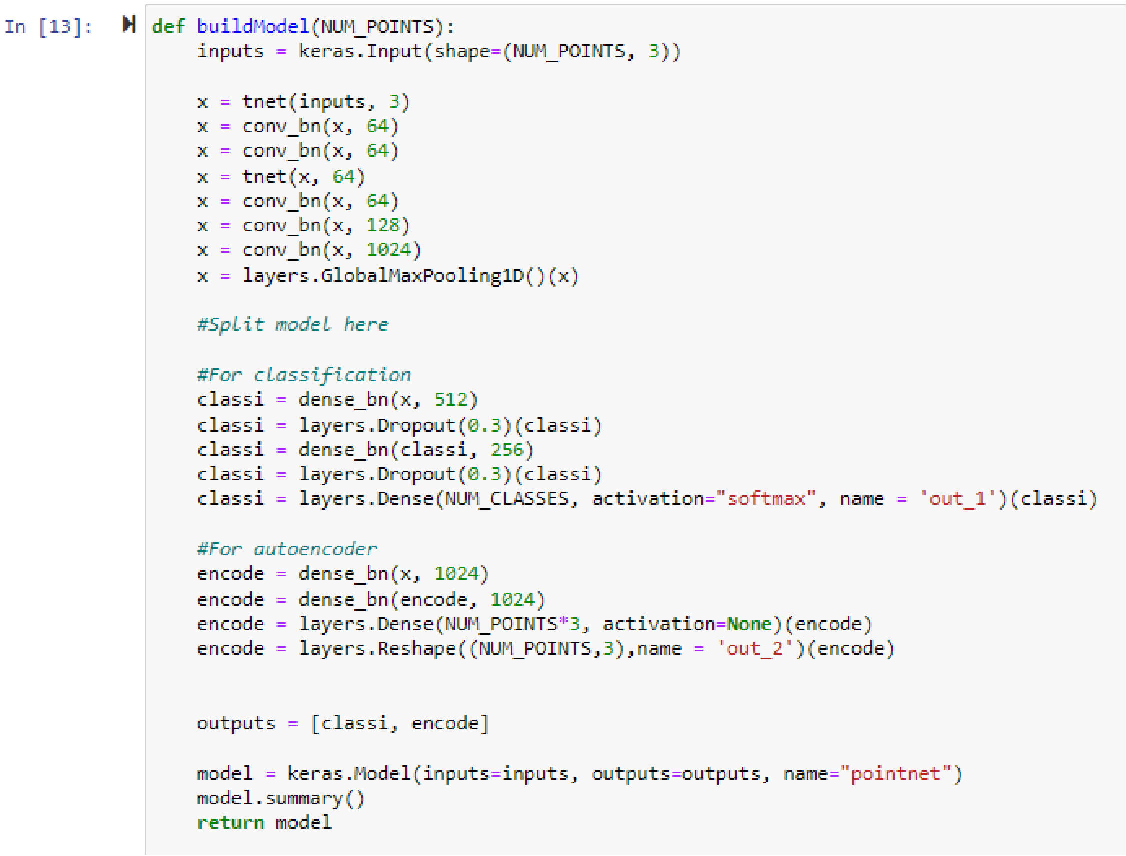
```
#Testing
for i in tqdm(range(len(test_df))):
    df = pd.read_csv(test_df.loc[i,'loc'], header = None, sep = ' ', usecols=[0,1,2])
    #print(df)

    dfSampled = df.sample(n=num_points, replace=True)
    #print(dfSampled)

    array = dfSampled.to_numpy()
    #print(array)
    test_points.append(array)

    label = [mapping[test_df.loc[i,'label']],test_df.loc[i,'damage']]
    test_labels.append(label)
```

