university of groningen / faculty of mathematics and natural sciences / artificial intelligence

Learning Optimal View Selection for Multi-View Object Representation

Andrei-Lucian Miculiță

university of groningen / faculty of mathematics and natural sciences / artificial intelligence

University of Groningen

Learning Optimal View Selection for Multi-View Object Representation

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of
S.H. Mohades Kasaei, Prof Dr (Artificial Intelligence, University of Groningen)
and
Prof. Dr. R. Carloni (Artificial Intelligence, University of Groningen)

Andrei-Lucian Miculiță (s4161947)

August 7, 2023

Acknowledgments

The author would like to give thanks to Dr. S.H. Mohades Kasaei, the supervisor, as well as Prof. Dr. R. Carloni, the co-supervisor, for their questions, feedback, guidance and expertise throughout the course of this work.

The author is also deeply grateful for the constant support and understanding of their family, friends and colleagues, despite numerous setbacks and challenges, which caused delays in project completion. Finally, the author would like to thank his therapist, whose encouragement played a significant role in overcoming personal obstacles and navigating the ups and downs of this project.

Abstract

Object recognition, pose estimation, and grasp affordance tasks for robots rely heavily on an accurate understanding of an object's geometry. However, this can often be challenging due to the absence of a reliable object representation. In this thesis, we propose a novel approach for learning a descriptive object representation by using a good view selection policy. Our method, called Maximum Entropy Viewpoint Selection (MEVS), selects the most informative view of an object to classify it. We present two alternative approaches for MEVS: (i) a differentiable rendering-based approach, optimizing view entropy, and (ii) a point cloud embedding-based approach, using PointNet++ for predicting depth entropy. MEVS is adaptable to new situations and environments, making it particularly valuable for robots deployed in dynamic settings. To assess our approach, we conduct evaluations on two adaptations of the ModelNet10 dataset, in a multi-view pipeline based on a ResNet backbone. We find that the point cloud embedding-based method is suitable for real-time applications and can consistently outperform random view selection in finding the most informative views of objects it has not seen before.

Keywords

Continuous Learning, Viewpoint Selection, Information Entropy, Point Clouds, Neural Networks, 3D Point Cloud Classification

Thesis Outline

The thesis is organised as follows: Chapter 1 provides an introduction to the topics of this thesis. Chapter 2 provides some more concrete examples of previous work, some of which this thesis builds on. The main research questions are summarized in Chapter 3. Chapter 4 presents the proposed method for the estimation of the optimal set of views for the representation of a given object. Chapter 5 presents the experimental results obtained with the proposed methods. Finally, Chapter 6 presents the conclusions and possible avenues for future work.

Contents

List of Figures

List of Tables

1 Introduction

One major challenge in robotic manipulation is to plan and execute a sequence of actions that achieves a desired goal. To achieve this, when manipulating objects, the robot must be able to perceive the object's shape and properties. This is a challenging task, as these are almost never known a priori. In this thesis, we focus on the problem of object recognition, which is the task of classifying the object into a set of known categories. This is a fundamental task in robotics, as it is required for many other tasks, such as grasping, manipulation, and navigation.

Specifically, we focus on the problem of object recognition based on multiple views of the object, a particularly demanding task due to the diverse viewpoints and substantial appearance variations an object can exhibit due to them. Based on an initial view of the object, the robot must be able to estimate the optimal set of additional viewpoints to explore, capture images from these viewpoints, and then recognize the object based on the compiled set of images.

1.1 Object Representation

The problem of representing 3D objects has been a central topic in both computer vision and computer graphics for many years. In computer vision, the goal is to extract useful information from given images. The information required depends on the application, but for many applications, 3D information is required. This is particularly the case for applications involving recognition, where it is desirable to be able to represent 3D objects in a way that is invariant to viewpoint. In computer graphics, on the other hand, the goal is to generate images. In order to do this, it is necessary to be able to represent 3D objects in a way that makes it easy to determine what the object would look like from any viewpoint. Historically, the two problems have usually been considered separately, with different representations being used for each.

For the representation and classification of 3D objects, several types of descriptors can be used. The following sections present some of the most common types of descriptors.

1.1.1 3D Descriptors

3D descriptors are based on the 3D shape of the object. They contain information about the object's geometry, and are therefore invariant to viewpoint. The most common types of 3D descriptors are the following:

- **Meshes**: A mesh (or polygonal mesh) is a collection of vertices, edges, and faces that represent the surface of a 3D object (a polyhedron). The faces are most commonly triangles, but they can also be quadrilaterals, or any other polygon. Most commonly, meshes are stored as a set of 3D coordinates (x, y, z) for the vertices, and a set of faces, where each face is a list of vertex indices. The vertices are connected by edges, which are not explicitly stored. Meshes are a very common representation of 3D objects, and are used in many applications, including computer graphics, computer vision, and robotics. Meshes are also used to represent 3D objects in the original ModelNet dataset [76], which is used in this thesis.

- **Voxel Grids**: A voxel grid is a 3D grid of voxels, where each voxel is a cube. The grid is usually aligned with the coordinate axes, and the size of the voxels is usually constant. The voxels are usually represented as a 3D array of booleans, where each element in the array is either true or false. If the element is true, the voxel is occupied, and if it is false, the voxel is empty. Voxel grids can be thought of as a discretization of the 3D space, where each voxel represents a small volume of space. Thus, the size of the voxels determines the resolution of the grid. Voxels can also contain other information, such as color, a probability of occupancy, or a density value. Another way of thinking about voxel grids is as a raster image extended to 3D, where each pixel is instead a voxel.

- **Point Clouds**: A point cloud is a set of points in 3D space. The points are usually sampled from the surface of the object, and are therefore a good representation of the object's geometry. However, point clouds are not very compact, usually requiring thousands of points to represent a complex object. They are also not very robust to noise, as a single outlier can have a large effect on the shape of the point cloud. A point cloud can be represented as a set of 3D coordinates (x, y, z) for each point. The points can also contain other information, such as color (r, g, b), or a normal vector (nx, ny, nz). As opposed to voxel grids, point clouds are more unstructured, as they do not have a fixed resolution, and the point coordinates can take continuous values (to be precise, they take values represented by floating point numbers, which lose precision the larger they are).

However, another aspect to take into account is the fact that for real-life applications, such 3D representations are computationally more difficult to obtain in comparison with 2D representations, such as camera projections. This can be an issue, as in unstructured environments, robots have to be fast learners, and thus must be able to quickly integrate new information once it is presented to them [31].

1.1.2 2D Descriptors

The use of 2D descriptors, such as camera projections, can be more suitable for real-life applications, as they are easier to obtain. However, 2D descriptors come with their own set of issues. In particular, the use of 2D descriptors can lead to the loss of information, as the 3D structure of the object is not preserved. This can be a problem, as this structure is important for its representation. The most common types of 2D descriptors are the following:

- **Images**: An image is a 2D array of pixels, where each pixel is a color value. The color value can be represented in a variety of ways, but the most common is RGB, where each pixel is represented by three values, one for each of the red, green, and blue channels.

- **Depth Maps**: A depth map is a 2D array of depth values, where each depth value is a distance from the camera to the object. The depth values are usually represented as floating point numbers, and are usually obtained by using a depth sensor, such as a depth camera or a laser scanner. They are also sometimes referred to as range images. They can also be thought of as a particular subset of point clouds, which are discretized in a 2D plane in front

of the camera, and where the depth values are the distance from the camera to the point. This type of space is called the screen space. To convert from screen space coordinates to world space coordinates, assuming a pinhole camera model, we require the camera's intrinsic matrix, which contains the focal length and the principal point, and the camera's extrinsic matrix, which contains the camera's position and orientation. Even without the latter, we can obtain a correct representation of the object, but in an arbitrary position and orientation. Assuming a simple case where the camera origin and the world origin are aligned, the skew coefficient is zero, and the image sensor is centered, the formula for converting from screen space coordinates to world space coordinates is the following:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = z \begin{bmatrix} 1/f_x & 0 & 0 & 0 \\ 0 & 1/f_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \\ 1/d \end{bmatrix} \tag{1}$$

where x, y, and z are the world space coordinates, the middle term is the inverse of the intrinsic matrix assuming the principal point offset and the skew coefficient are zero, and the last term contains the screen space coordinates. For more complex cases, the reader is referred to [9].

These two descriptors can be combined to obtain RGB-D images, which are images that contain both color and depth information. This representation is often used in Computer Vision, as the sensor data is usually obtained in this format. In this project, we operate with both RGB-D images and point clouds.

1.2 Object Classification

The problem of object classification is well studied in computer vision and has been applied to a number of applications, including industrial inspection, autonomous driving, and robotics. Its goal is to assign an object to a certain class out of a given set, given a set of features that describe the object. Thus, it is often split into two tasks: feature extraction and classification. For the first, the features extracted should capture the discriminative properties of the object. These features can be 2D or 3D, and are most commonly based on shape, color, or texture. They can be hand-crafted (designed by a human), or learned (obtained by a machine learning algorithm). Some examples of features are depth (distance from camera), surface normals (direction of the surface), curvature (rate of change of the surface normal), contours (the boundary between the object and the background), edges (the boundaries between regions of different colors), local binary patterns (a histogram of the local neighborhood of a pixel), and SIFT (Scale-Invariant Feature Transform) features (a histogram of the local neighborhood of a pixel, where the neighborhood is defined by a Gaussian kernel). Another aspect to take into account when selecting features is their robustness to factors such as the object's pose, illumination and background. In this thesis, we only deal with neural-network-based classification, which is currently the most popular type of classification, but there are other machine learning methods that can be used for classification, such as Support Vector Machines (SVMs) and Random Forests.

1.2.1 Neural Networks

The use of neural networks for object classification has been a topic of interest for many years. Neural networks are a class of machine learning algorithms that are inspired by the structure of the brain. They are composed of a set of interconnected units, called neurons, which take a set of inputs, multiply them by a set of weights, add a bias, and then apply a non-linear function to the result.

Using neural networks for object classification was first proposed by LeCun et al. [40], who used them to classify handwritten digits. The idea was to use a neural network to learn the mapping between the input image and the output class label, where the input is a normalized 2D image representing a digit, and the output class label is a number between 0 and 9. This network was trained using backpropagation and gradient descent, which is a supervised learning algorithm that minimizes the error between the output of the network and the desired output [58]. It works by propagating the error backwards through the network, and then updating the weights of the network using the gradient of the error with respect to the weights. This error is called the loss, and can be defined using various loss functions (see Section 1.2.4).

1.2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural network that is particularly suited for image classification. They were first proposed by Lecun et al. [39]. They are composed of a set of convolutional layers, which are used to extract features from the input image, and a set of fully connected layers, which are used to classify the image. The convolutional layers contain convolutional filters, which are used to extract features from the input image. These convolutional filters are applied to the input image, and the result is passed through a non-linear activation function, such as the rectified linear unit (ReLU).

One of the main advantages of CNNs, as opposed to classical (i.e. fully connected) neural networks, is weight sharing, which means that the same convolutional filter is applied to the entire input image. This allows the network to learn features that are invariant to translation, which is a desirable property for object recognition in images. One particular extension of CNNs is the use of 3D convolutional filters, which are used to extract features from 3D voxel grids. Given the fact that, as stated earlier, 3D voxel grids can be thought of as an extension of 2D (pixel) images, it is natural to use an extension of 2D convolutional filters into 3D to extract features from them [4, 19, 61, 48].

1.2.3 Residual Neural Networks

Residual neural networks (ResNets) are a type of neural network that was first proposed by He et al. [22]. What makes them different from other neural networks is that they use skip connections, which are connections that jump over one or more layers (see Figure 1). A particular subset of ResNets are DenseNets, which have several parallel skips [24]. The main reasons for the use of skip connections are to reduce the vanishing gradient problem, which is a problem that occurs when the gradient of the loss function is very small, and to increase the depth of the network, allowing it to learn more complex features. The skip connections allow the network to learn residual

mappings, i.e. the difference between the input and the output of a residual block. These mappings are added to the output of the block, creating shortcuts that help in the backpropagation of gradients and enable training of deep networks more effectively. The original ResNet architecture is composed of multiple layers of residual blocks, which are composed of a set of convolutional layers, and a skip connection that jumps over the convolutional layers, without additional processing [22]. To improve performance, the number of layers can be increased; the authors of the original ResNet architecture evaluated networks with up to 152 layers. ResNets have been shown to be very effective for image classification, achieving an ensemble top-5 error of 3.57% on ImageNet and winning the 1st place in the ILSVRC 2015 classification competition [22].

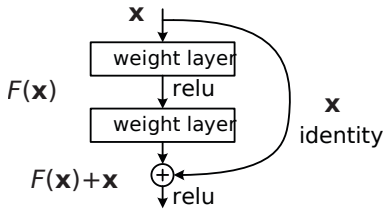


Figure 1: A residual block.

1.2.4 Loss Functions

The loss function is used to measure the error between the output of the network and the desired output. It is used to update the weights of the network during the training phase, as well as to evaluate the performance of the network during the testing phase. Different loss functions are used for classification problems, where the output of the network is a probability distribution over the classes, and regression problems, where the output of the network is a continuous value. The most common loss functions are the following:

- **Mean Squared Error (MSE)**: The MSE is defined as the mean of the squared difference between the output of the network and the desired output. It is used for regression problems. It is also known as L2 loss. The MSE is given by:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2}$$

  where N is the number of samples, y_i is the desired output of the i-th sample, and $\hat{y}_i$ is the output of the network for the i-th sample.

- **Mean Absolute Error (MAE)**: The MAE is defined as the mean of the absolute difference between the output of the network and the desired output. It is also used for regression problems. It is also known as L1 loss. As opposed to the MSE, it is more robust to outliers, as it does not penalize large errors more, but it is less efficient, as its derivative is not continuous at 0. The MAE is given by:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{3}$$

using the same notation as for the MSE.

- **Cross-Entropy Loss (CE)**: The CE is defined as the negative log-likelihood of the output of the network. It is used for classification problems. As it is logarithmic, it penalizes large errors more than small errors. It is also sometimes called Logistic Loss and Multinomial Logistic Loss. It is given by:

$$\mathrm{CE} = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij}) \tag{4}$$

  where N is the number of samples, C is the number of classes, y_{ij} is the desired output of the i-th sample for the j-th class (either a 0 or a 1, indicating if the sample belongs to the class), and $\hat{y}_{ij}$ is the output of the network for the i-th sample for the j-th class.

Loss functions can be altered and combined to suit the needs of the problem at hand. For example, terms can be added to the loss function to penalize the network for being too complex, which is known as regularization (see Section 1.2.6).

1.2.5 Optimization Algorithms

The optimization algorithm is used to update the weights of the network during the training phase. The first optimization algorithm that was proposed for training neural networks, as described in section 1.2.1, was the gradient descent algorithm. Other such algorithms have been proposed since then, to address some of its shortcomings. The following are some of the most common:

- **Stochastic Gradient Descent (SGD)**: The SGD is a variant of the gradient descent algorithm, where the gradient is computed using a randomly selected subset of the training data, instead of the entire dataset. This makes it much faster. It can be traced back to the Robbins-Monro algorithm, which was proposed by Robbins and Monro [57].
- **RMSProp**: The RMSProp is an improvement upon the gradient descent algorithm, where an adaptive learning rate is computed using a moving average of the squared gradients (with exponential forgetting). It was proposed by Geoffrey Hinton in one of his Coursera lectures [1].
- **Adam**: Adam (Adaptive Moment Estimation) is a variant of the RMSProp algorithm, which incorporates Momentum [58]. Thus, it uses a moving average of both the first and second moments of the gradients. It was proposed by Kingma and Ba [34], and is one of the most commonly used optimization algorithms for training neural networks.

1.2.6 Regularization

Regularization is a technique that is used to prevent overfitting, which is a common problem in machine learning. Overfitting occurs when the model is too complex for the amount of data that

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

is available, and it memorizes the training data, instead of learning the underlying distribution. Regularization is used to prevent this problem by penalizing the network for having large weights, which forces the network to learn a simpler model. A more complete survey of regularization techniques can be found in Kukaka et al. [37]. To give a short overview of regularization techniques, the following ones are commonly used:

- **Dropout**: Dropout is a regularization technique that was proposed by Srivastava et al. [68]. It consists of randomly dropping a set of neurons during the training phase, which forces other neurons to compensate for the missing neurons.

- **Batch Normalization**: Batch normalization is a regularization technique that was proposed by Ioffe and Szegedy [25]. It consists of normalizing the output of each layer, which causes the network to learn faster and more robustly:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}} \tag{5}$$

  where μ is the mean of the minibatch, σ is the standard deviation of the minibatch, and ε is a small constant for numerical stability.

- **Weight Decay**: Weight decay (or L_2 regularization) is a regularization technique that uses a penalty term on the L_2 norm of the weights, which is added to the loss function:

$$L(\mathbf{w}) = L_0(\mathbf{w}) + \frac{1}{2}\lambda \sum_i w_i^2 \tag{6}$$

  where L is the loss function, λ is a hyperparameter that controls the strength of the regularization, and w is the weight vector. Within the context of deep learning it was proposed by Hinton [23], based on earlier work by Tikhonov [72].

- **Data Augmentation**: Data augmentation is a regularization technique that can be used to address the issue of limited training data. It consists of generating new training examples by applying random transformations to the training data, such as rotation, translation, and scaling. It is different from other regularization techniques, as it is applied to the training data, instead of the network. In the context of point clouds, frequent transformations are rotation (usually around the vertical axis), translation, scaling, flipping, ground removal, and jitter (adding random translations to the points individually) [21]. These can be done both globally (for the entire point cloud at once) and locally (whereby a random transformation is applied in every annotated bounding box) [21].

  Data augmentation can make use of domain knowledge based on the dataset, as only some transformations are valid for a given problem. For example, rotating an image of a text character too much could change its meaning, and thus its label. A more comprehensive survey of data augmentation is given by Shorten and Khoshgoftaar [64].

