TUM Department of Civil, Geo and Environmental Engineering
Chair of Computational Modeling and Simulation
Prof. Dr.-Ing. André Borrmann

Point Cloud Classification as a Support for BIM-based Progress Tracking

Tim Breu

Masterthesis
for the Master of Science degree in Civil Engineering

Author:	Tim Breu
Immatriculation Number:	[redacted]
Supervisor:	Prof. Dr.-Ing. André Borrmann
	Alexander Braun M.Sc.
Date of Issue	01. August 2018
Date of Submission:	01. February 2019

Abstract

In construction site progress tracking, a point cloud of the construction site is compared to the 3D BIM-model of the planned building. If a sufficiently large number of points can be detected in the vicinity of a building component, this component is likely built. The schedule, deposited in the BIM-model, is used to derive the overall progress of the construction site and to detect delays. Problems occur when the quality of the point cloud makes the detection of objects ambiguous. Occluded areas, sparse regions, or temporary objects aggravate the comparison between the point cloud and the model. Object detection in point clouds would improve this workflow significantly.

Pointwise classification is generally an unsolved problem. The low quality of the point clouds in use, renders the application of classical feature descriptors impossible. Therefore, this thesis investigates how pointwise classification can be applied to point clouds with the help of deep learning. Different deep neuronal network architectures are explored. In order to train the architecture of choice, a dataset is needed. Because there is no shape dataset for construction site related objects, a generator is developed which can generate datasets based on common mesh files.

The theory behind machine learning in general and neuronal networks in specific is explained. Followed by the workflow to create a dataset and to train the neuronal network. Results and suggestions for further investigations complete this work.

Zusammenfassung

In der automatischen Baufortschrittskontrolle wird ein BIM-basiertes 3D Modell mit einer Punktwolke verglichen. Wenn sich genügend Punkte in der Nähe eines Bauteiles befinden, kann man davon ausgehen, dass das tatsächlich fertig gestellt wurde. Mit dem im BIM-Modell beinhalteten bauteilspezifischen Ablaufplan kann eine Aussage über den Baufortschritt getroffen werden. Problematisch wird es, wenn die Punktwolke nicht eindeutig Aufschluss über Teilbereiche der Baustelle gibt. Das kann bei Verdeckungen, lichten Stellen oder temporären Objekten, wie Schalungen, vorkommen. Wenn Objekte auch unabhängig vom BIM-Modell erkannt werden können, erhöht das die Genauigkeit der Fortschrittskontrolle signifikant.
Die generelle Klassifizierung von Punkten in einer Punktwolke ist zur Zeit eine ungelöste Fragestellung. Hinzu kommt, dass die verwendeten Punktwolken für die Baufortschrittskontrolle keine gute Qualität aufweisen und deshalb klassische Methoden zur Klassifizierung untauglich sind. Diese Arbeit befasst sich damit, wie punktweise Klassifizierung mittels Deep Learning auf Punktwolken angewandt werden kann. Es werden unterschiedliche Architekturen auf Tauglichkeit untersucht. Damit man ein Neuronales Netz trainieren kann braucht man einen Datensatz. Das es wenige Datensätze für Punktwolken und gar keine für Baustellen gibt, wird ein Datensatz künsltich generiert. Die Objekte in diesem Datensatz basieren auf üblichen geometrischen Netzen.
Es wird die Theorie hinter maschinellem Lernen und im speziellen Neuronalen Netzten beleuchtet und die Arbeitsweise des Datensatzgenerators erklärt. Ergebnisse und Vorschläge für das weiter Vorgehen runden die Arbeit ab.

Contents

1 Introduction **1**
1.1 Context . . . 2
1.2 Motivation . . . 3
1.2.1 Thesis . . . 4
1.3 Related Work . . . 5
1.3.1 Construction Progress Monitoring . . . 5
1.3.2 Point Cloud Classification . . . 6

2 Learning **8**
2.1 The Curse of dimensionality, but Manifolds . . . 8
2.2 Machine Learning . . . 10
2.2.1 Supervised Learning Algorithms . . . 12
2.2.2 Unsupervised Learning Algorithms . . . 12
2.3 Linear Regression or "Hello-World" . . . 12
2.4 Task . . . 14
2.5 Experience is Data . . . 15
2.5.1 Dataset Selection . . . 16
2.6 Performance . . . 17
2.6.1 The Loss . . . 17
2.6.2 Measure . . . 18
2.7 Gradient Descent . . . 19
2.7.1 Stochasitc Gradient Descent . . . 20
2.8 Validation . . . 21
2.8.1 Overfitting . . . 21
2.8.2 Training, Validation and Test Set . . . 23
2.9 Regularization . . . 25
2.9.1 L2 Regularization . . . 25
2.9.2 Dropout . . . 26
2.9.3 Other methods . . . 27

3 Deep Neuronal Networks **28**
3.1 Layers and Activations . . . 29
3.1.1 Perceptron . . . 29
3.1.2 Multilayer Perceptron or Fully Connected . . . 30
3.1.3 Softmax and Sigmoid . . . 31
3.1.4 ReLu . . . 33
3.1.5 Convolutional Layers . . . 33
3.1.6 Max Pooling Layers . . . 36
3.2 Automatic Differentiation . . . 37
3.3 Backporpagation . . . 39
3.4 Entropy vs. Maximum Likelihood . . . 42
3.4.1 Cross Entropy . . . 44
3.4.2 Maximum Likelihood Estimation . . . 45
3.4.3 The Link . . . 48
3.4.4 Softmax-Loss . . . 49
3.5 Inference . . . 50
3.6 Miscellaneous . . . 51

4 PointNet **52**
4.1 Mesh vs. Point . . . 52
4.1.1 3D Scanning and Reconstruction . . . 53
4.2 Boundary Conditions . . . 54
4.3 Spatial Transformation Networks . . . 55
4.4 PointNet Architecture . . . 56
4.4.1 Classification . . . 58
4.4.2 Segmentation . . . 58
4.5 PointNet++ . . . 59
4.6 Summary . . . 60

5 Point Cloud Generation **61**
5.1 Generator Architecture . . . 62
5.1.1 Aggregation . . . 63
5.1.2 Arrangement . . . 64
5.1.3 Sampling Slices . . . 67
5.2 HDF5 Dataset . . . 69
5.3 Training . . . 71
5.4 Improvements . . . 71

6 Results **73**
6.1 Choice of Implementation . . . 73
6.2 Evaluation . . . 74

6.2.1 Slices . . . 75
6.2.2 Test runs . . . 77
6.2.3 Inference . . . 79

7 Outlook & Improvements **81**
7.1 Deep Learning Approach . . . 81
7.1.1 Additional Features . . . 81
7.2 Dataset . . . 82
7.3 Disaster: Max Pooling . . . 82
7.4 Point Descriptors . . . 83
7.5 Conclusion . . . 84

Abbreviation

NN	Neuronal Network
MPL	Multilayer Perceptron
DNN	Deep Neuronal Network
CNN	Convolutional Neuronal Network
RNN	Recurrent Neuronal Network
AD	Automatic Differentiation
BP	Backpropagation
ML	Machine-Learning
KLD	Kullback-Leibler divergence
IoU	Intersection over Union
MSE	Mean Square Error
MLE	Maximum Likelihood Estimation
API	Application Programming Interface
ReLu	Rectified Linear Unit
SGD	Stochastic Gradient Decent
BIM	Building Information Modeling
IFC	Industry Foundation Classes
ICP	Iterative-Closest-Point algorithm
PCA	Principal Component Analysis
AEC	Architectural Engineering and Construction
B-rep	Boundary Representation
Lidar	Light detection and Ranging
STN	Spatial Transformer Network
MSG	Multi-Scale Grouping
MRG	Multi-Resolution Grouping
mla	machine learning algorithm
i.i.d.	independent and identically distributed
dl	deep learning
wrt.	with respect to

Chapter 1

Introduction

Learning has always been a challenging task for computers. What is the difference between storing data in a database and learning this data? It turns out that there are two main differences. We encode information into physical quantities like bits and bytes, which have well-defined positions inside a storage medium, e.g. a hard drive and a certain value. For the storage, we use file descriptors, a query language like SQL, or cloud service like Dropbox to retrieve the desired information. Because the structure of the stored data and its position are well defined, retrieving it is an algorithmic problem. An SQL query will always return the same data under the same initial conditions, the time this takes would depend on the amount of data stored in the database. How come that our brains do not exhibit this kind of trait? It can easily remember your high school teacher's name, but fails when you think about what your colleague wore yesterday. This property is called auto-associative. It can be compared with an unstable system, which, if disturbed, seeks for the minimal energy state. The disturbance is your thought and the minimal energy state the answer for your high school teacher's name. But why does this not always succeed? Because we learned this piece of information - namely the teacher's name - but we have not learned the coworker's wardrobe. But we could because of we - as humans - have the ability to do so. How we do it exactly is still a question of research and the example described above a massive oversimplification. Indeed, the study of the human brain is one of the most interesting research topics nowadays. But, to be able to make a machine learn something, there is no need to fully understand the brain at first. As often in biology and engineering, humanity tries to replicate nature but finds a suitable solution along the way. A plane flies because of the same underlying physical rules as well as a bird does but with totally different strategies. As well as our brain works with Neurons, a synthetically neuronal network does, too. But to be efficient we use graphics cards and no brain tissue [55].

Research means to pursue a solution for problems. Problems which are not immediately important for people on a daily basis. With solutions, most people have not even heard from. This thesis is a connection between deep learning and the task to track progress on construction sites. Deep learning is an abstract field of study located in the computer science department, whereas progress track has a fundamental relation to a concrete problem in reality. This contrast makes this thesis so interesting.

1.1 Context

Industrial manufacturing demands automation. In order to automate manufacturing, the manufactured products must be split into processes which require space, time, material, working hours and most importantly the relation between all of them. The birth hour of automation was Henry Ford inventing the assembly line [14]. Since then, the car has become an overengineered symbol of our modern age. It is not surprising that cars became what they are today. Technical machines where each assembly processes can be tracked, each material knows its origin and even the smallest parts are optimized to near perfection. But it makes sense to optimize cars in specific; they are identical to a high degree. Parts can be shared among different types, and any kind of improvement for one model is generally applicable to others. A whole ecosystem of car companies and their suppliers evolved, and work together. This is only made possible through a high standardization in planning and an almost vanishing margin for interpretation even in reality. Therefore, the lifecycle of a car is well defined in contrast to the building industry.

Any kind of building is a unique piece of collaboration. It starts with the idea, planned by the architect, approved by the civil engineer, build by the different construction companies, managed by facility firms and used by people, to be finally torn down by a wrecking company. This lifecycle is large for buildings and information is scattered among different parties, which have no motivation to store this information longer than needed. Information, like construction plans, building processes, material quantities and many more are not preserved for later use. And even if it would, there is little standard generally, and in particular no standard for storing it in a digital way.
Building Information Modeling (BIM) tries to overcome this problem by defining an international standard for a building lifecycle. This standard is, of course, not paper-based. The goal is to map as many information possible around a detailed and semantically described 4D representation of the building. In theory, there should be one place for information and a well-defined way to retrieve it. Construction sites are a pivotal part in the life of a building, and due to their nature, they are prone to create discrepancies between the plan and the reality. Saying this, a lot of ad hoc decisions and interpretation is common on construction sites, due to missing or not available information. Again, BIM tries to overcome these problems

by defining a data format, called Industry Foundation Classes (IFC). This format is derived from the STEP model used in the automobile industry. It defines a relation between entities for a BIM model and describes the material, properties, and processes. This model forms the basis where every contributing party is meant to update the model accordingly, but can also retrieve current information out of it. Conflicts can, therefore, be resolved as early as possible and with the most information at hand, and not when it is most urgent. This saves money for all parties while granting the most freedom for design choices. Construction plans can be seen as the language to exchange information among trades. In the beginning, these plans were drawn by hand. This evolved to computer-aided design but got stuck in 2D without semantical information. Even today, some plans are still created purely two dimensional. There is no way to validate these kinds of plans in an efficient and computer-based way. Errors might thus be discovered only on construction sites. BIM with IFC fosters a standard where all 2D plans are derived from one 3D model of the building, which guarantees consistency. Besides the spatial extent, this model carries detailed information for every building component of the building. Therefore, even a 2D plan has additional and correct information of properties it describes. It is an ambitious task to standardize the lifecycle of a building and to convince every participant to comply with this standard. But it is also a huge effort for companies to familiarize themselves with the spirit behind BIM. Especially, when they do not see the bigger picture, but only more work for them. It is the task of research and politics to make these advantages visible particularly in reality and not only in theory.
BIM has many facets and it would be out of scope to enumerate them all. Please see [3] for an exhaustive introduction to BIM. Still, the motivation for this thesis stems from a common problem during the construction process.

1.2 Motivation

Like in other disciplines and especially in construction, deadlines are present. They define the timeline for the building process. Deadlines are also the basis for supplements either the client or the construction company has to pay. Documentation of the construction process is a monotonic yet time-consuming task, which is also prone to errors made by humans. From experience, we know that even today this documentation is primarily paper-based.
For example, the total amount of wastewater connector is sought over several plans, because the construction company claims that in the tendering less of them are listed. The company is responsible to build according to the construction plan, but can only charge up the number of connectors stated in the tendering. The solution is to count all the occurrences of connectors on all plans by hand, which is a tedious job.
If this construction site was managed according to BIM standards, the tendering would have derived the correct amount of connectors from the 4D mode and every plan would be derived from the model, too. Therefore, the frequency of such questions would be minimal and a

short query to the model would yield the correct answer. This is the inevitable advantage of BIM driven planning and execution.
The Architectural Engineering and Construction (AEC) industry relies upon an accurate planning phase, but also needs feedback during construction. An own research branch has developed, dedicated to the question of how progress on active construction sites can be received. Embedded in the context of BIM, a comparison between the actual and the planned progress is drawn, based on a point cloud of the construction site at a specific moment. The point cloud can be obtained by a variety of remote sensing techniques or photogrammetric methods, which create a point cloud of the construction site.

Constructed parts of the building like walls, windows or ceilings, are represented in the BIM model together with their instantiation time and duration. This temporal relation is derived from a work schedule. If the construction goes according to the plan, the BIM would represent the reality accurately. The work from Braun et al. [5][6], harnesses detailed information about construction progress by comparing point cloud snapshots of the construction site with 3D geometry of the BIM. The point clouds are generated from airborne UAVs (e.g. drones) through photogrammetric methods, see [54]. They are aligned with the 3D model of the constructed building through an Iterative-Closest-Point algorithm (ICP). When the point cloud is well aligned over the 3D model, existing building components have a significant amount of points in their vicinity. Based on the work schedule, the components are compared to the construction state as "as planned" vs. "as built". This comparison gives information if a certain building component is constructed in time or delayed. In order to accurately reflect the construction state, the quality of the point cloud is of vital importance. The quality is generally not guaranteed due to occlusion of important building components and clutter or objects only temporarily on site. Further research from Braun et al. is aiming at a better perception of the construction site as a whole, see [8]. The detected building components are projected back into the images used to generate the initial point cloud from. In [26], a Convolutional Neuronal Network (CNN) is used to detected formwork, which is normally not modeled in a BIM, but occludes walls in construction process. For more details the reader is referred to [7].

1.2.1 Thesis

The main ideas revolve around the comparison between a BIM and the real-world point cloud representation. Obviously, a BIM cannot represent all objects present at an ongoing site. But objects not represented in the model cannot be captured. Thus, their appearance on a construction site cannot be transformed into knowledge. In the worse case, they lower the quality of the point cloud. Therefore, this thesis investigates a way to support the aforementioned ideas and derive additional information out of point clouds through a Deep Neuronal Network (DNN).

PointNet [46], a DNN, specifically designed for acting upon point clouds, is used to detect common objects on site. Because of the lack of appropriate training data, a generator is developed. The generator generates random point clouds by arranging mesh objects, like container, houses, excavators etc. in a specified area called a *scene*. An algorithm samples points from the surface of the mesh objects and then preprocesses them, in order to make them trainable by PointNet. Detected objects from PointNet, in turn, can help to better derive the progress state of the construction site.

1.3 Related Work

This topic is split into two parts as we discuss related work in the context of progress track on construction sites and general point cloud perception through pointwise classification.

1.3.1 Construction Progress Monitoring

Han et al. recently started an effort to collect a Construction Material Library(CML) suited to train CNNs on materials common on construction sites [20]. They use integrated information models(IIM), which again are overlays of an as-built point cloud of the construction site and the BIM. They generate the point cloud via photogrammetry and can, therefore, calculate the position of the camera which took the pictures. Based on IIMs they developed a crowdsourced BIM-guided web platform to enhance their CML. The platform processes the point cloud and aligns it with the BIM, afterwards it provides an adapted labeling tool based on LabelMe [49] for construction site annotations. This allows to incorporate material information from the BIM into the labeling process. At last, the annotated images are quality controlled.
Using the crowdsourcing approach to label images for classification is a common procedure. It becomes difficult to produce quality labels when the workforce is not familiar with the objects or materials they are confronted with. Using the Amazon crowdsource platform MTurk, this might be a valid concern. Han et al. expect their tool to circumvent these difficulties by providing strong guidance and good quality control.

In 3D reconstruction, the spatial data is not easily obtained. The techniques range from laser-scanners, 3D range cameras or stereo-photography. It either lacks accuracy, involves manual work or is expensive. An ever appearing problem with image-derived point clouds is their absolute scale. Therefore, the point clouds must be scaled manually on basis of a known length in the point cloud or their generating images. Brilakis et al. tackle the problem of point cloud scale with a novel algorithm by placing reference objects of certain measures on site [47]. The algorithm detects these objects like cubes or sheets of paper and derives the scale from them. These point clouds can then be used to generate better IIMs.
Furthermore, Brilakis et al. developed a framework which enables the user to progressively

monitor a construction site on basis of inexpensive stereo cameras [57]. His novel algorithm enhances the capturing of point clouds.

Bosché et al. [4] obtain the construction progress through laser scan derived point clouds. He proposes a system which co-registers the point cloud with the BIM using ICP. A virtual lasers scan simulation is then carried out on the BIM. The virtual laser scanner is positioned at the same position as the initial lasers scanner in reality. The simulated points are compared to the point cloud and based on the overlap, the progress can be derived.

1.3.2 Point Cloud Classification

Scaffolds occlude important parts of the building during construction. Therefore, photogrammetry fails to capture the underlying facade well enough if at all. Yet, experts can derive the current state of the construction from the existence and arrangement of scaffolds. The recent work from Xu et al. [62] deals with detection of scaffolds on construction sites.
In the first step they "crop" the scaffold out of a point cloud. For the second step, they invented a feature descriptor for point clouds, called LSSHOT, which is based on Principal Component Analysis (PCA). PCA is a method to derive the main direction of variance for a dataset by analyzing the eigenvalues of its covariance matrix. With points in $\mathbb{R}^3$, this allows them to state in which direction the points in a specified support region extend the most. This information and the robustness of their descriptor, allows deriving the different building parts of scaffolds from low-quality point clouds. Their LSSHOT descriptor was tailored for scaffolds. Additionally, a random forest classifier was trained in a supervised manner to classify the descriptors' output. This technique seems powerful but is limited to scaffold detection. Additional effort is required to transfer it to different classification tasks on construction sites where the object's features might not be as well captured as for scaffolds.

CNNs - as we will discuss in 3.1.5 - have proved their robustness in many real-life applications. These networks consume sequential data like sound and images. Point clouds, in turn, have no inherent order. This leads to the problem that a CNN cannot be trained on point clouds right away. In computer vision and in simulations, voxels constitute the counterpart to pixels and structure 3D space. Voxels can become memory intensive quite fast because to capture low level features a high resolution is needed. Unfortunately, in 3D space, the memory consumption rises by the power of three. In [61] a CNN is trained on a voxel representation of 3D space. Because of the lack of training data they also used the generator approach. Out of ModelNet10 data, they modeled realistic scenes and applied an efficient tree-based 3D volumetric object representation to it. A CNN is trained on this data. With performance in mind, this approach was fast enough to be deployed on drones for real-time classification.
This approach seems quite interesting, but the implementation is just constrained to classi-

fication of objects in one scene. But not classification of each pixel. Still, their performance is intriguing.

Chapter 2

Learning

Nowadays, Machine-Learning (ML) is a buzzword. It evokes a fascination on people, probably because learning is a property only associated with life. The recent rise of DNN and their capabilities have brought ML into public knowledge while their underlying basics are normally left out in discussions. Chapter 3 provides a comprehensive introduction to Neuronal Network (NN) while this chapter focuses on the fundamental concepts of ML, which apply to NN as well.
Before we jump into the details, we provide an introduction to the topic by investigating the problem space mathematically but also from the intuition.

2.1 The Curse of dimensionality, but Manifolds

Let's imagine a problem from solid mechanics in 3D space. Every point, describing the solid, has three coordinates. The solid is divided into 1000 elements along each axis, thus makes a total of 1,000,000,000 elements. Each coordinate value is a one-byte floating point number. We would need 3 TB alone to store the points.
Solving the equation system naively would require a stiffness matrix of 10^{18} bytes or 1 exabyte. Admittedly, $1,000^3$ is not a large problem at all. What would happen now, when we add another dimension to the space the problem lives in[1]? The number of values per points would increase to 4 and we would need $1,000^4$ elements. Which leads to 1,000,000,000,000 elements and 4,000 TB to store the points alone. These numbers are just not tradable for us at the moment. The number of memory needed to store such problems grows exponentially with each dimension we add. This is one occurrence of "The Curse of Dimensionality" [17].

[1] This might seem designed, but there is an ongoing effort to incorporate time into finite element formulation, which would be a 4D-problem. This is called space-time finite elements [11].

In ML, the space we analyze is way bigger than the problem described above. Luckily, we are searching for familiar patterns in this space and do not need to store it.
Intuitively, for image recognition, there must be high-density regions in the wide space of feature configurations, where features, we are interested in, cluster. Or, to put it differently, we would encounter familiar images by uniformly sampling each pixel in an image, when familiar patterns would be distributed randomly.

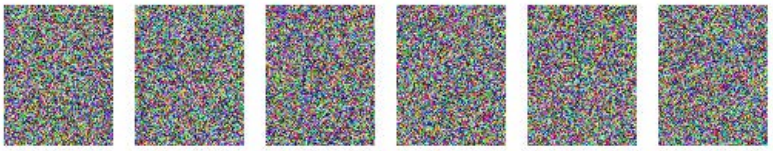


Figure 2.1: random points in search space

The images displayed above are randomly picked "points" in the design space of all images with 256-bit color and a resolution of 180×140. We can view an image as a point where each pixel values is stretched out in one dimension. Therefore, our noisy images belong to the design space of dimensions $1 \times 25,200$. They obviously do not represent any recognizable object. But this space does contain points which have meaning to us and there is a non-zero chance of picking one randomly. In comparison to our solid mechanical problem from 2.1, a space with 25,200 dimensions would be ridiculous. We can illustrate this with a small example.
Given the above images with 25,200 dimensions. We constrain each pixel's value to the range $[0, \ldots, 255]$ and sample the pixels uniformly at random. The amount of possible unique images we could create is $256^{25200} \approx 10^{60680}$. This number is incomprehensibly large[2].
Manifold theory says that we can capture regions in high dimensional space by constructing manifolds [17]. These are lower dimensional connected regions in a higher dimensional space. In three dimensions a plane or a sphere is a manifold. In dimensions higher than 4, humans struggle to visualize such entities.
We can imagine a ML algorithm to describe manifolds in the design space. Similar to the explicit equation of a plane 2.1.

$$P(x, y; a, b, c, d) = \frac{-ax - by - d}{c} = z \tag{2.1}$$

The input vector is limited to the (x, y) plane and (a, b, c, d) are the parameters which control the shape. This enables us to transition from one point to another, just by moving along the manifold (plane). As long as the manifold captures the desired data sufficiently accurate, moving on the manifold guarantees that the generated outcome is not arbitrary.
When we talk about (mathematical) spaces, inherently, the question of distance arises. How

[2] There are 10^{80} atoms in the observable universe

far are the images respectively the points apart from each other? We could use the L_1 or L_2[3] distance for example pictures. In [29], the authors used the image of a woman and distort them in different ways. It is shown that these distances completely fail to generalize and assign high distances to the images, although for us they occur as the same woman. This is not too surprising as these measures work on the lowest level possible.

Figure 2.2: Trees: Every Picture referrers the idea of a tree
(all images taken form www.pixabay.com)

Figure 2.2 shows examples on the manifold "tree". Undoubtedly, these images refer to trees but they look completely different.
Our brains are outstanding classifiers and far more complex than any NN nowadays. But at the core the brain's nerve cells also form a neuronal network with "parameters".
Our brains have their own parameters each, still we would agree that the images in 2.2 show trees. Additionally, there are more elements in certain scenes, elements like a mountain or other trees but we would still agree that a tree is in *focus*. This is the manifold "tree" intersecting with other manifolds of kind "forest" or "mountain".
We can, therefore, conclude that our brains created a manifold where the input points (image) are mapped to the idea of a "tree". Manifolds generate structure and, therefore, reduce the dimensionality of the design space. This is a rather loose interpretation of manifold theory and certainly not sound, but it helps our intuition of what ML algorithms try to accomplish. For details in manifold theory, see [56].

2.2 Machine Learning

The main idea behind ML algorithms is to convert an input to an output as a function does. The main difference is that we don't define the logic behind this function, but rather let the function figure out the logic by its own. It does so through a process called *training* where it processes examples. This is called *data-driven* approach [17].
The function is a mathematical model with tunable parameters for which the data is provided. During training, an optimization algorithm, see 2.7, tries to find suitable values for these parameters to solve a given task. Tasks define the way, machine learning algorithms process an example. An example is represented as a vector in $\mathbb{R}^n$ where every entry is called a

[3] These are just the summed differences of the two image points represented as $[1 \times n]$ dimensional vectors. L_2 is the Euclidean distance.

feature. These are values which represent the data in an example. Image pixels are a common appearance for features.
The ongoing research has brought a great variety of different techniques for problems efficiently solvable with ML. We need to formalize learning in order to be able to apply it to machines.

