Faculty of Science and Technology
Department of Physics and Technology

# Deep Image Clustering with Tensor Kernels and Unsupervised Companion Objectives

—

**Daniel Johansen Trosten**

*FYS-3941 Master's thesis in applied physics and mathematics 30 SP – June 2019*

## Abstract

Deep image clustering is a rapidly growing branch of machine learning and computer vision, in which deep neural networks are trained to discover groups within a set of images, in an unsupervised manner. Deep neural networks have proven to be immensely successful in several machine learning tasks, but the majority of these advances have been in supervised settings. The process of labeling data for supervised applications can be extremely time-consuming, or even completely infeasible in many domains. This has led researchers to shift their focus towards the deep clustering field. However, this field is still in its infancy, meaning that it includes several open research questions, regarding e.g. the design and optimization of the algorithms, the discovery of meaningful clusters, and the initialization of model parameters.

In an attempt to address some of these open questions, a new algorithm for deep image clustering is developed in this thesis. The proposed Deep Tensor Kernel Clustering (DTKC) consists of a convolutional neural network (CNN), which is trained to reflect a common cluster structure at the output of all its intermediate layers. Encouraging a consistent cluster structure throughout the network has the potential to guide it towards meaningful clusters, even though these clusters might appear to be nonlinear in the input space. The cluster structure is enforced through the idea of companion objectives, where separate loss functions are attached to each of the layers in the network. These companion objectives are constructed based on a proposed generalization of the Cauchy-Schwarz (CS) divergence, from vectors to tensors of arbitrary rank. Generalizing the CS divergence to tensor-valued data is a crucial step, due to the tensorial nature of the intermediate representations in the CNN. Furthermore, an alternate initialization strategy based on self-supervised learning, is also employed. To the author's best knowledge, this is the first attempt at using this particular self-supervised learning approach to initialize a deep clustering algorithm.

Several experiments are conducted to thoroughly assess the performance of the proposed DTKC model, with and without self-supervised pre-training. The results show that the models outperform, or perform comparable to, a wide range of benchmark algorithms from the literature.

**Acknowledgments**

First of all, I would like to thank my supervisor Professor Robert Jenssen, and co-supervisor Michael C. Kampffmeyer, for their guidance, their enthusiasm, and their willingness to wade through the endless river of questions asked by me while writing this thesis.

I would also like to thank my fellow students, and especially Andreas and Mads. Our collaboration and discussions have proven to be an invaluable source of inspiration, motivation, and knowledge.

Lastly, to my family, and to Eva: Thank you for your love and support.

Daniel Johansen Trosten,
Tromsø, June 2019.

# Table of Contents

# List of Tables

# List of Figures

# Abbreviations

| | |
|---|---|
| **ACC** | unsupervised clustering ACCuracy |
| **CNN** | Convolutional Neural Network |
| **CS** | Cauchy-Schwarz |
| **CVI** | Cluster Validity Index |
| **DDC** | Deep Divergence-based Clustering |
| **DEC** | Deep Embedded Clustering |
| **DNN** | Deep Neural Network |
| **DTKC** | Deep Tensor Kernel Clustering |
| **FC** | Fully-Connected |
| **KDE** | Kernel Density Estimation |
| **KL** | Kullback-Leibler |
| **MLP** | MultiLayer Perceptron |
| **NMI** | Normalized Mutual Information |
| **RKHS** | Reproducing Kernel Hilbert Space |
| **RNN** | Recurrent Neural Network |
| **SS** | Self-Supervised |
| **SVD** | Singular Value Decomposition |

# Part I / Introduction

# 1. What is machine learning?

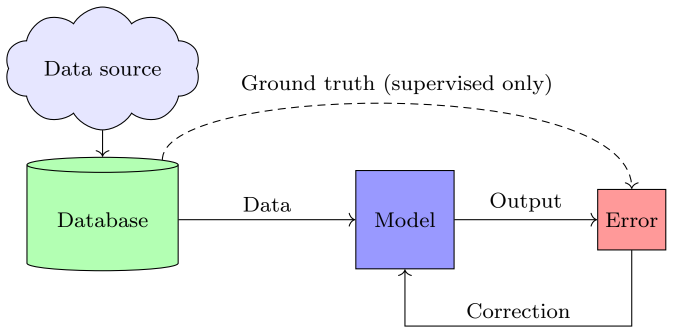


Figure 1: *An illustration of a generic machine learning system.*

*Machine learning* is a field of study which resides in the intersection between mathematics, statistics, and computer science. It is also considered to be an integral part of the general field of *artificial intelligence*, as machine learning components would allow these intelligent systems to learn from the environment they inhabit. The basic principles of machine learning have been summarized by several textbook authors [1, 2, 3, 4], and although there are some differences in phrasing and terminology, these authors all succeed in conveying the same message about the purpose of the machine learning field. Namely that it considers the design of computer systems for performing specific tasks, without them being explicitly programmed with this exact task in mind. This is achieved by constructing the system such that it can *learn* from examples, making it capable of adapting to the current task at hand. The examples are often referred to as the *training data*, and are – together with the learning algorithm itself – crucial components of all machine learning systems.

The potential of this symbiotic relationship between data and algorithm is precisely why we have seen massive developments in machine learning in recent years. Our society is in a constant state of digitalization, and the generation and collection of massive amounts of data is a natural byproduct of this process. The size of the datasets has long since outgrown the capabilities of manual analysis and explicit programming, which means that the design of automated systems is absolutely necessary when these datasets are to be translated into actionable insight. The digitalization has also brought with it an increase in computational capacity, which has led researchers to design increasingly complex systems for dealing with the ever growing amounts of data.

Figure 1 shows a conceptual overview of a machine learning system. The data is collected, and passed to the model, which then produces a set of outputs. The outputs are processed by a *cost function* or *loss function*, whose objective is to quantify the error made by the system. This error is in turn used to compute potential corrections that should be made to improve the model's performance. The loss function can either be a function of the output only, or it can be computed by comparing the output to a collection of desired responses, referred to as the *ground truth*. The difference between these two approaches is described in more detail below.

The various machine learning tasks can be roughly divided into four different categories [1, 3]. These are:

- *Supervised learning*: The training data consists of input-output pairs, and the system is trained to reproduce the correct output for each input. In this case we refer to the output examples as "ground truth" or "labels", and say that the training data is labeled.
- *Unsupervised learning*: The training data consists of only input observations. The training data is therefore said to be "unlabeled". Due to the lack of output examples, it is up to the machine learning practitioner to specify which type of output one wishes the system to produce. This is usually done through the design of the algorithm.
- *Semi-supervised learning*: A combination of supervised learning and unsupervised learning. In this case the training data consists of both input-output pairs, as well as observations where only the input is available.
- *Reinforcement learning*: The machine learning system represents an agent capable of interacting with a specific environment, whose goal is to maximize some reward. The input to the system is information about the environment, and the output is a sequence of actions. The system receives information about the reward for a particular set of actions, and has to adapt according only to this information.

# 2. Clustering

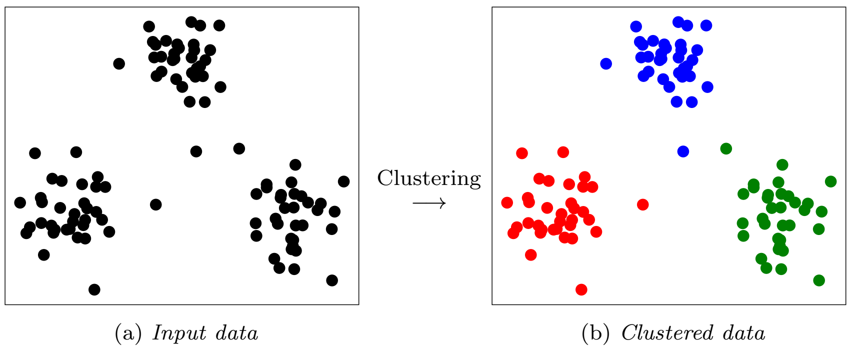


Figure 2: *An illustration of the clustering process for two-dimensional points. The "natural" groups in the input data are automatically discovered by the system, and shown here with different colors.*

In this thesis we will study a particular branch of unsupervised learning, namely clustering, which refers to the process of discovering underlying group structure in an unlabeled dataset. The process is illustrated for a simple two-dimensional dataset in Figure 2. The supervised counterpart to clustering, classification, requires labeled data. However, when the labeling process is not an intrinsic property of the data generating system, the labeling has to be done manually to make supervised learning a viable option. Although several datasets have been manually labeled for the design of classification systems, it remains a prohibitive task in many fields, especially within those containing massive amounts of data. This makes clustering the only feasible option in these fields. The ability to identify and quantify potential groupings in a dataset has proven to be immensely useful in a wide range of applications including, but not limited to, image segmentation and text processing [5], medical image analysis [6], satellite image segmentation [7], texture analysis [8], and several others.

The need to understand the massive amounts of unlabeled data has led to the development of a multitude of clustering algorithms. Comprehensive reviews of many of these algorithms can be found in e.g. [9] or [10]. In the last couple of years, the clustering field has seen a shift in methodology towards methods based on *Deep Learning* [11, 12], resulting in the birth of the *Deep Clustering* subfield. Deep learning architectures, also referred to as deep neural networks, are known for their versatility, as well as their ability to handle more complex data types, such as digital images or multivariate time series [13, 14].

Images and time series are examples of data types that cannot necessarily be said to reside in a vector space, which is an assumption made by many of the classical clustering algorithms. Deep clustering algorithms seek to leverage the representational power of deep learning architectures in an unsupervised manner, aiming to translate the success of supervised deep learning to the domain of unsupervised learning. This translation has been identified as a main next goal for machine learning research [11]. Recently, state-of-the-art results have been reported for complex vectorial data [15, 16, 17, 18], digital images [19, 20, 21], and multivariate time series [22]. A review of some of these methods is provided in Part III.

## 2.1. Key challenges in the current deep clustering scene

The process of developing the current state-of-the-art deep clustering techniques has posed many important questions and challenges along the way. Some of those considered to be most important are the following:

- **How should the deep clustering models be optimized?** The most common approach to deep clustering has been to use some deep neural network to embed each input observation in a vector space, and then pass these embeddings to a clustering module, whose output is the final prediction of the model. The deep neural network and the clustering module can then be jointly optimized to improve the clustering result, by minimizing some unsupervised loss function, computed at the output of the network. However, we do not know if this is the best approach to combining the deep neural network and the clustering module. There is also some uncertainty regarding the specification of the loss function, which can be observed in the more general field of clustering as well. The size of the literature and the large amount of proposed clustering methods are consequences of the fact that the mathematical specification of the "optimal clustering" can be somewhat challenging [23].

- **How do we make sure that the clusters we find actually make sense in the input space?** This question is somewhat loosely formulated, but it amounts to enforcing some preservation of cluster structure between the embedding space, which was discussed in the preceding question, and the input space [24]. If our clustering is based on perceived cluster structure in the embedding space, it is crucial that this structure reflects some meaningful cluster structure in the input space as well.

- **How do we quantify cluster structure for images and image representations?** One approach to encourage the preservation of cluster structure throughout the deep neural network is to enforce a common cluster structure at the output of the layers in the network. Convolutional neural networks (CNNs) [25] have been the go-to deep learning architecture for image clustering [19, 20, 21]. However, as we will see later, the layers of a CNN do not produce vectorial-representations, meaning that one cannot rely on vector-based methods from the literature to quantify the cluster structure for these representations.

- **How should we initialize the parameters of the deep neural networks?** Deep neural networks have to be optimized starting at some initial point in the parameter space. It was pointed out by e.g. Kampffmeyer et al.

[20] that randomly initializing the parameters of their clustering network made it prone to getting stuck in local optima, resulting in sub-optimal clusterings. Other works have focused on the use of autoencoders to alleviate some of the difficulties of training randomly initialized networks in an unsupervised manner [15, 24], but their results do not by any means indicate that the problem is "solved".

To the author's best knowledge, these questions still remain as open research topics in the field of deep clustering. The potential impact of advancements within these topics constitutes the motivational foundation for this thesis.

## 3. Contributions

The focus of this thesis will be on image clustering. Digital images are an extremely common data type, and the processing of these images makes up the very backbone of systems for e.g. autonomous driving [26, 27], medical imaging [28], and remote sensing [29]. Many of these systems rely on CNNs [25] – a particular neural network architecture which has proven to be very successful within the field of computer vision [13, 30]. These networks have a strong connection to the domain of tensor theory, with early work including e.g. [31, 32, 33]. However, the research on this topic is still very much in its early stages.

As an answer to the questions outlined above, the connection between CNNs and tensors is utilized to construct a model for image clustering. This model is referred to as Deep Tensor Kernel Clustering (DTKC), and draws inspiration from Deep Divergence-based Clustering (DDC) [20]. The objective of DTKC is not only to cluster the images, but to do so under constraints on the cluster structure of the intermediate layers, as well as the output layer. This is done by leveraging the idea of companion objectives [34], coupled with tensor kernels [35], and information theory. The recent concept of self-supervised learning [36] is also explored as an initialization strategy. The key contributions of the thesis are summarized as follows:

- The Cauchy-Schwarz (CS) divergence [37] is a in information theoretic measure which is central in DDC. In this thesis, the CS divergence is generalized from vectors to tensors of arbitrary rank, using kernel density estimation [38], and tensor kernels [35]. This allows us to describe the cluster structure in a tensor-valued dataset, by considering the divergence between probability density functions representing the respective clusters.

- The generalized CS divergence is used to construct an unsupervised companion objective for each of the layers in a convolutional neural network. These companion objectives are integral parts of DTKC, and are essentially terms which are added to the final loss function, that enforce a consistent cluster structure in earlier layers of the CNN.

- Self-supervised learning, as described in [36], is employed as an initialization strategy for the proposed model, resulting in DTKC-SS. To the author's best knowledge, this is the first attempt at using this technique to initialize deep clustering models.

Figure 3 shows an overview of the contributions and how they relate to their respective domains. The ideas originate from recent developments in machine learning, as well as advanced concepts from mathematics and statistics.

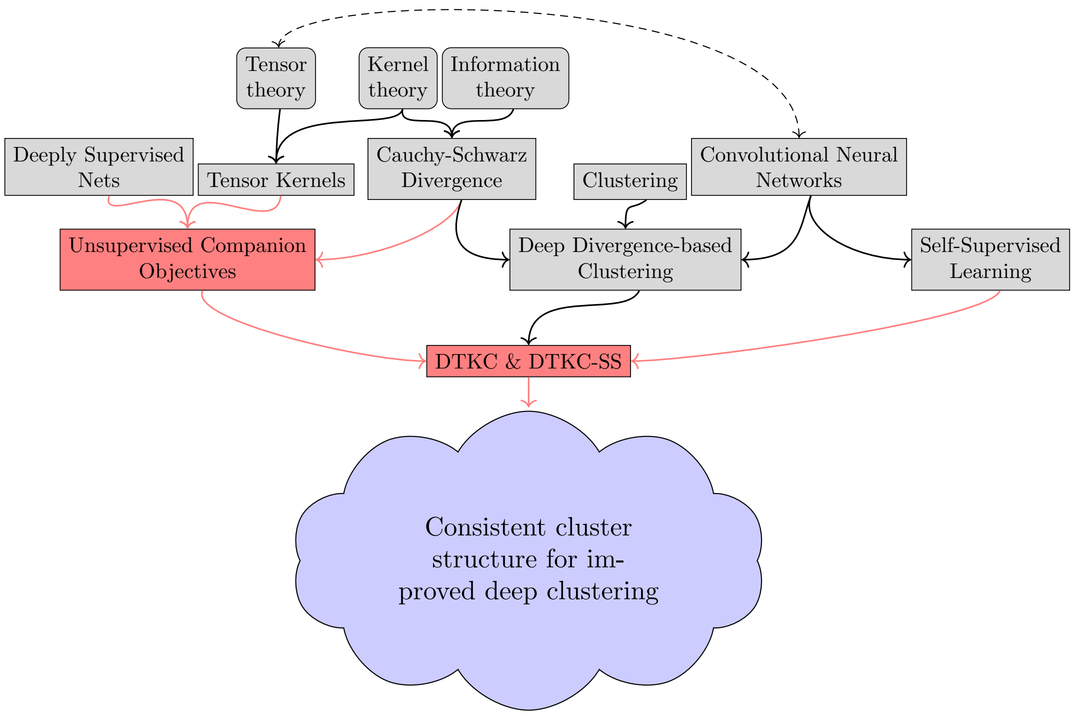

Tensor theory
Kernel theory
Information theory
Deeply Supervised Nets
Tensor Kernels
Cauchy-Schwarz Divergence
Clustering
Convolutional Neural Networks
Unsupervised Companion Objectives
Deep Divergence-based Clustering
Self-Supervised Learning
DTKC & DTKC-SS
Consistent cluster structure for improved deep clustering

Figure 3: *Diagram relating the main contributions (red squares and arrows) and the domains they come from.*

## 4. Thesis outline

After this introduction, we will proceed to Part II which covers the basic definitions and terminology of the clustering field. This part also includes a review of clustering algorithms that do not belong to the deep clustering field, as they remain relevant for a thorough understanding of the clustering problem.

Part III will take us closer to the deep clustering scene, as it begins with an introduction of the most frequently used deep learning architectures. These will also be revisited in later parts, both in theoretical and experimental contexts. The final section of Part III builds on the understanding of these models, and introduces a several deep clustering algorithms.

The theoretical and methodological contributions of this thesis are described in Part IV. Here we will go through the relevant information theory, as well as the theory on tensor kernels. This part ends with a thorough explanation of the proposed DTKC model.

Part V covers the experiments performed to evaluate DTKC. This part describes the datasets, model implementation, and provides all the experimental results. The section containing the results is designed to thoroughly assess the effects of the unsupervised companion objectives and self-supervised pre-training. It is therefore divided into subsections, where the goal of each subsection is to investigate and discuss very specific aspects of the proposed model. These analyses are followed by a discussion whose aim is to address the more general observations made during the experimental process, as well as to provide some thoughts on future work, and the outlook of the contributions made in this thesis.

Part VI gives some concluding remarks, summarizes the contributions, the outcome of the experiments, and what impacts they have on the future of the deep clustering field.

## 5. Notes from the author

A paper related to the work described in this thesis has been published by the author in the 2019 IEEE International Conference on Acoustics, Speech, and Signal Processing. The paper proposes RDDC [22], which is a model for deep clustering of variable length time series, based on recurrent neural networks and DDC. A second paper which is related to unsupervised image processing, is scheduled for publication in the 2019 Scandinavian Conference on Image Analysis. This paper proposes UCSN [39], which is a CNN-based model for unsupervised feature extraction from images.

# Part II / Clustering

In this part we will go through some of the background material that is necessary to understand the clustering problem. This will also include a definition of clustering which is more formal than what was given in the preceding part, which helps us build a consistent framework for terminology and mathematical notation. Following this will be a brief overview on data transformation, and a discussion on the quantitative evaluation of clusterings – the latter of which introduces the important concept of *cluster validity indices* (CVIs).

In the last section of this part we will go through several of the well-known clustering algorithms in the literature. Although these algorithms do not fall within the field of deep clustering, they are included as they represent important advancements in the more general clustering field. They also provide some relevant background and motivation for the deep clustering methods introduced later.

## 6. Definitions and background theory

Many different clustering definitions have been formulated throughout the years, all of which come with their own strengths and weaknesses regarding the formalization of the problem, and the formulation of specific clustering algorithms. The following definitions related to clustering are similar to those given by Theodoridis and Koutroumbas [1], and provide a general framework which can be further expanded upon when specific algorithms are introduced.

**Definition 1.** A *clustering* of a dataset $\mathcal{X} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$ is a partitioning of $\mathcal{X}$ into $k$ sets, $\mathcal{C}_1, \dots, \mathcal{C}_k$. These sets are commonly referred to as *clusters*. In general, the partitioning can be performed such that the sets $\mathcal{C}_1, \dots \mathcal{C}_k$ are either fuzzy (soft) or crisp (hard), producing fuzzy clusterings or crisp clusterings, respectively. For a given clustering to be valid, it should also satisfy the following:

(i) $\mathcal{C}_j \neq \emptyset, \; j = 1, \dots, k$

(ii) $\bigcup_{j=1}^{k} \mathcal{C}_j = \mathcal{X}$

(iii) $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset, \; i \neq j, \; i, j = 1, \dots, k$ (Hard clusterings only).

In the case of a fuzzy clustering, we need to introduce some object which determines the grade of an observations membership in a specific cluster. This object is the cluster membership function, and is defined as follows:

**Definition 2.** The *cluster membership function* $\mathcal{U}_j : \mathcal{X} \to [0,1]$ for a cluster $\mathcal{C}_j$, is a function which takes an observation as input, and outputs the grade of membership for the given observation in the cluster $\mathcal{C}_j$. Additionally, we require that the grade of cluster memberships for a specific data point over a set of clusters is normalized, such that: $\sum_{j=1}^{k} \mathcal{U}_j(\boldsymbol{x}_i) = 1$.

From this last definition it can be seen that hard clustering actually is a special case of soft clustering. If we take the range of $\mathcal{U}_j$ to be the set $\{0,1\}$ instead of the interval $[0,1]$, we recover the binary "in" (1) or "not in" (0) options. In this case, condition (iii) of Definition 1 is actually equivalent with the normalizing condition of Definition 2.

When introducing specific algorithms this formulation of hard clustering will be used to somewhat ease the mathematical notation, and help bridge the gap between soft and hard clustering, when such a transition is necessary. To further aid this cause, it is useful to introduce the cluster membership vectors, and the cluster membership matrix:

**Definition 3.** The *cluster membership vector* for a given observation $\boldsymbol{x}_i$ and a given clustering $\mathcal{C}_1, \dots, \mathcal{C}_k$ is the vector:

$$\boldsymbol{u}_i = \begin{bmatrix} u_{i1} & \dots & u_{ik} \end{bmatrix}^T$$

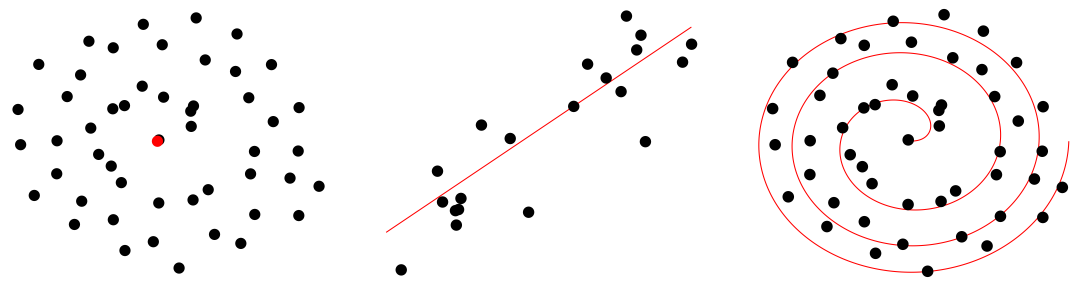
where the shorthand $u_{ij} = \mathcal{U}_j(\boldsymbol{x}_i)$ is used.

**Definition 4.** The *cluster membership matrix* for a data set $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$ and a given clustering $\mathcal{C}_1, \dots, \mathcal{C}_k$ is the matrix:

$$\boldsymbol{U} = \begin{bmatrix} u_{11} & \dots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{n1} & \dots & u_{nk} \end{bmatrix}$$

where $u_{ij} = \mathcal{U}_j(\boldsymbol{x}_i)$.

## 6.1. Cluster prototypes



(a) *Point* (b) *Line (Hyperplane)* (c) *Spiral*

Figure 4: *Examples of cluster prototypes*

In addition to the cluster itself, many clustering algorithms rely on some additional object which describes or summarizes the cluster in some way. These objects are commonly referred to as *cluster prototypes* or *cluster representatives*, as they provide some insight into characteristic properties of the cluster that they represent. Figure 4 provides some examples of different cluster prototypes, in the case that they are geometrical objects lying in the same space as the input observations. An interesting note about the cluster prototypes in Figures 4a and 4c is that the points contained in the cluster are identical in the two cases. However, the cluster prototypes are very different, which helps prove the fact that the choice of cluster prototype is not at all obvious in many cases.

## 6.2. Clustering based on distance to prototype

The main motivation behind introducing the cluster prototypes is that they provide a convenient formulation of a general clustering paradigm: Informally stated, an observation should be assigned to the cluster whose representative is the closest to said observation. The key component of this statement is a notion of distance, proximity, or dissimilarity, which is a crucial part of many clustering algorithms.

**Definition 5.** Suppose that the dataset is embedded in some space $X$. Then a *distance function* on $X$ is a function $d : X \times X \to \mathbb{R}$ satisfying, for each $\boldsymbol{x}, \boldsymbol{y} \in X$:

(i) $d(\boldsymbol{x}, \boldsymbol{y}) = d(\boldsymbol{y}, \boldsymbol{x})$

(ii) $d(\boldsymbol{x}, \boldsymbol{y}) \geq 0$.

A distance function is said to be a *metric* on $X$ if it also satisfies

(iii) $d(\boldsymbol{x}, \boldsymbol{y}) = 0$ if and only if $\boldsymbol{x} = \boldsymbol{y}$

(iv) $d(\boldsymbol{x}, \boldsymbol{z}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z})$ for all $\boldsymbol{z} \in X$.

If the cluster prototype is a subset of $X$ and not a single point, we define the distance between a point, $\boldsymbol{x}$ and the prototype, $\Theta$ as:

$$d(\boldsymbol{x}, \Theta) = \min \left\{ d(\boldsymbol{x}, \boldsymbol{\theta}) : \boldsymbol{\theta} \in \Theta \right\}.$$

Choosing a good distance measure is very important for algorithm performance. Although this choice can be very domain-dependent, a few go-to distance functions are provided below.

### Vector space distance functions

Traditionally, it is assumed that the data lies in $\mathbb{R}^m$, where $m$ is the number of attributes contained in each observation. In this case, the number of distance functions in the literature is massive, and naturally, some are applied more often than others. Arguably, the most commonly used distance function is the well-known Euclidean distance,

$$d_{\mathrm{E}}(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}$$

where $\boldsymbol{x} = [x_1, \ldots, x_m]$ and $\boldsymbol{y} = [y_1, \ldots, y_m]$. There is also the squared Euclidean norm:

$$d_{\mathrm{E}}^2(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{m} (x_i - y_i)^2.$$

The former is a specific instance of the $L^p$ norm, defined as

$$d_p(\boldsymbol{x}, \boldsymbol{y}) = \left( \sum_{i=1}^{m} (x_i - y_i)^p \right)^{1/p}, \quad p \geq 1.$$

In the event that the data is not normalized, it can be beneficial to account for the different attributes having different numerical ranges. This is done in the *Mahalanobis distance* which is

$$d_{\mathrm{M}}(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{y})}$$

where $\boldsymbol{\Sigma}$ is the (estimated) covariance matrix of the data generating distribution. The Mahalanobis distance effectively measures distance in units of standard deviation, eliminating the problem of different numerical ranges.

In some cases, it may be useful to have a distance function which is bounded, such as the *cosine distance*

$$d_{\text{cos}}(\boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{\boldsymbol{x}^T \boldsymbol{y}}{||\boldsymbol{x}||\,||\boldsymbol{y}||}$$

where $||\cdot||$ denotes the Euclidean norm. An important property of the cosine distance is that it only depends on the angle between the vectors, meaning that any two parallel vectors will have cosine distance 0.

### Time series distance functions

If the data generating process is sequential in nature, it might be wise to take this into account when selecting the distance function. In Dynamic Time Warping (DTW) [40], the two given sequences are aligned using varying shifts along the time axis, such that the Euclidean distance between the aligned sequences is minimized. This minimum distance is referred to as the DTW distance between the two sequences.

Complexity Invariant Distance (CID) [41] is another time series distance function, in which the Euclidean distance is corrected by a correction factor, which depends on the complexities of the sequences. The key motivation behind the CID is that the distance between time series of different complexities should be large. Suppose we have two (multidimensional) time series, $\boldsymbol{x}_t$ and $\boldsymbol{y}_t$, $t = 1, \ldots, T$. The CID between them is given by:

$$d_{\text{CID}}(\boldsymbol{x}, \boldsymbol{y}) = CF(\boldsymbol{x}, \boldsymbol{y}) \cdot \sqrt{\sum_{t=1}^{T} ||\boldsymbol{x}_t - \boldsymbol{y}_t||^2}$$

where $||\cdot||^2$ denotes the squared euclidean norm, and

$$CF(\boldsymbol{x}, \boldsymbol{y}) = \frac{\max\{CE(\boldsymbol{x}), CE(\boldsymbol{y})\}}{\min\{CE(\boldsymbol{x}), CE(\boldsymbol{y})\}}$$

is the correction factor, and

$$CE(\boldsymbol{x}) = \sqrt{\sum_{t=1}^{T-1} ||\boldsymbol{x}_t - \boldsymbol{x}_{t+1}||^2}$$

is the complexity estimate.

### Image distance functions

Suppose now that our data set consists of $c$-channel $N \times M$ images, where a specific image $\boldsymbol{A}$ has components $\boldsymbol{A} = [A_{ijl}]$, with $i = 1, \ldots, N,\ j = 1, \ldots, M,\ l = 1, \ldots, c$. The Normalized Hausdorff Metric between two images, $\boldsymbol{A}$ and $\boldsymbol{B}$ can then be formulated as

$$d_{\mathrm{H}}(\boldsymbol{A}, \boldsymbol{B}) = \max_{i,j} \left\{ d(\mathcal{A}_{ij}, \mathcal{B}), d(\mathcal{A}, \mathcal{B}_{ij}) \right\}.$$

Here $\mathcal{A}$ and $\mathcal{B}$ are the sets of $(c+2)$-dimensional vectors obtained from $\boldsymbol{A}$ and $\boldsymbol{B}$. That is:

$$\mathcal{A} = \{(i, j, A_{ij1}, \ldots, A_{ijc}) : i = 1, \ldots, N,\ j = 1, \ldots M\}$$
$$\mathcal{B} = \{(i, j, B_{ij1}, \ldots, B_{ijc}) : i = 1, \ldots, N,\ j = 1, \ldots M\}$$

and $\mathcal{A}_{ij}$ and $\mathcal{B}_{ij}$ are the elements of $\mathcal{A}$ and $\mathcal{B}$ corresponding to the pixel located at $(i, j)$. $d$ is some *vectorial* distance function on $\mathbb{R}^{c+2}$, where the distance between a vector and a set of vectors is computed as in Definition 5. Although the formulation is somewhat cryptic, the key intuition behind the Normalized Hausdorff metric is that it measures the maximal inter-pixel distance between two images [42].

In some image processing tasks, the images compared by the system can be warped relative to each other by some affine transformation. If we let the set of warps we wish to consider be denoted $\mathcal{T}$, we can define a modified distance function which accounts for this [43]:

$$d_{\mathcal{T}}(\boldsymbol{A}, \boldsymbol{B}) = || \arg\min_{\boldsymbol{T} \in \mathcal{T}} d(\boldsymbol{A}, \boldsymbol{TB}) ||_F^2$$

where $d$ is some image-distance function, $\boldsymbol{TB}$ is the transformation $\boldsymbol{T}$ applied to the image $\boldsymbol{B}$, and $||\cdot||_F^2$ denotes the squared Frobenius norm of the transformation (sum of squared elements). This distance function effectively searches through the set of allowable transformation, and finds the transformation $\boldsymbol{T}$ such that $\boldsymbol{A}$ is most similar to $\boldsymbol{TB}$. The norm of this transformation is then taken to be the distance between the two images.

## 6.3. Cluster shapes

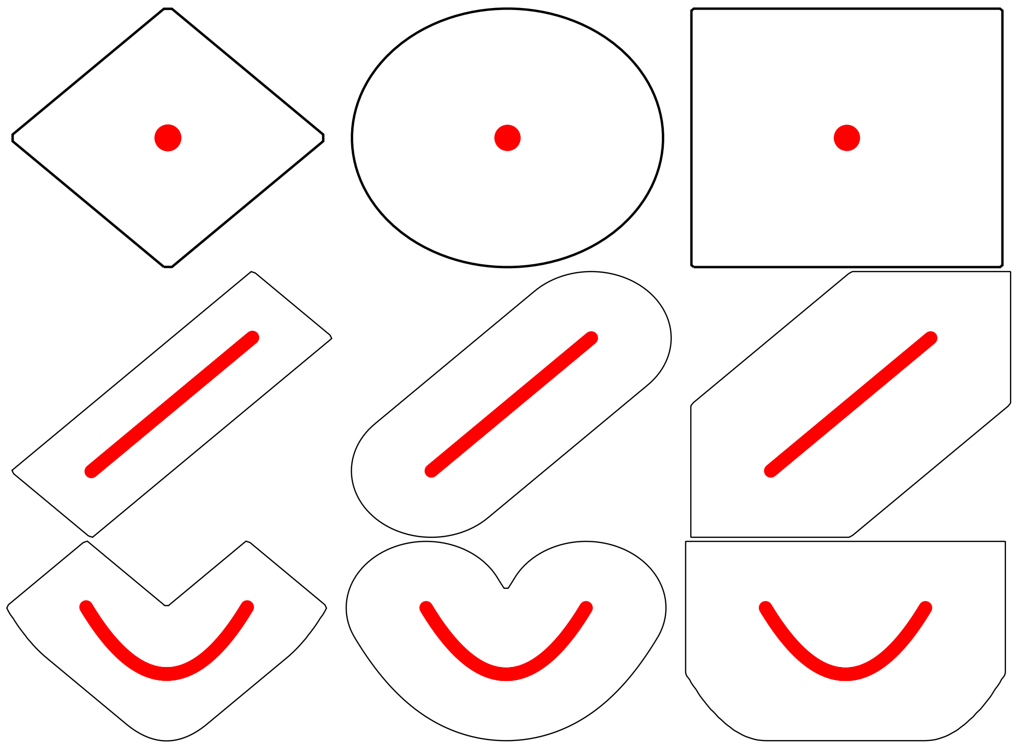


Figure 5: *Cluster prototypes (red) and a level curve (black), indicating the points that are equidistant from the cluster prototype. All distance functions are $L^p$ norms with $p = 1$ (left) $p = 2$ (middle), and $p \to \infty$ (right). The level curve indicates the most compact cluster shape in each of the cases.*

The choice of distance function together with the choice of cluster prototype determines a key aspect of the clustering algorithm, namely the geometrical shape of the “ideal” cluster. In this case, “ideal” refers to the shape which makes the cluster most compact, in the sense that the volume contained by the cluster-shape is maximized for a fixed surface area. This shape is determined by looking at the level surfaces (curves) of the distance function. For a constant $c > 0$, the level surface is the set $\{\boldsymbol{x} : d(\boldsymbol{x}, \Theta) = c\}$, where $d$ is the distance function, and $\Theta$ is the cluster prototype. Some examples of level curves and cluster prototypes are shown in Figure 5.

## 6.4. Data transformation

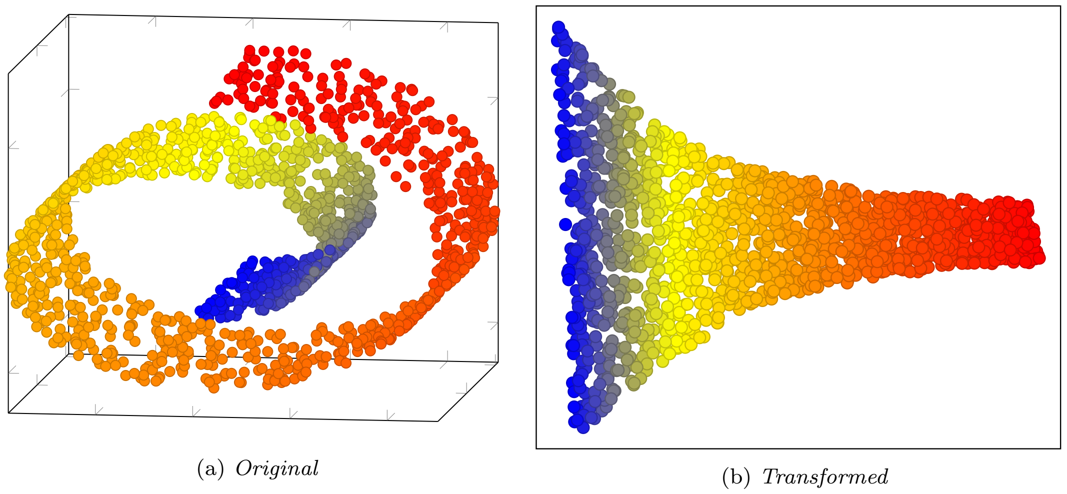

(a) *Original* (b) *Transformed*

Figure 6: *The Swiss-roll dataset "unwrapped" using Locally Linear Embedding [44].*

One important notion to consider before delving into the realm of clustering algorithms, is the one concerning data transformation and dimensionality reduction – and more generally – data preprocessing. As it turns out, real world data is not always as nice to work with, and do therefore often require some processing before being sent to the chosen clustering algorithm. This is especially important when it comes to high dimensional data, such as digital images, or streams from a large collection of industrial sensors. Large dimensionality often leads to high computational complexity, which in turn causes longer training and inference times. Moreover, one might encounter a particular set of mathematical challenges as well. These are often collectively referred to as the *Curse of Dimensionality* [45]. Lastly, higher dimensionality can lead to more complex geometrical structure, potentially making the specification of a "good" distance function difficult.

These issues have led to the development of many techniques for *feature learning* over the years. The goal of these is essentially to learn compressed vectorial representations for the input data. A comprehensive review of these methods is beyond the scope of this thesis. However, notable contributions include Principal Component Analysis (PCA) [46], $t$-distributed Stochastic Neighborhood Embedding ($t$-SNE) [47], Multidimensional Scaling (MDS) [48], Laplacian Eigenmaps [49], Locally Linear Embedding (LLE) [44], and Uniform Manifold Approximation and Projection (UMAP) [50]. The last three of these methods are motivated by the concept of *manifold unwrapping*, where one attempts to "unwrap" a manifold embedded in a high dimensional space, and then embed the unwrapped manifold in a lower dimensional space. The classical Swiss-roll example is shown in Figure 6, where the initially curved data manifold has been "flattened" using LLE.

Methods like these have been extensively used by machine learning practitioners to map the raw data to vector spaces where the clustering algorithms can work more comfortably. However, as we will see later, a new trend within the field is to combine the data transformation step with the clustering algorithm to create an end-to-end pipeline for joint feature learning and clustering.

# 7. Cluster evaluation

Finding a good clustering of a dataset is the essence of cluster analysis. However, how do we determine whether a clustering is good or bad? This question has led to the development of mathematical quantities referred to as *Cluster Validity Indices* (CVIs), whose objective is to say something about the quality of a given clustering, with respect to some predetermined conditions or assumptions. CVIs are usually classified into two categories [1, 51]:

- *Internal CVIs*, which measure the quality of a clustering without any external information. The majority of internal CVIs are based on notions of between-cluster separability and within-cluster compactness, and do therefore require the quantization of these ideas.
- *External CVIs*, which measure the quality of a clustering based on some fixed external "solution". The most common case is when the clusters produced by the algorithm should reflect some categorization which is known beforehand. One can then compare the cluster memberships with the ground truth labels of the categorization to determine the quality of the clustering.

In both these categories, the corresponding CVIs are test statistics entering in statistical hypothesis tests. The exact formulation of the tests, and their corresponding null hypothesis are different for the different CVIs, but they are all based on the idea that *"[...] $H_0$ should be a statement of randomness concerning the structure of $X$"* [1], where $X$ denotes the input space. In other words, there is no apparent cluster structure under the null hypothesis. Due to the often difficult task of sampling under the null hypothesis, it has become more common to report the value of the test statistic itself, rather than the outcome of the test. Although this removes the need to sample under the null hypothesis, it requires the evaluator to have a more thorough understanding of the test statistic.

## 7.1. Internal CVIs

The literature on clustering evaluation contains a large amount of suggested internal CVIs, and comprehensive reviews can be found in e.g. [52] or [53]. The following internal CVIs are some of the most frequently used indices, and have been shown to perform well in several applications [52, 53].

- **Calinski-Harabasz (CH)** [54]. This index is defined as:

$$CH = \frac{n-k}{k-1} \frac{\sum_{i=1}^{k} |\mathcal{C}_i| d_E(\bar{\boldsymbol{x}}, \boldsymbol{m}_i)}{\sum_{i=1}^{k} \sum_{\boldsymbol{x} \in \mathcal{C}_i} d_E(\boldsymbol{x}, \boldsymbol{m}_i)}$$

  where $\bar{\boldsymbol{x}} = \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{x}_j$ is the overall mean of the dataset, $\boldsymbol{m}_i = \frac{1}{|\mathcal{C}_i|} \sum_{\boldsymbol{x} \in \mathcal{C}_i} \boldsymbol{x}$ is the mean of the $i$-th cluster, and $d_E(\cdot, \cdot)$ denotes the Euclidean distance function. The Calinski-Harabasz index is a classical take on the separability-over-compactness criterion as the numerator looks at the average distance between cluster means and the global mean, while the denominator considers the distance between cluster means and the observations assigned to the respective clusters. This means that a large value of $CH$ corresponds to a "good" clustering.

- **Davies-Bouldin (DB)** [55]. The DB index is defined as:

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{j=1,\ldots,k,\ j \neq i} \frac{S(\mathcal{C}_i) + S(\mathcal{C}_j)}{d_E(\boldsymbol{m}_i, \boldsymbol{m}_j)}$$

  where $S(\mathcal{C}_i) = \frac{1}{|\mathcal{C}_i|} \sum_{x \in \mathcal{C}_i} d_E(\boldsymbol{x}, \boldsymbol{m}_i)$ is the summed within-cluster distance for cluster $i$, and $\boldsymbol{m}_i$ and $d_E(\cdot, \cdot)$ are defined as above. The interpretation of this index is somewhat more involved than than the aforementioned CH index. However, it can be interpreted as the average maximum "closeness" between clusters. The fraction in the expression is the ratio between the average sum of squares in the respective clusters, and the distance between their respective means. Thus, if the numerator is large, while the denominator is small, the ratio will be large, and the clusters will appear close to each other. This means that a small value of $DB$ corresponds to a "good" clustering.

It should be noted that in their original formulations, both of these indices use the Euclidean distance function as a measure of dissimilarity. In theory, one could use any distance function, as long as it allows for the computation of both point-to-point distances, as well as point-to-cluster distances. However, as is also the case with several clustering algorithms, the choice of distance function is a critical one, and can severely degrade the performance of the index if not made appropriately.

