Department of Informatics
King's College London
United Kingdom

7CCSMPRJ Individual Project

3D Generative Adversarial Modelling for Data Augmentation of Human Motions

---

Name: **Adrian Salazar Gomez**
Student Number: 1870060
Course: Data Science

Supervisor: Dr Brendan Michael

This dissertation is submitted for the degree of MSc in Data Science.

Abstract

Three dimensional data provides meaningful information that other kind of data cannot provide. The complexity of 3D datasets limits the methodologies that can be used to get useful information from 3D data. Deep learning models are able to manage this type of data,in exchange deep learning requires much data to perform well. Data Augmentation uses existing data to create new data with the ability to improve deep learning methodologies. However, traditional data augmentation methodologies are not useful to improve the classification of 3D data. This research shows that Generative Adversarial has the ability to synthesise data that can be used to improve the performance of 3D classifiers in dataset of all sizes.

Keywords: Machine Learning, Computer Vision, Generative Models, GAN, Deep Learning, Action Recognition, 3D Data, Data Augmentation

Nomenclature

$ANNs$ Artificial Neural Networks
$CNNs$ Convolutional Neural Networks
$GANs$ Generative Adversarial Networks
$3D$ Three dimensional
$G()$ Generator
$D()$ Discriminator
$\theta^{(D)}$ Discriminator parameters
$\theta^{(G)}$ generator parameters
ω Parameters in a Artificial Neural Network

Contents

List of Figures

List of Tables

1 Introduction

Understanding three dimensional(3D) information of an object is an important tasks on areas such as computer vision [1], augmented reality [2], virtual reality [3], medicine [4], and robotics [5]. This data provides more information than standard images, particularly, in situations where volume, shape, and motion characteristics play an important role. Although most of the analysis of 3D data require the implementation of machine learning methods, the complexity of 3D representations has limited the application of traditional machine learning. Deep learning methodologies have demonstrated good results in handling 3D data for supervised and unsupervised tasks [6] [7]. However, deep learning models require a large amount of data to perform well and are sensitive to class imbalances. This often proves to be problematic with 3D data as in practical scenarios the amount of data available is limited, particularly, in medicine. Additionally, the collection of 3D data requires special tools such as LiDAR scanner or RGBD cameras.

In computer vision, traditional methods to increase the amount of data available consist on producing small modification of the original data such as image rotation and flipping. These techniques tend to fall short on improving deep learning models performance since the generated variance is minimum. Generative adversarial networks(GANs) are deep learning structures able to learn the distribution of a dataset and synthesise non seen instances with similar characteristics as the original. The data produced by GANs has the potential to introduce enough variation to improve significantly the results of deep learning models, even when data is very limited [8] [9]. With 3D data, traditional augmentation methods do not work well [10] and traditional 3D generation models do not introduce enough variance as are based on mixing part of 3D data to generate new ones [11]. 3D GANs have the ability to learn 3D distribution and synthesise new 3D data. However, to the best of our knowledge, the capacity of 3D GANs to generated data for augmentation purposes has not been tested. This research evaluates the suitability of 3D GANs to augment 3D datasets and improve 3D deep learning methodologies.

To evaluate the capacity of 3D GANs to augment 3D datasets, the research uses a 3D deep learning classifier to identify actions represented in 3D characterisations of humans performing actions. Then analyses the impact of the augmentation process on the classifier. The results confirms the capacity of GANs to improve the performance of 3D deep learning models, even when the data set is limited in size. Additionally, the research evaluates some aspects of the augmentation process that must be considered to maximise the performance of a 3D GANs augmentation process. To our knowledge, this is the first data augmentation strategy suggested for 3D data using 3D GANs.

2 Background

2.1 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a type of machine learning structure proposed in [12] typically used for semi-supervised and supervised task. GANs model the distributions of high dimensional data even when the number of labelled data is scarce. This learnt distributions can be used to synthesise data with similar characteristic as the original data, image processing [13], style transfer [14], data augmentation [8], anomaly detection [15] and classification [16]. Because, the potential of GANs, the literature is continuously proposing new GANs structures that add new functionalities to the original structure. In most of the cases, the experiments made with GANs use images data. However, other types of data such as audio [17], text [18] and graph data [19] have been proposed to its implementation with GANs.

GANs are made of two structures a generator *G()* and a discriminator model *D()*. The generator *G()* generate data that comes from the same distribution as the real data. Whereas the discriminator *D()* differentiates between real data and synthetic data generated by *G()*. The generator *G()* is a differentiable function that uses a set of parameters $\theta^{(G)}$ to synthesise data by mapping a latent space z inferred from a prior distribution to a sample with the same characteristics as a sample from the real data distribution *pmodel*. The generator learns the parameters $\theta^{(G)}$ by feeding synthesised data to the discriminator and learning to fool it. Hence, the generator learns to generate realistic data without any contact with real data x. The typical structure of the generator is a deep artificial neural network model. The discriminator *D()* is a differentiable function that uses a set of parameters $\theta^{(D)}$ to map an input to a probability of the input being from the same probability distribution as the real data. The input of the discriminator consist on a set of synthesised data *G(z)* and real data x. The discriminator learns the parameters $\theta^{(D)}$ with a normal supervised learning approach with the goal to label properly real or fake data. Figure 1 illustrates the standard structure of a GAN.

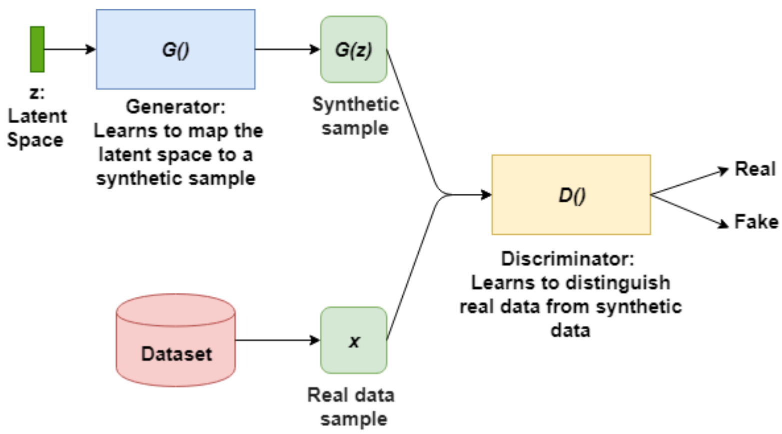


Figure 1: Traditional structure of a Generative Adversarial Network

GANs training procedure is a two-player minimax game where the discriminator tries to maximise the classification performance and the generator tries to minimise the discriminator classification performance. Equation 2.1 is the traditional training objective function in GANs. Whereas equation 2.2 is a modified objective function proposed in [20]. The modified functions produces stronger training signals or gradient to avoid a vanishing gradient situation where $\theta^{(D)}$ and $\theta^{(G)}$ do not change. Section 3.3 contains detailed information about the vanishing gradient problem and the potential solutions suggested in the literature.

$$\min_G \max_D L(D,G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.1}$$

$$L(D,G) = \tag{2.2}$$
$$\max_D [E_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z))] + \max_G \mathbb{E}_{z\sim p_z(z)}[\log D(G(z)))]$$

Where $E_{z\sim p_z(z)}[\log(1 - D(G(z))]$ is the log of the probability of the discriminator of predicting that the generated data is not from the real distribution and $E_{x\sim p_r(x)}[\log D(x)]$ is the log of the probability of the discriminator to classify real data as real. In the original formulation 2.1, the discriminator parameters $\theta^{(D)}$ are trained by maximising $\log D(x)$ whereas the generator parameters $\theta^{(G)}$ are trained by minising $\log(1-D(G(z)))$. In equation 2.2 the generator is trained by minimising $\log(1 - D(G(z))$ and the discriminator is trained by maximising $\log D(G(z))$. Initial GANs structures use stochastic gradient descent to update the model parameters. Later structures use the optimisation methodology Adam to update the weights [21]. The updates are made sequentially where either $\theta^{(D)}$ or $\theta^{(G)}$ is updated, while the other parameter is fixed.

Typically, the training stops when the game reaches a Nash equilibrium where one of the players does not change its decision independently of the other player. In most of the cases, there is not a Nash equilibrium and the training stops when the generation does not improve in quality [12]. A generator is said to be optimal when pmodel = *p(G(x))* whereas as discriminator is optimal when $D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p(G(z))))}$.

One of the biggest problem with GANs is that there is not an standard methodology to measure the performance of the generation process [20]. Thus, complicating the training process. Numerous statistics have been proposed but are designed for particular cases. Section 3.3 covers GANs measures proposed in the literature. This research evaluates the performance of GANs by the increase of accuracy triggered by adding synthetic data to the training set of a classifier.

The popularity of GANs has led to numerous modifications of the original structure. Most of the prominent modifications are Deep Convolutional GANs (DC-GANs), Conditional GANs(C-GANs) [22], Cycle-GANs [14] and Bidirectional GANs [23]. This projects lies heavily on DC-GANs and 3D-GANs. Section 3.3 describes in detail the different types of GANs and their applications.

2.2 Artificial Neural Networks & Deep Learning

An Artificial Neural Network (ANN) is a parametric machine learning model that uses a series of parameters ω to map an input to an output. The basic unit in ANNs are the input layer, hidden layer, and output layer. The input and the output layers represent the input and output of the model respectively. Whereas the hidden layer transforms the input into the output using model parameters. The parameters are learned by using gradient descent or Mean Squared Error Methods. Gradient descent methodologies are more frequent than Mean Squared Errors method. The learning policy tries to minimise a selected cost function given training data. Figure 2 shows a standard ANN structure.

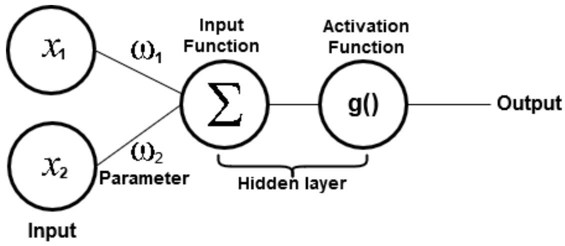


Figure 2: Artificial Neural Network structure

A Deep learning structure is a sequence of ANN layers. This structure can approximate complex non-linear functions. The parameters are trained by using a stochastic back propagation process. This method is a recursive method that transmits the gradient of the last layers to the initial layers of the Deep learning structure. Typically, deep learning structures are feed-forward. A feed forward structure presents the input signal to the network in sequential order without cycles. Some proposed structures contain cycles. Figure 3 represent a simple deep learning structure with three hidden layers.

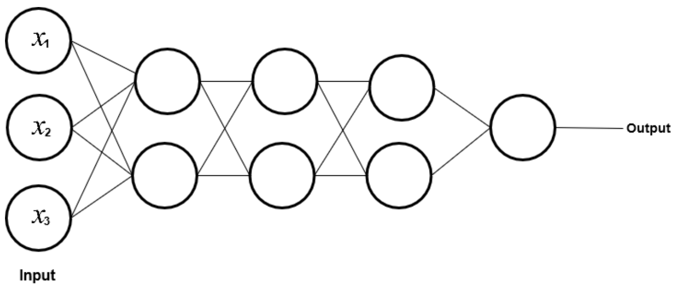


Figure 3: Deep Learning basic structure

There are almost an unlimited number of deep learning structures as there are large number of parameter to combine such as the type of hidden layers, number of hidden layer, types of activation function, and update processes. This leads to the implementation of suggested structures that have demonstrated to perform well on specific tasks. One of the most studied and used structures are Convolutional Neural Networks.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are deep learning structures frequently used for image classification. These structures reduce complex hierarchical structure, generally images, into a simplified representation that is easier to classify. Then, in most of the CNNs suggested structures, a standard feed forward deep learning network maps the resulting simplified representation into a class. The key elements in a CNNs structure are the convolutional filters, pooling layers, and Rectifier Linear Unit (ReLu) layers.

The key within a CNNs structure are the convolutional filters or Kernels and the strides. The kernels are the window that perform convolution operations over the input. The convolution operation performs a dot product between the network parameters and the model parameter within the window. Then, the output of the dot products are summed up into a value. The network parameters are learned to minimise the loss function of the structure. After each convolution the kernel moves based on the stride. The size of the Kernel window depends on the input size,however, the standard size is 3×3. Strides indicate the number of steps that the kernel takes after performing one convolution operation. Figure 4 represents a filter of 2×2 size with a stride of size 2.

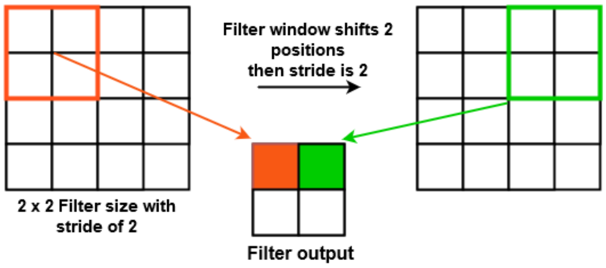


Figure 4: Stride and filter example

Another key elements in a CNNs is the pooling layer. Pooling layers reduce the dimensions of the output of a convolutional layer. These layers are filters that parse the entire convolutional layer output keeping the most relevant features for each step of the window. Finally, the Rectifier Linear Unit changes the negative values of the max pooling output to 0. A CNNs consist on a series of convolutional, max pooling and Re-Lu layer that reduces the input size until the resulting input is simple enough to be managed by simple neural networks. Fig 5 illustrates a standard CNNs structure.

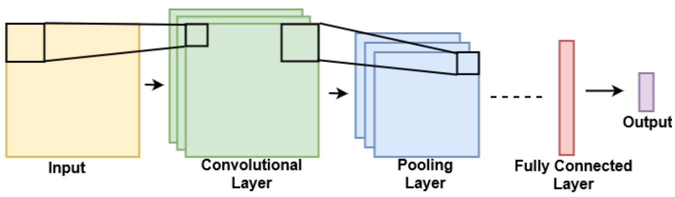


Figure 5: Convolutional Neural Network structure

Although most of the CNNs structures are designed for 2D images, the convolution can also involve a third dimension by adapting the kernel size to include an additional dimension in the convolutions. Figure 6 illustrates a convolution in 3D data. Fig 5

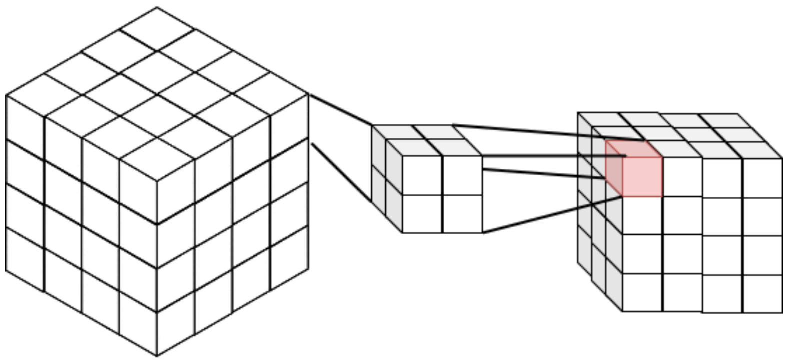


Figure 6: 3D convolution

2.4 Performance Measures: Confusion Matrix and Accuracy

Accuracy and Confusion Matrix are typical evaluation methods for classification methods. Accuracy, formulated in equation 2.3, is the ratio of number of correctly classified instances over the total number of instances.

$$Accuracy = \frac{Number\, of\, correct\, predictions}{Total\, Number\, of\, predictions\, made} \tag{2.3}$$

Confusion matrix evaluates the performance of a classification model on each of the classes. In a confusion matrix representation, typically, each row of the matrix represents the instances in a predicted class while each column represents actual instances in a class. Figure 7 shows the confusion matrix of a binary classifier and the possible outcomes.

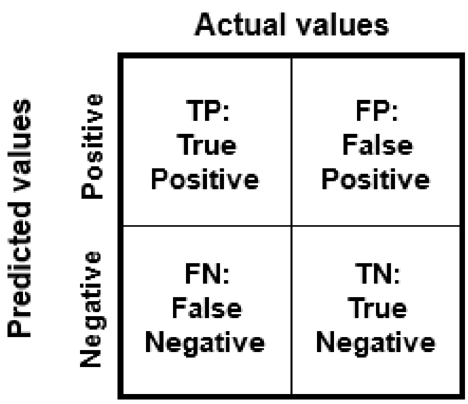
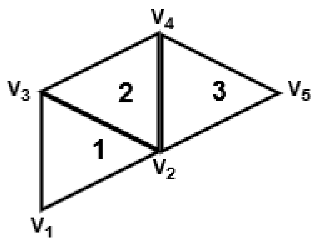


Figure 7: Confusion Matrix example

Confusion matrix reports the number of correct an incorrect classifications broken down class. Whereas Normalised Confusion matrix reports the proportion of correct an incorrect classifications broken down by class. Normalised confusion matrices allow the direct comparison between the individual performance of each class.

2.5 3D Data

This research uses three types of three dimensional data, namely voxelgrids, point clouds and triangular meshes. Triangular meshes represents the surface of 3D objects with a set of triangles that are interconnected by their vertices. Figure 8 illustrates a simple representation of a triangular meshes, complex representation contain more information such as adjacent triangles and edges. Processing triangular meshes can be simplified by doing calculation on the common vertices rather than for each single triangle triangles. Figure 9 shows an example of a triangular mesh representing a 3D object.



Triangle Index	Vertices
1	(1,2,3)
2	(2,3,4)
3	(2,4,5)

Figure 8: Triangular mesh example

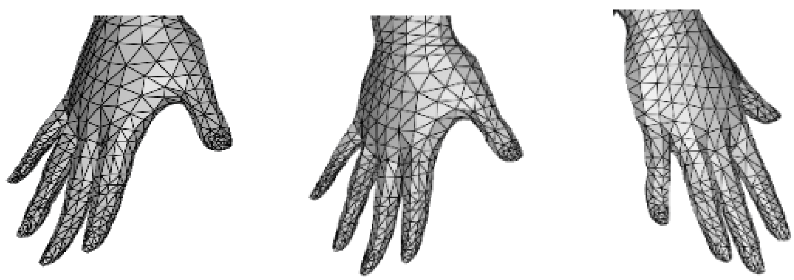


Figure 9: Triangular mesh example

Point clouds represent geometric objects as a set of points in a *x,y,z* space in a Euclidean coordinate frame [24]. Point clouds are represented as a $N \times 3$ matrix, where N is the number of points. Normally, N is labelled as the point clouds resolution, the higher the number of point used to represent an object the higher the fidelity of the representation. Point clouds are considered an standard format to represent 3D data since are the output format of common scanning devices such as LIDAR scanners, RGBD cameras, and Kinect [25]. Figure 10 shows a point cloud representation in a *x,y,z* space.

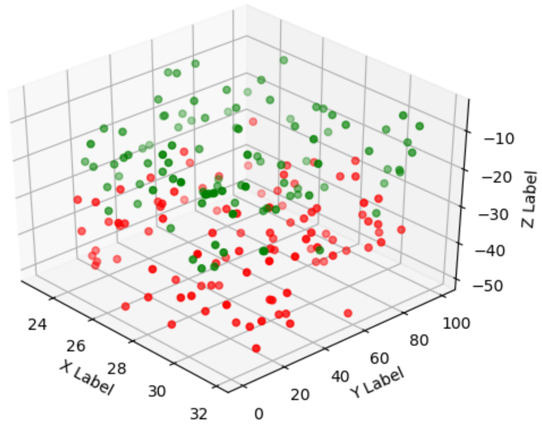


Figure 10: Point cloud representation

A Voxelgrids is a grid in a three dimensional space and a voxels is a point in the three dimensional grid. In contrast to point clouds and triangular meshes, voxels are not represent with x,y,z coordinates. Instead, voxels are represented as a value in a grid that indicates the position of the voxels based upon the other voxels in the grid. The value of a voxels in the grid is usually binary, with 0 indicating that there is not voxel in the coordinate and 1 represents a space occupied by a voxel. Otherwise, to represent voxels in grey-scale, the values in the grid could take values in the (0-1) range. Typically, voxelgrids are represented with 3D arrays. One problem with voxelgrid representations is their sparsity and large dimensionality, a $64 \times 64 \times 64$ array has 264,144 coordinates. Typically, voxelgrids are used in medicine [26] and landscape representations [27]. Sections 3.2 and 3.1 show detailed information about the applications of voxelgrids and methodologies to process them. Figure 11 shows an voxelgrid example.

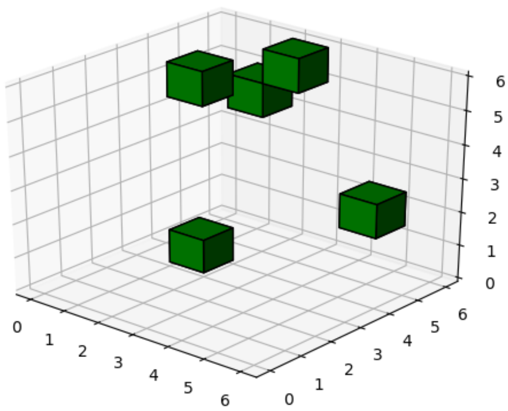


Figure 11: 6x6x6 voxel grid representation

3 Related Work

Identification of human actions has been actively researched in computer vision, however, is a yet under-explored problem because the complexity of modelling human motion. The development of three dimensional representations of real world objects and Generative Adversarial Networks has converged with computer vision resulting in a promising research domain to approach action recognition. This section presents a review of the previous work done in three dimensional computer vision, Generative adversarial networks and its applications in three dimensional computer vision and data augmentation since it is relevant to the work presented in this research.

3.1 3D data

Three dimensional (3D) depictions of objects are a key element in areas such as computer vision [1], augmented reality [2], virtual reality [3], medicine [4], and robotics [5]. 3D data can represent spatial details that are impossible to convey with conventional 2D pictures. The main obstacle to manipulate 3d representation is the high computational and memory cost as a result of the additional dimension [6]. There are multiple formats to represent 3D objects, the most frequent are view-based projections, triangular meshes, volumetric grids, and point clouds. Section 2.5 explains each 3D data format in detail. Figure 12 illustrates the differences between different 3D formats

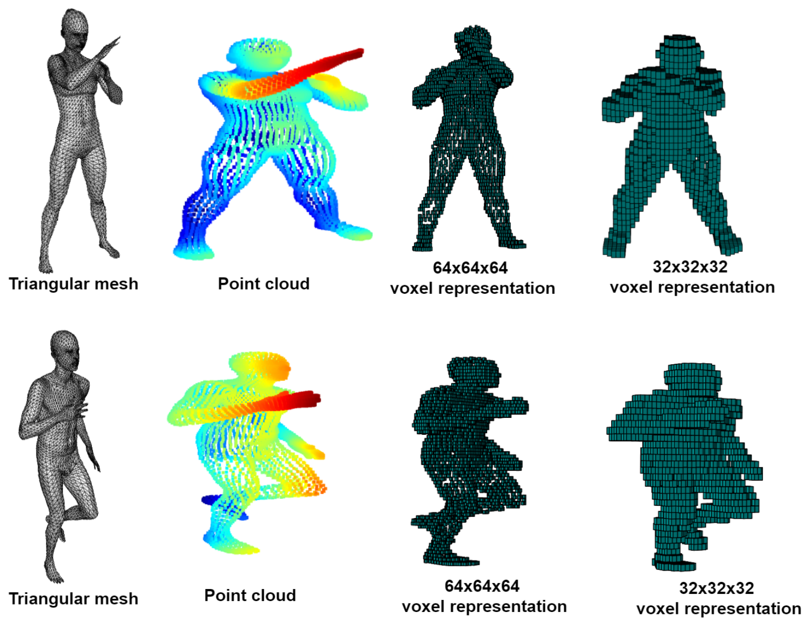


Figure 12: Standard 3D data formats

Multi view projections simplify the analysis of 3D objects but can only show the 3d objects surfaces and does not consider internal information. Triangular meshes and point clouds encoding format scale better than other formats because its reduced size. Additionally, point clouds are the standard output format of common scanning devices such as LIDAR, RGBD cameras, and Kinect [24]. However, point clouds and triangular meshes do not have uniform dimensions and do not convey any information about the neighbour points of a point and the spaces that are not occupied [25].Voxelgrids are hard to handle because the large dimensionallity but Voxelgrids can convey spatial and neighbour information that other 3D formats cannot convey. Within the computer vision domain, particularly in the shape recognition branch, the focus is in the elaboration of classification structures able to efficiently process and understand 3D data. Then, implement these classifiers to improve processes in fields where 3D data is frequent.

3.2 3D objects classification methodologies

The large dimensionality of 3D data condition the methodologies that can be used to perform classification in this format. Convolutional neural networks (CNNs) tend to perform better than other methods on big dimensional data which makes CNNs the preferred methodology to classify 3D data. Another approach to classify 3D data is based on the simplification of 3D data into a reduced space that can be used by standard classifiers. Following this approach De Deuge et al. [28] uses unsupervised Deep learning to reduce the data dimensionality, and then applies a nonlinear SVM in the reduced space. Shape descriptors can be extracted from 3D data and then feed a fully connected neural network with the descriptor [29]. Generally, the classification methodologies based on the simplification of 3D data, frequently, do not scale well on large dataset and are slower than methods based on 3D CNNs [6]. On the other side, within the 3D CNNs classification methodologies, there are several approaches that depends on the input format; volumetric CNNs, multi-view CNNs, point cloud CNNs, and spectral CNNs.

Volumetric CNNs use voxelized shapes as input. Voxelized representations are able to provide neighbour information between the elements in the 3D space and distinguish between free and occupied spaces [6]. However, Volumetric CNNs are limited by the computational cost of handling big dimensional and sparse high resolution 3d voxelized data [7]. As a result, multiple 3D CNNs deep learning structures have been proposed to make convolutional processes tractable and improve its performance. Shapenet [30] and Voxnet [6] are pioneer 3D volumetric CNNs structures to perform shape classification. Other applications of volumetric CNNs include generative models [31] and variational autoencoders [32]. Volumetric CNNs have been also used for video classification where the third dimension is the time dimension instead of volume [33].

Multiview CNNs transform 3D images into multiple 2D images. Then standard 2D CNNs are implemented for their classification [34] [35]. This approach avoids the high computational cost and memory limitations of 3D data. Multiview CNNs performance relies on 2D CNNs structures, the method to render 3D images into 2D, and the methodology to combine multiple classification result into a single classification. FusionNet [36] ensembles volumetric and multiview CNNs to boost the performance of 3D classifiers.

Spectral CNNs can classify 3D data given in mesh format. Theses methodologies are limited to manifold 3D meshes and it is no clear how this methodology can be applied to non-isometric meshes. [37] use spectral CNNs to classify 2D data projected into a 3D maninfold while [38] use the methodology to classify 3D human shapes.