> "A computer program is said to learn from experience E with respect to some class of task T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." - Mitchell [39, 17]

This statement is mostly selfcontained and the details will be explained in the following chapters. We can further specify the terms *task*, *performance* and *experience* by defining building blocks for ML algorithms [17].

quality dataset	objective function
optimization procedure	model

Experience for a ML algorithm is provided through a dataset. Even large datasets are only a minimal example of the vast size of the design space they are sampled from, as we saw in section 2.1.
Therefore, it is important to put a strong emphasis on quality of the data provided to machine learning algorithms. We will have a closer look on datasets in section 2.5.
The task, what the model is meant to archive, defines the model's architecture.
In ML the objective function compares the output from a model to a provided *label*. It is called *loss* or *cost* function, see 3.1.
Generally, in optimization problems, we try to minimize an objective function with respect to its variables $\boldsymbol{\theta}$. The optimization procedure is the way we try to minimize the cost function with respect to (wrt.) the model parameters.
Two fundamentally different approaches exist in optimization. A gradient-based approach and a heuristical approach. Gradient-based approaches require the cost function to be differentiable wrt. the parameters of interest.
The heuristical approach does not need a differentiable cost function. It can be summarized by making educated guesses where the minimum in the design space is located. These guesses are, of course, backed by sound mathematical theories. Evolutionary and genetical algorithms are well-known types of this kind.
Gradient-based algorithms have proved their superiority throughout the last years because of the reincarnation of *backpropagation* as an efficient way to compute derivatives [17].
The different building blocks can be combined rather flexible, but that they work well together is not given. To combine the right blocks or invent new types is a question of research.

2.2.1 Supervised Learning Algorithms

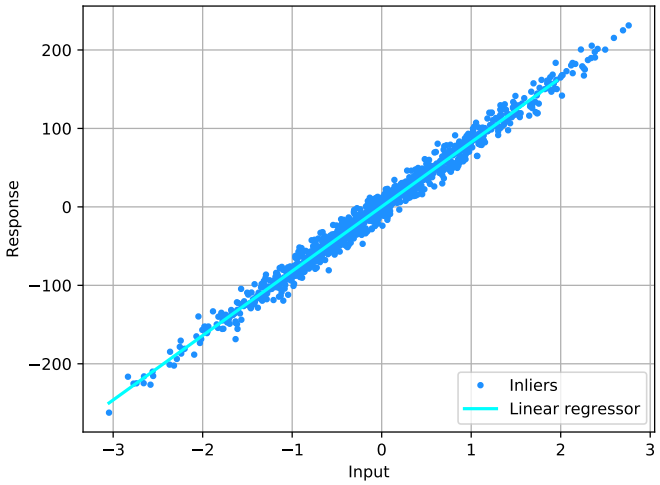
Supervised techniques require *labeled* data. Therefore, a dataset must provide a correct "prediction" for each input it contains. The algorithm's task is to find the minimizing parameters for a model on the training set by processing the input and the label. But these parameters must also minimize the cost on examples which the net has not seen during training. If it does so, we say the net generalizes well instead of learning the training set by hard.
Gathering enough data for training a model, is an expense that should not be underestimated. If a problem requires supervised learning techniques in the first place, it is most likely that there exist no other algorithms efficiently solving it. This lead to a circular dependency, which is broken by humans annotating huge amount of data [16]. See For example benchmark dataset like [35], [10], [32] and section 2.5 where different datasets are investigated in more detail.
Classical supervised tasks are regression tasks (2.3) or classification tasks as our pointwise classification of point clouds. Image classification and segmentation falls under this category as well.

2.2.2 Unsupervised Learning Algorithms

Unsupervised learning algorithms do not need labeled data. Their label is either their input or the input altered by a function.
Autoencoders are famous representatives of this type. By applying convolutions (3.1.5) and max pool (3.1.6) layers stepwise, they break down the features of their input into a relatively small feature vector where a similar net in reverse order tries to upsample the feature vector to its initial input. This can be interpreted as a very general compression algorithm.
Other techniques are applied to data where a correlation between sample points is presumed. An example is Principal Component Analysis, where a reduction in the dimensionality along the most and the less variance is sought [29].
Unsupervised learning is an exciting topic as it does not require labels, but it is also less applicable to task where external information is needed.

2.3 Linear Regression or "Hello-World"

In order to build our intuition for ML algorithms, we start with the "Hello-World" example linear regression. First of all, we need the data of interest, which is displayed in figure 2.3. We can clearly see a linear correlation between the x and y values. Next, we define the linear model for the general case for an arbitrary number of dimensions. We adopt the common convention for variables. Normal letters indicate scalars a, bold letters vectors $\boldsymbol{a}$, and capital bold letters matrices $\boldsymbol{A}$.



$$\hat{y} = \boldsymbol{w}^{\intercal}\boldsymbol{x} + b \tag{2.2}$$

Figure 2.3: dataset with linear correlation

The weights $\boldsymbol{w}$ and the input $\boldsymbol{x}$ form a scalar product where we can think of a weight influencing each input. A high value w_i increases the influence of x_i on $\hat{y}$, whereas a low value of w_i indicates that this particular input does not matter much for the prediction.
The updated equation 2.2 is just one dimensional and thus represents a line. The weight vector $\boldsymbol{w}$ and the sample vector $\boldsymbol{x}$ are reduce to a scalar.
Now, we need to connect the training data and the parameter $\boldsymbol{w}$ of the model. This is done through the loss function.
We use the Mean Square Error (MSE) [17], [59] and denote the predictions from our model as $\hat{\boldsymbol{y}}$.

$$MSE = \frac{1}{m} \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_{L2}^2 \tag{2.3}$$

We now take the gradient wrt. the weight w and set it to zero,

$$\nabla_w MSE = \frac{1}{m} \nabla_w \|\boldsymbol{x}w - \boldsymbol{y}\|_{L2}^2 \tag{2.4}$$

$$\Rightarrow \quad \nabla_w \left(w^2 \boldsymbol{x}^T\boldsymbol{x} - 2w\boldsymbol{x}^T\boldsymbol{y} + \boldsymbol{y}^T\boldsymbol{y}\right) = 0 \tag{2.5}$$

$$\Rightarrow \quad w = \left(\boldsymbol{x}^T\boldsymbol{x}\right)^{-1}\boldsymbol{x}^T\boldsymbol{y} \tag{2.6}$$

Please note that $\boldsymbol{x}$ now refers to all scalar sample points and not to a sample vector of more than one entry. This formulation resembles the normal equation for the general linear regression problem. Because we use a scalar weight, example, and label, we can define the normal equations in terms of vectors instead of matrices. A complete derivation of the normal equations 3.35 can be found in [17].

$$\boldsymbol{w} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{2.7}$$

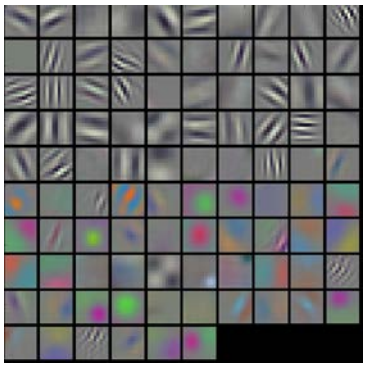
We derived a closed formula for the weights. This is only possible in the linear case as closed forms are usually unobtainable. Nevertheless, this formula contains pitfalls.
At first, we invert a matrix, which is generally a bad idea. Second, the matrix is generated from $\boldsymbol{X}$ and its transpose, which squares the condition number of the matrix we try to invert. This is even worse. The simplest solution to this problem is converting the expression 3.35 to an equation system and tackle it with the whole arsenal of linear equation solvers, iterative or direct [38].

$$\left(\boldsymbol{X}^T \boldsymbol{X}\right) \boldsymbol{w} = \boldsymbol{X}^T \boldsymbol{y} \tag{2.8}$$

We learn that even one of the simplest machine learning algorithm contains pitfalls. Our task was to predict a value y given a value x.
Our performance measure or cost function was the MSE, which we can intuitively interpret as minimizing the Euclidean distance between the labels $\boldsymbol{y}$ and the label $\hat{\boldsymbol{y}}$.
In this simple case the labels are the dataset.
The optimization procedure was the one-time derivation of the normal equations and its subsequent solving with a variety of different possible methods.

2.4 Task

A task defines the way a ML should process a given example. An example's data is called a *feature*. Structured examples form datasets.
Pixels in image processing tasks are a typical example of features. In case of NNs, features are not limited to examples. Layers can also output features or feature maps, which do not necessarily have an intuitive interpretation.



The images in 2.4 show a subset of the features in the first convolutional layer from the famous AlexNet. The net learns high-frequency gray-scale features, which are grouped in the top, and low-frequency color features grouped below [29].

Figure 2.4: Features in AlexNet
from http://cs231n.github.io/understanding-cnn/

Important to note is that the process of learning is not the task. The algorithm improves its performance on the task by learning [17]. The following is a non-exhaustive list of common tasks encountered in ML.

Classification is a task, where an input is mapped to one class i out of a set of k classes, $f : \mathbb{R}^n \rightarrow \{1, \ldots, k\}$. For instance, in image classification, an image is mapped to one pivotal point shown in the image. This could be an image of a dog mapped to the number 1 from the set of numbers {1,2,3,4}, where each number corresponds to the actual animal {"dog","cat","bird","fox"}. We often refer to classification output from a model as "hot-one" encoded. This means the output is a vector in $\mathbb{R}^n$ where the detected class k_i is one and all other entries are zero [17].

Structured output in general, enforces meaningful relationship among elements of the output vector of a NN. For example, the xy-coordinate of an element in an image, returning a hierarchical tree-structure of a sentence or most prominent image segmentation. Image segmentation can be seen as an extension of classification in a way, that every input pixel is mapped to a class.

Regression is a technique to estimate the relationship among variables. The net learns a compressed, yet not exact, representation of the dataset. This representation can predict a numeric value based on the input. It is similar to classification $f : \mathbb{R}^n \rightarrow \mathbb{R}$, except the format of the output.

A more fine-grained list of tasks can be found in [17].

2.5 Experience is Data

For ML algorithms datasets are the source of experience. We can divide learning algorithms into two categories, unsupervised learning algorithms and supervised learning algorithms. Reinforcement Learning is also mentioned here for completeness but not investigated further as its experience comes from a feedback loop not from a dataset. See [28] for an introduction.
Datasets bare an underestimated effort to obtain, yet their quality is highly important in ML. The trained model's assumptions are only based on the training data. If the model is trained on ambiguous data, it has no chance to perform satisfyingly on real-world data. There is generally no simpler way to generate annotate real-world data than to do it by hand. Otherwise, there would not be the necessity for ML in the first place.
Research groups provide datasets for learning tasks they work on and for others to independently verify their findings.

2.5.1 Dataset Selection

The **MNIST** database of handwritten digits [35] contains 70,000 images (28x28) divided into 60,000 training and 10,000 test images. It is a subset of the larger NIST dataset [31]. Each handwritten digit is associated with its real digit. The images are gray-scale, small in size and the label describes the image completely, which makes it ideal for introduction to deep learning.

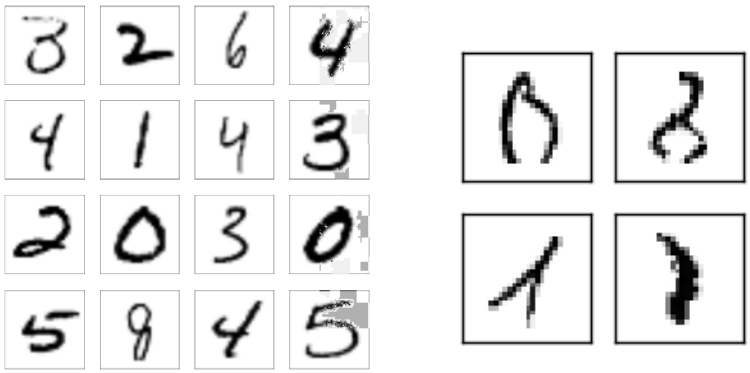


Figure 2.5: left: random sample from MNIST,
right: degenerated data

Many different techniques have been applied to the MNIST dataset e.g, NN, CNN, support vector machines and k-nearest neighbors. The mean error rate on MNIST for classification from the top ten performing architectures is 0.28% [2]. This means that not even three digits out of 1,000 are misclassified. This is even more impressive when compared with the degenerated examples from figure 2.5.

The **CIFAR-10** dataset consists of 60,000 colored 32x32 images, 50,000 for training and 10,000 for testing. It comprises ten classes like *car, deer, truck* or *dog*, which describes the pivotal part of the image. **CIFAR-100** extends the labels to 100 where each image is additionally classified by one superclass. For example, "fruit and vegetables" is the superclass containing "apples", "mushrooms", "oranges", "pears", "sweet peppers". The mean accuracy for the top performing architectures is 94.94 [2]

Figure 2.6: random sample from CIFAR dataset
https://www.cs.toronto.edu/~kriz/cifar.html

ImageNet is an ongoing research project which tries to organize images in a tree of *synsets* [23]. A synset groups similar pictures together and names their content. Each synset can have zero or many subsets, which describe the images in more detail. The deeper we traverse the hierarchy the more specific the image description becomes. An image contained in a leaf node inherits all the categories from its parents thus forms a transitive relation. ImageNet is inspired by WordNet which organizes English words in a hierarchical manner. Figure 2.7 shows an example for the word *star*.

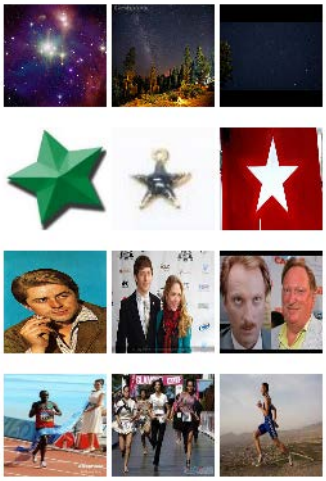


Figure 2.7: ImageNet synsets for star

The topmost example is from the synset "star", at level 5 in the tree. The description reads: "Any celestial body visible (as a point of light) from the Earth at night". Its supersets are: Root → natural objects → celestial body, heavenly body → star.
The second row images are from another synset called "star" at depth 5, but it is reached differently. Root → Misc → plane figure, 2-dimensional figure → star. And described like: "A plane figure with 5 or more points; often used as an emblem".

There are 40 synsets associated with the word star in the same manner described above e.g."rock stars", "track stars" or the "starfish". Most synsets refer to plants with the word star in their names. Considering that there are more than 14,000,000 images and 21000 synsets indexed in ImageNet, we get an impression of how much effort is put into generating quality datasets.

2.6 Performance

We divide the term performance into two parts. On the one side the objective function and on the other a measurable quantity.

2.6.1 The Loss

In 2.3 we encountered our first loss function, namely the Mean Square Error. In the deep learning community, the term loss and cost function are used interchangeably.
An attempt to differentiate them both is, that the loss (function) refers to the value we

get after processing one example or minibatch from the dataset. The cost (function) is the accumulated loss for the whole training set. The term objective function stems from the optimization community and is not used very often in the context of deep learning (dl). Throughout this thesis, we use cost and loss interchangeably, but stick to this definition when appropriate.
One huge drawback is that the loss function does not carry explicit meaning. Is a loss of 42 good in our context or not? This is hard to interpret. Further, in classification, we infer our model with test data and obtain a probability distribution over all classes. This is simply too much information. What we want to know is if the model predicted the example's class correctly.
We can conclude that the loss function is the mathematical connection between the prediction of the ML algorithm and the corresponding label. Because its differentiable wrt. the model's parameters, it allows calculating the gradient of them. In 2.7 we will learn a method, which uses the gradient to drive the parameters in the right direction so that the loss becomes small. The specific way how the gradient is obtained is discussed in 3.3.

2.6.2 Measure

Performance measure indicates how good a ML algorithm performs on a task in a human-interpretable way. It mainly depends on the task which measure is applicable. We can imagine a performance measure as the distilled information from the output of a machine learning algorithm (mla).
For classification tasks, the measure is *accuracy*, which is the ratio between all examples and the correctly classified examples. The error rate is defined as $error\ rate = 1 - accuracy$. For image segmentation tasks, this is straight forward, as this is a pixelwise comparison.
For tasks where a machine learning algorithm is meant to predict a bounding box normally the Intersection over Union (IoU) is used. This is the union area of the ground truth and the predicted bounding box. So a IoU of 1 is best, whereas 0 is worst. [17].
A careful analysis has to be done before designing a performance measure because it is not always clear what exactly depends on it.
In the context of point clouds, this seems straight forward: Accuracy, by pointwise comparison. This is also the way it is implemented in code. A further idea was to define a threshold for correct classified points per object and measure how many objects the network would detect correctly. This measure is weaker compared to the pointwise comparison but would give better insight into real-life applications 4.4.

2.7 Gradient Descent

Let's revisit our model from the "Hello-Word" example in section 2.3. The solution provided worked quite well. With the huge variety of linear solvers bigger and more complex dataset should not be a problem to train. But one crucial assumption still limits us to come up with a general machine learning algorithm: Linearity.
What if our model introduces non-linearities? There is no way to state the normal equations anymore, and therefore we have to use a different technique called Gradient descent [17], [38]. Gradient descent is an iterative method for minimizing an objective function. In its simplest form it can be written in the following manner:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \tag{2.9}$$

In other words, we are looking for a parameter set $\boldsymbol{\theta}^*$ which minimizes the objective function $f(\boldsymbol{\theta})$. We start with a random guess of our parameters $\boldsymbol{\theta}_0$ and follow the negative gradient "downhill". In reality, $\boldsymbol{\theta}_0$ is chosen in a smart way to help the gradient find step areas on the manifold. This hill-climbing analogy does, of course, only hold in three dimensions whereas ML problems can have millions of dimensions.
The η in front of the gradient expression in equation 2.12 is the so called *learning rate*. It is our first *hyperparameter*. In the hill-climbing analogy, it is the step size we take, as we move downhill. The implication is that large steps might miss the minimum while tiny ones take too long to finally arrive. It is our task to find a balancing learning rate. Plenty of effort needs to be carried out to answer the question of the optimal learning rate under certain circumstances, but in the end it depends on the specific setup [17], [42], [9]. Some advanced methods try to adept it on the fly [30].
We could also tackle our normal equations with gradient descent and if our weight matrix happens to be positive definite then gradient descent is guaranteed to converge, too [38].
But this assumption does not hold any longer for NNs, as the non-linearities in the activations convert the initial problem to a non-convex optimization problem. Giving up the field of convex optimizations with all its beneficial properties is a risky step. There is no guarantee anymore that we will find a global minimum of our objective function because suddenly there are local minima and saddle points all over the optimization space.
Nevertheless, research has shown that the benefits overcome the disadvantages, as NNs have no problem to describe non-linear relations in the dataset which is their most important advantage.
We stated the general method of how we solve for an optimizing vector $\boldsymbol{\theta}^*$, but this does not answer the question of how we obtain the gradient. Even worse for NNs, because their gradients have the same dimension as the number of trainable parameters. Section 3.3 explains the procedure of how we can compute the gradient for NNs in an efficient way and enable us to use gradient-based optimization for NNs.

2.7.1 Stochasitc Gradient Descent

A great variety of different gradient descent flavors exist[4]. In case of ML, Stochastic Gradient Decent (SGD) is the most important one. We will discuss it in the context of NNs, but it should be noted that it is not limited to it.
Equation 2.14 states a general loss function. The loss calculated from the loss function always refers to the whole training set (note the sum from 1 to m, which is the number of examples in the training set) [17].

$$C(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{x}^i, y^i, \boldsymbol{\theta}) \tag{2.10}$$

The gradient would just be $\nabla_{\boldsymbol{\theta}} L$, summed over all examples m and normalized by $\frac{1}{m}$. Therefore, the gradient must be computed for all training examples at once to obtain **one** step in the right direction.
Section 2.5 showed that current datasets contain examples up to hundred thousands, which is prohibitively large to calculate the gradient for. This gradient would be the most exact representation of the gradient at the position $\boldsymbol{\theta}_i$ wrt. the whole dataset.
Fortunately, SGD allows us to approximate the gradient. We just sample a subset from the training set called a *minibatch*. With the minibatch, we incidentally introduced another hyperparameter which influences our training results. Only the examples from the minibatch are used in SGD to calculate the gradient. The resulting gradient is only an approximation for the real gradient on the whole training set at position $\boldsymbol{\theta}_i$ but still good enough. We obtain

$$\boldsymbol{g}(\boldsymbol{\theta}) = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^i, y^i, \boldsymbol{\theta}), \tag{2.11}$$

for the gradient [17]. Notice that m' is only the size of the minibatch. Using this method decouples the gradient calculation from the total number of training examples. Our initial equation for gradient descent changes now to

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \boldsymbol{g} \tag{2.12}$$

Calculating the gradient is a constant matrix-vector multiplication and storage. The examples can also be processed independently from each other. This task is asking for parallelism. There already exists a device made for exactly these two purposes: graphics cards. Normally, the minibatch size is aligned to fit the graphics card's memory such that one minibatch occupies the most memory. This way the graphics card is used efficiently and a good starting point for the hyperparameter "minibach size" is obtained, too.
SGD structures training. Let's assume a training dataset with 1000 examples and an optimal minibatch size of 10 examples. We would need to process 100 minibatches, in order to cycle

[4] Conjugate Gradient, Momentum based methods, etc

through the complete training set once. One cycle is called an *epoch*, which is yet another hyperparameter we need to get right.

2.8 Validation

In section 2.3 we derived linear regression with generated data. For such simple cases where we have a linear correlation in 1D, we do not need much testing. We can verify the result by looking at the plot[5]. For higher dimensional data in classification tasks, we need another way to verify our results.

2.8.1 Overfitting

ML models have *capacity* which determines the model's ability to fit certain function families. Obviously, a model, which is linear in its input, simply cannot represent quadratic correlation even with infinite examples. But using a very general model with too much capacity for the example data, leads to the phenomenon called *overfitting*. We need to find a balance between the model's ability to fit the data while not overfitting it.

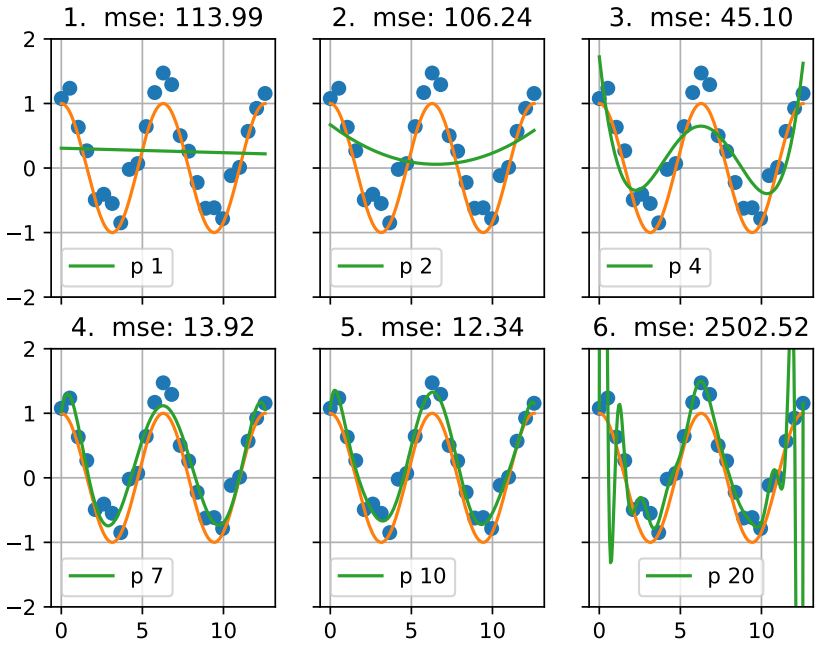


Figure 2.8: model capacity

[5]This is generally discouraged and ironically called the "eye-norm"

Figure 2.8 shows a linear regression model[6] with increasing polynomial degree p. The polynomial degree is directly correlated with the model's capacity. It's our task to find a suitable one. This p is another parameter in the series of hyperparameters which influences our model's behavior.
For the first three values of p the model clearly struggles to find a suitable function to represent the data. We say the model is biased (towards certain values) but has very low variance. Its capacity is simply too low, we call this *underfitting*. We can see that with a polynomial degree of 7 the model fits the data best.
The last plot shows the model which is almost not biased at all(it fits the values almost exactly) but has a very high variance (especially on the edges of the plot). Thus it highly overfits the data.
How do we find the right parameter for p? We use a performance measure, in this case, the Mean Square Error.

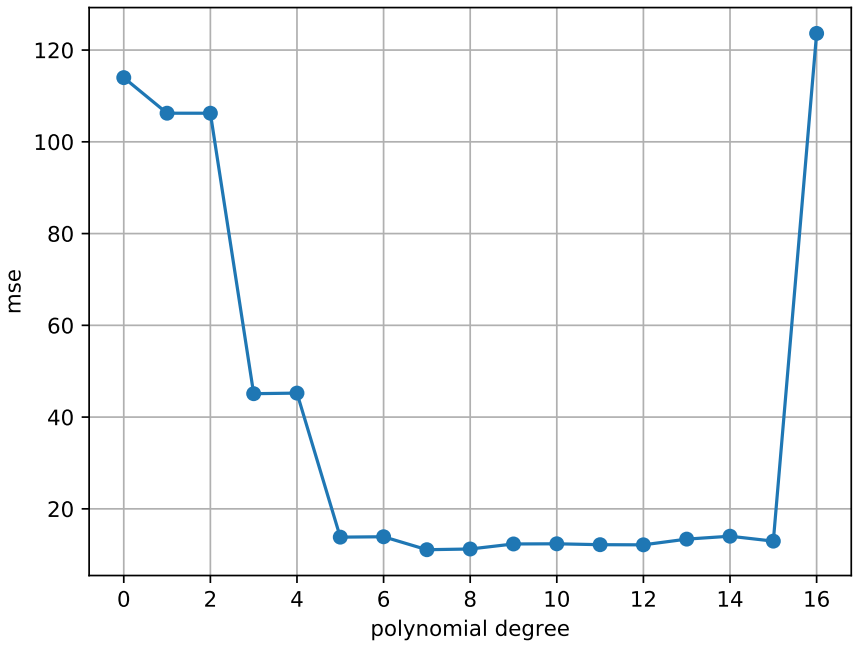


Figure 2.9: hyperparameter tuning

Plot 2.9 shows a simple analysis, where we plot the MSE against the polynomial degree. The plot reveals the optimal choice for p equal to 7.
Other degrees would yield an equally good result, but an important rule in ML is that one should always use the least complex model. Actually, this does not only apply to ML and is commonly known as *Ocam's Razor* [17].

