Hellenic Mediterranean University
Department of Informatics Engineering

APPLICATIONS OF MACHINE LEARNING AND OBJECT RECOGNITION IN VIRTUAL WORLDS

Thesis

Tzermia Chrysoula tp4702
Thesis advisor: Athanasios Malamos

1/12/2021

Content

1. Introduction

Machine learning and especially neural networks (NN) is becoming a standard and stable technology with several applications in our everyday life. Every time you block an email or mention it as spam, an NN algorithm may work behind your account to analyze the content and classify potential future emails with similar features to the unwanted messages. Another very common example is face lock screens. These applications use NNs in order to prevent people from accessing the devices.

Neural Networks (NN) are an important part of machine learning. The idea behind NNs is to simulate the way human brain neurons work. After training with large datasets of input data, neurons are able to perform recognition, classification or just decision tasks by doing endless correlations between features of inputs.

On the other hand Computer Graphics (CG) are here to create a better environment and interface for users. Nowadays their authors are aiming to represent the real world as realistic as possible. Laser scans as well as other similar machines (e.g cameras) have contributed to the easier creation of computer graphicsby capturing 3D data of existing objects. The evolution of computer graphics is remarkable and proportional to the requirements of users. Computer Graphics is considered as a mature technology and the applications has becoming more and more demanding.

3D objects may be represented in two ways. Usually, physical objects are digitized through 3d scanning and consequently are delivered into point clouds. However synthetic graphics are created by authors and usually with CAD application. Cad applications deliver 3D meshes. In a later chapter we explain the difference between point clouds and meshes.

Machine learning techniques are used to classify point clouds or meshes (geometries) by training algorithms using points or polygons trying to extract geometrical characteristics.

2. Background knowledge

Before we analyze the main theme of the thesis, let's define some general and important knowledge.

2.1. History

Keywords and phrases

The term **artificial intelligence (AI)** refers to intelligence functions performed by machines designed to reproduce the capabilities of the human brain through combination of algorithms. More specifically, artificial intelligence is what allows certain machines to perceive and respond to the environment around them in a way similar to human brain. This implies the ability to perform tasks such as learning and problem solving.

Machine learning is a collection of algorithms and methods that improve the efficiency of a machine in performing “intelligent” tasks such as Pattern Recognition, Feature Extraction, Prediction, Regression and Clustering.

Deep Learning is a machine learning approach in which a Neural Network (NN) is connected to numerous "levels" of basic processing units, one after the other, so that the entry into the system is successively through each of them.

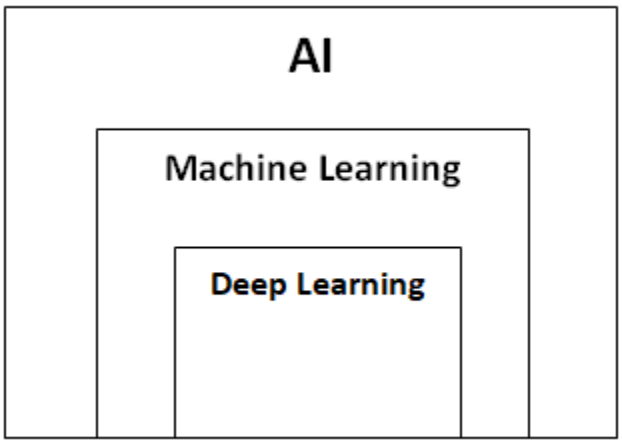


Figure 1: AI includes Machine Learning and Deep Learning is a part of Machine learning.

Historical evolution

Artificial intelligence has been present for more than 50 years. However, the evolution of computers, the availability of innumerable data and new algorithms have created new fields of growth and development. Initially, AI research focused on issues such as problem solving and symbolic methods. For example, in 2003, DARPA produced intelligent personal assistants, long before Siri, Alexa and Cortana became popular.

1950s - 1970s

First steps of neural Networks

people are impressed with “thinking” machines.

1980s – 2010s

The advancement of machine learning

Today

AI and Machine Learning explosion

Years of experiments lead to the “explosion: of artificial inteligence and machine learning applications.

2.2. Neural networks

Key words and phrases

Learning is one of the most important functions of human intelligence. Nowadays, both hardware and software have evolved and optimized to simulate this intelligence via Machine Learning

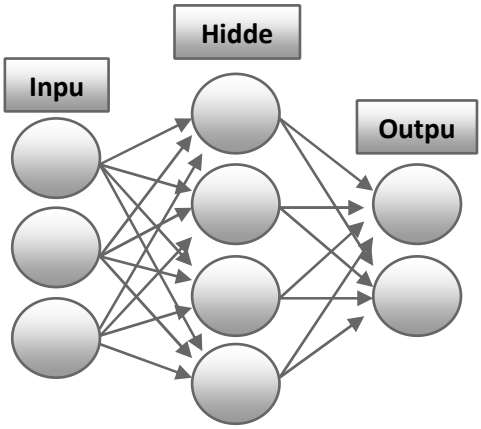
Segmentation is a technique in which several data are separated into categories based on one or more of their characteristics. During segmentation, each object belonging to a single class is highlighted with distinct hues to help computer vision recognize it.

Classification is the process of grouping items into distinct categories according to some common features.

In a neural network, **fully linked (FL) layers** are those in which all of the inputs from one layer are connected to each activation unit of the following layer, in contradiction to **partial connected** neural networks contain a portion of the total number of available connections for a given model.

Overview

In order to understand neural networks, let us suppose that computers are like curious toddlers who are now beginning to get to know the world. So first we try to give them as much information as possible, that's the input layer. The toddlers collect and combine the information they receive (Input layer), they remember more those who were given more emphasis (Hidden layer) and they draw conclusions (Output layer). In the rest of this chapter we further analyze the way that NN work.



Learning methods structured in layers are known as **neural**

figure 2: Neural Network architecture.

Neurons

Neurons are the building blocks of the network. Each such node receives a set of numeric inputs from different sources (either from other neurons or from the environment), performs a calculation based on these inputs, and produces an output. This output is either directed to the environment or fed as input to other neurons in the network. Basically, a neuron is a mathematical function,

$$f(x_1, x_2) = w_1x_1 + w_2x_2$$

known as **Linear combination of weight and inputs**, where $\boldsymbol{x}_k$ stands for the input number, $\mathbf{k}$ for the index of input and $\mathbf{w_k}$ for the Weights vector . As you can see from this function, the weight vector is assigned to each input number to show the uniqueness of each neuron. The output of this function is arithmetic.

Another useful function is the **activation function**. It shows the importance of the neuron's input for the prediction by activating the proper neuron. If $\mathbf{x_{ki}}$ is the **i-gate** input of the $\mathbf{k}$ neuron, $\mathbf{w_{ki}}$: the i-gate synaptic weight of k neuron and $\boldsymbol{\varphi(\cdot)}$ is the activation functions of the neural network , $\mathbf{y_k}$ is the output of the k neuron as seen on the above equation :

$$y_k = \varphi(\sum_{i=0}^{N}(x_{ki}w_{ki} + bias))$$

On the **k-gate** neuron there is an important weight $\mathbf{w_{k0}}$ named bias (is a form of neuron without input) or threshold with value $\mathbf{w_{k0}} = \mathbf{1}$. If the total sum of the other inputs of the neuron is bigger than this value, the neuron activated.

The **activation function** can be:

- step transfer function,

$$\varphi(\chi) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

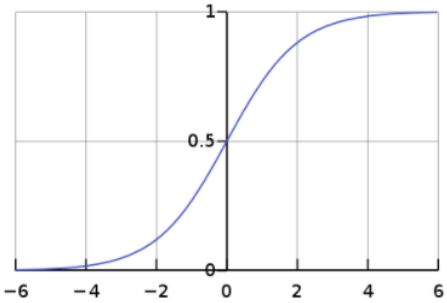


Figure 3: Sigmoid function

The step function is not considered useful as activation function for the Neural Networks because, according to math, it is derivative is infinite and that is a huge disadvantage. Therefore, there is the need to create an activation function similar to step transfer function but, at the same time, continuous and producible throughout their scope.

- linear transfer function,

$$\varphi(x) = ax$$

- non-linear transfer function
  The non-linear activation function commonly used in neural networks is called the sigmoid function. (Figure 3)
  The sigmoid function formula is

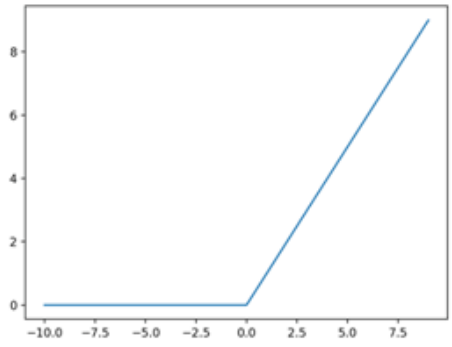


Figure 4: ReLU activation function

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} = 1 - S(-x)$$

and it can be used to scale our values between 0 and 1.

- Stochastic transfer function.
- ReLU function (Figure 4)
- etc

The main feature of neural networks is the ability to learn. By the word “learn” we define the gradual improvement of the net to solve problems, such as the gradual approach of a function. Learning is achieved through training, an iterative process of gradually adjusting the network parameters (usually its weights) to appropriate values to solve the problem under consideration with sufficient success. Once a network is trained, its parameters are usually "frozen" at the appropriate values and from this moment onwards it is in working order.

ReLU is also a very useful non-linear activation function.

$$ReLU(x) = max\ (x, 0)$$

ReLU stands for **Rectified Linear Unit**(Figure 4), it is a commonly used activation function when we refer to deep learning and it is preferable to the sigmoid function. The $\boldsymbol{x}$ value stands for the number we give as input, so if that number is greater than 0 we take that number, but if that number is smaller than 0 we take 0.

To sum up, neurons could be described from the following formula

$$f(x_1, x_1) = \ max\ (0, w_1x_1 + w_2x_2)$$

that takes one or more numbers as inputs and outputs, an activation number.

There are three types of neurons and layers: the input neurons (on the input layer), the output neurons (on the output layer) and the hidden neurons (on the hidden layer).

The **input neurons** are the combination between the environment inputs of the network and the hidden neurons. They do not perform any calculation. On the other hand, the **hidden neurons** multiply each input by the corresponding synaptic weight (which stands for the size or the durability of the connection between two nodes) and calculate the total sum of the products. This sum is fed as an argument to the chosen activation function, which is implemented internally by each node. The value that the function receives for this argument is also the output of the neuron for the current inputs and weights. Finally the **output neurons** transfer to the environment the final numerical output values of the network.

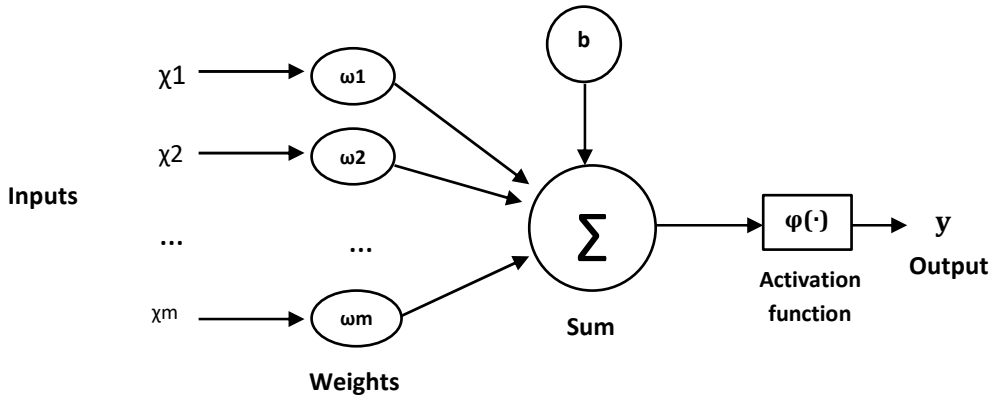


figure 5: Schematic diagram of an artificial neural network.

Training

Before we analyze the training methods, it is important to define the loss function. This function evaluates our neural network for a certain task.

$$L(y, \hat{y}) = \frac{1}{M} \sum_{i=0}^{m} (y, \hat{y})^2$$

M stands for the number of testing samples, y is the output we wish to receive from the network, $\hat{y}$ is what we actually received by running our example to the network and **i** is the index of a training data. In order to understand this function, let us suppose that we have a dataset of 3D chairs and 3D beds (two classes). We

create a class map where y =0 is the label for the chairs and y =1 is the label for the beds, in other words the number we wish to receive from the NN. However, while training NN the output $\hat{y}$is not always the appropriate, thus we ask for a chair and NN replies that it is a bed! Thus, the loss function is considering how many bad guesses we had.
The output value of the loss function must be as small as possible.

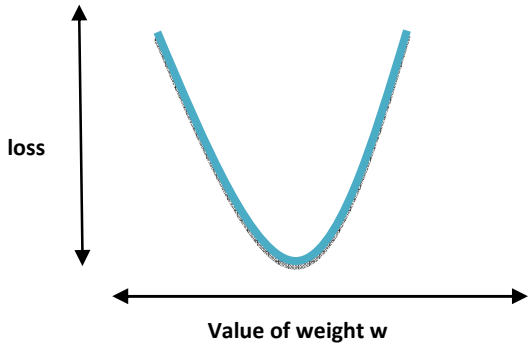


Figure 6: gradient of loss function

Now it is important to define the **Learning rate** (lr) and the **epoch**. The lr value should be small (often between 0.0 and 1.0) because we want to train NN in small steps, keeping loss function to small values if possible. lr is a positive number that further determines the frequency we update the weights in the hidden layer of the network. For instance, if the lr value is too small the weights are updating slowly so the training has slow progress and if this value is too large the model may not predict anything accurately because weights and the corresponding outputs "jump" into totally different values of loss function. A suggested number could be 0.1 or 0.01. That means we apply weight 0.1 or 0.01 to the parameters.
The image below is an example of what happens to the gradient loss function on each category.

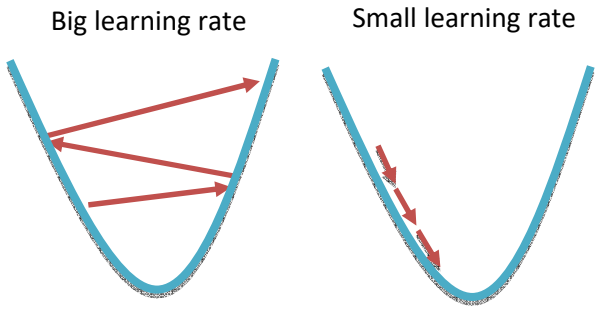


figure 7: Learning rate performance

An **epoch** is a machine learning term that refers to how many passes the machine learning algorithm has made across the full training dataset. **Batches** are commonly used to organize data collections (especially when the amount of

data is very large). The number of epochs equals the number of iterations if the batch size is the whole training dataset. In the construction of various models, several epochs are employed. When the dataset size is **d**, the number of epochs is **e**, the number of iterations is **i** and the batch size is **b**, the general relationship is

$$d \cdot e = i \cdot b$$

There are many algorithms whose application aims to adjust the values of the weights of an Artificial Neural Network. Learning methods can be classified into two categories: **supervised learning** and **unsupervised learning**.

The use of labeled datasets distinguishes **supervised learning** as a machine learning technique. These datasets are used to "train" or "supervise" algorithms so that they can effectively identify data and forecast outcomes. The model can track its accuracy and learn over time by using labeled inputs and outputs. Supervised learning is divided into two more categories: structural learning and temporal learning.

Unsupervised learning: The included algorithms of this category are called “self-organized” and they are procedures that do not require an "external" teacher or supervisor to be presented. Unsupervised learning analyzes and clusters unlabeled data sets using machine learning methods. These algorithms uncover hidden patterns in data. Some examples that represent unsupervised learning include the Hebbian algorithm, the min-max algorithm and the differential Hebbian algorithm.
Most of the training processes are offline.

Types of neural networks

Depending on the number of nodes on the hidden layer, neural networks separated into deep neural networks that are commonly used for deep learning and shallow or non-deep neural networks for simpler calculations.

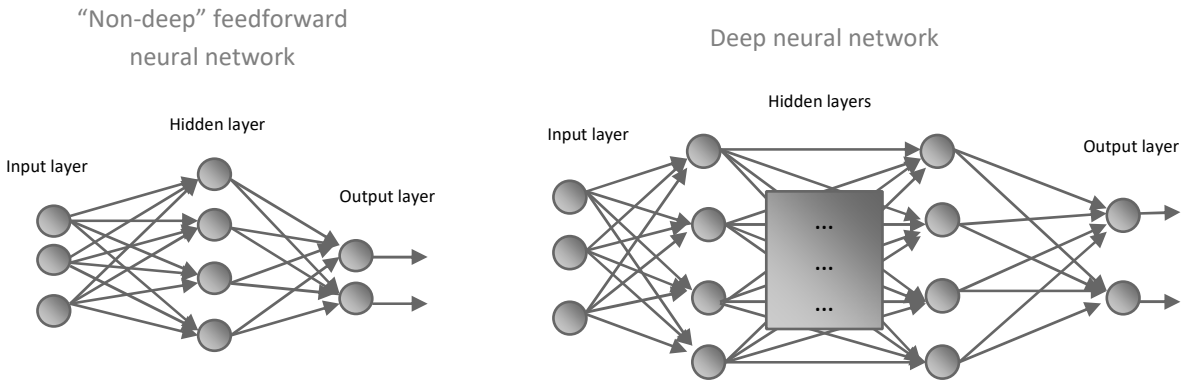


figure 8: Deep VS Shallow Neural Networks

The most famous deep neural networks are the convolutional neural networks (ConvNet or CNN) which are commonly used to process 2D data, such as images. These networks work by directly extracting features from data, a method that automatically makes models accurate from tasks of computer vision. Their architecture is simple and includes one input layer, many hidden layers with ReLu activation function and an output layer.

