DirCNN: Rotation Invariant Geometric Deep Learning

YANNICK SAIVE
DirCNN: Rotation Invariant Geometric Deep Learning

YANNICK SAIVE
Abstract

Recently geometric deep learning introduced a new way for machine learning algorithms to tackle point cloud data in its raw form. Pioneers like PointNet and many architectures building on top of its success realize the importance of invariance to initial data transformations. These include shifting, scaling and rotating the point cloud in 3D space. Similarly to our desire for image classifying machine learning models to classify an upside down dog as a dog, we wish geometric deep learning models to succeed on transformed data. As such, many models employ an initial data transform in their models which is learned as part of a neural network, to transform the point cloud into a global canonical space. I see weaknesses in this approach as they are not guaranteed to perform completely invariant to input data transformations, but rather approximately. To combat this I propose to use local deterministic transformations which do not need to be learned. The novelty layer of this project builds upon Edge Convolutions and is thus dubbed \textbf{DirEdgeConv}, with the directional invariance in mind. This layer is slightly altered to introduce another layer by the name of \textbf{DirSplineConv}. These layers are assembled in a variety of models which are then benchmarked against the same tasks as its predecessor to invite a fair comparison. The results are not quite as good as state of the art results, however are still respectable. It is also my belief that the results can be improved by improving the learning rate and its scheduling. Another experiment in which ablation is performed on the novel layers shows that the layers main concept indeed improves the overall results.
Sammanfattning

DirCNN: Rotationsinvariant geometrisk deep learning

Declaration

I hereby, declare that this thesis is my own work and that, to the best of my knowledge and belief, it contains no material which has been accepted or submitted for the award of any other degree or diploma. I also declare that, to the best of my knowledge and belief, this thesis contains no material previously published or written by any other person except where due reference is made in the text of the thesis.

Yannick Saive
May 27, 2019
Contents

1 Introduction 1
   1.1 Research Question ............................................. 1
   1.2 Outline of Thesis ............................................. 2

2 Fundamental Theory 3
   2.1 Linear Algebra .................................................. 3
   2.2 Probability theory and optimization for machine learning ................................................. 5
   2.3 Neural network fundamentals .................................... 9
   2.4 Graphs .............................................................. 16

3 Specialized Theory 19
   3.1 Problems with point clouds ..................................... 19
   3.2 Requirements of a model ........................................ 20
   3.3 Voxel approach ................................................... 20
   3.4 Geometric deep learning ........................................ 21
   3.5 Dataset ............................................................ 28
   3.6 Previous results on ModelNet ................................... 28

4 Extension 29
   4.1 The DirSplineConv layer ...................................... 29
   4.2 The DirSplineConv3D layer ..................................... 33
   4.3 From 3D features to 3D features ............................... 35
   4.4 Connection to other models .................................... 38
   4.5 Novel models ...................................................... 39

5 Experiments 42
   5.1 Experiments on DirDCNN ....................................... 42
   5.2 Experiments on DirCNN .......................................... 43
   5.3 Experiments on DirSplineCNN .................................. 45
   5.4 Experiments on DirCNN with pooling ........................... 47
   5.5 Experiments on ModelNet40 .................................... 47
      5.5.1 DirDCNN40+grclus ........................................ 47
      5.5.2 DirSplineDCNN40+grclus ................................ 48
   5.6 Outlook ........................................................... 50

6 Conclusion 51

References 52
Chapter 1

Introduction

This work will focus on a new area of research in the domain of machine learning. In 2016 a novel approach of using neural networks to work on point cloud data was introduced, and since then a variety of works have built up on this. Coined Geometric Deep Learning (GDL), the aim of this field of study is to understand how to work with point clouds. To generate the data point clouds are typically sampled from the surface of a virtual object, or recorded with 2.5D cameras. Before GDL these were either converted to a voxel grid, or the graph Laplacian was used and the resulting matrix was fed into a neural architecture. For GDL the point clouds are used raw, meaning only their 3D coordinates are used and fed into a machine learning model. While voxel grids will be mentioned in order to highlight the differences between point clouds and 2D images and how learning is done on them, they will not be used in experiments.

All underlying data structures allow for a variety of machine learning tasks. These could be classification, where, for example, a voxel grid is classified as a chair. Segmentation methods try to map a point from the chair to whether it is part of a leg, the armrest etc. Scene segmentation extend upon this by sampling points from a room and estimating what points belong to the walls, floor, tables etc. Finally, geometric deep learning is used in point matching, where two intersecting scenes are superimposed to create one larger scene. An example of this is shown in Figure 1.1.

More abstract tasks can also be explored. For those, point clouds are usually converted to graphs, where the points coordinates are in higher dimensional feature spaces. Examples are non Euclidean graphs, which could depict social graphs or comment histories on online blogs. Others have done chemical analysis using the new techniques of GDL on molecules, however presumably this is not a task suited for this project, which relies on many generic points in a point cloud.

Research Question

PointNet[24], PointNet++[23], DGCNN[29], PointCNN[19] and others, all make use of either a spatial transform or of reordering the points. The goal of these is to make the entire network invariant to initial rotations, translations and reordering of the point cloud. The issue with this is, that these invariants are only achieved approximately by using a global canonical coordinate system and having a neural network
learn them. This project makes away with global canonical coordinate systems and uses instead only local neighborhoods, which are aligned in 3D space through the use of deterministic mathematical tools as opposed to learned neural networks. The advantage of this is that invariance is provable and that the network will have fewer parameters as there is no transformer network. To evaluate this hypothesis the networks introduced in this project are compared on the same benchmarks as the previously mentioned models.

**Outline of Thesis**

The following will introduce the outline of this work. While this Chapter has focused on giving the reader an overview of the field of research, its application in the real world and its hopes for the future, we need to take one step back, and start at the theory. In Chapter 2 the basic theory is introduced, including machine learning basics and a discussion of the underlying data. Next, in Chapter 3, we introduce the theory which is necessary for Geometric Deep Learning, and some models from other research are introduced. As models are evaluated on a dataset, we introduce ModelNet in this chapter, and the results of other models on this dataset.

As a novelty, Chapter 4 introduces directional layers, motivates them and evaluates them on a proof of concept dataset. After introducing the novel layers, a full model is defined, and its structure is compared to other models.

The model is then evaluated in Chapter 5 on the ModelNet dataset. This chapter includes a discussion of why the results obtained are comparable to that of other research, the experiments themselves and the results.

Finally, this work is closed with a conclusion in Chapter 6.
Chapter 2

Fundamental Theory

This chapter focuses on a range of preliminaries necessary for the later discussions in this work. Basic linear algebra results are reviewed and the notation which is used is introduced. Machine learning models are then introduced and it is shown how they learn. Afterwards, neural networks are discussed, and their application domains highlighted. Finally, the datatype that is considered in this work is defined. The following chapter uses these preliminaries to build the foundations necessary to discuss the contribution of this work and its evaluation in Chapter 5.

Linear Algebra

A vector space $V$ over a field $\mathbb{F}$ is a set of points which is closed under scalar multiplication and finite vector addition. In this work I will only be looking at vector spaces defined over Euclidean space with finite dimension, such that $V \subset \mathbb{R}^n$. A basis of $V$ is a maximally linearly independent set of vectors $\{b_1, \ldots, b_n\}$, such that they span $V$. This means any $v \in V$ can be uniquely written as $v = \sum_{i=1}^{n} c_i b_i$ with $c_i \in \mathbb{R}$. Let the coordinates of $v$ under $B$ be defined as $[v]_B := (c_1, \ldots, c_n)$, and note that under the canonical basis of $\mathbb{R}^n$, $E_n = \{e_1, \ldots, e_n\}$ with $e_i \in \mathbb{R}^n$ all zeros except for a 1 in the $i$'th position, we have that any $v \in V$ satisfies $v_i = [v]_{E_n,i}$. Given two bases $B_1 = \{b_1, \ldots, b_n\}$ and $B_2 = \{\beta_1, \ldots, \beta_n\}$ of $V$, the linear map which transforms $[v]_{B_1}$ to $[v]_{B_2}$ is called the change of basis transform from $B_1$ to $B_2$. We write

$$T_{B_2,B_1} : \mathbb{R}^n \to \mathbb{R}^n,$$

$$[v]_{B_1} \mapsto [v]_{B_2}.$$  

We identify the linear map with its matrix form, which is given by

$$T_{B_2,B_1} = T_{B_2,E_n} T_{E_n,B_1} = [\beta_1, \ldots, \beta_n]^{-1}[b_1, \ldots, b_n].$$

Finally, an orthonormal basis of a vectorspace $V$ is a basis $B$ where each vector $b_i \in B$ has a norm of 1, and each pair of vectors $b_i, b_j \in B$ is orthogonal to each other. We write $\|b_i\|_2 = 1$ and $\langle b_i, b_j \rangle = 0$ for any $b_i \neq b_j$. Because the inverse of a matrix with orthogonal columns is equal to its transpose, we get that $T_{B,E_n}^{-1} = T_{B,E_n}^T = T_{E_n,B}$, for an orthonormal basis $B$. Hence, if $B_1$ and $B_2$ are orthonormal basis, then $T_{B_2,B_1} = [\beta_1, \ldots, \beta_n]^T[b_1, \ldots, b_n]$.  

3
Consider now \( X \in \mathbb{R}^{d \times n} \), where each column corresponds to a sample of an experiment with \( d \) features. The Principal Component Analysis (PCA) is a transform of the feature space which minimizes the covariance between features. This means where \( XX_T^{(i,j)} \neq 0 \), we get that in the transformed space the covariances are zero. Concretely, we notice that \( XX_T \) is symmetric positive semidefinite. Thus \( XX_T \) is diagonalisable and the eigenvalues are non negative and the eigenvectors are orthogonal. Let \( \lambda_1 \geq \cdots \geq \lambda_d \geq 0 \) be the eigenvalues to the associated eigenvectors \( b_1, \ldots, b_d \), and let \( B := \{ b_1, \ldots, b_d \} \) be an orthonormal basis. The change of basis transform \( T_{B,E} \) coincides with that of the diagonalisation of \( XX_T \)

\[
XX_T = T_{E_n,B} \Lambda T_{B,E_n},
\]

where \( \Lambda = \text{diag}(\lambda_1, \ldots, \lambda_d) \). When considering the covariance of the transformed features \( \tilde{X} := T_{B,E_n}X \),

\[
\tilde{X} \tilde{X}^T = T_{B,E_n} XX_T T_{E_n,B} = \Lambda,
\]

we see that they covariances are all zero, as \( \Lambda \) is a diagonal matrix. The vectors \( b_i \) are called the \( i \)’th principal components, and correspond to the direction in featurespace in which the dataset \( X \) shows the \( i \)’th strongest variance.

In Chapter 4 we will need to find the plane through the origin which best fits a sample of points \( X \in \mathbb{R}^{3 \times n} \) in three dimensional space. As it turns out, the plane is easily found by using PCA. First, note that \( XX_T \in \mathbb{R}^{3 \times 3} \), such that \( B \) contains only three principal components. Then the plane is spanned by the first two principal components, as can be easily verified. The square error between a plane with normal \( v \), \( \|v\|_2 = 1 \), and all points in \( X \) is given by

\[
\sum_{i=1}^{n} \langle x_i, v \rangle^2 = v^T XX_T v,
\]

as the points are projected onto \( v \). Without loss of generality we can assume that \( \|b_1\|_2 = \|b_2\|_2 = \|b_3\|_2 = 1 \), and recall that \( \langle b_3, b_1 \rangle = \langle b_3, b_2 \rangle = 0 \). We see that for any \( v \in \mathbb{R}^3 \) with \( \|v\|_2 = 1 \), which we can write as \( v = a_1 b_1 + a_2 b_2 + a_3 b_3 \), we have that

\[
v^T XX_T v = (a_1 b_1 + a_2 b_2 + a_3 b_3)XX_T (a_1 b_1 + a_2 b_2 + a_3 b_3) = a_1^2 \lambda_1 + a_2^2 \lambda_2 + a_3^2 \lambda_3 \geq (a_1^2 + a_2^2 + a_3^2) \lambda_3 = \lambda_3,
\]

as \( a_1^2 + a_2^2 + a_3^2 = 1 \) and \( \lambda_1 \geq \lambda_2 \geq \lambda_3 \). Because \( b_3^T XX_T b_3 = \lambda_3 \), we conclude that a plane with normal \( b_3 \) minimizes the square error between the plane and the datapoints in \( X \), and it is unique if \( \lambda_1 \geq \lambda_2 > \lambda_3 \).

As it turns out, we are not interested in the eigenvalues of \( XX_T \), but rather just in the vectors \( b_1, b_2 \) and \( b_3 \). A much more computationally efficient and stable way of finding this orthonormal basis is by means of the singular value decomposition (SVD). The SVD of a real matrix \( X \in \mathbb{R}^{d \times n} \) decomposes \( X \) into two orthogonal matrices \( U \in \mathbb{R}^{d \times d} \) and \( V \in \mathbb{R}^{n \times n} \) and a diagonal1 matrix \( \Sigma \in \mathbb{R}^{d \times n} \) such that

\[
X = U \Sigma V^T.
\]

1Here, a non square matrix is considered diagonal, if only those elements of the matrix that are on the diagonal are non-zero.
The matrix $\Sigma$ is unique, if one decides to order the diagonal elements by size, as these are always non-negative. The uniqueness of $U$ and $V$ depends on many factors, like $d < n$, $d = n$ or $d > n$, and the rank of $X$. In Chapter 5 we will always have $d = 3$ and $n \gg 3$, and hence we assume that $X$ has full rank. This leads to $U$ being unique up to multiplication by -1 for every column.

