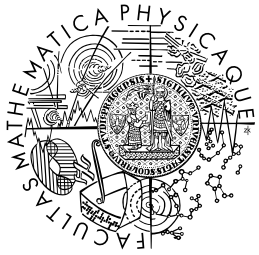


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Miroslav Krabec

3D Object Classification Using Neural Networks

Department of Software and Computer Science Education

Supervisor of the master thesis: doc. Ing. Jaroslav Křivánek, Ph.D.

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

This thesis is dedicated to everyone who supported me during the writing process of the thesis. Especially, I would like to thank my supervisor doc. Ing. Jaroslav Křivánek, Ph.D. for his thorough counseling and also the whole research group for their time and insights.

Title: 3D Object Classification Using Neural Networks

Author: Bc. Miroslav Krabec

Department: Department of Software and Computer Science Education

Supervisor: doc. Ing. Jaroslav Křivánek, Ph.D., Department of Software and Computer Science Education

Abstract: Classification of 3D objects is of great interest in the field of artificial intelligence. There are numerous approaches using artificial neural networks to address this problem. They differ mainly in the representation of the 3D model used as input and the network architecture. The goal of this thesis is to explore and test these approaches on publicly available datasets and subject them to independent comparison, which has not so far appeared in the literature. We provide a unified framework allowing to convert the data from common 3D formats. We train and test ten different network on the ModelNet40 and ShapeNetCore datasets. All the networks performed reasonably well in our tests, but we were generally unable to achieve the accuracies reported in the original papers. We suspect this could be due to extensive, albeit unreported, hyperparameter tuning by the authors of the original papers, suggesting this issue would benefit from further research.

Keywords: deep learning, classification, neural networks, 3D

Contents

1. Introduction

In this chapter we provide reasons and motivation for working on the problem of 3D classification, define the problem, and give a quick outline of the thesis.

1.1 Motivation and Goals

Recognition and generation of 3D shapes is quickly becoming one of the widely researched topics in the field of artificial intelligence. It can be applied in a vast number of fields such as driving of autonomous cars, analysis of medical scans as well as various fields of computer graphics. We approach this problem from the standpoint of computer graphics as we are interested in developing tools for content creators, architects and interior designers. In order to develop such tools it is necessary to start with the simplest problem – classification. There are many more or less successful approaches to 3D classification, most of them employing some kind of artificial neural network. However their relative performance has never been objectively evaluated.

Therefore, the goal of this thesis is to test existing techniques for 3D classification, find out how difficult it is to replicate their reported results, and finally to compare and evaluate them on publicly available datasets.

1.2 Problem Statement

As we intend to develop tools for computer graphics, our input is in the form of a 3D mesh. A 3D mesh is usually supplied as a list of vertices – triplets of coordinates in euclidean space and a list of faces – usually three vertices each forming a triangle. 3D mesh files can contain other information such as texture coordinates and materials, but we will be ignoring these. Our goal is to classify mesh files into several given categories (classes such as “car”, “chair”, “sofa”, etc.) using methods of supervised learning.

We can define the problem formally as follows: we are given a set of training examples $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ where, in our case, x_i is a 3D shape representation and y_i is a numerical encoding of corresponding label. Each shape belongs to exactly one class. The goal of classification can be formulated as learning a parametric model $P : X \to Y$, where X is a space of possible 3D shapes and Y is a space of labels. This model should be able to predict the correct class label for each model from X.

As the mesh format is not suitable for direct use with neural networks, we have to be able to convert meshes to other representations: voxel grids, images or point clouds.

1.3 Scope of the Thesis

We limit the scope of this thesis in the following ways: we will perform classification only on aligned 3D shapes, as not all networks can easily cope with arbitrary rotations. We will only consider simple classification, although most of

the networks can be extended to perform part segmentation as well as new shape generation. We will not test any large ensembles of networks: although they produce better results, they are usually big and cumbersome to use, while achieving only marginal improvements. We also consider only networks with publicly available code as implementing all the different techniques is far beyond the scope of this work.

1.4 Thesis Outline

In Chapter 1 we present the basics of the problem as well as our motivation for this work. In Chapter 2 we provide a brief introduction to artificial neural networks. In Chapter 3 we introduce different approaches to 3D classification and their implementations. In Chapter 4 we discuss chosen methods, describe datasets, procedures of data conversion and the technical setup of our framework. In Chapter 5 we present the setup and results of our experiments, comparison of tested the networks and an analysis of the dataset. In the final Chapter 6 we summarize our findings and suggest possible directions for future work.

2. Theoretical Background

This chapter contains a general overview of artificial neural networks as well as a quick description of software frameworks for machine learning.

2.1 Artificial Neural Networks

Artificial neural networks are widely successful in a great number of areas of computer science, often achieving better than human efficiency. This section offers a brief introduction to the principles of artificial neural networks. For more in depth information we recommend Goodfellow et al. [2016].

2.1.1 Feedforward Neural Networks

Artificial neural networks are computing systems inspired by the structure of central nervous system of animals and humans. A basic computing unit of the network is called *a neuron*, which performs some simple computations on its inputs and produces its output. Neurons are connected by weighted connections creating a network.

Artificial neural networks are parametric models – parameters are usually called *weights* and are learned during the training of the network. On the other hand *hyperparameters* are parameters whose values are set before the learning process begins. Artificial neural networks can be trained for a variety of tasks by minimizing some objective function, called *loss function* in this context. In the case of classification a *cross-entropy* loss is commonly used (Goodfellow et al. [2016]). The networks are trained by some variant of gradient descent algorithm – backpropagating the error of the trained task through the network in direction from outputs to inputs.

Feedforward Neural Networks are networks where the information is passed only in one direction, so the graph of the network is an directed acyclic graph. The simplest network architecture is the so called *Single-layer perceptron*. It consists of a single layer of output neurons; the inputs are fed directly to the outputs via a series of weights. Figure 2.1 shows a diagram of a small single-layer perceptron. Each neuron can also have a bias value and an activation function. Then the output of a neuron is computed:

$$output_j = f(\sum_i w_{ij} \times inputs_i + b_j)$$

Where f is an activation function, w_{ij} is a weight of the connection from i-th input neuron to j-th output neuron and b_j is its bias value. This notation assumes that the activation function accepts a full vector of input values and outputs a corresponding value at each vector component.

The activation function can be an arbitrary function but is required to be non-linear and differentiable. Most commonly used are the sigmoid function, hyperbolic tangent and rectified linear unit (ReLU), defined as $ReLU(x) = \max(0, x)$. ReLU is not a differentiable function in $x = 0$, but in practice this usually does not

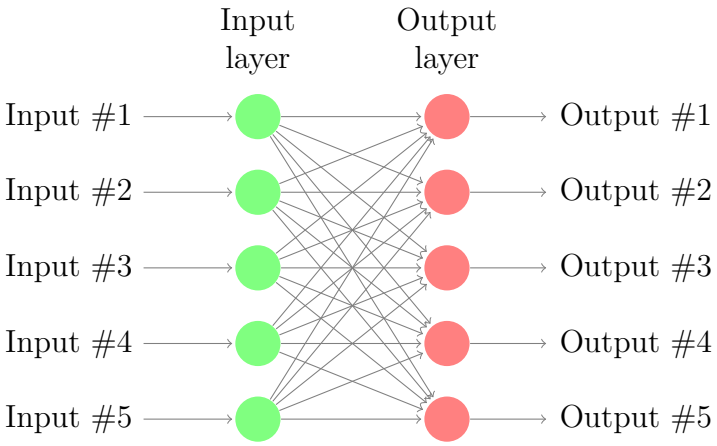


Figure 2.1: Single-layer perceptron

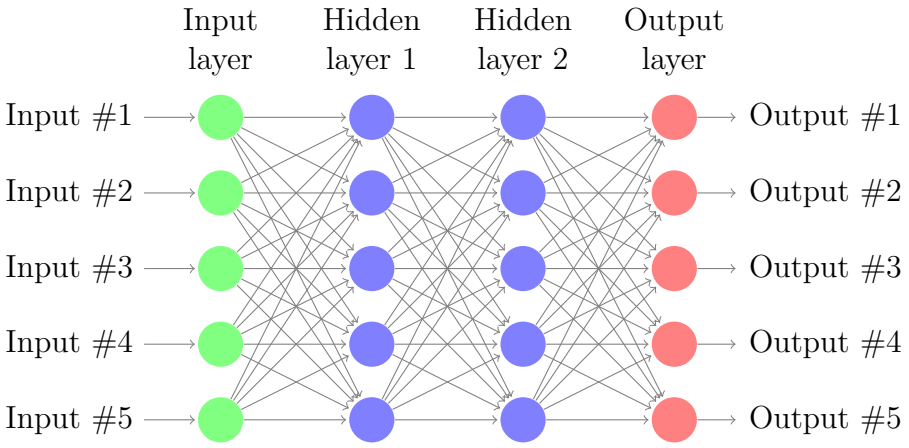


Figure 2.2: Multi-layer perceptron

matter and software frameworks implementing artificial neural networks handle this by returning one of the one-sided derivatives.

We can insert one or more layers of neurons between input and output obtaining a *Multi-layer perceptron*. A neuron from layer k gets its input from all neurons from layer $(k-1)$ and passes its output to all neurons in the layer $(k+1)$. Therefore this type of layer is called *fully connected layer* or *dense layer*. We can also rewrite all the weights to a matrix form and obtain the so called *weight matrix*. For dense layer k we get one matrix W^k, where W^k_{ij} is weight of the connection from neuron i to neuron j. Using the matrix notation, output of a fully connected layer k can be expressed as follows:

$$output_k = f(W^k \times output_{k-1} + b^k)$$

Where f is an activation function, W^k is the weight matrix for layer k and b^k is a corresponding vector of biases.

Multi-layer perceptrons, despite being old (first appearing in Rosenblatt [1961]), still create the basis in modern systems as fully connected layers are used in almost all other types of neural networks especially as last layers in the network, producing feature vectors or classification distributions. Figure 2.2 shows a diagram of a multi-layer perceptron.

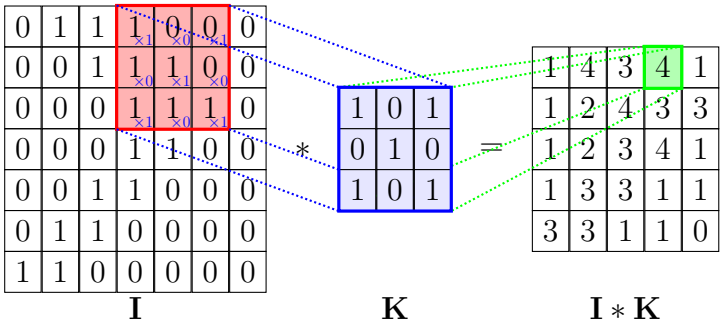


Figure 2.3: Illustration of 2D convolution with one 3×3 filter and valid padding.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks, first introduced in LeCun et al. [1989], are designed to capture the spatio-temporal data better than the standard fully connected layers. They are most successful in processing two-dimensional images therefore we will describe this case. However, also one-dimensional and three dimensional convolutions are used as shall be described later.

Weights of the convolutional layers are connected to only a small region of the data and shared across the spatio-temporal dimensions. In the case of images this means that the convolutional layer is connected to only small patches of the image and weights are shared among these patches. Weights of the convolutional layers are typically called *filters* or *kernels*. A common approach is to slide the filter across the whole image, computing local features for each pixel. Two-dimensional convolution can be defined in a following way:

$$Conv(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

where I is the input image and K represents the weights of the kernel. Figure 2.3 shows an illustration of 2D convolution.

The speed of the sliding window is a parameter called *stride*. By having a stride greater than one, we skip some pixels in the input, and obtain an image with lesser width and height.

We also need to take care of how the convolution behaves on the borders of the input images. These behaviors are called *padding schemes*. The most usual type of padding is *zero padding*, which counts areas outside of the picture as having value of zero and preserves original dimensions. Another approach, *valid padding*, ignores the edge pixels of the input image altogether, sliding the filter only across valid positions – this produces a smaller output image.

By stacking more convolutional layers atop each other and creating a deep convolutional network, global features of the image can be extracted.

Another important type of layer used in the convolutional neural networks is a *pooling layer*. It is usually used on feature maps obtained by convolutional layers. The goal of the pooling is to get some translation invariance in produced features. Similarly to the convolution, pooling also scans the entire input feature map in a sliding-window fashion. However it does not perform convolution but maximum, average or a similar aggregating function. Unlike with convolution, stride bigger than one is used when using pooling in order to reduce the size of the

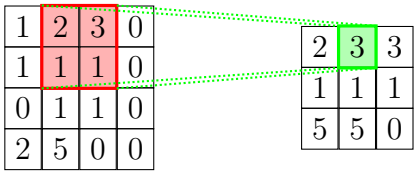


Figure 2.4: Illustration of 2×2 2D maximum pooling with valid padding.

output features. The most commonly used type of pooling – maximum pooling – can be described by the following formula:

$$maxpool(I)[x, y] = \max_{\substack{0 \leq i < d_h \\ 0 \leq j < d_w}} I[x + i - \lfloor \frac{d_h}{2} \rfloor, y + j - \lfloor \frac{d_w}{2} \rfloor]$$

where x and y are coordinates of a pixel in the picture, d_h and d_w are sizes of the sliding window. Figure 2.4 shows a diagram illustrating maximum pooling in two dimensions.

Convolutional layers are much more efficient than fully connected layers as they share their parameters across the input and have been very successful in a variety of image, video, and natural language processing tasks. This can be transferred to our 3D classification task as several networks use rendered images of 3D models, employing 2D convolutions, with great success.