1.3 Next-best View Selection

The task of object recognition and classification is challenging due to the large intra-class variance that is exhibited by an object class. This variance is a result of the object's geometric and photometric properties being influenced by a number of factors such as the viewpoint, illumination, background, and occlusion.

In the past, the object recognition problem has been tackled from a single view perspective. In this approach, the object is recognised from only one image, and does not take into account the information that can be gained from other views that the object can be seen from. This approach has a number of limitations. Firstly, the information gained from a single view is limited by the viewpoint of the camera. As a result, it may not be possible to recognise the object due to self-occlusion. Secondly, while it may be possible to recognise the object from a number of viewpoints, it is not straightforward to determine which viewpoint will result in the most reliable recognition.

In order to tackle this problem, the multi-view object recognition approach has been introduced, improving joint object categorization and pose estimation [16] [80]. Provided the prevalence of multi-view data in real-world applications, it is important to be able to encode a description of the object that can aggregate the information gained from multiple images. In this approach, the task of object recognition is carried out in a step-wise fashion, by first observing the object from a random view, and then using this view to select other views that are likely to be useful.

Next-best view selection is the process of determining the next camera position to be used in order to obtain the most information about the object being observed. This is done by maximizing the information gain, while optionally taking into account other factors, such as the cost of obtaining this new view (such as the energy cost or time taken to move the camera, the predicted amount of data required), as well as physical constraints (such as the limited configuration space of the robot's arm).

This is an active approach to object recognition as the viewpoint of the scene is changed in order to gain the most information from the scene. Active vision is a branch of computer vision that focuses on improving the vision process by actively interacting with the environment. In this way, the process can benefit in terms of both speed and accuracy.

1.4 Differentiable Rendering

Differentiable rendering is a field of research that can be used to combine computer graphics with computer vision (as mentioned in Section 1.1). It is the process of rendering a scene in a way that is differentiable with respect to the scene parameters. As an area that has grown rapidly in the past few years, differentiable rendering is finding many applications in a wide variety of fields, including inverse graphics, robotics, and computer vision. A brief overview of the field of differentiable rendering follows, including the mathematical principles behind the techniques, as well as the many applications that they are being used for.

The aim of differentiable rendering is to create rendering pipelines that are differentiable, meaning that they are able to take gradients of some error function with respect to the parameters of the rendering pipeline. A key benefit of this is composability: by putting together differentiable rendering pipelines, it is possible to create complex end-to-end rendering pipelines that are differentiable

themselves in their entirety.

1.4.1 Inverse Rendering

Inverse rendering is the process of estimating the scene parameters that best explain a given set of observations. This can be formulated as the following optimisation problem defined in Equation 7.

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\theta) \tag{7}$$

where θ is the set of scene parameters, and $\mathcal{L}(\theta)$ is the loss function measuring the error between the observations and the rendered images. θ can be, for example, the combination of intrinsic and extrinsic camera parameters, along with the parameters of the scene geometry and lighting.

In general, the inverse rendering problem is ill-posed. This is due to the fact that the observations only provide information about the scene from a small number of viewpoints, and for most scenes there are many possible viewpoints that could explain the observations. To solve this problem, priors need to be placed on the scene parameters. One way of doing this is to encode the priors in the form of a differentiable rendering function. This differentiable rendering function is then used in the loss function, and the scene parameters are found by minimising the loss.

Li [43] gives a straightforward explanation of differentiable rendering functions. Since the forward rendering integral includes a discontinuous visibility term, it is not differentiable in traditional renderers [43]. Both approximate solutions [32, 74] and exact solutions [43] exist for this issue. The approximate methods, which are used in this thesis, are detailed in Sections 2.2 and 2.3.

1.4.2 Automatic Differentiation

Automatic differentiation (AD or *autodiff*) refers to the conversion of a program composed of a number of elementary operations (a computational graph) that has a set of input and a set of outputs into a program that computes the derivatives of the outputs with respect to the inputs [2, 75]. In short, it relies on the chain rule, applied repeatedly to the elementary operations in the computational graph, parsed in reverse order.

Automatic differentiation is also used in deep learning frameworks such as PyTorch [53] and TensorFlow [1]. They leverage autodiff by building the graph of the operations performed on tensors during the forward pass of the neural network, in order to compute the gradients during the backward pass. This enables developers to more easily define and train complex neural networks, as they do not need to manually define gradient functions.

Similarly, by treating the rendering algorithm as a computational graph, it is possible to use autodiff to compute the derivatives of the loss function with respect to the scene parameters. The scene parameters are usually some combination of the geometry, material properties, lighting, and camera parameters. This differentiability can then allow for the inverse rendering problem to be solved by gradient descent.

In general, differentiable rendering can be broken down into two main components:

- A forward rendering component, which computes the images to be rendered from the scene parameters.

- A differentiable component, which computes the gradients of the loss function with respect to some (usually not all) scene parameters.

When discussing differentiable rendering, also of interest is the work of Mildenhall et al. [49], which represents a scene as a neural radiance field. It takes a sparse set of input view images and uses a differentiable volumetric renderer to render the scene from any viewpoint. It has been used for synthesising novel views of scenes, as well as for detecting objects and segmenting scenes. One benefit of this approach is that it can render scenes with complex lighting effects (reflections and refractions) with high fidelity. However, to date, no method has been proposed to encode priors into neural radiance fields which could be used to infer the geometry of occluded objects/parts of objects. This makes neural radiance fields unsuitable for use in the maximum entropy view selection framework we propose, which operates on the assumption that the model should predict the entropy of unseen views even when they contain currently occluded parts of objects. However, encoding such priors may be a promising avenue for future research.

1.5 Point Cloud Embedding

Point cloud embedding refers to the process of converting a point cloud into a lower-dimensional representation, i.e. a vector. This vector representation can then be used as input to a machine learning model, to perform tasks such as classification, segmentation, and generation. Segmentation refers to the process of assigning a label to each point in the point cloud, while generation refers to the process of generating a new point cloud from the vector representation. The point cloud input consists of a set of points with positions, along with other optional features such as colour and normal vectors.

There are several techniques for employing neural networks for point cloud embedding, including projection-based methods and voxel-based methods. Projection-based methods project the point cloud onto a 2D plane, which may subsequently be processed using convolutional neural networks (CNNs). On the other hand, voxel-based methods convert the point cloud into a voxel grid, to be processed with 3D convolutional neural networks (3D CNNs).

A variety of techniques have recently achieved good performance on various point cloud benchmarks. The PointNet and PointNet++ designs, for example, have demonstrated success in tasks such as classification and segmentation, while the VoxelNet architecture has demonstrated effectiveness in object recognition in point clouds [8, 54, 83]. We detail the PointNet architectures, which are used in this thesis, in Section 2.4.

2 Related Work

A brief overview of the works related to the topic of this thesis is provided in this section. On the topic of Multi-view Object Recognition, we describe the state of the art in the field. On the topics of differentiable rendering and point cloud embeddings, we treat in particular three main solutions: (i) The Neural 3D Mesh Renderer [32], (ii) Differentiable Direct Volume Rendering [74] and (iii) PointNet and PointNet++ [8, 54].

2.1 Multi-view Object Recognition

Several methods have been proposed for object recognition, which can be broadly classified into two categories: 2D and 3D. As several well-performing neural network architectures have been proposed for 2D object recognition, it is natural to extend these as pretrained backbone networks for multi-view object recognition. The use of multiple views can be beneficial for object recognition, as views can provide complementary information about the object, reducing the effect of occlusion. Qi et al. [55] provide a comprehensive review of the state of the art in multi-view object recognition. Other works that provide a review of 3D object recognition, while not referring exclusively to multi-view object recognition include [59], [6], and [18].

The Multi-View Convolutional Neural Network (MV-CNN) is one of the earliest approaches that was proposed for multi-view object recognition [69]. Su et al. [69] achieved substantially improved results than earlier attempts by employing Convolutional Neural Networks on multiple rendered 2D views of the object. This is owing, in part, to the high resolution that 2D images provide.

Given a point cloud representation of an object, a scale- and rotation-invariant method of performing multi-view object classification is to use three orthographic projections, such as in OrthographicNet [30]. This method relies on the eigenvector of the covariance matrix of the 3D object (represented as a point cloud), which is used to determine the three orthogonal directions. The object is then classified based on a synthesized collection of 2D images captured from these directions (forming a rotation invariant global feature) [30].

Alternatively, Zhou et al. [81] propose the Multi-View Saliency Guided Deep Neural Network (MVSG-DNN). This system has the following steps: projecting rendered views, learning the visual context (using CNNs), followed by the selection of the most representative views for classification, based on saliency, and finally, the compilation of 3D object descriptors with a classification LSTM, for object retrieval and classification. This method achieves comparable results to MVCNN.

Other approaches that make use of rendered 2D views include the Relation Network [78], RotationNet [27], MLVCNN (Multi-Loop-View CNN) [26], view-GCN (View-based Graph CNN) [73], DRCNN (Dynamic Routing CNN) [71], GVCNN (Group-View CNN) [17], joint CNN and LSTM [47], DeepPano (Deep Panoramic Representation) [63], and Geometry Images [65].

Besides the set of views themselves, other factors might have to be considered when performing multi-view object recognition. For example, the trajectory of the camera can be optimized to respect real-world constraints of the robotic arm, such as the range of motion of the joints, or the amount of time required to perform the task. To this end, Sock et al. [66] propose a method that uses deep

reinforcement learning to learn an optimal trajectory. An agent is trained in simulated scenarios, such that it learns which camera moves will result in the most accurate 6D object pose hypotheses. This method successfully obtains a set of near-optimal viewpoints, while respecting the other real-world constraints.

Korbach et al. [36] is another example of a deep reinforcement learning approach. They use the Soft Actor-Critic (SAC) algorithm to learn a policy for selecting the next best view. An actor learns to select actions that maximize the expected reward, while a critic learns to estimate the value function; the soft version adds an entropy term to the reward, which encourages exploration. In their work, SAC is used to learn the poses that maximize the classification confidence difference and sample efficiency, while minimizing time expenditure. The algorithm is shown to be able to classify objects with an accuracy of up to 96.33% on the TEOS dataset [67].

A comparable method is to train an end-to-end network to learn multi-view descriptors from rendered point clouds. Li et al. [42]'s "hard-forward soft-backward" technique addresses the issue of differentiability by employing the differentiable renderer SoftRas [45] in the backward pass, fusing per-triangle contributions in a "soft" probabilistic manner. This renderer aids in the derivation of 3D information from 2D pictures by eliminating the rasterization, which makes differentiation difficult because of discretization. For the case treated, SoftRas was adapted to operate with point cloud data.

However, the adaptation to point clouds can be avoided by recreating a polygonal surface from the point cloud. Methods for this include PolyFit [51] or the Open3D Library's Surface Reconstruction methods: alpha shapes [15], ball pivoting [3], and Poisson surface reconstruction [33] [82].

Often times, Multi-view Object Recognition methods simultaneously tackle other tasks such as pose estimation and grasp detection, as these also benefit from multi-view information. Parisotto et al. [52], for example, do so, and address the limitations of pre-defined virtual camera poses in MVCNN methods by developing a deep object-agnostic entropy estimation model that can predict the best viewpoints of a given 3D object. The obtained views of the object are then fed to the network, which then branches to simultaneously predict the pose and category label of the target object. This approach allows for more flexibility in viewpoint selection and improves the accuracy of both object recognition and pose estimation.

Similarly, Kasaei et al. [29] perform object recognition and grasp detection simultaneously. An encoder-decoder architecture is used, with an augmented memory capacity, that can estimate a pixel-wise grasp configuration. The proposed method can learn in an open-ended manner, and can learn new object categories with very few examples, and grasp never-seen-before objects, both in simulation and in the real world.

Other grasp synthesis methods have been proposed, such as the one by Li et al. [44], who use Deep Residual U-Nets to synthesize a grasping approach. A more recent method is MVGrasp [28], which uses a mixed autoencoder for multi-view object grasping. Multiple views of a given scene are generated; a view selection module then selects the best view for grasping and feeds it into the grasp network, which generates a pixel-wise synthesis (predicting quality, angle, and width of the grasp). Then, the grasp configurations are ranked by quality and transformed from 2D to 3D (from the orthographic view to the object's reference frame) and finally executed.

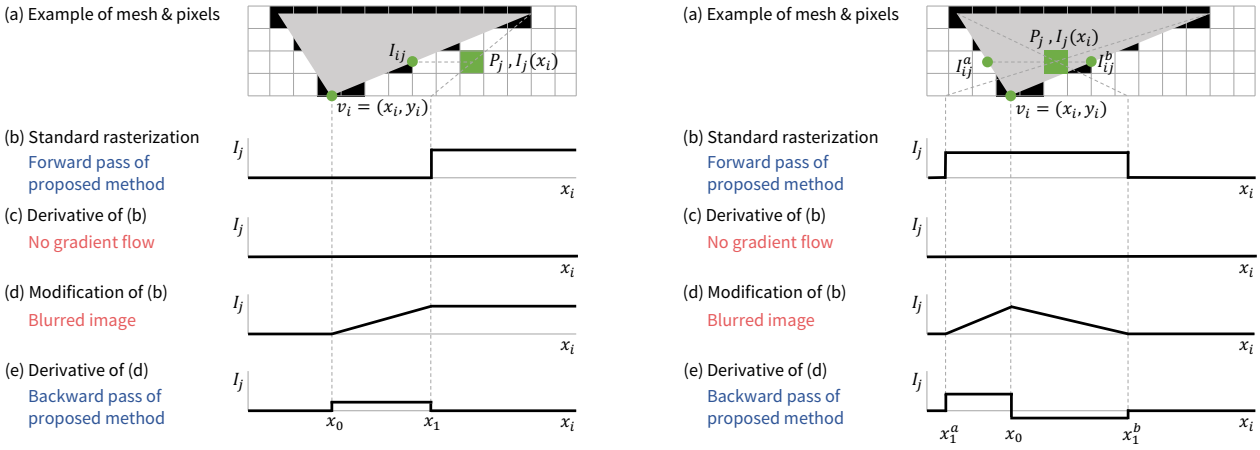


Figure 2: Illustration of the method used by the Neural 3D Mesh Renderer for computing the gradient flow of a mesh [32]. Left: $v_i = \{x_i, y_i\}$ is the i-th vertex of the mesh, and P_j is the j-th pixel with color I_j. The initial location of the vertex is x_0, and x_1 is the location of the vertex such that the edge of the face is in the middle of the pixel. Thus I_j becomes $I_i j$ when x_i is moved to x_1. Right: a similar case is treated when P_j is inside the face, and there are two directions that x_i can be moved in, either towards x_1^a or x_1^b, thus I_j becomes $I_i j^a$ or $I_i j^b$ respectively.

2.2 Neural 3D Mesh Renderer

For the purpose of optimizing camera positions continuously, we need to render 3D meshes in a differentiable manner. There is a great variety of methods to differentiate rendering algorithms, as shown by Zeltner et al. [79].

One of the earliest solutions in differentiable rendering was the Neural 3D Mesh Renderer [32]. Whereas traditional rendering methods use the rasterization technique (which is a discrete operation and therefore prevents backpropagation), the Neural 3D Mesh Renderer uses an approximate gradient for rasterization, which enables the integration of rendering into neural networks. Figure 2 illustrates the method used for computing the gradient flow of a mesh.

The Neural 3D Mesh Renderer offers, for example, the possibility of performing single-image 3D mesh reconstruction with silhouette image supervision. Other applications include 2D-to-3D style transfer or 3D DeepDream, which they both perform with 2D supervision for the first time [32]. For the purpose of next-best-view selection, what is of interest is the possibility of finding the set of camera parameters which minimize a given loss function. This is done by optimizing the camera position and orientation, which is a continuous operation. One example provided by the authors is defining the loss function as the absolute difference between the rendered image and a reference image. This leads to the camera position converging into the value which recreates the given reference image as closely as possible.

2.3 Differentiable Direct Volume Rendering (DiffDVR)

Differentiable Direct Volume Rendering (DiffDVR) is a rendering solution for 3D volumes, that provides differentiability for all continuous parameters of the volume rendering process [74]. By providing it with a problem-specific objective function, DiffDVR can be used to steer the parameters towards an optimal solution. Its ability to be independent of the number of sampling steps through the volume, and to consider small-scale changes, is achieved by enforcing a constant memory footprint, making it especially suitable for volume rendering. Both the external parameters of the rendering process and the volumetric density field can be optimized automatically using DiffDVR.

Two examples of its use are automatic viewpoint selection and reconstruction of a 3D density field from images. The first example uses differentiable entropy as the objective function, while the second example uses an absorption model to optimize per-voxel densities. Of the two examples, the first one is the most relevant to this thesis, as it can be used to automatically select the best viewpoint for the object recognition task. Figure 3 shows an example of the automatic viewpoint selection using DiffDVR, on a volume of a tooth.

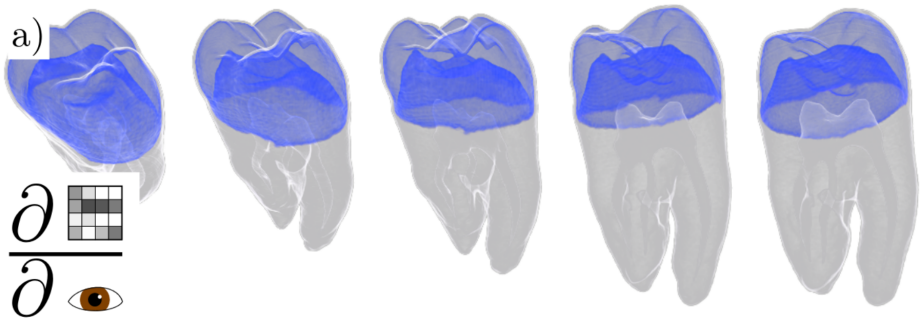


Figure 3: Viewpoint optimization for DiffDVR [74]. In the example given, a scan of a tooth is rotated in order to maximize the entropy of the rendered image of the tooth.