Figure 9-14: Adding damage label as a testing dataset target



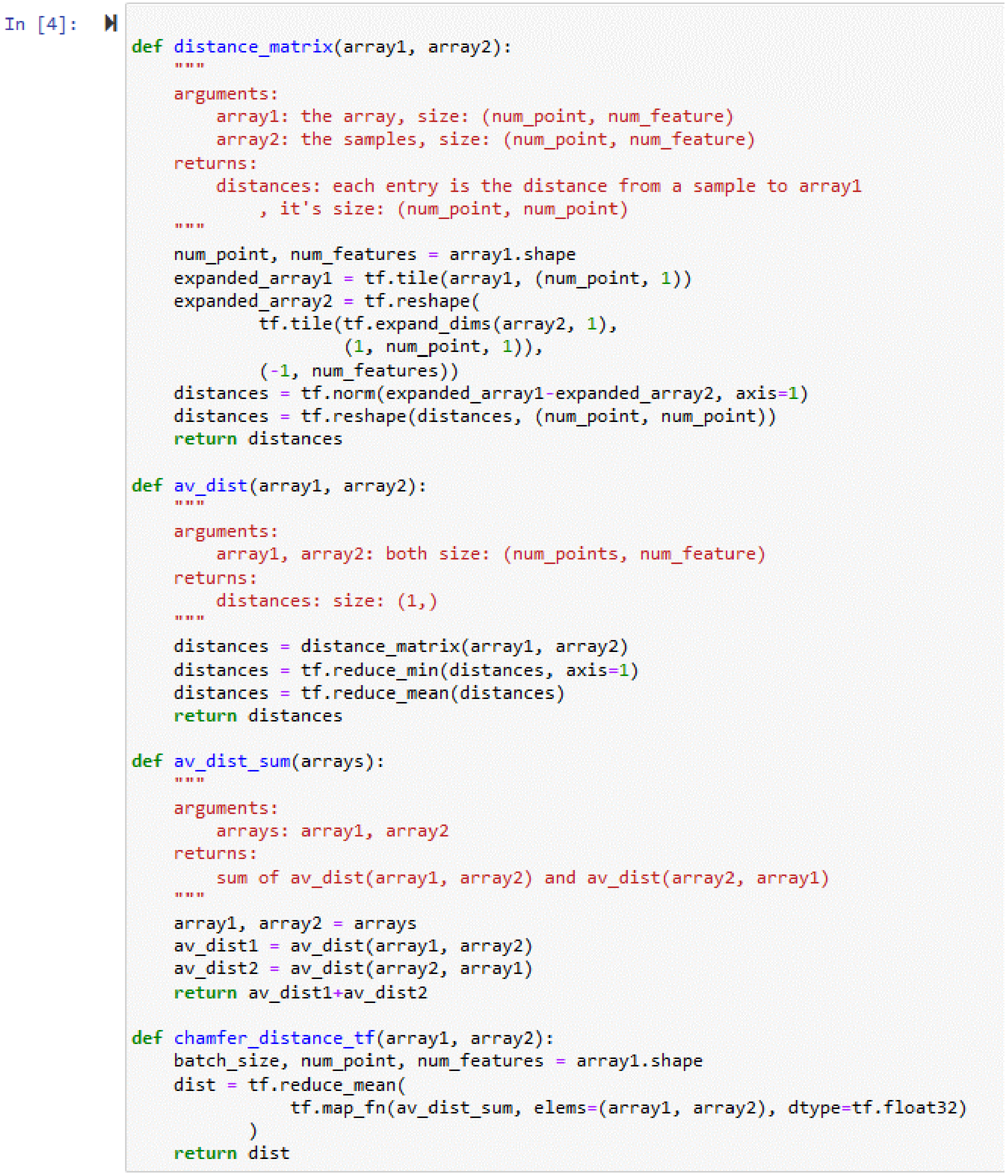
```
def getZScore(all_prob_pred, numpoints):
    all_max_prob = np.max(all_prob_pred[:], axis = 1)
    test_mean = np.mean(all_max_prob)
    print("Mean:" + str(test_mean))
    test_std =np.std(all_max_prob)
    print("Std:" + str(test_std))

    #calculate z-score for each test prediction
    zScore = (np.abs(all_max_prob - test_mean))/test_std
```