Point cloud CNNs use point cloud volumetric data to perform classification. Most of the 3D data extraction methodologies produce point clouds by default. Therefore, no data processing is required which avoids loss of information. Additionally, voxel grids or multi-view data are highly voluminous data representations that might result on a computational intractability. One of the difficulties in the development of point clouds classifiers is the unordered structure of the point clouds. Because the unordered structure of point clouds, the classification models must be invariant to the input feeding order and have to be able to capture the relationship between the unordered points. Some networks have been proposed to perform classification with point clouds. Kd-network [39] represents point cloud information with kd-trees that are the input of a CNN structure. Point net[7] is a CNN structure that admits point cloud inputs and PointNet++ [40] is a variation of Point net that creates a hierarchical structure of point clouds where Point net is applied recursively on each of the local structures.

3.3 Generative Adversarial Networks

Generative adversarial networks (GANs) are deep learning generative models. Proposed in [12], GANs are able to model high dimensional data distributions by employing two deep learning structures, namely the generator and the discriminator. The discriminator differentiates between synthesised data and real data while the generator tries to fool the discriminator with synthesised data. Section 2.1 covers technical details of GANs. Among multiple applications, GANs have been used to study the representation and manipulation of data distributions, improve machine learning methodologies, deal with missing or incomplete data, outlier detection and synthesis of realist images [41].

There are five major GANs architectures; fully connected GANs, Convolutional GANs, Conditional GANs, Inference GANs, and adversarial autoencoders [42]. Each of these architectures share the same basic adversarial mechanisms but with structural changes and different functionalities.

Fully connected GANs (Figure 1) are a primitive GANs structure presented in [12] where the generator and the discriminator use fully connected networks. This structure is limited to the generation of simple data. Convolutional GANs modify the structure of fully connected GANs with Convolutional neural networks (CNNs) taking advantage of the suitability of CNNs to handle complex images [43]. Wu et al. [31] extends the concept of convolutional GANs to the 3D data domain by using 3D CNNs. 3D GANs are covered in detail in section 3.3.2. Conditional GANs (CGANs), represented in figure 13, were suggested in [22] where the generator and the discriminator are class conditional. CGANs improve the generation of multi-modal data and allows the synthesis of a particular class.

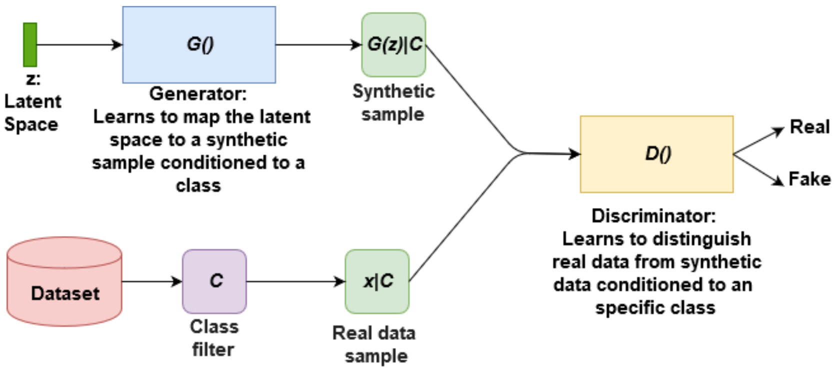


Figure 13: Conditional GANs (CGANs) standard structure

GANs with inference models include an inference mechanism to map real data to the latent space z. The inference system provides GANs with the ability to perform conditional generation, semi-supervised learning and sample reconstruction [16] [23]. Figure 14 illustrates and ALI or BiGAN structure, a standard inference GANs. Illustrated in figure 15, adversarial autoencoders employ an autoencoder in the standard GANs architecture. The autoencoders encoder output aims to match the distribution of the latent space z whereas the autoencoders decoder tries to reconstruct the original image from the encoder's output. In this framework, the discriminator differentiates between GANs latent spaces distributions and the output of the encoder. Adversarial Autoencoders have applications in clustering and semi supervised and supervised learning [44].

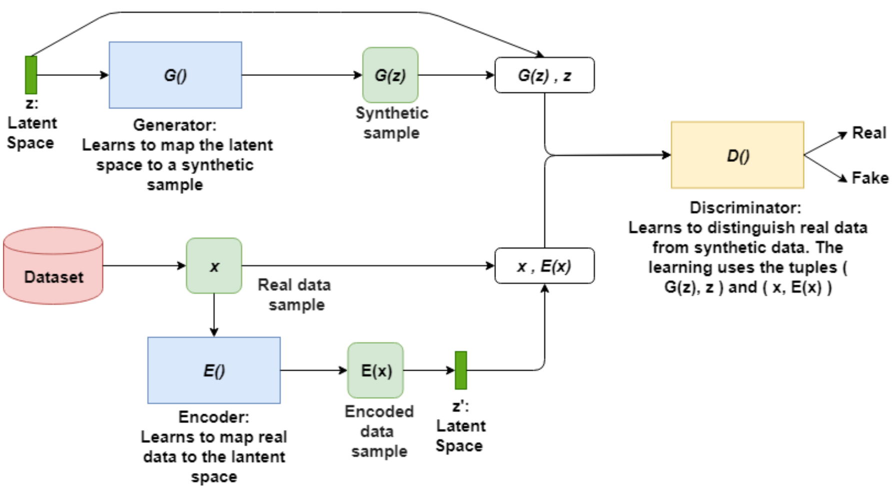


Figure 14: ALI/BiGAN basic structure

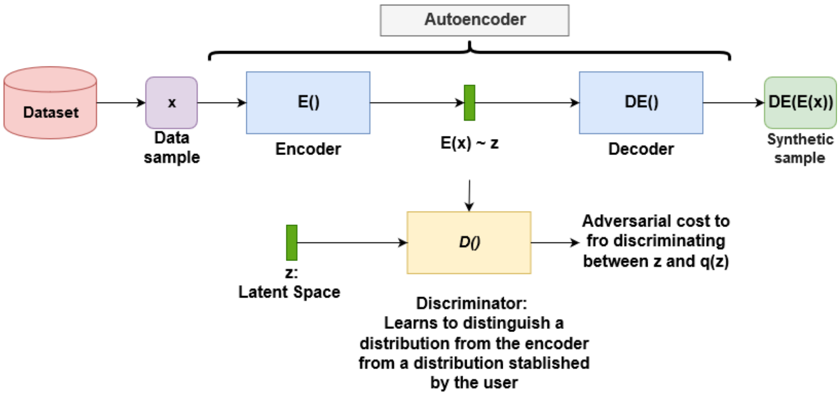


Figure 15: Adversarial autoencoder GAN basic structure

Although image synthesis is a frequent application of GANs [43] [41], GANs can be applied for other tasks such as data augmentation [8], improving performance of reinforcement learning models [45] [46], inference [16] [23], semi-supervised learning [47], imitate agent policies [48] , data privacy [10], anomaly detection [49], and domain transfer [50]. Section 3.3.1 covers the application of GANs for data augmentation.

Despite the great success of GANs, GANs training process is unstable and challenging [51]. GANs training is based on the zero-sum non-cooperative game that converges in a Nash equilibrium when one of the players does not change its decision independently of the other player decision. This is the optimal point in the GANs minimax objective function represented in equation 2.2. However, this equilibrium is not guaranteed [20]. Even when the model converges, there is not guaranty that the distribution of the generated data is close to the probability distribution of the original data [52].

Ideally GANs can represent all the distributions within a data set. However, one common problem during the GANs training is 'mode collapse' where the generator only synthesises the same family of samples or just a single type of sample to easily fool the discriminator [51]. Mode collapse arises from situations where the generator is trained extensively without updating the discriminator. Then, the generator finds the data that best fool the discriminator. The diversity of the generator can be improved by using multiple GANs to cover all the modes of the distribution [53]. Vanishing gradient is another common problem during GANs training process where the discriminator loss converges suddenly to zero and the model stops learning [51]. This problem is usually triggered by the discriminator learning faster than the generator. Then, the distributions $p(x)$ and $p(G(z))$ do not overlap and the discriminator can differentiate between them easily. Because the generator is trained via the discriminator, the generator does not receive gradient updates when the discriminator loss converges to 0. Adding noise to the generator have a positive effect on avoiding this problem [51].

Some solutions to improve GANs training relies on modifying the generator and discriminator structure [43], adding noise to the discriminator [54] , limiting the discriminator training if its accuracy is under a specific threshold accuracy [31], and modifying the generator and discriminator cost functions [55] [56]. In addition, Salimans et al. [20] suggests several approaches to improve GANs training process. The first method changes the generator objective to match the generated images with the discriminator's intermediate activation of the real data. This modification aims to increase the amount of information available. A second methodology, heuristic averaging, aims to speed up model convergence. Heuristic averaging consist on penalising the network weights if these weights deviate from the running average of previous weights. The third, mini-batch discriminator enables the discriminator to be aware of the differences between the generated distribution and the real distribution as a whole. The method compares the distance between batches of real data and synthesised data. Then, the extra feature is used as input for the discriminator to avoid mode collapse. A fourth methodology, one-sided label smoothing establishes the discriminator target for real data as 0.9 instead of 1 to smooth the discriminator decisions and prevent an overconfident discriminator. The fifth approach, virtual batch reduces the dependency of an instance to other instances by normalising every instance within a training mini-batch. The normalisation is based on the statistics of a reference batch retrieved at the beginning of the training process.

While much progress has been made to understand and improve GANs training process[51] [20] [52], there still remain the challenge of measuring GANs performance. There is not an effective methodology to evaluate quantitatively the fidelity of the synthetic images and it is not clear whether different GANs methodologies should be compared [41]. Some used evaluation methodologies are the like-hood estimation [12], and human inspections [57]. The absence of a procedure to measure the generation quality complicates the hyper-parameter tuning process. This is particularly concerning because GANs sensitivity to hyper-parameters [58].

3.3.1 Data augmentation with generative adversarial networks

Data Augmentation is a promising application of GANs aiming to solve problems experienced by deep learning models when the dataset is not big enough. Deep learning models have demonstrated unprecedented performance on machine learning tasks. In exchange, these models require large amounts of data to avoid overfitting and lack of generalisation. Additionally, imbalanced datasets result on the model to fall short.

The literature has developed several techniques to avoid loosing performance because overfitting. One approach is to add additional processes to existing deep learning structures such as batch normalisation [59], normalisation layers [60] and dropout [61]. When the training data is particularly small , these techniques cannot capture properly input invariances that are useful for the training process [8]. Another approach is to generate additional data by modifying the original data with augmentation processes.

Augmentation methodologies apply transformations to the original dataset to create new data and improve the generalisation ability of the classifiers. Common augmentation techniques in 2D and 3D computer vision are flips, rotations, gaussian noise, and random translations [62] [63] [6]. The application of these techniques is a common practise for large and small datasets because the proven benefits [64]. However, normal augmentation techniques does not represent the underlying data distributions, are limited to simple data variances, and produce highly correlated training data [10]. These limitations motivated the implementation of image synthesis methods able to induce variability to the augmented data while representing the underlying data distributions.

GANs can model wide large invariance and produce data that comes from the original data distribution. Consequently, the literature has started to test the ability of GANs as a method for data augmentation purposes. [8] proposed Data Augmentation Generative Adversarial Network (DAGAN), a GANs framework based on conditional GANs able to transform data withing the same domain. DAGAN transforms data that belong to a class into data of the desired class. DAGAN is implemented to synthesise data that belongs to a class with low frequency to balance the dataset and increase the classification performance on multiple 2D image public datasets. [9] augments the images of a dataset with standard augmentation methods and then synthesises new data with GANs using the already augmented data as a input to improve the liver lesion classification. [10] synthesises brain tumor MRI scans with image-to-image GANs that modify the characteristics of the original images to obtain new images. In addition, This research proves the capacity of GANs to create anonymous synthetic data to be used to train effective classification and segmentation methods. [65] achieves a superior bone lesion classifier by using synthetic data from GANs. To do so, the research uses cycle GANs to synthesise images with bone lesions from a particular part of the body from images without lesions using images with bone lesion from a different part of the body. Finally,[66] suggest Conditional Progressive Growing of GANs (CPGGANs) to synthesise MRI images of brain images with bounding boxes indicating brain metastases to improve the performance of object detection classifiers such as YOLO [67] or R-CNNs [68].

Data augmentation with GANs is particularly suitable in medicine related tasks because the lack of labelled data and the strict privacy requirements. To the best of our knowledge,there is not a proposed GANs data augmentation experiment that uses 3D data and implements a specialised 3D GANs structure for the augmentation process.

3.3.2 3D Generative Adversarial Modeling

Initial GANs architectures work only with 2D data such as images. The increasing popularity of 3D data and the development of 3D deep learning structures instigated the development of 3D GANs structures [31]. 3D GANs make possible to obtains the benefits of using GANs in domains where 3D data is used extensively.

Initial 3D generative methods reconstruct and generate new 3D images with non-parametric approaches based on retrieving and combining elements from the dataset [11] [69]. With this approach, 3D synthesis was constrained by the availability of morphological 3D templates, supervision during the process, and the 3D elements available in the repository [31]. Most of these methods use 3D data formats that can be represented in 2D such as CAD wire-frames [70] , meshes and skeletons [71]. Another 3D image synthesis approach is based on deep learning methods such as Recurrent Neural Networks [72] , Deep Belief Networks [30], Deep Convolutional Auto-encoders [73], and Capsule Networks [74]. Using 3D deep learning synthesis methods, [75] [76] synthesise 3D data from 2D data , [72] reconstructs 3D images, [77] simplifies 3D images into discriminative representation, and [78] transforms the 3D data format from point clouds to voxelgrids. Voxelgrids and point clouds are typical 3D data formats used in 3D deep learning synthesising methods. 3D image synthesis with deep learning requires, in most of the cases, full supervision and are limited by the variance that can synthesise.

3D GANs architectures aim to overcome the problems of previous generative methods. Implementations of 3D GANs claim that GANs, in contrast of primitive 3D synthesis methods and other deep learning generation methods, does not require structural templates, does not borrow items from the dataset, generate realistic object with variations, and does not require supervision [31]. However, 3D GANs are particularly hard to train because the big size and complex distributions of 3D data [79]. Depending on the type of 3D data format used as an input for GANs, there are two approaches; GANs that works with voxel grids and GANs that use point cloud 3D data.

Motivated by the lack of GANs methodologies for data in 3D formats, Wu et al. [31] suggested a 3D GANs framework based on volumetric CNNs able to synthesise voxelgrid 3D shapes. Besides synthesis, 3D GANs, once trained, can map complex 3D voxelgrids into an informative feature representation which are able to improve the classification processes. [79] identifies the complex training process of voxelgrids based 3D GANs. As a result, the research proposes a 3D GANs structure to make improvements in training and convergence time. This simplified structure uses a reduced size voxelgrid input and a training objective function based on the Wassertein distance with gradient normalisation [55]. Voxelgrid based 3D GANs have been applied for 3D image edition [80].

Achlioptas et al. [24] proposed the first GANs architecture for point clouds. The motivation of generating point clouds lies on avoiding unnecessary transformations when the target modality is in point cloud format. This initial point cloud based GANs uses fully connected layers for the discriminator and 1D-convolutional neural networks for the generator. [81] modified the initial point cloud GANs architecture. This modified version uses graph convolutions for the generator to capture the structural information of the input and improve the generation quality. [82] uses a tree structure to rearrange the input data and make the architecture proposed in [81] computationally tractable.

3D GANs development has been focused on developing structures to improve synthesis quality and training stability. To the best of our knowledge, there are not equivalents of well known GANs frameworks compatible with 2D images in 3D GANs framework. Additionally, there are not implementation of 3D GANs in the data augmentation, anomaly detection, data privacy, and domain adaptation frameworks. This research uses proposed 3D GANs architectures with the training improvements suggested by [20] to augment 3D datasets and improve the classification performance of human actions encoded in 3D data.

4 Approach

Three dimensional data is able to better represent the reality of data and their associated problems. However, its acquisition is not as simple as with 2D images and is harder to manipulate because its large dimensions [6]. Deep learning based on convolutional neural networks is a promising methodology to process 3D data for classification problems [30] [6]. As a drawback, deep learning do not perform well with small training sets which is a common problem with 3D data as is acquisition is not straightforward.

Traditionally, data scarcity is solved using augmentation methods that slightly modify the original data [62]. Generative adversarial Networks are a promising technique to synthesise data and augment a dataset as are able to generate realistic data with not seen variations [12] [8]. Data augmentation with GANs has been done frequently with 2D data [8] [10] [65]. However, no augmentation scheme has been suggested with 3D GANs.

This research evaluates GANs as method to improve the performance of 3D based classifiers with synthesised data. To do so, the research evaluates four key aspects of the augmentation process in a classification experiment. This experiment uses a 3D deep learning classifier to map 3D frames of a human doing an action to the action that the human is doing in the frame. Then, a 3D GANs generates synthetic labelled frames that are used to create an augmented dataset to train a new deep learning classifier.

In the experiment, the first aspect to evaluate is whether the augmentation of 3D data with GANs has the ability to increase the overall performance of a 3D classifier. The second is to study if GANs are able to synthesise meaningful data for all the different classes represented in the dataset or just a specific group of labels. The third evaluates whether the number of synthetic instances that are used for the augmentation process has an impact on the classification performance. Finally, to analyse if a 3D based classifier can reach good performance in detecting actions just using 3D frames. This four analysed aspect in the experiment can be translated into research questions:

- Can Three Dimensional Generative Adversarial Networks increase the performance of deep learning models through augmentation methods? Does this increase of performance depend on the number of instances available for training?
- How many synthetic instances have to be added to the original dataset to maximise the performance of the augmentation strategy ?
- Do Generative Adversarial Networks synthesise the data from the different label with the same quality? Where quality is measured as the improvement in classification performance for the specific label
- Does volumetric data provide enough information to be used in an frame based action classifier?

To the best of our knowledge no research has evaluated the potential of 3D based GANs to augment a 3D dataset and no research has used an action recognition classifier with a frame based approach while using volumetric data. The research opens the door to application of GANs into areas where 3D data is used such as robotics and medicine.

4.1 Dataset

This experiment uses the public available human action dataset Dynamic FAUST. The dataset was created by Bogo et al. [83] and contains 3D scans of human subjects in motion. The dataset contains information of 10 subject doing 14 different actions. The actions are represented as a sequences of frames where the subjects are represented as 3D triangular meshes. Figure 16 represents an action as a sequence of frames

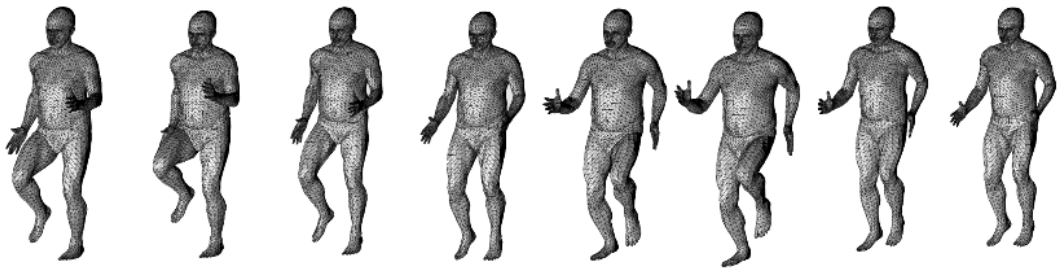


Figure 16: Dynamic FAUST: action as a sequence of frames

The dataset contains a total of 40,000 frames and the number of frames per action depends completely on the action and the subject who is doing the action. The actions represented in the dataset are: punching, running on spot, chicken wings, moving hips, moving knees, jumping jacks, shake arms, shake shoulders, shake hips, one leg loose, one leg jump, soft hop with two legs, one leg hop, and jiggling on toes. Appendix A show the label assigned to each action. Figure 17 shows a sample of the frames within the Dynamic FAUST dataset.

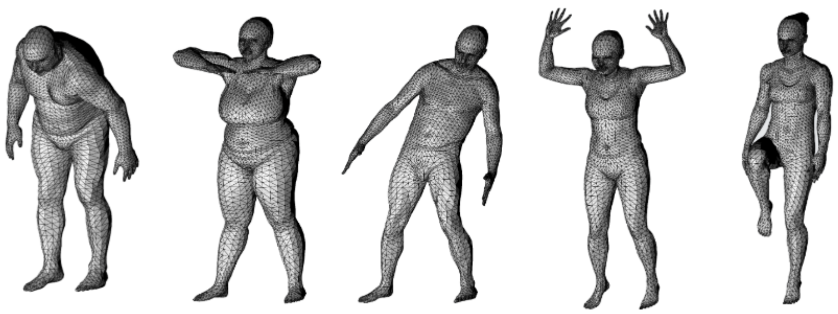


Figure 17: Dynamic FAUST frame samples

As a limitation, the dataset do not provide the equivalent 2D images of the three dimensional frames that were used to create the 3D objects. Consequently, a comparison between the classification of 3D frames and 2D frames can not be made in fair conditions.

4.2 Data Pre-Processing

The original dataset experienced two transformations; a transformation of the 3d format to represent the frames and a filtering process for the frames that are used for classification.

In first place, the original dataset format, triangular mesh, is transformed into point clouds and voxelgrids because the lack of methodologies for triangular meshes. Althought there are 3D classification methodologies that use triangular meshes, these methodologies are limited to specific shapes [37] and there is not a GANs methodology compatible with the format. Whereas, other 3D classifiers that use formats such as point clouds and voxelgrids have less restrictions and are more developed than mesh based classifiers [6] [7]. Additionally, there are GANs frameworks for point clouds and voxelgrids [31] [24].

To transform triangular meshes into point clouds, the vertices of the triangles were transformed into points in a x,y,z plane and the links between vertices were deleted resulting in a point cloud. Then, point clouds are transformed into voxelgrids using the spatial occupancy method [84]. This method overlaps a grid of voxels over the point cloud space and for each voxel in the grid a binary decision is made based on whether the voxels grid is occupied by points clouds. If a voxel is occupied, the grid coordinate gets the status of occupied (1) otherwise the grid receives the status on an empty space (0). The quality and the fidelity of the voxelgrid representation increases with the dimensions of the overlapping voxel grid. However, an increase in quality increases the computation complexity as the number of voxels increases as the cube of the dimensions of the voxelgrid. This research uses voxelgrids of $32 \times 32 \times 32$ and $64 \times 64 \times 64$ size as are the standard sizes in the domain [6] [30] [31]. All the frames were transformed into point clouds and then into voxelgrids. Figure 18 illustrates the voxelisation process [85].

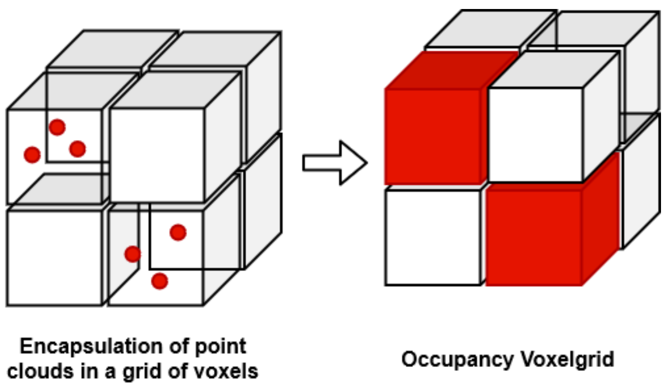


Figure 18: Voxelization process

This research uses a 3D classifier trained with frames of an action to classify a human action just with one frame. The performance of this methodology relies heavily on the quality of the information provided in the frames used for training [86]. Consequently, once the frames are transformed into point clouds and voxelgrids, the frames are filtered to keep only informative frames.

The initial frames and the last frames of a sequence do not provide any information about the action and are removed from the whole sequence. Each action is made by a sequence of frames that represent the different situations in an action. However, in this dataset, the initial frames of an action do not contain any relevant information as the subject is in a steady state. Then, after several frames, the subject starts to perform an action. The same issue happens with the last action frames.

In addition, after removing the non representative frames, consecutive frames that present similar information are smoothed into a single frame. In the dataset actions are presented as frames of an animation animation, to produce an animation, consecutive frames have to be similar. However, these similar frames provides duplicate information. To remove the duplicate information, the frames are grouped in sequences of five consecutive frames as suggested in [87]. In each group of frames, out the five frames, one is kept in the dataset and the other four are removed. Hence, keeping differentiated frames that represent key parts of an action. After the pre-processing stage, the number of frames available in the dataset is 2634. All the action present a similar number of frames. Figure 19 represents filtered actions in the different 3D formats used in this research.

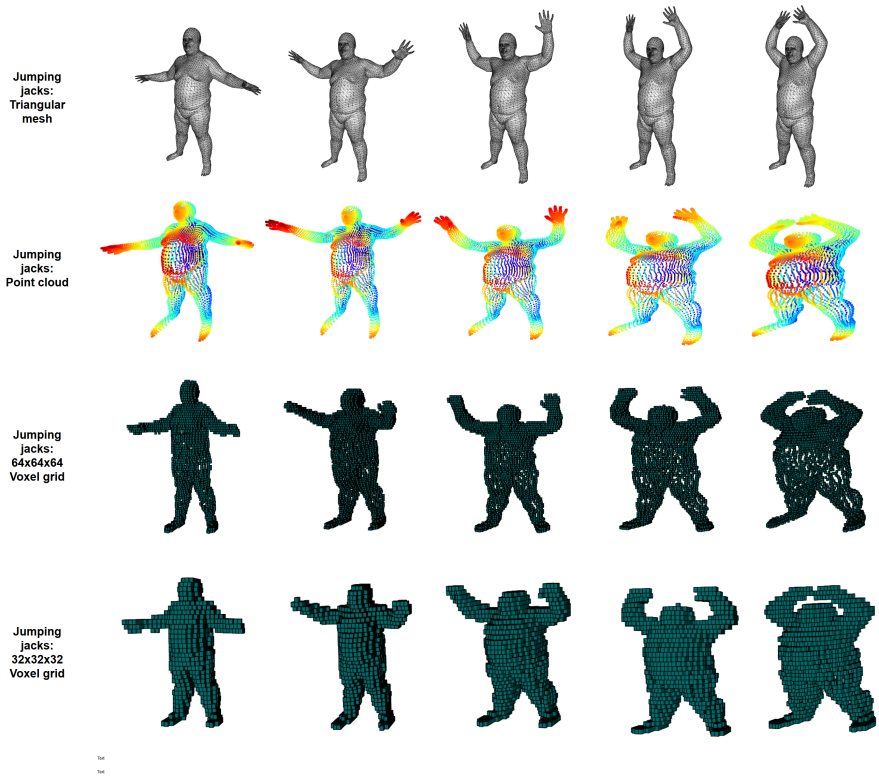


Figure 19: Frames of a sequence in different 3D formats

4.3 Dataset Split

To investigate the effect on the amount of data needed for an effective augmentation strategy. After the pre-processing stage, three different dataset with different sizes were created, namely small, medium, and big datasets. The small dataset contains 20% of the processed set ,the medium 60% and the big dataset contains all the instances available in the processed set. The medium and small set were created while keeping the same proportion of frames per action as the proportion of frames per action in the original dataset. The frames used to created the datasets were chosen randomly.