Another type of neural networks are the artificial neural networks. They are consisting of collections of artificial neurons inspired by the biological neural neural networks of human brain. Multi-layer perceptron is a type of feedforward artificial neural network (ANN). The name MLP is confusing, referring to networks built of many layers of perceptrons (with threshold activation) in some cases and any feedforward ANN in others.

The main distinction between a typical ANN and a CNN is that a CNN only has one completely linked layer (the last layer), whereas in an ANN, each neuron is connected to all other neurons.

Better accuracy and training time

In the fields of classification and recognition, deep neural networks are a game-changer. Deep networks have allowed robots to identify pictures, sounds, and even play games with an accuracy that is nearly unattainable for humans. To attain a high degree of accuracy, these networks must be trained using a large amount of data and, as a result, require computer power. Thus, reducing training time and enhancing model accuracy is a timeless target for the research community. As a result, in literature appear few simple rules that may help in optimization in time/accuracy curve.

Data Preprocessing
The fact that your neural network is only as good as the input data used to train it emphasizes the need of data pre-processing. Neural networks may not be able to attain the necessary degree of accuracy if critical data inputs are lacking. On the other hand, if data are not preprocessed appropriately, it may have an impact on the network's accuracy and performance in the future.

Data Normalization
Normalization is the process of transforming data into a scale that is consistent across all dimensions. The most common technique to achieve this is to split the data by the standard deviation of each dimension. It makes sense only if you have reason to suppose that distinct input features have different scales yet are equally important to the learning process.

Batch Normalization
Batch normalization is a method for training very deep neural networks that standardizes each mini-inputs batch's to a layer. This stabilizes the learning

process and significantly reduces the number of training epochs needed to create deep networks.

2.3. Computer Graphics

Key words and phrases

Vertex: a point (typically in 3D space), with features, such as coordinates, color, normal vector, and texture coordinates.

Edge: a line that connects two vertices.

Faces: a closed set of edges with three edges on a triangle face and four edges on a quad face. A coplanar collection of faces is known as a **polygon**. Polygons and faces are interchangeable in systems that enable multi-sided faces. A polygonal mesh may be thought of as an unstructured grid or undirected graph with extra geometry, form, and topological attributes.

Surfaces: a combined collection of faces in order to form a flat area.

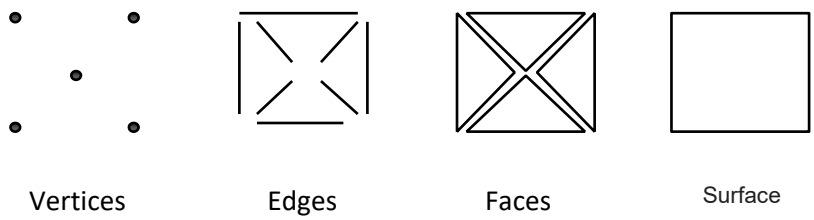


figure 9

Overview

Inputs of computer graphics are symbolic descriptions of the optical scene and the outputs are digital images or videos, with or without interaction with the user. Computer graphics have a variety of uses, for instance: on graphical user interfaces (GUI), video games, virtual reality (VR), movies, animations, on advertising and data visualization. Also, it is a very useful tool for the illustration of forms, architectural design and logos creation. The appropriate data can be entered into the computer via digitizing devices such as scanners, digital cameras, or via the keyboard and mouse. When the result is displayed on the screen, the user can manipulate it by moving it horizontally and vertically or by rotating it, editing it or extending it using a mouse or more specialized peripherals (such as light pen).

As we mentioned before, computer graphics create synthetic images from mathematical expressions of data (model). This model can be a description of an imaginary or natural digitized scene with polygonal surfaces. In order to create

an image it is moreover important to consider the way light interacts with model objects. Image processing and 3D modeling softwares are used to import, create and edit computer graphic with the use of dots, lines, curves, etc. This process is known as rendering.

Geometry

Geometry is the form of math used to describe the physical world. In order to project a three - dimensional object it uses X axis for the depth, forward and back, Y for the horizontal, side to side, and Z for the vertical, up and down (figure 10: three dimensions).

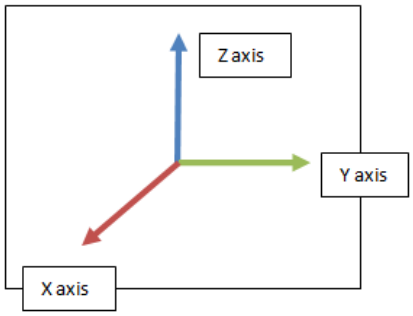


figure 10: three dimensions

Euclidean Space is the first Mathematical Space that was used to "house" the Geometric Shapes. In modern Mathematics, it is common to determine Euclidean Space by using Cartesian Coordinates and the theory of Analytic Geometry.

There are two basic transformations.

1. **Translation**, which means shifting the plane so that each of its points shifts in the same direction and at the same distance.
   For instance, to perform a translation of a point from position (x, y) to another (x_1, y_1) we add the shift vector (Tx, Ty), that represents the distance, to the original coordinates.

$$\boldsymbol{x_1} = \boldsymbol{x} + \mathbf{T}_{\boldsymbol{x}}$$
$$\mathbf{y_1} = \mathbf{y} + \mathbf{T_y}$$

   To perform a translation in tree-dimensions from (x, y, z) to position (x_1, y_1, z_1) we add the shift vector (Tx, Ty, Tz)

$$\boldsymbol{x_1} = \boldsymbol{x} + \mathbf{T}_{\boldsymbol{x}}$$
$$\mathbf{y_1} = \mathbf{y} + \mathbf{T_y}$$
$$\mathbf{z_1} = \mathbf{z} + \mathbf{T_z}$$

2. **Rotation,** in which each point on the plane rotates around a fixed point at the same angle.
   The matrix that represents the rotation around a **2D plane** is shown below.

$$R_\beta = \begin{bmatrix} \cos\beta & -\sin\beta \\ sin\beta & cos\beta \end{bmatrix}$$

And the rotation of a 2D vector in a plane is done as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The above matrix can be generalized in order to be useful for a **3D world** by adding a third coordinate. So if we would like to rotate a point about z-axis we use the following matrix,

$$R_{\beta,z} = \begin{bmatrix} cos\beta & -sin\beta & 0 \\ sin\beta & cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

From the other hand, if we wish to perform the same rotation to x-axis it is preferable to use the above one

$$R_{\beta,x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\beta & -sin\beta \\ 0 & sin\beta & cos\beta \end{bmatrix}$$

and, for our last case, to perform a rotation around y-axis we use the above formula

$$R_{\beta,y} = \begin{bmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{bmatrix}$$

Rigid transformation includes the above two transformations. It doesn't change the size or form of an input object.

Color

Another feature of graphics is the color. When light falls on a surface, it is either reflected or absorbed depending on the color and the surface reflectivity. White surfaces reflect the full spectrum of colored light, while black surfaces reflect nothing and absorb light. On the other hand, a shiny surface fully reflects while dull surfaces absorb mostly. These actions are perceived by the human eye and the brain as the color of the object.

Additive Color Model (RGB Model) describes a template that uses the three basic colors (where R stands for red, G for green and B for blue) which are visible because of the transmission of light.

Subtractive Color Model (CMYK Model) uses the reflection of light in order to describe object colors that are magenta (M), cyan (C), yellow (y) or one of their derivatives.

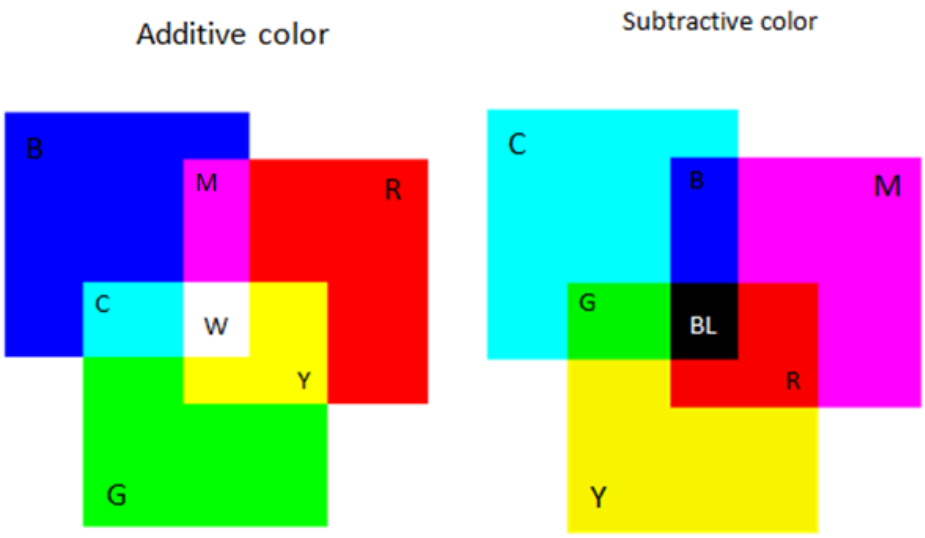


figure 11: Additive color VS Subtractive color

Direct Coding is the process used to apply color on a certain pixel by using a fixed amount of memory storage space. For example, if we use one bit for each main color then one pixel has 3 bits. Each bit can be equal to 0 (off) or 1 (on) , so , to create the color magenta we set the first bit (red) and the third bit (blue) equal to 1 and the second bit (green) equal to 0.

Color depth is the number of bits used to describe the color of each pixel (or area in the vector graphics).The current standard is 24-bit color depth for screens and 32-bit for prints (screen and printing are using different color patterns). Also, there are graphics with greater color depth, intended for special uses, as the human eye cannot distinguish more than 16.7 million color gradients. For the internet applications Bitmap Graphics are preferable because vector graphics are not supported by older versions of browsers that are still used by a relatively large percentage of internet users.

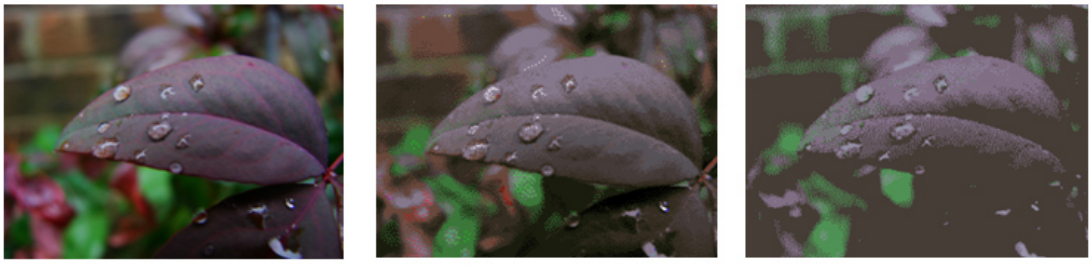


figure 12: Color Depth. The first image is an example of an 8 bit.png with 256 colors, the second is a 4 bit.png with 16 colors and the last one is a 2 bit.png with 4 colors.

Types of computer graphics

There are four types of computer graphics:

2D computer graphics (figure 13: 2D scene from the game "Super Mario".) are used to create graphical user interfaces (GUI), but also for illustrations of books, magazines and other publications. After their composition, they are stored in digital image files and their further processing is a part of digital image processing. Two-dimensional graphics can be divided into Vector Graphics and Bitmap Graphics.
The type of graphics is usually recognized by the extension of the file name in which they are stored (the part of the name to the right of the dot that separates the name of a file). The most common types are: ".svg", ".cdr", ".ai" (for Vector Graphics) and ".tif", ".bmp", ".jpg", ".gif", ".png" (for Bitmap Graphics).

3D computer graphics (figure 14: 3D scene from the game "Detroit Become Human".) are an attempt to display three-dimensional graphics on a two-dimensional screen of a digital device. Their function is based on the spatial description of three-dimensional objects through points and mathematical formulas in a coordinate system, and then displays the coordinates of their points in two dimensions during the performance phase. Such graphics are commonly used by programs such as computer games and virtual worlds.3D graphics are also used in cinema to compose scenes from virtual worlds and to create special effects using modern digital technology (instead of mechanical or additional effects).

figure 14: 3D scene from the game "Detroit Become Human".

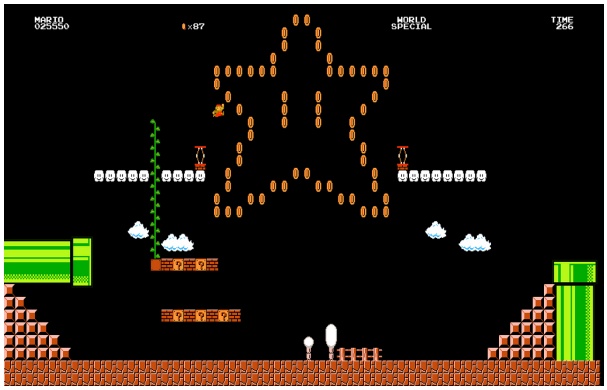


figure 13: 2D scene from the game "Super Mario".

Static computer graphics are budgeted and pre-processed graphic objects (coordinates of points and surfaces, their colors, lighting and textures)which are not rendered at the time they are displayed, but have been rendered once when created. Then they are stored and played as a video file, so they cannot be interactive. An example of such graphics are small videos, which are displayed in video games, and which have been "shot" once and each time we watch them remain the same. To create them we use a suitable program for creating graphics and animation, such as 3D Studio Max, Maya, Lightwave, Blender, Cinema4D, etc.

Real time computer graphics are graphic objects (coordinates of points and surfaces, colors, lights, and textures) that are rendered visually when a computer program is running, whenever that happens, by re-executing the appropriate commands / calculations by the processor. Displaying them requires a real-time graphics engine, such as Ogre3D, Crystal Space and game machines. Also, real-time graphics can be interactive, with the graphics machine responding appropriately to user inputs (from peripherals such as a mouse or keyboard), but this is not necessary. There are several standard libraries for programming them, such as OpenGL and Direct3D.

Mesh

Polygon meshes are a big part of computer graphics (particularly 3D computer graphics) and geometric modeling. Different polygon mesh representations are utilized for various purposes and aims. Boolean logic (Constructive solid geometry), smoothing, simplification, and many more operations may be done on meshes. Also, there are several algorithms for Ray tracing, collision detection, and rigid-body dynamics with polygon models.

A polygon mesh is a collection of vertices, edges, and faces that determines the geometry of a polyhedral object in 3D computer graphics. The rendering becomes easier with triangle faces (triangle mesh), quadrilaterals (quads), or other basic convex polygons (n-gons) but meshes can also be made up of concave polygons or even polygons with holes. The simplest mesh representation consists of a vertex list and a polygon list. Triangular elements are frequently used to define polygons. Triangles are useful in a variety of geometrical computations, including point inclusion checks, area and normal calculations, and interpolation of vertex characteristics, because they are always both planar and convex.

The three-dimensional coordinates of the mesh vertices defined in an appropriate coordinate frame are stored in the vertex list, while the polygon list includes integer values that index into the vertex list. The front facing side of each polygon is usually indicated by an anticlockwise arrangement of vertices with respect to the outward face normal direction. In lighting computations and culling processes, the distinction between the front and rear sides of a polygon becomes crucial. If the polygon list represents a set of linked triangles can be employed a triangular strip which represents a more efficient and compact data structure. The first triangle is identified by the first three indices in a triangle strip. Along with the previous two indices, the fourth index reflects.

In contrast to polygon meshes, volumetric meshes openly represent both the surface and volume of a structure, whereas polygon meshes only clearly represent the surface (the volume is implicit).

The mesh geometry is specified by the model definition files, which contain information on vertices, polygons, color values, texture coordinates, and perhaps many more vertex and face related properties. The topology of the mesh is defined by the adjacency and incidence connections between mesh components, which are widely employed by numerous mesh processing techniques.

The assumption that the provided mesh is a polygonal manifold is prevalent in the development of mesh data structures and associated algorithms. A polygonal manifold is a mesh that meets two criteria: no edge is shared by more than two faces, and faces sharing a vertex may be sorted so that their vertices excluding the shared vertex form a simple chain.

A non-manifold mesh can include edges shared by more than two polygons or vertices with several chains of neighboring vertices. Many mesh processing methods struggle to conduct local alterations around a vertex in a non manifold mesh because the neighborhood of that vertex may not be topologically equivalent to a disc. The methods in this article presume that the provided mesh meets the polygonal manifold criteria.

Mesh data is stored and shared in graphics applications using a variety of file formats. A variety of such file formats save values in binary and compressed formats to save space.

Off file format is a geometry definition file format that contains the description of a geometric object's constituent polygons. It can hold 2D or 3D objects, and it can also represent higher-dimensional things with simple additions. The basic standard was initially established for Geomview, a geometry visualization software, but it has since been adopted by other softwares. The structure of this format is shown above.