Figure 9-15: Z-Score calculation



```
def buildModel(NUM_POINTS):
    inputs = keras.Input(shape=(NUM_POINTS, 3))

    x = tnet(inputs, 3)
    x = conv_bn(x, 64)
    x = conv_bn(x, 64)
    x = tnet(x, 64)
    x = conv_bn(x, 64)
    x = conv_bn(x, 128)
    x = conv_bn(x, 1024)
    x = layers.GlobalMaxPooling1D()(x)

    #Split model here

    #For classification
    classi = dense_bn(x, 512)
    classi = layers.Dropout(0.3)(classi)
    classi = dense_bn(classi, 256)
    classi = layers.Dropout(0.3)(classi)
    classi = layers.Dense(NUM_CLASSES, activation="softmax", name = 'out_1')(classi)

    #For autoencoder
    encode = dense_bn(x, 1024)
    encode = dense_bn(encode, 1024)
    encode = layers.Dense(NUM_POINTS*3, activation=None)(encode)
    encode = layers.Reshape((NUM_POINTS,3),name = 'out_2')(encode)


    outputs = [classi, encode]

    model = keras.Model(inputs=inputs, outputs=outputs, name="pointnet")
    model.summary()
    return model
```

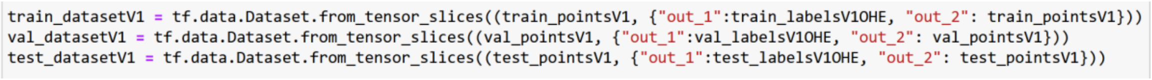
Figure 9-16: Keras implementation of multi-output architecture



In [4]:

```
def distance_matrix(array1, array2):
    """
    arguments:
        array1: the array, size: (num_point, num_feature)
        array2: the samples, size: (num_point, num_feature)
    returns:
        distances: each entry is the distance from a sample to array1
            , it's size: (num_point, num_point)
    """
    num_point, num_features = array1.shape
    expanded_array1 = tf.tile(array1, (num_point, 1))
    expanded_array2 = tf.reshape(
            tf.tile(tf.expand_dims(array2, 1),
                    (1, num_point, 1)),
            (-1, num_features))
    distances = tf.norm(expanded_array1-expanded_array2, axis=1)
    distances = tf.reshape(distances, (num_point, num_point))
    return distances

def av_dist(array1, array2):
    """
    arguments:
        array1, array2: both size: (num_points, num_feature)
    returns:
        distances: size: (1,)
    """
    distances = distance_matrix(array1, array2)
    distances = tf.reduce_min(distances, axis=1)
    distances = tf.reduce_mean(distances)
    return distances

def av_dist_sum(arrays):
    """
    arguments:
        arrays: array1, array2
    returns:
        sum of av_dist(array1, array2) and av_dist(array2, array1)
    """
    array1, array2 = arrays
    av_dist1 = av_dist(array1, array2)
    av_dist2 = av_dist(array2, array1)
    return av_dist1+av_dist2

def chamfer_distance_tf(array1, array2):
    batch_size, num_point, num_features = array1.shape
    dist = tf.reduce_mean(
                tf.map_fn(av_dist_sum, elems=(array1, array2), dtype=tf.float32)
            )
    return dist
```

Figure 9-17: CD TensorFlow calculation method [53]



```
train_datasetV1 = tf.data.Dataset.from_tensor_slices((train_pointsV1, {"out_1":train_labelsV1OHE, "out_2": train_pointsV1}))
val_datasetV1 = tf.data.Dataset.from_tensor_slices((val_pointsV1, {"out_1":val_labelsV1OHE, "out_2": val_pointsV1}))
test_datasetV1 = tf.data.Dataset.from_tensor_slices((test_pointsV1, {"out_1":test_labelsV1OHE, "out_2": test_pointsV1}))
```