2.4 PointNet and PointNet++

Several methods have been proposed for embedding point clouds into a feature space [41, 77, 46, 70, 35]. Of these, PointNet and PointNet++ are the most relevant to this thesis.

PointNet is a neural network that directly consumes point clouds, and is well suited for applications such as object classification, part segmentation, and scene semantic parsing [8]. It has several advantages over traditional convolutional architectures for 3D data. By directly consuming point clouds, an important type of geometric data structure, it avoids unnecessary voluminous data and quantization artifacts that can obscure the underlying structure of the data. At the time of its publication, PointNet was the first neural network that directly consumed point clouds, and was able to achieve state-of-the-art performance on several tasks such as object classification, part segmentation, and scene semantic parsing [8].

Concretely, it represents a point cloud as a set of 3D points, where each point is its (x,y,z) coordinates, along with other feature channels (such as color or normals). It uses a symmetric function to aggregate information across all points in the set, thus respecting the permutation invariance of the points. This allows it to operate on unordered point clouds, without the need for a canonical ordering. The output can be either a single global feature vector (for tasks such as classification), or a feature vector for each point (for tasks such as segmentation).

PointNet++ is an extension of the original PointNet architecture that uses a hierarchical grouping strategy to capture local geometric structures [54]. This is to address PointNet's limitation in recognizing fine-grained patterns. It works by partitioning the input point cloud into nested subsets, and then applying PointNet recursively on each subset [54]. The network is able to learn local geometric features with increasing contextual scales, and is able to learn deep point set features efficiently and robustly. As of the time of its publication, PointNet++ was the state-of-the-art on several benchmarks of 3D point clouds. A diagram of the PointNet++ architecture is shown in Figure 4.

PointNet++ proposes two grouping strategies: Single-Scale Grouping (SSG) and Multi-Scale Grouping (MSG). Muti-Scale Grouping (MSG) deals with the issue of variations in point density in different areas. It does this by adaptively combining features from multiple scales, which allows it to handle point sets with varying densities. However, it is more computationally expensive than Single-Scale Grouping (SSG), with a double increase in execution time, due to the multi-scale region feature extraction.

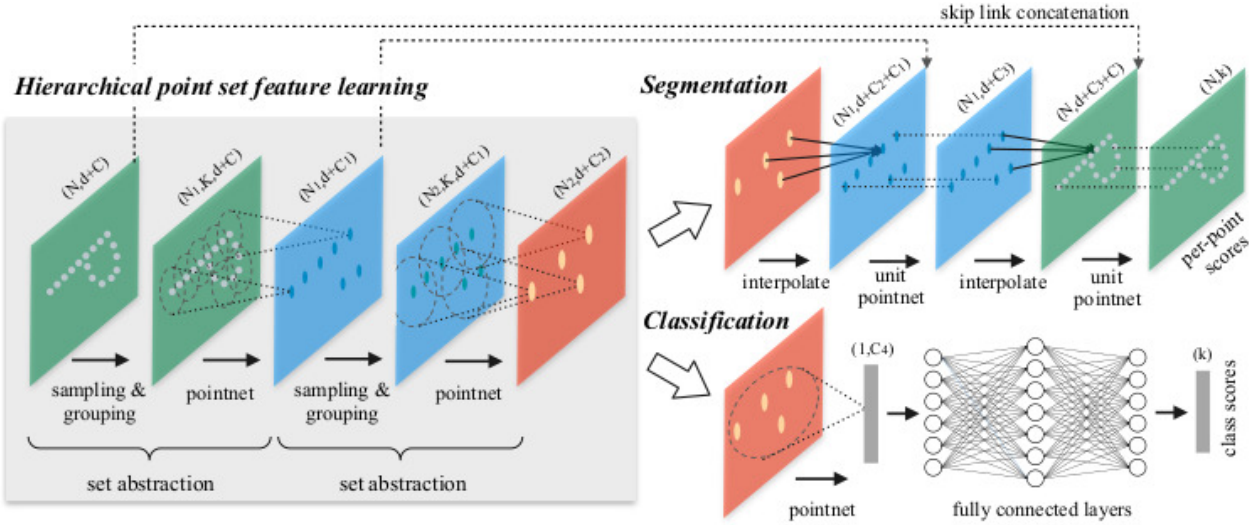


Figure 4: Diagram of the PointNet++ architecture [54].

3 Research Questions

To summarize, this thesis focuses on the following problems:

Q1. Firstly, whether maximum entropy viewpoint selection (MEVS) is a viable approach for **viewpoint selection for object recognition**.

Q2. Secondly, whether differentiable rendering can be used for MEVS.

Q3. Thirdly, whether a neural network using point clouds as input can be used for MEVS.

Q4. Furthermore, discovering which of the approaches is best for the estimation of **the optimal set of views for the representation of a given object**.

4 Methods

In this project, several methods were used to find the maximum entropy view for the camera. We describe them in Section 4.2 These were then combined with the backbone networks (described in Section 4.3) to form the complete pipeline (described in Section 4.4). We start with the dataset used, in Section 4.1.

4.1 Dataset

For a real-world use case, the data obtained would be from an RGB-D camera. It would be a point cloud, and thus any dataset used for training or evaluating our methods would have to be converted to point clouds. Several datasets were considered (such as ShapeNet [7], RGB-D[38], and MIRO [27]), but eventually the ModelNet10 dataset [76] was used, which is a synthetic dataset consisting of 10 classes of objects, in mesh format. The classes of the ModelNet10 dataset are: bathtub, bed, chair, desk, dresser, monitor, night stand, sofa, table and toilet. The dataset contains 4899 meshes, divided into 3991 training meshes and 908 test meshes. Classes are imbalanced, with chairs being the most common class, and bathtubs being the least common (see Figure 5).

We generate the point cloud data by sampling points from the meshes in ModelNet10, based on a set of partial views centred on the object. For this, we base our work on the work of Parisotto et al. [52], which provides a set of scripts to generate the views. The views were generated by rotating the camera around the object along the latitude and longitude, and rendering the object at each rotation. To collect the views, one method that can be used is to rotate in fixed increments along each of these, and render the object at each rotation. One issue with this is that the number of views sampled at the top and bottom of the object would be much larger than the number of views sampled at the sides of the object.

Therefore, to obtain uniformly distributed viewpoints, we first used a set of coordinates representing the vertices of Platonic solids (the tetrahedron, cube, octahedron, dodecahedron and icosahedron). These viewpoints were then converted to spherical coordinates and the θ and ϕ values were used to generate the views. The problem this brought was that it would not allow increasing the number of viewpoints arbitrarily. To solve this, a Fibonacci sphere was used to generate the viewpoints. This is an algorithm that allows the even distribution of an arbitrary number of points on a sphere. See Figure 6 for an example of the points generated by running the Fibonacci sphere algorithm for 10 and 40 iterations respectively. The formula used to generate a Fibonacci sphere with N points is given by:

$$
\begin{gathered}
t_i = (\frac{i}{N}, \frac{i}{\theta}) \text{ for } 0 \leq i \leq N \\
\text{where} \\
\theta = \frac{1+\sqrt{5}}{2} = lim_{n\to\infty}\frac{F_n}{F_{n-1}} \\
(x,y) \to (\theta,\phi) : (cos^{-}1(2x-1) - \pi/2, 2\pi y) \\
(\theta,\phi) \to (x,y,z) : (cos(\theta)cos(\phi), cos(\theta)sin(\phi), sin(\theta))
\end{gathered}
\tag{8}
$$

We retain the order the points are generated in, as a canonical ordering to be used later (see Figure 7a).

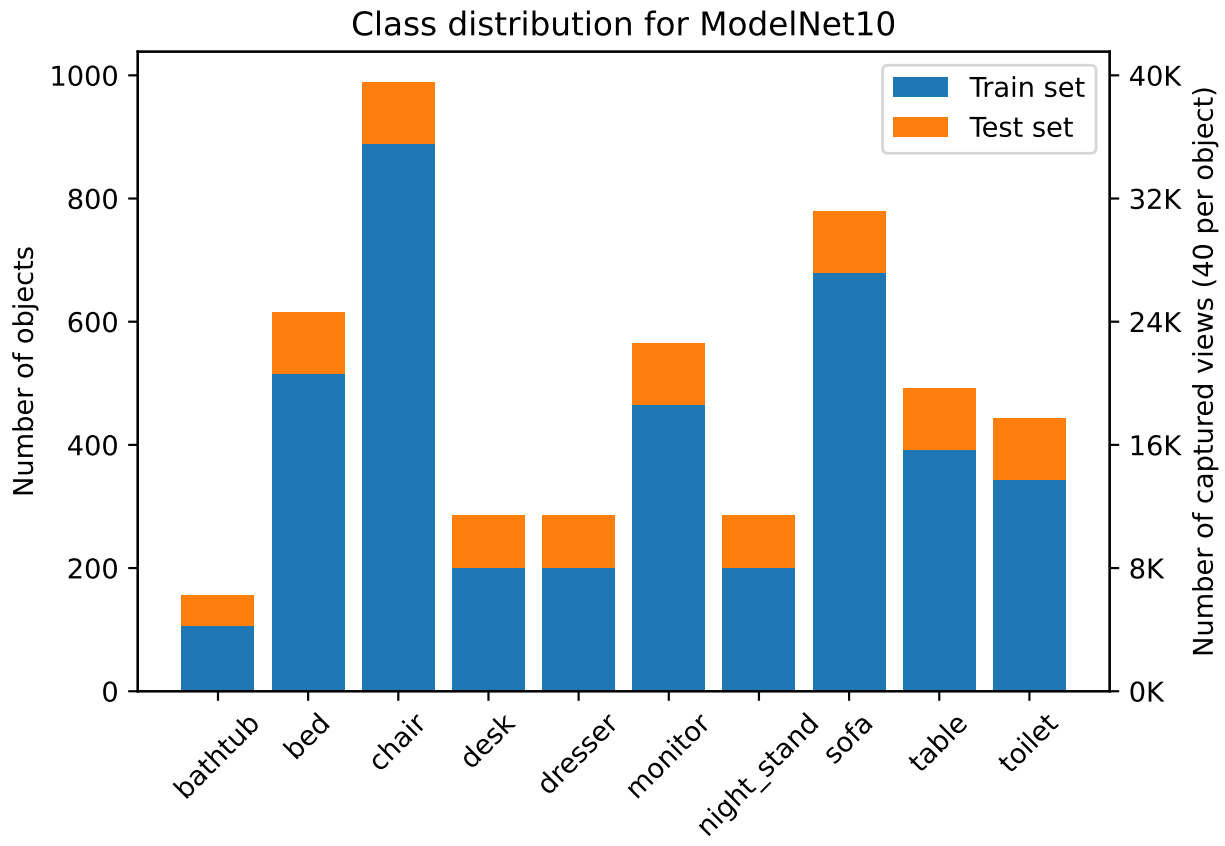


Figure 5: Class distribution of the ModelNet10 dataset, split into training and test sets. On the x-axis are the classes, and on the left y-axis are the number of object meshes in each class. We also include the number of views captured (for the 40-view dataset) on the right y-axis. For the 10-view dataset, the number of views is 10 times the number of meshes.

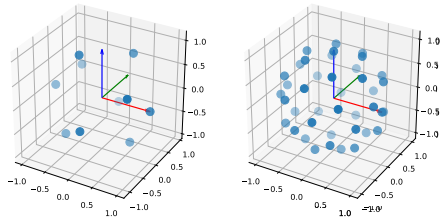
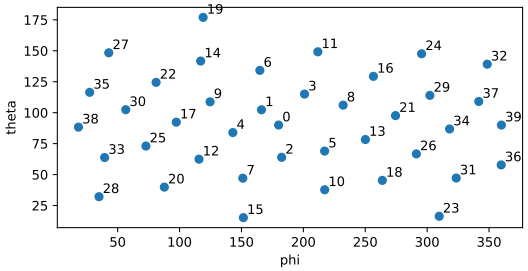
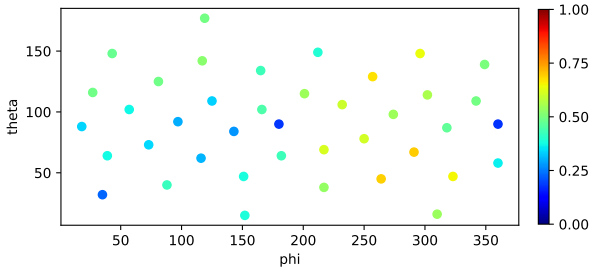


Figure 6: (left) Distribution of 10 points generated using the Fibonacci sphere, in 3-dimensional space. (right) Distribution of 40 points generated using the Fibonacci sphere. Note that the points in the left image are not a subset of the points in the right image.



(a) Distribution of 40 points generated using the Fibonacci sphere, projected in 2D using the Mercator projection. It can be seen that the points are not evenly distributed when projected (there are fewer points at the top and bottom), as the Mercator projection distorts the shape of the sphere. This is not a problem for the sampling, as on the sphere the points are evenly distributed. We also include the canonical ordering of the points, which is the order in which they are generated.



(b) Depth entropies associated with 40 viewpoints generated using the Fibonacci sphere, projected in 2D using the Mercator projection. The object concerned is a bathtub. The depth entropy is normalized to the interval [0, 1]. It can be seen that the depth entropy has local minima and maxima. In the Appendix (Figure 22), we include example captures at the points with the lowest and highest depth entropies.

Figure 7: The Fibonacci sphere algorithm generates points that are evenly distributed on a sphere.

We capture two datasets, one with 10 views per object, and one with 40 views per object. Thus the dataset sizes are 48990 and 195960 respectively. We capture the partial point clouds from each viewpoint using the Open3D package, which provides a method `capture_depth_point_cloud` as part of its `Visualizer` class. The viewport width and height were both set to 224 pixels, leading to a maximum number of points of 50176. As the object only occupied part of the viewport, the number of points captured was always lower (as shown in Figure 8). The number of points was relevant, as we set the input size of the point cloud embedding network based on it, to ensure as many views as possible could be processed.

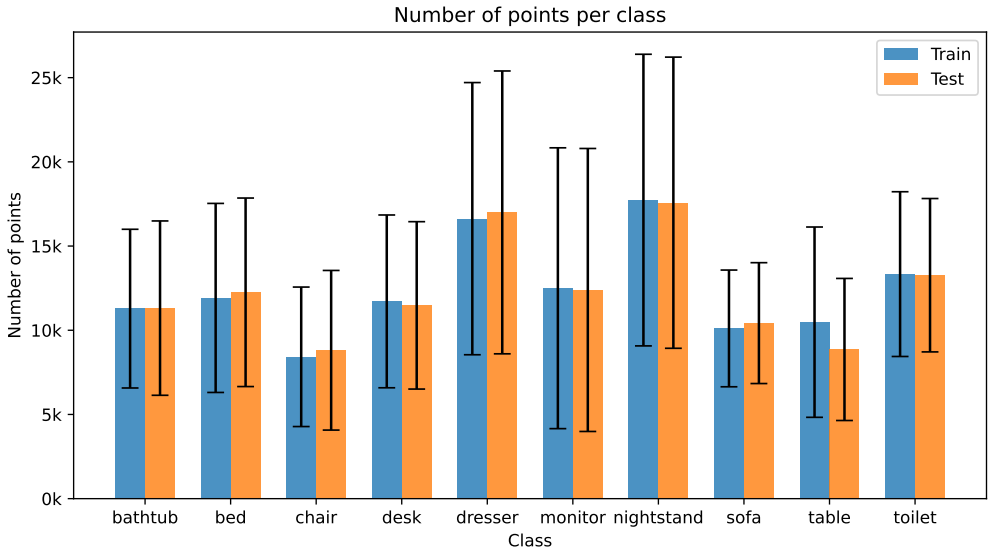


Figure 8: Number of points per view in the partial point clouds for each class. For the entire dataset, the mean number of points per view was 12337.24, and the standard deviation was 5829.

As detailed in Section 1.2.6, the synthetic point cloud dataset was augmented. The augmentations used were:

- scaling the point cloud (with a factor sampled uniformly from the interval [0.8, 1.25])
- rotating the point cloud around the z-axis (with an angle sampled uniformly from the interval [0, 2π])
- rotating the point cloud around all axes (with angles sampled uniformly from the interval [0, 0.06])
- translating the point cloud (with a factor sampled uniformly from the interval [-0.1, 0.1])
- adding jitter noise to the point cloud (with a standard deviation of 0.01, and clipping the values to be between -0.05 and 0.05),
- randomly removing points from the point cloud (for each point cloud, a random probability between 0 and 0.875 was chosen, and points were removed with that probability by setting them to the first point in the point cloud)

These augmentations were only applied when training, and not to the test set.

A depth image was also captured, with the `capture_depth_image` method. This was the equivalent of converting the coordinates from world space to screen space. This depth image was then used to compute the depth entropy of each view. The formula used for this was the Shannon entropy, which is given by Equation 9 [62, 20].

$$H = -\mathbb{E}[\log p(X)] = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) \tag{9}$$

Where p(x) is the frequency of the pixel value x in the image, and $\mathcal{X}$ is the set of all possible pixel values.

These values were then normalized to be between 0 and 1, by dividing by the maximum sampled depth entropy when training (which was 5.46). Figure 9 shows the distribution of all the computed depth entropies, after normalization. The mean depth entropy was 0.329, and the standard deviation was 0.156. The distribution has a fat tail, which means that there are a few views with very high depth entropy values.

