A Machine Learning Approach to Indoor Localization Data Mining

Master's Thesis In Technology
University of Turku
Department of Future Technologies
Software Engineering
2020
Samuel Lindqvist

UNIVERSITY OF TURKU
Department of Future Technologies

SAMUEL LINDQVIST: A Machine Learning Approach to Indoor Localization Data Mining

Master's Thesis In Technology, 105 p., 0 app. p.
Software Engineering
November 2020

Indoor positioning systems are increasingly commonplace in various environments and produce large quantities of data. They are used in industrial applications, robotics, asset and employee tracking just to name a few use cases. The growing amount of data and the accelerating progress of machine learning opens up many new possibilities for analyzing this data in ways that were not conceivable or relevant before. This paper introduces connected concepts and implementations to answer question how this data can be utilized. Data gathered in this thesis originates from an indoor positioning system deployed in retail environment, but the discussed methods can be applied generally.

The issue will be approached by first introducing the concept of machine learning and more generally, artificial intelligence, and how they work on a general level. A deeper dive is done to subfields and algorithms that are relevant to the data mining task at hand. Indoor positioning system basics are also shortly discussed to create a base understanding on the realistic capabilities and constraints that these kinds of systems encase.

These methods and previous knowledge from literature are put to test with the freshly gathered data. An algorithm based on existing example from literature was tested and improved upon with the new data. A novel method to cluster and classify movement patterns was introduced, utilizing deep learning to create embedded representations of the trajectories in a more complex learning pipeline. This type of learning is often referred to as deep clustering.

The results are promising and both of the methods produce useful high level representations of the complex dataset that can help a human operator to discern the relevant patterns from raw data and to be used as an input for subsequent supervised and unsupervised learning steps. Several factors related to optimizing the learning pipeline, such as regularization were also researched and the results presented as visualizations. The research found that pipeline consisting of CNN-autoencoder followed by a classic clustering algorithm such as DBSCAN produces useful results in the form of trajectory clusters. Regularization such as L1 regression improves this performance.

The research done in this paper presents useful algorithms for processing raw, noisy localization data from indoor environments that can be used for further implementations in both industrial applications and academia.

Keywords: indoor positioning, machine learning, artificial intelligence, deep clustering

Contents

Acronyms

AI artificial intelligence.

BLE Bluetooth low energy.

CNN convolutional neural network.

DBSCAN density based spatial clustering of applications with noise.

DL deep learning.

HDBSCAN hierarchical density based spatial clustering of applications with noise.

LOS line of sight.

LSTM long-short term memory.

MDS multidimensional scaling.

ML machine learning.

MLP multilayer perceptron.

NLOS no line of sight.

PCA principal component analysis.

ReLu rectified linear unit.

RNN recursive neural network.

RTLS real-time locating system.

T-SNE t-distributed stochastic neighbor embedding.

UWB ultra-wideband.

VAE variational autoencoder.

Glossary

artificial intelligence Computer system which attempts to perform an intellectual task.

autoencoder A neural network which learns a constrained internal representation (called latent representation) of the input data and reconstructs its inputs.

backpropagation Method in which neural network's layers' partial derivatives with respect to loss function are calculated recursively starting from output layer.

black box model An algorithm which is hard to interpret and its inner workings cannot be explained in human understandable manner.

classification A machine learning task in which the output is a single or multiple discrete values of known classes.

clustering Type of unsupervised learning task, in which data points similar to each other are grouped, and often separated from noise.

curse of dimensionality An effect in which high dimensional space makes data points appear sparser as the volume of the space increases fast, causing many numerical methods including machine learning algorithms to perform poorer than in lower dimensions.

data augmentation Techniques for artificially inflating the input dataset with new samples created from existing ones.

data mining The act of discovering patterns from raw data.

decision tree A hierarchical machine learning model which consists of a tree of ”questions” ending with a prediction in the leaf node. They are easy to interpret, which makes them an important algorithm when the inner workings of the model must be understood.

deep learning A subfield of machine learning in which multiple processing layers learn increasingly abstract representations of the input data.

dimensionality reduction The act of removing dimensions (e.g. features) from the data to remove irrelevant data to make some algorithms work better and to avoid noise making learning task harder. Useful also for visualization.

discretization Converting continuous data to a domain in which samples are from pool of discrete values (technically categories).

domain expert Human operator who has deep intellectual understanding of a task's domain (i.e. law) and can transfer this knowledge to computer program through rule sets or labelled data.

expert system Early type of machine learning program, which follows a large catalogue of hardcoded if-then clauses, created by domain experts.

exploratory data analysis A subfield of data analytics in which the data is processed with unsupervised methods to gain insights from it.

feature In machine learning, feature refers to a single property of the input data. Analogous with column in tabular data.

heuristic A practical problem specific method that is often not optimal but works sufficiently well and is often simple.

hierarchical model A clustering model which creates a hierarchy of clusters of increasing or decreasing granularity.

hyperparameter A parameter that controls some aspect of a machine learning algorithm, usually chosen by human. A *parameter* in model is something that is controlled by the algorithm itself.

indoor positioning system Hardware and/or software system which finds the location (and maybe orientation) of a target object in indoor environment.

information loss The loss of patterns in data that could be used to perform the task better, usually resulting from transforming the data into another form.

label In machine learning tasks, especially classification the target ”value” of the sample. For example the class of an image.

latent representation The form of the data inside autoencoder's smallest layer. The condensed representation of the input data in fewer dimensions.

line of sight Direct line between two points unobstructed by anything but air.

localization The act of finding the location of an object. Also sometimes ”locating”.

machine learning Subfield of artificial intelligence, in which a program learns to perform a task using a dataset.

neural network A machine learning algorithm in which layer(s) of *neurons* are interconnected and produce output based on *weigth* and *bias* of the connections from inputs or previous layer.

nondeterministic The output of a method is not certain between the runs given the same inputs.

offline When applied to data systems and machine learning, refers to processing utilizing data gathered prior to current moment, usually in calibration phase of sorts.

online When applied to data systems and machine learning, refers to the fact that processing happens in real time and based on only the current data, in contrast to offline processing.

positioning The act of finding coordinates *and orientation* of target. However, the term is used virtually inseparably from *locating* and not always including the orientation.

regression A machine learning task in which the output a continuous value and not one of pre-existing classes.

regularization The act of restricting the training of a machine learning model by introducing penalty terms or transformations which steer the model to learn better generalizations and prevent overfitting.

representation learning The act of the machine learning algorithm learning patterns from raw data on itself, instead of engineer aggregating the raw data into a more suitable form beforehand i.e. learning the representation automatically, as opposed to manual work.

supervised learning Machine learning task in which representations are learned from existing data-label pairs.

technology stack Collection of technologies used together in a single solution.

tensor A generalization of vectors or scalars, which describes relations between tensors, scalars or vectors. The ”dimensionality” of tensor is called *rank*. Vectors are tensors of first rank.

trajectory Path traversed by an object from point a to point b.

unsupervised learning Machine learning task in which representations are learned automatically from data.

white box model An algorithm which is ”transparent” and easy to comprehend about. Its inner workings can be investigated and reasoned about.

Chapter 1

Introduction

Indoor positioning systems are systems consisting of hardware and software that provide (usually) the real time location of a device or person in an indoor environment. Indoor environment can for example be a store, retirement home, hospital or a sports hall and has properties that make using a common locating system like GPS non-ideal. [1]

Machine learning refers to algorithms and software systems which learn intrinsic patterns from complex datasets, utilizing the vast amounts of cheap computing power that is available today to mine datasets that also are larger and more accessible due to increasing prevalence of sensors and connected devices in our environment. Despite this the history of machine learning is almost as old as the history of computers, the first neural models going back to the 1950s. The recent breakthrough of machine learning has been largely due to the increasing amount of cheap processing power in the form of highly performant CPUs and especially GPUs available to researchers and practitioners. [2]

This thesis looks into utilizing machine learning to mine high level representations from data gathered by indoor positioning systems and the various ways those representations can be utilized. This hasn't been a popular topic in literature despite both indoor positioning and machine learning being popular topics today and new implementations and

products being developed at an increasing pace. There is a significant gap and thus large potential for making new innovations by combining the raw localization data with modern machine learning innovations such as deep learning and other data mining techniques.

The central research question of this thesis is how can we find intrinsic patterns from the kind of data that localization systems make available, how that data has to be processed so the algorithms perform well and what kind of possibilities and insights does this combination provide. The level of research in this thesis is general and not every method will be thoroughly dissected to limit the scope of the thesis. This research will open a large area for future reserach.

The hypothesis is that modern machine learning methods, especially those related to image processing, provide many great high level insights to the raw data as the positional data can be trivially transformed into an image. Image processing is one of the most popular fields of machine learning and has a large catalog of readily available research and algorithms to almost directly apply to the current problem.

Chapter 2 contains an overview of indoor localization as a concept as well as the theoretical basics for such systems. This helps the reader to understand not only how these systems work, but also why they can be challenging to work with as they have their own limitations and considerations. Existing literature on the subject will be briefly enumerated as there isn't much material easily available on the subject.

Machine learning, and more generally artificial intelligence and relevant theory will then be presented in chapters 3, 4 and 5. Several concepts and common algorithms in the field relevant to the research are presented to help the reader to understand the basic principles and how they can utilize the available data.

Chapter 6 contains the experimental portion of this thesis, in which the gathered data with applied preprocessing steps is presented and discussed. The algorithms that were used to experiment with the data are presented as well as their results. The methodology is to first apply methods found in the literature where applicable and to improve them upon for this use case. When experimenting with completely novel implementations the work relies on sources that are closely related to the problem, such as using image processing theory to work on transformed representations of indoor trajectories.

These algorithms only scratch the surface of what is possible by combining the data available in indoor positioning systems with the performance of modern machine learning models and creates a ready foundation upon which many more domain specific systems can be built.

Chapter 2

Background on indoor positioning systems

Indoor positioning system refers to a technical system with the goal of locating objects in indoor environment. These systems are often interchangeably referred to as *indoor positioning systems* or *real-time locating systems (RTLS)* as well, but there is a distinction between locating and positioning something, as *positioning* also includes finding out the *orientation* of the object which is often not done by systems dubbed RTLS. These terms are however used so often to refer to the same thing that this distinction is often meaningless.

This chapter looks into literature on issues relevant to the research problem and provides background on positioning technologies. Central research problem of this thesis is how to utilize raw traces acquired from indoor positioning system to discover higher level features by data mining. The specific scope for the real system discussed in this thesis is retail store environment. However, most of the discussion is relevant to all indoor environments and localization systems as the considerations are similar despite variance in properties of the systems.

There exists a very limited amount of existing literature on the subject of utilizing data from indoor localization systems and at the time of writing none discussing retail environments were found. The research subject is very novel and references to previous research is mostly relating to trajectory clustering and general machine learning technicalities. There exists a large amount of literature on various technical solutions to locating objects in indoor environment, such as those listed in [1] but most of this remains in theoretical space and lab environments. Limited amount of documented deployed applications exists.

This section will introduce the central themes relating to the subject of this research and a high level overview of the technical infrastructure that can be assumed to comprise an RTLS system. The rest of this thesis will handle mostly the resulting raw data on a much higher level but understanding where it comes from also helps to understand its constraints and limitations as well. It also helps discover potential use cases other than the ones mentioned here.

2.1 Overview of published research

One closely related system called *InTraRoute* presented by Prenton et. al. in 2014 [3] was found to be very informative and relevant. Their findings are the best matching source on trajectory clustering algorithms and serves as the technical basis for the implementation covered in section 6.2. *InTraRoute* system was used in the original paper to classify common routes used by hospital employees in indoor environment. The traces were collected using WiFi positioning with mean localization error of 15-30 meters. The system presented in this paper will look into many more variables than just coarse location of the customer and attempt to extract a more abstract representation of the person carrying located device than the *InTraRoute* system, but the work provides a good base understand-

ing for many of the central problems relating to trajectory data mining.

Survey of Indoor Localization Systems and Technologies [1] published in 2019 by Zafari et. al. provided also a valuable higher level overview of various existing implementations in both research and industry as well as an overview of the related low-level technologies. It should be noted that the comparison tables provided in literature have limited applicability in industry applications as they are often theoretical or obtained in controlled laboratory settings which do not provide a realistic basis for comparing the real-world effectiveness of various solutions. A further discussion on choosing the right technology stack is provided later in this chapter.

2.2 Indoor vs. outdoor positioning

Most of the literature regarding route estimation handles routes and traces in outdoor environment. Outdoor position data is usually gathered with GPS and the predicted routes are fixed and known prior, like from freeways down to pedestrian paths. It is easy to see why large scale route estimation is more attractive problem for machine learning practitioners, because the applications are usually economically profitable ones like navigation, traffic control and fleet control all the way to navigating self-driving cars. The data in outdoor environments is abundant and cheap to get, because infrastructures are maintained by public institutions and data is often open source. [3]

In addition to these enonomic aspects, indoor navigation and route estimation is - perhaps surprisingly - a much harder problem stemming from at least three core aspects defined in [3]:

Path-density vs. position accuracy refers to the issue that in indoor environments routes commonly traversed paths exist significantly more densely than in outdoor envi-

ronments, and the positioning accuracy is generally worse. This makes designing reliable and well generalizing systems and algorithms a lot more complex problem. In outdoor environment positioning accuracy might be few meters with GPS and the common routes (roads, for example) separated by tens of meters, whereas in indoor environment routes might be mere meters from each other and the accuracy of positioning varies greatly.

Flat path hierarchy refers to the fact that in indoor environment paths are equal to each other, whereas in outdoor environment paths have a hierarchy from small local paths to highways. Paths high in the hierarchy can often be assumed to be more important, which reduces the total number of routes to consider and other properties, such as speed, can also be used to approximate the path. In an indoor environment there is generally no knowledge of which routes should be preferred, especially since there is often no prior knowledge of the layout of the indoor environment. There is thus no trivial way to prioritize hallway from a sidestep to a toilet, for example.

Cost-effective scalability refers to the issue that indoor positioning systems more often than not require installation of custom hardware on site. This already is a large obstacle for many implementers, but one has to consider future implementations of the system as well and optimize system parameters like cost and accuracy around it, further complicating the high level design of the system.

2.3 Technologies for indoor locating systems

Indoor localization is a wild west of numerous different competing technologies and approaches, all with their own upsides and downsides. There is a lack of common standards and creating such standards would be hard as they could not cover many use-cases, because the demands of different systems vary a lot. When designing an indoor locating system selecting the correct technology stack in an early stage is of utmost importance

because it can make or break the whole product as changes down the line are expensive to make, especially if the solution requires dedicated hardware. One major challenge in such systems is the difficulty of prior evaluation. The environments are extremely diverse and the system must be built to dynamically adapt to the needs of specific use-cases. The specifications of various implementations often list *median localization error* and such measures, but they can be misleading as the reported values are almost always idealized and the performance in some environments can be significantly worse due to local factors. Following enumerates just some of the major questions the engineers have to ask when designing a localization system for indoor use.

1. What is the required locating accuracy and frequency of the business case?
2. How do walls and obstacles affect the signal?
3. What is the average environment in our business case like? Do we need to potentially adapt to other environments with current stack as well?
4. What are the power constraints of the system?
5. What localization algorithm do we use and where is it run?
   (a) Does the algorithm require offline training phase?
   (b) Can the algorithm be run on the edge (i.e. on some/all of the devices)?
6. How does the system interfere with other signals? (e.g. Wi-Fi, Bluetooth on the 2.4GHz band)
7. Are there regulatory constraints for example for signal power or communication bands?
8. Does the system require specialized hardware or can it use existing infrastructure?

These outline just some of the constraints that have to be considered, but by no means cover everything. The purpose is to show that there is currently virtually no ”general-purpose” solution to create an RTLS as each business case has so individual requirements. A change in battery life requirements for example can completely change the technology stack and make some systems impossible to implement.

The main design and engineering problem is obviously the actual locating technology, including the algorithmics, hardware and communication protocols which are discussed later. The business case has a major impact on this, for example an inventory tracking solution has a far less strict frequency and accuracy requirements than robot tracking solution. Generally, the less strict the non-functional requirements for the system are, the cheaper and more scalable it is.

A significant factor for the cost and ease of adoption is whether the system works with existing infrastructure and common protocols, or does it require specialized hardware and separate physical installation. Having to install and calibrate separate hardware on-site quickly accelerates the cost of the system into intractable ranges. For niche cases that are not critical or otherwise do not create much additional value, like proximity based services such as directed advertising, utilizing existing infrastructure such as wireless access points or customers' mobile devices is a good choice.

2.3.1 Communication technologies used for localization

This section iterates some of the most relevant wireless communication technologies in general use today that have good potential for indoor localization solutions.

Wi-Fi is a family of communication technologies generally used to provide wireless network access through *wireless local area network* (WLAN). Hardware supporting these

technologies can be found everywhere today, including but not limited to laptops, mobile devices and remote peripherals such as printers. Access points are also rather ubiquitous in public spaces and especially in retail sites such as shopping malls and establishments such as hospitals. They are used to provide connectivity for guests, employees and devices. Range varies, usually from tens of meters in indoor environment to hundreds of meters with line of sight and capable equipment. Wi-Fi utilizes 2.4GHz and 5GHz ISM bands and it tends to require line of sight between communicating devices and walls and other obstacles can affect the signal strength significantly. For indoor localization purposes this is not ideal, but this is countered by the ready availability of infrastructure which makes Wi-Fi a potent platform for coarser locationing purposes. [1]

ZigBee is a protocol for *low-rate wireless personal area network* LR-WPAN communication for wireless sensor networks or point-to-point communication. It is generally used for purposes such as home automation and other close distance low-powered applications. The specialized hardware, low range, data rate and adoption rate does not make it a favorable protocol for indoor localization.

Bluetooth and especially the newer Bluetooth low energy (BLE) standard are specifications that are widely used in indoor localization as well as for many other purposes. Classic Bluetooth was used mainly to connect personal appliances such as TVs, wireless mouses and alike, but the newer BLE specification aims for low-powered wireless networks with a range similar to ”full” Bluetooth (up to 100 meters) with reduced bandwidth and power usage. It is widely used in wireless devices. Indoor localization implementations using Bluetooth exists in industry as well as large number of literature concerning the matter. It has been widely adopted and existing infrastructure is largely in place, as almost all mobile phones have Bluetooth capabilities by default. One downside of Bluetooth is that it does not perform well if there is no line-of-sight between communicating parties and walls and other obstacles do have a major impact on the signal quality. It

is also prone to interference as it operates in same bands as Wi-Fi. One popular use of Bluetooth is in proximity detection and both Apple and Google have published their own standards for this. [1]

Ultra-wideband (UWB) is a technology for transmitting radio messages using short pulses over large spectrum, exceeding 500MHz in width. It does not interfere with other narrow band transmissions in the same ranges due to its modulation technique, which is based on pulse-timing and various other methods. It has a high bandwidth and lower power usage than Wi-Fi for example. It has good penetration, it is virtually immune to multi-path fading (due to its modulation scheme) and it allows separating direct transmissions from indirect ones with good efficiency. These points give it many ideal properties for indoor use. Traditionally it was used for radar imaging but recently it has found usage in sensor networking and indoor localization. As a downside it does have moderate power consumption compared to BLE and it requires specialized hardware, although recently device manufacturers have started including UWB chips inside consumer mobile devices. UWB based indoor localization exists in the market and alongside BLE it is the most prominent choice when higher accuracy is needed. [1]

2.3.2 Algorithms for localization

When the actual physical infrastructure exists that enables tags to be communicated with or detected remotely, these raw signals have to be converted to their approximate locations. Low signal power, high inherent noise and other adverse effects present in indoor environments makes this algorithmic-wise more complex task than is obvious. In the following the most basic locating methods will be shortly explained. Usually a wide range of heuristics are applied to improve the performance, but these are often not general purpose solutions.

Some implementations require a calibration phase in which data is gathered from the environment. This is referred to as **offline** data gathering, in contrast to **online** phase later in which this previously gathered data is used to aid in running the localization algorithms.

Fingerprinting

Fingerprinting is a method in which usually a large amount of data is gathered from the environment in offline phase with associated manually entered labels. This data is used to train a machine learning model that then later predicts the target label or coordinates of the object based on raw data in online phase. In simpler terms, the model is trained with the data from the environment to learn what different labels ”look like”, usually from RF-data perspective. The system then tries to assign the new data to closest matching class. Most often this is used to implement a localization system with a very coarse level of accuracy. The task is commonly a classification task and the result is for example the room or approximate part of building the device currently resides in. The amount of data required for higher accuracy is large and mistakes in offline phase easily translate into really confusing prediction errors in online phase. The dataset has to be representative of the environment as the models cannot learn about peculiarities such as reflections and obstacles that are not represented in the training data. One benefit of this method is that it does not care about line-of-sight. Multi-path propagation and such noise that normally is an issue can actually be beneficial in classification task as they are inherent properties of the environment that can help distinguishing one class from another. [1]

Fingerprinting can work on its own but an offline calibration phase can also be used in various ways to improve localization done with other methods. One example of this is in [4] where the researchers gathered data from the environment to predict whether a received signal was result of line-of-sight (LOS) or non-line-of-sight (NLOS) propagation to aid traditional trilateration localization.