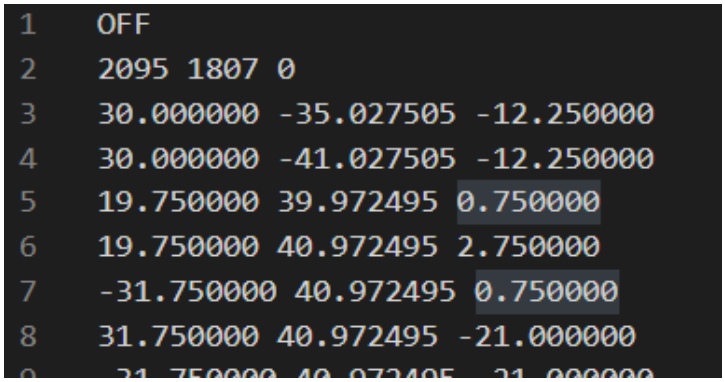


figure 15: view of an object in “off” file format

The header keyword OFF should be on the first line. Optional remark lines beginning with the character # can be added after this line. The entire number of vertices, faces, and edges should be represented by three integer values nv, nf, and one on the first non-comment line. The number of edges (ne) is never more than zero. The vertex list comes after the line above. The list's number of vertices must equal the number nv. The index 0 is assigned to the initial vertex, while the index nv—1 is assigned to the last vertex.

```
2097    -26.143000 34.260195 0.750000
2098    3 4 5 6
2099    3 5 4 7
2100    3 8 9 10
2101    3 9 8 11
2102    3 12 13 14
```

figure 16

The face list comes after the vertex list. Each line has a collection of integers n, i1, i2,...,in, where n is the number of vertices of that face and the remaining numerals are the face indices. Color values can be added to each face as 3 or 4 integer values in the range [0, 255] or floating-point values in the range [0, 1] in either RGB or RGBA notation.

Point Cloud

A point cloud is a large collection of small individual points plotted in three dimensions. Each virtual point on a wall, window or any other surface the laser beam comes into touch with would represent a real point.

The scanner automatically calculates a 3D X, Y, Z coordinate location for each point using the vertical and horizontal angles formed by the laser beam to provide a set of 3D coordinate measurements that frequently include the color value recorded in RGB and intensity. These characteristics can then be converted into a digital 3D model that provides more information.

figure 17: point cloud of a chair included in ModelNet10

The more points in the representation, the more detailed it is, allowing minor features and texture details to be specified more clearly and accurately.

The method of aligning point clouds with 3D models or other point clouds is known as point set registration. The point cloud of a product can be matched to an existing model and examined to check for changes in industrial metrology or inspection using industrial computed tomography. The point cloud may also be used to obtain geometric measurements and tolerances.

While point clouds may be viewed and studied directly, they are frequently transformed to polygon mesh or triangle mesh models, or CAD models via a process known as surface reconstruction.

Converting a point cloud to a 3D surface may be done in a variety of ways. Some methods, such as Delaunay triangulation, alpha forms, and ball pivoting generate a network of triangles over the point cloud's existing vertices, whilst others transform the point cloud to a volumetric distance field and rebuild the implicit surface by using a marching cubes technique.

Scanners generate raw data in a variety of forms. For 3D modeling, there are hundreds of file types to choose from. Some of these file types are compatible with different processing applications, and each piece of software has varied exporting capabilities.

What you intend to do with the data and who requires it is also related to output formats. If you want to keep the data, you should save it as an ASCII file, which saves the point cloud as a simple, generic collection of XYZ coordinates that you may even open in a text document as a last option. However, keep in mind that ASCII eliminates any color or vector.

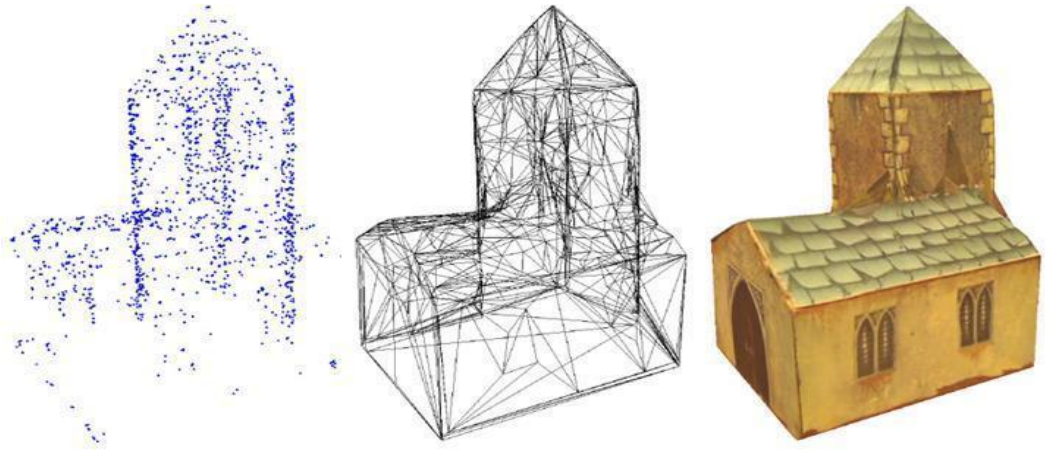


figure 18: Point cloud converted to triangles and then mesh

XYZ files are ASCII or binary database files with a separator character between each column in a row. Within a point cloud, each row represents a point. Each column represents one of the point's component points.

By convention, these files have filename extensions of ".xyz" , ".csv", or ".txt", although the Point Cloud XYZ may read and write files with any extension.

Voxel

The terms volumetric and pixel are combined to form the word voxel. In three-dimensional space, a voxel may be thought of as a 3D pixel. It represents a three-dimensional picture (or several slices of two-dimensional images) that shows a volume, similar to how a pixel is an element of a two-dimensional image.

A voxel is a single sample, or data point, on a three-dimensional grid with uniform spacing. This data point can be made up of a single item of information, such as opacity, or many pieces of information, such as color and opacity.
The space between each voxel is not recorded in a voxel-based dataset;
each voxel represents only a single point on this grid, not

a volume. This missing information may be rebuilt and/or estimated, e.g. via interpolation, depending on the kind of data and the dataset's intended application.

A three-dimensional shape in the form of a mesh can be represented as a possible distribution of binary values in a three-dimensional grid. We define the inside of a grid surface using a voxel value = 1 and the outside (or empty space) = 0.

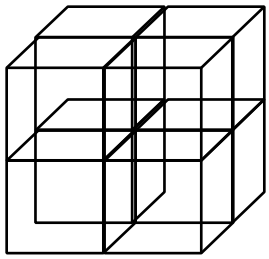


figure 19: Voxel grid

Voxels are commonly employed in medical and scientific data visualization and analysis.

Rendering

Rendering is called the process in which there is a realistic display of models and environments, using colors, textures, lighting and shading. The time required to complete the model-space is proportional to its complexity. The program used for this process is called renderer. The production process of the final photorealistic depicted scene is a complex process because there are many parameters that must be taken into account in order to produce a result that is close to reality. The rendering process can last fractions of a second or up to a whole day to produce a single image per frame.

The rendering formula

$$L_0(x, w_0, \lambda, t) = L_e(x, w_0, \lambda, t) + \int_\Omega f_r(x, w_i, w_0, \lambda, t) L_i(x, w_i, \lambda, t)(w_i \cdot n) dw_i$$

Where **n** stands for the normal surface, $\mathbf{w_i} \cdot \mathbf{n} = \cos\theta_i$, $\mathbf{L_0}$ stands for the output light, $\mathbf{L_e}$ is the emitted light and $\mathbf{L_i}$ stands for the incoming light. So $\mathbf{L_0(x, w_0, \lambda, t)}$ describes the outward directed total spectral radiance of **λ** wavelength for **x** position to $\mathbf{w_0}$ direction at **t** time (= output spectral radiance).

The sampling problem is a challenge that any rendering system must cope with. Essentially, the rendering process uses a finite number of pixels to portray a continuous function from picture space to colors. Any spatial waveform that may be exhibited must have at least two pixels, which is proportional to picture resolution, according to the Nyquist–Shannon sampling theorem (or Kotelnikov theorem). In basic words, this means that an image cannot contain characteristics, such as color or intensity peaks or troughs, that are smaller than one pixel.

High frequencies in the picture function will generate unpleasant aliasing in the final image if a naïve rendering technique is applied without any filtering. Aliasing is most commonly seen as jaggies, or jagged edges on objects with visible pixel grids. All rendering algorithms (if they are to generate good-looking images) must employ some form of low-pass filter on the image function to eliminate high frequencies, a process known as antialiasing, in order to remove aliasing.

3. *Application*

Keywords and phrases

The fully connected **FC layers** of the network are used to identify certain global configurations of the characteristics observed by the lower levels. They are commonly found at the top of the network architecture, after the input has been condensed to a compact representation of features (by the previous, usually convolutional layers). Each node in the FC layer learns its own set of weights from the nodes below it.

Dropout layers are crucial in CNN training because they prevent the training data from being overfit.

Overfitting is a term used in data science to describe when a statistical model although fits its training data perfectly, we still training.

Support-vector machines (SVM) are supervised learning models that examine data for classification and regression analysis, along with accompanying learning algorithms.

Cross-validation is a resampling approach that tests and trains a model on different iterations using different sections of the data.

Softmax is an activation function that outputs the probability for each class summing up to 1.

TensorFlow: open-source library included in Python supported by Google Brains.

Keras: sub library included in TensorFlow. Proper for Deep learning.

3.1. *Theoretical analysis*

APIs and datasets

The main idea of this thesis is the 3D object detection, via cloud of points. To perform this task we use deep learning algorithms implemented in a python library developed by the Google team of artificial intelligence (Google Brain), called **Tensorflow**.

Keras is a simple, flexible and powerful application programming interface included in Tensorflow library. It is an open-source software ideal for deep neural networks and designed to enable fast calculations.

For the implementation it was necessary to use a large dataset with 3D objects. So after some research we find the ModelNet10, a dataset created for

computer vision research tasks, computer graphics, and robots. It includes a comprehensive collection of 3D object in ".off" file format.

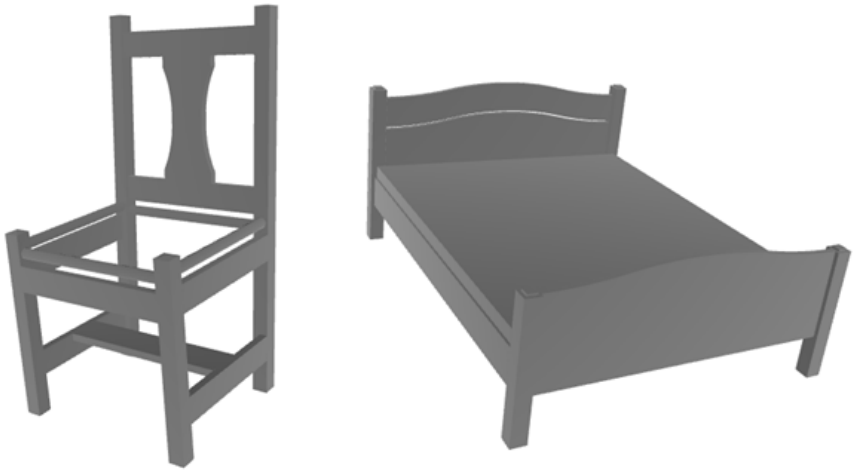


figure 20: ".off" file format objects included in ModelNet

ModelNet10 is a dataset with 10 categories (classes). Each of them includes a large amount of data splitted into train and test files. There is also a bigger dataset named ModelNet40 that includes the objects from ModelNet10 and 30 more categories.

Max - pooling / pooling

Max pooling is a discretization method based on samples. The goal is to reduce the dimensionality of an input representation (image, hidden-layer output matrix, etc.) based on assumptions that could be made from features included in the binned sub-regions. It aims to reduce overfitting by offering a simplified version of the representation. It also minimizes the computational cost by lowering the number of parameters in the learning process, as well as providing basic translation invariance to the internal representation.

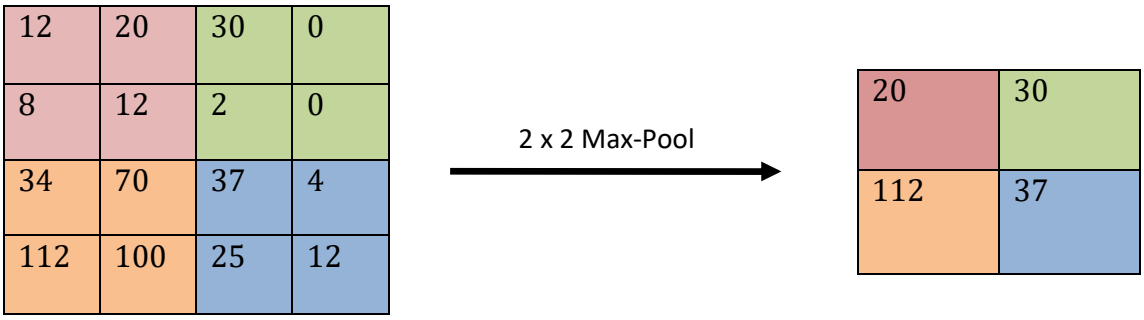


figure 21: max-pooling example

PointNet

PointNet is an innovative, highly efficient net that uses neural networks to detect 3D objects without rendering. It was created by the team of Stanford University (PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation) to provide several applications for scene semantic parsing to objects classification. It is appropriate for unordered input sets because it uses a simple symmetric structure. It is a unified architecture that learns both global and local point characteristics, allowing it to perform a variety of 3D recognition tasks in a simple, quick, and effective manner.

The network effectively learns a set of optimization functions/criterion that selects critical points (also mentioned as interesting or informative) in the point cloud and encodes the rationale for their selection. The network's last fully connected layers combine these ideal critical points into a global descriptor for the entire form (shape classification) or forecast per point labels (shape segmentation). Since each point may be transformed individually by affine transformations it is required that points be preprocessed before inserting them to the algorithm. This preprocessing tries to canonicalize the input and makes it independent to potential rotation or translation of the initial model before the PointNet analyzes it.

There are three important modules of the PointNet. The max pooling layer, which uses a symmetric function to aggregate data from all points, a local and global information combination structure, and two joint alignment networks to align both input points and point characteristics. In this approach a **symmetric function** takes n vectors as input and returns a new vector that is invariant to the order of the input vectors. The +, for example, is symmetric binary function. The basic idea is to use a symmetric function on altered items in the set to approximate a generic function defined on a point set:

$$f(\{x_1, \dots, x_n\}) \approx g(h(x_1), \dots, h(x_n))$$

They estimate **h** using a multi-layer perceptron network, and **g** using a combination of a single variable function and a max pooling function. We can learn a number of **f**'s from a collection of **h**'s to capture distinct aspects of the set. The result of the previous part is a vector [f1, f2,....fn], which is the input set's global signature. This will now function perfectly because the SVM can be simply trained to produce a classifier output. However, we need a combination of local and global information for **point segmentation**. After computing the global feature vector, we feed it back to the point feature by concatenating global features with per point features to get the desired outcome. This approach can anticipate per-point quantities by relying on both global and local semantics.

If a point cloud is subjected to geometric modifications the semantic labeling must remain invariant. As a result, we anticipate our point set's learned

representation to be invariant to these alterations.

A **mini-network (T-net)** predicts an affine transformation matrix, which we then apply directly to the coordinates of input points. The mini-network is made up of core modules such as point independent feature extraction, maximum pooling, completely linked layers, and it mimics the huge network. The T-net is discussed in further depth in the appendix. The goal of T-net is to use its own small network to learn an affine transformation matrix. The T-net is utilized twice in this architecture. The first time the input features (n, 3) are transformed into a canonical representation. The second is an affine transformation for feature space alignment (n, 3). They confine the transformation to be near to an orthogonal matrix as in the original study.

To align features from distinct input point clouds, we may use another alignment network on point features and predict a feature transformation matrix. The **transformation matrix** in the feature space, on the other hand, has a far larger dimension than the spatial transform matrix, making optimization much more challenging. As a result, we include a regularization term in the softmax training loss (Softmax Activation + Cross-Entropy Loss). The feature transformation matrix is constrained to be close to an orthogonal matrix:

$$L_{reg} = ||I - AA^T||^2$$

where **A** is the feature alignment matrix predicted by a mini-network. As a result, an orthogonal transformation is preferred since it does not lose information in the input. By including the regularization term in the optimization, the optimization becomes more stable, and the model performs better.

The below diagram represents the architecture of PointNet (figure 22: PointNet architecture). The classification network receives n points as input, executes input and feature transformations, and then uses max pooling to aggregate point features. The result is a classification score for each of the k classes. It combines global and local characteristics and generates point scores. The letters “mlp” stand for multi-layer perceptron (MLP), while the numbers in parentheses represent the layer sizes of the perceptron. For example, “mlp(64, 64)” means that we have 2 hidden layers with size 64 (number of neurons) on each. In a classification net, dropout layers are employed for the last mlp. With ReLU, batch normalization is utilized for all layers.