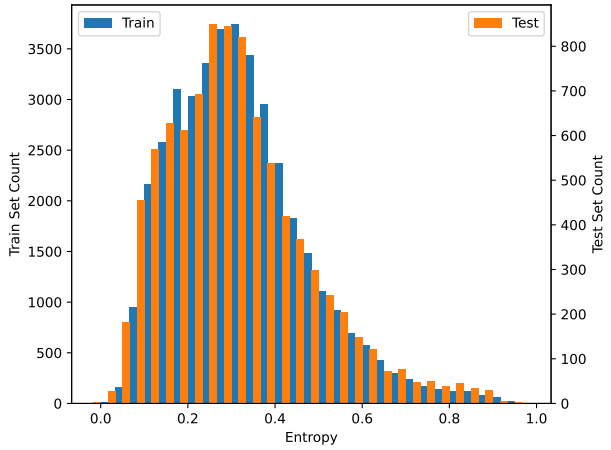
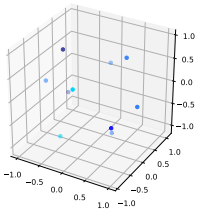


Figure 9: Distribution of the depth entropies for all the objects, by train/test split. The depth entropy is normalized to the interval [0, 1]. The horizontal axis represents the depth entropy, and the vertical axis represents the number of views with that depth entropy. To make it easier to compare the y-axis, two scales are used: the left y-axis is for the train set, and the right y-axis is for the test set.

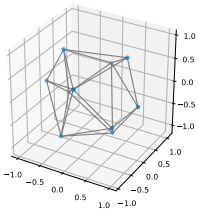
Finally, for classifying the objects, an RGB image was captured using the `capture_screen_image` method. This is necessary for the classification with the ResNet model, as it is a CNN that takes RGB images as input. Later on, the depth image was also used as input for the ResNet model, to see if it would improve the results.

When switching from the discrete latitudes and longitudes to the Fibonacci sphere, one issue was the loss of the neighborhood relation between the viewpoints. Indeed, to get the neighbors of a point on a grid, we can simply look at the points with indices (i-1, j), (i+1, j), (i, j-1) and (i, j+1), where i and j are the indices of the point. The neighbors of a point on the Fibonacci sphere are not as easy to find, as they are not indexed; a graph structure can be used. Thus to obtain this neighborhood relation, we used the Delaunay triangulation [11]. The triangulation only had to be computed once, as the viewpoints were the same for all the objects. For this, the Python library Stripy [50] was used, which provides an interface to the Fortran library `STRIPACK` [56]. This resulted in a node-weighted graph, where the nodes are the viewpoints, and the node weights

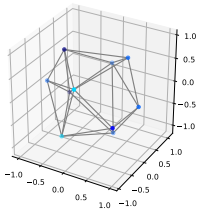
are the depth entropies of the viewpoints (see Figures 10 and 11). Note that all edges are of equal weight, as they simply represent that two viewpoints are neighbors. This graph could then be parsed to find the local maxima of the depth entropy, which correspond to the viewpoints with the highest depth entropy in their neighborhood. Multiple algorithms were benchmarked for this, and in the end, the simplest solution was chosen as it was fastest: parsing the node list and adding all nodes with higher depth entropy than their neighbors to a result list (see Appendix A for the algorithm).



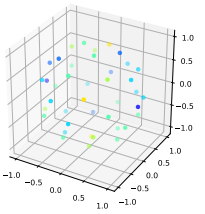
(a) 10 views, colored by depth-entropy.



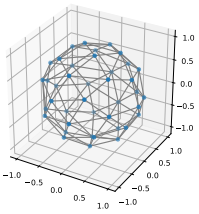
(b) 10 views, connected in a view graph obtained with the Delaunay triangulation.



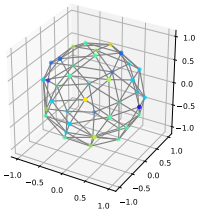
(c) 10 views, colored by depth-entropy; connected in a view graph obtained with the Delaunay triangulation.



(d) 40 views, colored by depth-entropy.

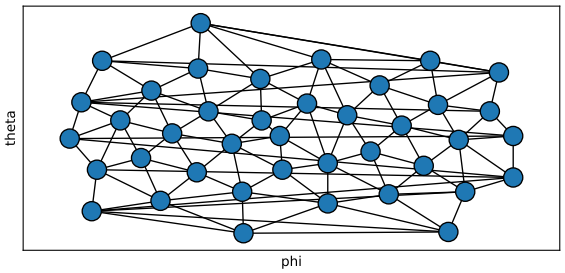


(e) 40 views, connected in a view graph obtained with the Delaunay triangulation.

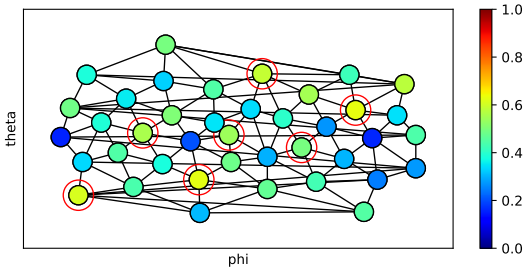


(f) 40 views, colored by depth-entropy; connected in a view graph obtained with the Delaunay triangulation.

Figure 10: Distribution of points generated using the Fibonacci sphere, in 3-dimensional space. The top row shows the distribution for 10 points, and the bottom row for 40 points. The left column shows the points with depth-entropy assigned, the middle column shows the points connected in a view graph obtained with the Delaunay triangulation, and the right column shows both, which is the data structure we use to find the local maxima of the depth-entropy. The same color scale is used as for Figure 7b. For easier visualization, animations of these structures are also provided in the project repository, under directory `assets/animations/`.



(a) Graph representing the neighborhood relation between the viewpoints. The nodes are the viewpoints, and the edges represent that two viewpoints are neighbors. This configuration is reused for all the objects.



(b) The depth entropies of the viewpoints are computed using the depth images. We then populate the graph node weights with these depth entropies. Then we can use the algorithm described in Appendix A to find the local maxima of the depth entropy (shown circled in red).

Figure 11: We triangulate the viewpoints on the Fibonacci sphere to obtain a graph, which we can then use to find the local maxima of the depth entropy. Note that the edges wrap around the sphere, so that the viewpoints on the leftmost part of the projection are neighbors with the viewpoints on the rightmost part.

4.2 Next Best View Selection

4.2.1 Model based on differentiable rendering

The initial approach used for the project was to take advantage of differentiable rendering. The idea was to use a differentiable renderer to render the object from an initial viewpoint, to obtain a partial view of the object. Then, we could use the information of this partial view to optimize the next viewpoint's parameters, such that the expected (rendered) image is as good as possible. First, it was necessary to define a monotonically decreasing function of some feature of the views which must be maximized. Then, by freezing the parameters of the rendering pipeline, except for the camera position (the camera's extrinsic parameters, and the object's geometry), we can compute the gradient of the feature with respect to the camera parameters.

This was done using two renderers:

- Neural 3D Mesh Renderer [32], which is a differentiable renderer that can be used to render 3D meshes, with a given set of camera parameters. The feature maximized was the amount of the object which was visible in a circle in the center of the rendered image. Several other features were considered, such as the amount of edges in the rendered image, the entropy of the rendered image, or the entropy of the depth. The Neural 3D Mesh Renderer was then used to render the object from an initial view, and the loss was computed. The loss was then used in a stepwise fashion to compute the gradient of the feature with respect to the camera parameters, and the gradient was used to update the camera parameters. This was repeated until the loss converged to a local minimum.

- DiffDVR [74], which instead handles 3D volumes (i.e. voxels) with a given set of camera parameters.

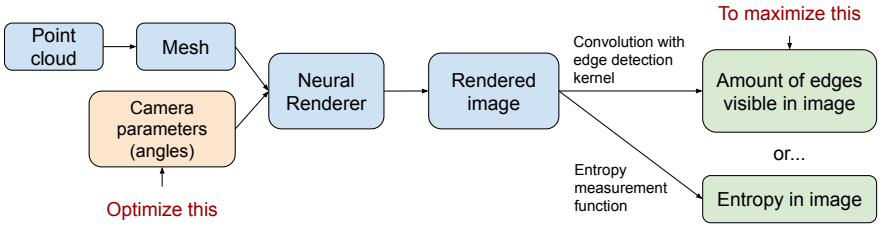
For DiffDVR, the mesh data from ModelNet10 was converted to a binary occupancy grid using the Open3D library's voxelization method [82]. Here, a script was used that was initially developed by Parisotto et al. [52].

As neither of these renderers were meant for point clouds, for a real pipeline, a step would have to be converting the point cloud to an appropriate representation. Thus:

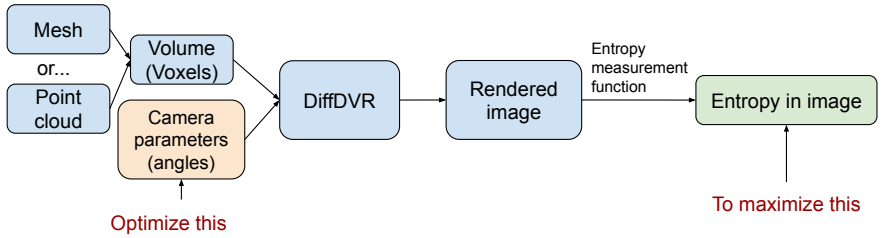
- for the Neural 3D Mesh Renderer, the point cloud can be converted to a mesh using the Open3D library's Poisson Surface Reconstruction method [33, 82]. It might seem counter-intuitive to capture a point cloud based on a mesh, and then convert it back to a mesh, but we want to simulate a real-world scenario, where we do not have access to the full mesh, but only to a partial RGB-D view.

- for DiffDVR, again we can use the Open3D library's voxelization method [82], but this time with a point cloud as input.

The diagrams describing these approaches are shown in Figure 12. Both of these methods were found to be computationally intensive, as the differentiable renderers require a training run for each view. This means that, while the differentiability of the renderers allow for their use in training neural networks, it is unfortunately not suitable for use in a real-time application.



(a)



(b)

Figure 12: Pipelines describing the models based on differentiable rendering. (a) Pipeline for the Neural 3D Mesh Renderer. (b) Pipeline describing the model based on DiffDVR. Note that it can only accept 3D volumes as input, so for point cloud inputs, we will need a conversion step.

4.2.2 Model based on point cloud embedding

The second approach used for the project was to use a point cloud embedding network to embed the point cloud into a latent space, then use a fully connected network to predict the depth entropy of each viewpoint. This can be seen as a multi-output regression problem, where the input is the point cloud, and the output is the depth entropy of each viewpoint. Then, the viewpoint with the highest depth entropy would be selected as the optimal viewpoint. This was done using the PointNet++ [54] network, with single-scale grouping. A diagram describing the preparation of the approach can be seen in Figure 13.

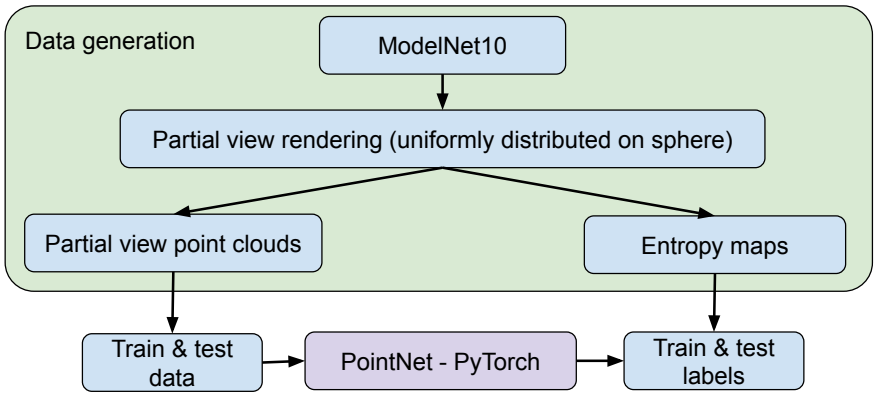


Figure 13: Pipeline describing how data is obtained for the model based on point cloud embedding. To begin, the dataset is generated based on ModelNet10 [76]. On the one hand, the partial view point clouds are collected. On the other hand, the depth entropy of the capture from each view is also collected, in an entropy map. The partial views are treated as the input to the network, and the depth entropy map is treated as the ground truth. The network is trained to predict the depth entropy map from the partial views.

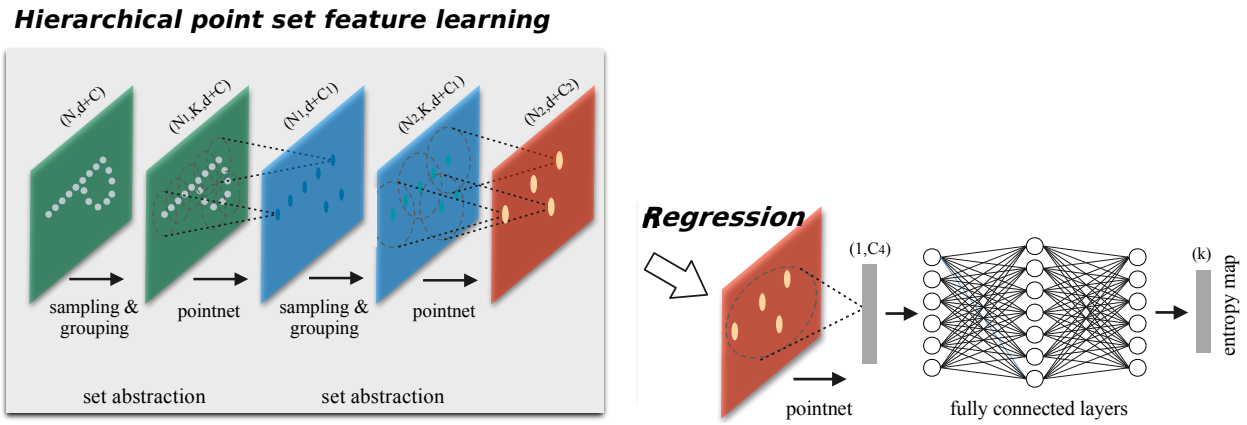
The advantage of this approach is that it is much faster than the first approach, as it does not require a backpropagation pass through the renderer for each viewpoint. This allows it to be used in real-time applications, such as a robotic arm. However, it does not regress the viewpoint coordinates themselves, and thus has a discrete, fixed set of 40 possible viewpoints, which may not be desired in some applications.

The network architecture is as follows:

- The input is a point cloud with 4096 points.
- A set abstraction module (SAModule) is used to downsample the point cloud. This is done by grouping points together, and then applying a symmetrical function to each group. This is done three times, with the number of points in each group decreasing each time. The first SAModule uses a radius of 0.2, the second uses a radius of 0.4, and the third uses a radius of 1. The number of points in each group is 512, 128, and 1024 respectively.

- The output of the last SAModule is flattened, and then passed through a fully connected network with 3 hidden layers. Batch normalization is used after each hidden layer, and dropout is used before the last hidden layer with a dropout rate of 0.5. The output of this network is the depth entropy of each viewpoint, in the canonical order shown in Figure 7a.

The network architecture can be seen in Figure 14.



(a) Feature learning

(b) Regression

Figure 14: The network architecture of the PointNet++ network used for feature learning and multi-output regression. This figure is largely based on the one found in the original PointNet++ paper [54], with the modification of the output of the network being the depth entropy of each viewpoint.

The total number of parameters in the network is 1,471,509 for 10 views, and 1,479,219 for 40 views. Section 5 shows that this network is able to learn a good representation of the point cloud, and is able to predict the depth entropy of each viewpoint with high accuracy.

4.3 Backbone for classification

For the classification of the images rendered from the selected viewpoints, two ResNet [22] architectures were tested as backbones, with different types of input:

- ResNet-18 [22], which is a 18-layer ResNet architecture. A network pretrained on ImageNet [13] is fine-tuned on a dataset of images/depth captured from the viewpoints. The total number of parameters in the network is 11,191,262.
- ResNet-34 [22], which is a 34-layer ResNet architecture. Again, this network is pretrained on ImageNet, but only fine-tuned on the depth. The total number of parameters in the network is 21,306,862.

The architecture of residual networks is further described in Section 1.2.3 and Figure 15. In both cases, the input of the network is a 224x224 image/depth, and the output is a C-dimensional vector,

where C is the number of classes (10 in the case of ModelNet10). A softmax layer is applied to the output of the network to obtain a probability distribution over the classes.

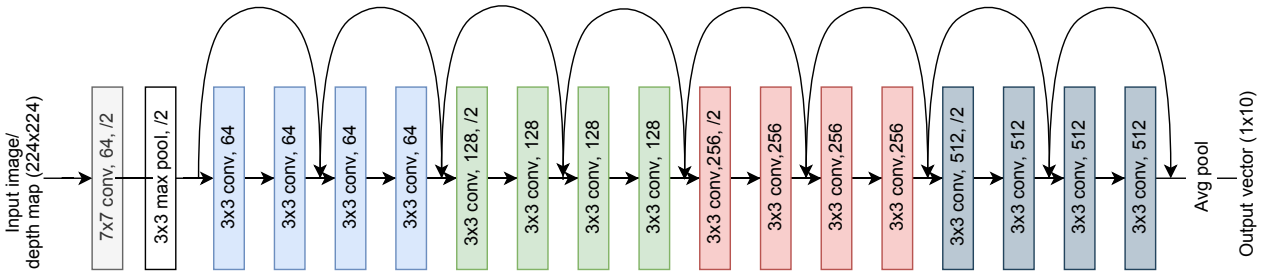


Figure 15: The network architecture of the ResNet-18 network used as a backbone for classification. Architecture changes are minor compared to the one found in the original ResNet paper [22]; the output is changed to a 10-dimensional vector.