Trilateration

This technology is most commonly known of being used in *global positioning system* (GPS) but it is applied widely in indoor localization as well, both in 2D and 3D spaces. Trilateration refers to methods utilizing travel time of signals between points, in this case between beacons and tags. The speed of light is a well-known constant and thus when the travel time of a message between two devices is known the distance between them can be calculated. When two distances are known, the position of the tag can be approximated figuratively by drawing two spheres with radii of the calculated distances. In two dimensions, the two resulting intersection points are the candidates for the position of the tag. If the previous position of the device is known the selection between candidates can be made, but often this is not feasible in indoor environments. For this purpose, at least three well-known reference points are needed to produce a single intersection point.

2.4 Utilizing indoor location data

Indoor localization has a plethora of use cases. Some of the most commonly mentioned in literature are healthcare, indoor navigation and industrial environments. More specifically, indoor localization systems could be used for example to locate patients, machinery, customers and many other objects at the sites. This can be used in real-time but the vast amounts of data gathered from the environments can also be utilized for knowledge discovery and optimization purposes after the fact. Path trajectories and their utilization rates could be used to direct services where they are most likely needed, to prevent indoor traffic and improve emergency plans, optimize site layouts and to find general insights with business or other incentives from the subject behavior on the site. Aggregating data from a large history of trajectories helps to discover patterns that would be impossible for any human observer to find by themselves. This is the common theme in data mining.

Retail environment, which is the context of the RTLS in this thesis, has many use cases for data mining customer or object trajectories. This includes for example customer analytics for sales boosting and congestion control. One example would be to optimize the layout of a hypermarket based on customer flow.

Chapter 3

Primer on artificial intelligence and deep learning

Artificial intelligence (AI) is a term often seen in today's tech media, often closely related with other adjacent terms machine learning (ML) and deep learning (DL). To make sense of this paper's topic it is important to give some context to these terms and explain how they relate to each other. What makes these terms so hard is the fact that their definitions are very fluid and it is hard to pinpoint what exactly falls under the terms because that depends on the context, who we are speaking to and even the time period. Artificial intelligence is probably the hardest of these terms, because it can be defined for example as *an effort to automate intellectual tasks normally performed by humans*. This can be used to define almost any automated task as artificial intelligence, regardless of how simple it is. Nowadays in the age of deep learning and self-driving cars we don't usually refer to long lists of if-else clauses as artificial intelligence, but that's what it referred to in the 80's, more generally known as *symbolic AI*. Nowadays in technological context artificial intelligence refers usually to systems utilizing machine learning, but it is important to understand that it does not necessarily need to be the case. Machine learning is a subclass of methods under the definition of artificial intelligence and deep learning falls under the

definition of machine learning. These important topics will be discussed in more detail in the following chapters.

3.1 Machine learning

> A computer program is said to learn from experience *E* with respect to some task *T* and some performance measure *P*, if its performance on T, as measured by P, improves with experience *E*.
>
> — Tom Mitchell, in *Machine Learning* (1997) [5]

The previous quote formalizes one description of machine learning quite nicely. Artificial intelligence refers to machines (computers) doing intelligent tasks usually performed by humans. Machine learning is a subset of artificial intelligence, referring to methods and algorithms which *learn from data*, as opposed to having the rules manually programmed by a domain expert, as was done during the height of the *expert systems* in the 80s. ML algorithms find the important patterns in the data by themselves, often outperforming manually written algorithms in simple problems and enabling solutions to complex problems like computer vision. [6]

The fundamental idea of machine learning algorithms is that they need just data and arguments called *hyperparameters*. They do not need to be directed by a human expert, but their learning can be optimized by tuning the input data and the previously mentioned hyperparameters.

There exists a plethora of different algorithms that can learn from data by themselves, the most important nowadays being the *neural network* which will be described in more detail later in the thesis. Different algorithms are good for different kinds of tasks and environments, there is no silver bullet, but *neural network*, especially combined with a

technique called *deep learning* is especially relevant in modern implementations for *perceptive* tasks, such as image processing and recognition.

In the following are described the most common classes of machine learning systems and machine learning tasks relating to them.

Supervised learning refers to types of machine learning tasks in which the system learns from pairs of feature-label points for which the correct output is known. A classic example would be learning to categorize the type of animal in a picture given a large set of animal pictures with the correct description of animal present in the image. This type of learning is generally divided into two types of tasks, classification and regression. In *classification*, there exists a finite number of discrete classes, for which the correct category is predicted. *Regression* on the other hand refers to predicting a continuous value from the solution space. This would be for example predicting the exact price of a house. One can see example of this in table 3.1 in which *LABEL* column is the known target value of that row. [6]

CRIM	ZN	CHAS	AGE	TAX	PTRATIO	**LABEL**
0.00632	18.0	0.0	65.2	296.0	15.3	24.0
0.02731	0.0	0.0	78.9	242.0	17.8	21.6
0.02729	0.0	0.0	61.1	242.0	17.8	34.7
0.03237	0.0	0.0	45.8	222.0	18.7	33.4
0.06905	0.0	0.0	54.2	222.0	18.7	36.2
0.02985	0.0	0.0	58.7	222.0	18.7	28.7
0.08829	12.5	0.0	66.6	311.0	15.2	22.9

Table 3.1: A small labeled sample from the widely used *Boston Housing* dataset

Unsupervised learning refers to type of learning in which there are no labels available, only raw data from which the system tries to learn on its own. Classification and regression are not possible because there is no prior information available, the algorithms are rather used to learn new insights of the input data. *Clustering* is a task in which the algorithm attempts to find groups of similar points from the data and is the most common form of unsupervised learning. *Dimensionality reduction* means lowering the complexity of the input data without losing too much information. This is done by automatically learning what is informative in data and removing irrelevant noise. [6]

In most scenarios, indoor positioning data is not labeled so unsupervised learning is the subtype of learning that has the largest focus in this thesis.

Semi-supervised learning is a combination of supervised and unsupervised learning. In this kind of learning, the system emerges with patterns and insights on the input data in an unsupervised manner but requires the help of a human director to continue learning. It generally refers to kinds of machine learning tasks in which some input from external operator is required to ”kickstart” or steer the learning [6]. One relevant example would be to tag part of the indoor paths produced by a RTLS and use that to steer the training process.

3.2 Deep learning

> Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction [2]

The preceding quote summarizes the core idea of what deep learning is quite well. The definition is used very liberally especially as it is such a trendy topic in artificial intelligence, but the central ideas are quite intuitive and easy to understand. Deep learning refers

to type of machine learning which consist of multiple layers of processing that each takes as an input the output of the previous layer. The input to the first layer is the raw data, and each layer extracts more and more abstract patterns from it. Thus the stack of layers learns representations of increasing abstraction, from simple motifs to high level features that can be used e.g., for classification or regression [2]. Some alternative names for deep learning could be *layered representations learning* or *hierarchical representations learning* [7].

Classical example of getting a better grasp of how this works is to look at the type of learning problem deep learning has excelled in recent years: image processing. The input to first layer of processing is the raw image data and the first layers in the model extract simpler representations of patterns in the data. This information is usually the different kind of edges that appear in the image. From these simpler edges more and more complex motifs are constructed in later layers, eventually becoming abstractions of actual objects in the image such as an eye, an elephant, a car and so on [2]. This is a simplification of what the networks might learn, but in reality the features might be more abstract, complex and harder to comprehend. It is however a good way to understand the idea of deep learning.

The type of learning in which the representations present in data are learned by the model itself instead of constructed by hand with the help of domain expertise is called *representation learning*. It is one of the reasons deep learning is such a powerful breed of artificial intelligence methods and which separates it from traditional machine learning models, often referred to as *shallow learning* [7].

The type of machine learning model almost always associated with deep learning is the previously discussed neural network, but deep learning as a subfield of machine learning

methods does not include a notion of any specific algorithm. The ”layers of learning” in the model can consist of any type of learners, but neural networks as an inherently layered model, in which all of the layers can be simultaneously optimized using backpropagation, is an exceptionally good fit for deep learning.

Deep learning is a very effective method and the recent results and technological advancements are impressive, but it should be kept in mind that deep learning is not a silver bullet that is applicable to every problem. One key requirement for training deep neural networks is a large amount of applicable data, which is not something that is always available. Deep learning is fit for problems in which the solution is constructed by stacking up layers of increasingly higher levels of abstraction, ending with representation that is fit for solving the research problem. Most common applications of deep learning under this definition are perceptive problems like computer vision, audio processing and natural language processing. Sometimes a smaller MLP works, but even these require relatively large amount of data. If one is faced with a relatively simple problem with smallish amount of data, traditional shallow algorithms like logistic regression, support vector machines and alike might be worth of consideration. They should be tried anyways as a sanity check, as if neural network cannot beat the ”reference line” set by a trivial method, one should aim for a simpler solution. [7]

3.3 Deep learning for sequence processing

Sequential data is any data in which the ordering of the data points is important. This could be for example outdoor temperature during the day or the value of a market asset. To gain insights from the dataset and to train machine learning models to predict the value of time-dependent variable in the future, it is important to include the notion of time in the data. This has traditionally been a challenging task for machine learning algorithms,

as most of the shallow learning methods don't have any obvious way to include ordering to data. One method is to aggregate data over certain timespan into singlular scalar values, like mean, standard deviantion or absolute change over given time. Even traditional feed-forward multilayer perceptrons (simple neural networks) are not good at this task, as they neither have a notion of time embedded in their architecture. Flattening a time series into a single input vector also is a very brittle method for more or less the same reasons as learning local patterns in images doesn't work well with feed-forward networks. Network can learn to find a pattern at certain point of time or a motif in a given point in input image, but it does not recognize it when it appears in unknown context. For image processing convolutional neural networks are obvious choice over simple networks and for sequence processing there exists a type of network than excels in handling time-series data: recurrent neural network. Sequence processing is however not a relevant subject for this thesis' research problem so it will not be further discussed here. The location data does have a notion of time in it, but it is handled in different way by embedding this information in the input tensor.

Chapter 4

Important algorithms and subtypes of machine learning

The following chapter goes over some of the major machine learning algorithms that are thought relevant for the thesis' research. However, the field of machine learning is so wide and constantly evolving that all popular algorithms or families of algorithms cannot be enumerated here. The chapter is limited to introduce some of the major algorithms especially related to *clustering*.

4.1 Neural networks

Neural networks are a very old and very popular category of machine learning algorithms. Implementations have been around since the introduction of the *perceptron* algorithm in 1958, but have been limited by the lack of large amounts of computing power and training data required by the algorithm. Neural networks were largely overshadowed by other algorithms in state-of-the-art implementations until recent times. [7]

A neural network is a directed graph of neurons which are organized in layers. The network starts from an input layer, which receives the input data row by row or in small

batches, goes through varying amount of hidden layers and ends up in output layer whose structure is directed by the problem definition. A regression task ends with one output neuron, producing a continuous value, or a statistical distribution of probabilities for different classes in the case of classification. A wide variety of different tasks aside from the aforementioned can be achieved with neural networks and there exists a wide variety of different architectures and topologies many of which will be discussed in this and the upcoming chapters.

So how does a neural network work on a low level? A single neuron is a unit of processing which receives as its input the outputs of the previous layer weighted by a tensor called *weights* of the neuron and optionally summed with a *bias*. The output of a single neuron is the weighted sum of the input values passed through a nonlinearity called *activation function*. A scalar called *bias* is optionally added to the value before passing it to activation function. Together with an actication function like ReLu, which sets values below a given treshold to 0, this can be used to inhibit neurons with small values from affecting the output and thus helping to avoid overfitting.

Output of a layer in multilayer perceptron (MLP) is described by the following formula, where f_a is the activation function like rectified linear unit (ReLu), W is the weight matrix and b is the bias scalar. *Dot* refers to tensor product of weight matrix and the input tensor, multiplying input vectors by each neuron's weight vector.

$$Output = f_a(dot(W, input) + b)$$

The purpose of the activation function is to introduce a nonlinearity into the process. Multiple layers each performing some nonlinear transformation on the data is what enables

the network to extract the complex non-linear patterns in the dataset. Without activation functions the entire network would reduce into a large linear transformation and the beneficial properties of neural networks would go largely unused. There exists a large catalogue of activation functions used in various cases, but a few have proven to be the ”strong defaults” used by almost all modern networks. [7]

By far the most popular activation function used in hidden layers today is *rectified linear unit (ReLu)*. It is formulated in the following.

$$f(x) = max(0, x)$$

As can be seen it is extremely simple and computationally efficient. It is simply the positive part of the argument. ReLu is the most commonly used activation function, but some others such as *tanh* and *leaky ReLu* are seen, the latter being a variation of ReLu which doesn't drop to exactly zero, avoiding issues such as dead nodes. [8]

Selecting activation function for hidden layers is, like many other issues in machine learning, more of an art than a science. This is however not the case for selecting the final layer's activation. It is dictated solely by the problem definition and there exists well known rules on how it should be selected. In regression tasks, activation function is generally omitted (a single node in the final layer outputs the continuous value), in multi-class classification it is *softmax* and in binary classification the *sigmoid* function. [7]

A very simple neural network might consist only of a single layer of neurons, but more commonly there are multiple *hidden layers* (layers that are not inputs or outputs) following each other. A simplified structure of such multi-layered network, referred to as multilayer perceptron (MLP) is shown in figure 4.1.

Training a neural network (i.e. tweaking the weights and biases so the predictions generalize well on unseen data) works generally as follows:

1. Weights are initialized to small positive values, biases usually to zero.
2. A forward pass (output of network given inputs) is calculated either using
   (a) A single sample (stochastic gradient descent)
   (b) A group of samples (mini-batch gradient descent)
   (c) Entire training data (batch gradient descent)
3. Predictions are compared to expected values and average *loss* is calculated using the selected function.
4. The *gradient*, which is a collection of partial derivatives of the network's free parameters relative to loss value is calculated.
5. *Optimizer* algorithm updates the free parameters to lower the loss value of previous forward pass. It can be simply applying the gradients multiplied with *learning rate* but it can also apply additional dampening features.
6. The process continues from point 2 until a termination condition is met.

The previous points give a very high level overview of the process which has many parameters to choose and hyperparameters to tweak. Loss function, optimizer, stopping conditions and performance metrics are left for the engineer to choose and can affect both the training of the network as well as performance of the final model. Loss function is often very simple to choose, as different problem definitions have usually only a handful of metrics to choose from and usually one or two ”industry standards” [7]. For example binary classification utilizes most often *binary crossentropy* as its loss function. Other properties of the training process and network architecture are often harder to choose and optimize properly. Often creating the model devolves into trying everything and seeing

what works best, although a good understanding of the problem and relevant algorithms helps to choose a good first guess.

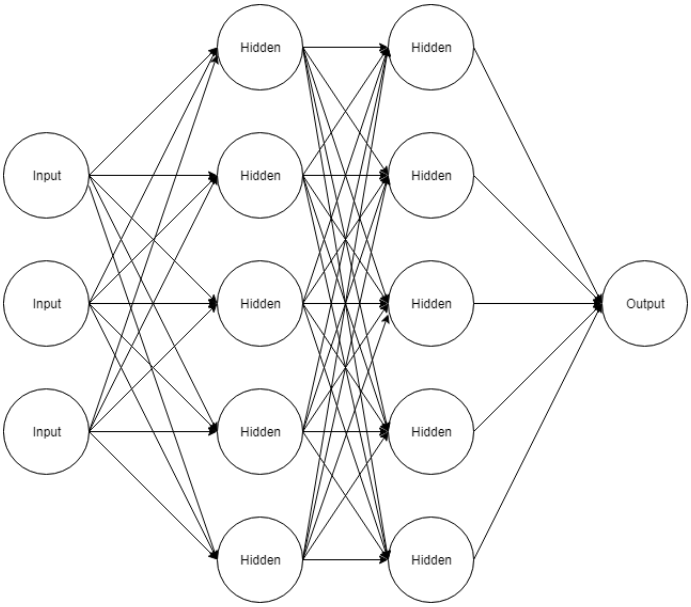


Figure 4.1: Simplified visualization of a fully connected MLP network with three input neurons (three features), two hidden layers with five hidden units each and a single output neuron. This could be a network for regression task with a single continuous output value.

4.2 Convolutional neural networks

Convolutional neural networks are a central neural architecture that was introduced in early 90's, but has only risen in popularity during the last decade after it demonstrated exceptional performance in image processing and other computer vision tasks. Several efficient implementations running on GPU had been developed during the 2000's, but convolutional neural networks became more widely known after publishing of AlexNet in 2012 in the paper "ImageNet Classification with Deep Convolutional Neural Networks" [9]. The network demonstrated high performance in ImageNet classification task. Currently CNNs are the dominating architecture in computer vision tasks, but it performs well in many other classes of tasks, such as sequence processing [7].

Convolutional neural networks are formed of stages, or layers, as any other deep neural network architecture. These layers extract each a representation on a higher level of abstraction from the outputs of the previous layer. Convolutional layer contains a set of filters called a *filter bank*. Each of these filters is an N-rank tensor of values that each are trained to look for certain features in the input data. The input data, which is a tensor of rank one or higher is fed into the layer. This data is processed with each of the filters separately using convolution operation, creating a new output channel for each filter.

Figuratively, thinking CNN operating on image data, each filter in a layer is a weight matrix which is multiplied with the input matrix in many different positions. The filter, which has smaller dimensionality than the input image, is multiplied with the image at every position, moving one or more pixels at a time. The filter output is a tensor with values of larger magnitude in positions that matched the filter's significant values (i.e. matched the image feature the filter was looking for). This process is clarified in equation 4.1 in which the first matrix is the input, the second the filter weights and the third the output of the convolution. This result is passed through a nonlinear function and processed the same as in other types of neural networks. The output tensor is by default smaller than the input, but this can be mitigated by adding *padding* around the input so that the size stays constant in the network. [2]

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 2 & 1 \\ 1 & 4 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \end{bmatrix} \tag{4.1}$$

Thus, if a layer has, say, 32 filters, it will output 32 channels, each of which represents the presence of a certain feature in the input data. Understanding CNN becomes more concrete when we think the operation as looking for local features in the image. Each layer of the network represents different layer of abstraction. One can think of an image processing CNN as starting with small local features like edges, then looking for the combinations of those motifs, gradually leading up to more abstract motifs like ears, eyes or letters. The final level of abstraction is usually on the level of entire objects, like humans, dogs and so on [7]. The convolutional part of a network usually consists of repeating sets of convolutional layers and pooling layers. These pooling layers perform a sort of dimensionality reduction operation, using some operation like mean of spatially local groups of given size to output a tensor of reduced size, in which the values represent the general values of a given area, on a higher level. This is useful because the convolution operation does not reduce the size of the input tensor, but increases the number of channels and keeps the other dimensions the same (there are exceptions to this). As succeeding convolutional layers represent higher levels of abstraction which also cover larger areas in the original data, it makes sense to reduce the size of the representation. An elephant is (usually) present in far larger areas in input image than a single pixel, and an economic recession is present in large timespans of stock market data, not just a single value. They are general repeating themes, or motifs that the convolutional neural network is exceptionally good at finding.

To put it in a nutshell, convolutional neural networks are good at finding repeating local patterns in input data, and construct and learn higher levels of representation from combinations of these. The gist of using convolutional kernels is that they find the same pattern in any spot in the input data. This is the thing that makes them far superior to

traditional deep learning models which receive flattened input data and learn meaningful values present in certain neurons. When CNN learns some pattern, it can find it in other locations as well, even if it had not seen it there before. Convolutional layers are translation invariant (or by derivation - time invariant in sequence processing tasks), however other operations used in CNN such as pooling layers are not to the same degree. Relative invariance to translations is what makes it the model of choice for signal processing tasks, like computer vision, audio processing and sequence processing. [2]

4.3 Decision Trees

Large neural networks dominate today's machine learning landscape. In applications such as machine vision and natural language processing the networks can grow huge, having millions or even billions of parameters. It is a rather intractable task to decode the inner workings of such networks, even simpler networks usually learn such complex patterns and relationships that translating the decision making processes to human audience is virtually impossible. However, there exists numerous technical and non-technical reasons for understanding the decision making processes of the models, which we will iterate later in this thesis. There exists various methods of interpreting neural networks and other complex models such as gradient boosters which we will also touch later, but there exists also many simpler models that are interpretable by themselves. One such model is called a *decision tree* which we will overview in this section. Models which are not interpretable are generally referred to as *black box* models. Such models include neural networks and random forests to name a few. Interpretable models, i.e. models whose inner workings are comprehensible to humans are called *white box* models. This classification includes for example linear regression and decision trees.

Decision trees are a series of connected nodes, starting from a single *root node* and ending

in a leaf node which contains the predicted output. A node ”asks” a single question about a feature in the input row. A *binary tree* contains two connected nodes for non-leaf nodes and thus the answer has a boolean value. Trees with multiple answers per node exists but are not as commonly used, because popular algorithms such as CART (Classification and regression tree) produce binary trees and multiple answers can be split into multiple binary questions. The tree is followed into the next node based on the answer and the chain of questions continues until a leaf node is reached. The value of the prediction is then the most probable class in that node. [6]

The ”goodness” of the split is evaluated by some metric to find the optimal split in the current subset of training instances. Two of the most common metrics are *gini impurity* and *entropy impurity* of the split [6]. These measure the *impurity* of the resulting subsets, a pure subset containing mostly instances of one class. The algorithm thus targets to find splits that result in a good split into different classes. One downside of decision trees are the they are *greedy* algorithms i.e. they only consider optimizing the current split and don’t care how it effects the future splits. There exists various variations of the algorithms to train and optimize the trees. Some, such as the popular CART are stochastic which means that every run might not result in similar tree. This can be mitigated by running the algorithm many times and taking the mode of the results.