Once having found the SVD of $X$ it can be seen that

$$XX^T = U\Sigma V^TV\Sigma^TU^T = U(\Sigma\Sigma^T)U^T;$$

as $V$ is orthogonal. Hence $U = T_{E_n,B}$ and $(\Sigma\Sigma^T) = \Lambda$.

### Probability theory and optimization for machine learning

In this section we will introduce the necessary content to understand what, in a general sense, the task at hand is. Furthermore we will explore what it means for a model to learn, and how this is done.

#### A model and loss

A supervised learning task is one, where a dataset $X, Y$ is given, with features $X = \{x_1, \ldots, x_n\}$ with $x_i \in \mathbb{R}^d$, and labels $Y = \{y_1, \ldots, y_n\}$ with $y_i \in \{0, 1\}^C$, where $C \in \mathbb{N}$ is the number of different labels, and each label $y_i$ is a one-hot encoding. This means that $y_i$ has exactly one entry which is 1, and all others are 0. If the $c$'th component of $y_i$ is one, then we say that $x_i$ belongs to the class $c$. The goal then is to find a mapping $f$ which, maps a datapoint $x_i$ to $y_i$.

We assume that $x_i, y_i$ are jointly sampled from $p_{\text{data}}(x_i, y_i)$, the data generating distribution, and the pairs are all independent and identically distributed (iid). Our goal is to estimate the conditional probability $p_{\text{data}}(y_i|x_i)$, meaning given a feature $x_i$, we wish to know what label $y_i$ corresponds to it. Define $p_{\text{model}}(x_i, y_i; \theta)$ as a distribution dependent on a parameter $\theta$, which aims to map the pair $x_i, y_i$ to its probability density under the real data generating probability density, i.e. $p_{\text{model}}(x_i, y_i; \theta)$ shall approximate $p_{\text{data}}(x_i, y_i)$. The maximum likelihood estimator (MLE) for $\theta$ is

$$\theta^* := \arg \max_{\theta} p_{\text{model}}(X, Y; \theta)$$

$$= \arg \max_{\theta} \prod_{i=1}^{n} p_{\text{model}}(x_i, y_i; \theta).$$

As the logarithm is a concave function, and scalar multiplication doesn’t change the maximum of a function either, we can rewrite the MLE as

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^{n} \log p_{\text{model}}(x_i, y_i; \theta)$$

$$= \arg \max_{\theta} \frac{1}{n} \sum_{i=1}^{n} \log p_{\text{model}}(x_i, y_i; \theta).$$
As we assume $x_i, y_i$ to be sampled iid, this can be expressed as an expectation,

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{x, y \sim p_{\text{data}}} \left[ \log p_{\text{model}}(x, y; \theta) \right].$$

Now we can compare the MLE to the Kullback Leibler divergence between $p_{\text{data}}$ and $p_{\text{model}}$,

$$D_{\text{KL}}(p_{\text{data}} \| p_{\text{model}}) = \mathbb{E}_{x, y \sim p_{\text{data}}} \left[ \log p_{\text{data}}(x, y) - \log p_{\text{model}}(x, y; \theta) \right]$$

and realize that minimizing $D_{\text{KL}}$ can be achieved by plugging in $\theta^*$, as the left hand term in the expectation is independent of $\theta$. To finish this discussion we finally note that the cross entropy $H(p, q) := \mathbb{E}_p[-\log q]$ between distributions $p$ and $q$ is a measure of how distinct $q$ is from $p$. As $D_{\text{KL}}(p\|q) = H(p, q) - H(p)$, with $H$ the Shannon entropy, minimizing $D_{\text{KL}}$ corresponds to minimizing $H(p, q)$. This shows that finding the MLE is precisely what we need to do, and that there are many intermediary equations that we can actually use to do so. It is, for example, computationally more stable to minimize the negative log likelihood (NLL), $-\sum_i \log p(x_i, y_i; \theta)$, than computing $\prod_i p(x_i, y_i|\theta)$.

For the conditional probability $p_{\text{model}}(y|x_i; \theta)$, we can look at the conditional NLL

$$\theta^* = \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n - \log p_{\text{model}}(y_i|x_i, \theta).$$

It is now time to tie the function $f : \mathbb{R}^d \rightarrow \mathbb{R}^C$, which is going to do the prediction, and $p_{\text{model}}$ together. To this end we introduce the *softmax* function $\sigma : \mathbb{R}^c \rightarrow (0, 1)^C$ via its components

$$\sigma_c : \mathbb{R}^C \rightarrow (0, 1),$$

$$z \mapsto \frac{e^{z_c}}{\sum_{j=1}^C e^{z_j}},$$

and set

$$p_{\text{model}}(y|x; \theta) := \prod_{c=1}^C \sigma_c(f_\theta(x))^y_c.$$

One can quickly verify that $\sum_{c=1}^C p_{\text{model}}(e_c|x; \theta) = 1$ with $e_c$ a one-hot encoding. With this we see that the NLL becomes

$$- \log p_{\text{model}}(Y|X; \theta) = \sum_{i=1}^n \sum_{c=1}^C - y_{i,c} \log \sigma_c(f_\theta(x_i)) =: L(X, Y; \theta),$$

which we define as the *loss*, and is commonly refereed to as the *categorical cross entropy loss*.

**Optimization**

We defined $p_{\text{model}}(y|x; \theta) = \prod_{c=1}^C \sigma_c(f_\theta(x))^y_c$ for a model dependent on $\theta$, however in the context of neural networks it is customary to include the *softmax* in the model, such that

$$\log p_{\text{model}}(y|x; \theta) = \sum_{c=1}^C y_c \log(f_\theta(x)_c).$$  (2.1)
Neural networks will be introduced in the next section. This leads to the categorical cross entropy loss
\[
L(X, Y; \theta) = -\sum_{i=1}^{n} \sum_{c=1}^{C} y_{i,c} \log(f_{\theta}(x_i)_c),
\]
where \(f_{\theta}(x_i)_c\) is the \(c\)'th component of \(f_{\theta}(x_i) \in \mathbb{R}^C\). To train a model \(f_{\theta}\) we wish to minimize the loss function as a function of \(\theta\).

Using gradient decent we decrease the loss of \(f_{\theta}\) by updating the initial parameters \(\theta_0\) by many applications of
\[
\theta_{t+1} := \theta_t - \lambda \nabla_{\theta} L(X, Y; \theta_t),
\]
where \(\lambda\) is the learning rate. The learning rate will be discussed shortly.

The shortcoming of this equation is, if \(n\) is very large, meaning there are many training examples, it takes a long time to compute a single step of gradient decent. To combat this, we note the following. As previously already assumed, the samples of \(X, Y\) are iid. Then it is possible to rewrite the loss as
\[
L(X, Y; \theta) = \sum_{i} L(x_i, y_i; \theta),
\]
and, as the \(\nabla\) operator is linear, we have
\[
\nabla_{\theta} L(X, Y; \theta) = \sum_{i} \nabla_{\theta} L(x_i, y_i; \theta).
\]
The learning rate \(\lambda \in \mathbb{R}_{>0}\) can be chosen arbitrarily, such that dividing the loss by \(n\) can later be compensated by the learning rate. Hence, looking at
\[
\frac{1}{n} \sum_{i=1}^{n} \nabla_{\theta} L(x_i, y_i; \theta),
\]
we see that this approximates the expected value of the gradient of the loss for a sample \(x, y \sim p_{\text{data}}\) drawn from the true data generating process. The key insight now is that instead of computing the whole gradient, we can simply compute a gradient which in expectation is correct. Stochastic gradient descent (SGD) samples \(x_i, y_i\) from the dataset and performs a gradient update based on
\[
\theta_{t+1} = \theta_t - \lambda \nabla_{\theta} L(x_i, y_i; \theta_t).
\]
While the problem of slow computation is overcome by this, the gradient now has a large variance, while in expectation is still correct. To reduce the variance, mini-batch gradient descent has become the norm, where a batch of \(n' < n\) samples \(X' := \{x'_1, \ldots, x'_{n'}\}, Y' := \{y'_1, \ldots, y'_{n'}\}\), is sampled from the data, and a gradient step is performed using
\[
\theta_{t+1} := \theta_t - \lambda \frac{1}{n'} \sum_{i=1}^{n'} \nabla_{\theta} L(x'_i, y'_i; \theta_t).
\]
Further improvements to the gradient step can be made by using Adam (adaptive moment estimation) [12] optimization. The algorithm used by Adam is shown in
Algorithm 1. Adam optimization. Given a dataset $X, Y$, a minibatch size $n'$ and parameters $\alpha$ (stepsize), $\beta_1, \beta_2 \in [0, 1)$ (exponential decay rates), $\epsilon > 0$ and initial model parameters $\theta_0 \in \mathbb{R}^N$, this algorithm returns optimized parameters $\theta_t$.

1: $m_0 \leftarrow 0 \in \mathbb{R}^N$
2: $v_0 \leftarrow 0 \in \mathbb{R}^N$
3: $t \leftarrow 0$
4: **while** $\theta_t$ not converged **do**
5: $t \leftarrow t + 1 \in \mathbb{N}$
6: $X', Y' \leftarrow$ new minibatches sampled from $X, Y$ of size $n'$
7: $g_t \leftarrow \nabla_\theta L(X', Y'; \theta_{t-1})$
8: $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
9: $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
10: $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
11: $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
12: $\theta_t \leftarrow \alpha \hat{m}_t / (\sqrt{\hat{v}_t + \epsilon})$
13: **return** $\theta_t$

Algorithm 1. Multiplication, division, squaring and taking the square root of a vector is always done pointwise in the algorithm. Instead of updating the parameters only by using the newest gradient, as was done in the previously discussed methods, it uses the first and second moment of the gradient.

Until line 8 this algorithm is identical to SGD with minibatches. In lines 8,9 the unbiased moment of the gradient is computed, and in lines 10,11 the unbiased second moment. Line 12 updates the parameters $\theta_t$, in a similar fashion to SGD, however it uses the moments instead of the pure gradient. This is the algorithm that will be used for optimization in Chapter 5.

The learning rate is a very important variable in the training of a model. Not only does it influence how much time and resources need to be spent to train a model $f_\theta$ to optimize the parameters, but it also impacts how good the final model is. If the learning rate is too small, it is possible for the model to be stuck in a local minimum of the loss in regards to $\theta$, because the steps it takes are too small to get out of the valley. On the other hand, a too large learning rate might be unable to make precise enough adjustments to learn effectively, and a larger learning rate still leads to a divergence and infinite gradients. Testing different initial learning rates was prevalent until recently, where [26] introduced cyclic learning rates. The key idea is to let the learning rate cycle between a minimal and maximal value through training, as the authors claim the short term negative effects of a too large learning rate are outweighed by the long term positive effects. To find the minimal and maximal values, short training runs are performed with a wide range of learning rates, and then the minimal a maximal learning rate are chose to be the smallest and biggest learning rates which showed reasonable improvements of the model. This is called cyclic learning rate scheduling, and will be applied during model training in Chapter 5.
Figure 2.1: An example of the loss of the training set and evaluation set of a typical training run. The model is at its best when the evaluation loss is the lowest.

Model Evaluation

In most supervised learning problems a dataset is split into a training and evaluation set, with about 80% of the data in the former. While training the model may only see the training portion of the dataset. When evaluating the model, the evaluation set is used. The advantage of this is that models do not have the option of learning examples by heart, since the evaluation set is hidden during training. What usually occurs during training is that models overfit on a dataset, meaning the loss on the training dataset continuously gets smaller, however the loss on the validation set gets larger. This means that the model prefers learning features by heart as opposed to understanding general patterns in the data. This can be spotted in Figure 2.1. Many techniques exist which try to combat the overfitting behavior of models, some of which are discussed in the next section.

Looking at a models loss is usually not the most informative metric of how good it is. As we will focus only on multiclass classification problems, we will instead look at a models accuracy. It is defined as the ratio between correctly labeled examples to all examples,

$$\text{acc} = \frac{\text{nr. correctly labeled examples}}{\text{nr. examples}},$$

and is a number between 0 and 1.