2.1.3 Recurrent Neural Networks

Another widely used class of artificial neural networks are *recurrent neural networks* (RNNs). In contrast to the previously described networks, they take also their previous state as their input, representing a kind of memory. Recurrent neural networks are well suited for processing of sequential data, such as video, text or speech. A typical architecture of an RNN is the *encoder-decoder* architecture. Encoder produces a feature vector by processing the input sequence. Decoder then constructs an output sequence from the feature vector. To give the network control over its memory, two types of cells were devised: *long short-term memory unit* (LSTM) (Hochreiter and Schmidhuber [1997]) and *gated recurrent unit* (GRU) (Cho et al. [2014]). An important concept, which leads to better performance, is *attention* (Bahdanau et al. [2014]) – it allows the network to learn which parts of the input sequence are important and how they correspond to the output sequence.

Only one of the networks tested uses the RNN architecture and techniques described in this section, and so we refer to Goodfellow et al. [2016] for further information.

2.1.4 Classification

This section describes a general scheme of solving the problem of classification, as defined in Section 1.2, using an artificial neural network. The architecture of

the network can be generally split into two parts. Firstly, some kind of network is employed which produces a one-dimensional feature vector, usually an output of a dense layer. This part varies across the different systems and can be implemented using 3D convolutions, 2D convolutions, or dense layers in some hierarchical structure. The second part is formed by a dense layer with the number of neurons equal to the number of categories of classified data. This last layer uses the identity function as its activation function and its output is used for computing the loss function and for classification. The output of i-th neuron can be thought of as the weight of the belief of the network that the classified model belongs to the category with index i. For inference the *argmax function* (returns the index of the highest value) is used on the output of the last dense layer. To transform the output of the last dense layer to a probability distribution, the *softmax function*, defined as $\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$, is used. The loss function used for training is usually the cross entropy between the correct category probability distribution (vector of zeros with a one at the index of the correct class) and the softmax of the output of the last dense layer. The cross entropy of two probability distributions is computed according to the following formula: $H(p, q) = -\sum_{\forall x} p(x) \log(q(x))$.

2.1.5 Regularization

Overfitting is a common problem in machine learning. It is a phenomenon when a model represents the training set too well, memorizing individual training examples, and fails to generalize. To avoid this problem several regularization techniques are employed. We can add regularization directly to the optimized loss function. This is usually implemented by adding a term, which keeps the weights of the network in low absolute values.

Dropout (Srivastava et al. [2014]) is a stochastic regularization technique; during training, at every iteration, it randomly selects some nodes and ignores them along with all of their incoming and outgoing connections. Therefore each iteration has a different set of nodes and this results in a different set of outputs. It is a very effective technique which can be thought of as training a whole ensemble of networks at once. Dropout is usually applied after each fully connected layer.

Another common way how to avoid overfitting is by increasing the size of the training set by *data augmentation*. Specifics of data augmentation depend on the kind of data we are working with, but in general the dataset is increased by creating new instances from the training dataset. For example, with image data, we can use geometric transformations such as mirroring, rotating, translating or scaling.

These techniques are standard when training neural networks and are employed in one form or another in all the networks we tested.

2.1.6 Training

In recent years very deep (tens of layers) neural networks are used. To train such networks several improvements were developed. Mini-batch training is used in almost all cases. The training dataset is divided into small chunks (typically 32 or 64 examples) and one batch is presented, gradient is computed and weights are updated correspondingly in each step of the training. This is much more efficient

than computing the gradient of the whole dataset, and it additionally has some positive regularization effects.

For deep networks it is no longer sufficient to use a simple gradient descent algorithm. To speed up and stabilize the training process, algorithms with momentum, such as Nesterov momentum (Sutskever et al. [2013]), have been used. An important hyperparameter of the network is the learning rate, which controls the speed of training. Set the learning rate too small and the network can not learn at all, too big and the weights may diverge. To solve this problem, algorithms with adaptive learning rates, such as RMSProp (Hinton et al. [2012]) or ADAM (Kingma and Ba [2014]), are used.

Another obstacle in training is that the information contained in the gradient gets lost in deep neural networks during the backpropagation phase of the algorithm. It either diminishes to zero or grows rapidly and diverges. This can be avoided by a good choice of the activation function (in modern networks mainly ReLU is used) or by some normalization technique. A prime example of this is *batch normalization* (Ioffe and Szegedy [2015]), which normalizes the inputs of the layer by subtracting the mean of the batch and dividing by its standard deviation. These changes would be discarded by the learning algorithm, so we add two learnable parameters representing the mean and standard deviation, and the output of the layer is again denormalized using these parameters. This effectively allows the network to learn the correct scaling of the weights using only two parameters instead of changing the whole network, which leads to a much greater stability of the training.

Another widely used technique allowing training of very deep networks are *residual connections* (Szegedy et al. [2016]). A residual connection allows the network to skip the layer and choose to work with the input instead. This makes copying the information through the network possible and helps reduce the vanishing gradient problem.

The above described techniques are used in several neural networks for 3D classification we tested, as some of them use very deep 2D or 3D convolutional networks.

2.2 Deep Learning Frameworks

In recent years several software frameworks for machine learning in general and for deep learning in particular have been developed. They are usually implemented in C++ for performance reasons but provide a Python API for more convenient use. All of the following libraries are open-source and publicly available. They also implement support for running the machine learning algorithms on the GPU – a key feature for fast training of deep neural networks.

One of the most widely used deep learning frameworks is TensorFlow (Martín Abadi et al. [2015]) developed by a team at Google. It is a symbolic math library and implements all the standard neural layers as well as many of the latest developments in the area of deep learning.

PyTorch (Paszke et al. [2017]) is a Python extension of the Torch machine learning library. It is primarily developed by Facebook. It focuses on simple usage and Python integration and implements all the standard functions as well as extended tools for various areas of machine learning.

Caffe (Jia et al. [2014]) is a deep learning framework originally developed at University of California, Berkeley. It offers good speed and training models without the need to write any code, just network definitions have to be provided. It does not seem to be developing as quickly as the aforementioned frameworks. There was also Caffe2 developed by Facebook, but it was merged into PyTorch.

Theano (Theano Development Team [2016]) is a Python library for manipulating and evaluating mathematical expressions, primarily developed by the Montreal Institute for Learning Algorithms at the Université de Montréal. Lasagne (Dieleman et al. [2015]) is a lightweight library which uses Theano for machine learning computations. It also does not seem to be developing quickly enough at present.

Other popular deep learning frameworks, which we do not use in our work, include Microsoft Cognitive Toolkit (Seide and Agarwal [2016]) developed by Microsoft as well as Keras (Chollet and others [2015]) which is a high-level API focusing on user friendliness and uses TensorFlow as its backend, but is modifiable to work with other frameworks as well.

3. Survey of 3D Classification Methods

In recent years many techniques for classifying 3D shapes by means of artificial neural networks have been devised. In this chapter we present most of the commonly used and successful of them. As mentioned in previous chapters the usual mesh format is not suitable for processing by a neural network directly so we divide the networks according to the format they use as their input: voxel-based, multi-view, and point-cloud-based. Table 3.1 shows a list of neural networks described in this chapter.

3.1 Voxel-based Neural Networks

As 2D convolutional neural networks were a great breakthrough in image recognition, it is natural to try to generalize this approach to three dimensions. Instead of pixels we use 3D occupancy grid of *volume elements*, or *voxels*. Convolutions can be easily extended to work in three dimensions. Convolutions seem to be suitable for the task as they can make use of the spatial structure of the problem. However in 3D, they are computationally demanding and voxel grids have high memory requirements as their size grows with the cube of the resolution. For this reason only relatively small resolutions can be used, the most usual being 32^3.

3.1.1 VoxNet

First of the successful systems applying 3D convolutions to occupancy grids is VoxNet (Maturana and Scherer [2015]), which we use as an example of a network using a shallow convolutional architecture. In VoxNet, the occupancy grid is processed by 3D convolutions which extract local features and lower the resolution. The convolution result is passed to a ReLU layer to achieve nonlinearity. Maximum pooling is then performed in order to get better representation and to further lower the number of parameters needed. Finally the occupancy grid is flattened and passed to a fully connected layer which outputs resulting feature vector. Figure 3.1 shows a diagram of the VoxNet architecture.

As is common with the neural networks, data augmentation is a very important part of the training process. VoxNet uses rotation along the vertical axis as its main augmentation technique. During training it creates n copies of each input instance, each rotated by $360/n$ degrees. Typical values of n range from 8 to 24. At evaluation time it presents all rotations of the input object to the network and then uses pooling across the rotations to get the class prediction. A TensorFlow implementation of VoxNet (Maturana and Scherer [2016]) is available.

Results of VoxNet were improved by Orientation boosted voxel nets for 3D Object Recognition (Sedaghat et al. [2016a]), wherein the classification task was augmented with an orientation estimation task. An implementation in Caffe is available (Sedaghat et al. [2016b]). FusionNet (Hegde and Zadeh [2016]) combines a 3D convolutions on voxel representation with a multi-view approach.

Network	Reference	Framework	In
Voxel			
VoxNet	Maturana and Scherer [2015]	TensorFlow	No
ORION	Sedaghat et al. [2016a]	Caffe	No
FusionNet	Hegde and Zadeh [2016]	Not available	No
VRN	Brock et al. [2016]	Theano	Yes
O-CNN	Wang et al. [2017]	Caffe	Yes
AO-CNN	Wang et al. [2018a]	Caffe	Yes
Multi-view			
VGG-voting	Simonyan and Zisserman [2014]	TensorFlow	Yes
MVCNN	Su et al. [2015]	TensorFlow	Yes
MVCNN2	Su et al. [2018b]	PyTorch	Yes
RotationNet	Kanezaki et al. [2018]	Caffe	Yes
Seq2Seq	Zhizhong et al. [2018b]	TensorFlow	Yes
Point cloud			
PointNet	Qi et al. [2016a]	TensorFlow	Yes
PointNet++	Qi et al. [2017b]	TensorFlow	Yes
SO-Net	Li et al. [2018]	PyTorch	Yes
KD-Net	Klokov and Lempitsky [2017a]	Theano	Yes
GraphNet	Dominguez et al. [2018]	TensorFlow	No

Table 3.1: List of the examined neural networks. The table gives a reference to the original paper, the framework used in publicly available code and whether or not we included the network in our testing.

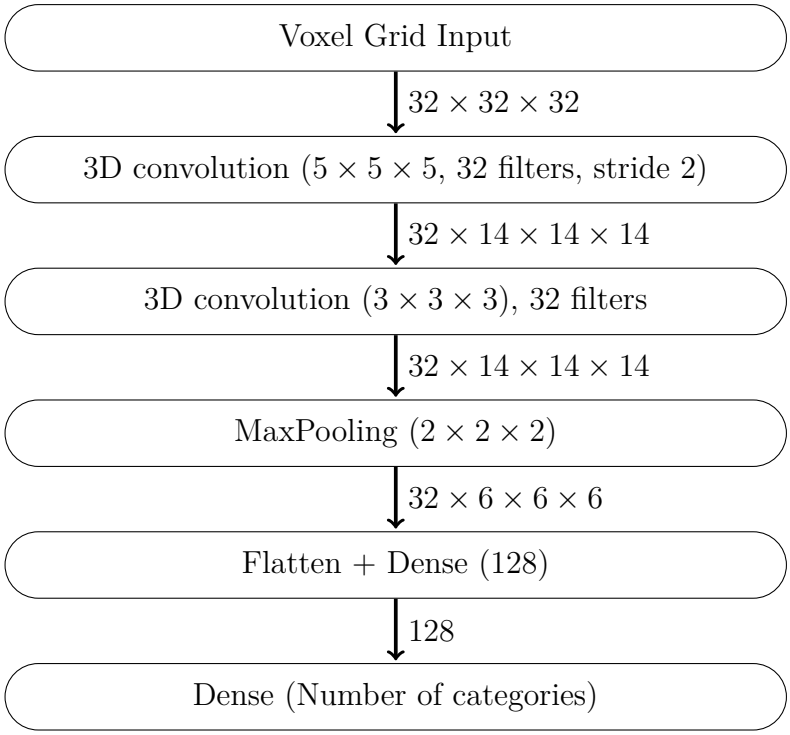


Figure 3.1: VoxNet architecture

3.1.2 Voxception Residual Network

Voxception Residual Network (Brock et al. [2016]) is inspired by deep residual convolutional networks for image recognition which are the state of the art approach for this task. It uses Inception-style modules (Szegedy et al. [2016]), batch normalization (Section 2.1.5), residual connections (Section 2.1.6) and stochastic network depth (Huang et al. [2016]). The Voxception network consists of several sequential voxception modules. These modules should enable for information to propagate through the network through many possible "pathways", while still maintaining simplicity and efficiency.

For example, one of the basic blocks concatenates the result of a $3 \times 3 \times 3$ and the a $1 \times 1 \times 1$ convolution, so the network can learn which of these filters to apply. A diagram of this block with added residual connection is shown in Figure 3.2. There are several types of the so called Voxception blocks with residual connections with pre-activation[1] employed in the network.

The best performing architecture consists of Voxception blocks as well as downsampling blocks which enable the network to choose the best downsampling methods (e.g. convolutions with stride bigger than one, or pooling). The deepest path through the network is 45 layers, and the shallowest path (assuming all droppable non-residual paths are dropped) is 8 layers deep. Figure 3.3 shows a diagram illustrating the Voxception Residual Network architecture. The model is quite big and slow to train, authors report one epoch taking around six hours on a single Titan X GPU for ten thousand training examples, which is in line with our results. Voxel grid resolution of 32^3 is used. The network is trained using 24 rotations of each input instance along the vertical axis and using binary voxel

[1] Nonlinearity is used before the addition.