If training is left unconstrained the decision tree will overfit the data and result in leaf nodes containing only a single sample in them and perfect accuracy in training data. This will obviously scale really badly and decision trees are either *pruned* afterwards or *regularized* while training. Pruning means removing the leaf nodes using a set of rules such as the statistical significance of the information gain [6] until the tree matches the rules. Popular constraints for regularization while training include setting the *max depth* of the tree or *max samples per leaf node* to stop the training when constraint boundaries are met.

Another downside of decision trees is that the decisions boundaries they make are very orthogonal in the feature space, meaning that the don't perform well if the information is laid in a rotated manifold. Decision trees generally don't need any kind of pre-processing but in this case applying manifold learning can help to unroll these complex manifolds. This however does lead into reduced interpretability and in such case other model might be a better choice. [6]

The main advantage of decision trees is obviously their simple interpretation. The models can be laid out as a chain of simple rules that can be easily followed by a human observer and thus decision trees can be used as data mining tool to find simplified relationships between inputs and labels. This is later important when *distillation* is introduced. Inference of decision trees is also extremely fast and scales well as the chain of rules can be reduced into a chain of *if-clauses* in code. Decision trees can be thus used to create an optimized version of a more complex model for use on constrained environments, such as IoT devices or mobile applications.

Decision trees as well as many other classical machine learning models cannot compete with deep neural networks in representation learning in extremely complex problems such as audio processing or machine vision but they do still have their own use cases. Gaining insights to the predictions by creating white box models is one and the relevant one for the purposes of this thesis. Another case is when the training dataset is very small. Many real world problems do not have a large enough dataset to allow training complex models without overfitting and data augmentation may not be a feasible solution. In this case simpler model might do just fine as they require vastly fewer training instances to find patterns that generalize well.

An example of constrained decision tree can be seen in figure 4.2 in which a decision tree with *max depth* of 3 is fitted on popular *Boston Housing Data* dataset. Limiting the tree to be this small reduces the predictive power of the model but it makes the main patterns legible for human readers (for example, high crime rate (CRIM) and air pollution (NOX) correlate with low housing prices). Decision trees can be utilized in this way to *interpret* and *explain* more complex models.

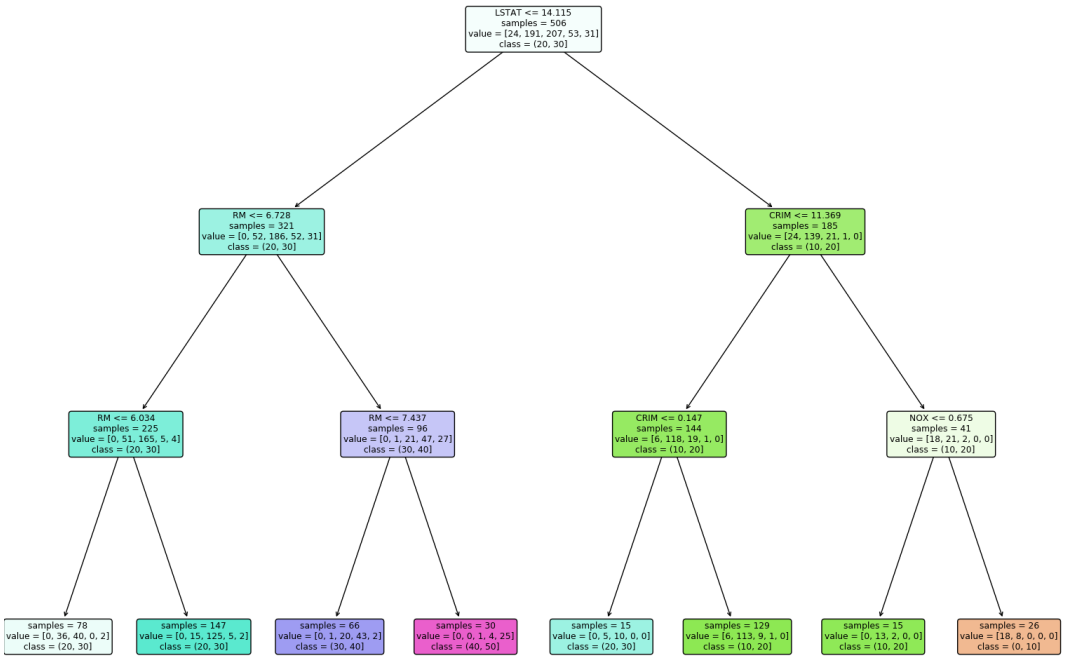


Figure 4.2: Decision tree

4.4 Clustering

4.4.1 General on clustering

Clustering is a central task in exploratory data mining in which points are grouped together using some *distance metric*. A cluster should consist of data points that are similar to each other, and specifically more similar to each other than points in other clusters or outside any cluster (noise) [10]. It has a myriad of use cases, but most common are knowledge discovery in high-dimensional space, or visualization in lower dimensions, both of

which will be very relevant later in this thesis.

Clustering has a lot of practical applications in industry and science and has thus been an important research topic for a long time. The introduction of deep learning has also given boost to many classical clustering algorithms (for example by incorporating them as part of more complex model) as well as introduced many new ones by utilizing the representational power of neural networks [11]. Classical examples of clustering would be classifying images by their categories, novels by their genre, customers and even images of tissue samples. Clustering provides an invaluable aid in finding patterns in seemingly chaotic and high dimensional data and can do this either automatically or aid a domain expert in exploring the data visually.

The cases in which clustering is utilized can be very different. In some cases the domain is well known and even the base truth (the expected number of clusters) is known prior, which has a major influence on the choice of algorithm and many other aspects as well. In many cases however, the underlying groups and patterns are mostly unknown as is the case in this thesis. The uncertainty about the results also makes evaluating the clusters almost as hard as the clustering itself. The act of exploring the data to gain insights from it is referred to as *exploratory data analysis* and is mostly done with unsupervised algorithms.

The choice of distance metric is vital for clustering task. This metric is usually chosen in a task-specific manner and must be a good fit to measure distances in the task's domain. A classic example of distance metric would be euclidean distance which is often used in spatial space. Less common tasks usually require some hand-crafted distance metric.

Many clustering algorithms make strong assumptions about the data (like Gaussian distri-

bution) and are sensitive to noise and variance from these assumptions. Clustering often suffers from the complexity of the data and phenomena such as *curse of dimensionality* which refers to data appearing sparser in larger space [6]. Significant preprocessing is often needed, such as learning better representations from raw data before the data is input into clustering algorithm. This will be discussed in more detail later alongside *deep clustering*.

Clustering algorithms can be divided using many characteristics. Examples would be the parameters they require, the assumptions about the data they make or if they allow noise in the data or not. A good classification measure is the type of the *cluster model*, i.e. which kind of properties does a notion of ”cluster” have in some specific algorithm. The following describes briefly some of the most common cluster model families and some representative algorithms from them. These partitions are not universal and often have different names assigned to them.

4.4.2 Connectivity based models

Models of this class are also often called *hierarchical models*. They build a hierarchy of clusters of increasing or lowering densities. The largest cluster usually includes every data point in the data set and smallest data points consist of a single individual. If the clustering starts from one single cluster and starts breaking it down into smaller clusters it is called *divisive*. If it starts from every data point being its own cluster it is called *agglomerative*. Generally hierarchical clustering methods are slow even though there exists many implementations that optimize the process. [12]

Agglomerative hierarchical clustering is a common type of connectivity based clustering algorithm. It starts with every data point being its own cluster. Distance between every cluster is calculated for example using the distance between closest points or the

average distance between every point and two closest clusters are combined at every step until all clusters have merged together. This algorithm requires the user to select the number of clusters manually, for example through visual inspection of all possible stages. The process is usually visualized using a *dendrogram* as demonstrated in figure 4.3.

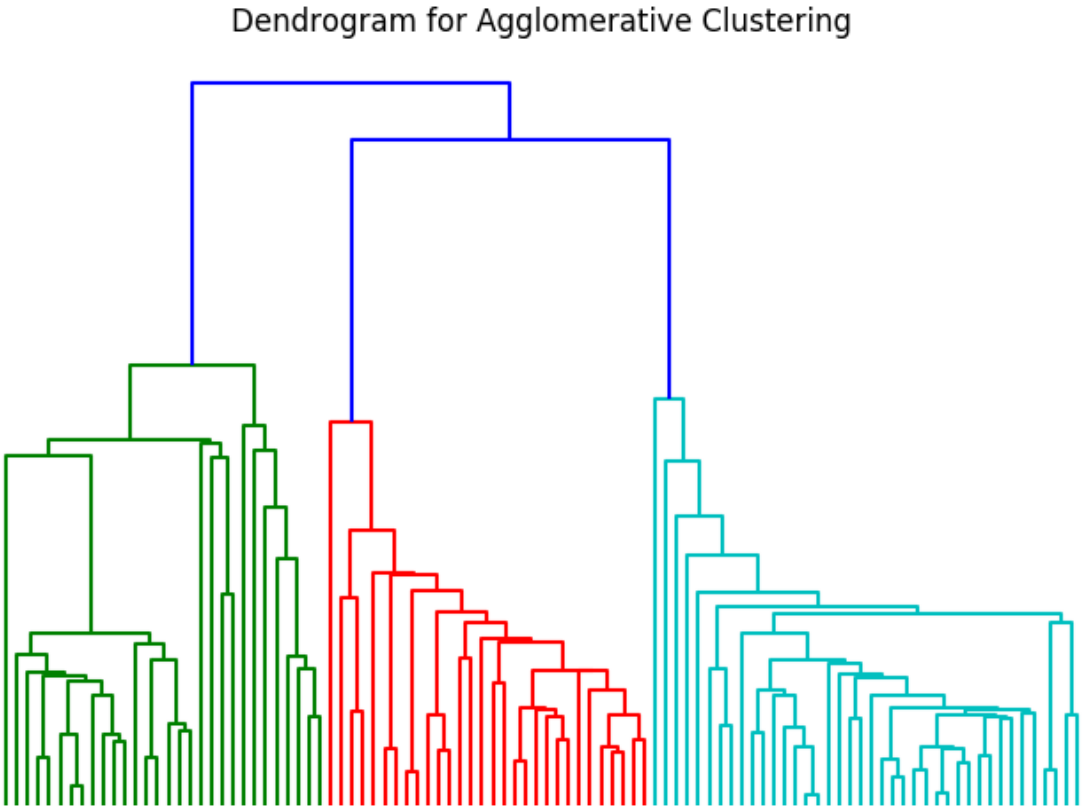


Figure 4.3: Dendrogram resulting from agglomerative clustering. One can observe the different clusters as dense regions of connectivity. How ”low” the lines connect represents how close they are to each other.

4.4.3 Distribution based models

These models usually make some general assumptions about the statistical structure of clusters and attempt to find parameters for distributions which generated the clusters. They are often sensitive to noise.

Gaussian mixture model makes the assumption that every cluster in the data was generated by a gaussian distribution and attempts to find the parameters (mean, standard deviation) for the distributions using *expectation maximization algorithm*. It has the advantage that clusters can have varying shapes and properties (different densities etc.), but the central assumption about the Gaussian distributions is very restrictive one and often does not work in real world scenarios. Most implementations also require the selection of cluster count manually. [10]

Figure 4.4 displays a randomly generated dataset with GMM applied to it.

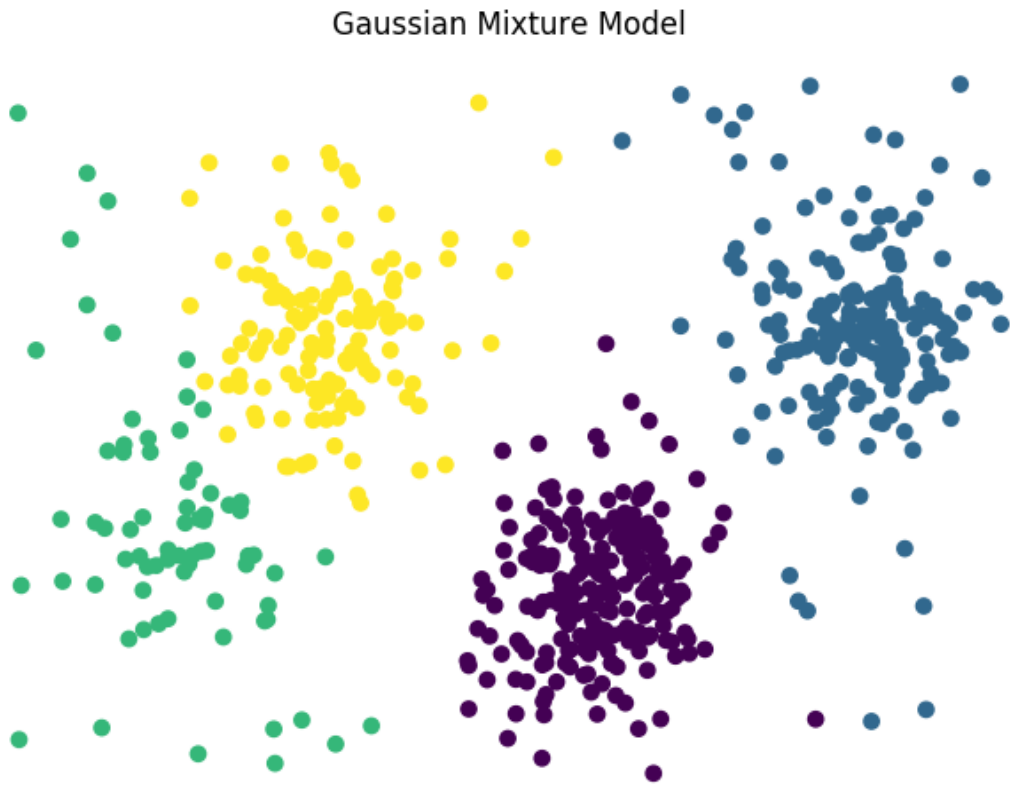


Figure 4.4: Gaussian mixture model fit on artificial dataset with four clusters and noise. The number of clusters had to be provided manually. Also, every data point is included in some cluster. The clusters were spherical which is an ideal condition for GMM (the distributions are gaussian).

4.4.4 Centroid based models

This is a very popular family of clustering algorithms based on the concept of *central vector* defining the cluster. Points are grouped together in the cluster whose centroid is closest to them by some metric. They are computationally very efficient and simple which makes them popular when handling large amounts of data. They also scale well with the amount of data and can be distributed with relative ease as clustering is affected only by the neighborhood of the data point.

K-means clustering is one of the most popular examples from this category. A fixed predetermined number of cluster centers are positioned randomly in the data space and all data points are grouped under the closest centroid. Cluster centers are then moved to the mean of the points assigned to it and process repeated iteratively until the centroids stop moving and thus termination condition is met. Different runs can result in different groupings which makes the method *nondeterministic*. This can be mitigated by running the algorithm multiple times and selecting the most common result (i.e. mode). K-means splits the data space into *Voronoi diagram*, classifying every data point to some group, which means this algorithm is not a good fit when data contains noise. The number of clusters has to be also known in prior, which restricts the use of this method in many applications, including this thesis' problem. It is however extremely simple which is why it's usually the first clustering algorithm people are taught about. [10]

4.4.5 Density based models

This class includes many of the most popular algorithms used in real-world applications. They are based on the assumption that clusters are defined by changes in densities in the data space, clusters being areas of similar higher density while the lower density data outside clusters is noise. Real world data is often very noisy which makes these models very effective in practical applications. Density based clustering algorithms also have

other good properties that often make them a prefereable choice in non-toy applications, like the previously mentioned fact that they are robust to noise, and even more importantly, they do not require the user to define the number of clusters in prior.

Mean-shift clustering is a hill-climbing algorithm in which randomly selected cluster centroids *shift* towards higher densities of data, resembling K-means clustering in many ways. Instead of shifting towards the *mean* of the group's points the centroid shifts towards direction of *higher density*. This algorithm is non-parametric and does not require user to define the number of clusters as it includes a processing step which removes "worse" (less dense) candidates for clusters. [13]

DBSCAN which stands for *density based spatial clustering of applications with noise* is perhaps most commonly used clustering algorithm in general and the most relevant algorithm in the clustering tasks of this paper. It has two parameters, ϵ (epsilon) and *minpoints* which control the clustering process. It works by starting from a random unvisited point and finds the neighborhood points using some distance metric. If there are at least *minpoints* unlabeled neighbors in distance equal or less than ϵ the point and its neighbors start a new cluster. When the cluster does not grow anymore a new unvisited point is selected. This process continues until all points have been visited. Points which were not included in any cluster are labeled as noise. This algorithm works very well in many scenarios, but choosing the parameters can be hard and must be done carefully. The clustering might also not work ideally if the clusters have different densities, but there exists many optimized derivative implementations like *OPTICS* [14] that assess these issues. Figure 4.5 shows a fake dataset similar to figure 4.4 processed with DBSCAN. The main difference is that sparse points outside clusters are labeled as noise, unlike some other algorithms which put every single point under some cluster. [13]

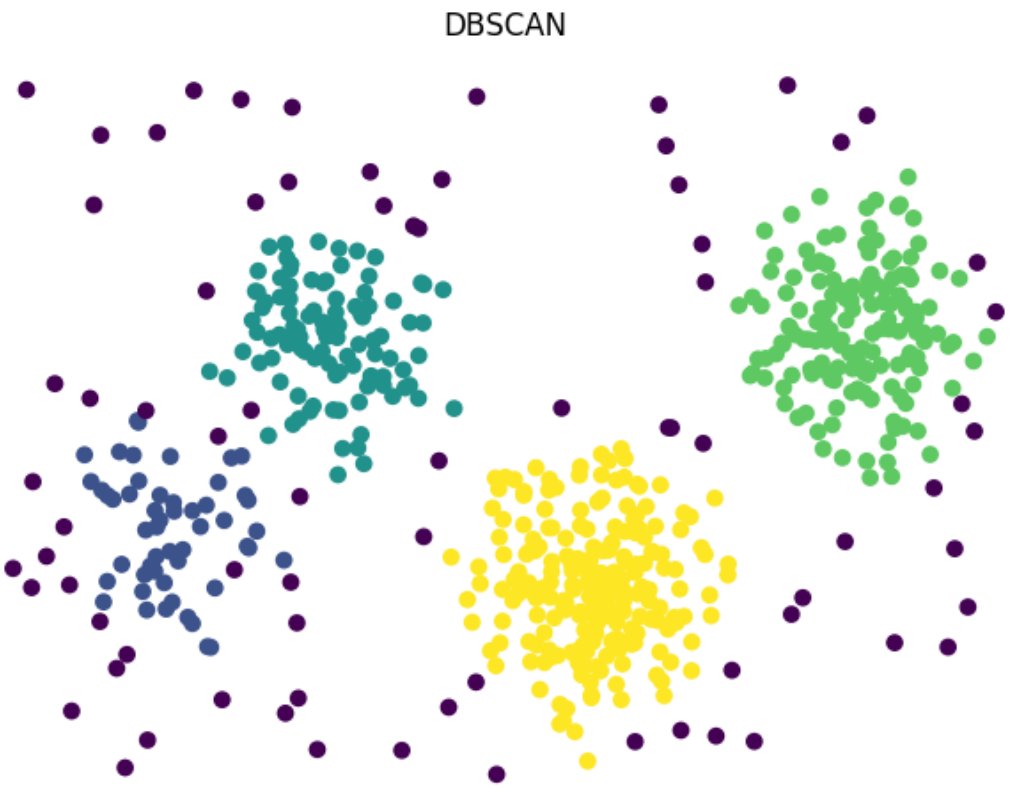


Figure 4.5: DBSCAN run on artificial dataset with noise. With manually adjusting the parameters the algorithm can differentiate clusters from noise (black color).