Neural network fundamentals

Within recent years Neural Networks have found large success, and even set new benchmarks in many areas of machine learning tasks, including image classification, natural language processing, beating the best chess engines and even performing well in online real time strategy games such as Dota 2 and Starcraft. Machine learning employing Neural Networks is often refereed to as deep learning, as neural networks have many layers stacked on top of each other.
An overview of neural networks

A neural network (NN) is a function \( f : \mathbb{R}^d \to \mathbb{R}^C \), which is composed of several functions \( f^{(1)}, \ldots, f^{(L)} \), called layers,

\[
f(x) := f^{(L)} \circ \cdots \circ f^{(1)}(x)
\]

where we define the input as \( h^{(0)} := x \), the output as \( \hat{y} := h^{(L)} := f(x) \) and the hidden activations \( h^{(l)} := f^{(l)}(h^{(l-1)}) \) recursively. All layers share a similar structure, namely

\[
f^{(l)}(h^{(l-1)}) := \sigma^{(l)}(W^{(l)}h^{(l-1)} + b^{(l)}),
\]

where \( W^{(l)} \) is the \( l \)th layers weight matrix, \( b^{(l)} \) its bias and \( \sigma^{(l)} \) its activation function. The dimensions of \( W^{(l)} \) and \( b^{(l)} \) depend on the specific setup of each layer. We define these dimensions via the hidden activations. Let \( d_l \in \mathbb{N} \) be such that \( h^{(l)} \in \mathbb{R}^{d_l} \). The activation function is usually a component wise applied non-linear function. As is evident, without the non-linearity the whole neural network would be a composition of affine linear transformations, which itself would be an affine linear transformation again. They will be further discussed in a following section. Notice finally the slight abuse of notations as \( L \) is both used for the loss and the number of layers in a NN. For a multiclass classification problem it is most common to use a softmax function as the final activation function \( \sigma^{(L)} \).

Backpropagation

To use SGD or Adam optimization to optimize a neural network, we need to find the gradient of the loss function. For brevity we define the parameters \( \theta := \{W^{(i)}, b^{(i)} | i \in \{1, \ldots, l\}\} \) of a NN to be the collective of all weights and biases, and identify the terms \( L(X, Y; \theta) = L(X, Y; \{W^{(i)}, b^{(i)} | i \in \{1, \ldots, l\}\}) \). In the following, we write \( f_\theta \) for the NN \( f \) with parameters \( \theta \). To perform the computation of \( \nabla_\theta L(x, y; \theta) \) for a neural network \( f_\theta \) and a datasample \( x, y \), the most widespread approach is to use gradient backpropagation. This is an algorithm where each weight and bias of a NN \( f_\theta \) is computed efficiently, as we will see. The key building blocks of backpropagation are the chain rule, and reusing previously calculated partial gradients. Using \( L \) as defined in (2.1), we see

\[
\nabla_\theta L(x, y; \theta) = -\sum_{c=1}^{C} y_c \nabla f_\theta(x)_c = -\sum_{c=1}^{C} \frac{y_c}{f_\theta(x)_c} \nabla f_\theta(x)_c,
\]

such that we can now focus on the derivatives of \( f_\theta \). To simplify the notation we will write \( z^{(l)} := W^{(l)}h^{(l-1)} + b^{(l)} \) and remember that \( f_\theta(x)_j = h_j^{(L)} = \sigma_j^{(L)}(z^{(L)}) \).

Let us look at the derivative of weights \( w^{(L)}_{ik} \) in the last layer, and compute

\[
\frac{\partial f_\theta(x)_j}{\partial w^{(L)}_{ik}} = \frac{\partial}{\partial w^{(L)}_{ik}} \sigma_j^{(L)}(W^{(L)}h^{(L-1)} + b^{(L)}),
\]

\[
\frac{\partial f_\theta(x)_j}{\partial w^{(L)}_{ik}} = \sum_{i=1}^{C} \frac{\partial h_j^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial w^{(L)}_{ik}} = \sum_{i=1}^{C} \frac{\partial \sigma_j^{(L)}}{\partial z_i^{(L)}} (z^{(L)}) h_k^{(L-1)}. 
\]
For layer $l$, we have
\[
\frac{\partial h^{(l)}}{\partial w_{ik}^{(l)}} = \sum_{i=1}^{d_l} \frac{\partial h^{(l)}}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ik}^{(l)}},
\]
\[
\frac{\partial h^{(l)}}{\partial b_k^{(l)}} = \sum_{i=1}^{d_l} \frac{\partial h^{(l)}}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial b_k^{(l)}},
\]
because $\frac{\partial z_i^{(l)}}{\partial b_k^{(l)}} = 1$. The formula
\[
\frac{\partial h^{(l)}}{\partial h^{(l-1)}} = \frac{\partial h^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial h^{(l-1)}}
\]
grants the name backpropagation. By composing these rules we can find the derivative of the loss with respect to any parameter in $\theta$. Let us, for example compute
\[
\frac{\partial h^{(L)}}{\partial w_{ik}^{(L-2)}} = \frac{\partial h^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial h^{(L-1)}} \frac{\partial h^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial h^{(L-2)}} \frac{\partial h^{(L-2)}}{\partial w_{ik}^{(L-2)}}
\]
\[
= \frac{\partial h^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial h^{(L-1)}} \frac{\partial h^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial h^{(L-2)}} \sum_{i=1}^{d_{L-2}} \frac{\partial h^{(L-2)}}{\partial z_i^{(L-2)}} \frac{\partial z_i^{(L-2)}}{\partial w_{ik}^{(L-2)}}.
\]
As is evident, the matrices $\frac{\partial h^{(l)}}{\partial z^{(l)}}$ and $\frac{\partial z^{(l)}}{\partial h^{(l-1)}}$ are used over and over again, so an efficient implementation of this algorithm is paramount.

**Activation functions**

Let us now come back to the activation function. When choosing an activation function like the sigmoid, which maps $x \mapsto \frac{e^x}{1+e^x}$ or the tangens hyperbolicus, mapping $x \mapsto \frac{e^x-e^{-x}}{e^x+e^{-x}}$, which both have derivatives in $(0, 1)$, we see that the long product of their derivatives leaves a small number. This means that for a network with many layers, the values of $\frac{\partial h^{(L)}}{\partial w_{ik}^{(L-2)}}$ will get smaller as $L - l$ increases. When performing a step of SGD, the weights will barely be changed. This problem is referred to as the vanishing gradient problem, and several solutions have been proposed, such as Highway Networks [28], Residual Networks [8], and different activation functions. One of those is to use an activation function which doesn’t suffer from small derivatives, such as the rectified linear unit [22],

ReLU : $\mathbb{R} \rightarrow \mathbb{R}$,

$x \mapsto \max\{0, x\}$,

or the exponential linear unit [1]

ELU : $\mathbb{R} \rightarrow \mathbb{R}$,

$x \mapsto \begin{cases} 
  x & \text{if } x > 0, \\
  \alpha(e^x - 1) & \text{if } x \leq 0.
\end{cases}$
Both solve the problem of vanishing gradients, however the ReLU function leads to dead neurons. When the input to ReLU is negative, its output is 0, and so is its gradient. This means that the learning progress is also stopped, hence the name. The activation function ELU was proposed as a solution to this issue. Because the functions are not symmetric, a new problem by the name of bias shift arises, and will be discussed in a following section.

We will mainly be using ELU as an activation function, because other experiments in this area of research use it. This helps to make the results found here and those of other research comparable.

**Convolutional layer**

A special type of layer of a NN is a convolutional layer. A network which is primarily made up of these layers is called a convolutional neural network (CNN)[16]. Its main use of application is on structured data, such as images, language or audio input. It is called a convolutional layer, because a set of filters \( \{F_1, \ldots, F_m\} \) are each convolved with the input \( h^{(l-1)} \). In the following we extend the previous definition of a layer \( f^{(l)} \) by allowing higher dimensional in- and output. From now on the input to a network is a matrix. As such we will refer to a single datasample as \( X, y \), as opposed to \( x, y \). Before getting to the definition of convolution, let \( X \in \mathbb{R}^{d_1 \times d_2} \), and define the partial matrix

\[
X_{[i_1:i_2, j_1:j_2]} := \begin{bmatrix}
x_{i_1,j_1} & \cdots & x_{i_1,j_2-1} \\
\vdots & \ddots & \vdots \\
x_{i_2-1,j_1} & \cdots & x_{i_2-1,j_2-1}
\end{bmatrix}.
\]  

(2.3)

Further \( X_{[i]} := X_{[i,d_1+1,1:2]} \), which means to say a colon represents the whole dimension, and an integer means only a single slice of the dimension. Let

\[
\otimes : (\mathbb{R}^{d_1 \times d_2}, \mathbb{R}^{d_1 \times d_2}) \to \mathbb{R}^{d_1 \times d_2}; \\
A \otimes B \to \sum_{ij} a_{ij}b_{ij}
\]

be the tensor contraction.

In the case of the input to the layer being a 2D grayscale image, \( h^{(l-1)} \in \mathbb{R}^{d_{1,1} \times d_{1,2}} \), the filters \( F_i \) too are in \( \mathbb{R}^{f_{1,1} \times f_{1,2}} \). The convolution of \( F_i \) with \( h^{(l-1)} \) is defined as

\[
F_i, h^{(l-1)} \mapsto (F_i \otimes h^{(l-1)}_{[j:j+f_{1,1},k:k+f_{1,2}]})_{j,k}
\]

(2.4)

for all \( j, k \) such that the partial matrix is defined. Then

\[
\text{conv2d} : \mathbb{R}^{f_{1,1} \times f_{1,2} \times m}, \mathbb{R}^{d_{1,1} \times d_{1,2}} \to \mathbb{R}^{(d_{1,1} - f_{1,1} + 1) \times (d_{1,2} - f_{1,2} + 1) \times m},
\]

\[
\{F_1, \ldots, F_m\}, h^{(l-1)} \mapsto ((F_1 \otimes h^{(l-1)}_{[j:j+f_{1,1},k:k+f_{1,2}]})_{j,k}, \\
\ldots, (F_m \otimes h^{(l-1)}_{[j:j+f_{1,1},k:k+f_{1,2}]})_{j,k}),
\]

which is just convolving each filter independently with the input. This definition is best understood by looking at Figure 2.2. It can be easily adjusted to work on
Figure 2.2: A visualization of how a filter $F_1$ is convolved with an input with 2 dimensions. The output for this filter is the same size as the input image, as the padding=\texttt{same} is considered. The elements of the filter $F_1$ are first multiplied with those elements of the input which are directly below the filter, and then added. The result is saved in the output, symbolized by a circle.

higher, or lower dimensional input, such as one dimensional audio inputs in $\mathbb{R}^{d_{l-1}}$ (\texttt{conv1d}) or arbitrary tensors in $\mathbb{R}^{d_{l-1}\times\cdots\times d_{l-1}, m}$ (\texttt{convmd}).

As a layer in a neural network we can define $f^{(l)} = \sigma^{(l)}(\text{conv2d}(h^{(l-1)}) + b^{(l)})$, where we omit the filters in notation. The elements in the filters are considered part of the parameters $\theta$, such that they can be optimized using SGD. There are other definitions of \texttt{convmd}, where not just the partial matrices are considered that were defined in (2.3), but also those such that filters overlap the input matrix. In the case where filters overlap the matrix, we fill the undefined matrix indices (for example $x_{0,0}$) with zeros. This is considered as the \textit{padding}. What was defined in (2.4) is called \textit{valid}, while padding such that $\text{conv2d} : \mathbb{R}^{f_{1,1}\times f_{1,2}\times m}, \mathbb{R}^{d_{l-1}\times d_{l-1}, 2} \rightarrow \mathbb{R}^{d_{l-1}\times d_{l-1}, 2\times m}$ is called \textit{same} padding. Another variation adds a \textit{stride} to the convolution, which means that a filter is only applied to some of the partial matrices. In particular, a stride $s \in \mathbb{N}$, would result in a convolution with a filter $F_i$

$$ \left( F_i \otimes h^{(l-1)}_{[js+fs,ks+ks]} \right)_{j,k}. $$

This obviously changes the output size of the $\text{conv2d}$ layer. The output of the convolutional layer $f^{(l)}$ is made up of $m$ feature maps.

Convolutional NNs are some of the most successfully NN architectures, mainly due to their nature of acting locally, and being much more parameter sparse than dense layers, which are introduced shortly.

\textbf{Pooling layer}

Another important variation of a layer is the \textit{pooling} layer. This layer reduces the size of the input. It too considers partial matrices and commonly returns the maximum, sum or average of the partial matrix. Concretely, for a 2 dimensional input, a stride of 2 and a pool size of $(3,3)$, and with max as a pooling function,

$$ \text{pool2d} : \mathbb{R}^{d_{l-1,1}\times d_{l-1,2}} \rightarrow \mathbb{R}^{d_{l,1}\times d_{l,2}}, $$

$$ h^{(l-1)} \mapsto (\max h^{(l-1)}_{[2j+3,2k+3]})_{j,k} $$
for all \(j, k\) such that the partial matrix is defined. As with the conv2d layer, the stride, pool size and pooling function are omitted from the notation. In case the dimension of the input is not neatly tiled by the pooling filters, \texttt{valid} would drop the last, incomplete partial matrix, and \texttt{same} would pad the incomplete partial matrix with zeros. Pooling layers are usually applied directly after convolutional layers. In that case, similar to how convolutions of different filters work independently of one another, the pooling works independently on the output of the \(m\) filters:

\[
\text{pool2d} : \mathbb{R}^{d_{l-1} \times d_{l-1}} \rightarrow \mathbb{R}^{d_l \times d_l},
\]

\[
(h^{(l-1)}) \mapsto \left(\max_{j,k} h^{(l-1)}_{[2j:2j+3,2k:2k+3,1]}j,k, \ldots, \max_{j,k} h^{(l-1)}_{[2j:2j+3,2k:2k+3,m]}j,k\right).
\]

Finally, it is important to note that a pooling layer does not have any parameters, however as a layer \(f^{(l)} : \sigma^{(l)}(\text{pool2d}(h^{(l-1)}))\) it still plays a role in the derivative of the loss.