Figure 9-18: Dataset generation with input point clouds being set as the target data for the autoencoder

```
class MyHyperModel(keras_tuner.HyperModel):

    def build(self, hp):
        inputs = keras.Input(shape=(NUM_POINTS, 3))

        x = tnet(inputs, 3, hp)
        filt = hp.Int("units_base", min_value=32, max_value=256, step=32)
        x = conv_bn(x,filt, "layer1",hp)
        x = conv_bn(x,filt, "layer2",hp)
        x = tnet(x, filt, hp)
        x = conv_bn(x,filt,"layer3",hp)
        x = conv_bn(x,(filt*2), "layer4",hp)
        x = conv_bn(x,(filt*16), "layer5",hp)
        x = layers.GlobalMaxPooling1D()(x)

        #Split model here

        #For classification
        classi = dense_bn(x,(filt*8), "classi1",hp)
        classi = layers.Dropout(0.3)(classi)
        classi = dense_bn(classi,(filt*4), "classi2",hp)
        classi = layers.Dropout(0.3)(classi)
        classi = layers.Dense(NUM_CLASSES, activation="softmax", name = 'out_1')(classi)

        #For autoencoder
        encode = dense_bn(x, (filt*16), "enc1", hp)
        encode= dense_bn(encode, (filt*16),"enc2", hp)
        encode = layers.Dense(NUM_POINTS*3, activation=None)(encode)
        encode = layers.Reshape((NUM_POINTS,3),name = 'out_2')(encode)


        outputs = [classi, encode]

        model = keras.Model(inputs=inputs, outputs=outputs, name="pointnet")
        model.summary()

        model.compile(
        loss={"out_1":"categorical_crossentropy",
              "out_2": chamfer_distance_tf},
        optimizer=keras.optimizers.Adam(learning_rate=hp.Float("learning_rate",
                                                               min_value=0.001,
                                                               max_value=1,
                                                               step=10,
                                                               sampling="log")),
        metrics=['Accuracy'],
    )
        return model
```

```
#Hyperparamters: num filters, momentum and activation function
def conv_bn(x,filt,layerName,hp):
    x = layers.Conv1D(filters=filt,
                      kernel_size=1, padding="valid")(x)
    x = layers.BatchNormalization(momentum=hp.Float(f"momentum_{layerName}",min_value=0.0,max_value=0.8,step=0.1,))(x)
    return layers.Activation(activation=hp.Choice(f"activation_{layerName}", ["relu", "tanh"]))(x)

#Hyperparamters: num filters, momentum and activation function
def dense_bn(x,unit, layerName,hp):
    x = layers.Dense(units= unit)(x)
    x = layers.BatchNormalization(momentum=hp.Float(f"momentum_{layerName}",min_value=0.0,max_value=0.8,step=0.1,))(x)
    return layers.Activation(activation=hp.Choice(f"activation_{layerName}", ["relu", "tanh"]))(x)


def tnet(inputs, num_features,hp):

    # Initalise bias as the indentity matrix
    bias = keras.initializers.Constant(np.eye(num_features).flatten())
    reg = keras.regularizers.OrthogonalRegularizer(factor=0.001)

    x = conv_bn(inputs,num_features, "tnet1:"+ str(num_features),hp)
    x = conv_bn(x,(num_features*2), "tnet2:"+ str(num_features),hp)
    x = conv_bn(x,(num_features*16), "tnet3:"+ str(num_features),hp)
    x = layers.GlobalMaxPooling1D()(x)
    x = dense_bn(x, (num_features*8),"tnet4:"+ str(num_features),hp)
    x = dense_bn(x, (num_features*4), "tnet5:"+ str(num_features),hp)
    x = layers.Dense(
        num_features * num_features,
        bias_initializer=bias,
        kernel_regularizer=reg,
    )(x)

    feat_T = layers.Reshape((num_features, num_features))(x)
    # Apply affine transformation to input features
    return layers.Dot(axes=(2, 1))([inputs, feat_T])
```