Hierarchical Density-Based Spatial Clustering of Applications with Noise or HDBSCAN in short is a more advanced algorithm derived from DBSCAN. HDBSCAN extends DBSCAN into a hierarchical clustering algorithm. It performs DBSCAN with varying ϵ values and then uses hierarchical clustering to find the most stable clustering. One of its strengths is that it can handle varying density clusters which is something DBSCAN falls short in. Secondly, it gets rid of the aforementioned epsilon parameter which is central to DBSCAN and retains *minimum cluster size* as its only required parameter. It is thus easier to use and more robust to varying densities of data, but also more computationally demanding than its parent algorithm. It is also harder to tweak if it does not produce satisfying results, due to the lack of hyperparameters. Figure 4.6 shows HDBSCAN applied to a fake dataset, demonstrating its ability to find clusters of different densities. The cluster

in top left has lower density than the two other large clusters, but it is still found with high fidelity. A fourth cluster was created from noise in the bottom left, but this could be fixed by increasing the cluster minimum size hyperparameter. [15]

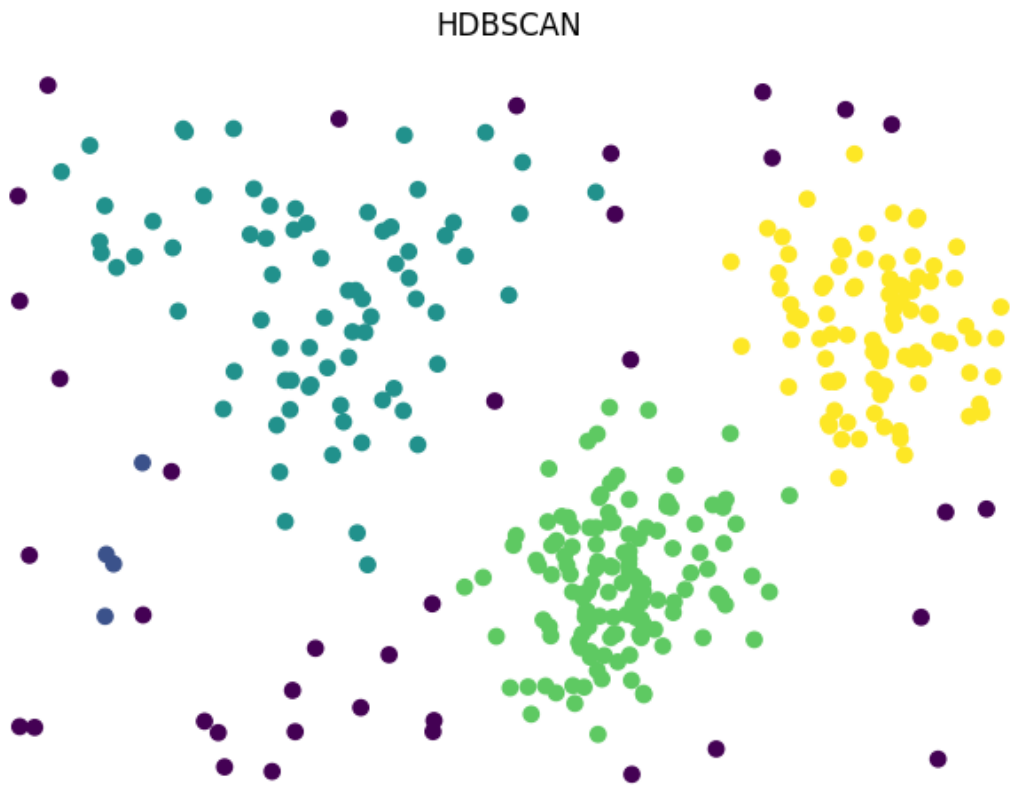


Figure 4.6: HDBSCAN run on similar dataset as in previous figure. The clusters have different densities but the algorithm still finds three clusters even with the noise (dark purple color).

4.5 Deep autoencoder

Autoencoders are a type of neural network that learn an identity mapping of the inputs. This means that the input and correct output are same and the training of the network means minimizing the *reconstruction error*. This error is usually the l_2-norm between the output and input, although different loss function might be used based on the task. *Deep* in deep autoencoder refers networks which have more than a single hidden layer

and is not a well defined term. The same prefix is also used for different types of neural networks with multiple hidden layers. [16]

Identity mapping is a trivial task, but autoencoders introduce strong *regularization* in the form of dimensionality reduction by using narrower layers in the middle of the network. Autoencoder consists of two parts: encoder and decoder. Encoder consists of hidden layers with decreasing number of hidden units, which causes the representations to be compressed into a low-dimensional representation. The dimensionality of this encoded representation can be controlled by choosing the number of hidden units to use in the encoded layer. The second part of the network is called the decoder and does the opposite action to encoder, reconstructing the original input from the encoded representation. The sub-networks are usually identical, mirrored copies of each other but this does not necessarily need to be the case. Decoder architecture can be drastically different from the encoder as long as it is able reconstruct the original input. An example of autoencoder network is seen in figure 4.7. [16]

Autoencoder can be expressed with the following notation:

$$\overline{X} = D(E(X))$$

where $\overline{X}$ is the output of the autoencoder, D is the decoder, E is the encoder and X is the original input to the network. Training autoencoder is finding optimal weights that minimize the difference between $\overline{X}$ and X.

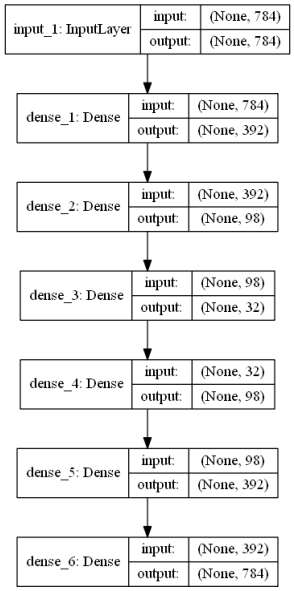


Figure 4.7: An example of autoencoder network architecture, showing the full network. Encoder consists of layers dense_1, dense_2 and dense_3 while decoder is dense_4, dense_5 and dense_6.

Autoencoders are trained using the same data as both inputs to the network and the expected outputs. After the training has converged and the performance of resulting full network is acceptable, the weights can be used to split encoder and decoder into their own neural network models, encoder acting as a dimensionality reducing model outputting low-dimensional representations and decoder recreating the original representations. Figure 4.9 shows the latent representations and reconstructed images from a simple autoencoder network. The latent representation is a single vector but for visualization purposes it is on two columns.

Autoencoders have numerous use cases. Some of them are described next.

Dimensionality reduction is an obvious use case of autoencoders, because that's what every autoencoder is doing implicitly. Ignoring the decoder part of the trained full network, encoder can be used to transform high dimensional input data into low dimensional representation. This low dimensional derivative can then be used for further analysis, for

example in visualizing the high dimensional data or using it as a new dataset for clustering. [16]

Data compression is essentially the same implementation as previously mentioned dimensionality reduction, but used for different purposes. Autoencoders can be used as lossy data compression method by compressing data with encoder and later inflating it with the decoder. As of writing this thesis, autoencoders are generally not used for this purpose because they don't perform better than arithmetic compression methods like JPEG and generalize worse, because autoencoder can not work with patterns it has never seen before. [7]

Noise reduction is a less obvious use for autoencoders, but when one considers what autoencoders learns it becomes clear that autoencoders are great fit for it. To reconstruct the original representation from compact representation the network has to learn informative features from the data it is trained on. Noise and artifacts are obviously irrelevant to the actual representations, and thus the network ignores them. The network has to be trained on clear data without noise and the network can then receive noisy data and output the data with noise removed or at least reduced. In real world, however, clean data is not always available in great enough numbers, but there exists advanced variants like *Robust Deep Autoencoders* which can learn to remove noise even without clean training data. [17]

Outlier/anomaly detection refers to finding data points significantly differing from "normal". Data from normal execution of a system can be fed into autoencoder which finds the usual patterns present in the system. When something out of ordinary happens in the system, the deviation can be detected from the hidden representation. Autoencoders have been successfully used for example in detecting anomalies in supercomputer execution [16] and intrusion detection in cybersecurity applications. Autoencoders are

applied to trajectory data outlier detection in section 6.3.6. There are various ways of detecting outliers and anomalies, depending on the structure of the data and the nature of the anomalies. In simple cases, in which the dataset has a single feature outliers can be detected with a boxplot. In this case outlier is usually considered to be value outside $1.5 * IQR$ range, IQR being interquantile range $Q_3 - Q_1$ of the values. Another way of of detecting outliers could be simply picking value a number of standard deviations away or farther from mean or median of the distribution. Both of these methods can be integrated with more complicated autoencoder networks by using autoencoder's *reconstruction error* as the error value, and labeling values with high enough error as outliers. This works because with large enough quantity of data autoencoder learns the structure of common values better than uncommon ones, resulting in higher error for samples deviating from the norm. Reconstruction error can be for example *mean absolute error* (MAE) or *mean squared error* (MSE). [16]

Regularization refers to applying constraints to the learning process to make it "harder" for the model by applying some forms of penalties or transformations to some of the model's properties. This is done to force the model to learn more general representations and thus prevent overfitting the training data. A common method of regularization is introducing dropout by randomly making outputs of some neurons 0. This would prevent the model from making decisions based only on a few activations and thus overfitting. [18]

In autoencoders a common form of regularization is known as *Lasso regression* or *l1-regularization*. It adds a penalty term to loss function, defined in equation 4.2 [8]. λ is the *learning rate*, signifying the magnitude of the regularization. $\sum_{j=1}^{p} |\beta_j|$ means that the magnitude of activation at every neuron is summed together. In practice this means that the more neurons have high nonzero values, the higher the penalty will be.

$$\lambda \sum_{j=1}^{p} |\beta_j| \tag{4.2}$$

This penalizes cases in which many neurons would be activated and steers the encoder-decoder network find parameters which represent the inputs with as *few neurons as possible*, acting as a form of *feature selection* for the encoded representation. Without this regularization, autoencoder approximates principal component analysis (PCA) [6]. From input-output viewpoint it seems that l1-constraint would just ”smudge” the output resulting in lossier compression, but the regularization actually forces the network to find more concise and informative latent representations which means that the output is represented with fewer bits in encoded form. This can be a desirable property in some applications. [6]

L2-regularization, also known as *ridge regression* has also been used in some implementations and is a common type of regularization in other types of neural networks. It penalizes the magnitude of each neuron, steering the network to have small positive activation. These are often combined together. In this thesis l1 norm will be used extensively in later experimental sections. [11].

Another common method of increasing the robustness of an autoencoder is to introduce noise into the inputs. The reconstruction loss is still computed according to the original image without noise. This forces the network to learn better latent representations that ignore the noise and results in better learned features, but has also the practical application of being used in denoising applications for example for images and audio. Example image with added noise is visualized in figure 4.8. [19]

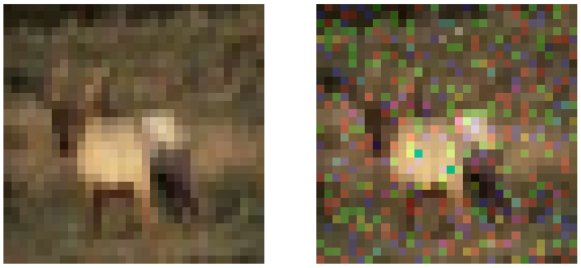


Figure 4.8: An image drawn from CIFAR-10 dataset with added noise on the right.

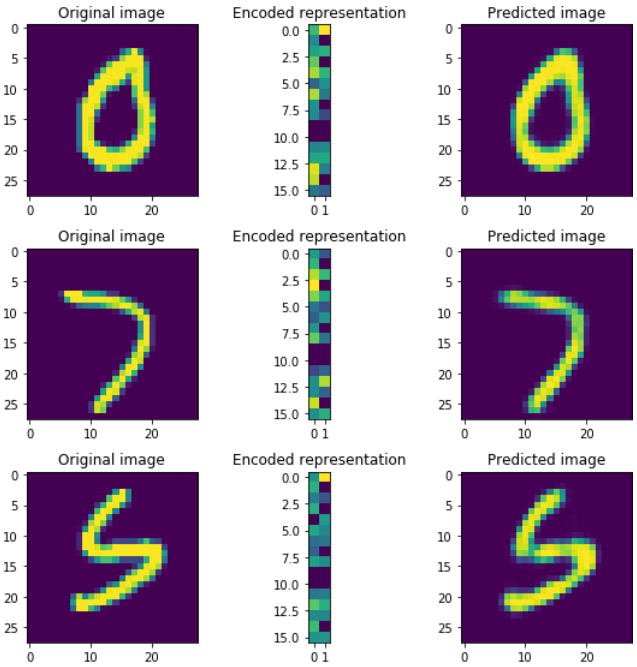


Figure 4.9: Data at different layers in autoencoder. Images from MNIST dataset on the left, encoded representation in the middle and matching reconstructed image on the right.

4.6 Variational autoencoders

Variational autoencoder (VAE) is a different kind of, more advanced autoencoder network that has found much success in image processing and data generation. Instead of traditional type of autoencoders which find (hopefully) well-structured subspaces representing the original data in a more compact way, VAE finds latent distributions which generate/reconstruct the original data with maximal accuracy. In other words, it finds the distributions of the properties that create the data of the input space. VAE consist of encoder and decoder parts, which are not directly connected in the same way as they are in regular autoencoders. The encoder part has at least two outputs which instead of the direct low-dimensional representation output the properties of the underlying distribution, for example in the case of Gaussian distribution this would be the *mean* and *variance* of every parameter. The decoder would then sample from these distributions and decode the randomly generated samples back to original shape. The network is trained with reconstruction error and usually additional regularization on the latent distributions. The network thus learns a number of well-formed latent distributions which generate the original data space with optimal accuracy. Variational autoencoders are a very promising type of network that provides more interesting and often better latent representation, but have the underlying assumption that the data is generated from a number of distributions. They are often used in image processing and have found many useful applications in that field. If the location data is converted into an image processing task, which it is later in the thesis, VAEs could also be experimented with. [7] [20]

4.7 Deep clustering

While modern advances in neural networks and deep learning have given significant boost to tasks such as object detection, classification and generative tasks such as speech synthesis, most prominent clustering algorithms are still those that have been in use for years,

even decades. One fundamental reason for the massive boost in neural network's performance is that they lend themselves well for efficient GPU implementations [21]. Neural networks by themselves however can't really do clustering tasks. Their capability to discover good underlying representations in massive amounts of high dimensional data however is a property that can be used as part of clustering framework (or "pipeline", rather), as a separate phase or as a joint task along with cluster creation. This task of leveraging deep learning for clustering is generally referred to as *deep clustering* [19]. The following section gives an overview of how deep learning is integrated into clustering tasks and the common properties of these implementations. The following chapter further discusses a certain neural network architecture in more detail, called the *autoencoder*.

Clustering is a relatively easy task in low dimensions and small datasets with little noise. Seeing how K-means algorithm separates groups in toy datasets like *iris* gives a good and important intuition on how these algorithms work and different groups are found. In real implementations however, even more so when dealing with so called *big data* applications, the datasets can be massive and very high dimensional. Clustering algorithms can be sensitive to these kinds of datasets and generally work better in lower dimensions [19]. Most algorithms in clustering are also parametric and require hand-picking some or multiple hyperparameters in prior such as ϵ in DBSCAN which has a major impact on the end results. Picking these parameters can be hard even with simple tasks, and it becomes even harder the more complicated and harder to reason about the dataset is. Most deep clustering frameworks leverage the really attractive property of neural networks, that is their ability to learn good representations on their own. A common paradigm in deep clustering is to perform feature extraction with a neural network as the first step before the lower dimensional representation is fed into some another clustering algorithm [19].

A naive way of boosting clustering would be to first perform dimensionality reduction/fea-

ture extraction with a network such as autoencoder or pre-trained convolutional neural network and then as a second step use the representations of the inner layers as input to clustering. This works, but there exists better ways as well such as training the model for two objectives jointly. These objectives are called the *network loss* L_n, which is the loss function for learning the representations, and *clustering loss* L_c which is some clustering algorithm specific way of measuring the quality of the clustering. These can be joined to a *joint loss function* which optimizes both objectives at the same time. [19]

The joint loss is presented in equation 4.3. The equation includes a *hype parameter* λ (hype, *not* **hyper**parameter) which specifies the impact of one objective in the total. When $\lambda = 0$ the objective would be optimized completely based on clustering loss and could lead to corrupted representations [19]. A common selection of the parameter is 0.5 which would divide the loss equally, but there exists variety of scheduling strategies which change the loss function parameters progressively. One common strategy is to pre-train the representation learning by training only with the network loss at first and then with both losses.

$$L_t = \lambda L_n + (1 - \lambda) L_c \tag{4.3}$$

There exists a variety of different architectures for deep clustering and a large catalog of different loss functions and other properties to choose from. A taxonomy of deep clustering algorithms and related losses is presented at [11] as a means to help selecting the proper components.

Most popular deep clustering architectures are based on the autoencoder network. Autoencoder based pipelines will also be the main focus in this thesis. Autoencoder is a two-part neural network which first compresses the input into low dimensional represen-

tation and then tries to reconstruct the original input from the lower dimensional representation. This constraint forces the network to learn good representation that enable it to recreate the original input with as little loss of information as possible. The ”middle” layer or the output of the *encoder* network can then act as a new representation of the original input data which can be used for clustering. The network loss in autoencoders is completely separate from clustering loss and is called the *reconstruction loss*. It is the difference between inputs and predicted outputs. A remarkable clustering architecture utilizing autoencoders is *Deep Clustering Network* which jointly optimizes the network and a k-means clustering loss. [19]

Not all deep clustering applications are based on the autoencoder. *Clustering deep neural networks* are regular, often pre-trained multilayer perceptrons or convolutional neural networks which are trained with only the clustering loss. They however often learn bad representations due to the lack of network loss. [11]

Generative models like *generative adversial networks* and *variational autoencoders* can also be used in conjunction with clustering. Their central property is that they don't just learn the arbitrary structure of the data space, they learn the *statistical distributions* of the input space and can create new samples from them, leading to their common use in generating new inputs.

Generative models or CDNNs are good candidates to explore when researching deep clustering, but autoencoders are easy to implement and work in almost every domain while other architectures require extensive handcrafting of the architecture and creating losses specific to the given task. The experiments of this thesis focus on autoencoder based implementations.

4.8 Data augmentation

4.8.1 General on data augmentation

Machine learning models require vast amounts of training data to dig out the underlying abstract patterns that are present in the data space. Depending on the task training data might or might not be readily available. Gathering proper dataset is one of the hardest tasks in a machine learning project. Selecting a model is usually relatively easy task and training the model requires only processing time. Gathering data can be expensive and challenging task, especially if the data is real-world data that cannot be scraped from the internet or bought from some vendor. The quality of the training data also more or less determines how well the end product will generalize to new data. Machine learning models can only be taught to find patterns that are present in the set that is fed into them, they cannot learn patterns that they have never seen. It is thus imperative that the input domain is understood so well so that a representative dataset can be created. [7]

There exists a class of techniques for artificially inflating the dataset. This is generally referred to as *data augmentation*. In data augmentation, a domain expert introduces realistic noise to the dataset, generating new inputs from existing ones. These transformations must preserve the label, i.e. they can mangle the input data in ways that can be found in ”nature” but must not change the informative content of the sample, ending up with bogus data that negatively impacts the model. When data augmentation is done correctly, it does not just duplicate the input, but creates a completely new input which contains the same patterns as the original one but with differing raw representation. A concrete example is rotating an image: a rotated image of an elephant still represents an elephant but the raw pixel values have changed in most locations.

Data augmentation can make the model more robust and help it differentiate the actual

informative patterns from noise. Selecting the methods of introducing noise to the input is not an easy task and requires a level of domain expertise. The data engineer performing the transformations must think of the data and the input domain and realize data that *could have been* found in the domain. There are no thumb rules as how this is done, as it is completely task specific, but it is most commonly found in literature regarding computer vision and easiest to understand in that domain. [22]

4.8.2 Data augmentation in computer vision

In computer vision the inputs are images, usually from either two-dimensional or three-dimensional world. Noise present in those domains is relatively easy to think about. An image of car represents a car even if it was stretched, mirrored, in different color scheme or zoomed in or out. Image usually represents the same target even if it is from different angle or a reflection in a window. Performing these kinds of transformations is a relatively common task and popular toolsets are readily available. In figure 4.10 five input images from *CIFAR-10* dataset are mangled with various transformations including shearing, rotating and zooming. The figures are very different from each other but still clearly represent the same type of object. Augmenting the training dataset with these kinds of images forces the model to ignore irrelevant noise present in these images and learn the patterns that actually relate to the label of the image.

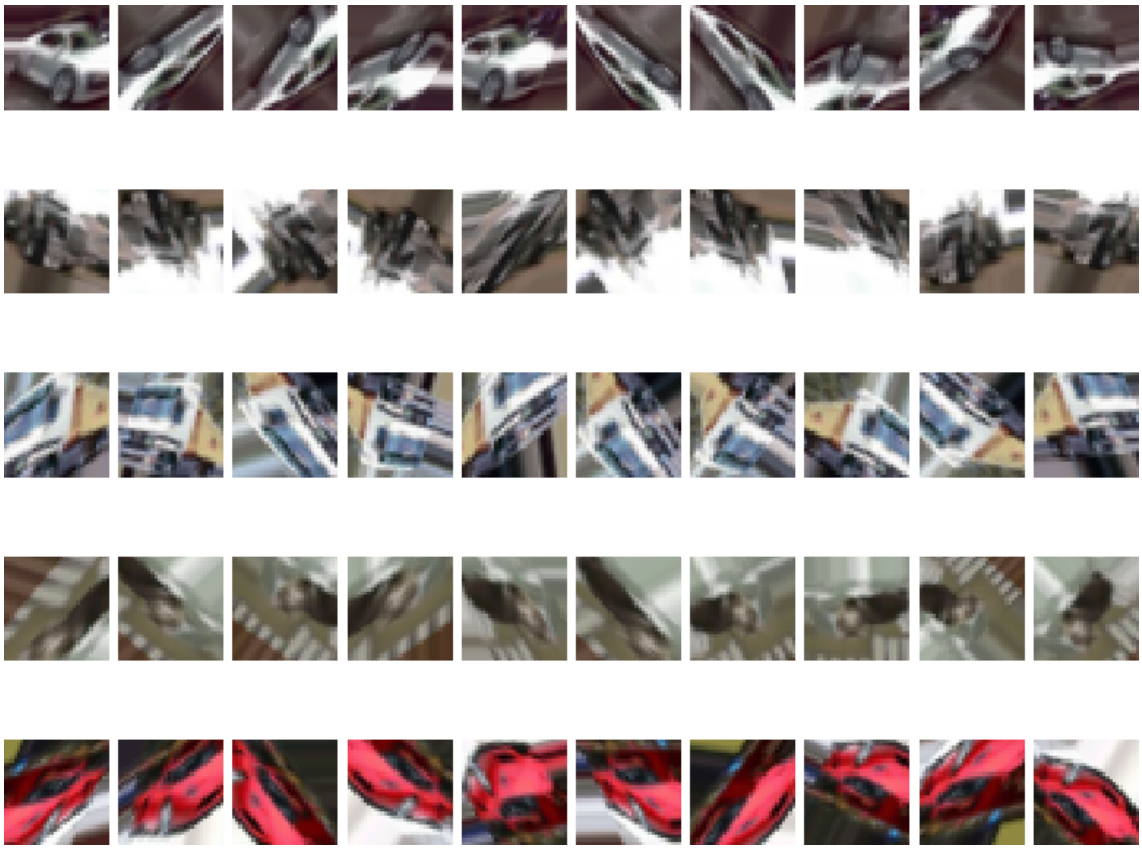


Figure 4.10: Augmented images generated from CIFAR-10 image dataset

4.8.3 Augmenting trajectory data

Real-world images have a rather well understood structure and augmenting transformations, but what about trajectories/paths and other indoor behavior that is of interest in this thesis? A path in building cannot be mirrored or zoomed out as that would completely change the path and just create bogus data. One possibility would be to cut paths into pieces and combine these into paths that *could* have been traversed, but this can be very complex and it would destroy the information about which paths are actually traversed. Another common method would be to introduce *jitter* to the raw paths. It means moving some of the path's points randomly. This noise can be drawn for example from a Gaussian distribution with some probability p for each point in the path. If the path is binned rather than the absolute coordinates, this jitter could be implemented by moving

the bin/cell by 1 location randomly. This would create paths that traverse more or less the same route but the coordinates have a variation that could be present naturally if the target made small sidesteps or there were inaccuracies in the locationing method which are both very possible scenarios. For the purposes of the research in this paper, jitter is the best and perhaps one of the only viable methods to augment path data. If the path is considered as an image, adding random Gaussian or uniform noise could also be a viable alternative.