4.4 Pipeline

The pipeline proposed in this thesis is shown in Figure 16. The pipeline consists of two main parts: viewpoint selection and classification. The viewpoint selection part is responsible for selecting the viewpoints from which to capture the images. A configuration file is used to specify the set of possible viewpoints, in a canonical order (as described in 4.1), and their adjacency. The pipeline starts with a random viewpoint. The ResNet backbone network is used to classify the captured image/depth from that viewpoint, and the output (before softmax) is stored in an accumulator. A softmax is applied to this output, and if its maximum value (confidence) is greater than a threshold, we classify the object as the class with the highest probability. Otherwise, we proceed to the viewpoint selection part. The viewpoint selection part then selects the next viewpoint to capture from the set of possible viewpoints. Now, we have a new image, and we repeat the process. This process goes on until the confidence score (maximum value in the accumulator) is greater than the threshold, or the maximum number of viewpoints has been reached. As methods based on differentiable rendering proved impractical, two methods for selecting the next best viewpoint are implemented:

- **ResNet + Random selection:** A new (unattempted) viewpoint is selected at random from the set of possible viewpoints. We treat the performance of this method as the baseline.
- **ResNet + PointNet++:** We retrieve the input for the PointNet++ network (a partial point cloud), to obtain the depth entropy of each viewpoint, in a canonical order. We then store the depth entropies in the nodes of a graph, and use our algorithm to find the local maxima of the depth entropy, using the method described in Figure 11 and Appendix A. We select the highest depth entropy unattempted local maximum; if all local maxima have been attempted, we select the highest depth entropy unattempted viewpoint.

As three different backbone ResNets are used, as well as two different PointNet++ architectures,

six different ResNet + PointNet++ pipeline configurations are possible. Results for each of these configurations, as well as for single-view classification, are presented in Section 5.3.3.

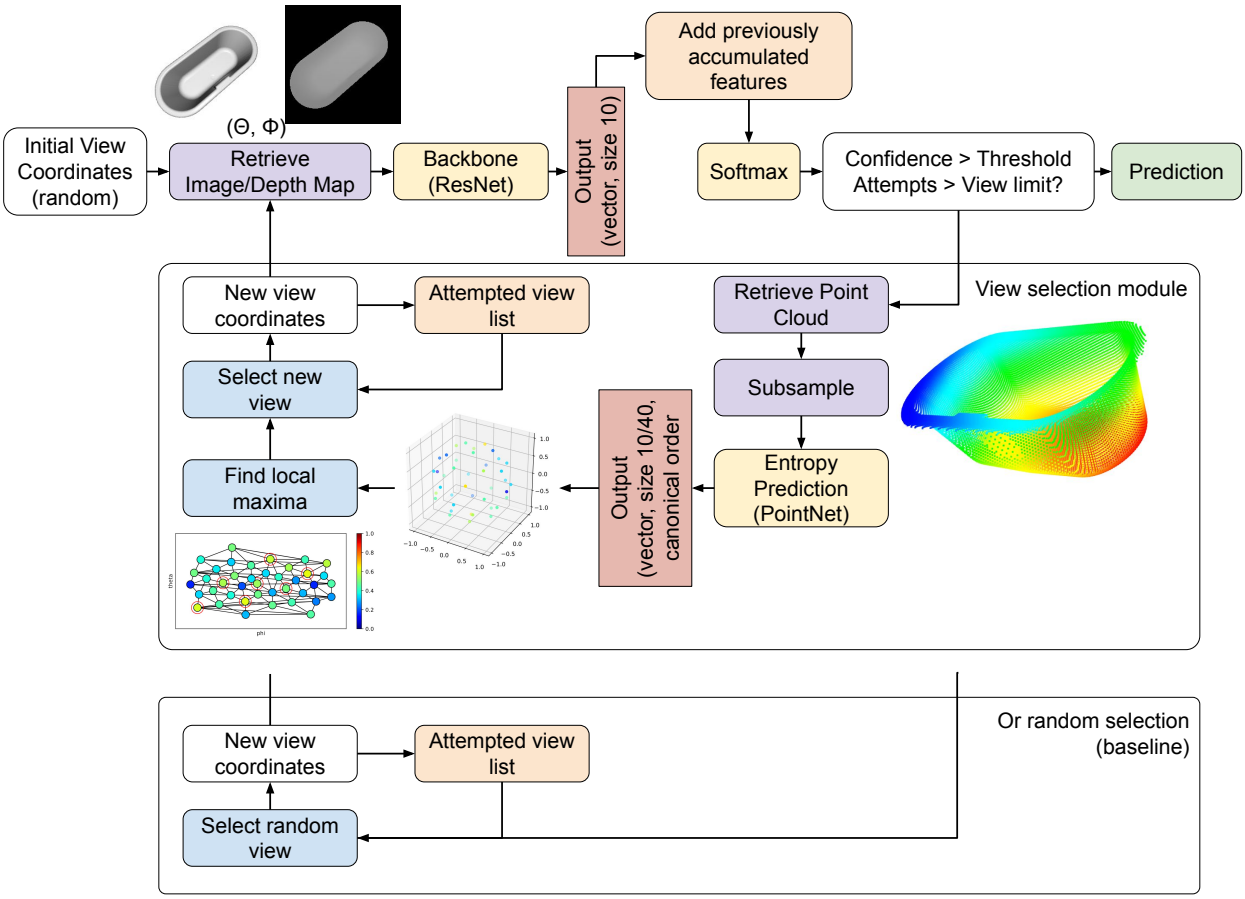


Figure 16: The pipeline used for viewpoint selection and classification. We describe both our proposed method and the baseline method. White: simple operations, such as generating a number or performing a calculation. Purple: loading data. Yellow: neural networks/differentiable operations. Blue: parsing operations. Red: output vectors. Orange: accumulators. Note that the point cloud on the right is colored by the distance from the capturing camera and then rotated, with blue points being closest and red being furthest.

5 Experiments and Results

5.1 Tools and Technologies

The approaches compared are built based on the official code repositories for the Neural Renderer, DiffDVR and PointNet++ respectively. The main project repository, including the pipeline used to run the experiments, is available at: `https://github.com/AndreiMiculita/nbv_mevs`. Some of the code was modified to allow for the experiments to be run, as follows:

- The Neural Renderer was modified to allow for the use of a custom loss function. The implementation of this method is available in the main project repository: `https://github.com/AndreiMiculita/nbv_mevs`.
- For DiffDVR the application was already implemented by the original authors in the form of a script for estimating the maximum entropy view of a volume, and thus only required minimal modification to allow for the input of a point cloud. A Docker container was used to run the application, which was built on top of a Docker image for DiffDVR.[2] This circumvented the need to install the required dependencies on the host machine. The implementation of this method is available in a fork of the original repository: `https://github.com/AndreiMiculita/DiffDVR`.
- The PointNet++ was modified to allow for multi-output regression, in which each output represented the depth entropy at a certain viewpoint of the object The modified code is available at `https://github.com/AndreiMiculita/Pointnet2_PyTorch`, on the branch `am/thesis`. This implementation is built using the PyTorch Lightning framework [3], which allows for easy training and evaluation of neural networks.

For the purpose of these experiments, we note there is a fundamental difference between using the Neural Renderer and DiffDVR on one hand, and PointNet++ on the other. Namely, in the case of the Neural Renderer and DiffDVR, as the optimization process involves adjusting camera parameters through gradient descent, the application requires training the networks for each new use. For PointNet++, view entropies are instead regressed by the network, so it was trained only once, and the results were obtained by running the forward pass on the generated dataset. Thus, for the purposes of our application, the training step of the differential renderers can be considered equivalent to the inference step for PointNet++.

Given the nature of the experiments, a consumer machine was sufficient for running them. The relevant specifications of the machine can be found in Table 1.

[2] The Docker image is available at `https://hub.docker.com/r/xetaiz/diffdvr`.

[3] `https://www.pytorchlightning.ai/`

Model	Lenovo Legion S7 15IMH5
Operating System	Linux Mint 20.3
CPU	Intel i7-10875H (16) @ 5.100GHz
GPU	NVIDIA GeForce RTX 2060 with Max-Q Design (1920 CUDA cores, 6144 MB dedicated memory, 192-bit memory interface)
Memory	31817MiB total
CUDA version	11.0
cuDNN version	8.1.1
Python version	3.7
PyTorch version	1.7.1

Table 1: Specifications of machine used for running experiments.

5.2 Performance Criteria

The performance of the approaches was evaluated using the following criteria:

- **Computation Time**: The time taken to estimate the optimal views.
- **Entropy Learning Accuracy**: It refers to how well the point cloud embedding network can learn the depth entropies associated with each object from the synthetic depth entropy dataset.
- **Instance Accuracy**: The formula for the instance accuracy is given by:

$$\text{Instance Accuracy} = \frac{\sum_{i=1}^{C} TP_i + TN_i}{\sum_{i=1}^{C} P_i + N_i} \tag{10}$$

  where C is the number of classes, TP_i is the number of true positives for class i, TN_i is the number of true negatives for class i, P_i is the number of positive samples for class i and N_i is the number of negative samples for class i. More intuitively, it represents the ratio of correctly classified objects to the total number of objects. **Unless specified otherwise, we always refer to this when we mention accuracy results.**

- **Class Accuracy**: The formula for the class accuracy is given by:

$$\text{Class Accuracy} = \frac{1}{C} \sum_{i=1}^{C} \frac{TP_i + TN_i}{P_i + N_i} \tag{11}$$

  where C is the number of classes, TP_i is the number of true positives for class i, TN_i is the number of true negatives for class i, P_i is the number of positive samples for class i and N_i is the number of negative samples for class i. More intuitively, it represents the average

instance accuracy across all classes. The advantage of this metric is that it takes into account the class imbalance in the dataset; still, in a practical setting, not all error types have equal costs (this is especially true for medical applications), so the class accuracy still does not provide a complete picture of the performance.

- **GPU Memory**: The GPU memory used by the approaches, which was measured using the *max_memory_allocated()* function in PyTorch.

5.3 Results

Below we describe the results obtained from the experiments, measured by the performance criteria described in Section 5.2.

5.3.1 Computation Time

Computation time was measured using the Python *time* library. The time required to estimate the maximum entropy view of an object was measured for each of the approaches. In terms of computation time, the methods based on differentiable rendering are orders of magnitude slower. For each class in the ModelNet10 dataset, Table 2 describes the mean and standard deviation of the time required using the Neural Renderer and DiffDVR.

The Neural Renderer is the slowest, taking 773.37 seconds on average to estimate the next best view of an object. DiffDVR is faster, taking 57.90 seconds on average to estimate the maximum entropy view. PointNet++ is the fastest, taking 3.08 milliseconds on average for the 10-view model, and 3.27 milliseconds on average for the 40-view model. Table 3 contains a per-class breakdown of the time required to find the maximum depth entropy view using PointNet++. These are computed, for each class, based on averaging 10 training runs of the Neural Renderer and DiffDVR, and 20 inference runs of PointNet++, each with randomly selected objects of that class.

For the training of the Neural Renderer, the camera parameters were optimized using the Adam optimizer with a learning rate of 0.01. Multiple custom loss functions were used, and training was stopped when the loss went below a certain threshold. For DiffDVR, Stochastic Gradient Descent (SGD) was used to optimize the camera parameters, with a learning rate of 5.0. For both the Neural Renderer and DiffDVR, the starting camera parameters were determined by randomly choosing a point with the Fibonacci sphere sampling method described in Section 4.1. The training process had a high degree of interpretability, allowing for example to visualize the rendered images at each step of the optimization process, as well as the loss at each step, which was useful for debugging. The optimization process for DiffDVR can be seen in file `assets/animations/camera_optimization_diffdvr.gif`.

The starting point can have a large impact on the time required when using differentiable rendering, as the optimization process can get stuck in a local minimum. Thus, the standard deviation of the time required to estimate the maximum entropy view is also much higher for the Neural Renderer and DiffDVR, compared to PointNet++, which is expected due to the stochastic nature of the optimization process. For PointNet++, we also find no significant difference in computation time between object classes, which is expected as the point cloud embedding network always takes the same input size. Nevertheless, after obtaining these results we decided to stop using the differentiable renderers for the rest of the experiments, as the Neural Renderer was too prone to getting stuck in local minima, and both were too slow to be practical.

Class	Mean (NR) (s)	Std (NR) (s)	Mean (DiffDVR) (s)	Std (DiffDVR) (s)
bathtub	1432.70	2089.01	51.57	7.80
bed	353.45	490.20	65.16	7.05
chair	1337.13	2156.19	50.16	5.95
desk	141.35	134.38	56.39	8.22
dresser	1046.44	1947.38	72.07	12.19
monitor	157.32	126.20	61.60	11.97
night_stand	282.79	394.93	73.70	8.86
sofa	1291.09	2576.58	49.03	5.76
table	687.84	1475.83	45.57	6.02
toilet	1003.60	1035.29	53.77	5.77
all	773.37	1529.62	57.90	12.46

Table 2: Mean and standard deviation of computation time required for differentiable renderer-based methods, by class. NR stands for Neural Renderer. The times are averaged over 10 runs, each with a randomly selected object of the class. For the Neural Renderer, computation times tended to be grouped in 2 clusters, one with times around 60 seconds, and another with times around 1200 seconds. The latter cluster was likely due to the optimization process getting stuck in a local minimum, and never reaching a satisfactory loss value.

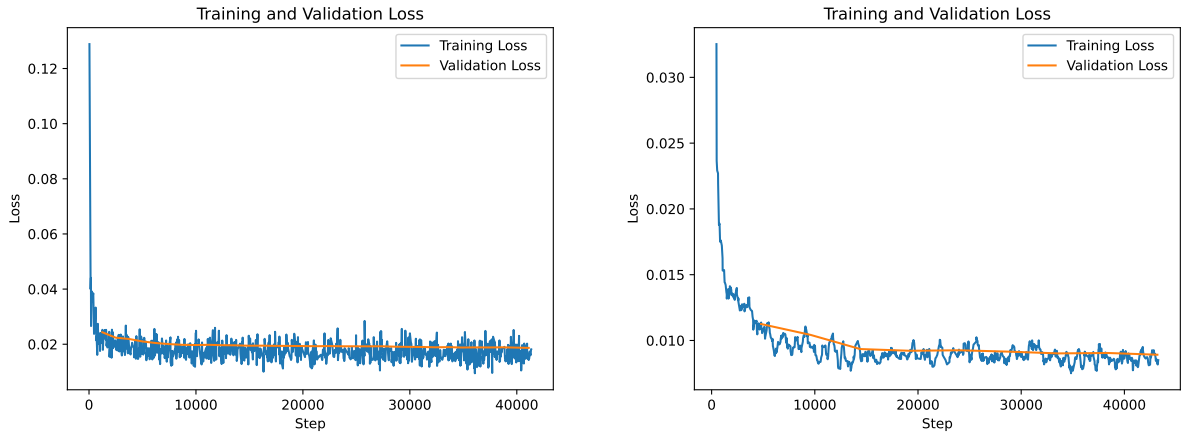
Class	Mean, 10-view (ms)	Std, 10-view (ms)	Mean, 40-view (ms)	Std, 40-view (ms)
bathtub	3.2206	0.9244	3.2671	0.3269
bed	3.0439	0.3676	3.3769	0.8497
chair	2.8651	0.3188	3.1444	0.3932
desk	3.3377	1.0106	3.3209	0.8186
dresser	3.1733	1.1299	3.2474	0.4565
monitor	3.0540	0.7938	3.4304	0.8939
night_stand	3.0631	0.5021	3.3742	0.6317
sofa	2.9605	0.4326	3.1232	0.3401
table	2.9721	0.2797	3.1579	0.4987
toilet	3.1604	0.8039	3.2225	0.4058
All	3.0851	0.7320	3.2665	0.6077

Table 3: Mean and standard deviation of computation time required for PointNet++, by class. This refers to inference time only. Both 10-view and 40-view models are evaluated. The times are averaged over 50 runs, each with a randomly selected object of the class.

5.3.2 Entropy learning

When training the PointNet++ model, the depth entropy learning progress is measured using the average of the *Mean Squared Error* (MSE) between the predicted depth entropies and the ground truth depth entropies equivalent to each point cloud.

It was trained for up to 200 epochs, with early stopping applied (with a patience of 5 epochs, based on validation loss) to prevent overfitting. It was trained using the Adam optimizer with a learning rate of 0.001, and a learning rate scheduler with a decay rate of 0.7 and a decay step of 20000. The batch size was set to 32, and the number of points was set to 4096. Before training, the samples were cached using the `lmdb` format, which allows for faster loading of the training data. The plot in Figure 17a shows the trend of the depth entropy learning loss of the point cloud embedding network during training, for each model. The results are shown in Table 4.



(a) 10-view model

(b) 40-view model

Figure 17: Trend of depth entropy learning loss (Mean Squared Error) during training, for a 10-view and a 40-view model (trained for 49 and 9 epochs respectively). The difference in epochs is because of the different number of training samples per epoch. The training and validation losses are shown.

For a 40-view model, the depth entropy learning loss is shown in Figure 17b and Table 4.

5.3.3 Classification accuracy

To build the pipeline, several backbone networks were trained on the ModelNet10 dataset. Once these were obtained, along with the point cloud embedding network, the pipeline was set up, as described in Section 4.4. The classification accuracy was then measured on the test set of the ModelNet10 dataset.