These datasets are created to evaluate the capacity of GANs to improve deep learning models in different scenarios with different data limitations. The evaluation is made by applying the proposed classifier and augmentation scheme into all three datasets. Then, comparing the impact of the data augmentation impact across datasets. In each of the three created sets, 80% of the dataset is used as a training set and 20% as a testing set.

4.4 Action Classification

The research aims to use a classifier able to handle 3D data and use it to identify frames of human actions. Consequently, evaluating the potential of classifiers that use 3D data as an input and the capacity of 3D GANs to improve classification processed with data augmentation. The proposed 3D classifier classifies one single 3D frame into an action or label. Although identifying a human action just from a single frame is possible and has showed good results, the methodology can be implemented into complex action recognition processes by classifying multiple frames of an sequence of frames and using a voting system to identify the action represented in the sequence [87]. Hence, if the classifier has a high performance in mapping a frame to an action, an action classifier based on voting multiple frames should have a high performance. Normally the number of frames used to detect an action is between 1-7 [86]. This approach has been used previously with 2d data [86]. Although the methodology has good performance, it does not use volumetric information. It is expected that the 3d information will boost the classifier as it contains valuable information. Not similar experiments have been done using 3D based classifiers.

Volumetric CNNs are an attractive methodology to do classification while considering spatial information. Volumetric CNNs tend to reach good performance compared with other classifiers. Other 3D classification methodologies such as simple classifiers, point cloud deep learning and multi-view classifiers are not in line with the project approach or perform worse than volumetric CNNs. Althought, Point clouds based classifiers avoid losing information because no data transformation is required, this methods do not explore spatial and neighbour characteristics and tend to perform worse than Volumetric grids. [7]. Simple classifiers can not handle the large dimensions of 3D representation of humans actions [6]. Finally, multi-view classifiers perform well but does not explore thoroughly the spacial characteristics of the data [34]. The major problem with volumetric CNNs is to find an structure to handle the large dimension of voxelgrid data.

Deep learning 3D CNNs structures are made of multiple layers interconnected using volumetric CNNs as a back bone layer. The most frequent layers in 3D volumetric structures are the input layer(I), fully connected layer(FC), and pooling layers (P) [6]. However, there is an unlimited number of combination of layers and hyperparameters. The 3D action classifier employed in this project follows a Voxnet architecture [6]. Voxnet has proven to reach similar performance to other volumetric CNNs structures such as Shapenet [30] but Voxnet requires a smaller number of parameters. Voxnet is a feed-ford with a layer structure C(32,5,2) - C(32,3,1) - P(2) - FC(128) - FC(K) where K represents the number of classes, *C()* a convolutional layer, P a max pooling layer and FC a fully connected layer. In C , the first parameter indicates the filter size, the second the stride and the third the padding parameters. The output of the convolutional and fully connected layers is passed through a leaky rectified non-linearity unit (ReLU) [88] with parameter 0.1. To avoid model overfitting, the output of each layer is passed through a dropout regularisation process with a dropout rate of 0.5 [89]. The last fully connected layer activation function is a softmax nonliterary that provides a probabilistic output. Figure 20 illustrates a Voxnet architecture.

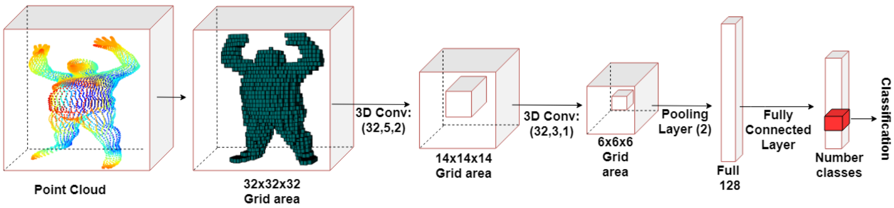


Figure 20: VoxNet Architecture

The input of a Voxnet structure is a set of of voxelgrids $\boldsymbol{X}$=$\{x_1, x_2, x_3...x_n\}$ where x_n represent a frame of an action as grid of size $I \times J \times K$ where I=J=K=32. The grid values are integers in the (0-1) range where 1 represents an space occupied by a single voxel and 0 an empty space. The classifier output is a label assigned to a single instance x_n. In this experiment, the labels are the actions to classify. An instance is labelled with the class with the highest probabilistic output in the softmax non-linearity layer. The labels are the actions represented in the dataset.

The network hyperparameters of the experiment 3D classifier follow the configuration suggested in Maturana et al. [6]. The model weights are trained with Stochastic Gradient Descent with momentum rate of 0.9 and a learning rate of 0.01. The training objective function is a multinomial negative log-likehood. The batch size is 32. The structure parameters are initialised using a zero-mean Gaussian distribution.

4.5 3D Generation & Data Augmentation

The performance of deep learning models depends on the amount and quality of the data used for training [8]. This research evaluates the ability of 3D GANs to synthesise new data and improve the performance of 3D deep learning models. To do so, the trained GANs add synthetic 3D data from the same distribution as the original data but with unseen variations to the training set. Then, the potential of the augmented data set is evaluated by comparing the performance of a classifier trained with and without synthetic samples. The performance of the augmentation lies heavily on the configuration of the implemented GANs and the type of GANs used [10].

Depending on the format of the synthesised data, there are two types of 3D GANs; point cloud based GANs [24] and voxelgrids based GANs [31]. In this experiment, GAN generates in voxelgrid format because the classifier uses voxelgrids. Hence, avoiding loss of information when transforming the data. The structure of the generator and the discriminator is crucial for the ability of GANs to synthesise good looking images [79]. The implemented voxelgrid GANs is based on the original 3D GANS structure proposed in Wu et al.[31] and the training improvements suggested in Salimans et al. [20].

The generator is a feed forward deep learning structure made of five volumetric CNNs. The number of channels is {512, 256, 128, 64, 1}. All the volumetric CNNs have kernels of size 4 and all the layers but the first layer have a stride length of 2, the first layer has a stride of length 1. The structure includes ReLU and batch normalisation layers after every volumetric CNNs. The Generator input is a 200-size vector, this vector is retrieved from a Gaussian distribution (0, 0.33) as is empirically shown that improves the model convergence and the synthesis quality [51]. The generator output is a voxelgrid matrix of $64 \times 64 \times 64$ dimension with values in the (0-1) range. Althought, the original 3D GANs structure suggest to use $min\ \log(1 - D(G(z)))$ as generator loss function where $D(G(z)$ is the generator performance, a generator loss $max\ \log D(G(z))$ is used as provides stronger gradients and avoid gradient vanishing problems [20]. Figure 21 illustrates the discriminator.

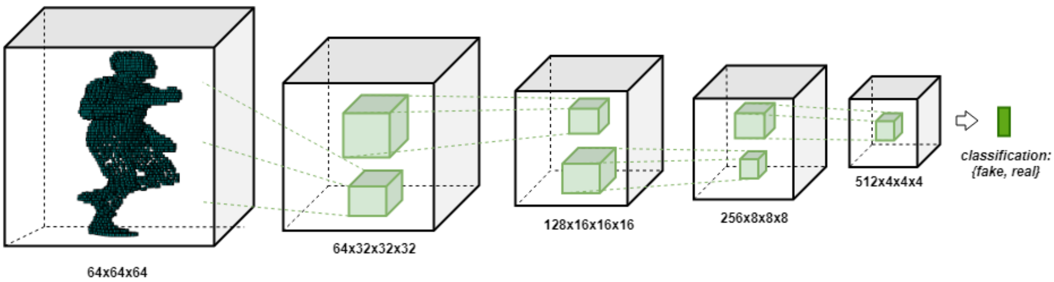


Figure 21: The discriminator in 3D-GANs

The discriminator is a feed-forward deep learning structure made of five Volumetric CNNs. The number of channels in each CNNs layer is {64,128,256,512,1}. Each volumetric convolutional layer has a kernel size of 4 and a stride length of 2, the last layer has a stride length of 1 instead of 2. In addition, there are leaky ReLU layers with parameter 0.2 and batch normalisation layers after every volumetric CNN layer. The last volumetric CNNs layer has a sigmoid activation function. The discriminator's input is a voxelgrid matrix of $I \times J \times K$ dimensions where I=J=K=64. The output is in the range (0-0.9) instead of (0-1) because smooths the discriminator decision and avoids an overconfident discriminator. If the output is above 0.5 the instance is labelled as real while igf the output is below 0.5 the instace is classified as fake. [20]. Figure 22 illustrates the discriminator.

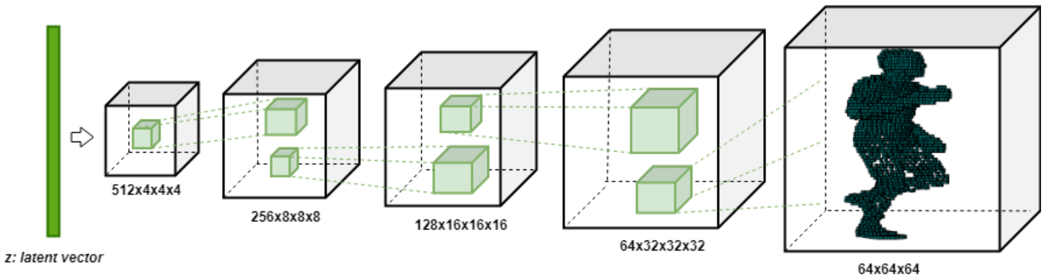


Figure 22: The generator in 3D-GANs

GANs are highly sensitive to the training configuration [58]. However, the lack of a standard method to measure synthesis quality and the long training process complicate the hyperparameters tuning process. In this project, the hyperparameters configuration is based on the original configuration with few modifications. The model parameters are trained with ADAM optimiser [21] with a $\beta = 0.5$. The discriminator batch size is 32. The model parameters are initialised using Xavier initialisation method [90]. Figure 23 represents the assembled 3D GANs structure.

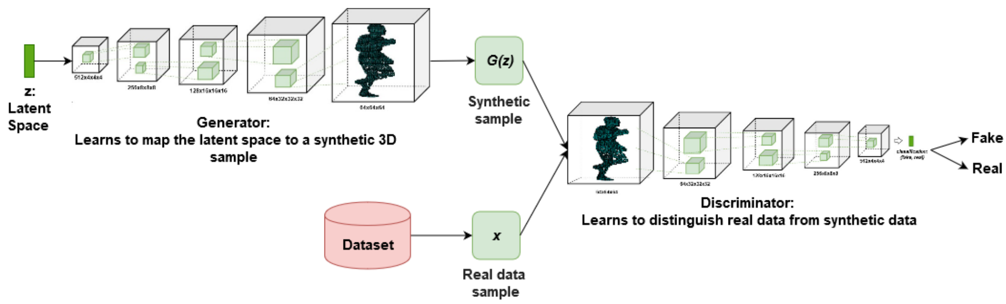


Figure 23: 3D-GAN standard structure

A common problem in 3D GANs is the discriminator learning quicker than the generator because the generation of voxelgrids is harder than distinguishing between synthesised and real voxelgrids [43]. This leads to the discriminator to differentiate instances perfectly while not issuing gradients. Without gradients, the generator cannot be updated resulting in a vanishing gradient [51]. To regulate the learning pace, this experiment GANs implements an adaptive training strategy [31] where the discriminator is updated only if the discriminator accuracy of the last batch is below 80%.

The GANs were trained with a generator learning rate of 0.0025 and a discriminator learning rate of 0.00005 as suggested in the original 3D GANs configuration. Other learning rates were analysed, if the discriminator learning rate is above 0.00005 the model tends to fall into a vanishing gradient. Whereas, if the discriminator leaning rate is below 0.00005 the model has a lower synthesis quality. Figure 24 shows the evolution of the GANs loss functions and discriminator accuracy when the discriminator learning rate is above 0.00005. In this figure the discriminator learns faster than the generator because the superior learning rate. The accuracy is always above 0.5 and the discriminator loss is always close to 0 leading to a gradient vanishing problem.

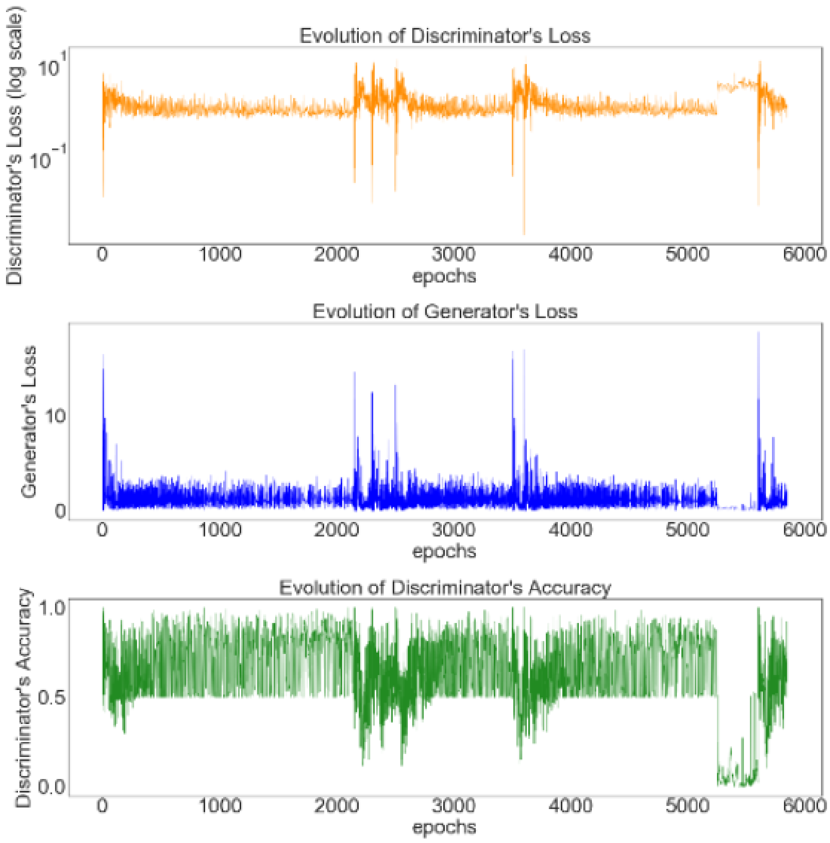


Figure 24: GANs training evolution whit high discriminator learning rate

A frequent stopping criteria in GANs is to stop the training process when there is not improvement in synthesis quality [12]. During this research 3D GANs training process, there is a point where the discriminator starts to learn at a faster pace than the generator despite the measures applied to avoid it. This triggers a vanishing gradient that leads to a continuous decrease of synthesis quality. Figure 25 shows the evolution of the discriminator accuracy in two GANs training process. In the first one, the discriminator gets a continuous accuracy of 100% after the 3500 epoch. Whereas in the second training process, the GANs enter into a vanishing gradient after the 4000 epoch.

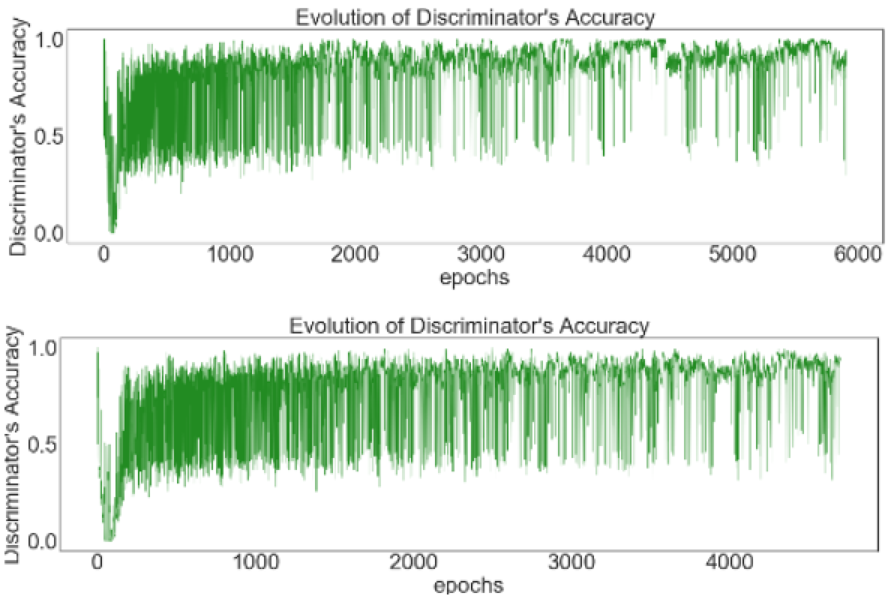


Figure 25: Vanishing Gradient in two GANs training process

The synthesis quality peaks in the epochs before the generator turning into a strict discriminative behaviour. This event happens for every label and in all the datasets in the experiment. Consequently, in this project, the stopping criteria is the generator reaching a constant accuracy of 100%. Figure 26 shows the evolution of the quality of the synthesised data before the training process reaches a vanishing gradient and after.

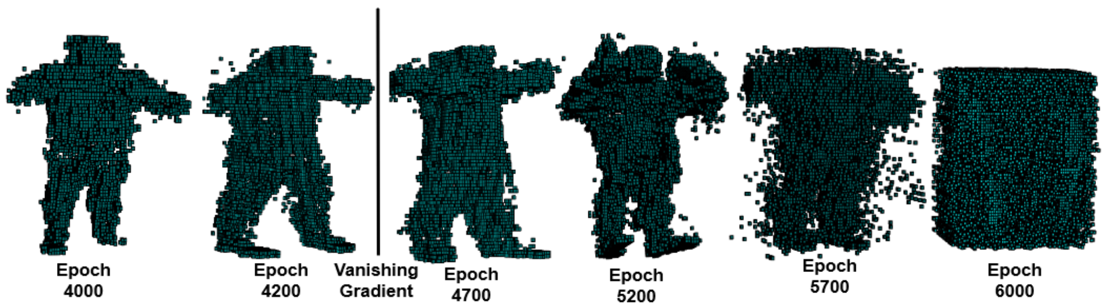


Figure 26: Synthesis quality: Vanishing Gradient

To generate labelled data, GANs are trained only using the data that belong to a class. In addition, training one GANs per class reduces the risk of the generator synthesising only few instances types to fool the discriminator [51]. As a result, multiple individual GANs are trained for every label in each dataset. The augmentation process finishes when the synthesised labelled data is added to the original dataset as a training data. Figure 27 illustrates the data augmentation process with GANs

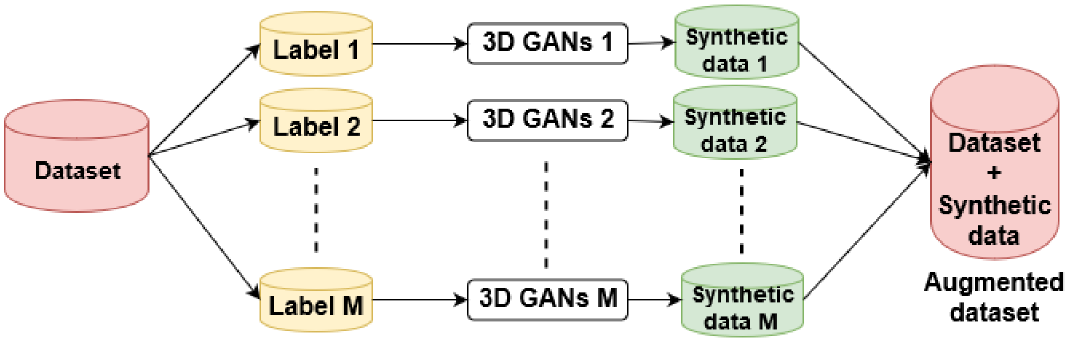


Figure 27: Data Augmentation Process

4.6 Evaluation

The proposed classification methodology is trained with a training set made of 80% of the instances of the processed set while 20% of the training set is used as a held out validation set. The classifier is evaluated with a testing set made of 20% of the processed data. The validation set is used to track whether the model is overfitting and to evaluate, during the training process, the best epoch to stop the training. The resulting model is the configuration of the model in the epoch of the training process with the highest validation accuracy. Figure 28 shows the evolution of the training and validation set.

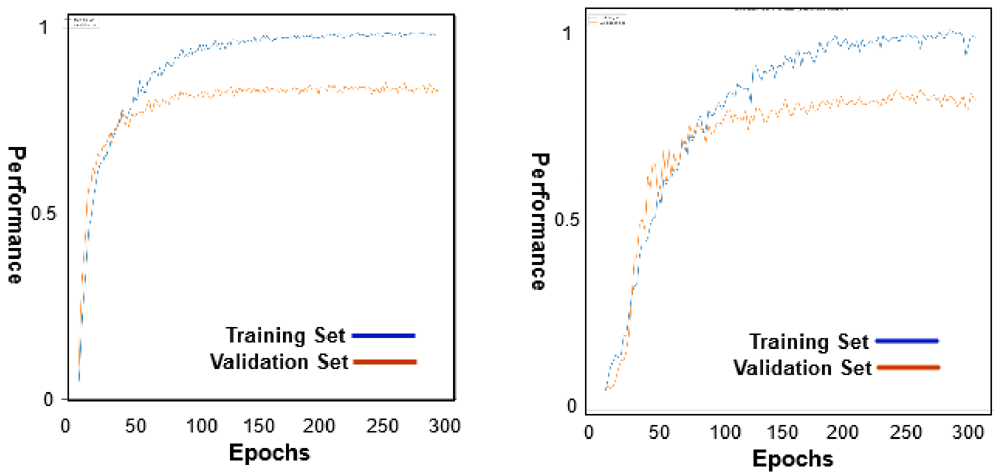


Figure 28: Example of Validation Evolution of 3D Classifier

Accuracy and confusion matrix are used to evaluate the classification. The evaluation follows a 10 fold cross validation. The accuracy reports the overall performance of the model while the confusion matrix is reported to evaluate the performance of the classifier for each individual action. The performance of the classifier is reported for the three proposed datasets, namely small, medium, and big. The comparison of the performance across datasets will show how the 3D classification of actions is affected by the size of the dataset.

To evaluate the impact of the data augmentation process, GANs are trained with the training set of each of the datasets. The output of the GANs is added to the training set to create an augmented set as showed in figure 27. Then, the 3D classifier is trained with the same approach as the non augmented data. However, the model is trained with the synthesised and training data. The number of synthetic instances to add is determined by the augmentation that returns the best performance in the classification stage. The performance of the classifier trained with and without augmented dataset are compared to evaluate the potential of an augmentation process made with GANs in different scenarios. Additionally, the confusion matrices of the augmented and non augmented dataset are compared to evaluate whether GANs can improve the action detection of all the actions or just detection of a selected number of action. Hence prove if GANs synthesise all the label with the same quality or just instances of specific actions. In the augmented sets, the 10-fold validation is ensured to not use synthetic data as testing data. Figure 29 summarises this project stages

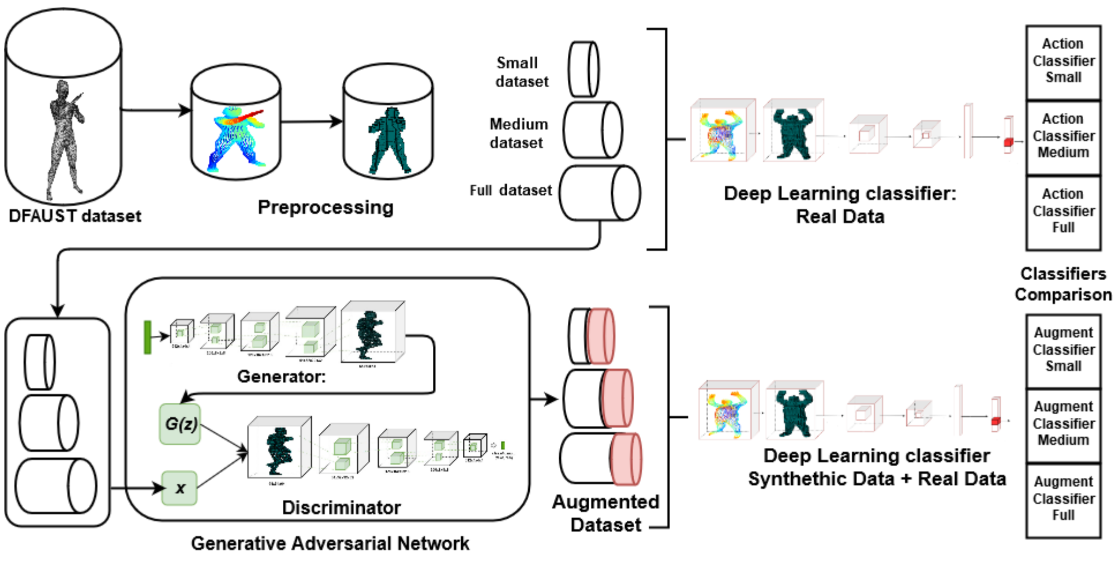


Figure 29: Project pipeline

5 Results

5.1 Qualitative Analysis

A GANs with the configuration stated in section 4.5 was trained for each label in each of the three dataset. Resulting in multiple models able to synthesised new labelled data. An initial analysis was made to evaluate the variety and quality of the generated data.

The initial visual analysis of the synthesised data reveals that, visually, there is no difference between the synthesised data from the three different dataset. Additionally, the GANs did not enter into a complete model collapse state as the trained GANs are able to generate different variation for each label. Figure 30 illustrates synthesised data samples where the object in each row belong to the same class. In each row, the first three objects are the synthetic and the last two are original objects.

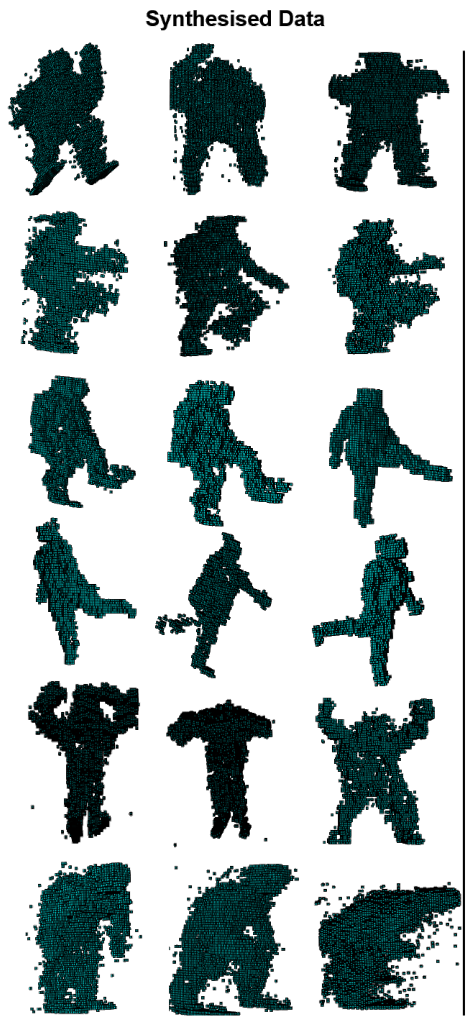


Figure 30: Real and Synthethic data sample

The synthesised objects are similar, but not identical, to the original samples. Although these imperfections, the synthesised 3D objects show that empirically the generator is able to represent the distribution of complex 3D models.