### Dense and flatten layers

In a CNN it is common to have the final few layers of the network to be \textit{fully connected layers}, often called \textit{dense layers}. These are very similar to how layers were defined in (2.2). However, when the input is a feature map, it is common to consider the feature maps independent of each other

\[
dense : \mathbb{R}^{d_{l-1} \times m} \rightarrow \mathbb{R}^{d_l \times m},
\]

\[
h^{(l-1)} \mapsto (W_1^{(l)} h^{(l-1)}_{[1:1]}, \ldots, W_m^{(l)} h^{(l-1)}_{[m:]})
\]

where the matrices \(W_1^{(l)}, \ldots, W_m^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}\). To combine all the features a \textit{flatten layer} is used,

\[
\text{flatten} : \mathbb{R}^{d_l \times m} \rightarrow \mathbb{R}^{d_l m},
\]

\[
h^{(l-1)} \mapsto (h^{(l-1)}_{[1:1]} | \ldots | h^{(l-1)}_{[m:]})
\]

where | corresponds to a concatenation. This layer too does not have any parameters.

A stack of a few dense layers is abbreviated as a multilayer perceptron (MLP). We write \texttt{mlp}(\(a_1, \ldots, a_n\)) for a MLP with \(a_i\) hidden activations in the \(i^{\text{th}}\) layer.

### Batch normalization

The whitening of the input data has many advantages when training a neural network [15], and other machine learning algorithms as well. We wish to extend this to the inner layers of a neural network, especially since functions like ReLU and ELU are non symmetric. To do so we use Batch Normalization [9], which combats internal covariance shift on a feature by feature basis. In particular, the \texttt{BatchNorm} layer takes as input a mini batch of inputs \(x_1, \ldots, x_m\), computes the mean

\[
\mu := \frac{1}{m} \sum_{i=1}^{m} x_i
\]
and variance

\[ \sigma^2 := \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu)^2. \]

The layers input is then normalized,

\[ \hat{x}_i := \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \]

and scaled and shifted by learnable parameters \( \gamma \) and \( \beta \),

\[ y_i := \gamma \hat{x}_i + \beta. \]

The derivatives of \( \gamma \) and \( \beta \) are shown in [9], which are important for the backpropagation step and their optimization.

**Regularization**

By *regularization* we refer to the techniques of changing a model with the goal of having it perform better on the validation test, but accepting that it usually performs worse on the training set or takes longer to learn. Common techniques are weight regularization like L1 and L2 regularization, which try to limit the size of the weights. Dropout [27] randomly sets some hidden activations to zero, forcing the model to be able to cope with partial information only. Other techniques focus on the training data, distorting it by random rotations, scaling or other transformations.

**Rotation invariant convolutional filters**

A map \( f \) is considered *invariant* under the transform \( \rho \), if \( f(x) = f(\rho(x)) \). When \( f \) is the NN and \( \rho \) is a rotation, shift or scaling transform, then it is natural that one wishes \( f \) to be invariant under \( \rho \). In practical terms, turning an image by 30 degrees does not alter the content of the image, such that the image should still be classified the same. In Figure 2.3 the filters of AlexNet [14] are shown. One can see that many filters are very similar, but rotations of one another. Rotation invariance of NN has been studied in the past [7], [30], [20] for a variety of tasks. In essence [20] use fewer filters than other models, reducing the amount of parameters which need to be learned substantially. Each filter is then applied many times to the same

Figure 2.3: Figure taken from [14].
domain, however always under different rotations. The rotations are in this case at $22.5^\circ$, spanning the whole circle. The benefits of this are manifold, for example training speed is improved, and rotational invariance is achieved.

While [20] uses all outputs of the different rotations of 2D images, [21], working on 2D manifolds embedded in 3D space, only uses the maximum activation, i.e. only that rotation which gave the biggest output for a given filter. This is more in line of the contributions of this work, and will be further discussed in Section 4.1.

In [10] the Radon transformation was used to align a patch of the image along a principal direction. This means that not the filter was rotated, but rather the underlying data. According to the authors this works well when the underlying patches are anisotropic, which is commonly the case for textures. However, as [20] also point out, the approach lacks consistency for isotropic textures and patches where there are more than one principal direction. Albiet the criticisms this approach is closest to the one being used here. The underlying data for [10] consisted of 2D Images, and here will be 3D point clouds. Section 4.1 will discuss why the shortcomings do not extend to 3D point clouds.

**Graphs**

The data which will be studied in the following sections are point clouds in $\mathbb{R}^3$, which is a collection of vectors $x_i \in \mathbb{R}^3$. We write $X \in \mathbb{R}^{3 \times n}$, where each column of $X$ corresponds to the cartesian coordinates of a point in the point cloud. Depending on context we consider $X$ to be the set $\{x_1, \ldots, x_n\}$, or the matrix in $\mathbb{R}^{3 \times n}$. Two structured approaches of these points will be explored, namely voxel grid and graph representations. In the later stages we will only work with the graph representation, however the voxel representation is very instructive in highlighting the differences between ordered 2D image data and 3D point cloud data, and how deep learning is done on these data types.

**Voxel grid**

A voxel grid of a point cloud $X \in \mathbb{R}^{3 \times n}$ is a multi dimensional matrix $V \in \mathbb{N}^{d \times d \times d}$, where $\delta := 1/d$ is the size of the voxel $v_{ijk}$. For a point cloud with $x \in [0, 1] \times [0, 1] \times [0, 1]$ for all $x \in X$, the elements of $V$ are given by

$$v_{ijk} = \text{sgn}\left(\{x \in X\} \cap [\delta(i - 1), \delta i] \times [\delta(j - 1), \delta j] \times [\delta(k - 1), \delta k]\right)$$

$$= \begin{cases} 0, & \text{if the voxel is empty} \\ 1. & \end{cases}$$

In words, a voxel is activated, i.e. $v_{ijk} = 1$, if a point is inside it, otherwise it is zero. Compared to a 2D image, where the information is dense, a voxel grid is usually sparse, a problem discussed further in Section 3.3. Furthermore the number of voxels increases dramatically as $d$ increases, due to the curse of dimensionality. This means that in practice $d$ is kept small, leading to coarse representations of the underlying data, as can be seen in Figure 2.4. In Figure 2.4a we can see that
A voxel representation of a cube. The voxel grid was created by \( n = 512 \) points sampled uniformly from the surface of a cube, and there are \( d = 32 \) voxels along each axis.

Another voxel grid made from points sampled from a cube, where \( n = 2048 \) and \( d = 8 \).

Figure 2.4: Two voxel grids.

the low number of points, \( n = 512 \), and the high number of voxels, \( 32^3 \), leads to a representation that lacks many voxels. In Figure 2.4b we can see that all voxels are filled. However the representation is so coarse, that the object in question could be a cylinder or a cube.

Graph representation

Another representation of a point clouds structure is by capturing it in a graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{F}) \), where \( \mathcal{V} = \{v_i\}_{i=1}^n \) is a set of vertices, s.t. \( v_i = x_i \). The edges \( \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \) indicate which vertices are adjacent to one another, and we call \( \mathcal{N}(x_i) \) the neighbors of \( x_i \), which is the set of all adjacent nodes of \( x_i \). The vertex features \( \mathcal{F} \subseteq \mathcal{V} \times \mathbb{R}^f \) describe the vertices further. In notation we write \( f_{v_i} \) for the vertex features of a vertex \( v_i \). Possible features of a vertex could be red-green-blue (RGB) colors, in which case \( f = 3 \), or more generally, the output features of a convolutional layer. This concept will be explored in Section 4.3. A common way to chose which vertices are adjacent to one another in this field is to use the \( k \)-nearest neighbors (\( k \)-NN) algorithm, as done by [3], [24] and [29], among others. The \( k \)-NN algorithm works as follows. For a point \( v_0 \in \mathcal{V} \), let \( v_1, \ldots, v_n \) be a reordering of the points in \( \mathcal{V} \setminus \{v_0\} \), such that

\[
\|v_0 - v_1\|_2 \leq \|v_0 - v_2\|_2 \leq \cdots \leq \|v_0 - v_n\|_2.
\]

The \( k \in \mathbb{N} \) nearest neighbors of \( v_0 \) are the points \( v_1, \ldots, v_k \). We write \( k\text{-NN}(v) \) for the set (or matrix, depending on context) of the \( k \)-nearest neighbors of \( v \). In notation, \( \mathcal{V} \) is omitted.
(a) A point cloud representation of a triangular pyramid. There are \( n = 2048 \) points sampled uniformly from the surface. The green dot in the bottom left corresponds to \( v_0 \) and the orange points are its 50 nearest neighbors.

(b) A voxel representation of a triangular pyramid with \( n = 2048 \) and \( d = 16 \).

Figure 2.5: The point cloud representation with \( k\)-NN(\( v_0 \)) shown, and the voxel grid of the same triangular pyramid.

The edges \( E \) of a graph \( G \) are then defined by

\[
(v_i, v_j) \in E \iff v_j \in k\text{-NN}(v_i),
\]

unless otherwise stated.

In Figure 2.5 an example of a graph representation is shown, with its voxel grid counterpart. In particular it is of interest, that the voxel representation in Figure 2.5b needs \( 16^3 = 4096 \) voxels, however the much more precise representation in Figure 2.5a only needs \( n = 2048 \) points.
Chapter 3
Specialized Theory

This chapter explains two approaches of how deep learning has been applied to point clouds in the past. The three most common approaches are using a voxel representation of the 3D point cloud, converting the 3D point cloud to a graph and then using the Laplacian matrix, and using a message passing architecture. The approach of using the Laplacian has several drawbacks in this particular domain of 3D object classification, and will not be discussed further.

While convolutional neural networks on the voxel representation of a 3D point cloud are simple extensions to the CNN defined in Section 2.3, problems arise due to the nature of the underlying data. These will be explored, and hence justify why message passing networks are currently heavily studied. After the exploration of its drawbacks, voxels will not be visited again.

With the introduction of PointNet [24] in 2016 a new area of research has opened. PointNets architecture was the primary source of inspiration for the architecture of PointCNN and DGCNN, the networks most closely linked to the model which will be defined in Section 4.5 and studied in section Chapter 5. Other networks, such as SO-Net [18] and SplineCNN [6] try to extend the idea of convolution and pooling to graph based deep learning. Deep learning on such graphs is refereed to as Geometric Deep Learning.

Finally the datasets which are explored will be introduced, and the results of other state of the art models on these datasets are presented.

Problems with point clouds

This section explains the difficulties of extending the highly successful neural networks from two dimensional input images to three dimensional input point clouds. First we will look at images and how their information is stored and processed in a neural network. A single pixel of an image is made up of channels, where only one channel is used for grayscale images or three are used for a RGB image. These channels can typically take values between [0, 1] or integers values in the range of [0, 255], called intensities. The location in 2D space of a pixel is implicitly given by where it is saved in memory, in particular, the pixels occupy a regular grid. It is not possible for pixels to lie outside of this grid. When computing, for example, a convolution between a filter $F_1$ and the input image $h^{(0)}$, the pixel intensity is
multiplied by the correct entry of the filter, according to 2.4. For a pooling layer the implicit location of the pixels is also very important.

In contrast to structured image data, a point cloud exhibits completely different types of information. In the following chapters, the input to a model will always be a point cloud with zero features attached to each vertex, meaning there is no color or other information, just the location. Instead of having an intensity with implicit location, a point simply exists at a specific location in 3D space. Further more, its position can be arbitrary, it is not locked into a fixed grid. In an image each pixel is occupied, meaning its full of information. The 3D space surrounding the point cloud is mainly empty, as can be seen visually in Figure 2.5a. Finally, the point cloud is unordered. In contrast to images, where it is clear which pixel is left or right from another pixel, in a 3D point cloud we are not afforded this information. Indeed, it is impossible to order the whole 3D cartesian space.

All of these differences make it difficult to apply the highly successful techniques developed for CNNs to point cloud data in a straight forward fashion.

Requirements of a model

When classifying images, there are some desirable properties which we wish our model to have. These are, for example, that if an input image is mirrored, then the prediction of what is depicted in this image should not change. Similarly, we have some properties which we wish any geometric deep learning model to satisfy. These include

- **Rotational invariance**: Any rotation performed on the point cloud should not influence the prediction,

- **Scale invariance**: Any scaling (uniform in all directions) of the point cloud should not influence the prediction,

- **Shift invariance**: Any shifting of the point cloud should not influence the prediction,

- **Invariance under point order**: The order of which the points are listed in the point cloud should not influence the prediction.

Only the fourth item is unique to 3D point cloud data, and would not arise in a structured dataset like 2D images or 3D voxels.