Single-view classification As described in Section 4.3, a ResNet-18 and ResNet-34 backbone were trained on the ModelNet10 dataset, to classify objects based on a single view. Transfer learn-

Class	Mean, 10-view	Std, 10-view	Mean, 40-view	Std, 40-view
bathtub	0.018	0.013	0.022	0.012
bed	0.008	0.008	0.009	0.006
chair	0.006	0.007	0.021	0.014
desk	0.009	0.006	0.036	0.016
dresser	0.026	0.025	0.028	0.026
monitor	0.009	0.010	0.018	0.013
nightstand	0.032	0.017	0.020	0.015
sofa	0.003	0.003	0.010	0.006
table	0.008	0.011	0.019	0.018
toilet	0.007	0.007	0.014	0.013
all	0.012	0.015	0.017	0.015

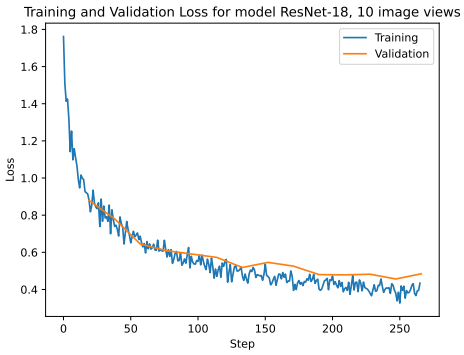
Table 4: Mean and standard deviation of the MSE (mean squared error) between the predicted depth entropies and the ground truth depth entropies, for each class, for a 10-view model and a 40-view model, measured on the validation set. We can see that for some classes, learning to predict depth entropies is more difficult than for others. When predicting 10 views, the sofa and the chair are easiest, while the nightstand and the dresser are the hardest. For 40 views, the bed and the sofa are easiest, while the desk and the dresser are hardest.

ing from pretrained models was used as much as possible, to reduce the training time. The ResNet-18 network was first initialized with weights pre-trained on ImageNet, provided by the PyTorch library. It was then trained on a small dataset we generate based on 10 views collected uniformly around each object. In all cases, for optimization, the Adam optimizer [34] was used, with a learning rate of 0.001 and weight decay of 0.0001. The loss function used in all cases was the cross-entropy loss, and the batch size was 16. A validation set was used to monitor the training progress, with a size of 20% of the training set. The maximum number of epochs allowed was 50, but the training stopped early when the validation loss did not decrease for 5 epochs, to avoid overfitting. As the purpose of this training was to obtain a backbone to enable comparison of view selection methods (not to obtain the best classification accuracy of the backbone itself), few hyperparameters were evaluated, and smaller models were favoured. The best training and validation losses obtained are shown in Table 5, and their trend over time is shown in Figure 18a.

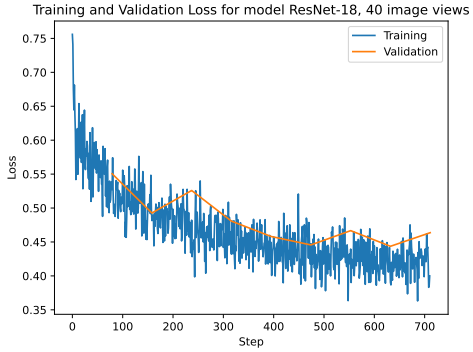
Once obtained, the first backbone was used in the pipeline for multi-view classification. Once the pipeline was set up, it was fine-tuned on a bigger (40-view) dataset, with the same hyperparameters as before. The trend of training and validation losses are shown in Figure 18b and Table 5.

Backbone	Training loss	Validation loss
ResNet-18 (10 views/object, image)	0.434	0.484
ResNet-18 (40 views/object, image)	0.400	0.463
ResNet-18 (40 views/object, depth)	0.385	0.418
ResNet-34 (40 views/object, depth)	0.350	0.379

Table 5: Best training and validation losses obtained during training of ResNet-18 and ResNet-34 on 10-view and 40-view datasets of captured views of ModelNet10 objects.



(a) Trend of training and validation loss during training of ResNet-18 on a 10-view dataset of images of ModelNet10 objects. 14 epochs were needed to reach the best validation loss.

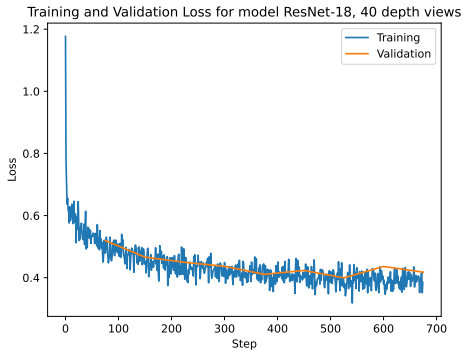


(b) Trend of training and validation loss during fine-tuning of ResNet-18 on 40-view dataset. Losses start at lower values, since the model already has some knowledge of the data. Here, 9 epochs were needed.

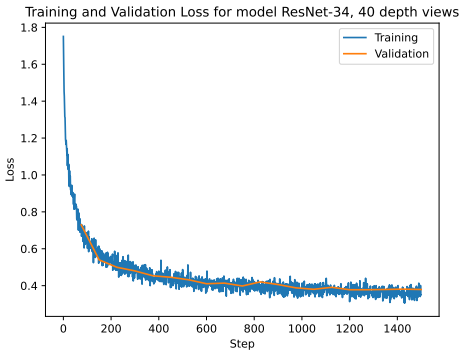
Figure 18: Trend of loss during training of ResNet-18 on a 10-view dataset of images of ModelNet10 objects (left) and fine-tuning of ResNet-18 on 40-view dataset (right).

One other ResNet-18 backbone was trained on the depth captures obtained from the synthetic dataset. For this, again, to speed up training, the weights were initialized with the weights of the model trained on images. The training and validation losses are shown in Figure 19a.

The ResNet-34 backbone was also trained on the same depth captures. The training and validation losses are shown in Figure 19b. As the architecture of the ResNet-34 backbone is different, the weights were not initialized from a previously trained model, but with again with ImageNet weights.



(a) Trend of training and validation loss during training of ResNet-18 on 40-view dataset of depth captures. We see again that loss decreases much faster, as the model is pretrained on images, and so already has some knowledge of the data. 9 epochs were needed until early stopping.



(b) Trend of training and validation loss during training of ResNet-34 on 40-view dataset of depth captures. Since weights were not initialized from a previous model, loss starts at a similar value to the first model's loss. Training also took longer, with 20 epochs until early stopping.

Figure 19: Trend of loss during training of ResNet-18 on a 10-view dataset of depth captures of ModelNet10 objects (left) and training of ResNet-34 on 40-view dataset of depth captures (right).

All models were then tested on the test dataset. Table 6 shows a comparison of the test accuracies obtained. Each model is only tested on the dataset of the same modality as it was trained on. Also, we find that each model performs better on the dataset with the same number of views per object as its training dataset, even when the test dataset has fewer views per object.

Backbone	Test acc. (10-view dataset)	Test acc. (40-view dataset)
ResNet-18 (10 views/object, image)	79.34	66.46
ResNet-18 (40 views/object, image)	73.27	77.98
ResNet-18 (40 views/object, depth)	76.72	80.09
ResNet-34 (40 views/object, depth)	78.21	81.31

Table 6: Single view test accuracies (%) of backbone ResNet-18 and ResNet-34 models trained on 10-view and 40-view per object datasets of image and depth captures of ModelNet10 objects.

Multi-view classification Two view selection methods were evaluated for the complete multi-view classification pipeline: random and point cloud embedding. These are described in Section 4.4. The same backbone networks were used for all methods, namely the last 3 models in Table 6. Again, the test set of ModelNet10 was used for evaluation. For the first test, we use a 10-view PointNet++ model for depth entropy prediction, coupled with a ResNet-18 model for classification based on image captures; the confidence threshold was set to 0.99 and the maximum

number of views to 4. Each mesh was classified 3 times starting from a different random viewpoint, and the average accuracy was taken. We obtain a classification accuracy of 86.56% for the random view selection method and 89.43% for the point cloud embedding method. As expected, both methods perform better than a single-view model, with our proposed point cloud embedding method performing better than baseline random view selection.

For a qualitative analysis, we show the confusion matrices for the two methods in Figure 23 (in the Appendix). We find that classes with similar shapes, such as the *table* and *desk* classes, or the *nighstand* and *dresser* classes, are more difficult to distinguish. While the objects have different sizes, the meshes are all normalized to fit in a unit cube, so the size information is lost. Taking this size information into account could be a future direction for improvement.

However, for a more thorough quantitative evaluation of the PointNet++ method, the classification accuracy was measured with varying minimum confidence thresholds and maximum number of views. This was done to determine in which contexts the method makes the biggest difference. In a real-world application, we cannot freely choose them, as we need to make a tradeoff between accuracy and speed; thus we need to know how the method performs for different values. The accuracy was measured for minimum confidence thresholds of 0.5, 0.6, 0.7, 0.8, 0.9, 0.95 and 0.99 and maximum number of views of 2, 3, 4, 5, 6, 7, 8, and 9. Accuracies were then averaged across backbone models (ResNet-18 and ResNet-34) and across datasets (image and depth). The average improvement results of this grid search are shown in Table 7 (and in more detail in Table 10 in the Appendix) and visualized in Figure 20. We can clearly see from this table that the PointNet++ method is better than the Random method for most cases. The advantage of the PointNet++ method increases with the maximum number of views allowed, and with the minimum confidence threshold. We see in Table 13 (in the Appendix) that the higher these are, the more views are selected on average. Thus an advantage for the PointNet++ method is expected, as it is more likely to select informative view than the Random method, and the more views are allowed, the bigger a difference it will make. Even so, when comparing the number of views required to the random method, our proposed method usually needs fewer views to reach the confidence thresholds (see Table 14).

For a maximum number of views of 1, the PointNet++ method is equivalent to the Random method, as it selects a single view at random; this case can be disregarded. For almost all other cases however, we can see a clear improvement when using the PointNet++ method, compared to the Random method. In short, the more views are allowed, the better the PointNet++ method performs.

For a 40-view PointNet++ model, the accuracy was again measured for different minimum confidence thresholds and maximum number of views. The results of this grid search are shown in Figure 25 (in the Appendix) and Tables 11 and 12. The results are less clear than for the 10-view model, with smaller improvements, but we can still see that the PointNet++ method performs better than the Random method for most cases.

Of all experiments, the highest accuracy obtained for any configuration was 92.31%, for a maximum number of views of 9 and a confidence threshold of 0.99, using a ResNet-18 backbone network trained on depth, with 10 possible views. Table 8 shows the best results obtained for each backbone network, for both the 10 and 40 possible views. We report also the class accuracy, which is the average accuracy for each class, and thus is less affected by class imbalance (see Section 5.2).

MV \ CT	0.5	0.6	0.7	0.8	0.9	0.95	0.99
2	1.248	1.285	-0.661	1.101	1.358	1.689	0.661
3	-0.330	0.147	1.468	1.872	1.579	1.028	0.844
4	0.771	0.844	0.624	2.093	1.175	1.028	2.166
5	-0.551	0.587	0.477	1.836	1.285	1.285	2.056
6	0.110	-0.110	-0.881	3.965	1.211	2.643	0.551
7	-0.330	0.330	1.872	0.110	2.203	0.661	1.211
8	0.330	1.322	-0.110	3.084	-0.220	2.863	3.304
9	0.771	1.322	0.991	0.881	1.762	2.423	0.661

Table 7: Grid search results for the proposed pipeline, for a 10-view model. MV stands for maximum number of views, and CT stands for confidence threshold. The table shows the measured improvement in accuracy of the PointNet++ method, compared to the Random method, for a 10-view model. The numbers represent the accuracy difference, in percentage points, of the PointNet++ method and the Random method, for a certain minimum confidence threshold and maximum number of views. Cells contents are green if the PointNet++ method is better than the Random method, and black otherwise. Figure 20 shows the improvements in a more visual way.

Method	Accuracy	Class accuracy
ResNet-18 (images) + PointNet++ (10 views)	90.63%	90.33%
ResNet-18 (images) + PointNet++ (40 views)	89.21%	89.08%
ResNet-18 (depth) + PointNet++ (10 views)	92.31%	92.11%
ResNet-18 (depth) + PointNet++ (40 views)	89.99%	89.62%
ResNet-34 (depth) + PointNet++ (10 views)	89.43%	89.02%
ResNet-34 (depth) + PointNet++ (40 views)	89.32%	89.13%

Table 8: Best results obtained for each backbone network, for both the 10-view and 40-view models. We report both the accuracy (per instance) and the class accuracy (averaged over classes).

Interestingly, the 10-view configurations perform slightly better than the 40-view configurations; this could be because the entropy prediction (PointNet++) network finds it easier to learn 10 views than 40 views, and underfits for the latter. Increasing the variance of the PointNet++ could help to improve the results for the 40-view model. Also, ResNet-18 performs better than ResNet-34; this could be because, as a smaller network, it is less confident in its predictions, thus requiring more views, leading to incorrect predictions canceling each other out.

For comparison, we also report accuracies obtained with an oracle method: a method that always selects the optimal view, based on the ground truth labels for depth entropies. This method is strictly hypothetical, but it is useful to see how much better the PointNet++ method could perform if it was perfect. Section C in the Appendix shows the results of this oracle method, for 40 possible

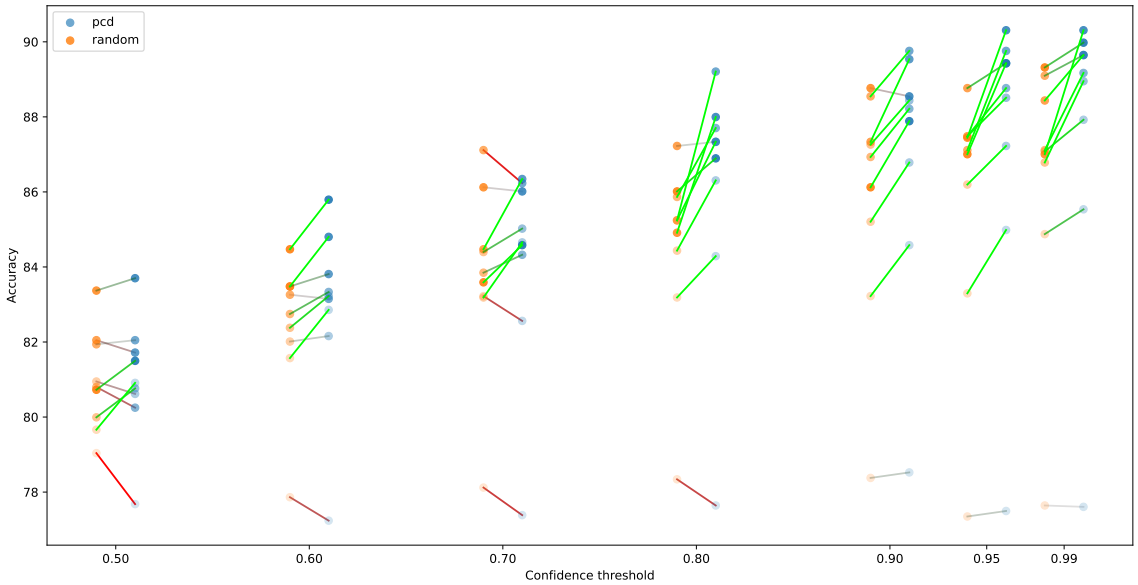


Figure 20: Average of classification accuracies vs minimum confidence threshold and maximum number of views for a 10-view model. The accuracy is measured on the test set of ModelNet10; it is represented on the vertical axis. The horizontal axis represents the minimum confidence threshold; points are offset horizontally for better visibility, but the actual values are shown on the axis as ticks. The points are coloured based on the method used to select the views (see legend). The maximum number of steps is indicated by the alpha value of the points (the more steps, the more opaque the point). For an easier comparison between the methods, the points are connected by lines (green if the PointNet++ method is better than the Random method, red if worse). The specific values can be seen in Table 10.

views. We only test it up to 5 maximum views attempted. The best accuracy we obtain is 90.53%, for a maximum number of views of 5 and a confidence threshold of 0.99 (see Table 15 in the Appendix); the backbone network used for this accuracy is a ResNet-18 trained on depth. Overall we find little difference in performance between the PointNet++ method and the oracle method, which means that the PointNet++ has learned to predict depth entropies; whatever other limitations there are, they are likely not due to the PointNet++ network. Interestingly, we find that even with the oracle method, ResNet-18 often performs better than ResNet-34. This might be because the ResNet-34 network is more complex, and thus more prone to overfitting, or because the ResNet-18 network is less confident in its predictions, and thus more likely to select more views (which is beneficial to the accuracy).

5.3.4 GPU Memory

We report the GPU (graphics processing unit) memory used by four methods (random selection, Neural Renderer, DiffDVR and PointNet++) in Table 9. This was measured using the *max_memory_allocated()* PyTorch function. The Neural Renderer and DiffDVR were run outside of the proposed pipeline,

as they were not integrated into it. Results show that the memory used by the PointNet++ method is higher than the memory used by the Random method, but the difference is not significant. The Neural Renderer and DiffDVR however use considerably more memory than any of the other two methods, which is another argument in favour of the PointNet++ method. This is likely because these differentiable rendering methods use gradient descent to find the optimal view, which requires storing the intermediate tensors for backpropagation and gradient computation. We find a very small difference (under 1MB) between the memory used by the PointNet++ method for 10 views and 40 views; this is because only the last layer of the PointNet++ network differs. However, the choice of backbone between ResNet-18 and ResNet-34 networks does show a difference, with the latter using about 40 MB more memory.

Method	Allocated GPU Memory (MB)
ResNet-18 + Random view selection	69.76 MB
ResNet-34 + Random view selection	109.23 MB
Neural Renderer	1364.00 MB
DiffDVR	496.49 MB
ResNet-18 + PointNet++ (10 views)	81.95 MB
ResNet-18 + PointNet++ (40 views)	81.97 MB
ResNet-34 + PointNet++ (10 views)	121.42 MB
ResNet-34 + PointNet++ (40 views)	121.45 MB

Table 9: Memory used by the three methods. This is measured as the maximum memory allocated by the GPU, when determining best views on the test set of ModelNet10.

6 Conclusion

6.1 Summary of Main Contributions

The work presented introduces an efficient method of estimating the optimal views required for multi-view object representation, termed Maximum Entropy View Selection (MEVS). A great advantage of MEVS is the fact that it is object category-agnostic, requiring no information about the class of the object being analyzed. While differentiable rendering proved to be inefficient, an alternative method based on point cloud embeddings was proposed. The results show that the point cloud-based approach is much faster, while still being able to select the most informative views of the object. This is a desirable property, as it allows the method to be used in real-time applications, such as robotic manipulation. Point cloud data is also more readily available than CAD models, requiring less preprocessing, and can be obtained using a variety of sensors, such as RGB-D cameras or LiDARs.

We also propose a novel method of sampling views of the object, which is scalable to any number of views. We introduce a data structure for entropy prediction and an algorithm for parsing the view graph, which is used to select the optimal views.