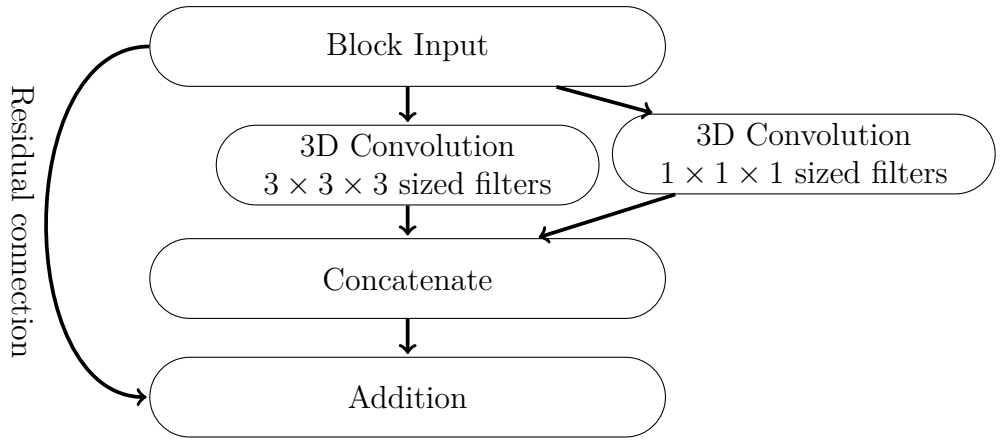


Figure 3.2: Example of a simple residual block

representation but with binary voxel range $\{-1, 5\}$ instead $\{0, 1\}$ to encourage the network to pay more attention to positive entries.

As there is not much new research done in the area of voxel based classification we chose a single Voxel Residual Network as a representative of this category. It still achieves accuracy comparable to the latest networks and has publicly available code. It is implemented in Theano with Lasagne (Brock [2016]) and offers several models to train and make ensemble from. We opted for only one of these models as it is very time demanding to train even a single network. The model chosen is the model reported by authors as the best one and described in detail in the original paper. This approach still represents the state of the art in voxel based classification as ensemble of similar models reports 95.54% accuracy on ModelNet40 dataset which remains one of the highest reported.

3.1.3 Octree and Adaptive Octree Networks

The Octree-based Convolutional Neural Network (Wang et al. [2017]) uses another data structure for representing 3D data – an octree (Meagher [1980]). An octree is a tree where each node has exactly eight children, partitioning the space into finer and finer cubes. This basically means voxelization of the 3D model, but in this case only voxels on the object boundaries are considered. This can be implemented efficiently and represented in a format suitable for GPU computation. In each leaf node a normal vector of the surface is stored. Authors then present an efficient way of performing convolutions on octrees and construct a hierarchical structure of shared layers for individual levels of the tree. The computation proceeds from the finest leaf octants and continues upwards to the root of the tree. This approach gives good results but the octrees are of a fixed maximum depth and therefore can waste memory on flat regions where a simple planar approximation would be sufficient. This problem is solved by Adaptive Octree (Wang et al. [2018a]) representation which uses such planar patches as a representation in leaf nodes. Therefore flat areas of the original mesh can be represented by a simple leaf on a higher level of a tree, while more complex areas are subdivided into finer details. Several (12 in our case) rotations of the 3D model are used during both training and evaluation to achieve better results. Authors

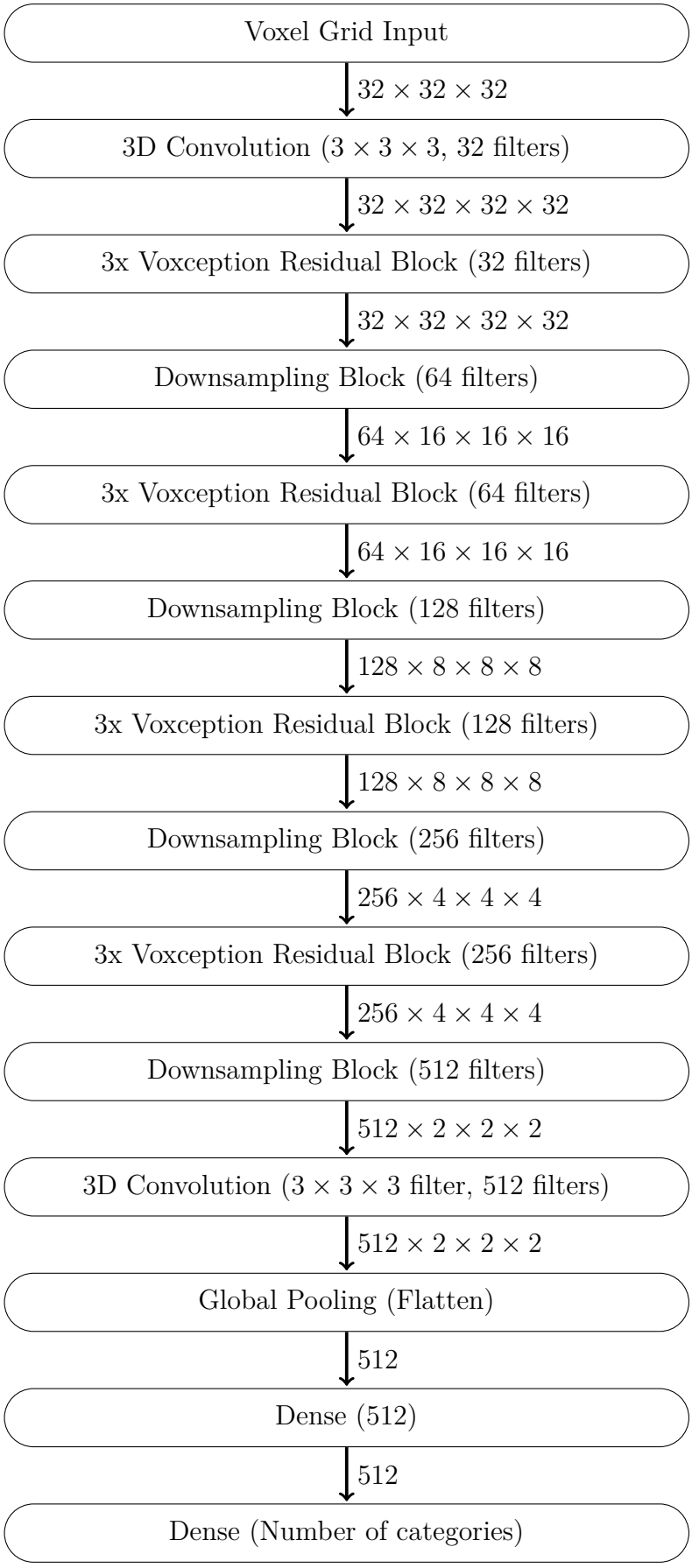


Figure 3.3: The Voxception Residual Network (VRN) architecture.

offer an implementation of both networks in Caffe as well as tools for converting mesh data to octrees and adaptive octrees (Wang et al. [2018b]).

3.2 Multi-view-based Neural Networks

Another approach, harnessing the power of 2D convolutions and huge image datasets, are the so called multi-view neural networks. A general setup of multi-view networks is as follows: they use rendered images of the 3D model from different angles as an input, these views are then passed to some pre-trained image processing network, and finally some technique for combining features from different views is employed. Such techniques range from simple pooling across the views to employing recurrent neural network to process them as a sequence.

Multi-view approaches can be considered the state of the art in this area as they achieve excellent results. For a fair comparison of these methods we use the same sets of images and twelve views of each 3D model rotated around the vertical axis.

3.2.1 Multi-view Convolutional Networks

For training a multi-view based network we need to fine-tune some already pre-trained image recognition network. We use two different networks: the smaller and older AlexNet (Krizhevsky et al. [2012]) and the state of the art deep network VGG (Simonyan and Zisserman [2014]). This offers a simple method of 3D classification; we train the image network on the views of a rotated 3D model without any regard for the multi-view nature of the dataset. During evaluation we perform voting across the views. We chose VGG for this task as it performs better on image recognition tasks. We use a publicly available implementation of VGG (machrisaa [2017]) with 19 weighted layers in Tensorflow. As we shall see later, even this simple approach yields results comparable with the most sophisticated networks.

The first multi-view approach to appear (Su et al. [2015]) uses shared convolutional layers individual views, then uses max pooling across the views to combine the features. Resulting features are fed to another convolutional network and then classified. We chose to test this approach as it is the first multi-view approach to achieve good results and it is simple enough to serve as a baseline for similar approaches. From several available implementations of this network we have chosen a Tensorflow implementation (Lee [2016]). In this case, we use AlexNet as the pre-trained image network, which is recommended by the authors of the code and supported in the code structure. Figure 3.4 shows a diagram illustrating the multi-view architecture.

The revisited but similar approach is used by Su et al. [2018b], which divides the training phase into two stages. In the first stage the network is trained only using one view at a time and later, during the second phase, pooling across the views is employed. Authors also explore different pre-trained image network architectures and different image rendering techniques, significantly improving accuracy of this method. It uses a pre-trained VGG convolutional network as its base. It offers a publicly available implementation in PyTorch (Su et al. [2018a])

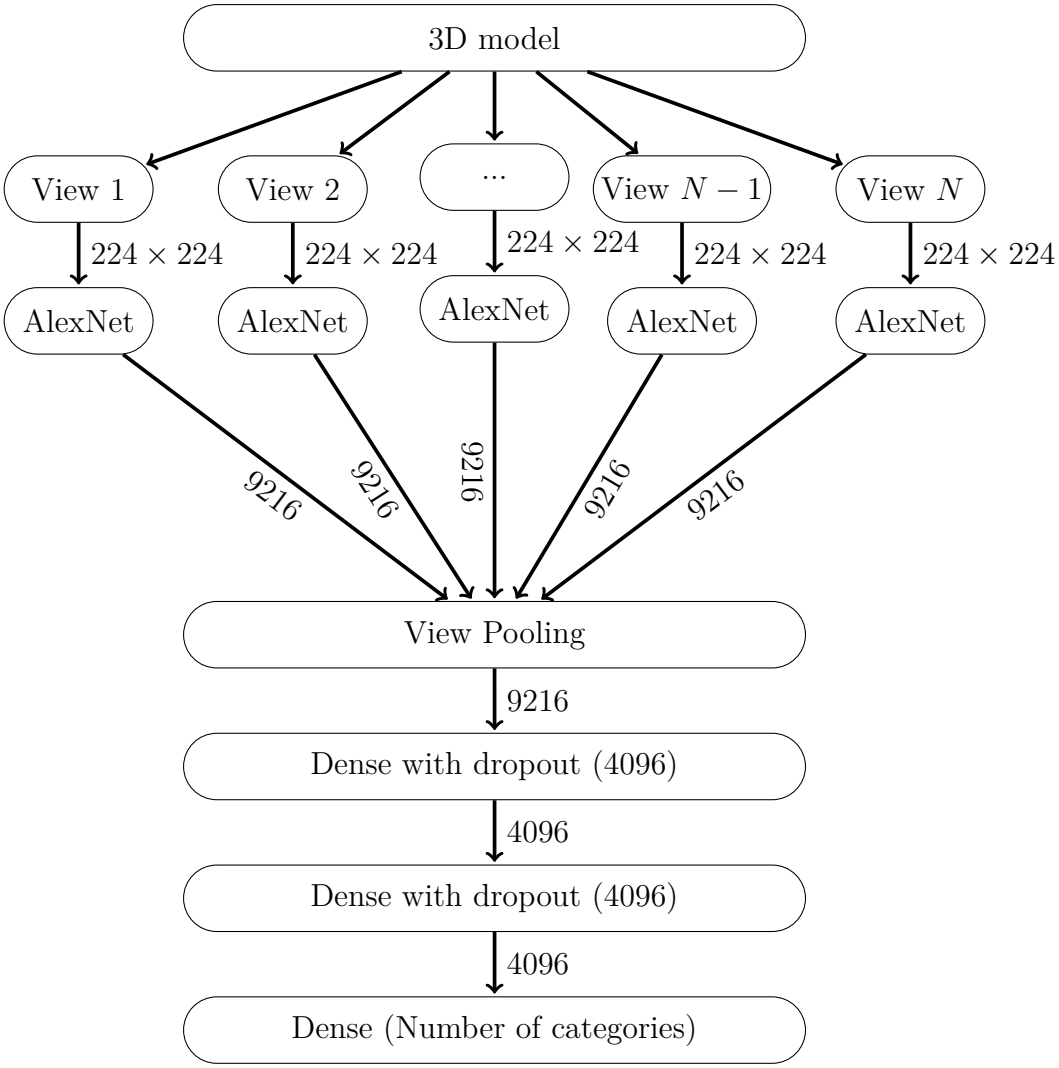


Figure 3.4: Multi-view architecture, as used in Lee [2016]

which we have chosen to test as it promises some of the best accuracies achieved on ModelNet40 so far.

3.2.2 RotationNet

RotationNet (Kanezaki et al. [2018]) reports the highest achieved accuracy on the ModelNet40 dataset so it is of particular interest to us. It combines the multi-view classification with an unsupervised pose estimation task. Unlike other multi-view networks, it does not provide information about the position of the viewpoints to the network, i.e., they can be rotated arbitrarily, hence the name of the network. To achieve this, a new category is added to the set of original classification categories. The meaning of this category is "this is not the correct viewpoint". All the possible rotations of the viewpoints are tried and the most probable according to predicted categories is chosen. Authors offer two implementations, one in Caffe (Kanezaki [2018]), and another one in PyTorch. We have chosen to test this network as it reports very high accuracy on ModelNet40 and inherently contains pose estimation, which can be useful for future work.