Voxel approach

Before getting into the actual topic of this thesis, we wish to motivate why a voxel approach is not always suited to deep learning on 3D point cloud data. Some of the disadvantaged were already alluded to in the previous chapter, while introducing voxels in Section 2.4.

After converting the point cloud \( X \) to a voxel grid \( V \), we can conduct convolutions with 3D filters, just as explained in Section 2.3. One problem which arises here is
that the voxel grid is only filled with zeros and ones. This binary data makes it much harder for the convolutions to generalize. Furthermore, most voxels in a voxel grid are zero for any given shape. This leads to large computational overhead where filters are effectively multiplied with zeros. This also leads to excessively large training times, because most parts of the gradient vector are zero. These and other problems are discussed in [25]. OctNet [25] tries to overcome some of these issues by having voxels of different size, depending on the complexity of the data within the voxels. An image representation of this is show in Figure 3.1a. Kd-Net [13] extends on this idea by separating the 3D space into continuous sized rectangles, building a tree. This is shown for a 2D point clouds in Figure 3.1b.

Even though this is better, it still does not comply to all of our requirements, as discussed in section 3.2. Namely rotational invariance is very hard to achieve for models working on voxel grids. For images in 2D this is achieved, or rather approximated, by rotating the input image randomly. However in 3D there are two dimensions of freedom for rotation, making this approach very difficult. Furthermore, scale invariance is not accounted for. Shift invariance can be achieved easily by using convolutional layers, just as in 2D. One advantage of this approach is that invariance under point order is a given, as the point cloud is converted into a voxel grid.

**Geometric deep learning**

**PointNet**

The emergence of PointNet [24] and its application of deep-learning methods to unstructured point clouds opened a new field of research. In the years following its publication, many classification models took inspiration from PointNet, and tried to overcome some of its limitations. After introducing PointNet its limitations are discussed and we take a look at some of its successors.

PointNet is best introduced by explaining its layers, represented in Figure 3.2. The input is a point cloud, where the rows and columns are flipped to how it was
defined in Section 2.2. The reason for this is that in machine learning it is common to have a data sample span a row, while in linear algebra the convention is to have a data point as a column vector. As we combine SVD with machine learning, these conventions clash, however require only to transpose the point cloud matrix $X$. For PointNet, $n = 1024$ points are sampled from the surface of a CAD model, and normalized into the unit sphere. As a final preprocessing step, each point has a Gaussian white noise with zero mean and standard deviation of 0.02 added to it. The CAD models are from ModelNet [31], introduced in the next section.

In the first step, the points are fed into an input transform. This layer first computes a rotational matrix and then transforms all 1024 points. The rotational matrix is output by the T-net, a neural network with the goal of finding a rotation which rotates the points into a canonical space [24]. This means, that no matter the initial orientation, after this transform the objects rotation is always the same. This layer tries to emulate rotational invariance. After the rotation each point is individually fed into a mlp which extracts 64 features from the point. A criticism of this is, that this layer does not take advantage of the local surrounding of the points, which has proven very useful in CNNs [29]. Another input transform transforms the features in 64 dimensions into a canonical space again, and a further mlp with 1024 outputs is used for feature extraction. Maxpooling is applied over these 1024 features for the $n = 1024$ points. The 1024 features are then fed into a final mlp with $C$ outputs, where $C$ is the number of different labels in the dataset. The argmax of the output corresponds to the predicted label. For training the network a softmax layer is added, weight regularization and dropout are introduced, and categorical cross entropy loss is used with an Adam optimizer. The finer details can be found in [24], and the training data and performance of the model are discussed in the following sections.

Apart from not leveraging the benefits that local convolutions offer in two dimensions, local pooling also does not exist in the model. The authors are aware of this, and have subsequently presented PointNet++ [23], combating some of PointNets weaknesses. We don’t go into PointNet++, as it is irrelevant for us.
Edge convolution

We introduce now the Dynamic Graph CNN (DGCNN) [29] which tackles the shortcomings of PointNet by extending convolutions to graphs.

The layers of the network fall under the category of a message passing layer (MPL). In a MPL the features of a vertex are sequentially updated by its neighbors. While possible, it was not necessary to consider the point cloud as a graph for PointNet, as it does not consider edges. In the following it will be necessary. A MPL is a layer which transforms a graph $G^{(l)}$ to $G^{(l+1)}$. For ease of notation we write $G$ for $G^{(l)}$.

To this extent, let $G$ be an input graph with vertices, edges and features, and $G^{(l+1)}$ be the output graph, with $V^{(l+1)} := V$, and its edges and features defined shortly. Let $h_\theta$ be an edge function, $h_\theta : V \times V \times F \times F \to \mathbb{R}$, $(v_i, v_j, f_{v_i}, f_{v_j}) \mapsto e_{i,j}$, which takes as input a pair of vertices and its features, which are connected by an edge. We call the output an edge feature, as it describes the connection between the two vertices. See Figure 3.3 for a visualization. After computing all edge features, they are propagated into the central node. To this end, let $\Box$ be an aggregation function, such as max, mean or sum, which operates pointwise. For example, with $\Box = \sum$, 

$$f_i' := \Box_{v_j \in N(v_i)} h_\theta(v_i, v_j, f_{v_i}, f_{v_j}) = \sum_{v_j \in N(v_i)} e_{i,j}.$$ 

These features are further transformed by another parametric function $\gamma_\phi$, and stored in the output graph $G^{(l+1)}$:

$$F^{(l+1)} \ni f_{v_i}^{(l+1)} := \gamma_\phi(f_i').$$

Finally the edges of $G^{(l+1)}$ are for now left undefined, however are typically defined by $k$-NN, either for the vertices $v_i \in \mathbb{R}^3$ or its features $f_{v_i} \in \mathbb{R}^m$ (see (3.1)), as done by [29]. This describes fully how many state of the art models implement the concept of convolution on graphs.

A MPL satisfies invariance under point order. As there is still some freedom in the choice of $h_\theta$ and $\gamma_\phi$, we can also enforce rotational, scale and shift invariance, as we will see later.

The network of DGCNN is depicted in Figure 3.4, and the main novelty is the EdgeConv layer. The EdgeConv layer is a MPL which takes as input a graph $G$ with $n = 1024$ vertices with features. The edges are defined by $k$-NN on the features, meaning

$$e_{i,j} \in E \iff f_{v_j} \in k\text{-NN}(f_{v_i}).$$ (3.1)

The edge function is given by

$$e_{i,j} := h_\theta(f_{v_i}, f_{v_j} - f_{v_i}).$$
Figure 3.3: On the left vertices $v_1, \ldots, v_5$ are neighbors of the central node $v_0$. The edges feature $e_{0,i}$ are computed with the edge function $h_{\theta}$, and visualized as red curves between the vertices. In the next step these edge features are aggregated with $\Box$, which is not visualized.

Figure 3.4: The architecture of DGCNNs classification network.

for a MLP $h_{\theta}$. The aggregation $\Box = \sum$ is the sum function, and $\gamma_{\theta} = \text{id}$. Hence

$$f_{v_i}^{(l+1)} = \sum_{v_j \in \mathcal{N}(v_i)} h_{\theta}(f_{v_i}, f_{v_j} - f_{v_i}).$$

This network is able to capture the local information for a vertex because it convolles over its neighbors and considers their difference in feature space, $f_{v_j} - f_{v_i}$, but simultaneously keeps global information due to the term $f_{v_i}$ in $h_{\theta}$.

The first layer of DGCNN is an input transform, just like in PointNet. Next there are two EdgeConv layers. As the input graph has no vertex features, the authors of [29] set $\mathcal{F}^{(0)} := \mathcal{V}^{(0)}$, i.e. the coordinates are used as features. The following layers are similar as those in PointNet again. This allows for a valid comparison of the two models.

SplineCNN

The network SplineCNN [6] has been applied to a variety of tasks in [6], however its application to classification is not comparable to the task covered here. As such, the network as a whole will not be presented, but rather its SplineConv MPL. The base building block of it are $B$-spline functions, which are defined as follows.
The one dimensional basis B-splines with $K + 1 \in \mathbb{N}$ supports are defined by

$$B_k : [0, 1] \rightarrow \mathbb{R},$$

$$x \mapsto \begin{cases} 0 & \text{if } x \notin \left[ \frac{k-1}{K}, \frac{k+1}{K} \right], \\ x - \frac{k-1}{K} & \text{if } x \in \left[ \frac{k-1}{K}, \frac{k}{K} \right], \\ \frac{k+1}{K} - x & \text{if } x \in \left[ \frac{k}{K}, \frac{k+1}{K} \right], \end{cases}$$

for each $k \in \{0, \ldots, K\}$. The $d$ dimensional basis functions are then defined for each tuple $(k_1, \ldots, k_d)$ as

$$B_{(k_1, \ldots, k_d)} : [0, 1]^d \rightarrow \mathbb{R},$$

$$(x_1, \ldots, x_d) \mapsto \prod_{i=1}^d B_{k_i}(x_i),$$

and with a $d$ dimensional matrix $W \in \mathbb{R}^{K+1 \times \cdots \times K+1}$ used to index the weights of the basis splines, a B-spline

$$B_W : [0, 1]^d \rightarrow \mathbb{R},$$

$$(x_1, \ldots, x_d) \mapsto \sum_{w_{k_1,\ldots,k_d} \in W} w_{k_1,\ldots,k_d} B_{(k_1,\ldots,k_d)}(x_1, \ldots, x_d)$$

is defined. A closed B-spline is one where the edge of the spline is zero:

$$x \in \partial [0, 1]^d \Rightarrow B_W(x) = 0,$$

which is the case if

$$\exists i : k_i = 0 \text{ or } k_i = K \Rightarrow w_{k_1,\ldots,k_d} = 0.$$

Refer to Figure 3.5 for an example of a closed B-spline and a basis spline. These are linear B-splines, and can be extended to be made up of polynomials of higher degree, however [6] achieved best results with linear splines. This is why we limit ourselves to these as well. B-splines act as filters, similar to convolutional filters. We take a collection of weight matrices $W_1, \ldots, W_m$, and set

$$h_\theta(v_i, v_j, f_{v_i}, f_{v_j}) = \left( f_{v_j} B_{W_1}(v_i - v_j) \right), \ldots, \left( f_{v_j} B_{W_m}(v_i - v_j) \right),$$

where all the learnable parameters are in $\theta$. Notice that $v_i - v_j$ is not necessarily in $[0, 1]^d$, however [6] uses appropriate scaling and shifting. The aggregation method used is $\Box = \sum$. Finer details on SplineCNN can be found in [6].

### Pooling on graphs

In this section we will look at pooling methods on graphs, notably graclus [4] and farthest point sampling (FPS). Others exist, but will not be considered here.

The name graclus comes from graph clustering, and is a very simple method to reduce a graphs number of vertices. The unweighted variant described in [4] is described now. Let $G = (\mathcal{V}, \mathcal{E})$ be a graph without features. The aim is to find a new graph $G^{(l+1)}$ with fewer vertices, and with neither edges nor features. To do so we first define a matching function $\pi : \mathcal{V} \rightarrow \mathbb{N}$, which satisfies
1. for any $v \in \mathcal{V}$ there is at most one $w \in \mathcal{V}\{v\}$ for which $\pi(v) = \pi(w)$ holds,
2. for all pairs $v, w \in \mathcal{V}$ with $\pi(v) = \pi(w)$ we have that $v \in \mathcal{N}(w)$ and $w \in \mathcal{N}(v)$.

This means that $\pi$ only matches pairs, and pairs need to be neighbors. We call a vertex $v$ matched if there exists $w \in \mathcal{V}\{v\}$ such that $\pi(v) = \pi(w)$. Furthermore $\pi$ is maximal, if there are no two vertices connected by an edge which are both unmatched. The algorithm used to find such a matching $\pi$ is presented in Algorithm 2.

**Algorithm 2**

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ this algorithm returns a matching function $\pi$.

1: randomize the order of $\mathcal{V}$
2: for $v \in \mathcal{V}$ do
3: \hspace{1em} $\pi(v) \leftarrow \infty$
4: for $v \in \mathcal{V}$ do
5: \hspace{1em} if $\pi(v) = \infty$ then
6: \hspace{2em} if $\emptyset \neq \mathcal{N}(v) \cap \pi^{-1}(\{\infty\})$ then
7: \hspace{3em} \hspace{1em} $w \leftarrow \text{select}(\mathcal{N}(v) \cap \pi^{-1}(\{\infty\}))$
8: \hspace{2em} \hspace{1em} $\pi(v) \leftarrow \text{next}$
9: \hspace{2em} $\pi(w) \leftarrow \pi(v)$
10: return $\pi$

Line 1 of the algorithm makes sure that the matching is randomized. Lines 2 and 3 prepare the matching function. Lines 4 through 9 then perform the actual matching. We first select a vertex $v$ (line 4), making sure we select all vertices at some point. If this vertex is already matched (line 5), chose the next vertex. Otherwise, see if this vertex has an unmatched neighbor (line 6). The select function chooses arbitrarily one unmatched neighbor $w$ of $v$ (line 7), and the next function assigns the next natural number to $\pi(v)$ (line 8). The same number is assigned to $\pi(w)$ (line 9). The matching function $\pi$ returned (line 10) maps a vertex to an integer in $\mathbb{N}$ if it is matched, and to $\infty$ if it is unmatched.
Given a matching function \( \pi \) for a graph \( G \) we can define a new graph \( G^{(l+1)} \) with vertices \( V^{(l+1)} \) such that for any matched pair \( v, w \in V \) exactly one vertex is in \( V^{(l+1)} \). Alternatively one could take their mean, such that \( \frac{v + w}{2} \in V^{(l+1)} \). All unmatched vertices will be in \( V^{(l+1)} \). There are several ways to transfer the features \( F \) from \( G \) to \( G^{(l+1)} \). For unmatched vertices \( v \) the most natural way is to copy the feature \( f_v \) from \( F \) to \( F^{(l+1)} \). For matched vertices \( v, w \) an aggregation method \( \Box \), such as max or \( \sum \), is used. Thus \( \Box \{f_v, f_w\} \in F^{(l+1)} \).