5.2 Augmentation Size

Once GANs are trained, there is a complete control over the number of instances to synthesise. Although, an unlimited number of synthetic samples can be added to the original set, the number of samples to add should be considered. Augment the dataset with excessive synthetic samples saturates the classifiers with similar information and increases the chances of training a model that does not generalise well. Contrarily, augmenting a dataset with few synthetic instances do not employ the potential of the augmentation. In this project, the augmentation size impact is evaluated by comparing the accuracy of the proposed classifiers trained with multiple augmentation schemes.

The experiment tries five different augmentation sizes for each dataset. The original datasets are augmented with synthetic data in a proportional number of the size of the original set, with 10%, 20%, 30%, 40%, and 50% as used proportions. The number of synthetic instances per each class added to the original set was considered to keep the proportion of classes as in the original dataset. Table 1 shows the accuracy results of the data augmentation in each of the three different dataset for each of the proposed augmentation sizes. The average accuracy and the standard are reported as the experiment is repeated several times with the cross validation evaluation method.

Percentage Augmented	**10%**	**20%**	**30%**	**40%**	**50%**
Average Accuracy in Small Dataset	0.759	0.759	0.791	0.797	0.794
Standard Deviationof Accuracy between experiments Small Set	0.015	0.026	0.021	0.01	0.019
Average Accuracy in Medium Dataset	0.834	0.829	0.858	0.861	0.858
Standard Deviation Accuracy between experiments Medium Set	0.012	0.008	0.009	0.007	0.009
Average Accuracy in Full Dataset	0.897	0.9	0.909	0.916	0.913
Standard Deviation of Accuracy between experiments in Full Set	0.002	0.005	0.007	0.005	0.008

Table 1: Accuracy of 3D classifier with multiple augmented sets

In all the datasets, the accuracy improved as the number of synthetics samples used for augmentation increased, up to an augmentation proportion of 40% of the size of the original set. Above this proportion, when more synthesised data is added, the augmentation fails to improve the accuracy. The classifier accuracy decreases because the synthetic data cannot provide more meaningful information and the classifier stars to be feed up with similar information. Proving that the number of synthetic samples added to the original set has an impact on the performance, this research uses the best augmentation policies for further comparisons. Figure 31 illustrates the increase of performance with the different augmentation strategies in each different data sets.

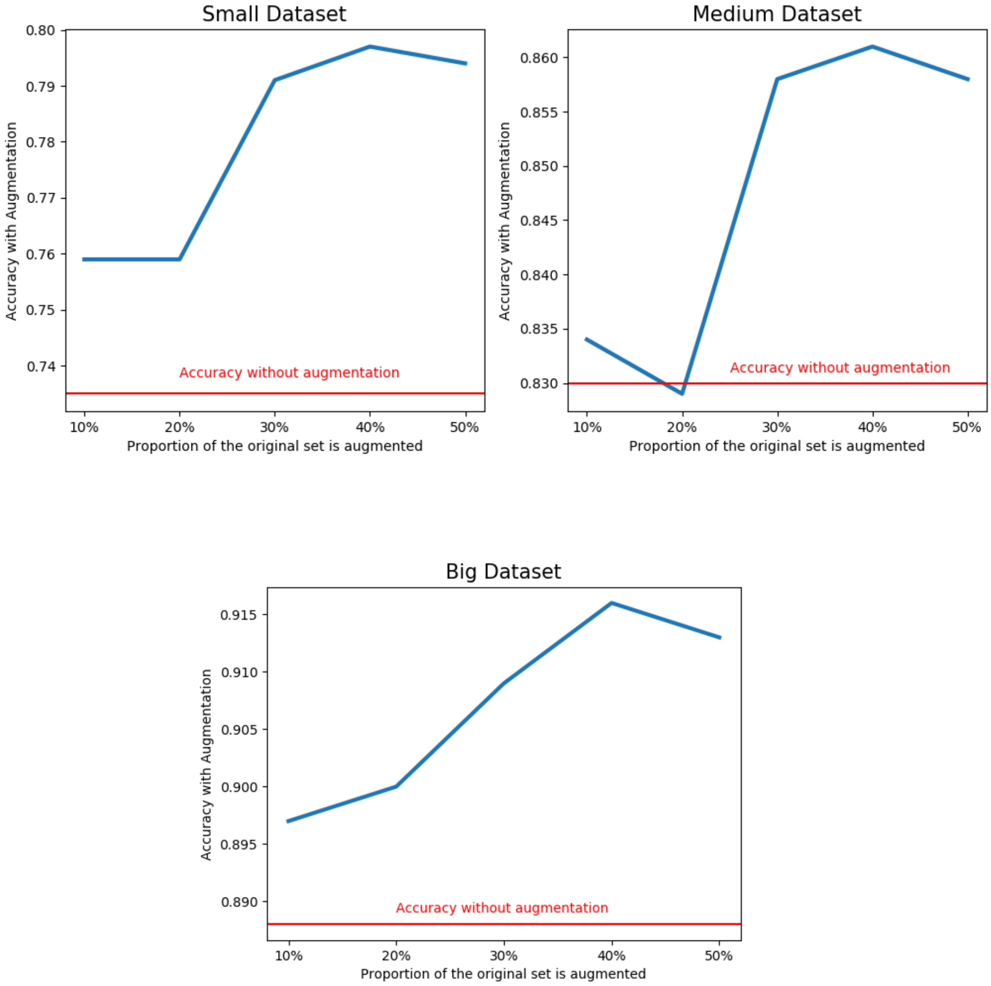


Figure 31: Accuracy fluctuation among the proposed augmentation strategies

5.3 Overall Results

The potential of the proposed augmentation strategy to improve the classification of 3D data is tested by comparing the performance of a 3D classifier trained with the original and augmented datasets. The impact of the dataset size on the performance of the augmentation methodology is tested by comparing the performance fluctuation due to the augmentation across the proposed sets. Table 2 shows the average accuracy and the standard deviation results of the multiple repetitions of the 3D classifier trained with the original and the augmented datasets in each of different dataset. The table, also, reports the fluctuation of performance between the augmented and baseline classifiers.

	Small Dataset	Medium Dataset	Full Dataset
Average Accuracy in Standard Classifier	0.735	0.83	0.888
Standard Deviation StandardClassifier	0.027	0.006	0.006
Average Accuracy in Augmented Classifier	0.797	0.861	0.916
Standard Deviation Augmented Classifier	0.01	0.007	0.005
% Increase of Accuracy with Augmentation	**+8.43%**	**+3.73%**	**+3.15%**

Table 2: Performance comparison between augmented classifiers and non augmented

The proposed augmentation improves the classification performance in all the datasets, increasing the classification accuracy in the big dataset by 3.15% (88.8% to 91.6%) and in the medium by 3.75% (83% to 86.1%). The proposed augmentation methodology performs particularly well in small datasets, the accuracy in the small set increased by 8.43% (73.5% to 79.7%). Figure 32 illustrates the evolution the performance across the augmented an non augmented sets.

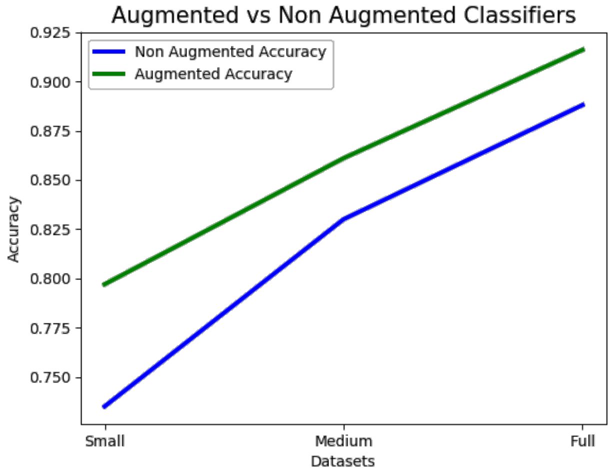


Figure 32: Augmented Classifiers vs Standard Classifiers

The results of the augmentation process confirms that, in overall, GANs can generate data from multiple distributions and use these representation to improve the performance of 3D classifiers. However, not all the distribution are equally easy to reproduce and some classes might be wrongly represented. This research evaluates whether GANs are able to represent all the distributions by comparing the classification performance of each single class between the augmented and normal classifier across the different datasets. Then, identify the wrongly represented classes to improve the model performance a posteriori.

The percentage of instances that the 3D classifier correctly classifies per class is retrieved from the normalised confusion matrices of each model. Then, the class performance is compared between the augmented and original datasets in all three dataset variants. Appendix B shows the normalised confusion matrices for each model. Tables 3, 4, and 5 show the percentage of instances properly identified in each class for the augmented and non augmented classifiers. The tables also provides information about the fluctuation in single class accuracy between augmented a non augmented cases.

Label	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Label Accuracy Small Classifier	0.57	1.00	0.57	0.67	0.85	0.75	0.88	0.63	0.78	0.70	0.50	0.67	0.88	0.78
Label Accuracy Small Augmented Classifier	0.57	1.00	0.71	0.67	0.92	0.67	0.88	0.88	0.56	0.70	0.71	0.73	1.00	0.78
Small dataset Increase label Accuracy	0.00	0.00	0.25	0.00	0.09	-0.11	0.00	0.40	-0.29	0.00	0.43	0.09	0.14	0.00

Table 3: Small Dataset Label Accuracy comparison

Label	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Label Accuracy Medium Classifier	0.73	0.97	1.00	0.97	0.88	0.56	0.77	0.83	0.86	0.71	0.61	0.74	0.92	0.89
Label Accuracy Medium Augmented Classifier	0.82	0.97	1.00	0.97	1.00	0.60	0.82	1.00	0.77	0.75	0.75	0.85	0.96	0.93
Medium dataset Increase label Accuracy	0.13	0.00	0.00	0.00	0.14	0.07	0.06	0.21	-0.11	0.05	0.23	0.15	0.04	0.04

Table 4: Medium Dataset Label Accuracy comparison

Label	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Label Accuracy Full Classifier	0.86	0.86	0.97	0.86	0.97	0.86	0.95	0.89	0.93	0.85	0.71	0.87	0.98	0.87
Label Accuracy Full Augmented Classifier	0.97	0.97	0.94	0.91	0.94	0.85	0.95	0.97	0.93	0.96	0.85	0.91	0.98	0.85
Full dataset Increase Accuracy	0.13	0.12	-0.03	0.06	-0.03	0.00	0.00	0.09	0.00	0.12	0.21	0.05	0.00	-0.02

Table 5: Full Dataset Label Accuracy comparison

In overall, the classification performance of the individual labels increases with the augmentation process showing the ability of GANs to model the distributions of multiple situations and synthesise non-seen data from those distributions. There are classes such as 3 (moving hips), 6 (shake arm), and 13 (jiggling on toes) whose classification performance remains invariant after the augmentation. Contrarily, the classification performance of class 8 (shake hips) decreases as a result of the augmentation. This shows that, certainly, there are distributions that are harder to learn and either GANs cannot model those distributions or require a different configuration of parameters to learn. One simple solution is to avoid using GANs to synthesise labelled data for those classes that GANs cannot learn their distribution. However, knowing a priori that GANs cannot generate a distribution properly is a complicated task as there is not a good measure to evaluate the fidelity of the GANs learned distributions [20]

Overall, the results confirm that the proposed classifier detects actions using 3D objects even when the number of data available is limited. The presented augmentation strategy improves the classification performance of classifiers trained on datasets of all sizes, particularly, small dataset. The efficacy of the augmentation strategy depends on the number of synthetic samples added to the original set. Finally, GANs are not always able to represent all the distributions within a dataset which reduces the performance of the augmentation strategy if wrong representations are included in the training set.

6 Conclusion

This research proposes a method to synthesise 3D human representations using 3D generative adversarial networks to improve the performance of deep learning models for the classification of 3D images. The research outlines that structural modifications of the original 3D GANs structure improves the generation quality of complex 3D data distributions. The proposed 3D GANs learns the distributions within a dataset to introduce novel labelled objects into the training set of a deep learning classifier. This results in improvements in classification performance in all kinds of datasets, particularly, in low-data settings. However beyond that, this research shows the limitations of GANs to produce a variety of non-seen information. Consequently, the number of synthetic instances used for augmentation should be considered. Furthermore, this research identifies that GANs can learn incorrectly some distributions withing a dataset which leads to a performance decrease of the augmentation. Finally, the good results obtained by the proposed 3D classifiers and the ability of 3D based GANs to learn 3D data distributions for its generation confirms the suitability of 3D data to represent information.

6.1 Further Research

Future work should be focused on adapting 2D based GANs structures into 3D based GANs. 2D based GANs structures able to perform domain transfer such as cycle GANs or pix2pix GANs have shown good results on augmenting small datasets. Another research path lies on comparing the performance of GANs based on the generation of voxels and GANs based on the generation of point clouds for the augmentation of 3D datasets. Additionally, the development of methods to augment datasets only with meaningful synthesised samples or a method to evaluate GANs learned distributions might have a positive impact on the augmentation process. Finally, the proposed strategy needs to be evaluated in other domains where the implementation of 3D data will lead to improvements. The suggested methodology is potentially suitable for domains such as human robot interaction to improve agents perception of the real world or in medicine to boost the detection of diseases with 3D data provided by scans, body sensors, and wearables.

References

[1] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, *et al.*, "Shapenet: An information-rich 3d model repository," *arXiv preprint arXiv:1512.03012*, 2015.

[2] M.-D. Yang, C.-F. Chao, K.-S. Huang, L.-Y. Lu, and Y.-P. Chen, "Image-based 3d scene reconstruction and exploration in augmented reality," *Automation in Construction*, vol. 33, pp. 48–60, 2013.

[3] F. Argelaguet and C. Andujar, "A survey of 3d object selection techniques for virtual environments," *Computers & Graphics*, vol. 37, no. 3, pp. 121–136, 2013.

[4] A. Payan and G. Montana, "Predicting alzheimer's disease: a neuroimaging study with 3d convolutional neural networks," *arXiv preprint arXiv:1502.02506*, 2015.

[5] L.-Y. Gui, K. Zhang, Y.-X. Wang, X. Liang, J. M. Moura, and M. Veloso, "Teaching robots to predict human motion," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 562–567, IEEE, 2018.

[6] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 922–928, IEEE, 2015.

[7] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 652–660, 2017.

[8] A. Antoniou, A. Storkey, and H. Edwards, "Data augmentation generative adversarial networks," *arXiv preprint arXiv:1711.04340*, 2017.

[9] M. Frid-Adar, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan, "Synthetic data augmentation using gan for improved liver lesion classification," in *2018 IEEE 15th international symposium on biomedical imaging (ISBI 2018)*, pp. 289–293, IEEE, 2018.

[10] H.-C. Shin, N. A. Tenenholtz, J. K. Rogers, C. G. Schwarz, M. L. Senjem, J. L. Gunter, K. P. Andriole, and M. Michalski, "Medical image synthesis for data augmentation and anonymization using generative adversarial networks," in *International Workshop on Simulation and Synthesis in Medical Imaging*, pp. 1–11, Springer, 2018.

[11] H. Huang, E. Kalogerakis, and B. Marlin, "Analysis and synthesis of 3d shape families via deep-learned generative models of surfaces," in *Computer Graphics Forum*, vol. 34, pp. 25–38, Wiley Online Library, 2015.

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[13] T.-C. Wang, M.-Y. Liu, J.-Y. Zhu, A. Tao, J. Kautz, and B. Catanzaro, "High-resolution image synthesis and semantic manipulation with conditional gans," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8798–8807, 2018.

[14] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 2223–2232, 2017.

[15] H. Zenati, C. S. Foo, B. Lecouat, G. Manek, and V. R. Chandrasekhar, "Efficient gan-based anomaly detection," *arXiv preprint arXiv:1802.06222*, 2018.

[16] J. Donahue, P. Krähenbühl, and T. Darrell, "Adversarial feature learning," *arXiv preprint arXiv:1605.09782*, 2016.

[17] S. Pascual, A. Bonafonte, and J. Serra, "Segan: Speech enhancement generative adversarial network," *arXiv preprint arXiv:1703.09452*, 2017.

[18] W. Fedus, I. Goodfellow, and A. M. Dai, "Maskgan: better text generation via filling in the_," *arXiv preprint arXiv:1801.07736*, 2018.

[19] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann, "Netgan: Generating graphs via random walks," *arXiv preprint arXiv:1803.00816*, 2018.

[20] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in neural information processing systems*, pp. 2234–2242, 2016.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[22] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[23] V. Dumoulin, I. Belghazi, B. Poole, O. Mastropietro, A. Lamb, M. Arjovsky, and A. Courville, "Adversarially learned inference," *arXiv preprint arXiv:1606.00704*, 2016.

[24] P. Achlioptas, O. Diamanti, I. Mitliagkas, and L. Guibas, "Learning representations and generative models for 3d point clouds," *arXiv preprint arXiv:1707.02392*, 2017.

[25] C.-H. Lin, C. Kong, and S. Lucey, "Learning efficient point cloud generation for dense 3d object reconstruction," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[26] G. Chételat, B. Landeau, F. Eustache, F. Mézenge, F. Viader, V. de La Sayette, B. Desgranges, and J.-C. Baron, "Using voxel-based morphometry to map the structural changes associated with rapid conversion in mci: a longitudinal mri study," *Neuroimage*, vol. 27, no. 4, pp. 934–946, 2005.

[27] D. Maturana and S. Scherer, "3d convolutional neural networks for landing zone detection from lidar," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3471–3478, IEEE, 2015.

[28] M. De Deuge, A. Quadros, C. Hung, and B. Douillard, "Unsupervised feature learning for classification of outdoor 3d scans," in *Australasian Conference on Robitics and Automation*, vol. 2, p. 1, 2013.

[29] Y. Fang, J. Xie, G. Dai, M. Wang, F. Zhu, T. Xu, and E. Wong, "3d deep shape descriptor," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2319–2328, 2015.

[30] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1912–1920, 2015.

[31] J. Wu, C. Zhang, T. Xue, B. Freeman, and J. Tenenbaum, "Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling," in *Advances in neural information processing systems*, pp. 82–90, 2016.

[32] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Generative and discriminative voxel modeling with convolutional neural networks," *arXiv preprint arXiv:1608.04236*, 2016.

[33] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 1, pp. 221–231, 2012.

[34] C. R. Qi, H. Su, M. Nießner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view cnns for object classification on 3d data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5648–5656, 2016.

[35] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view convolutional neural networks for 3d shape recognition," in *Proceedings of the IEEE international conference on computer vision*, pp. 945–953, 2015.

[36] V. Hegde and R. Zadeh, "Fusionnet: 3d object classification using multiple data representations," *arXiv preprint arXiv:1607.05695*, 2016.

[37] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[38] J. Masci, D. Boscaini, M. Bronstein, and P. Vandergheynst, "Geodesic convolutional neural networks on riemannian manifolds," in *Proceedings of the IEEE international conference on computer vision workshops*, pp. 37–45, 2015.

[39] R. Klokov and V. Lempitsky, "Escape from cells: Deep kd-networks for the recognition of 3d point cloud models," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 863–872, 2017.

[40] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," in *Advances in neural information processing systems*, pp. 5099–5108, 2017.

[41] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," *arXiv preprint arXiv:1701.00160*, 2016.

[42] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[43] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[44] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey, "Adversarial autoencoders," *arXiv preprint arXiv:1511.05644*, 2015.

[45] C. Finn, P. Christiano, P. Abbeel, and S. Levine, "A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models," *arXiv preprint arXiv:1611.03852*, 2016.

[46] C. Finn and S. Levine, "Deep visual foresight for planning robot motion," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2786–2793, IEEE, 2017.

[47] A. Odena, "Semi-supervised learning with generative adversarial networks," *arXiv preprint arXiv:1606.01583*, 2016.

[48] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in neural information processing systems*, pp. 4565–4573, 2016.

[49] T. Schlegl, P. Seeböck, S. M. Waldstein, U. Schmidt-Erfurth, and G. Langs, "Unsupervised anomaly detection with generative adversarial networks to guide marker discovery," in *International Conference on Information Processing in Medical Imaging*, pp. 146–157, Springer, 2017.

[50] D. Yoo, N. Kim, S. Park, A. S. Paek, and I. S. Kweon, "Pixel-level domain transfer," in *European Conference on Computer Vision*, pp. 517–532, Springer, 2016.

[51] M. Arjovsky and L. Bottou, "Towards Principled Methods for Training Generative Adversarial Networks," *arXiv e-prints*, p. arXiv:1701.04862, Jan 2017.

[52] S. Arora, R. Ge, Y. Liang, T. Ma, and Y. Zhang, "Generalization and equilibrium in generative adversarial nets (gans)," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 224–232, JMLR. org, 2017.

[53] I. O. Tolstikhin, S. Gelly, O. Bousquet, C.-J. Simon-Gabriel, and B. Schölkopf, "Adagan: Boosting generative models," in *Advances in Neural Information Processing Systems*, pp. 5424–5433, 2017.

[54] C. K. Sønderby, J. Caballero, L. Theis, W. Shi, and F. Huszár, "Amortised map inference for image super-resolution," *arXiv preprint arXiv:1610.04490*, 2016.

[55] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," *arXiv preprint arXiv:1701.07875*, 2017.

[56] S. Nowozin, B. Cseke, and R. Tomioka, "f-gan: Training generative neural samplers using variational divergence minimization," in *Advances in neural information processing systems*, pp. 271–279, 2016.

[57] Q. Chen and V. Koltun, "Photographic image synthesis with cascaded refinement networks," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1511–1520, 2017.

[58] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, "Are gans created equal? a large-scale study," in *Advances in neural information processing systems*, pp. 700–709, 2018.

[59] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[60] S. Ioffe, "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models," in *Advances in neural information processing systems*, pp. 1945–1953, 2017.

[61] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[62] Z. Hussain, F. Gimenez, D. Yi, and D. Rubin, "Differential data augmentation techniques for medical imaging classification tasks," in *AMIA Annual Symposium Proceedings*, vol. 2017, p. 979, American Medical Informatics Association, 2017.

[63] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[64] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[65] A. Gupta, S. Venkatesh, S. Chopra, and C. Ledig, "Generative image translation for data augmentation of bone lesion pathology," *arXiv preprint arXiv:1902.02248*, 2019.

[66] C. Han, K. Murao, T. Noguchi, Y. Kawata, F. Uchiyama, L. Rundo, H. Nakayama, and S. Satoh, "Learning more with less: conditional pggan-based data augmentation for brain metastases detection using highly-rough annotation on mr images," *arXiv preprint arXiv:1902.09856*, 2019.

[67] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[68] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," in *Advances in neural information processing systems*, pp. 379–387, 2016.

[69] J. W. Tangelder and R. C. Veltkamp, "A survey of content based 3d shape retrieval methods," in *Proceedings Shape Modeling Applications, 2004.*, pp. 145–156, IEEE, 2004.

[70] C. Li, M. Zeeshan Zia, Q.-H. Tran, X. Yu, G. D. Hager, and M. Chandraker, "Deep supervision with shape concepts for occlusion-aware 3d object parsing," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5465–5474, 2017.

[71] H. Sundar, D. Silver, N. Gagvani, and S. Dickinson, "Skeleton based shape matching and retrieval," in *2003 Shape Modeling International.*, pp. 130–139, IEEE, 2003.

[72] C. B. Choy, D. Xu, J. Gwak, K. Chen, and S. Savarese, "3d-r2n2: A unified approach for single and multi-view 3d object reconstruction," in *European conference on computer vision*, pp. 628–644, Springer, 2016.

[73] R. Girdhar, D. F. Fouhey, M. Rodriguez, and A. Gupta, "Learning a predictable and generative vector representation for objects," in *European Conference on Computer Vision*, pp. 484–499, Springer, 2016.

[74] Y. Zhao, T. Birdal, H. Deng, and F. Tombari, "3d point capsule networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[75] X. Yan, J. Yang, E. Yumer, Y. Guo, and H. Lee, "Perspective transformer nets: Learning single-view 3d object reconstruction without 3d supervision," in *Advances in Neural Information Processing Systems*, pp. 1696–1704, 2016.

[76] D. J. Rezende, S. A. Eslami, S. Mohamed, P. Battaglia, M. Jaderberg, and N. Heess, "Unsupervised learning of 3d structure from images," in *Advances in Neural Information Processing Systems*, pp. 4996–5004, 2016.

[77] Y. Xiang, W. Choi, Y. Lin, and S. Savarese, "Data-driven 3d voxel patterns for object category recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1903–1911, 2015.

[78] A. Sharma, O. Grau, and M. Fritz, "Vconv-dae: Deep volumetric shape learning without object labels," in *European Conference on Computer Vision*, pp. 236–250, Springer, 2016.

[79] E. Smith and D. Meger, "Improved adversarial systems for 3d object generation and reconstruction," *arXiv preprint arXiv:1707.09557*, 2017.

[80] J. Liu, F. Yu, and T. Funkhouser, "Interactive 3d modeling with a generative adversarial network," in *2017 International Conference on 3D Vision (3DV)*, pp. 126–134, IEEE, 2017.

[81] D. Valsesia, G. Fracastoro, and E. Magli, "Learning localized generative models for 3d point clouds via graph convolution," 2018.

[82] D. W. Shu, S. W. Park, and J. Kwon, "3d point cloud generative adversarial network based on tree structured graph convolutions," *arXiv preprint arXiv:1905.06292*, 2019.

[83] F. Bogo, J. Romero, G. Pons-Moll, and M. J. Black, "Dynamic faust: Registering human bodies in motion," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 6233–6242, 2017.

[84] M. W. Jones and R. Satherley, "Shape representation using space filled sub-voxel distance fields," in *Proceedings international conference on shape modeling and applications*, pp. 316–325, IEEE, 2001.

[85] S. Fang and H. Chen, "Hardware accelerated voxelization," *Computers & Graphics*, vol. 24, no. 3, pp. 433–442, 2000.

[86] K. Schindler and L. J. Van Gool, "Action snippets: How many frames does human action recognition require?," in *CVPR*, vol. 1, pp. 3–2, 2008.

[87] A. Fathi and G. Mori, "Action recognition by learning mid-level motion features," in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, IEEE, 2008.