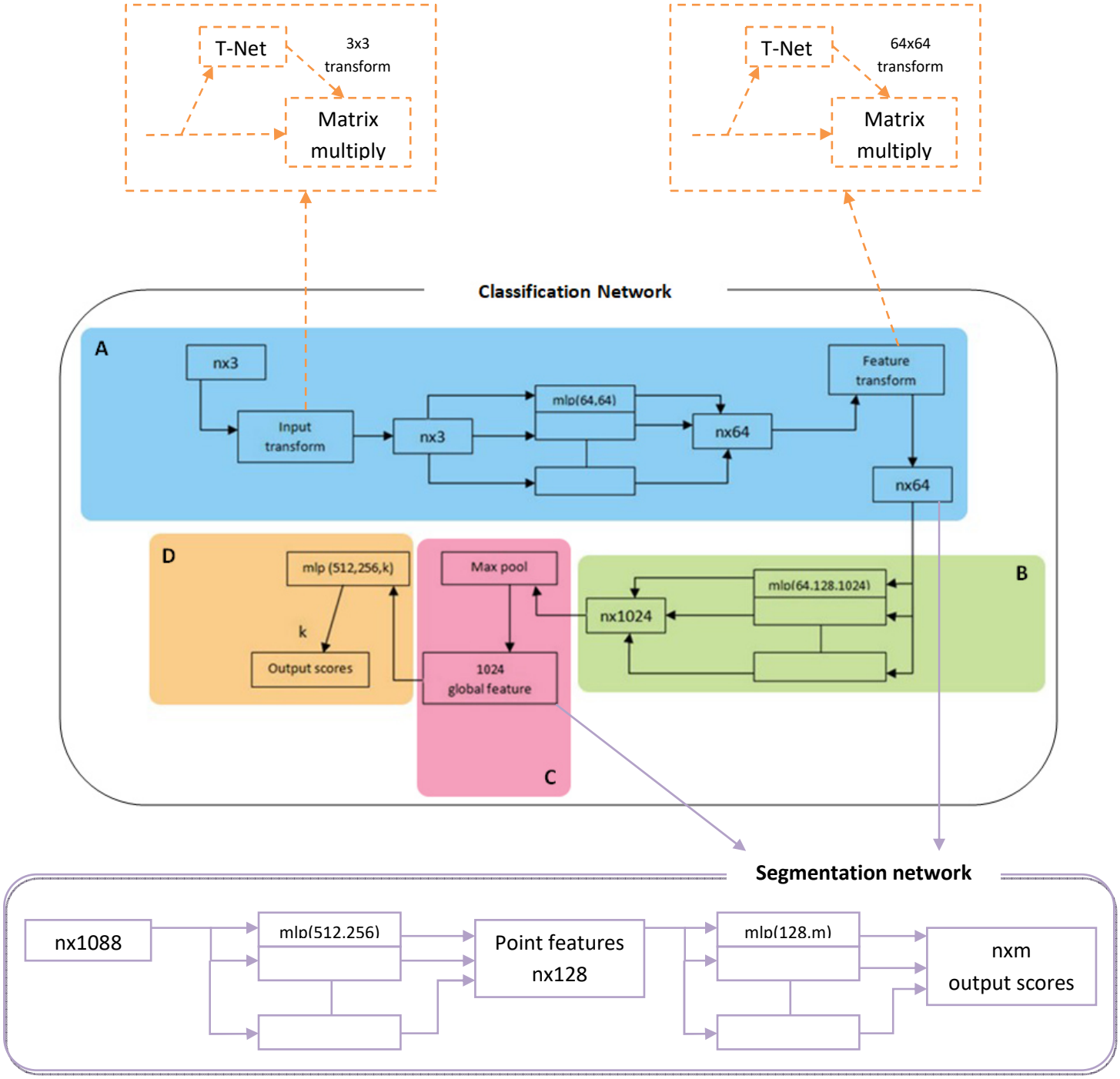


figure 22: PointNet architecture

More specifically the **classification network** maps each of the n points from 3 dimensions to 64 dimensions by using a shared multi-layer perceptron. It is critical that each of the n points has its own multi-layer perceptron (**A** on diagram). Similarly, each n point is transferred from 64 to 1024 dimensions in the following layer (**B** on diagram). A max pooling is further used to construct a global feature vector in R^{1024} (**C** on diagram). Finally, the global feature vector is mapped to k output classification scores using a three-layer fully connected network (FCN) (**D** on diagram).

Each of the n input points in the **segmentation network** must be allocated to one of k segmentation classes. Because segmentation relies on both local and global features, the points in the 64-dimensional embedding space

(local point features) are concatenated with the global feature vector (global point features) to produce a per-point vector in R^{1088}. In other words, after computing the global feature vector, the algorithm feeds it back to the point feature by concatenating global features with per point features to get the desired outcome. This approach can anticipate per-point quantities by relying on both global and local semantics. MLPs are used on the n points to reduce the dimensionality from 1088 to 128 and subsequently to m, resulting in an array of n x m.

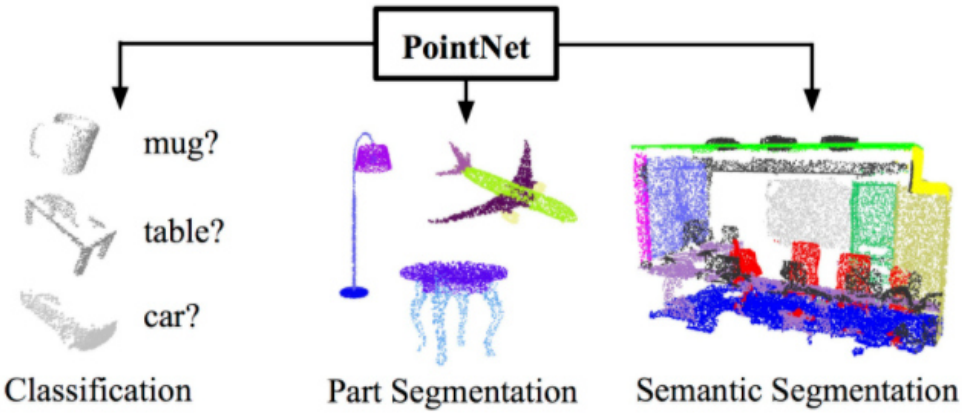


figure 23: stages of PointNet architecture

The higher-level design of PointNet motivates the activities that make up the T-Net (figure 24: T-Net architecture). MLPs (or fully connected layers) are used to translate the input points to a higher-dimensional space independently and identically: max pooling is employed to encode a global feature vector, which is subsequently reduced to R^{256} with FC layers. The final FC layer's input-dependent features are then merged with globally trainable weights and biases to produce a 3-by-3 transformation matrix.

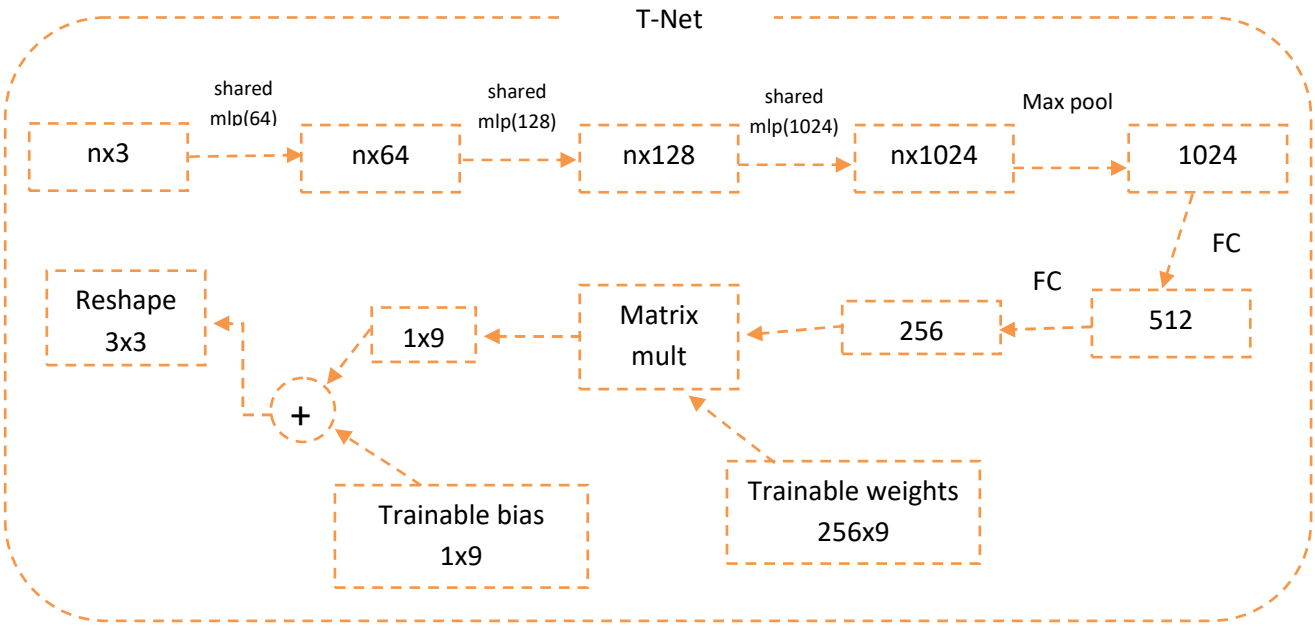


figure 24: T-Net architecture

The **global feature** vector may be used to derive a significant amount of intuition. To begin, as previously stated, the dimensionality of the vector, referred to as the bottleneck dimension and denoted by K, is directly related to the expressiveness of the model. Naturally, a higher K value leads to a more complicated — and, more importantly, correct — model, and vice versa. K=1024, for example, is used in the design of PointNet. Also keep in mind that the feature vector was the outcome of a well-thought-out symmetric function (for permutation invariance). PointNet employs maximum pooling. The output of max pooling compresses the n points in the input point cloud to a subset of points, similar to how the max operator compresses numerous real-valued inputs to a single value. In reality, the global feature vector can be contributed to by no more than K points. The critical point set is made up of points that contribute to and define the global feature vector, and it encodes the input with a sparse collection of key points.

More intriguingly, the network learns to summarize an input point cloud using a sparse collection of important points, which closely matches to the skeleton of objects according to visualization (figure 25: visualization of objects).

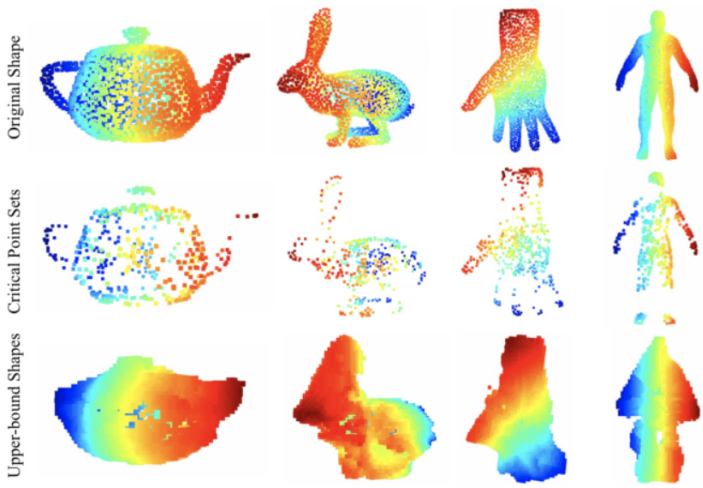


figure 25: visualization of objects

3.2. Implementation

The team from the web page of Keras made a program based in the architecture of PointNet. In this approach they used ModelNet10 dataset thus the application is built to read objects in ".off" file format and process a sample of their vertices.

This thesis deals with the idea of using the program developed by Keras and convert it to read and process raw data in xyz file format.

Off file format to xyz

To convert the elements included in “ModelNet10” form “.off” file format to “.xyz” we follow the below steps.

First of all, we define the file that contains our dataset. For this purpose we use the “Path()” command with the name of the file in quotation marks as shown below.

```
path = Path("ModelData2")
```

Then we read the “.off” data from the file with the command “glob.glob(os.path.join(,))”. The “glob.glob(*pathname(sting)*)” is a method that returns a list of path names that matches the input parameter. The use of “os.path.join(,)” as shown below allows us to have access to the included files of “path” .

```
folders = glob.glob(os.path.join(path, "[!README]*"))
```

In order to have access to each value included in “folders” element we use a simple “for” loop. First, we only process the data included in categories “bed” and “chair” (second and third) because the point clouds of these two classes can be characterized as similar. For this purpose, we make two “if” statements to delimit the datasets we get. In the first one we rule out the possibility of reading the files included in “bathtub” which is the forth category. The second “if” ends the process when it’s time to read data from third class, named “desk”. Finally, we use the “train_set” list to append the values of “train” files included in “bed” and “chair” classes and the “test_set” list to append the elements of the “test” files.

```
path = Path("ModelNet10")
```

```
train_set = []
test_set = []

folders = glob.glob(os.path.join(path, "[!README]*"))

for i, folder in enumerate(folders):

    if(os.path.basename(folder)!= 'bathtub'):
        if(os.path.basename(folder)!= 'desk'):
            print("Class: {}".format(os.path.basename(folder)))

            #training set------------------------------------------------------------
            train_files = glob.glob(os.path.join(folder, "train/*"))
            for f in train_files:
                train_set.append(trimesh.load(f))

            #test set----------------------------------------------------------------
            test_files = glob.glob(os.path.join(folder, "test/*"))
            for f in test_files:
                test_set.append(trimesh.load(f))

        else:
            break
```

figure 26: code to access the data of a specific folder

Both “train_set” and “test_set” includes trimesh objects from both “chair” and “bed” classes.

The following figure shows the output of the “train_set” list. As you can see each trimesh object has vertices and faces. For example, the shape of vertices in the first object is 689 rows and 3 columns.

```
train_set
```

```
[<trimesh.Trimesh(vertices.shape=(689, 3), faces.shape=(1807, 3))>,
 <trimesh.Trimesh(vertices.shape=(16394, 3), faces.shape=(35299, 3))>,
 <trimesh.Trimesh(vertices.shape=(564, 3), faces.shape=(2440, 3))>,
 <trimesh.Trimesh(vertices.shape=(720, 3), faces.shape=(2200, 3))>,
 <trimesh.Trimesh(vertices.shape=(2684, 3), faces.shape=(6761, 3))>,
 <trimesh.Trimesh(vertices.shape=(5152, 3), faces.shape=(10045, 3))>,
```

figure 27: values included in “train_set” list

In order to save our dataset as point clouds we create two new lists to append the vertices of each trimesh. From the above output results we can see that each trimesh has different shapes, so we need to save each object with fixed number of points based on the shape of vertices.

The command “a, k = test_set[i].vertices.shape” outputs two values. The “a” value refers to vertices and represents the number of the i-gate trimesh’s rows of the “test_set” list. The second value represents the number of columns of the same list and trimesh. So, in our case the first value is more useful.

The following code includes the above methodology and exports two lists (train_set, test_set) with point cloud elements in groups.

```
#test-----------------------------------------------
test_data = []
y=len(test_set)

for i in range(0,y):
    a,k=test_set[i].vertices.shape
    test_data.append(test_set[i].sample(a))

#train-----------------------------------------------
train_data = []
w=len(train_set)

for i in range(0,w):
    a,k=train_set[i].vertices.shape
    train_data.append(train_set[i].sample(a))
```

figure 28: code for storing different point clouds

Output

```
train_data
```

```
[array([[ 10.56697185,  12.47527929, -20.13980091],
        [ 18.5012    ,   7.49366004,  -2.94584635],
        [ 18.5012    ,  -7.12256399, -18.66665498],
        ...,
        [-13.576     ,  -6.08271295, -15.27553229],
        [ 18.5012    ,  10.55476128,   2.88986818],
        [ -8.22417995,  23.975     ,  -7.91750118]]),
 array([[ 12.07405136, -10.52685468,  -9.7886103 ],
        [  3.25833017, -24.51366726,  -1.59635629],
        [  9.52719931, -13.3781037 ,  -0.1974349 ],
        [-13.29796662,   3.48473814,   7.22682267],
        [  8.30665613,  18.81410422,  -3.33966503],
        [ -4.54104042,   8.57433185, -11.375     ],
```

figure 29: values included in “train_data” list

The last part is to save our data to “xyz” file format for further use. So, we create files with the name “p.xyz” for each list included in “train_set” and append the values in three columns.

```
nnum3=[]
k1=0
for i in range (0,len(train_set)):
    nnum3.append("p{}.xyz".format(i))

for i in range (0, len(train_data)):
    with open(nnum3[k1], mode='w') as f:
        for j in range(len(train_data[k1])):
            f.write("%f    "%float(train_data[k1][j][0].item()))
            f.write("%f    "%float(train_data[k1][j][1].item()))
            f.write("%f    \n"%float(train_data[k1][j][2].item()))
    k1=k1+1
```

figure 30: “xyz” files for training set

The output files are saved in the same path as the python code.

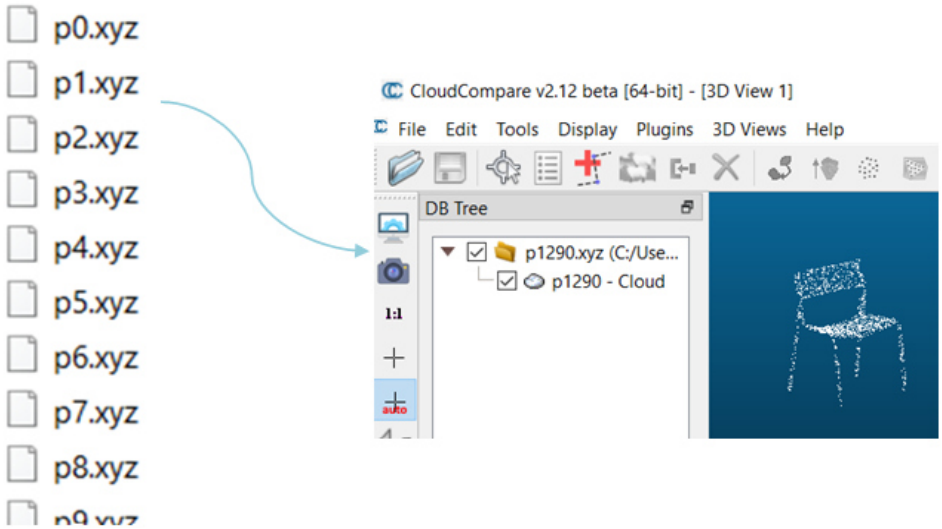


figure 31: View of a "xyz" file included in train set

We follow the same methodology for the "test_set" and create files with the name "ptest.xyz".

```
nnum3=[]
k1=0
for i in range (0,len(test_set)):
    nnum3.append("ptest{}.xyz".format(i))

for i in range (0, len(test_data)):
    with open(nnum3[k1], mode='w') as f:
        for j in range(len(test_data[k1])):
            f.write("%f    "%float(test_data[k1][j][0].item()))
            f.write("%f    "%float(test_data[k1][j][1].item()))
            f.write("%f    \n"%float(test_data[k1][j][2].item()))
    k1=k1+1
```

figure 32: "xyz" files for test set

The output results are shown below.