The pooling layer \texttt{graclus} is then the mapping from \( G \) to \( G^{(l+1)} \), where the aggregation method is omitted from notation.

Farthest point sampling is an easier algorithm to understand. It takes as its input the vertices \( V \), meaning that it does not regard edges. After choosing the first point randomly, the following point chosen is the furthest point from all already chosen points. To determine the furthest point a metric is used which, in Algorithm 3, is denoted by \( d \). The algorithm stops once it has sampled a ratio \( r \in (0, 1) \) of all points. The algorithm is depicted in Algorithm 3.

### Algorithm 3

Given just the vertices \( V \) and a ratio \( r \in (0, 1) \), this algorithm returns \( V^{(l+1)} \), which has less vertices than the original \( V \).

1: randomize the order of \( V \)
2: \( V^{(l+1)} = \emptyset \)
3: chose randomly \( v \in V \), and add \( v \) to \( V^{(l+1)} \)
4: while \( \frac{|V^{(l+1)}|}{|V|} < r \) do
5: \( v \leftarrow \arg \max_{v \in V} d(v, V^{(l+1)}) \)
6: add \( v \) to \( V^{(l+1)} \)
7: return \( V^{(l+1)} \)

PointCNN

PointCNN[19] is another network which works similarity to DGCNN. Its main novelty is orthogonal to what is presented in this work, thus PointCNN will not be discussed fully. However, this network is a motivation of the layers presented in the following chapter, thus we discuss briefly its \( \mathcal{X} \)-Transform. Previously mentioned models heavily relied on an input transform to rotate the point cloud into a canonical 3D space. Here, the points are not rotated, but ordered. The component which I would like to highlight, is that the ordering happens on a per-neighborhood basis, meaning for each node all its neighbors are ordered. The ordering of points in a neighborhood is determined by the output of a neural network, hence the ordering is learned. Further detail can be found in [19].

PyTorch Geometric

PyTorch Geometric\(^1\) [5] is a deep learning framework built on top of PyTorch, which is able to handle many types of geometric data. It is used extensively in this work, and without it this work would not have been possible.

\(^1\)https://github.com/rusty1s/pytorch_geometric
### Dataset

In this work two datasets will be used. The first, GeometricShapes, will be used to evaluate qualitatively whether the model which is constructed in the following section behaves as expected. This dataset was created for this project and is published with the rest of PyTorch Geometric [5]. It contains 40 primitive 3D and 2D shapes, such as spheres, circles, cubes and pipes. See Figure 4.2 for all shapes. Only one object exists for each class, however when sampling the point cloud from the surface of an object, the samples are distinct every time. This allows the model to learn generalizations. Furthermore, this dataset is not used to evaluate the model, but serves simply to visualize the internal workings of the model. It will be heavily used in the following chapter, where the model is constructed and evaluated step by step.

The other dataset which will be considered is ModelNet [31]. It exists in two variants, one with 10 classes and one with 40 classes. The dataset is split up into a training and test set, such that comparisons of different models rely on exactly the same data. This model will be used when evaluating and comparing the model to other models in the area of geometric deep learning. The objects classes include tv-monitors, tables, bathtubs and a range of other common items. Each class contains many models of the same label, unlike the GeometricShapes dataset.

### Previous results on ModelNet

To compare the results of the model introduced in the next chapter we look at results of other models on the ModelNet40 dataset. The accuracies are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
<th>Input Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointNet[24]</td>
<td>89.2</td>
<td>1024</td>
</tr>
<tr>
<td>PointNet++[23]</td>
<td>90.7</td>
<td>1024</td>
</tr>
<tr>
<td>DGCNN [29]</td>
<td>92.2</td>
<td>1024</td>
</tr>
<tr>
<td>PointCNN [19]</td>
<td>92.2</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table 3.1: Accuracy in % of comparable models on the ModelNet40 classification task. All models use 1024 input points.

A myriad of other machine learning models exist and have competed on ModelNet40, however most of them are of different nature, meaning they do not employ geometric deep learning. Hence they will not be used for comparison. Notable results are however the Voxel based OctNet [25] with 83.8%, a tree-based extension Kd-Net (depth 15) [13] with 88.5% and the multiview based RotationNet [11] with 97.37% accuracies.
Chapter 4

Extension

This chapter will introduce the extensions made to previous models.

Several difference between machine learning on ordered data, such as images or audio files and geometric deep learning have been identified. Those include that rotational and scale invariants need to be achieve through different means, and the problem with the order of points in point clouds arises. An extension to previous geometric deep learning layers should also respect these invariants.

I propose to extend the idea of directional filters from images as described in Section 2.3 to GDL. This will be done in stages to verify that each step of the derived algorithm works as intended. GeometricShapes will be used for visualization.

First, in Section 4.1 we explore how the \( k \)-NN of a central point \( x_0 \) can be rotated such that they respect a common direction. Next we explore feature extraction on these rotated points, and verify its effectiveness through ablation. Finally we consider embedding features in 3D space which respect the rotation of the rotated points.

After defining the new layers we connect them back to other models which were discussed in Section 3.4 to highlight their novelties and origins.

Finally this chapter will define the final model which will be used in the next chapter, focusing on experiments and results. The model will be made up of some new layers, and some previously defined layers.

The DirSplineConv layer

This section will present the building blocks used later when defining models and motivates them. Their correctness is shown and a simple ablation test proves their effectiveness. Finally we introduce the models evaluated in the next chapter.

Basis transformation

Given a graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}) \) without vertex features, chose a vertex \( v_0 \in \mathcal{V} \) and assume that the neighbors of \( v_0 \) in \( \mathcal{G} \) are defined by its \( k \)-NN. An example of this is given in Figure 2.5a. Let \( x_1, \ldots, x_k \) be the centered neighbors, such that \( x_i := v_i - v_0 \) with \( v_i \in k\text{-NN}(v_0) = \mathcal{N}(v_0) \). When considering \( X = (x_1, \ldots, x_k) \in \mathbb{R}^{3 \times k} \) as a matrix, we can easily find the plane of best fit through the origin using the SVD

\[
X = U \Sigma V^T.
\]
As described in Section 2.1, $U_3 \in \mathbb{R}^3$ is the normal of the plane of best fit, and $U$ itself is orthonormal. Consider $U$ as the basis transform $T_{B,E}$, with the basis $B := \{U_1, U_2, U_3\}$. Transforming each point $x_i = [x_i]_E$ to $[x_i]_B$ is easily achieved by

$$\tilde{X} := U^TX.$$  

The interesting property of $\tilde{X}$ is that its variance along the third dimension is very low, and high in the first two. One strong assumption is made here, namely that the neighborhood of any vertex is on a 2D manifold embedded in 3D space. This is justified, as this is indeed the case for models from ModelNet and GeometricShapes. The projection is depicted in Figure 4.1, with $X$ on top and $\tilde{X}$ on the bottom. Two problems are already seen here. On the right, all nearest neighbors are used to produce the basis transform $U$. This means that $U_3$ is not parallel to the normal of the face on which the central node is located. This can be seen in the bottom right, as the face is slanted. When using only 6 nearest neighbors to compute $U$, as done for the middle two figure in Figure 4.1, the plane is parallel to the xy-plane. On the left a section of a sphere is shown. After rotation it is convex, meaning its opening upwards. The central figure shows a concave shape, i.e. its opening is downwards.

In Section 2.1 we discussed, that the results of the SVD are unique when ordering $\Sigma$ by size up to a multiplication by $-1$. To ensure that $U$ is always consistent, we compute the center of mass of $\tilde{X}$, calling it $\tilde{x} \in \mathbb{R}^3$. If its third entry is negative, then we flip the sign of the third row of $\tilde{X}$. This effectively forces the shape to always be convex. From now on we always assume that $\tilde{X}$ is already convex, and we call it a cluster.

We have previously discussed, that directional filters are bad at working on isotropic data. These issues do not arise here, as just discussed.

**Feature extraction from a cluster**

Given the cluster $\tilde{X}$ as described in the previous section, it is now the goal to extract features from it. To do so we use SplineConv as defined in Section 3.4.

Let $m \in \mathbb{N}$ be the number of 3 dimensional B-spline filters with $K+1 \in \mathbb{N}$ supports. Each filter has an associated weight matrix $W_j \in \mathbb{R}^{K+1 \times K+1 \times K+1}$ and bias $b_j \in \mathbb{R}$. The support of a B-spline is in $[0,1]^3$, thus we shift and scale the cluster $\tilde{X}$ as follows:

$$M := \max\{\|\tilde{x}_i\|\}$$

$$Z := \frac{\tilde{X}}{M} + \frac{1}{2}.$$  

(4.1)

Let

$$f_j^{(l+1)} := b_j + \frac{1}{K} \sum_{z_i \in Z} B_{W_j}(z_i),$$  

(4.2)

and $f^{(l+1)} := (f_1^{(l+1)}, \ldots, f_m^{(l+1)})$. Careful inspection shows that this is a special case of the SplineConv, where the input features are always 1.

Unlike PointNet or DGCNN, this layer only looks at the relative location between points, and not at the global location. PointNet only uses the global location after
Figure 4.1: On the top are the 50 nearest neighbors of a central node, shifted such that the central node is in the origin. On the right, all $k = 50$ neighbors are used to find $U$. On the left and in the middle, only 6 are used. The oriented points are plotted below.

the input transform, DGCNN uses both relative and global coordinates after the input transform. Only using local coordinates is also done by PointCNN [19], who rationalize this by arguing that a layer working on local neighborhoods of points should also only use local coordinates.

**Feature extraction for each point**

In a graph $G$ each vertex $v$ can be mapped to its cluster $\tilde{X}_v$ and then mapped to its features $f_v^{l+1}$. We can now define the $\text{DirSplineConv}$ as the map mapping $G$ to $G^{(l+1)}$ where $\mathcal{V}^{(l+1)} := \mathcal{V}$, $\mathcal{E}^{(l+1)} := \mathcal{E}$, and $\mathcal{F}^{(l+1)} := \{f_v^{l+1} | v \in \mathcal{V}\}$. The parameters are omitted from the notation.

**Correctness**

Let us now show that $\text{DirSplineConv}$ does satisfy the invariance properties which we desire from it. When taking a cluster, we center it at the origin, hence shift invariance is maintained. Further, rotational invariance is maintained as the cluster $X$ (which may have any rotation) is rotated, such that $X$ is always concave with least variance in the up/down direction. The scale invariance is achieved in equation (4.1). Finally, as $\text{DirSplineConv}$ is a MPL, it is invariant to the ordering of points.

Backpropagation for this layer is also trivial, as there are no added parameters in addition to those from the $\text{SplineConv}$ layer.
Figure 4.2: All shapes from the GeometricShapes dataset. The color of each vertex is determined by the output of DirSplineConv. The first three feature were used as RGB values.

**Proof of concept for DirSplineConv**

The idea behind the DirSplineConv layer is that it is able to return the same features if the central node of a cluster is in a similar surrounding. For example, all vertices which lie on the edge of a cube should have similar features, however they should be different from those of points which lie on faces or corners. A small test model was implemented, and the results are displayed in Figure 4.2. The color of each vertex is determined by the output DirSplineConv for the vertex. All outputs are then fed pointwise into the sigmoid function, and the first three activations are used as RGB values. It is evident from this that all 2D objects have a distinct purple color. Round shapes like spheres are pink, edges are green, thin cylinders are orange and thicker cylinders are green. This shows that the DirSplineConv layer behaves as expected. More rigorous experimental discussion is done in the next chapter.

**Ablation**

While the proof of concept showed that the DirSplineConv layer worked as intended, it is not obvious whether the projection actually benefits the learning, or whether the internal spline convolution alone is enough to extract features from the point cloud.

Two quick experiments were done where in the first, the DirSplineConv layer is used exactly as previously explained, and in the second the $U$ matrix is always set to be the identity. Effectively this means that the clusters are not rotated, however they are still scaled and shifted to fit into the box $[0, 1]^3$ nicely. Figure 4.3 shows the results. The training with ablation is slower and seems to converge at a higher loss.
Figure 4.3: On the left, the \texttt{DirSplineConv} layer is used as intended. On the right, the transformation matrix $U$ is always set to the identity. Apart from this, both experiments are identical. The negative log likelihood is plotted against a training of 1000 epochs. On the left, the loss reaches roughly 0.7, and is at about 1.0 after only 400 epochs. On the right, the loss of 1.0 is only reached towards the end of training.