## 7.2. External CVIs

As the name implies, the task of external CVIs is to measure the quality of the clustering with respect to some predetermined ground truth. In the following descriptions, let

$$y_i = j : u_{ij} \geq u_{il}, l = 1, \dots, k$$

be the index of the predicted cluster for observation $i$, and let $r_i$ be the corresponding ground truth label for observation $i$. Then we have the following external CVIs:

- **Unsupervised Clustering Accuracy (ACC)**. This is perhaps the most frequently used external CVI in the recent years, and is defined as

  $$ACC = \max_{m \in \mathcal{M}} \frac{1}{n} \sum_{i=1}^{n} \delta(m(y_i) - r_i)$$

  where $\delta(\cdot)$ denotes the Kronecker delta function, and $\mathcal{M}$ denotes the set of all possible bijective mappings from $\{1, \dots, k\}$ to itself. The unsupervised clustering accuracy is essentially the best possible accuracy when one attempts to assign each of the clusters to different categories.

- **Normalized Mutual Information (NMI)**. This external CVI is rooted in information theory, and is defined as

  $$NMI = \frac{I(\boldsymbol{y}, \boldsymbol{r})}{\frac{1}{2}(H(\boldsymbol{y}) + H(\boldsymbol{r}))}$$

  where we let $\boldsymbol{y} = [y_1, \dots, y_n]^T$ and $\boldsymbol{r} = [r_1, \dots, r_n]^T$ to unburden the notation. $I(\cdot, \cdot)$ denotes the mutual information:

  $$I(\boldsymbol{y}, \boldsymbol{r}) = \sum_{i=1}^{k} \sum_{j=1}^{k} P_{ij}^{yr} \ln \frac{P_{ij}^{yr}}{P_i^y P_j^r}$$

  and $H(\cdot)$ denotes the entropy:

  $$H(\boldsymbol{y}) = -\sum_{i=1}^{k} P_i^y \ln P_i^y.$$

  The $P$'s denote relative frequencies of cluster indices occurring in the respective vectors, and are defined as

  $$P_i^y = \frac{1}{n} \sum_{l=1}^{n} \delta(y_l - i), \quad P_{ij}^{yr} = \frac{1}{n} \sum_{l=1}^{n} \delta(y_l - i)\delta(r_l - j).$$

  The NMI measures the mutual dependence between the cluster assignments and the ground truth labels. A high NMI implies that the ground truth labels are – to a large degree – explained by the cluster memberships. This indicates a good clustering.

## 7.3. Choosing the correct CVI

As it turns out, both external and internal CVIs have their own pitfalls that should be avoided. With internal CVIs, one has to chose a distance function – or at the very least – some function capable of correctly capturing dissimilarities in the input space. On the other hand, when using external CVIs, one has to be certain that the specific categorization is indeed the "best" possible clustering. In the latter case, the existence of another clustering that appears more natural to the algorithm, would completely invalidate the evaluation procedure.

These drawbacks are thoroughly discussed by von Luxburg et al. [23], and lead to the conclusion that the evaluation of a given clustering has to be done with respect to the problem at hand. This conclusion also emphasizes the importance of qualitative analysis when evaluating the performance of a clustering algorithm. For instance, qualitative analysis might reveal that the clustering reflects some other categorization than the one used with external CVIs – or that the clustering has indeed learned to identify similar objects, even though this was not reflected by the distance function used in internal CVIs. We will see examples of the necessity of these considerations in Part V, which describes the experiments and their respective outcomes.

# 8. Clustering algorithms

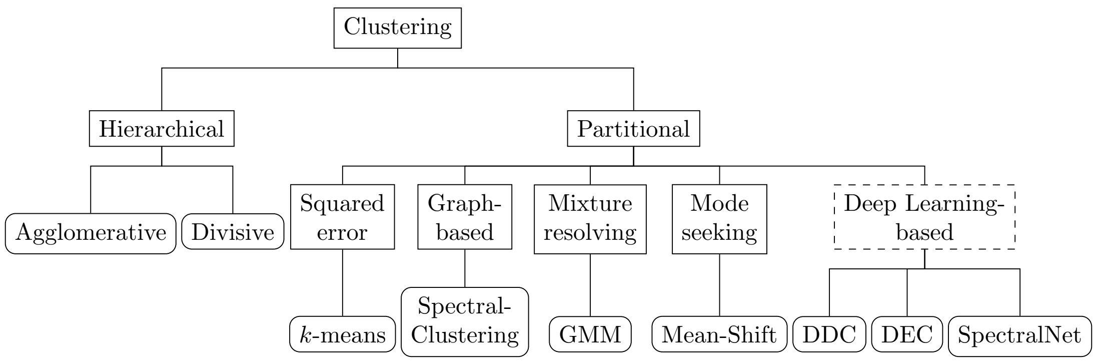


Figure 7: *A collection of clustering paradigms, and associated algorithms. Based on [9], but augmented to include the more recent deep learning-based clustering algorithms.*

Since the concept of clustering was first introduced, a large number of approaches and algorithms have surfaced over the years. Figure 7 outlines some of the main paradigms under which most of the commonly used clustering techniques are developed. According to the two top branches of Figure 7, a clustering algorithm can be either hierarchical or partitional. What is characteristic for hierarchical algorithms, is that they produce a hierarchy of clusters, where at the one end, all observations are assigned to separate clusters, and at the other end, all observations are assigned to the same cluster. Partitional algorithms on the other hand, produces a single clustering by partitioning the input space into a fixed number of partitions. The observations are then assigned to clusters based on which partition they lie within. A direct consequence of many approaches to partitional clustering is that the number of clusters has to be known beforehand. Clearly, this is unproblematic if one seeks the partitioning between a known set of classes present in the dataset. However, this might be problematic for more exploratory applications. A simple way to get around this problem has been to run the algorithm several times using a different number of clusters each time, and then use a CVI to chose the number of clusters. Although most of the algorithms we will cover in this paper fall within the partitional category, a couple of approaches to hierarchical clustering will be included as well, where the number of clusters can be determined based on intrinsic quantities computed during the clustering process.

## 8.1. Hierarchical algorithms

As can be seen in Figure 7, there are two main approaches to hierarchical clustering, namely agglomerative and divisive. In agglomerative hierarchical clustering, the algorithm is initialized with all observations placed in separate clusters. Pairs of clusters are then iteratively merged, according to some criterion, until all observations are assigned to the same cluster. Divisive hierarchical clustering works in the opposite direction, meaning that the algorithm is initialized with all observations belonging to the same cluster. Then, at each iteration, a cluster is split according to that same criterion. The divisive algorithm terminates when all observations lie in separate clusters.

## 8.2. Single-link, average-link and complete-link

The criterion which governs the merging or splitting of clusters is the most crucial part of the hierarchical clustering algorithms. Suppose that at the current iteration we have the clusters $\mathcal{C}_1, \dots, \mathcal{C}_k$. For agglomerative clustering, the criterion is as follows:

Merge clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ if:

$$d(\mathcal{C}_i, \mathcal{C}_j) = \min_{\substack{a,b=1,\dots,k \\ a \neq b}} d(\mathcal{C}_a, \mathcal{C}_b). \tag{1}$$

I.e, merge the two clusters which are the closest. For the divisive approach, we split cluster $\mathcal{C}_i$ into clusters $\mathcal{C}_a$ and $\mathcal{C}_b$ if:

$$i = \arg\max_{i=1,\dots,k} \left\{ \max_{\substack{\mathcal{C}_c \cap \mathcal{C}_d = \emptyset \\ \mathcal{C}_c \cup \mathcal{C}_d = \mathcal{C}_i}} d(\mathcal{C}_c, \mathcal{C}_d) \right\} \tag{2}$$

and

$$d(\mathcal{C}_a, \mathcal{C}_b) = \max_{\substack{\mathcal{C}_c \cap \mathcal{C}_d = \emptyset \\ \mathcal{C}_c \cup \mathcal{C}_d = \mathcal{C}_i}} d(\mathcal{C}_c, \mathcal{C}_d). \tag{3}$$

Here, we search over all allowable sub-clusters of each cluster, and find the two sub-clusters which are the furthest apart from each other. These are then taken as separate clusters during the next iteration.

The function $d$ is in this case a distance function measuring distance between *clusters*. Depending on how $d$ is computed, the *linkage* of the algorithm is said to be either single-link, average-link, or complete-link. For two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, the different linkages are defined as follows:

- *Single-link*: The distance between the two closest points from the different clusters:
$$d_{\text{sl}}(\mathcal{C}_i, \mathcal{C}_j) = \min_{\boldsymbol{x} \in \mathcal{C}_i, \boldsymbol{y} \in \mathcal{C}_j} d_X(\boldsymbol{x}, \boldsymbol{y}).$$
- *Average-link*: The average distance between all points from different clusters:
$$d_{\text{al}}(\mathcal{C}_i, \mathcal{C}_j) = \frac{1}{|\mathcal{C}_i| \cdot |\mathcal{C}_j|} \sum_{\boldsymbol{x} \in \mathcal{C}_i, \boldsymbol{y} \in \mathcal{C}_j} d_X(\boldsymbol{x}, \boldsymbol{y}).$$
- *Complete-link*: The distance between the two most distant points from the different clusters:
$$d_{\text{cl}}(\mathcal{C}_i, \mathcal{C}_j) = \max_{\boldsymbol{x} \in \mathcal{C}_i, \boldsymbol{y} \in \mathcal{C}_j} d_X(\boldsymbol{x}, \boldsymbol{y}).$$

$d_X$ is a distance function on the input space $X$, following Definition 5.

The specification of the merging/splitting criterion allows us to formulate the two hierarchical clustering algorithms, which can be found in Algorithms 1 and 2.

```
input  : Raw dataset $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ or distance-matrix
         $\boldsymbol{D} = [d_{ij}],\ d_{ij} = d_X(\boldsymbol{x}_i, \boldsymbol{x}_j)$
output: Cluster hierarchy $\{\{\mathcal{C}_1^1, \ldots, \mathcal{C}_n^1\}, \{\mathcal{C}_1^2, \ldots, \mathcal{C}_{n-1}^2\}, \ldots, \{\mathcal{C}_1^n\}\}$
Initialize $\mathcal{C}_1^1, \ldots, \mathcal{C}_n^1 = \{\boldsymbol{x}_1\}, \ldots, \{\boldsymbol{x}_n\}$
for $t \leftarrow 1$ to $n$ do
    Find clusters $\mathcal{C}_i^t$ and $\mathcal{C}_j^t$ satisfying Eq. (1).
    Form the next hierarchy-level by merging $\mathcal{C}_i^t$ and $\mathcal{C}_j^t$, and leaving the
      other clusters as-is.
end
```
**Algorithm 1:** Agglomerative hierarchical clustering.

```
input  : Raw dataset $\mathcal{X} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$ or distance-matrix
         $\boldsymbol{D} = [d_{ij}],\ d_{ij} = d_X(\boldsymbol{x}_i, \boldsymbol{x}_j)$
output: Cluster hierarchy $\{\{\mathcal{C}_1^1\}, \{\mathcal{C}_1^2, \mathcal{C}_2^2\}, \dots, \{\mathcal{C}_1^n, \dots \mathcal{C}_n^n\}\}$
Initialize $\mathcal{C}_1^1 = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$
for $t \leftarrow 1$ to $n$ do
    Find clusters $\mathcal{C}_i^t = \mathcal{C}_a^{t+1} \cup \mathcal{C}_b^{t+1}$ satisfying Eq. (2) and Eq. (3).
    Form the next hierarchy-level by splitting $\mathcal{C}_i^t$ into $\mathcal{C}_a^{t+1}$ and $\mathcal{C}_b^{t+1}$, while
     leaving the other clusters as-is.
end
```

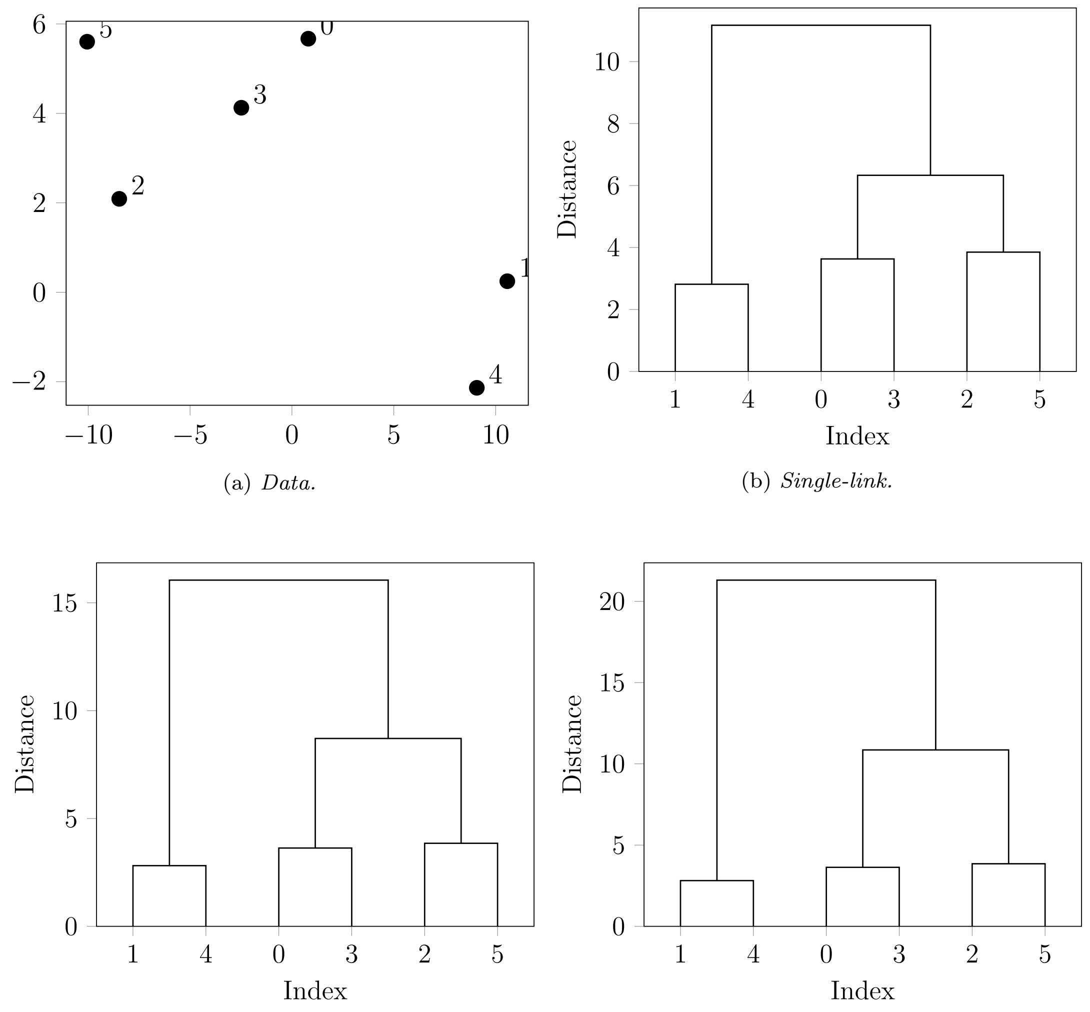
**Algorithm 2:** Divisive hierarchical clustering.

## 8.3. The dendrogram

The dendrogram is a great tool for visualizing the cluster hierarchy produced by a hierarchical clustering algorithm. For simplicity, we will only consider the agglomerative case[1]. Example dendrograms for the three different linkage types are shown in Figure 8. When looking at a dendrogram, there are two main elements to consider: (i) The actual cluster assignments produced, and (ii) at each step, what was the distance between the merged/divided cluster? The latter consideration brings with it the important notion of *cluster lifetime*, which in turn can help determine the level of the hierarchy that should be taken as the final clustering of the dataset. The lifetime of a cluster is defined as $|d_{\text{created}} - d_{\text{absorbed}}|$, where $d_{\text{created}}$ is the distance at which two clusters were merged to create the current cluster. $d_{\text{absorbed}}$ is the distance at which the current cluster was absorbed into a larger cluster [1]. Note that both of these quantities can be extracted directly from the dendrogram.

The idea behind a cluster's lifetime is that it says something about both how compact the cluster is, and how isolated it is from other clusters. A compact cluster will be merged at a low distance, whereas an isolated cluster will be absorbed at a large distance. Hence, the difference between the distances will be large, producing a large cluster lifetime. Similar approaches to determining the optimal level in the hierarchy have been proposed by [1, 52, 56].

[1]The divisive case is analogous, as the dendrograms can be read top-down instead of bottom-up.



(a) *Data.* (b) *Single-link.* (c) *Average-link.* (d) *Complete-link.*

Figure 8: *Dendrograms produced when performing hierarchical clustering on a small data set. Although the link-types produce the same clusters, the distances at which the clusters are merged, are different.*

## 8.4. Partitional algorithms

### 8.4.1. $k$-means

Perhaps the most frequently used clustering algorithm to date is the $k$-means algorithm [57]. The algorithm arises naturally from the general prototype-based method outlined earlier, by choosing the squared Euclidean distance measure, along with point cluster prototypes. Suppose we are interested in partitioning the data set into $k$ disjoint clusters $\mathcal{C}_1, \ldots, \mathcal{C}_k$, associated with point-prototypes $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_k$. This can be formulated as a loss function minimization problem, with loss function

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j=1}^{k} u_{ij} ||\boldsymbol{x}_i - \boldsymbol{\theta}_j||^2 \tag{4}$$

where $u_{ij} = \mathcal{U}_j(\boldsymbol{x}_i)$ is the $j$-th cluster membership function evaluated at the $i$-th observation. In $k$-means, the clusterings are assumed to be hard, thus causing the $u_{ij}$ to be non-differentiable. We will therefore settle for the two-stage greedy optimization algorithm:

1. Treat the cluster prototypes as fixed, and recompute the cluster membership functions such that Eq. (4) is minimized.
2. Treat the cluster membership functions as fixed, and recompute the cluster prototypes such that Eq. (4) is minimized.

In stage 1, we can see that Eq. (4) is minimized if a point is assigned to the *closest* cluster prototype. That is:

$$u_{ij} = \begin{cases} 1, & ||\boldsymbol{x}_i - \boldsymbol{\theta}_j||^2 = \min\limits_{l=1,\ldots,k} ||\boldsymbol{x}_i - \boldsymbol{\theta}_l||^2 \\ 0, & \text{otherwise} \end{cases}. \tag{5}$$

In stage 2 we can minimize $\mathcal{L}$ by ordinary gradient-based minimization. For the $j$-th prototype, we get

$$\nabla_{\boldsymbol{\theta}_j} \mathcal{L} = -2 \sum_{i=1}^{n} u_{ij} (\boldsymbol{x}_i - \boldsymbol{\theta}_j)$$

equating the gradient to zero and solving for $\theta_j$ gives

$$\boldsymbol{\theta}_j^{\text{new}} = \frac{\sum_{i=1}^{n} u_{ij} \boldsymbol{x}_i}{\sum_{i=1}^{n} u_{ij}} = \frac{1}{|\mathcal{C}_j|} \sum_{\boldsymbol{x} \in \mathcal{C}_j} \boldsymbol{x} \tag{6}$$

which is the mean of cluster $\mathcal{C}_j$, hence the name *$k$-means*. The optimization process is summarized in Algorithm 3.

#### 8.4.1.1. Convergence

It can be shown that the optimization procedure outlined above converges to a local minima of the loss function in a finite number of steps [58]. The proof is informally summarized below. First, make the following observations:

1. Due to the hard nature of the algorithm, there is finite, but possibly large, number of different clusterings.
2. At each iteration, the loss function $\mathcal{L}$ decreases or stays constant.

```
input  : Raw dataset $\mathcal{X} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$, number of clusters, $k$.
output: Cluster membership matrix $\boldsymbol{U} = [u_{ij}]$.
Initialize $\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_k$ randomly.
while not converged do
    Recompute cluster assignment matrix $\boldsymbol{U} = [u_{ij}]$:
        $u_{ij} = \begin{cases} 1, & ||\boldsymbol{x}_i - \boldsymbol{\theta}_j||^2 = \min_{l=1,\dots,k} ||\boldsymbol{x}_i - \boldsymbol{\theta}_l||^2 \\ 0, & \text{otherwise} \end{cases}.$
    Recompute cluster prototypes $\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_k$:
        $\boldsymbol{\theta}_j = \frac{1}{|\mathcal{C}_j|} \sum_{\boldsymbol{x} \in \mathcal{C}_j} \boldsymbol{x}, \ j = 1, \dots, k.$
end
```

**Algorithm 3:** $k$-means.

3. If the clustering algorithm produces the same two clusterings at two consecutive iterations, it has converged.

From observation 1, it is evident that, at some iteration, the algorithm will revisit a previous solution (clustering). Thus, the algorithm has entered a cycle. If the cycle had length greater than 1, by observation 2, this would imply that the loss function would take on a value at the current iteration, which is lower than the value of the loss function which was previously visited. Hence, the cycle has length 1, and the algorithm has converged by observation 3.

It is very important to stress that this proof only guarantees convergence to a *local* minimum of the loss function. There might therefore exist better clusterings which are never visited by the algorithm. Due to this fact, it is common to run the $k$-means algorithm from several different initializations, and then choose the clustering which resulted in the lowest value of the loss function.

#### 8.4.1.2. Initialization

In some cases, the random initialization might not be sufficient for producing good initial prototypes. To alleviate this, many different initialization schemes have been proposed to help speed up the convergence of the algorithm, and improve the quality of the final clustering. One option is to choose $k$ random observations from the dataset, and use these as the initial prototypes [1]. This ensures that the prototypes are initialized close to the data, regardless of the location of the data distribution. A more robust method is the $k$-means++ initialization technique [59], in which prototypes are iteratively chosen such that the probability of them being far away from each other, is large. This helps scatter the initial prototypes such that most of the populated input space is explored during the optimization procedure.

### 8.4.2. Spectral clustering

In contrast to the aforementioned methods, spectral clustering does not make an explicit vector-space assumption. Rather, the data is represented as a weighted undirected graph $G = (V, E, W)$, where each vertex represents an observation, $V = \mathcal{X} = \{\boldsymbol{x}_1, \ldots \boldsymbol{x}_n\}$. $W = [w_{ij}]$ is a symmetric and nonnegative affinity-matrix, which contains the weights of each edge, describing how strongly two vertices (observations) are connected. A large $w_{ij}$ means that observations $i$ and $j$ are strongly connected, or *similar* in some sense. In this case, the clustering problem can therefore be seen as cutting the graph to produce $k$ disconnected subgraphs. This train of thought is what initially led to the minimization problem [60]:

$$\min_{\mathcal{C}_1,\ldots,\mathcal{C}_k} \sum_{j=1}^{k} \text{cut}(\mathcal{C}_j, \mathcal{C}_j^c), \tag{7}$$

where $\mathcal{C}_j^c$ denotes the complement-set of $\mathcal{C}_j$, and

$$\text{cut}(\mathcal{C}_j, \mathcal{C}_j^c) = \sum_{\boldsymbol{x}_i \in \mathcal{C}_j} \sum_{\boldsymbol{x}_l \in \mathcal{C}_j^c} w_{il}. \tag{8}$$

Hence, the minimization problem of Eq. (7) can be seen as minimizing the total weight of the cut edges, effectively finding the partitioning such that the summed similarities of points assigned to different clusters, is minimized. However, the solution to the minimization problem in Eq. (7) tends to favor very small and isolated clusters, meaning that the resulting clusters might be very imbalanced with respect to the number of observations assigned to them. This led to the formulation of the *normalized cut* [61]:

$$\text{ncut}(\mathcal{C}_j, \mathcal{C}_j^c) = \frac{\text{cut}(\mathcal{C}_j, \mathcal{C}_j^c)}{\text{vol}(\mathcal{C}_j)}$$

and the minimization problem

$$\min_{\mathcal{C}_1,\ldots,\mathcal{C}_k} \sum_{j=1}^{k} \text{ncut}(\mathcal{C}_j, \mathcal{C}_j^c). \tag{9}$$

The normalization term $\text{vol}(\mathcal{C}_j)$ is defined as

$$\text{vol}(\mathcal{C}_j) = \sum_{\boldsymbol{x}_i \in \mathcal{C}_j} d_{ii}$$

where $\boldsymbol{D} = [d_{ii}]$ is a diagonal "degree" matrix, with elements $d_{ii} = \sum_{l=1}^{n} w_{il}$. The degree $d_{ii}$ of an observation $\boldsymbol{x}_i$ tells us "how connected" $\boldsymbol{x}_i$ is to the other observations. The volume of a cluster is the sum of the degrees of its elements, which means that a cluster containing few elements weakly connected to all other observations, will have a small volume. This is precisely how the normalization of the cut works in Eq. (9). If we define the matrix $\boldsymbol{Y} = [y_{ij}]$, where

$$y_{ij} = \frac{u_{ij}}{\sqrt{\mathrm{vol}(\mathcal{C}_j)}} = \begin{cases} \frac{1}{\sqrt{\mathrm{vol}(\mathcal{C}_j)}}, & \boldsymbol{x}_i \in \mathcal{C}_j \\ 0, & \text{otherwise} \end{cases}, \tag{10}$$

it can be shown that [62]:

$$\sum_{j=1}^{k} \mathrm{ncut}(\mathcal{C}_i, \mathcal{C}_i^c) \propto \mathrm{tr}(\boldsymbol{Y}^T \boldsymbol{L} \boldsymbol{Y})$$

where $\mathrm{tr}(\cdot)$ denotes the matrix trace, and $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$ is the graph Laplacian. Furthermore, observe that $\boldsymbol{Y}^T \boldsymbol{D} \boldsymbol{Y} = \boldsymbol{I}_k$, which is the $k \times k$ identity matrix.

To allow for more efficient optimization, we will relax the hard memberships from Eq. (10). With this relaxation, we can approximate the minimization problem in Eq. (9) as

$$\arg \min_{\boldsymbol{Y} \in \mathbb{R}^{n \times k}} \mathrm{tr}(\boldsymbol{Y}^T \boldsymbol{L} \boldsymbol{Y}), \quad \text{s.t.} \quad \boldsymbol{Y}^T \boldsymbol{D} \boldsymbol{Y} = \boldsymbol{I}_k. \tag{11}$$

This is the well known trace-minimization problem [2], which is solved by taking the columns of $\boldsymbol{Y}$ to be the $k$ eigenvectors of $\boldsymbol{L}$ corresponding to its $k$ smallest eigenvalues. When the solution, $\boldsymbol{Y}^*$, to the relaxed optimization problem has been found, it still remains to recover the original hard cluster memberships $u_{ij}$. This can be done with e.g. $k$-means Shalev-Shwartz and Ben-David [2], thresholding [1, 61], or spectral rotation [63]. The complete Spectral Clustering algorithm is summarized in Algorithm 4.

```
input  : Affinity-matrix $\boldsymbol{W}$, number of clusters $k$.
output: Cluster membership matrix $\boldsymbol{U} = [u_{ij}]$.

Compute the eigenvectors of $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$ corresponding to the $k$
  smallest eigenvalues. These are the columns of $\boldsymbol{Y}^*$.

Find $\boldsymbol{U}$ by running $k$-means on $\boldsymbol{Y}^*$, or by using some other
  thresholding method.
```

**Algorithm 4:** Spectral clustering.

#### 8.4.2.1. Computing the affinity-matrix

Up until this point, we have assumed that the affinity matrix is known. This is not the case for most applications, which makes the computation of the affinity-matrix a critical step of spectral clustering. Perhaps the most widely used class of affinity matrices are computed using the $\varepsilon$-connected Gaussian affinity function:

$$w_{ij} = \begin{cases} \exp\left(-\frac{d(\boldsymbol{x}_i, \boldsymbol{x}_j)}{2\sigma^2}\right), & d(\boldsymbol{x}_i, \boldsymbol{x}_j) < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

where $\varepsilon$ and $\sigma$ are hyperparameters, and $d$ is some symmetric distance function.

### 8.4.3. Mixture models

The class of mixture models arise naturally when one envisions a probabilistic approach to the clustering problem. Suppose that the clusters $\mathcal{C}_1, \dots \mathcal{C}_k$ are represented by $k$ different probability distributions $p_1, \dots, p_k$, resulting in the marginal data distribution:

$$p(\boldsymbol{x}) = \sum_{j=1}^{k} \pi_j p_j(\boldsymbol{x}; \boldsymbol{\phi}_j)$$

where $\boldsymbol{\phi}_j$ is a vector of parameters, and $\pi_j$ denotes the probability of an observation belonging to $\mathcal{C}_j$, satisfying $\sum_{j=1}^{k} \pi_j = 1$. Let us introduce stochastic versions of the cluster indicators, such that

$$U_{ij} = \begin{cases} 1, & \boldsymbol{X}_i \in \mathcal{C}_j \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

where $\boldsymbol{X}_i$ can be thought of as the stochastic (unobserved) $i$-th element of the dataset. Then we have

$$p(u_{i1}, \dots, u_{ik}) = \prod_{j=1}^{k} \pi_j^{u_{ij}}$$

and

$$p(\boldsymbol{x}_i, u_{i1}, \dots, u_{ik}) = \prod_{j=1}^{k} \pi_j^{u_{ij}} p_j(\boldsymbol{x}_i; \boldsymbol{\phi}_j)^{u_{ij}}.$$

This gives the log-likelihood

$$l(\boldsymbol{\pi}, \boldsymbol{\Phi} \mid \mathcal{X}, \boldsymbol{U}) = \sum_{i=1}^{n} \sum_{j=1}^{k} U_{ij}(\ln \pi_j + \ln p_j(\boldsymbol{x}_i; \boldsymbol{\phi}_j)) \tag{13}$$

where we let $\boldsymbol{\pi} = [\pi_1, \dots, \pi_k]^T$, $\boldsymbol{\Phi} = [\boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_k]^T$ and $\boldsymbol{U} = [U_{ij}]$, to unburden the notation.

Since the $U_{ij}$ in Eq. (13) are unobserved, we cannot maximize the log-likelihood directly. Hence, the mixture resolving is usually done using the Expectation Maximization (EM) algorithm [64], which is an iterative two-step procedure for maximizing the log-likelihood, where at iteration $t$, we do:

1. E-step: Compute the expected log-likelihood $Q$, conditioned on the current parameters $(\boldsymbol{\pi}^{(t)}, \boldsymbol{\Phi}^{(t)})$, and the observed data $\mathcal{X}$:
$$Q(\boldsymbol{\pi}, \boldsymbol{\Phi} \mid \boldsymbol{\pi}^{(t)}, \boldsymbol{\Phi}^{(t)}) = \mathrm{E}(l(\boldsymbol{\pi}, \boldsymbol{\Phi} \mid \mathcal{X}, \boldsymbol{U}) \mid \boldsymbol{\pi}^{(t)}, \boldsymbol{\Phi}^{(t)}, \mathcal{X})$$

2. M-step: Update the parameters by maximizing the expected log-likelihood with respect to the parameters:
$$(\boldsymbol{\pi}^{(t+1)}, \boldsymbol{\Phi}^{(t+1)}) = \arg\max_{(\boldsymbol{\pi}, \boldsymbol{\Phi})} Q(\boldsymbol{\pi}, \boldsymbol{\Phi} \mid \boldsymbol{\pi}^{(t)}, \boldsymbol{\Phi}^{(t)}).$$

With the log-likelihood from Eq. (13), we get

$$Q(\boldsymbol{\pi}, \boldsymbol{\Phi} \mid \boldsymbol{\pi}^{(t)}, \boldsymbol{\Phi}^{(t)}) = \sum_{i=1}^{n} \sum_{j=1}^{k} p_{ij} (\ln \pi_j + \ln p_j(\boldsymbol{x}_i; \boldsymbol{\phi}_j))$$

where

$$p_{ij} = \mathrm{E}(U_{ij} \mid \boldsymbol{\pi}^{(t)}, \boldsymbol{\Phi}^{(t)}, \mathcal{X}) = \frac{\pi_j^{(t)} p_j(\boldsymbol{x}_i; \boldsymbol{\phi}_j^{(t)})}{\sum_{l=1}^{k} \pi_l^{(t)} p_l(\boldsymbol{x}_i; \boldsymbol{\phi}_l^{(t)})}. \tag{14}$$

Maximizing $Q$ with respect to $\pi_j$ gives

$$\pi_j^{(t+1)} = \frac{1}{n} \sum_{i=1}^{n} p_{ij}.$$

whereas the explicit form of $\boldsymbol{\Phi}^{(t+1)}$ depends on the choice of probability distributions $p_1, \ldots, p_k$. When it comes to determining the output of the algorithm, we must remember that in the case of mixture models, the $U_{ij}$ are unobserved stochastic variables, and thus, cannot be ”computed” directly. Hence, we estimate these using their expected value, conditioned on the parameters of the final model, as well as the observed data. This expectation is precisely the $p_{ij}$ defined in Eq. (14).

#### 8.4.3.1. Gaussian mixture models

Up until this point, we have assumed some general form of the probability distributions $p_1, \ldots, p_k$. This is nice for the sake of generality, but when we actually want to compute something, these have to be specified. A common choice is to let each of the distributions be a Gaussian with different means and covariance-matrices:

$$p_j(\boldsymbol{x}; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) = \frac{1}{(2\pi)^{\delta/2} \sqrt{\det \boldsymbol{\Sigma}_j}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_j)\right)$$

where $\delta$ is the dimensionality of the input space. Using these probability density functions, we get an explicit form for the update equations for $\boldsymbol{\phi}_j = (\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$:

$$\begin{aligned}
\boldsymbol{\mu}_j^{(t+1)} &= \frac{\sum_{i=1}^n p_{ij}\boldsymbol{x}_i}{\sum_{i=1}^n p_{ij}} \\
\boldsymbol{\Sigma}_j^{(t+1)} &= \frac{\sum_{i=1}^n p_{ij}(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t+1)})(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t+1)})^T}{\sum_{i=1}^n p_{ij}}
\end{aligned}$$

The complete EM-algorithm for a Gaussian mixture model is presented in Algorithm 5.

```
input  : Raw dataset $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, number of mixture
         components (clusters), $k$.
output: Expected membership matrix $\boldsymbol{P} = [p_{ij}]$.
Initialize $\boldsymbol{\mu}_1^{(1)}, \ldots, \boldsymbol{\mu}_k^{(1)}$, $\boldsymbol{\Sigma}_1^{(1)}, \ldots, \boldsymbol{\Sigma}_k^{(1)}$.
while not converged do
    Compute the $p_{ij}$ using the current parameters:
        $$p_{ij} = \frac{\pi_j^{(t)} p_j(\boldsymbol{x}_i; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}{\sum_{l=1}^k \pi_l^{(t)} p_l(\boldsymbol{x}_i; \boldsymbol{\mu}_l, \boldsymbol{\Sigma}_l)}$$
    Update $\boldsymbol{\pi}$, $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$, and $\boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_k$ according to
        $$\pi_j^{(t+1)} = \frac{1}{n}\sum_{i=1}^n p_{ij}$$
        $$\boldsymbol{\mu}_j^{(t+1)} = \frac{\sum_{i=1}^n p_{ij}\boldsymbol{x}_i}{\sum_{i=1}^n p_{ij}}$$
        $$\boldsymbol{\Sigma}_j^{(t+1)} = \frac{\sum_{i=1}^n p_{ij}(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t+1)})(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t+1)})^T}{\sum_{i=1}^n p_{ij}}$$
end
```

**Algorithm 5:** Gaussian mixture model using the EM-algorithm

### 8.4.4. Mean-shift clustering

The mean-shift clustering algorithm is similar in spirit to the aforementioned mixture-resolving algorithms. However, instead of explicitly modeling several distributions, mean-shift clustering only considers the modes of the data distribution $p(\boldsymbol{x})$, and does so in a nonparametric manner. Hence, the critical assumption made in mean-shift clustering is that the data distribution is multimodal, and that different modes correspond to different clusters. Let us start with the Kernel Density Estimator [38] for the data distribution[2]:

$$\hat{p}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} K(||\boldsymbol{x} - \boldsymbol{x}_i||)$$

where $K$ is some function (kernel) satisfying

$$0 \leq K(x) < \infty, \forall x \in \mathbb{R} \quad \text{and} \quad \int_{-\infty}^{\infty} K(x)\mathrm{d}x = 1.$$

The gradient of $\hat{p}(\boldsymbol{x})$ is

$$\begin{aligned}\nabla\hat{p}(\boldsymbol{x}) &= \frac{2}{n}\sum_{i=1}^{n}(\boldsymbol{x} - \boldsymbol{x}_i)K'(||\boldsymbol{x} - \boldsymbol{x}_i||) \\ &= \boldsymbol{m}(\boldsymbol{x})\frac{2}{n}\sum_{i=1}^{n} K'(||\boldsymbol{x} - \boldsymbol{x}_i||) \end{aligned} \tag{15}$$

where

$$\boldsymbol{m}(\boldsymbol{x}) = \boldsymbol{x} - \frac{\sum_{i=1}^{n} \boldsymbol{x}_i K'(||\boldsymbol{x} - \boldsymbol{x}_i||)}{\sum_{i=1}^{n} K'(||\boldsymbol{x} - \boldsymbol{x}_i||)}$$

is the *mean-shift vector*. Note that the gradient in Eq. (15) is on the form $c(\boldsymbol{x})\boldsymbol{m}(\boldsymbol{x})$ for some scalar function $c$. Hence, the vector $\boldsymbol{m}(\boldsymbol{x})$ points towards the direction of steepest ascent from $\boldsymbol{x}$. This forms the basis for the mean-shift algorithm for finding the modes of a probability density function. Starting from an initial point $\boldsymbol{x}^{(0)}$ the algorithm iteratively updates $\boldsymbol{x}^{(t)}$ according to the update equation

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} + \boldsymbol{m}(\boldsymbol{x}^{(t)}),$$

which will eventually converge to a mode of $\hat{p}(\boldsymbol{x})$ [65].

The clever trick used to employ the mean-shift algorithm for clustering, is to start the algorithm once from each of the observations in the dataset. Then, all observations converging to the same mode are said to be clustered together. This also brings with it the key property of not having to specify the number of clusters, which is instead determined natively as a part of the algorithm. The mean-shift clustering procedure is summarized in Algorithm 6.

[2]Note that Kernel Density Estimation will be revisited in Part IV.

```
input  : Raw dataset $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$
output: Cluster membership matrix $\boldsymbol{U} = [u_{ij}]$.
foreach $\boldsymbol{x}^{(0)} \in \mathcal{X}$ do
    while not converged do // Mean-shift procedure
        $\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} + \boldsymbol{m}(\boldsymbol{x}^{(t)})$
    end
    Save the point that the mean-shift algorithm converged to
end
Generate $\boldsymbol{U}$ such that all observations that converged to the same
 mode are clustered together.
```

**Algorithm 6:** Mean-shift clustering.

#### 8.4.4.1. Choice of kernel

Although the algorithm has been fully described, the critical choice of a sensible kernel remains. This choice can be highly problem dependent, but one go-to kernel from the literature is the Gaussian kernel:

$$K_{\text{Gauss}}(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

where $\sigma$ is a positive bandwidth parameter. The choice of $\sigma$ determines how smooth the estimate for $\hat{p}(\boldsymbol{x})$ is, and can therefore also influence the number of modes present in $\hat{p}(\boldsymbol{x})$, as well as the number of clusters identified by the algorithm.

# Part III / Deep clustering

The objective of this part is twofold: First we will thoroughly review the deep learning architectures that are used later in this thesis. The review will also contain information on how these models are trained, as well as an explanation of the self-supervised learning approach [36].

After reviewing the deep learning models, we will shift our focus towards some examples of deep clustering models, and a thorough explanation of these. One of these models is the aforementioned Deep Divergence-based Clustering (DDC) [20], which was included due to its relation to the DTKC model proposed in this thesis.

## 9. Frequently used deep learning models

As was previously alluded to in Section 6.4, the more recently developed deep clustering algorithms attempt to combine the data transformation and clustering steps into a single architecture, which can then be simultaneously optimized by minimizing some loss function. As the name "deep clustering" implies, the architectures used for the feature learning step are borrowed from the field of deep learning [11]. These are models belonging to a category often referred to as *deep neural networks* (DNN), as early instances were developed based on the anatomy of the human brain [66].

In the last few decades, a large number of models have been devised under the DNN theme, and naturally, some have been used more than others. The goal of this section is therefore to introduce three types of models that are frequently used in the literature, namely multilayer perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). Both the MLP and the CNN are integral parts of the DTKC model, and will therefore appear in both the model description in Section 13, and in the preceding experiments in Part V. The RNN on the other hand, is in Section 17.9 used to modify the DTKC model, making it applicable to time series clustering.

Following the explanation of these architectures, we will go through autoencoders, as well as the self-supervised learning technique, which is used as an alternate initialization strategy for some of the experiments later in this thesis. Finally, we will take a closer look at neural network optimization, and how this can be done for general loss functions.

## 9.1. Multilayer perceptron

### 9.1.1. The perceptron

The perceptron [66] is a simple linear model for vectorial data. This model is often regarded as the very model that later grew to become the large class of models we know as deep neural networks. The perceptron maps an input vector $\boldsymbol{x}$ to an output scalar $y$ using the linear transformation:

$$y = g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 \tag{16}$$

where $\boldsymbol{w}$ and $w_0$ are the parameters of the model. Originally, the perceptron was formulated as a binary classifier, where the output $y$ would be either close to $-1$ or 1, indicating the classifier's decision. This then motivated the *perceptron loss function*, which is

$$\mathcal{L} = \sum_{\boldsymbol{x} \in \mathcal{X}^*} \delta_{\boldsymbol{x}} (\boldsymbol{w}^T \boldsymbol{x} + w_0)$$

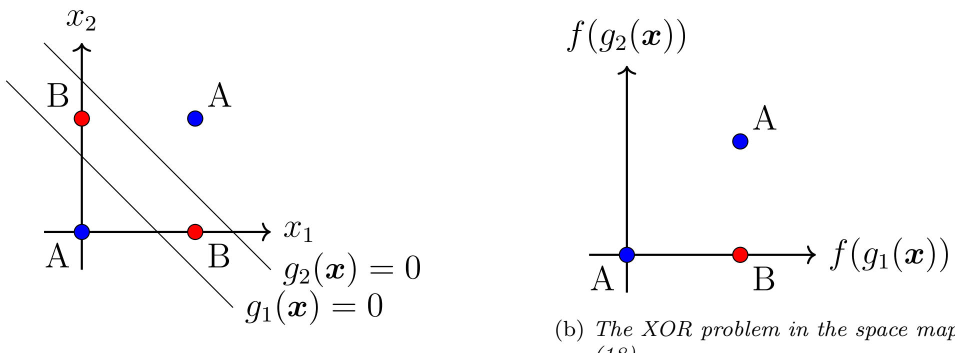
where $\mathcal{X}^*$ are the set of training observations that are misclassified by the system, and $\delta_{\boldsymbol{x}} = \pm 1$, depending on which class $\boldsymbol{x}$ belongs to.