[6]The model is linear with respect to its parameters θ but not in its input. As long as the model is linear in its parameters we refer to it as linear regression [17].

This kind of analysis is necessary when models depend on hyperparameters. When the model's training phase is time-consuming, tuning such parameters is an involving task.
Remark: We picked a polynomial function to fit a cosine with noise. This sound like we used the wrong model. In real-world application, we have no knowledge of what the underlying distribution might be, so using polynomials is not a bad guess. They are easy to handle, and can become complex enough through a simple increase of the polynomial degree.
However, it is possible that the model is just not the right one to satisfyingly fit the data. The only possible solution, in this case, is to use another model.

2.8.2 Training, Validation and Test Set

In previous derivations, we did not distinguish between different parts of the dataset nor did we attempt to verify our results. In section 2.8.1, we found a promising value for p, but the only thing that we could conclude was that it fits our data. But is it robust for real-life application, too?
A common standard has emerged in order to classify the performance of machine learning algorithms and their hyperparameters. The dataset is divided into three parts, namely *training, validation* and *test* set, as shown in figure 2.10. The training set is chosen to be the largest one where the model is trained on.
The validation set is a 10%-20% subset of the training set [29]. The sets must not share any examples among each other.

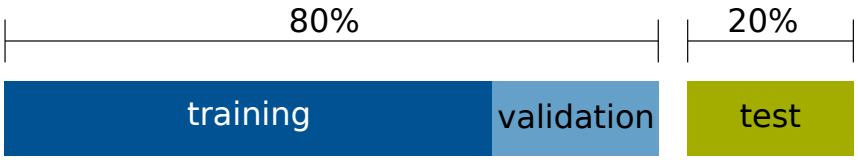


Figure 2.10: dataset split

The validation set is used to validate the accuracy of the model on examples the model has not seen before. We, therefore, obtain a training error and a validation error. The error of the validation set is obviously bounded by the error on the training set. Or put differently, the error on the training set is lower than the error on the validation set.

$$error_{train} < error_{validation} \tag{2.13}$$

An extended training on the training set, when the training error has already saturated, would increase the validation error. Our model would simply learn the noise from the training set. Exactly this is *overfitting*. But now we have a tool at hand to indicate it.
We can now compare the training error and the validation error throughout the training, and determine the overall performance of the network. We can also tune our hyperparameters.

See figure 2.11 where the green curve can be interpreted as the validation error.

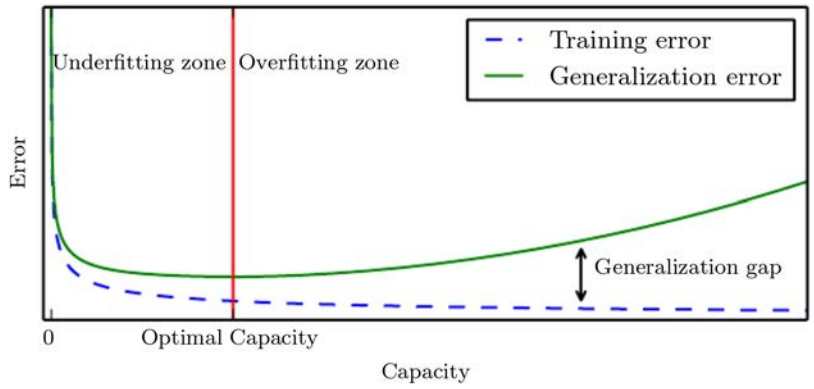


Figure 2.11: idealized curves for a training run
figure taken from [17]

Both errors decay rapidly in the beginning because the machine learning algorithm captures the direction with most variance relative easily. They both level off after a certain amount of training examples. The validation error starts to increase slowly whereas the training error constantly decreases. At this point, overfitting has begun. Normally, the two errors tend to oscillate a lot, therefore, it is not immediately clear when this point is reached.
Equipped with this knowledge, we can now tweak the model's hyperparameters until a sufficient performance on the **validation set** is reached. We now know that the model has captured the underlying distribution well. Unfortunately, the model is now biased towards the validation set because the hyperparameters depend on it.
In order to still see how good the model has learned to generalize, the performance on the test set is evaluated. Like the examples from the validation set, the test set's data has to be from the same distribution but must not contain an example from the validation nor the training set. After testing the model on the test set, its hyperparameters must not change anymore to make a valid statement of the model's performance.
On first sight, we might think that a low training error is desirable under any circumstances, but this is definitely a question of balance between learning noise and generalization. Take for instance a dataset of human faces. Clearly, the skin color would be biased, because it is impossible to cover all existing skin colors in one dataset. One way we can think of overfitting is that the model would match the skin color of noses precisely and assign lower confidence to noses with slightly different colors. We can conclude that constant training on the train set might increase performance of the model. But if overfitting happens we give up the most desirable property of ML: The ability to generalize beyond the training set.

2.9 Regularization

We seek the minimum of a loss function with respect to a given training set, but hope that it also minimizes the corresponding validation set. If this is the case, we say that our learning algorithm *generalizes well*. An extended training on the training set, when the training error already has saturated, would increase the validation error. Our model would simply learn the noise from the training set. This is again *overfitting*.
We can tackle overfitting with regularization. With gradient descent, the gradient and therefore the machine learning algorithm's weights are not constrained mathematically. Nothing prevents single weights to become relatively large and dominate the neuron's activation. We can add a regularization term to our loss function which constrains the values of the weights either absolutely or indirect.

2.9.1 L2 Regularization

The L2 regularization is very common throughout NNs design [29].

$$C(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^i, y^i, \boldsymbol{w}) + \frac{1}{2}\lambda \boldsymbol{w}^2 \tag{2.14}$$

We just add $\frac{1}{2}\lambda \boldsymbol{w}^2$ to the loss function and introduced another hyperparameter λ the *regularization strength*. We can interpret this term as a constraint for weights to not get too large because any large weight would contribute quadratically to the loss. Therefore, the weights layers become diffuse. The L2 regularization is also known as weight decay because through the nature of this formula the weights decay with the rate $-\lambda \boldsymbol{w}$[7].
As the regularization term in equation 2.14 is multiplied by the learning rate anyway, we are allowed to just divide the whole term by half. This is for convenience, and simplifies the derivative. Figure 2.12 shows data which is divided linearly with some outliers.
In the left image the network overfits the data as it splits it very precisely. The middle network manages to capture the linearity better, whereas the right model accepts the outliers as noise. This is accomplished with different regularization strengths.
Side note: These 2D representations of the NNs decision boundaries are only made possible through low feature dimensionality. These can be interpreted as manifolds diving the design space, as we learned in the section 2.1.

[7] It is clear that the regularization term in 2.14 drives the weights towards zero. Unfortunately, this sharp distinction depends on the optimization methods and their actual implementation, as shown in [36]

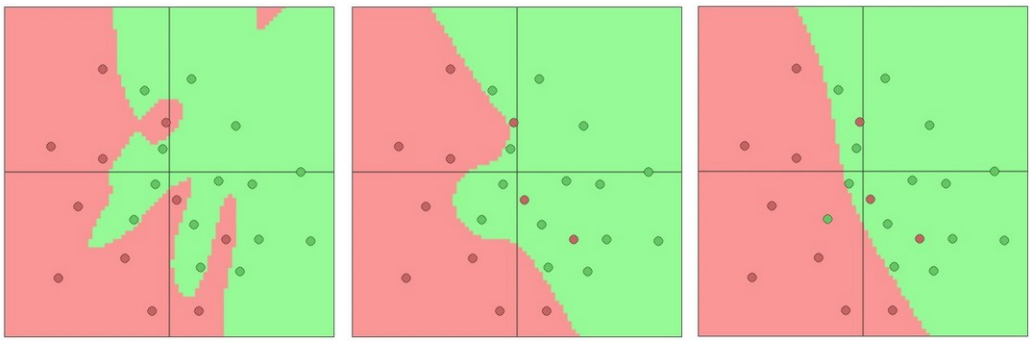


Figure 2.12: Regularization
taken from [29]

2.9.2 Dropout

Dropout has become a standard regularization method, due to a simple yet efficient idea. During training, neurons per layer become deactivated with a hyperparameter probability p. We assume a layer with 100 neurons where dropout is applied. With $p = 0.5$, 50 out of 100 neurons would just not contribute to the next layer's activations. Figure 2.13 visualizes this idea [53].

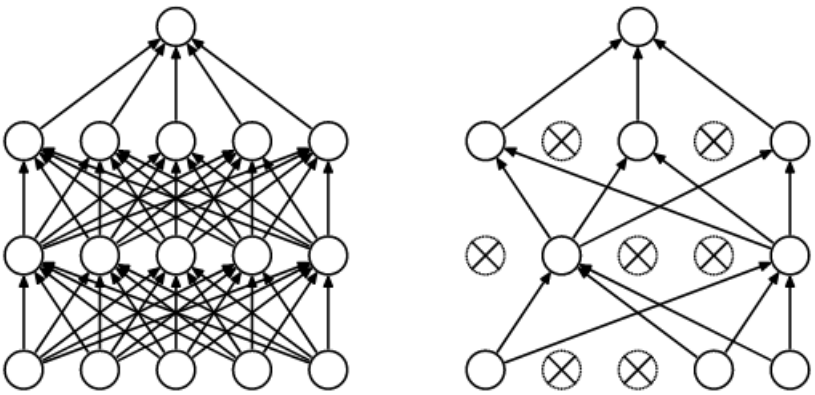


Figure 2.13: dropout visualized
taken from [53]

The training phase of the network can be seen as a training of exponentially many subnets with shared weights. During the test phase all neurons are used, but with their weights scaled.
The implication is that the neurons do not co-adapt as they would when trained normally. They become more robust with respect to the activations received from the layer below and cannot form complex relations which work well on the training set but fail to generalize well. The motivation comes from evolution where sexual reproduction also depends on genes from

two genders plus some random mutations. The argument is that genes which work together in random combinations are more robust than from organisms which reproduce on their own genes [53].

2.9.3 Other methods

There are many different methods which are not described here in detail because of limited scope. For completeness they are mentioned here. Like L2 regularization, the same method can be stated without the quadrature, which is then called L1 regularization, see [29],[41]. This regularization drives the weights to become sparse which a few dominating weights. This makes the network very robust against noise.
An even simpler method is called max norm constraint where an upper bound for each weight is defined. Therefore, the network cannot explode during training.
A recently introduced method is batch normalization, which yields quite promising results for real applications, see [24].

Chapter 3

Deep Neuronal Networks

We have merely scratched the surface of ML in chapter 2. Nevertheless, in the following chapter, we want to shed light on Deep Neuronal Networks, which are often coined black boxes. DNNs are a mighty subcategory of supervised machine learning algorithms. They have gained a massive hype since Alex Krizhevsky won the ImageNet competition in 2012 [33]. DNNs proved their capabilities in the following years. They allowed for breakthroughs in vision, speech recognition, and even gameplay. AlphaGo was a DNN trained by Google, which improved its play style by repeated matches against itself [51]. Go is considered the most complex game in terms of possible game scores. In comparison to chess, no heuristic approach is applicable, thus it stood unmanageable for computers to comprehend for a long time. But eventually, AlphaGo beat the Japanese Go master Lee Sedol in 2016.

The design of NN neuronal networks is inspired by the human brain. One of the most intriguing physical objects on planet earth. It is capable of perceiving light, sound, heat etc. as well as feel emotions, and deduce complex interactions with the environment out of it.
But interestingly, the brain's complexity arises from a rather small object called a *Neuron*. Neurons are interconnected with each other and therefore form a neuronal network. A neuron *fires* when the input from all previous neurons it is connected to add up to a given threshold. This leads to a cascading effect, which only in sum allows complex computations.
Like any human adaption from nature, a neuron in an artificial neuronal network has not much in common with the physical object in our brain. It is rather a mathematical description of components which build up and resemble the neuronal network in our brain.

Starting off at the beginning with the most simplest neuron, the so-called *perceptron*. The idea of a perceptron was first introduced by Warren S. McCuloch and Walter H. Pitts in 1943, who investigated biological neurons. And later in the 1960th picked up by Frank Rosenblatt who discovered the mathematical theory around it [48].

3.1 Layers and Activations

DNN are stacked layers of neurons where the following layer consumes the output of the previous layer. The type of layer defines the way the neurons are interconnected among each other. The connections can be depicted as a graph. We will only focus on *feedforward* DNN without *skip connections*. Feedforward means that there are no loops allowed in the computational graph, like in a Recurrent Neuronal Network (RNN). Skip connections allow connection beyond the immediate layer, like in ResNet [21]. We start with the predecessor of neurons in a NN the perceptron, and form our idea of DNNs as we investigate more complex neuron arrangements.

3.1.1 Perceptron

A perceptron is a function which takes an input vector $\mathbf{x} = \{x_1, x_2, \dots x_n\}$ dots it with a weight vector $\mathbf{w} = \{w_1, w_2, \dots w_n\}$ and subtracts a *bias* b from it. We denote this result as a *logit* and call it z. The logit is than processed by a *activation function* $f(z)$, if the result of the activation function is larger or equal than zero the perceptron's output will be 1 otherwise 0. We will see different kinds of activation functions in the following sections. For now we simply assume $f(z) = z$. We can formulate an equation which describes a linear perceptron:

$$p(\mathbf{w}, \mathbf{x}, b) = \begin{cases} -1 \text{ if } f(\mathbf{w}^T\mathbf{x} - b) < 0 \\ \ \ 1 \text{ if } f(\mathbf{w}^T\mathbf{x} - b) \geq 0 \end{cases} \tag{3.1}$$

Please note that, *linear* refers to the activation function not the fact that we dot the input and weight vector, which is similar to linear regression. We can interpret weights as a measure of how much a certain input should contribute to the function's output. A high value favors a given input, whereas a low or even negative one does not. The bias can be seen as a parameter which simply shifts the activation function, see 2.3.
At first glance, nothing would limit us to build an artificial neuronal network with perceptrons. But unfortunately, the perceptron is not mathematically continuous. It is only capable of evaluating to 0 or 1. As we will see, training makes heavy use of the gradient[1], thus this is not a desired property of perceptrons.
Furthermore, perceptrons are not universal, for instance, they are not able to learn the XOR equation, which was a discouraging outcome for the inventors [17].
The perceptron was the initial attempt to recreate real neurons behavior. But we have already seen a similar function, as we developed our first linear regressor in 2.3, which had a more appealing property as it was at least continuous.

[1] Of course, the field of evolutionary algorithms overcome this drawback. But these algorithms are not deeper investigated in this thesis as the prevailing algorithms are gradient based.

3.1.2 Multilayer Perceptron or Fully Connected

We can state a more suitable form of the perceptron and call it a neuron.

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} - b \tag{3.2}$$

We encounter that a neuron is the same as our linear regressor stated in section 2.2. We could stack more of them and form a layer [29].

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x} - \boldsymbol{b} \tag{3.3}$$

What changed now is that we order the weight vector into a matrix $\boldsymbol{W}$ with dimensions $n \times m$ and a bias vector of dimension m as. The dimension m corresponds to the amount of neurons per layer, whereas the dimension n indicates the dimensions of the example data $\boldsymbol{x}$. We have made ourselves a classifier out of many neurons. The output dimension is the same as the input dimension $\boldsymbol{x}$. Nothing stops us from passing the output from the first layer into another one.

$$\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{W}_2^T\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{b}_2 = \boldsymbol{W}_2^T(\boldsymbol{W}_1^T\boldsymbol{x} - \boldsymbol{b}_1) - \boldsymbol{b}_2 \tag{3.4}$$

We build ourselves a fully connected network. In deed, this is a small network as it only consists of an input and an output layer. We could stack as many layers this way as we want, which we call hidden layers.
It is important to note that $\boldsymbol{W}_2, \boldsymbol{W}_1$ do not need to share the same dimension m, as long as they are the same in n, thus n corresponds to the number of neurons we want have in the layers.
We have passed the output from one layer to the next unchanged. This is problematic as shown in equation 3.5 [38].

$$\boldsymbol{A}_1 * (\boldsymbol{A}_2 * \boldsymbol{X}) = (\boldsymbol{A}_1 * \boldsymbol{A}_2) * \boldsymbol{X} = \boldsymbol{A} * \boldsymbol{X} \tag{3.5}$$

Activations are necessary to break the linearity between the layers in a DNN. Without them we could stack as much layers as we want but still obtain a linear classifier.
If we added a non-linearly to the output of a layer, the associative property of the equation 3.5 would not hold any longer. We just denote the activation function with $\sigma(x)$, and apply it elementwise to the output of $f(\boldsymbol{x})$.

$$\boldsymbol{f}(\boldsymbol{x}) = \sigma(\boldsymbol{W}^T\boldsymbol{x} - \boldsymbol{b}) \tag{3.6}$$

Because we applied f elementwise, nothing changed in the output dimensionality of $\boldsymbol{f}(\boldsymbol{x})$ and we can further use it to bulid up layers.

$$\boldsymbol{g}(\boldsymbol{x}) = \sigma(\boldsymbol{W}_2^T\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{b}_2) = \sigma(\boldsymbol{W}_2^T\sigma(\boldsymbol{W}_1^T\boldsymbol{x} - \boldsymbol{b}_1) - \boldsymbol{b}_2) \tag{3.7}$$

This is the most simplest from of a DNN, also known as the Multilayer Perceptron (MPL) (although we actually used the formulation of a neuron rather than a perceptron)[17]. How the output is interpreted depends on the choice for σ. We are not bound to always use the same σ as our activation function. As long as the function is differentiable with respect to the weights, we can use it. Commonly, hidden layers share a type of activation and only in the last layer a different function is chosen [42].

3.1.3 Softmax and Sigmoid

Let us assume a DNN for a classification task where we do not treat the output in any way different than the inner layers. These output values, or scores, would be hard to interpret. They are not bounded, can be negative and they do not share any relationship among each other. The only measure we would have is that we assume that the highest value is the class score, as done with Support Vector Machines [29].
Fortunately, the *softmax* function provides an escape from this dilemma and further opens up new ways of interpreting DNN predictions [17], [29].

$$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \tag{3.8}$$

The softmax function takes a vector $\boldsymbol{x}$ and maps it to the same vector space, $f : \mathbb{R}^k \to \mathbb{R}^k$. It squeezes the values in the range $[0, 1]$ and makes them sum to 1. This looks suspiciously like a discrete distribution over all classes k. Of course, we intuitively talked about predictions, in a

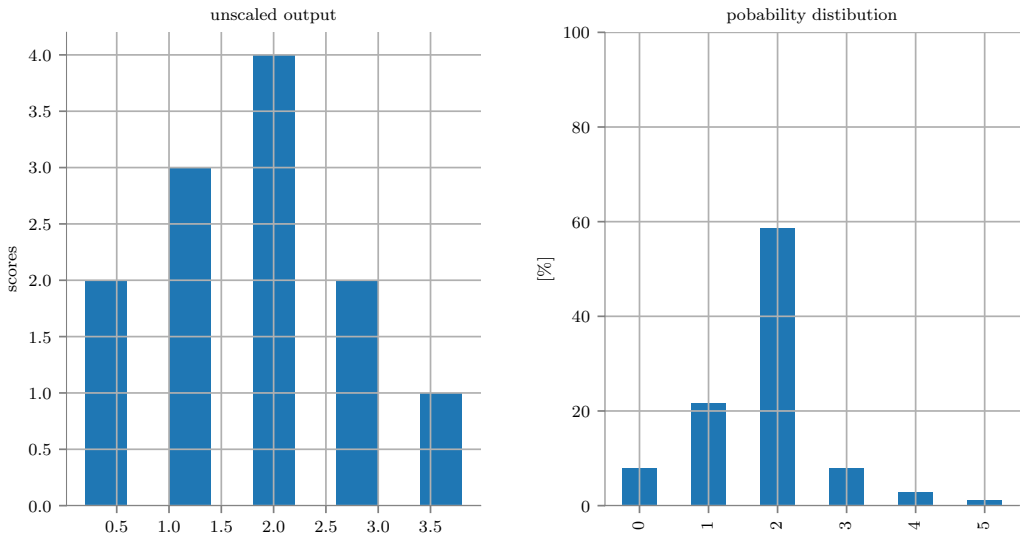
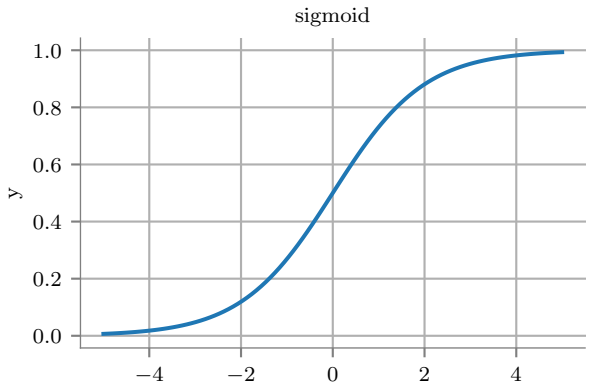


Figure 3.1: softmax function applied to scores

statistical sense, when we mean the output of a DNN, but strictly speaking this is only made possible through special treatment in the output layer. The binary counterpart of softmax is the Sigmoid function [59], see 3.9.



$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.9}$$

Figure 3.2: Sigmoid function

It squeezes a number from $\mathbb{R}$ into the range from $]0; 1[$. In former years it was even used as an activation in hidden layers. The Sigmoid is discouraged today because its learning slows down tremendously when the weight is very wrong. A small example illustrates this. The derivative of the sigmoid function can be written in terms of the Sigmoid function itself.

$$\sigma'(z) = (1 - \sigma(z))\sigma(z) \tag{3.10}$$

The complete derivation can be found in [17] or [29]. We can now plug in different values and observe that for high or low values the gradient is very small as shown in figure 3.3. We will see in section 3.3, that backpropagation is based on repeated multiplication of local

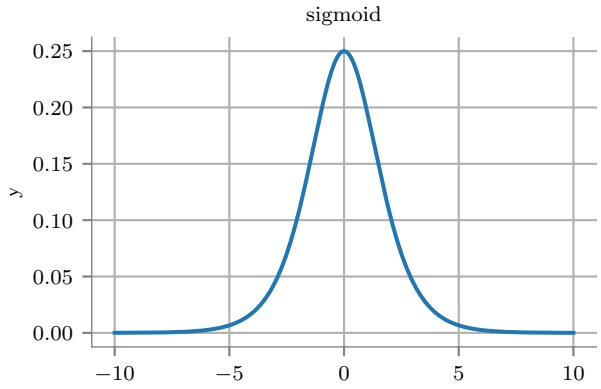
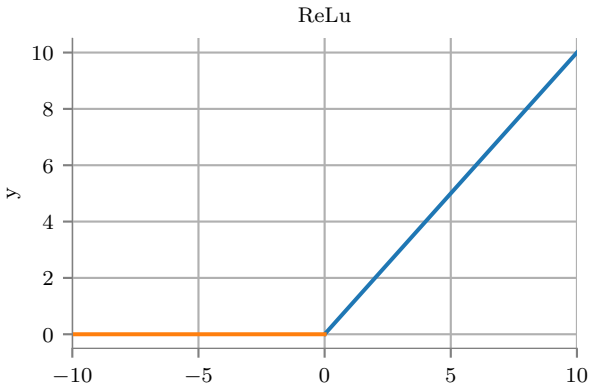


Figure 3.3: derivative of the sigmoid function

gradients. When the gradient is small multiplication will enhance the error, which leads to the phenomenon of *vanishing gradients* [29].

3.1.4 ReLu

In oder to overcome the problem with vanishing gradients the *Rectified Linear Unit (ReLu)* was developed. The ReLu is extremely simple. It adds the non-linearity by constraining the input value to zero when it is below zero.



$$ReLu(z) = max(0, z) \tag{3.11}$$

Figure 3.4: Rectified Linear Unit (ReLu)

The figure above shows the plot with the zero values in orange. The ReLu is computationally way more efficient than the Sigmoid function. It does not even involve arithmetic operations. It allows significantly faster training, see [29]. Unfortunately, ReLus can be become locked either by wrong initialization or unfavorable gradients. Imagine a neuron with a negative weight w_i bigger in magnitudes than the remaining weights and bias for this neuron. This would influence the neuron such that the ReLu would always be zero. In turn, no gradient would flow through it anymore. Whole parts of the network could be locked this way. There are remedies for this scenario as Leaky ReLus or *maxout* [17]. But in general, right weight initialization and good regularization should be enough to not run into problems.

3.1.5 Convolutional Layers

Layers, other than fully connected, have the purpose to reduce the number of connections per layer by exploiting a certain structure of the example data. For convolutions, this structure comes in form of sequential data which can be exploited when the data in the vicinity of one feature helps to understand the dataset in depth. A simple example helps to explain this idea.