[88] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[89] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[90] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

A Table Number corresponding with the classified actions

Label	Action
0	punching
1	running on spot
2	chicken wings
3	moving hips
4	moving knees
5	jumping jacks
6	shake arms
7	shake shoulders
8	shake hips
9	one leg loose
10	one leg jump
11	soft hop with two legs
12	one leg hop
13	jiggling on toes

Table 6: Label assigned to each action

B Confusion Matrices

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.5714	0	0	0.2857	0	0	0	0.1429	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0.1429	0	0.5714	0	0	0	0	0	0.1429	0	0	0.1429	0	0
3	0	0	0	0.6667	0.1667	0	0	0	0	0	0	0.0833	0.0833	0
4	0	0	0	0	0.8462	0	0	0	0	0.0769	0.0769	0	0	0
5	0	0	0	0	0	0.75	0	0	0	0	0	0.125	0	0.125
6	0	0	0.125	0	0	0	0.875	0	0	0	0	0	0	0
7	0	0.125	0	0.125	0	0	0	0.625	0	0	0	0.125	0	0
8	0	0	0	0	0	0	0.1111	0	0.7778	0	0	0	0.1111	0
9	0	0	0	0	0	0	0	0	0	0.7	0.2	0	0.1	0
10	0	0.1667	0	0	0.0833	0.0833	0	0	0	0.0833	0.5	0	0.0833	0
11	0	0	0	0	0	0	0	0	0	0	0	0.6667	0.3333	0
12	0	0	0	0.125	0	0	0	0	0	0	0	0	0.875	0
13	0	0	0	0.1111	0	0.1111	0	0	0	0	0	0	0	0.7778

Figure 33: Normalised Confusion Matrix of the classifier trained with the small dataset

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.5714	0	0	0.4286	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0.7143	0	0	0	0	0	0.1429	0	0	0.1429	0	0
3	0	0	0	0.6667	0	0	0	0.0833	0	0	0	0.0833	0.1667	0
4	0	0	0	0	0.9231	0	0	0	0	0.0769	0	0	0	0
5	0	0.1111	0	0	0	0.6667	0	0	0	0	0	0.1111	0	0.1111
6	0.125	0	0	0	0	0	0.875	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0.875	0	0	0	0.125	0	0
8	0.1111	0	0	0.1111	0	0	0.1111	0	0.5556	0	0	0	0.1111	0
9	0	0	0	0	0	0	0	0	0	0.7	0.2	0	0.1	0
10	0	0.0714	0	0.0714	0.0714	0	0	0	0	0.0714	0.7143	0	0	0
11	0	0	0	0	0	0	0	0.0909	0	0	0.0909	0.7273	0.0909	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0.1111	0	0.1111	0	0	0	0	0	0	0	0.7778

Figure 34: Normalised Confusion Matrix of the classifier trained with the augmented small dataset

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.7273	0.0455	0	0	0	0	0.0455	0	0.0909	0	0	0.0909	0	0
1	0	0.9714	0	0	0	0	0	0	0	0	0.0286	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0.9714	0	0	0	0.0286	0	0	0	0	0	0
4	0	0	0	0	0.875	0.025	0	0	0	0.075	0.025	0	0	0
5	0	0	0	0	0.08	0.56	0	0	0	0.04	0.16	0.04	0	0.12
6	0.0455	0.0455	0	0	0	0	0.7727	0	0.0455	0	0	0	0.0455	0.0455
7	0.0435	0	0	0.0435	0	0	0	0.8261	0	0	0	0.0435	0.0435	0
8	0.0357	0	0	0.0357	0	0	0.0357	0	0.8571	0	0	0	0	0.0357
9	0	0	0	0	0.0714	0	0	0	0	0.7143	0.2143	0	0	0
10	0	0.0278	0	0	0.1389	0	0	0	0.0278	0.1389	0.6111	0.0556	0	0
11	0	0	0	0	0	0	0	0	0	0	0.037	0.7407	0.2222	0
12	0	0	0	0.04	0	0	0	0	0	0	0	0.04	0.92	0
13	0	0	0	0	0	0	0	0	0.0357	0	0	0	0.0714	0.8929

Figure 35: Normalised Confusion Matrix of the classifier trained with the medium dataset

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.8182	0	0	0	0	0	0.0455	0.0455	0.0455	0	0	0.0455	0	0
1	0.0286	0.9714	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0.9714	0	0	0	0.0286	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0.04	0.6	0	0	0.04	0.04	0.16	0	0	0.12
6	0.0455	0	0	0	0.0455	0	0.8182	0.0455	0	0	0	0	0	0.0455
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0.0667	0	0.0667	0.1	0	0.7667	0	0	0	0	0
9	0	0	0	0	0.1429	0	0	0	0	0.75	0.1071	0	0	0
10	0	0.0556	0	0	0.0556	0	0	0	0	0.0833	0.75	0.0556	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0.8519	0.1481	0
12	0	0	0	0	0	0	0	0	0	0	0	0.04	0.96	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0.0714	0.9286

Figure 36: Normalised Confusion Matrix of the classifier trained with the augmented medium dataset

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.8611	0	0	0.0278	0	0	0.0278	0.0278	0.0278	0	0	0	0.0278	0
1	0.0172	0.8621	0	0	0.0345	0	0.0172	0	0	0.0172	0.0517	0	0	0
2	0	0	0.9714	0	0	0	0	0	0.0286	0	0	0	0	0
3	0	0.0172	0	0.8621	0.0517	0	0	0	0	0	0	0.0345	0.0345	0
4	0	0.0149	0	0.0149	0.9701	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0.0476	0.8571	0	0	0.0476	0	0.0238	0	0	0.0238
6	0	0	0.0263	0	0	0	0.9474	0	0	0	0	0	0	0.0263
7	0.0263	0	0	0.0263	0	0	0	0.8947	0	0	0	0	0.0526	0
8	0.0217	0	0	0	0	0	0.0217	0.0217	0.9348	0	0	0	0	0
9	0	0	0	0	0.0851	0	0	0	0	0.8511	0.0426	0.0213	0	0
10	0	0.0517	0	0	0.069	0	0	0	0	0.1207	0.7069	0.0517	0	0
11	0.0222	0	0	0.0222	0	0	0	0.0222	0	0	0.0222	0.8667	0.0444	0
12	0	0	0	0.0238	0	0	0	0	0	0	0	0	0.9762	0
13	0	0	0	0.0213	0.0213	0.0213	0.0213	0	0.0426	0	0	0	0	0.8723

Figure 37: Normalised Confusion Matrix of the classifier trained with the big dataset

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.971	0.000	0.029	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1	0.000	0.966	0.000	0.000	0.034	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	0.000	0.000	0.943	0.000	0.000	0.000	0.000	0.057	0.000	0.000	0.000	0.000	0.000	0.000
3	0.000	0.017	0.000	0.914	0.017	0.000	0.000	0.000	0.000	0.000	0.000	0.052	0.000	0.000
4	0.000	0.000	0.000	0.000	0.940	0.000	0.000	0.015	0.000	0.015	0.030	0.000	0.000	0.000
5	0.000	0.024	0.000	0.000	0.000	0.854	0.000	0.000	0.073	0.000	0.024	0.000	0.000	0.024
6	0.000	0.000	0.026	0.000	0.000	0.000	0.947	0.000	0.000	0.000	0.000	0.000	0.000	0.026
7	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.974	0.000	0.000	0.000	0.000	0.026	0.000
8	0.043	0.000	0.000	0.000	0.000	0.000	0.022	0.000	0.935	0.000	0.000	0.000	0.000	0.000
9	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.956	0.044	0.000	0.000	0.000
10	0.000	0.016	0.000	0.016	0.016	0.000	0.000	0.000	0.000	0.066	0.852	0.016	0.016	0.000
11	0.022	0.000	0.000	0.000	0.000	0.000	0.000	0.022	0.000	0.000	0.022	0.911	0.022	0.000
12	0.000	0.000	0.000	0.024	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.976	0.000
13	0.000	0.000	0.000	0.021	0.000	0.043	0.021	0.000	0.043	0.000	0.021	0.000	0.000	0.851

Figure 38: Normalised Confusion Matrix of the classifier trained with the augmented big dataset

C Code

```
#import packages we are suing for the experiment
import tensorflow as tf
import numpy as np
import scipy.io as io
from scipy.io import loadmat
#import skimage.measure as sk
import os
import sys
import h5py
import numpy as np
import scipy.io as io

import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from tqdm import *


def get_list_elements_without_pattern_not_current_directory(directory_to_search):
    #Comprehension list that by given a directory, explores all the elements
    files= [element for element in os.listdir(directory_to_search)]
    # return a list
```

```
    return files

#function to create an array of an specific label
def create_array_labels(label,number_instances):
    #
    label_array=np.full((number_instances),label)
    #
    return label_array

def saveFromVoxels(voxels, path):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.voxels(voxels, facecolors='b', edgecolor='k')
    fig.savefig(path)
    plt.close()
    plt.clf()
    plt.cla()
    del fig

def create_folder_in_path_check_folder_created(path_creation, path_to_create):
    #get a list of the elements that are in the directory we want to create
    directories_in_directory_where_eant_create=\
    get_list_elements_without_pattern_not_current_directory(path_creation)
    #Get the paths of the elements in the directory  where we want to create a
    #new directory
    directories_path_in_directory_where_eant_create=\
    [path_creation+'/'+path for path in \
    directories_in_directory_where_eant_create]
    #If the directory we want to create has not been created before, create one
    if path_to_create not in directories_path_in_directory_where_eant_create:
        #create the directory
        os.mkdir(path_to_create)

def double_voxels_dimension(voxel_array_to_transform):
    number_voxels=voxel_array_to_transform.shape[0]
    #Get the number of voxels per dimension
    one_dimension_voxels=voxel_array_to_transform.shape[1]
    double_dimension=one_dimension_voxels*2
    print (double_dimension)
    #create an array to store the transformed instances
    voxel_array_transformed=np.zeros((number_voxels,double_dimension,\
    double_dimension,double_dimension,1))

    #loop through all the instances in the array we want to transfrom
    for voxel_tranformation_index in range(number_voxels):
        #get the array that we want to transform
        voxel_to_transform=voxel_array_to_transform[
```

```
    voxel_tranformation_index]
        #modify the voxels
        voxel_to_transform=np.pad(voxel_to_transform,(0, 0),\
        'constant',constant_values=(0, 0))
        #Square voxels
        voxel_to_transform=nd.zoom(voxel_to_transform,\
        (2, 2, 2), mode='constant', order=0)\
        .reshape((double_dimension,double_dimension,double_dimension,1))
        #Add the transformed voxel to the array that contains the
    transformed
        #voxels
        voxel_array_transformed[voxel_tranformation_index]=\
        voxel_to_transform
    #return the modified array
    return voxel_array_transformed

#Fucntion to reduce the size of a set of voxels
def half_voxels_dimension(voxel_array_to_transform):
    number_voxels=voxel_array_to_transform.shape[0]
    #Get the number of voxels per dimension
    one_dimension_voxels=voxel_array_to_transform.shape[1]
    half_dimension=int(one_dimension_voxels/2)

    #create an array to store the transformed instances
    voxel_array_transformed=np.zeros((number_voxels,half_dimension,\
    half_dimension,half_dimension,1))
    #loop through all the instances in the array we want to transfrom
    for voxel_tranformation_index in range(number_voxels):
        #get the array that we want to transform
        voxel_to_transform=voxel_array_to_transform[
    voxel_tranformation_index]
        #modify the voxels
        voxel_to_transform=np.pad(voxel_to_transform,(0, 0),\
        'constant',constant_values=(0, 0))
        #Square voxels
        voxel_to_transform=nd.zoom(voxel_to_transform,\
        (0.5, 0.5, 0.5), mode='constant', order=0)\
        .reshape((half_dimension,half_dimension,half_dimension,1))
        #Add the transformed voxel to the array that contains the
    transformed
        #voxels
        voxel_array_transformed[voxel_tranformation_index]=\
        voxel_to_transform
    #return the modified array
    return voxel_array_transformed

#
    ######################################################################################

#The below function were taken from the github post:
#https://github.com/meetshah1995/tf-3dgan

#Tensoflow function to create a set of weight for our models
```

```
def init_weights(shape, name):
    return tf.get_variable(name, shape=shape,\
    initializer=tf.contrib.layers.xavier_initializer())


#Fucntion to create biases for the generator and discriminator
def init_biases(shape):
    return tf.Variable(tf.zeros(shape))


#Batch normalisation layer with tensorflow
def batchNorm(x, n_out, phase_train,scope='bn'):
    with tf.variable_scope(scope):
        #BEta parameter in batch normalisation
        beta = tf.Variable(tf.constant(0.0, shape=[n_out]),name='beta',\
                                   trainable=True)
        #Gamma parameter
        gamma = tf.Variable(tf.constant(1.0, shape=[n_out]),name='gamma',\
                                    trainable=True)
        #Ema
        batch_mean, batch_var = tf.nn.moments(x, [0,1,2], name='moments')
        ema = tf.train.ExponentialMovingAverage(decay=0.5)
        #function to calculate the mean value of the update
        def mean_var_with_update():
            ema_apply_op = ema.apply([batch_mean, batch_var])
            with tf.control_dependencies([ema_apply_op]):
                return tf.identity(batch_mean), tf.identity(batch_var)
        #
        mean, var = tf.cond(phase_train,
                                    mean_var_with_update,
                                    lambda: (ema.average(batch_mean),\
                                             ema.average(batch_var)))

        #Otput of as a result of the ooutput normalisation
        normed = tf.nn.batch_normalization(x, mean, var, beta, gamma, 1e-3)

    return normed

#Batch normalisation funtion
class batch_norm(object):
  def __init__(self, epsilon=1e-5, momentum = 0.9, name="batch_norm"):
    with tf.variable_scope(name):
      self.epsilon  = epsilon
      self.momentum = momentum
      self.name = name

  def __call__(self, x, train=True):
    return tf.contrib.layers.batch_norm(x,decay=self.momentum,
                      updates_collections=None,
                      epsilon=self.epsilon,
                      scale=True,
```

```
                      is_training=train,
                      scope=self.name)

#Function to create a threeshold for the discriminator
def threshold(x, val=0.5):
    x = tf.clip_by_value(x,0.5,0.5001) - 0.5
    x = tf.minimum(x * 10000,1)
    return x

#LEaky relu layer
def lrelu(x, leak=0.2):
    return tf.maximum(x, leak*x)


# def lrelu(x, leak=0.2):
#      f1 = 0.5 * (1 + leak)
#      f2 = 0.5 * (1 - leak)
#      return f1 * x + f2 * abs(x)

'''
#######################       Global Parameters       #########################
'''
n_epochs    = 6000
batch_size = 64
g_lr        = 0.0025
d_lr        = 0.00005
beta        = 0.5
d_thresh    = 0.8
z_size      = 200
leak_value = 0.2
cube_len    = 64


#fucntion to create a generator
weights = {}
def generator(z, batch_size=batch_size, phase_train=True, reuse=False):

    strides    = [1,2,2,2,1]

    with tf.variable_scope("gen", reuse=reuse):
        z = tf.reshape(z, (batch_size, 1, 1, 1, z_size))
        g_1 = tf.nn.conv3d_transpose(z, weights['wg1'], (batch_size,4,4,4,512),\
                                    strides=[1,1,1,1,1], padding="VALID")
        g_1 = tf.contrib.layers.batch_norm(g_1, is_training=phase_train)
        g_1 = tf.nn.relu(g_1)

        g_2 = tf.nn.conv3d_transpose(g_1, weights['wg2'], (batch_size,8,8,8,256),\
                                    strides=strides, padding="SAME")
        g_2 = tf.contrib.layers.batch_norm(g_2, is_training=phase_train)
```

```
        g_2 = tf.nn.relu(g_2)

        g_3 = tf.nn.conv3d_transpose(g_2, weights['wg3'], (batch_size,16,16,16,128),\
                                     strides=strides, padding="SAME")
        g_3 = tf.contrib.layers.batch_norm(g_3, is_training=phase_train)
        g_3 = tf.nn.relu(g_3)

        g_4 = tf.nn.conv3d_transpose(g_3, weights['wg4'], (batch_size,32,32,32,64),\
                                     strides=strides, padding="SAME")
        g_4 = tf.contrib.layers.batch_norm(g_4, is_training=phase_train)
        g_4 = tf.nn.relu(g_4)

        g_5 = tf.nn.conv3d_transpose(g_4, weights['wg5'], (batch_size,64,64,64,1),\
                                     strides=strides, padding="SAME")
        #Choose between an sigmoid or tangh activation function. I got the best
        #the best result with the sigmoid which outputs values between 1 and 0
        g_5 = tf.nn.sigmoid(g_5)
        #g_5 = tf.nn.tanh(g_5)

    #print statements
    print(g_1, 'g1')
    print(g_2, 'g2')
    print(g_3, 'g3')
    print(g_4, 'g4')
    print (g_5, 'g5')

    return g_5

#function to generate the discriminator
def discriminator(inputs, phase_train=True, reuse=False):
    #strides that the discriminator will use
    strides    = [1,2,2,2,1]
    #Piece of code to add the multiple layers
    with tf.variable_scope("dis", reuse=reuse):
        d_1 = tf.nn.conv3d(inputs, weights['wd1'], strides=strides, padding="SAME")
        d_1 = tf.contrib.layers.batch_norm(d_1, is_training=phase_train)
        d_1 = lrelu(d_1, leak_value)

        d_2 = tf.nn.conv3d(d_1, weights['wd2'], strides=strides, padding="SAME")
        d_2 = tf.contrib.layers.batch_norm(d_2, is_training=phase_train)
        d_2 = lrelu(d_2, leak_value)

        d_3 = tf.nn.conv3d(d_2, weights['wd3'], strides=strides, padding="SAME")
        d_3 = tf.contrib.layers.batch_norm(d_3, is_training=phase_train)
        d_3 = lrelu(d_3, leak_value)
```

```

        d_4 = tf.nn.conv3d(d_3, weights['wd4'], strides=strides, padding="SAME")
        d_4 = tf.contrib.layers.batch_norm(d_4, is_training=phase_train)
        d_4 = lrelu(d_4)

        d_5 = tf.nn.conv3d(d_4, weights['wd5'], strides=[1,1,1,1,1], padding="VALID")
        d_5_no_sigmoid = d_5
        d_5 = tf.nn.sigmoid(d_5)
    #print statements
    print(d_1, 'd1')
    print(d_2, 'd2')
    print(d_3, 'd3')
    print(d_4, 'd4')
    print(d_5, 'd5')

    return d_5, d_5_no_sigmoid


#Function to set up the initialization weights
def initialiseWeights():

    global weights
    xavier_init = tf.contrib.layers.xavier_initializer()

    weights['wg1'] = tf.get_variable("wg1", shape=[4, 4, 4, 512, 200], initializer=xavier_init)
    weights['wg2'] = tf.get_variable("wg2", shape=[4, 4, 4, 256, 512], initializer=xavier_init)
    weights['wg3'] = tf.get_variable("wg3", shape=[4, 4, 4, 128, 256], initializer=xavier_init)
    weights['wg4'] = tf.get_variable("wg4", shape=[4, 4, 4, 64, 128], initializer=xavier_init)
    weights['wg5'] = tf.get_variable("wg5", shape=[4, 4, 4, 1, 64], initializer=xavier_init)

    weights['wd1'] = tf.get_variable("wd1", shape=[4, 4, 4, 1, 64], initializer=xavier_init)
    weights['wd2'] = tf.get_variable("wd2", shape=[4, 4, 4, 64, 128], initializer=xavier_init)
    weights['wd3'] = tf.get_variable("wd3", shape=[4, 4, 4, 128, 256], initializer=xavier_init)
    weights['wd4'] = tf.get_variable("wd4", shape=[4, 4, 4, 256, 512], initializer=xavier_init)
    weights['wd5'] = tf.get_variable("wd5", shape=[4, 4, 4, 512, 1], initializer=xavier_init)

    return weights

#Function to ensemble the gan and train it
def trainGAN(dataset, path, label, is_dummy=False, checkpoint=None):
  #'/content/drive/My Drive/working/merged_data/analysis_gans_final/full
```

```
dataset/3/biasfree_350.cptk'):

    weights =  initialiseWeights()

    z_vector = tf.placeholder(shape=[batch_size,z_size],dtype=tf.float32)
    x_vector = tf.placeholder(shape=[batch_size,cube_len,cube_len,
cube_len,1],\
                                    dtype=tf.float32)

    net_g_train = generator(z_vector, phase_train=True, reuse=False)

    d_output_x, d_no_sigmoid_output_x = discriminator(x_vector,
phase_train=True,\
                                                         reuse=False)
    d_output_x = tf.maximum(tf.minimum(d_output_x, 0.99), 0.01)
    summary_d_x_hist = tf.summary.histogram("d_prob_x", d_output_x)

    d_output_z, d_no_sigmoid_output_z = discriminator(net_g_train,
phase_train=True, reuse=True)
    d_output_z = tf.maximum(tf.minimum(d_output_z, 0.99), 0.01)
    summary_d_z_hist = tf.summary.histogram("d_prob_z", d_output_z)

    # Compute the discriminator accuracy
    n_p_x = tf.reduce_sum(tf.cast(d_output_x > 0.5, tf.int32))
    n_p_z = tf.reduce_sum(tf.cast(d_output_z < 0.5, tf.int32))
    d_acc = tf.divide(n_p_x + n_p_z, 2 * batch_size)

    # Compute the discriminator and generator loss
    # d_loss = -tf.reduce_mean(tf.log(d_output_x) + tf.log(1-d_output_z))
    # g_loss = -tf.reduce_mean(tf.log(d_output_z))

    d_loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=
d_no_sigmoid_output_x,\
                                                       labels=tf.ones_like(
d_output_x))
    d_loss += tf.nn.sigmoid_cross_entropy_with_logits(logits=
d_no_sigmoid_output_z,\
                                                        labels=tf.
zeros_like(d_output_z))
    g_loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=
d_no_sigmoid_output_z,\
                                                       labels=tf.ones_like(
d_output_z))

    d_loss = tf.reduce_mean(d_loss)
    g_loss = tf.reduce_mean(g_loss)

    summary_d_loss = tf.summary.scalar("d_loss", d_loss)
    summary_g_loss = tf.summary.scalar("g_loss", g_loss)
    summary_n_p_z = tf.summary.scalar("n_p_z", n_p_z)
    summary_n_p_x = tf.summary.scalar("n_p_x", n_p_x)
    summary_d_acc = tf.summary.scalar("d_acc", d_acc)
```

```
    net_g_test = generator(z_vector, phase_train=False, reuse=True)

    para_g = [var for var in tf.trainable_variables() if any(x in var.name for x in ['wg', 'bg', 'gen'])]
    para_d = [var for var in tf.trainable_variables() if any(x in var.name for x in ['wd', 'bd', 'dis'])]

    # only update the weights for the discriminator network
    optimizer_op_d = tf.train.AdamOptimizer(learning_rate=d_lr,beta1=beta).minimize(d_loss,var_list=para_d)
    # only update the weights for the generator network
    optimizer_op_g = tf.train.AdamOptimizer(learning_rate=g_lr,beta1=beta).minimize(g_loss,var_list=para_g)

    saver = tf.train.Saver()

    with tf.Session() as sess:

        sess.run(tf.global_variables_initializer())
        #Load checkpoints in case we need to retrain our model
        if checkpoint is not None:
            saver.restore(sess, checkpoint)

        if is_dummy:
            volumes = np.random.randint(0,2,(batch_size,cube_len,cube_len,cube_len))
            print('Using Dummy Data')
        else:
          volumes = dataset.astype(np.float)
          print ('using own data')
        # volumes *= 2.0
        # volumes -= 1.0

        #Lists to keep track of the loss fucntions and accuracies of the generator
        #and discriminator
        loss_function_generator=[]
        loss_function_discriminator=[]
        discriminator_accuracy=[]


        for epoch in range(n_epochs):

            idx = np.random.randint(len(volumes), size=batch_size)
            x = volumes[idx]
            z_sample = np.random.normal(0, 0.33, size=[batch_size, z_size]).astype(np.float32)
            z = np.random.normal(0, 0.33, size=[batch_size, z_size]).astype(np.float32)
            # z = np.random.uniform(0, 1, size=[batch_size, z_size]).astype(np.float32)

            # Update the discriminator and generator
```

```
            d_summary_merge = tf.summary.merge([summary_d_loss,
                                                summary_d_x_hist,
                                                summary_d_z_hist,
                                                summary_n_p_x,
                                                summary_n_p_z,
                                                summary_d_acc])

            summary_d, discriminator_loss = sess.run([d_summary_merge,d_loss],feed_dict={z_vector:z, x_vector:x})
            summary_g, generator_loss = sess.run([summary_g_loss,g_loss],feed_dict={z_vector:z})
            d_accuracy, n_x, n_z, d_x,d_z = sess.run([d_acc, n_p_x, n_p_z,d_output_x,d_output_z],feed_dict={z_vector:z, x_vector:x})

            #print("nx_nz:",n_x, n_z, "\nd_x:",d_x.reshape(batch_size), "d_z:",d_z.reshape(batch_size))

            print ("nx",n_x,"nz",n_z)
            if d_accuracy < d_thresh:
                sess.run([optimizer_op_d],feed_dict={z_vector:z, x_vector:x})
                print('Discriminator Training ', "epoch: ",epoch,', d_loss:',discriminator_loss,'g_loss:',generator_loss, "d_acc: ", d_accuracy)

            sess.run([optimizer_op_g],feed_dict={z_vector:z})

            #Append values to the lists that keep track of the results
            loss_function_generator.append(generator_loss)
            loss_function_discriminator.append(discriminator_loss)
            discriminator_accuracy.append(d_accuracy)

            print('Generator Training ', "epoch: ",epoch,', d_loss:',discriminator_loss,'g_loss:',generator_loss, "d_acc: ", d_accuracy)


###########  END OF THE CODE retrieve from https://github.com/meetshah1995/tf-3dgan #################
            #Print generations and store generated data
            if epoch % 50 == 0:
                #generate data
                voxel_volumes= sess.run(net_g_test,feed_dict={z_vector:z_sample})
                z = np.random.normal(0, 0.33, size=[batch_size, z_size]).astype(np.float32)
                voxels_volumens_II= sess.run(net_g_test,feed_dict={z_vector:z})
                #get random numbers as index to retrieve generated instances
                id_ch = np.random.randint(0, batch_size, 4)

                #plot the random generated data
                for i in range(3):
```

```
              print(voxel_volumes[id_ch[i]].max())
              if voxel_volumes[id_ch[i]].max() > 0.5:
                voxels = np.squeeze(voxel_volumes[id_ch[i]])
                #filter the voxels to binary values
                voxels[voxels < 0.5] = 0
                voxels[voxels >= 0.5] = 1
                #modify the shape of the voxels
                voxels=nd.zoom(voxels,\
                (0.5, 0.5, 0.5), mode='constant', order=0)\
                .reshape((32,32,32))
                #save image
                saveFromVoxels(voxels,path+"/img_{}_{}".format(epoch, i))

            #concatenate arrays
            generated_array=np.concatenate((voxel_volumes, voxels_volumens_II),axis=0)
            del voxel_volumes
            del voxels_volumens_II

            #get information of our generations and create a label array
            number_generated_instances=generated_array.shape[0]
            label_array=create_array_labels(label, number_generated_instances)

            #save the array
            with h5py.File(path+"/generated_data_array_{}".format(epoch)+'.h5', 'w') as hf:
              hf.create_dataset("generated_data",data=generated_array)
              hf.create_dataset("label",data=label_array)
            hf.close()

            #delete the generated array
            del generated_array

        #save a checkpoint of the model to load it to generate extra data or
        #to keep going with the training in case of interruption
        if epoch % 50 == 0:
            saver.save(sess, save_path = path + '/biasfree_' + str(epoch)+'.cptk')

            #save the evolution of the loss fucntions
            with h5py.File(path+"/evolution_loss_functions{}".format(epoch)+'.h5', 'w') as hf:
              hf.create_dataset("generator_loss",data=loss_function_generator)
              hf.create_dataset("dicriminator_loss",data=loss_function_discriminator)
              hf.create_dataset("dicriminator_accuracy",data=discriminator_accuracy)
```

```
                hf.close()

def generateGAN(path, label, trained_model_path=None,epoch='last', n_batches=10):
  weights = initialiseWeights()

  z_vector = tf.placeholder(shape=[batch_size,z_size],dtype=tf.float32)
  net_g_test = generator(z_vector, phase_train=True, reuse=False)

  sess = tf.Session()
  saver = tf.train.Saver()

  with tf.Session() as sess:
      sess.run(tf.global_variables_initializer())
      saver.restore(sess, trained_model_path)

      #generate data
      #
      for i in range(n_batches):
        if i == 0:
          #next_sigma = float(raw_input())
          z_sample = np.random.normal(0, 0.33, size=[batch_size, z_size]).astype(np.float32)
          generated_data=sess.run(net_g_test,feed_dict={z_vector:z_sample})

        else:
          #next_sigma = float(raw_input())
          z_sample = np.random.normal(0, 0.33, size=[batch_size, z_size]).astype(np.float32)
          generated_samples=sess.run(net_g_test,feed_dict={z_vector:z_sample})
          generated_data=np.concatenate((generated_data, generated_samples), axis=0)

  number_generated_instances=generated_data.shape[0]
  label_array=create_array_labels(label,number_generated_instances)

  #generate the dataset
  with h5py.File(path+"/generated_data_array_{}_{}".format(epoch,str(label))+'.h5', 'w') as hf:
    hf.create_dataset("generated_data",data=generated_data)
    hf.create_dataset("labels",data=label_array)
  hf.close()


def train_multiple_dataset_with_multiple_labels(list_data_sets_paths,\
    list_dataset_names,labels,root_directory):
    number_datasets_to_analyse=len(list_data_sets_paths)
    number_labels_analysis=len(labels)
    for dataset_index in range(list_data_sets_paths):
        #get the name oof the dataset we are analysis
        dataset_analysis=list_data_sets_paths[dataset_index]
```

```
        model=list_dataset_names[dataset_index]
        for label_index in range(number_labels_analysis):
            #get the labels of analysis
            label=labels[label_index]

            #creare directories to store results and get the data
            current_directory=root_directory
            results_directory=current_directory+'/gans_results'
            dataset_directory=results_directory+'/'+str(model)
            label_directory=dataset_directory+'/'+str(label)
            #Create directories
            create_folder_in_path_check_folder_created(current_directory,results_directory)
            create_folder_in_path_check_folder_created(results_directory,dataset_directory)
            create_folder_in_path_check_folder_created(dataset_directory,label_directory)
            #

            #get the data
            dataset= h5py.File(dataset_analysis, 'r')
            attributes_training=np.array(dataset.get('attributes_training'))
            attributes_testing=np.array(dataset.get('attributes_testing'))
            labels_training=np.array(dataset.get('labels_training'))
            labels_testing=np.array(dataset.get('labels_testing'))
            dataset.close()
            del dataset
            del attributes_testing

            #transform the data
            boolean_mask=np.where(labels_training == label)[0]

            #Get the instances that correspond with the labels
            gans_data=attributes_training[boolean_mask][:500]
            gans_data=gans_data.reshape((-1,32,32,32))

            #Transform the data to 64x64x64 format
            gans_data=double_voxels_dimension(gans_data)

            #Training process
            trainGAN(gans_data,label_directory,label)

######################       END FUNCTIONS    ############################

#Create the first and  second models
#list_data_sets_paths=['merged_dataset_0.2labelled_instances.h5',\
#'merged_dataset_0.4labelled_instances.h5','merged_dataset_0.6labelled_instances.h5',\
#'merged_dataset_0.8labelled_instances.h5','merged_dataset.h5']