The minimization is done using the gradient descent algorithm, which computes the gradient of the loss function with respect to the weights, and then updates these weights according to the computed gradient. That is:

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + \Delta \boldsymbol{w} = \boldsymbol{w}_{old} - \lambda \nabla_{\boldsymbol{w}_{old}} \mathcal{L}(\boldsymbol{w}_{old}) \tag{17}$$

where $\lambda$ is the learning rate, which is a user-specifiable hyperparameter that scales the magnitude of the parameter-update $\Delta \boldsymbol{w}$.

### 9.1.2. The XOR problem, and perceptrons working together



(a) *The XOR classification problem.*

(b) *The XOR problem in the space mapped to by Eq. (18).*

Figure 9: *XOR problem in original space and transformed space.*

Consider the classification task of Figure 9a, where $[0, 0]$ and $[1, 1]$ belong to class A, and $[1, 0]$ and $[0, 1]$ belong to class B. It can be easily verified that there exists no straight line which perfectly separates the two classes, and thus a linear classifier, like the perceptron, would perform rather poorly in this case.

However, what if two perceptrons are used instead of one? From Figure 9a it can be easily seen that the space between $g_1$ and $g_2$ should be classified to class B, and the space either below $g_1$ or above $g_2$ should be classified to class A. Now consider applying the mapping

$$\boldsymbol{x} \mapsto \begin{bmatrix} f(g_1(\boldsymbol{x})) \\ f(g_2(\boldsymbol{x})) \end{bmatrix}, \qquad f(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{18}$$

to the points in the classification problem of Figure 9a. Note that $g_i(\boldsymbol{x})$ is on the form given in Eq. (16). The result of this mapping is presented in Figure 9b, and as the plot clearly shows, the classes are now linearly separable in this new space. Thus, by using the mapping of Eq. (18) the classification task was transformed from nonlinear to linear. Based on this, a new perceptron could be trained in the mapped-to space, and the classification task would be completed.

Figure 10 provides a graphical overview of the data flow for the two-dimensional two-layer perceptron. The output will be either 0 or 1, depending on which class $\boldsymbol{x}$ is classified to.

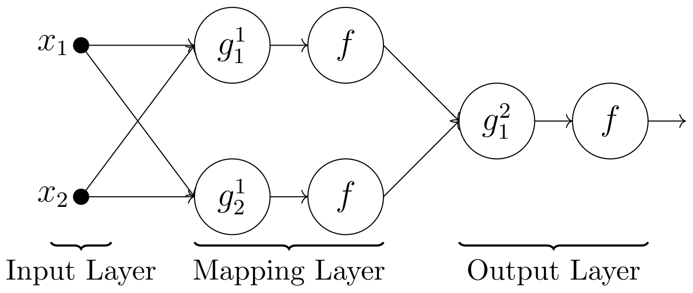
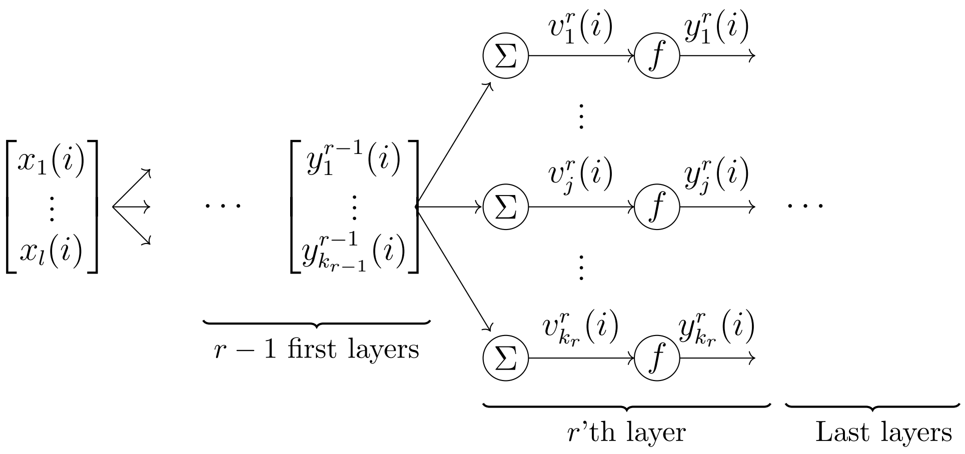


Figure 10: *A visual representation of the two-layer perceptron.*

At this point it is important to note that one mapping will not always be sufficient to make the data linearly separable. The solution to this is the *multilayer perceptron* (MLP), which adds more perceptrons, either in the form of increasing the size of the mapping layer, adding more mapping layers, or both. The individual layers of an MLP are often referred to as *fully-connected* (FC) layers, as each node in a layer receives the output of all nodes in the preceding layer.

### 9.1.3. Training and the backpropagation algorithm

Training a multilayer perceptron is usually done with the *backpropagation* algorithm [67]. In essence, it performs gradient descent with respect to some loss function by analytically computing the gradient, and then updating the parameters accordingly. We will now deviate from the original perceptron loss function,



Symbols:

- $k_r$ number of nodes in layer $r$.
- $[x_1(i), \ldots, x_{k_0}(i)]^T$: Input vector $i$.
- $[y_1^{r-1}(i), \ldots, y_{k_{r-1}}^{r-1}(i)]^T = \boldsymbol{y}^{r-1}$: Output vector from layer $r-1$.
- $\Sigma$: Operator performing the inner product $(\boldsymbol{w}_j^r)^T \boldsymbol{y}^{r-1}(i) + w_0^r$ in layer $r$, node $j$.
- $v_j^r(i)$: Result of inner product in layer $r$, node $j$.
- $y_j^r(i)$: Activation function $f$ applied to the result of the inner product in layer $r$, node $j$.

Figure 11: *A snapshot of the data flow for layer $r$ and input vector $i$ in a multilayer perceptron.*

and instead assume that the loss function is on the form

$$\mathcal{L} = \sum_{i=1}^{N} \varepsilon(i) \tag{19}$$

where $\varepsilon(i)$ is a function representing the severity of the error made when trying to classify the training vector $\boldsymbol{x}_i$, and $N$ is the number of training vectors. Then, calculating the gradient of the loss function with respect to the weights of node $j$ in a layer $r \leq L$ gives:

$$\nabla_{\boldsymbol{w}_j^L} \mathcal{L} = \sum_{i=1}^{N} \nabla_{\boldsymbol{w}_j^L} \varepsilon(i) = \sum_{i=1}^{N} \frac{\partial \varepsilon(i)}{\partial v_j^r(i)} \nabla_{\boldsymbol{w}_j^r} v_j^r(i) \quad \text{(By the chain rule).}$$

Clearly, the dependence of $v_j^r(i)$ in $\varepsilon(i)$ can be quite complex. For now however, define:

$$\delta_j^r(i) = \frac{\partial \varepsilon(i)}{\partial v_j^r(i)} \tag{20}$$

and observe that:

$$\nabla_{\boldsymbol{w}_j^r} v_j^r(i) = \nabla_{\boldsymbol{w}_j^r} \boldsymbol{y}^{r-1}(i)^T \boldsymbol{w}_j^r = \boldsymbol{y}^{r-1}(i).$$

This gives:

$$\Delta \boldsymbol{w}_j^r = -\lambda \sum_{i=1}^{N} \boldsymbol{y}^{r-1}(i) \delta_j^r(i) \tag{21}$$

which is the desired correction for the weights in node $j$ in layer $r$. $\lambda$ is the learning rate. Note that if $r = 1$, then $\boldsymbol{y}^{r-1}(i)$ would be the training vector $\boldsymbol{x}_i$.

These last two results require the activation function $f$ to be continuously differentiable, a property clearly not possessed by $f$ in Eq. (18). To fix this, the step-activation is swapped for a smooth function with the same asymptotic properties. Commonly used activation functions include the hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

or the sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Another popular choice is the softmax function, which is a normalized version of the sigmoid, where one ensures that the outputs from the nodes in a given layer, say $l$, sum to one:

$$\text{softmax}(y_i) = \frac{\sigma(y_i^l)}{\sum_{j=1}^{k_l} \sigma(y_j^l)}.$$

This activation function is frequently used when one wants the outputs $y_1^l, \dots y_{k_l}^l$ to reflect some probability mass function.

## 9.2. Convolutional neural networks

Convolutional neural networks (CNNs) [25] represent a type of deep neural networks that mimics the behavior of the multilayer perceptron, where each layer is made out of simple processing units (nodes), and each layer produces its output based on all the outputs from the preceding layer. However, in contrast to the inner product in MLPs, the units in a CNN uses the well known convolution operator to produce their output. As we will see in the following, the convolution operator is well suited for data which is inherently grid-based, such as images and time series.

### 9.2.1. The convolution operator

Since CNNs will be used to process images in this thesis, the convolution operator will be introduced here in the context of images. Suppose we have an $N_A \times M_A$ image $\boldsymbol{A}$, and an $N_B \times M_B$ image $\boldsymbol{B}$. In many image processing tasks, the latter image can be referred to as the *filter* or *kernel*[3], as it has been carefully selected to emphasize some property of the image. The convolution of the image $\boldsymbol{A}$ with $\boldsymbol{B}$ is the new image:

$$(\boldsymbol{A} \star \boldsymbol{B})(i,j) = \sum_a \sum_b \boldsymbol{A}(a,b)\boldsymbol{B}(i-a, j-b) \tag{22}$$

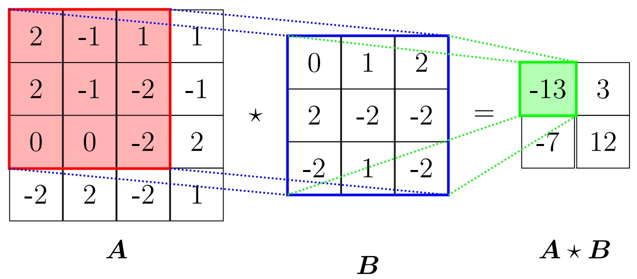
where the dimensions of the resulting image $\boldsymbol{A} \star \boldsymbol{B}$, and thereby also the summation limits, depend on which type of convolution is being performed. The three most common forms of convolution are:

- *Valid:* Only the indices for which $1 \leq a \leq N_A$, $1 \leq b \leq M_A$ and $1 \leq i-a \leq N_B$, $1 \leq j-b \leq M_B$ are summed over. The result has dimensions $|N_A - N_B| + 1 \times |M_A - M_B| + 1$. See Figure 12a.
- *Same:* The first input image is zero-padded such that the output has the same size as $\boldsymbol{A}$. See Figure 12b.
- *Full:* Either of the images are padded such that all overlaps with size greater than or equal to one, are included in the output. The resulting image has dimensions $N_A + N_B - 1 \times M_A + M_B - 1$. See Figure 12c.

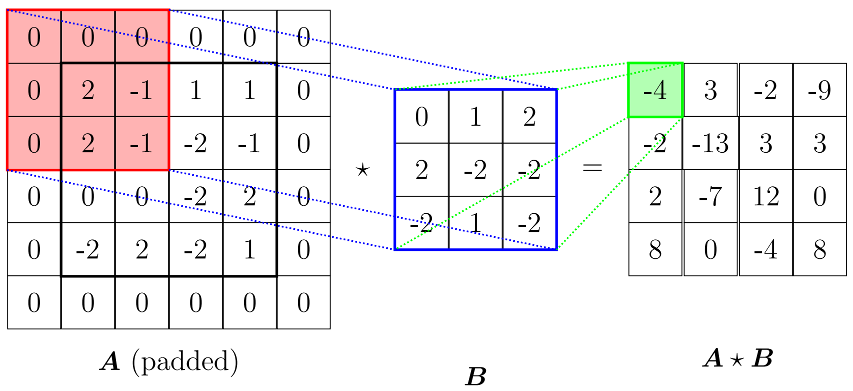
#### 9.2.1.1. Examples

Two examples of image filtering using convolution are shown in Figure 13. The image in Figure 13b was obtained by convolving the original image with a filter of size $11 \times 11$, where each value was $1/11^2$. This effectively makes each output

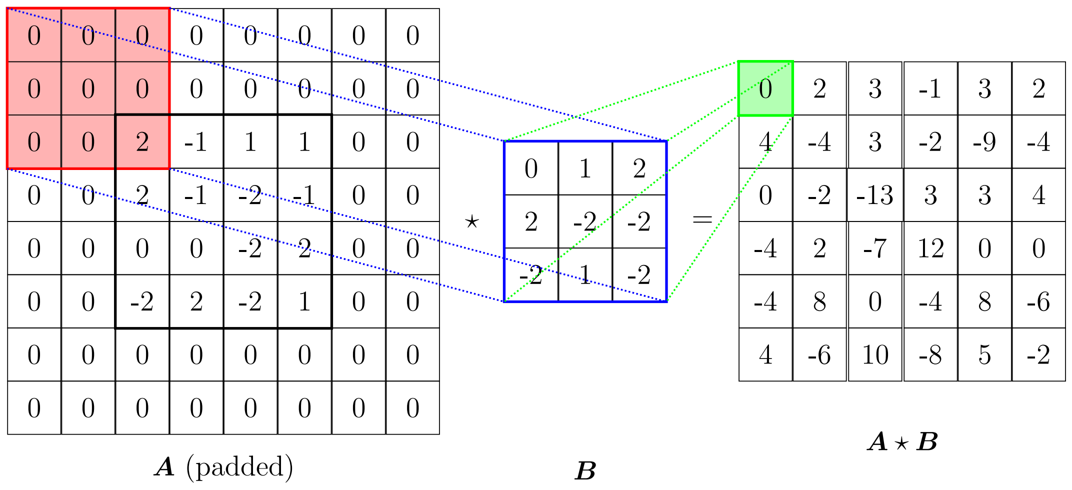
[3]We will refrain from using this name, as it can be confused with other types of "kernels" defined later.



(a) *Valid convolution.*



(b) *Same convolution.*



(c) *Full convolution.*

Figure 12: *Different types of padding schemes for discrete convolution. Note that the filter,* $\boldsymbol{B}$ *is flipped up-down and left-right before the element-wise multiplication, as is indicated in Eq. (22).*

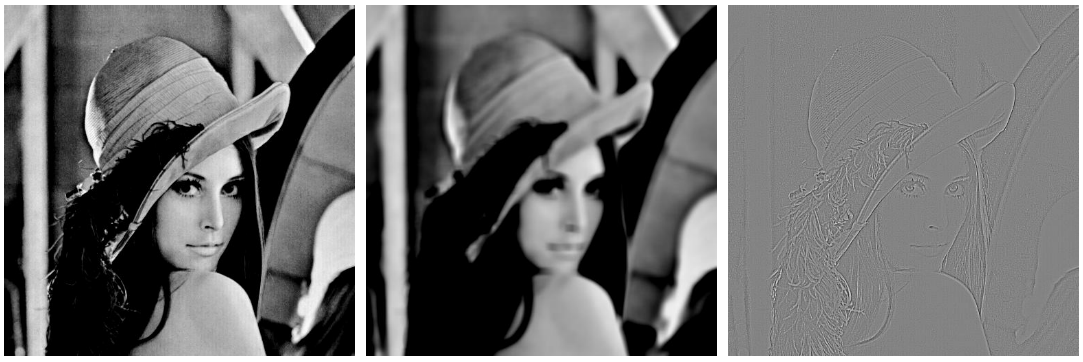

(a) *Original* (b) *Smoothed* (c) *Edges detected*

Figure 13: *Examples of filtered images using convolution.*

pixel a local average of its neighboring pixels, resulting in a smoothed image. The image in Figure 13c was obtained using the Laplacian filter:

$$\boldsymbol{B}_{\text{Laplace}} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

which is a widely used edge-detection filter.

### 9.2.1.2. Properties of the convolution operator

The convolution operator is both associative: $(\boldsymbol{A} \star \boldsymbol{B}) \star \boldsymbol{C} = \boldsymbol{A} \star (\boldsymbol{B} \star \boldsymbol{C})$, and commutative: $\boldsymbol{A} \star \boldsymbol{B} = \boldsymbol{B} \star \boldsymbol{A}$. Additionally, it has two nice properties which makes it especially suitable for image processing; the first one being linearity:

$$\boldsymbol{A} \star (b\boldsymbol{B} + c\boldsymbol{C}) = b(\boldsymbol{A} \star \boldsymbol{B}) + c(\boldsymbol{A} \star \boldsymbol{C})$$

where $b$ and $c$ are arbitrary scalars. The second property is translation equivariance:

$$T(\boldsymbol{A}) \star \boldsymbol{B} = T(\boldsymbol{A} \star \boldsymbol{B})$$

where $T$ is a translation function, with $T(\boldsymbol{A})(i,j) = \boldsymbol{A}(i + t_1, j + t_2)$, where $t_1$ and $t_2$ are integers. This last property means that the response for a specific object in an image will be the same regardless of that object's position in the image. The response will simply be located at the corresponding position in the output image.

### 9.2.1.3. Images with multiple channels

There are several ways to handle the case of images with multiple channels when designing the convolution operator. If we have a filter with a single channel, the classical solution is to simply perform the convolution channel-by-channel. Another option is to define a filter with the same number of channels as the

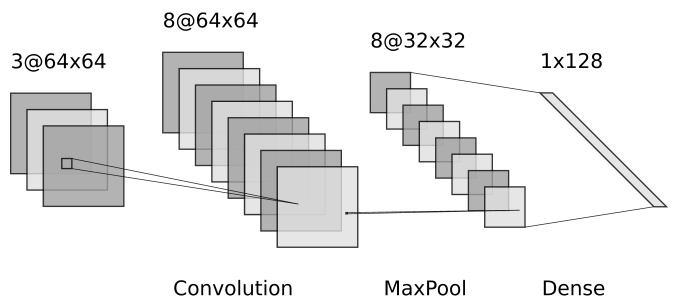


Figure 14: *A simple convolutional neural network. "Dense" refers to a perceptron layer, which is commonly used as a last layer to produce vectorial outputs.*

input image, say $C$, and then generalize the convolution operator in the following manner:

$$(\boldsymbol{A} \star \boldsymbol{B})(i,j) = \sum_{a} \sum_{b} \sum_{c=1}^{C} \boldsymbol{A}(a,b,c)\boldsymbol{B}(i-a, j-b, c) \tag{23}$$

which produces an output image with one channel. Adding more output channels can be solved by simply adding more filters.

### 9.2.2. Convolutional layers

A convolutional neural network, is a deep neural network where one or more of the layers are *convolutional layers*. In a convolutional layer, the nodes perform convolutions, as in Eq. (23), instead of the inner product done by the MLP. An example of a simple convolutional layer can be found in the first transition in Figure 14. Here a 3-channel $64 \times 64$ input image is processed by 8 filters, producing 8 1-channel images of size $64 \times 64$, or equivalently, one 8-channel image of size $64 \times 64$.

Note that there is some ambiguity in the literature regarding the term "convolutional layer" [12]. One option is to strictly define the layer as above, where the layer consists of only the convolution operator. However, this can make the explanation of more complex CNN-based models somewhat cumbersome, which has led to the inclusion of activation functions and pooling operations (explained below) in the term "convolutional layer". In this thesis, we will adhere to this second, somewhat looser definition – in order to make the description of future models easier.

#### 9.2.2.1. Activation function

Recall that convolution is a linear operator, and that the composition of linear operators is itself linear. Thus, a convolutional neural network consisting solely of convolutional layers will be a linear map from the input space to the output space. To take potentially nonlinear structure in the data into account, a nonlinear activation function is usually applied within convolutional layers, in an element-wise fashion. Early work on CNNs focused on squashing nonlinearities, like the sigmoid or hyperbolic tangent [25]. However, recently it has been shown that networks using rectifier-activations tend to improve performance by allowing for better gradient flow during training [68, 69, 70]. The rectified linear unit (ReLU) is defined as:

$$\mathrm{ReLU}(x) = \max(0, x).$$

Its derivative is the unit-step function:

$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{ReLU}(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

which, in contrast to the derivatives of squashing nonlinearities, does not vanish for large values of $x$. This property is precisely what helps improve gradient flow during training. Note that $\mathrm{ReLU}(x)$ is not actually differentiable at $x = 0$, which is a problem since backpropagation requires differentiable loss functions. The common approach to bypass this problem is to simply define the gradient $\frac{\mathrm{d}}{\mathrm{d}x}\big|_{x=0}\mathrm{ReLU}(x)$ to be equal to zero [69].

#### 9.2.2.2. Pooling

Depending on the problem at hand, it might be beneficial for objects located at approximately the same location in the image to have the same output. An extreme example of such a problem is an object detector, where the only task is to detect whether a specific object is in the image, or not. At some point, the network has to translate the information ”object located at position $(x, y)$”, to just ”object exists somewhere in the image”. This motivates the use of Pooling-layers, which divide the input image into non-overlapping patches of equal size, and then compute some scalar statistic for each patch. The statistic can be e.g. the maximum value, or the average of the given patch. When the former is used, the operation is referred to as *max pooling* or *MaxPool*. As is indicated in Figure 14, the Pooling operation is applied on a channel-by-channel basis. Pooling layers also have the added benefit of reducing the size of the intermediate representations, decreasing memory consumption, and allowing for faster processing in subsequent layers.

#### 9.2.2.3. Weight sharing

As was previously stated, the most important distinction between the multilayer perceptron and the convolutional neural network, is that the CNN replaces the MLPs inner product with convolution. This also brings with it the notion of *shared weights* [25]. Since the convolution operator ”slides” the same filter across the input image, the filter can be chosen to have dimensions much smaller than the input. Hence, the same weights (elements of the filter) are used at multiple locations in the input, i.e. they are shared across the spatial dimensions of the image. An MLP designed to process the same input would need to have a weight vector with number of elements equal to the number of pixels in the input, leading to an increase in the number of parameters compared to the CNN. Thus, the memory consumed by a CNN can be drastically smaller than the memory consumed by a similarly sized MLP.

### 9.2.3. Computing the gradient of a convolutional layer

In contrast to classical signal processing tasks, the filters used in convolutional neural networks are not explicitly selected, but rather learned from data. Usually, this is done with gradient descent, similar to the training of MLPs. Suppose we have a loss function $\mathcal{L}$, and we are interested in computing its gradient with respect to $K(i, j, l)$, which will denote element $(i, j, l)$ of the filter $K$. Define:

$$Y = a(V), \quad V = X \star K,$$

as the output of the layer, after application of the activation function. Here, $X$ is the input to the convolution layer, and $a$ is the activation function, which is applied element-wise. From the chain-rule, we get:

$$\frac{\partial \mathcal{L}}{\partial K(i,j,l)} = (\nabla_Y \mathcal{L})^T \frac{\partial Y}{\partial K(i,j,l)} \tag{24}$$

where we assume $\nabla_Y \mathcal{L}$ has been computed in later layers of the network. Thus, it remains to compute:

$$\frac{\partial Y}{\partial K(i,j,l)} = \left[\frac{\partial Y(1,1)}{\partial K(i,j,l)}, \ldots, \frac{\partial Y(m,n)}{\partial K(i,j,l)}\right]^T$$

where $(m, n)$ are the dimensions of $Y$. For element $(x, y)$ we get:

$$\begin{aligned}
\frac{\partial Y(x,y)}{\partial K(i,j,l)} &= a'(V(x,y)) \frac{\partial V(x,y)}{\partial K(i,j,l)} \\
&= a'(V(x,y)) \sum_{abc} \frac{\partial}{\partial K(i,j,l)} K(a,b,c) X(x-a, y-b, c) \\
&= a'(V(x,y)) X(x-i, y-j, l).
\end{aligned}$$

Substituting this result back into Eq. (24) gives:

$$\frac{\partial \mathcal{L}}{\partial K(i,j,l)} = (\nabla_Y \mathcal{L})^T \begin{bmatrix} a'(V(1,1)) X(1-i, 1-j, l) \\ \vdots \\ a'(V(m,n)) X(m-i, n-j, l) \end{bmatrix}$$

which is the desired update for $K(i,j,l)$. Analogously, we can obtain the derivative of $\mathcal{L}$ with respect to the input, $X$ as:

$$\frac{\partial \mathcal{L}}{\partial X(i,j,l)} = (\nabla_Y \mathcal{L})^T \begin{bmatrix} a'(V(1,1))K(1-i,1-j,l) \\ \vdots \\ a'(V(m,n))K(m-i,n-j,l) \end{bmatrix}$$

which can then be propagated backwards to earlier layers. The generalization to several filters, and several inputs is straightforward.

## 9.3. Recurrent neural networks

We will now shift our focus to the type of data which incorporates some form of sequential dependence. This type of data mainly includes sequences, and more specifically, time series, which are sequences formed through the evolution of time. Due to the nature of the data-generating process, the vast majority of these sequences contain some form of dependence between different timesteps. Therefore, it is most natural that the way we process element $t$ of the sequence should not only depend on observation $t$ itself, but also on the preceding elements, 1 through $t-1$, so that we are able to capture this temporal dependence.

Recurrent Neural Networks (RNNs) are a class of deep neural networks designed to take the temporal dependence into account. In general, the RNN model is a pair of recurrence relations on the form

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}; \boldsymbol{\theta}_f), \quad \boldsymbol{y}^{(t)} = g(\boldsymbol{h}^{(t)}; \boldsymbol{\theta}_g) \tag{25}$$

where $\boldsymbol{x}^1, \dots, \boldsymbol{x}^T$ is the input sequence, $\boldsymbol{h}^{(t)}$ is the *state* of the RNN at time $t$, $\boldsymbol{y}^{(t)}$ is the output of the RNN at time $t$, and $(\boldsymbol{\theta}_f, \boldsymbol{\theta}_g)$ are parameter vectors. From this we can see that the current state $\boldsymbol{h}^{(t)}$ depends on all previous states and inputs through the recurrent connection with $\boldsymbol{h}^{(t-1)}$. The exact form of the functions $f$ and $g$ can vary based on which type of RNN is being used. Moreover, it should be noted that both the parameter vectors $\boldsymbol{\theta}_f$ and $\boldsymbol{\theta}_g$, and the functions $f$ and $g$ do not depend on $t$, meaning that these are constant with respect to time. Hence, the RNN also employs the concept of parameter sharing, which we previously saw in CNNs.

### 9.3.1. The Elman RNN

The Elman RNN (ERNN) [71] is one of the first types of RNN to be proposed. Here, the function $f$ consists on an affine transformation of the pair $(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)})$, followed by a sigmoid-activation function:

$$\boldsymbol{h}^{(t)} = \sigma(\boldsymbol{W}\boldsymbol{x}^{(t)} + \boldsymbol{U}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}). \tag{26}$$

The output of the ERNN at time $t$ is then

$$\boldsymbol{y}^{(t)} = \sigma(\boldsymbol{W}_y \boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$

where $(\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{b}, \boldsymbol{W}_y, \boldsymbol{b}_y)$ are learnable parameters, and $\sigma(x) = (1 + e^{-x})^{-1}$ is the sigmoid-function, which is applied element-wise.

### 9.3.2. Vanishing or exploding gradients

RNNs are trained using a generalization of the backpropagation algorithm, referred to as backpropagation through time (BPTT)[4]. However, it was quickly discovered that the simple ERNN-architecture had problems with this training regime, especially when it came to modeling long-term dependencies [72, 73]. For now, consider the simple RNN computing

$$\boldsymbol{h}^{(t+1)} = f(\boldsymbol{h}^{(t)})$$

for some function $f$. The Jacobian $d\boldsymbol{h}^{(t+1)}$ of $\boldsymbol{h}^{(t+1)}$ is

$$d\boldsymbol{h}^{(t+1)} = df(\boldsymbol{h}^{(t)}) d\boldsymbol{h}^{t} = df(\boldsymbol{h}^{(t)}) df(\boldsymbol{h}^{(t-1)}) \cdots df(\boldsymbol{h}^{(1)})$$

by the chain rule, where $df$ denotes the Jacobian of $f$. If we take $f$ to be a linear mapping with matrix $\boldsymbol{A}$, we get $d\boldsymbol{h}^{(t+1)} = \boldsymbol{A}^t$, which means that, for an initial gradient $\boldsymbol{g}^{(1)}$, we have $\boldsymbol{g}^{(t)} = \boldsymbol{A}^t \boldsymbol{g}^{(1)}$ for the gradient at time $t$. Let $\boldsymbol{A} = \boldsymbol{E}\boldsymbol{\Lambda}\boldsymbol{E}^{-1}$ be the eigendecomposition of $\boldsymbol{A}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\boldsymbol{A}$. $\boldsymbol{A}^t$ then has eigendecomposition $\boldsymbol{A}^t = \boldsymbol{E}\boldsymbol{\Lambda}^t\boldsymbol{E}^{-1}$. Thus, eigenvalues whose absolute values are not close to one will either explode or vanish, also causing the gradient to act similarly [12]. Furthermore, it was shown in [73] that the gradient of a long-term interaction is vanishingly small compared to the gradient of a short-term interaction, meaning that one cannot simply solve the problem by "constraining the parameters such that the eigenvalues of the Jacobian are close to one".

In practice, the assumptions made about the form of the recurrence relation in the preceding argument, are somewhat unrealistic. However, it turns out that simpler nonlinear RNN architectures, like the ERNN, still exhibit the same type of behavior [12].

### 9.3.3. Multilayer RNNs

Similarly to the aforementioned DNN architectures, RNNs can also be stacked to produce a "deeper" model. This is done by taking the hidden state $\boldsymbol{h}^{(t)}$ and passing it on to another RNN operation, instead of using it to compute the output

[4]BPTT will be revisited in more detail later in this section.

directly. For an RNN of depth $D$, we get one sequence of hidden states for each layer:

$$\begin{aligned}
\boldsymbol{h}_1^{(t)} &= f_1(\boldsymbol{h}_1^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}_{f,1}) \\
\boldsymbol{h}_2^{(t)} &= f_2(\boldsymbol{h}_2^{(t-1)}, \boldsymbol{h}_1^{(t)}; \boldsymbol{\theta}_{f,2}) \\
&\vdots \\
\boldsymbol{h}_D^{(t)} &= f_D(\boldsymbol{h}_D^{(t-1)}, \boldsymbol{h}_{D-1}^{(t)}; \boldsymbol{\theta}_{f,D}).
\end{aligned}$$

The output $\boldsymbol{y}^{(t)}$ is then computed as a function of the topmost hidden state:

$$\boldsymbol{y}^{(t)} = g(\boldsymbol{h}_D^{(t)}; \boldsymbol{\theta}_g)$$

### 9.3.4. Bidirectional RNNs

In some applications, it might be beneficial to model the system based on temporal dependence in both the forward and backward directions. With RNNs, this can be done by having two separate instances of the recurrence relations, where the first instance receives the original sequence $\boldsymbol{x}_1, \dots \boldsymbol{x}_T$, while the second instance receives the *reversed* sequence $\boldsymbol{x}_T, \dots \boldsymbol{x}_1$. I.e:

$$\boldsymbol{h}_F^{(t)} = f_F(\boldsymbol{x}^{(t)}, \boldsymbol{h}_F^{(t-1)}; \boldsymbol{\theta}_{f,F}) \qquad \boldsymbol{h}_B^{(t)} = f_B(\boldsymbol{x}^{(T-t+1)}, \boldsymbol{h}_B^{(t-1)}; \boldsymbol{\theta}_{f,B})$$

where the subscripts $F$ and $B$ denote "forwards" and "backwards", respectively. The output is then computed based on both of the current hidden states:

$$\boldsymbol{y}^{(t)} = g(\boldsymbol{h}_B^{(t)}, \boldsymbol{h}_F^{(t)}; \boldsymbol{\theta}_g).$$

### 9.3.5. Gated RNNs

To overcome the gradient problems often exhibited by simpler RNN architectures, many modifications to the classical models have been proposed. Among these are the Long Short-Term Memory (LSTM) [74], and Gated Recurrent Unit (GRU) [75]. Both of these architectures keep a persistent state-vector which is only updated through element-wise affine transformations, whose slopes change over time. This allows for better gradient flow throughout the network.

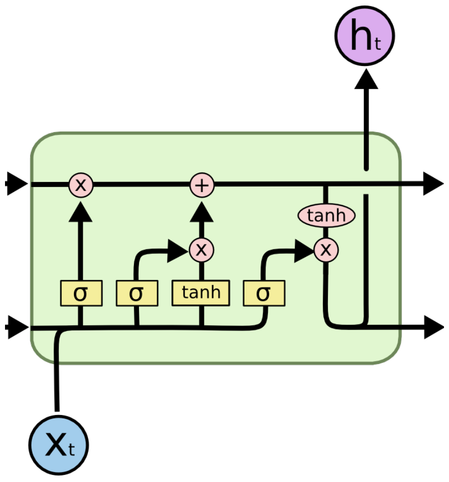


Figure 15: *The computational flow within an LSTM. The horizontal line on the top indicates the cell state* $\mathbf{c}^{(t)}$*, while the horizontal line at the bottom indicates the hidden state* $\mathbf{h}^{(t)}$*. Figure from [76]*

### 9.3.5.1. Long short-term memory

A complete overview of the connections within the LSTM is given in Figure 15. For each time step $t$, the *cell state* $\boldsymbol{c}^{(t)}$ and *hidden state* $\boldsymbol{h}^{(t)}$ are computed as:

$$\begin{aligned}
\boldsymbol{f}^{(t)} &= \sigma(\boldsymbol{W}_f\boldsymbol{x}^{(t)} + \boldsymbol{U}_f\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f) \\
\boldsymbol{i}^{(t)} &= \sigma(\boldsymbol{W}_i\boldsymbol{x}^{(t)} + \boldsymbol{U}_i\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i) \\
\tilde{\boldsymbol{c}}^{(t)} &= \tanh(\boldsymbol{W}_c\boldsymbol{x}^{(t)} + \boldsymbol{U}_c\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c) \\
\boldsymbol{c}^{(t)} &= \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{c}}^{(t)} \\
\boldsymbol{o}^{(t)} &= \sigma(\boldsymbol{W}_o\boldsymbol{x}^{(t)} + \boldsymbol{U}_o\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o) \\
\boldsymbol{h}^{(t)} &= \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{c}^{(t)})
\end{aligned}$$

where $\odot$ denotes element-wise multiplication. If we first look at the cell state $\boldsymbol{c}^{(t)}$, we see that it is indeed updated by an affine transformation. The slope, decided by $\boldsymbol{f}^{(t)}$ is referred to as the output of the forget-gate, which decides how much of the previous cell state that should be kept (or forgot). The output of the input-gate, $\boldsymbol{i}^{(t)}$ decides how much of the new proposed cell state, $\tilde{\boldsymbol{c}}^{(t)}$, that should be added to the current cell state. The hidden state $\boldsymbol{h}^{(t)}$ is not updated via an affine transformation, but rather as a squashed version of the cell state, which is scaled by the output of the output-gate, $\boldsymbol{o}^{(t)}$.

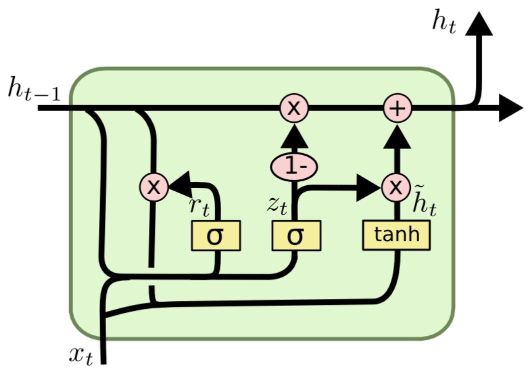


Figure 16: *Flowchart for the Gated Recurrent Unit. Figure from [76]*

#### 9.3.5.2. Gated recurrent unit

The computations within the GRU are illustrated in Figure 16. The update-equations are as follows:

$$\begin{aligned}
\boldsymbol{h}^{(t)} &= (\boldsymbol{1} - \boldsymbol{z}^{(t)}) \odot \boldsymbol{h}^{(t-1)} + \boldsymbol{z}^{(t)} \odot \tilde{\boldsymbol{h}}^{(t)} \\
\boldsymbol{z}^{(t)} &= \sigma(\boldsymbol{W}_z \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_z \boldsymbol{x}^{(t)} + \boldsymbol{b}_z) \\
\tilde{\boldsymbol{h}}^{(t)} &= \tanh(\boldsymbol{W}(\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{b}) \\
\boldsymbol{r}^{(t)} &= \sigma(\boldsymbol{W}_r \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r \boldsymbol{x}^{(t)} + \boldsymbol{b}_r).
\end{aligned}$$

The GRU can be seen as a simplification of the LSTM, where the forget-gate and input-gate have been replaced by a single update-gate ($\boldsymbol{z}^{(t)}$). The cell state has also been removed, meaning that only the hidden state remains. Furthermore, the updating of the hidden state is now being done via an affine transformation, similarly to the cell state in the LSTM. The coefficients of the transformation are determined by the output-gate.

### 9.3.6. Computing the gradient in an RNN

As was previously stated, RNNs can be trained using "backpropagation through time" (BPTT), which is a variant of the ordinary backpropagation algorithm adapted to the RNN model. Suppose that, at time $t$, we have a scalar loss function $\mathcal{L}^{(t)}(\boldsymbol{y}^{(t)})$, which we wish to compute the gradient of, with respect to the parameter vectors $\boldsymbol{\theta}_f$ and $\boldsymbol{\theta}_g$, for the general RNN in Eq. (25). Starting with $\boldsymbol{\theta}_g$ we get:

$$\nabla_{\boldsymbol{\theta}_g} \mathcal{L}^{(t)} = (\nabla_{\boldsymbol{y}^{(t)}} \mathcal{L}^{(t)})^T \cdot d_{\boldsymbol{\theta}_g} \boldsymbol{y}^{(t)}$$

where $d_{\boldsymbol{\theta}_g} \boldsymbol{y}^{(t)}$ denotes the Jacobian of $\boldsymbol{y}^{(t)}$ with respect to $\boldsymbol{\theta}_g$. For $\boldsymbol{\theta}_f$ we get:

$$\nabla_{\boldsymbol{\theta}_f} \mathcal{L}^{(t)} = (\nabla_{\boldsymbol{y}^{(t)}} \mathcal{L}^{(t)})^T \cdot d_{\boldsymbol{h}^{(t)}} \boldsymbol{y}^{(t)} \cdot d_{\boldsymbol{\theta}_f} \boldsymbol{h}^{(t)}$$

and:

$$d_{\boldsymbol{\theta}_f}\boldsymbol{h}^{(t)} = d_{\boldsymbol{\theta}_f}\boldsymbol{h}^{(t)^*} + d_{\boldsymbol{h}^{(t-1)}}\boldsymbol{h}^{(t)} \cdot d_{\boldsymbol{\theta}_f}\boldsymbol{h}^{(t-1)}$$

where $\boldsymbol{h}^{(t)^*}$ is a variable which is identical to $\boldsymbol{h}^{(t)}$, with the exception of $\boldsymbol{h}^{(t-1)}$ being treated as a constant.

For the Elman-RNN, the elements of the Jacobians are:

$$\begin{aligned}
\frac{\partial y_i^{(t)}}{\partial w_{y,jk}} &= \sigma'(v_{y,i})\delta(i,j)h_k^{(t)} & \frac{\partial y_i^{(t)}}{\partial b_{y,j}} &= \sigma'(v_{y,i})\delta(i,j) \\
\frac{\partial h_i^{(t)^*}}{\partial w_{jk}} &= \sigma'(v_i)\delta(i,j)x_k^{(t)} & \frac{\partial h_i^{(t)^*}}{\partial u_{jk}} &= \sigma'(v_i)\delta(i,j)h_k^{(t-1)} \\
\frac{\partial h_i^{(t)^*}}{\partial b_j} &= \sigma'(v_i)\delta(i,j) & \frac{\partial h_i^{(t)}}{\partial h_j^{(t-1)}} &= \sigma'(v_i)u_{ij}
\end{aligned}$$

where $v_{y,i}$ denotes the $i$-th element of $\boldsymbol{W}_y\boldsymbol{h}^{(t)}+\boldsymbol{b}_y$, and $v_i$ denotes the $i$-th element of $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{U}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}$. The function $\delta(i,j)$ is the Kronecker-delta, which is 1 if $i = j$, and 0 otherwise. The generalization to bidirectional RNNs, multiple layers, and several inputs is straightforward.

## 9.4. Autoencoders

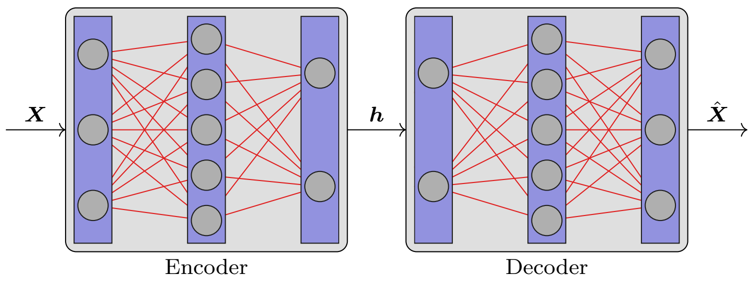


Figure 17: *A simple autoencoder. The input $\boldsymbol{X}$ is processed by the encoder, producing $\boldsymbol{h} = Enc_{\boldsymbol{\theta}_e}(\boldsymbol{X})$. Then the reconstructed input is computed by the decoder as $\hat{\boldsymbol{X}} = Dec_{\boldsymbol{\theta}_d}(\boldsymbol{h})$.*

The autoencoder [77, 78, 12] is a neural network based model designed for unsupervised feature learning. In essence, the main objective of an autoencoder is to reconstruct an input observation as best it can. This might seem tautological at first, as the trivial identity mapping is indeed capable of perfectly reconstructing its input. Thus, it is required by the vast majority of autoencoders that at least one of the intermediate representations within the autoencoder has a dimensionality which is *smaller* than that of the input data. This is sometimes referred to as the *bottleneck principle*, and has led to the success of autoencoders in many machine learning tasks [77, 79, 80, 15] [5].

[5]These types of autoencoders are sometimes also referred to as *undercomplete*, whereas autoencoders implementing the constraint of nontriviality through other means are referred to as *overcomplete*. The latter will not be covered in more detail in this thesis.