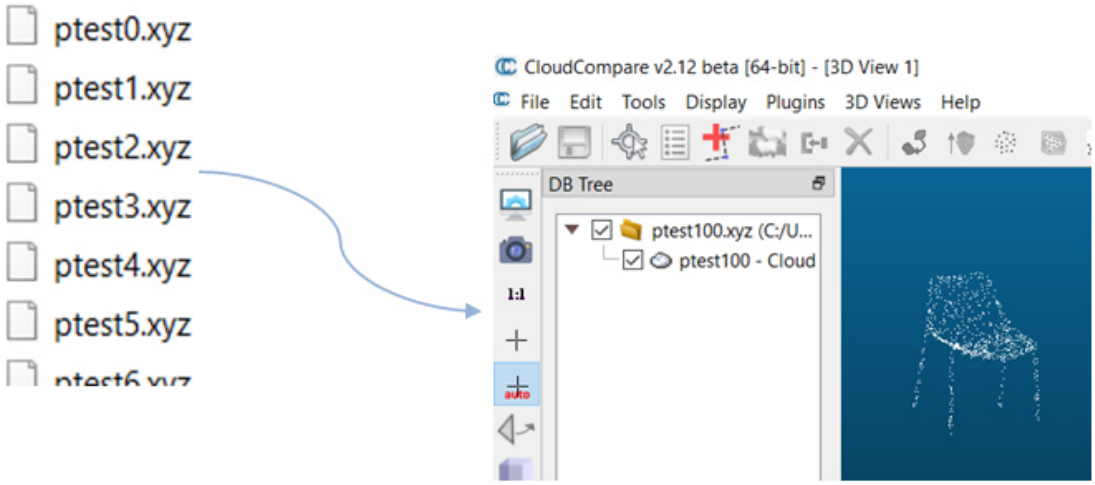


figure 33: View of a "xyz" file included in test set

The last step is to transfer all the "xyz" data to a new file (named "New") divided into two categories with names "bed" and "chair". Each of these categories includes two subfiles called "train" and "test".

3.2.1. *Data preprocessing*

Before building the net, it is important to preprocess our data. The first step is to read the new file with ".xyz" objects by using the following command.

Path = Path("New")

We use the ready-made code about PointNet from Keras web page and change it to read xyz files. For this purpose we replace the "samples(*integer*)" command to "vertices.view()" with "np.ndarray" as an input. The whole implementation is done in a function called "setsAndlabels()" and the outputs of this function is 4 array lists and a class map that shows the labels of each category ("bed" and "chair").

```
def setsAndlabels():

    train_points = []
    train_labels = []
    test_points = []
    test_labels = []
    class_map = {}
    folders = glob.glob(os.path.join(path, "[!README]*"))

    for i, folder in enumerate(folders):
        print("processing class: {}".format(os.path.basename(folder)))
        # store folder name with ID so we can retrieve later
        class_map[i] = folder.split("\\")[-1]
        # gather all files
        train_files = glob.glob(os.path.join(folder, "train/*"))
        test_files = glob.glob(os.path.join(folder, "test/*"))

        for f in train_files:
            train_points.append(trimesh.load(f).vertices.view(np.ndarray))
            train_labels.append(i)

        for f in test_files:
            test_points.append(trimesh.load(f).vertices.view(np.ndarray))
            test_labels.append(i)

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

```
train_set = []
train_labels = []
test_set = []
test_labels = []
class_map = {}

train_set, test_set, train_labels, test_labels, CLASS_MAP = setsAndlabels()
```

```
processing class: bed
processing class: chair
```

figure 34: function for storing ".xyz" data

From the output result of "CLASS_MAP" we can see that label "0" is assigned to class "bed" and label "1" is assigned to "chair". So, after training, the computer must detect beds as "0" and chairs as "1".

```
CLASS_MAP
```

```
{0: 'bed', 1: 'chair'}
```

figure 35: "CLASS_MAP" value output

So, the values included in "train_labels" and "test_labels" must be "0" and "1" depending on the category they represent.

```
test_labels
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1])
```

```
train_labels
```

```
array([0, 0, 0, ..., 1, 1, 1])
```

figure 36: "train_labels" and "test_labels" outputs

"Train_set" includes data from paths "bed/train" and "chair/train" (where "train" is a subfile of the file "chair").

```
train_set
```

```
[array([[ -5.802605, -40.027505,  -0.61301 ],
        [ -7.666625, -32.35231 ,   0.75     ],
        [ 24.415532,  28.637627,   7.726409],
        ...,
        [ -6.8919  ,  33.2821  ,   6.963308],
        [ 31.029121,  -7.370752,  -7.5     ],
        [ -1.162228,  28.764363,   5.211174]]),
 array([[-10.418241, -37.5     ,  -4.401239],
        [-21.6248  , -24.101593,  -5.378961],
        [-21.120787,  30.765713, -11.      ],
        ...,
```

figure 37: "train_set" output

"Test_set" includes data from "chair/test" and "bed/test" (where "test" is a subfile of the file bed).

```
test_set
```

```
[array([[-30.987113, -63.169572,  16.232581],
        [-19.942932, -36.353568,   9.125    ],
        [-37.237715,   1.060098,   7.268998],
        ...,
        [-17.18878 , -24.289883,   1.875    ],
        [ 35.762271,   0.362119, -16.771195],
        [ 24.668917, -64.74995 , -34.982248]]),
 array([[ 31.376967,  24.9375  ,  -1.075436],
        [ 27.702898, -32.631055,   1.691669],
        [  9.12588 ,   8.303286,   1.937626],
        ...,
```

figure 38: "test_set" output

By using the commands "len(train_set)" and "len(test_set)" we can see the actual number of elements included in "train_set" and "test_set" lists. For example, "train_set" is a list of 1404 elements (figure 39).

```
len(train_set)
```

```
1404
```

figure 39: elements included in "train_set".

With the use of the same command, we can see the number of points included in one element. For instance the element ”0” of “train_set” (train_set[0]) has 689 points and the element ‘1’ of the same set (train_set[1]) has 16394 points.

```
len(train_set[0])
```

```
689
```

```
len(train_set[1])
```

```
16394
```

figure 40: number of points included in one element of a list.

As we mentioned before, each list element has different number of points. The architecture of PointNet requires the same number of points for every element included in “training_set” and “test_set”. For this purpose, we create two list arrays to save our new datasets with “target” number of points for each element included. So, by using a “for” loop, we read the elements of test_set one by one. If the number of points of one element is greater than the “target” value the program asks for a random number (by using the command “random.choice()”) between the bounds of this element . This number represents a row of “test_list”, so since we work with “xyz” files this row includes the coordinates of a point. This process is repeated until the number of inputs included in “list1” are equal to “target” value. In this way we achieve the random reduction of the number of points for one element without changing its original form.
In case we want to increase the number of points in random positions we again ask for a random number between the element’s bounds to detect a certain point from “test_set” list .Then by adding the value 0.09 to each axis of the detected point we create a new vertex. Finally we append the new point to “list1” and repeat the prosses until the number of “target” points is achieved.
For elements with equal number of points to “target” value we save them as they are. Finally we append the “list1” to “te_list” for later use as a new test set.

```
for i, folder in enumerate(test_set):
    print("processing test_set")
    test_list=[]
    list1=[]
    test_list=folder
    y1=len(test_list)
    print(y1)
    #number of points > target ------------------------------
    if (y1>target):
        print("bigger")
        num2=[]
        l=[]
        for i in range (0,len(test_list)):
            l.append(i)

        for i in range(0,target):
            a=random.choice(l)
            list1.append(test_list[a])
    #number of points < target ------------------------------
    elif (y1<target):
        print("smaller")
        num2=[]
        l=[]

        for i in range(0,len(test_list)):
            list1.append(test_list[i])

        y=target-(len(list1))
        for i in range (0,len(test_list)):
            l.append(i)

        for i in range(0,y):
            a=random.choice(l)
            tl=[]
            tl.append(test_list[a][0]+0.09)#for x axis
            tl.append(test_list[a][1]+0.09)#for y axis
            tl.append(test_list[a][2]+0.09)#for z axis
            list1.append(np.array(tl))
```

```
import random

te_list=[]
target=10000
print({'target': target})
```

```
#number of points = target ------------------------------
else:
    print("equal")
    for i in range(0,len(test_list)):
        list1.append(test_list[i])

with open(nnum3[k1], mode='w') as f:
    for i in range(len(list1)):
        f.write("%f    "%float(list1[i][0].item()))
        f.write("%f    "%float(list1[i][1].item()))
        f.write("%f    \n"%float(list1[i][2].item()))
    k1=k1+1

te_list.append(list1)
```

figure 41: code for generating sub clouds from “test_set”

We follow the same methodology for “train_set”.

```
#train_set-----------------------------------------------
tr_list=[]

for i, folder in enumerate(train_set):
    print("processing train_set")
    train_list=[]
    list1=[]
    train_list=folder
    y1=len(train_list)
    print(y1)
    if (y1>target):
        print("bigger")
        num2=[]
        l=[]
        for i in range (0,len(train_list)):
            l.append(i)

        for i in range(0,target):
            a=random.choice(l)
            list1.append(train_list[a])

    elif (y1<target):
        print("smaller")
        num2=[]
        l=[]

        for i in range(0,len(train_list)):
            list1.append(train_list[i])

        y=target-(len(list1))
        for i in range (0,len(train_list)):
            l.append(i)

        for i in range(0,y):
            a=random.choice(l)
            tl=[]
            tl.append(train_list[a][0]+0.09)
            tl.append(train_list[a][1]+0.09)
            tl.append(train_list[a][2]+0.09)
            list1.append(np.array(tl))
    else:
        print("equal")
        for i in range(0,len(train_list)):
            list1.append(train_list[i])

    with open(nnum3[k1], mode='w') as f:
        for i in range(len(list1)):
            f.write("%f    "%float(list1[i][0].item()))
            f.write("%f    "%float(list1[i][1].item()))
            f.write("%f    \n"%float(list1[i][2].item()))
        k1=k1+1

    tr_list.append(list1)
```

figure 42: code for generating sub clouds from “train_set”

For reasons of further data processing is useful to save the new data to “.txt” files. So, before the above program, we create a number of “txt” files (with the name “points.txt”) equal to the summary of the elements included in “train_set” list and “test_set” list to save the datasets before exiting the “for” loops.

```
#names for the new txts----------------------------------------
nnum2=len(test_set)+len(train_set)
nnum3=[]
k1=0
for i in range (0,nnum2):
    nnum3.append("points{}.txt".format(i))
#---------------------------------------------------------------
```

figure 43: code for generating multiple ”.txt” files.

The output of the above commands are shown to figure 44 where "{'target':10000}" is the number of points we wish for each element of our datasets, the numbers "5126", "11302" and "163" are the actual numbers of points for each element and the indications "smaller", "bigger" means that the actual numbers of points are smaller or bigger than "target" number of points.

```
{'target': 10000}
processing test_set
5126
smaller
processing test_set
11302
bigger
processing test_set
163
smaller
```

figure 44

The images below shows the output of the above methodology. The left one shows a case of random reduction points and the right one represents a case of increasing the number of points in random positions.

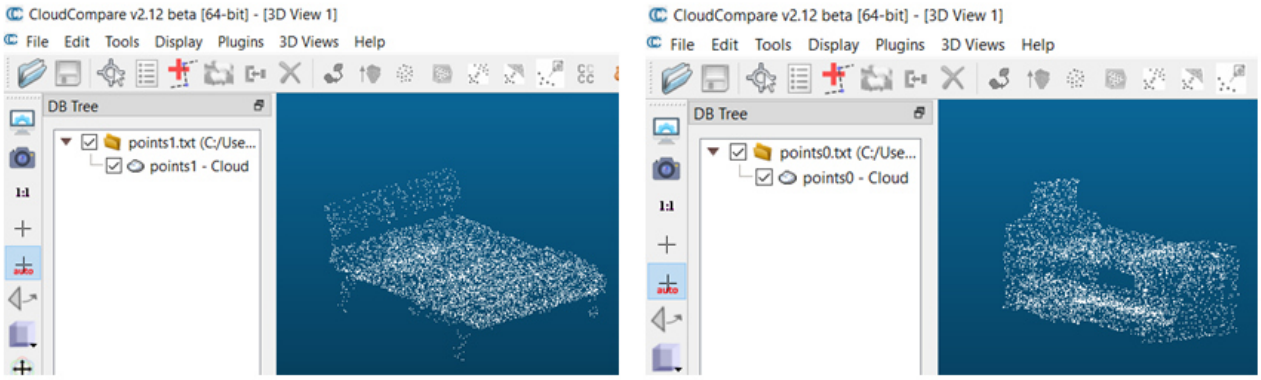


figure 45: output of subclouds.

3.2.2. *Data processing*

The next step is to normalize the device coordinates between -1 and 1 for each element of our new array lists (tr_list, te_list). For this purpose, we use the following function to normalize our feature x in range 0 and 1.

$$x' = \frac{x - minx}{maxx - minx}$$

And then, to normalize it in range -1 and 1 we use the function

$$x'' = 2\frac{x - minx}{maxx - minx} - 1$$

In general, to get a normalized value between “a” and “b” we use the following function.

$$x''' = (b - a)\frac{x - minx}{maxx - minx} + a$$

All the above calculations where done in a function called “scale_numpy_array” for each element of “tr_list” and “te_list”. The results are saved in lists, “v” and “v2”, and assigned to “tr_list” and “te_list” respectively.

```
def scale_numpy_array(data):
    scaled_unit = (data - np.min(data)) / (np.max(data) - np.min(data))
    scaled_unit = 2*scaled_unit
    scaled_unit = scaled_unit -1
    return scaled_unit

v=[[ 0 for i in range(3) ] for j in range(len(tr_list))]

for i in range (0,len(tr_list)):
    v[i]=scale_numpy_array(tr_list[i])
```

```
def scale_numpy_array(data):
    scaled_unit = (data - np.min(data)) / (np.max(data) - np.min(data))
    scaled_unit = 2*scaled_unit
    scaled_unit = scaled_unit -1
    return scaled_unit

v2=[[ 0 for i in range(3) ] for j in range(len(te_list))]

for i in range (0,len(te_list)):
    v2[i]=scale_numpy_array(te_list[i])
```

```
tr_list=v
te_list=v2
```

figure 46: normalization code

To make sure that the calculations are done correctly we assign an item from each list in a “txt” file and check the values in CloudCompare program.