3.2.3 Sequential Views to Sequential Labels

Sequential Views to Sequential Labels network (Zhizhong et al. [2018b]) employs recurrent neural networks and treats multiple views as a sequence of images. It uses a classic encoder-decoder architecture. In order to do this it does not treat its output as a single vector but as a sequence of labels. It uses a pre-trained convolutional network, fine-tuned on the single-image classification task. We opted for the VGG network described above. The last fully connected layer of size 4096 is used as a feature vector for single views and is fed into the encoder as a sequence. Both the encoder and decoder consist of GRU cells (Section 2.1.3) and attention is used. An implementation in TensorFlow is available (Zhizhong et al. [2018a]) and we used it to test this network.

3.3 Point-cloud-based Neural Networks

An altogether different representation of spatial data is a point cloud. A point cloud is an unordered set of points in Euclidean space, representing the surface of the object. It is a natural output format of laser scanning devices used by robots or autonomous cars and also in medical scanning. We can easily construct a point cloud from a mesh by sampling its faces. Point clouds are neither structured nor ordered as voxels or images are, which poses a problem to neural networks.

3.3.1 PointNet and PointNet++

The first network to successfully overcome all the difficulties of processing unordered point clouds was PointNet (Qi et al. [2016a]). Its main idea lies in using only symmetric functions, i.e., functions for which the order of arguments does not matter. Each point is processed independently by a series of multi-layer perceptrons sharing their weights. Then a global feature vector is constructed using maximum pooling, which is a symmetric function. Another important feature of PointNet are learn-able geometric transformations which ensure some invariance to rotation or jittering (random small translations) of the input point cloud. Rotation and jittering are also used as data augmentation during the training. Figure 3.5 shows a diagram of the PointNet architecture. Although PointNet achieves reasonable results it does not provide any mechanism for learning local features.

PointNet++ (Qi et al. [2017b]) presents a hierarchical structure inspired by convolutional neural networks which solves the problem of extracting local features. It clusters close sets of points together and runs the original PointNet on such neighborhoods. For this purpose iterative farthest point sampling and multi-resolution grouping (which ensures good representation for regions with different density) are used. Thusly obtained local features are represented by the centroid of the original neighborhood and clustered again in a hierarchical manner. Finally, a classic fully connected layer is employed to extract global features and classification. Several (12 in our case) rotations of the 3D model are used during both training and evaluation to achieve better results. An implementation of both PointNet and PointNet++ is available in TensorFlow (Qi et al. [2016b,

2017a]) and we tested both of them, as they achieve reasonable accuracy, promise better scalability, and are considerably faster than some other methods.

3.3.2 Self-Organizing Network

The PointNet++ architecture lacks the ability to reveal the spatial distribution of the input point cloud during the hierarchical feature extraction. Self-Organizing Network (Li et al. [2018]) solves this problem by constructing a *self organizing map* (SOM) which represents the point cloud better than simple centroids used in PointNet++. Each point of the original point cloud is associated with k nearest SOM nodes and for each such node a mini point cloud is constructed. This also ensures that the mini point clouds are overlapping which was shown to be a key feature. These mini point clouds are processed by a series of fully connected layers similar to the original PointNet. This process yields a local feature vector for each of the original SOM nodes, which are then used for constructing a global feature vector by means of max pooling across the nodes. An implementation in PyTorch (Li [2018]) is available, which contains also code for creating self organizing maps from point clouds. We have chosen to test this network as it seems to offer a significant improvement for a cost of only quick data preprocessing.

3.3.3 KD-Network

A kd-tree (Bentley [1975]) is a data structure suitable for storing and searching in a set of points of higher dimension. Its 3D variant is used as an input format for KD-Network (Klokov and Lempitsky [2017a]). First a kd-tree is constructed over a point cloud. The tree is then fed to a series of fully connected layers in a recursive manner starting in the leaf nodes and continuing to the root, where the global feature vector is extracted and used for classification. Weights of the fully connected layers are shared by nodes on the same level of the tree, where the tree is split along the same coordinate.

During training it uses several geometric perturbations as data augmentation as well as randomized kd-tree construction. This approach can process raw point clouds but requires heavy preprocessing when constructing the kd-trees. An implementation in Theano with Lasagne (Klokov and Lempitsky [2017b]) is supplied by the authors, which also provides a framework for kd-tree construction from a point cloud.

3.3.4 Graph Based Convolutional Network

For the completeness' sake we mention a graph based approach by Dominguez et al. [2018], which constructs a graph from a point cloud. Vertices of the graph are the original points and edges are constructed to the six nearest neighbors and sorted by their direction by an arbitrary sorting. Then special graph convolutions are applied repeatedly simplifying the structure of the graph and extracting local features. This approach is interesting from the theoretical standpoint but the training is very slow and it does not achieve state of the art results. A TensorFlow implementation of this network is available (Dominguez [2018]).

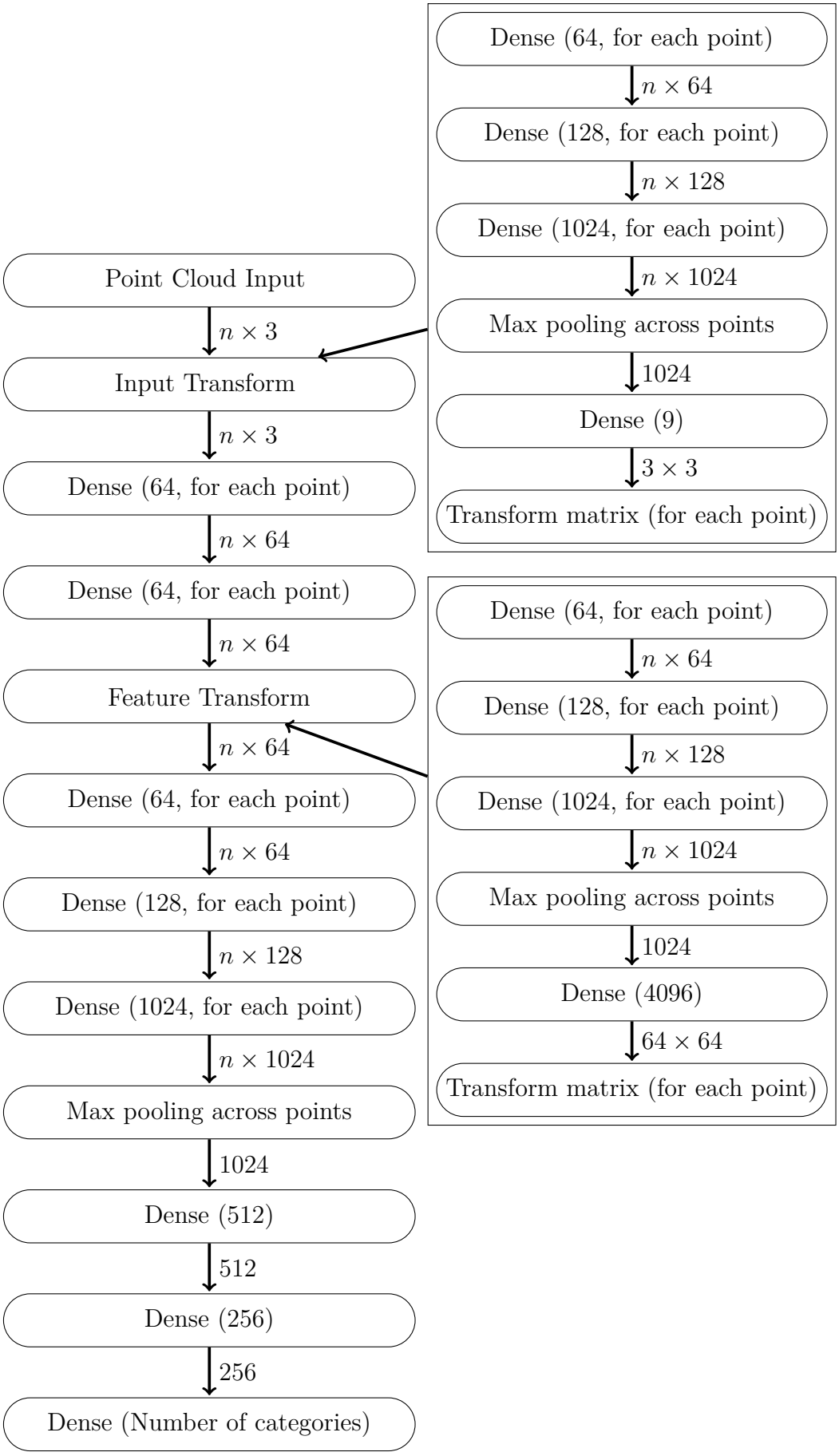


Figure 3.5: PointNet architecture. Layers labeled "for each point" are applied for each point separatly with same weights. n is number of input points.

4. Methods

We begin this chapter by introducing the data used for our experiments as well as some additional sources for further research. We continue by briefly discussing the methods of converting 3D meshes to various other representations and we end with presenting our choice of software.

4.1 Datasets

In this section we introduce the datasets of 3D models which we used or considered to use for training and testing.

4.1.1 ModelNet40

ModelNet (Wu et al. [2014]) is one of the most well-known and commonly used datasets containing annotated 3D models in a mesh format. It was developed by a team at Princeton University. Its subset, called ModelNet40, is used as a benchmark for testing different approaches. We therefore decided to use this dataset as a main focus for our evaluations. ModelNet40 contains forty different categories and 12,311 individual models. The dataset has an official split to training and testing subsets, which we adhere to in all cases. The test set contains 2,648 models and is never used for training. Figure 4.1 shows examples of models in ModelNet40.

The models in original ModelNet40 are not aligned and have widely different scales. Therefore, when preprocessing the data for neural networks, we rescale all models to fit a unit sphere and we use a manually aligned version of the dataset (Sedaghat et al. [2016a]). Also the categories are not equally populated. For example there are over 700 airplane models and only over 100 wardrobe models. The exact numbers of models in particular categories can be found in Table 4.1. ModelNet40 contains files in .off format so our scripts have to be able to read this particular format. The dataset is available to download for academic purposes.

4.1.2 ShapeNetCore

ShapeNet (Chang et al. [2015]) is an ongoing effort to establish a richly-annotated, large-scale dataset of 3D shapes. ShapeNet is a collaborative effort between researchers at Princeton, Stanford, and Toyota Technological Institute at Chicago. We used its subset called ShapeNetCore, which contains 51,209 individual models in 55 categories. There is also an official split to training, test, and validation sets. However, this split does not contain all models and is not divided uniformly. We therefore decided to construct our own split – 80% of models in each category is assigned to the training set and the rest to the test set. By doing this we obtained a training set with 40,939 models and a test set with 10,270 models.

Table 4.2 lists the exact numbers of models in particular categories. During our exploration of the dataset we noticed that some models are assigned to more than one category so we were forced to choose one of them somewhat arbitrarily.

Category	Train	Test	Category	Train	Test
airplane	626	100	laptop	149	20
bathtub	106	50	mantel	284	100
bed	515	100	monitor	465	100
bench	173	20	night stand	200	86
bookshelf	572	100	person	88	20
bottle	335	100	piano	231	100
bowl	64	20	plant	240	100
car	197	100	radio	104	20
chair	889	100	range hood	115	100
cone	167	20	sink	128	20
cup	79	20	sofa	680	100
curtain	138	20	stairs	124	20
desk	200	86	stool	90	20
door	109	20	table	392	100
dresser	200	86	tent	163	20
flower pot	149	20	toilet	344	100
glass box	171	100	tv stand	267	100
guitar	155	100	vase	475	100
keyboard	145	20	wardrobe	87	20
lamp	124	20	xbox	103	20
			Total	**9843**	**2468**

Table 4.1: List of ModelNet40 categories and the number of training and test models in each category.

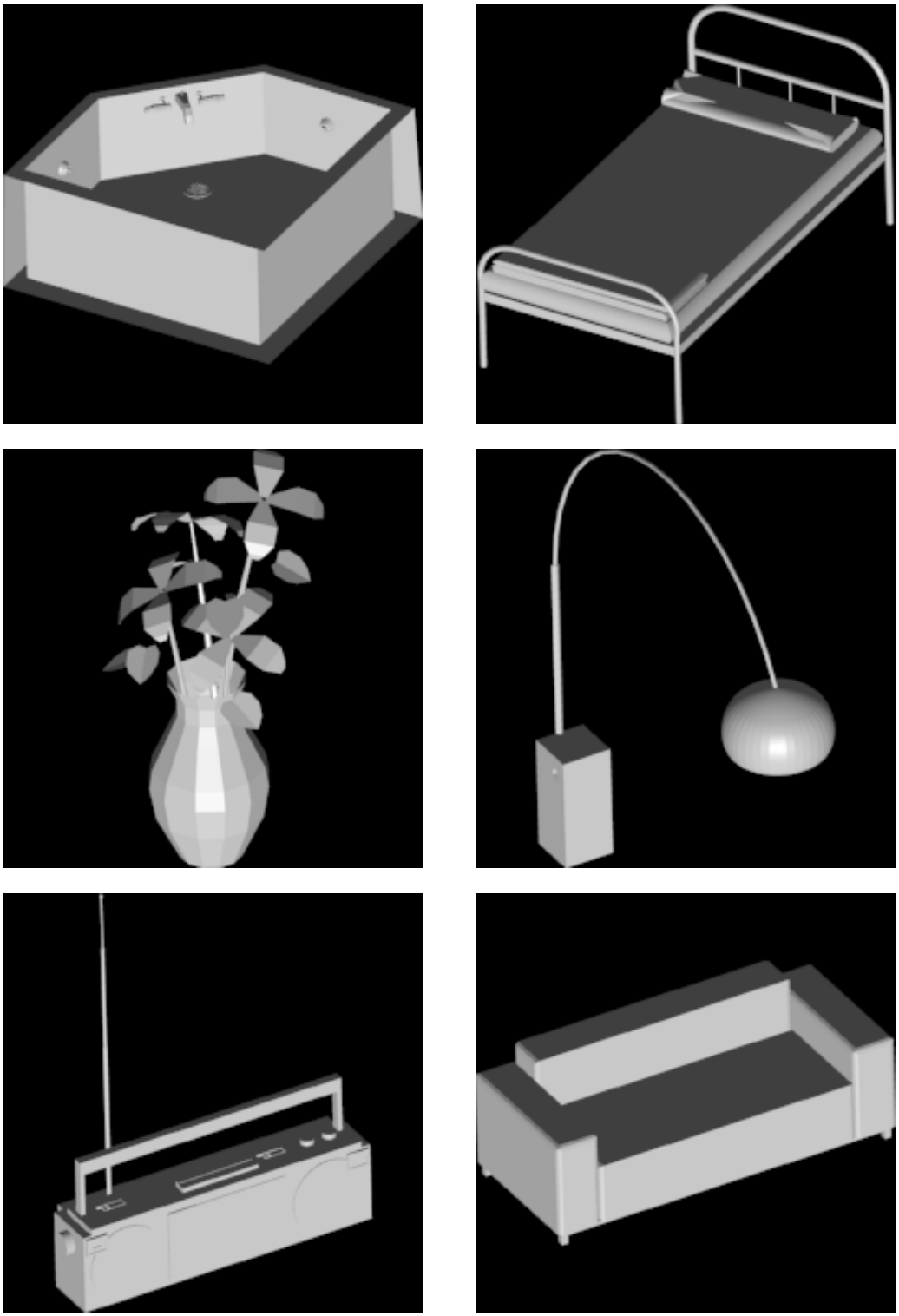


Figure 4.1: Illustration of models in ModelNet40 (bathtub, bed, flower pot, lamp, radio and sofa)

Our final split into sets and categories can be found in the *shapenetsplit.csv* file in the electronic attachments.