Mathematically, the autoencoder consists of a pair of parameterized functions, namely the encoder $\mathrm{Enc}_{\boldsymbol{\theta}_e} : X \to H$, and the decoder $\mathrm{Dec}_{\boldsymbol{\theta}_d} : H \to X$, where $X$ is the space containing the input data, and $H$ is a vector space which contains the learned representations. $\boldsymbol{\theta}_e$ and $\boldsymbol{\theta}_d$ denote parameter vectors for the encoder and decoder, respectively. The bottleneck principle now reads $\dim(H) < \dim(X)$, which immediately implies $\mathrm{Enc}_{\boldsymbol{\theta}_e} \neq \mathrm{Id}$ and $\mathrm{Dec}_{\boldsymbol{\theta}_d} \neq \mathrm{Id}$, where Id denotes the identity mapping. This causes the aforementioned trivial solution to be avoided. Figure 17 shows an example of a simple autoencoder. Here, both the encoder and decoder consist of three fully-connected layers.

The autoencoder can represented by the composition $\mathrm{Dec}_{\boldsymbol{\theta}_d} \circ \mathrm{Enc}_{\boldsymbol{\theta}_e} : X \to X$, whose reconstruction error we seek to minimize. For a dataset $\mathcal{X} = \{\boldsymbol{X}_1, \dots, \boldsymbol{X}_n\}$ we have the loss function

$$\mathcal{L}_{\mathrm{ae}}(\boldsymbol{\theta}_e, \boldsymbol{\theta}_d; \mathcal{X}) = \frac{1}{n} \sum_{i=1}^{n} ||\boldsymbol{X}_i - (\mathrm{Dec}_{\boldsymbol{\theta}_d} \circ \mathrm{Enc}_{\boldsymbol{\theta}_e})(\boldsymbol{X}_i)||_X^2$$

where $|| \cdot ||_X^2$ denotes some norm on $X$. If $X$ is a vector space, the squared Euclidean norm is frequently used. When the autoencoder's training procedure is finished, the encoder $\mathrm{Enc}_{\boldsymbol{\theta}_e}$ can be used to obtain the learned representation for arbitrary input elements from $X$.

### 9.4.1. Connection to principal component analysis

If we let $X$ be a vector space, and $\mathrm{Enc}_{\boldsymbol{\theta}_e}$ and $\mathrm{Dec}_{\boldsymbol{\theta}_d}$ be linear transformations, it can be shown that minimizing the mean squared error loss $\mathcal{L}_{\mathrm{ae}}$ is equivalent to performing PCA on zero-mean data [77]. Thus, in this simple case, the matrix for the encoding transformation can be obtained by stacking eigenvectors from the estimated covariance matrix of the input data.

### 9.4.2. Deep autoencoders and other variations

Many different takes on the classical autoencoder framework have been proposed over the years. One of the most noticeable trends – which has also influenced the entire field of deep neural networks, due to the increase in computational capacity – is to build deeper models consisting of an increasing amount of layers, both in the encoder and in the decoder.

It should also be noted that the autoencoder framework is not limited to MLPs, but can also employ both convolutional and recurrent layers. A CNN-based autoencoder was proposed by Masci et al. [81], and an RNN-based autoencoder was introduced by Srivastava et al. [82].

Although all layers of a deep autoencoder can be trained simultaneously, a greedy layer-wise training procedure has been shown to yield better representations in some cases [83, 80, 81]. These *stacked* autoencoders are formed by first training a shallow two-layer autoencoder, and then using the representation provided by this autoencoder to train another two-layer autoencoder "between" the two initial layers. This process can be repeated until the desired number of layers is reached.

Another commonly applied modification to the autoencoder framework is to train with a *denoising criterion* [79, 80]. In this case, the autoencoder is trained to reconstruct a noise-free version of an input observation, based on a noisy version of said input. Thus, the denoising autoencoder introduces a stochastic corruption process $\eta$ which produces a corrupted version of a given input observation. The loss function is therefore

$$\mathcal{L}_{dae}(\boldsymbol{\theta}_e, \boldsymbol{\theta}_d, \mathcal{X}) = \frac{1}{n} \sum_{i=1}^{n} ||\boldsymbol{X}_i - (\text{Dec}_{\boldsymbol{\theta}_d} \circ \text{Enc}_{\boldsymbol{\theta}_e} \circ \eta)(\boldsymbol{X}_i)||_X^2.$$

Vincent et al. [80] empirically demonstrate that the features learned using a stacked denoising autoencoder are better suited for classification and reconstruction, compared to similarly constructed stacked autoencoders.

## 9.5. Self-supervised learning

Self-supervised learning [36] is an approach to learning image representations in an unsupervised manner. The main idea is to construct a model that can be trained in a supervised fashion, based on an unlabeled dataset. One approach to this is through the task of *context prediction*, where the model learns to predict the location of an image patch, relative to another patch extracted from the same image. This approach to self-supervised learning is the one we will use throughout this thesis. Doersch et al. [36] argue that a model which is trained to perform this task learns to recognize characteristic parts of the objects present in the dataset. This is indeed intuitive as determining the relative location of one patch with respect to another becomes much easier when one is able to identify the object in the image.

### 9.5.1. Model description

The model used for the prediction task is a convolutional neural network, followed by a number of fully-connected layers. The input to the model is illustrated in Figure 18. Note that the different patches outlined in Figure 18 are not contiguous, but rather separated and jittered relative to each other. This is done to avoid "trivial" solutions where the model simply learns to continue patterns or textures in the image.

The input $\boldsymbol{X}$ consists of the two image-patches, which are processed independently by the convolutional layers of the model, as well as the first fully connected layer. Note that the output of the last convolutional layer has to be vectorized

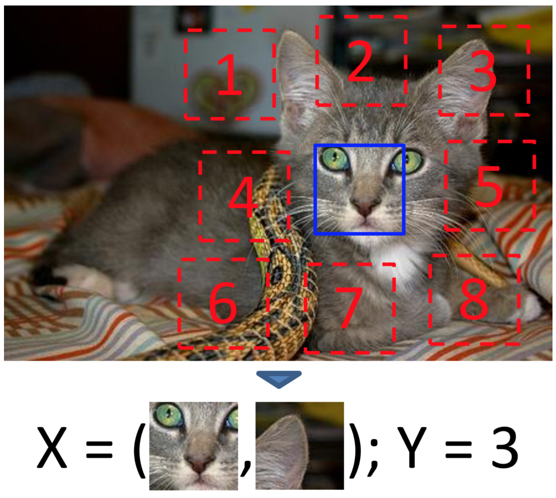


Figure 18: *Input to the self-supervised model. Figure from [36]*

(flattened) before being processed by the fully-connected layer. The vectorial representations produced for the two patches by the first fully connected layer are then concatenated, and processed by three more fully-connected layers. The last of these layers produces an 8-dimensional vector containing the predicted probabilities of the different relative positions.

When training the model, these probabilities are compared with the correct position (position 3 in the example of Figure 18), so that they agree as best as possible. This can be done using an ordinary supervised loss function, such as the cross-entropy loss function [1]:

$$\mathcal{L}_{ce} = -\sum_{i=1}^{n}\sum_{j=1}^{8}\left(y_{ij}\ln\frac{\hat{y}_{ij}}{y_{ij}} + (1-y_{ij})\ln\frac{1-\hat{y}_{ij}}{1-y_{ij}}\right)$$

where $n$ is the number of patch-pairs, $\hat{y}_{ij}$ is the predicted probability of patch-pair $i$ having relative position $j$, and

$$y_{ij} = \begin{cases} 1, & \text{True relative position of pair } i \text{ is } j \\ 0, & \text{otherwise} \end{cases}$$

is the one-hot encoded true relative position for patch-pair $i$.

### 9.5.2. From patches to full images

The model outlined above is designed to process patches extracted from a given image, rather than the full image. Thus, to obtain one representation for the whole image, one has to slightly modify the model. Recall that the convolution-operator does not require a fixed size input, meaning that the convolutional layers of the model can be left as-is. The fully-connected layers following the concatenation can be discarded, since we only have one image, and thus, nothing to concatenate. This leaves the first fully connected layer, which still requires an input with fixed dimensionality. This layer is transformed to a convolutional layer following [84]. The transformation removes the vectorization-step after the last convolutional layer, and replaces the first fully-connected layer with an equivalent convolutional layer, whose parameters are obtained from the weight matrix belonging to the fully-connected layer it replaces. After this modification, the model is capable of processing an input image of arbitrary size.

The transformed model can now be used as a feature extractor directly, or to initialize the parameters of another convolutional neural network designed for other image-processing tasks.

## 9.6. Training DNNs

### 9.6.1. Gradient descent

In the description of MLPs, CNNs, and RNNs, we saw examples of gradient computations used to determine the desired weight updates for the respective models. This is because DNNs are commonly trained using the backpropagation algorithm [67]. Suppose we have a loss function $\mathcal{L}(\boldsymbol{\theta}; \mathcal{X})$ which measures the performance of the model with parameters $\boldsymbol{\theta}$, evaluated on the dataset $\mathcal{X}$. The function $\mathcal{L}$ is assumed to be large for "bad" models, and small for "good" models, which means that we seek the parameter vector $\boldsymbol{\theta}$ which minimizes $\mathcal{L}$. Often, $\mathcal{L}$ is taken to be the negative log-likelihood of the sample: $\mathcal{L} = -\sum_{\boldsymbol{x} \in \mathcal{X}} \ln p_x(\boldsymbol{x})$, where the dataset is assumed to contain independent and identically distributed samples from the distribution $p_x$.

In gradient descent, the current parameter vector $\boldsymbol{\theta}_i$ is updated according to

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \lambda \nabla_{\boldsymbol{\theta}_i} \mathcal{L}(\boldsymbol{\theta}_i; \mathcal{X}) \tag{27}$$

which specifies an iterative procedure that can be repeated until some stopping criterion is met. In other words, at each iteration we update $\boldsymbol{\theta}$ by moving it along the opposite direction of the gradient evaluated at that point. Recall that the gradient always points in the direction of steepest ascent, meaning that the above update corresponds to moving $\boldsymbol{\theta}$ in the direction of steepest descent. The learning rate $\lambda$, is a hyperparameter which specifies the length of the step taken when updating the parameter vector. The selection of $\lambda$ is both model dependent and data dependent, which can make it difficult to determine a good value for this hyperparameter.

### 9.6.2. Stochastic mini-batch gradient descent

The algorithm outlined above updates the parameters based on the gradient computed on the full dataset $\mathcal{X}$. If the dataset is large, the computational cost of computing the gradient can be prohibitively large. Stochastic mini-batch gradient descent aims to alleviate the computational cost by randomly partitioning the dataset $\mathcal{X}$ into a set of mini-batches, $\mathcal{B}_1, \dots, \mathcal{B}_m$, and then update $\boldsymbol{\theta}$ based on the gradient computed on each mini-batch. The algorithm is summarized in Algorithm 7.

```
input: Raw dataset $\mathcal{X}$, loss function to minimize $\mathcal{L}$.
Initialize $\boldsymbol{\theta}$ randomly.
while Convergence criterion not met do
    Randomly partition $\mathcal{X}$ into $m$ mini-batches, $\mathcal{B}_1, \dots, \mathcal{B}_m$
    for $\mathcal{B}$ in $\mathcal{B}_1, \dots, \mathcal{B}_m$ do
        Compute the gradient: $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$
        Update the parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \boldsymbol{g}$
    end
end
```

**Algorithm 7:** Stochastic mini-batch gradient descent.

From a statistical viewpoint, stochastic mini-batch gradient descent offers a compromise between computational cost and variance minimization. The latter of which is a consequence of the averaging effect obtained when performing gradient descent on the full dataset [12, 85].

### 9.6.3. The Adam optimization technique

A natural consequence of deep learning models beginning to gain traction, was that the literature simultaneously saw an increase in publications about improving the basic gradient descent theme set by the aforementioned methods. Many of these works focused on the learning rate $\lambda$, and how it should be adapted to achieve faster convergence to a minimum of potentially lower value. Among these methods are AdaGrad [86], AdaDelta [87], and RMSprop [88]. The Adam [89] optimization technique also falls within this category, as it adapts the learning rate based on running estimators of the mean and (uncentered) variance of the gradient. These estimators are

$$\boldsymbol{m}_i = \beta_1 \boldsymbol{m}_{i-1} + (1 - \beta_1)\boldsymbol{g}_i$$
$$\boldsymbol{v}_i = \beta_2 \boldsymbol{v}_{i-1} + (1 - \beta_2)\boldsymbol{g}_i^2$$

where $\boldsymbol{g}_i = \nabla_{\boldsymbol{\theta}_i} \mathcal{L}(\boldsymbol{\theta}_i)$ is the gradient at iteration $i$, and $\boldsymbol{g}_i^2$ is its element-wise square. $\beta_1$ and $\beta_2$ are hyperparameters[6]. The update rule is:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \frac{\eta}{\sqrt{\tilde{\boldsymbol{v}}_i} + \varepsilon} \tilde{\boldsymbol{m}}_i \tag{28}$$

[6]The authors suggest $\beta_1 = 0.9$ and $\beta_2 = 0.999$ as default values.

where

$$\tilde{\boldsymbol{m}}_i = \frac{\boldsymbol{m}_i}{1-\beta_1^i} \qquad \text{and} \qquad \tilde{\boldsymbol{v}}_i = \frac{\boldsymbol{v}_i}{1-\beta_2^i}$$

are bias-corrected modifications of $\boldsymbol{m}_i$ and $\boldsymbol{v}_i$, respectively. $\eta$ is a learning rate, and $\varepsilon$ is a small constant which is included to avoid potential numerical instabilities. From Eq. (28) it is apparent that Adam differs from ordinary gradient descent in two main ways:

(i) The mean-gradient $\tilde{\boldsymbol{m}}_i$ is used instead of the current gradient $\boldsymbol{g}_i$;

(ii) The learning rate is adapted based on the uncentered variance of previous gradients $\lambda_i = \frac{\eta}{\sqrt{\tilde{\boldsymbol{v}}_i}+\varepsilon}$.

Property (i) introduces more of the previously mentioned averaging-effect, which can also help "push" the updating-scheme forwards at points where the current gradient is small [1]. A large variance implies that the algorithm is uncertain about which direction it should advance the gradient, and property (ii) therefore ensures that only small steps are taken in potentially uncertain directions [89].

### 9.6.4. Batch normalization

Batch normalization is a normalization scheme which can be applied to the layers of DNNs to improve performance and reduce the required number of training steps [90]. The normalization attempts to alleviate *Internal Covariate Shift* (ICS) which is defined as "*the change in the distribution of network activations due to the change in network parameters during training*" [90]. ICS is problematic as it requires a particular layer of the network to continuously adapt to a new distribution of outputs from the preceding layer.

Suppose we let $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$ be the vectorial outputs for the current batch at the layer to be normalized, at the current training iteration. Batch normalization then normalizes these outputs according to:

$$\tilde{\boldsymbol{y}}_i = \boldsymbol{\gamma} \odot (\boldsymbol{y}_i - \boldsymbol{\mu}) \oslash \boldsymbol{\sigma} + \boldsymbol{\beta} \tag{29}$$

where $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable parameters, and $\odot$ and $\oslash$ denote element-wise multiplication and element-wise division between vectors, respectively. $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the empirical mean and standard deviations of the output components, for the given batch. These are computed as:

$$\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{y}_i, \quad \boldsymbol{\sigma} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (\boldsymbol{y}_i - \boldsymbol{\mu}_i) \odot (\boldsymbol{y}_i - \boldsymbol{\mu}_i)}$$

where the square root is taken element-wise. To understand why this particular transformation reduces ICS it is useful to view it as the swo-stage transformation:

$$\tilde{\boldsymbol{y}}_i = \boldsymbol{\gamma} \odot \boldsymbol{y}_i^* + \boldsymbol{\beta}, \quad \boldsymbol{y}_i^* = (\boldsymbol{y}_i - \boldsymbol{\mu}) \oslash \boldsymbol{\sigma}$$

where the elements of $\boldsymbol{y}_i^*$ has empirical mean and variance equal to 0 and 1, respectively.

The orignal authors of batch normalization, Ioffe and Szegedy [90], claim that this standardization of moments constrains the distribution of the layer's outputs, thus reducing the ICS. $\tilde{\boldsymbol{y}}_i$ is then simply a linear transformation of $\boldsymbol{y}_i^*$ with parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, which is included to restore the representational power of the network after normalization.

However, in more recent work by Santurkar et al. [91], they show that the observed optimization improvements come from a smoother loss landscape, and not from stabilized distributions of activations. The smoothing of the loss landscape make the gradients more predictive and well behaved, allowing for more efficient optimization [91].

## Batch normalization in CNNs

The transformation in Eq. (29) assumed that the outputs from the given layer were vectors. However, this is not the case with convolutional neural networks, where each output is represented as a 3-dimensional array of numbers, which means that the transformation has to be modified to accommodate these data types. This is done by slicing each array along the third dimension, forming a set of vectors from each array:

$$\boldsymbol{Y}_i \mapsto \{\boldsymbol{y}_{i,1}, \dots, \boldsymbol{y}_{i,m}\}$$

where $\boldsymbol{Y}_i \in \mathbb{R}^{D_1 \times D_2 \times D_3}$ is the $i$-th output of the layer. The slicing process produces $m = D_1 \cdot D_2$ vectors which have dimension $D_3$. See Figure 19 for an illustration.

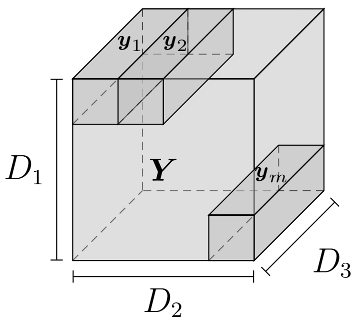


Figure 19: *Slicing an array $\boldsymbol{Y}$ along the third dimension to form a set of vectors $\boldsymbol{y}_1, \dots \boldsymbol{y}_m \in \mathbb{R}^{D_3}$, where $m = D_1 \cdot D_2$ .*

The transformation in Eq. (29) can then be performed on the set of vectors obtained by slicing all the arrays in the given batch. This causes the the normalization to have an effective batch size of $n \cdot m = n \cdot D_1 \cdot D_2$, as this is the total number of vectors obtained by slicing all $n$ arrays.

### Inference with batch normalized networks

From Eq. (29) it is apparent that introducing batch normalization after one or more layers in a neural network causes the output for a given input to depend on the rest of the elements in the current batch. Although this is fine during training, it is not a desired effect when the network is used for inference, as one typically wants a fixed correspondence between input and output. This is solved by keeping running estimates of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ during training, such that these reflect the moments of the layer outputs. When the network is used for inference, these estimates can then be used in place of the empirical moments, which will result in a fixed input-output relationship.

# 10. Deep clustering algorithms

The architectures described in the preceding section have proven to potentially yield large performance improvements in supervised classification tasks. Examples include AlexNet by Krizhevsky et al. [13] for large-scale image classification, and the Deep LSTM by Graves et al. [14] for speech recognition. However, these models are trained in a supervised fashion, as is the case with the majority of deep learning based success stories. Supervised training requires the data to be labeled before it can be used to train the model, which can be a prohibitive process for many applications.

This section will serve as an overview of related work, and introduce some of the impactful recent advances in deep clustering. The methods covered include Deep Embedded Clustering [15], SpectralNet [18], and Deep Divergence-based Clustering [20]. A few more algorithms are also covered in less detail, in order to provide a broad overview of the deep clustering field.

The design pattern for these models is to take one of the classical deep learning architectures described previously, and combine it with a *clustering module*, which computes the cluster membership vectors based on the representation provided by the deep neural network. The network and the clustering module are then trained simultaneously by minimizing an unsupervised loss function, to determine their respective parameters. The joint optimization causes the network to learn features that are well suited for the clustering module, while the clustering module learns to cluster these features in an optimal way.

Throughout this section, $f_{\boldsymbol{\theta}} : X \to Z$ will denote the neural network mapping with parameters $\boldsymbol{\theta}$, which maps from the input space $X$ to the space of learned representations $Z$. The clustering module will be denoted $g_{\boldsymbol{\phi}} : Z \to U$, where $\boldsymbol{\phi}$ is a vector of parameters, and $U$ is the space of cluster membership vectors. For an observation $\boldsymbol{X}_i$, we have the learned representation $\boldsymbol{z}_i = f_{\boldsymbol{\theta}}(\boldsymbol{X}_i)$, and cluster membership vector $\boldsymbol{u}_i = g_{\boldsymbol{\phi}}(\boldsymbol{z}_i) = (g_{\boldsymbol{\phi}} \circ f_{\boldsymbol{\theta}})(\boldsymbol{X}_i)$.

## 10.1. Deep Embedded Clustering

In Deep Embedded Clustering (DEC) [15], the feature extractor is parameterized by a multilayer perceptron, and the clustering module is based on soft-assignments to a set of centroids. More specifically, suppose we have $k$ centroids $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k \in Z$ associated with $k$ respective clusters. The soft cluster assignment of observation $i$ to cluster $j$ is then computed using Student's $t$-distribution:

$$u_{ij} = \frac{\left(1 + \frac{||\boldsymbol{z}_i - \boldsymbol{\mu}_j||^2}{\alpha}\right)^{-\frac{\alpha+1}{2}}}{\sum_{l=1}^{k} \left(1 + \frac{||\boldsymbol{z}_i - \boldsymbol{\mu}_l||^2}{\alpha}\right)^{-\frac{\alpha+1}{2}}} \tag{30}$$

where $\alpha$ is a hyperparameter specifying the distribution's degrees of freedom[7].

The loss function is constructed by forcing the distribution of cluster assignments $\mathbb{U}$ closer to a target distribution $\mathbb{P}$, by means of the Kullback-Leibler (KL) divergence:

$$\mathcal{L} = \mathrm{KL}(\mathbb{P}||\mathbb{U}) = \sum_{i=1}^{n} \sum_{j=1}^{k} p_{ij} \ln \frac{p_{ij}}{u_{ij}}. \tag{31}$$

The specification of target distribution is clearly a nontrivial choice, but the authors of DEC, Xie et al. [15], state that it should be chosen to have the following properties:

(i) Strengthen predictions.

(ii) Put more emphasis on high-confidence assignments.

(iii) Normalize loss contribution of each centroid.

They then proceed to suggest the distribution

$$p_{ij} = \frac{u_{ij}^2 / f_j}{\sum_{l=1}^{k} u_{il}^2 / f_l}$$

where $f_j = \sum_{i=1}^{n} u_{ij}$ is the frequency of the $j$-th cluster. Note that the division by $f_j$ helps $\mathbb{P}$ satisfy property (iii). However, to the author's best knowledge, no further proof on the fulfillment of conditions (i) or (ii) has been given. To this end, the following analysis is provided here: Let the difference between $p_{ij}$ and $u_{ij}$ be $\delta_{ij}$. Then we have $p_{ij} = u_{ij} + \delta_{ij}$, and:

$$u_{ij} + \delta_{ij} = \frac{u_{ij}^2 / f_j}{s^2}$$

[7]The authors of the DEC leave $\alpha = 1$ for all experiments.

where we let $s_i^2 = \sum_{l=1}^{k} u_{il}^2/f_l$ to unburden the notation. After some algebra, we obtain:

$$s_i^2 - \frac{u_{ij}}{f_j} = -\delta_{ij}\frac{s_i^2}{u_{ij}}.$$

Note that $s_i^2, u_{ij}$ and $f_j$ are all positive, implying that if $\frac{u_{ij}}{f_j} > s_i^2$ then $\delta_{ij} > 0$ and $p_{ij} > u_{ij}$. Thus, if $u_{ij}$ is large (high confidence assignment), the loss function will force it to become even larger by pulling it closer to $p_{ij}$, effectively strengthening the prediction. Conversely, if $\frac{u_{ij}}{f_j} < s_i^2$ then $p_{ij} < u_{ij}$, meaning that the contribution of this specific assignment will be weighted down in the loss function (by multiplication of $p_{ij}$). This leads to more emphasis being put on high-confidence assignments.

### 10.1.1. Training

The clustering parameters (centroids) $\boldsymbol{\phi} = [\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k]$, and network parameters $\boldsymbol{\theta}$, are estimated using stochastic mini-batch gradient descent. The gradient of the loss with respect to the $j$-th centroid is:

$$\nabla_{\boldsymbol{\mu}_j}\mathcal{L} = -\frac{\alpha+1}{\alpha}\sum_{i=1}^{n}\left(1+\frac{||\boldsymbol{z}_i-\boldsymbol{\mu}_j||^2}{\alpha}\right)^{-1}(p_{ij}-u_{ij})(\boldsymbol{z}_i-\boldsymbol{\mu}_j).$$

For the $i$-th embedded observation, the gradient is:

$$\nabla_{\boldsymbol{z}_i}\mathcal{L} = \frac{\alpha+1}{\alpha}\sum_{j=1}^{k}\left(1+\frac{||\boldsymbol{z}_i-\boldsymbol{\mu}_j||^2}{\alpha}\right)^{-1}(p_{ij}-u_{ij})(\boldsymbol{z}_i-\boldsymbol{\mu}_j)$$

which can be used to compute the weight updates in the MLP.

### 10.1.2. Initialization

DEC is not trained end-to-end using the aforementioned loss function. Rather, the parameters of the MLP are initialized by a stacked autoencoder (see Section 9.4), which is an architecture previously proven to provide representations suitable for clustering [80, 92]. When the autoencoder's optimization procedure terminates, the decoder is discarded, and the parameters of the encoder are taken to be the initial parameters for the mapping $f_{\boldsymbol{\theta}}$. The centroids $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ are initialized by running $k$-means on the hidden representations $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$, obtained by the autoencoder-initialized MLP. The MLP-parameters and centroids are then fine-tuned using the loss function in Eq. (31).

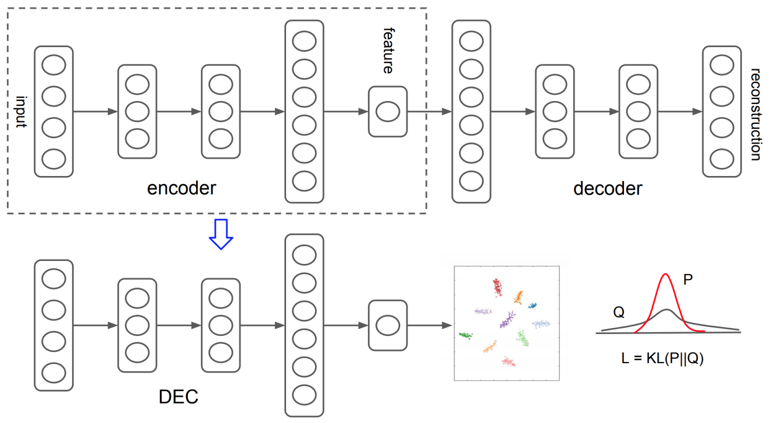


Figure 20: *An overview of the DEC architecture (from [15]). The feature extractor $f_{\boldsymbol{\theta}}$ is first trained as a stacked autoencoder, and then fine tuned using the clustering loss.*

## 10.2. SpectralNet

As the name implies, SpectralNet [18] offers a deep learning based approach to the spectral clustering algorithm described in Part II. Recall that for ordinary spectral clustering, we sought an output $\boldsymbol{y}$ for a given input $\boldsymbol{x}$, without explicitly considering the mapping $\boldsymbol{y} = \varphi(\boldsymbol{x})$. Instead, the solutions for a given dataset was formulated as an eigendecomposition of the graph Laplacian, formed from a set of input vectors $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$. This approach has a few notable drawbacks, the two most important being generalizability and scalability: The ordinary spectral clustering approach requires the full eigendecomposition to be repeated when clustering a previously unseen observation. Moreover, the computational complexity can make the eigendecomposition prohibitive for large datasets.

SpectralNet aims to alleviate these two shortcomings by modeling $\varphi$ explicitly using an approximation $f_{\boldsymbol{\theta}}$, which is parameterized by a neural network[8]. Suppose for now that we have a known symmetric affinity matrix $\boldsymbol{W} = [w_{ij}]$. The loss function used to train SpectralNet is then

$$\mathcal{L} = \frac{2}{n^2}\,\mathrm{tr}(\boldsymbol{Y}^T\boldsymbol{L}\boldsymbol{Y}) = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}||\boldsymbol{y}_i - \boldsymbol{y}_j||^2 \tag{32}$$

where $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n = f_{\boldsymbol{\theta}}(\boldsymbol{x}_1), \ldots, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ are the rows of $\boldsymbol{Y}$, and $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$ is the graph Laplacian, defined as in Section 8.4.2. This loss function is proportional to the loss function of ordinary spectral clustering, but it is minimized using gradient descent, rather than through an eigendecomposition.

To avoid the trivial solution of mapping all inputs to the same output, the minimization of Eq. (32) is performed subject to the constraint

$$\frac{1}{n^2}\boldsymbol{Y}^T\boldsymbol{Y} = \boldsymbol{I}_k$$

[8]The original authors does not explicitly state which neural network architecture they use for their experiments.

where $\boldsymbol{I}_k$ denotes the $k \times k$ identity matrix. This constraint is implemented as a linear orthogonalization layer, which orthogonalizes the columns of the matrix $\boldsymbol{Y}$. Thus, the complete SpectralNet pipeline is (see Figure 21):

$$\boldsymbol{y}_i = \boldsymbol{O}\tilde{\boldsymbol{y}}_i, \quad \tilde{\boldsymbol{y}}_i = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

where $\boldsymbol{O}$ denotes the matrix of the linear mapping performed by the orthogonalization layer, which can be obtained through the Cholesky decomposition of $\boldsymbol{Y}^T\boldsymbol{Y}$ [18].

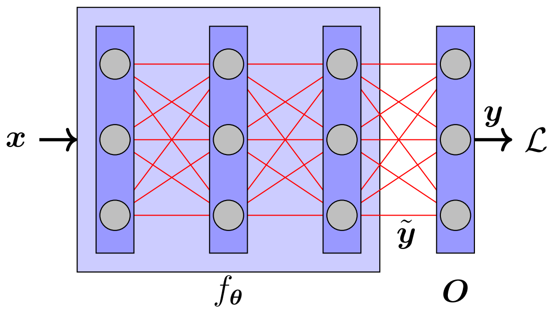


Figure 21: *An illustration of the SpectralNet architecture. The input $\boldsymbol{x}$ is processed by the neural network $f_{\boldsymbol{\theta}}$, producing $\tilde{\boldsymbol{y}}$. This representation is then orthogonalized using the linear transformation $\boldsymbol{O}$, to produce the output $\boldsymbol{y}$.*

### 10.2.1. Training

During training, the data is presented to the architecture in the form of randomly sampled mini-batches of size $n$. Due to the addition of the orthogonalization constraint, the optimization procedure alternates between the following two steps:

1. Sample a mini-batch, and compute the matrix $\boldsymbol{Y}$ based on current estimates of $\boldsymbol{O}$ and $\boldsymbol{\theta}$, and use it to update the orthogonalization matrix $\boldsymbol{O}$
2. Sample a new mini-batch, and recompute the matrix $\boldsymbol{Y}$. Compute the gradient of the loss function $\mathcal{L}$, to update the network parameters $\boldsymbol{\theta}$.

In the latter step, the gradient[9]

$$\nabla_{\tilde{\boldsymbol{y}}_a}\mathcal{L} = \frac{4}{n^2}\left(\sum_{i=1}^{n} w_{ai}(\boldsymbol{y}_a - \boldsymbol{y}_i)\right)^T \boldsymbol{O}$$

is propagated down to the neural network to update the parameters $\boldsymbol{\theta}$. When the network has been trained, the final $\boldsymbol{y}_1, \ldots \boldsymbol{y}_n$ are clustered using $k$-means.

[9]Not provided by SpectralNet's authors [18], but included here for completeness.

### 10.2.2. Computing the affinity matrix

As is the case with ordinary spectral clustering, the specification of an affinity matrix remains as a critical choice. The $\varepsilon$-connected Gaussian affinity

$$w_{ij} = \begin{cases} \exp\left(-\frac{d(\boldsymbol{x}_i, \boldsymbol{x}_j)}{2\sigma^2}\right), & d(\boldsymbol{x}_i, \boldsymbol{x}_j) < \varepsilon \\ 0, & \text{otherwise} \end{cases} \tag{33}$$

can most certainly be used, but it still requires the specification of a suitable distance function in the input space. In SpectralNet this distance function is learned from data, rather than being specified directly. This is accomplished by training an unsupervised *siamese* neural network [18, 93] to embed the input data in a vector space where the Euclidean distance can be used as a suitable distance function. An unsupervised siamese network is any network producing vectorial representations, trained to minimize the contrastive loss

$$\mathcal{L}_s = \sum_{i=1}^{n} \sum_{j=1}^{n} e_s(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

where

$$e_s(\boldsymbol{x}_i, \boldsymbol{x}_j) = \begin{cases} ||\boldsymbol{z}_i - \boldsymbol{z}_j||^2, & \boldsymbol{x}_j \text{ is } \boldsymbol{x}_i\text{'s nearest neighbor} \\ \max(0, 1 - ||\boldsymbol{z}_i - \boldsymbol{z}_j||)^2, & \text{otherwise} \end{cases} .$$

Here, $\boldsymbol{z}_i$ denotes the siamese embedding of $\boldsymbol{x}_i$. The distance function is then

$$d(\boldsymbol{x}_i, \boldsymbol{x}_j) = ||\boldsymbol{z}_i - \boldsymbol{z}_j||$$

which can be used to compute the affinity matrix $\boldsymbol{W}$ through Eq. (33). The contrastive loss function causes the siamese network to learn an approximate adaptive nearest neighbor metric, which tends to more accuratly describe similarity structure, than the Euclidean metric computed in the input space. This is demonstrated experimentally by Shaham et al. [18]. Note that the siamese network is constructed and trained prior to training of the neural network $f_{\boldsymbol{\theta}}$, and thus treated as fixed when the latter is being trained.

## 10.3. Deep Divergence-based Clustering

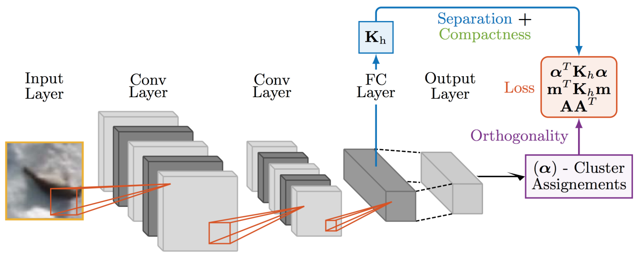

Separation +
Compactness
$\mathbf{K}_h$
Input Layer
Conv Layer
Conv Layer
FC Layer
Output Layer
Loss
$\boldsymbol{\alpha}^T\mathbf{K}_h\boldsymbol{\alpha}$
$\mathbf{m}^T\mathbf{K}_h\mathbf{m}$
$\mathbf{A}\mathbf{A}^T$
Orthogonality
($\boldsymbol{\alpha}$) - Cluster Assignements

Figure 22: *An overview of the DDC model for image clustering. Figure from [20]. Note that the notation is slightly different, as we have $\boldsymbol{\alpha} = \boldsymbol{u}$ and $\boldsymbol{h} = \boldsymbol{z}$.*

In contrast to the two aforementioned algorithms which were vector based, Deep Divergence-based Clustering (DDC) [20] is an approach to image clustering utilizing ideas from deep learning. In DDC, the mapping $f_{\boldsymbol{\theta}}$ is a convolutional neural network, which previously has been shown to perform well on supervised image processing tasks [25, 13]. The last layer of the CNN is a fully-connected layer which receives the flattened output from the last convolutional layer, and produces the representation $\boldsymbol{z}$. The clustering module $g_{\boldsymbol{\phi}}$ consists of another fully-connected layer, which produces the soft cluster membership vector (see Figure 22).

DDC's loss function is constructed based on both the learned representations $\boldsymbol{z}$, and the cluster membership vectors $\boldsymbol{u}$. It is designed to enforce the following requirements:

(i) *Cluster compactness and separability:* In the representation space $Z$, individual clusters should be compact, while different clusters should be well separated.

(ii) *Orthogonal cluster membership vectors:* Cluster membership vectors pointing to different clusters should be orthogonal in $\mathbb{R}^k$.

(iii) *Closeness to simplex corner:* Each cluster membership vector should be close to a corner of the standard simplex in $\mathbb{R}^k$ (defined as: $\{(a_1, \dots, a_k) \in \mathbb{R}^k_{\geq 0} : \sum_{i=1}^k a_i = 1\}$).

The loss function is the sum of three terms, each of which tackles one of the properties outlined above:

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3.$$

The first loss term enforces the separability and compactness condition through the Cauchy-Schwarz (CS) divergence between $k$ probability density functions $p_1, \dots, p_k$ [37]:

$$D_{cs}(p_1, \dots, p_k) = -\ln\left(\frac{1}{k}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\frac{\int p_i(\boldsymbol{z})p_j(\boldsymbol{z})\mathrm{d}\boldsymbol{z}}{\sqrt{\int p_i^2(\boldsymbol{z})\mathrm{d}\boldsymbol{z}\int p_j^2(\boldsymbol{z})\mathrm{d}\boldsymbol{z}}}\right).$$

Note that $\frac{1}{k}$ is used as a normalization constant, instead of $\binom{k}{2}^{-1}$, which was used by Jenssen et al. [37]. In order to describe DDC in its original form, we will stick to $\frac{1}{k}$ in this section.

Maximizing $D_{cs}$ corresponds to minimizing the argument of the logarithm, resulting in:

$$\mathcal{L}_1 = \frac{1}{k} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\int p_i(\boldsymbol{z}) p_j(\boldsymbol{z}) \mathrm{d}\boldsymbol{z}}{\sqrt{\int p_i^2(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} \int p_j^2(\boldsymbol{z}) \mathrm{d}\boldsymbol{z}}}.$$

Suppose that each of the probability density functions represent their own cluster. The numerator of term $(i, j)$ in $\mathcal{L}_1$ is the integrated overlap between between clusters $i$ and $j$. A small value of the integrated overlap leads to clusters that are well separated. The denominator is the product of integrated self-overlap for clusters $i$ and $j$. These quantities will be large if both clusters are compact. Hence, a combination of compact and well-separated clusters will result in $\mathcal{L}_1$ taking a small value.

The probability density functions $p_1, \dots, p_k$ are unknown, and have to be estimated from data. Using the kernel density estimator [38] we get[10]:

$$p_j(\boldsymbol{z}) = \frac{1}{|\mathcal{C}_j| \sigma^{\dim Z}} \sum_{\boldsymbol{z}_j \in \mathcal{C}_j} K\left( \frac{||\boldsymbol{z} - \boldsymbol{z}_j||}{\sigma} \right)$$

where $K$ is chosen to be a Gaussian

$$K(x) = \frac{1}{\sqrt{2\pi}} \exp\left( -\frac{x^2}{2} \right)$$

and $\sigma$ is a bandwidth parameter. If we assume for now, that the cluster membership functions produce hard assignments, we can rewrite $\mathcal{L}_1$ as:

$$\mathcal{L}_1 = \frac{1}{k} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_j}{\sqrt{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j}}$$

where $\boldsymbol{K} = [\kappa_{ij}]$ is the kernel matrix whose elements are the pairwise similarities $\kappa_{ij} = K\left( \frac{||\boldsymbol{z}_i - \boldsymbol{z}_j||}{\sigma} \right)$. $\boldsymbol{v}_j$ denotes the $j$-th *column* of the $n \times k$ cluster assignment matrix $\boldsymbol{U}$, which can be formed row-wise from the cluster assignment vectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_n$. To make the loss function differentiable, we can now relax the hard membership constraint, and allow for soft assignments instead.

The second loss term enforces the orthogonality between the cluster assignments vectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_n$. The matrix $\boldsymbol{U}\boldsymbol{U}^T$ consist of pairwise inner products between cluster assignment vectors, and thus, small elements in the upper (or lower)

[10]Note the small abuse of notation where $p_j$ denotes both the true pdf and the kernel density estimate.

triangular part of this matrix would correspond to orthogonal cluster assignment vectors. This gives the loss term

$$\mathcal{L}_2 = \operatorname{triu}(\boldsymbol{U}\boldsymbol{U}^T) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \boldsymbol{u}_i \boldsymbol{u}_j$$

which is the sum of the strictly upper triangular part of $\boldsymbol{U}\boldsymbol{U}^T$. However, this sum also enforces orthogonality between vectors pointing to the same cluster, and thus introduces a regularizing effect to the optimization by repelling the cluster assignment vectors away from each other.

The last loss term ensures that the cluster assignment vectors lie close to a corner of the simplex containing the assignments $\boldsymbol{u}_1, \dots, \boldsymbol{u}_n$. Let $\boldsymbol{M}$ be the matrix whose elements are:

$$m_{ij} = \exp(-||\boldsymbol{u}_i - \boldsymbol{e}_j||^2)$$

where $\boldsymbol{e}_j$ denotes the $j$-th corner of the simplex ($j$-th cartesian basis vector). Then we have the loss term:

$$\mathcal{L}_3 = \frac{1}{k} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_j}{\sqrt{\boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_i \boldsymbol{m}_j^T \boldsymbol{K} \boldsymbol{m}_j}}$$

where $\boldsymbol{m}_j$ denotes the $j$-th column of $\boldsymbol{M}$. Due to its resemblance to $\mathcal{L}_1$, $\mathcal{L}_3$ can be interpreted analogously: The distribution of cluster assignment vectors should be compactly centered around separate simplex corners.