Figure 3.5 shows a convolution in one dimension. The gray squares indicate the input. The yellow squares show the output. The output is a simple dot product between the green weights and the input (a bias term is omitted for brevity). The amount of weights we use per layer is a hyperparameter F and called the *receptive field* [29].
Additionally, there is the *stride* S, which specifies the value the receptive field slides along one

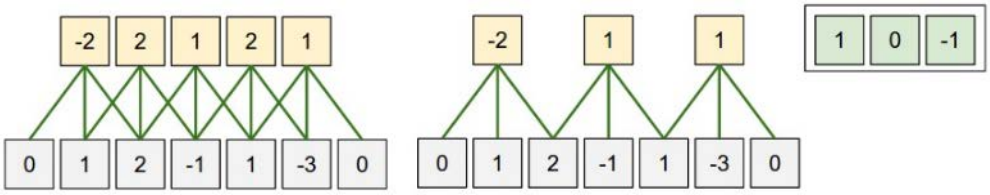


Figure 3.5: Convolution examples in 1D. Left example: stride S = 1, padding P = 1, receptive filed F = 3, input W = 5. Output is therefore $(5 - 3 + 2 * 1)/1 + 1 = 5$. The left example is similar only the stride changed to S = 2; The output is $(5 - 3 + 2 * 1)/2 + 1 = 3$.
(Image and values taken from https://cs231n.github.io/convolutional-networks/).

dimension. The stride is 1 for the left example and it is 2 for the right example in figure 3.5. The input vector is extended with zeros at the beginning and at the end. This is called padding P, and is an optional hyperparameter. It extends the input dimensions artificially with zeros such that the receptive field and the stride fit together or to produce a certain output. If the padding was not there on the left example the output dimension would only be 3 instead of 5.
We can use the following formula to calculate the number of output neurons

$$O = (W - F + 2P)/S + 1 \tag{3.12}$$

where W is the size of the input vector and O is the output [29]. See figure 3.5 for calculations. Convolutions are mostly explained by a sliding window analogy which can be misleading sometimes. When we expect the neurons (yellow squares) to be there and ready to calculate their output by shared weights, then there is no need to use a sliding analogy as it suggests motion. It is only useful to explain the reason how the exact amount of neurons have come together. Also in actual implementations, no sliding is carried out.
Doubtless, CNN became famous for their great ability to percept images. In 2012 Alex Krizhevsky [33] won the ImageNet competition with a CNN trained on his computer at home. He rediscovered the work from LeCun in 1989 where he tried to read handwritten postal zip code [34]. Therefore, the idea to use convolutions as learnable filters is actually older than commonly expected. Unfortunately for LeCunn, back in his days the hardware and datasets were not good enough for a major breakthrough.
In the same manner, as we did it for one dimension, we can formulate convolutions in higher spaces. For example, for a 3D tensor with spatial extensions $[Hight \times Width \times Depth]$. This accidentally coincides with the representation of images in computer memory, where the depth is 3 and maps to the colors RGB (red, green, blue).
We define an oversimplified input gray-scale image X with dimensions $[7, 7, 1]$, see figure 3.6. We now need to group our weights to conform to the new input dimensions. It is common to use a symmetric receptive filed and therefore we define our weights as $[3 \times 3 \times 1]$ (blue). The depth equals the depth of the input tensor. In order to understand the output dimensions, we now slide the weights along one dimension with a stride of one. Because of symmetry, it is

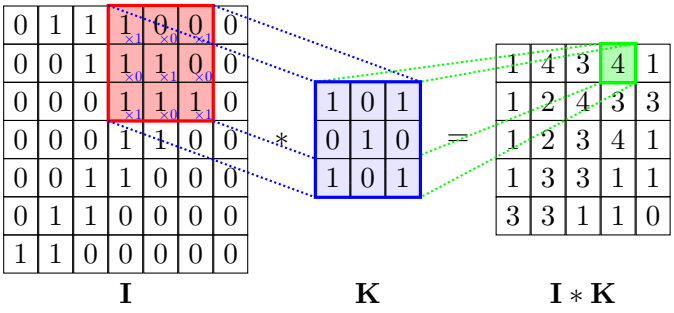


Figure 3.6: two-dimensional convolution operation
https://github.com/PetarV-/TikZ/tree/master

actually irrelevant which dimension we choose. Because we used no padding the output is a tensor with dimensions $[5 \times 5 \times 1]$. We can check the result by using equation 3.12. We have a stride $S = 1$, a receptive field $F = 3$, no padding $P = 0$ and input $W = 7$ which defines our output as follows $(7 - 3 + 2 * 0)/1 + 1 = 5$. Which is correct.

We now convolved one filter and produced one output or feature map. To processes colored images we need to extend the filter. Straight forward, we define a new input X with dimensions $[7, 7, 3]$ and adapt out weights F accordingly $[3, 3, 3]$. Using this extended dimensions, nothing would change in the output feature map!
The only thing what changes is, that we need to take the scalar product through all three dimensions. We can image figure 3.6 stacked three times, where each layer of the input $X[x, y, :]$ and the weights $W[x, y, :]$ have distinct values. We carry out the convolutions for each layer and obtain three output maps. The final result is the sum of all three feature maps. We can conclude that the depth of the input does not influence our calculation on the width and height of our output.
Even better, we can manually adjust the depth of the new convolutional layers. We just define a new set of weights F_2 with the same dimensions of our first one, namely $[3, 3, 3]$. We do the same calculations and obtain a new output. If we stack them up, we would end up with another 3-dimensional tensor for further processing. The amount of different filters we want to stack up is a hyperparameter and can be seen as the number of filters we want to apply to the input X. Interestingly, we consumed a 3D tensor and produced another one. Let's assume for a moment we used padding and the first two dimensions were the same as the ones from the input, e.g.[7,7,3] and [7,7,n]. Then the output would produce n different "images" of features, found in the original image.
To become a fully adequate layer for a neuronal network, we still need to add a non-linearity. Most commonly ReLu is applied element-wise to every output value. Convolutions have the appealing property that they are invariant to transformation in the input. This becomes intuitively clear when we recall that the weights for one feature map are shared. The feature map would detect an object irrelevant of its position.

A more in-depth introduction to Convolutional Neuronal Networks can be found in [29], [42], [9], whereas the theory is covered in [17].

3.1.6 Max Pooling Layers

Convolutions are tightly bound to max pool layers. As complex as convolutions might be, the pooling operation is quite simple. The idea behind pooling is that convolutions, despite their efficient, still produce a lot of unnecessary information. Figure 3.7 shows one simplified feature map.

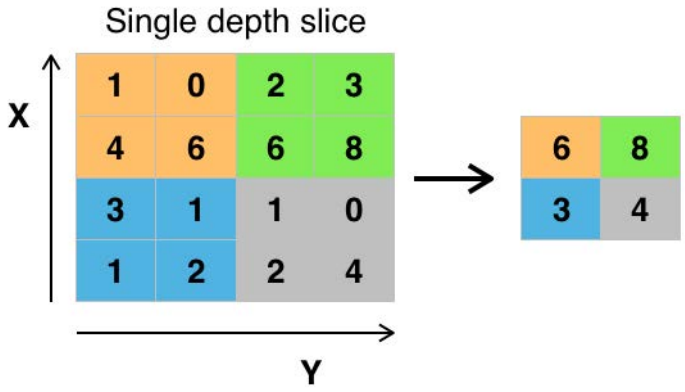


Figure 3.7: max pool operation visualized
https://de.wikipedia.org/wiki/Convolutional_Neural_Network#/media/File:Max_pooling.png

We assume that high values correspond to the detection of a certain feature. When one feature is highly present than the output in its vicinity is not of so much interest anymore and therefore discarded. During pooling the output from a convolutional layer is downsampled. By an example division into filter sizes of $[2 \times 2]$ and stride 2, the highest value is allowed through and forms the output. This procedure reduces the amount of information to a quarter.
The values for the filter size and the stride are common values and have turned out to be generally applicable. But nothing hinders us to choose different values. However, a filter size of more than 2 is probably too destructive.
Empirically, max pooling has shown good results by almost none computational cost. There is also average-pooling or L_2 norm pooling ([17]), which is mentioned here for completeness. Max pool layers have come in criticism over the years, because of their destructiveness on spatial relations among features. A discussion can be found in 7.3. Unfortunately, there is no new technique in sight which provides a practical solution to this problem.

3.2 Automatic Differentiation

We now have an understanding of DNN architectures, and we know that they are trainable with a method called gradient descent 2.7. The remaining question is how we obtain the gradient in an efficient way for a large number of parameters.
Automatic Differentiation (AD) is a research field which tries to provide access to the derivatives of (programmed) functions defined by computational algorithms. This makes AD different from numerical and symbolic differentiation in an interesting way.
In order to get a good understanding of the different differentiation approaches, we review them both.

Numeric Differentiation: Equation 3.13 shows the mathematical definition of derivatives, but can also be interpreted as an approximation when the limit is omitted. The method of finite differences, for example, builds directly on top of it [38]. This formula is straight forward to program, but evaluation suffers from truncation as well as round-off errors. Truncation errors arise from h not actually being zero. Round-off errors are the inaccuracy added from machine-precision [38].

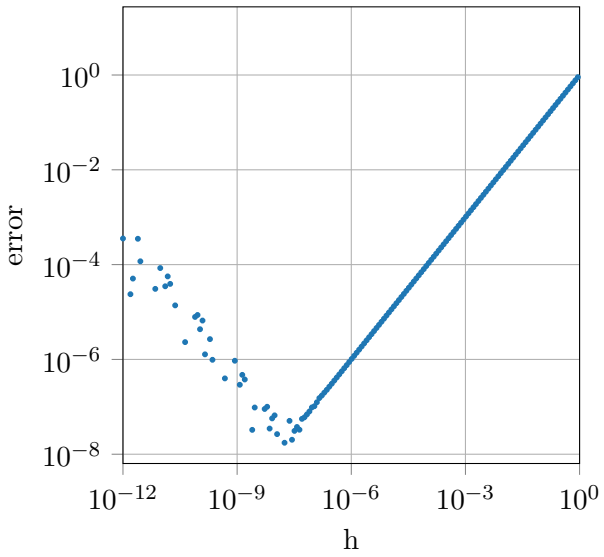


Figure 3.8: numerical error

$$\frac{d\,f(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.13}$$

Plot 3.8 shows the error for the evaluation of equation 3.13 of the function $f(x) = x^2$. The kink in the graph indicates the limit of accuracy we can reach with a 64 bit float number. As h becomes smaller than approximately 10^{-7}, the error increases. This behavior does not reflect

the actual math. A simple remedy to overcome this problem includes a slight reformulation of 3.13.

$$\frac{d\,f(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \tag{3.14}$$

More sophisticated remedies have been developed to alleviate round-off error related problems but at the core this property remains.

Symbolic Differentiation does not suffer from precision errors because it evaluates the derivative exactly. This technique comes close to the human handcrafted derivatives. Basically, the derivative is not approximated but actually calculated. The end result is in anyway correct. But usually, a much simpler and more efficient form of the derivative exists. Furthermore, this technique is not capable of deriving control flow operations like `if/else`.

Automatic Differentiation neither suffers from roundoff errors nor does it produce inefficient analytical expressions.
It is after all very memory intensive. The fundamental property of any sequential computer algorithm is that it can be unrolled into a graph. In other words, any arbitrary complex computer algorithm can be decomposed into a primitive mathematical expression. AD exploits exactly this structure instead of trying to work on its mathematical abstraction. It is important to note that this will not lead to a closed form of the gradient, but will give us only the gradient with respect to some input. Normally, our datasets.
AD is applied in two steps which we denote *forward pass* and *backward pass*. The forward pass traverses the graph in its initial direction from input to output. Every node in the graph denotes a certain operation as granular as summation or division. Indeed, it is common to group sequential parts of the graph into one *node*. Each node stores its own output value calculated from all input values defined by the graph. This doubles the amount of memory used for every node. The final node produces the actual output of the function.
Now, we can traverse backward through the graph and calculate the gradient with respect to every node by applying the chain rule. Again, the gradient is stored per node, increasing the memory demand even further.
The chain rule generalizes very well to higher dimensions and therefore tensors. Where the gradient of a matrix-vector multiplication is the backpropagated gradient times the respective Jacobian. The Jacobian is in most cases sparse or simply a multiple of the input vector, so efficient algorithms try to avoid storing it completely.
The backward pass is commonly known as the famous Backpropagation (BP) algorithm. As we can see in next chapter 3.3 the BP algorithm is only a piece – arguably an important one – in the whole filed of automatic differentiation which is not reflected in most literature. BP is the workhorse computing the gradient. All deep learning frameworks like Tensorflow, Theano, (Py)Torch, MS CNTK etc. abstract all the math away from the user, so in the end,

the forward and the backward pass is carried out automatically. This is a huge benefit and makes deep learning even accessible for users not familiar with multivariable calculus.

3.3 Backporpagation

Backpropagation is a method to obtain the gradient of an objective function. What is done with the gradient is in the domain of the optimization method. In other words, we could use gradient descent for training a DNN and compute the gradient a different way e.g. with finite differences.
Indeed, this was done in the early 60th. Results were obtained so slowly that NNs where abandoned from research. This was just not the right technique. Gradient descent is a basic method out of a group of first-order optimization techniques which require the gradient to optimize an objective function. Second order optimization methods, like the Newton method, require the Hessian matrix to optimize the objective function. Theses methods usually converge much faster, but are computationally more involving. For deep learning they are not of relevance outside of research.
BP is a method, proved highly efficient, to obtain the gradient. In other words, the optimization algorithm can be anything to your liking, like Stochastic Gradient Descent, Adam or RMSProp, as long as it depends on the gradient of the objective function.

In order to better understand Backpropagation (BP) we need be clear about its fundamental building blocks. First of all, we quickly recap the chain rule, one pivotal point [40].

Chain Rule

$$\frac{\partial}{\partial x}\left[f\left(u\right)\right] = \frac{\partial}{\partial u}\left[f\left(u\right)\right]\frac{\partial u}{\partial x} \tag{3.15}$$

If the derivate of a function with respect to the non-primary x variable is seeked, than this equals the derivative of the same function with respect to to its primary variable u times the derivative of the primary variable u with respect to to the non-primary variable x. This holds true for nested variables as well, which is the reason for the name: *chain rule*.

The Gradient

The word *gradient* is used very loosely in every day's conversations. The gradient has a well defined mathematical meaning, but people often refer more to its abstraction as a calculated

property (of the neuronal net), guiding the way to a global minimum (eventually). The exact meaning even differs from discipline to discipline. Strictly speaking, the gradient is a multi-variable mathematical operator which generalizes the derivative to higher dimensions. It is generally denoted with the ∇-operator. For three dimensions it can be written like this [40]:

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} \tag{3.16}$$

In most cases the *gradient* is referred to as the result of the application of the gradient-operator to a function. We will mostly stick to this loose definition, because, as we will see, it is not the intention of BP to give a closed form for the gradient nor write out every partial derivate for all parameters.
The gradient's outcome is a vector-field which defines a vector for each point of the function. Each vector is perpendicular to the nouveau-lines and points in the direction with the highest descent of the function. The gradient vector's magnitude indicates the slop of the function. The gradient is often associated with $\mathbb{R}^3$ and Cartesian coordinates mainly due to problems given in 3D space. Nevertheless, the gradient generalizes well with higher dimension where, for instance, the gradient of the velocity field for fluid flow is the acceleration tensor in $\mathbb{R}^4$. What makes intuitively sense as the acceleration a is the derivative of the velocity v, $\dot{v} = a$.

Let's focus on the gradient in the context of machine learning.

$$f(x) = w_1 x^2 + w_2 x + w_3, \qquad f'(x) = 2 \cdot w_1 x + w_2 \tag{3.17}$$

Equation 3.17 is a simple parabola and its derivative. It might seem cumbersome but it is useful to define the equation for each node in order to apply the chain rule. Figure 3.9 shows the computational graph for this function. Every node is denoted with a unique name and signals witch math operation is carried out with its inputs.

$$f = w_3 + a \qquad \begin{matrix} a = c + b & c = w_1 * d \\ b = w_2 * x & d = x * x \end{matrix} \tag{3.18}$$

We try to achieve the same derivate, by applying the chain rule, as we get in 3.17. The results for the derivatives with respect to all variables are given in 3.19

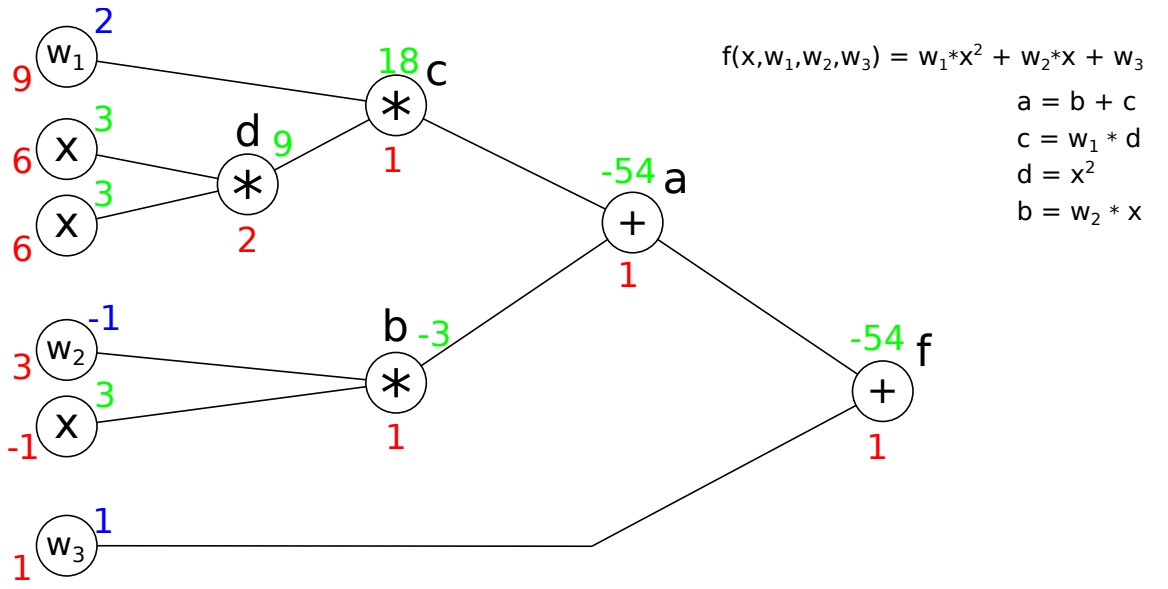


Figure 3.9: Computational graph of equation 3.17

$$\begin{aligned}
\frac{\partial f}{\partial f} &= \frac{\partial f}{\partial a} = 1 \\
\frac{\partial f}{\partial b} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial b} = 1 \\
\frac{\partial f}{\partial c} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial c} = 1 \\
\frac{\partial f}{\partial d} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial c}\frac{\partial c}{\partial d} = w_1
\end{aligned}
\qquad
\begin{aligned}
\frac{\partial f}{\partial w_1} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial c}\frac{\partial c}{\partial w_1} = d \\
\frac{\partial f}{\partial w_2} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial b}\frac{\partial b}{\partial w_2} = x \\
\frac{\partial f}{\partial w_3} &= 1
\end{aligned}
\tag{3.19}$$

and the most interesting one:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\left(\frac{\partial a}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial a}{\partial c}\frac{\partial c}{\partial d}\frac{\partial d}{\partial x}\right) = 1 \cdot (1 \cdot w_2 + 1 \cdot w_1 \cdot 2x) = 2 \cdot w_1 x + w_2 \tag{3.20}$$

This experiment shows that the calculation for the derivative wrt. x works. Admittedly, this seems unnecessarily sophisticated, and we do not want to store the gradient that way.
Let's focus on the numbers in figure 3.9. The input is $x = 3$ depicted in green. The weights are $\boldsymbol{w} = [2, -1, 1]$ in blue. We carry out the forward pass by calculating the intermediate results shown in green as well. The forward pass ends when the function is evaluated completely.
We start backpropagation by evaluating the first derivative $\frac{\partial f}{\partial f}$ which is obviously 1 denoted in red. With the help of the equations in 3.19, we can propagate the error back to x. In the last layers we can see, that the input x has different outputs. To obtain the general gradient, we need to sum them up. The result shows the overall influence of x on f. We can now compare equation 3.20 with the gradient we obtained with backpropagation. And come to

the same result.

$$\frac{\partial f(x, \boldsymbol{w})}{\partial x} = 2 \cdot w_1 x + w_2 = 2 \cdot 2 \cdot 3 - 1 = 11$$
$$\frac{\partial f(x, \boldsymbol{w})}{\partial x} = 6 + 6 - 1 = 11 \tag{3.21}$$

Incidentally, we have also calculated the gradient wrt. the weights. This gradient actually comes for free. And as we fixed x we can now start gradient descent. The gradient on the weights reads $\nabla_w f = [9, 3, 1]$. We recall the gradient descent equation 2.12.

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \tag{3.22}$$

To obtain $\boldsymbol{\theta}_{i+1}$, which are the weights in our case, we need to subtract the gradient from current weights.

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla_{\boldsymbol{w}} f(\boldsymbol{x}; \boldsymbol{w}) = \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} - 1 \cdot \begin{pmatrix} 9 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} -7 \\ -4 \\ 0 \end{pmatrix} \tag{3.23}$$

For $\eta = 1$ we get our new weights. The update process is oversimplified here, and many more sophisticated update techniques were developed.
It is important to note that the gradient in backpropagation is a completely local phenomenon per node. It is determined only by the node's input and output during forward pass and the backpropagated error. In the example from figure 3.9 there was actually no need to use the value of the forward pass in the calculations. But if the graph had more complex nodes, their output would be necessary, too.
This procedure scales to higher dimensions without problems. For detailed information the reader is referred to [17] and [29].

3.4 Entropy vs. Maximum Likelihood

Entropy is a measure of information a source emits and how it is encoded on a limited channel. Information can be measured in bits, nats or whole words, depending on the base of the logarithm used in equation 3.24. In deep learning, its usual to use the natural logarithm with nats, whereas bits, with the logarithm of base 2, are used in information theory [17].
For further derivation, it is not utterly important to always refer to "information" in the context of loss functions. But it gives the loss one way of interpretation and a unit.
The entropy of a discrete distribution y is given by the following equation:

$$H(y) = \sum_i y_i \log(\frac{1}{y_i}) = -\sum_i y_i \log(y_i) \tag{3.24}$$

The minus in 3.24 does intuitively fit as the logarithm for values between $[0, 1]$ is negative. Thus gives a positive entropy. Let's assume we define the random variable X to be the number of students for each course of study at the faculty "Bau Geo Umwelt"(BGU)

$$X = \{CivilEnginieering, Geodesie, \\ Enginieering, Geology)\} \tag{3.25}$$

and a probability distribution over the random variable X:

$$y = P(X) = \{0.47, 0.06, 0.31, 0.16\}. \tag{3.26}$$

The goal now is to represent the four courses in a manner which reduces the amount of bits used to send it over a channel. We imagine a bit sequence for each course, naively we could define it like so:

$$\begin{aligned} Civil\ Engineering &= \{1,0,0,0\} \\ Env.\ Engineering &= \{0,1,0,0\} \\ Geodesy &= \{0,0,1,0\} \\ Geology &= \{0,0,0,1\} \end{aligned} \qquad \begin{aligned} Civil\ Engineering &= \{1,1\} \\ Env.\ Engineering &= \{1,0\} \\ Geodesy &= \{0,1\} \\ Geology &= \{0,0\} \end{aligned}$$

The left encoding is known as one-hot encoding, commonly used as labels for ML tasks. These are definitely not the most saving encoding we can come up with, although the suggestion on the right is much better than on the left. The entropy of distribution y is given by,

$$H(y) = -(0.47 * \log_2(0.47) + 0.06 * \log_2(0.06) + \\ 0.31 * \log_a(0.31) + 0.16 * \log_2(0.16)) = 1.72\ [bit] \tag{3.27}$$

The result can be interpreted such that we would need 1,72 bit on average to encode y, to obtain the most saving encoding. One, out of a view, optimal encodings would be the following:

$$\begin{aligned} Civil\ Engineering &= \{1\} \\ Env.\ Engineering &= \{0\} \\ Geodesy &= \{0,1\} \\ Geology &= \{0,0\} \end{aligned}$$

Because, Civil and Environmental Engineering are much more frequent it is better to encode it with viewer bits to save bandwidth on the channel.

We can also make an image our information source by letting it emit pixels one after another. Noise has high entropy, whereas the entropy of the tree's image is only high where the tree is apparent. We can see that pure information does not instantly mean that the image shows something recognizable. Less information is often better than too much.

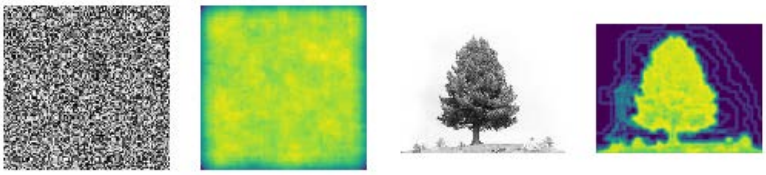


Figure 3.10: the entropy of noise and a tree
image from www.scipy.org

3.4.1 Cross Entropy

Admittedly, we are not interested in encodings, but a measure for how good our prediction is. We introduce cross entropy as a measure of how good an arbitrary distribution $\hat{y}$ is able to approximate y.

$$H(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_i y_i \log(\frac{1}{\hat{y}_i}) = -\sum_i y_i \log(\hat{y}_i) \tag{3.28}$$

In our case we could assume, that the students are spread equally over all courses, which gives us a uniform distribution of $\hat{y}_i = 0.25$ for every course. Doing the math yields a cross entropy of

$$H(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_i y_i \log_2(\hat{y}_i) = 2.0 \ [bit] \tag{3.29}$$

Subtracting the entropy of the target distribution and our approximation, gives us an additional "fee" of our transmission of $H(y, \hat{y} - H(y)) = 0.298 \ [bits]$. The fee comes from the fact that we use $\hat{y}$ instead of y for our encoding. This difference is called the Kullback-Leibler divergence (KLD) [17].

$$KL(\boldsymbol{y}||\hat{\boldsymbol{y}}) = H(\boldsymbol{y}, \hat{\boldsymbol{y}}) - H(\boldsymbol{y}) = \sum_{i=1}^{n} y_i \log(\frac{y_i}{\hat{y}_i}) \tag{3.30}$$

If $\boldsymbol{y} = \hat{\boldsymbol{y}}$ the KLD would be zero and the approximation would be the original distribution. This starts to look like a "loss" for NNs because the KLD indicates how much our approximation is apart from our target distribution.
What happens when we would use KLD as our loss function.

$$\nabla_{\hat{\boldsymbol{y}}} KL(\boldsymbol{y}|\hat{\boldsymbol{y}}) = \nabla_{\hat{\boldsymbol{y}}} H(\boldsymbol{y}, \hat{\boldsymbol{y}}) - \nabla_{\hat{\boldsymbol{y}}} H(\boldsymbol{y}) = \nabla_{\hat{\boldsymbol{y}}} H(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{3.31}$$

We can clearly see, that minimizing the KLD with respect to our prediction is the same as minimizing the cross entropy, because the entropy of our target distribution does not depend on $\hat{\boldsymbol{y}}$.
So, we are left with the cross entropy as our loss function for the m-th training example,

$$Loss_m = -\sum_{i=1}^{n} y_i log(\hat{y}_i) \tag{3.32}$$

By summing over all training examples, we end up with our cross entropy as a suitable cost function.

$$Cost = -\sum_{m=1}^{k}\sum_{i=1}^{n} y_i^m log(\hat{y}_i^m) \tag{3.33}$$

It is important to notice, that looking at our loss, we always refer to one training example. Whereas, when we specified our cost we took a step back and took all training examples into account. We leave the cost function as it is for now, but will return to it at the end of the chapter.