All models in ShapeNetCore are already aligned and scaled to fit a unit sphere. The categories are not distributed equally at all as you can see from the table.

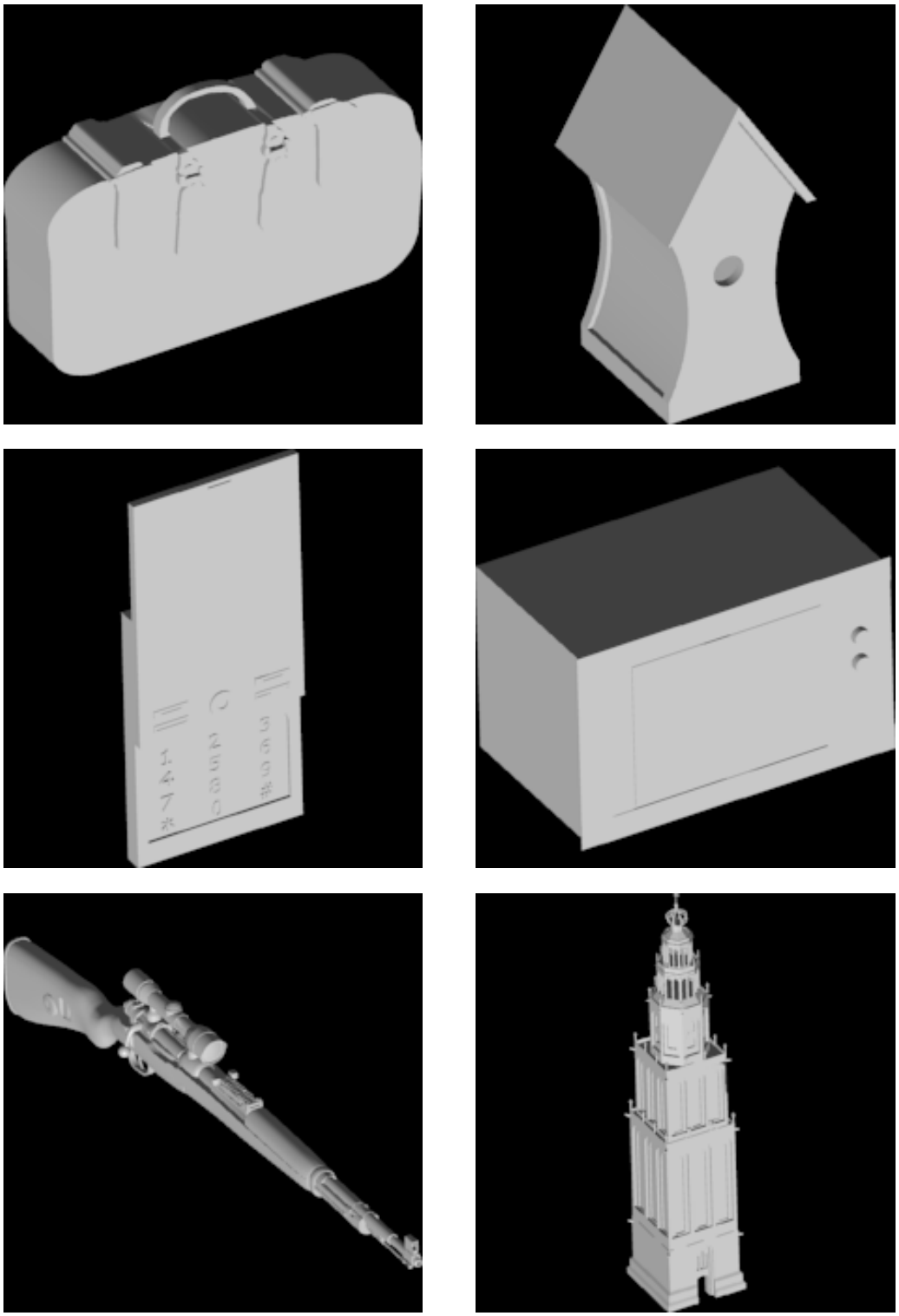


Figure 4.2: Illustration of models in ShapeNetCore (bag, birdhouse, cellular telephone, microwave, rifle, tower)

The dataset is also freely available to download for academic purposes. Figure 4.2 shows examples of models in ShapeNetCore.

Category	Train	Test	Category	Train	Test
airplane	3235	810	jar	466	114
ashcan	275	68	knife	339	85
bag	66	17	lamp	1853	464
basket	82	21	laptop	360	91
bathtub	684	172	loudspeaker	1274	319
bed	184	49	mailbox	74	19
bench	1451	362	microphone	53	14
birdhouse	58	15	microwave	122	30
bookshelf	362	90	motorcycle	269	68
bottle	395	102	mug	171	43
bowl	146	39	piano	191	48
bus	751	188	pillow	76	20
cabinet	1247	315	pistol	247	60
camera	90	23	pot	439	110
can	84	21	printer	132	33
cap	44	12	remote control	52	14
car	2811	703	rifle	1864	467
cellular telephone	665	166	rocket	68	17
chair	5391	1354	skateboard	121	31
clock	521	130	sofa	2406	603
computer keyboard	51	13	stove	174	44
dishwasher	74	19	table	6702	1676
display	874	218	telephone	206	52
earphone	58	15	tower	98	25
faucet	593	149	train	311	78
file	230	59	vessel	1550	388
guitar	637	160	washer	133	34
helmet	129	33	**Total**	**40939**	**10270**

Table 4.2: List of ShapeNetCore categories and the number of training and test models in each category.

4.1.3 Other 3D Datasets

In this section we mention several publicly available datasets containing 3D models which can be used for further research.

Both ModelNet and ShapeNet contain many more models than the standardized subsets we used for our evaluation. Therefore there is an option to download the whole datasets or to construct custom subsets.

A Large Dataset of Object Scans (Choi et al. [2016]) is a dataset focusing on video scan to 3D model reconstruction but we suppose it can be used for learning classification as well.

ObjectNet3D (Xiang et al. [2016]) focuses on image to 3D model reconstruction and contains a large number of 3D models that can be used for classification training.

SUNCG dataset (Song et al. [2017]) contains entire indoor scenes but is annotated on the level of single objects and therefore can be parsed and used for classification.

SceneNN (Hua et al. [2016]) dataset contains a large number of scenes which are richly annotated and can be split into single objects.

4.2 Data Conversion

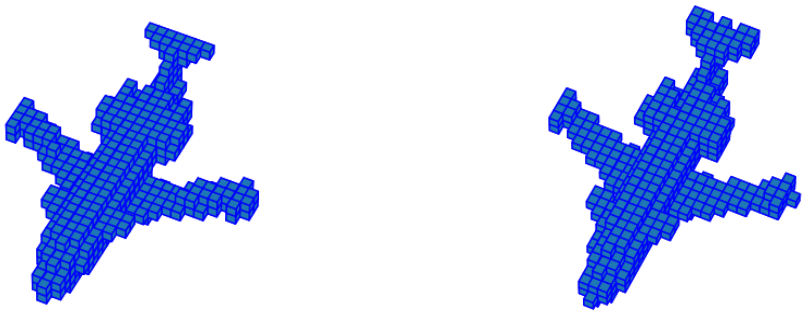
As mentioned in previous chapters, mesh files, in which most existing 3D models are saved, are not suitable for direct processing by neural networks. Therefore we have to be able to convert meshes to voxels, images, and point clouds.

4.2.1 Mesh to Voxels

In order to use voxel-based systems we need to convert mesh files to voxel occupancy grids. For this purpose we have chosen the OpenVDB library (Museth et al. [2013]), which is free, open-source and offers Python scripting. OpenVDB provides voxelization as one of its core functions, implemented in C++. We supply Python scripts for voxelization of ModelNet40 and ShapeNetCore datasets using Python multiprocessing to parallelize the computation. Still, it can take several hours to process the whole dataset as we need to voxelize multiple rotations for each model.

4.2.2 Mesh to Images

For multi-view-based neural network, we have to be able to render images taken from arbitrary viewpoints of a 3D mesh. First, we tried to replicate results used by Su et al. [2015] and we used PBRT (Pharr and Humphreys [2010]), physically based rendering software with publicly available code. This turned out to be a plausible approach. We also used the original scripts using Phong shading (Bishop and Weimer [1986]) implemented in Blender to render the images. Later in our research we found Blender scripts from Su et al. [2018b]. They provide two different rendering options – shaded images and depth images. These achieve better accuracy than both ours and Phong shaded images and the rendering is considerably faster. In our framework we provide all four approaches implemented



(a) Original voxel representation provided by authors of VRN

(b) Our voxelization using OpenVDB

Figure 4.3: Illustration of voxel representation

with Python scripts and multiprocessing support. Figure 4.4 shows one of the airplane models rendered by the four different scripts.

4.2.3 Mesh to Point Cloud

For the use of point-cloud-based neural networks we have to construct a point cloud from a 3D mesh. This is a much more straightforward problem than the conversions described above. A point cloud is created by random sampling from the polygons forming the mesh. First, a polygon is selected with a probability proportional to the area of that polygon. Then a random point is sampled within the selected polygon by generating random barycentric coordinates. We provide a Python script with support for multiprocessing and this is sufficiently fast for our purposes. When we failed to achieve the reported results of the original papers, we inspected the data provided by Qi et al. [2016b] and it seemed to be more regular than our uniformly sampled points. Authors comment on their sampling method in the following way: "*We uniformly sample 1024 points on mesh faces according to face area and normalize them into a unit sphere.*" Therefore we tried two more sampling methods to replicate the original PointNet data. We employed Lloyd's algorithm (Lloyd [1982]) which samples the mesh very regularly using the Point Cloud Utils library (Williams [2019]). We also tried to use a low-discrepancy sequence sampling which should give more regular but seemingly random samples. For this we used a Sobol sequence (Sobol' [1967]). Figure 4.5 shows a visual comparison of our methods as well as original PointNet point cloud, as can be seen we did not manage to replicate the desired look of the point cloud. Nonetheless as discussed later, the sampling method did not have a significant impact on the classification accuracy.

(a) Phong shading in Blender

(b) Our PBRT rendering

(c) Depth image in Blender

(d) Shaded image in Blender

Figure 4.4: Illustration of different image representations

4.3 Technical Setup

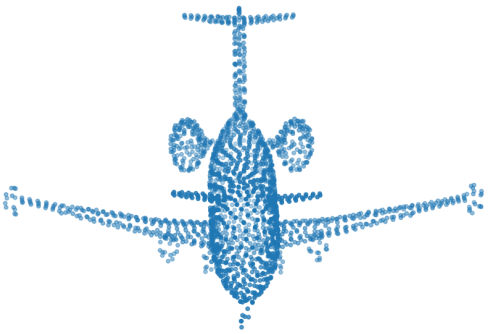
This section provides a brief summary of software choices we made. For information about the prerequisites and structure of our framework, please consult the manual (Appendix C).

As one of our main goals is to provide the academic community with easy-to-run code, we opted for a solution using Docker (Merkel [2014]). Docker is a program used to run software packages called containers. Containers are isolated bundles of software, libraries, and configurations. The specification of a container is called an image. An image is defined by a Dockerfile which is a text file, allowing automatic installation of all dependencies, setting up configurations, etc. Every neural network and data conversion package is thus a completely independent piece of software, which can be run almost without any prerequisites. We consider this to be one of the main contributions of our work.

As all the machine learning frameworks we encountered support handling by Python scripts and Python is the most commonly used programming language in machine learning and artificial intelligence, we naturally use it for most of

(a) Original point cloud provided by the authors of PointNet (2048 points)

(b) Uniform sampling (2048 points)



(c) Lloyd sampling (2048 points)

(d) Sobol sequence sampling (2048 points)

Figure 4.5: Illustration of point cloud representations

our code. We also preferred libraries for data conversion which support Python. Some of the neural networks are implemented in such a way that they accept a purely pythonic file format as their input. A library not supporting Python would require one more data conversion step.

We currently support only Linux, but Docker can be run on Windows as well and we believe that our framework can be extended to run on Windows without great difficulties.