Figure 9-19: Multi-output model hyperparameter tuning implementation

Appendix B: Hyperparameter Summary

momentum_tnet1:3: 0.1

activation_tnet1:3: relu

momentum_tnet2:3: 0.0

activation_tnet2:3: relu

momentum_tnet3:3: 0.2

activation_tnet3:3: tanh

momentum_tnet4:3: 0.8

activation_tnet4:3: relu

momentum_tnet5:3: 0.5

activation_tnet5:3: relu

units_base: 32

momentum_layer1: 0.2

activation_layer1: tanh

momentum_layer2: 0.30000000000000004

activation_layer2: tanh

momentum_tnet1:32: 0.5

activation_tnet1:32: relu

momentum_tnet2:32: 0.8

activation_tnet2:32: relu

momentum_tnet3:32: 0.5

activation_tnet3:32: tanh

momentum_tnet4:32: 0.0

activation_tnet4:32: tanh

momentum_tnet5:32: 0.30000000000000004

activation_tnet5:32: relu

momentum_layer3: 0.0

activation_layer3: relu

momentum_layer4: 0.4

activation_layer4: relu

momentum_layer5: 0.2

activation_layer5: tanh

momentum_classi1: 0.0

activation_classi1: tanh

momentum_classi2: 0.5

activation_classi2: relu

momentum_enc1: 0.2

activation_enc1: relu

momentum_enc2: 0.0

activation_enc2: tanh

learning_rate: 0.01

momentum_tnet1:64: 0.6000000000000001

activation_tnet1:64: tanh

momentum_tnet2:64: 0.6000000000000001

activation_tnet2:64: tanh

momentum_tnet3:64: 0.6000000000000001

activation_tnet3:64: tanh

momentum_tnet4:64: 0.1

activation_tnet4:64: relu

momentum_tnet5:64: 0.6000000000000001

activation_tnet5:64: tanh

momentum_tnet1:128: 0.0

activation_tnet1:128: relu

momentum_tnet2:128: 0.0

activation_tnet2:128: relu

momentum_tnet3:128: 0.6000000000000001

activation_tnet3:128: relu

momentum_tnet4:128: 0.1

activation_tnet4:128: relu

momentum_tnet5:128: 0.30000000000000004

activation_tnet5:128: relu

momentum_tnet1:224: 0.0

activation_tnet1:224: tanh

momentum_tnet2:224: 0.8

activation_tnet2:224: relu

momentum_tnet3:224: 0.7000000000000001

activation_tnet3:224: tanh

momentum_tnet4:224: 0.30000000000000004

activation_tnet4:224: relu

momentum_tnet5:224: 0.5

activation_tnet5:224: relu

momentum_tnet1:160: 0.4

activation_tnet1:160: tanh

momentum_tnet2:160: 0.2

activation_tnet2:160: tanh

momentum_tnet3:160: 0.0

activation_tnet3:160: relu

momentum_tnet4:160: 0.1

activation_tnet4:160: tanh

momentum_tnet5:160: 0.2

activation_tnet5:160: relu

Score: 27.590384483337402

Appendix C: Model Summary

```
__________________________________________________________________________________________
___
 Layer (type)                Output Shape         Param #     Connected to
==========================================================================================
==============
 input_2 (InputLayer)          [(None, 4096, 3)]    0           []

 conv1d_11 (Conv1D)            (None, 4096, 3)      12          ['input_2[0][0]']

 batch_normalization_19 (BatchN  (None, 4096, 3)     12          ['conv1d_11[0][0]']
 ormalization)

 activation_19 (Activation)     (None, 4096, 3)      0           ['batch_normalization_19[0][0]']

 conv1d_12 (Conv1D)            (None, 4096, 6)      24          ['activation_19[0][0]']

 batch_normalization_20 (BatchN  (None, 4096, 6)     24          ['conv1d_12[0][0]']
 ormalization)

 activation_20 (Activation)     (None, 4096, 6)      0           ['batch_normalization_20[0][0]']

 conv1d_13 (Conv1D)            (None, 4096, 48)     336         ['activation_20[0][0]']

 batch_normalization_21 (BatchN  (None, 4096, 48)    192         ['conv1d_13[0][0]']