3.4.2 Maximum Likelihood Estimation

What if we do want to directly predict a model $\hat{y}(\boldsymbol{x};\boldsymbol{\theta})$. Our model is a NN with $\boldsymbol{\theta}$ as its parameters and $\boldsymbol{x} \in \boldsymbol{X}$ are the training examples. We can ask the question:
"Given a training set $\boldsymbol{X}$, how do the model's parameters look like, that the likelihood of the model predicting the data is maximized".
We investigate a slightly more complex example than 3.26. Namely, the average age of students at the faculty BGU 3.11.
The samples are drawn from a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with $\mu = 23$ and $\sigma = 3$[2].
The Maximum Likelihood Estimation (MLE) gives us a way to compute the target distribution under the assumption that the data is independent and identically distributed (i.i.d.).
We can state the likelihood function

$$L(\boldsymbol{\theta}|x_1, x_2, x_3, , \ldots, x_n) = f(x_1, x_2, x_3, \ldots, x_n|\boldsymbol{\theta}) = \prod_{i=1}^{n} f(x_i|\boldsymbol{\theta}) \tag{3.34}$$

The "|" divides the argument into variables and parameters. $L(\boldsymbol{\theta}|x_1, x_2, x_3, \ldots, x_n)$ indicates that L is dependent on $\boldsymbol{\theta}$ and the parameters x_i are fixed. Through the multiplication, the argument's variables and parameters are switched.
By now, equation 3.34 is just a way to make $\boldsymbol{\theta}$ our variable. We now seek a $\boldsymbol{\theta}$ which maximizes our likelihood function because at this point the distribution estimates our samples $x_1, x_2, x_3, \ldots, x_m$ best. Straight forward, we formulate the product of 3.34, with f being the

[2] these values are assumptions

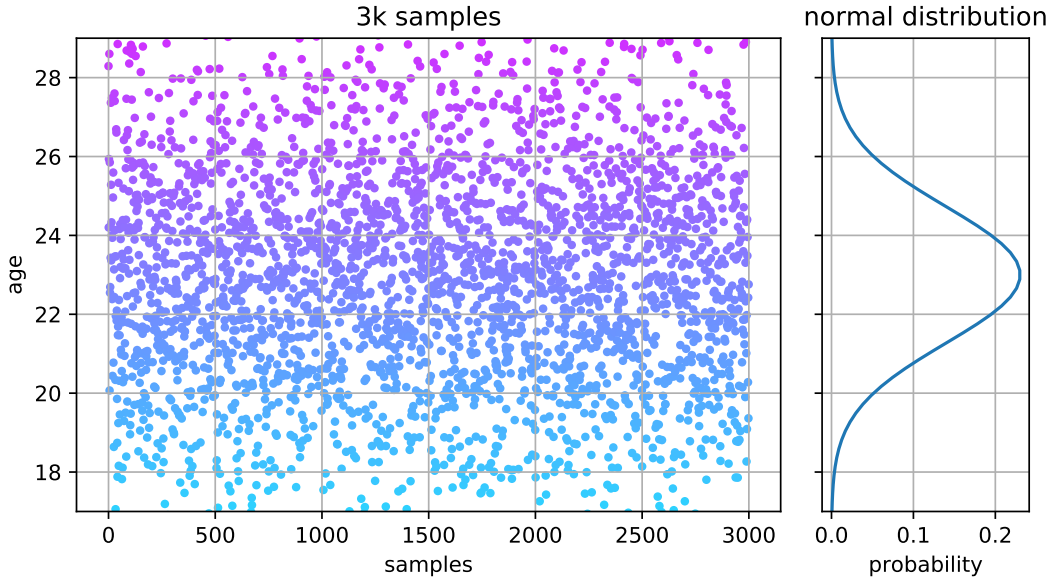


Figure 3.11: age samples of students from BGU

normal distribution

$$f = \mathcal{N}(x|\mu,\, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{3.35}$$

$$\begin{aligned} \prod_{i=1}^{n} f(x_i|\boldsymbol{\theta}) = \mathcal{N}(x_1|\mu,\, \sigma^2) \times \mathcal{N}(x_2|\mu,\, \sigma^2) \times \\ \mathcal{N}(x_3|\mu,\, \sigma^2) \times \cdots \times \mathcal{N}(x_n|\mu,\, \sigma^2) \end{aligned} \tag{3.36}$$

The keen reader might noticed that using the normal distribution as our f is not a NN. That's because derivation with only two variables than view hundred is simpler and as we initially sampled our data points from a normal distribution, chances are high that we can recover out initial μ and σ. Of course, f can be a different model.
To find the desired values for μ, σ, we need to take the respective derivatives.
From calculus we know that deriving a product is much more cumbersome than a sum. Therefore, we apply a small trick and define the log-likelihood as our function, we want to maximize. We are allowed to do this because the logarithm is strictly monotonic and does only scale the likelihood function. The maximizing parameters $\boldsymbol{\theta}$ stay the same. In turn the gain is significant as, we can now write the product as a sum,

$$\log(L) = \log(\prod_{i=1}^{n} f(x_i|\boldsymbol{\theta})) = \sum_{i} \log(\mathcal{N}(x_i|\mu,\, \sigma^2)) \tag{3.37}$$

Using the definition of 3.35 equation 3.37 can be written as

$$\begin{aligned}\ln(L) &= \sum_{i=1}^{n}\left[-\frac{(x_i-\mu)^2}{2\sigma^2} + \ln\left(\frac{1}{\sigma\sqrt{2\pi}}\right)\right] \\ &= -n\ln(\sigma) - \frac{n}{2}\ln(2\pi) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2\end{aligned} \tag{3.38}$$

As we are free in our choice of the logarithm's base, we choose the natural logarithm because it cancels nicely with the exponential function in the normal distribution.

$$\begin{aligned}\frac{\partial \ln(L)}{\partial \mu} &= -\frac{1}{2\sigma^2}\sum_{i=1}^{n}2(x_i-\mu) = 0 \qquad \Rightarrow \quad \mu = \frac{1}{n}\sum_{i=1}^{n}x_i \\ \frac{\partial \ln(L)}{\partial \sigma} &= -n\frac{1}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{n}(x_i-\mu)^2 = 0 \qquad \Rightarrow \quad \sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n}}\end{aligned} \tag{3.39}$$

Figure 3.12 shows how the error decreases as the number of samples increases. This property hold independent of the model used [17]. Of course, the model must be able to represent the data. In other words, using more samples for the MLE always reduces the error of the parameters.

A small error on the training set is not per se what we want. As NN are generally over parameterized, they tend to overfit the training data, which, in turn, leads to a worse generalization, see 2. In this section we have used $f = \mathcal{N}(x|\mu, \sigma^2)$ as out model. Without

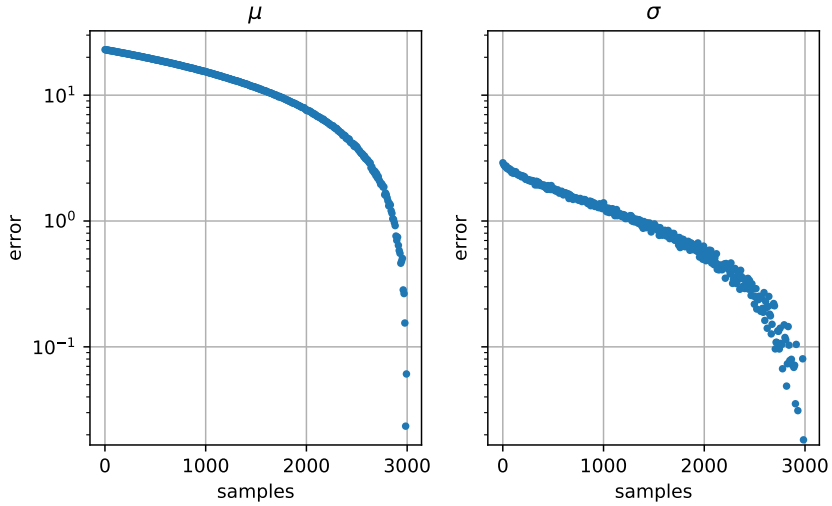


Figure 3.12: parameter error

problems, we could also define f to be the Poisson distribution or MSE. Indeed, using the normal distribution in the MLE is the same as minimizing the MSE, we just predict the mean and guess a sigma. This is derived in [17].

But what really is of interest for us is using the MLE as a loss function. We recall equation 3.37. But for simply inserting our prediction vector we do not obtain a solution directly. Because the class of the input $\boldsymbol{x}$ is in range $[1, 2, 3, \ldots, k]$, we would need to figure out the correct label at backpropagation time. We can do better by carefully examining the MLE for the NN case.

$$\boldsymbol{x} \in \boldsymbol{X} \quad \hat{\boldsymbol{y}}(\boldsymbol{x}|\boldsymbol{\theta}) \quad \boldsymbol{y} \qquad L_m = \sum_{i=1}^{n} log(\hat{y}_i^m) \tag{3.40}$$

Equation 3.40 shows the log likelihood function for the m-th training example $\boldsymbol{x}$ in $\boldsymbol{X}$. The sum over all entries of the discrete distribution's vector – our model's predictions $\hat{y}_i$ – gives us the likelihood of the example $\boldsymbol{x}$. We know that our labels are mutually exclusive, and thus form a one-hot encoded vector. This encoding represents four classes with the '1' indicating the respective class. Because the values lie in the range from $[0, 1]$ we can also interpret this as a distribution over all classes where the mass density sits on the correct class. Thus, '0' indicates 0% of the input being the class and 1 indicates 100%.
Equation 3.41 shows an example label and the corresponding predictions.

$$\boldsymbol{y}^m = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \qquad \hat{\boldsymbol{y}}^m = \begin{pmatrix} 0.2 \\ 0.4 \\ 0.98 \\ 0.1 \end{pmatrix} \qquad \hat{y}^m_{\{\boldsymbol{y}=1\}} = \hat{y}_i^m = 0.98 \tag{3.41}$$

The only prediction of interest is the one where the label's vector entry is 1. With this knowledge, we can define the cost function with respect to the whole training set as

$$Cost = \sum_{m=1}^{M} log(\hat{y}^m_{\{\boldsymbol{y}=1\}}) = \sum_{m=1}^{M} log(\hat{y}_i^m) \tag{3.42}$$

Please note that we do not need to sum over the prediction's vector anymore. Each example produces exactly one probability $\hat{y}_i$ which is of interest for us.

3.4.3 The Link

We revisit the cross entropy cost from equation 3.43.

$$Cost = -\sum_{m=1}^{M} \sum_{i=1}^{n} y_i^m log(\hat{y}_i^m) \tag{3.43}$$

The assumption of our one-hot encoded label still hold, therefore we are can see that all entries except one for the inner sum become 0. This again leaves us with almost the same

cost as derived from the MLE. Using the numbers from 3.41 clarifies this statement.

$$Cost = -\sum_{m=1}^{M} \begin{pmatrix} y_1^m log(\hat{y}_1^m) \\ y_2^m log(\hat{y}_2^m) \\ y_3^m log(\hat{y}_3^m) \\ y_4^m log(\hat{y}_4^m) \end{pmatrix} = -\sum_{m=1}^{M} \begin{pmatrix} 0\ log(0.2) \\ 0\ log(0.4) \\ 1\ log(0.98) \\ 0\ log(0.1) \end{pmatrix} = -\sum_{m=1}^{M} log(\hat{y}_i^m) \tag{3.44}$$

We obtain the negative MLE as a cost function. In optimization, problems usually stated as a minimization of the objective function. It is more a matter of preference instead of having mathematical advantages, because there is no difference between maximizing a function f or minimizing its corresponding negative function $-f$. Therefore, we stick with the negative cost function and state:

Minimizing the KLD between two distributions is the same as minimizing the cross-entropy between them and also the same as minimizing the negative-log-likelihood of the model. The KLD is a distance measure, so apart from cross entropy and likelihood, we can also think of decreasing the "distance" between the predictions and the labels.

In literature, the cost or loss, for the most part, is denoted as the cross entropy. This may be because of its straight forward formulations including the labels as $\boldsymbol{y}$ by default. The cross-entropy does not limit us in using one-hot encoded labels but also allows soft labels in a sense that the probabilities of the input do not need to be mutually exclusive. In other words, we are allowed to use a discrete distribution as a label where the entries are not restricted to 0 or 1.

Deep learning implementations e.g TensorFlow or PyTorch, support this difference through extra layers which are called `softmax_cross_entropy_with_logits()` or `sparse_softmax_cross_entropy_with_logits()` [18], [15]. Logits refer to the unscaled output of the last layer, similar to the z variable defined in 3.1.1.

In case of one-hot encoded vectors, we run into a slight problem. Imagine $C = \{0, 1, 0, 0\}$ as the target distribution. If $\hat{y}_i$ reaches zero for the target probability equal to 1, we would calculate log(0) which is minus infinity. Thus, reaching the target probability is not even possible.

Nevertheless, if the prediction manages to come close to the target this suffices anyway, because we are only interested in the maximum value. For example $max(\hat{\boldsymbol{y}}) = \{0.3, 0.98, 0.04, 0.02\} = \hat{y_2}$, does not depend on $\hat{y_1}$ being 0.3 or 0.5. Deep learning implementations take care of the argument of the logarithm not being zero.

3.4.4 Softmax-Loss

In section 3.1.3 we introduced the softmax function as a way to normalize each entry of a vector. With the findings in section 3.4.1 we can now deduce a combination of the cross-

entropy and the softmax. We recall both equations.

$$Cost = -\sum_{m=1}^{M} log(\hat{y}_i^m) \qquad s(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \tag{3.45}$$

As we have seen in3.4.3, only one entry is of interest. Thus, the softmax reduces to one equation and we can plug it in our cost function.

$$Cost = -\sum_{m=1}^{M} \ln\left(\frac{e^{\hat{y}_i^m}}{\sum_{j=1}^{k} e^{\hat{y}_j^m}}\right) = -\sum_{m=1}^{M}\left[\hat{y}_i^m - \ln\left(\sum_{j=1}^{k} e^{\hat{y}_j^m}\right)\right] \tag{3.46}$$

Equation 3.46 benefits the design process of neuronal networks because the intermediate step, explicitly defining a softmax for the output scores, is abstracted away. Additionally, the implementation handles corner cases much more reliably and efficiently. Equation 3.46 also implies a subtle fact. Cross-entropy does only implicitly penalize other than the desired prediction. Its main concern is to increase the target probability $\hat{y}_i$.

3.5 Inference

The NN has been trained successfully and the test metric has reached a satisfying level. Now, the time has come, to use the network in a production environment. This is generally called inference.

When we compare the training state of a NN with the inference state we need to consider *latency* and *throughput*. It is clear that training requires a high throughput in order to be efficient and latency is of minor importance [12]. In real-life applications, like image classification at Google, Facebook providing image captions for blind people [60] or in self-driving vehicles, latency is the decisive factor.

In the training state, the network requires to store the gradient for every tensor in each layer. But when the net is inferred the gradient is obviously not needed anymore as we are not going to alter the parameters of the NN. Thus, BP is omitted completely. BP heavily relies on a fine-grained key of the network's graph in order to apply the chain rule correctly. The absence of BP, in turn, allows us now to generously compress the network's layers and parallelize them on the GPU[3]. This decreases the latency and reduces the memory consumption approximately by half. Further analysis of the graph and its parameters can detect parts which are unlikely to influence the result due to their low activations. This parts can safely be *pruned* reducing the size of the network in total [19].

[3] assuming that almost all serious applications run on GPUs nowadays.

3.6 Miscellaneous

Computational power is by far the most important aspect when training a neuronal network. Unfortunately, even the fastest CPU is not built for massive parallel code. Luckily graphics card have been around doing exactly this. Let's have a look at how graphics cards work on an abstract basis, leaving out the technical details and hardware implementations.
Every Pixel's value is computed by the same instructions similarly a construction pipeline. Assuming the CPU wants to display a space shuttle on the screen, this data is only available in binary in the RAM of the computer. The space shuttle is composed out of different geometry primitives. Each of which has its own local coordinate system. The graphics card first demanding task is calculating the transformation matrices for each object, with respect to the world coordinate system, the space shuttle exists in. Immediately thereafter it renders the geometry consisting out of triangles. The 3D-Geometry is than projected into the perspective of the viewer in the 2D-plane. All triangles which are not part of the scene are clipped e.g. excluded from the pipeline. This process is hardwired in graphics cards due to the enormous speed benefit and the relative simplicity. A, so called, shader adds the color or texture to each remaining triangle. The end result is then sent to the monitor.
These steps above describe the necessary tasks to convert abstract binary data to *one* pixel's value. Because these instructions don't differ for all pixels this pipeline can be applied to all of them at *once*.
This oversimplification does, of course, not give justice to the complex topic of computer graphics but it is sufficient to understand its relevance in deep learning. In comparison, a CPU is deliberately designed to carry out different complex tasks in parallel but only on a view cores. To be precise the tasks are called processes and their jobs can range from a simple counter to a resource-intensive sorting procedure.
How can we now exploit the parallel capabilities a graphics card provides in terms of training a deep neuronal network?
When we recall the formula of the loss function from 2.14 we can observe that we need to average the loss over the whole minibatch. If we would run the training sequentially the program would consume a training example one after the other. It would store the loss for every training pair and sum over it in the end, providing the loss for that mini-batch. Like in the case of pixels, the network's graph definition stays the same for every training example. Furthermore, the computation for each example is *independent* from each other, meaning there is no circumstance where one process (training example) requires data from another.
In practice, it is advisable to fit the size of a mini-bach to the memory of the underlying hardware.

Chapter 4

PointNet

Geometry and computation are very tied subjects. Since human eyesight is the best way to get data into the brain, it does not surprise that we use monitors to interact with computers. Computer vision is the collection of methods to make images perceptible for machines and data visible for humans. Many methods have real-time constraints, which means a certain computation has to be finished after a fixed amount of time. Modern monitors have a refresh rate of 60 Hz, which constrains the computation time for one frame to roughly 1.6 ms. If this constraint is violated by a tiny margin, humans percept it as a lag spike. To meet these constraints, graphics cards had been developed.
Not only hardware is necessary for flawless computer graphics, but data structures also play an important role. They define the possibilities, what is representable.

4.1 Mesh vs. Point

In order to visualize an object on a computer screen, the object must be defined. At the lowest level, 3D points alone are used to define the discrete position of an object's surface. We call this representation a *point cloud*.
Point clouds, in general, are three-dimensional collections of points in the Euclidean space. Each point is described through its coordinates. Normally, Cartesian coordinates are chosen but through transformation, points can be mapped to arbitrary coordinate systems as long as the transformation preserves the metric. This is done when large landscape scans are populated into regional maps. The metric in Euclidean space is simply the distance between one point and the origin and evaluated by the L^2 Norm. Points are related to each other by distance.
The simple description of point clouds makes it possible to automatically retrieve them from the real world. The downside is the massive amount of data generated in the range of gigabytes. Depending on their generation, they might be scaled incorrectly or other objects

occlude parts which leads to sparse regions. Point clouds inherently lack the information about geometrical membership. There is no incorporated information which identifies points belonging to the same object. Even further, no way exists to unambiguously detect surfaces and their orientation in space. With the absence of surfaces, there is no chance to make statements about volume or use point clouds directly in simulations. Point clouds can carry additional information per point, either from measurement or from calculations, like color or normal vectors. This again increases the overall amount of data needed to store the point cloud.

An intuitive idea is to connect neighboring points with each other, such that they form triangles. This allows for fewer points in regions where the object is flat, as the triangles comply with them well enough. Regions where high curvature is present, more triangles are needed. Incidentally, this also defines a volume if the mesh is closed. And finally, normal vectors are defined through the triangle. The famous but simple STL format defines a mesh this way by storing three points for each triangle and the normal vector. The application which processes STL files is responsible to check if there are gaps between triangles or if triangles intersect. Originally developed for 3D printing, STL files do not meet the advanced requirements for today's graphics.

STL stores the vertices for each triangle separately although triangles normally share their nodes with neighboring ones. The Boundary Representation (B-rep) model overcomes this drawback of STL files but makes the geometry representation more complex. In B-rep models, every node is stored only once. Through a graph based relation, the nodes are connected to lines, which are connected to surfaces which form volumes. This topology removes the redundant nodes and opens up for complex descriptions of volumes and even holes.

All mesh representations carry more information than a point cloud. But, in the end, meshes are completely arbitrary data structures which try to describe the 3D reality efficiently. They have no inherent relation to reality, which makes it hard to retrieve them from a real scene. Therefore, we try to utilize the simplest 3D representation, a point cloud, to track the progress of the construction sites in reality and accept their drawbacks.

4.1.1 3D Scanning and Reconstruction

There are active and passive methods to obtain point clouds. Active methods emit some measurable quantity whereas passive methods rely on ambient light or moving sources. We will not cover passive methods, as they do not have any importance for progress track problems. Active methods can roughly be divided in **time-of-flight** methods and **triangulation** methods.
Time-of-flight methods are based on the relation $d = c*t$ where c is the speed of light, t is time and d is the distance. They send out a laser pulse which is reflected from an object's surface.

The duration between the dispatch of the pulse and the arrival of its reflection is measured. These methods can operate over a large distance. The more accurate the measurement of the duration is, the more accurate is the distance. This is also the downside of time-of-flight methods, as their accuracy range in millimeters. Light detection and Ranging (Lidar) is based on this technique and gained popularity as it is used for autonomous vehicles [54].
Triangulation techniques exhibit the opposite trait as they are not suitable for long-range measurements, but produce very accurate results in short-range. They are based on a constant laser point on the target surface. A sensor recognizes the position of the point's reflection. The laser emitter, the sensor lens, and the target point form an imaginary triangle. The angle of the laser emitter and sensor, as well as their distance, is known. The angle of the reflection hitting the sensor lens is proportional to the distance of the target surface. Out of this relation the distance can be calculated.
The photogrammetric approach tries to derive point clouds from 2D images of the same scene. This is done through feature tracking throughout all the images. Because this feature detector is quite robust to scale, rotation and illumination it accurately finds the features even when the image quality is low. Out of the correlation of the same features in different images, a distance is derived. Unfortunately, this distance is not absolute and must be scaled manually. Photogrammetric point clouds have color information by design as pictures are not gray-scale anymore. In the context of progress tracking, this is an advantage.

4.2 Boundary Conditions

In contrast to image recognition, feature recognition tasks in point clouds deal with some substantially different problems. Point clouds are scattered unordered datasets. In images, the spatial relation of pixels is fixed. A CNN does exactly exploit this property, see 3.1.5. Point clouds do still represent the same information even when the points are permuted. This is an important realization.
Points form subsets in space which build up important features a NN needs to capture. Features like edges or skewed areas appear locally and might not have a large spatial extent. These features add up and describe the whole object, which, in turn, might span over half the construction site. Therefore, features must be matched locally, but the NN needs to be able to learn that some features form objects belong together over a large distance. To conclude the NN must

- be invariant to !N permutations
- heavily rely on the distance between points
- be insensitive to affine transformations of the input point set

An interesting fact is described in [46], where the authors initially trained their net with normalized point clouds and obtained good results. This is, of course, a meaningful preprocessing, but violates the restriction that the net's prediction must not be influenced by absolute values. Loosely speaking, the net learned the simple fact that points with z-value near zero most likely describe the floor of a room. Admittedly, it is quite challenging to find good data augmentation, to break the dependency to absolute values.
As briefly explained in 1.3.2 often a volumetric approach is chosen. Voxels structure space the way pixels structure images. These structures can be equally exploited by CNNs. This approach is problematic as voxels do only indicate that one or more points are present inside. The exact information of the arrangement of these points is lost. This means we lose information on the lowest, yet most important level, but gain a data structure which is ready to be processed by a CNN. We choose PointNet mainly because its architecture makes it possible to consume point clouds directly, and therefore preserves local features.

4.3 Spatial Transformation Networks

PointNet makes use of Spatial Transformer Network (STN)s. They were originally developed by Google to provide an attention mechanism to label house numbers on street view images. Apparently, they can also be used for point clouds. To better understand the workflow of PointNet, we quickly discuss them in the general context and see later how they are applied to point clouds.
A spatial transformer network STN, see details in [25], enables attention mechanisms for larger neuronal networks. It alters the input image in a way that only parts with most interesting information will be passed to the next layers. Thus, it derives an (affine) transformation matrix out of the input image, which is, in turn, applied to the same input leaving it augmented. The augmented data will be further processed by the later layers without any additional effort. The location in the network's graph, where the STN is applied, is not restricted to the net's input layer and can be applied even sequentially.
Although the size of the transformation matrix, e.g. the number of parameters, can be chosen arbitrary, it is usually set to a meaningful number. Originally developed for a CNN, the authors choose a matrix of size $\mathbb{R}^2 \times \mathbb{R}^3$, which makes six parameters θ total. These parameters are learned in the same training cycles the net would undergo anyway and due to its size not much overhead is added.
Figure 4.1 shows the architecture of a STN. The localization net can be any kind of neuronal network which has a regression layer at the end and a continuous activation for backpropagation. *Regression layer* in this context simply describes a network type where the output values are not squashed into a function, e.g. softmax, but used unaltered as the parameter values. The grid generator and sampler are constructs, often used in computer graphics for texture maps. This is a further advantage as the computation is already carried out on

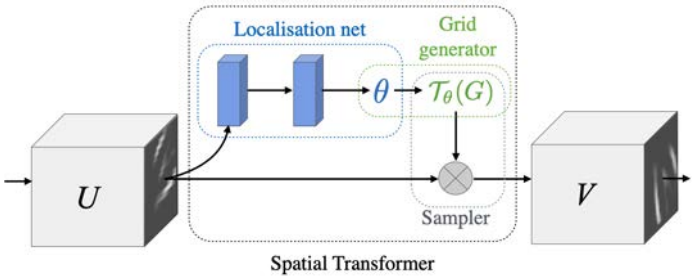


Figure 4.1: Spatial Transformer Network

graphics cards.
It be must be emphasized that the STN is responsible for picking the right transformation, as only the transformed part of the slice is further processed. PointNet's authors, therefore, regularize the loss function with a term which makes the transformation matrix near orthogonal. Further, as seen in figure 4.1, the computational graph is split into two parallel paths. Nevertheless, it still classifies as a feedforward network because there are still no loops present. The matrix multiplication of the STN's output and the input image, is modeled as a normal operation, where the gradient is split accordingly. One part of the gradient is backpropagated into the STN, whereas the other skips it. Finally, they joined together at position U in figure 4.1.