While this is not an extensive analysis, it does suggest that the model works as intended and a clear improvement becomes apparent. Detailed experiments and their discussion is reserved for the next Chapter.

\section*{The \texttt{DirSplineConv3D} layer}

This section introduces a small extension which is made to the \texttt{DirSplineConv} layer. First, let us examine a drawback of the layer. After recording the directional features $\mathcal{F}^{(l+1)} = \{f_v^{(l+1)} | v \in \mathcal{V}\}$, we lose the directional information of the features, as the features are just scalars. Arguably the directional information is still present because all the vertices are still present, however if we remove some vertices by means of pooling, then some of this directional information is lost.

Assume the underlying \texttt{SplineConv} has $m$ filters, then $f_v^{(l+1)} \in \mathbb{R}^m$. To overcome the loss of direction, we consider the $m$ features $f_v^{(l+1)}$ as vectors in $\mathbb{R}^3$ pointing upwards. This is depicted in Figure 4.4c. Afterwards we can transform the vector back by multiplying it by the inverse of $U^T$. Note that, since $U$ is orthonormal, this is just $U$. As a basis transformation, we can think of the vector $(0, 0, f_v^{(l+1)})^T$ as coordinates under the basis $B$, and after multiplication with $U$ the vector is transformed back into the canonical basis $E_3$. This is depicted in Figure 4.4d, with its caption going into more detail.

The \texttt{DirSplineConv} layer mapped a graph $\mathcal{G}$ to a new graph $\mathcal{G}^{(l+1)}$ such that $f_v^{(l+1)} \in \mathcal{F}^{(l+1)}$ satisfies

$$f_v^{(l+1)} \in \mathbb{R}^m.$$ 

For the new \texttt{DirSplineConv3D} layer $f_v^{(l+1)} \in \mathcal{F}^{(l+1)}$ satisfies

$$f_v^{(l+1)} \in \mathbb{R}^{m \times 3}.$$
Figure 4.4: Figure 4.4a depicts a cluster before rotation. In Figure 4.4b the points are rotated, such that the direction of least variance points upwards. Spline convolution is used to extract features from the rotated cluster. These features are then embedded in 3D space, simply by taking vectors which point up/down, as shown in Figure 4.4c. In Figure 4.4d both the feature vectors and the cluster are transformed back into the original direction.
The step by step process of \texttt{DirSplineConv3D} is depicted in Figure 4.4.

\section*{From 3D features to 3D features}

The previously defined \texttt{DirSplineConv} and \texttt{DirSplineConv3D} layers both take no features as input, but just the coordinates of the vertices. However they output features, either in 1D or in 3D. To be able to stack layers on top of each other, as is usually done in NNs, we need to be able to work with vertices that contain features. Furthermore, with all emphasis thus far having been on preserving and extracting orientational information, we wish to continue this trend.

Before we get into the actual definition of such a layer, let us look at Figure 4.5, where the transformation of a cluster is depicted. On the left, a cluster in a random rotation with 3 directional features at each vertex is depicted. Then, as in the \texttt{DirSplineConv} layer, all vertices and features are scaled and shifted, such that the central point is in the origin and the shape is facing upwards. Then, the thus far undefined transform takes place, using all features and vertex positions and transforming them to new features for the central node. These are depicted as longer arrows. Then, the new features and vertices are transformed back into their original position, just like in the \texttt{DirSplineConv3D} layer.

Using a spline convolution in the middle step is probably a bad idea. To justify this, let us remember equation (4.2) in \texttt{DirSplineConv},

\[ f_j^{(l+1)} := b_j + \frac{1}{k} \sum_{z_i \in Z} B_{W_j}(z_i). \]

If we now concatenate the features of the central vertex and of the neighbor \(v_i\) to \(z_i\), we are left with a 9 dimensional vector as input to the Spline \(B_{W_j}\). With only 3 supports, we would already have \(3^9 = 19683\) parameters for a single filter. As such a different approach must be taken.

The layer which will be defined now is closely linked to the \texttt{EdgeConv} layer as introduced in Section 3.4. The DGCNN network was introduced with the edge function \(h_\theta(f_{v_i}, f_{v_j} - f_{v_i})\). The authors have also experimented using edge functions of the form \(h_\theta(f_{v_i}, f_{v_j})\) or \(h_\theta(f_{v_i} - f_{v_j})\). We too will use an edge function, however will discuss the exact form of its inputs later.
The DirEdgeConv layer

Let the new layer, by the name of DirEdgeConv be identical to the DirSplineConv layer, except that we replace the spline convolution

\[ f_{j}^{(l+1)} := b_j + \frac{1}{k} \sum_{z_i \in Z} B_{W_j}(z_i) \]

with an edge function

\[ f_{j}^{(l+1)} := \frac{1}{k} \sum_{z_i \in Z} h_{\theta}(\cdot), \]

where for now the input is unspecified, but can be any combination of \( v_i, v_j, f_{v_i} \) and \( f_{v_j} \).

The edge function

To chose an edge function to use, six are compared. A neural network model is built, consisting of a DirEdgeConv3D with layer mlp(64,64,64), a DirEdgeConv layer with mlp(128), a pooling layer and a mlp(1024,512,265,C) at the end. The first DirEdgeConv3D layer has an edge function with signature \( h_{\theta}(v_j) \), and DirEdgeConv has different signatures in different experiments. The six signatures tested are

\[ h_{\theta}(f_{v_i}, f_{v_j}), \quad (4.3) \]
\[ h_{\theta}(f_{v_i}, f_{v_i} - f_{v_j}), \quad (4.4) \]
\[ h_{\theta}(v_j, f_{v_j}), \quad (4.5) \]
\[ h_{\theta}(v_j, f_{v_i}, f_{v_j}), \quad (4.6) \]
\[ h_{\theta}(f_{v_i}), \quad (4.7) \]
\[ h_{\theta}(v_j). \quad (4.8) \]

In Figure 4.6 the losses of different edge functions are plotted for 300 epochs of training on the GeometricShapes dataset. None of the results seem very different at first glance, however this might be due to the very simple underlying dataset. All models achieve accuracies of close to 100%, and in some epochs even exactly 100%. The least loss of the different signatures is shown in Table 4.1.

<table>
<thead>
<tr>
<th>( h_{\theta} )</th>
<th>(4.3)</th>
<th>(4.4)</th>
<th>(4.5)</th>
<th>(4.6)</th>
<th>(4.7)</th>
<th>(4.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loss</td>
<td>0.1993</td>
<td>0.1742</td>
<td>0.1824</td>
<td>0.2224</td>
<td>0.1979</td>
<td>0.1996</td>
</tr>
</tbody>
</table>

Table 4.1: Lowest loss of different signatures.

In agreement with [29] it seems that (4.4) performs best. Note that (4.8) does not actually use the output of the first DirEdgeConv3D layer, yet still performs better than (4.6) which has all information available.

The DirEdgeConv3D layer

Analog to the extension from DirSplineConv to DirSplineConv3D we can extend DirEdgeConv to DirEdgeConv3D.
Figure 4.6: All models were trained with the same initial parameters for 300 epochs. Their training loss is plotted, where the loss is the negative log likelihood. Top left: (4.3), top right: (4.4), middle left: (4.5), middle right: (4.6), bottom left: (4.7) and bottom right (4.8). The lowest loss is achieved by (4.4).
Connection to other models

The layers which were introduced in the previous section took inspiration from various sources. The main novelty of them is inspired mainly by directional convolutional filters, which are examined in [20]. A related approach has been studied in [19], where instead of rotating the points in space and using a layer which is equivariant to point ordering, they used a learnable point ordering to incorporate direction into their model. The key idea is that the learned ordering would correspond to a canonical orientation of the points.

After the rotation of the points in 3D space an edge convolution is applied, once analogue to the edge convolution from DGCNN, and once by applying spline convolution. These approaches were clearly inspired by [29] and [6].

The 3D extension was inspired by considering the following. PointNet used a transformer network to initially transform all points into a canonical space. The layers introduced here only transform clusters, and after the feature extraction this transformation is lost. To retain directional information 3D features were introduced.

The shortcoming of the models inspired by PointNet, which were already alluded to in the introduction deserve another mention. In particular we have always made sure that each step along the way is invariant to the four invariances mentioned in section 3.2. These were rotational, scale and shift invariance, and invariance under point order. PointNet and PointNet++ solved invariance under point order by simply working on points individually and then, as soon as different points were combined, a symmetric aggregation was used. Models built with message passing layers do not have this problem, as discussed earlier. However, the other three invariances are not rigorously respected. The initial input transform transforms all points into a canonical 3D space, however as this rotation matrix is the output of a neural network with learned parameters, it is not guaranteed that the invariance is met. It is only approximated, or the model learns to use different rotations.

PointCNN lacks further invariance as it also relies on point order in a cluster, and does not address the rotational invariance. The point order which is used is indeed the output of a neural network, and is not even continuous under shifting points. The authors suggest that point order in a cluster is enough for classification, and back this up with good results. As this project tries to achieve invariance in all cases, it is a stark contrast. It must be mentioned that PointCNNs idea of bypassing an initial input transform to work on clusters directly was a main source of inspiration for the layers presented in this chapter.

This motivated DirSplineConv and DirEdgeConv. The main advantage that these layers have is that they work only on local neighborhoods and that they manage to rotate local clusters deterministically. Hence rotational invariance is strictly adhered to. Furthermore, an advantage is that these rotation matrices are independent of parameters, and need not be learned. This means that the network has less parameters.
Novel models

In the following chapter many models are evaluated. To make sure that all details are understood we will now define one model, and thoroughly explain each of its layers. A graphical representation will also be provided. In the next chapter, where we do the experiments on a variety of models only brief explanations of the differences are given.

DirDCNN

The first model which is evaluated, named DirDCNN tries to copy DGCNN (refer to Figure 3.4) as much as possible, to invite a fair comparison. As this model was previously only introduced to show the concept of an edge convolution, we shall now take the time to go into more detail.

The input to the model is simply a point cloud sampled from the surface of an object at any random scale and orientation. In Figure 4.7 this is labeled by (1). To convert this point cloud $X$ into a graph $G = (V, E)$ the $k$-NN algorithm is used in the vertex space (2).

Since the DirEdgeConv3D layer is invariant to initial rotation of the object, DirDCNN will not include an input transform layer. The EdgeConv layer with an inner $\text{mlp}(64, 64, 64)$ is replaced with a DirEdgeConv3D layer, which also has an inner $\text{mlp}(64, 64, 64)$, as depicted by step (4). This inner $\text{mlp}$ corresponds to $h_\theta$. As we only have the vertices at this point, and no features $F$, we use

$$h_\theta(v_j) = \text{mlp}(64, 64, 64)(v_j).$$

The aggregation function chosen is, just as in DGCNN, the max function. As with the first, the second EdgeConv is replaced by a DirEdgeConv layer. Here the edge function is a function of $f_j \in \mathbb{R}^{64 \times 3}$,

$$h_\theta(f_j).$$

Refer to step (8). From this point forward, both DGCNN and DirDCNN behave the same. This means pooling and $\text{mlp}(1024)$ is copied, and so is the final $\text{mlp}(512, 256, C)$. It is worth mentioning that DGCNN has included a batch normalization layer after every step, a practice that is duplicated. Furthermore, after each layer there is a ReLU activation function added. Only the last layer has a softmax function added, as could be expected.

The D from DGCNN was thus far not explained. In Section 3.4 we have already alluded to the idea that the edges of a graph can be defined via its $k$-NN in feature space. This is exactly what DGCNN is doing, meaning

$$e_{i,j} \in E \Leftrightarrow f_{v_j} \in k\text{-NN}(f_{v_i}).$$

This means that for the first DirEdgeConv3D layer, the edges are defined by $k\text{-NN}(v_i)$, and for the DirEdgeConv layer by $k\text{-NN}(f_{v_i})$. In Figure 4.7 this corresponds to step (6). In both cases we have $k = 20$. 
In Figure 4.1 we have shown that using all \( k = 20 \) vertices might not be a good idea to find the best rotation matrix. Of the \( k = 20 \) nearest neighbors we chose the closest 7 to compute the rotation matrix.

After having introduced all the similarities, it is worth mentioning that DirDCNN has a few, albeit not many more parameters in its EdgeConv layers. This is due to the fact that input features are mapped into 3D space, meaning there are 3 times more parameters in \( W^{(l)} \) when applying

\[
\sigma^{(l)}(W^{(l)}h^{(l-1)} + b^{(l)}).
\]

On the other hand DGCNN includes a spatial transform, which is entirely cut out here.

**DirCNN**

The authors of the network DGCNN, standing for Dynamic Graph Convolutional Neural Network have experimentally concluded that the inclusion of step (6) is beneficial. We will replace step (6) by \( k\text{-NN}(v) \), and call this model DirCNN. Note in particular that (2) and (6) give the exact same edges since the vertices have not changed. This means that in practice we can skip this step.