5. Experiments and Results

In this chapter we describe the setup and results of our experiments. We introduce the hardware, methods of training and discuss our results.

5.1 Hardware

We conducted all our experiments on the same machine running Linux operating system. It was equipped with two AMD RYZEN Threadripper 1950X CPU units (16 cores each) and 128 GB of RAM. However, we did not use this directly as it is much more efficient to train neural networks on GPUs. We had four NVIDIA GeForce GTX 1080 Ti GPUs at our disposal, however we conducted our experiments only using one of them. We opted for this as we wanted to have some fair comparison of training time and not all the frameworks support running on multiple GPUs, or at least not without some heavy modifications of the code.

5.2 Accuracy

Accuracy is a simple metric computed as the number of correctly classified examples divided by the total number of all examples. It is usually given in percents and is a standard for classification task evaluation. As is the case with our datasets, when the examples are not distributed equally, accuracy can be skewed by categories containing more examples. Therefore we additionally compute *average class accuracy*, which is computed as $\frac{1}{N}\sum_{i=1}^{N}\frac{\text{correctly classified in category i}}{\text{total in category i}}$, where N is the number of categories. We believe that this number is somewhat more descriptive.

5.3 Testing on the Artificial Datasets

To compare the performance of various methods, it is necessary to use some standardized datasets. We chose ModelNet40 and ShapeNetCore for this purpose; their description can be found in Section 4.1.

As the main goal of this thesis is to explore the possibilities of using the previously introduced neural networks in practice, we are not interested in chasing couple of percents on artificial datasets. Rather we focus on general performance and convenience of use. Thus we did not spend huge amounts of time on hyperparameter tuning in order to increase the accuracy as this would lead to overtraining of hyperparameters on the test set. Although this problem could be solved by employing a validation set, standard ModelNet40 split does not support this option and hyperparameter tuning would be cumbersome to do in a real-world setting. So we opted for using the hyperparameters described in the original papers if available or used the default setting in the original code. More information about the hyperparameter setting can be found in Appendix A.

Another important decision was to choose the stopping condition of training. As each of the networks takes different time to complete one training epoch (a period during which each training example is presented once) it would not

Network	Reported Accuracy	Reported Class Acc.	Measured Accuracy	Measured Class Acc.
Voxel				
VoxNet	83.00 %	–	–	–
ORION	89.70 %	–	–	–
FusionNet	90.80 %	–	–	–
VRN	91.33 %	–	90.32 %	88.00 %
O-CNN	90.60 %	–	88.29 %	83.09 %
AO-CNN	90.50 %	–	91.08 %	87.97 %
Multi-view				
VGG	–	–	90.86 %	88.00 %
MVCNN	90.10 %	–	88.83 %	86.24 %
MVCNN2	95.00 %	92.40 %	90.64 %	89.13 %
RotationNet	97.37 %	–	92.12 %	89.86 %
Seq2Seq	93.31 %	–	91.26 %	88.54 %
Point cloud				
PointNet	89.20 %	86.20 %	86.60 %	84.00 %
PointNet++	91.90 %	–	89.00 %	85.28 %
SO-Net	93.40 %	87.30 %	89.00 %	85.50 %
KD-Net	91.80 %	88.50 %	88.10 %	83.90 %
GraphNet	91.13 %	–	–	–

Table 5.1: List of ModelNet40 accuracies. The first two columns give reported accuracy and reported average class accuracy, if available. The last two columns present our results.

be fair to set some fixed number of epochs. Therefore we stop the training after convergence, i.e., when the value of the loss function on the test set is not improving for several epochs. Some of the networks do not really stabilize in test set, so we wait for the stabilization of the training loss and report the accuracy of the test set averaged over the last ten epochs. More detailed statistics can be found in Table B.1 and Table B.2 for ModelNet40 and ShapenetCore respectively.

5.3.1 Time and Memory Requirements

Table 5.2 shows approximate times of training. These times are not conclusive, as training time depends on the used hardware. However, the differences among the types of networks are considerable. In general we can say that the point-cloud-based networks are quite fast, processing tens of examples each second. Octree and Adaptive-octree networks proved to be the most efficient – processing hundred and two hundred examples per second, respectively. Although multi-view-based networks achieve better accuracies, they are considerably slower. Depending on the network, the simplest architecture can process tens of examples per second,

Network	Epochs	Total (hours)	One epoch (minutes)	Examples per second
Voxel				
VRN	20	120	394	0.5
O-CNN	200	4	1.15	140
AO-CNN	200	2	0.65	250
Multi-view				
VGG	60	20	25	6
MVCNN	200	8	2.33	70
MVCNN2	60	14	13.33	12
RotationNet	200	8	2.73	60
Seq2Seq	300	1	0.25	650
Point cloud				
PointNet	200	10	2.22	70
PointNet++	200	4	1.07	150
SO-Net	400	6	0.9	180
KD-Net	200	8	2.61	60

Table 5.2: Table of approximate training times on ModelNet40.

deeper networks require more time, processing around ten examples per second. The Seq2Seq network seems very fast and it indeed is, but the fact that the separate fine-tuning of VGG is required to use this network, must be considered. The slowest by far is voxel-based VRN which needs two seconds for a single training example and requires approximately a week to train.

As for the memory requirements, Table 5.3 shows approximate sizes of the neural networks, which roughly correspond to the number of trainable parameters of the networks and also memory requirements during inference. Memory usage during training depends on the batch size of the network and size of the input.

5.3.2 Results on ModelNet40

We trained and tested all the networks introduced previously and all the input data variants on the ModelNet40 dataset (described in Section 4.1.1). The results we achieved with comparison to reported accuracies are listed in Table 5.1.

We present the detailed results of individual networks below. In this section by "parameters" we mean non-trainable hyperparameters such as learning rates, momentum value, number of training epochs, etc.

- **VRN (Voxception Residual Network)**

  The authors report an accuracy of 91.33% and we were able to achieve 90.32%. The original paper does not provide exact training parameters, so we used the default parameters of the code. Our lower accuracy be caused by using our own data, created by our own script which can be inferior

Network	Size of the model (MB)
Voxel	
VRN	50
O-CNN	2
AO-CNN	2
Multi-view	
VGG	500
MVCNN	800
MVCNN2	500
RotationNet	230
Seq2Seq	30
Point cloud	
PointNet	40
PointNet++	18
SO-Net	10
KD-Net	8

Table 5.3: Approximate sizes of saved models, roughly corresponding to the number of trainable parameters of the model.

to original voxelization. The authors provide some example data but not complete ModelNet40. We did not perform additional experiments because of the long training time of this network.

- **O-CNN and AO-CNN (Octree and Adaptive Octree convolutional neural networks)**

  The authors report accuracy of 90.6% and 90.5% for O-CNN and AO-CNN, respectively. We were able to achieve 88.29% and 91.08%. To prepare the data, we used scripts provided by the authors and they also state the exact training parameters.

- **Multi-view convolutional neural networks**

  According to Su et al. [2018b] the quality of the input images is not negligible in the case of multi-view-based approaches to 3D classification. We have tested these networks on four different sets of images. First, the original images provided by Su et al. [2015], then our own images rendered in PBRT and two variants of images rendered by scripts provided by Su et al. [2018b]. Detailed results can be found in Table 5.4. The technique of rendering the images is discussed in Section 4.2.2.

- **MVCNN (Multi-view Convolutional Neural Network)**

  The authors report accuracy of 90.1% on images rendered using Phong shading, but we achieved accuracy of only 83.99%. We were able to get higher accuracy of 88.83% on newer shaded images. However the authors

Network	**Accuracy**	**Class Accuracy**
VGG (PBRT)	87.93 %	84.99 %
VGG (Phong)	90.78 %	83.21 %
VGG (depth)	89.14 %	85.75 %
VGG (shaded)	90.78 %	88.00 %
MVCNN (PBRT)	87.62 %	85.54 %
MVCNN (Phong)	83.99 %	79.60 %
MVCNN (depth)	87.83 %	85.07 %
MVCNN (shaded)	88.83 %	86.24 %
MVCNN2 (PBRT)	90.26 %	88.44 %
MVCNN2 (Phong)	88.97 %	87.32 %
MVCNN2 (depth)	90.52 %	89.13 %
MVCNN2 (shaded)	90.64 %	88.38 %
RotationNet (PBRT)	91.09 %	87.78 %
RotationNet (Phong)	89.22 %	86.85 %
RotationNet (depth)	91.46 %	88.62 %
RotationNet (shaded)	92.12 %	89.86 %
Seq2Seq (PBRT)	88.10 %	84.47 %
Seq2Seq (Phong)	89.09 %	82.78 %
Seq2Seq (depth)	89.08 %	85.97 %
Seq2Seq (shaded)	91.26 %	87.07 %

Table 5.4: Comparison of multi-view methods.

do not provide the training parameters in the paper so we had to default to parameters used in the code.

- **MVCNN2 (Multi-view Convolutional Neural Network 2)**

  The authors report the highest achieved accuracy of 95% on the shaded variant of the images. We achieved only 90.64% on the shaded images and 90.52% on the depth images. The training parameters are not provided in the paper, so we used the default values from the code.

- **RotationNet**

  The authors report accuracy of 97.37% using original Phong shaded images by Su et al. [2015] but we have been able to achieve only 92.12%. The reported value of 97.37% is however a maximum achieved over more training sessions. The authors report average accuracy of 93.70% using AlexNet as the pretrained image network, which is much closer to our result. They do not give all the training parameters in the paper but most of the important ones are mentioned.

- **SEQ2SEQ (Sequential Views to Sequential Labels)**

  The authors claim that they achieved accuracy of 92.5% only with VGG and voting across twelve views. We failed to replicate this and and achieved only 90.86%. Therefore we could not achieve the reported 93.31% accuracy of the Seq2Seq network but only maeasured 91.26%. However this means that the single view VGG with voting performed better than most of the more complex networks and the recurrent Seq2Seq did not bring any significant improvement.

- **PointNet and PointNet++**

  The authors report accuracy of 89.2% and 91.9% for PointNet and PointNet++ respectively. We were able to achieve only 86.60% and 89.00%. The authors provide most of the training parameters in their papers, so we use them along with the default values given in the code. We trained with point cloud data provided by the authors as well as our own converted data (sampling techniques are described in Section 4.2.3) . Full results of these tests are given in Table 5.5.

- **SO-NET (Self-Organizing Network for Point Cloud Analysis)**

  The authors report the highest achieved accuracy of 93.4% on an experiment with 5,000 input points. They also provide most of the training parameters in their paper. We were able to achieve only 88.90%. We used our own sampling of point cloud which can be the source of the discrepancy.

- **KD Network**

  The authors report the highest achieved accuracy of 91.8% and we have achieved only 88.10%. The authors do not provide all the training parameters so we defaulted to the values used in code. All the code to sample point clouds and construct the trees we used is supplied by the authors so this could not be the source of the disparity between the reported and measured accuracy.

Network	Accuracy	Class Accuracy
PointNet (original)	86.60 %	84.00 %
PointNet (uniform)	82.65 %	81.00 %
PointNet (lloyd)	85.25 %	78.97 %
PointNet (sobol)	85.87 %	80.92 %
PointNet++ (original)	89.00 %	85.28 %
PointNet++ (uniform)	87.57 %	85.30 %
PointNet++ (lloyd)	88.53 %	85.13 %
PointNet++ (sobol)	88.54 %	84.54 %

Table 5.5: Comparison of differently sampled point clouds in PointNet architectures.

5.3.3 Difficult Categories

In this section we explore the results in more detail – we discuss the accuracies on individual categories of ModelNet40. We show which categories are generally hard to recognize and which, on the other hand, did not cause any problems. We also give a brief account about pairs of categories which were mistaken most often and their illustrations. We believe that this information can be useful when designing custom category hierarchies.

We compute the accuracies per class and average these across all the trained networks. There are several categories which were almost always correctly classified and all the networks learned to recognize them fairly quickly. These categories are “airplane”, “laptop”, “guitar” and “keyboard”, which all achieved more than 99.00% average accuracy, 100% for most of the networks. Another successfully recognized categories are ‘car”, “bed”, “chair”, “monitor”, “person”, “bottle” and ”sofa”, all achieving more than 95% average accuracy. On the other hand the most difficult category by far was a “flower pot”, which was recognized only in 14.6 percent of cases. This is probably caused by the small number of examples of this category as well as very similar categories of “flower” and “vase”. Other generally difficult categories include “wardrobe”, “cup”, “night stand”, “bench” and “radio”, achieving no more than 75% accuracy. You can find average accuracies per category in Table 5.6

When we take a look at the pairs of categories most often mistaken one for the other, we find out that besides the above mentioned “flower pot”, the most mistakes were made classifying a “table” as a “desk”. This sounds quite reasonable, as the borderline between these categories is blurry even by human standards. The same applies for the category of “wardrobe”, which was commonly mistaken for “bookshelf”, “dresser” or even an “xbox”. The most commonly made mistakes can be found in Table 5.7.

Category	Accuracy	Cases	Category	Accuracy	Cases
flower pot	14.60%	20	door	90.40%	20
wardrobe	63.00%	20	piano	90.96%	100
cup	68.00%	20	range hood	91.00%	100
night stand	70.14%	86	tent	92.60%	20
bench	73.20%	20	bookshelf	93.04%	100
radio	73.20%	20	cone	93.60%	20
xbox	75.40%	20	glass box	93.92%	100
stool	76.40%	20	mantel	94.92%	100
table	76.44%	100	sofa	95.24%	100
dresser	76.70%	86	bottle	95.32%	100
vase	77.08%	100	person	96.20%	20
desk	79.35%	86	monitor	96.44%	100
sink	80.00%	20	chair	97.00%	100
tv stand	82.04%	100	bed	97.64%	100
bathtub	83.68%	50	car	98.56%	100
plant	83.68%	100	toilet	98.72%	100
lamp	83.80%	20	keyboard	99.00%	20
stairs	87.00%	20	guitar	99.48%	100
bowl	88.40%	20	laptop	99.60%	20
curtain	88.80%	20	airplane	99.72%	100

Table 5.6: Average accuracies per category of ModelNet40, sorted from worst to best.