Chapter 5

Interpretable machine learning

It is a relatively easy task to treat predictors as black box machines that take in some data and output results they are trained to predict. Sometimes this is enough, but it can be beneficial to understand at least on some level how model ends up with the output it does. Reverse engineering the model can help to understand the inner workings and thus help in optimizing the architecture. The rise of artificial intelligence and its fast adoption in numerous industries also brings up many ethical questions, especially if the models are dealing with decisions that have an impact in peoples' lives, through employment, validity for mortgage or insurance and alike. The models should not make predictions for example based on people's ethnicity or gender and such qualities, even if it would result in good looking accuracy from pure input-output viewpoint. These kinds of errors are surprisingly easy to make accidentally if the engineer does not understand the model well. The reasons to gain deeper understanding in the model's inner representations and how they relate to predicted outputs are thus many, even if it was just for pure curiosity and validation. In data mining interpreting the models is also a central problem as the goal is to gain useful insights to the patterns present in data, not just successfully fitting the model.

This chapter presents a general overview of various methods of interpreting deep neu-

ral networks, namely through visualization and knowledge transfer.

5.1 Knowledge transfer

Different types of machine learning models learn patterns and represent them in different ways. Some models like neural networks can learn very complex and non-linear representations of the input data while some shallow models like decision trees learn hierarchical representations that are easy to reason about. Different models have their own strengths and weaknesses. Some are simple and fast but fail to meet expectations when the underlying patterns are complex or the data is high dimensional. Some, like neural networks excel in high dimensional and complex data but end up being hard to interpret. Learned representations can be transferred from more complex model into a simpler one, possibly leading into superior performance than if it was trained normally. Varying terminology is used to refer to this process, some sources coin it as *distillation* [23] but it is not a well established term. A simpler model trained based on a more complex one is referred to as that model's *surrogate* [24].

5.1.1 Global surrogate

A global surrogate is a simple, or at least simpler model that is trained with the outputs of a more complex model. The simple model uses the outputs of the complex one as its target values, as opposed to the actual labels from data. The idea behind this is that the complex model does not learn to output just the label, but also outputs a *distributions* of predictions. This distribution itself contains learned information about the raw data and can be useful for improving the performance of the simple model. In digit classification, it is informative to know if a 3 looks more like a 7 or 8, for example. A simple model such as decision tree could be used to output this same distribution for the same inputs, with the hopes that this would drive it to learn better rules which could then be used to ”interpret”

the black box model. Even wrong classifications should be fed into the surrogate as the fact that certain samples were classified incorrectly could contain useful hidden information. This is a useful technique especially for optimizing complex and computationally heavy models for simpler systems and improving the performance of simpler models. Regarding interpretation it should be noted that the actual hidden representations are not necessarily transferred from the complex model to the simple one. This can happen, but there's no guarantee for it as the surrogate model is trained with just the outputs. This technique cannot be thus used to say for certain why or how a black box model does what it does. [24]

5.1.2 Distillation

Distillation is a knowledge transfer model closely related to global surrogate, introduced by the Google Brain team in 2015 [25]. The paper separates the output of the classifier into two scenarios, a *hard* target classification and *soft* target classification which separates it from "usual" global surrogate method.

Distillation works by first training a cumbersome, large model that can actually be an ensemble of models without concerning the resource usage. In neural networks, output layer activation function for multi-class classification problem is usually *softmax* which is formalized as follows.

$$q_i = \frac{\exp z_i/T}{\sum_j \exp z_j/T}$$

In which q_i is the class probability, z_i is the class logit and z_j the logits of other classes. T is the *temperature* parameter which in normal case is 1. *Hard targets* refers to phase in which model is optimized to minimize the loss between correct labels (class) and the predicted output. When training to create soft targets however, temperature parameter is increased so the output entropy is higher and the predictions "softer".

After this model is trained, a simpler model which does not need to be a neural network is trained to predict the same statistical distribution of outputs the cumbersome model produces instead of the correct labels. Distilled model uses same temperature parameter as was used to create the target distribution. The distilled model does not need to ever see the correct labels, but the soft targets can be mixed with the actual hard targets for improved performance. The results presented by Google are promising, with distilled models showing significant improvement over same models trained with raw data. Practically this is the same process as the previously discussed global surrogate, but with the addition of the *temperature* parameters. [25]

An interesting application of distillation regarding the research problem was presented in 2017 by partly the same team from Google Brain. In this approach, *hierarchical features* learned by neural network are distilled into *hierarchical decisions* in a rather shallow *soft decision tree* for the sole purpose of demystifying the learned patterns and providing clear ”thought flow” of the network to make sense how it ends up with the output distributions it does. The soft decision tree contains distributions of probabilities in its nodes, instead of hard if-else clauses. With distillation as well it must be noted that the surrogate and the complex model are independently trained so it is not a silver bullet for demystifying the inner workings of a neural network. [23]

5.2 Visualization

Visualization is one of the most important and intuitive ways to interpret the outputs and inner workings of machine learning models. Artificial intelligence systems, even ones utilizing machine learning are not ”intelligent” in the sense that they do not have same kind of deeper reasoning about the underlying problem that a human has. They only understand the patterns in the data domain they have been trained with, and whether that

understanding is of any use, and how it is used, is on the discretion of a human operator. Most of the machine learning models in use are not autonomous systems but act as a tool to help human operators do their intelligent task more efficiently. This is especially true in unsupervised learning and data mining in which the algorithm can't possibly even have any way of evaluating its results in the same way as in supervised learning with labeled data. The result of an unsupervised learning task are the patterns in data. A human is needed to transform these results into *knowledge*.

Visualizing data is a technique way older than the age of computer. Plotting on paper with a pen being one example. As computers have evolved in computing power so have visualization methods and algorithms. Humans have the inherent flaw that they cannot comprehend data in high dimensions, trying to comprehend data with more than three dimensions is already an almost impossible task for a human. Thus, to visualize structured high-dimensional data it has to be *projected* down to one, two or three dimensions so it can be presented in various graphical forms. Some methods to handle high dimensional data are presented in the following.

Multidimensional scaling (MDS) is a form of dimensionality reduction which is useful in visualizing data from which a *distance matrix* can be formed in a meaningful way. MDS is a family of algorithms which try to preserve the distances between pairs of data points in low dimensional representation. MDS is an optimization algorithm which tries to minimize a loss function, which in a common implementation is called a *strain*. For MDS to work, a distance function between two data points must be defined and it must describe the distance of two points in a meaningful way for the output to make any sense. An example is shown in figure 5.1. In this example, the original space is two-dimensional so MDS is used just to improve the visualization, but the original domain could be high dimensional. Euclidean distance, or some other power of Minkowski distance, is a common choice of distance metric when the inputs are spatial points. A custom distance function

can be crafted for more complex domains. Distance between indoor paths could be visualized using MDS, using some distance metric which measures the distance between two trajectories. [26]

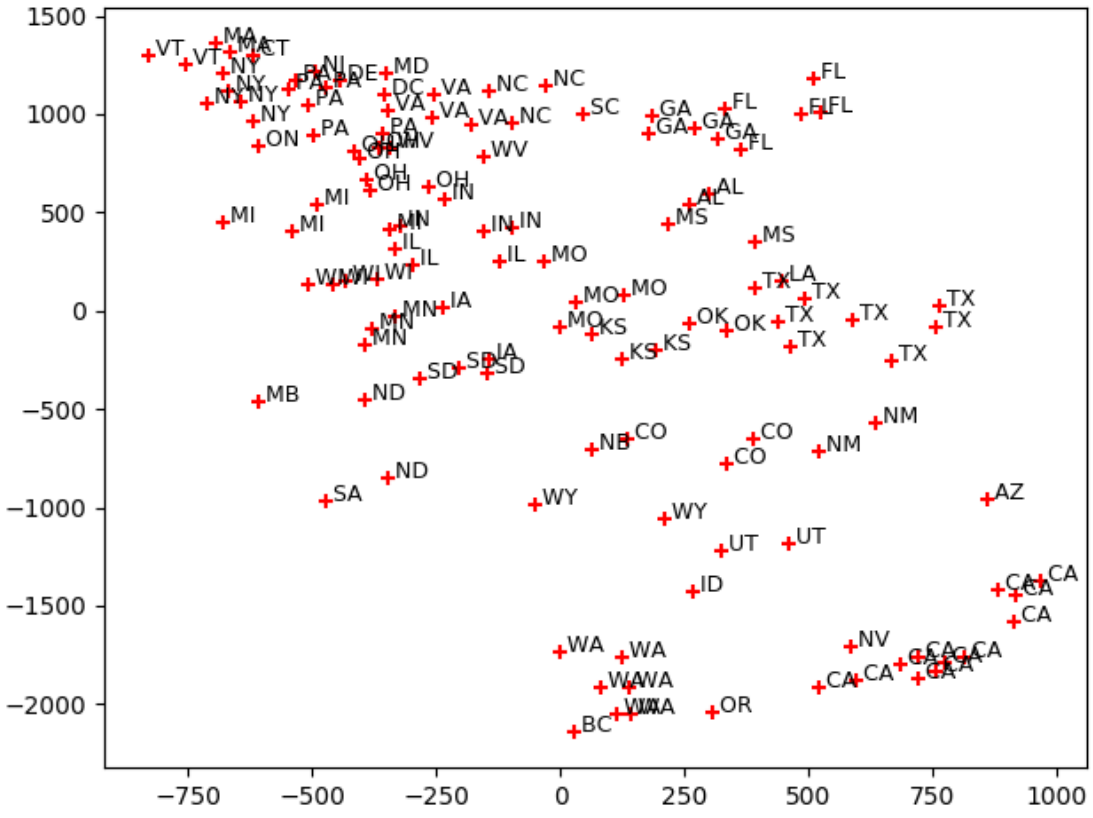


Figure 5.1: US cities plotted using MDS with Euclidean distance as the distance metric. The labels are abbreviations of their respective states.

Principal component analysis (PCA) is a popular method for doing dimensionality reduction based on mapping high dimensional input space into lower dimensional space while attempting to preserve as much of the *variance* in the data as possible. A dataset with high dimensions probably has less informative features and correlated features which mostly add just noise and make interpreting the data harder. The output of the transformation is a set of variables called the *principal components*. They are uncorrelated to each other (orthogonal) and the result of a linear combination of the original features. The features are ordered in a way that the first principal component encodes most of the variation

in the data, i.e. is the most informative. One then chooses a cut-off point and retains all of the principal components below. Like with other visualization methods mentioned here, there are many ways to do PCA, for example using the *Singular Value Decomposition* method which results in a matrix with all of the principal components. [6]

PCA has a plethora of good applications both as a visualization tool in exploratory data analysis and also as a preprocessing tool for many other algorithms. PCA can be used as a *feature selection* tool by preserving only some of the principal components while discarding less important ones. This loses some information but makes sense if we can get rid of many useless features while preserving a good portion (>95% or so) of the variation. PCA can also be used for plotting a high dimensional dataset by selecting one, two or three of the most important principal components and drawing a plot using those transformed variables. It is of importance to note that principal components are their own features and cannot be directly used to reason about the original features. In figure 5.2 there is a 30-dimensional artificial dataset projected down to two dimensions and plotted as a two dimensional scatter plot.

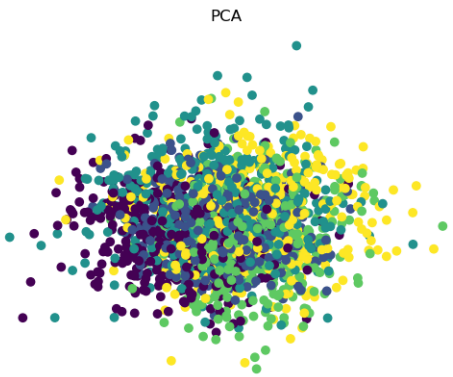


Figure 5.2: Scatter plot of 30 dimensional artificial data with six clusters projected to two dimensions using Principal Component Analysis (PCA).

T-distributed stochastic neighbor embedding (T-SNE) is another algorithm that projects high-dimensional data to fewer dimensions. T-SNE performs a nonlinear transformation by fitting a probability distribution over data point pairs in high dimensions and in low-dimensions and then minimizing the *Kullback-Leibler divergence* of the two distributions using gradient descent. This means that data points close to each other using chosen distance metric in high dimensions are also ideally close to each other in the new representation. The transformation is not linear unlike in PCA, rather the two projections are completely different representations of the same points but with similar distances between point pairs. T-SNE can be of significant aid in exploratory data science in which clusters are expected or searched for as it seems to automatically cluster similar data points into usually well-defined clusters. These clusters however are not necessary meaningful and thus must be manually inspected to gain insights about them. Different runs of the algorithm can result in different results as a result of optimization algorithm finding different local minima. An example of T-SNE output can be found in figure 5.3. The input data is similar as in figure 5.2 but the clusters are here clearly more distinct even in two dimensions. [27]

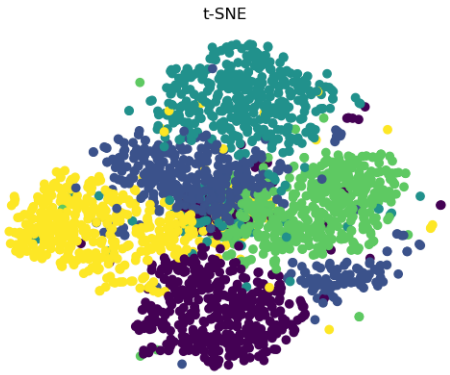


Figure 5.3: Scatter plot of 30 dimensional original data projected to two dimensions using T-SNE.

5.3 Inherently interpretable models

Some simple models have an advantage that due to their simplicity they are inherently interpretable in their raw form, in the sense that humans can use their trained form in full or simplified form to gain insights about the data patterns without external methods or a surrogate model. There are two major classes of ML models that have this characteristic (although it is subjective what is interpretable), *linear models* and *hierarchical models*. Examples from the first category include *linear regression*, *logistic regression* and *generalized linear model* (GLM). Most prominent example of the latter is decision tree which is already discussed in length. More complex models such as *random forests* can be utilized for example to explain feature importance, but these models quickly become very complex. Decision tree height can be restricted to a rather shallow tree for simplified explanations while giving up some expressive power. There also exists some models created specifically for interpretability, such as *Rulefit* which combines decision trees and linear models to create a linear model of features derived with decision trees. Models that are transparent and interpretable are referred to as *white box models* as opposed to opaque, hard to penetrate models which are referred to as *black box models*. [24]

Chapter 6

Experiments

This chapter introduces the experiments that were carried out while aiming to answer the research questions of this thesis. The main goal of the experimental part was to use machine learning to uncover patterns from indoor localization data. The initial goal was to cluster the paths and to create a predictive model which would predict the class (e.g. cluster label) of a moving object as it is active on the site, but due to the complexity of the task the scope was reduced to exploring trajectory clustering.

The data used in these experiments was gathered from the database of a large scale indoor localization system used for various purposes, including customer analytics, loss prevention and other custom use cases. The located objects are here referred to as *tags*, short for *smart tag*. These tags were located using a system based on ultra-wideband (UWB) communications protocol, approximating the location based on distances measured from a set of fixed *beacons*. The tags were located only when they were awake, i.e. on the move. During other periods they were in standby mode, preserving the battery. The boundaries of these sleep periods form a logical starting and ending points for their *trajectories*, although additional logic had to be applied.

First, in section 6.1 the available data is described as well as the preprocessing steps

that were common to all tasks. Then in section 6.2 a ”classical” type of trajectory clustering algorithm based on the *InTraRoute* discussed in section 2.1 and introduced in [3] is presented. A more complex framework based on deep clustering discussed in section 4.7 is then described in section 6.3.

The practical work was carried out using a toolset including tensor processing library **Tensorflow 2**, deep learning library **Keras**, **Python** language libraries **Matplotlib**, **Numpy**, **Pandas**, **Scikit-Learn**, **Scipy** and many other minor libraries.

6.1 Data and preprocessing

This section will go over the datasets that were used in the experiments, how they were gathered and what kind of steps were taken to preprocess and refine the data before it was used as input for machine learning algorithms.

6.1.1 Datasets

Two large datasets were used to produce the results and figures presented in this thesis. The initial data was the raw approximate locations of the located objects (smart tags) stored in the database of an indoor locationing system. The data was recorded at 1Hz and accuracy can be assumed to vary between less than a meter at best to around 5 meters at worst, including some infrequent outliers from locationing errors. The data was queried directly using SQL. This data also includes details about the state of the tag such as the quality of the measurements and whether the tag is going to standby, which were essential to filter out some outliers and to group the samples under trajectories.

The two main datasets are described in the following table.

Name	Environment	N	Preprocessed N	Time span	Rows / day
DS 1	Sports store	3435646	395791	∼1.5 years	∼7000
DS 2	Hypermarket	4351549	3678699	7 days	∼620 000

The differences between the two dataset are quite evident. The latter dataset is from a significantly more active environment. The preprocessing steps which were the same (with manually adjusted parameters) also shaved off around 15% of the data in *DS 2* while removing nearly 88% of the data in *DS 1*. The main reason for this is that majority of the data in the sports store dataset was insignificant short paths caused by the temporally short-lived movements of the tags. The differences in the datasets are also explained by the objects the tags were attached to. First dataset had the tags attached to hand-picked selection of products while in the hypermarket scenario the tags were tracking movement of shopping baskets. From this fact already it is obvious that *DS 2* has way higher potential to produce interesting insights about the customer behavior in retail environment, because they track the entire customer trajectory in the store and target a more representative portion of the customers. *DS 1* was retained and experimented with because it's a much simpler environment and provided a good proving ground for ideas. It was simpler to adjust the algorithms for example due to tighter *path hierarchy* in the smaller environment.

A small sample of the data is provided in table 6.1. Extra columns are removed from this sample and the timestamp is in human readable format instead of Unix timestamp.

X	Y	Timestamp	GoingToSleep
38244	-40960	08:30.7	False
38244	-40960	08:30.7	False
38237	-40885	08:30.7	False
36606	-40080	08:33.7	False
37054	-40226	08:35.7	False
37349	-40404	08:36.7	False
37199	-40342	08:37.7	False
37371	-40422	08:38.7	False
37251	-40374	08:39.7	True

Table 6.1: Example of what the data could look like.

6.1.2 Preprocessing and cleaning

The raw dataset consists of a flat table of timestamped tag locations. This data has the approximate coordinates (relative to arbitrary origo on the site) as well as tag's state. This data has to be aggregated into well-defined *trajectories* for further processing. The location data is also highly variable in quality, even within a single trajectory and thus some significant aggregation and cleaning steps are necessary before the data is fed into any machine learning model. The most significant issue to tackle with preprocessing is to filter out *noise* i.e. irrelevant trajectories, which in this case does not refer to actual locationing noise but to paths that are not interesting and can skew the dataset and make gaining insights harder and distort the training of ML models.

Creating trajectories

The data was grouped into trajectories with a well-defined starting and ending point. The starting point was chosen to be the first location after previous sleep period or after a sig-

nificant time period has passed since last message from the tag (as is in the case when tag has moved out of the range of the beacons). The subsequent locations are then appended to the trajectory until the tag informs that it will go to sleep due to inactivity and all of the optional ending conditions have been met. In *DS 1* there were no additional conditions, but due to its sheer size in *DS 2* the tags were additionally required to be within the specified zone around the checkouts. Due to the tags being attached to shopping baskets this is a very reasonable condition since the tags will end in that area at the end of their trajectory unless they are abandoned somewhere within the store (in which case the entire path was filtered out). Without this additional step, a large portion of the trajectories would have ended prematurely, as the basket left alone for a moment causes the tag to go to sleep.

Filtering noise

To improve the quality of the dataset, many heuristic conditions were used to filter the dataset after the trajectories were created. These conditions aim to remove short paths (tags waking up and going back to sleep, or maybe moving just a few meters), invalid paths and otherwise suspicious paths based on prior expert knowledge.

Some of the used filters are listed in the following. Not all were necessarily used simultaneously.