```

```
list_data_sets_paths=['merged_dataset_0.2labelled_instances.h5',\
'merged_dataset_0.4labelled_instances.h5','merged_dataset_0.6labelled_instances.h5',\
'merged_dataset_0.8labelled_instances.h5','merged_dataset.h5']

#name of the models we are going to use
#list_dataset_names=['0.20 dataset','0.40 dataset','0.60 dataset','0.80 dataset',\
#'full dataset']
list_dataset_names=['0.20 dataset','0.40 dataset','0.60 dataset','0.80 dataset',\
'full dataset']

#labels of the dataset we are going to use
#labels=[0,1,2,3,4,5,6,7,8,9,10,11,12,13]
labels=[0,1,2,3,4,5,6,7,8,9,10,11,12,13]

#train_multiple_dataset_with_multiple_labels(list_data_sets_paths,\
#list_dataset_names,labels,root_directory)

path_check_check_point='G:/gans_project_root_directory/processed_data/\
gans_results/1/checkpoints_and_arrays/biasfree_3950.cptk'

result_generation='G:/gans_project_root_directory/processed_data/gans_results/1/new_generated_data'

generateGAN(result_generation, 1, trained_model_path=path_check_check_point,\
epoch='3900',n_batches=10)
```

Listing 1: 3D GANs code

```
#load utils from keras
#Import Keras tools we use to implements GANs
import tensorflow as tf
from tensorflow import keras

#Load utils from skleanr
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import train_test_split

#Load utils from standard libraries
import h5py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import seaborn as sns
sns.set_style('white')


class IIID_classification():

    def __init__(self):
```

```
        self.horizontal_axis=16
        self.vertical_axis=16
        self.volume_axis=16
        self.color_channels=3
        self.input_size=(self.horizontal_axis,self.vertical_axis,\
        self.volume_axis, self.color_channels)
        self.number_classes=10
        self.one_dimension_size=4096

        #Training parameters Good combinations:(30,80),
        self.epochs=2
        self.batch=86
        #batch_size=128, epochs=50
        self.validation_split=0.20
        self.learning_rate=0.001

    #Normally 3D model are in h5 format.Open h5 fles and separate the instances
    #within them into training and testing files.
    def open_h5(self,file_to_open='aaa'):
        with h5py.File(file_to_open+".h5", 'r') as h5:
            attributes_training,labels_training=h5["X_train"][:],h5["y_train"][:]
            attributes_testing,labels_testing= h5["X_test"][:], h5["y_test"][:]

            return attributes_training,labels_training,attributes_testing,\
            labels_testing

    #In most of the datasets the 3D data is in 1D. So,  we have to process
    #this data for its visualization and posterio analysis

    #Find the rgb values of our dataset
    def add_rgb_dimention(self, instance):
        #Choose the color map we are using
        scaler_map = cm.ScalarMappable(cmap="Oranges")

        #Transform the instance. The -1 fits automatically the size to the
        #dimension
        instance= scaler_map.to_rgba(instance)[:, : -1]

        return instance

    #Process to transform our 1D data to 3D data
    def add_color_dimension(self, dataset_to_transform):
        dataset_with_color_coordinates=np.ndarray((\
        dataset_to_transform.shape[0],self.one_dimension_size,3))

        #Loop through all the instance to add the color coordinates
        for instance_index in range(dataset_to_transform.shape[0]):
```

```
            dataset_with_color_coordinates[instance_index]=\
            self.add_rgb_dimention(dataset_to_transform[instance_index])

        return dataset_with_color_coordinates

    def reshape_dataset(self, dataset):
        #convert our data set to a 'number of instance' + 4D dimensional dataset
        # the '-1' automatically calculates the remaining dimension.
        dataset = dataset.reshape(-1,self.horizontal_axis, self.vertical_axis,\
        self.volume_axis, self.color_channels)

        return dataset

    def one_hot_encode_labels(self,labels):
        #convert target variable into one-hot
        labels = keras.utils.to_categorical(labels,self.number_classes)

        return labels

    def Conv(self,filters=16, kernel_size=(3,3,3),activation='relu',\
    input_shape=None):
        if input_shape:
            return keras.layers.Conv3D(filters=filters,kernel_size=kernel_size,\
            padding='Same', activation=activation, input_shape=input_shape)
        else:
            return keras.layers.Conv3D(filters=filters,kernel_size=kernel_size,\
            padding='Same', activation=activation)


    #3D convolutional networks require a tensor innput of five dimensions:
    #number of instances per batch, horizontal dimension , vertical dimension,
    #volumen dimension, number of color channels.
    def convolutional_IIID_network(self):
        #Common structure of 3CNN
        ## input layer
        input_layer= keras.layers.Input((self.input_size))

        ## Add the 3D convolutional layers with different characteristics
        #The parenthesis after the layer connect the previous layer with
        #the layer we have already created.
        conv_layer1 = keras.layers.Conv3D(filters=8, kernel_size=(3, 3, 3),\
        activation='relu')(input_layer)

        #Add more 3D convolutional layers and 3D maxpool layers.
        conv_layer2 = keras.layers.Conv3D(filters=16, kernel_size=(3, 3,
```

```
3),\
        activation='relu')(conv_layer1)

        ## add max pooling to obtain the most imformatic features
        pooling_layer1 = keras.layers.MaxPool3D(pool_size=(2, 2, 2))\
        (conv_layer2)

        conv_layer3 = keras.layers.Conv3D(filters=32, kernel_size=(3, 3,
3),\
        activation='relu')(pooling_layer1)

        conv_layer4 = keras.layers.Conv3D(filters=64, kernel_size=(3, 3,
3),\
        activation='relu')(conv_layer3)
        pooling_layer2 = keras.layers.MaxPool3D(pool_size=(2, 2, 2))\
        (conv_layer4)

        #perform batch normalization on the convolution outputs before
feeding\
        #it to MLP architecture
        pooling_layer2 = keras.layers.BatchNormalization()(pooling_layer2
)
        flatten_layer = keras.layers.Flatten()(pooling_layer2)

        #Fully connected layer of top of the convolutions to classify the
 model
        #First transform the results of the convoluation into a 1D format
        dense_layer1 = keras.layers.Dense(units=2048, activation='relu')\
        (flatten_layer)
        dense_layer1 = keras.layers.Dropout(0.4)(dense_layer1)
        dense_layer2 = keras.layers.Dense(units=512, activation='relu')\
        (dense_layer1)
        dense_layer2 = keras.layers.Dropout(0.4)(dense_layer2)
        output_layer = keras.layers.Dense(units=self.number_classes,\
        activation='softmax')\
        (dense_layer2)

        ## define the model with input layer and output layer
        model=keras.models.Model(inputs=input_layer, outputs=output_layer
)

        #Compile the model
        model.compile(loss="categorical_crossentropy",\
        optimizer=keras.optimizers.Adadelta(lr=0.1),metrics=["accuracy"])

        return model

    def convolutional_IIID_network_other_Structure(self):
        #Normal feed-forward structure
        cnn_three=keras.models.Sequential()

        #USe the fucntion Conv that we create before to cast the 3D
        #convolutional network
```

```
        cnn_three.add(self.Conv(8, (3,3,3), input_shape=self.input_size))
        cnn_three.add(self.Conv(16, (3,3,3)))

        #model.add(BatchNormalization())
        cnn_three.add(keras.layers.MaxPool3D())
        #cnn_three.add(keras.layers.Dropout(0.25))

        #
        cnn_three.add(self.Conv(32, (3,3,3)))
        cnn_three.add(self.Conv(64, (3,3,3)))
        cnn_three.add(keras.layers.BatchNormalization())
        cnn_three.add(keras.layers.MaxPool3D())
        cnn_three.add(keras.layers.Dropout(0.25))

        #Fully connected layer of top of the convolutions to classify the
    model
        #First transform the results of the convoluation into a 1D format
        cnn_three.add(keras.layers.Flatten())

        #Add more fully connected layers
        cnn_three.add(keras.layers.Dense(4096, activation='relu'))
        cnn_three.add(keras.layers.Dropout(0.5))
        cnn_three.add(keras.layers.Dense(1024, activation='relu'))
        cnn_three.add(keras.layers.Dropout(0.5))
        #The input layer contains as many as neurons as different classes
        cnn_three.add(keras.layers.Dense(self.number_classes,activation='
    softmax'))

        #Compile the model
        cnn_three.compile(optimizer='adam',loss = "
    categorical_crossentropy",\
        metrics=["accuracy"])

        #return the model
        return cnn_three

    def voxnet(self):
        #Common structure of 3CNN
        ## input layer
        input_layer= keras.layers.Input((self.input_size))
        ## Add the 3D convolutional layers with different characteristics
        #The parenthesis after the layer connect the previous layer with
        #the layer we have already created.
        conv_layer1 = keras.layers.Conv3D(filters=32, kernel_size=(5,5,5)
    ,\
        strides=(2,2,2),activation='relu')(input_layer)

        #Add more 3D convolutional layers and 3D maxpool layers.
        conv_layer2 = keras.layers.Conv3D(filters=32, kernel_size=(3, 3,
    3),\
        strides=(1,1,1),activation='relu')(conv_layer1)

        ## add max pooling to obtain the most imformatic features
```

```
        pooling_layer1 = keras.layers.MaxPool3D(pool_size=(2, 2, 2))\
        (conv_layer2)
        #
        flatten_layer = keras.layers.Flatten()(pooling_layer1)

        #Fully connected layer of top of the convolutions to classify the model
        #First transform the results of the convoluation into a 1D format
        dense_layer1 = keras.layers.Dense(units=128, activation='relu')\
        (flatten_layer)
        dense_layer1 = keras.layers.Dropout(0.5)(dense_layer1)
        output_layer = keras.layers.Dense(units=self.number_classes,\
        activation='softmax')(dense_layer1)

        ## define the model with input layer and output layer
        model=keras.models.Model(inputs=input_layer,outputs=output_layer)

        #Compile the model
        model.compile(loss="categorical_crossentropy",\
        optimizer=keras.optimizers.SGD(lr=self.learning_rate,momentum=0.9),\
        metrics=["accuracy"])

        return model

    #Fucntion to create a callback to stop the training process given a set
    #of characteristics
    def generate_stopping_criteria(self, monitor_metric='val_accuracy',\
    callback_patience=20):
        #create the callback object to stop the training process. Patience
        #is the number of iterations without improvement that have to happen to
        #stop the training process. Monitor is the performance measure that we
        #consider to stop the training process.
        stopping_call_back=keras.callbacks.EarlyStopping(monitor=monitor_metric,\
        mode='max',verbose=1,patience=callback_patience)
        #return the callback
        return stopping_call_back

    #Fucntion to save our model given a specific criteria
    def generate_model_saving_criteria(self, monitor_metric='val_accuracy',\
    model_name='best_model'):
        #generate hte name of the h5 that will store the best model
        model=model_name+'.h5'
        #generate the callback to store the best model. The monitor metric
        #is the measure that we want to maximize with out model
        saving_call_back=keras.callbacks.ModelCheckpoint(model,\
```

```
        monitor=monitor_metric,mode='max',verbose=1,save_best_only=True)
        #return the call back
        return saving_call_back

    # Train model with the parametrs indicated in the constructor
    def train_model(self, model_to_train, attributes_training,
labels_training,\
    callback_list):
        #Train the model
        train_model=model_to_train.fit(x=attributes_training,y=
labels_training,\
        batch_size=self.batch, epochs=self.epochs,\
        validation_split=self.validation_split, verbose=1, shuffle=True,\
        callbacks=callback_list)
        #Return the trained model
        return train_model

    #The following fucntion saves the model that we are training.
    def save_smodel(self, model_to_save):
        saved_model=model_to_save.save('3d_classifier.h5')

    #Evaluate the model after the training process.
    def model_evaluation(self, model_to_evaluate, attributes_testing,\
    labels_testing, path='path',title='confusion_matrix'):
        #Predict the labels using the model we trained
        class_prediction=model_to_evaluate.predict(attributes_testing)
        #Because the model is one hot encoded we have and we used softmax
        #activation fucntion
        class_prediction=np.argmax(class_prediction, axis=1)

        #Calculate the accuracy score of our model.
        accuracy_model=round(accuracy_score(class_prediction,
labels_testing),3)

        #Confusin matrix. The confucion matrix will indicate which labels
 are
        #hard to predict and other potential problem in out model.
        confusion_matrix_model=confusion_matrix(labels_testing,
class_prediction)

        #transform the numpy array to a pandas dataframe
        confusion_matrix_model=pd.DataFrame(confusion_matrix_model,\
        index = range(self.number_classes),\
        columns = range(self.number_classes)).astype('int')

        #Plot the confusion matrix
        plt.figure(figsize=(20,20))
        sns.heatmap(confusion_matrix_model, annot=True)
        plt.title(title+'      '+'accuracy: '+str(accuracy_model))
        plt.savefig(path+'.png')
        plt.clf()
        #return the model evaluation object
        return accuracy_model
```

```
    #PLot the evolution of the training process
    def plot_validation_score(self, model, path='path',\
    title='validation_training'):
        #We are going to plot the training and validation scores. We will create
        #a .png create the figure we are going to plot our accuracy and the
        #value of the loss fucntion
        plt.figure(figsize=[20,20])
        #Plot the accuracy evolution
        plt.plot(model.history['accuracy'])
        plt.plot(model.history['val_accuracy'])
        plt.title('Model training and validation_'+title)
        plt.ylabel('performance')
        plt.xlabel('epoch')
        plt.legend(['training set','validation set'], loc='upper left')

        #save figure we created
        plt.savefig(path+'.png')
        plt.clf()

    #The following function is to test the performance of our model in different
    #set of our training set with different sizes.
    #This could be useful to evaluate the performance of the model in when
    #we have small datasets.
    #list_of_models is a list that contains the models we are using
    ##list_of_models is a list that contains the splits or reduction of the
    #dataset  we are using
    def dataframe_record_experiment_results(self,list_of_models, list_of_splits,\
    attributes_training,labels_training,attributes_testing,labels_testing,\
    number_data_splits=3,random_seed=0):
        #Get the number of models
        number_models=len(list_of_models)

        #Get the dataset reduction we want to apply
        number_data_splits=len(list_of_splits)

        #create an array to store the results
        results_array=np.zeros((number_data_splits,number_models))

        #loop through the models and
        for model_index in range(number_models):
            #loop through the splits
            for split_index in range(number_data_splits):
                #Know the model we are using and how much are we gonna reduce
                #the dataset
```

```
                data_division=list_of_splits[split_index]
                model=list_of_models[model_index]

                #reduce the dataset
                attributes_reduced, attributes_discard,\
                labels_reduced, labels_discard = train_test_split(
                attributes_training, labels_training,\
                test_size=data_division, random_state=random_seed)

                #Train, evaluate and plot the model
                model_train= self.\
                train_model(model, attributes_reduced, labels_reduced)

                #Evaluate the model
                performance=self.model_evaluation(model,\
                attributes_testing,labels_testing,\
                title='confusion_matrix_'+'model_'+str(model_index)+'_'+\
                'used_data_'+str(data_division))

                #Plot the evoluation of the validation score and the training
                #accuracy
                self.plot_validation_score(model_train,\
                file_name='validation_training'+'model_'+str(model_index)+'_'+\
                'used_data_'+str(data_division))

                results_array[split_index,model_index]=performance

        #Issue a csv file with the result of our model in different data
        #reductions
        pd.DataFrame(results_array).to_csv('results.csv')

        return results_array
```

Listing 2: 3D Deep Learning classifiers

```
#load utils from keras
#Import Keras tools we use to implements GANs
import tensorflow as tf
from tensorflow import keras

#Load utils from sklearn
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import train_test_split

#Load utils from standard libraries
import h5py
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import seaborn as sns
```

```
sns.set_style('white')
import os

#import the code we created
from IIId_classifiers import IIID_classification

# ###############################################################################

#fucntion to create folders in a given path

def get_list_elements_without_pattern_not_current_directory(directory_to_search):
    #Comprehension list that by given a directory, explores
    files= [element for element in os.listdir(directory_to_search)]
    #
    return files



def create_folder_in_path_check_folder_created(path_creation,path_to_create):
    #
    directories_in_directory_where_eant_create=\
    get_list_elements_without_pattern_not_current_directory(path_creation)
    #
    directories_path_in_directory_where_eant_create=\
    [path_creation+'/'+path for path in \
    directories_in_directory_where_eant_create]
    #
    if path_to_create not in directories_path_in_directory_where_eant_create:
        #
        os.mkdir(path_to_create)

# ###############################################################################


#Load the dataset
def analysis_data_and_class_creation(data_set_name,epochs=1000, batchs=32,\
validation=0.50,learning_rate=0.001):
    #Open the h5 file with the function within the class
    dataset= h5py.File(data_set_name, 'r')
    attributes_training=np.array(dataset.get('attributes_training'))
    attributes_testing=np.array(dataset.get('attributes_testing'))
    labels_training=np.array(dataset.get('labels_training'))
    labels_testing=np.array(dataset.get('labels_testing'))
    dataset.close()

    #Transform the dataset
```

```
    number_voxels_columns=attributes_training.shape[1]
    #
    dimension_axis=int(round(math.pow(number_voxels_columns,1/3.)))
    #
    attributes_training=attributes_training.reshape(-1,dimension_axis,\
    dimension_axis,dimension_axis,1)
    #
    attributes_testing=attributes_testing.reshape(-1,dimension_axis,\
    dimension_axis,dimension_axis,1)
    #
    number_different_labels=len(np.unique(labels_training))

    #Load the class
    IIID_classifier= IIID_classification()
    IIID_classifier.horizontal_axis=dimension_axis
    IIID_classifier.vertical_axis=dimension_axis
    IIID_classifier.volume_axis=dimension_axis
    IIID_classifier.color_channels=1
    IIID_classifier.input_size=(IIID_classifier.horizontal_axis,\
    IIID_classifier.vertical_axis,IIID_classifier.volume_axis,\
    IIID_classifier.color_channels)
    IIID_classifier.number_classes=number_different_labels
    IIID_classifier.one_dimension_size=number_voxels_columns

    #Training parameters Good combinations:(30,80),
    IIID_classifier.epochs=epochs
    IIID_classifier.batch=batchs
    #batch_size=128, epochs=50
    IIID_classifier.validation_split=validation
    IIID_classifier.learning_rate=learning_rate

    #One hot encode training and testing labels
    labels_training=labels = keras.utils.to_categorical(labels_training)
    #return the classifier object
    return IIID_classifier


###################            TRAINING                ######################
######################            TRAIN MODEL I            ##################

#Fucntion to analyse a large number of dataset and store the results of a chosen
#model into a csv. Hence, evaluate the perfromance of the model in multiple
#datasets. Addtionally we can repeat the evaluation process several times to
#calculate the average and std measure of the performance.
def multiple_data_model_analysis(list_datasets, list_titles, list_models,\
number_analysis_per_dataset=5,patience=50, csv_title='csv_multiple_data'):
```

```
    #create a directory to save the results
    #get current directory
    current_directory=os.getcwd()
    #directory we will create
    directory_results=current_directory+'/'+'results_analysis_datasets'
    #create the dataset
    create_folder_in_path_check_folder_created\
    (current_directory,directory_results)

    #Basic analysis of the number of data we have to analyse
    number_sets_to_analyse=len(list_data_sets)
    number_model_analysis=len(model_names)

    #Create an array to store the results. Every column
    #is a different dataset
    results_array=np.zeros((number_model_analysis*4,
number_sets_to_analyse))

    #Create and 3d classifier object with the chracteristics of our data
    IIID_classifier=analysis_data_and_class_creation(\
    list_data_sets[number_model_analysis-1],epochs=1000, batchs=32,\
    validation=0.50,learning_rate=0.001)

    #Loop through all the datasets
    for dataset_index in range(number_sets_to_analyse):
        #Get the dataset of analysis
        data_set_of_analysis=list_data_sets[dataset_index]
        #Get the title/label of the dataset we are analysing.
        #The title will appear on the confusion matrix, table of
        #results and other visualizations.
        title_data_analysis=titles[dataset_index]

        print 'dataset: '+' '+data_set_of_analysis+' '+
title_data_analysis

        #create a numpy array to store the resuls of the model
        #on the dataset after a number of repetitions
        results_model_array=np.zeros(number_analysis_per_dataset)
        results_model_accuracy_array=np.zeros(number_analysis_per_dataset
)

        #repeat the analysis as inidcated in 'number_analysis_per_dataset
'
        for analysis_index in range(number_analysis_per_dataset):
            print 'analysis_number: '+str(analysis_index)
            #load the classification model
            IIID_model_I=IIID_classifier.voxnet()

            #load the data from dataset
            dataset= h5py.File(data_set_of_analysis,'r')
            attributes_training=np.array(dataset.get('attributes_training
'))
            attributes_testing=np.array(dataset.get('attributes_testing')
```

```
)
            labels_training=np.array(dataset.get('labels_training'))
            labels_testing=np.array(dataset.get('labels_testing'))
            dataset.close()

            #Transform the dataset
            number_voxels_columns=attributes_training.shape[1]
            #get the one of the dimensions of the cuboid grid
            dimension_axis=int(round(math.pow(number_voxels_columns,1/3.)
))
            #transfrom the attributes
            attributes_training=attributes_training.reshape(-1,
dimension_axis,\
            dimension_axis,dimension_axis,1)
            #testing attributes
            attributes_testing=attributes_testing.reshape(-1,
dimension_axis,\
            dimension_axis,dimension_axis,1)
            #One hot encode training and testing labels
            labels_training=labels=keras.utils.to_categorical(
labels_training)

            #Generate the callbacks for saving the model
            saving=IIID_classifier.generate_model_saving_criteria\
            (monitor_metric='val_accuracy',model_name=\
            directory_results+'/'+'best_model'+\
            ' '+title_data_analysis+' '+str(analysis_index))

            #Generate callbacks to stop the training porcess. patience
            #is the number of iterations without an improvements
            stopping=IIID_classifier.generate_stopping_criteria\
            (monitor_metric='val_accuracy',callback_patience=patience)
            #generate a list with the callbacks functions we just created
            list_callbacks=[saving,stopping]

            #Traning process
            IIID_model_I_train= IIID_classifier.\
            train_model(IIID_model_I, attributes_training,\
            labels_training,list_callbacks)

            #Calculate the accuracy and plot the evalution of the
            #training and validation performance throughout the
            #training process for both models.
            IIID_model_I_accuracy=\
            IIID_classifier.model_evaluation\
            (IIID_model_I,attributes_testing,labels_testing)

            #get the max validation score during the training process
            max_val_score=max(IIID_model_I_train.history['val_accuracy'])

            #append the max validation score to the array
            results_model_array[analysis_index]=max_val_score