 ormalization)

 activation_21 (Activation)     (None, 4096, 48)     0           ['batch_normalization_21[0][0]']

 global_max_pooling1d_3 (Global  (None, 48)          0           ['activation_21[0][0]']
 MaxPooling1D)

 dense_11 (Dense)              (None, 24)           1176        ['global_max_pooling1d_3[0][0]']

 batch_normalization_22 (BatchN  (None, 24)          96          ['dense_11[0][0]']
 ormalization)

 activation_22 (Activation)     (None, 24)           0           ['batch_normalization_22[0][0]']

 dense_12 (Dense)              (None, 12)           300         ['activation_22[0][0]']

 batch_normalization_23 (BatchN  (None, 12)          48          ['dense_12[0][0]']
 ormalization)

activation_23 (Activation)      (None, 12)           0           ['batch_normalization_23[0][0]']

 dense_13 (Dense)              (None, 9)            117         ['activation_23[0][0]']

 reshape_2 (Reshape)           (None, 3, 3)         0           ['dense_13[0][0]']

 dot_2 (Dot)                   (None, 4096, 3)      0           ['input_2[0][0]',
                                                                 'reshape_2[0][0]']
```

```
conv1d_14 (Conv1D)            (None, 4096, 32)     128         ['dot_2[0][0]']

 batch_normalization_24 (BatchN  (None, 4096, 32)    128         ['conv1d_14[0][0]']
 ormalization)

 activation_24 (Activation)     (None, 4096, 32)     0          ['batch_normalization_24[0][0]']

 conv1d_15 (Conv1D)            (None, 4096, 32)     1056        ['activation_24[0][0]']

batch_normalization_25 (BatchN  (None, 4096, 32)    128         ['conv1d_15[0][0]']
 ormalization)

 activation_25 (Activation)     (None, 4096, 32)     0          ['batch_normalization_25[0][0]']

 conv1d_16 (Conv1D)            (None, 4096, 32)     1056        ['activation_25[0][0]']

 batch_normalization_26 (BatchN  (None, 4096, 32)    128         ['conv1d_16[0][0]']
 ormalization)

 activation_26 (Activation)     (None, 4096, 32)     0          ['batch_normalization_26[0][0]']

 conv1d_17 (Conv1D)            (None, 4096, 64)     2112        ['activation_26[0][0]']

 batch_normalization_27 (BatchN  (None, 4096, 64)    256         ['conv1d_17[0][0]']
 ormalization)

 activation_27 (Activation)     (None, 4096, 64)     0          ['batch_normalization_27[0][0]']

 conv1d_18 (Conv1D)            (None, 4096, 512)    33280       ['activation_27[0][0]']

 batch_normalization_28 (BatchN  (None, 4096, 512)   2048        ['conv1d_18[0][0]']
 ormalization)

 activation_28 (Activation)     (None, 4096, 512)    0          ['batch_normalization_28[0][0]']

 global_max_pooling1d_4 (Global  (None, 512)         0          ['activation_28[0][0]']
 MaxPooling1D)

 dense_14 (Dense)              (None, 256)          131328      ['global_max_pooling1d_4[0][0]']

 batch_normalization_29 (BatchN  (None, 256)         1024        ['dense_14[0][0]']
 ormalization)

 activation_29 (Activation)     (None, 256)          0          ['batch_normalization_29[0][0]']

 dense_15 (Dense)              (None, 128)          32896       ['activation_29[0][0]']

 batch_normalization_30 (BatchN  (None, 128)         512         ['dense_15[0][0]']
 ormalization)

 activation_30 (Activation)     (None, 128)          0          ['batch_normalization_30[0][0]']

 dense_16 (Dense)              (None, 1024)         132096      ['activation_30[0][0]']