## 10.3.1. Training

In contrast to DEC, DDC can be trained end-to-end from randomly initialized parameters. Training is done by minimizing the loss function $\mathcal{L}$ using mini-batch stochastic gradient descent. However, the gradient computations were not provided by Kampffmeyer et al. [20]. For the sake of completeness, they are provided here. First of all, notice that we have:

$$\nabla \mathcal{L} = \nabla \mathcal{L}_1 + \nabla \mathcal{L}_2 + \nabla \mathcal{L}_3.$$

Starting with the derivative of $\mathcal{L}_1$ with respect to an output $u_{ab}$:

$$\frac{\partial \mathcal{L}_1}{\partial u_{ab}} = \frac{1}{k} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{D_1(i,j) \cdot \sqrt{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j} - D_2(i,j) \cdot \frac{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_j}{2\sqrt{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j}}}{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j}$$

where:

$$D_1(i,j) = \frac{\partial}{\partial u_{ab}} \boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_j = \delta(i,b) \sum_{m=1}^{n} k_{am} u_{mj} + \delta(j,b) \sum_{l=1}^{n} k_{la} u_{li}$$

and:

$$\begin{aligned} D_2(i,j) &= \frac{\partial}{\partial u_{ab}} \boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j \\ &= 2\left( \delta(i,b) \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j \sum_{m=1}^{n} k_{am} u_{mi} + \delta(j,b) \boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \sum_{m=1}^{n} k_{am} u_{mj} \right). \end{aligned}$$

The gradient of $\mathcal{L}_1$ wrt. an embedded point $\boldsymbol{z}_a$ is:

$$\nabla_{\boldsymbol{z}_a} \mathcal{L}_1 = \frac{1}{k} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{D_3(i,j) \cdot \sqrt{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j} - D_4(i,j) \cdot \frac{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_j}{2\sqrt{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j}}}{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j}$$

where:

$$\begin{aligned} D_3(i,j) &= \nabla_{\boldsymbol{z}_a} \boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_j \\ &= \sum_{m=1}^{n} u_{ai} u_{mj} k_{am} \left( \frac{\boldsymbol{z}_m - \boldsymbol{z}_a}{\sigma^2} \right) + \sum_{l=1}^{n} u_{li} u_{aj} k_{la} \left( \frac{\boldsymbol{z}_l - \boldsymbol{z}_a}{\sigma^2} \right) \end{aligned}$$

and:

$$D_4(i,j) = \nabla_{\boldsymbol{z}_a} (\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j) = D_3(i,i) \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j + \boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i D_3(j,j)$$

Moving on to $\mathcal{L}_2$ we have:

$$\nabla_{\boldsymbol{u}_a} \mathcal{L}_2 = \sum_{i=1}^{n-1} u_i + \sum_{j=a+1}^{n} u_j$$

and finally, for $\mathcal{L}_3$, we get:

$$\frac{\partial \mathcal{L}_3}{\partial u_{ab}} = \frac{1}{k} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{D_5(i,j) \cdot \sqrt{\boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_i \boldsymbol{m}_j^T \boldsymbol{K} \boldsymbol{m}_j} - D_6(i,j) \cdot \frac{\boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_j}{2\sqrt{\boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_i \boldsymbol{m}_j^T \boldsymbol{K} \boldsymbol{m}_j}}}{\boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_i \boldsymbol{m}_j^T \boldsymbol{K} \boldsymbol{m}_j}$$

where:

$$\begin{aligned} D_5(i,j) &= \frac{\partial}{\partial u_{ab}} \boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_j \\ &= 2 \sum_{t=1}^{n} k_{at} m_{tj} m_{ai} (e_{ib} - u_{ab}) + 2 \sum_{s=1}^{n} k_{sa} m_{si} m_{aj} (e_{jb} - u_{ab}) \end{aligned}$$

and:

$$D_6(i,j) = \boldsymbol{m}_i^T \boldsymbol{K} \boldsymbol{m}_i D_5(j,j) + \boldsymbol{m}_j^T \boldsymbol{K} \boldsymbol{m}_j D_5(i,i).$$

To summarize, any derivative with respect to component(s) of $\boldsymbol{U}$ are used to update the parameters of the output layer $\boldsymbol{\phi}$, through the MLP-gradients provided in the previous part. The parameters the CNN are updated using $\nabla_{\boldsymbol{z}} \mathcal{L}_1$, as well as the gradients that are propagated down through the output layer. The updating is done according to the CNN gradients which were also provided in the previous part.

## 10.4. Other algorithms

The purpose of this subsection is to introduce a few more deep clustering algorithms, in order to further increase our insight into the deep clustering field. These particular algorithms were chosen since they fall within the "DNN + clustering module" framework, which has been common for the previously discussed algorithms as well. As we will see later, this shared framework means that all these algorithms can be modified using the unsupervised companion objectives proposed in this thesis – to potentially improve the clustering performance. The algorithms are:

- **Improved Deep Embedded Clustering (IDEC)** [24]. As the name implies, IDEC is closely related to the previously described DEC algorithm. However, the authors of IDEC argue that the fine-tuning stage of DEC "*leads to non-representative meaningless features and this in turn hurts clustering performance*" [24]. To alleviate this IDEC keeps the decoder-part of the autoencoder during fine-tuning, in contrast to DEC, where it is discarded. The other parts of the IDEC model are shared with DEC.
- **Deep Clustering Network (DCN)** [17]. This method is also similar to IDEC, since it uses an autoencoder with a clustering module attached to the code-space. However, instead of the soft clustering module used by IDEC, DCN uses a hard $k$-means clustering module to produce the cluster assignments. Due to the non-differentiability of the hard cluster assignments, DCN adopts the following three-stage optimization procedure: (i) Update autoencoder parameters with gradient descent on reconstruction loss; (ii) recompute cluster assignments; (iii) recompute cluster centers. This optimization procedure is repeated until convergence.
- **Discriminatively Boosted image Clustering (DBC)** [21]. DBC is another autoencoder-based algorithm which is designed specifically for image clustering. It uses a fully-convolutional autoencoder, which is an autoencoder consisting of only convolutional layers [81, 94]. DBC's training procedure begins with pre-training the autoencoder, and when the pre-training finishes, the decoder is discarded, and the encoder is fine-tuned using DEC's clustering loss.

Lastly, the deep clustering literature also contains a few algorithms that does not directly adhere to the "DNN + clustering module" framework. These algorithms include: (i) Joint Unsupervised LEarning (JULE) [19], which uses a CNN for hierarchical clustering; (ii) Information Maximizing Self-Augmented Training (IMSAT) [95], which depends heavily on data augmentation; (iii) Variational Deep Embedding (VaDE) [96], which is a generative model based on variational autoencoders [97]; and (iv) Categorical Generative Adversarial Network (CatGAN) [16], which is another generative model based on generative adversarial networks [98]. However, a thorough review and comparison of these methods falls outside the scope of this thesis.

# Part IV / Proposed method

In this part we will go through the mathematical foundations for the model proposed in this thesis. The methodology includes ideas and theory from several different areas of research, including nonparametric statistics, information theory, kernel methods and reproducing kernel Hilbert spaces, differential geometry and tensor analysis, and finally deep learning.

The first section in this part considers the problem of estimating an unknown probability density function from a vectorial dataset, without any explicit assumptions on the distribution itself. These concepts are generalized to tensors of arbitrary rank in the second section by building upon the work done by Signoretto [35]. Then, in the last section of this part we will see how all these concepts come together to form the main contribution of this thesis, whose goal is to address some of the challenges that arise with current state-of-the-art methods for deep clustering.

## 11. Kernel density estimation

Kernel Density Estimation (KDE) [38, 99] is one of the most widely used methods for estimating unknown probability distributions. In contrast to many other estimation techniques, KDE is *nonparametric*, meaning that there are no direct assumptions on the underlying distribution of the data. Kernel density estimates have been used to describe methods previously in this thesis, but since it represents an integral part of the proposed method, it is explained in more detail in this section.

Suppose we have a dataset consisting of $n$ vector-valued random variables $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n \in \mathbb{R}^D$ drawn from an unknown distribution $p_{\boldsymbol{X}}(\boldsymbol{x})$. The KDE is a generalization of the well-known histogram, and is formulated as follows:

$$\hat{p}_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{n\sigma^D} \sum_{i=1}^{n} K\left(\frac{||\boldsymbol{x} - \boldsymbol{X}_i||}{\sigma}\right)$$

where $||\cdot||$ denotes the Euclidean norm on $\mathbb{R}^D$, $\sigma$ is a bandwidth parameter, and $K$ is a distribution or *kernel* on $\mathbb{R}$ satisfying

$$\text{(i)} \quad 0 \leq K(x) < \infty \ \forall x \in \mathbb{R}$$

$$\text{(ii)} \quad K(x) = K(-x) \ \forall x \in \mathbb{R}$$

$$\text{(iii)} \quad \int_{-\infty}^{\infty} K(x) \mathrm{d}x = 1.$$

Furthermore, one can constrain $\int_{-\infty}^{\infty} x^2 K(x)\mathrm{d}x = 1$, causing the bandwidth parameter $\sigma$ to represent the scale-parameter of the kernel. Perhaps the most commonly used kernel in the literature is the Gaussian kernel:

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}. \tag{34}$$

In essence, KDE with a Gaussian kernel places a $D$-dimensional spherical Gaussian distribution around each observation, and then averages the contributions from each of the observations to create the final estimate. Setting the bandwidth parameter $\sigma$ is a highly nontrivial task, and several automated procedures have been proposed to alleviate this burden. Jenssen et al. [37] suggest determining $\sigma$ by minimizing the asymptotic mean integrated squared error (AMISE) between $\hat{p}_{\boldsymbol{X}}$ and $p_{\boldsymbol{X}}$ resulting in

$$\sigma_{\text{AMISE}} = \hat{\sigma} \left( \frac{4}{(2D+1)n} \right)^{\frac{1}{D+4}}$$

where

$$\hat{\sigma} = \frac{1}{nd-1} \sum_{j=1}^{D} \sum_{i=1}^{n} \left( X_{ij} - \bar{X}_i \right)^2$$

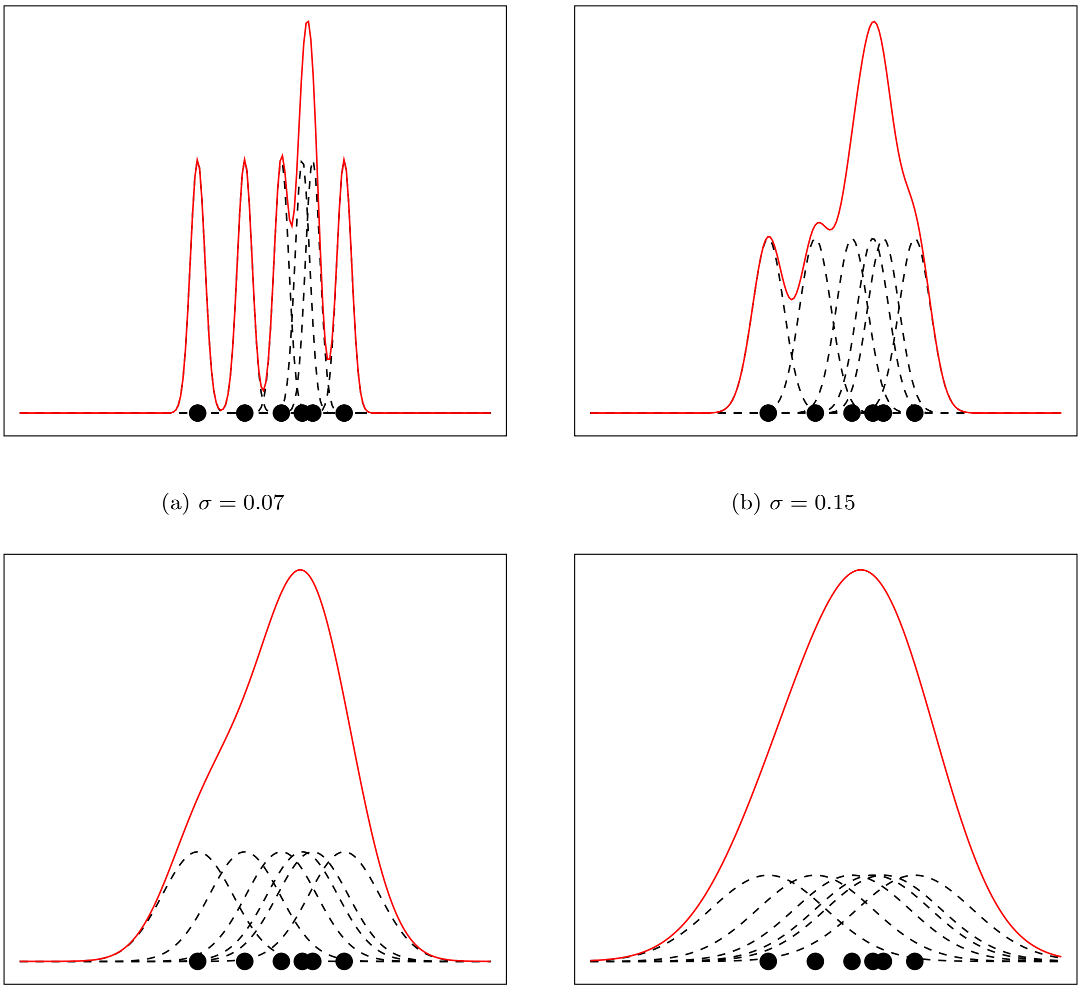
is the averaged sample standard deviation along each dimension.

Another rule of thumb which has been used in previous information-theoretic clustering algorithms [100, 20], is to let $\sigma$ be 15 % of the median pairwise distances between the observations in the dataset:

$$\sigma = 0.15 \cdot \text{median} \left\{ d_{ij} \right\}_{i,j=1}^{n}$$

where $d_{ij} = ||\boldsymbol{X}_i - \boldsymbol{X}_j||$.

Figure 23 shows examples of kernel density estimates using different values for the bandwidth $\sigma$. From the plots, it is apparent that a smaller value of $\sigma$ causes the estimates to exhibit larger variations, as more of the mass in each kernel lies closer to its respective observation.



(a) $\sigma = 0.07$

(b) $\sigma = 0.15$

(c) $\sigma = \sigma_{AMISE} = 0.338$

(d) $\sigma = 0.55$

Figure 23: *Univariate kernel density estimate (red) for* 5 *observations (●). The dashed curves indicate individual contributions from the respective observations to the total estimate. The kernels are Gaussians with varying scales. Note the large difference in the smoothness of the different estimates.*

## 11.1. Kernel density estimation and Mercer kernels

As it turns out, there is a connection between kernel density estimation, and the theory of Mercer kernels and reproducing kernel Hilbert spaces. The link was first outlined by Jenssen et al. [37], and is based on a specific information theoretic quantity referred to as the Cauchy-Schwarz (CS) divergence. However, before delving into the exact formulation, we will first outline the relevant background theory on Hilbert spaces and Mercer kernels.

### 11.1.1. Mercer kernels and reproducing kernel Hilbert spaces

Informally, a Hilbert space $\mathcal{H}$ is a possibly infinite dimensional linear space which possesses an inner-product operation[11]. When the dimensionality of $\mathcal{H}$ is finite, it reduces to an Euclidean space. Let $\boldsymbol{\phi} : \mathbb{R}^D \to \mathcal{H}$ be a mapping from the $D$-dimensional input space to a Hilbert space $\mathcal{H}$, and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D$. Then, according to *Mercer's Theorem* [1], there exists a function $k(\boldsymbol{x}, \boldsymbol{y})$ such that

$$\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{y}) \rangle_{\mathcal{H}} = k(\boldsymbol{x}, \boldsymbol{y})$$

where $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ denotes the inner procuct on $\mathcal{H}$. Furthermore, $k$ is a symmetric continuous function satisfying

$$\int_C \int_C k(\boldsymbol{x}, \boldsymbol{y}) g(\boldsymbol{x}) g(\boldsymbol{y}) \mathrm{d}\boldsymbol{x} \mathrm{d}\boldsymbol{y} \geq 0 \tag{35}$$

for any $g(\boldsymbol{x}), \boldsymbol{x} \in C \subset \mathbb{R}^D$ such that

$$\int_C g(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x} < \infty.$$

In this case, $k$ is referred to as a *Mercer Kernel*, and $\mathcal{H}$ is called a *Reproducing Kernel Hilbert Space* (RKHS). Interestingly, the above also holds in reverse, meaning that any symmetric continuous function satisfying Eq. (35) specifies an inner product in some RKHS.

It is important to note that Mercer's theorem does not provide any guidance on how to find the mapping $\boldsymbol{\phi}$ explicitly. However, many machine learning algorithms can be formulated through inner products, meaning that one can map the data from $\mathbb{R}^D$ to $\mathcal{H}$ implicitly, through the kernel function $k$. This has led to the "kernelization"[12] of classical linear algorithms, resulting in e.g. Kernel $k$-means, Kernel Principal Component Analysis [103], and the Kernel Support Vector Machine [104]. Although linear in $\mathcal{H}$, these techniques become nonlinear

[11]A formal definition can be found in e.g. [101] or [102]

[12]The translation is also sometimes referred to as the "Kernel Trick".

in $\mathbb{R}^D$ due to the potential nonlinearity of the mapping $\boldsymbol{\phi}$. It can be shown that the Gaussian kernel

$$k_\sigma(\boldsymbol{x}, \boldsymbol{y}) = e^{-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2\sigma^2}} \tag{36}$$

satisfies the kernel-conditions (see e.g. [105]), making it a popular choice in these methods as well. Note that we have omitted the normalization factor since Mercer's theorem does not require the total integral of the kernel to be equal to one.

### 11.1.2. The Cauchy-Schwarz divergence in a Hilbert space

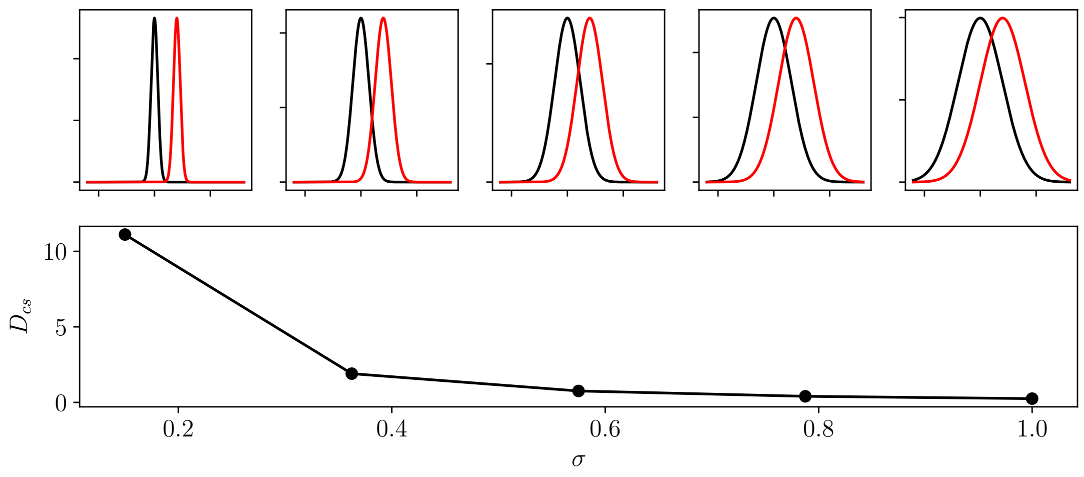


Figure 24: *The CS divergence between two probability density functions at different widths and overlaps.*

The CS divergence is an approach to measuring the similarity between probability density functions. For a pair of densities $p_1$ and $p_2$, the CS divergence is given by

$$D_{cs}(p_1, p_2) = -\ln\left(\frac{\int p_1(\boldsymbol{x})p_2(\boldsymbol{x})\mathrm{d}\boldsymbol{x}}{\sqrt{\int p_1(\boldsymbol{x})^2\mathrm{d}\boldsymbol{x} \int p_2(\boldsymbol{x})^2\mathrm{d}\boldsymbol{x}}}\right).$$

As was previously argued in the explanation of DDC, densities that are compact and well separated will result in a large CS divergence.

Figure 24 illustrates how the CS divergence changes for a pair of Gaussians when $\sigma$ increases. The plots show that an increasing $\sigma$ leads to densities that are overlapping and less compact – which is reflected by the decrease in $D_{cs}$.

For covenience, denote the argument of the logarithm $d_{cs}(p_1, p_2)$, such that $D_{cs}(p_1, p_2) = -\ln(d_{cs}(p_1, p_2))$. Now suppose the functions $p_1$ and $p_2$ are unknown, and we are interested in estimating the CS divergence between them from the random samples $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ and $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n$ drawn from $p_1$ and $p_2$, respectively. Using the kernel density estimator, the numerator in $d_{cs}(\hat{p}_1, \hat{p}_2)$ becomes

$$\int \hat{p}_1(\boldsymbol{x})\hat{p}_2(\boldsymbol{x})\mathrm{d}x = \frac{1}{(n\sigma)^{2d}} \sum_{i,j=1}^{n} \int K\left(\frac{||\boldsymbol{x} - \boldsymbol{X}_i||}{\sigma}\right) K\left(\frac{||\boldsymbol{x} - \boldsymbol{Y}_j||}{\sigma}\right) \mathrm{d}\boldsymbol{x}.$$

If $K$ is defined as in Eq. (34), we get

$$\int \hat{p}_1(\boldsymbol{x})\hat{p}_2(\boldsymbol{x})\mathrm{d}x = \frac{1}{(n\sigma)^{2d}} \sum_{i,j=1}^{n} K\left(\frac{||\boldsymbol{X}_i - \boldsymbol{Y}_j||}{\sqrt{2}\sigma}\right)$$

by the convolution theorem for Gaussians [37]. Similar calculations for the denominator in $d_{cs}(\hat{p}_1, \hat{p}_2)$ give

$$d_{cs}(\hat{p}_1, \hat{p}_2) = \frac{\sum\limits_{i,j=1}^{n} k_{\sqrt{2}\sigma}(\boldsymbol{X}_i, \boldsymbol{Y}_j)}{\sqrt{\sum\limits_{i,j=1}^{n} k_{\sqrt{2}\sigma}(\boldsymbol{X}_i, \boldsymbol{X}_j) \sum\limits_{i,j=1}^{n} k_{\sqrt{2}\sigma}(\boldsymbol{Y}_i, \boldsymbol{Y}_j)}}$$

where we've used the Gaussian Mercer kernel $k_{\sqrt{2}\sigma}(\cdot,\cdot)$ from Eq. (36). Since $k_{\sqrt{2}\sigma}(\cdot,\cdot)$ represents an inner product in a RKHS, we can write

$$\begin{aligned} d_{cs}(\hat{p}_1, \hat{p}_2) &= \frac{\sum\limits_{i,j=1}^{n} \langle \boldsymbol{\phi}(\boldsymbol{X}_i)\boldsymbol{\phi}(\boldsymbol{Y}_j)\rangle_{\mathcal{H}}}{\sqrt{\sum\limits_{i,j=1}^{n} \langle \boldsymbol{\phi}(\boldsymbol{X}_i)\boldsymbol{\phi}(\boldsymbol{X}_j)\rangle_{\mathcal{H}} \sum\limits_{i,j=1}^{n} \langle \boldsymbol{\phi}(\boldsymbol{Y}_i)\boldsymbol{\phi}(\boldsymbol{Y}_j)\rangle_{\mathcal{H}}}} \\ &= \frac{\langle \boldsymbol{M}_1, \boldsymbol{M}_2\rangle_{\mathcal{H}}}{\langle \boldsymbol{M}_1, \boldsymbol{M}_1\rangle_{\mathcal{H}} \langle \boldsymbol{M}_2, \boldsymbol{M}_2\rangle_{\mathcal{H}}} \\ &= \cos\angle(\boldsymbol{M}_1, \boldsymbol{M}_2)_{\mathcal{H}} \end{aligned}$$

where $\boldsymbol{M}_1 = \sum\limits_{i=1}^{n} \boldsymbol{\phi}(\boldsymbol{X}_i)$ and $\boldsymbol{M}_2 = \sum\limits_{i=1}^{n} \boldsymbol{\phi}(\boldsymbol{Y}_i)$ and $\angle(\cdot,\cdot)_{\mathcal{H}}$ denotes the angle between two elements in $\mathcal{H}$. Thus, we can interpret $d_{cs}(\hat{p}_1, \hat{p}_2)$ as the cosine between the "kernelized" mean vectors from the respective samples.

# 12. Tensors and tensor kernels

In the preceding section we assumed that the input to the kernel density estimation procedure was vectorial. However, in the case of more complex data types such as time series, images, or videos, the vector-space assumption might not hold. Thus, a modification of current algorithms, or construction of entirely new algorithms, is necessary to correctly model these datatypes. Luckily, there exists within the realm of mathematics, a framework on which we can base these modifications or new algorithms. These objects are referred to as *tensors*, and can be regarded as generalizations of the well known vectors. The coordinate-free formal definition of a tensor is not immediately useful to us, so we will throughout this thesis assume that we have a basis, and that all tensors are expressed through this basis. Informally, this allows us to view a tensor as a "hypercube of numbers". Hence, we have the following definition:

**Definition 6.** A *tensor* $\boldsymbol{T}$ of rank $r$ is an $r$-dimensional collection of elements:

$$\boldsymbol{T} = [T_{i_1,i_2,\ldots,i_r}]$$

where $i_k \in \{1,\ldots D_k\}$ for $k = 1,\ldots,r$. The tuple $(D_1,\ldots,D_r)$ is referred to as the *shape* of the tensor.

From the definition above, we immediately have three well known examples of tensors:

1. *Vectors*: A vector $\boldsymbol{v} = [v_i]$ is a rank-1 tensor.
2. *Matrices*: A matrix $\boldsymbol{M} = [m_{ij}]$ is a rank-2 tensor.
3. *Color images*: A color image $\boldsymbol{I} = [i_{jkl}]$ is a rank-3 tensor.

Similarly to vectors and vector spaces, tensors reside in tensor spaces, where element-wise addition and scalar multiplication are defined analogously. In the following, we will add additional structure to the tensor space, in the form of distances and kernels.

## 12.1. Tensor kernels

### 12.1.1. The naïve kernel

Recall that for rank-1 tensors (vectors) we have the kernel:

$$k_\sigma(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2\sigma^2}\right) = \prod_{i=1}^{D_1} \exp\left(-\frac{(x_i - y_i)^2}{2\sigma^2}\right).$$

The latter equality shows that this kernel belongs to a particular class of kernels, namely *product kernels*. Generalizing this kernel to tensors $\boldsymbol{X}$ and $\boldsymbol{Y}$ of rank $r$ gives the naïve kernel [35]:

$$k_\sigma^{\text{naïve}}(\boldsymbol{X}, \boldsymbol{Y}) = \prod_{i_1=1}^{D_1} \cdots \prod_{i_r=1}^{D_r} \exp\left(-\frac{(X_{i_1\cdots i_r} - Y_{i_1\cdots i_r})^2}{2\sigma^2}\right)$$

where we compute a kernel for each of the components, and then use these to form a product kernel. However, as is also pointed out by Signoretto [35], this causes the kernel function to ignore inter-component structure within and between the respective tensors. Mathematically, this means that the kernel is invariant to a fixed permutation rule $P$:

$$k_\sigma^{\text{naïve}}(\boldsymbol{X}, \boldsymbol{Y}) = k_\sigma^{\text{naïve}}(P(\boldsymbol{X}), P(\boldsymbol{Y})).$$

This effect can be especially destructive for e.g. images, where these can be transformed beyond recognition by permuting the spatial indices.

### 12.1.2. Matricization-based tensor kernels

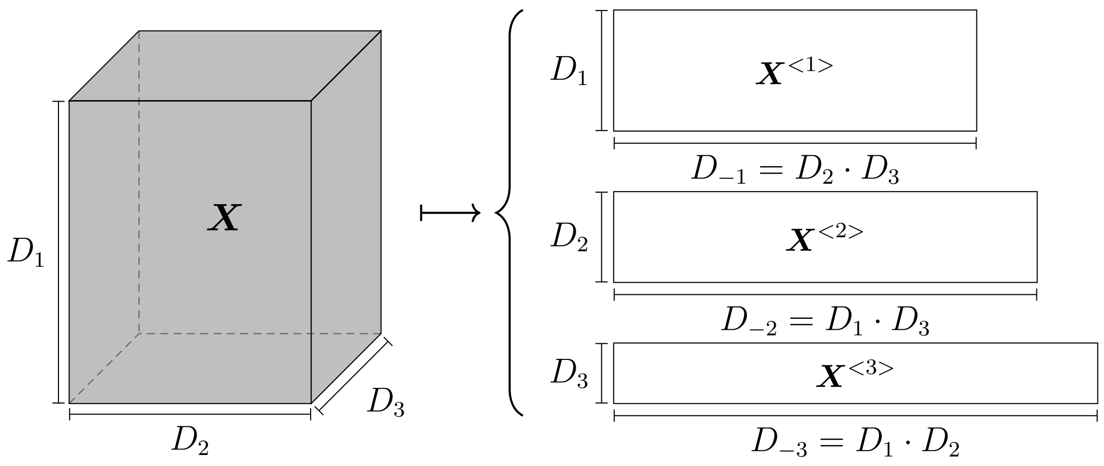


Figure 25: *Matricization of a rank-3 tensor $\boldsymbol{X}$.*

The problem outlined above calls for a more robust kernel which takes inter-component structure into account. To this end, Signoretto [35] suggests defining a product kernel over the *matricizations* of the input tensors:

$$k_\sigma^{\text{tensor}}(\boldsymbol{X}, \boldsymbol{Y}) = \prod_{m=1}^{r} k_\sigma^m(\boldsymbol{X}^{<m>}, \boldsymbol{Y}^{<m>})$$

where $\boldsymbol{X}^{<m>}$ is the matricization of $\boldsymbol{X}$ along dimension $m$ (and similarly for $\boldsymbol{Y}$). More specifically, this is the matrix obtained by rearranging the elements of $\boldsymbol{X}$ into a matrix with shape $(D_m, D_{-m})$, $D_{-m} = D_1 \cdots D_{m-1} \cdot D_{m+1} \cdots D_r$. The matricization $\boldsymbol{X}^{<1>}$ is therefore:

$$\boldsymbol{X}^{<1>} = \begin{bmatrix} X_{1,1,\dots,1} & \dots & X_{1,1,\dots,D_r} & \dots & X_{1,D_2,\dots,1} & \dots & X_{1,D_2,\dots,D_r} \\ X_{2,1,\dots,1} & \dots & X_{2,1,\dots,D_r} & \dots & X_{2,D_2,\dots,1} & \dots & X_{2,D_2,\dots,D_r} \\ & & & \vdots & & & \\ X_{D_1,1,\dots,1} & \dots & X_{D_1,1,\dots,D_r} & \dots & X_{D_1,D_2,\dots,1} & \dots & X_{D_1,D_2,\dots,D_r} \end{bmatrix}.$$

Note that the matricization for an arbitrary dimension $m$ can be obtained similarly by reordering the dimensions of $\boldsymbol{X}$, and then reorganizing the elements according to the rule given above.

Figure 25 shows an example of the matricization process where the rank-3 tensor $\boldsymbol{X}$ is transformed to the matrices $\boldsymbol{X}^{<1>}$, $\boldsymbol{X}^{<2>}$ and $\boldsymbol{X}^{<3>}$. In this case the tensor $\boldsymbol{X}$ could be an image, or the output of a convolutional layer in a CNN – the latter being the connection between tensors and CNNs mentioned in the introduction of this thesis.

Thus, it remains to specify the form of the components $k_\sigma^m(\cdot,\cdot)$, in the product kernel. We will stick to the Gaussian kernel, but use a distance function on the *Grassmann manifold* spanned by the respective matricizations (see Figure 26). Considering data matrices as points on the Grassmann manifold is the key concept in *Grassmannian learning* [106, 105, 107]. Results from this field indicate that using a distance function on the Grassmann manifold tends to improve performance of distance-based machine learning systems for tensor data.

If we couple the Gaussian kernel with a generic distance function $d_{\mathcal{G}(D_m, D_{-m})}$ on the Grassmann manifold, we get the kernel component:

$$k_\sigma^m(\boldsymbol{X}^{<m>}, \boldsymbol{Y}^{<m>}) = \exp\left(-\frac{d_{\mathcal{G}(D_m,D_{-m})}(\boldsymbol{X}^{<m>}, \boldsymbol{Y}^{<m>})}{2\sigma^2}\right).$$

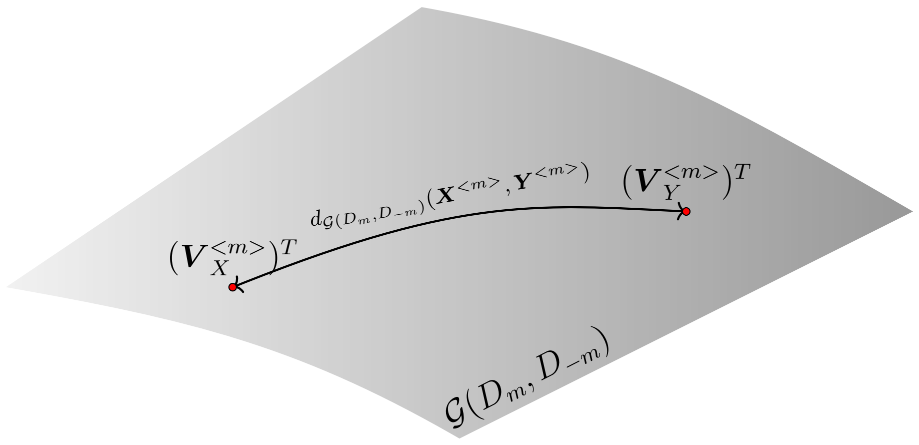
Here $\mathcal{G}(D_m, D_{-m})$ denotes the Grassmann manifold, which consists of all $D_m$-dimensional linear subspaces of $\mathbb{R}^{D_{-m}}$. This interpretation of the matricizations assumes that we have $D_m \leq D_{-m}$, as the dimensionality of the subspace has to be less than or equal to the dimensionality of the parent-space. This assumption does not necessarily hold for general tensors, which means that the computations have to take this into account to ensure the "validity" of the approach. Matricizations for which $D_m > D_{-m}$ are therefore transposed prior to the distance computation, following [35], meaning that we essentially consider distances on $\mathcal{G}(D_{-m}, D_m)$ instead. Throughout the rest of this thesis we will assume that $D_m < D_{-m}$, and that the transposition has been made whenever this does not hold.

Strictly speaking, it is not the matrix $\boldsymbol{X}^{<m>}$ itself that lies on the Grassmann manifold, but rather the span of its rows. Therefore, it is useful to represent points on the manifold by $D_m \times D_{-m}$ orthonormal matrices. These orthonormal representations can be obtained through the singular value decomposition (SVD) of the original input matrices. Recall that, for a real matrix $\boldsymbol{X}^{<m>}$ with shape $(D_m, D_{-m})$ satisfying $D_m < D_{-m}$, we have

$$\boldsymbol{X}^{<m>} = \boldsymbol{U}_X^{<m>} \boldsymbol{\Sigma}_X^{<m>} (\boldsymbol{V}_X^{<m>})^T \tag{37}$$

where $\boldsymbol{\Sigma}_X^{<m>}$ is a diagonal matrix with shape $(D_m, D_m)$ and nonnegative real numbers on the diagonal. $\boldsymbol{U}_X^{<m>}$ and $\boldsymbol{V}_X^{<m>}$ are orthonormal matrices with shapes $(D_m, D_m)$ and $(D_{-m}, D_m)$ respectively. Moreover, it can be shown that the row span of $(\boldsymbol{V}_X^{<m>})^T$ is equal to the row span of $\boldsymbol{X}^{<m>}$ [108]. Thus one can take $(\boldsymbol{V}_X^{<m>})^T$ to be the orthonormalized representation of $\boldsymbol{X}^{<m>}$. The distance function is illustrated in Figure 26 as the distance between $(\boldsymbol{V}_X^{<m>})^T$ and $(\boldsymbol{V}_Y^{<m>})^T$, which are points on the Grassmann manifold.

### 12.1.3. Distance functions on Grassmann manifolds



$d_{\mathcal{G}(D_m, D_{-m})}(\boldsymbol{X}^{<m>}, \boldsymbol{Y}^{<m>})$
$(\boldsymbol{V}_X^{<m>})^T$
$(\boldsymbol{V}_Y^{<m>})^T$
$\mathcal{G}(D_m, D_{-m})$

Figure 26: *The distance $d_{\mathcal{G}(D_m, D_{-m})}(\boldsymbol{X}^{<m>}, \boldsymbol{Y}^{<m>})$ between two matricizations $\boldsymbol{X}^{<m>}$ and $\boldsymbol{Y}^{<m>}$ on the Grassmann manifold. Since the Grassmann manifold consists of linear subspaces, the matricizations are represented by the respective orthonormalized matrices $(\boldsymbol{V}_X^{<m>})^T$ and $(\boldsymbol{V}_Y^{<m>})^T$.*

There exists a variety of different distance functions on Grassmann manifolds in the literature [106, 107]. Among the most commonly used distances, we find the geodesic (arc length) distance, and the projection distance. The former is computed using the principal angles between the respective subspaces [106]. The computation of these angles requires another SVD, which substantially increases the computational cost of the algorithm. Moreover, it can be shown that a Gaussian kernel using the geodesic distance is not positive semidefinite [105]. Conversely, the projection distance, when coupled with a Gaussian, results in a positive semidefinite kernel. This distance function is based on the *orthogonal projection operator* which takes an arbitrary element of $\mathbb{R}^{D_{-m}}$ and projects it orthogonally to the relevant subspace. For a matrix $\boldsymbol{X}^{<m>}$, the projection operator is given by the product $\boldsymbol{V}_X^{<m>}(\boldsymbol{V}_X^{<m>})^T$ [35]. Considering the Frobenius norm[13] between projection operators gives the distance function:

$$d^{\text{proj}}_{\mathcal{G}(D_m,D_{-m})}(\boldsymbol{X}^{<m>},\boldsymbol{Y}^{<m>}) = ||\boldsymbol{V}_X^{<m>}(\boldsymbol{V}_X^{<m>})^T - \boldsymbol{V}_Y^{<m>}(\boldsymbol{V}_Y^{<m>})^T||_F$$

where $||\cdot||_F$ denotes the Frobenius norm. Moreover, it can be shown that:

$$d^{\text{proj}}_{\mathcal{G}(D_m,D_{-m})}(\boldsymbol{X}^{<m>},\boldsymbol{Y}^{<m>}) = \sqrt{2(D_m - \operatorname{tr}((\boldsymbol{V}_Y^{<m>})^T\boldsymbol{V}_X^{<m>}(\boldsymbol{V}_X^{<m>})^T\boldsymbol{V}_Y^{<m>}))}$$

which is more efficient to compute compared to the previous expression [35].

Using the projection distance, we obtain the tensor kernel:

$$\begin{aligned} k_\sigma^{\text{tensor}}(\boldsymbol{X},\boldsymbol{Y}) &= \prod_{m=1}^{r} \exp\left(-\frac{D_m - \operatorname{tr}((\boldsymbol{V}_Y^{<m>})^T\boldsymbol{V}_X^{<m>}(\boldsymbol{V}_X^{<m>})^T\boldsymbol{V}_Y^{<m>})}{\sigma^2}\right) \\ &= \exp\left(-\frac{\sum_{m=1}^{r}(D_m - \operatorname{tr}((\boldsymbol{V}_Y^{<m>})^T\boldsymbol{V}_X^{<m>}(\boldsymbol{V}_X^{<m>})^T\boldsymbol{V}_Y^{<m>}))}{\sigma^2}\right) \\ &= \exp\left(-\frac{d^2_{\text{tensor}}(\boldsymbol{X},\boldsymbol{Y})}{2\sigma^2}\right) \end{aligned} \tag{38}$$

where we let $d_{\text{tensor}}$ be the tensor distance function constructed from the projection distance between the different matricizations for notational simplicity. It can be shown that, due to the positive semidefiniteness of the product kernels, this kernel is also positive semidefinite [105].

### 12.1.4. Computing $\boldsymbol{V}_X^{<m>}$

It can be readily seen that the distance function described above only depends on one of the three outputs from the SVD, namely $\boldsymbol{V}_X^{<m>}$. Moreover, it can be shown that $\boldsymbol{V}_X^{<m>}$ can be formed by stacking the $D_m$ eigenvectors of $\boldsymbol{X}^{<m>}(\boldsymbol{X}^{<m>})^T$ corresponding to the nonzero eigenvalues, column-wise [108]. We can therefore compute $\boldsymbol{V}_X^{<m>}$ directly with an eigendecomposition, instead of using a generic SVD solver.

However, this approach requires the eigendecomposition of matrices with shape $(D_{-m}, D_{-m})$ – which, in practical applications – can be quite large. To alleviate this potential computational burden, we will instead make use of the fact that we

[13]The Frobenius norm of a matrix is the square root of the sum of its squared elements.

can obtain eigenvectors of $\boldsymbol{X}^{<m>}(\boldsymbol{X}^{<m>})^T$ by linearly transforming the eigenvectors of $(\boldsymbol{X}^{<m>})^T\boldsymbol{X}^{<m>}$. Suppose that $\boldsymbol{v}$ is an eigenvector of $(\boldsymbol{X}^{<m>})^T\boldsymbol{X}^{<m>}$ with eigenvalue $\lambda \neq 0$. Then we have:

$$\begin{aligned}
&(\boldsymbol{X}^{<m>})^T\boldsymbol{X}^{<m>}\boldsymbol{v} = \lambda\boldsymbol{v} \\
\Rightarrow\ &\boldsymbol{X}^{<m>}(\boldsymbol{X}^{<m>})^T\boldsymbol{X}^{<m>}\boldsymbol{v} = \lambda\boldsymbol{X}^{<m>}\boldsymbol{v} \\
\Rightarrow\ &\boldsymbol{X}^{<m>}(\boldsymbol{X}^{<m>})^T\boldsymbol{w} = \lambda\boldsymbol{w}
\end{aligned}$$

thus, we have shown that $\boldsymbol{w} = \boldsymbol{X}^{<m>}\boldsymbol{v}$ is an eigenvector of $\boldsymbol{X}^{<m>}(\boldsymbol{X}^{<m>})^T$ with eigenvalue $\lambda$.

This result means that we can compute the eigenvectors of $(\boldsymbol{X}^{<m>})^T\boldsymbol{X}^{<m>}$, whose shape is $(D_m, D_m)$, and then linearly transform these to form $\boldsymbol{V}_X^{<m>}$. Since we assumed $D_m < D_{-m}$ this will result in reduced computational complexity compared to eigendecomposing $\boldsymbol{X}^{<m>}(\boldsymbol{X}^{<m>})^T$ directly, whose shape is $(D_{-m}, D_{-m})$.