Category	Mistaken For	Percentage
flower pot	plant	54.40%
flower pot	vase	22.60%
table	desk	18.92%
cup	vase	18.40%
night stand	dresser	14.60%
stool	chair	14.20%
wardrobe	bookshelf	14.00%
plant	flower pot	10.52%
dresser	night stand	10.42%
wardrobe	dresser	8.00%
bench	table	8.00%
desk	table	6.37%
wardrobe	xbox	6.20%
vase	cup	6.12%
cup	bowl	5.60%

Table 5.7: List of the most commonly made mistakes made by all the networks on ModelNet40.

Network	Accuracy	Class Accuracy
Voxel		
VRN	88.98 %	76.71 %
O-CNN	90.75 %	75.48 %
AO-CNN	91.33 %	77.14 %
Multi-view		
VGG (shaded)	91.90 %	81.00 %
MVCNN (shaded)	89.26 %	77.23 %
MVCNN2 (shaded)	92.22 %	85.56 %
RotationNet(shaded)	93.08 %	80.52 %
Seq2Seq (shaded)	92.35 %	81.77 %
Point cloud		
PointNet (uniform)	82.53 %	64.51 %
PointNet++ (uniform)	83.67 %	67.68 %
SO-Net (uniform)	87.87 %	73.36 %
KD-Net	81.23 %	53.84 %

Table 5.8: List of ShapeNetCore accuracies.

5.4 ShapeNetCore Results

We conducted several experiments on the ShapeNetCore dataset (Section 4.1.2). It is about five times bigger than the ModelNet40 and has 55 categories. The training of the networks therefore takes five times longer, therefore we did not train on all variants of inputs as with ModelNet40, but chose only those which were most successful previously. Table 5.8 shows the achieved accuracies on ShapeNetCore dataset. As can be seen from the table, the differences in measured accuracies across the networks are greater than in the case of ModelNet40, so the comparisons are clearer. The Multi-view networks, achieving around 92%, are performing much better than the point-cloud-based networks. Voxel-based VRN achieves reasonable accuracy of 88.98%, but one training epoch takes about 30 hours. It is possible that this result can be improved by training for longer time as we managed to train the network only for ten epochs. The octree-based networks are reasonably fast and achieve accuracy comparable to multi-view networks.

Also the differences between the accuracy and the average class accuracy is much higher. This is probably caused by the fact that the categories in ShapeNetCore dataset are not populated equally at all.

The detailed per category accuracies are given in Table 5.9. Categories “telephone” and “cellular phone” are mixed together, so models of cellular phones are to be found in “telephone” category and vice versa. This is the reason of 28% accuracy achieved in classifying the “telephone” category.

Category	Accuracy	Cases	Category	Accuracy	Cases
microphone	20.00%	14	bench	82.14%	362
telephone	28.37%	52	train	82.95%	78
tower	28.92%	25	can	85.30%	21
camera	46.09%	23	bathtub	86.16%	172
basket	46.42%	21	display	86.18%	218
file	48.08%	59	cabinet	86.58%	315
remote	48.57%	14	cap	86.67%	12
stove	53.41%	44	pistol	86.93%	60
rocket	54.71%	17	helmet	89.09%	33
mailbox	58.82%	19	keyboard	90.77%	13
jar	59.60%	114	lamp	90.86%	464
birdhouse	64.00%	15	faucet	92.00%	149
washer	64.41%	34	mug	92.09%	43
clock	64.49%	130	sofa	93.82%	603
pot	65.78%	110	pillow	94.00%	20
bag	67.06%	17	bus	94.36%	188
bookshelf	68.42%	90	table	95.29%	1676
dishwasher	68.95%	19	laptop	95.75%	91
microwave	71.00%	30	skateboard	96.45%	31
bowl	73.37%	39	chair	96.49%	1354
printer	73.57%	33	knife	96.82%	85
ashcan	74.40%	68	rifle	96.94%	467
bed	74.52%	49	vessel	97.24%	388
earphone	74.67%	15	airplane	97.56%	810
cellular	75.48%	166	guitar	98.94%	160
piano	78.75%	48	car	98.98%	703
bottle	79.48%	102	motorcycle	99.12%	68
loudspeaker	79.65%	319			

Table 5.9: Average accuracies per category of ShapeNetCore, sorted from worst to best.

6. Conclusions

In this section we give a brief summary of our work, discuss our results and offer some ideas for further research in the area of 3D classification.

6.1 Summary

In this work we explored different artificial neural-network-based systems for classification of 3D models. We performed a broad survey and gave a brief description of the individual systems. We also used the code published by the authors of the original publications and used it to successfully train and test several of these networks. In order to achieve this, we constructed a reasonably independent framework which enables to run and compare networks implemented in different machine learning frameworks. The conversions of 3D meshes to point clouds, voxel grids and multi-view images are also part of our framework. We have tested all the networks and data conversion on ModelNet40 and ShapeNetCore datasets so our framework is ready to be used on these and we believe it can be extended on similarly structured datasets without much difficulties.

As for our results, we generally failed to replicate the original results on ModelNet40. All the networks performed reasonably well, but we have achieved a couple percent less than the reported accuracy. This can be caused by the sensitivity of neural networks to hyperparameter tuning. We suppose that the authors of the original papers spent a considerable amount of time on making their networks achieve the best results possible. In general they do not describe the exact training setup in their publications and we did not have enough resources to perform an exhaustive hyperparameter search for all the networks. The relative performance of the networks was confirmed by our results; networks with higher reported accuracy generally performed better in our experiments as well. The differences are more significant with ShapeNetCore, multi-view networks outperforming the point-cloud-based networks.

The networks also differ greatly in the required training time, from weeks to several hours, and this can be a very important fact to consider when choosing an approach in practice. The voxel-based neural networks are very slow, multi-view-based networks are considerably faster and point-cloud-based and octree-based networks are the fastest.

6.2 Future Work

The research in the field of artificial intelligence in general and in the area of neural networks in particular is progressing very quickly. As we were working on this thesis there already appeared some new publications (Yu et al. [2018], You et al. [2018], Feng et al. [2018]) improving multi-view approaches to 3D classification which seem to be the most promising approach. It would be worth including these new systems to our framework.

Almost all of the networks we have tested describe in the papers and implement in the code a version for part segmentation of the 3D models. This is a

problem of dividing a 3D model to logical parts such as dividing the model of a table to the top desk and the legs. It would be worth comparing all the different networks as well.

We hope that this work will lead to some practical improvements in the fields such as interior design, where artificial intelligence can solve menial tasks currently solved by experts.

Bibliography

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]*, September 2014. URL http://arxiv.org/abs/1409.0473. arXiv: 1409.0473.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975. ISSN 00010782. doi: 10.1145/361002.361007. URL http://portal.acm.org/citation.cfm?doid=361002.361007.

Bishop and Weimer. Fast Phong Shading. *Computer Graphics (20) 4 pp. 103-106*, 20, 1986. doi: 10.1145/15886.15897.

Andrew Brock. VRN in Theano with lasagne, 2016. URL https://github.com/ajbrock/Generative-and-Discriminative-Voxel-Modeling.

Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. Generative and Discriminative Voxel Modeling with Convolutional Neural Networks. *arXiv:1608.04236 [cs, stat]*, August 2016. URL http://arxiv.org/abs/1608.04236. arXiv: 1608.04236.

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. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv:1406.1078 [cs, stat]*, June 2014. URL http://arxiv.org/abs/1406.1078. arXiv: 1406.1078.

Sungjoon Choi, Qian-Yi Zhou, Stephen Miller, and Vladlen Koltun. A Large Dataset of Object Scans. *arXiv:1602.02481*, 2016. URL http://redwood-data.org/3dscan/index.html.

François Chollet and others. *Keras*. 2015. URL https://keras.io.

Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, Diogo Moitinho de Almeida, Brian McFee, Hendrik Weideman, Gábor Takács, Peter de Rivaz, Jon Crall, Gregory Sanders, Kashif Rasul, Cong Liu, Geoffrey French, and Jonas Degrave. *Lasagne: First release.* August 2015. doi: 10.5281/zenodo.27878. URL http://dx.doi.org/10.5281/zenodo.27878.

Miguel Dominguez. G3dnet in Tensorflow, 2018. URL https://github.com/WDot/G3DNet.

Miguel Dominguez, Rohan Dhamdhere, Atir Petkar, Saloni Jain, Shagan Sah, and Raymond Ptucha. *General-Purpose Deep Point Cloud Feature Extractor*. March 2018. doi: 10.1109/WACV.2018.00218.

Yifan Feng, Zizhao Zhang, Xibin Zhao, Rongrong Ji, and Yue Gao. GVCNN: Group-View Convolutional Neural Networks for 3d Shape Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL https://www.deeplearningbook.org/.

Vishakh Hegde and Reza Zadeh. FusionNet: 3d Object Classification Using Multiple Data Representations. *arXiv:1607.05695 [cs]*, July 2016. URL http://arxiv.org/abs/1607.05695. arXiv: 1607.05695.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural Networks for Machine Learning: Lecture 6a Overview of mini-batch gradient descent, 2012. URL http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 9:1735–80, 1997. doi: 10.1162/neco.1997.9.8.1735.

Binh-Son Hua, Quang-Hieu Pham, Duc Thanh Nguyen, Minh-Khoi Tran, Lap-Fai Yu, and Sai-Kit Yeung. SceneNN: A Scene Meshes Dataset with aNNotations. In *International Conference on 3D Vision (3DV)*, 2016. URL http://www.scenenn.net/.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. Deep Networks with Stochastic Depth. *arXiv:1603.09382 [cs]*, March 2016. URL http://arxiv.org/abs/1603.09382. arXiv: 1603.09382.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]*, February 2015. URL http://arxiv.org/abs/1502.03167. arXiv: 1502.03167.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3063-3. doi: 10.1145/2647868.2654889. URL http://doi.acm.org/10.1145/2647868.2654889. event-place: Orlando, Florida, USA.

Asako Kanezaki. RotationNet in Caffe, 2018. URL https://github.com/kanezaki/rotationnet.

Asako Kanezaki, Yasuyuki Matsushita, and Yoshifumi Nishida. RotationNet: Joint Object Categorization and Pose Estimation Using Multiviews from Unsupervised Viewpoints. *arXiv:1603.06208 [cs]*, March 2018. URL http://arxiv.org/abs/1603.06208. arXiv: 1603.06208.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. URL http://arxiv.org/abs/1412.6980. arXiv: 1412.6980.

Roman Klokov and Victor Lempitsky. Escape from Cells: Deep Kd-Networks for the Recognition of 3d Point Cloud Models. *arXiv:1704.01222 [cs]*, April 2017a. URL http://arxiv.org/abs/1704.01222. arXiv: 1704.01222.

Roman Klokov and Victor Lempitsky. KD-Net in Theano and Lasagne, 2017b. URL https://github.com/Regenerator/kdnets.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257. event-place: Lake Tahoe, Nevada.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 1989.

Tang Lee. Multi-View CNN in Tensorflow, 2016. URL https://github.com/WeiTang114/MVCNN-TensorFlow.

Jiaxin Li. SONET in TensorFlow, 2018. URL https://github.com/lijx10/SO-Net.

Jiaxin Li, Ben M. Chen, and Gim Hee Lee. SO-Net: Self-Organizing Network for Point Cloud Analysis. *arXiv:1803.04249 [cs]*, March 2018. URL http://arxiv.org/abs/1803.04249. arXiv: 1803.04249.

S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489.

machrisaa. VGG in Tensorflow, 2017. URL https://github.com/machrisaa/tensorflow-vgg.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL http://tensorflow.org/.

D. Maturana and S. Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *IROS 2015*, 2015.

Daniel Maturana and Sebastian Scherer. VoxNet in TensorFlow, 2016. URL https://github.com/Durant35/VoxNet.

Donald Meagher. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. October 1980.

Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), March 2014. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=2600239.2600241.

Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. OpenVDB: An Open-source Data Structure and Toolkit for High-resolution Volumes. In *ACM SIGGRAPH 2013 Courses*, SIGGRAPH '13, pages 19:1–19:1, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2339-0. doi: 10.1145/2504435.2504454. URL http://doi.acm.org/10.1145/2504435.2504454. event-place: Anaheim, California.

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. In *NIPS-W*, 2017.

Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010. ISBN 0-12-375079-2 978-0-12-375079-2.

Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3d Classification and Segmentation. *arXiv:1612.00593 [cs]*, December 2016a. URL http://arxiv.org/abs/1612.00593. arXiv: 1612.00593.

Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet TensorFlow, 2016b. URL https://github.com/charlesq34/pointnet.

Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet++ TensorFlow, 2017a. URL https://github.com/charlesq34/pointnet2.

Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. *arXiv:1706.02413 [cs]*, June 2017b. URL http://arxiv.org/abs/1706.02413. arXiv: 1706.02413.

Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington DC, 1961.

Nima Sedaghat, Mohammadreza Zolfaghari, Ehsan Amiri, and Thomas Brox. Orientation-boosted Voxel Nets for 3d Object Recognition. *arXiv:1604.03351 [cs]*, April 2016a. URL http://arxiv.org/abs/1604.03351. arXiv: 1604.03351.

Nima Sedaghat, Mohammadreza Zolfaghari, Ehsan Amiri, and Thomas Brox. ORION in Caffe, 2016b. URL https://github.com/lmb-freiburg/orion.

Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2945397. URL http://doi.acm.org/10.1145/2939672.2945397. event-place: San Francisco, California, USA.