```

```
            #plot the evolution of the validation and training score
            IIID_classifier.plot_validation_score(IIID_model_I_train,\
            path=directory_results+'/'+'voxnet '+title_data_analysis+' '+\
            str(analysis_index), title='voxnet '+title_data_analysis+' '+\
            str(analysis_index))

            #Get the model accuracy and plot the confusion matrix on the
            #training set
            IIID_model_I_accuracy=\
            IIID_classifier.model_evaluation(IIID_model_I, attributes_testing,\
            labels_testing,path=\
            directory_results+'/'+'confusion matrix voxnet '+\
            title_data_analysis+' '+str(analysis_index), title=\
            'confusion matrix voxnet '+title_data_analysis+' '+str(analysis_index))

            #Store the model accuracy
            results_model_accuracy_array[analysis_index]=IIID_model_I_accuracy

            del IIID_model_I
            del IIID_model_I_train
            del IIID_model_I_accuracy
            del list_callbacks

        #calculate the mean and std of the obtained results
        mean_val_accuracy=round(np.mean(results_model_array),3)
        std_val_accuracy=round(np.std(results_model_array),3)
        mean_accuracy=round(np.mean(results_model_accuracy_array),3)
        std_accuracy=round(np.std(results_model_accuracy_array),3)

        #Move the results to the results array
        results_array[0,dataset_index]=mean_val_accuracy
        results_array[1,dataset_index]=std_val_accuracy
        results_array[2,dataset_index]=mean_accuracy
        results_array[3,dataset_index]=std_accuracy

    #transform the numpy array with the results into a dataframe
    dataframe_results=pd.DataFrame(data=results_array,index=['mean validation',\
    'std validation','mean accuracy','std_accuracy'],columns=list_datasets)
    #transform the data frame into csv
    export_csv=dataframe_results.to_csv(\
    directory_results+'/'+csv_title+'.csv',index=True,header=True)

# ##############################################################################
#
```

```
################################################################################

#Create the first and  second models
list_data_sets=['merged_dataset_0.2labelled_instances.h5',\
'merged_dataset_0.4labelled_instances.h5',\
'merged_dataset_0.6labelled_instances.h5',\
'merged_dataset_0.8labelled_instances.h5','merged_dataset.h5']

titles=['0.20 dataset','0.40 dataset','0.60 dataset','0.80 dataset',\
'full dataset',]

model_names=['voxnet']

multiple_data_model_analysis(list_data_sets,titles,model_names,\
number_analysis_per_dataset=5,patience=30, csv_title='csv_multiple_data')
```

Listing 3: Multiple 3D data analysis

```
#Import standard libraries
import os
import numpy as np
import h5py
import matplotlib.pyplot as plt
import open3d as o3d
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import scipy.io as io
import scipy.ndimage as nd
import random

#functions we are using
#fucntion to manage directories
def half_voxels_dimension(voxel_array_to_transform):
    number_voxels=voxel_array_to_transform.shape[0]
    #Get the number of voxels per dimension
    one_dimension_voxels=voxel_array_to_transform.shape[1]
    half_dimension=int(one_dimension_voxels/2)

    #create an array to store the transformed instances
    voxel_array_transformed=np.zeros((number_voxels,half_dimension,\
    half_dimension,half_dimension,1))
    #loop through all the instances in the array we want to transfrom
    for voxel_tranformation_index in range(number_voxels):
        #get the array that we want to transform
        voxel_to_transform=voxel_array_to_transform[
    voxel_tranformation_index]
        #modify the voxels
        voxel_to_transform=np.pad(voxel_to_transform,(0, 0),\
        'constant',constant_values=(0, 0))
        #Square voxels
        voxel_to_transform=nd.zoom(voxel_to_transform,\
```

```
        (0.5, 0.5, 0.5), mode='constant', order=0)\
        .reshape((half_dimension,half_dimension,half_dimension,1))
        #Add the transformed voxel to the array that contains the
transformed
        #voxels
        voxel_array_transformed[voxel_tranformation_index]=\
        voxel_to_transform
    #return the modified array
    return voxel_array_transformed


def double_voxels_dimension(voxel_array_to_transform):
    number_voxels=voxel_array_to_transform.shape[0]
    #Get the number of voxels per dimension
    one_dimension_voxels=voxel_array_to_transform.shape[1]
    double_dimension=one_dimension_voxels*2
    print (double_dimension)
    #create an array to store the transformed instances
    voxel_array_transformed=np.zeros((number_voxels,double_dimension,\
    double_dimension,double_dimension,1))

    #loop through all the instances in the array we want to transfrom
    for voxel_tranformation_index in range(number_voxels):
        #get the array that we want to transform
        voxel_to_transform=voxel_array_to_transform[
voxel_tranformation_index]
        #modify the voxels
        voxel_to_transform=np.pad(voxel_to_transform,(0, 0),\
        'constant',constant_values=(0, 0))
        #Square voxels
        voxel_to_transform=nd.zoom(voxel_to_transform,\
        (2, 2, 2), mode='constant', order=0)\
        .reshape((double_dimension,double_dimension,double_dimension,1))
        #Add the transformed voxel to the array that contains the
transformed
        #voxels
        voxel_array_transformed[voxel_tranformation_index]=\
        voxel_to_transform
    #return the modified array
    return voxel_array_transformed

def get_list_elements_pattern_not_current_directory(directory_to_search,
pattern):
    #Comprehension list that by given a directory, explores
    pattern_files= [element for element in os.listdir(directory_to_search
) if\
    element.endswith("."+pattern)]
    #
    return pattern_files

def get_list_elements_without_pattern_not_current_directory(
directory_to_search):
    #Comprehension list that by given a directory, explores
```

```
    files= [element for element in os.listdir(directory_to_search)]
    #
    return files

#funntion to get automatically the path of a given file in the curren directory
def generate_directory_path_contains_current_directory(folder_name):
    #Get current directoy
    current_directory=os.getcwd()
    #Create a directory path to explore
    directory_to_explore=current_directory+'/'+folder_name
    #Create a directory path to explore
    return directory_to_explore

def create_folder_in_path_check_folder_created(path_creation, path_to_create):
    #
    directories_in_directory_where_eant_create=\
    get_list_elements_without_pattern_not_current_directory(path_creation)
    #
    directories_path_in_directory_where_eant_create=\
    [path_creation+'/'+path for path in \
    directories_in_directory_where_eant_create]
    #
    if path_to_create not in directories_path_in_directory_where_eant_create:
        #
        os.mkdir(path_to_create)

#function to plot the loss functions
def loss_gans_plot(evolution_loss_function_discriminator,\
evolution_loss_function_generator,evolution_accuracy,results_directory, label):
    plt.figure(figsize=(20,20), dpi=80)

    #First subgraph with discriminator loss
    plt.rc('xtick',labelsize=30)
    plt.rc('ytick',labelsize=30)
    plt.subplot(3, 1, 1)
    plt.plot(evolution_loss_function_discriminator,'darkorange', lw=0.4)
    plt.yscale('log')
    plt.title("Evolution of Discriminator's Loss", fontsize=30)
    plt.xlabel('epochs',fontsize=30)
    plt.ylabel("Discriminator's Loss (log scale)",fontsize=30)


    #Second subgraph with generator loss
    plt.subplot(3, 1, 2)
    plt.plot(evolution_loss_function_generator,'blue', lw=0.4)
    plt.title("Evolution of Generator's Loss", fontsize=30)
    plt.rc('xtick',labelsize=30)
    plt.rc('ytick',labelsize=30)
```

```
    plt.xlabel('epochs',fontsize=30)
    plt.ylabel("Generator's Loss",fontsize=30)


    #Third subgraph with accuracy

    plt.subplot(3, 1, 3)
    plt.plot(evolution_accuracy,'forestgreen',lw=0.45)
    plt.title("Evolution of Discriminator's Accuracy", fontsize=30)
    plt.rc('xtick',labelsize=30)
    plt.rc('ytick',labelsize=30)
    plt.xlabel('epochs',fontsize=30)
    plt.ylabel("Discriminator's Accuracy",fontsize=30)
    #plt.plot(evolution_loss_function_generator, lw=1)

    #Space between subgraphs in the main graph
    plt.subplots_adjust(hspace=0.4)

    #plt.show()
    plt.savefig(results_directory+'/_evolution_loss_accuracy'+str(label))

def get_loss_data_from_checkpoints(data_directory,checkpoint_to_append):
    evolution_loss_function_discriminator=[]
    evolution_loss_function_generator=[]
    evolution_accuracy=[]
    elements_to_explore=['dicriminator_loss','generator_loss',\
    'dicriminator_accuracy']
    number_elements_to_explore=len(elements_to_explore)

    for elements_to_explore_index in range(number_elements_to_explore):
        element_to_explore=elements_to_explore[elements_to_explore_index]

        for checkpoint_index_dis in range(len(checkpoint_to_append)):
            checkpoint=checkpoint_to_append[checkpoint_index_dis]
            #get the dataset
            data_path=data_directory+'/evolution_loss_functions'+
    checkpoint+'.h5'

            dataset= h5py.File(data_path, 'r')
            loss_values=np.array(dataset.get(element_to_explore))
            dataset.close()
            del dataset

            for information in loss_values:
                if elements_to_explore_index==0:
                    evolution_loss_function_discriminator.append(
    information)
                elif elements_to_explore_index==1:
                    evolution_loss_function_generator.append(information)
                elif elements_to_explore_index==2:
                    evolution_accuracy.append(information)

    return evolution_loss_function_discriminator,
```

```
    evolution_loss_function_generator,evolution_accuracy

def display_generated_data(data_path,list_epochs,\
epoch_in_list_to_display=0,instance_to_display=0,binarise=True,
    reduce_voxel=False):
    #get the path of the dataset we want
    epoch_to_analyse=list_epochs[epoch_in_list_to_display]

    #get the directory of the epoch
    generated_data_path=data_path+'/generated_data_array_'+
    epoch_to_analyse+'.h5'

    #Get the dataset
    dataset= h5py.File(generated_data_path, 'r')
    generated_data=np.array(dataset.get('generated_data'))
    dataset.close()

    #get the intance of the set we want to display
    generated_instance=generated_data[instance_to_display]


    if binarise == True:
        generated_instance=np.where(generated_instance>=0.5,1,0)

    if reduce_voxel==True:
        generated_instance=nd.zoom(generated_instance,\
        (0.5, 0.5, 0.5), mode='constant', order=0)\

    #information about the reshape of the instance to displance and
    reshaping
    generated_instance_shape_voxel=generated_instance.shape[0]

    generated_instance=generated_instance.reshape((\
    generated_instance_shape_voxel,generated_instance_shape_voxel,\
    generated_instance_shape_voxel))

    #plot
    fig=plt.figure()
    ax = fig.gca(projection='3d')
    ax.grid(False)
    plt.axis('off')
    ax.voxels(generated_instance,facecolors='aqua', edgecolor="k")
    plt.show()

def original_data(data_path,list_epochs,\
label=0,instance_to_display=0,binarise=True,reduce_voxel=False):
    #get the directory of the epoch
    generated_data_path=data_path

    #Get the dataset
    dataset= h5py.File(generated_data_path, 'r')
    generated_data=np.array(dataset.get('attributes_training'))
    labels_training=np.array(dataset.get('labels_training'))
```

```
    dataset.close()

    #filter by label
    filter=np.where(labels_training==label)
    generated_data=generated_data[filter]
    labels_training=labels_training[filter]

    #get the intance of the set we want to display
    generated_instance=generated_data[instance_to_display]

    generated_instance=generated_instance.reshape((32,32,32))

    if binarise == True:
        generated_instance=np.where(generated_instance>=0.5,1,0)

    if reduce_voxel==True:
        generated_instance=nd.zoom(generated_instance,\
        (0.5, 0.5, 0.5), mode='constant', order=0)\

    #information about the reshape of the instance to displance and
    reshaping

    generated_instance=np.pad(generated_instance,(0, 0),\
    'constant',constant_values=(0, 0))
    #Square voxels
    generated_instance=nd.zoom(generated_instance,\
    (2, 2, 2), mode='constant', order=0)\
    .reshape((64,64,64,1))

    generated_instance_shape_voxel=generated_instance.shape[0]

    generated_instance=generated_instance.reshape((\
    generated_instance_shape_voxel,generated_instance_shape_voxel,\
    generated_instance_shape_voxel))

    #plot
    fig=plt.figure()
    ax = fig.gca(projection='3d')
    ax.grid(False)
    plt.axis('off')
    ax.voxels(generated_instance,facecolors='aqua', edgecolor="k")
    plt.show()

def get_path_generated_data(data_path,list_epochs,\
epoch_in_list_to_display=0,instance_to_display=0,binarise=True,
    reduce_voxel=False):
    #get the path of the dataset we want
    epoch_to_analyse=list_epochs[epoch_in_list_to_display]

    #get the directory of the epoch
    generated_data_path=data_path+'/generated_data_array_'+
    epoch_to_analyse+'.h5'

```

```
    return generated_data_path

#function to merge dataset. Te original datasets and the generated one
def merge_dataset_with_augmented(augmented_path, 
    path_dataset_to_be_augmented,\
filters=[1,2,3], filter=True, label=2, reduce_voxels=True):

    #get the dataset we want to increase and the subset within it
    dataset= h5py.File(path_dataset_to_be_augmented, 'r')
    attributes_training=np.array(dataset.get('attributes_training'))
    attributes_testing=np.array(dataset.get('attributes_testing'))
    labels_training=np.array(dataset.get('labels_training'))
    labels_testing=np.array(dataset.get('labels_testing'))
    dataset.close()

    #Loop through all the augmented dataset and add the to the original 
    set
    #including the labels

    augmented_dataset= h5py.File(augmented_path, 'r')
    generated_data=np.array(augmented_dataset.get("generated_data"))
    labels_generated_data=np.array(augmented_dataset.get("labels"))


    print (generated_data.shape)

    if filter == True:
        generated_data=generated_data[filters]
        number_instances=generated_data.shape[0]
        label_array=np.full((number_instances),label)
        print (label_array)
        labels_generated_data=label_array
        print (labels_generated_data)

    generated_data=generated_data.reshape((generated_data.shape[0],
    generated_data.shape[1],generated_data.shape[2],generated_data.shape
    [3]))
    #Reduce the size of the generated data
    if reduce_voxels==True:
        generated_data=half_voxels_dimension(generated_data)

    generated_data=generated_data.reshape((generated_data.shape[0],32768)
    )
    print (attributes_training.shape)
    print (generated_data.shape)
    #concatenate the data sets
    attributes_training=np.concatenate((attributes_training,
    generated_data),axis=0)
    labels_training=np.concatenate((labels_training,labels_generated_data
    ),axis=0)

    #safety prints
    print(generated_data.shape)
```

```
    print (labels_generated_data.shape)
    print (attributes_training.shape)
    print (labels_training.shape)
    unique, counts = np.unique(labels_training, return_counts=True)
    print (unique)
    print (counts)

    #save the dataset we have augmented

    with h5py.File(path_dataset_to_be_augmented, 'w') as hf:
      hf.create_dataset('attributes_training',data=attributes_training)
      hf.create_dataset('attributes_testing',data=attributes_testing)
      hf.create_dataset('labels_training',data=labels_training)
      hf.create_dataset('labels_testing',data=labels_testing)
    hf.close()

def merge_dataset_with_original(original_dataset_path,
    path_dataset_to_be_augmented,\
label=0, filters=[1,2,3], filter=True, reduce_voxels=True):

    #get the dataset we want to increase and the subset within it
    dataset= h5py.File(path_dataset_to_be_augmented, 'r')
    attributes_training=np.array(dataset.get('attributes_training'))
    attributes_testing=np.array(dataset.get('attributes_testing'))
    labels_training=np.array(dataset.get('labels_training'))
    labels_testing=np.array(dataset.get('labels_testing'))
    dataset.close()

    #get the aumented data of a specific label
    augmented_dataset= h5py.File(original_dataset_path, 'r')
    generated_data=np.array(augmented_dataset.get('attributes_training'))
    labels_generated_data=np.array(augmented_dataset.get('labels_training
    '))
    augmented_dataset.close()

    filter_I=np.where(labels_generated_data==label)
    generated_data=generated_data[filter_I]
    labels_generated_data=labels_generated_data[filter_I]

    generated_data=generated_data[filters]
    labels_generated_data=labels_generated_data[filters]

    #concatenate the data sets
    attributes_training=np.concatenate((attributes_training,
    generated_data),axis=0)
    labels_training=np.concatenate((labels_training,labels_generated_data
    ),axis=0)

    #safety prints
    print(generated_data.shape)
    print (labels_generated_data.shape)
    print (attributes_training.shape)
    print (labels_training.shape)
```

```
    unique, counts = np.unique(labels_training, return_counts=True)
    print (unique)
    print (counts)

    #save the dataset we have augmented
    with h5py.File(path_dataset_to_be_augmented, 'w') as hf:
      hf.create_dataset('attributes_training',data=attributes_training)
      hf.create_dataset('attributes_testing',data=attributes_testing)
      hf.create_dataset('labels_training',data=labels_training)
      hf.create_dataset('labels_testing',data=labels_testing)
    hf.close()


#################             END OF FUNCTIONS                     #####################


# ###############################################################################
#################            PATHS/DIRECTORIES                     #####################

#visualization tool for generated data and for the evalution of the loss
#functions

#Set up directories
#root directory
current_directory=os.getcwd()

#directory where the data is stored
data_directory='G:/gans_project_root_directory/processed_data/gans_results_0.2/7/checkpoints_and_arrays'

#from the data directory get the name of our data
data_names=\
get_list_elements_pattern_not_current_directory(data_directory,'h5')

#set up directory where we are going to save the visualisations
visualizations_directory='G:/gans_project_root_directory/visualizations/visualization_IIID_figures/generation_visualization/2'

if not os.path.exists(visualizations_directory):
    os.makedirs(visualizations_directory)

loss_visualizations_directory='G:/gans_project_root_directory/visualizations/loss_functions/lr01'
if not os.path.exists(loss_visualizations_directory):
    os.makedirs(loss_visualizations_directory)

new_generations_directory='G:/gans_project_root_directory/processed_data/gans_results/0_0.9_lr'
if not os.path.exists(new_generations_directory):
```

```
    os.makedirs(new_generations_directory)

path_dataset_to_merge='G:/gans_project_root_directory/processed_data/voxel_datasets'
# ##############################################################################
#open the dataset and get the information we want
#####


#epochs to get from a data set
epoch_to_get=['5000']

#################        VISUALISATION GENERATED ######################
display=True
if display ==True:
    display_generated_data(data_directory,epoch_to_get,\
    epoch_in_list_to_display=0,instance_to_display=40, binarise=True,reduce_voxel=False)

#################        VISUALISATION ORIGINAL ######################
#visualise the original data and not the augmented
display_original=False
original_set_path='G:/gans_project_root_directory/processed_data/voxel_datasets/merged_dataset.h5'
if display_original==True:
    original_data(original_set_path,epoch_to_get,\
    label=4,instance_to_display=68,binarise=True,reduce_voxel=False)

#Because lack of memory RAM the process stops several times, hence I have to
#merge the results of several points where the process stopped

#checkpoint label 0
dictionary_checkpoint_to_get={'0':['1050','2750','4650','5500'],
'1':['1350','3050','3950','4550','5150'],'2':['1800','4250','5950'],\
'3':['350','3850','4550','5500'],'4':['550','2100','5950'],\
'5':['1600','2350','4150','5200','5950'],'6':['800','5900'],\
'7':['650','3850','4950','5850','5950','6450'],\
'8':[],'9':[],'10':[],'11':[],'12':[],'13':[],\
'lr01':['5650','4450','4250','4050','3900','3800','2950','2800','2650','2500',\
'2350','1600','1450','1300','1150','1000','800','600','450','250'],\
'lr01':['5850','3700','3550','3350','2350','2250','600','450','250']}

################## CHECK CHECKPOINT GANS #######################################
experimenting_checkpoint=False
if experimenting_checkpoint==True:
    path_to_open=data_directory+'/evolution_loss_functions250.h5'
```

```

    dataset= h5py.File(path_to_open, 'r')
    loss_values=np.array(dataset.get('dicriminator_loss'))
    dataset.close()
    del dataset
    print (len(loss_values))


########### VISUALISATION  LOSS ###########################################
visualise=False
if visualise == True:
    evolution_loss_function_discriminator,evolution_loss_function_generator,\
    evolution_accuracy=\
    get_loss_data_from_checkpoints(data_directory,dictionary_checkpoint_to_get['lr01'])


    loss_gans_plot(evolution_loss_function_discriminator,\
    evolution_loss_function_generator,evolution_accuracy,\
    loss_visualizations_directory,7)


################# ANALYSIS AUGMENTED SET #######################################
#Analysis of the dataset to be augmented for sanity check
analysis_set=False
if analysis_set==True:
    #get the dataset we want to increase and the subset within it
    dataset= h5py.File('G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet/0.2_augmented/augmented_0.2_10.h5', 'r')
    attributes_training=np.array(dataset.get('attributes_training'))
    attributes_testing=np.array(dataset.get('attributes_testing'))
    labels_training=np.array(dataset.get('labels_training'))
    labels_testing=np.array(dataset.get('labels_testing'))
    dataset.close()

    uniqueValues, occurCount = np.unique(labels_training, return_counts=True)
    print (attributes_training.shape)
    print (uniqueValues)
    print (occurCount)
    #np.savetxt(\
    #'G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet/0.2_augmented/0.2_shape.txt',\
    #np.concatenate((uniqueValues,occurCount),axis=0),fmt="%s")

# ##############################################################################
list_generated_sets_paths=[]
```

```
#dat
#list of dataset to augment
datasets_to_augment=['G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet/full_augmented/augmented_full_10.h5',\
'G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet/full_augmented/augmented_full_20.h5',\
'G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet/full_augmented/augmented_full_30.h5',\
'G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet/full_augmented/augmented_full_40.h5',\
'G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet/full_augmented/augmented_full_50.h5']

#get dataset to augment
augmentation=False
if augmentation==True:
    for augmentation_set in [0,1,2,3,4]:
        path_dataset_to_be_augmented=datasets_to_augment[augmentation_set]
        print (path_dataset_to_be_augmented)
        analysis_set=True
        if analysis_set==True:
            #get the dataset we want to increase and the subset within it
            dataset= h5py.File(path_dataset_to_be_augmented, 'r')
            attributes_training=np.array(dataset.get('attributes_training'))
            attributes_testing=np.array(dataset.get('attributes_testing'))
            labels_training=np.array(dataset.get('labels_training'))
            labels_testing=np.array(dataset.get('labels_testing'))
            dataset.close()
            uniqueValues, occurCount = np.unique(labels_training, return_counts=True)
            print (attributes_training.shape)
            print (uniqueValues)
            print (occurCount)

        if augmentation_set == 0:
            random_generation=3

        if augmentation_set == 1:
            random_generation=7

        if augmentation_set == 2:
            random_generation=10

        if augmentation_set == 3:
            random_generation=13

        if augmentation_set == 4:
            random_generation=18


```

```
        filter_random=random.sample(range(1, 40), random_generation)
        #results path
        results_path='G:/gans_project_root_directory/processed_data/augmented_voxels_dataset_voxnet'

        augmented_dataset_name_list=['augmented_0.20_dataset','augmented_0.40_dataset',\
        'augmented_0.60_dataset','augmented_0.80_dataset','augmented_full dataset']

        ############ AUGMENTATION ######################################################
        generated_dataset_path='G:/gans_project_root_directory/processed_data/gans_results/13/checkpoints_and_arrays/generated_data_array_5250.h5'
        #augmenting the data
        augmentation=True
        filter_random=random.sample(range(1, 40), random_generation)
        if augmentation==True:
            merge_dataset_with_augmented(generated_dataset_path,path_dataset_to_be_augmented,\
            filters=filter_random, filter=True, label=13, reduce_voxels=True)

        generated_dataset_path='G:/gans_project_root_directory/processed_data/gans_results/13/checkpoints_and_arrays/generated_data_array_5250.h5'
        #augmenting the data
        augmentation=True
        filter_random=random.sample(range(1, 40), random_generation)
        if augmentation==True:
            merge_dataset_with_augmented(generated_dataset_path,path_dataset_to_be_augmented,\
            filters=filter_random, filter=True, label=13, reduce_voxels=True)

        generated_dataset_path='G:/gans_project_root_directory/processed_data/gans_results/13/checkpoints_and_arrays/generated_data_array_4900.h5'
        #augmenting the data
        augmentation=True
        filter_random=random.sample(range(1, 40), random_generation)
        if augmentation==True:
            merge_dataset_with_augmented(generated_dataset_path,path_dataset_to_be_augmented,\
            filters=filter_random, filter=True, label=13, reduce_voxels=True)


#################################################################################

```

```
augmentation_original=False
if augmentation_original==True:
    path_dataset_to_be_augmented=datasets_to_augment[4]
    print (path_dataset_to_be_augmented)
    ####################  To delete
   ############################################
    original_set_path='G:/gans_project_root_directory/processed_data/
   voxel_datasets/merged_dataset.h5'
    for label_loop in [0,1,2,3,4,5,6,7,8,9,10,11,12,13]:
        filters_random=filter_random=random.sample(range(1, 40), 5)
        merge_dataset_with_original(original_set_path,
   path_dataset_to_be_augmented,\
        label=label_loop, filters=filters_random, filter=True,
   reduce_voxels=False)
    for label_loop in [11,12]:
        filters_random=filter_random=random.sample(range(1, 40), 2)
        merge_dataset_with_original(original_set_path,
   path_dataset_to_be_augmented,\
        label=label_loop, filters=filters_random, filter=True,
   reduce_voxels=False)


analysis_set=False
if analysis_set==True:
    #get the dataset we want to increase and the subset within it
    dataset= h5py.File(path_dataset_to_be_augmented, 'r')
    attributes_training=np.array(dataset.get('attributes_training'))
    attributes_testing=np.array(dataset.get('attributes_testing'))
    labels_training=np.array(dataset.get('labels_training'))
    labels_testing=np.array(dataset.get('labels_testing'))
    dataset.close()

    uniqueValues, occurCount = np.unique(labels_training, return_counts=
   True)
    print (attributes_training.shape)
    print (uniqueValues)
    print (occurCount)
    #np.savetxt(\
```