- Filter out paths with fewer than N unique points
- Filter out paths which end very near where they started from
- Filter out paths which have alarmingly large jumps (in temporal or spatial space)
- Filter out paths which start or end in invalid regions (such as backrooms)

Different environments required different values for the filters which were manually chosen based on examination of the dataset. Heavy handed filtering meant that a large amount of data was thrown away but due to its ready availability this could be afforded.

Creating new data

Some algorithms discussed in this paper, namely autoencoders discussed in 4.5, can benefit from added noise and all algorithms benefit from additional training data. For this reason several ways of augmenting the dataset were experimented with. Common image transformations like translation, skewing and rotation could not be used in this case since it would change the informational content of the data. The only trivial way of inflating the dataset was found to be adding random Gaussian or uniform noise to the location data (numeric coordinates), or directly to the trajectory image (intensity in a given area). This was done by drawing a sample from a Gaussian distribution with standard distribution (σ) controlling the intensity of the noise. To add noise directly to the image, the absolute of value was then multiplied with a Bernoulli distribution with probability p controlling the probability of noise at given coordinate to only pick some noise samples. This matrix was then added to the original image data. When the input data was converted into binary form, the noise was also binary (0 or 1) with probability p, and it was *xor*ed into the input data. An example of created noisy sample can be seen in figure 6.1. It should be noted that the previously described method could only be used with the later discussed deep clustering method, the *InTraRoute* based method could not work with such noise. Creating new paths by modifying coordinates could be used for the "shallow" method but this is questionable as the model does not attempt to learn any patterns from the data but directly clusters the input and adding fake data to this mix could hurt the clustering. Another way of adding noise by breaking trajectories to subpaths was experimented with but found to be exceedingly complex and it was abandoned for now to limit the thesis' scope. As discussed in 4.8, the intensity of randomness should be within realistic boundaries. The amount of noise within the locationing system is not constant and there doesn't exists any trivial method to select it automatically, so the parameters were chosen by hand for the spefific environment.

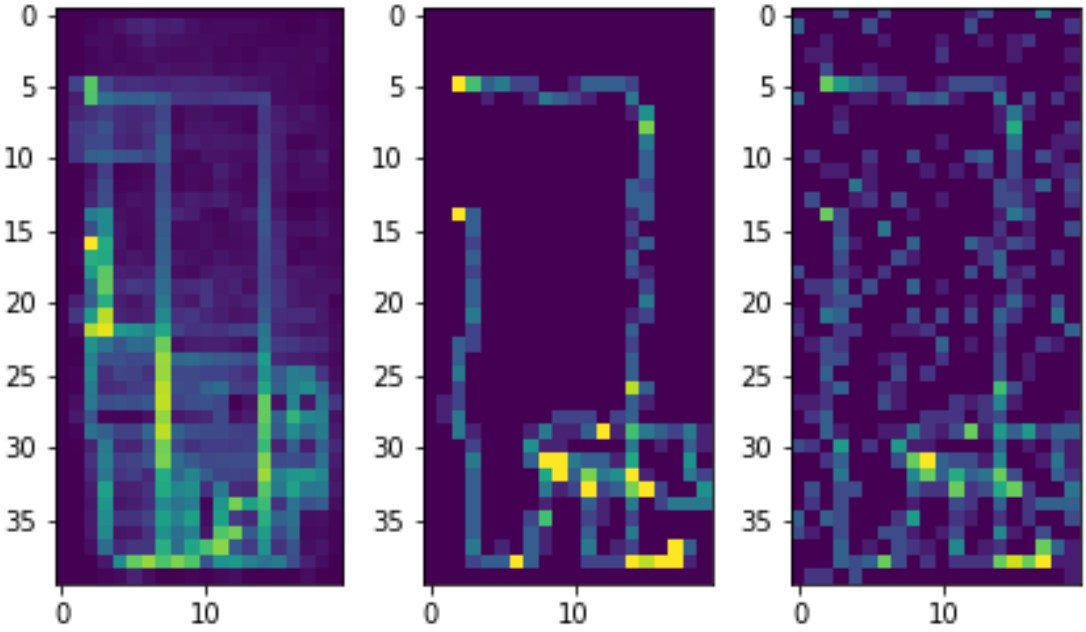


Figure 6.1: From left to right: All filtered trajectories from *DS 2* combined into single heatmap, a single trajectory from *DS2*, and rightmost the same trajectory with added noise

Converting the data into an image

The initial form of the location data described previously is *tabular*. A single sample represents the location and the state of a tag at given moment. The data is temporal in nature, which means the single trajectory, which is considered to be the meaningful unit of our research, consists of multiple samples on a temporal axis. Data like this could be used directly as an input for a model that can handle such data, the canonical choice for this would be recursive neural network (RNN) or long-short term memory (LSTM) based neural network. Working with time-series data is complicated, but with a simple trick we can convert the trajectory into a single sample and leverage all of the powerful tools available for image processing. This can be achieved by converting multiple rows of location data into a two-dimensional image. The gist of this trick is easy to understand if we imagine the observer (model) looking at the trajectory from bird-eye view. Additional data,

such as ordering or tag's state can also be trivially added into this image by adding new layers, making it technically a 3D image. Working with images is far easier from both algorithmic and human perspective; image processing is a wildly popular subfield in ML and observing the results and debugging issues is easy when the state is a visualization by itself!

Discretization is the process of turning continuous data into discrete data, that is giving it a discrete value. The nature of the data at hand, namely it's variably inaccurate nature means that putting the data into *bins* doesn't lose much information and makes the data a lot easier to handle. In this case we do this by splitting the site's dimensions into G_w x G_h grid of bins/cells and assign each sample the coordinate of the cell it falls under. The coordinate system was cropped to match the site's blueprint. Samples that are outside of the grid can either be ignored or be given the coordinate of the closest cell, in this research we did the latter. After the location data has been converted into this new coordinate system, each trajectory can be trivially converted into a single tensor of shape $G_w * G_h * 1$ which can then be used as an input for a convolutional neural network (CNN) as will be done later. The values in cells can either be binary or the sum of the samples that fell into that cell. The latter actually becomes a *heatmap* of the trajectory and a useful visualization. Both of these options were experimented with and the latter was found to be generally more useful.

In figure 6.1 there can be seen a combined heatmap of all trajectories in *DS 2* and a single trajectory created with the previously described method. This is the basic form of data that was used as an input for the deep clustering pipeline.

6.2 Improved shallow clustering framework

Being almost the only pre-existing framework for clustering indoor trajectories the *InTraRoute* framework discussed in 2.1 was used as the starting point when beginning the experiments on trajectory clustering. It was found to work to a some degree without customization, but the methods presented in the original paper were found to not scale well into a more accurate and higher density path environment as used in system at hand. Several improvements were made to make path clustering work better in this use case and these improvements are presented in this chapter. This algorithm, or rather a set of algorithms is referred here to as *improved shallow clustering framework* due to being an improvement upon *InTraRoute* and being based on *shallow*, or ”classical” methods instead of *deep* models such as neural network.

6.2.1 Data and preprocessing

The data was such as discussed in 6.1.1 and was preprocessed using the steps in 6.1.2. The raw locations were also converted from the original ”absolute” coordinate space into the aggregated *grid* space as described in 6.1.2 but **not** converted into an image tensor, rather the data with new coordinates were retained in the tabular form. This is the form of data that was used as input in this algorithm.

Trajectories were further processed to remove duplicates i.e. sequential points that fell into the same cell. These were removed because even though they do provide information about how long the target stays at one point, they affect similarity measures negatively and do not provide interesting information regarding the path traversals. Trajectories typically contain many identical locations at the beginning and the end of the trajectory and removing these was found to be essential in making the clustering algorithm work well. The algorithm described in this chapter was also found to be very sensitive to the processing

steps and the quality of the data, way more so than the neural network based implementation described later in this thesis.

InTraRoute paper mentioned that it removed *loops* from the trajectory to reduce noise that could prevent two otherwise similar paths from being clustered together. Implementation details however were not provided, so it was implemented independently here using Algorithm 1.

Algorithm 1 Removing Loops from path

```
function REMOVELOOPS(P)
    Input: P is a path represented as a list of coordinates
    Output: Path P with loops removed
    prev ← null
    graph ← graph()
    for x, y in P do
        node ← (x, y)
        if HasNode(graph, node) then
            if prev <> node then
                loop ← ShortestPath(graph, node, prev)
                loop ← loop + node
                prevLoopNode ← null
                removeNodes ← array()
                for loopNode in loop do
                    if prevLoopNode <> null and loopNode <> node then
                        RemoveEdge(graph, prevLoopNode, loopNode)
                    end if
                    if loopNode <> node then
                        removeNodes ← removeNodes + loopNode
                    end if
                    prevLoopNode = loopNode
                end for
                RemoveNodes(removeNodes)
            end if
        else
            AddNode(node)
            if prev <> null then
                AddEdge(graph, prev, node)
            end if
        end if
        prev ← node
    end for
    return graph
end function
```

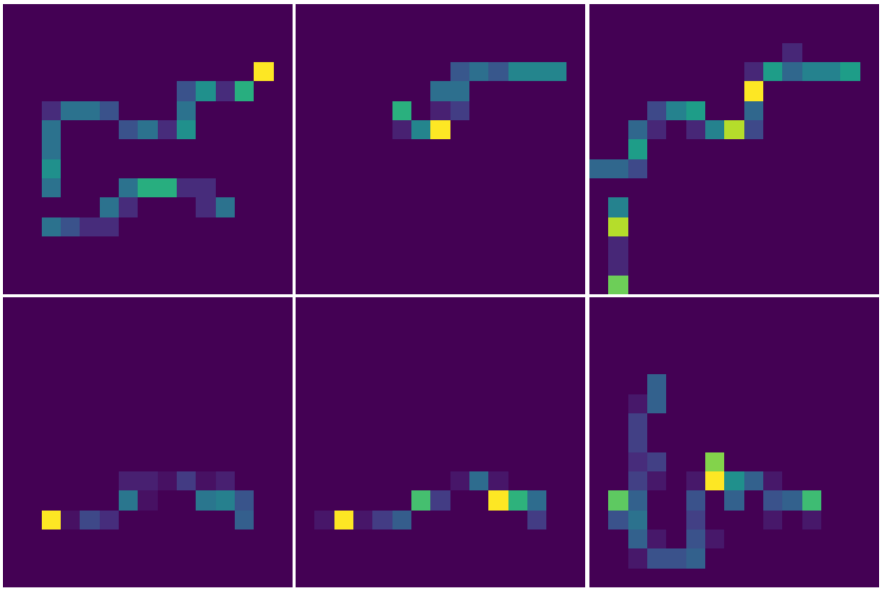


Figure 6.2: Several paths binned with grid size $W_c = 15$, brighter color indicates more points inside the respective cell

6.2.2 Trajectory clustering

After preprocessing the trajectories in tabular form are clustered together using the following algorithm. This is done using mostly the same procedure as described originally in [3] with some changes, primarily the change of distance metric. The clustering algorithm goes through all of the trajectories, indexed by the *trajectory ID* created as described in 6.1.2. The order of iteration does not matter but the algorithm is *undeterministic* so a different order might result in slightly different clusters. The clustering algorithm works as follows.

1. If there are no clusters, the first trajectory starts the first cluster

2. Distances between a trajectory and *cluster representative trajectories* of clusters are calculated using a chosen *distance metric* $f(x, y)$

3. The trajectory is added to the cluster if the distance is below selected *treshold value*

4. An updated cluster representative trajectory is calculated when a new path is added to it

5. If the trajectory is not added to any existing cluster, it starts a new cluster

When all of the paths are clustered, the clusters with too few members in it can be filtered out, considered to be outliers. These clusters generally are not interesting and add noise, but it depends on context.

A cluster of paths is represented by a *cluster representative trajectory* which can be understood as the average route taken by tags traversing that path. The representative route is calculated by creating a *graph* out of the paths in the cluster. In this graph nodes are cell coordinates and edges represent traversals between two cells. These edges have associated *weight* with them which is the *inverse square* $\frac{1}{x^2}$ of the count of times this traversal happens within this cluster. Practically, this means the more often a traversal between two cells happens, the smaller the weight or "cost" this edge has.

Clusters also have a well defined *start* and *end point*. All of the start and end points *in original coordinate space* of cluster's paths are averaged and the cell that average point falls into becomes the cluster's start and end point respectively. The shortest path from this start node into the end node is calculated using *Djikstra's algorithm* with the weights calculated as previously described. The representative path can be retrieved from the cell coordinates of nodes belonging to this shortest path. To calculate the path in original coordinate space a mean or median of every point in the cells of the representative path are used. In figure 6.3 one can see the raw paths that were clustered together (in grey) and the representative path for the cluster (in red). The representative path traverses through cells selected for the representative, black crosses are the median of all raw path points inside

those cells forming the path.

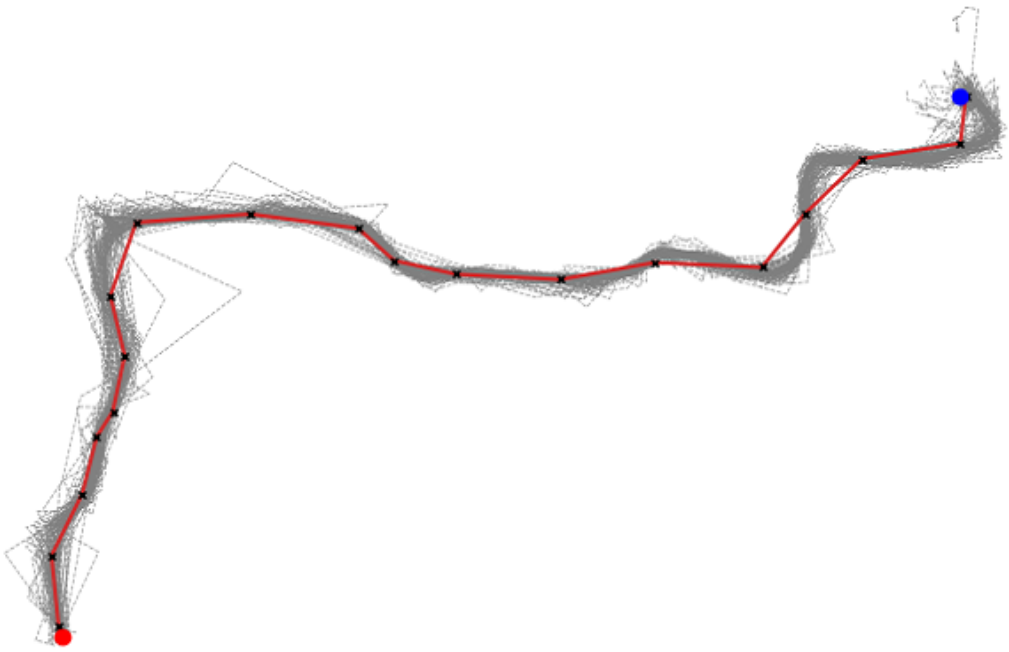


Figure 6.3: Raw paths (in grey) and their cluster representative path (in red)

6.2.3 Distance metrics

Distance metric is an essential component of the clustering algorithm, deciding whether a trajectory becomes a part of a cluster or not. The choice of the metric affects not only the outcomes but the choice of other parameters as well.

There exists a wide variety of distance metrics for estimating the similarity between two curves or trajectories. Most of them however are not suitable for this scenario for one reason or another, usually because they make such assumptions about the curves that do

not fit indoor trajectories. Many of them are described more thoroughly in [3] and [28].

The simplest way of measuring the distance between two trajectories would be to sequentially calculate the *Euclidean distance* $\sqrt{(p_{x1} - p_{x2})^2 + ... + (p_{n1} - p_{n2})^2}$ (in which n is the dimensionality of the coordinate system) of each matching point pair between the paths. This is however very brittle as it assumes the trajectories would proceed at the same pace and breaks instantly if the traversals happen at different speeds so it is not a viable alternative here. Other interesting possible measures not further discussed here include *dynamic time warping* and *Hausdorff distance* [28].

Jaccard distance also known as *Jaccard index* and *Jaccard similarity coefficient* is presented in [3] as the distance measure they used to cluster paths in a hospital environment. It is a rather simple measure and can be described verbally as *intersection over union*. It operates with sets so the paths have to be transformed into sets of discrete coordinates. Thus, the function measures the relative similarity between two *unordered* sets of coordinates or any arbitrary values and gives a similarity value to them. Each XY cell coordinate represents its own unique value but they have no other relation to each other. To test this distance measure, cell's XY coordinates were transformed from tuple into a string and then stored in a set. The function is defined formally in 6.1, in which S_1 is the set of first path's coordinates and S_2 the set for the second path.

$$J(S_1, S_2) = \frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|} \tag{6.1}$$

This was tested with real-world data and it demonstrated moderate performance, but it has inherent drawbacks that prevent it from being the distance measure for our improved framework. Firstly, it considers the trajectories *unordered bags of arbitrary values* and thus all information about traversal direction is lost. This is not good as it would cluster together two similar looking paths that would run in the exact opposite directions. In some

cases this could be okay but generally we would want to consider them as being different paths. Secondly, and even more importantly the coordinates *lose their relative distances*. This means that every point right next to cluster's representative path is considered to bear no relation as the unique coordinates do not intersect. This makes the choice of grid size very essential for clustering. A large grid size means that paths clustered together most likely are very similar, but many obviously similar paths are not considered to be in the same cluster. Too small grid size makes it easier to cluster similar paths together as they are more likely to traverse through same cells, but it might also include paths that have clearly different behavior than other paths in the same cluster.

Jaccard index was the initial choice of distance measure for clustering as it was used in the existing implementation described [3]. The InTraRoute system considered paths on a very high level and on a large scale (positioning accuracy was $\sim$ 20 meters in the paper). In this kind of environment Jaccard index might perform better, but with the finely grained locations in this thesis a different distance metric should be used.

Discrete Frechet Distance was found to be the best way for measuring distance between two indoor trajectories. It takes into account the ordering and is very robust against small to modest variations between the paths. The algorithm is an easily computable variation of the continuous *Frechet distance* introduced by a French mathematician Maurice Fréchet in 1906 [29]. The discrete version approximates the Frechet distance of "real" curves by using a series of points (polygonal curves). This happens to be exactly what we have in our dataset so it is a very natural fit for the current problem.

Intuitively Frechet distance can be defined as follows (taken from [29]): "A man is walking a dog on a leash: the man can move on one curve, the dog on the other; both may vary their speed, but backtracking is not allowed. What is the length of the shortest leash that is sufficient for traversing both curves?" Algorithm 2 gives a more formal definition

of the algorithm introduced by Eiter et. al.

Algorithm 2 Discrete Frechet Distance [29]

d is a distance metric, usually Euclidean distance

function DISCRETEFRECHET(P, Q)
  Input: Polygonal curves P (u_1, ... u_p) and Q (v_1, ... v_p)
  Output: Distance between curves P and Q
  $ca \leftarrow array[1...p, 1...p]$
  for $i = 1$ to p **do**
    for $j = 1$ to q **do**
      $ca(i, j) = -1.0$
    end for
  end for
  return $c(p, q)$
end function

function C(i, j)
  if $ca(i, j) > -1$ **then return** $ca(i, j)$
  else if $i = 1$ and $j = 1$ **then**
    $ca(i, j) = d(\, u_1,\, v_1)$
  else if $i > 1$ and $j = 1$ **then**
    $ca(i, j) = max(\, c(i - 1, 1),\, d(u_i, v_1)\,)$
  else if $i = 1$ and $j > 1$ **then**
    $ca(i, j) = max(\, c(1, j - 1),\, d(u_1, v_j)\,)$
  else if $i > 1$ and $j > 1$ **then**
    $ca(i, j) = max(\, min(\, c(i - 1, j), c(i - 1, j - 1), c(i, j - 1)),\, d(u_1,\, v_j))$
  else
    $ca(i,\, j) = \infty$
  end if
  return ca(i, j)
end function

6.2.4 Insights

We took the existing algorithm first described in [3] and modified it to work in our use case. Specifically, the very coarse original clustering algorithm was modified to work in environments with a much higher localization accuracy and higher path density. This set

of algorithms provides a framework which can be modified to fit different use cases, and improved upon in any way that a specific use case calls for. The distance metric especially is a component that is easy to switch. The algorithm can also include much more finely grained logic relating to how clusters are formed or how these clusters are converted into actual trajectories. The loop removal algorithm can also be considered optional, as it adds quite a bit of complexity and was in practice found to not affect the result in a significant way.

The framework has many parameters that all have trade-offs regarding the outcome. There exists no general thumb rule how they should be chosen as it depends on the requirements of the use case. Some of the (hyper)parameters of the framework are listed in the following.

- Size of the grid $G_w * G_h$
- Limits of the grid in continuous coordinates (area to be binned) [1]
- Minimum start-end cell distance
- Minimum unique cells covered
- Distance metric $f(P_1, P_2)$
- Clustering treshold T
- Minimum number of paths in a cluster

Needless to say, that's a lot of parameters to choose from, especially since each one of them requires some level of deeper understanding of the algorithm and the data. The presented algorithm nevertheless presents a viable framework for clustering a large number

[1] In our experiments created from blueprint bounds

of raw trajectories and has the potential of being used as a basis for many implementations. It has many advantages, including its computational effectiveness and simplicity (easy to understand, easy to debug). It is a good choice for simpler environments, the best results were recieved when working with *DS 1*. As the environment gets more complex, so does the configuration of the framework.