```
with open("test.txt", mode='w') as f:
        for i in range(len(v[0])):
            f.write("%f    "%float(v[0][i][0].item()))
            f.write("%f    "%float(v[0][i][1].item()))
            f.write("%f    \n"%float(v[0][i][2].item()))
```

```
with open("test2.txt", mode='w') as f:
        for i in range(len(v2[0])):
            f.write("%f    "%float(v2[0][i][0].item()))
            f.write("%f    "%float(v2[0][i][1].item()))
            f.write("%f    \n"%float(v2[0][i][2].item()))
```

figure 47: code for ".txt" examples of point clouds

Before normalization

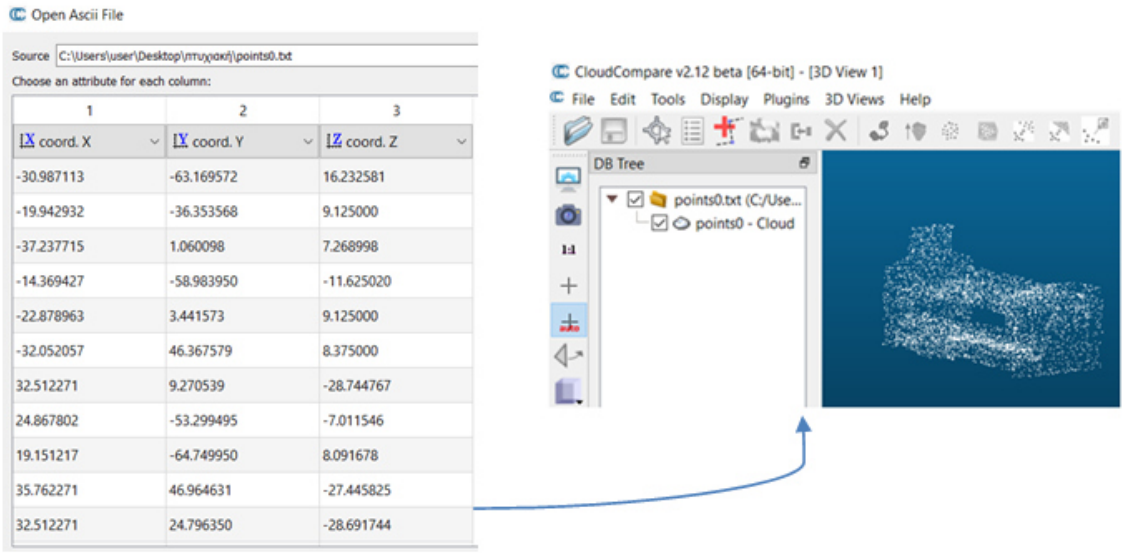


figure 48: non-normalized point cloud

After normalization

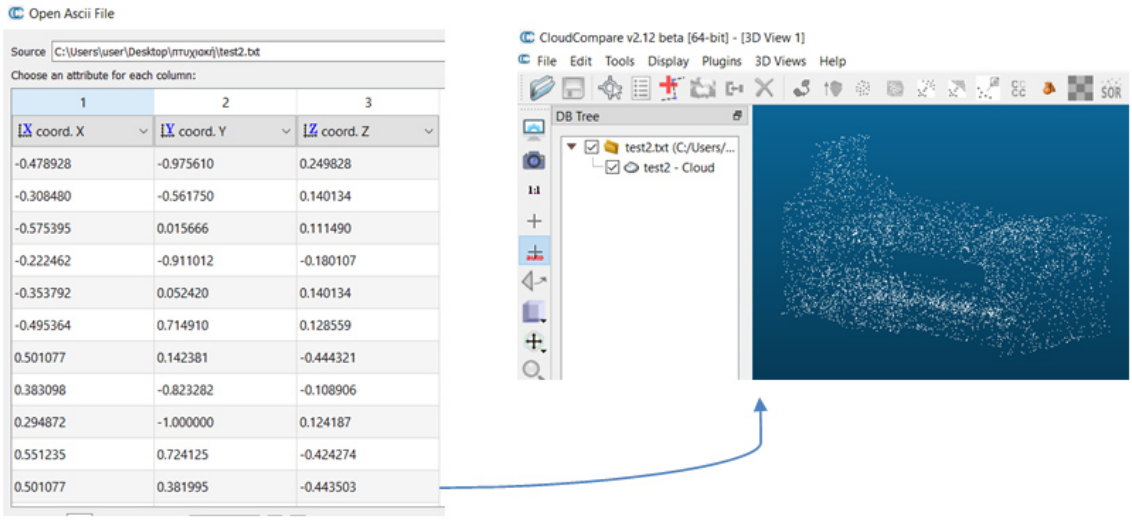


figure 49: normalized point cloud

As you can see the values are normalized without effecting the shape of the object.

The shuffle buffer size is set to the total size of the dataset since the data was previously organized by class. To random jitter and shuffle train dataset, the team of Keras developed an augmentation function. The new datasets are called “train_dataset” and “test_dataset”.

The command “tf.random.uniform(shape, minval = 0, maxval= None, None,dtype=tf.float32, seed=None, name=None)” produces a tensor of the provided shape filled with values from a uniform distribution in the range minval to axval, with the lower limit included but not the higher bound.

```
def augment(points, label):
    # jitter points
    points += tf.random.uniform(points.shape, -0.005, 0.005, dtype=tf.float64)
    # shuffle points
    points = tf.random.shuffle(points)
    return points, label


train_dataset = tf.data.Dataset.from_tensor_slices((tr_list, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices((te_list, test_labels))

train_dataset = train_dataset.shuffle(len(tr_list)).map(augment).batch(32)
test_dataset = test_dataset.shuffle(len(te_list)).batch(32)
```

figure 50: code for shuffle and jitter

The batch size represents groups of data. In our case, batch size is equal to 32 (bach(32)). This means that during training the algorithm takes the first 32 samples from “tr_list” and trains the network. Then it takes the next 32 samples and trains again the network.

In case you want a view of your data you can use the following commands

```
data = test_dataset

points, labels = list(data)[0]
points = points
labels = labels
```

```
fig = plt.figure(figsize=(50, 50))
for i in range(2):
    ax = fig.add_subplot(8, 1, i + 1, projection="3d")
    ax.scatter(points[i, :, 0], points[i, :, 1], points[i, :, 2],c='g')
    ax.set_axis_off()
plt.show()
```

Output

figure 51: view of point clouds

3.2.3 PointNet

In the "General information" section we do a theoretical analysis about PointNet architecture. So, for the code part, the network is built as shown below. The authors use the smaller 2 classes of ModelNet10 dataset and duplicate the network design presented in the original research, but with half the number of weights on each layer.

The below code shows the architecture of the main net.

```
inputs = keras.Input(shape=(target, 3))

x = tnet(inputs, 3)
x = conv_bn(x, 32)
x = conv_bn(x, 32)
x = tnet(x, 32)
x = conv_bn(x, 32)
x = conv_bn(x, 64)
x = conv_bn(x, 512)
x = layers.GlobalMaxPooling1D()(x)
x = dense_bn(x, 256)
x = layers.Dropout(0.3)(x)
x = dense_bn(x, 128)
x = layers.Dropout(0.3)(x)

outputs = layers.Dense(2, activation="softmax")(x)

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

figure 52: main architecture of PointNet

"keras.Input( )" command creates a keras tensor which represents the input of a keras object. In our case, the input value has "target" number of rows and 3 columns.

The next command is “tnet( )”, a function that describes the transformer network. It has similar architecture with the main net.

```
def tnet(inputs, num_features):

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

    x = conv_bn(inputs, 32)
    x = conv_bn(x, 64)
    x = conv_bn(x, 512)
    x = layers.GlobalMaxPooling1D()(x)
    x = dense_bn(x, 256)
    x = dense_bn(x, 128)
    x = layers.Dense(
        num_features * num_features,
        kernel_initializer="zeros",
        bias_initializer=bias,
        activity_regularizer=reg,
    )(x)
    feat_T = layers.Reshape((num_features, num_features))(x)
    # Apply affine transformation to input features
    return layers.Dot(axes=(2, 1))([inputs, feat_T])
```

figure 53: architecture of T-net

“Keras.initilization.Constant()” command is a Tensor initializer that creates constant values tensors

“np.eye( )” command returns a two-dimensional numpy array, like a matrix, with zeros everywhere except from diagonal which has ones.

In Python, we may flatten a matrix to one dimension by using the “ndarray.flatten( )” method.

All the above commands are useful for the creation of bias value. In order to create the “reg” value “OrthogonalRegularizer( )” function is used.

```
class OrthogonalRegularizer(keras.regularizers.Regularizer):
    def __init__(self, num_features, l2reg=0.001):
        self.num_features = num_features
        self.l2reg = l2reg
        self.eye = tf.eye(num_features)

    def __call__(self, x):
        x = tf.reshape(x, (-1, self.num_features, self.num_features))
        xxt = tf.tensordot(x, x, axes=(2, 2))
        xxt = tf.reshape(xxt, (-1, self.num_features, self.num_features))
        return tf.reduce_sum(self.l2reg * tf.square(xxt - self.eye))
```

figure 54: regularization code

The command “keras.regularizers.Regularizer” allow us to apply penalties during optimization on layer parameters or layer activity. These penalties are added together in the network's loss function. This “OrthogonalRegularizer( )” function includes two subfuctions called __init__( ) and __call__( ).

The functions below are referred in many places of model's architecture.

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


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

figure 55: code for applying a convolution layer

"layers.Conv1D( )" applies an 1D convolution layer, "layers.Dense( )" represents a regular densly-connected neural network layer, "layers.BatchNormalization( )" is a layer that normalizes the data it receives and "layers.Activation( )" with "relu" as input applies ReLU activation function to "x" output.

There is also another command that we use to take the largest value to down sample the input representation and it is called "layers.GlobalMaxPooling1D( )".

The last two command that are used in "tnet( )" are the "layers.Reshape( )" and the "layers.Dot( )". The first one represents a layer that reshapes the input into a given target shape and the second command computes a dot product between samples in two tensors.

The steps Convolution / Dense -> Batch Normalization ->ReLU Activation Function make up each convolution and fully-connected layer (with the exception of end layers).

So going back to the main architecture of PointNet we can see that, except from the commands that are already defined above, there is also the "layers.Dropout()" line in which dropout is applied to the input.

Finally by using "keras.Model( )" we create a model with inputs , outputs and the title "pointnet".

The summary of our model ("model.summary( )") is shown above.

```
Model: "pointnet"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_1 (InputLayer)         [(None, 2048, 3)]         0
_________________________________________________________________________________________________
conv1d (Conv1D)              (None, 2048, 32)          128         input_1[0][0]
_________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 2048, 32)       128         conv1d[0][0]
_________________________________________________________________________________________________
activation (Activation)      (None, 2048, 32)          0           batch_normalization[0][0]
_________________________________________________________________________________________________
conv1d_1 (Conv1D)            (None, 2048, 64)          2112        activation[0][0]
_________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 2048, 64)       256         conv1d_1[0][0]
_________________________________________________________________________________________________
activation_1 (Activation)    (None, 2048, 64)          0           batch_normalization_1[0][0]
_________________________________________________________________________________________________
conv1d_2 (Conv1D)            (None, 2048, 512)         33280       activation_1[0][0]
_________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 2048, 512)      2048        conv1d_2[0][0]
_________________________________________________________________________________________________
activation_2 (Activation)    (None, 2048, 512)         0           batch_normalization_2[0][0]
_________________________________________________________________________________________________
global_max_pooling1d (GlobalMax (None, 512)            0           activation_2[0][0]
_________________________________________________________________________________________________
dense (Dense)                (None, 256)               131328      global_max_pooling1d[0][0]
_________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 256)            1024        dense[0][0]
_________________________________________________________________________________________________
activation_3 (Activation)    (None, 256)               0           batch_normalization_3[0][0]
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 128)               32896       activation_3[0][0]
_________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 128)            512         dense_1[0][0]
_________________________________________________________________________________________________
activation_4 (Activation)    (None, 128)               0           batch_normalization_4[0][0]
_________________________________________________________________________________________________
dense_2 (Dense)              (None, 9)                 1161        activation_4[0][0]
_________________________________________________________________________________________________
reshape (Reshape)            (None, 3, 3)              0           dense_2[0][0]
_________________________________________________________________________________________________
dot (Dot)                    (None, 2048, 3)           0           input_1[0][0]
                                                                   reshape[0][0]
```

figure 56: model summary 1

conv1d_3 (Conv1D)	(None, 2048, 32)	128	dot[0][0]
batch_normalization_5 (BatchNor	(None, 2048, 32)	128	conv1d_3[0][0]
activation_5 (Activation)	(None, 2048, 32)	0	batch_normalization_5[0][0]
conv1d_4 (Conv1D)	(None, 2048, 32)	1056	activation_5[0][0]
batch_normalization_6 (BatchNor	(None, 2048, 32)	128	conv1d_4[0][0]
activation_6 (Activation)	(None, 2048, 32)	0	batch_normalization_6[0][0]
conv1d_5 (Conv1D)	(None, 2048, 32)	1056	activation_6[0][0]
batch_normalization_7 (BatchNor	(None, 2048, 32)	128	conv1d_5[0][0]
activation_7 (Activation)	(None, 2048, 32)	0	batch_normalization_7[0][0]
conv1d_6 (Conv1D)	(None, 2048, 64)	2112	activation_7[0][0]
batch_normalization_8 (BatchNor	(None, 2048, 64)	256	conv1d_6[0][0]
activation_8 (Activation)	(None, 2048, 64)	0	batch_normalization_8[0][0]
conv1d_7 (Conv1D)	(None, 2048, 512)	33280	activation_8[0][0]
batch_normalization_9 (BatchNor	(None, 2048, 512)	2048	conv1d_7[0][0]
activation_9 (Activation)	(None, 2048, 512)	0	batch_normalization_9[0][0]
global_max_pooling1d_1 (GlobalM	(None, 512)	0	activation_9[0][0]
dense_3 (Dense)	(None, 256)	131328	global_max_pooling1d_1[0][0]
batch_normalization_10 (BatchNo	(None, 256)	1024	dense_3[0][0]
activation_10 (Activation)	(None, 256)	0	batch_normalization_10[0][0]
dense_4 (Dense)	(None, 128)	32896	activation_10[0][0]
batch_normalization_11 (BatchNo	(None, 128)	512	dense_4[0][0]
activation_11 (Activation)	(None, 128)	0	batch_normalization_11[0][0]
dense_5 (Dense)	(None, 1024)	132096	activation_11[0][0]
reshape_1 (Reshape)	(None, 32, 32)	0	dense_5[0][0]

figure 57: model summary 2

```
dot_1 (Dot)                     (None, 2048, 32)     0           activation_6[0][0]
                                                                 reshape_1[0][0]
_____________________________________________________________________________________
conv1d_8 (Conv1D)               (None, 2048, 32)     1056        dot_1[0][0]
_____________________________________________________________________________________
batch_normalization_12 (BatchNo (None, 2048, 32)     128         conv1d_8[0][0]
_____________________________________________________________________________________
activation_12 (Activation)      (None, 2048, 32)     0           batch_normalization_12[0][0]
_____________________________________________________________________________________
conv1d_9 (Conv1D)               (None, 2048, 64)     2112        activation_12[0][0]
_____________________________________________________________________________________
batch_normalization_13 (BatchNo (None, 2048, 64)     256         conv1d_9[0][0]
_____________________________________________________________________________________
activation_13 (Activation)      (None, 2048, 64)     0           batch_normalization_13[0][0]
_____________________________________________________________________________________
conv1d_10 (Conv1D)              (None, 2048, 512)    33280       activation_13[0][0]
_____________________________________________________________________________________
batch_normalization_14 (BatchNo (None, 2048, 512)    2048        conv1d_10[0][0]
_____________________________________________________________________________________
activation_14 (Activation)      (None, 2048, 512)    0           batch_normalization_14[0][0]
_____________________________________________________________________________________
global_max_pooling1d_2 (GlobalM (None, 512)          0           activation_14[0][0]
_____________________________________________________________________________________
dense_6 (Dense)                 (None, 256)          131328      global_max_pooling1d_2[0][0]
_____________________________________________________________________________________
batch_normalization_15 (BatchNo (None, 256)          1024        dense_6[0][0]
_____________________________________________________________________________________
activation_15 (Activation)      (None, 256)          0           batch_normalization_15[0][0]
_____________________________________________________________________________________
dropout (Dropout)               (None, 256)          0           activation_15[0][0]
_____________________________________________________________________________________
dense_7 (Dense)                 (None, 128)          32896       dropout[0][0]
_____________________________________________________________________________________
batch_normalization_16 (BatchNo (None, 128)          512         dense_7[0][0]
_____________________________________________________________________________________
activation_16 (Activation)      (None, 128)          0           batch_normalization_16[0][0]
_____________________________________________________________________________________
dropout_1 (Dropout)             (None, 128)          0           activation_16[0][0]
_____________________________________________________________________________________
dense_8 (Dense)                 (None, 10)           1290        dropout_1[0][0]
=====================================================================================
Total params: 748,979
Trainable params: 742,899
Non-trainable params: 6,080
```

figure 58: model summary 3

As you can see the net has a value of 748.979 total parameters of which the 742.899 are trainable and the rest 6.080 are non-trainable.

The value of **trainable parameters** is adjusted/modified during training according to their gradient. **Non-trainable parameters** are those whose value is not optimized as a function of their gradient during training.

3.2.4. *Training*

Once the model has been defined, it may be trained by using “compile()”and “it()” functions, just like any other conventional classification model.

```
model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    metrics=["sparse_categorical_accuracy"],
)

model.fit(train_dataset, epochs=20, validation_data=test_dataset)
```

figure 59: code for training a model

The code for training a PointNet is shown above. This particular one is scheduled to compare the “train_dataset” with “test_dataset” for 20 epochs and learning rate equal to 0.001. The indication “categorical_crossentropy” refers to a loss function used in multi-class classification models with two or more output labels. A single category encoding value of 0s and 1s is applied to the output label.

The images below shows the training progress of the net.

During training

```
Epoch 1/20
 9/44 [=====>.....................] - ETA: 1:46 - loss: 3.3471 - sparse_cate
gorical_accuracy: 0.6146
```

After training

```
44/44 [==============================] - 189s 4s/step - loss: 1.2835 - sparse_categorical_accuracy: 0.9473 - val_loss: 1003.9
509 - val_sparse_categorical_accuracy: 0.9800
Epoch 18/20
44/44 [==============================] - 186s 4s/step - loss: 1.2888 - sparse_categorical_accuracy: 0.9416 - val_loss: 2314.8
860 - val_sparse_categorical_accuracy: 0.9900
Epoch 19/20
44/44 [==============================] - 188s 4s/step - loss: 1.3195 - sparse_categorical_accuracy: 0.9366 - val_loss: 206016
9.7500 - val_sparse_categorical_accuracy: 0.9900
Epoch 20/20
44/44 [==============================] - 187s 4s/step - loss: 1.3018 - sparse_categorical_accuracy: 0.9487 - val_loss: 143266
944.0000 - val_sparse_categorical_accuracy: 0.9600
```

figure 60: training process

The above output shows us that for the 20th epoch loss equals to 1.3018, “val_loss” is 143266944.0000, “sparse_categorical_accurancy” is 94.87% and “val_sparse_categorical_accurancy” is 96%.

The only difference between “sparse_categorical_accurancy” and “val_sparse_categorical_accurancy” is that the first one is based on our training dataset, whereas the metric prefixed with “val” is based on our test dataset. We are overfitting our model on our training dataset if the metric on our test dataset stays the same or decreases while it increases on our training dataset. This means the model is trying to fit on noise in the training dataset, leading our model to perform worse on out-of-sample data.