Listing 4: Augmentation and visualisation

```
#Import standard libraries
import os
import numpy as np
import pandas as pd
import sys
import csv
import requests
import xml.etree.ElementTree as ET
import xmltodict
import shutil
import h5py
import open3d as o3d
from mpl_toolkits.mplot3d import Axes3D
```

```python
import numpy as np
import matplotlib.pyplot as plt

#glob lists the elements in the current directory with a specific pattern
import glob
from matplotlib import pyplot as plt
import matplotlib.patches as patches
from voxelgrid import VoxelGrid
from mpl_toolkits.mplot3d import Axes3D
from sklearn import preprocessing


#functions we are using
#fucntion to manage directories
def get_list_elements_pattern_not_current_directory(directory_to_search, pattern):
    #Comprehension list that by given a directory, explores
    pattern_files= [element for element in os.listdir(directory_to_search) if\
    element.endswith("."+pattern)]
    #
    return pattern_files

def get_list_elements_without_pattern_not_current_directory(directory_to_search):
    #Comprehension list that by given a directory, explores
    files= [element for element in os.listdir(directory_to_search)]
    #
    return files

#funntion to get automatically the path of a given file in the curren directory
def generate_directory_path_contains_current_directory(folder_name):
    #Get current directoy
    current_directory=os.getcwd()
    #Create a directory path to explore
    directory_to_explore=current_directory+'/'+folder_name
    #Create a directory path to explore
    return directory_to_explore

def create_folder_in_path_check_folder_created(path_creation, path_to_create):
    #
    directories_in_directory_where_eant_create=\
    get_list_elements_without_pattern_not_current_directory(path_creation)
    #
    directories_path_in_directory_where_eant_create=\
    [path_creation+'/'+path for path in \
    directories_in_directory_where_eant_create]
    #
    if path_to_create not in directories_path_in_directory_where_eant_create:
```

```
        #
        os.mkdir(path_to_create)

def open_obj_to_data_frame(obj_file):
    data = pd.read_csv(obj_file,delimiter=' ',names=['cat','x','y','z'],\
    skiprows=2)

    data_frame_point_cloud=data.loc[data['cat'] == 'v']

    return data_frame_point_cloud[['x','y','z']]

def normalize_dataframe_to_array(dataframe):
    #Name of the coordinates. We use this to parse the dataframe
    coordinates=['x','y','z']

    #Shape of the dataframe.We use this info to create a numpy array with
    #the same characteristics
    number_points=dataframe.shape[0]
    number_coordinates=dataframe.shape[1]

    #Create a dataframe
    normalised_pointcloud_array=np.zeros((number_points,number_coordinates))

    counter=0
    #Loop through the coordinate
    for coordinate in coordinates:
        #Get the column we want to normalise
        column_to_normalise=np.array(dataframe[coordinate].values.\
        astype(float)).reshape(-1,1)

        #Get the normalizer
        min_max_scaler=preprocessing.MinMaxScaler()

        #Normalise the column
        normalised_column= min_max_scaler.fit_transform(column_to_normalise)

        #put the normalised column into the normalised array
        normalised_pointcloud_array[:,counter]=normalised_column.flatten()

        #Add one to the counter
        counter=counter+1

    return normalised_pointcloud_array

#Function to voxelize a single point cloud using the functions in the voxelgrid
def cloud_voxelize_binary_values(poin_cloud,voxgrid_dimension=[32,32,32]):
    #Get the voxel object
    grid=VoxelGrid(poin_cloud, x_y_z=voxgrid_dimension)
```

```
    #From the voxel object get an array that indicates the number of point
    #within each boxel
    dimensional_cuadatric_array=np.array(grid.vector)

    #if a voxel is not empty assign value 1 to the voxel. Otherwise, assign the
    #value 0
    dimensional_cuadatric_array=np.where(dimensional_cuadatric_array>0,1,0)
    #
    number_voxels=voxgrid_dimension[0]*voxgrid_dimension[1]*\
    voxgrid_dimension[2]
    #
    dimensional_cuadatric_array=\
    dimensional_cuadatric_array.reshape(1,number_voxels)
    #
    return dimensional_cuadatric_array


#Set up directories
#root directory
current_directory=os.getcwd()
#directory where the data is stored
data_directory='G:/gans_project_root_directory/hips/50004_hips/'
#from the data directory get the name of our data

data_names=\
get_list_elements_pattern_not_current_directory(data_directory,'obj')

#set up directory where we are going to save the visualisations
visualizations_directory='G:/gans_project_root_directory/visualizations'
#create the visualization directory

create_folder_in_path_check_folder_created(current_directory,\
visualizations_directory)

for element_index in range(len(data_names)):
    #get the obj that we want to
    obj_item_path=data_directory+'/'+data_names[element_index]

    #open the file and transform it to a normalise pointcloud
    point_cloud=open_obj_to_data_frame(obj_item_path)
    point_cloud=normalize_dataframe_to_array(point_cloud)

    #od3 object and point cloud plotting
    three_dimensional_object=o3d.geometry.PointCloud()
    three_dimensional_object.points=o3d.utility.Vector3dVector(point_cloud)
    o3d.visualization.draw_geometries([three_dimensional_object])

    #voxels 64
```

```
    sixty_four_voxel=\
    cloud_voxelize_binary_values(point_cloud,voxgrid_dimension
    =[64,64,64])

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.grid(False)
    plt.axis('off')
    ax.voxels(sixty_four_voxel.reshape((64,64,64)),facecolors='aqua',
    edgecolor="k")
    plt.show()

    #voxel 32
    three_two_voxel=\
    cloud_voxelize_binary_values(point_cloud,voxgrid_dimension
    =[32,32,32])

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.grid(False)
    plt.axis('off')
    ax.voxels(three_two_voxel.reshape((32,32,32)),facecolors='aqua',
    edgecolor="k")
    plt.show()
```

Listing 5: Visualisation of obj files and triangular meshes

```
#Import standard libraries
import os
import numpy as np
import pandas as pd
import sys
import csv
import requests
import xml.etree.ElementTree as ET
import xmltodict
import shutil
import h5py

#glob lists the elements in the current directory with a specific pattern
import glob
from matplotlib import pyplot as plt
import matplotlib.patches as patches
from voxelgrid import VoxelGrid
from mpl_toolkits.mplot3d import Axes3D
from sklearn import preprocessing

def get_list_elements_without_pattern_not_current_directory(
    directory_to_search):
    #Comprehension list that by given a directory, explores
    files= [element for element in os.listdir(directory_to_search)]
    #
    return

```

```
def get_list_elements_pattern_not_current_directory(directory_to_search,
    pattern):
    #Comprehension list that by given a directory, explores
    pattern_files= [element for element in os.listdir(directory_to_search
    ) if\
    element.endswith("."+pattern)]
    #
    return pattern_files

def get_list_elements_without_pattern_not_current_directory(
    directory_to_search):
    #Comprehension list that by given a directory, explores
    files= [element for element in os.listdir(directory_to_search)]
    #
    return files

#funntion to get automatically the path of a given file in the curren
    directory
def generate_directory_path_contains_current_directory(folder_name):
    #Get current directoy
    current_directory=os.getcwd()
    #Create a directory path to explore
    directory_to_explore=current_directory+'/'+folder_name
    #Create a directory path to explore
    return directory_to_explore

def create_folder_in_path_check_folder_created(path_creation,
    path_to_create):
    #
    directories_in_directory_where_eant_create=\
    get_list_elements_without_pattern_not_current_directory(path_creation
    )
    #
    directories_path_in_directory_where_eant_create=\
    [path_creation+'/'+path for path in \
    directories_in_directory_where_eant_create]
    #
    if path_to_create not in
    directories_path_in_directory_where_eant_create:
        #
        os.mkdir(path_to_create)

#fucntion to create folders in a given path
def create_folders_in_path(path,folder_names_list):
    #create folders in a given path. The folders names are given by a
    list
    for folder_name in folder_names_list:
        path_new_directory= path+ '/'+ folder_name
        create_folder_in_path_check_folder_created(path,
    path_new_directory)


########################      OBJ TRANSFORMATION
```

```
########################
#
def open_obj_to_data_frame(obj_file):
    data = pd.read_csv(obj_file,delimiter=' ',names=['cat','x','y','z'],\
    skiprows=2)

    data_frame_point_cloud=data.loc[data['cat'] == 'v']

    return data_frame_point_cloud[['x','y','z']]

#
def normalize_dataframe_to_array(dataframe):
    #Name of the coordinates. We use this to parse the dataframe
    coordinates=['x','y','z']

    #Shape of the dataframe.We use this info to create a numpy array with
    #the same characteristics
    number_points=dataframe.shape[0]
    number_coordinates=dataframe.shape[1]

    #Create a dataframe
    normalised_pointcloud_array=np.zeros((number_points,
    number_coordinates))

    counter=0
    #Loop through the coordinate
    for coordinate in coordinates:
        #Get the column we want to normalise
        column_to_normalise=np.array(dataframe[coordinate].values.\
        astype(float)).reshape(-1,1)

        #Get the normalizer
        min_max_scaler=preprocessing.MinMaxScaler()

        #Normalise the column
        normalised_column= min_max_scaler.fit_transform(
    column_to_normalise)

        #put the normalised column into the normalised array
        normalised_pointcloud_array[:,counter]=normalised_column.flatten
    ()

        #Add one to the counter
        counter=counter+1

    return normalised_pointcloud_array


#Function to voxelize a single point cloud using the functions in the
    voxelgrid
def cloud_voxelize_binary_values(poin_cloud,voxgrid_dimension=[32,32,32])
    :
    #Get the voxel object
```

```
    grid=VoxelGrid(poin_cloud, x_y_z=voxgrid_dimension)

    #From the voxel object get an array that indicates the number of point
    #within each boxel
    dimensional_cuadatric_array=np.array(grid.vector)

    #if a voxel is not empty assign value 1 to the voxel. Otherwise, assign the
    #value 0
    dimensional_cuadatric_array=np.where(dimensional_cuadatric_array>0,1,0)
    #
    number_voxels=voxgrid_dimension[0]*voxgrid_dimension[1]*\
    voxgrid_dimension[2]
    #
    dimensional_cuadatric_array=\
    dimensional_cuadatric_array.reshape(1,number_voxels)
    #
    return dimensional_cuadatric_array

#
def create_array_labels(label,number_instances):
    #
    label_array=np.full((number_instances),label)
    #
    return label_array

#
def transform_cloud_points_into_single_file_voxels(path_contains_folders_we_want_analyse,\
list_folders_to_parse,labels_list,voxgrid_size=[32,32,32]):
    #
    number_folders_explore=len(list_folders_to_parse)
    #get the path of the folders that we want to explore given a path and the
    #name of the folders
    paths_to_explore=[path_contains_folders_we_want_analyse+'/'\
    'results'+'/'+folder_to_parse for folder_to_parse in list_folders_to_parse]

    #
    results_path=path_contains_folders_we_want_analyse+'/'+'results_voxels'
    #
    create_folder_in_path_check_folder_created(\
    path_contains_folders_we_want_analyse,results_path)

    #create the paths of the folder to store the voxels
    results_point_cloud_directories=[path_contains_folders_we_want_analyse+'/'+\
    'results_voxels'+'/'+folder_to_analyse+'_'+'voxels' for folder_to_analyse in\
```

```
    list_folders_to_parse]

    #create the folders to store the results
    for directory in results_point_cloud_directories:
        create_folder_in_path_check_folder_created(\
        results_path,directory)

    #
    for folder_to_explore_index in range(number_folders_explore):
        #
        store_results_folder=results_point_cloud_directories\
        [folder_to_explore_index]
        #
        folder_to_explore=paths_to_explore[folder_to_explore_index]
        #
        label=labels_list[folder_to_explore_index]
        #
        action_of_analysis=list_folders_to_parse[folder_to_explore_index]
        #
        elements_in_folder_to_explore=\
        get_list_elements_pattern_not_current_directory(folder_to_explore,\
        'obj')
        #
        number_intems_to_voxelise=len(elements_in_folder_to_explore)
        #
        number_voxels=voxgrid_size[0]*voxgrid_size[1]*voxgrid_size[2]
        #
        voxel_matrix=np.zeros((number_intems_to_voxelise,number_voxels))
        #
        labels_array=create_array_labels(label,number_intems_to_voxelise)
        #
        for element_to_voxelize_index in range(number_intems_to_voxelise):
            element_to_voxelise=folder_to_explore+'/'+\
            elements_in_folder_to_explore[element_to_voxelize_index]
            #
            point_cloud=open_obj_to_data_frame(element_to_voxelise)
            #
            point_cloud=normalize_dataframe_to_array(point_cloud)
            #
            voxel_transformation=\
            cloud_voxelize_binary_values(point_cloud,voxgrid_dimension=voxgrid_size)
            #
            voxel_matrix[element_to_voxelize_index]=voxel_transformation
        #
        with h5py.File(store_results_folder+'/'+action_of_analysis+'.h5'\
        , 'w') as hf:
            hf.create_dataset('attributes', data=voxel_matrix)
            hf.create_dataset('labels', data=labels_array)
        hf.close()
```

```
    return 'done'


########################       IMPLEMENTATION #############################
#Folder to explore
folders_to_explore=['punching','running_on_spot','chicken_wings','hips',\
'knees','jumping_jacks','shake_arms','shake_shoulders','shake_hips',\
'one_leg_loose','one_leg_jump','light_hopping_loose','light_hopping_stiff',\
'jiggle_on_toes']
#Jump list
jump_list=[45,50,30,50,40,40,40,50,35,40,45,45,40,30]
#labels to assign to each action
labels_list=[0,1,2,3,4,5,6,7,8,9,10,11,12,13]
#Get current directory
current_directory=os.getcwd()

#retrieve_desired_actions(current_directory,folders_to_explore,jump_list,jump=3)

transform_cloud_points_into_single_file_voxels(current_directory,\
folders_to_explore,labels_list,voxgrid_size=[32,32,32])
```

Listing 6: Preprocessing: transform point clouds to voxels for multiple folders and delete the initial frames and smoothing of the frames

```
#Import standard libraries
import os
import numpy as np
import pandas as pd
import sys
import csv
import requests
import xml.etree.ElementTree as ET
import xmltodict
import shutil
import h5py

#glob lists the elements in the current directory with a specific pattern
import glob
from matplotlib import pyplot as plt
import matplotlib.patches as patches
from voxelgrid import VoxelGrid
from mpl_toolkits.mplot3d import Axes3D
from sklearn import preprocessing
from sklearn.model_selection import train_test_split

def get_list_elements_without_pattern_not_current_directory(directory_to_search):
    #Comprehension list that by given a directory, explores
    files= [element for element in os.listdir(directory_to_search)]
    #
    return files
```

```

def create_folder_in_path_check_folder_created(path_creation, path_to_create):
    #
    directories_in_directory_where_eant_create=\
    get_list_elements_without_pattern_not_current_directory(path_creation)
    #
    directories_path_in_directory_where_eant_create=\
    [path_creation+'/'+path for path in \
    directories_in_directory_where_eant_create]
    #
    if path_to_create not in directories_path_in_directory_where_eant_create:
        #
        os.mkdir(path_to_create)

#fucntion to create folders in a given path
def create_folders_in_path(path,folder_names_list):
    #create folders in a given path. The folders names are given by a list
    for folder_name in folder_names_list:
        path_new_directory= path+ '/'+ folder_name
        create_folder_in_path_check_folder_created(path, path_new_directory)


def merge_hpy_file(results_root_directory,path_folders_with_files, list_folders_information_merge,\
reduce_set=False, percentage_to_reduce_dataset=0.80):
    #
    paths_to_explore=[path_folders_with_files+'/'+folder_analysis+'_point_cloud' for\
    folder_analysis in list_folders_information_merge]
    #
    directory_to_create_results=results_root_directory+'/'+'merged_data'
    #
    create_folder_in_path_check_folder_created(results_root_directory,\
    directory_to_create_results)
    #
    number_paths_to_explore=len(paths_to_explore)
    #
    path_to_explore=paths_to_explore[0]
    #
    file_name=list_folders_information_merge[0]
    #
    file_name_path=path_to_explore+'/'+file_name+'.h5'
    #
    hf = h5py.File(file_name_path, 'r')
    #
    attributes= np.array(hf.get('attributes'))
    #
    labels=np.array(hf.get('labels'))
```

```
    #
    hf.close()
    #
    for path_to_explore_index in range(1,number_paths_to_explore):
        #
        path_to_explore=paths_to_explore[path_to_explore_index]
        #
        file_name=list_folders_information_merge[path_to_explore_index]
        #
        file_name_path=path_to_explore+'/'+file_name+'.h5'
        #
        hf = h5py.File(file_name_path, 'r')
        #
        attributes_to_concatenate=np.array(hf.get('attributes'))
        #
        attributes=np.concatenate((attributes,attributes_to_concatenate),
     axis=0)
        #
        labels_to_add=np.array(hf.get('labels'))
        #
        labels=np.concatenate((labels,labels_to_add), axis=0)
        #
        hf.close()

    if reduce_set == True:
        #
        attributes_to_maintain,attributes_to_delete,\
        labels_to_maintain,labels_to_delete=\
        train_test_split(attributes,labels,\
        test_size=percentage_to_reduce_dataset,stratify=labels,
    random_state=42)
        #
        percentage_data_kept=1-percentage_to_reduce_dataset
        print (attributes_to_maintain.shape)
        print (labels_to_maintain.shape)
        print (np.unique(labels_to_maintain))
        print (np.unique(labels_to_maintain, return_counts=True)[1])
        #
        attributes_training,attributes_testing,\
        labels_training,labels_testing=\
        train_test_split(attributes_to_maintain,labels_to_maintain,\
        test_size=0.20,stratify=labels_to_maintain,\
        random_state=42)
        #
        with h5py.File(directory_to_create_results+'/'+'merged_dataset_'
    +\
        str(round(percentage_data_kept,3))+'labelled_instances'+'.h5', 'w
    ') as hf:
            hf.create_dataset('attributes_training', data=
    attributes_training)
            hf.create_dataset('labels_training', data=labels_training)
            hf.create_dataset('attributes_testing', data=
    attributes_testing)
```

```
            hf.create_dataset('labels_testing', data=labels_testing)
        hf.close()
        print ('training')
        print (attributes_training.shape)
        print('testing')
        print (labels_training.shape)
        print (np.unique(labels_training))
        print (np.unique(labels_training, return_counts=True)[1])
        #
        return 'done'

    else:
        #
        attributes_training,attributes_testing,labels_training,labels_testing=\
        train_test_split(attributes,labels,test_size=0.20, stratify=labels,\
        random_state=42)
        #
        print (attributes_training.shape)
        print (labels_training.shape)
        print (np.unique(labels_training))
        print (np.unique(labels_training, return_counts=True)[1])
        #
        with h5py.File(directory_to_create_results+'/'+'merged_dataset'+'.h5'\
        , 'w') as hf:
            hf.create_dataset('attributes_training', data=attributes_training)
            hf.create_dataset('labels_training', data=labels_training)
            hf.create_dataset('attributes_testing', data=attributes_testing)
            hf.create_dataset('labels_testing', data=labels_testing)
        hf.close()

        return 'done'

'''
def create_training_testing_sets(hpy_file_path, path_store_splited_file):
    #
    hf = h5py.File(hpy_file_path, 'r')
    #
    attributes= np.array(hf.get('attributes'))
    print (attributes.shape)
    #
    labels=np.array(hf.get('labels'))
    print(labels.shape)
    #
    attributes_training,attributes_testing,labels_training,labels_testing=\
    train_test_split(attributes,labels,test_size=0.15, stratify=labels,\
    random_state=42)
    #
```

```
    hf.close()
    #
    with h5py.File(path_store_splited_file+'/'+'trainig_testing_dataset
    '+'.h5'\
    , 'w') as hf:

    return print('done')
'''


########################       IMPLEMENTATION
    #############################
#Folder to explore
folders_to_explore=['punching','running_on_spot','chicken_wings','hips',\
'knees','jumping_jacks','shake_arms','shake_shoulders','shake_hips',\
'one_leg_loose','one_leg_jump','light_hopping_loose','light_hopping_stiff
    ',\
'jiggle_on_toes']

#Jump list
jump_list=[45,50,30,50,40,40,40,50,35,40,45,45,40,30]
#labels to assign to each action
labels_list=[0,1,2,3,4,5,6,7,8,9,10,11,12,13]
#Get current directory
current_directory=os.getcwd()
#Create path to explore
path_explore=current_directory+'/'+'results_pointclouds'
#

merge_hpy_file(current_directory,path_explore,folders_to_explore)
#
splits=[0.8, 0.6, 0.4, 0.2]
for split in splits:
    print (split)
    merge_hpy_file(current_directory,path_explore,folders_to_explore,\
    reduce_set=True, percentage_to_reduce_dataset=split)
```

Listing 7: Preprocessing: create training and testing sets

```
## Code retrieved from https://www.kaggle.com/roestigraben/grid-of-voxels
    -to-train-linear-model
import numpy as np
import open3d as o3d
import h5py
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from voxelgrid import VoxelGrid
import random
import os


def get_list_elements_without_pattern_not_current_directory(
    directory_to_search):
    #Comprehension list that by given a directory, explores
```

```python
    files= [element for element in os.listdir(directory_to_search)]
    #
    return files

def create_folder_in_path_check_folder_created(path_creation, path_to_create):
    #
    directories_in_directory_where_eant_create=\
    get_list_elements_without_pattern_not_current_directory(path_creation)
    #
    directories_path_in_directory_where_eant_create=\
    [path_creation+'/'+path for path in \
    directories_in_directory_where_eant_create]
    #
    if path_to_create not in directories_path_in_directory_where_eant_create:
        #
        os.mkdir(path_to_create)

#fucntion to create folders in a given path
def create_folders_in_path(path,folder_names_list):
    #create folders in a given path. The folders names are given by a list
    for folder_name in folder_names_list:
        path_new_directory= path+ '/'+ folder_name
        create_folder_in_path_check_folder_created(path, path_new_directory)

def reduce_dimension_point_cloud(point_cloud_to_reduce):
    #create a od3 object
    point_cloud = o3d.PointCloud()
    #with the od3 trasnform the point cloud array into a od3 numpy array
    point_cloud.points = o3d.Vector3dVector(point_cloud_to_reduce)
    #o3d.draw_geometries([point_cloud])
    #reduce the dimension of the point cloud
    reduced_point_cloud= o3d.geometry.voxel_down_sample(point_cloud, voxel_size=0.035)
    #o3d.visualization.draw_geometries([reduced_point_cloud])
    #tranform the 3od object back into a numpy array
    reduced_point_cloud= np.asarray(reduced_point_cloud.points)
    #return the reduced point cloud
    return np.array(reduced_point_cloud)


#function to polish the shape of the point clouds
def modify_randomly_point_dimensions(point_cloud,dimension):
    difference_point_cloud_dimensions=dimension-point_cloud.shape[0]
    if difference_point_cloud_dimensions < 0:
        difference_point_cloud_dimensions=-difference_point_cloud_dimensions
        #generate random numbers between 0 and the dimension of the point cloud to
```

```
        #modify
        random_instances=random.sample(range(0,point_cloud.shape[0]),\
        int(difference_point_cloud_dimensions))
        #delete the instances randomly selected
        normalised_point_cloud=np.delete(point_cloud,random_instances,
    axis=0)

    else:
        #generate random numbers between 0 and the dimension of the point
     cloud to
        #modify
        random_instances=random.sample(range(0,point_cloud.shape[0]),\
        int(difference_point_cloud_dimensions))
        #retrieve the random instances from the point cloud
        retrieved_instances=point_cloud[random_instances]
        #concatenate the retrieved instances to the the point cloud
        normalised_point_cloud=\
        np.concatenate((point_cloud,retrieved_instances),axis=0)
    #return the normalised point cloud
    return normalised_point_cloud

#function to transform an entire array/dataframe
def reduce_point_cloud_dataset(dataset):
    #get the number of instances in the dataset
    number_instances_dataset=dataset.shape[0]

    #loop through all the intance in the dataset
    for element_to_process_index in range(number_instances_dataset):
        #First we retrieve the first element in the dataset and the we
        #concatenate more elements to it
        if element_to_process_index ==0:
            #get the point cloud to reduce dimensionallity
            element_to_process=dataset[element_to_process_index]
            #reduce the dimension of the point cloud
            reduced_point_cloud=reduce_dimension_point_cloud(
    element_to_process)
            #polish the shape of the point cloud
            reduced_point_cloud=\
            modify_randomly_point_dimensions(reduced_point_cloud,1800)
            # get the dimensions of the reduced point cloud
            rows_point_reduced_cloud=reduced_point_cloud.shape[0]
            columns_point_reduced_cloud=reduced_point_cloud.shape[1]
            #reshape the point cloud in a way that we can concatenate
    more
            #point clouds to it
            final_point_cloud_array=\
            reduced_point_cloud.reshape\
            ((1,rows_point_reduced_cloud,columns_point_reduced_cloud))
        #if is not the first element just append instances to the
    original
        else:
            element_to_process=dataset[element_to_process_index]
            #get the point cloud into a dataframe
```

```
            reduced_point_cloud=reduce_dimension_point_cloud(element_to_process)
            #polish the shape of the point cloud
            reduced_point_cloud=\
            modify_randomly_point_dimensions(reduced_point_cloud,1800)
            # get the dimensions of the reduced point cloud
            rows_point_reduced_cloud=reduced_point_cloud.shape[0]
            columns_point_reduced_cloud=reduced_point_cloud.shape[1]
            #reshape the point cloud and concatenate it to the main results
            #structure
            reduced_point_cloud=\
            reduced_point_cloud.reshape((1,\
            rows_point_reduced_cloud,columns_point_reduced_cloud))
            #concatenation
            final_point_cloud_array=\
            np.concatenate((final_point_cloud_array,reduced_point_cloud),axis=0)

    #return the entire modified set
    return final_point_cloud_array

################# IMPLEMENTATION ###############################################

#get current directory
current_directory=os.getcwd()
#directory to store results
results_directory=current_directory+'/'+'processed_point_clouds'

#create a directory to store the processed point clouds
create_folder_in_path_check_folder_created(current_directory,results_directory)

#directory with the data
data_directory=current_directory+'/'+'merged_data'

datafiles_to_process=['merged_dataset.h5',\
'merged_dataset_0.8labelled_instances.h5',\
'merged_dataset_0.6labelled_instances.h5',\
'merged_dataset_0.4labelled_instances.h5',\
'merged_dataset_0.2labelled_instances.h5']

datafiles_name=['full_pt','0.8 dataset_pt','0.6 dataset_pt','0.4 dataset_pt',\
'0.2 dataset_pt']

#get the number of databases
number_datasets=len(datafiles_to_process)

#loop through all the data
for dataset_index in range(number_datasets):
    #get the dataset name
```

```
    dataset_name=datafiles_to_process[dataset_index]

    #get the data
    dataset= h5py.File(data_directory+'/'+dataset_name , 'r')
    attributes_training=np.array(dataset.get('attributes_training'))
    attributes_testing=np.array(dataset.get('attributes_testing'))
    labels_training=np.array(dataset.get('labels_training'))
    labels_testing=np.array(dataset.get('labels_testing'))
    dataset.close()

    #sanity check with print statements
    print (attributes_training.shape)

    #process the datasets
    attributes_training= reduce_point_cloud_dataset(attributes_training)
    attributes_testing= reduce_point_cloud_dataset(attributes_testing)

    #sanity check with print statements
    print (attributes_training.shape)

    #save the dataset
    with h5py.File(results_directory+'/'+datafiles_name[dataset_index]+'.h5'\
    , 'w') as hf:
        hf.create_dataset('attributes_training', data=attributes_training)
        hf.create_dataset('labels_training', data=labels_training)
        hf.create_dataset('attributes_testing', data=attributes_testing)
        hf.create_dataset('labels_testing', data=labels_testing)
    hf.close()
```

Listing 8: Algorithm point clouds to voxels