We explore several contexts for using the method, to evaluate the tradeoffs between accuracy and computational cost. Clear improvements in accuracy over the baseline are observed. We empirically find that the proposed method makes a bigger difference in accuracy when used in conjunction with a less powerful network, such as ResNet-18, as opposed to the more powerful ResNet-34. A possible explanation is that the more powerful network predicts with more confidence, and thus fewer additional views are required to reach a certain confidence threshold. This means that the view selection method is given less room to improve the accuracy of the prediction.

6.2 Future Work

Even given its high computational cost and currently impractical time requirements, differentiable rendering it is still a promising method for view selection. Future work could focus on using improved differentiable renderers.

Another possible improvement, which was not explored in this work, would be to use the complete ModelNet40 dataset. It contains 12,311 CAD models of 40 object categories, and, being a much more diverse dataset, would allow for a more thorough evaluation of the proposed method [76]. However, neither of these datasets contain color information, which is an important factor in object recognition. Given this, it is remarkable that the proposed method is able to achieve such improvements, even without color information.

The input of the depth entropy prediction network can be adapted to make use of Vector Neurons, as proposed by Deng et al. [12]. These enable rotation invariance or equivariance, which is a desirable property for MEVS, as it would allow the network to learn to select the optimal views, regardless of the orientation of the object.

We also find that discarding size information by normalizing all object meshes negatively impacts the ability of the system to separate certain classes. Future work could focus on finding a way

to incorporate size information into the system, either by removing the normalization step, or by treating the size differently for the backbone network and the view selection method. The backbone itself could also be changed to another type of network such as a Vision Transformer [14]; this architecture might have an advantage in using multiple views, handling occlusion patterns more explicitly through the use of attention mechanisms.

The output of the depth entropy prediction network could also be changed from a flat vector representation to an undirected graph representation. This could be done by using graph embeddings, first proposed by Scarselli et al. [60]. While the current implementation does use a graph as an intermediate step for detecting local maxima, using a proper graph embedding for the output could allow the network to exploit possible relationships between neighboring views when learning. Another option for the output would be spherical CNNs [10]. These could be used to exploit the spherical nature of the view coordinates. Initially developed for cosmological applications, spherical CNNs are graph CNNs which operate on the HEALPix sampling of the sphere [10].

The content of the output could also be changed to something other than depth entropy. For example, surfaces that are angled with respect to the camera result in a high depth entropy but are not always informative. As we see that even with the oracle view selection method, the accuracy still has room for improvement, it is possible that depth entropy is not the best metric for view selection. A possible solution would be to use some other metrics, such as a measure of the amount of edges in the image (e.g. Canny edge detection [5] followed by summing the entire image), or a measure of the confidence in the correct classification of the object (e.g. the maximum value of the softmax output).

Another factor to consider is that the currently presented method is not evaluated (or optimized) in terms of movement costs, which is an important factor in robotic applications. The only metric collected is the number of views required to reach a certain confidence threshold. Future work could focus on optimizing the travel distance, by precomputing it for view pairs, then taking it into account when parsing the view graph and selecting the optimal views. This is nontrivial, as the travel distance is dependent on the robot's joint configuration and collision avoidance, and entails solving an inverse kinematics problem in a complex cost space.

The designed system can be expanded to pose estimation, by using the captured views to train a neural network to estimate the pose of the object; possibly jointly using branching. This can be done treated as a 6-DoF regression problem, where the network is trained to predict the 6D pose of the object, given views of the object. After, or in parallel with estimating the pose, we can also compute the optimal gripper position for grasping the object. This task (described also in Section 2.1) is complex, requiring a collision-free trajectory, and must result in the object being solidly immobilized in the gripper.

To sum up, the proposed MEVS method offers an effective and object category-agnostic approach for multi-view object representation. Through the use of point cloud embeddings, MEVS achieves real-time performance, making it particularly well-suited for robotic manipulation and other time-sensitive applications. By extending the system, we can continue advancing the field of computer vision and enable more efficient and accurate multi-view object representation for a wide range of practical applications.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171–190, 2000. ISSN 0377-0427. doi: 10.1016/S0377-0427(00)00422-2. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[3] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, 1999. doi: 10.1109/2945.817351.

[4] Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. Generative and discriminative voxel modeling with convolutional neural networks, 2016.

[5] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8:679 – 698, 12 1986. doi: 10.1109/TPAMI.1986.4767851.

[6] L. E. Carvalho and A. von Wangenheim. 3d object recognition and classification: a systematic literature review. *Pattern Analysis and Applications*, 22(4):1243–1292, February 2019. doi: 10.1007/s10044-019-00804-4.

[7] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository, 2015.

[8] R. Qi Charles, Hao Su, Mo Kaichun, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 77–85, July 2017. doi: 10.1109/CVPR.2017.16.

[9] E.R. Davies. Chapter 18 - image transformations and camera calibration. In E.R. Davies, editor, *Computer and Machine Vision (Fourth Edition)*, pages 478–504. Academic Press, Boston, fourth edition edition, 2012. ISBN 978-0-12-386908-1. doi: 10.1016/B978-0-12-386908-1.00018-5.

[10] Michaël Defferrard, Martino Milani, Frédérick Gusset, and Nathanaël Perraudin. DeepSphere: a graph-based spherical CNN. In *International Conference on Learning Representations (ICLR)*, 2020.

[11] B. N. Delaunay. Sur la sphère vide. *Bull. Acad. Sci. URSS*, 1934(6):793–800, 1934.

[12] Congyue Deng, Or Litany, Yueqi Duan, Adrien Poulenard, Andrea Tagliasacchi, and Leonidas Guibas. Vector neurons: A general framework for so (3)-equivariant networks. *arXiv preprint arXiv:2104.12229*, 2021.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai,

Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[15] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. In *IEEE Trans. Inf. Theory*, 1983.

[16] Mohamed Elhoseiny, Tarek El-Gaaly, Amr Bakry, and A. Elgammal. A comparative analysis and study of multiview cnn models for joint object categorization and pose estimation. *undefined*, 2016.

[17] Yifan Feng, Zizhao Zhang, Xibin Zhao, Rongrong Ji, and Yue Gao. Gvcnn: Group-view convolutional neural networks for 3d shape recognition. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 264–272, 2018. doi: 10.1109/CVPR.2018.00035.

[18] Abubakar Sulaiman Gezawa, Yan Zhang, Qicong Wang, and Lei Yunqi. A review on deep learning approaches for 3d data representations in retrieval and classifications. *IEEE Access*, 8:57566–57593, 2020. doi: 10.1109/access.2020.2982196.

[19] Rohit Girdhar, David F. Fouhey, Mikel Rodriguez, and Abhinav Gupta. Learning a predictable and generative vector representation for objects, 2016.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[21] Martin Hahner, Dengxin Dai, Alexander Liniger, and Luc Van Gool. Quantifying data augmentation for lidar based 3d object detection, 2020.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[23] G E Hinton. Learning translation invariant recognition in massively parallel networks. In *Volume I: Parallel Architectures on PARLE: Parallel Architectures and Languages Europe*, page 1–13, Berlin, Heidelberg, 1987. Springer-Verlag. ISBN 0387179437.

[24] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2016.

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[26] Jianwen Jiang, Di Bao, Ziqiang Chen, Xibin Zhao, and Yue Gao. MLVCNN: Multi-loop-view convolutional neural network for 3d shape retrieval. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):8513–8520, July 2019. doi: 10.1609/aaai.v33i01.33018513.

[27] Asako Kanezaki, Yasuyuki Matsushita, and Yoshifumi Nishida. RotationNet: Joint object categorization and pose estimation using multiviews from unsupervised viewpoints, 2018.

[28] Hamidreza Kasaei and Mohammadreza Kasaei. Mvgrasp: Real-time multi-view 3d object grasping in highly cluttered environments. *Robotics and Autonomous Systems*, 160:104313, 2023.

[29] Hamidreza Kasaei, Sha Luo, Remo Sasso, and Mohammadreza Kasaei. Simultaneous multi-view object recognition and grasping in open-ended domains. *arXiv preprint arXiv:2106.01866*, 2021.

[30] S Hamidreza Kasaei. Orthographicnet: A deep transfer learning approach for 3-d object recognition in open-ended domains. *IEEE/ASME Transactions on Mechatronics*, 26(6):2910–

2921, 2020.

[31] S. Hamidreza Kasaei, Jorik Melsen, Floris van Beers, Christiaan Steenkist, and Klemen Voncina. The State of Lifelong Learning in Service Robots: Current Bottlenecks in Object Perception and Manipulation, 2020.

[32] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3D Mesh Renderer. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[33] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson Surface Reconstruction. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP '06, page 61–70, Goslar, DEU, 2006. Eurographics Association. ISBN 3905673363.

[34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[35] Roman Klokov and Victor Lempitsky. Escape from cells: Deep kd-networks for the recognition of 3d point cloud models, 2017.

[36] Christian Korbach, Markus D. Solbach, Raphael Memmesheimer, Dietrich Paulus, and John K. Tsotsos. Next-best-view estimation based on deep reinforcement learning for active object classification, 2021.

[37] Jan Kukaka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *ArXiv*, abs/1710.10686, 2017.

[38] Kevin Lai, Liefeng Bo, Xiaofeng Ren, and Dieter Fox. A large-scale hierarchical multi-view RGB-d object dataset. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, May 2011. doi: 10.1109/icra.2011.5980382.

[39] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[40] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[41] Jiaxin Li, Ben M. Chen, and Gim Hee Lee. So-net: Self-organizing network for point cloud analysis, 2018.

[42] Lei Li, Siyu Zhu, Hongbo Fu, Ping Tan, and Chiew-Lan Tai. End-to-End Learning Local Multi-view Descriptors for 3D Point Clouds, 2020.

[43] Tzu-Mao Li. Differentiable visual computing, 2019.

[44] Yikun Li, Lambert Schomaker, and S. Hamidreza Kasaei. Learning to Grasp 3D Objects using Deep Residual U-Nets, 2020.

[45] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning, 2019.

[46] Yongcheng Liu, Bin Fan, Shiming Xiang, and Chunhong Pan. Relation-shape convolutional neural network for point cloud analysis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8895–8904, 2019.

[47] Chao Ma, Yulan Guo, Jungang Yang, and Wei An. Learning multi-view representation with lstm for 3-d shape recognition and retrieval. *IEEE Transactions on Multimedia*, 21(5):1169–1182, 2019. doi: 10.1109/TMM.2018.2875512.

[48] Daniel Maturana and Sebastian Scherer. Voxnet: A 3d convolutional neural network for real-

time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928, 2015. doi: 10.1109/IROS.2015.7353481.

[49] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[50] Louis Moresi and Ben Mather. Stripy: A python module for (constrained) triangulation in cartesian coordinates and on a sphere. *Journal of Open Source Software*, 4(38):1410, 2019. doi: 10.21105/joss.01410.

[51] Liangliang Nan and Peter Wonka. PolyFit: Polygonal surface reconstruction from point clouds. In *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, October 2017. doi: 10.1109/iccv.2017.258.

[52] Tommaso Parisotto, Subhaditya Mukherjee, and Hamidreza Kasaei. More: simultaneous multi-view 3d object recognition and pose estimation. *Intelligent Service Robotics*, pages 1–12, 2023.

[53] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[54] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space, 2017.

[55] Shaohua Qi, Xin Ning, Guowei Yang, Liping Zhang, Peng Long, Weiwei Cai, and Weijun Li. Review of multi-view 3d object recognition methods based on deep learning. *Displays*, 69: 102053, September 2021. doi: 10.1016/j.displa.2021.102053.

[56] Robert J. Renka. Algorithm 772: Stripack: Delaunay triangulation and voronoi diagram on the surface of a sphere. *ACM Trans. Math. Softw.*, 23(3):416–434, sep 1997. ISSN 0098-3500. doi: 10.1145/275323.275329.

[57] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, September 1951. doi: 10.1214/aoms/1177729586.

[58] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[59] Silvio Savarese and Li Fei-Fei. Multi-view object categorization and pose estimation. *Studies in Computational Intelligence*, 285:205–231, 2010. ISSN 1860949X. doi: 10.1007/978-3-642-12848-6_8.

[60] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

[61] Nima Sedaghat, Mohammadreza Zolfaghari, Ehsan Amiri, and Thomas Brox. Orientation-boosted voxel nets for 3d object recognition, 2017.

[62] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[63] Baoguang Shi, Song Bai, Zhichao Zhou, and Xiang Bai. Deeppano: Deep panoramic representation for 3-d shape recognition. *IEEE Signal Processing Letters*, 22(12):2339 – 2343, 2015. doi: 10.1109/LSP.2015.2480802. Cited by: 322.

[64] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1), July 2019. doi: 10.1186/s40537-019-0197-0.

[65] Ayan Sinha, Jing Bai, and Karthik Ramani. Deep learning 3d shape surfaces using geometry images. In *European Conference on Computer Vision*, 2016.

[66] Juil Sock, Guillermo Garcia-Hernando, and Tae-Kyun Kim. Active 6D Multi-Object Pose Estimation in Cluttered Scenarios with Deep Reinforcement Learning, 2019.

[67] Markus D. Solbach and John K. Tsotsos. Blocks world revisited: The effect of self-occlusion on classification by convolutional neural networks, 2021.

[68] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[69] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view Convolutional Neural Networks for 3D Shape Recognition, 2015.

[70] Hang Su, Varun Jampani, Deqing Sun, Subhransu Maji, Evangelos Kalogerakis, Ming-Hsuan Yang, and Jan Kautz. Splatnet: Sparse lattice networks for point cloud processing. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2530–2539, 2018. doi: 10.1109/CVPR.2018.00268.

[71] Kai Sun, Jiangshe Zhang, Junmin Liu, Ruixuan Yu, and Zengjie Song. Drcnn: Dynamic routing convolutional neural network for multi-view 3d object recognition. *IEEE Transactions on Image Processing*, 30:868–877, 2021. doi: 10.1109/TIP.2020.3039378.

[72] A. N. Tikhonov. On the stability of inverse problems. *Proceedings of the USSR Academy of Sciences*, 39:195–198, 1943.

[73] Xin Wei, Ruixuan Yu, and Jian Sun. View-gcn: View-based graph convolutional network for 3d shape analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[74] Sebastian Weiss and Rüdiger Westermann. Differentiable direct volume rendering. volume 28, pages 562–572, 2022. doi: 10.1109/TVCG.2021.3114769.

[75] R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8): 463–464, aug 1964. ISSN 0001-0782. doi: 10.1145/355586.364791.

[76] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes, 2014.

[77] Yaoqing Yang, Chen Feng, Yiru Shen, and Dong Tian. Foldingnet: Point cloud auto-encoder via deep grid deformation, 2018.

[78] Z. Yang and L. Wang. Learning relationships for multi-view 3d object recognition. *Proc. CVPR*, page 7505 – 7514, 2019. Cited by: 2.

[79] Tizian Zeltner, Sébastien Speierer, Iliyan Georgiev, and Wenzel Jakob. Monte Carlo Estimators for Differential Light Transport. *Transactions on Graphics (Proceedings of SIGGRAPH)*, 40(4), August 2021. doi: 10.1145/3450626.3459807.

[80] Jing Zhao, Xijiong Xie, Xin Xu, and Shiliang Sun. Multi-view learning overview: Recent progress and new challenges. *Information Fusion*, 38:43–54, 11 2017. ISSN 15662535. doi: 10.1016/j.inffus.2017.02.007.

[81] H. Zhou, A. Liu, W. Nie, and J. Nie. Multi-View Saliency Guided Deep Neural Network for 3-

D Object Retrieval and Classification. *IEEE Transactions on Multimedia*, 22(6):1496–1506, 2020. doi: 10.1109/TMM.2019.2943740.

[82] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A Modern Library for 3D Data Processing. *arXiv:1801.09847*, 2018.

[83] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection, 2017.

Appendices

A Local Maxima in View Graph

Result: Set of nodes with higher weights than neighbors
Input : `graph` - A list of `Node` objects
Output: `result` - A set of `Node` objects
`result` $\leftarrow \varnothing$
for *each* `node` ***in*** `graph` **do**
  `is_higher_weight` $\leftarrow$ **true**;
  for *each* `neighbor` ***in*** `node.neighbors` **do**
    if `neighbor.weight` $\geq$ `node.weight` **then**
      `is_higher_weight` $\leftarrow$ **false**;
      break;
    end
  end
  if `is_higher_weight` **then**
    `result` $\leftarrow$ `result` $\cup$ `node`;
  end
end
return `result`;

Algorithm 1: Finding local maxima in a view graph. The *graph* is a list of *Node*s, and each *Node* has a list of *neighbors* (other *Node*s) and a *weight*. The *result* is another set of *Node*s. We iterate through each *Node* in the *graph*, and check if it has a higher weight than all its neighbors; if so, we add it to the *result*, which we return at the end. Variations attempted were storing nodes in a heap queue, skipping previously visited nodes, and pre-sorting the nodes by weight. However, these did not improve the performance. Thus the algorithm is $O(|V|^2)$, where $|V|$ is the number of nodes in the graph. We benchmark on the 10-view and the 40-view graph; the algorithm took on average 0.144 ms and 0.584 ms respectively.

Figure 21 shows the result of this algorithm on an example graph.

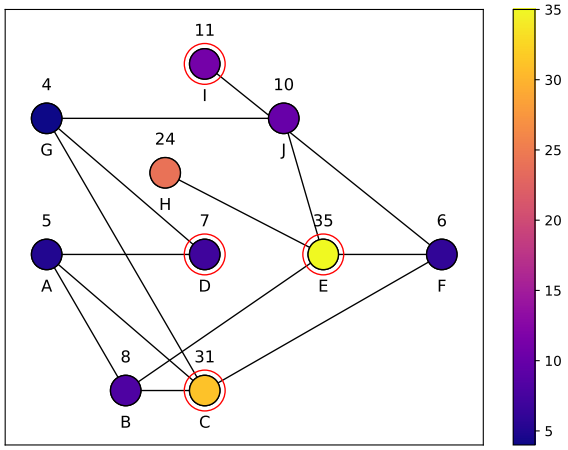


Figure 21: Example graph with local maxima (nodes E, D, C, I) highlighted in red. The node names are shown under each node, and the node weights above. Note how each local maximum has a higher weight than all its neighbors. This figure also can be obtained by running the script `node_weighted_graph/check_functionality.py` in the `nbv_mevs` repository.