The value of our cross-validation data's cost function is “val_loss”, whereas the value of our training data's cost function is “loss”.
To test the above accuracy, the team of Keras used the below code and had the following results.

```
data = test_dataset.take(1)

points, labels = list(data)[0]
points = points[:8, ...]
labels = labels[:8, ...]

# run test data through model
preds = model.predict(points)
preds = tf.math.argmax(preds, -1)

points = points.numpy()

# plot points with predicted class and label
fig = plt.figure(figsize=(50, 50))
for i in range(len(points)):
    ax = fig.add_subplot(10, 1, i + 1, projection="3d")
    ax.scatter(points[i, :, 0], points[i, :, 1], points[i, :, 2])
    ax.set_title(
        "pred: {:}, label: {:}".format(
            CLASS_MAP[preds[i].numpy()], CLASS_MAP[labels.numpy()[i]]
        )
    )
    ax.set_axis_off()
plt.show()
```

Output

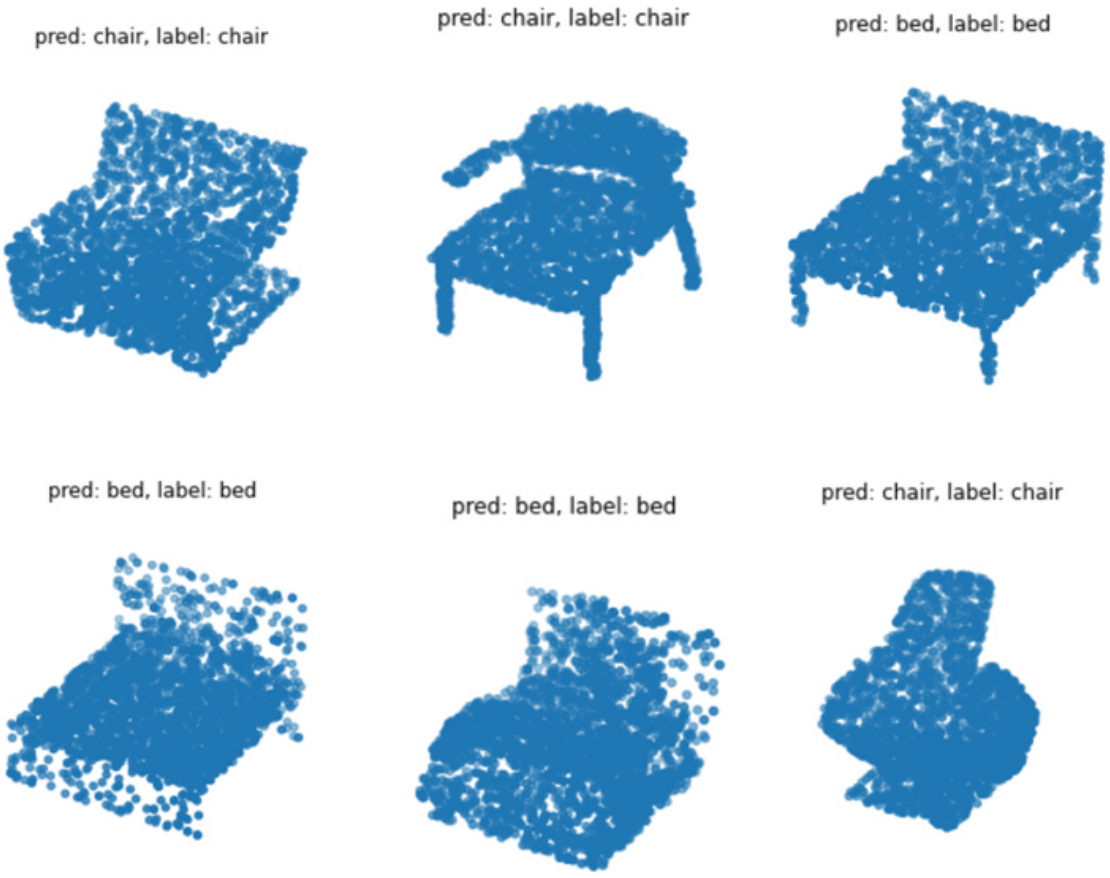


figure 61: results of training

4. Results

All the below training processes have stable learning rate equal to 0.001 and batch size equal to 32.

4.1. Dataset (1)

First we use a dataset that includes 2 classes with 1404 training samples (889 chair data, 515 bed data) and 200 test samples (100 on each of the two classes).

By following the above code, we train the net for two cases. In our first case we use non-normalized data .The results are shown in the matrix below where "Points" represents the number of points included on each testing and training element of the dataset, "Epoch" stands for the number of training epochs and "Accuracy (Acc)" is the values of "Val_categorical_acc".

Points	Epochs	Accuracy (Acc)	loss
2000	20	89%	1.2995
2000	30	95%	1.2102
1000	20	94%	1.3045
1000	30	97%	1.2076
800	20	96%	1.2613
800	30	90.5%	1.2089
500	30	94%	1.2401
300	20	86%	1.2751
300	30	88%	1.2397

Each row represents a training occasion. For example on the first case we test the net for 20 epochs, with 2000 points on each element of the dataset, the accuracy level was less than 90% and the loss value equals to 1.2995.

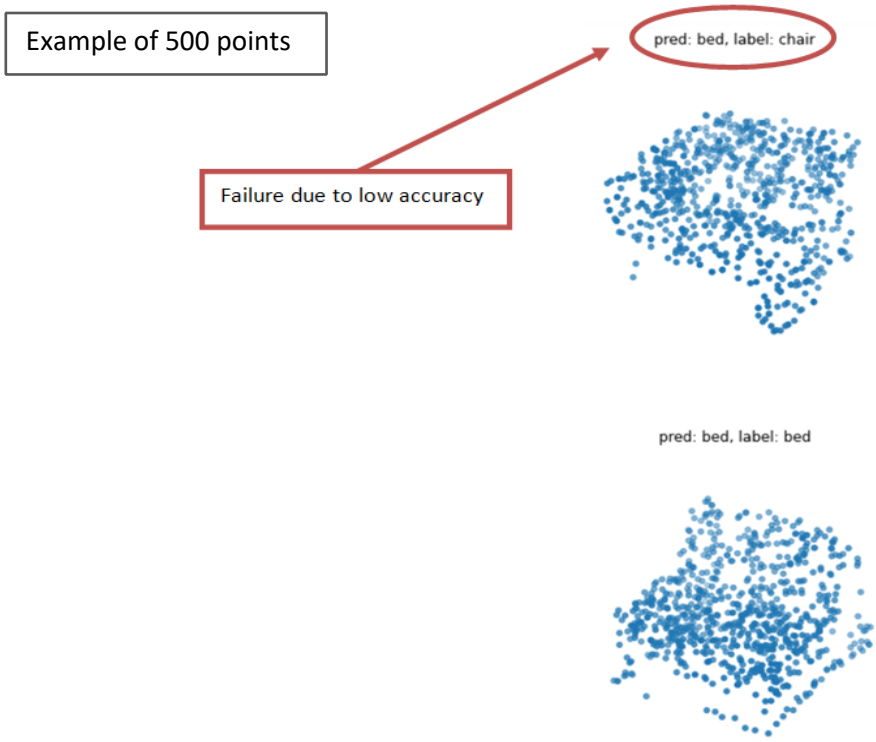


figure 62: results of non normalized dataset (1)

Seeing the above accuracy results we can say that the network performs well when using many elements for training. Also, in some cases, the accuracy levels are decreasing during epoch progress.

In our second training case we use normalized coordinate values between -1 and 1. The results are shown in the matrix below.

Points	Epochs	Accuracy (Acc)	loss
2000	30	98%	1.0709
1000	30	97%	1.1387
3000	20	95%	1.0934
3000	30	100%	1.0628
800	20	99.5%	1.1492
500	20	98.5%	1.0711
500	30	95.5%	1.1344
300	30	100%	1.0960

Normalization applies identical weights to all the data variables to avoid the guidance of model's performance in one way just because of bigger models in dataset. It is not necessary to normalize every dataset for machine learning. It is only necessary when the ranges of characteristics are different.

The table above shows the improvement of loss values and accuracy levels comparing to the previews matrix where non-normalized coordinates had been used.

Example of 500 points

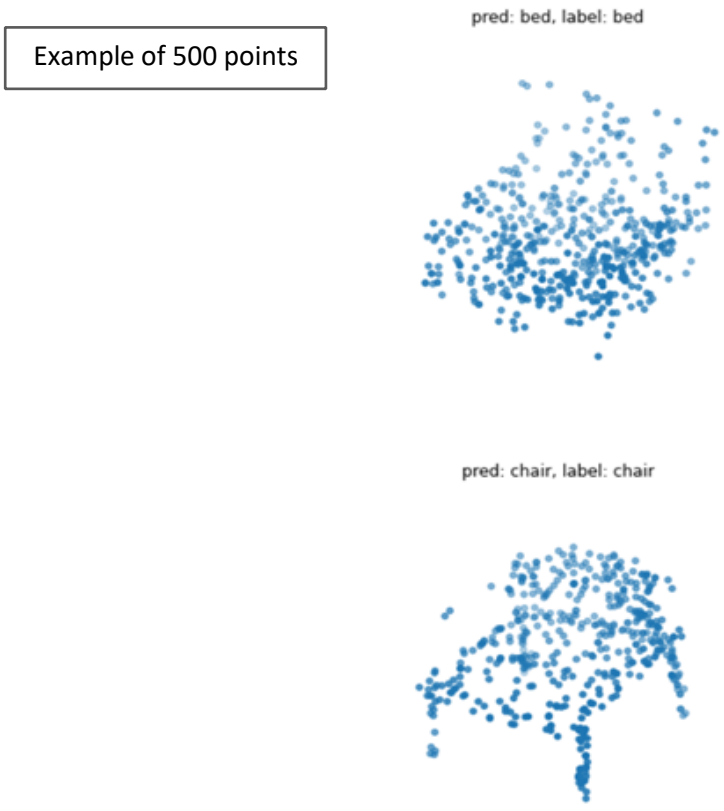


figure 63: results of normalized dataset (1)

4.2. Dataset (2)

Seeing the results of training with **Dataset(1)** we wonder how the system reacts after training with less values in 2 classes. For this purpose we create a dataset with 200 training samples (100 on each class) and 20 test samples (10 on each class) that we choose randomly from the original dataset.
Again we train PointNet for two cases.

Points	Epochs	sparse_ categorical_ accuracy	val_sparse_ categorical_ accuracy	loss
2000	30	72%	77%	1.8486
1000	40	86%	72%	1.5760
3000	30	83%	72.7%	1.6047
3000	40	81%	86.3%	1.5517
1500	40	85%	90 %	1.4940
2400	20	82%	59%	1.8048
2400	40	89%	90%	1.4564
500	30	80%	86%	1.6447
500	40	85%	90%	1.6946
800	30	73%	81%	1.8266
800	40	87%	86%	1.6460
300	40	83%	81%	1.6281

The matrix above represents the results of training with non-normalized coordinate values. Comparing with the accuracy and loss values of "**Dataset (1)**" we can see that PointNet has a lower accuracy level and the "loss" value is bigger after training with smaller dataset and non-normalized coordinates.

Points	Epochs	sparse_ categorical_ accuracy	val_sparse_ categorical_ accuracy	loss
2000	30	99%	95%	1.1026
1000	30	99%	100%	1.1247
3000	30	100%	95%	1.0390
1500	40	100%	95 %	1.0855
2400	20	99%	86%	1.0796
2400	40	100%	90%	1.0353
500	30	99%	95%	1.1156
500	40	97%	90%	1.2425
800	30	99%	95%	1.0928
800	40	95%	100%	1.2747
300	30	99%	100%	1.1238

The table above represents the training with normalized coordinates and shows

the improvement of accuracy and loss levels.

4.3. Dataset (3)

Finally we create an even smaller dataset with 100 random training samples (50 on each category/class), 20 random test samples and train again the net with normalized and non-normalized coordinates.

Points	Epochs	sparse_ categorical_ accuracy	val_sparse_ categorical_ accuracy	loss
2000	40	88%	90%	1.4255
1000	40	82%	81%	1.5353
3000	30	76%	50%	1.5476
3000	40	80%	50%	1.6270
2400	20	84%	86%	1.4742
2400	40	90%	86%	1.4159
500	30	76%	90%	1.0417
500	40	86%	95%	1.4739
800	40	73%	54%	1.7860
800	30	75%	95%	1.7033

Points	Epochs	sparse_ categorical_ accuracy	val_sparse_ categorical_ accuracy	loss
2000	40	98%	90%	1.0671
1000	40	100%	100%	1.0414
3000	30	99%	95%	1.1134
3000	40	100%	100%	1.0333
2400	20	98%	95%	1.1055
2400	40	100%	100%	1.0355
500	30 (απο11)	100%	100%	1.0417
500	40	100%	100%	1.0741
800	30	100%	100%	1.0679
800	40	100%	100%	1.0456
300	40	99%	100%	1.2131

The last matrix contains the results of the training with normalized coordinate values and, as we can see, they are improved.
All the above results lead as to the conclusion that no matter the dataset size we can achieve a better accuracy by using normalized coordinate values.

4.4. Dataset (4)

After all the above results, we made sure that the network works properly. So we create a new test set, irrelevant to ModelNet10, with only one bed and one chair. For this purpose, we download two new objects in ".obj" file format. By using Meshlab we change them to ".off" and then convert them to ".xyz" file format in the same way as the previous datasets. Finally, we train the net again with the train set included in "ModelNet10" dataset and use our new test set to see whether the objects will be recognized correctly.

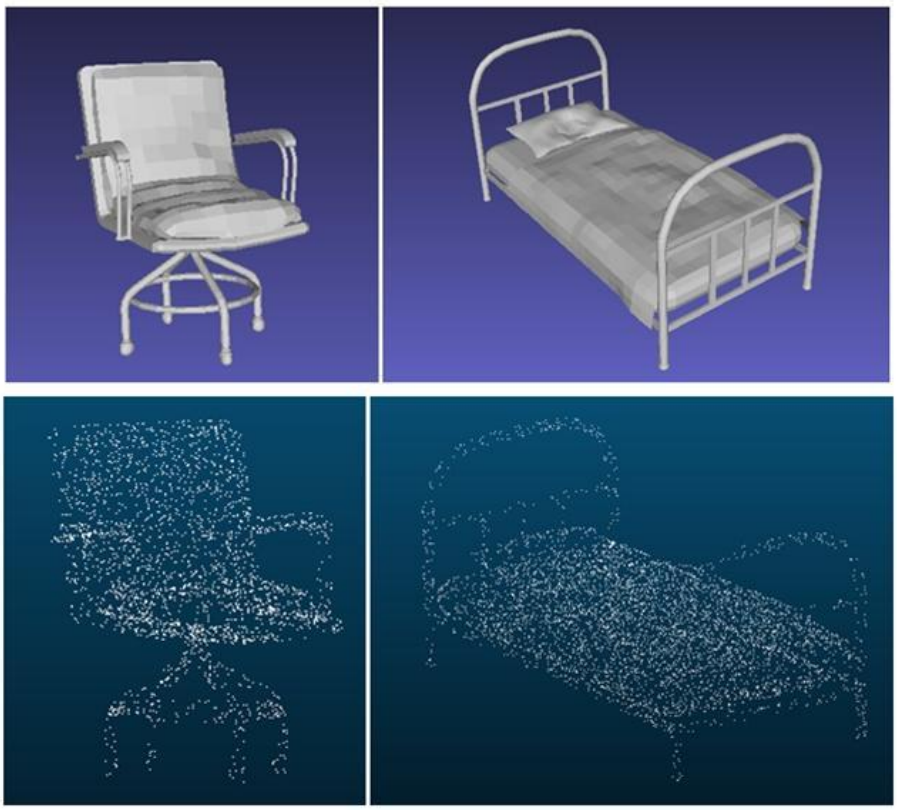


figure 64: : ".obj" data included in the new test set with their point clouds.

The network successfully recognized the objects after using the training set included to "ModelNet10" for "bed" and "chair" categories. The training was 100% accurate for 1000 points, 20 epochs and normalized coordinate values.

The results are shown bellow.

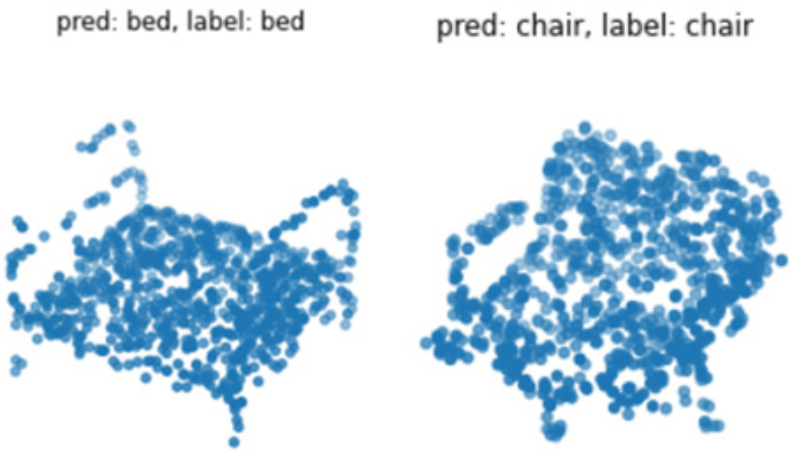


figure 65: training outputs

It is worth mentioning that the accuracy results are likely to differ from those reported above because each time we train a network the weights are updated differently.