4.4 PointNet Architecture

In order to understand PointNet, we give an overview of all building blocks and explain them in detail afterward. See [46] and [45] for comprehensive information.

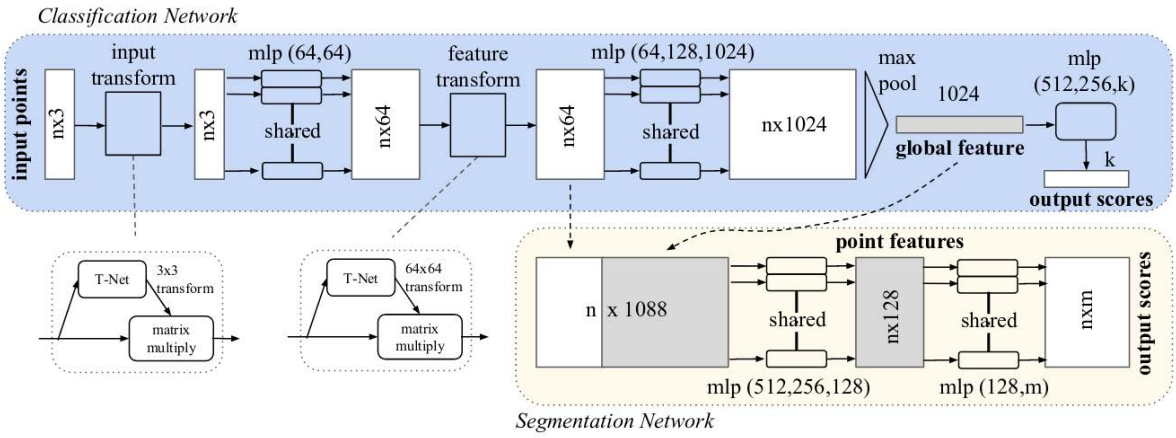


Figure 4.2: PointNet architecture
taken from [46]

Figure 4.2 shows the architecture of PointNet[1]. The blue area marks the classification part whereas the yellow part refers to segmentation. We discuss all tensor dimensions without the additional batch dimension for clarity. Furthermore, only the key dimensions of certain layers are mentioned, as these numbers are pretty common (power of 2), and do not contribute to a better understanding of the concept. They are mentioned in the 4.2.

PointNet consists out of spatial transformers, shared MPLs, one max pool layer and fully connected layers. Right in the beginning the input point cloud of form $[N \times 3]$ is split into two computational graphs. One is passed through the transformer network one is unaltered. PointNet uses STNs for the input layer and between the max pool layers. Evidently, the implementation uses a $\mathbb{R}^3 \times \mathbb{R}^3$ transformation matrix to process points of dimension $\mathbb{R}^3$. With this additional networks they gain an improvement of 0.8% [17].

The augmented point is then passed in a MPL with shared weights. The idea of weight sharing within a MPL seems counterintuitive on the first glance. An MPL actually defines itself through a full connection. It does make sense, though, when viewed in the context of convolutions with a filter size of $[1 \times 1]$. Throughout one filter all weights are shared. As the extent of the kernel is one, e.g the kernel processes one point at a time, the weights only depend on the dimension of the input in our case (x, y, z) (or x, y, z, R, G, B when we include color). This brilliant idea decreases the weights per filter dramatically and pays respect to the invariance of points. If the kernel size was larger, we would need to define proximity between points, which we do not have since there could be $N!$ permutations in the input. At last, it benefits the implementation, because convolutional layers are a fundamental and highly optimized building block of every deep learning framework.

After the first sequence of STN, shared MPL and another STN, the calculated local point features are buffered. A stacked sequence of shared MPLs aggregate the global features out of the local ones. Finally, the max pool operation reduces these features to a global feature vector of dimension 1024. The max pool operation breaks the dependence on the order of the points. It does so by symmetry. The problem, still, is that we filter the features per point but cannot aggregate the global feature vector which is independent of the input ordering of the points. In general, the remedy for the above-stated permutation problem is simply a symmetric function [40]. Common examples are:

$$f(x) = \max\{x_1, x_2, x_3 \dots x_n\} \tag{4.1}$$

$$f(x) = x_1 + x_2 + x_3 \dots x_n \tag{4.2}$$

The max pool operation works on the max operation shown in 4.1. Evidently, we can see that this function is independent of the input order.

[1] There is also the *vanilla* version without the spatial transformer network. We won't consider it in our discussion, as it is not for significance for our task.

4.4.1 Classification

The global feature vector carries enough information for classification purpose. It is processed by a simple stack of fully connected layers (or real MPL without weight sharing). They regress the global feature vector of size 1024 to a vector of size k, where k is the number of classes the net was trained on. Classification was not investigated in this thesis, because it does not have significance for the progress tracking. To apply classification, we would need to preprocess the construction site into pieces which the net could unambiguously classify. We want to explicitly circumvent this task.

4.4.2 Segmentation

If we could identify the class of an object by pixelwise classification, we had more information at hand. And we could state more precise progress of the site. Therefore, we make use of segmentation, which assigns a class to each point separately. In theory, points of the same class cluster and form an object of interest. PointNet archives this through the combination of features. We recall that we buffered our local point features before we extracted the global features out of it. PointNet concatenates the 1024 global features with the 64 local features and obtains a new feature vector of size 1088, see 4.2. This vector is sampled down again through repeated application of shared MPLs until the last layer. Here a softmax classifier uses one-hot encoded labels per point. PointNet is trained with cross-entropy loss, both classification and segmentation, see 3.4

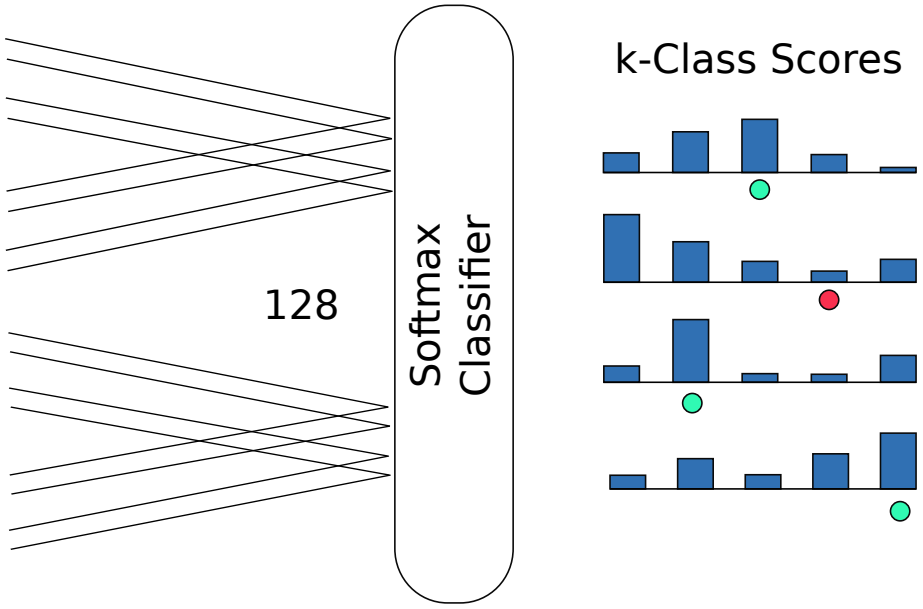


Figure 4.3: Softmax layer visualized

Figure 4.3 visualizes the last segmentation layers. The black lines indicate 128 the feature vector size in the second last layer where the number of features are reduced to the number of classes per points. Each neuron predicts a probability for each point being a specific class i. The dots indicate their label. The color green means the highest probability of the prediction corresponds to the label. Red means a wrong prediction. It is important to note that, despite the fact that PointNet is invariant to input permutations, once the points are processed their order must not change anymore.

4.5 PointNet++

PointNet provides an efficient way to process point clouds, but its design lacks hierarchical feature aggregation wrt. point neighborhoods on different scales. PointNet derives its suggestion for pointwise segmentation only from per point features and global features through one pooling operation. This is suboptimal, as the surrounding of points encode highly important information of the class it belongs to. This opens up the question again how to structure space and how to define a "neighborhood" which is robust even in sparse regions?
PointNet++ [45] is PointNet's successor which tries to find an answer to this question. The idea revolves around set abstraction layers, which consist out of a *sampling* layer, a *grouping* layer and, indeed, a PointNet layer. A set abstraction layer processes its input matrix to form the desired output matrix in the following way: $N \times (d + C) \rightarrow N' \times (d + C')$. N is the number of input points, d the dimension of the metric space (normally 3 for (x, y, z)) and C is the dimension of features, which can be color or normals in the input layers. The amount of input points is reduced to N' when processed in a set abstraction layer. This does not matter for classification, but for segmentation it introduces complexity.
The **sampling** layer samples a subset N' of all input points N in an iterative farthest point manner. The authors claim better performance compared to random sampling. Intuition confirms that as points which are farthest apart are most likely key feature points which form the skeleton of the shape they represent. The subsampled centroids N' define the local region through the ball-metric.
The **grouping** layer takes the subsampled points from the previous sampling layer and additionally a small set of centroid points within the set and groups them. These centroid points form the "neighborhood" depending on the ball radius around them. The output is of form: $N' \times K \times (d + C')$. Where K indicates the number of points per group. C' is the dimension of new aggregated features. K can have different values, which correspond to the points within the ball radius around the centroid.
The next layer is, indeed, a **PointNet** layer. As we know from 4.4, PointNet has by design no problem consuming a varying number of points. Each point in one group K is transformed into a local reference frame defined by the centroids contained in N'. PointNet now extracts features, exactly the same way as it does in its original version. Differently, the region where

it is applied is determined by the sampling and grouping algorithm. The extracted features only correspond to this regions.
The remaining questions are. How to aggregate features throughout the set abstraction layers and how does this aggregation influences the choice of the parameters used in these layers? The locality approach introduces a problem when the point cloud becomes sparse locally. The lowest region can therefore not produce robust features and may propagate wrong information downstream. Two remedies for this problem were investigated.
Multi-Scale Grouping (MSG) solves this by constructing different scale groups for the PointNet layers and concatenates the features. This approach is straight forward but is computationally more intensive as the PointNet layer is used multiple times on one centroid. The more promising approach is called Multi-Resolution Grouping (MRG) where the features aggregated in a convolutional manner. Each level, except the lowest, concatenate their computed features with the one from the level beneath per centroid. This way the feature vector gathers feature information from each level.
Regardless of which approach was used, none of them produced feature vectors for all points because they were subsampled in the beginning. The authors used interpolation between points of the same level and use skip links between levels to restore the missing information for input points which were not processed. The interested reader is referred to [45] for details.

4.6 Summary

The PointNet flavors provide an advanced technique to extract features from point clouds. They address the problems of consuming an unordered set of points while being invariant to input permutations. Their findings are solely based on benchmark datasets. This makes it hard to estimate PointNets applicability for real-life tasks as construction sites in advance. Experience has shown, that there are two limiting factors for PoinNet(++)'s application which might be hard to overcome. The first one is the simple fact that training data is from crucial importance. Generating high-quality data with realistic perturbations is a most challenging task.
The second factor is PointNet's limit to use 2500 points per input. This number is not large in the world of point clouds and interesting features might only appear over large distances. Future research will show if there is enough room for improvement to overcome these limitations.
Despite these facts, PointNet seems to be the only universal approach to classify point clouds. Manually feature engineered approaches might work better, but this highly depends on the circumstances.

Chapter 5

Point Cloud Generation

The pivotal task of this thesis was to develop a concept to use PointNet [46] [45], or its successors, in order to detect certain objects in point clouds from construction sites. PointNet, as explained in chapter 4, is a DNN trained in a supervised manner. This leads to the common question where to get labeled point cloud data from? Especially from the domain of construction sites.
Recent research, as explained in chapter 1, uses point clouds to derive construction progress and delays. These point clouds have no labels. Labeling them would require hundreds of work hours, and a framework which is suitable for this task, as used in the research of [13], similar to [49] for pictures. Their annotating framework was designed for detailed room scans and is not optimized for clustered heterogeneous construction sites. Even if it was, the task would be too time-consuming still to label enough point clouds to train PointNet.
As the DNN get deeper and more complex during the last years a common technique called *transfer learning* [58] was established in the deep learning community. This technique provides weights from an already trained NN. These nets are usually trained on current datasets, see 2.5, and therefore do not need to be trained from scratch. The user would allow gradient updates only on the last view layers, assuming that the former layers have learned to generalize well enough. This allows for a significantly smaller dataset.
Also this approach was discarded, as a significantly smaller dataset would have needed time-consuming annotations, too. Further, the selection of trained PointNet models was limited to the one trained by Charles R. Qi et. al. Using a trained model makes it hard to alter the model architecture as the weights and biases are tied to it. This is not ideal for research problems. It would have also limited us to TensorFlow as the deep learning framework.

Because of these reasons, the decision fell on data generation. The idea was to generate point clouds by sampling them from mesh (stl, ply, obj,etc.) representations, see 4.1. This has some advantages:

- infinite amount of data to train, evaluate and test
- the flexibility to use any mesh object in the dataset
- the possibility for fine-grained data augmentation
- the possibility to compose objects out of other objects

The framework was programmed with these features in mind. These advantages do, of course, not come without disadvantages. The generated point clouds do not reflect low-level features from real-world data, and fail to resemble a construction site overall. Without alternatives to the proposed plan, the task was to generate the point cloud as random as possible to make the net learn a broad variety of features.

5.1 Generator Architecture

We provide an overview of the architecture but will not explain code details. Python is the programming language of our choice. Mainly because it is all-purpose and object-oriented. It comes with bindings to all major deep learning frameworks, has nice integration of n-dimensional array manipulation though Numpy [44] and a large community, fostering scientific coding through Anaconda.
The goal is to obtain a ready to train point cloud training set as a file. Usually, datasets are created in a file structure. We discouraged this approach in favor of the hdf5-file format as the storage for the point cloud data. Hdf5, explained in 5.2, does not just store massive amount of data in a simple structured way, but also allows for metadata which was the decisive criteria. Metadata allows having everything packed up in one large file which can be easily deployed on a server. A drawback is, that generally, every data piece in a hdf5-file is not simply accessible without a Python script. We split the data generation process into three stages.

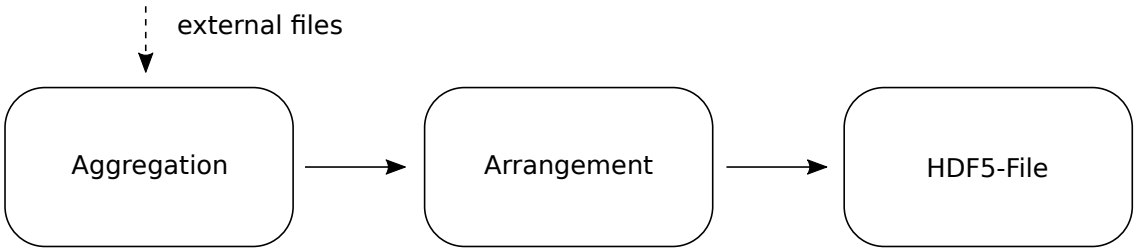


Figure 5.1: generator stages

5.1.1 Aggregation

The first task was to handle different mesh file formats in order to load the mesh into memory. A lot of external libraries were tested and only a view were taken as dependencies. The core geometrical functionally formed the Trimesh library[1].
From experience, these libraries provide a mesh object with numerous methods. It is easy to retrieve the mesh's vertices or the connection graph as numpy arrays.
To not lose functionality, a wrapper class around the Trimesh mesh object was created which we denote `WrapMesh`. The wrapmesh holds information of the mesh, a name and the class label for future point sampling. Additional effort was set upon directly generating objects like a house, container and scaffold. These objects did not come from a file.
A strong requirement was to be able to use objects composed out of many warpmeshes. Because scaling and rotation shears the object when its center of gravity does not coincide with the origin, the element class is introduced. The `Element` aggregates wrapmesh objects, and worked as a proxy for wrapmesh methods in order to provide a cleaner Application Programming Interface (API).

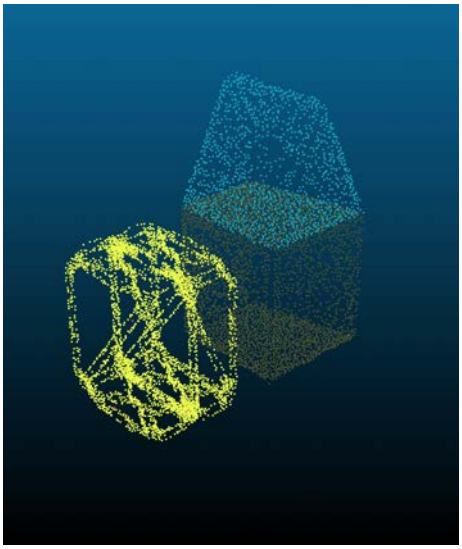


Figure 5.2: element composed out of three WrapMehes

Figure 5.2 displays an element which houses three different wrapmesh objects. The colors indicate the different class labels as *scaffold*, *roof* and *body*. Please note, that labels don't carry any transitive information. A roof is not a subcategory of house, for example. PointNet's architecture would not allow for that kind of relations. Nevertheless, due to the implementations through the element class these relations can be mapped easily. The element also computes the ground truth for the whole mesh on the xy-plane which is important later on. In order to create a meaningful point cloud from elements, they must be distributed over an

[1] https://trimsh.org/index.html

area while their default size should vary.
For this reason, the factory pattern was used. Providing a factory for each element together with a `Transformer` class which transformed the elements in stipulated random ways. The transformer class generated a transformation matrix tailored to the size and type of the element. To apply transformation correctly the object's center of mass is moved to the origin. We used homogeneous coordinates for transformation, as shown in equation 5.1.

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad S_{x,y}(\boldsymbol{b}) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Tl_{x,y}(\boldsymbol{c}) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

Where R is the rotation matrix, S scales the element and Tl is the translation to the origin. To combine them we multiply them from the right.

$$T(\alpha, \boldsymbol{b}, \boldsymbol{c}) = Tl_{x,y} \times S_{x,y} \times R_z \tag{5.2}$$

Another class, called `ElementSequence`, holds all element factories, and invokes the construction process in each factory. For every iteration, all the elements have the same probability to be created.
Figure 5.3 shows STL meshes and their corresponding point clouds.

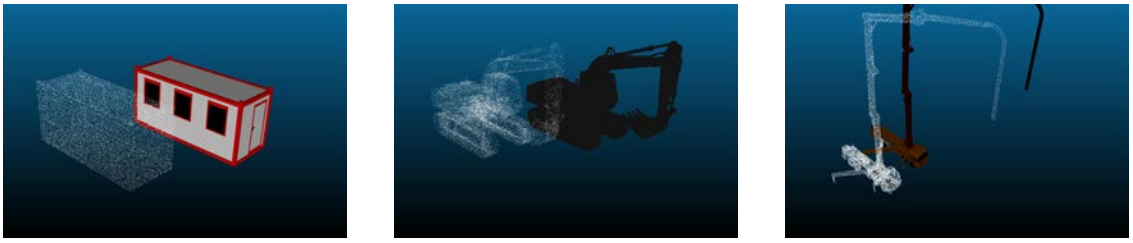


Figure 5.3: STL examples and their point cloud representation (container,excavator,concrete pump)

At this sage we have created the elementsequence object that can be iterated over. In each iteration step a randomly selected factory creates a scaled and rotated element. The transformation is determined by the transformer class. The elements must then be positioned in a specified area, which we denote as a *Scene*.

5.1.2 Arrangement

The arrangement was an unexpectedly involving task. The random sized object must not intersect but should be placed as random as possible. In the following explanation, "element" still refers to the object, but for simplicity, only the ground truth representation is shown.
Figure 5.5 shows the possible arrangement strategies for different elements. A `Scene` object

defines the size of the populated area and holds the elements. The gray rectangles constitute the ground truth for each element. The ground truth is derived from the scaled and rotated elements and is axis aligned, see figure 5.4. Therefore, rectangles touching each other does not mean the point clouds intersect.

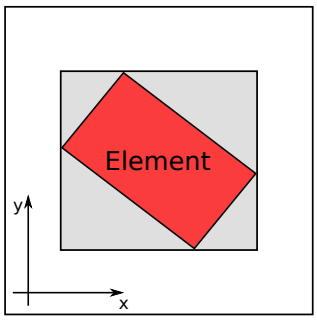


Figure 5.4: ground truth and element area

For the **random approach**, the idea was to sample points uniformly at random, and use them as the element's center, see image in figure 5.5. The random approach was abandoned quickly because the scene was populated too sparsely. And intersections couldn't be excluded completely.

The **closest to mid** approach tried to iteratively move randomly placed objects near to the origin. One object after another is moved along an imaginary line which goes through the origin and the element's center point (projected on the x,y-plane). One iteration moved every object a distance t towards the origin. If two objects intersected, no movement would happen.

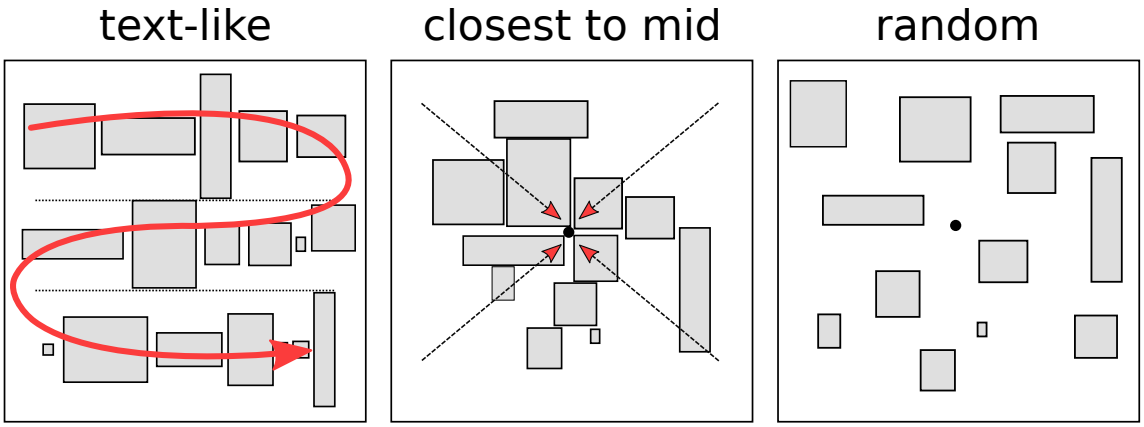


Figure 5.5: three scene arrangement solution

The iteration ends when no objects are able to move closer to the origin. This approach generally suffered from a high computational workload. At unfortunate placements, the objects

got locked. Intersections were calculated through the Trimesh intersection module which was faulty, and did not recognize intersections correctly. The incomplete documentation complicated the pursuit of a solution. Therefore, this method was discarded as well.
The **text-like** approach interpreted each element in the queue as "character" and placed them on a straight line. The two largest elements defined the minimum distance between two rows as depicted as dotted lines in the left image in figure 5.5. This method worked quite well, but also produced too much space between elements. In addition, small elements placed around a bigger element were not possible.
The final decision fell on a mathematical problem called *packing problem*, more precisely the *knapsack problem* [1]. We used the python library `rectpack`[2] as an implementation, instead of implementing it again. Figure 5.6 shows a solution plot for the problem fitting n rectangles in a predefined area.

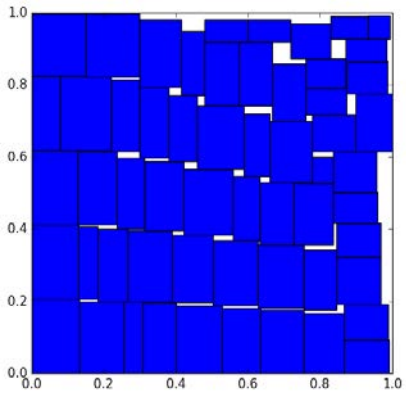


Figure 5.6: Knapsack problem
https://github.com/secnot/rectpack

The packing algorithm was used to process each generated element sequentially. It started in the lower left corner and placed the elements best fitting. The Knapsack packing algorithm was primary created for problems including more than one area, which are called *bins* in this context. Based on a *fitness* measure, the rectangles where placed in the best fitting bin. It was also totally fine to not find a suitable bin for the algorithm, which then simply discarded this element. This led to subtle bugs. Because of the algorithm's nature, all elements were clustered near the origin. Additionally, relative small elements filled gaps created by larger ones and therefore amplify clustering.
Figure 5.7 shows example plots for a scene. This approach was random enough and intersections could be ruled out.