K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

I. M. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86 – 112, 1967. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(67)90144-9. URL http://www.sciencedirect.com/science/article/pii/0041555367901449.

Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. Semantic Scene Completion from a Single Depth Image. *Proceedings of 29th IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

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*, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view Convolutional Neural Networks for 3d Shape Recognition. *arXiv:1505.00880 [cs]*, May 2015. URL http://arxiv.org/abs/1505.00880. arXiv: 1505.00880.

Jong-Chyi Su, Matheus Gadelha, and Rui Wang. Multi-view CNN in Pytorch, 2018a. URL https://github.com/jongchyisu/mvcnn_pytorch.

Jong-Chyi Su, Matheus Gadelha, Rui Wang, and Subhransu Maji. A Deeper Look at 3d Shape Classifiers. *arXiv:1809.02560 [cs]*, September 2018b. URL http://arxiv.org/abs/1809.02560. arXiv: 1809.02560.

I Sutskever, J Martens, G Dahl, and G Hinton. On the importance of initialization and momentum in deep learning. *30th International Conference on Machine Learning, ICML 2013*, pages 1139–1147, 2013.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *arXiv:1602.07261 [cs]*, February 2016. URL http://arxiv.org/abs/1602.07261. arXiv: 1602.07261.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. O-CNN: Octree-based Convolutional Neural Networks for 3d Shape Analysis. *ACM Transactions on Graphics (SIGGRAPH)*, 36(4), 2017.

Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. Adaptive O-CNN: A Patch-based Deep Representation of 3d Shapes. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 37(6), 2018a.

Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. O-CNN in Caffe, 2018b. URL https://github.com/Microsoft/O-CNN.

Francis Williams. Point Cloud Utils, 2019.

Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d ShapeNets: A Deep Representation for Volumetric Shapes. *arXiv:1406.5670 [cs]*, June 2014. URL http://arxiv.org/abs/1406.5670. arXiv: 1406.5670.

Yu Xiang, Wonhui Kim, Wei Chen, Jingwei Ji, Christopher Choy, Hao Su, Roozbeh Mottaghi, Leonidas Guibas, and Silvio Savarese. ObjectNet3d: A Large Scale Database for 3d Object Recognition. In *European Conference Computer Vision (ECCV)*, 2016. URL http://cvgl.stanford.edu/projects/objectnet3d/.

Haoxuan You, Yifan Feng, Rongrong Ji, and Yue Gao. PVNet: A Joint Convolutional Network of Point Cloud and Multi-View for 3d Shape Recognition. *arXiv:1808.07659 [cs]*, August 2018. URL http://arxiv.org/abs/1808.07659. arXiv: 1808.07659.

Tan Yu, Jingjing Meng, and Junsong Yuan. Multi-view Harmonized Bilinear Network for 3d Object Recognition. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 186–194, 2018.

Han Zhizhong, Shang Mingyang, and Liu Zhenbao. SEQ2seq in Tensorflow, 2018a. URL https://github.com/mingyangShang/SeqViews2SeqLabels.

Han Zhizhong, Shang Mingyang, and Liu Zhenbao. SeqViews2seqlabels: Learning 3d Global Features via Aggregating Sequential Views by RNN With Attention. *IEEE Transactions on Image Processing*, 28(2):658 – 672, September 2018b.

List of Figures

List of Tables

List of Abbreviations

AO-CNN adaptive octree neural network
API application programming interface
CPU central processing unit
GRU gated recurrent unit
GPU graphics processing unit
LSTM long short-term memory unit
MVCNN multi-view convolutional neural network
O-CNN octree convolutional neural network
RAM random access memory
ReLU rectified linear unit
RNN recurrent neural network
Seq2Seq sequence to sequence
SOM self-organizing map
VGG visual geometry group network
VRN voxel residual network

A. Training parameters

In this attachment we present some of the most important parameters we used to train the neural networks.

- **VRN**

  training epochs: 20
  batch size: 24
  learning rate: 0.002 for 10 epochs and then 0.0002
  number of rotations: 24

- **O-CNN**

  training epochs: 50
  batch size: 64
  learning rate: 0.1, divided by ten every ten epochs
  number of rotations: 12

- **AO-CNN**

  training epochs: 50
  batch size: 64
  learning rate: 0.1, divided by ten every ten epochs
  number of rotations: 12

- **VGG**

  training epochs: 20
  batch size: 60
  learning rate: 0.0001, multiplied by 0.75 every three epochs
  number of views: 12

- **MVCNN**

  training epochs: 200
  batch size: 64
  learning rate: 0.0001
  number of views: 12

- **MVCNN2**

  training epochs: 30+30
  batch size: 64
  learning rate: 0.00005
  number of views: 12

- **RotationNet**

  training epochs: 200
  batch size: 40
  learning rate: 0.0001 divided by ten every fifty epochs
  number of views: 12

- **Seq2Seq**

  training epochs: 200
  batch size: 32
  learning rate: 0.0002
  number of views: 12

- **PointNet**

  training epochs: 200
  batch size: 64
  number of points: 2048
  learning rate: 0.0001 multiplied by 0.8 every 20 epochs
  number of rotations: 12

- **PointNet++**

  training epochs: 200
  batch size: 32
  number of points: 2048
  learning rate: 0.0001 multiplied by 0.7 every 20 epochs
  number of rotations: 12

- **SO-Net**

  training epochs: 400
  batch size: 8
  number of points: 5000
  learning rate: 0.001 divided by two every 40 epochs
  number of rotations: 1

- **KD-Net**

  training epochs: 200
  batch size: 16
  number of points: 2048
  learning rate: 0.001
  number of rotations: 12

B. Detailed results

The tables above (Table B.1 and Table B.2) show more detailed results of our experiments.

Network	Mean	Max	Min
Voxel			
VRN	90.32	90.84	88.82
OCTREE	88.29	91.25	82.37
OCTREE ADAPTIVE	91.08	93.68	86.59
Multi-view			
MVCNN (depth)	87.83	88.94	87.07
MVCNN (PBRT)	87.62	89.10	86.35
MVCNN (phong)	84.00	84.76	83.06
MVCNN (shaded)	88.83	89.79	88.09
MVCNN2 (depth)	90.52	91.65	89.55
MVCNN2 (PBRT)	90.26	91.21	88.57
MVCNN2 (phong)	88.97	89.79	88.13
MVCNN2 (shaded)	90.64	91.77	89.34
VGG (depth)	89.27	89.65	88.88
VGG (PBRT)	87.80	88.63	87.25
VGG (phong)	87.69	87.90	87.41
VGG (shaded)	90.86	91.11	90.66
ROTNET (depth)	91.46	91.65	91.25
ROTNET (PBRT)	91.09	91.17	91.00
ROTNET (phong)	89.22	94.59	85.71
ROTNET (shaded)	92.12	92.22	91.94
SEQ2SEQ (depth)	89.08	89.14	88.94
SEQ2SEQ (PBRT)	88.10	88.21	88.05
SEQ2SEQ (phong)	87.09	87.24	86.95
SEQ2SEQ (shaded)	91.26	91.41	91.13
Point cloud			
KDNET	88.10	88.65	87.64
POINTNET (lloyd)	85.25	85.66	84.85
POINTNET (uniform)	82.65	83.67	81.60
POINTNET (original)	86.60	87.60	85.78
POINTNET (sobol)	85.87	86.18	85.17
POINTNET++ (lloyd)	88.36	88.78	87.84
POINTNET++ (uniform)	87.57	88.13	86.99
POINTNET++ (original)	89.00	89.42	88.86
POINTNET++ (sobol)	88.54	88.74	88.21
SONET (2048 points)	85.23	85.53	85.01
SONET (5000 points)	89.00	89.42	88.45

Table B.1: Detailed statistics of ModelNet40 experiments. The statistics are computed over the last 10 training epochs.

Network	Mean	Max	Min
Voxel			
VRN	88.98	89.62	88.36
OCTREE	90.75	92.69	86.76
OCTREE ADAPTIVE	91.33	93.26	87.67
Multi-view			
MVCNN (shaded)	89.23	89.54	88.65
MVCNN2 (shaded)	92.22	92.51	91.92
VGG (shaded)	91.90	92.43	89.80
ROTNET (shaded)	93.09	98.83	70.70
SEQ2SEQ (shaded)	92.35	92.47	92.10
Point cloud			
KDNET	81.23	81.92	80.75
POINTNET (uniform)	82.53	82.89	82.24
POINTNET++ (uniform)	83.67	83.80	83.48
SONET (5000 points)	87.87	88.23	87.47

Table B.2: Detailed statistics of ShapeNetCore experiments. The statistics are computed over the last 10 training epochs.

C. Manual

This section contains instructions how to use the code we used to conduct our experiments.

C.1 Requirements

To run the code you will need a computer with Linux-based operating system and NVIDIA GPU.

You will need to install the following:

- NVIDIA drivers
  (Installation guide here: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/)
- Docker version 1.12 or higher
  (Installation guide here: https://docs.docker.com/install/)
- NVIDIA Container Runtime for Docker
  (Installation guide here: https://github.com/NVIDIA/nvidia-docker)

Each neural network is an independent Docker image and all its dependencies are installed when building the image. All code is written in python.

C.2 Datasets Setup

The code is made to work with ModelNet40 and ShapeNetCore datasets. The easiest way to run it with custom dataset is to restructure your data so it resembles the structure of one of these datasets.

Modelnet40

- Download the dataset (http://modelnet.cs.princeton.edu/). For experiments we used manually aligned version of the dataset.
- Unpack the downloaded archive.

ShapeNetCore

- Dowload the dataset (https://www.shapenet.org/download/shapenetcore). You need to register and wait for a confirmation email.
- Unpack the downloaded archive.

C.3 General Setup

Each network is implemented as a separate Docker image. To learn more about Docker, images and containers visit https://docs.docker.com.

Each neural network is contained in one directory in /dockers. None of the networks accepts mesh files as their input directly, so some data conversion is required. All data conversion is implemented in Docker with the same structure as neural networks themselves. The code for data conversion is located in */dockers/data_conversion*. More details on the structure of the electronic attachments is given in Appendix D.

Each directory contains two important files: *config.ini* and *run.sh*, which you will need to open and edit. Another important file is Dockerfile which contains the definition of the Docker image. Remaining files contain the code which differ from the original network implementation. Original network code is downloaded automatically when building the image and you can find the download link below.

run.sh is a runnable script which builds the Docker image, runs the Docker container and executes the data conversion or neural network training and evaluation. You will need to setup a couple of variables here:

- name – will be used as a name of the Docker image and Docker container. You can leave this at default value unless it is in conflict with some already existing image or you want to run more instances of this image at once. With data conversion scripts the name is the name of the converted dataset and directory of the same name will be created. The name of the image can be changed by changing variable image_name in this case.
- dataset_path – contains the path to the root directory of the dataset on your filesystem. (Used as input.)
- out_path – contains the path to the directory where training logs and network weights will be saved. This directory will be created automatically.
- GPU – index of the GPU which will be visible to Docker container. Have to be a single integer. We currently do not support multiple GPUs.
- docker_hidden – Must be one of "t" or "d". With "t" the container will be run in interactive mode, meaning it will run in your console. With "d" it will in detached mode i.e. in the background. For more information check Docker documentation.

config.ini contains most of the relevant parameters of the network or data conversion. The file is split to sections where each section is started by [SECTION] statement. Then on each line a parameter in format key = value. Explanation of network parameters is located in later sections.

C.4 Data conversion

To convert your dataset you need to set the parameters described above and then simply run the *run.sh* script in your shell console. This will convert the dataset to various formats directly readable by the neural networks.
Parameters for data conversion in config.ini file:

- data – path to the dataset inside the container. Does not have to be changed.

- output – path to the directory inside the container where converted dataset will be saved. Does not have to be changed.
- log_file – path and name of the file where progress of the data conversion will be written. By default its located in the output directory and called log.txt.
- num_threads – maximum number of threads to use.
- dataset_type – The type of dataset being converted. Must be one of "modelnet" or "shapenet" currently.

C.5 Neural Networks

Each of the neural networks is implemented in python but in different framework. That is why we used the Docker infrastructure. We try to present a unified framework to easily test and train the networks without changing the code. This section will briefly introduce used networks and some of their most important parameters.

Parameters common to all neural networks:

- name – will be used as the name of the experiment used in log files.
- data – path to the dataset inside the container. Does not have to be changed.
- log_dir – path to the directory inside the container where logs and weights will be saved. Does not have to be changed.
- num_classes – the number of classes in the dataset. (40 for ModelNet40 and 55 for ShapeNetCore)
- batch_size – size of the batch for training and testing neural networks.
- weights – if you want to test or fine-tune already trained network, this should be the number of this model. If you want to train from scratch, this should be -1.
- snapshot_prefix – name of the file where weights will be saved. The number of the training epoch when these weights are saved will be added to this.
- max_epoch – number of epochs to train for. One epoch means one pass through the training part of the dataset.
- save_period – the trained network will be saved every epoch divisible by save_period.
- train_log_frq – frequency of logging during training. It is roughly the number of examples seen by network.
- test – if you want to only test an already trained network, set this to "True". weights parameter has to have a valid value bigger than -1. Should be "False" for training.

D. List of electronic attachments

The following diagram shows the directory structure of the electronic attachment to this thesis:

```
materials
    papers
dockers
    data_conversion
        kdnet_data
        mvcnn_data_blender
        mvcnn_data_pbrt
        octree_data
        pnet_data
        vrnens_data
    kdnet
    mvcnn
    mvcnn2
    octree
    octree_adaptive
    pointnet
    pointnet2
    rotnet
    seq2seq
    sonet
    vgg
    vrnens
```

For further research convenience we enclose original papers describing tested neural networks in *materials/papers*.