 reshape_3 (Reshape)           (None, 32, 32)       0          ['dense_16[0][0]']

 dot_3 (Dot)                   (None, 4096, 32)     0          ['activation_25[0][0]',
                                                                 'reshape_3[0][0]']

 conv1d_19 (Conv1D)            (None, 4096, 32)     1056        ['dot_3[0][0]']

 batch_normalization_31 (BatchN  (None, 4096, 32)    128         ['conv1d_19[0][0]']
 ormalization)

 activation_31 (Activation)     (None, 4096, 32)     0          ['batch_normalization_31[0][0]']

 conv1d_20 (Conv1D)            (None, 4096, 64)     2112        ['activation_31[0][0]']

 batch_normalization_32 (BatchN  (None, 4096, 64)    256         ['conv1d_20[0][0]']
 ormalization)
```

```
activation_32 (Activation)     (None, 4096, 64)     0           ['batch_normalization_32[0][0]']

 conv1d_21 (Conv1D)            (None, 4096, 512)    33280       ['activation_32[0][0]']

 batch_normalization_33 (BatchN  (None, 4096, 512)   2048        ['conv1d_21[0][0]']
 ormalization)

 activation_33 (Activation)     (None, 4096, 512)    0           ['batch_normalization_33[0][0]']

 global_max_pooling1d_5 (Global  (None, 512)         0           ['activation_33[0][0]']
 MaxPooling1D)

 dense_17 (Dense)              (None, 256)          131328      ['global_max_pooling1d_5[0][0]']

 batch_normalization_34 (BatchN  (None, 256)         1024        ['dense_17[0][0]']
 ormalization)

 dense_19 (Dense)              (None, 512)          262656      ['global_max_pooling1d_5[0][0]']

 activation_34 (Activation)     (None, 256)          0           ['batch_normalization_34[0][0]']

 batch_normalization_36 (BatchN  (None, 512)         2048        ['dense_19[0][0]']
 ormalization)

 dropout_2 (Dropout)           (None, 256)          0           ['activation_34[0][0]']

 activation_36 (Activation)     (None, 512)          0           ['batch_normalization_36[0][0]']

 dense_18 (Dense)              (None, 128)          32896       ['dropout_2[0][0]']

 dense_20 (Dense)              (None, 512)          262656      ['activation_36[0][0]']

 batch_normalization_35 (BatchN  (None, 128)         512         ['dense_18[0][0]']
 ormalization)

 batch_normalization_37 (BatchN  (None, 512)         2048        ['dense_20[0][0]']
 ormalization)

 activation_35 (Activation)     (None, 128)          0           ['batch_normalization_35[0][0]']

 activation_37 (Activation)     (None, 512)          0           ['batch_normalization_37[0][0]']

 dropout_3 (Dropout)           (None, 128)          0           ['activation_35[0][0]']

 dense_21 (Dense)              (None, 12288)        6303744     ['activation_37[0][0]']

 out_1 (Dense)                 (None, 4)            516         ['dropout_3[0][0]']

 out_2 (Reshape)               (None, 4096, 3)      0           ['dense_21[0][0]']

==================================================================================
==================
Total params: 7,378,821
Trainable params: 7,372,491
Non-trainable params: 6,330
__________________________________________________________________________________
________
```

Curriculum Vitae

Name:	Kaitlin Hutton
Post-secondary Education and Degrees:	Western University London, Ontario, Canada 2017 – 2021 B.E.Sc.
	Western University London, Ontario, Canada 2021 – 2023 M.E.Sc.
Related Work Experience:	Teaching Assistant Western University 2021 – 2023
	Research Assistant Western University 2021 – 2023