B Captured depth example

Figure 22 shows the depth entropies along with some corresponding depth captures.

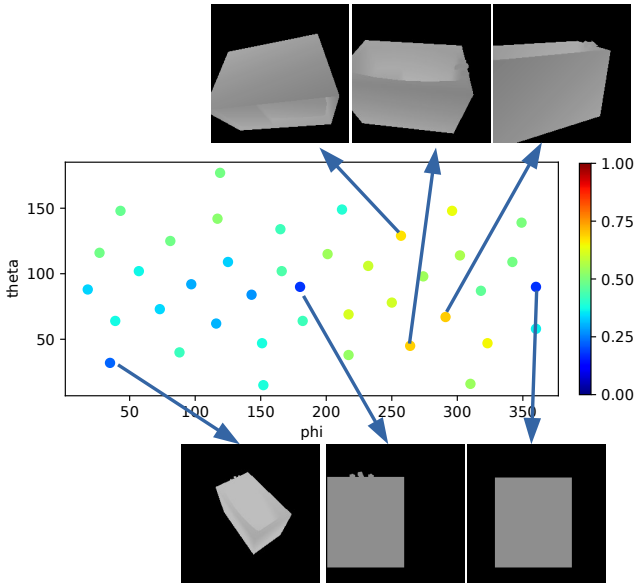


Figure 22: For the same object as in Figure 7b, we show depth captures for the top 3 highest depth entropy views (top), and the 3 lowest (bottom). We see that the highest depth entropies tend to be from angled views, while the lowest depth entropies tend to be from views that are more orthogonal to the object surfaces, or if the closest point of the object is further from the camera.

A Confusion Matrices

We report the confusion matrices for the baseline and our method in Figure 23 and the difference in Figure 24. For both, the confidence threshold is 0.99 and the maximum number of views is 4.

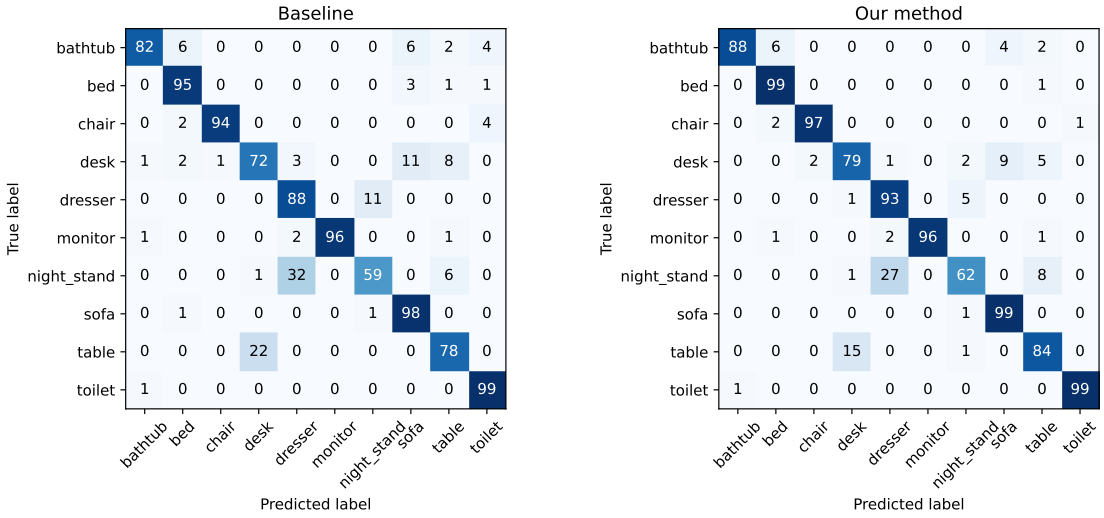


Figure 23: Confusion matrices for multi view classification using random viewpoint selection (left) and PointNet++ depth entropy prediction (right). Numbers are percentages of the total number of objects in each class.

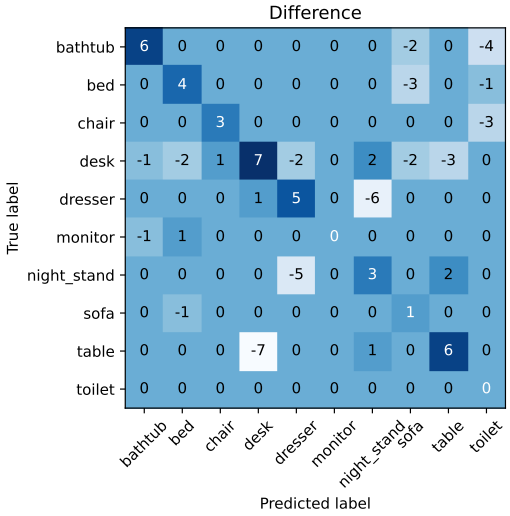


Figure 24: Difference between the confusion matrices for multi view classification using random selection and PointNet++ view selection. Numbers are percentages. We see that the proposed method performs better for all classes; all values on the main diagonal are greater than or equal to 0.

B Average improvements

Below we see some more thorough comparisons of the average improvements of the PointNet++ method, compared to the Random method. In all tables, MV stands for maximum number of views, and CT stands for confidence threshold. 'pcd' and 'random' stand for PointNet++ and Random View Selection, respectively.

A Accuracy results

A.1 Results for 10 possible viewpoints

We report the average accuracies and improvement in accuracy of the PointNet++ method, compared to the Random method, for 10 possible viewpoints in Tables 10 and 7, respectively.

MV \ CT	0.5	0.6	0.7	0.8	0.9	0.95	0.99
2 (random)	79.66	81.57	**83.22**	83.19	83.22	83.30	84.88
2 (pcd)	**80.91**	**82.86**	82.56	**84.29**	**84.58**	**84.99**	**85.54**
3 (random)	**80.95**	82.01	83.19	84.43	85.21	86.20	87.08
3 (pcd)	80.62	**82.16**	**84.65**	**86.31**	**86.78**	**87.22**	**87.92**
4 (random)	79.99	82.38	84.40	85.24	87.26	87.48	86.78
4 (pcd)	**80.76**	**83.22**	**85.02**	**87.33**	**88.44**	**88.51**	**88.95**
5 (random)	**80.80**	82.75	83.85	85.87	86.93	87.48	87.11
5 (pcd)	80.25	**83.33**	**84.32**	**87.70**	**88.22**	**88.77**	**89.17**
6 (random)	81.94	**83.26**	**87.11**	85.24	88.55	87.11	89.10
6 (pcd)	**82.05**	83.15	86.23	**89.21**	**89.76**	**89.76**	**89.65**
7 (random)	**82.05**	83.48	84.47	87.22	87.33	88.77	88.44
7 (pcd)	81.72	**83.81**	**86.34**	**87.33**	**89.54**	**89.43**	**89.65**
8 (random)	83.37	83.48	**86.12**	84.91	**88.77**	87.44	87.00
8 (pcd)	**83.70**	**84.80**	86.01	**88.00**	88.55	**90.31**	**90.31**
9 (random)	80.73	84.47	83.59	86.01	86.12	87.00	89.32
9 (pcd)	**81.50**	**85.79**	**84.58**	**86.89**	**87.89**	**89.43**	**89.98**

Table 10: Average accuracy over all pipelines using ResNet backbones, with PointNet++ trained for 10 views.

A.2 Results for 40 possible viewpoints

We report the average accuracies and improvement in accuracy of the PointNet++ method, compared to the Random method, for 40 possible viewpoints in Tables 11 and 12, respectively. We visually represent the improvements in Figure 25.

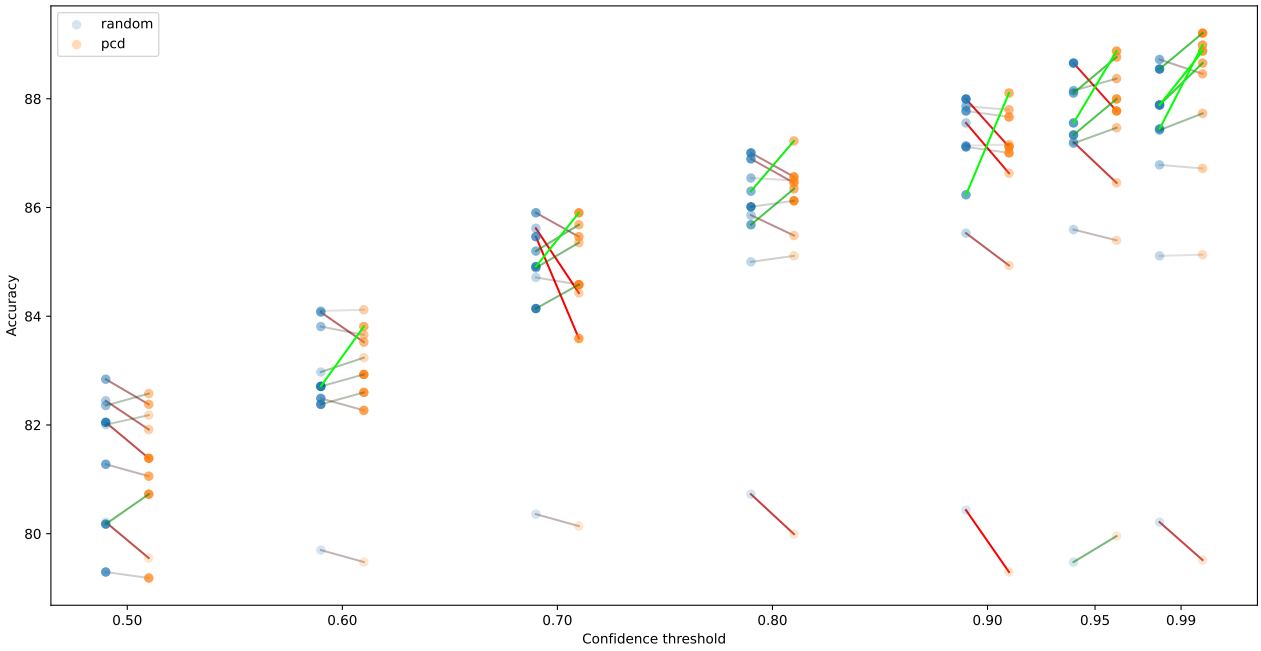


Figure 25: Average accuracy vs minimum confidence threshold and maximum number of views for a 40 possible viewpoints. The average accuracy is represented on the vertical axis; otherwise the data is represented in the same way as in Figure 20. The specific values can be seen in Table 11.

MV \ CT	0.5	0.6	0.7	0.8	0.9	0.95	0.99
2 (random)	82.00	82.97	**84.71**	85.00	**85.53**	**85.59**	85.11
2 (pcd)	**82.18**	**83.24**	84.58	**85.11**	84.93	85.40	**85.13**
3 (random)	**82.44**	84.10	**85.62**	**85.86**	**87.56**	**87.20**	**86.78**
3 (pcd)	81.92	**84.12**	84.43	85.48	86.63	86.45	86.72
4 (random)	82.36	**83.81**	84.89	**86.54**	87.14	87.18	87.42
4 (pcd)	**82.58**	83.66	**85.35**	86.50	**87.16**	**87.47**	**87.73**
5 (random)	**82.84**	**84.07**	85.20	86.30	**87.86**	88.15	**88.72**
5 (pcd)	82.38	83.52	**85.68**	**87.22**	87.80	**88.37**	88.46
6 (random)	**81.28**	82.71	**85.90**	85.68	86.23	88.11	87.89
6 (pcd)	81.06	**83.81**	85.46	**86.34**	**88.11**	**88.77**	**88.66**
7 (random)	**79.30**	**82.49**	**85.46**	**86.89**	**87.78**	87.56	87.44
7 (pcd)	79.19	82.27	83.59	86.45	87.67	**88.88**	**88.99**
8 (random)	80.18	82.38	84.91	**87.00**	**87.11**	87.33	87.89
8 (pcd)	**80.73**	**82.60**	**85.90**	86.56	87.00	**88.00**	**88.88**
9 (random)	**82.05**	82.71	84.14	86.01	**88.00**	**88.66**	88.55
9 (pcd)	81.39	**82.93**	**84.58**	**86.12**	87.11	87.78	**89.21**

Table 11: Average accuracy over all pipelines using ResNet backbones, with PointNet++ trained for 40 views.

MV \ CT	0.5	0.6	0.7	0.8	0.9	0.95	0.99
2	0.176	0.264	-0.132	0.110	-0.595	-0.198	0.022
3	-0.529	0.022	-1.189	-0.374	-0.925	-0.749	-0.066
4	0.220	-0.154	0.463	-0.044	0.022	0.286	0.308
5	-0.463	-0.551	0.485	0.925	-0.066	0.220	-0.264
6	-0.220	1.101	-0.441	0.661	1.872	0.661	0.771
7	-0.110	-0.220	-1.872	-0.441	-0.110	1.322	1.542
8	0.551	0.220	0.991	-0.441	-0.110	0.661	0.991
9	-0.661	0.220	0.441	0.110	-0.881	-0.881	0.661

Table 12: Average improvement in accuracy over all pipelines using ResNet backbones, with PointNet++ trained for 40 views. This is an aggregated version of Table 11, subtracting every 2 rows. Figure 25 shows the improvements in a more visual way.

B Number of viewpoints attempted

MV \ CT	0.5	0.6	0.7	0.8	0.9	0.95	0.99
1 (random)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1 (pcd)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2 (random)	1.11	**1.20**	**1.30**	**1.38**	**1.50**	1.58	1.73
2 (pcd)	**1.11**	1.20	1.28	1.38	1.49	**1.59**	**1.74**
3 (random)	**1.13**	**1.23**	**1.39**	**1.54**	1.71	1.88	2.18
3 (pcd)	1.11	1.22	1.36	1.51	**1.74**	**1.89**	**2.18**
4 (random)	**1.12**	1.25	**1.42**	1.59	1.86	**2.07**	**2.47**
4 (pcd)	1.12	**1.26**	1.40	**1.62**	**1.87**	2.07	2.46
5 (random)	1.11	**1.26**	**1.42**	1.65	1.94	**2.20**	**2.68**
5 (pcd)	**1.12**	1.26	1.42	**1.66**	**1.96**	2.18	2.66
6 (random)	1.10	**1.24**	**1.36**	**1.62**	1.94	2.19	**2.76**
6 (pcd)	**1.11**	1.21	1.34	1.57	**1.99**	**2.23**	2.73
7 (random)	**1.10**	**1.25**	**1.41**	**1.64**	**1.97**	**2.32**	**2.96**
7 (pcd)	1.09	1.21	1.38	1.61	1.94	2.24	2.84
8 (random)	1.08	1.22	**1.41**	**1.64**	1.93	**2.35**	**3.07**
8 (pcd)	**1.11**	**1.24**	1.38	1.61	**2.01**	2.28	2.98
9 (random)	**1.11**	1.21	**1.45**	**1.65**	2.03	**2.37**	**3.09**
9 (pcd)	1.10	**1.23**	1.37	1.60	**2.06**	2.36	3.05

Table 13: Average number of points attempted over all backbones, for both 10 and 40 possible views.

MV \ CT	0.5	0.6	0.7	0.8	0.9	0.95	0.99
1	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	0.002	-0.000	-0.017	-0.002	-0.009	0.010	0.000
3	-0.014	-0.010	-0.025	-0.026	0.023	0.009	0.000
4	-0.001	0.012	-0.013	0.028	0.010	-0.006	-0.011
5	0.004	-0.003	-0.003	0.010	0.021	-0.017	-0.017
6	0.004	-0.037	-0.024	-0.054	0.050	0.045	-0.028
7	-0.006	-0.043	-0.021	-0.026	-0.031	-0.077	-0.127
8	0.030	0.018	-0.030	-0.032	0.078	-0.063	-0.088
9	-0.007	0.020	-0.083	-0.045	0.035	-0.011	-0.042

Table 14: Average decrease in number of points attempted over all backbones, for both 10 and 40 possible views.

C Oracle method

We also report the results of the oracle method, which consists of selecting the highest depth entropy view for each point, based on the ground truth labels. We report the results for 40 possible views.

MV \ CT	0.5	0.6	0.7	0.8	0.9	0.95	0.99
2 (ResNet-18, image)	81.39	79.85	81.17	82.49	83.04	82.05	82.93
3 (ResNet-18, image)	80.29	82.38	83.48	84.69	84.36	85.02	84.69
4 (ResNet-18, image)	80.62	81.83	82.93	85.35	86.67	86.56	86.34
5 (ResNet-18, image)	80.07	81.06	84.58	85.90	85.02	86.78	86.56
2 (ResNet-18, depth)	81.06	82.16	82.93	83.26	83.26	83.70	84.36
3 (ResNet-18, depth)	82.71	82.27	84.69	86.34	86.45	87.89	87.78
4 (ResNet-18, depth)	82.60	83.26	85.68	87.33	86.56	88.33	88.44
5 (ResNet-18, depth)	82.27	83.59	86.56	86.23	88.22	88.33	90.53
2 (ResNet-34, depth)	82.49	83.92	84.69	85.24	84.69	84.80	85.24
3 (ResNet-34, depth)	84.03	83.59	85.13	85.57	85.68	87.00	87.00
4 (ResNet-34, depth)	83.15	83.92	85.02	85.57	87.78	87.89	87.78
5 (ResNet-34, depth)	82.82	84.36	85.46	86.45	87.33	88.11	88.99

Table 15: Accuracy of the oracle method, for different confidence thresholds (CT) and maximum number of views (MV), in conjunction with each backbone.