Until now, we worked with mesh objects from Trimesh. Sampling points from triangulated meshes seems straight forward but includes a pitfall. The naive approach would suggest to just use the nodes from the mesh as a point cloud. This limits the number of points to the nodes of the mesh which is hardly enough to call it a point cloud. Furthermore, the mesh nodes cluster in regions with high curvature, which leaves the number of points biased towards curvy regions.
We used `Pyntcloud`'s[3] sampling algorithm, based on the area of the triangle, to sample $\boldsymbol{n}$ points uniformly from the mesh. The number of points is regulated through a parameter

[2] https://github.com/secnot/rectpack
[3] https://github.com/daavoo/pyntcloud

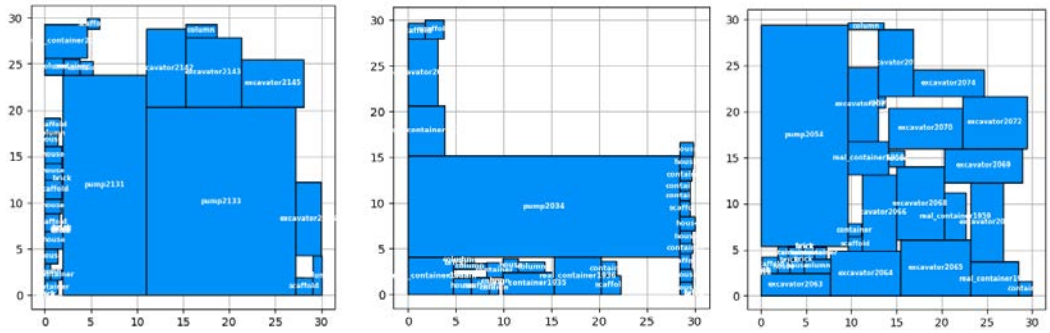


Figure 5.7: Ground truth of three scenes. Clusterings can be observed in the left and in the middle image. Sparse regions are unfortunately still present. These image illustrate different scales of the objects, for example excavator and column.

provided to the element factory. Small elements get a lower number of points whereas bigger elements get more. This parameter can be set in the corresponding element factory. Because the algorithm depends on the area of each triangle of the mesh, this sampling became a bottleneck when more than 50,000 points where requested. An open task remains to discard the use of Pyntcloud and use the underlying sampling algorithm directly because it is completely based on pandas [37][4]

5.1.3 Sampling Slices

In chapter 4, we described the PointNet in detail. It can process 2500 point at max directly. This is quite little in the domain of scene scans where 1 million points are still considered small. The generated scene had to be preprocessed in order to be consumed by PointNet. We will denote chunks of the point cloud with more than 2500 but less than 5000 points as *slices*.

With the split-of-concern-principle in mind, we had the idea to leave slicing to the PyTorch dataloader. The dataloader organizes the load process for a NN. It has a nice interface to augment the provided data on the fly and in parallel through so-called workers. Unfortunately, the dataloader and the hdf5-implementation did not work together at all, even though hdf5 aggressively supports parallelism. Only one worker thread worked as expected, which killed any performance boost.

The advantage of the dataloader would have been that only whole scenes were required. Additionally, we could have used the same procedure for real-world laser scans, too. Unfortunately, the slicing problem was computationally too expensive and not realizable in real time during the training process with only one worker. Therefore, slicing also became part of the dataset generation process. Two sampling techniques were further investigated.

[4]Pandas belongs to the famous scipy ecosystem and provides highly optimized two and one-dimensional data structures for statistical computation.

k-nearest Neighbors

The idea was to pick points at random and use their 2500 surrounding neighbors as one slice. First of all, we calculated a kd-tree for the point cloud. The resulting matrix contained the indices of the neighbor point as row vectors. By sampling a number between zero and the number of points in the point cloud, we used the number as an index to pick a slice and deleted the corresponding points from the point cloud. Then, we needed to recalculate the kd-tree in order to give credit to the missing points we sampled the slice from. This, approach suffered from redundant recalculations of the kd-tree.
Furthermore, this procedure created spatially widespread slices the smaller the point cloud became which is not a desired property for training data. Additionally, from uniform regions, the algorithm sampled the slices in a spherical fashion. This makes sense as a sphere defines the nearest neighbors on a plane region. This led to a heavily biased dataset in the sense that the network learned spherical features where none exist in reality. Due to the downsides described above, this approach was discarded.

Equal Slices

The equal slice approach mitigates the spherical sampling problem by equally partitioning of the point cloud in rectangles with a predefined size. Straight forward, we developed an algorithm which divided the point clouds into slices according to a given width and length (w, h). Figure 5.8 shows renderings from one scene. The renderings on the right display the full point cloud whereas the renderings on the left depict the slices cut out from the cloud.

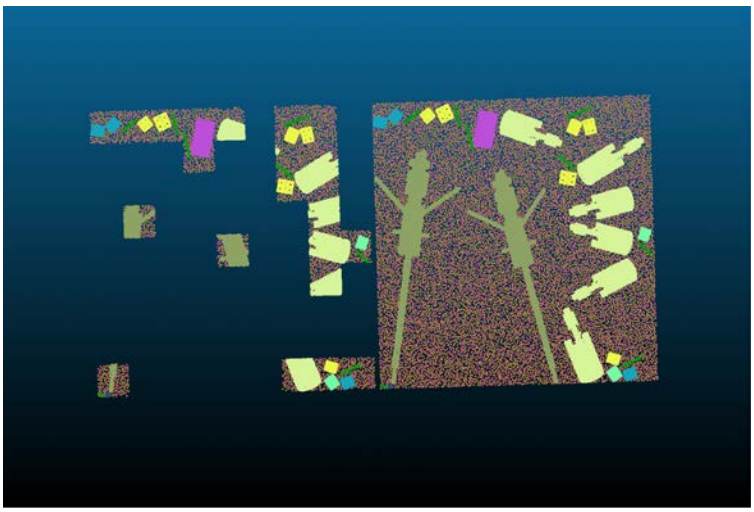


Figure 5.8: Four point cloud renderings depicting the same scene. The colors indicate the class for each point and have no further meaning. (pale yellow:excavator, green:bar, purple: container, blue: roof, brown: house body, beige: concrete pump, turquoise: container(cube), yellow:scaffold)

Due to the high variation of point density, there are only a few slices generated per scene which have enough points to be consumed. The measured ratio of all slices and slices with more than 2500 point is between 25% - 30%.
The hard to overcome issue is that, on the one side we are constrained by the upper bound of 2500 points per slice, but on the other we seek to classify large objects in a scene. If the object is larger than a slice, then its learnable features span across multiple slices. This destroys the connection between them, and the NN is not able to comprehend their correlation. This fundamental problem has a high priority in further investigations.

5.2 HDF5 Dataset

The Hierarchical Data Format (HDF5) was developed by the non-profit HDF-Group company to satisfy the needs of scientist to store massive amount of heterogeneous data in an ordered way. HDF5 is a self-describing binary file which is able to represent n-dimensional datasets without limitations in size. Its elements, in turn, can self be complex objects. It's meant for easy sharing, it is not platform agnostic and parallel I/O is natively supported. It provides meta attributes which describe the stored data.
The library is implemented in C but almost every common programming language provides wrappers around it, like h5py which is used in this thesis. HDF5 files are composed out of *groups*, *attributes*, *datasets*. Groups can be viewed as folders holding datasets. Datasets are n-dimensional arrays holding different data types and structures. Attributes are the meta information of the data stored and might be, for example, a simple string. Because hdf5 datasets are ordered and self-describing they suite very well as a data source for deep learning tasks.
The sampling algorithms explained in section 5.1.3 are interwoven with the creation of the hdf5-file. Nevertheless, for the algorithm it makes no difference how the actual points are sampled. Therefore, these sections can be treated independently. During the programming process, a lot of different hdf5-file structures were tested. In order to explain the dataset creation, we first need to look at the PyTorch dataloader and dataset class because they highly depend on each other.
The PyTorch dataloader gets a PyTorch dataset at instantiation time. During training, the dataloader is iterated over, which in turn iterates over the dataset. The dataset class defines an abstract base class for the user to derive from. It demands, that Python's famous magic methods `__len__` and `__getitem__` are defined. Where `__len__` refers to the "length" of the object and `__getitem__` is a method, which defines the logic of which item is returned when the object is iterated. These are only a small selection of python specific functions, but they make Python so effective. For more details the reader is referred to the Python documentation[5].

[5] https://docs.python.org/3/reference

The `__len__` method obviously refers to the length of the dataset whereas the `__getitem__` method returns a tuple of the actual data and its corresponding label. There is normally not too much computation carried out in the `__getitem__` method beside the loading procedure of the data and the label from disk. The dataloader, in turn, is responsible for data augmentation on the fly. For images, this includes cropping, transformation, adding filters etc.
In our case, the dataset gets the hdf5 file and must be able to understand its structure in order to load the data from it. Therefore, more logic is contained in the dataset class.
During the dataset creation, an user-defined number of scenes n is created. The slices of every scene belong to a *group* to render them later on. Every scene is partitioned in $i \times j$ slices which makes $n \times i \times j$ slices in total. If every slice had a sufficient number of points, we would know the exact number of slices per group and indexing would be no problem. But the exact number of slices per scene is only known after slicing.
As described in 5.1.3, the chosen *equal slicing* did not guarantee that all slices from the scene where suitable for the training phase. A simple remedy for this problem is to not store slices from the same scene together in one group of the hdf5-file but sequentially. Rendering scenes and evaluations become impossible this way because we do not know which slices belong to the same scene. A simple solution would have been to add a tag to each dataset which shows the corresponding scene, but this was unknown during the programming phase.
Figure 5.9 shows the diagram of the data storage format.

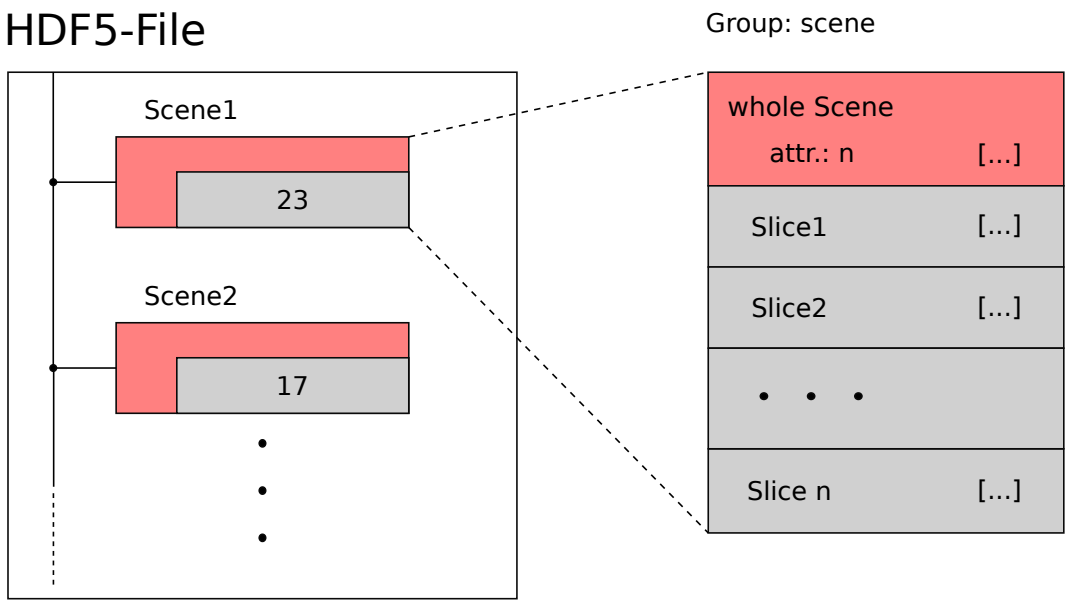


Figure 5.9: initial structure for the HDF5 file

On the left, we can see that the file is structured into m groups, which contains n slices (gray). Additionally, the whole scene is saved as well. This is done to compare the slices

with their corresponding scene where they are sampled from. And to detect the parts the algorithm has discarded in the rendering process. The numbers (23,17) indicate the number of slices in each scene group. The dataset knows this structure and iterates through all scene groups, while counting the visited slices, until the counter equals the slice index. This needs further optimization.
The duration to create a dataset is linearly dependent on the number of scenes we want in our dataset. Further studies have shown that sampling the points from large and complicated meshes, like the concrete pump, becomes a bottleneck. We can assume 2-4 hours for a sufficiently large dataset.

5.3 Training

The dataset generation process and the training are distinct phases. Nevertheless, the dataloader's part is quickly explained here.
The dataset class, which reads the hdf5 file, provides the raw slice to the dataloader. Because our constraint on the slice was that it must at least contain 2500 points and 5000 at max, the dataloader must subsample the slice further to yield exactly 2500 points. It does so randomly. This leads to an interesting fact, that the NN does not encounter the exact training example twice, which is highly desirable.
On demand, the dataloader augments the points by subtracting their mean from them. This gives credit to the fact that it is not important at all for the class at where it is placed in 3D space. The slice is translated into the origin by subtracting the mean from it. This form of augmentation is necessary in order to not bias the network with point values from one range, e.g. only positive. At inference time, this would perform poorly on point values in negative range.
When a dataset of pictures is normalized through its mean value, it is highly important that the mean is only calculated on the training set. This calculated mean is then applied to the test validation set and during inference, see [29]. For our 3D points, this is not important as we calculate the mean for each slice independently. But is also clear that the NN has only seen values defined by the measures of the slices it is trained on. This is a trade-off we have to take. A further consideration is to completely normalize the slices by squeezing all point values into the unit sphere.

5.4 Improvements

There are only a view inconveniences in this setup which became only clear after the first training runs. First of all, there is no gain for the complex class hierarchy only to carry the mesh representation until the sampling algorithm samples the points from them. At this

point we suggest, to discard meshes from the process completely, expect the very initial phase where they are loaded. The sampled point cloud would completely replace them leaving us with the simple Numpy array representation. This would simplify the whole generation process a lot. The disadvantage is that transformation must then be applied to the number of point in the point cloud, which could become a bottleneck as well. In the spirit of this simplification, the point cloud and the labels should be processed separately. Currently, the labels extend the point cloud array as a fourth dimension. For homogeneity reasons, the labels have the same floating point precision as the rest of the points, e.g. `float32`. This is a waste of memory as `uint8` would be more than sufficient for labels representing 20 classes at max.

The hdf5 file structure has proved its superiority when the user is familiar with it, but it can be drastically simplified. Two specific ideas need further investigation. Hdf5 provides pointer-like references to all hdf5 types like groups, attributes, or datasets. Additionally, it allows to reference regions **inside** a dataset. This allows for only on massive datasets, containing all points from all slices. The reference would be used to define where a slice begins end ends. Arranging the references into groups of scenes further allows identifying the whole scene.

Another idea was to not group the slice datasets into scene groups, but leave all of them in the root level. A simple integer attribute would allow sorting the slices by scenes. This approach seems simpler to implement.

Minor improvements contain revision of the `element factories` by providing a base class where every factory inherits from. At generation time, the factories get the class index as an argument independently from each other. No logic checks if a class number is accidentally assigned multiple times to different objects which is error-prone. Supplementary, the total amount of classes must be set manually, which could also be automated.

Chapter 6

Results

As we have now seen the workflow of PointNet and the generation of training data, we can now dedicate ourselves to the actual results.

6.1 Choice of Implementation

Beside the papers for PointNet[1] and PointNet++[2], Charles R. Qi et al. provide implementations on Github for research use. Both are implemented in TensorFlow. As explained in 4.5, PointNet++ uses farthest point sampling to generate subsets from the input point cloud. For this rather exotic purpose TensorFlow does no provide a default implementation. Therefore, they implemented a custom layer in C++, tide to a version 9.0 of Nvidia CUDA's deep learning library `cudnn`. To run PointNet++ at all, one needs to compile this layer at first. Additionally, one needs cudnn9.0 and therefore a Nvidia graphics card.
TensorFlow composes a graph of user-defined computations and compiles it to hardware instructions. A `session` is needed to run computations at all. The computational graph is heavily optimized for training but must not change after its instantiation. This makes it hard to debug, test and comprehend. On the other side TensorFlow is known for its incomparable speed.
These reasons led to the choice of a custom implementation of PointNet in PyTorch. PyToch, in contrast to TensorFlow, builds its graph during runtime repeatedly for one forward and backward pass. This design allows logical control flow statements, like `if/else`, in the computational graph. Further, tensors are accessible right away, and therefore much easier to debug. We used the implementation from fxia22[3] for this thesis.

[1] https://github.com/charlesq34/pointnet
[2] https://github.com/charlesq34/pointnet2
[3] https://github.com/fxia22/pointnet.pytorch

This code was adjusted in large part. Especially the input queue was completely rewritten in order to load data from the hdf5 file.

6.2 Evaluation

The first test run already brought interesting insights. Figure 6.1 plots show the test/train accuracy on the left and the test/train loss on the right. The pale colors indicate the real values, whereas the full colored lines represent the smoothed version.

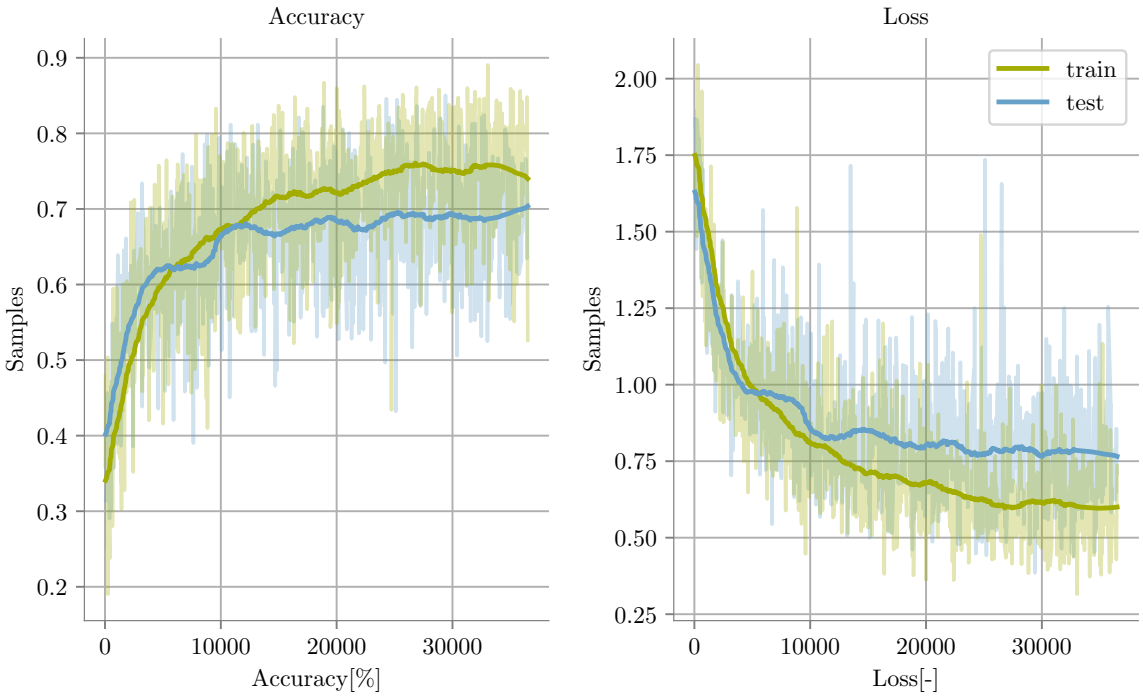


Figure 6.1: First Run

learning rate: 0.01 momentum: 0.9 batch size: 20
epochs: 250 classes: 11 slices: c.a. 6000

The training set, for this run, was generated with 11 different classes and approximately 6,000 slices. For training the Adam optimizer was used with a momentum of 0.9 and a batch size of 20.

At first glance, the graphs look pretty normal. The Accuracy rises and saturates as well as the loss, only in reverse. The training loss is also lower than the test loss which, in turn, leads to a higher train accuracy. Overfitting is not observable as the test loss still decreases, but cannot be fully excluded. On the whole, major mistakes in the training data and errors in the model do not seem likely.

Noticeable, though, is that the actual values for accuracy and loss are of high frequency. Oscillating graphs are not unusual, but to this extent, it is not. One possible explanation could be the batch size being too small. Therefore, we could not generate a stable gradient and the weights are exposed to biased updates. This increases the variance which, in turn, leads to unstable predictions, see [17].
Further, the train accuracy saturates early at a value of around 70%. As we cannot observe overfitting yet, this indicates, that the model's capacity is to low. Another explanation could be that the training data produces too similar objects with many similar features. Due to the small size of each slice, the features might be ambiguous. Increasing the slices is not a solution. During training 2500 points are randomly subsampled to meets the nets constraint. If we would extend the slice the overall point density decreases which destroys local features. Increasing the point limit PointNet consumes above 2500, would lead to problems with max pooling. Max pooling makes a DNN robust to slights perturbations in the input but also destroys the spatial relationship of features. Larger max-pooling layers would embrace this behavior.
A downside of PointNet's architecture is that it struggles to recognize scale. This can be observed by the frequent misclassification of bars. Their shape is quite similar to bricks. PoinNet's architecture enhances this behavior because it makes its predictions by low-level local features and global features. The max pool layers destroy the spatial relation. If the max pool layer filters feature for a cuboid, the local features cannot tell its spatial extent. Therefore, PoinNet's 'guessing' in this case. We believe, if the bar had distinct local features like notches, PointNet would recognize them better.

6.2.1 Slices

Figure 6.2 shows the evaluation on the test set. Green denote correctly classified points whereas red denotes the opposite. We can observe ten classes, like concrete pump, house (roof,body), scaffold, bar, brick, ground (up, down), container1, container2. Container1 is created out of a simple boolean operation. Container2 is sampled from a STL-file, see 5.3. We divided the ground into two regions, upper and lower, in order to test PointNet's ability to identify the same structure as two distinct classes only depending on its position. This works quite well. The other reason we used the ground is to already have a little number of points everywhere.

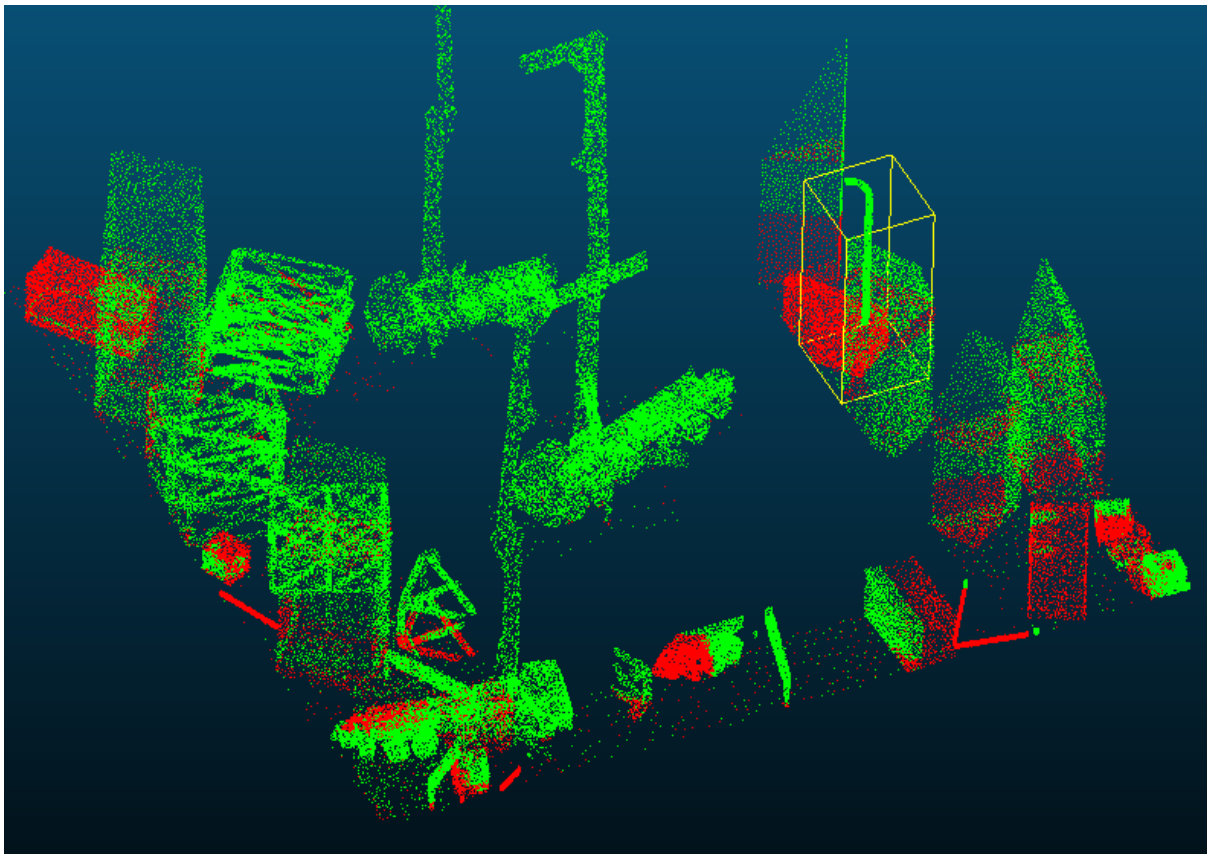


Figure 6.2: Evaluation on the test set

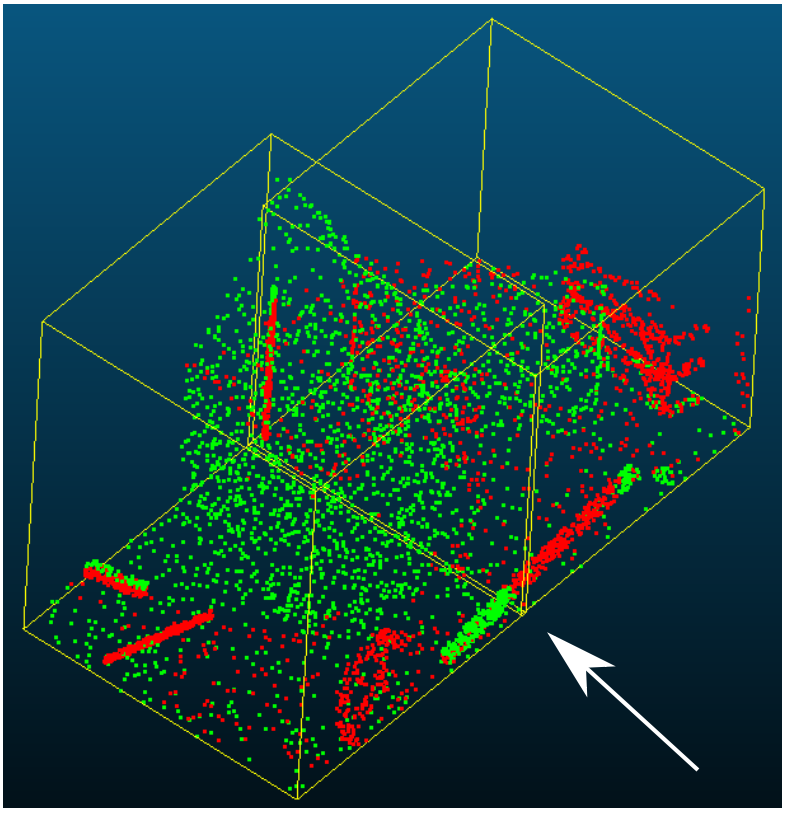


Figure 6.3: Information is not passed beyond two slices

Figure 6.3 shows two slices. On the left, the network manages to recognize the correct class. On the right site, it struggles. If the information exchange would happen between slices, such cases would be less frequent. It would be definitely worth investigating a dynamic slicing technique where objects are not cut.
We have to put this constraint into a bigger picture. The net cannot concatenate information throughout slices where one slice's spatial extent amounts only a few meters. Thus, the net has initially no chance to capture objects over a large extent. Whether this becomes a limitation, depends on the circumstances. In the scope of progress track, compact objects, which change their position a lot, are of interest. Objects like clutter, reinforcement steel, concrete bombs, formwork. Therefore, this approach seems still applicable.