4.5. Dataset (5)

Finally, we use the whole “ModelNet10” dataset to compare the accuracy results after training with normalized and non normalized data.

As we mentioned before “ModelNet10” includes 10 deferent categories with data in “.off” file format. Each of them contains a number of train and test data spitted into two subfiles with corresponding names. The first class is “bathtub” and includes 50 data for test and 106 data for training. The next class, “bed”, includes 100 test data and 515 train data. The category “chair” has 889 data for training and 100 for test. “desk”, “dresser” and “night_stand” categories includes 86 test and 200 train elements on each. The class named “monitor” includes 100 data for testing and 465 data for training. The category “sofa” has 100 test data and 680 train data. “table” class includes 100 test data and 392 train data. Finally the last category, “toilet” has 100 data for testing and 344 for training.

figure 66: ModelNet10 objects

To change the data format from ".off" to ".xyz" we follow the same preprocess methodology as described before but this time for all classes.

```
train_set = []
test_set = []

folders = glob.glob(os.path.join(path, "[!README]*"))

for i, folder in enumerate(folders):
    print("Class: {}".format(os.path.basename(folder)))

    #training set-------------------------------------------------------------
    train_files = glob.glob(os.path.join(folder, "train/*"))
    for f in train_files:
        train_set.append(trimesh.load(f))

    #test set-----------------------------------------------------------------
    test_files = glob.glob(os.path.join(folder, "test/*"))
    for f in test_files:
        test_set.append(trimesh.load(f))
```

figure 67: code to access a file

```
#test-----------------------------------------------
test_data = []
y=len(test_set)

for i in range(0,y):
    a,k=test_set[i].vertices.shape
    test_data.append(test_set[i].sample(a))

#train-----------------------------------------------
train_data = []
w=len(train_set)

for i in range(0,w):
    a,k=train_set[i].vertices.shape
    train_data.append(train_set[i].sample(a))
```

```
nnum3=[]
k1=0
for i in range (0,len(train_set)):
    nnum3.append("p{}.xyz".format(i))

for i in range (0, len(train_data)):
    with open(nnum3[k1], mode='w') as f:
        for j in range(len(train_data[k1])):
            f.write("%f    "%float(train_data[k1][j][0].item()))
            f.write("%f    "%float(train_data[k1][j][1].item()))
            f.write("%f    \n"%float(train_data[k1][j][2].item()))
    k1=k1+1
```

```
nnum3=[]
k1=0
for i in range (0,len(test_set)):
    nnum3.append("ptest{}.xyz".format(i))

for i in range (0, len(test_data)):
    with open(nnum3[k1], mode='w') as f:
        for j in range(len(test_data[k1])):
            f.write("%f    "%float(test_data[k1][j][0].item()))
            f.write("%f    "%float(test_data[k1][j][1].item()))
            f.write("%f    \n"%float(test_data[k1][j][2].item()))
    k1=k1+1
```

figure 68: code for creating ".txt" files

Then we split all the new data in 2 subfiles on each of the 10 categories without changing the original number of included objects.

By following the same methodology we train our data with non - normalized coordinates. The following matrix shows the results of training. Each row represents a case.

points	epochs	Sparse_categorical_acc	Val_categorical_acc	loss
3000	20	80%	77%	1.6830
2400	20	85%	88%	1.5425
2400	30	87%	90%	1.4609
1000	20	76%	83%	1.5968
800	20	81%	59%	1.6892
800	30	86%	89%	1.4981
500	20	85%	83%	1.5548

As we can see the accuracy results are good but not satisfactory. After training with normalized coordinates these results are way better. The below matrix shows improved accuracy levels and the loss value is lower.

points	epochs	Sparse_categorical_acc	Val_categorical_acc	loss
3000	20	93%	96%	1.1946
2400	20	96%	93%	1.2089
1000	20	96%	95%	1.1932
800	20	95%	88%	1.2377
800	30	97%	94%	1.1721
500	20	96%	93%	1.1969
500	30	97%	96%	1.1927

As a general conclusion from all the above results, after training with 2 or more classes and normalized or non–normalized data, we notice that the number of classes and data they contain does not affect the performance of the network when training with normalized coordinates.

4.6. Bonus category!

In addition of a project undertaken by the MC lab of Hellenic Mediterranean University we had in our disposal 20 clouds of points from churches. In this case we create a dataset with 2 categories and we split our data. The first class has churches without dome and took the name “Basilica”. The second category includes churches with dome and took the name “Basilica_with_dome”. Both the two names are based on the architectural rhythm of the churches.

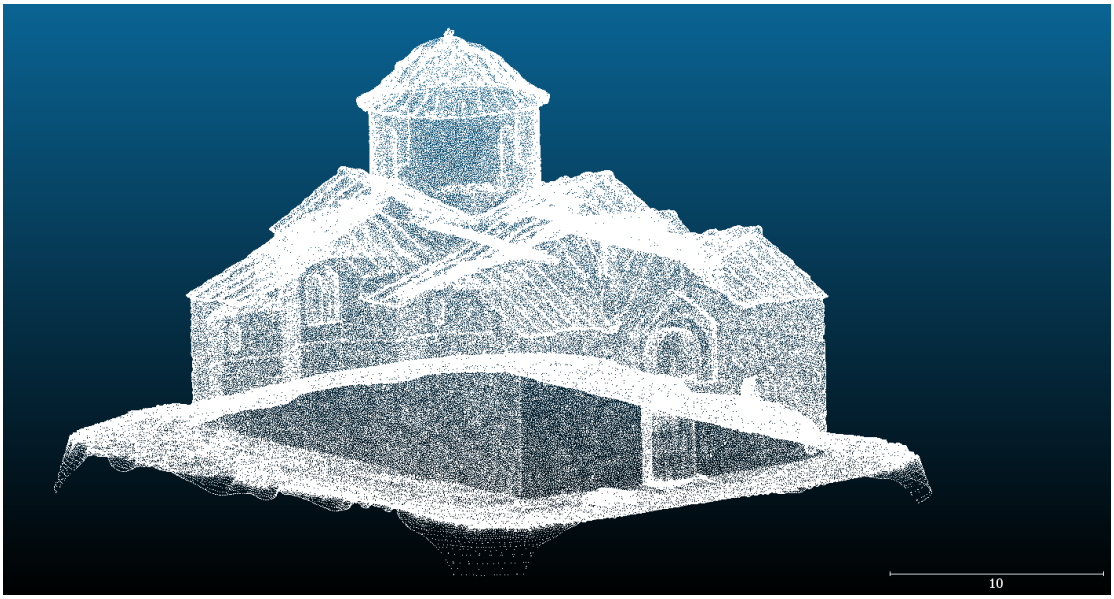


figure 69: point cloud of a church with dome.

The data preprocessing and processing was the same as the above datasets. Training achieved really fast because of the small amount of elements included in train and test set.

The blue matrix represents the results of training with non-normalized data and the green one with normalized coordinate values.

Points	Epochs	Accuracy (Acc)	loss
2000	30	50%	0.3070
2000	40	75%	0.2286
2000	50	75%	0.2220
1000	30	75%	0.3055
1000	60	50%	0.2185
3000	40	50%	0.2244
1500	40	50%	0.2366
800	40	50%	0.2275
2400	40	75%	0.2598
2400	60	50%	0.2107
500	60	75%	0.2146
4000	40	50%	0.2146
800	30	75%	0.2559
800	60	75%	0.2054
300	60	25%	0.2028

Points	Epochs	Accuracy (Acc)	loss
2000	30	75%	0.3560
2000	40	75%	0.2560
1000	30 (26/30 είχε 100% Acc)	100%	0.2976
3000	30	75%	0.2679
3000	50	75%	0.2116
1500	30	75%	0.2863
800	40 (30/40 είχε 100% Acc)	100%	0.2454
2400 (προτεινόμενο από keras)	30	75%	0.2877
2400	40	75%	0.2554
500	30	100%	0.3131
4000	30	25%	0.2760
800	30	100%	0.2937
300*	30	100%	0.2464

The accuracy results for normalized coordinate data shows that the network works better in fewer points when training with small datasets.

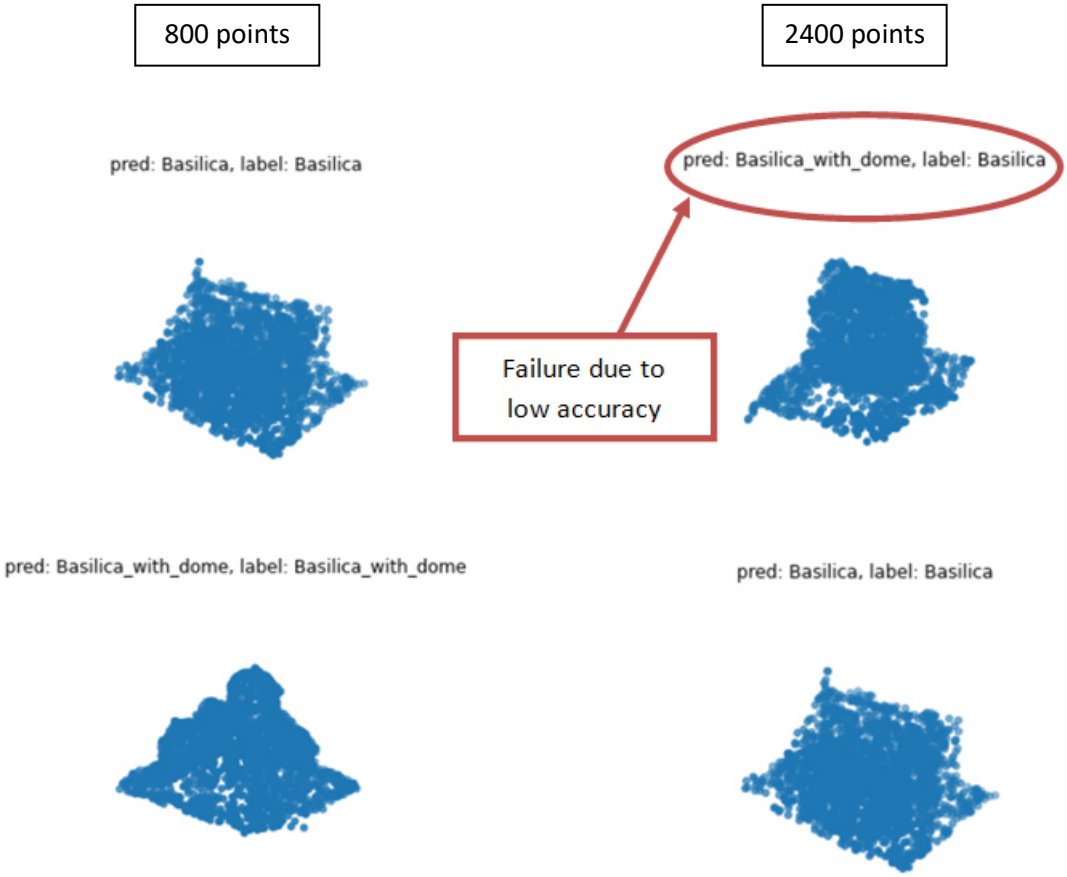


figure 70: Accuracy results

5. Conclusion

A point cloud is a form of data structure used to store geometric shape data. Because of its unstructured format, it's frequently converted into normal 3D voxel grids or collections of photos before being employed in deep learning systems.

Most of the times classifications are done in natural models that come directly from digitalization (laser scanning or photogrammetry) thus they are always in a point cloud format. For this purpose it is more efficient to use algorithms that do not required post processing of raw data. In this way we reduce computation requirements and human involvement in the process. In this thesis we deal with PointNet algorithms that consume raw data (cloud of points) instead of processed data (mesh).

In PointNet, the basic idea for classification and segmentation in point clouds is to calculate the distance between points. This allows us to condense points that are close into a single point by grouping them in small “boxes”. This method may be used to summarize geometric information and eventually name the complete point cloud. A contribution of our thesis is that we increased affiance of the algorithm by normalizing the coordinates of the points expressing all different models inside the same normalized coordination system.
In general, normalization is a data preparation method that is frequently used in machine learning. The goal is to convert the values of numeric columns in a dataset to a similar scale without distorting the ranges of values. It is frequently used to reduce training time and achieve better results.

In our experiments, the accuracy levels shows that no matter the dataset size, the detection levels are improved by using normalized coordinate values. Also the network works better with low number of points on each element for both training and test set. The combination of these two, makes PointNet a fast and effective network.

6. Bibliography

Deep Learning

1. https://el1.warbletoncouncil.org/inteligencia-artificial-2346
2. https://en.wikipedia.org/wiki/Artificial_intelligence
3. https://isqlplus.com/big-data-analytics/data-analytics-vs-ai-vs-machine-deep-learning/
4. https://en.wikipedia.org/wiki/Deep_learning
5. https://en.wikipedia.org/wiki/Machine_learning

Neural Networks

1. https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73
2. https://www.mathworks.com/discovery/deep-learning.html
3. https://en.wikipedia.org/wiki/Artificial_neural_network
4. https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964
5. https://en.wikipedia.org/wiki/Multilayer_perceptron
6. https://en.wikipedia.org/wiki/Backpropagation
7. https://www.sciencedirect.com/science/article/pii/S2666351121000358
8. Artificial Networks, Konstantinos Diamantaras
9. Deep Learning with Python, Francois Chollet
10. Geometric Deep Learning, Jonathan Masci, Emanuele Rodolà, Davide Boscaini, Michael M. Bronstein, Hao Li

Computer graphics

1. Graphics: Principles and algorithms, TheocharisTheocharis, Alexandros Bem
2. https://el.wikipedia.org/wiki/%CE%93%CF%81%CE%B1%CF%86%CE%B9%CE%BA%CE%AC_%CF%85%CF%80%CE%BF%CE%BB%CE%BF%CE%B3%CE%B9%CF%83%CF%84%CF%8E%CE%BD
3. https://en.wikipedia.org/wiki/Rendering_(computer_graphics)
4. https://www.haroldserrano.com/blog/rotations-in-computer-graphics
5. https://www.javatpoint.com/computer-graphics-translation
6. https://en.wikipedia.org/wiki/Voxel
7. https://en.wikipedia.org/wiki/Polygon_mesh
8. http://what-when-how.com/advanced-methods-in-computer-graphics/mesh-processing-advanced-methods-in-computer-graphics-part-1/
9. https://geoslam.com/point-clouds/
10. https://info.vercator.com/blog/what-are-point-clouds-5-easy-facts-that-explain-point-clouds

Application

1. http://stanford.edu/~rqi/pointnet/
2. https://www.geeksforgeeks.org/pointnet-deep-learning/

3. https://www.tensorflow.org/
4. https://keras.io/examples/vision/pointnet/
5. https://keras.io/examples/vision/pointnet_segmentation/
6. Deep Learning with Python, Francois Chollet
7. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation, Charles R. Qi*, Hao Su*, Kaichun Mo, Leonidas J. Guibas, Stanford University
8. PointNetLK: Robust & Efficient Point Cloud Registration using PointNet, Yasuhiro Aoki, Hunter Goforth, Rangaprasad Arun Srivatsan,Simon Lucey, Carnegie Mellon University, Fujitsu Laboratories Ltd, Argo AI
9. PCRNet: Point Cloud Registration Network using PointNet Encoding, Vinit Sarode, Xueqian Li, Hunter Goforth, Yasuhiro Aoki, Rangaprasad Arun Srivatsan, Simon Lucey, Howie Choset, Carnegie Mellon University, Fujitsu Laboratories Ltd., Argo AI, Apple
10. OINTNET FOR THE AUTOMATIC CLASSIFICATION OF AERIAL POINT CLOUDS, M. Soilán, R. Lindenbergh, B. Riveiro, A. Sánchez-Rodríguez
11. Voxel-Based 3D Point Cloud Semantic Segmentation: Unsupervised Geometric and Relationship Featuring vs Deep Learning Methods, Florent Poux * and Roland Billen
12. Hands-on Graph Neural Networks with PyTorch&PyTorch Geometric, Steeve Huang

7. Figures

1. Figure 1: AI includes Machine Learning and Deep Learning is a part of Machine learning. , Created with Microsoft World
2. figure 2: Neural Network architecture. , Created with Microsoft World
3. Figure 3: Sigmoid function, image from Wikipedia https://en.wikipedia.org/wiki/Sigmoid_function
4. Figure 4: ReLU activation function , image from Machine Learning Mastery https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/
5. figure 5: Schematic diagram of an artificial neural network., Created with Microsoft World
6. Figure 6: gradient of loss function, Created with Microsoft World
7. figure 7: Learning rate performance, Created with Microsoft World
8. figure 8: Deep VS Shallow Neural Networks, Created with Microsoft World
9. figure 9, Created in Microsoft World
10. figure 10: three dimensions, Created with Adobe Photoshop CC 2019
11. figure 11: Additive color VS Subtractive color, Created with Adobe Photoshop CC 2019
12. figure 12: Color Depth. The first image is an example of an 8 bit.png with 256 colors, the second is a 4 bit.png with 16 colors and the last one is a 2 bit.png with 4 colors., Image from Wikipedia https://en.wikipedia.org/wiki/Color_depth
13. figure 13: 2D scene from the game “Super Mario”., image from pixabaysel https://pixabay.com/fr/images/search/2d%20game/