**DirSplineCNN**

While DirDCNN is very close to DGCNN in architecture, we wish to include DirSplineConv3D into the experimental stage. As we will later see, DirCNN is much faster than DirDCNN, hence we will use DirCNN as a template. In Section 4.1 we have already discussed that this layer is only useful as the first layer to the model. Hence we replace the first DirEdgeConv3D layer with a DirSplineConv3D layer. This means that in Figure 4.7 we remove step (6) and replace step (4) by a DirSplineConv3D layer. The amount of Spline filters and the amount of supports they have will for now be left open, and different configurations are examined in the next chapter. Such a model we shall call DirSplineCNN.

**DirCNN + fps/graculus**

Two pooling methods will also be investigated. Again we take DirCNN as a template and again we replace step (6). For DirCNN+fps we set \( r = 0.25 \) and use the farthest point sampling algorithm to remove 75% of all points. We also remove the associated
features. If we consider the old edges, some of them will end in vertices which were removed. This means that we have to compute new edges, and we do this via $k$-NN($v$). As a result step (7) and (9) now have $n/4$ input points.

The keen reader will have noticed that we do a lot of computations in step (4) which are going to be dropped in the following stage. The implementation of the model is efficient and does not do these unnecessary computations, however the explanation is much easier to follow on the unoptimized model.

The second pooling used is graclus, which reduces the amount of vertices by roughly a factor of 2. Again we need to create new edges, and do this by $k$-NN($v$). The model using graclus as a pooling algorithm is denoted by DirCNN+graclus.
Chapter 5

Experiments

In this chapter we will use ModelNet10 and ModelNet40 to evaluate both models introduced in the previous section. As ModelNet10 is much smaller and easier than ModelNet40, we will use it in the beginning to decided on which model is the best. The reason for this is that it takes a very long time to train these models, and a larger dataset with more classes exaggerates this further. Once having decided on which is the best model, we shall test it on ModelNet40, in order to have a comparison to other models who competed on ModelNet40.

Experiments on DirDCNN

In the previous Chapter 4 DirDCNN was defined. In a first experiment we compare DirDCNN with a model where the rotation matrix $U$ is artificially set to the identity. This corresponds to the ablation test from Section 4.1, however is done on a real dataset. This ablation model is called NonDirDCNN, and is nearly identical to DGCNN. It mainly misses the input transformation, and is thus expected to perform worse than DGCNN. Because training on ModelNet40 takes too long, we do this test on ModelNet10. Since a different dataset is present, we no longer have to abide by the learning rate scheduling described by [29]. Instead we choose an initial learning rate of 0.001 and ever 2 epochs the learning rate $\lambda$ decays to $0.9\lambda$.

Results

In Figure 5.1 we compare the results of the two models. While after only 35 epochs it seems that both models could still improve on their test accuracy, the learning seems to be settling down and becoming more stable. While NonDirDCNN reaches its best accuracy 83.3% the non-ablated model DirDCNN reaches a higher accuracy of 86.0%.
Discussion

Further training would certainly improve both models, however as training takes several hours, it was decided to focus on other experiments as well. Nevertheless there are certain conclusions we can draw from this experiment.

Firstly, NonDirDCNN performs roughly 3% worse than DirDCNN. Secondly, the rate of learning in the beginning of training is significantly higher for DirDCNN. This can be expected, because the nature of the input after the cluster transformation is much more regular.

Experiments on DirCNN

In this experiment we test DirCNN. As it is quite fast we use it to explore the impact of learning rate, and how well it does after many epochs of training. Three experiments are run. The first (DirCNN(1)) has the same learning rate setup as the previous experiment. Because the model is much faster we let it train for more epochs than in the previous experiment.

The second model (DirCNN(2)) only changes the decay step from every 2 to every 6 epochs. This means that after 90 epochs the learning rate is 20% of its initial rate, reached previously after only 30 epochs.

The third and final experiment (DirCNN(3)) uses cyclic learning rates. An upper bound is set to 0.001 and a lower to 0.00003. Each cycle is completed after 6 epochs, and further an exponential decay of 0.95 is added. The learning rate is visualized in Figure 5.5.
CHAPTER 5. EXPERIMENTS

Results

In Figure 5.2 we see that DirCNN(1) ceases to learn after around epoch 40. At this stage the learning rate is $0.001 \cdot 0.9^{40/2} = 0.00012$, only 12% of the initial learning rate. This might be much too small. A maximum accuracy of 84.4% is achieved.

Figure 5.2: Accuracy of DirCNN(1) on ModelNet10.

In Figure 5.3 we see the training of DirCNN(2) with the same learning rate, however with a decay applied only every 6 epochs. The best accuracy of 85.0% is similar to that of DirCNN(1). Flattening of the accuracy curve starts later, at about 50-60 epochs, while for DirCNN(1) after epoch 35 no significant gains seem to be achieved.

Figure 5.3: Accuracy of DirCNN(2) on ModelNet10.

The accuracy and learning rate when using cyclic learning rates are shown in Figures 5.4 and 5.5. The best accuracy achieved was 83.4%. The learning rate has arguably decayed too fast, something that we will keep in mind when building the final model to be used on ModelNet40.

Figure 5.4: Accuracy of DirCNN(3) on ModelNet10.

Figure 5.5: Learning rate of DirCNN(3) on ModelNet10.
Discussion

All three learning rate schedules and parameters give roughly the same results. This is not unusual, as we heavily rely on batch normalization. Never the less this observation reinforces the belief that the the training used in the following section is adequate.

Experiments on DirSplineCNN

In this section we explore the DirSplineConv3D layer, and the impact the parameters have on the DirSplineCNN model. We recollect that the number of supports and the number of filters are of interest. We compare 4 different setups on the ModelNet10 dataset.

DirSplineCNN(1) is as defined in Section 4.5, and has 30 filters with 7 supports each. Refer to Table 5.1 for the other models.

<table>
<thead>
<tr>
<th>DirSplineCNN Nr.</th>
<th>Filters</th>
<th>Supports</th>
<th>Best Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>15</td>
<td>78.8%</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>15</td>
<td>79.3%</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>7</td>
<td>80.8%</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>7</td>
<td>83.3%</td>
</tr>
</tbody>
</table>

Table 5.1: The parameters and results of different DirSplineCNN models evaluated on 40 epochs of training on the ModelNet10 dataset.

The learning rate and scheduler is the same for all 4 experiments, namely we have an initial learning rate of 0.001, and a decay of 0.9 every 2 epochs.

Results

In the four plots of Figure 5.6 we see the test accuracy of the four DirSplineCNN models. The best accuracies of each model is also included in Table 5.1.

Discussion

It seems that both parameters have a large impact on the outcome of the experiment. As the number of filters increases the results improve, and quite drastically when going all the way up to 60 filters. An increase in the amount of supports seems to degrade the performance of the model. Furthermore, when taking a closer look at the validation accuracy in Figure 5.6, the accuracy seems much more turbulent, and after less than the 40 epochs it seems like the accuracy has plateaued. The models with only 7 supports are less turbulent, and extending the line imaginarily suggests a positive gradient.

Note also that in the evaluation the experiment on DirCNN (Section 5.2) we can see that the learning rate used here is not optimal. However this experiment is
Figure 5.6: The validation accuracy of different DirSplineCNN models is shown while training for 40 epochs.
not concerned with finding the best learning rate, but rather in comparing different parameters for the DirSplineConv3D layer.

**Experiments on DirCNN with pooling**

In this experiment we test DirCNN+fps and DirCNN+graclus. We use what we have learned in Section 5.2 and change the learning rate to the following. We continue using an initial learning rate of 0.001 with a decay of 0.9, however only decay every 6 epochs. A training run across more epochs is possible for these models because they run significantly faster.

**Results**

The results are shown in Figure 5.7. The maximum accuracies obtained are 82.3% and 84.2% respectively.

**Discussion**

Both models perform relatively well, considering the fact that they train significantly faster than the other models. Furthermore, they include a dynamic recalculation of edges. This is done by using $k$-NN($v$). DirDCNN uses $k$-NN($f$) and has thus far achieved best results, but was simultaneously by far the slowest.

**Experiments on ModelNet40**

DirDCNN40+graclus

**Model setup and parameters**

We now build a model to be trained on the ModelNet40 dataset, keeping in mind what we have learned through the previous experiments. The first layer shall be a DirEdgeConv3D layer, as before.
The previous experiments have lead to the conclusion, that dynamic recalculation of edges in feature space gives a considerable advantage, however is very slow. Pooling methods speed up the algorithm without a mayor loss of performance. Hence the final model shall include a graclus pooling layer with dynamic computation of edges in feature space.

The discussion in section 4.3 about the edge function $h_\theta$ has concluded that $h_\theta(f_v, f_v - f_y)$ gives best results, as already discussed by [29]. This suggests that the following DirDenseConv layer should use this signature instead. The model is visualized in Figure 5.8. Note that step (8) takes $n/2$ vertices. This is only approximate, as graclus returns a matching where some vertices are unmatched. This means sometimes there are a few more than $n/2$ vertices as input to this layer. This is however not a problem because the following layers only work on the graphs and layer (11) does max pooling, which works with an arbitrary amount of inputs.

The initial learning rate will be 0.001 using Adam optimization and a decay of 0.95 every 2 epochs. We shall train this model on ModelNet40 for 110 epochs.

## Results

In Figure 5.9 we can see the training accuracy for the first 50 epochs. After 50 epochs the algorithm was adjusted to use the validation dataset. Thus, after the first 50 epochs, the validation accuracy is plotted. This shows the huge difference between validation and test accuracy. While the model might still be learning at the end of the 110 epochs, the learning rate is already very low. The best validation accuracy obtained was 83.4%.

### DirSplineDCNN40+graclus

#### Model setup and parameters

Let the DirSplineDCNN40+graclus model be identical to DirDCNN40+graclus, except for that the DirEdgeConv3D layer is replaced by an DirSplineConv3D layer with 7 supports and 64 filters.

The learning rate is also 0.001 initially, however decays by 0.95 every 5 epochs. This allows for a longer effective training. The model is trained for 100 epochs.
Results

In Figure 5.10 we can see the validation accuracy when training for 100 epochs. It is interesting to note that, while in the beginning it seems that the training is quite effective, the high learning rate in the beginning leads to very volatile validation accuracy. Later on, towards epoch 60, the validation accuracy is much more consistent. A maximum of 82.4% is reached.

Discussion

The maximum accuracy of both models differs by less than 1 percent, and it is worth mentioning that due to the volatility this difference is not significant. This means it is hard to say which model is better. The fact that point clouds are sampled randomly from the ModelNet40 models means that there is a certain degree of randomness through the training and evaluation. Furthermore it is interesting that these two models perform similarly on ModelNet40 as the models previously introduced perform on ModelNet10. The best accuracy on ModelNet10 achieved was 86.0% and the lowest 78.8%. Other models which have been evaluated on both ModelNet10 and ModelNet40 show a differences of only 1 percent up to 10 percent. This suggests that the models evaluated here generalize well.
Outlook

Sparsity in SplineConv

In section 4.1 we introduced an approach which uses spline convolution on transformed clusters. Points in these transformed clusters are by design as close to possible to the horizontal plane. This means that few times the points will lie on the top/bottom part of the spline function. When speaking about Voxelgrids we have already mentioned that sparse training brings with it many disadvantages. This problem could be tackled in a few different ways, however in this work this problem is not further examined. Nevertheless it is important to note, that this could hinder learning or simply mean that there is a useless computational overhead.

Training Rate and Scheduling

Through all experiments it seems that the training rate and scheduling were not optimal. Because of the huge cost of training, financially as well as time consuming, this part was not fully optimized. The results could certainly have been improved with better parameters.
Chapter 6

Conclusion

The goal of this project was to implement a novel layer which works well in the framework of geometric deep learning. This layer aimed to be completely invariant towards point ordering, rotation, scaling and shifting. The \texttt{DirSplineConv} layer achieves this, while \texttt{DirEdgeConv} is not invariant towards scaling. As point clouds are always resized to fit into the unit circle, this lack of invariance is not important.

Of course these layers are to be evaluated in a variety of models. This was done successfully, comparing different edge functions, parameters for spline filters and the effect of pooling and dynamic recalculation of edges. These were compared on ModelNet10. No state of the art results were achieved here, nevertheless the performance is respectable. In particular PointNets accuracy on ModelNet10 is 77.6\%, which was surpassed by nearly 10\%.

The results on ModelNet40 were also not state of the art, however more inline of what other models achieved on the dataset. As previously mentioned, the learning rate and scheduling was a limitation to the performance of the models. Due to the high cost financially, the parameters to be used for ModelNet40 were estimated by experiments on ModelNet10. Whether this inference is legitimate is not clear, however due to the limitations this path was chosen.

In the earlier part of this project the geometric dataset was introduced. In retrospect this was extremely useful, because it allowed to test small models relatively quickly. Training on this dataset usually only took 10 to 15 minutes to reach maximum accuracies. Early on in the project accuracies of only 70\% were reached, however towards the end 100\% were reached. This allowed me to confidently say which changes worked well, and which did not. If I had used ModelNet10 for this 3 to 5 hours would have been needed. The final training on ModelNet40 took about 24 hours in both examples. Furthermore the geometric dataset allowed visualizations of the inner activations, as shown earlier.
Bibliography