6.2.2 Test runs

We extended the number of batches per epoch to 640. Despite the fact that more data is better, the increase in training data had not much influence on the results. Figure 6.4 show the graphs.

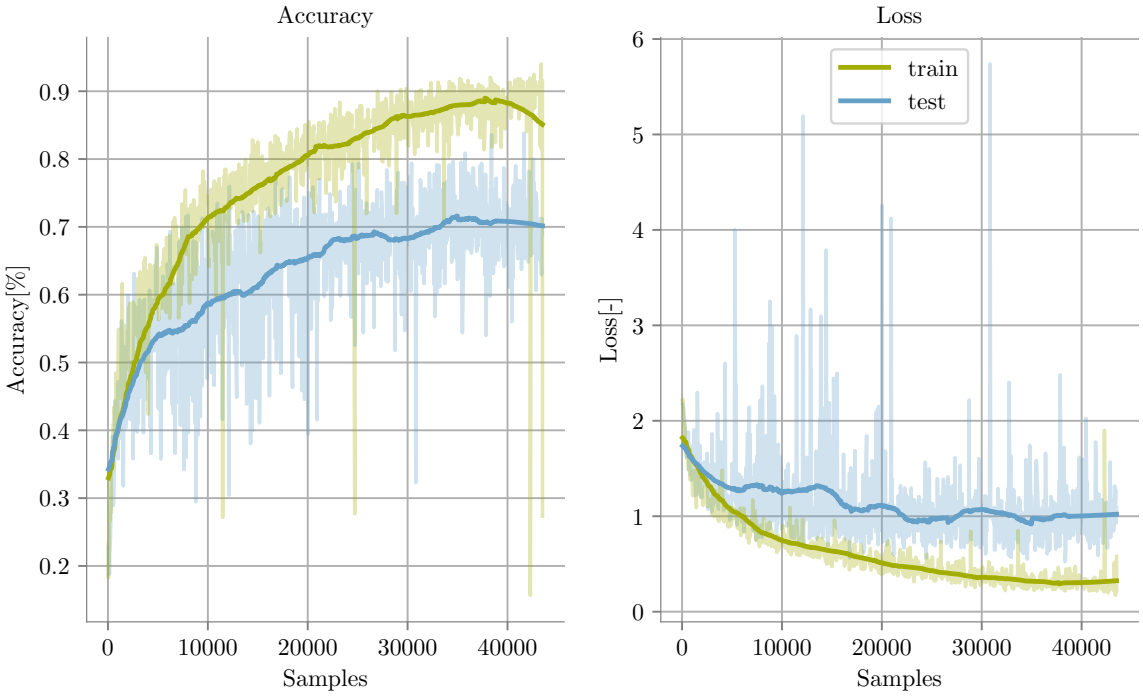


Figure 6.4: Run with larger dataset

learning rate: 0.01 momentum: 0.9 batch size: 20
epochs: 250 classes: 11 slices: c.a. 13000

We can observe a similar graph to 6.1. The training accuracy constantly increases, where as the train accuracy saturated around 70% as in the first run. This is a clear indicator that the NN overfits the dataset. The loss' graph confirms that behavior. The training loss decreases while the test loss has saturated. We can conclude that a training set with approximately 8000 slices is sufficient to train PointNet.
For the next test run, we reduced the number of classes to 3. The main objects of interest were *bars* and *formwork*. The rest was classified as clutter. Figure 6.5 shows the graphs. This stunning results must be viewed critically. We archived a very good test error of 3.68%.

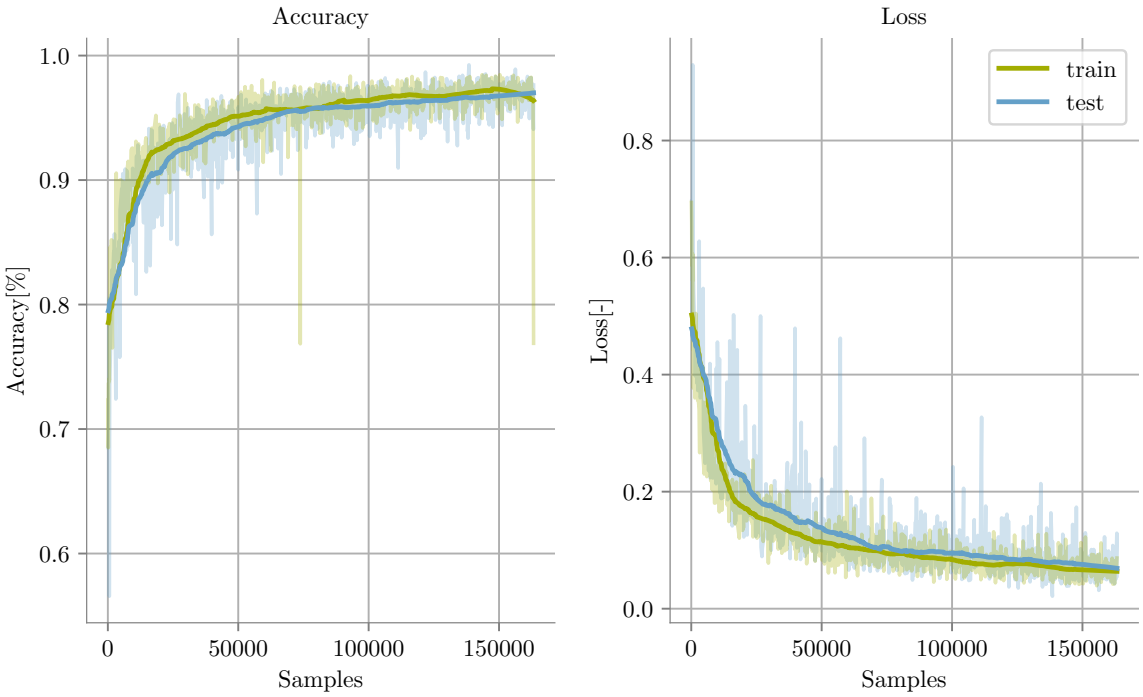


Figure 6.5: Run with only 3 classes

But as we only used three classes, the absolute amount of points belonging to classes *bar* and *scaffold* is already low. In other words, the net archives a pretty good accuracy by assigning *clutter* to all points. Figure 6.6 shows the models predictions as well as the boolean representation. Despite the fact that the model could assign *clutter* to all point, we observe quite a good prediction from the network. Even tiny objects like bricks are captured well enough.

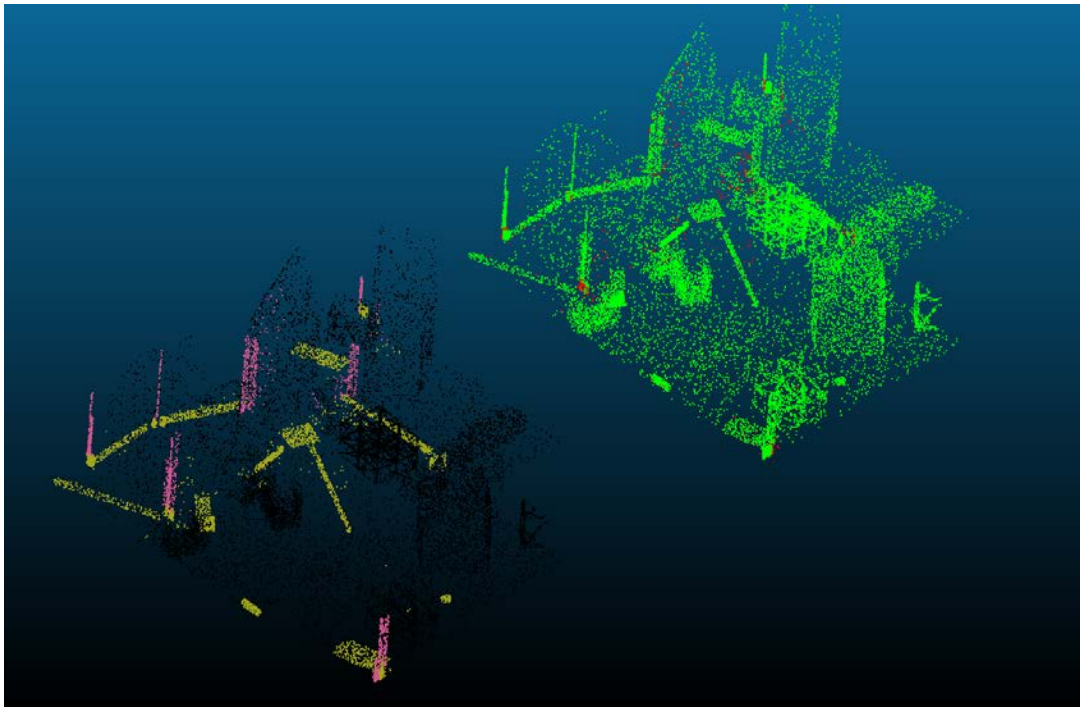


Figure 6.6: Prediction and boolean representation for 3 classes

6.2.3 Inference

In order to infer PointNet, data augmentation was necessary. The datasets contained the slices at positions where they were sampled from the scene. The scene is by design always located in the first quadrant. PointNet learned the object's position to be positive. To make inference invariant to the actual position of the slice in the coordinate system, we subtracted the mean from all points. This translated the slice to the origin. PointNet was now trained on these augmented data. We could not observe any significant changes in the classification error on the test set. In inference, in turn, we could be sure that PointNet was not biased by the absolute position of the slice.

Figure 6.7 depicts PointNet's prediction on real-world data. The colored point cloud on the right shows the pixelwise prediction of PointNet. The comparison to real-world objects is hard to draw, as we have no labels for it. The training dataset did not contain similar patterns which could be recognizable in the real-world point cloud. Nevertheless, we can observe that the predictions are not arbitrary. Similar predictions cluster where distinct objects can be observed in the input. The yellow regions in the back indicate the class *container*. There is obviously no container on the high ground, but a pile of formwork is located in that region, which has a similar shape. Furthermore, the scaffold is completely classified green, except its railings which are brown. This indicates a clear distinction between object shapes, the net is able to recognize.

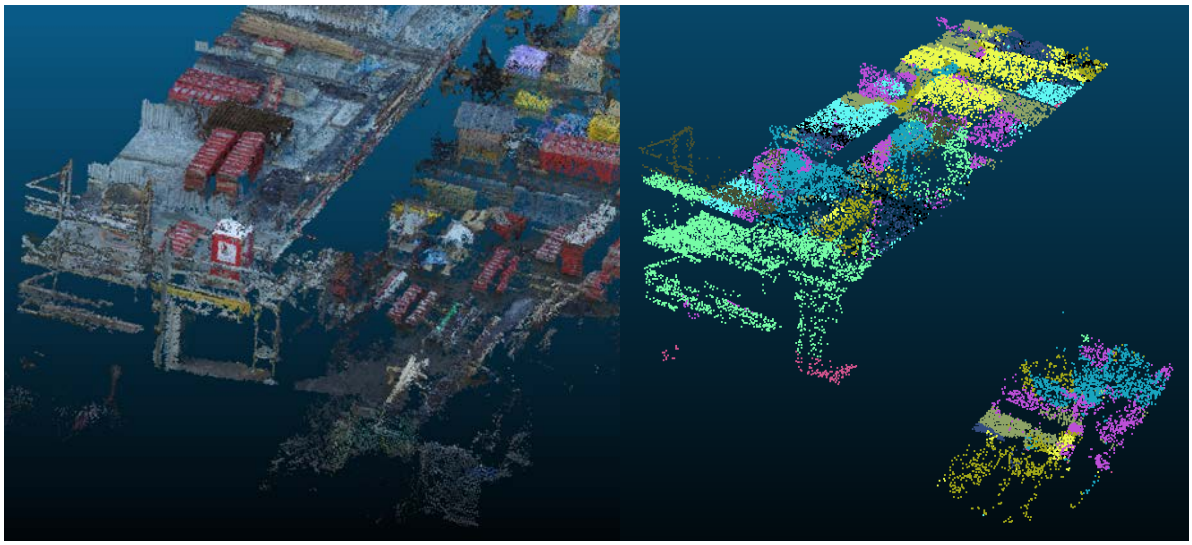


Figure 6.7: Inference on a real-world construction site point cloud

We can conclude that real-life application is not in reach yet. But the results are highly promising as they prove that the deep learning approach is the right choice for this problem.

Chapter 7

Outlook & Improvements

7.1 Deep Learning Approach

Deep learning is a universal tool. Its power combined with user-friendly frameworks lower the barrier for people to use it. But it is most likely not the right tool for medium complex daily tasks. We recall Occam's Razor and ask: Is deep learning the right approach for this problem? There are, of course, other promising methods available, see 7.4. Still, point cloud classification is an unsolved problem in general with many pathological corner cases in specific. Thus, applying deep learning seems like more than a justified approach. The decision to use PointNet was mainly due to the appealing property that it works on point clouds directly. In contrast, volumetric deep learning approaches lack the information on the lowest level. They render the point cloud unnecessary voluminous. It must be investigated if these architectures might yield better results than PointNet.

7.1.1 Additional Features

All our experiments were solely based on 3D coordinates without additional information. It is most likely that color information would drastically increase the performance and robustness of PoinNet. For example, the spatial arrangement of a scaffold's and a crane's struts may exhibit a similar feature vector, but color would completely determine the one from the other. Additionally, normals could also help PointNet to improve its predictions. In [46] the authors describe an experiment where they use PointNet to predict normals on a point cloud. This shows that PointNet is able to extract features corresponding to normals. If we would provide normal right from the beginning, PointNet does not need to learn them.
To add extra information to points, the generator becomes a lot more complex. Normals could be incorporated into the generation process without much effort. Color information is trickier as we need to sample it from textures.

7.2 Dataset

The quality of the dataset is crucial. Future work must decide if the generator approach will lead to sufficient improvements for real-life data. An important question is: Can a generator produce realistic enough point clouds to train PointNet?
We suggest an experiment in which we include real point cloud slices into the training process which represent clutter as a whole class. This would simplify the creation process and might help the NN to learn real-world pattern.
The classes per se need revision and a more careful definition. Using broader labels which include different objects, might help to generalize better in the first place. We suggest classes like 'vehicle' including concrete pumps, cranes, excavators, trucks and the like.
Composing different datasets where the focus lies on one specific classes would allow mixing datasets. For example, when the net struggles to recognize a truck, it might be useful to train on the vehicle dataset for a while.

7.3 Disaster: Max Pooling

As we have seen in 3.1.6 max pooling is a rather simple technique to reduce the complexity of the embedding space and extract the most important features out of it. Thus by design, information is definitely lost when passed through a max pool layer. The deep-learning community has an ongoing debate about max pool layers. One famous opponent said:

"The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster." – Geoffrey Hinton

But what is it, that makes max pooling a *disaster*. And what can we learn from Hinton in order to improve our PointNet implementation?
Geoffrey Hinton is a Professor of the Computer Science Department of the University of Toronto, Canada [22]. Coming from a background in psychology, he early believed in the power of neuronal networks. Cited as the "God Father of AI" in *The Telegraph*, he predicts a sinister future for the path AI-research is heading. As an opponent for *intelligent* lethal weapons, he signed an open letter from the Future of Life Institute [43] along with Stephen Hawking, Elon Musk, Steve Wozniak and Max Tegmark.
Scientifically, he is the inventor of CapsulesNet. A new neuronal network architecture which inherently preserves spatial relations between embeddings. His proposed technique to train the network is called "routing by agreement". This allows lower layers to decide to which neuron in the higher layer their output is routed. Therefore, the route through the network can be tracked and has an intuitive impact on the input. One downside is that routing by agreement implements a k-nearest-neighbor-like algorithm, which contains a loop. A loop is

not a usual construct in neuronal network architectures, thus it has not been optimized for modern deep-learning frameworks. Its results are state of the art on MNIST (2.5) dataset. But it lacks accuracy on CIFAR where the error is around 10.6%. This seems quite a lot, but as mentioned in thier paper [50], this is what all new major architectures achieved in the beginning.
Hinton aside, he is not the only one against pooling layers. 2015 a research group around Jost Tobias Springenberg published a paper called "Striving for Simplicity: The All Convolutional Net" [52]. As the name suggests, they use the property of convolutions to reduce the spatial size of the input image. Comparing with other architectures they archive an accuracy of 95.95% on the CIFAR-10 dataset what secured them the second place. But only place 19 on CIFAR-100 with an accuracy of 66.29% [2]. This suggests that their initial idea avoiding pooling layers does not scale to larger, meaning more complex, datasets.
Nevertheless, these findings definitely need more investigation wrt. our PointNet architecture where the pivotal point exactly is the max pool operation.

7.4 Point Descriptors

Before deep learning became famous, people engineered point descriptors. Point descriptors seek to extract information about one point of a point cloud and its neighbors which is invariant to scale, translation and noise. The idea is that robust point descriptors can be matched across point clouds. Therefore, a specific relation among point descriptors indicates the presence of an object. Point descriptors must be only obtained once. No time-consuming learning process is required. Their application on large datasets comes with a high computational cost. This is the disadvantage compared to DNN, which can be tweaked to real-time performance even on budget hardware. Two promising descriptors are explained in the following.
SHOT (Signatures of Histograms OrienTations) descriptors are inspired by the SIFT algorithm used in photogrammetry to generate point clouds out of images, see 1.3.2. The point cloud is divided into a spherical gird where the cosine between the feature point normal and the normal of neighboring points is computed. The results are packed in histogram bins. This way the histogram encodes information at this feature point. A classification algorithm (random forest) can be trained on histograms, recognizing common descriptors between the same objects. This is roughly the approach suggested in [62].
Spin Images uses the normals per feature point and rotate an imaginary canvas around it. During rotation, the canvas will be intersected by neighboring points. The frequency of the points intersecting one specific pixel is encoded in gray values. After rotation and smoothing the canvas contains an "image" of the neighborhood of the feature point. The canvas size is not scale-invariant and neither is the descriptor, see [27].
There are a large variety of descriptors: Histogram-based, transform-based, 2D view-based,

graph-based and combinations. None of them provides a general approach to robustly extract objects out of point clouds.

7.5 Conclusion

The perception of point clouds is an unsolved problem. In comparison to images, point clouds life in an even larger space. That is why it is evidently hard to extract useful information out of them. Common well-defined descriptor approaches for feature extraction have not solved this task yet. With the rise of more powerful deep learning techniques it seems straight forward to apply them on point sets. The pioneering CNNs only work on structured data, what point clouds are not. Because point clouds are only used in specific cases, there is only limited to none labeled data.

In this thesis, we investigated exactly this two issues. Many promising deep learning approaches were examined. Our decision fell on PointNet, as it works in point clouds directly. Hardly any preprocessing - in terms of voxelization - has to be done to use it. We used a custom implementation in PyTroch.

In order to overcome the lack of point clouds data, we developed a generator. The generator randomly places mesh objects on an user-defined area. The user can further specify arbitrary transformations which then are applied to the objects. Finally placed in the right position, points are sampled from the meshes. This leads to scenes which resemble constructions sites. These scenes are sliced to be consumed by PointNet. Out of these slices, with the help of the hdf5-file format, datasets were created. Because we sampled the points from known mesh types, every point has a label assigned to it.

We showed in chapter 6 that PointNet is able to capture objects from datasets. Depending on the dataset and the number of classes, we archive 75%-97% test accuracy. This number shows that this approach works. We show further that PointNet's prediction fully depends on the instances provided in the dataset. Based on our findings, the future effort has to be put on tuning the dataset and add color and normals to it. Furthermore, ways to use the successor version of PointNet, PointNet++, have to be investigated.

Bibliography

[1] John J. Bartholdi. *Building Intuition: Insights from Basic Operations Management Models and Principles (International Series in Operations Research & Management Science Book 115)*. Springer, 2008.

[2] Rodrigo Benenson. http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354.

[3] A. Borrmann, M. König, C. Koch, and J. Beetz. *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. VDI-Buch. Springer Fachmedien Wiesbaden, 2015.

[4] F. Bosche and C. Haas. PhD thesis, 08 2008.

[5] A. Braun, S. Tuttas, U. Stilla, and A. Borrmann. Classification of detection states in construction progress monitoring. In *11th European Conference on Product and Process Modelling*, Limassol, Cyprus, 2016.

[6] A. Braun, S. Tuttas, U. Stilla, and A. Borrmann. Incorporating knowledge on construction methods into automated progress monitoring techniques. In *23rd International Workshop of the European Group for Intelligent Computing in Engineering*, Krakow, Poland, 2016.

[7] A. Braun, S. Tuttas, U. Stilla, and A. Borrmann. Bim-based progress monitoring. In A. Borrmann, M. König, C. Koch, and J. Beetz, editors, *Building Information Modeling*. Springer, 2018.

[8] A. Braun, S. Tuttas, U. Stilla, and A. Borrmann. Process- and computer vision-based detection of as-built components on construction sites. In *Proc. of the 35nd ISARC 2018*, Berlin, Germany, 2018.

[9] Nikhil Buduma and Nicholas Locascio. Fundamentals of deep learning : designing next-generation machine intelligence algorithms. Paperback, December 2017.

[10] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao,

Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository. *CoRR*, abs/1512.03012, 2015.

[11] Eric T. Chung, Yalchin Efendiev, Wing Tat Leung, and Shuai Ye. Generalized multiscale finite element methods for space–time heterogeneous parabolic equations. *Computers & Mathematics with Applications*, 76(2):419 – 437, 2018.

[12] Nvidia Corporation. Gpu-based deep learning inference: A performance and power analysis. Technical report, NVIDIA, 2015.

[13] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2017.

[14] William R. Denslow. *10,000 Famous Freemasons from A to J Part One*. Kessinger Publishing, LLC, 2004.

[15] uber facebook. https://pytorch.org/docs/stable/index.html.

[16] Karën Fort, Gilles Adda, and K. Bretonnel Cohen. Amazon mechanical turk: Gold mine or coal mine? *Comput. Linguist.*, 37(2):413–420, June 2011.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[18] Google. https://www.tensorflow.org/api_docs/.

[19] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[20] Kevin Han and Mani Golparvar-Fard. Crowdsourcing bim-guided collection of construction material library from site photologs. 2017.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[22] Geoffrey Hinton. http://www.cs.toronto.edu/~hinton/fullcv.pdf.

[23] ImageNet. http://image-net.org/.

[24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[25] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *Advances in neural information processing systems*, pages 2017–2025, 2015.

[26] Katrin Jahr, Alex Braun, and André Borrmann. Formwork detection in uav pictures of construction sites. In *Proc. of the 12th European Conference on Product and Process Modelling*, Copenhagen, Denmark, 2018.

[27] Andrew Edie Johnson and Martial Hebert. Using spin images for efficient object recognition in cluttered 3d scenes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21:433–449, 1999.

[28] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[29] Andrej Karpathy. https://cs231n.github.io/.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[31] A. Kramida, Yu. Ralchenko, J. Reader, and and NIST ASD Team. NIST Atomic Spectra Database (ver. 5.6.1), [Online]. Available: `https://physics.nist.gov/asd` [2015, April 16]. National Institute of Standards and Technology, Gaithersburg, MD., 2018.

[32] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). *unknown*, 2009.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[34] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989.

[35] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[36] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[37] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[38] A. Meister. *Numerik linearer Gleichungssysteme: Eine Einführung in moderne Verfahren. Mit MATLAB®-Implementierungen von C. Vömel.* Springer Fachmedien Wiesbaden, 2014.

[39] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[40] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. BSB B. G. Teubner Verlagsgesellschaft, Nauka-Verlag, Leipzig, Moskau, 19 edition, 1979.

[41] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 78–, New York, NY, USA, 2004. ACM.

[42] Michael A. Nielsen. Neural networks and deep learning, 2018.

[43] Future of Live Institute. https://futureoflife.org.

[44] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[45] Charles R Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *arXiv preprint arXiv:1706.02413*, 2017.

[46] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016.

[47] Abbas Rashidi, Ioannis Brilakis, and Patricio Vela. Generating absolute-scale point cloud data of built infrastructure scenes using a monocular camera setting. *Journal of Computing in Civil Engineering*, 29:04014089–04014089, 10 2014.

[48] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[49] Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Freeman. Labelme: A database and web-based tool for image annotation. *International Journal of Computer Vision*, 77:157–173, 2007.

[50] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017.

[51] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[52] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[53] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[54] Uwe Stilla. *Topographic Laser Ranging and Scanning: Principles and Processing, Second Edition*. CRC Press, 2018.

[55] Max Tegmark. *Leben 3.0*. Ullstein, September 2018.

[56] Loring W. W. Tu. *An Introduction to Manifolds: Second Edition (Universitext)*. Springer, 2010.

[57] P Vasudev and Ioannis Brilakis. Progressive site modelling with videogrammetry. page 11, 01 2009.

[58] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(1):9, May 2016.

[59] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Amsterdam, 3 edition, 2011.

[60] Shaomei Wu, Jeffrey Wieland, Omid Fari, and Julie Schiller. Automatic alt-text: Computer-generated image descriptions for blind users on a social network service. 2017.

[61] Xiaofan Xu, João Amaro, Sam Caulfield, Gabriel Falcao, and David Moloney. Classify 3d voxel based point-cloud using convolutional neural network on a neural compute stick. 10 2017.

[62] Yusheng Xu, Sebastian Tuttas, Ludwig Hoegner, and Uwe Stilla. Reconstruction of scaffolds from a photogrammetric point cloud of construction sites using a novel 3d local feature descriptor. *Automation in Construction*, 85:76 – 95, 2018.

Eidesstattliche Erklärung

With this statement I declare, that I have independently completed this Master's thesis. The thoughts taken directly or indirectly from external sources are properly marked as such. This thesis was not previously submitted to another academic institution and has also not yet been published.

München, TT. Monat JJJJ

Tim Breu

Tim Breu
[redacted]