Figure 6.4 demonstrates two clusters created from a large amount of raw trajectories (non-clustered paths not visible) with different treshold values. It demonstrates how a choice of the *treshold* parameter affects the resulting clusters.

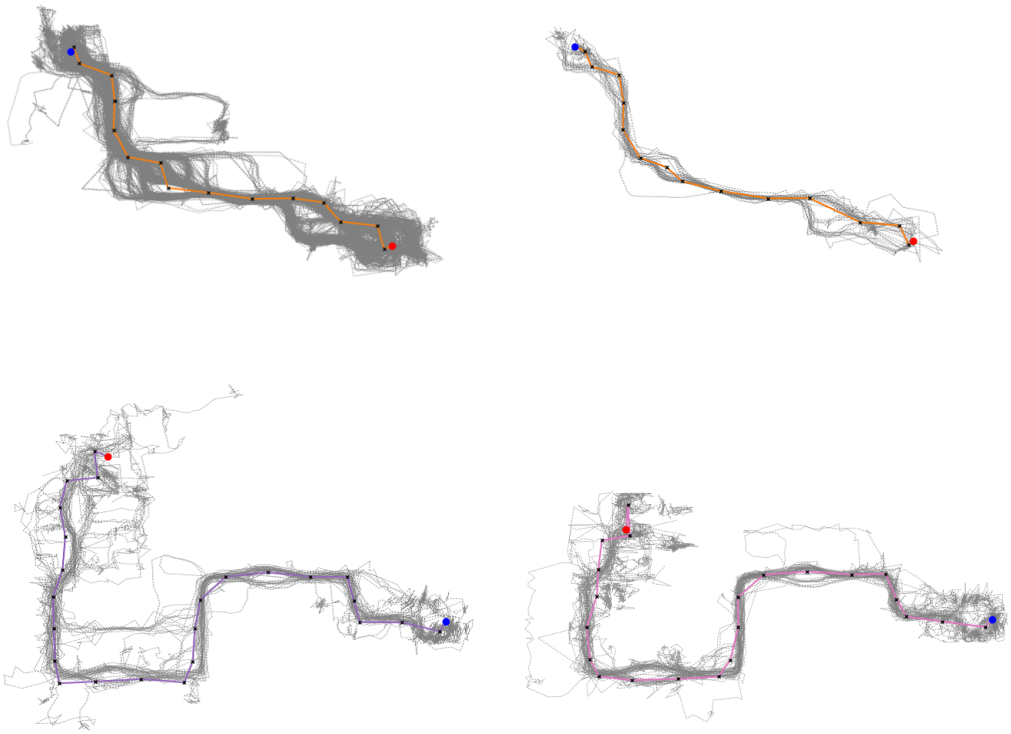


Figure 6.4: Same path clusters with different treshold values. Clusters on the right side have smaller (tighter) treshold. One can observe how tighter treshold filters out some less similar paths.

6.3 Deep learning approach

This chapter introduces an alternative, more advanced and more complex approach to the trajectory clustering problem. This chapter's solution is based on deep clustering method discussed in 4.7, relevant theory can also be found in sections relating to neural networks (4.1), autoencoders (4.5), DBSCAN and HDBSCAN (4.4.5) and T-SNE (5.2).

Chronologically, work on this section's implementation began after the previous section was complete. The goal was to find a new approach to the clustering problem based on neural networks. The work began by researching neural network based clustering methods found in literature as there was no specimen to use as a direct example. Sources such as [19], [30] and [11] provided good starting points and most of the practical work after that was just pure experimentation. The demonstrated implementation follows roughly the simplest model defined in [19], which consists of separate *representation learner* and *clustering* stages.

The overall collection of algorithms and parameters is referred here as both *framework* and a *pipeline* because it's really both. The framework consists of a pipeline of algorithms in which the output of the first part (representation learner) is used as the input for the second stage. Both of these parts can have different algorithms plugged into their place and they have many configurable parameters themselves.

6.3.1 Data and preprocessing

The input data was as described in 6.1.1 and was cleaned and preprocessed as described in 6.1.2. The significant preprocessing step that is different from previous chapter's algorithm is the conversion of tabular data into third rank tensors as described in 6.1.2. In a nutshell, this meant converting the list of samples which define the tag's location at dif-

ferent points of time into a two-dimensional matrix. The third dimension is required for convolutional neural network (CNN) which will be used. The third dimension contains different *properties* of the trajectory in spatial space. In this thesis' implementation there is only a single property (and thus the third dimension is always of size 1), the presence of the tag in the given area, or *intensity*. Practically, the input to the model is the *heatmap* of the trajectory. In more advanced models, the input tensor could also encode other properties such as *speed*, *order* etc. but to limit the scope of the thesis only the simplest form is discussed here. The model should require little to no changes as these additional properties are added, due to the neural network's ability to learn the representations intrinsically as shortly described in 3.2 making the model extremely dynamic and scalable.

Figures 6.1 and 6.2 display this data visually, with the dummy third dimension removed.

6.3.2 Architecture

The pipeline consists of two main parts, *representation learner* and *clusterer*. This architecture got inspiration from many of the examples described in [11] but is created from ground up for current use case and did not follow any specific example. One aspect separating the developed implementation from many of those in the literature is that the favourite clustering method in existing examples seems to be *k-means* or similar algorithm that requires the knowledge of the number of clusters in prior. One constraint that dictated the choice of algorithms in our use case was that there is no knowledge of the cluster count or other such properties in prior, which prevented us from first experimenting with known implementations before creating our own.

The representation learner's task is to transform the complex raw input data into a more concise and simple form that can be used as an input for the "shallow" clustering algorithm. Most of the clustering algorithms available would not work at all with the type of

data we have at hand, thus requiring significant preprocessing. The main transformation that has to be done is reducing the input dimensions into a single vector. This could be done manually, by crafting features such as *presence in given area* or *time spent in given area*, but the aim of deep clustering is to use neural network's properties to discover good features automatically. This minimizes information loss as the algorithm gets to learn what actually is informative and what is not. The exact architecture for the representation learner was an *autoencoder* network consisting of *convolutional layers*, or *convolutional autoencoder* for short. The working principles of an autoencoder are described in section 4.5. The first and last layer have the shape of the input data and the network is in double "funnel" shape. Each of the convolutional layers is succeeded by a *max pooling layer* which reduces the size of the data and a *batch normalization layer* which acts as regularization. The middle layer is preceded by a *flattening layer* which transforms the 3D data into single dimension, i.e. a vector. The middle layer's size decides the size of the latent representation, or the number of features the input data is converted into and is thus one of the major hyperparameters of the model. L2-regularization was used in convolutional layers and L1-regularization in the middle layer. The effect of this regularization is described later.

The representation learner transforms the input image into a relatively small one-dimensional representation which is then used as the input for the subsequent clustering phase. There exists some implementations which do both the representation learning and clustering in the same algorithm, usually the neural network outputting cluster properties directly. Some of these are described in [11] but they are more complex and introduce constraints, such as fixed cluster count, so they were not included in this thesis. The clustering algorithm works independently from the first step and finds clusters from the latent representations, so being part of the pipeline does not introduce any additional concerns in the simplest case. The clustering task in this architecture is straightforward: first train

the autoencoder, and then create clusters with the clustering algorithm. The clustering algorithms of choice in this implementation were *DBSCAN* and *HDBSCAN*, both discussed in section 4.4.5, due to their ability to handle noise and not requiring a fixed cluster count.

The method presented previously is the simplest and most straightforward way to do deep clustering, but still works remarkably well and above all, is easy to develop further and adjust manually. In literature, for example in [19] this architecture is referred to as *deep embedding network*. There exists a plethora of various constraints, losses, architectures and additional training methods to improve upon this, many of which are covered in [11] and [19]. A large number of different variations could be tried but they are out of scope for this thesis. Reader is advised to refer to the cited papers for deeper dive to optimizing deep clustering. A simple way to improve upon the presented architecture is by combining the neural network *reconstruction loss* with clustering algorithm's *clustering loss* into a single *joint loss function* as discussed in section 4.7. This is discussed in more detail in the referred theoretical section, but the main gist is that this makes the autoencoder learn features based on clustering performance instead of reconstruction performance. In the current model, the autoencoder is trained in such a way that makes the output of the *decoder* (latter) part as close to the original input as possible, which is not really interesting here. Some time was spent on this, but it was found that including the clustering loss in gradient descent requires the clustering algorithm to be differentiable which is not the case for most of the algorithms. There exists some differentiable implementations of popular clustering algorithms like *k-means*, but as continuing in this direction would have quickly become exceedingly complex, it was left out of this thesis' scope. An alternative direction would be to replace gradient descent with an optimizer that does not require calculating the gradient of the loss function, for example by utilizing genetic algorithms as described in [31]. Creating a joint loss function provides a clear point to continue the research and would be a subject of its own research paper.