### 12.1.5. Tensor kernel density estimation

The tensor distance function $d_{\text{tensor}}$ can be used to construct a kernel density estimator for an unknown probability density function in a tensor space. Generalizing the vector space KDE to a tensor-valued dataset $\boldsymbol{X}_1, \dots, \boldsymbol{X}_n$, we have:

$$\hat{p}(\boldsymbol{X}) \propto \sum_{i=1}^{n} K\left(\frac{d_{\text{tensor}}(\boldsymbol{X}, \boldsymbol{X}_i)}{\sigma}\right)$$

where the normalization constants have been omitted as they disappear in the expression for the CS divergence. $K$ is assumed to be a univariate standard Gaussian:

$$K(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}.$$

Using the exact same calculations as in the vector case allows us to write the CS divergence between two tensor-space-pdfs as:

$$D_{cs}(\hat{p}_1, \hat{p}_2) = -\ln\left(\frac{\sum\limits_{i,j=1}^{n} k_{\sqrt{2}\sigma}^{\text{tensor}}(\boldsymbol{X}_i, \boldsymbol{Y}_j)}{\sqrt{\sum\limits_{i,j=1}^{n} k_{\sqrt{2}\sigma}^{\text{tensor}}(\boldsymbol{X}_i, \boldsymbol{X}_j) \sum\limits_{i,j=1}^{n} k_{\sqrt{2}\sigma}^{\text{tensor}}(\boldsymbol{Y}_i, \boldsymbol{Y}_j)}}\right)$$

where $\boldsymbol{X}_1, \dots \boldsymbol{X}_n$ and $\boldsymbol{Y}_1, \dots, \boldsymbol{Y}_n$ denotes tensor-valued samples from $p_1$ and $p_2$, respectively. The RKHS interpretation of $D_{cs}$ is analogous to the vector-space case.

# 13. Deep Tensor Kernel Clustering

## 13.1. Motivation

The representational power of deep learning architectures is a double-edged sword when training deep clustering algorithms [24]. Although these models are indeed capable of producing vectorial representations from noisy and complex datatypes, they are also capable of drastically changing the between-sample similarities present in the dataset. The majority of deep clustering algorithms rely on the similarities computed from the network's output, meaning that one must adequately regularize the model such that input-similarities are somehow reflected in the output-similarities. Much work focuses on implementing this constraint through the use of autoencoders, where an additional clustering module is attached to the autoencoder's code space [17, 24, 21]. Recall that also the aforementioned DEC [15] pre-trains the embedding network as an autoencoder (using a reconstruction loss), and then fine-tunes the network using the clustering loss. Thus, the well known DEC also falls within the autoencoder-employing category of algorithms. However, the autonecoder is trained to reconstruct its input as best as possible, meaning that it does not necessarily produce representations suitable for clustering.

Even though the autoencoder approach is a popular one, there are algorithms that do well without the extra guidance provided by the reconstuction loss. Examples of such algorithms include SpectralNet [18] and DDC [20], which were both described earlier in this thesis. These algorithms are trained end-to-end using randomly initialized parameters without explicit regularization terms[14]. The ability to do without the autoencoder has the benefit of reducing the number of model parameters, potentially increasing training efficiency. Moreover, the end user does not have to worry about pre-training vs. fine-tuning, and other potential difficulties that might arise when combining the autoencoder and the clustering module. However, as is also the case with the autoencoder-based models, these algorithms are far from perfect. In DDC the authors state that the algorithm is indeed prone to getting stuck in local minima, potentially converging to sub-optimal clusterings.

[14]It should be noted however, that DDC's second loss term introduces a regularizing effect to the hidden representation.

## 13.2. Unsupervised companion objectives

In an attempt to alleviate some of the aforementioned difficulties, without sacrificing end-to-end trainability from randomly initialized parameters, a new approach to unsupervised loss function design is proposed in this thesis. The method is similar in spirit to *Deeply Supervised Nets* introduced by Lee et al. [34], where a collection of *companion objectives* are introduced to help guide the earlier layers of a supervised model. The main idea behind these companion objectives is to introduce a classifier at each layer in the network, and then jointly optimize these classifiers, as well as the final classification layer. In mathematical terms, we get the loss function:

$$\mathcal{L} = \mathcal{L}_{\text{classification}} + \sum_{i=1}^{L-1} \mathcal{L}_{\text{co}}^{i}$$

where $\mathcal{L}_{\text{classification}}$ denotes the ordinary loss function computed at the final layer, $L$ is the total number of layers, and $\mathcal{L}_{\text{co}}^{i}$ denotes a loss function (companion objective) computed at layer $i$. Although the Deeply Supervised Nets are – as the name implies – supervised, the main idea can be transferred to the domain of deep clustering.

Let us now assume that we have the standard deep clustering setup from Section 10:

$$\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{X}), \quad \boldsymbol{u} = g_{\boldsymbol{\phi}}(\boldsymbol{z})$$

where $f_{\boldsymbol{\theta}}$ denotes the neural network producing the learned representation $\boldsymbol{z}$, from the input $\boldsymbol{X}$, and $g_{\boldsymbol{\phi}}$ denotes the clustering module producing the cluster membership vector $\boldsymbol{u}$. Since $f_{\boldsymbol{\theta}}$ represents a neural network, it can be decomposed layer-wise as:

$$f_{\boldsymbol{\theta}} = f_{\boldsymbol{\theta}_L}^{L} \circ f_{\boldsymbol{\theta}_{L-1}}^{L-1} \circ \cdots \circ f_{\boldsymbol{\theta}_1}^{1}$$

where $f_{\boldsymbol{\theta}_l}^{l}$ is the mapping performed by layer $l$. If we let $\boldsymbol{Y}^l$ be the output of layer $l$, we have:

$$\boldsymbol{Y}^l = f_{\boldsymbol{\theta}_l}^{l}(\boldsymbol{Y}^{l-1})$$

with $\boldsymbol{Y}^0 = \boldsymbol{X}$ and $\boldsymbol{Y}^L = \boldsymbol{z}$. Note that we assume $\boldsymbol{X}$ and $\boldsymbol{Y}^1, \dots, \boldsymbol{Y}^{L-1}$ to be tensors of arbitrary rank, whereas $\boldsymbol{z}$ and $\boldsymbol{u}$ are assumed to be vectors (rank-1 tensors).

If we have a dataset $\mathcal{X} = \{\boldsymbol{X}_1, \dots, \boldsymbol{X}_n\}$ and a clustering loss function $\mathcal{L}_{\text{cluster}}(\mathcal{X}, \boldsymbol{\theta}, \boldsymbol{\phi})$, the proposed total loss function reads:

$$\mathcal{L} = \mathcal{L}_{\text{cluster}}(\mathcal{X}, \boldsymbol{\theta}, \boldsymbol{\phi}) + \lambda \sum_{l=1}^{L-1} \mathcal{L}_{\text{co}}^{l}(\boldsymbol{Y}_1^l, \dots \boldsymbol{Y}_n^l, \boldsymbol{U}) \tag{39}$$

where $\mathcal{L}_{\text{co}}^{l}(\boldsymbol{Y}_1^l, \dots \boldsymbol{Y}_n^l, \boldsymbol{U})$ is an unsupervised companion objective on the $l$-th layer, which depends on both the outputs of that layer and the cluster membership matrix produced by the clustering module. $\lambda$ is a hyperparameter which determines the strength of the companion objectives.

Following the idea of Deeply Supervised Nets, $\mathcal{L}^l_{\text{co}}$ should be designed to enforce a discriminative cluster structure at layer $l$. Similarly to DDC, this is done using the CS divergence, resulting in the companion objective:

$$\mathcal{L}^l_{\text{co}}(\boldsymbol{Y}^l_1, \dots \boldsymbol{Y}^l_n, \boldsymbol{U}) = \frac{1}{\binom{k}{2}} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\boldsymbol{v}_i^T \boldsymbol{K}^l \boldsymbol{v}_j}{\sqrt{\boldsymbol{v}_i^T \boldsymbol{K}^l \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K}^l \boldsymbol{v}_j}} \tag{40}$$

where $k$ is the number of clusters, $\boldsymbol{v}_i$ denotes the $i$-th *column* of the cluster membership matrix $\boldsymbol{U}$, and $\boldsymbol{K}^l = [\kappa^l_{ij}]$ is a kernel matrix whose elements are:

$$\kappa^l_{ij} = k^{\text{tensor}}_\sigma(\boldsymbol{Y}^l_i, \boldsymbol{Y}^l_j)$$

where $k^{\text{tensor}}_\sigma$ is the tensor kernel from the previous section. Note that the correct normalization factor $\binom{k}{2}^{-1}$ from [37] is used, instead of DDC's $\frac{1}{k}$.

Eq. (39) and Eq. (40) constitute the mathematical formulation of the proposed unsupervised companion objectives. In essence, these are designed to enforce similar cluster structure at each of the layers in the network, ensuring a more consistent similarity structure between the outputs of subsequent layers. Because the representational power of a single layer is limited, and each layer has its own companion objective, the similarity structure at later layers should more closely resemble the similarity structure at the earlier layers, compared to the unconstrained case.

## 13.3. Model overview

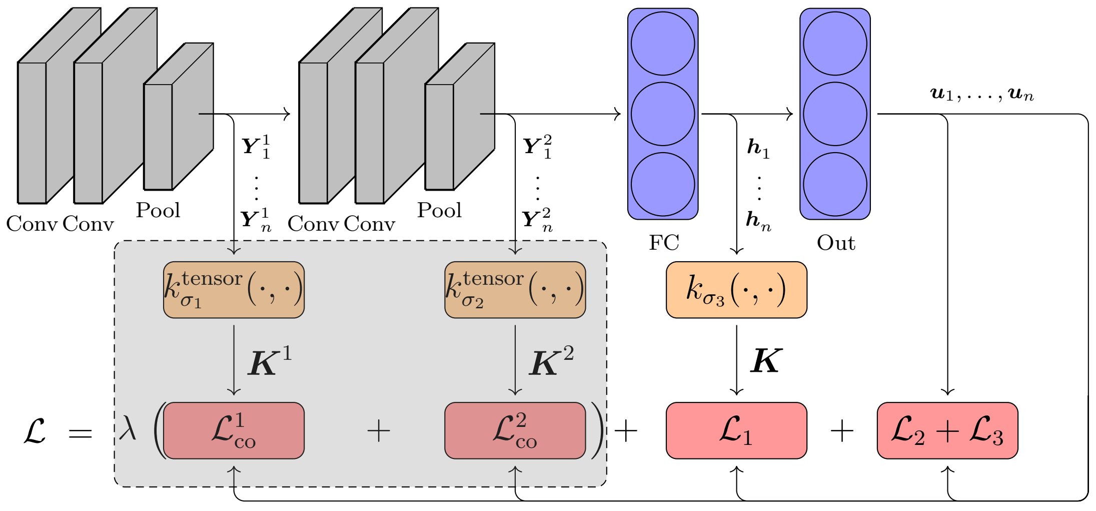

Conv
Conv
Pool
$Y_1^1$
$Y_n^1$
Conv
Conv
Pool
$Y_1^2$
$Y_n^2$
FC
$h_1$
$h_n$
Out
$u_1, \dots, u_n$
$k^{\text{tensor}}_{\sigma_1}(\cdot,\cdot)$
$k^{\text{tensor}}_{\sigma_2}(\cdot,\cdot)$
$k_{\sigma_3}(\cdot,\cdot)$
$K^1$
$K^2$
$K$
$\mathcal{L} = \lambda \left( \mathcal{L}^1_{\text{co}} + \mathcal{L}^2_{\text{co}} \right) + \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$

Figure 27: *An overview of the DTKC model. The dashed box shows the computation of the proposed unsupervised companion objectives. The tensor kernels are computed from the outputs of each pooling operation, and then used to compute the companion objectives for the respective layers.*

An overview of the complete DTKC model is shown in Figure 27. The "FC" and "Out" layers as well as the losses $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ are from DDC, which has been shown to work well for both image clustering and time series clustering [20, 22], with randomly initialized parameters. Moreover, the companion objectives closely resemble DDC's $\mathcal{L}_1$. The DDC loss function should therefore tend to agree with the companion objectives, so that they "pull in the same direction" during training. In order to strengthen this agreement, the normalization constant of $\mathcal{L}_1$ was changed from $\frac{1}{k}$ to $\binom{k}{2}^{-1}$ in DTKC.

Although DTKC shares its clustering module with DDC, it is important to emphasize that the unsupervised companion objectives can be coupled with any deep clustering algorithm, as long as it uses a deep neural network to produce the cluster membership predictions. The companion objectives have been introduced for tensors of arbitrary rank, meaning that they can be attached to any deep neural network which produces tensorial representations. To illustrate this generalizability, Section 17.9 includes some experiments where the companion objectives are used with a recurrent neural network to cluster sequential data.

# Part V / Experiments

This part describes the experiments that were performed in order to assess the performance of the proposed DTKC model. First, the datasets used for evaluation are summarized, and their contents are discussed briefly. The next section thoroughly covers the implementation details of DTKC, and the benchmark models to which it is compared. Following this is an overview of the CVIs used for qualitative evaluation, and then the experimental results. The final section in this part provides a discussion on different aspects of the performed experiments, as well as some thoughts on outlook and future work.

# 14. Datasets

A wide range of benchmark datasets were chosen to thoroughly evaluate the performance of DTKC. These datasets represent clustering tasks which are often encountered in computer vision, and are thus widely used in the literature [15, 24, 18, 20, 19]. An overview of the datasets can be found in Table 1, and plots showing some images from the respective datasets are shown in Figure 28. Compared to datasets for large-scale object detection in natural images, such as ImageNet [109] or the Pascal Visual Object Classes (VOC) [110], the chosen datasets include images with less complex scenery, and less within-class variation.

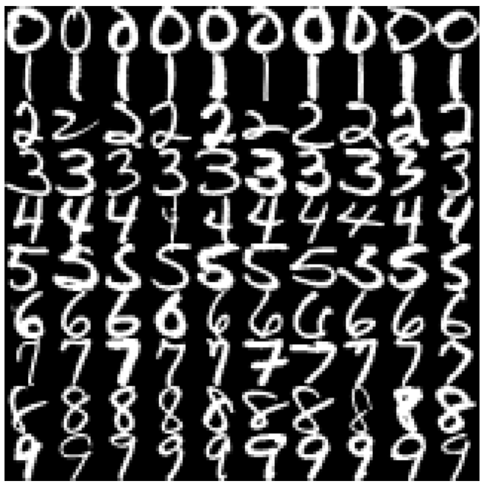
Table 1: *Overview of the datasets used for evaluation. $n$ and $k$ denote the total number of images, and the number of categories, respectively.*

| Name | Image size | Color | $n$ | $k$ | Contents |
|---|---|---|---|---|---|
| MNIST [111] | $28 \times 28$ | Gray | 60000 | 10 | Hand-written digits |
| USPS | $16 \times 16$ | Gray | 9298 | 10 | Hand-written digits |
| SVHN [112] | $32 \times 32$ | RGB | 99289 | 10 | House number digits |
| Fashion-MNIST [113] | $28 \times 28$ | Gray | 60000 | 10 | Clothing items |
| COIL-20 [114] | $128 \times 128$ | Gray | 1440 | 20 | Common objects |
| COIL-100 [115] | $128 \times 128$ | RGB | 7200 | 100 | Common objects |
| UMist [116] | $112 \times 92$ | Gray | 575 | 20 | Faces |

The MNIST, USPS and SVHN datasets contain images of digits. The task of the clustering system is therefore to group these images based on the depicted digit. Whereas the first two datasets contains distraction-free and noise-free grayscale images, the third dataset contains color images which can include both distractions in the form of other digits, as well as noise.

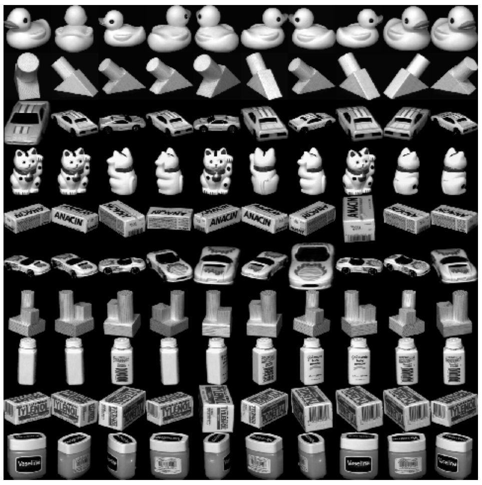
As the name implies, the format of the Fashion-MNIST dataset is similar to the MNIST dataset. The difference is that this dataset contains grayscale images of clothing items instead of digits, making it a somewhat harder clustering task [118]. The COIL-20 and COIL-100 datasets also both contain images of common

(a) *MNIST*


(b) *USPS*

(c) *SVHN*

(d) *Fashion-MNIST*


(e) *COIL-20*

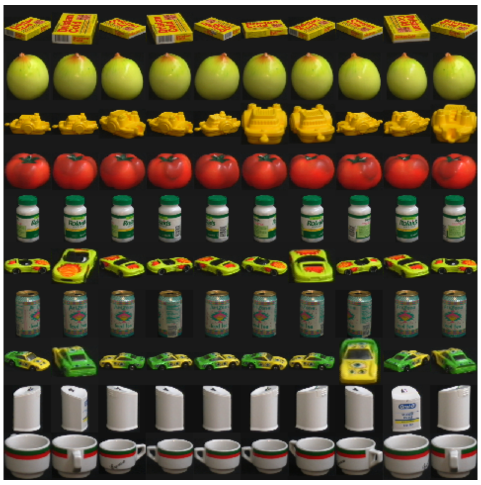

(f) *COIL-100*

Figure 28: *Random images drawn from the different datasets. Each row represents a different category. For the COIL-20 and COIL-100 datasets, the first* 10 *categories are shown. Note that images from the UMist dataset are omitted due to legal reasons [117].*

objects. However, in these sets each category consists of a single object depicted at different angles. This means that the source of within-category variation comes from the rotation of the given object. COIL-20 and COIL-100 contain 20 and 100 categories (objects), respectively. Although the number of categories is much larger for COIL-100, the added color information appears to be beneficial for recognizing the objects.

The UMist dataset consists of images of 20 different people where each person is depicted from several angles. The clustering task is to recognize the person in the image, independent of the imaging angle.

## 14.1. Dataset splits

In supervised learning it is common to split each dataset into a training set, a validation set, and a test set. The training set is then used to train the model, while the validation set is used for model selection and hyperparameter tuning. Finally, the test set is used to evaluate the performance of the final model. This last step is done to evaluate the generalization performance of the model [3]. However, in the recent literature on clustering, the common approach has been to use the full dataset for training, model/hyperparameter selection, and evaluation [119, 15, 24, 17, 18, 120, 121]. To obtain a fair comparison, the evaluation of the models proposed in this thesis is performed in this way. However, it should be stressed that this approach to evaluation does indeed neglect the generalization capabilities of the model, which can be important based on the application at hand.

# 15. Models

## 15.1. Classical models

Several benchmark algorithms were chosen to thoroughly assess the clustering performance of the proposed DTKC model. The following classical models were chosen as they represent some of the most widely used clustering algorithms:

- $k$-means [57].
- Average-link agglomerative hierarchical clustering.
- Spectral Clustering [61].

An overview of these algorithms was provided in Section 8.

### 15.1.1. Implementation details

The classical models were implemented using the `scikit-learn` module in Python. The images were vectorized as a preprocessing step for each of these algorithms, as they require vectorial inputs. The methods were configured as follows:

- $k$-means: The initial centroids were chosen according to the $k$-means++ initialization strategy. The algorithm was run for 300 iterations from 20 different initializations.
- Hierarchical Clustering: Agglomerative, average link.
- Spectral Clustering: A Gaussian affinity measure was used, where the value of $\sigma$ was randomly chosen. The clustering was repeated 20 times for different $\sigma$ values.

## 15.2. Deep clustering models

In addition to the classical models, several deep clustering models were also chosen for evaluation. These models were selected as they all employ similar approaches to joint feature extraction and clustering with deep neural networks, thus allowing for a fair performance comparison. The deep clustering models are:

- Deep Divergence-based Clustering (DDC) [20].
- Deep Divergence-based Clustering with self-supervised pre-training (DDC-SS).
- Deep Embedded Clustering (DEC) [15].
- SpectralNet [18].
- Improved Deep Embedded Clustering (IDEC) [24].
- Deep Clustering Network (DCN) [17].
- Discriminatively Boosted Clustering (DBC) [21].

(See Section 10 for an explanation of these algorithms). The results for DDC, DDC-SS, and DEC were obtained by the author of this thesis. The results for the remaining models were extracted from their respective references, which is why some models are missing in the tables below.

### 15.2.1. Implementation details

DDC was implemented in the `TensorFlow` framework in Python, mostly following the model specification in [20]. The implementation used here only deviates from the original formulation in the choice of CNN architecture (see Table 3) – a choice which proved to be beneficial for the performance of DDC. The reason for choosing the particular architecture is given in Section 15.3.1. DDC-SS is implemented in this manner as well, but has an additional self-supervised pre-training step. The parameters for the pre-training are the same as for DTKC-SS, and are also described in Section 15.3.1.

The DEC implementation was based on `DEC-Keras`[15]. The hyperparameters and network architecture were chosen according to the original article [15].

[15]`https://github.com/XifengGuo/DEC-keras`

## 15.3. DTKC and DTKC-SS

The experiments were performed with two different variants of the proposed model. The first is DTKC trained from randomly initialized network-parameters. The second variant is DTKC-SS, which is the same as DTKC, but with a CNN that has been pre-trained using self-supervised learning.

### 15.3.1. Implementation details

The specification of hyperparameters can be somewhat challenging for clustering applications, due to the aforementioned difficulties of quantitatively measuring clustering performance. In supervised learning, the hyperparameter specification is usually done by searching some portion of the hyperparameter space, and then selecting the configuration which resulted in the best model performance, with respect to some supervised performance measure. In clustering, one cannot assume that labeled data is available in the general case, meaning that it is not possible to use external CVIs. This leaves us with internal CVIs. These can be used, but often produce sub-optimal results due to their dependence on some distance (or dissimilarity) function in the input space.

At first, one might think that the creation of a small validation set by a domain expert is a possible approach, as this would allow for the use of external CVIs. However, there are two main problems with this approach:

(i) The validation set has to be sufficiently large to represent the majority of the data distribution, meaning that its creation can be a time consuming process.

(ii) If such a set was made available before training the algorithm, why not use a semi-supervised algorithm instead? This would tend to be beneficial as long as the desired categorization is represented in the validation set.

Unless something else is explicitly stated, the choice of the different hyperparameters were therefore made by inspecting the loss function during training. Both the value of the loss function, and its tendency to decrease during training, were considered when configuring the models. Low values of the loss function and small variations were considered favorable as they are indications of a good clustering, and a stable training procedure, respectively.

#### Batch size

Both DTKC and DTKC-SS were trained on batches of size 120. This batch size was chosen as it resulted in a stable training procedure. Note that it can be difficult to base the choice of batch size directly on the value of the loss function because the loss tends to depend on the batch size. Another consideration is that several of the terms in the loss function are based on kernel density estimation,

Table 2: *Parameters for the self-supervised pre-training.*

| | Parameter | |
|---|---|---|
| Dataset | Patch size | Jitter |
| MNIST | $7 \times 7$ | 1 |
| USPS | $5 \times 5$ | 0 |
| SVHN | $7 \times 7$ | 1 |
| Fashion-MNIST | $7 \times 7$ | 1 |
| COIL-20 | $24 \times 24$ | 4 |
| COIL-100 | $24 \times 24$ | 4 |
| UMist | $24 \times 24$ | 4 |

which requires a large number of samples to produce reliable estimates [99]. This last consideration will be revisited in Section 18.1.

## Initialization

The parameters of all convolutional layers, as well as the first fully-connected layer, were initialized according to [30]. The last fully-connected layer uses the initialization strategy proposed in [122]. For DTKC the randomly initialized network was trained end-to-end using the clustering loss function described in Section 13. For DTKC-SS the randomly initialized CNN was first trained using self-supervised learning (see Section 9.5), before being fine-tuned with the clustering loss described in Section 13. The parameters used for the self-supervised pre-training are listed in Table 2. These were selected such that the height and width of each image was approximately equal to three or four times the height and width of a patch, plus the jitter.

## Network architecture

The architecture of the CNN was fixed for DTKC and DTKC-SS, as well as the DDC benchmark model[16]. The fixed architecture strategy was decided upon mainly due to the difficulties of unsupervised hyperparameter tuning. Moreover, the purpose of these experiments is not necessarily to search high and wide for the best overall clustering performance on all datasets – but rather to thoroughly evaluate the performance of the proposed models compared to relevant benchmarks, as well as to inspect the effect of the proposed technique on intrinsic parts of the CNN. Keeping the network architecture fixed aligns with these goals, as a fixed architecture reduces the complexity of the experimentation by creating a more controlled experimental environment. This is compared to the converse case where the network architecture is treated more as a set of hyperparameters.

A summary of the CNN is given in Table 3, which also includes the "names" for the different parts of the network. Note that for the convolution operations, the

[16]The benchmark DDC was also modified in order to obtain the best possible comparison.

Table 3: *Overview of CNN used for experimentation. All convolutions are performed with "valid" padding, while the MaxPool operations use "same" padding. The number of units in the last layer is equal to the number of clusters for the given dataset ($k$).*

| | |
|---|---|
| Convolutional layer 1 | Convolution ($5 \times 5 \times 32$)<br>ReLU<br>Convolution ($5 \times 5 \times 32$)<br>Batch normalization<br>ReLU<br>MaxPool |
| Convolutional layer 2 | Convolution ($3 \times 3 \times 32$)<br>ReLU<br>Convolution ($3 \times 3 \times 32$)<br>Batch normalization<br>ReLU<br>MaxPool |
| Fully-connected layer 1 | Flatten<br>Fully-connected (100)<br>ReLU<br>Batch normalization |
| Output layer/ Clustering module | Fully-connected ($k$)<br>Softmax |

notation "$5 \times 5 \times 32$" means that 32 different convolutions are performed, each having filters with shapes $5 \times 5$. The depth of the filters is equal to the number of channels in the input to the convolution (i.e $D_3$ in tensor-shape notation). The parenthesized number at the fully-connected layers indicate the number of units in the output layer.

The network has proven to be sufficiently large for satisfactory performance, while simultaneously being small enough to have a manageable memory footprint and training time. Recall that the original DDC architecture used two convolution operations, each of which was followed by max-pooling operations. The addition of two more convolution operations to this network was done to increase the representational power of the CNN, which in turn should lead to more prominent differences between DTKC, and DDC. Note that max-pooling was omitted after the two new convolutions as these would cause the tensors in later layers to have negative dimensions. Batch normalization [90] was also added to increase the stability of the training procedure.

## Other parameters

- Optimizer: The models were trained using the Adam optimizer [89] with a learning rate of $10^{-4}$.
- Gradient clipping: Abnormally large gradients were sometimes observed during training. Gradients with a magnitudes larger than 10 were therefore

clipped to this magnitude [11]. This was also found to improve training stability.

- Epochs: The models were trained for a maximum of 100 epochs. A training run was terminated if no decrease in the loss function was observed over 30 epochs. This approach was observed to be sufficient for the training procedure to converge. Note that the models were saved at the epoch at which the loss-value was at its lowest.
- Runs: Each model was trained 20 times.
- Bandwidth: The $\sigma$ parameter for each kernel was set to 15 % of the mean pairwise distances between the activations from the respective layers, following [100, 20].
- Companion objective strength: The $\lambda$ parameter was set to 0.01 for all experiments. The reasons for this choice are further discussed in Section 17.4.

# 16. Metrics

For the quantitative evaluation of the proposed models and the comparison of these with the benchmark models, the following two external CVIs were chosen:

- Unsupervised clustering accuracy (ACC).
- Normalized mutual information (NMI).

These were chosen as they are the most frequently used performance measures in deep clustering [119, 15, 24, 17, 18, 120, 121], and therefore allows for an evaluation which is compatible with the literature. However, since these CVIs assess the clustering system by its capability of reproducing a specific set of clusters, they should not be blindly trusted as definitive measures of overall clustering performance. This notion is taken into account in the presentation and discussion of the results.

The metrics were computed for the models resulting in the lowest value of the overall loss function for each run. This resulted in 20 (ACC, NMI)-pairs, where each pair corresponds to the best observed performance for the given run, with respect to the loss function. These pairs were further aggregated to produce the following four summary statistics:

- *Best*: ACC and NMI for the run which resulted in the lowest value of the loss function. This is used as the primary measure of model performance, as the best model is selected in a completely unsupervised manner.
- *Mean*: Average ACC and NMI over the 20 runs.
- *Sd.*: Standard deviation for ACC and NMI over the 20 runs.

- *Max*: The highest observed ACC and NMI for the 20 runs. These are selected based on external CVI's and should therefore not be used for direct comparisons between unsupervised models. The only reason for including this statistic is to give an impression of the "best case" performace of the model.

This collection of statistics should give thorough insight into the capabilities of the different models with respect to the chosen CVIs, both in terms of performance and in terms of stability.

# 17. Results

The results of the experiment setup above are summarized and discussed in this section. The quantitative results are discussed first, followed by several subsections whose aim are to both present and discuss more specific aspects of the resulting models. It is important to note that the main purpose of this section is to thoroughly assess the effect of self-supervised pre-training and the unsupervised companion objectives. Thus, more emphasis is put on the comparison between the DDC-inspired models (DDC, DTKC, and DTKC-SS), and not necessarily on the comparison of these together with the benchmark models. We must also keep in mind that clustering evaluation is not always as straightforward as e.g. classification [23], meaning that some qualitative verification of the learned clusters is necessary. This is especially important when one uses external CVIs to evaluate the clusterings, as the clustering system can learn to cluster objects based on entirely different properties than those given by the ground truth labels.

Tables 4 and 5 show the results for the different experiments described above. In general, one can observe large gaps in performance, both between different models on the same dataset, as well as across different datasets. These variations indicate that the chosen datasets do indeed offer clustering tasks of varying difficulty. From Tables 4 and 5, we can make the following observations:

- **Deep clustering methods tend to outperform the classical methods.** This tendency is apparent for almost all datasets, with the UMist dataset being the only exception. This difference in performance between the classical and deep models is likely due to the fact that the classical models all use the squared Euclidean distance as a measure of dissimilarity – whereas the deep models does not make any explicit assumptions about the distance function in the input space.

  For the UMist dataset, it is possible that the deep clustering models struggle due to the small number of samples (see Table 1). This is consistent with previous work [123, 124], which has shown that the performance of deep learning methods tends to increase significantly with the number of observations present in the dataset.

- **DEC is mostly outperformed by DDC, DTKC, and DTKC-SS.** One probable reason for this particular result is that DDC, DTKC and DTKC-SS are based on CNNs, while DEC uses an MLP to extract the representation $\boldsymbol{z}$. Similar observations have been made in supervised learning, where CNNs regularly outperform MLPs in image processing tasks [13, 11].

  Although the absolute performance is somewhat worse for DEC compared to DDC, DTKC, and DTKC-SS, its standard deviations are consistently lower. This can indicate that the autoencoder pre-training is somewhat more stable compared to random initialization or self-supervised pre-training.

- **Self-supervised pre-training does not seem to increase overall performance.** Comparing the models pre-trained using self-supervised learning ( DDC-SS and DTKC-SS) to their randomly initialized couterparts (DDC and DTKC), reveals no overall indications of improved performance. The potential gain or loss in performance varies from dataset to dataset, implying that the effect of self-supervised pre-training is mostly data dependent.

  The results also show that the differences in standard deviations are marginal when comparing random initialization and self-supervised pre-training. The effect of the pre-training step is discussed further in Sections 17.7 and 18.2.

- **Clustering performance for the SVHN dataset is quite bad.** For the SVHN dataset, none of the clustering algorithms seem to be able to cluster the images based on the digit they contain. This is not surprising due to the complex structure of the images contained in this dataset (see Figure 28c). This is discussed further in Sections 17.1 and 17.2.

Combined with the experimental results, these observations provide an overview of the capabilities of the different models. The rest of this section will describe several different aspects of these results in more detail, covering key observations that are not directly visible in the result tables.

Table 4: *Resulting ACC and NMI for the MNIST, USPS, and SVHN experiments. The best results are highlighted in bold.*

MNIST

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| DBC | 0.96 | – | – | – | **0.92** | – | – | – |
| DCN | 0.83 | – | – | – | 0.81 | – | – | – |
| IDEC | 0.88 | – | – | – | 0.87 | – | – | – |
| SpectralNet | **0.97** | – | – | – | **0.92** | – | – | – |
| $k$-means | 0.51 | 0.54 | 0.03 | 0.58 | 0.49 | 0.5 | 0.01 | 0.52 |
| Hierarchical | 0.21 | 0.21 | 0.0 | 0.21 | 0.31 | 0.31 | 0.0 | 0.31 |
| Spectral | 0.5 | 0.55 | 0.02 | 0.56 | 0.51 | 0.51 | 0.01 | 0.52 |
| DEC | 0.83 | **0.87** | 0.03 | **0.95** | 0.85 | **0.86** | 0.01 | **0.89** |
| DDC | 0.91 | 0.77 | 0.07 | 0.91 | 0.83 | 0.73 | 0.06 | 0.83 |
| DDC-SS | 0.88 | 0.76 | 0.09 | 0.9 | 0.83 | 0.74 | 0.07 | 0.84 |
| DTKC | 0.94 | 0.77 | 0.08 | 0.94 | 0.88 | 0.74 | 0.07 | 0.88 |
| DTKC-SS | 0.87 | 0.76 | 0.07 | 0.87 | 0.84 | 0.74 | 0.06 | 0.84 |

USPS

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| DBC | 0.74 | – | – | – | 0.72 | – | – | – |
| IDEC | 0.76 | – | – | – | 0.78 | – | – | – |
| $k$-means | 0.67 | 0.63 | 0.04 | 0.69 | 0.63 | 0.62 | 0.02 | 0.65 |
| Hierarchical | 0.22 | 0.22 | 0.0 | 0.22 | 0.19 | 0.19 | 0.0 | 0.19 |
| Spectral | 0.59 | 0.65 | 0.02 | 0.66 | 0.6 | 0.6 | 0.0 | 0.61 |
| DEC | 0.76 | **0.76** | 0.0 | 0.77 | 0.78 | **0.78** | 0.0 | 0.79 |
| DDC | **0.81** | 0.69 | 0.06 | **0.83** | 0.77 | 0.7 | 0.03 | 0.77 |
| DDC-SS | 0.72 | 0.69 | 0.05 | 0.78 | 0.7 | 0.7 | 0.03 | 0.78 |
| DTKC | 0.78 | 0.7 | 0.06 | 0.79 | **0.8** | 0.73 | 0.05 | **0.81** |
| DTKC-SS | 0.69 | 0.68 | 0.05 | 0.76 | 0.69 | 0.69 | 0.04 | 0.75 |

SVHN

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| $k$-means | 0.13 | 0.13 | 0.0 | 0.13 | 0.02 | 0.02 | 0.0 | 0.02 |
| Hierarchical | **0.19** | 0.19 | 0.0 | 0.19 | 0.0 | 0.0 | 0.0 | 0.0 |
| Spectral | 0.13 | 0.14 | 0.0 | 0.14 | 0.02 | 0.02 | 0.0 | 0.02 |
| DEC | 0.18 | **0.2** | 0.01 | **0.23** | **0.07** | **0.09** | 0.02 | **0.14** |
| DDC | 0.15 | 0.14 | 0.01 | 0.17 | 0.02 | 0.02 | 0.01 | 0.04 |
| DDC-SS | **0.19** | 0.18 | 0.01 | 0.2 | 0.06 | 0.05 | 0.01 | 0.08 |
| DTKC | 0.17 | 0.18 | 0.01 | 0.21 | 0.05 | 0.06 | 0.01 | 0.09 |
| DTKC-SS | 0.14 | 0.14 | 0.01 | 0.17 | 0.02 | 0.02 | 0.01 | 0.05 |

Table 5: *Resulting ACC and NMI for the Fashion-MNIST, COIL-20, COIL-100, and UMist experiments. The best results are highlighted in bold.*

Fashion-MNIST

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| $k$-means | 0.47 | 0.52 | 0.04 | 0.61 | 0.51 | 0.51 | 0.01 | 0.54 |
| Hierarchical | 0.1 | 0.1 | 0.0 | 0.1 | 0.01 | 0.01 | 0.0 | 0.01 |
| Spectral | 0.48 | 0.5 | 0.02 | 0.52 | 0.5 | 0.5 | 0.01 | 0.52 |
| DEC | 0.56 | 0.55 | 0.01 | 0.58 | 0.6 | **0.6** | 0.01 | 0.62 |
| DDC | 0.58 | 0.55 | 0.06 | 0.65 | 0.52 | 0.49 | 0.04 | 0.56 |
| DDC-SS | 0.57 | **0.58** | 0.05 | **0.7** | 0.58 | 0.58 | 0.04 | **0.65** |
| DTKC | 0.63 | 0.56 | 0.04 | 0.63 | 0.55 | 0.5 | 0.03 | 0.56 |
| DTKC-SS | **0.65** | 0.55 | 0.05 | 0.65 | **0.62** | 0.55 | 0.03 | 0.62 |

COIL-20

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| DBC | **0.79** | – | – | – | **0.9** | – | – | – |
| $k$-means | 0.69 | 0.63 | 0.04 | 0.7 | 0.8 | 0.77 | 0.01 | 0.8 |
| Hierarchical | 0.35 | 0.35 | 0.0 | 0.35 | 0.67 | 0.67 | 0.0 | 0.67 |
| Spectral | 0.61 | 0.65 | 0.03 | 0.68 | 0.76 | 0.77 | 0.01 | 0.78 |
| DEC | 0.67 | 0.66 | 0.04 | 0.71 | 0.77 | 0.77 | 0.01 | 0.8 |
| DDC | 0.68 | 0.64 | 0.03 | 0.71 | 0.77 | 0.75 | 0.03 | 0.81 |
| DDC-SS | 0.71 | 0.67 | 0.03 | 0.72 | 0.81 | **0.79** | 0.02 | 0.82 |
| DTKC | 0.62 | 0.64 | 0.03 | 0.69 | 0.75 | 0.75 | 0.03 | 0.8 |
| DTKC-SS | 0.71 | **0.68** | 0.03 | **0.74** | 0.82 | **0.79** | 0.02 | **0.83** |

COIL-100

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| DBC | **0.78** | – | – | – | **0.91** | – | – | – |
| $k$-means | 0.62 | **0.61** | 0.01 | 0.63 | 0.83 | 0.83 | 0.0 | 0.84 |
| Hierarchical | 0.23 | 0.23 | 0.0 | 0.23 | 0.68 | 0.68 | 0.0 | 0.68 |
| Spectral | 0.44 | 0.56 | 0.05 | 0.61 | 0.76 | 0.81 | 0.02 | 0.83 |
| DEC | 0.59 | **0.61** | 0.02 | **0.65** | 0.84 | **0.85** | 0.01 | **0.86** |
| DDC | 0.61 | 0.59 | 0.02 | 0.63 | 0.83 | 0.83 | 0.01 | 0.84 |
| DDC-SS | 0.5 | 0.49 | 0.03 | 0.53 | 0.78 | 0.77 | 0.01 | 0.79 |
| DTKC | 0.64 | 0.6 | 0.02 | **0.65** | 0.85 | 0.83 | 0.01 | 0.85 |
| DTKC-SS | 0.51 | 0.5 | 0.02 | 0.52 | 0.77 | 0.77 | 0.01 | 0.78 |

UMist

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| $k$-means | 0.4 | 0.42 | 0.02 | **0.46** | 0.59 | 0.61 | 0.01 | 0.64 |
| Hierarchical | **0.46** | **0.46** | 0.0 | **0.46** | 0.66 | 0.66 | 0.0 | 0.66 |
| Spectral | 0.36 | 0.42 | 0.03 | **0.46** | 0.58 | 0.63 | 0.02 | 0.66 |
| DEC | 0.08 | 0.24 | 0.12 | 0.4 | **1.69** | **0.75** | 0.56 | **1.69** |
| DDC | 0.31 | 0.36 | 0.03 | 0.42 | 0.46 | 0.52 | 0.02 | 0.55 |
| DDC-SS | 0.35 | 0.35 | 0.02 | 0.4 | 0.51 | 0.51 | 0.02 | 0.54 |
| DTKC | 0.29 | 0.36 | 0.04 | 0.42 | 0.45 | 0.52 | 0.04 | 0.59 |
| DTKC-SS | 0.34 | 0.34 | 0.03 | 0.41 | 0.51 | 0.51 | 0.03 | 0.57 |

## 17.1. Visualization of learned clusters

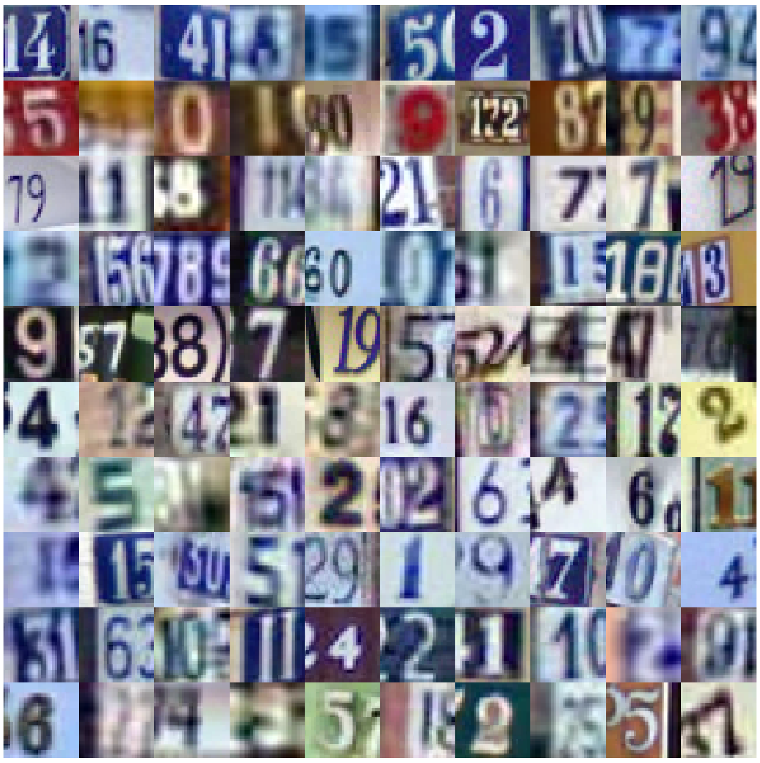


Figure 29: *Clusters produced by the best DTKC model for the SVHN dataset.*

The preceding evaluations are solely based on external criteria, meaning that all algorithms are measured by their ability to reproduce some predetermined categorization. This appears to be especially problematic for the SVHN dataset, where all methods perform very poorly according to the reported metrics. This behavior is unsurprising if we consider the type of images contained in the SVHN dataset. Figure 29 shows some example clusters extracted from the SVHN dataset. Each row of images corresponds to a cluster. The figure shows that there is indeed a very large variation in the appearance of the digits, as well as varying forms of distractions and noise. When inspecting these images, no obvious reason for this particular clustering reveals itself. The top row does seem to contain images that are more "blue-ish" than the rest of the clusters, while the second row contains images with more red colors. The bottom cluster on the other hand, seems to contain images that are more blurry than the rest. However, these arguments remain speculative, as these traits are not particularly prominent, and not without counterexamples.

When inspecting the learned clusters in this manner, one must also keep in mind that it is possible for the clusterings to be more or less random, and without any particular structure. A consequence of the design of most deep clustering algorithms is that they will always produce *some* clustering, regardless of the provided dataset, and presence of categories therein. Consequently, if no cluster structure is visible to the algorithm, it will end up producing some nonsensical clustering, which often tends to depend on the randomly initialized parameters of the model (if there are any). This somewhat unfortunate property has led researchers to discuss the necessity of algorithms producing clusterings that are, in a statistical sense, significantly different from random clusterings [23].

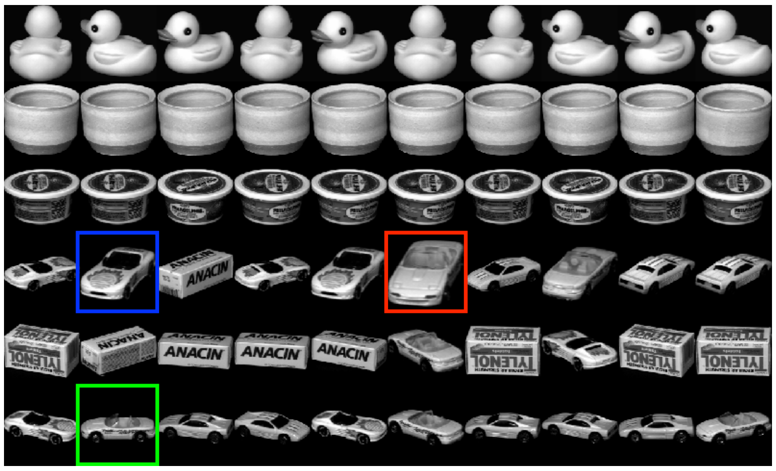
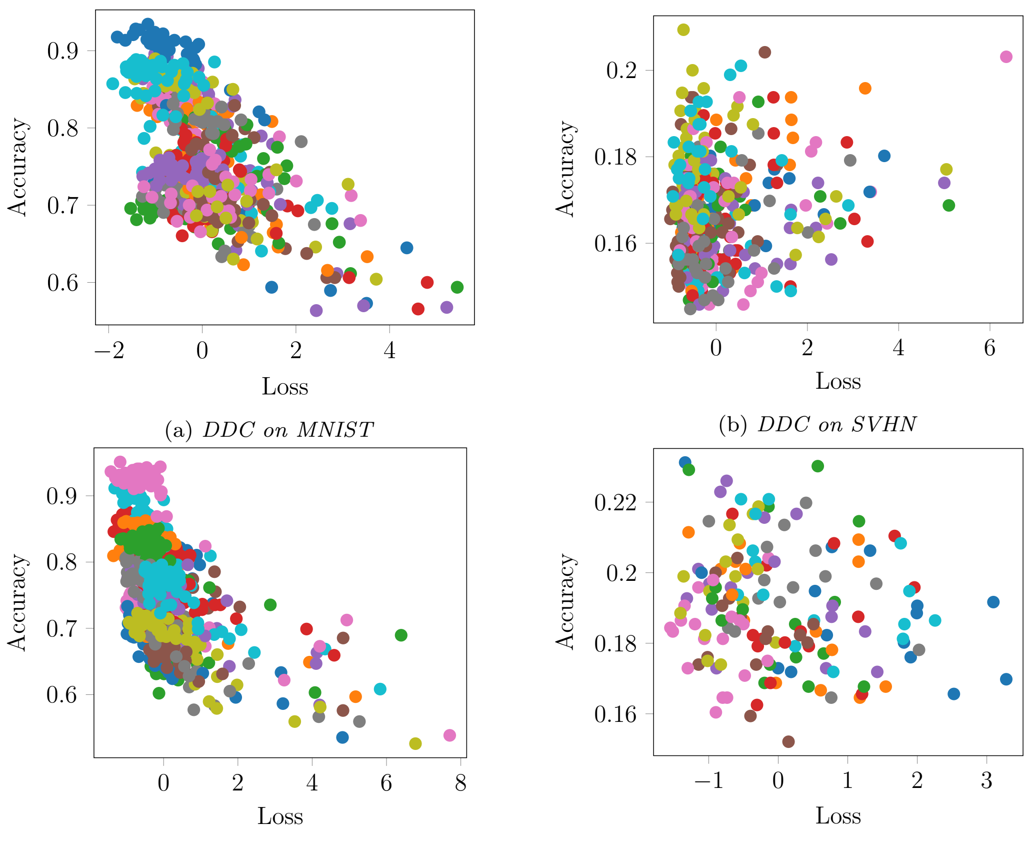


Figure 30: *Clusters produced by the best DTKC model for the COIL-20 dataset.*

Figure 30 shows some of the learned clusters for the COIL-20 dataset. Although not perfect, these clusters are much more interpretable than the SVHN clustering discussed above. Items that contain characteristic shapes, like the head of the rubber duck, or items that are rotationally symmetric, e.g. rows 2 and 3, are easily separated into their own clusters. However, the last three rows show clusters that are not as pure as the first three. In these clusters the algorithm is unable to distinguish between the three different cars, and sometimes confuses them with the Anacin-box.

The car confusion also poses the more complicated question of whether we *should* expect the three cars to be placed in three different clusters, regardless of their orientation. As an example, consider image 4-6 (row 4, column 6), and compare it to image 6-2 and image 4-2. How can we say that images 4-6 and 6-2 (same cars, different rotation) are more similar than images 4-6 and 4-2 (different cars, same rotation)? A system capable of determining the presence of a decal on the car's hood will correctly separate the two cars, whereas a system focusing on the overall shape of the car will not. The orientation of the cars is arguably a more low-level feature which is easier to identify than the decal on the hood. It is therefore more natural, and quite possibly favorable, that the system focuses on these "more obvious" clusterings [23].

## 17.2. Correspondence between loss and accuracy



(a) *DDC on MNIST*

(b) *DDC on SVHN*

(c) *DTKC on MNIST*

(d) *DTKC on SVHN*

Figure 31: *Loss vs. accuracy for the DDC and DTKC models on the MNIST and SVHN datasets. Different colors correspond to different runs. The loss-axis is normalized to allow for more accurate comparisons between models. Note that the early stopping criterion resulted in a noticeable difference in the number of samples for DDC and DTKC on the SVHN dataset.*

Table 6: *Results of fitting a simple linear regression model to the loss-accuracy pairs during training. s denotes the standard error of the slope estimate. The p-value is the result of a two-sided t-test for whether or not the slope is equal to zero.*

| Dataset | Model | Slope | $s$ | $R^2$ | $p$-value |
|---|---|---|---|---|---|
| MNIST | DDC | -0.044 | 0.002 | 0.333 | 0.0 |
| | DTKC | -0.041 | 0.002 | 0.264 | 0.0 |
| SVHN | DDC | 0.002 | 0.001 | 0.03 | 0.0 |
| | DTKC | -0.003 | 0.001 | 0.029 | 0.034 |

Another approach to more quantitatively evaluate the validity of an external CVI, is to check the correspondence between what the model thinks is a “good” clustering, and the resulting CVI. If a model is trained to minimize some loss function, it means that, from the model’s perspective, the lower the loss function, the better the clustering.

Figure 31 shows loss-accuracy pairs sampled during the 20 training runs, for the DDC and DTKC models, and for the MNIST and SVHN datasets. For the MNIST dataset there is a clear negative correspondence between loss and accuracy, while no such trend is visible for the SVHN dataset. These observations are confirmed by the results of Table 6, which contains summary statistics for a simple linear regression model, fitted to the respective sets of loss-accuracy pairs. The table shows that there is a statistically significant[17] negative linear relationship between loss and accuracy for the MNIST models.

When it comes to the SVHN dataset, there is actually a significant, but small positive trend for the DDC model, indicating that decreasing the loss function corresponds to a decrease in accuracy. Clearly, this is not the desired outcome if we want the clustering to reflect the depicted digits. There is a significant negative trend for DTKC, but this is still not large enough to result in satisfactory performance.

It should be noted that the $R^2$ values for the regression models are quite low, indicating that linear models are not very accurate in predicting the relationship between loss and accuracy. Moreover, the independence assumption might not hold for the sampled pairs, as multiple samples were extracted for each run. However, the regression model should still provide some overall insight into the true relationship between loss and accuracy.

## 17.3. Visualization of kernel matrices

To investigate the effect of the unsupervised companion objective on the layer-outputs, kernel matrices extracted from the CNNs of DDC and DTKC during training on the MNIST dataset, are shown in Figure 32. The kernels were computed from the output of the convolutional layers, using the same tensor kernel as in the unsupervised companion objectives. The rows and columns of the kernel matrices were sorted according to the ground truth for visualization purposes.

The class structure is not especially prominent for either of the two models at layer 1. However, the class structure is visible already at epoch 4 for both models at layer 2, with similar separability between classes for the two models at this stage. At epoch 20, the classes are somewhat more separable with DTKC, especially when it comes to the images containing 1's. This trend is also visible between the two models et epoch 50. This is as expected since the cluster structure coincides well with the class structure (see Table 4), and DTKC is trained to explicitly enforce this cluster structure at the intermediate layers.

[17]Using a significance level of 0.05.

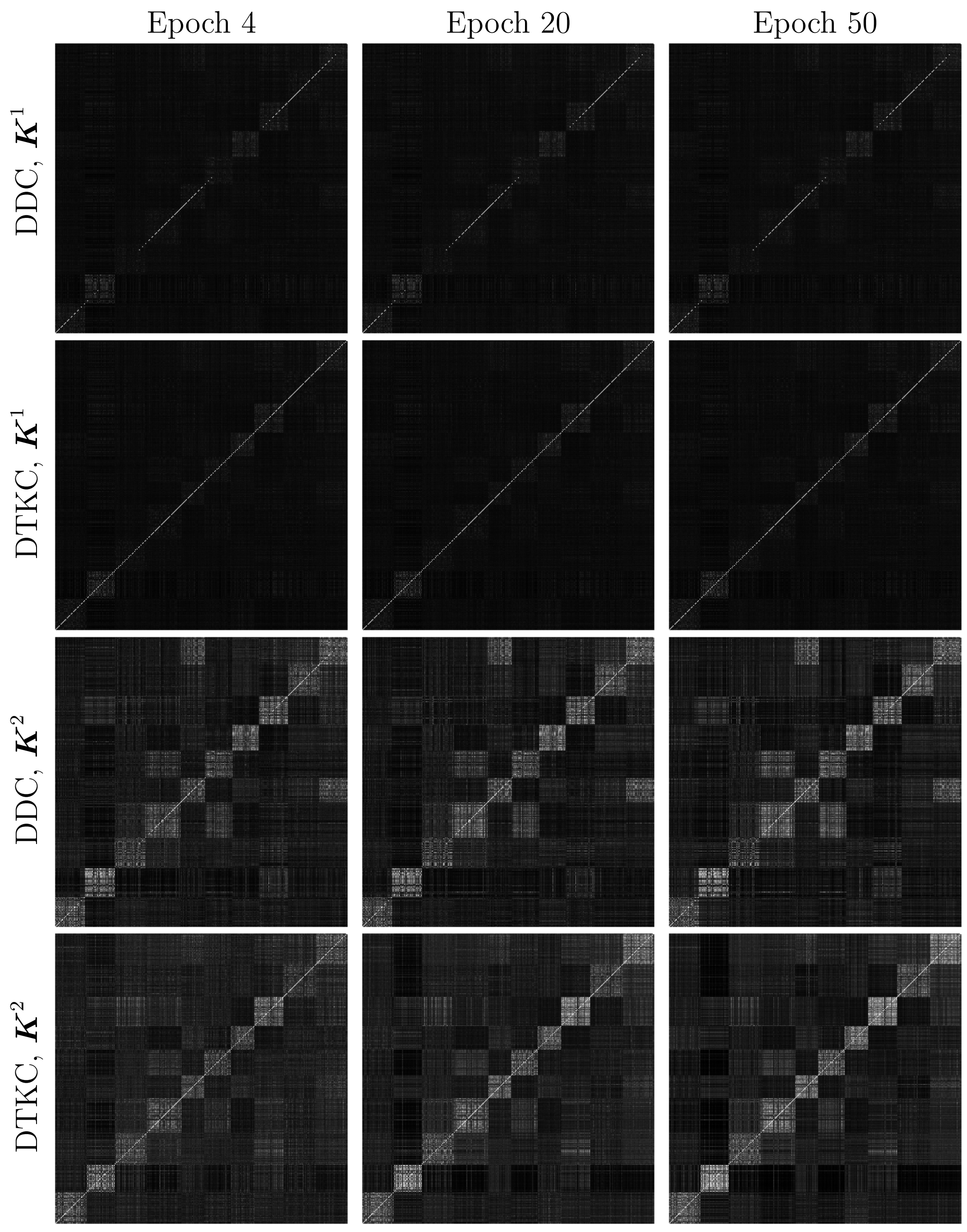


Figure 32: *Kernel matrices computed for a random subset of* 1024 *MNIST samples. Brighter pixels indicate higher values. The kernel matrices are sorted such that the bottom left block corresponds to the 0's, and the top right block corresponds to the 9's.*

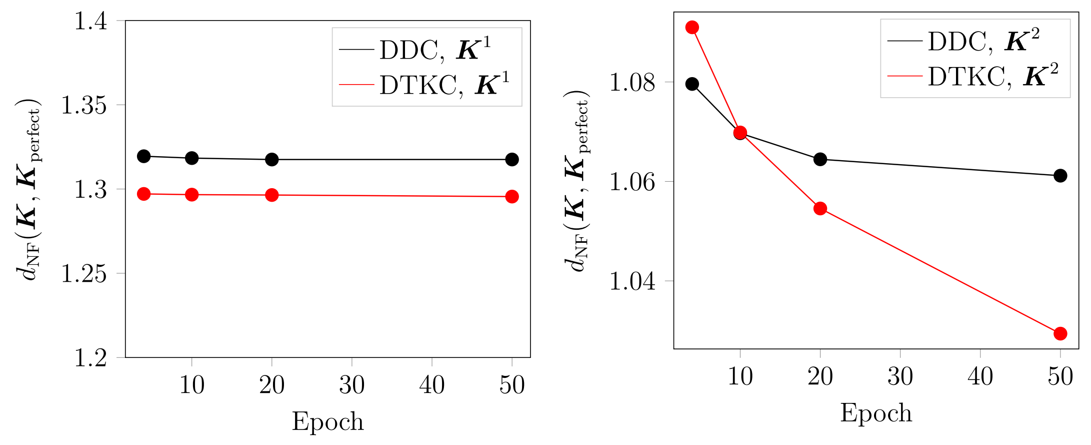


Figure 33: *Normalized Frobenius norm between the perfect kernel, and the kernel matrices shown in Figure 32.*

To further validate the claim that DTKC has learned intermediate embeddings which are more discriminative, the normalized Frobenius norm was computed between the extracted kernels, and the perfect kernel, defined as:

$$d_{\mathrm{NF}}(\boldsymbol{K}^l, \boldsymbol{K}_{\mathrm{perfect}}) = \left|\left| \frac{\boldsymbol{K}^l}{||\boldsymbol{K}^l||_F} - \frac{\boldsymbol{K}_{\mathrm{perfect}}}{||\boldsymbol{K}_{\mathrm{perfect}}||_F} \right|\right|_F$$

where:

$$K_{\mathrm{perfect},ij} = \begin{cases} 1, & \text{if observations } (i,j) \text{ belong to the same class} \\ 0, & \text{otherwise} \end{cases}$$

and:

$$||\boldsymbol{A}||_F = \sqrt{\sum_{i,j} A_{ij}^2}.$$

Figure 33 shows the resulting normalized Frobenius norms at different training epochs. The plots show that the kernels extracted from DTKC's CNN are more similar to the perfect kernel, reflecting the "true" cluster structure. Moreover, the normalized Frobenius norm has a steeper downward trend for the DTKC model at layer 2, which is consistent with the goal of the companion objectives. These observations therefore show that the addition of companion objectives has made the cluster structure more prominent within these intermediate representations.

## 17.4. Choosing the unsupervised companion objective strength $\lambda$

The specification of the strength of the unsupervised companion losses ($\lambda$) is a both crucial and difficult task. The approach taken in this thesis is somewhat similar to the approach outlined in Section 15.3, but does not require the creation of validation sets by domain experts. Here $\lambda$ was chosen based on a supervised validation procedure on the MNIST benchmark dataset, and the resulting value was used in all subsequent experiments. Clearly, one assumes with this approach that the hyperparameter configuration determined on the benchmark dataset generalizes to the other datasets, meaning that this approach is not problem free either.

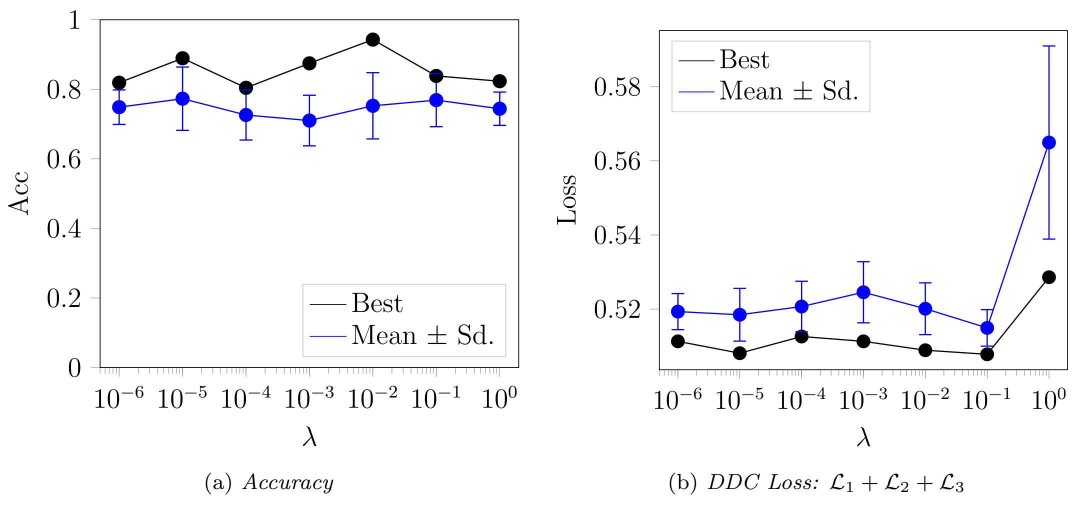


Figure 34: *Effect of the $\lambda$ hyperparameter on accuracy and DDC loss for the MNIST dataset.*

The resulting accuracies for the DTKC model on the MNIST dataset are shown in Figure 34a. The plot does not show a particularly large variation in accuracy, but the optimal value seems to be at around $\lambda = 0.01$. The small variation implies that the model is rather robust to variations in $\lambda$. This last property is crucial, as it makes the assumption about generalization to other datasets more feasible. Based on these observations, as well as the difficulties with unsupervised hyperparameter validation, $\lambda$ was set to 0.01 for all experiments.

Figure 34b shows the DDC loss ($\mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$) when training the DTKC model using different values of $\lambda$. One could think that this plot can be used to determine the optimal value of $\lambda$, making the hyperparameter choice entirely unsupervised. However, the validity of this approach depends on one important factor, namely the correspondence between minima of the clustering loss and minima of the companion objectives. This is explained by considering the following two cases:

- *Minima of companion objectives do not coincide with minima of clustering loss.* When this is the case, the optimization procedure will "pull" the point of convergence away from the minima of the clustering loss, which will result in an increase of the final clustering loss. Stronger companion objectives implies that the convergence point is pulled further away from the minimum of the clustering loss, and closer to the minimum of the companion objectives. Therefore, this case should result in higher clustering losses for higher values of $\lambda$. This increasing behavior makes the clustering loss an unsuitable measure of performance for the selection of $\lambda$.

- *Minima of companion objectives coincide with minima of clustering loss.* In this case, the point of convergence will be a minimum of both the clustering loss and the companion objectives. Thus, no "pulling" will be present, and no increasing trend should be observed. The clustering loss might therefore be used for selecting $\lambda$ in this case.

From Figure 34b it is not straightforward to determine which of the two cases that best fits the observed behavior. However, the large jump in DDC loss from $\lambda = 10^{-1}$ to $\lambda = 1$ indicates that the optimization procedure has been pulled away from a clustering-loss-minimum and towards a minimum of the companion objectives. Also note that the standard deviation is very large at $\lambda = 1$, meaning that companion objectives this strong resulted in a model which was unstable during training.

## 17.5. Effect of unsupervised companion objectives at individual layers

Table 7: *Resulting MNIST accuracies and DDC losses when different companion objectives are included in the model.*

| | | ACC | | DDC Loss | |
|---|---|---|---|---|---|
| $\lambda$ | Included terms | Best | Mean ± Sd. | Best | Mean ± Sd. |
| 0 | None | 0.91 | 0.77 ± 0.07 | 0.51 | 0.52 ± 0.01 |
| 0.01 | $\mathcal{L}_{\text{co}}^1$ | 0.80 | 0.79 ± 0.04 | 0.52 | 0.52 ± 0.00 |
| | $\mathcal{L}_{\text{co}}^2$ | 0.85 | 0.73 ± 0.07 | 0.51 | 0.53 ± 0.01 |
| | $\mathcal{L}_{\text{co}}^1 + \mathcal{L}_{\text{co}}^2$ | 0.94 | 0.75 ± 0.10 | 0.51 | 0.53 ± 0.01 |
| 1 | $\mathcal{L}_{\text{co}}^1$ | 0.94 | 0.73 ± 0.11 | 0.52 | 0.54 ± 0.01 |
| | $\mathcal{L}_{\text{co}}^2$ | 0.82 | 0.77 ± 0.05 | 0.79 | 0.80 ± 0.01 |
| | $\mathcal{L}_{\text{co}}^1 + \mathcal{L}_{\text{co}}^2$ | 0.82 | 0.74 ± 0.05 | 0.79 | 0.82 ± 0.02 |

This subsection describes an ablation study where the goal is to investigate the effect of the companion objectives on the different layers. The experiment includes the DDC base model, as well as the DTKC model, both trained on the MNIST dataset. For DTKC, the companion objectives were attached to layer 1 or 2, or both (see Table 3 and Figure 27). The experiments were repeated for $\lambda = 0.01$ and $\lambda = 1$. This was done to investigate the contributions of the different terms both at the "optimal" value, and at an "exaggerated" value.

Table 7 shows the resulting accuracies and DDC losses for the different model configurations. Perhaps the most prominent trend in this table is the increase in loss for the models with companion objectives at layer 2 with $\lambda = 1$. This is consistent with the findings of Section 17.4, where an increase in DDC loss was discovered at $\lambda = 1$, indicating that the model has put too much emphasis on minimizing the companion objective(s), instead of the DDC loss.

Another observation which can be made from Table 7 is that strong companion objectives at layer 1 is comparable to weak companion objectives at layers 1 and 2, and that these two configurations seem to give the best overall performance. The latter has a somewhat higher mean accuracy, and a lower standard deviation, but the differences are not large enough to provide conclusive evidence on the benefits of using one over the other. However, it is beneficial with respect to model design that the model including both companion objectives at the same strength, is among the best performing configurations. Unequal weighting or dropping terms entirely would significantly increase the model design complexity, which is already tricky due to the unsupervised nature of the clustering problem. An exhaustive study of these modifications is therefore beyond the scope of this thesis, and left as future work.

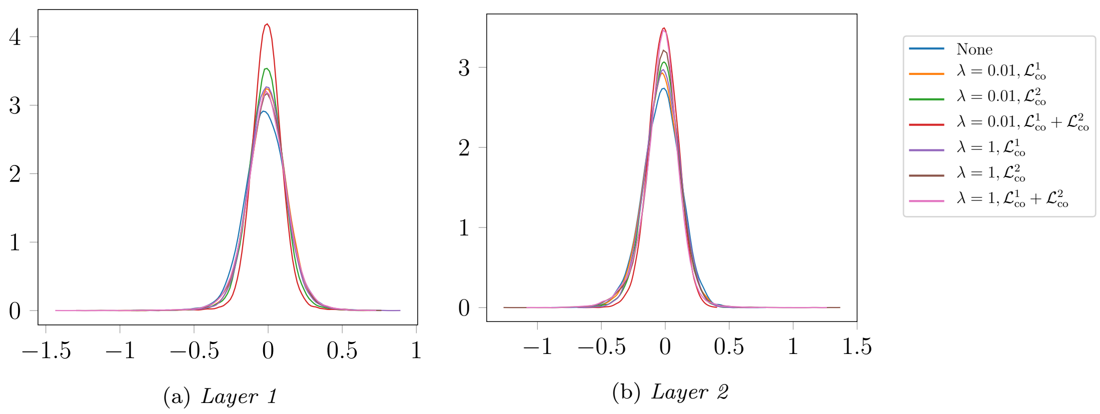


Figure 35: *Estimated distributions of learned parameters in the two convolutional layers for the different companion objective configurations. The estimates were computed using the Gaussian kernel density estimator with $\sigma = \sigma_{AMISE}$.*

To check whether the different model configurations has different effects on the parameters of the convolutional layers, kernel density estimators of the parameter-distributions were extracted for the run resulting in the lowest value of the total loss function. These are shown in Figure 35. From these plots we can make the following key observations:

- The configuration $\lambda = 0.01, \mathcal{L}_{\text{co}}^1 + \mathcal{L}_{\text{co}}^2$ has parameter values that are more compactly centered around zero than the other configurations for both layers, with the exception of $\lambda = 1, \mathcal{L}_{\text{co}}^1 + \mathcal{L}_{\text{co}}^2$ in layer 2.
- The base model $\lambda = 0$ has the widest parameter distribution in both layers.

These observations indicate that the unsupervised companion objectives can lead to smaller parameter values in the CNN. This is not particularly surprising as the companion objectives tend to penalize differences in kernels between layers, as long as the DDC loss is sufficiently low. Recall that, for DTKC:

$$\mathcal{L}_{\text{cluster}} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$$

where:

$$\mathcal{L}_1 = \frac{1}{\binom{k}{2}} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_j}{\sqrt{\boldsymbol{v}_i^T \boldsymbol{K} \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K} \boldsymbol{v}_j}},$$

where $\boldsymbol{K}$ denotes the kernel matrix extracted at the output of the first fully connected layer, and $\boldsymbol{v}_1, \dots, \boldsymbol{v}_k$ denotes the columns of the cluster assignment matrix $\boldsymbol{U}$. Furthermore, we have:

$$\mathcal{L}_{\text{co}}^m = \frac{1}{\binom{k}{2}} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\boldsymbol{v}_i^T \boldsymbol{K}^m \boldsymbol{v}_j}{\sqrt{\boldsymbol{v}_i^T \boldsymbol{K}^m \boldsymbol{v}_i \boldsymbol{v}_j^T \boldsymbol{K}^m \boldsymbol{v}_j}}.$$

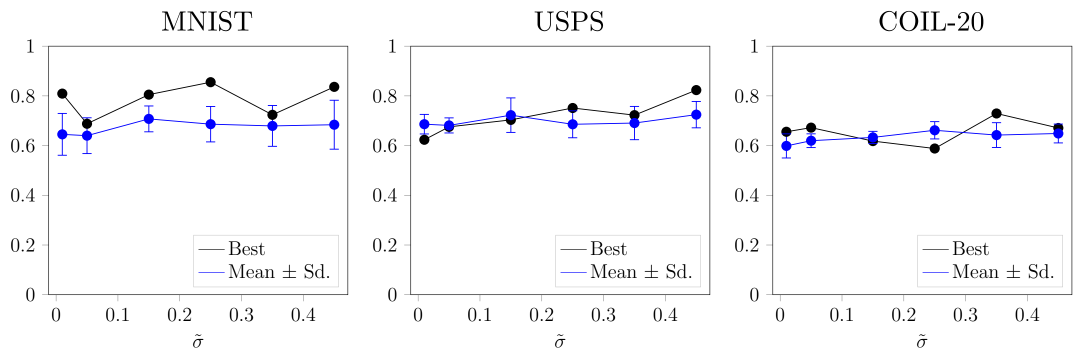
where $\boldsymbol{K}^m$ is the tensor kernel extracted at layer $m$. If we set $\boldsymbol{K} = \boldsymbol{K}^m$ we get $\mathcal{L}_1 = \mathcal{L}_{\text{co}}^m$, which means that similar kernels and low values of $\mathcal{L}_1$ will correspond to low values of $\mathcal{L}_{\text{co}}^m$. One way to obtain similar kernels is to make the mappings between the layers less complex, which can lead to smaller parameter values. This is because transformations similar to the identity would result in similar kernels, and such transformations are represented as sparse filters in convolutional layers (recall that the identity filter contains a single one surrounded by zeros).

## 17.6. Effect of the bandwidth parameter $\sigma$

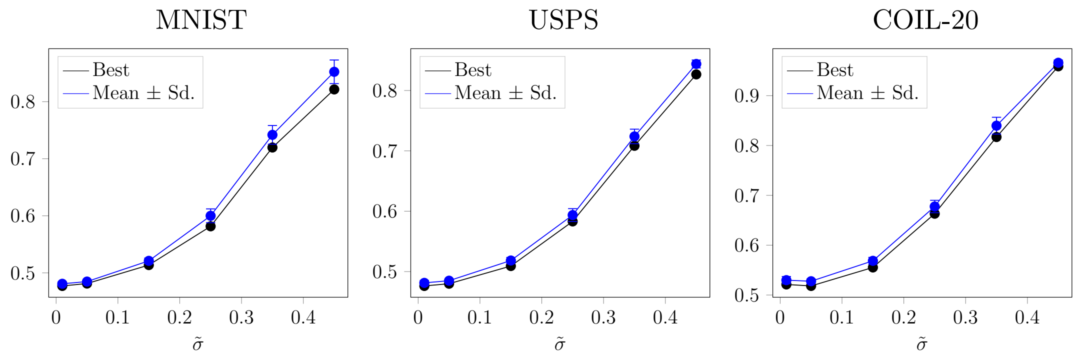
Recall that the bandwidth parameter $\sigma$ was chosen according to the "15 % of median distance" rule of thumb given in [100]. This rule of thumb has been successfully applied to other DDC-based models [20, 22] – which is why it was used for the experiments in this thesis as well. However, as was previously discussed in Section 11, the specification of $\sigma$ in the Gaussian kernel can be difficult, as well as critical for the performance of the resulting algorithm. The goal of this subsection is therefore to inspect the consequences of this choice, and more specifically, to investigate how sensitive the DTKC model is to variations in $\sigma$.

Note that the findings of this analysis were not used to specify the $\sigma$ parameter, in contrast to what was done in Section 17.4 for the $\lambda$ hyperparameter. There are two main reasons for this, the first being that introducing a similar procedure for $\sigma$ would result in a significant increase in the experimental complexity. The second reason is that for $\sigma$, we already have a rule of thumb which has been proven to work well with previous DDC-based models – which is not the case for the $\lambda$ hyperparameter.

To evaluate the effect of different $\sigma$ values on DTKC, the model was trained using variations of the aforementioned rule of thumb, on the MNIST, USPS, and



(a) *Accuracy*



(b) *DDC loss:* $\mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$

Figure 36: *Effect of the $\sigma$ hyperparameter on accuracy and DDC loss for the MNIST, USPS, and COIL-20 datasets.*

COIL-20 datasets. Specifically, for a given layer $l$, $\sigma^l$ was varied according to:

$$\sigma^l = \tilde{\sigma} \cdot \text{median}\left\{d_{ij}^l\right\}_{i,j=1}^n$$

where $\tilde{\sigma}$ is the multiplication factor[18], and $d_{ij}^l$ is the distance between output $i$ and output $j$ in layer $l$. Following the ordinary loss-function-computations, the tensor distance $d_{\text{tensor}}(\cdot,\cdot)$ was used for the $\sigma$ values in the companion objectives, while the Euclidean distance was used for $\sigma$ in the DDC loss function. Note that the same $\tilde{\sigma}$ was used for all layers, in order to reduce the complexity of the analysis.

The resulting accuracies and DDC losses are shown in Figure 36a and Figure 36b, respectively. The accuracy plots show that the DTKC is rather robust towards variations in the multiplication factor $\tilde{\sigma}$. Thus indicating that the potential loss or gain in performance is not particularly large when varying this hyperparameter. This is indeed an advantage due to the aforementioned difficulties of setting hyperparameters in unsupervised algorithms.

The loss curves in Figure 36b shows why the DDC loss function is unsuitable for specifying $\sigma$. The plots show a clear monotonically increasing trend, meaning

[18]Note that setting $\tilde{\sigma} = 0.15$ recovers the original rule of thumb.

that higher values of $\sigma$ results in increased DDC loss. The intuitive reason for this is that a larger $\sigma$ leads to larger values in the kernel matrix, resulting in wider and smoother kernel density estimates. Furthermore, recall that $\mathcal{L}_1$ penalizes overlapping and non-compact clusters, meaning that $\mathcal{L}_1$ will increase with the width of the density estimates. The loss function will therefore favor smaller values of $\sigma$, regardless of the resulting clustering performance.

Although DTKC did not prove to be especially sensitive to the choice of $\sigma$, its specification remains an important consideration for future work. Section 18.1 offers a broader discussion on the choice of $\sigma$ for tensor-based methods, and summarizes some thoughts on future work along those lines.

## 17.7. Visualization of importance

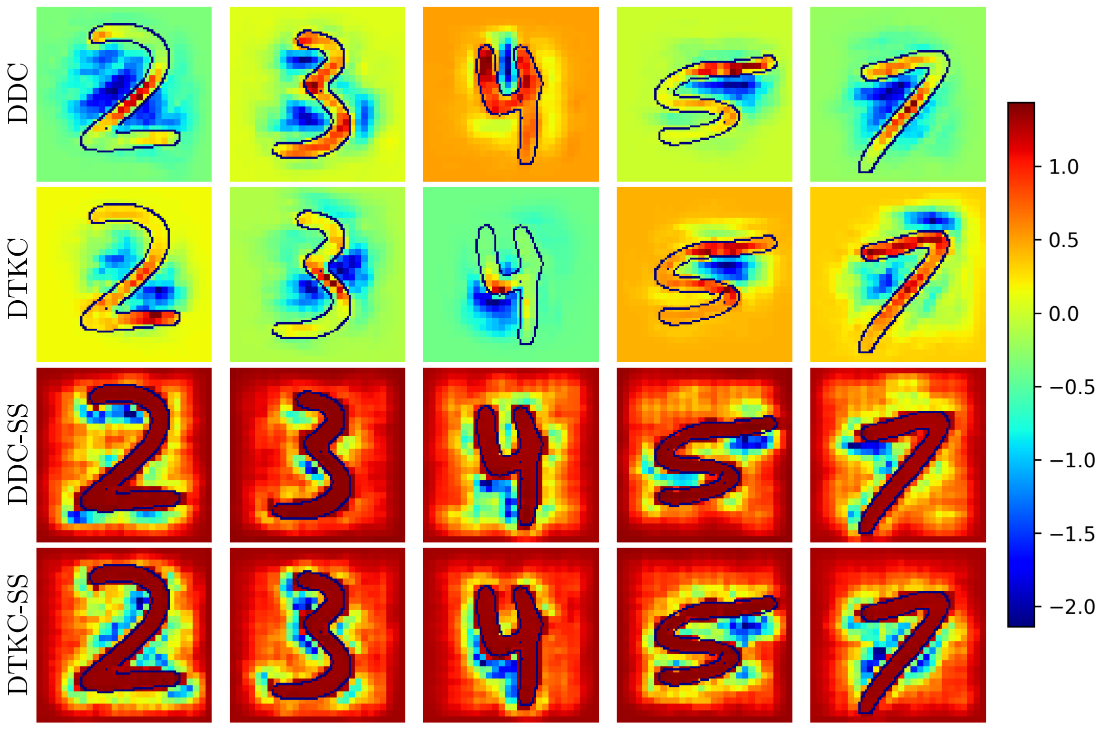


Figure 37: *Pixels in the input image important for the cluster assignment. The images were obtained using guided backpropagation [125].*

Another way to qualitatively assess the validity of the clusterings produced by the models, is to inspect which pixels in the input image that are important for the cluster assignment. These visualization techniques have been extensively used in supervised image classification [126, 125, 127], but not so much in unsupervised applications. Even though one can argue that the necessity for such visualization techniques is even greater in the unsupervised case, due to the potential ambiguity in the interpretation of the clusterings.

Guided backpropagation [125] is a recent technique for visualizing the importance of the input pixels with respect to an activation in a particular layer. Suppose we want to determine which pixels in the $i$-th input image $\boldsymbol{X}_i = [X_{i,xyc}]$, are most important to a the soft assignment $\alpha_{ij}$. In guided backpropagation this is done by computing an approximation of the gradient

$$\frac{\partial \alpha_{ij}}{\partial X_{i,xyc}}.$$

The approximated gradient differs from the actual gradient in that only positive values are kept when back-propagating through ReLU activations. This modification might seem arbitrary at first, but the reasoning is as follows: If an element of the input gradient to the ReLU activation is negative during the backward pass, this means that this particular element corresponds to a negative derivative with respect to the prediction, effectively "speaking against" that prediction. However, we are interested in explaining the prediction, and not the "lack" thereof, meaning that we should keep only positive values. Another argument is that the positive contributions of some gradients are "cancelled out" by the negative contributions of other gradients. This modification has been shown to vastly improve the resulting importance maps in supervised image classification [125].

Figure 37 shows the result of applying guided backpropagation to DDC, DDC-SS, DTKC, and DTKC-SS. The images show the gradient of the largest input to the softmax function in the last fully-connected layer, with respect to the input image. The reason for choosing the largest softmax input is that this value represents the predicted cluster membership of the input image. We want to inspect the importance of the assignment of digit $x$ to the cluster of $x$'s, as opposed to its assignment to a cluster corresponding to some other digits. Informally, one could say that we are explaining why the 1 was clustered as a 1, and not why it was not clustered as a 2.

The resulting importance plots are as expected for DDC and DTKC. In these plots we see that the system focuses on characteristic parts of the respective digits, such as the horizontal bar in the 5. The plots also show that the system assigns negative importance to areas surrounding the digit, meaning that if these were part of the digit, it would speak against the current prediction.

The results are quite different when it comes to the models pre-trained using self-supervised learning, namely DDC-SS and DTKC-SS. In contrast to the randomly initialized models, these models seem to put very much emphasis on the borders of the images. This might seem puzzling at first, as the surrounding parts of the images are homogeneous and mostly equal to zero. However, recall that during the self-supervised pre-training, the images are randomly divided into patches, whose relative positions are to be determined by the system. It is reasonable to assume that the edge of a given patch is important for determining its relative position with respect to some other patch, which means that the network – and especially the convolutionalized fully-connected layer – learns to focus on these edges. As these features are not immediately useful to the clustering system during the fine-tuning process, one could further conjecture that the importance of the edges would decrease with a longer fine-tuning process.

## 17.8. Classification-based companion objectives

The preceding results show that the companion objectives constructed from tensor kernels and Cauchy-Schwarz divergence helps guide the system towards more separated embeddings, and possibly better performance. It is therefore reasonable to ask the question of whether there exist other formulations of the companion objectives which yield similar results. One option – which is closely related to Deeply Supervised Nets [34] – is to construct the companion objectives based on an ordinary supervised classification system. Clearly, these “companion classifiers” will need some form of ground truth labels for training, but what if these could be obtained as part of the network?

Suppose we let $\boldsymbol{Y}^l_1, \dots, \boldsymbol{Y}^l_n$ be the tensorial outputs of layer $l$ for the current batch, and that we flatten these to form the vectors $\boldsymbol{y}^l_1, \dots, \boldsymbol{y}^l_n$. These can then be passed to a fully-connected layer with a softmax activation function:

$$\tilde{\boldsymbol{u}}^l_i = \text{softmax}(\boldsymbol{W}^l \boldsymbol{y}^l_i + \boldsymbol{b}^l)$$

where $(\boldsymbol{W}^l, \boldsymbol{b}^l)$ denote the parameters of the fully-connected layer. The outputs $\tilde{\boldsymbol{u}}^l_1, \dots, \tilde{\boldsymbol{u}}^l_n$ can now be compared to the outputs of the final layer, in order to optimize $(\boldsymbol{W}^l, \boldsymbol{b}^l)$. The $l$-th companion objective can thus be constructed based on e.g. the cross-entropy loss function [1]:

$$\mathcal{L}^l_{\text{co,clf}} = -\sum_{i=1}^{n}\sum_{j=1}^{k}\left( u_{ij} \ln \frac{\tilde{u}^l_{ij}}{u_{ij}} + (1 - u_{ij}) \ln \frac{1 - \tilde{u}^l_{ij}}{1 - u_{ij}} \right)$$

where $\boldsymbol{u}^l_i = [u^l_{i1}, \dots, u^l_{ik}]$ denotes the predicted cluster membership vector for the $i$-th input. This companion objective can then be used in place of $\mathcal{L}^l_{\text{co}}$ in the total loss function, which gives:

$$\mathcal{L} = \mathcal{L}_{\text{cluster}} + \lambda \sum_{i=1}^{L-1} \mathcal{L}^l_{\text{co,clf}} \tag{41}$$

which can be minimized using ordinary gradient descent. This setup essentially attaches a linear logistic regression classifier [3] to each of the layers, meaning that the companion objectives will try to enforce linear separability between clusters in earlier layers.

### Experiment setup

To investigate the classifier-based companion objective formulation, the loss function in Eq. (41) was used to train a model analogous to DTKC in the previously described experiments, which is referred to as DTKC-SM (DTKC-SoftMax). The network layout and all other training components were left exactly as described in Section 15.3, to obtain accurate comparisons. The DTKC-SM model was trained and evaluated on a representative subset of the datasets, namely MNIST, Fashion-MNIST and COIL-100.