6.3.3 Autoencoder network

The following listing provides one example of an autoencoder network used during the experimentation, as outputted by Keras' **summary()** method. It should be noted that this is by no means the ground truth or the suggested model, just one architecture that worked.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 44, 24, 1)         0
e1 (Conv2D)                  (None, 44, 24, 32)        320
e2 (BatchNormalization)      (None, 44, 24, 32)        128
e3 (MaxPooling2D)            (None, 22, 12, 32)        0
e4 (Conv2D)                  (None, 22, 12, 32)        9248
e5 (BatchNormalization)      (None, 22, 12, 32)        128
e6 (MaxPooling2D)            (None, 11, 6, 32)         0
e7 (Flatten)                 (None, 2112)              0
e8 (Dense)                   (None, 256)               540928
e9 (BatchNormalization)      (None, 256)               1024
encoded (Dense)              (None, 16)                4112
d1 (Dense)                   (None, 2112)              35904
d2 (Reshape)                 (None, 11, 6, 32)         0
d3 (UpSampling2D)            (None, 22, 12, 32)        0
d4 (Conv2D)                  (None, 22, 12, 32)        9248
d5 (BatchNormalization)      (None, 22, 12, 32)        128
d6 (UpSampling2D)            (None, 44, 24, 32)        0
d7 (Conv2D)                  (None, 44, 24, 32)        9248
d8 (BatchNormalization)      (None, 44, 24, 32)        128
decoded (Conv2D)             (None, 44, 24, 1)         289
=================================================================
Total params: 610,833
Trainable params: 610,065
Non-trainable params: 768
```

The network most likely could be a lot shallower and have less parameters than the example considering the relative simplicity of the data. The layers starting with *e* belong to the encoder subnetwork and the ones starting with *d* correspondingly to the decoder part. The layer called *encoded* is the smallest layer in the middle and the one that outputs the

encoded representations. In this instance it has 16 neurons and would consequently create a 16-wide representation of the original 44*24 matrix.

The convolution layers used here are regular ”3D” convolutions implemented by Keras' **Conv2D** layer. *Depth-wise separable convolutional layers*, implemented by **SeparableConv2D** in Keras were also successfully used. They perform the convolution in two steps, first depth-wise running a 2D convolution kernel for each layer separately, then combining them using a 1x1 depth-wise kernel. This is computationally significantly more efficient than ”regular” convolution and a good choice for use cases in which the depth-wise interactions do not carry any significant information, like this one. It can even help the learning as it ignores these interactions, which would add nothing but noise. The depth-wise separable convolution is notably successfully used in constrained networks for mobile applications, such as **MobileNet**. To learn more about this operation and understand why it fits some use cases well, reader should refer to [32].

This network is trained like any other autoencoder network, by using noisy training data as inputs and clean data as outputs. After training, full dataset is converted into latent representations using a separate model created by taking all layers from first layer to the *encoded* layer. This model is referred to as *encoder* here. A *decoder* model is also created from the encoded layer to the decoded layer, but its only use is validating and debugging the results as it does nothing for the clustering.

Various constraints were used as regularization when training the network. Noisy input data created using the method described in 6.1.2 was used to predict the clean input. Augmenting the training data with said noisy transformations was used. The network's layers were regularized with batch normalization (*BatchNormalization* layers in the listing).

The embedding layer was heavily regularized with L1-regression (Lasso regression) to steer the network to learn a sparse representation of the data, i.e. rather than learning the trajectories as a combination of many features the network was constrained to rather learn sparse, more representative features. Autoencoder using this type of regularization is sometimes referred to as *sparse autoencoder* [19]. This type of regularization is discussed in more detail in section 4.5. One can see the effect of L1-regression best via inspecting the output of the regularized layer with different regularization values. This is done in figure 6.5 in which there are the outputs of latent layer given three random samples with different values for L1 activity regularization. The samples are in different columns and values for L1 are in different rows. The lowest factor is on the top, a reasonable middle value in the middle and high regularization on the bottom row. One can easily see the effect of Lasso regression from top to bottom as the number of positive neuron activations decline which is exactly what this type of regularization penalizes.

By choosing a reasonable hand-picked value for L1 regularization the network can be steered to learn more representative singular features and ignore less important features which can be beneficial for clustering. This is reflected also as declining accuracy of the reproduced input which can be seen in figure 6.6a. The original input is in the leftmost column and the following columns go from low L1 value to very high L1 regularization. The difference between low regularization and a reasonable middle value isn't that obvious but the highly regularized outputs represented by sparse activation in figure 6.5 have smudgier reconstructions, but still have some features reproduced with recognizable quality. The highly regularized network has very informative and discriminating features, which can mean that they fit well for clustering in the follow-up phase. The fact that the reconstructions of the unregularized and somewhat highly regularized networks are almost the same, but the features are far more sparse in the latter one, means that adding some

level of regularization to the latent space is almost certainly beneficial for clustering tasks.

Autoencoder network can also be constrained by reducing the width of its latent layer. Figuratively, we make the ”waist” of the network smaller and force the input space to be constrained into smaller representation, again forcing the network to learn more informative representations and ignore noise. This could also be thought as a form of regularization although it’s inherent to how autoencoders work. Figure 6.6b shows the reconstructed input with different widths of the latent layer. Again as with the L1 values, the difference between the widest and middle one is not that clear but the smallest network with only 8 neurons has started to lose significantly more information in the reproduction. It must be noted that this is not a proper way to evaluate the final result of deep clustering pipeline as even though information is lost, the learned representations might be better fitted for clustering.

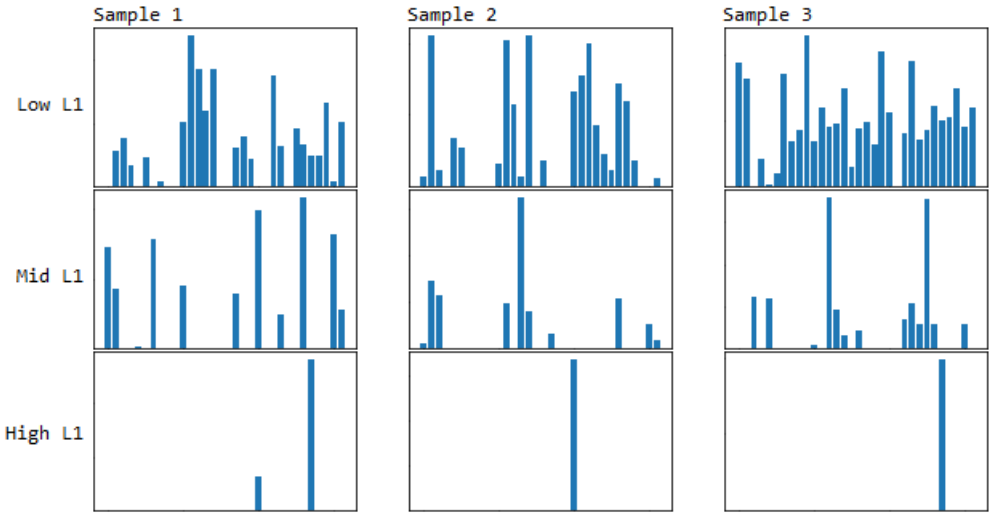
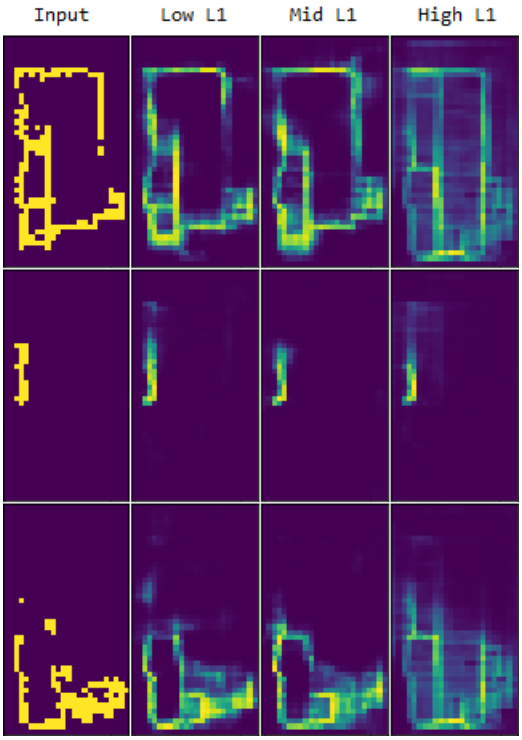
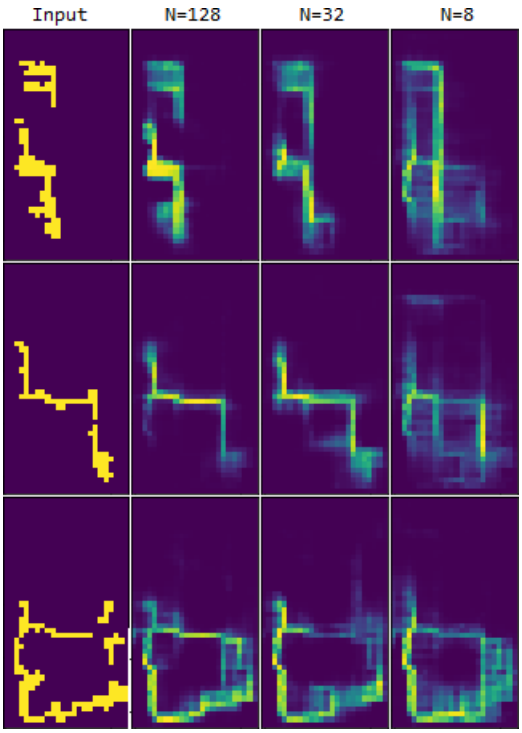


Figure 6.5: Output of latent layer on some randomly sampled inputs with different L1 factors



(a) Reconstructed inputs (DS2) with various L1 regularization values for latent layer output



(b) Reconstructed inputs (DS2) with different latent space sizes

Figure 6.6

6.3.4 Clustering

After the representation learning step has been completed and the original dataset is transformed into a lower dimensional representation the data can be fed into the clustering algorithm. This ended up being significantly easier and straight-forward than the representation learning phase. The clustering algorithm can be any that can work with the vector of continuous values and matches all other criteria dictated by the task. Here, DBSCAN and HDBSCAN were tested. All of the same considerations that were discussed with said methods in section 4.4.5 apply here. The main difference between the two is that DBSCAN has more parameters, of which the most important is ϵ which decides the size of the ”neigborhood” considered for each point. HDBSCAN does not have this parameter, but when the algorithm failed there was less parameters to adjust, meaning that

DBSCAN ended up being the algorithm of choice although both demonstrated mostly good performance.

Figures 6.7 and 6.8 show clusters found with DBSCAN and HDBSCAN in dataset *DS 1*, respectively. The number indicates the number of trajectories in that cluster. The clusters look visually good and reasonable. Both results have a single cluster that includes most of the trajectories, but knowing the source data this cluster seems reasonable as well.

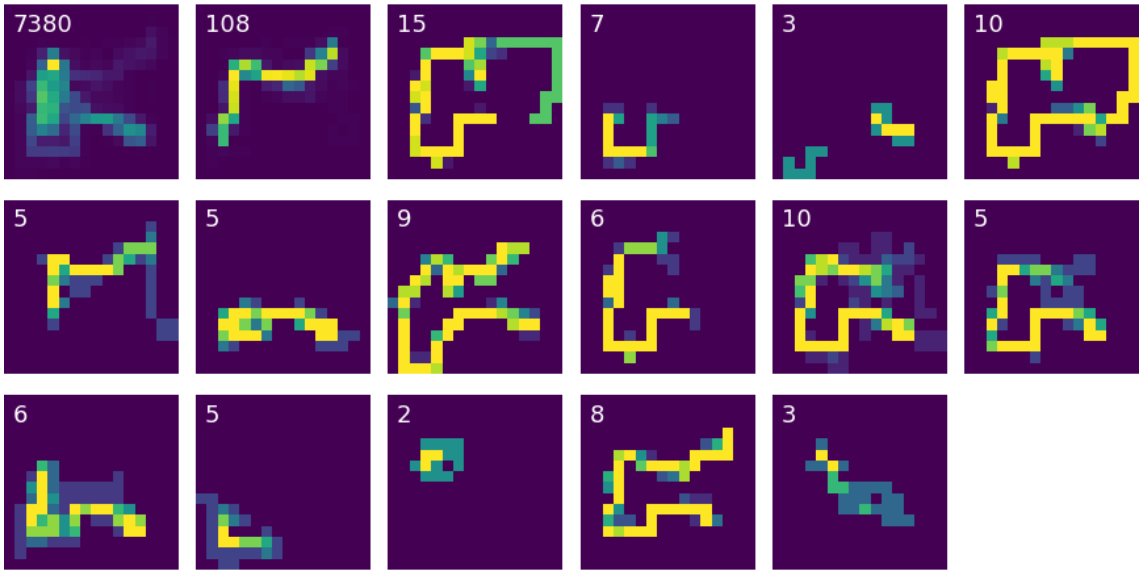


Figure 6.7: Clusters created with DBSCAN

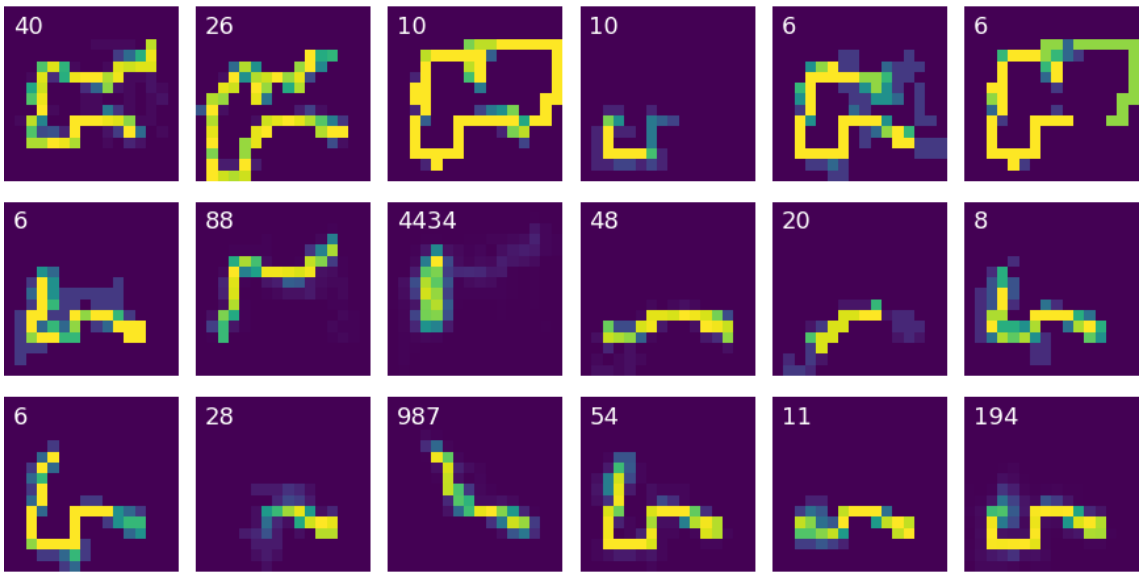


Figure 6.8: Clusters created with HDBSCAN

Preprocessing with **T-SNE** (discussed in section 5.2) was found to be a very useful preprocessing step for the embedded representations before clustering. By the best of our knowledge, T-SNE is not found in relevant literature being used for such a purpose, rather T-SNE is often ignored as a preprocessing step and used rather for visualization for manual inspection purposes. It was found that giving the embedded representations directly to a clustering algorithm such as DBSCAN resulted in poorer clustering in complex environments (especially in *DS 2*) and the parameters for the model were hard to adjust. This is probably mostly due to many clustering algorithms performing poorly in high dimensions. Increasing the width of the latent layer would also increase the dimensionality of the input data, meaning that without additional dimensionality reduction we may be forced to use undesirably small latent layer in the network, limiting the configuratibility of the model.

T-SNE was employed initially in this work for visualizing the embedded representations, but it was found that algorithms such as DBSCAN and HDBSCAN performed well without extensive manual adjustment using this data as input and resulted in better looking

clusters and less trajectories marked as noise. The latent dimension width could be increased to numbers such as 16, 32 or more from the original and T-SNE was used to project this representation down to just a few dimensions (2, 3 or 4) before clustering it. It is assumed that this works well especially for clustering tasks because T-SNE strives to preserve the distance of data points close to each other in the lower dimensional representation, thus acting as a sort of locality preserving transformation. It performs a kind of preliminary clustering before the actual clustering, creating ”islands” of similar data. It must be noted however that this results in some loss of information, but in turn makes the model adjustment significantly easier.

Figure 6.9 shows the T-SNE representation space with cluster assignments. It is obvious why clustering is easier to make from low-dimensional representation like this. Two dimensions are good for visualization, but actual clustering could and should be done in higher dimensions, in three or more to avoid losing too much discriminating information.

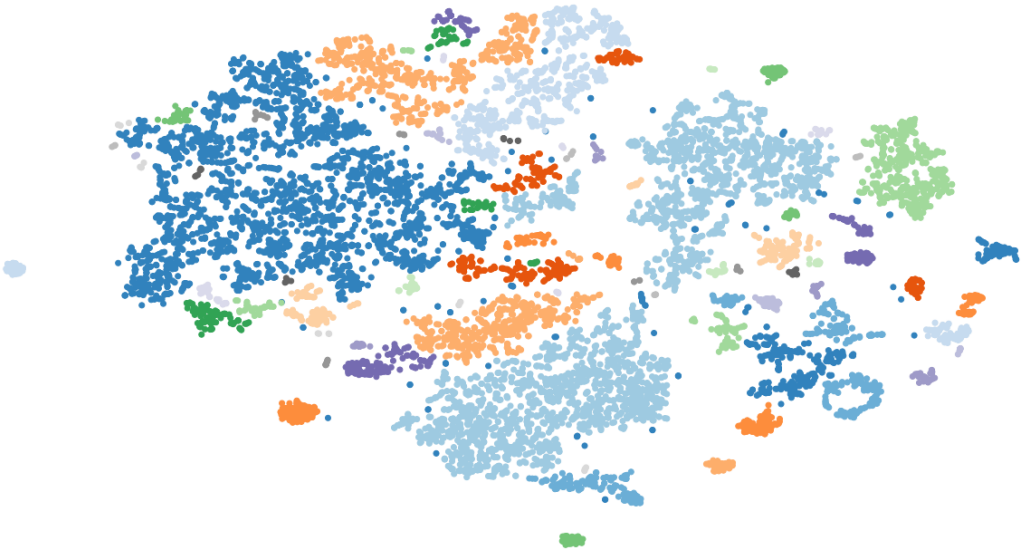


Figure 6.9: T-SNE output in 2 dimensions with cluster assignments from DBSCAN in different colors. Blue points separate from clusters are data marked as noise.

6.3.5 High level insights

After the clustering has been completed in a satisfactory way, the resulting clusters can be used to find high level insights about the underlying data and turn it into useful knowledge. This is another very wide subject and there exists a plethora of methods to continue the analysis, some which have already been presented in previous chapters. This section will discuss a few examples.

Clusters can be visualized in various ways. An easy and informative way is to create an intensity heatmap of all paths belonging to a cluster, this can be seen utilized in figures 6.6 and 6.11. This is achieved trivially by just calculating the sum of the 2D matrices resulting in a single matrix in which the number represents the intensity of activity in the area represented by that grid cell. The clusters can be plotted on the original blueprint or the transformed grid space, all at once (with different color per cluster) or one by one, whichever results in the best image. Most often in experiments done for this thesis this type of visualization resulted in clusters with the ”main trajectory” more illuminated and the dimmer auxiliary trajectories around it. This gives a good idea of the cluster at a glance, the way how the tags inside it acted and also how much variance there is inside the cluster.

T-SNE can be employed to visualize the embedding space. This helps to assess the quality of the representations, whether they create divisive enough features and such. This can also be used as an effective preprocessing step as described in the previous section. It also gives a high level visualization of the entire dataset in a way that can not be achieved in the original data space, comparable to the multidimensional scaling result in figure 5.1. This can be used in many ways in subsequent analysis. In figure 6.9 trajectories from *DS2* are transformed by the encoder down to 32 dimensions. This is then projected down to 2 dimensions for visualization using T-SNE. One can see the embedded representations

are good and result in good discrimination between different clusters. Colors represent cluster assignments made using HDBSCAN.

Another way to assess the properties of the clusters is to inspect the latent representation of samples belonging to a single cluster. This can be seen in figure 6.11, with the mean neuron activation of a cluster with matching mean trajectory embedded in the same subplot. This gives a very visual way to assess the quality of the embedding space and to investigate the similarity of clusters. One can observe in the figure that clusters are generally dominated by one or a few neurons. These features represent the characteristic behavior of that cluster. It can also be observed that clusters having similar features (partly sharing same trajectories) do have high activation in the same neurons. The same could also be observed on a wider scale in data not included in this figure. Figure 6.10 shows the mean activation for trajectories labelled as noise by the clustering algorithm. This has clearly a more even distribution, meaning that it does not represent any distinct features, which sounds reasonable.

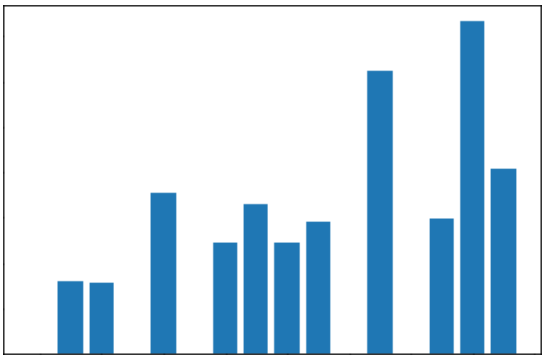


Figure 6.10: Mean latent layer output for cluster marked as noise from HDBSCAN (DS2, HDBSCAN).

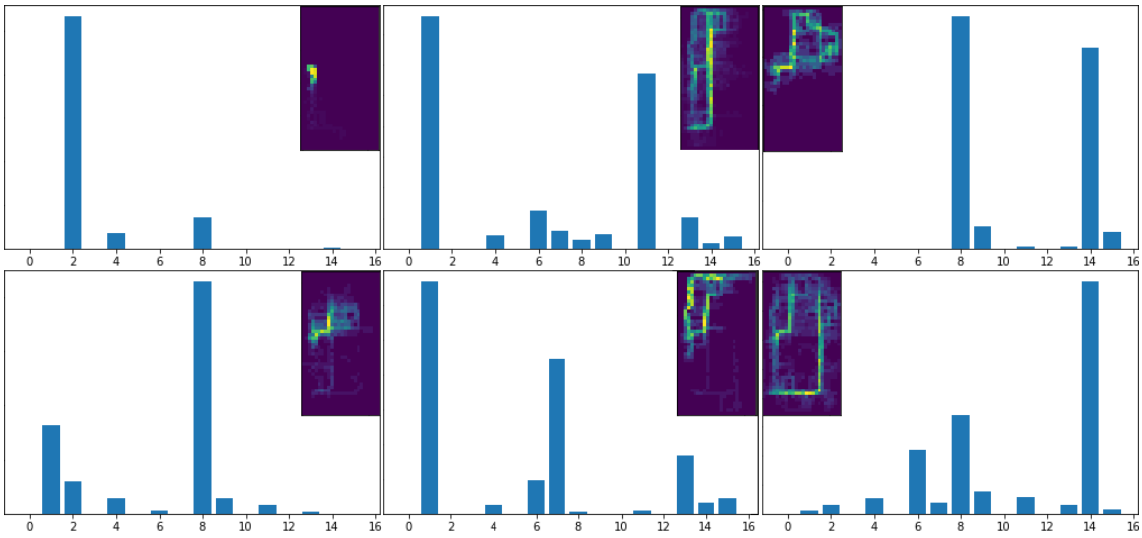


Figure 6.11: Cluster's trajectories' mean intensity with respective mean output of latent layer (DS2, HDBSCAN).

Using this clustering as a starting point, a wide array of further steps not in this thesis' scope could be taken. This could include for example projecting the clustering into a hierarchical model using the distillation/global surrogate process discussed in 5.1.1 and 5.1.2. The clusters could also be used to create predictive models that predict the class of the target while it is still on the trajectory. This could have many practical use cases.

6.3.6 Anomaly detection

Alongside the main clustering task, the autoencoder network was also experimentally used for *anomaly detection*, a task which is shortly described in section 4.5. As the neural network learns the dataset's intrinsic patterns the reconstruction error of the model can be used as a score that determines how "common" the sample is in the dataset. Figure 6.13 displays some of the trajectories from the dataset with their respective reconstruction errors (using mean absolute error (MAE)). Figure 6.12 displays the distribution of reconstruction errors. It can be observed that majority of the dataset (*DS 1*, here) consists of very short paths and the more complex and longer the trajectory gets, the higher its error value is. This is very dependent on the dataset though and has nothing to do with the

length of the trajectory. The most common trajectories will have the lowest error values.

This error value can be utilized in various ways. One way would be in preprocessing to filter out the most common, and thus maybe uninteresting paths. This requires manually finding a good cutoff treshold value, below which the paths would be removed from the dataset. If the dataset is very skewed towards the common case, this could prove a useful step, but no further reserach was done in this direction. This method can of course be trivially used to discover common behavior in the environment if that's important.

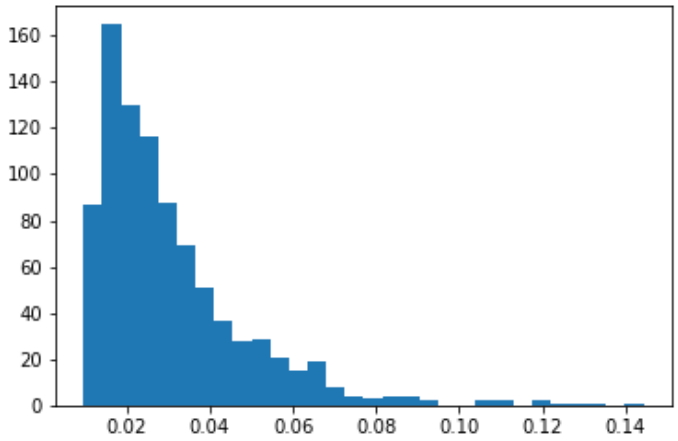


Figure 6.12: Distribution of reconstruction errors in *DS 1*

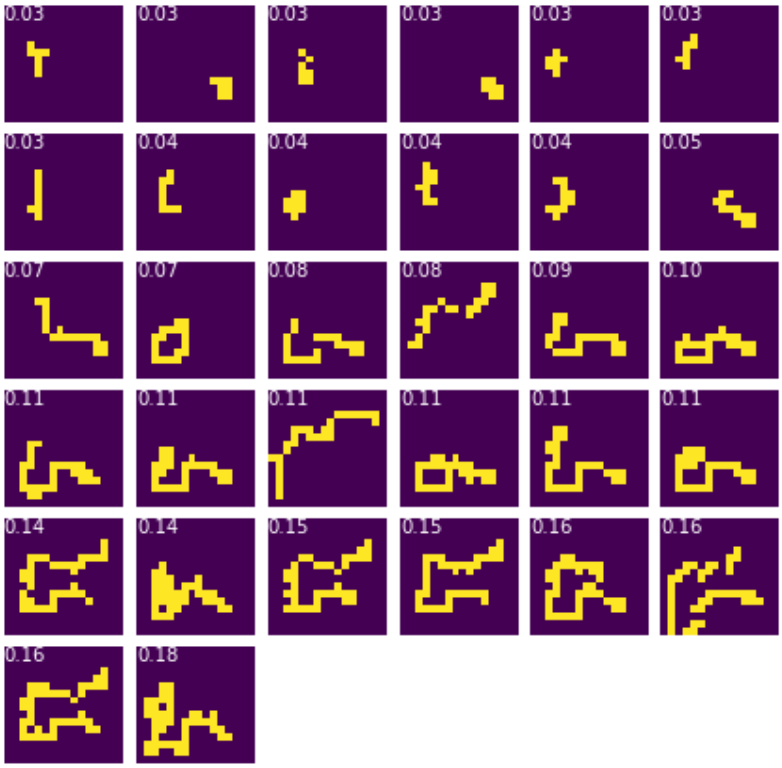


Figure 6.13: Some trajectories from *DS 1* with their respective reconstruction errors

Chapter 7

Conclusions

This thesis explored the world of machine learning in the context of indoor localization, focusing on the problem of clustering trajectories from an indoor localization system. In the first chapters of the thesis indoor localization and machine learning were presented as concepts and their basic technological components were concisely introduced. Later chapters went into more detail in exploring various machine learning and data analysis technologies relevant to the research problem, later put into action to develop two new clustering frameworks.

The main research problem was to bundle indoor trajectories demonstrating similarity together, a task more widely referred to as clustering. Indoor localization systems produce very large amounts of data of variable quality and thus such processing steps are needed to transform the raw data into usable form. This kind of analysis has many direct use cases including floor layout planning, congestion control and customer analytics to name a few prominent ones. The growing popularity and utilization of indoor localization systems calls for advanced methods of analysing the data they produce, creating a strong business case for continued research on the subject.

There is a good amount of existing literature on machine learning and indoor localization

but not on utilizing machine learning on indoor localization data. Most of the existing literature was research-oriented and practical experiments were very idealized and had limited applicability in real world scenarios. This thesis introduced novel algorithms developed based on data from a widely deployed real indoor localization system and is thus validated on real business case from the beginning.

Two different new frameworks for trajectory clustering were developed in the context of this thesis and described in detail in the text. First, based on a framework described in source literature, improved upon this system by introducing new distance metrics and tackling many issues that were left open or did not translate from the original system to more complex scenarios. Second, a neural network based *deep clustering* system was developed for more complex use cases. This included many innovations, including the use of *T-SNE* as an intermediary processing step. Both of these approaches resulted in systems that group similar trajectories together by their properties with good efficiency. The output allowed the operator to gain insights on the general movement patterns present in the site through visualizing the results. These systems already demonstrated good performance but they also act as ready-to-use platforms for continued research and applications. Many potential research directions and improvements were discussed and provide a good starting point to develop more advanced systems.

References

[1] F. Zafari, A. Gkelias, and K. K. Leung. A survey of indoor localization systems and technologies. *IEEE Communications Surveys Tutorials*, 21(3):2568–2599, thirdquarter 2019.

[2] Yann LeCun, Y Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[3] T. S. Prentow, H. Blunck, K. Grønbæk, and M. B. Kjærgaard. Estimating common pedestrian routes through indoor path networks using position traces. In *2014 IEEE 15th International Conference on Mobile Data Management*, volume 1, pages 43–48, July 2014.

[4] Salvatore Marano, Wesley Gifford, Henk Wymeersch, and Moe Win. NLOS identification and mitigation for localization based on UWB experimental data. *Selected Areas in Communications, IEEE Journal on*, 28:1026 – 1035, 10 2010.

[5] Tom M. Mitchell. *Machine learning*. MacGraw-Hill, 1997.

[6] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition, 2017.

[7] Francois Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.

[8] Andriy Burkov. *The hundred-page machine learning book*. Andriy Burkov, 2019.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[10] Dong Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2, 08 2015.

[11] Elie Aljalbout, Vladimir Golkov, Yawar Siddiqui, Maximilian Strobel, and Daniel Cremers. Clustering with Deep Learning: Taxonomy and New Methods. *arXiv e-prints*, page arXiv:1801.07648, Jan 2018.

[12] N Rajalingam and K Ranjini. Hierarchical clustering algorithm - a comparative study. *International Journal of Computer Applications*, 19, Apr 2011.

[13] Jarkko Mikael Piironen. Density-based clustering. Master's thesis, University of Eastern Finland, 2018.

[14] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 49–60, New York, NY, USA, 1999. Association for Computing Machinery.

[15] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. In Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, editors, *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. Anomaly Detection using Autoencoders in High Performance Computing Systems. *arXiv e-prints*, page arXiv:1811.05269, Nov 2018.

[17] Chong Zhou and Randy C. Paffenroth. Anomaly detection with robust deep autoencoders. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD 17*, 2017.

[18] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 06 2014.

[19] E. Min, X. Guo, Q. Liu, G. Zhang, J. Cui, and J. Long. A survey of clustering with deep learning: From the perspective of network architecture. *IEEE Access*, 6:39501–39514, 2018.

[20] Carl Doersch. Tutorial on Variational Autoencoders. *arXiv e-prints*, page arXiv:1606.05908, June 2016.

[21] Matthew D Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. *arXiv e-prints*, page arXiv:1311.2901, Nov 2013.

[22] Luke Taylor and Geoff Nitschke. Improving Deep Learning using Generic Data Augmentation. *arXiv e-prints*, page arXiv:1708.06020, Aug 2017.

[23] Nicholas Frosst and Geoffrey E. Hinton. Distilling a neural network into a soft decision tree. *CoRR*, abs/1711.09784, 2017.

[24] Christoph Molnar. *Interpretable machine learning: a guide for making Black Box Models interpretable*. Lulu, 2019. https://christophm.github.io/interpretable-ml-book/ (accessed 2020-11-01).

[25] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv e-prints*, page arXiv:1503.02531, Mar 2015.

[26] Sungkyu Jung. Stat 2221: Advanced applied multivariate analysis. https://www.stat.pitt.edu/sungkyu/course/2221Spring15 (accessed 2020-11-01).

[27] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 11 2008.

[28] Jiang Bian, Dayong Tian, Yuanyan Tang, and Dacheng Tao. A survey on trajectory clustering analysis. *arXiv e-prints*, page arXiv:1802.06971, Feb 2018.

[29] Thomas Eiter and Heikki Mannila. Computing discrete fréchet distance. Technical report, 1994. http://www.kr.tuwien.ac.at/staff/eiter/et-archive/cdtr9464.pdf (accessed 2020-11-01).

[30] Kai Tian, Shuigeng Zhou, and Jihong Guan. Deepcluster: A general clustering framework based on deep learning. In Michelangelo Ceci, Jaakko Hollmén, Ljupčo Todorovski, Celine Vens, and Sašo Džeroski, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 809–825, Cham, 2017. Springer International Publishing.

[31] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv e-prints*, page arXiv:1712.06567, December 2017.

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*, page arXiv:1704.04861, April 2017.