### Results

Table 8: *Resulting accuracy and NMI for the models with classifier-based companion objectives. The relevant results from the previous experiments are included for easier comparison.*

MNIST

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| DDC | 0.91 | **0.77** | 0.07 | 0.91 | 0.83 | 0.73 | 0.06 | 0.83 |
| DTKC | **0.94** | **0.77** | 0.08 | **0.94** | **0.88** | **0.74** | 0.07 | **0.88** |
| DTKC-SM | 0.87 | **0.77** | 0.08 | 0.87 | 0.84 | **0.74** | 0.07 | 0.84 |

Fashion-MNIST

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| DDC | 0.58 | 0.55 | 0.06 | **0.65** | 0.52 | 0.49 | 0.04 | **0.56** |
| DTKC | **0.63** | **0.56** | 0.04 | 0.63 | **0.55** | **0.5** | 0.03 | **0.56** |
| DTKC-SM | 0.56 | 0.54 | 0.05 | 0.62 | 0.51 | 0.49 | 0.03 | 0.54 |

COIL-100

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| DDC | 0.61 | 0.59 | 0.02 | 0.63 | 0.83 | **0.83** | 0.01 | 0.84 |
| DTKC | **0.64** | **0.6** | 0.02 | **0.65** | **0.85** | **0.83** | 0.01 | **0.85** |
| DTKC-SM | 0.59 | **0.6** | 0.02 | 0.62 | 0.83 | **0.83** | 0.01 | **0.85** |

The results in Table 8 show that the models using softmax-based companion objectives perform slightly worse than both DDC and DTKC on all datasets in terms of “Best” performance, whereas the “Mean” performance is similar for all models across all three datasts.

To more closely inspect the effect of the softmax-based companion objectives, the analysis in Section 17.3 was repeated for the DTKC-SM model. Figure 38 shows $\boldsymbol{K}^2$ for the DDC, DTKC and DTKC-SM models resulting in the lowest loss function[19]. From these plots it is apparent that, when compared to the DDC base model, the intermediate representations from the second layer are slightly more compact and well separated for the DTKC-SM model. On the other hand, when

[19]Plots of $\boldsymbol{K}^1$ are omitted as they were all similar.

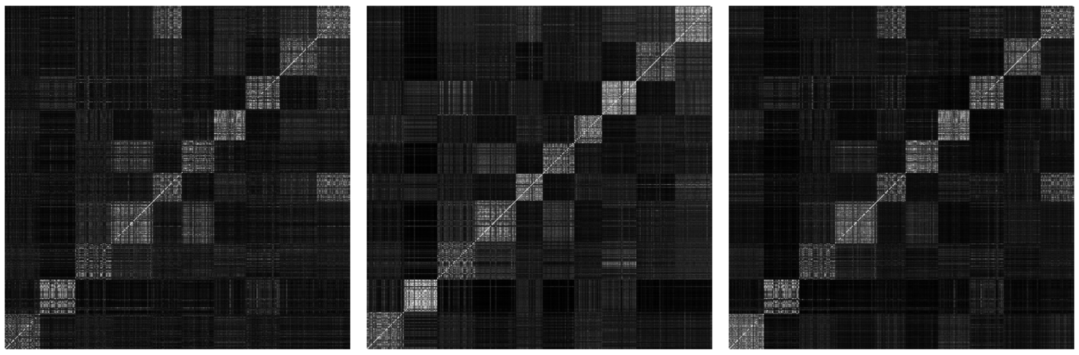

(a) *DDC* (b) *DTKC* (c) *DTKC-SM*

Figure 38: *Plots of $\boldsymbol{K}^2$ for the DTKC and DTKC-SM models resulting in the lowest values of the respective loss functions. The kernels were extracted from a random subset of* 1024 *MNIST samples.*

comparing DTKC-SM and DTKC, there is a noticeable difference in separability and compactness in favor of DTKC. This can intuitively be described by the fact that the logistic regression classifier favors linearly separable clusters, but not necessarily the compactness of these clusters.

It should also be noted that this analysis is somewhat biased towards DTKC as the tensor kernel framework is used to compute $\boldsymbol{K}^2$, which is the same kernel used in DTKC's companion objectives. Nevertheless, if one assumes that the tensor kernel is capable of correctly describing the geometry of the output space of layer 2, the analysis still holds with respect to between-cluster-separability and within-cluster-compactness in this space.

A significant benefit of using the softmax companion objectives instead of the companion objectives based on tensor kernels, is that it leads to a noticeable decrease in training time. Recall that DTKC requires the computation of several eigendecompositions at each training step, which significantly increases the training time. DTKC-SM on the other hand simply requires the computation of a matrix product, followed by a softmax nonlinearity – a computation which is much faster, especially when done with specialized deep learning frameworks such as `TensorFlow`. The extra parameters $(\boldsymbol{W}^l, \boldsymbol{b}^l)$ introduced by the softmax classifiers causes this increase in training speed to come at the cost of an increase in the model's memory footprint during training. This is not a problem for inference however, as the final prediction is not influenced by the softmax classifiers, which means that they can be discarded when the training process is finished.

## 17.9. Experiments with sequential data

Up until this point, the main focus of this part has been on image clustering. However, in [22] it was shown that DDC provides a promising framework for the clustering of sequential data as well. By replacing the CNN with an RNN it was shown that the resulting Recurrent Deep Divergence-based Clustering (RDDC) was able to outperform classical methods when clustering sequences. It is therefore natural to ask if the framework proposed in this thesis can help improve the performance of RDDC as well, by introducing stronger supervision in earlier layers of the RNN.

Recall that RNNs can be seen as layer-wise models, similarly to CNNs. The translation of the unsupervised companion objectives from CNNs to RNNs is therefore relatively straightforward. Suppose we have an input sequence $\boldsymbol{x}_i = \boldsymbol{x}_{i,1}, \dots, \boldsymbol{x}_{i,T}$ where $T$ is the length of the sequence. Recall from Section 9.3.3 that at layer $l$ we get the output sequence:

$$\boldsymbol{h}^l_{i,t} = f_{\boldsymbol{\theta}_l}(\boldsymbol{h}^l_{i,t-1}, \boldsymbol{h}^{l-1}_{i,t}),\ t = 1, \dots, T.$$

We can then use the last hidden states $\boldsymbol{h}^l_{1,T}, \dots, \boldsymbol{h}^l_{n,T}$ of layer $l$ to compute the companion objective. Note that in contrast to the CNN case, this will result in the companion objectives receiving rank-1 tensors (vectors) instead of rank-3 tensors. The regular Gaussian kernel was therefore used to compute the elements of the kernel matrix:

$$\boldsymbol{K}^l = [\kappa^l_{ij}], \quad \kappa^l_{ij} = \exp\left(-\frac{||\boldsymbol{h}^l_{i,T} - \boldsymbol{h}^l_{j,T}||^2}{2\sigma^2}\right)$$

### Models

The sequential experiments were performed with two different models: RDDC as described in [22], and RDTKC (Recurrent Deep Tensor Kernel Clustering). The latter refers to a DTKC-based model where the CNN has been replaced with an RNN. Following [22] both models used a two-layer bidirectional gated recurrent unit [75] (also see Section 9.3.5.2), together with the DDC clustering module. The dimensionality of the hidden states for each layer was set to 32, also following [22]. Batch size, optimizer, gradient clipping, epochs, runs, and bandwidth all follow the configuration specified in Section 15.3. The companion objectives in RDTKC were constructed as described above.

$k$-means, spectral clustering, and average-link hierarchical clustering were used as benchmark methods. Since these require vectorial inputs, the sequences were first zero-padded to have length equal to the maximum length in the dataset, and then reshaped into vectors.

## Datasets

The datasets used for evaluation are:

- *Character Trajectories* [128]. Sequences of vectors on the form $(x, y,\text{pen tip force})$, where each sequence corresponds to the trajectory of a hand-written letter in the English alphabet. Following [22] a subset consisting of the first 6 characters was selected. However, since almost all algorithms produced perfect clusterings on this subset, the set was augmented with the next 4 characters as well, resulting in a total of 10 characters.

- *Arabic Digits* [128]. Mel-frequency cepstrum coefficients [1] extracted from recordings of spoken Arabic digits. All 10 digits were used.

These datasets were chosen as they represent clustering or classification challenges of suitable complexity [22, 129, 130]. These are also openly accessible and well-known benchmark datasets, allowing for easier comparison with the literature. See Table 9 for a summary of relevant attributes.

Table 9: *Summary of attributes for the sequential datasets. "Lengths" denotes the range of sequence lengths contained in the datasets.* dim, $n$, *and* $k$ *represent the dimensionality of each sequence-element, the number of sequences, and the number of clusters, respectively.*

| Name | Lengths | dim | $n$ | $k$ | Contents |
|---|---|---|---|---|---|
| Character Trajectories | $[109, 198]$ | 3 | 1491 | 10 | Hand-written letters |
| Arabic Digits | $[4, 93]$ | 13 | 8800 | 10 | Spoken digits |

## Results

Table 10: *Results for the experiments with sequential data.*

Character Trajectories

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| $k$-means | **0.84** | **0.8** | 0.05 | **0.86** | **0.86** | **0.86** | 0.03 | **0.91** |
| Hierarchical | 0.51 | 0.51 | 0.0 | 0.51 | 0.73 | 0.73 | 0.0 | 0.73 |
| Spectral | 0.79 | 0.78 | 0.0 | 0.79 | **0.86** | 0.85 | 0.0 | 0.86 |
| RDDC | 0.64 | 0.58 | 0.09 | 0.71 | 0.73 | 0.68 | 0.09 | 0.81 |
| RDTKC | 0.8 | 0.65 | 0.08 | 0.8 | 0.83 | 0.74 | 0.06 | 0.83 |

Arabic Digits

| | ACC | | | | NMI | | | |
|---|---|---|---|---|---|---|---|---|
| Model | Best | Mean | Sd. | Max | Best | Mean | Sd. | Max |
| $k$-means | 0.71 | **0.67** | 0.05 | 0.72 | 0.6 | 0.6 | 0.02 | 0.61 |
| Hierarchical | 0.1 | 0.1 | 0.0 | 0.1 | 0.03 | 0.03 | 0.0 | 0.03 |
| Spectral | 0.64 | 0.65 | 0.02 | 0.68 | 0.59 | 0.58 | 0.01 | 0.59 |
| RDDC | 0.46 | 0.55 | 0.07 | 0.68 | 0.63 | **0.62** | 0.04 | 0.73 |
| RDTKC | **0.76** | 0.56 | 0.09 | **0.76** | **0.74** | 0.61 | 0.07 | **0.76** |

The results for the sequential experiments are given in Table 10. In contrast to the image clustering results, these show that the deep models (i.e. RDDC and RDTKC) are not capable of consistently outperforming the classical models. Moreover, the results for RDDC and RDTKC also show relatively low mean-accuracies and large standard deviations.

Comparing RDDC's results on the Arabic Digits dataset in Table 10 to [22] reveals that there is a relatively large gap between the "Best" accuracies. Since the implementation used here follows the implementation used in [22], this discrepancy is most likely due to the randomness of the RNN initialization.

The preceding observations strengthen the hypothesis about the necessity of a more consistent cluster structure. Moreover, the performance of $k$-means reveals a great deal about the geometry of the Character Trajectories input space. Recall that $k$-means uses the squared Euclidean distance as its dissimilarity measure, and thus, favors compact, hyperspherical clusters. This, together with $k$-means' accuracy, tells us that the character-classes are indeed compact and hyperspherical clusters in the input space. The poor performance of RDDC compared to $k$-means indicates that the RNN is not capable of preserving this Euclidean geometrical structure. The companion objectives of RDTKC seems to somewhat alleviate this problem, resulting in better performance than RDDC. However, its performance is still worse than $k$-means for the Character Trajectories dataset.

## Discussion

In summary, the results seem to show that using an RNN instead of a CNN makes the DDC clustering framework more prone to the aforementioned geometrical problems. In future work, it would therefore be interesting to investigate the use of one-dimensional convolutional architectures instead of recurrent networks to process the input sequences. However, if one insists on using an RNN, other ways to incorporate the geometric constraints could be explored. One option is to use more of the hidden states produced by the layers, instead of only using the last hidden state. For instance, all the hidden states in a layer could be stacked, resulting in the rank-2 tensor:

$$\boldsymbol{Y}_i^l = \begin{bmatrix} \boldsymbol{h}_{i,1}^l & \boldsymbol{h}_{i,2}^l & \cdots & \boldsymbol{h}_{i,T}^l \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}.$$

For a dataset of sequences $\boldsymbol{x}_1, \dots \boldsymbol{x}_n$, the corresponding $\boldsymbol{Y}_1^l, \dots, \boldsymbol{Y}_n^l$ could then be used to compute the kernel matrix $\boldsymbol{K}^l$ for layer $l$ using the tensor kernel framework.

# 18. Discussion and future work

## 18.1. Unsupervised companion objectives and tensor kernels

The generalized Cauchy-Schwarz divergence coupled with the idea of companion objectives is perhaps the most important contribution made in this thesis. Together, these two concepts play an essential role in DTKC – as they incorporate the much needed geometrical constraints on the individual layers of the deep neural network used by the model.

In the derivation of the unsupervised companion objectives we made one particularly important choice, namely to base the kernel density estimation on the tensor kernel framework proposed by Signoretto [35]. By using these kernels we assumed that the summed Grassmann distance between matricizations is a "good" distance measure, in the sense that it reflects the cluster structure we are seeking. However, since these kernels are computed at the output of layers in the network, it is not immediately straightforward to evaluate the validity of this assumption. In future work it would therefore be interesting to more thoroughly investigate this assumption, both from theoretical and experimental standpoints. One could also construct other tensor kernels based on e.g. domain knowledge, weak supervision, or semi-supervision.

Another important choice which was made in the construction of the unsupervised companion objectives, was to use the CS divergence as a measure of dissimilarity between the density functions describing the clusters. The CS divergence is only one of many different divergence measures in the literature on information theory, with other examples including the Kullback-Leibler (KL) divergence [131] and the more general Rényi divergence [132]. From the perspective of loss function minimization, there are no immediate problems with using either of these two divergences in place of the CS divergence. One reason for favoring the CS divergence however, is that it takes on a quite favorable form when expressed through the kernel density estimator. Recall from Section 11.1.2 that the KDE-based estimator for the CS divergence is a function of the kernel matrix $\boldsymbol{K}$ only, allowing for more straightforward loss computations. Moreover, the particular form of the CS divergence estimator also allows for the RKHS-based interpretation also outlined in Section 11.1.2.

Furthermore, it is important not to neglect the potential influence of the bandwidth parameter $\sigma$ in the kernel density estimator. Much research has been done on data-driven selection of this parameter, both in the univariate case [99, 133, 134, 135], and the multivariate vectorial case [99, 136]. The domain of density estimation for tensor data will most likely also benefit from similar advancements in automatic bandwidth selection. It is possible that much of the statistical intuition and theory that underlies the univariate and vectorial methods, is applicable in the more general case of bandwidth selection for tensor valued density estimators. However, the developments of new methods for selecting these bandwidths should take the (possibly non-Euclidean) geometrical structure into account, to correctly model the underlying data distribution.

It is a well known result from nonparametric statistics that kernel density estimation becomes difficult in higher dimensions [99], requiring massive sample sizes to obtain accurate estimates. One might therefore think that this result somewhat invalidates the computation of the unsupervised companion losses, as well as DDC's loss function, since these are computed for each mini-batch. However, note that these losses are concerned with density estimates that roughly reflect the overall shape and location of the respective clusters, and not necessarily the best possible modeling of the data distribution. The results obtained by DTKC, DDC [20], RDDC [22], as well as other classical information-theoretic approaches [137, 138, 139], indicate that these rougher estimates are sufficient for the clustering objective, regardless of their questionable accuracy.

Section 13 states that the unsupervised companion objectives can be coupled with any clustering algorithm using some form of deep neural network. This degree of generalizability is an important advantage of the unsupervised companion objectives, since new deep neural network architectures are likely to be introduced in the future. We are already seeing examples of this development with the introduction of Graph Convolutional Networks [140] and Transformers [141]. The tensor kernel framework [35] and the proposed CS divergence for tensors, are also general concepts that can be applied outside the field of deep clustering. One potential application is the promising work on understanding deep neural networks through information theory [142, 143]. Preliminary work by Yu et al. [144] on applying these methods to CNNs has focused on the naïve kernel, but it is possible that these methods can benefit from the tensor kernel utilized in this thesis.

The idea of introducing loss functions at the output of individual layers also opens up for an interesting pre-training strategy. Recall from Section 9.4 that the stacked autoencoder is built by training the network layer by layer, instead of training all layers at once. The unsupervised companion objectives could be used in a similar manner, where we initially only consider the first layer, and train it to minimize the first companion objective. Then we move on to the second layer, and train only the parameters of this layer to minimize the second companion objective. This process can be repeated until the desired number of layers is reached. Finally, we can fine-tune all layers and the clustering module by minimizing all the companion objectives, combined with the clustering loss. This could make it easier for the individual layers to learn transformations that enhance the cluster structure. However, the training of individual layers requires a clustering module attached to each layer, leading to an increase in memory consumption during training. As always, this is a tradeoff which has to be considered by the end-user.

## 18.2. Self-supervised pre-training

From the preceding results it is not immediately apparent whether or not self-supervised pre-training will result in increased clustering performance. Thus, when comparing to a randomly-initialized model, the tradeoff between added computational cost and potentially improved initialization is one that has to be considered based on the current task at hand. However, when compared to autoencoder-based pre-training, self-supervised learning offers an alternative pre-training strategy with a potentially significant reduction in the number of pre-training parameters. This is because the autoencoder's decoder might introduce a large number of extra parameters, depending on its architecture.

To reduce the cost of the pre-training step, one could experiment with setups where one pre-training run is used to initialize several fine-tuning runs. The extreme case being one single pre-training run for all fine-tuning runs. A concrete example of this strategy could be the following: Suppose our computational capacity limits us to 20 runs. Then, we could pre-train 4 times, and then perform 5 fine-tuning for each of the 4 pre-training runs. This would result in 20 runs in total, but we've effectively reduced the pre-training time by a factor of 5.

It is also possible to expand upon this line of thinking by including fine-tuning runs from random initializations into the allowed number of runs. For instance, instead of pre-training 4 times, we could pre-train 3 times, and then perform 5 fine-tuning runs for each of the now 3 pre-training runs. We could then do 5 runs from a random initialization, again resulting in a total of 20 runs. Diagrams illustrating these different strategies are shown in Figure 39.

| | | |
|---|---|---|
| 1 | pre-train | fine-tune |
| 2 | pre-train | fine-tune |
| 3 | pre-train | fine-tune |
| 4 | pre-train | fine-tune |
| 5 | pre-train | fine-tune |
| 6 | pre-train | fine-tune |
| 7 | pre-train | fine-tune |
| 8 | pre-train | fine-tune |
| 9 | pre-train | fine-tune |
| 10 | pre-train | fine-tune |
| 11 | pre-train | fine-tune |
| 12 | pre-train | fine-tune |
| 13 | pre-train | fine-tune |
| 14 | pre-train | fine-tune |
| 15 | pre-train | fine-tune |
| 16 | pre-train | fine-tune |
| 17 | pre-train | fine-tune |
| 18 | pre-train | fine-tune |
| 19 | pre-train | fine-tune |
| 20 | pre-train | fine-tune |

| | | |
|---|---|---|
| 1 | | fine-tune |
| 2 | | fine-tune |
| 3 | pre-train | fine-tune |
| 4 | | fine-tune |
| 5 | | fine-tune |
| 6 | | fine-tune |
| 7 | | fine-tune |
| 8 | pre-train | fine-tune |
| 9 | | fine-tune |
| 10 | | fine-tune |
| 11 | | fine-tune |
| 12 | | fine-tune |
| 13 | pre-train | fine-tune |
| 14 | | fine-tune |
| 15 | | fine-tune |
| 16 | | fine-tune |
| 17 | | fine-tune |
| 18 | pre-train | fine-tune |
| 19 | | fine-tune |
| 20 | | fine-tune |

| | | |
|---|---|---|
| 1 | | fine-tune |
| 2 | | fine-tune |
| 3 | pre-train | fine-tune |
| 4 | | fine-tune |
| 5 | | fine-tune |
| 6 | | fine-tune |
| 7 | | fine-tune |
| 8 | pre-train | fine-tune |
| 9 | | fine-tune |
| 10 | | fine-tune |
| 11 | | fine-tune |
| 12 | | fine-tune |
| 13 | pre-train | fine-tune |
| 14 | | fine-tune |
| 15 | | fine-tune |
| 16 | fine-tune | |
| 17 | fine-tune | |
| 18 | fine-tune | |
| 19 | fine-tune | |
| 20 | fine-tune | |

Figure 39: *Suggested pre-training and fine-tuning strategies for a total of* 20 *runs. The leftmost strategy was used for the experiments in this thesis, which is also the strategy with the highest computational cost.*

Another factor to consider is that the pre-training step might also require some careful configuration of its hyperparameters. Doersch et al. [36] argue that sufficient jitter and patch-spacing has to be applied in order for the network to learn features which reflect the objects present in the images. Furthermore, they observed that failure to meet these conditions led the network to only focus on pattern similarities along the image-edges. This is also consistent with the importance maps shown in Section 17.7, indicating that this "continuation of patterns" might be the case for the self-supervised network pre-trained on the MNIST dataset. However, due to the low resolution of the images considered in this thesis, increasing the jitter would require the patches to shrink accordingly. Although increased jitter was not further investigated here, it is possible that upsampling methods could be used to alleviate the problem of low resolutions.

Lastly, it is possible that self-supervised pre-training proves to be more beneficial for the clustering of images with more complex scenery than those used in this thesis. The results obtained by Doersch et al. [36] for ImageNet [109] and Pascal VOC [110], indicate that the system is capable of capturing visual similarity across images in these more diverse datasets.

## 18.3. Quantifying uncertainty for the "Best" statistic

The quantitative analyses done in the different subsections of Section 17 were – to a large degree – based on the "Best" statistic extracted from the 20 runs. But how closely does this *really* reflect the expected performance of the model? The answer to this depends on which of the two following situations we find ourselves in when trying to solve the problem at hand:

1. We have the computational resources to train the model from 20 different initializations, and then select the one resulting in the lowest loss value.

2. We have the computational resources to train the model *once*, and have to make due with the model we get from that single run.

Clearly, if we are in situation 2, the "Best" statistic will give an overly optimistic estimate of the true performance. In this situation, it would therefore be more accurate to consider the "Mean" statistic. Additionally, the "Sd." statistic provides us with an estimate of the uncertainty of the resulting performance in this situation.

On the other hand, if we actually have the resources to train the model 20 times – and thus find ourselves in situation 1 – we will get a more accurate point estimate from the "Best" statistic. But what if we want to say something about the uncertainty of this point estimate? Since the analytic computation of an estimator for this uncertainty can be somewhat tricky due to the complex distribution of the loss function, we can resolve to simulation techniques instead. *Bootstrapping* [145, 99] is one widely known such simulation technique, where the original sample is resampled a number of times to create a set of bootstrap samples. Each bootstrap sample can then be used to compute a value of the statistic – resulting in several values for the statistic in total. We can then compute e.g. the standard deviation based on these values. Suppose we let

$$\boldsymbol{R}_1 = [L_1, A_1], \ldots, \boldsymbol{R}_r = [L_r, A_r]$$

be vectors containing "loss" and "ACC" from the $r$ different training runs. We are then interested in estimating the standard deviation of the "Best" statistic:

$$\beta = \text{Best}(\boldsymbol{R}_1, \ldots, \boldsymbol{R}_r) = (A_i \;\text{ s.t. }\; L_i = \min\{L_1, \ldots, L_r\})$$

whose distribution is unknown. To estimate this standard deviation, we first draw the bootstrap samples:

$$\boldsymbol{R}^*_{b1}, \boldsymbol{R}^*_{b2}, \ldots, \boldsymbol{R}^*_{br}, \; b = 1, \ldots, B$$

where $B$ is the total number of bootstrap samples, and:

$$\boldsymbol{R}^*_{bi} \sim \mathbb{U}(\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_r\}), \; b = 1, \ldots, B, \; i = 1, \ldots, r$$

where $\mathbb{U}(\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_r\})$ denotes the discrete uniform distribution on the original sample. We then compute the statistic for each bootstrap sample:

$$\beta^*_b = \text{Best}(\boldsymbol{R}^*_{b1}, \ldots \boldsymbol{R}^*_{br}), \; b = 1, \ldots, B$$

and get the bootstrap estimators for "Best" and its standard deviation as:

$$\beta^* = \frac{1}{B}\sum_{b=1}^{B} \beta^*_b, \qquad \boldsymbol{\sigma}^*_\beta = \sqrt{\sum_{b=1}^{B} \frac{(\beta^*_b - \beta^*)^2}{B-1}}. \tag{42}$$

These are estimators of the expected "Best" performance, and its standard deviation.

Table 11 shows the resulting estimates computed for $B = 10000$ bootstrap samples. From these results we can make the following key observations:

- The expected accuracy is lower than the overall "Best", which can be found in Tables 4 and 5. This is because the overall "Best" is the highest observed value in the original sample, meaning that the bootstrap samples which does not include this value will pull the overall mean downwards[20].
- The deep models outperform the classical models for almost all datasets. This is consistent with the results in Tables 4 and 5.
- DEC has a noticeably lower standard deviation than the other deep models. This can be either due to the autoencoder initialization, or that DEC's fine-tuning process behaves more consistently across runs. The latter can be caused by a "nicer" loss functions with fewer local optima to get stuck in.

[20]The asymptotic probability ($r \to \infty$) of the overall "Best" being included in a bootstrap sample is $1 - e^{-1} \approx 0.632$ [3].

Table 11: *Estimates of expected "Best" accuracies ± standard deviation ($\beta^* \pm \sigma^*_\beta$). The estimates were computed based on the estimators in Eq. (42). The highest means are highlighted in bold.*

| | MNIST | USPS | SVHN | F-MNIST | COIL-20 | COIL-100 | UMist |
|---|---|---|---|---|---|---|---|
| $k$-means | 0.51 ± 0.0 | 0.67 ± 0.0 | 0.13 ± 0.0 | 0.5 ± 0.03 | 0.68 ± 0.02 | **0.62** ± 0.0 | 0.41 ± 0.02 |
| HC | 0.21 ± 0.0 | 0.22 ± 0.0 | **0.19** ± 0.0 | 0.1 ± 0.0 | 0.35 ± 0.0 | 0.23 ± 0.0 | **0.46** ± 0.0 |
| DEC | 0.87 ± 0.03 | 0.76 ± 0.0 | 0.18 ± 0.01 | 0.55 ± 0.0 | 0.65 ± 0.04 | 0.6 ± 0.0 | 0.09 ± 0.01 |
| DDC | 0.88 ± 0.04 | **0.77** ± 0.05 | 0.15 ± 0.0 | 0.6 ± 0.02 | 0.67 ± 0.02 | 0.61 ± 0.01 | 0.33 ± 0.03 |
| DDC-SS | 0.88 ± 0.02 | 0.72 ± 0.01 | **0.19** ± 0.01 | 0.56 ± 0.02 | **0.7** ± 0.02 | 0.51 ± 0.02 | 0.35 ± 0.01 |
| DTKC | **0.92** ± 0.04 | **0.77** ± 0.02 | 0.18 ± 0.01 | **0.61** ± 0.02 | 0.63 ± 0.01 | **0.62** ± 0.02 | 0.32 ± 0.04 |
| DTKC-SS | 0.86 ± 0.01 | 0.71 ± 0.02 | 0.14 ± 0.0 | **0.61** ± 0.06 | **0.7** ± 0.02 | 0.51 ± 0.0 | 0.35 ± 0.01 |

- DTKC and DTKC-SS outperform, or perform comparable to, the other deep models on all datasets, indicating that the performance of the proposed models is at least as good as DDC and DEC. Note that the benchmark models whose results were extracted from the literature, are not included here. This is because it was not possible to design and execute a comparable bootstrap when only the point estimates (or means) were reported for these models.

Lastly, and perhaps most importantly, the values of the estimated standard deviations are so large that they should be taken into consideration when evaluating the results. The key advantage of estimating uncertainty in this way, is that the bootstrap approach requires no extra training of models, making it very cheap compared to running multiple instances of the 20-run-setup. Bootstrapping should therefore be strongly considered as an option to quantify uncertainty when using the "Best" statistic based on multiple runs – especially when the alternative is to report the point estimate only.

## 18.4. Specifying the number of clusters

The number of clusters is an important parameter for many clustering algorithms, including DTKC. Although we have assumed the number of clusters to be a known quantity, this might not be the case in real-world applications – especially those designed for exploratory purposes.

The literature on determining the number of clusters seems to focus on mainly two different approaches: (i) Finding the number of clusters in the dataset before training the model [52, 146], and (ii) training several model instances with different numbers of clusters, and then selecting the best of these [147, 148]. Most of the methods within both these approaches rely heavily on the specification of a similarity (or dissimilarity) measure in the input space. As was previously discussed in Section 7, this choice can be highly nontrivial for more complex datatypes, such as digital images and time series.

For approach (ii), the requirement of a similarity measure could be bypassed by considering the loss function of the clustering algorithm, instead of the commonly used internal CVIs. However, this can also be problematic if the loss function itself is biased towards either small or large numbers of clusters. For the deep clustering methods, an alternative approach would be to keep using internal CVIs, but to compute these for the learned representations (denoted $\boldsymbol{z}$ in Section 10) instead of the input data. The specification of a similarity measure in this space is less challenging, as it tends to be an integral part of the subsequent clustering module [15, 20]. Sadly, this approach is not problem free either. Different models will result in different embeddings, which can negatively impact the validity of comparing the resulting CVIs. Thus, the determination of the number of clusters in complex datasets remains as an important direction for future work.

# Part VI / Conclusion

The purpose of this thesis has been to develop a new deep clustering algorithm for image clustering, as an answer to some of the open research questions in the field of deep clustering. The proposed DTKC employs unsupervised companion objectives to enforce cluster structure in earlier layers of its convolutional neural network. In the development of the unsupervised companion objectives, a connection between tensors and convolutional neural networks was utilized, in order to construct companion objectives capable of describing clusters of tensors. This work was inspired by ideas from several different fields of study, namely information theory, tensor theory, and deep learning.

Self-supervised learning in the form of context prediction [36] was also described and tested as a pre-training approach. Many deep clustering methods use autoencoders for pre-training, and self-supervised learning was recognized as an alternative to these, potentially reducing the number of extra parameters introduced in the pre-training stage.

In the experiments, both DTKC and DTKC-SS (DTKC with self-supervised pre-training), were compared to several benchmark algorithms from the literature – consisting of both classical algorithms, as well as algorithms from the more recent deep clustering field. The experimental results show that DTKC and DTKC-SS outperform, or perform comparable to, the benchmark algorithms. The experiments also describe the evaluation of two different variations of the base DTKC model, with the first being an alternate formulation of the unsupervised companion objectives. The second variation was a DTKC-based model in which a recurrent neural network was used in place of the convolutional neural network, resulting in a model for time series clustering.

The aforementioned connection between tensor theory and convolutional neural networks has received limited attention in the current deep learning scene. However, promising work on e.g. increasing network efficiency [31], and tensor factorization [32, 33], indicate that there is much untapped potential at the intersection between deep learning and tensor theory.

The preservation of geometrical structure in the input data is still a key challenge in deep clustering [24], and remains as an important direction for future work. To this end, the theoretical contributions made in this thesis show that the quantification of cluster structure can be done for tensors of arbitrary rank. More work along these lines can therefore lead to impactful advancements in the clustering field – especially for datatypes with more complex geometrical structure, such as images, sequences, and graphs.

# References

[1] S. Theodoridis and K. Koutroumbas. *Pattern Recognition.* Elsevier Acad. Press, Amsterdam, 4. edition, 2009. ISBN 978-1-59749-272-0. OCLC: 550588366.

[2] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, New York, NY, USA, 2014. ISBN 978-1-107-05713-5.

[3] E. Alpaydin. *Introduction to Machine Learning.* Adaptive Computation and Machine Learning. The MIT Press, Cambridge, Massachusetts, 3. edition, 2014. ISBN 978-0-262-02818-9.

[4] S. J. Russell, P. Norvig, E. Davis, and D. Edwards. *Artificial Intelligence: A Modern Approach.* Prentice Hall Series in Artificial Intelligence. Pearson, Boston Columbus Indianapolis New York San Francisco Upper Saddle River Amsterdam, Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo, 3., global edition, 2016. ISBN 978-1-292-15396-4.

[5] A. Rosenfeld, H. Huang, and V. Schneider. An application of cluster detection to text and picture processing. *IEEE Transactions on Information Theory*, 15(6):672–681, Nov. 1969. ISSN 0018-9448. doi: 10.1109/TIT.1969.1054378.

[6] T. Taxt and A. Lundervold. Multispectral analysis of the brain using magnetic resonance imaging. *IEEE Transactions on Medical Imaging*, 13(3): 470–481, Sept./1994. ISSN 02780062. doi: 10.1109/42.310878.

[7] A. H. S. Solberg, T. Taxt, and A. K. Jain. A Markov random field model for classification of multisource satellite imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 34(1):100–113, Jan. 1996. ISSN 0196-2892. doi: 10.1109/36.481897.

[8] H. H. Nguyen and P. Cohen. Gibbs Random Fields, Fuzzy Clustering, and the Unsupervised Segmentation of Textured Images. *CVGIP: Graph. Models Image Process.*, 55(1):1–19, Jan. 1993. ISSN 1049-9652. doi: 10.1006/cgip.1993.1001.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, Sept. 1999. ISSN 03600300. doi: 10.1145/331499.331504.

[10] A. K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, June 2010. ISSN 0167-8655. doi: 10.1016/j.patrec.2009.09.011.

[11] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553): 436–444, May 2015. ISSN 1476-4687. doi: 10.1038/nature14539.

[12] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25, pages 1097–1105, 2012.

[14] A. Graves, A.-R. Mohamed, and G. Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *International Conference on Acoustics, Speech and Signal Processing*, volume 38, Mar. 2013. doi: 10.1109/ICASSP.2013.6638947.

[15] J. Xie, R. Girshick, and A. Farhadi. Unsupervised Deep Embedding for Clustering Analysis. In *International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 478–487, June 2016.

[16] J. T. Springenberg. Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks. *arXiv:1511.06390 [cs, stat]*, Nov. 2015.

[17] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong. Towards K-means-friendly Spaces: Simultaneous Deep Learning and Clustering. In *International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3861–3870, Aug. 2017.

[18] U. Shaham, K. Stanton, H. Li, R. Basri, B. Nadler, and Y. Kluger. SpectralNet: Spectral Clustering using Deep Neural Networks. In *International Conference on Learning Representations*, 2018.

[19] J. Yang, D. Parikh, and D. Batra. Joint Unsupervised Learning of Deep Representations and Image Clusters. In *International Conference on Computer Vision and Pattern Recognition*, pages 5147–5156, 2016. doi: 10.1109/CVPR.2016.556.

[20] M. Kampffmeyer, S. Løkse, F. M. Bianchi, L. Livi, A.-B. Salberg, and R. Jenssen. Deep Divergence-Based Approach to Clustering. *Neural Networks*, 113:91–101, May 2019. ISSN 08936080. doi: 10.1016/j.neunet.2019.01.015.

[21] F. Li, H. Qiao, and B. Zhang. Discriminatively boosted image clustering with fully convolutional auto-encoders. *Pattern Recognition*, 83:161–173, Nov. 2018. ISSN 0031-3203. doi: 10.1016/j.patcog.2018.05.019.

[22] D. J. Trosten, A. S. Strauman, M. Kampffmeyer, and R. Jenssen. Recurrent Deep Divergence-based Clustering for Simultaneous Feature Learning and Clustering of Variable Length Time Series. In *International Conference on Acoustics, Speech and Signal Processing*, pages 3257–3261, May 2019. doi: 10.1109/ICASSP.2019.8682365.

[23] U. von Luxburg, R. C. Williamson, and I. Guyon. Clustering: Science or Art? In *International Conference on Machine Learning - Workshop on Unsupervised and Transfer Learning*, pages 65–79, June 2012.

[24] X. Guo, L. Gao, X. Liu, and J. Yin. Improved Deep Embedded Clustering with Local Structure Preservation. In *International Joint Conference on Artificial Intelligence*, pages 1753–1759, Melbourne, Australia, Aug. 2017. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-0-3. doi: 10.24963/ijcai.2017/243.

[25] Y. Lecun. Generalization and network design strategies. In *Connectionism in perspective*. Elsevier, 1989.

[26] R. Hadsell, P. Sermanet, J. Ben, A. Erkan, M. Scoffier, K. Kavukcuoglu, U. Muller, and Y. LeCun. Learning long-range vision for autonomous off-road driving. *Journal of Field Robotics*, 26(2):120–144, Feb. 2009. ISSN 1556-4959. doi: 10.1002/rob.20276.

[27] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Scene Parsing with Multiscale Feature Learning, Purity Trees, and Optimal Covers. In *International Conference on Machine Learning*, ICML'12, pages 1857–1864, USA, 2012. Omnipress. ISBN 978-1-4503-1285-1.

[28] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. A guide to deep learning in healthcare. *Nature Medicine*, 25(1):24, Jan. 2019. ISSN 1546-170X. doi: 10.1038/s41591-018-0316-z.

[29] X. Zhu, D. Tuia, L. Mou, G.-S. Xia, L. Zhang, F. Xu, and F. Fraundorfer. Deep Learning in Remote Sensing: A Comprehensive Review and List of Resources. *IEEE Geoscience and Remote Sensing Magazine*, 5:8–36, Dec. 2017. doi: 10.1109/MGRS.2017.2762307.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *International Conference on Computer Vision*, pages 1026–1034, Dec. 2015. doi: 10.1109/ICCV.2015.123.

[31] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition. *arXiv:1412.6553 [cs]*, Dec. 2014.

[32] J.-T. Chien and Y.-T. Bao. Tensor-Factorized Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, PP:1–14, Apr. 2017. doi: 10.1109/TNNLS.2017.2690379.

[33] S. Oymak and M. Soltanolkotabi. End-to-end Learning of a Convolutional Neural Network via Deep Tensor Decomposition. *arXiv:1805.06523 [cs, math, stat]*, May 2018.

[34] C.-Y. Lee, S. Xie, P. W. Gallagher, Z. Zhang, and Z. Tu. Deeply-Supervised Nets. *Journal of Machine Learning Research*, 38, 2015.

[35] M. Signoretto. *Kernels and Tensors for Structured Data Modelling.* PhD, Katholieke Universiteit Leuven – Faculty of Engineering, Dec. 2011.

[36] C. Doersch, A. Gupta, and A. A. Efros. Unsupervised Visual Representation Learning by Context Prediction. In *IEEE International Conference on Computer Vision*, pages 1422–1430, Santiago, Chile, Dec. 2015. IEEE. ISBN 978-1-4673-8391-2. doi: 10.1109/ICCV.2015.167.

[37] R. Jenssen, J. C. Principe, D. Erdogmus, and T. Eltoft. The Cauchy–Schwarz divergence and Parzen windowing: Connections to graph theory and Mercer kernels. *Journal of the Franklin Institute*, 343(6):614–629, Sept. 2006. ISSN 00160032. doi: 10.1016/j.jfranklin.2006.03.018.

[38] E. Parzen. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, Sept. 1962. ISSN 0003-4851. doi: 10.1214/aoms/1177704472.

[39] D. J. Trosten and P. Sharma. Unsupervised Feature Extraction – A CNN-Based Approach. In *Scandinavian Conference on Image Analysis*, Lecture Notes in Computer Science, pages 197–208. Springer International Publishing, 2019. ISBN 978-3-030-20205-7.

[40] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49, Feb. 1978. ISSN 0096-3518. doi: 10.1109/TASSP.1978.1163055.

[41] G. E. Batista, X. Wang, and E. J. Keogh. A Complexity-Invariant Distance Measure for Time Series. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 699–710, Philadelphia, PA, Apr. 2011. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-992-5 978-1-61197-281-8. doi: 10.1137/1.9781611972818.60.

[42] V. Di Gesù and V. Starovoitov. Distance-based functions for image comparison. *Pattern Recognition Letters*, 20(2):207–214, Feb. 1999. ISSN 01678655. doi: 10.1016/S0167-8655(98)00115-9.

[43] R. Souvenir and R. Pless. Image distance functions for manifold learning. *Image and Vision Computing*, 25(3):365–373, Mar. 2007. ISSN 0262-8856. doi: 10.1016/j.imavis.2006.01.016.

[44] S. T. Roweis and L. K. Saul. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500):2323–2326, Dec. 2000. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.290.5500.2323.

[45] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957. ISBN 978-0-691-07951-6.

[46] K. Pearson. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, Nov. 1901. ISSN 1941-5982. doi: 10.1080/14786440109462720.

[47] L. van der Maaten and G. Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[48] I. Borg and P. J. F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer Series in Statistics. Springer, New York, 1997. ISBN 978-0-387-94845-4.

[49] M. Belkin and P. Niyogi. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation*, 15:1373–1396, 2002.

[50] L. McInnes, J. Healy, and J. Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv:1802.03426 [cs, stat]*, Feb. 2018.

[51] R. Xu and D. C. Wunsch. *Clustering*. IEEE Press Series on Computational Intelligence. Wiley ; IEEE Press, Hoboken, N.J. : Piscataway, NJ, 2009. ISBN 978-0-470-27680-8. OCLC: ocn216937130.

[52] G. W. Milligan and M. C. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(2):159–179, June 1985. ISSN 1860-0980. doi: 10.1007/BF02294245.

[53] O. Arbelaitz, I. Gurrutxaga, J. Muguerza, J. M. Pérez, and I. Perona. An extensive comparative study of cluster validity indices. *Pattern Recognition*, 46(1):243–256, Jan. 2013. ISSN 0031-3203. doi: 10.1016/j.patcog.2012.07.021.

[54] T. Calinski and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics - Theory and Methods*, 3(1):1–27, 1974. ISSN 0361-0926. doi: 10.1080/03610927408827101.

[55] D. L. Davies and D. W. Bouldin. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2): 224–227, Apr. 1979. ISSN 0162-8828. doi: 10.1109/TPAMI.1979.4766909.

[56] J. Boberg and T. Salakoski. General formulation and evaluation of agglomerative clustering methods with metric and non-metric distances. *Pattern Recognition*, 26(9):1395–1406, Sept. 1993. ISSN 00313203. doi: 10.1016/0031-3203(93)90145-M.

[57] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. The Regents of the University of California, 1967.

[58] L. Bottou and Y. Bengio. Convergence Properties of the K-Means Algorithms. In *Advances in Neural Information Processing Systems*, volume 7, pages 585–592. MIT Press, 1995.

[59] D. Arthur and S. Vassilvitskii. K-means++: The Advantages of Careful Seeding. In *ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-624-5.

[60] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, Nov. 1993. ISSN 0162-8828. doi: 10.1109/34.244673.

[61] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, Aug. 2000. ISSN 0162-8828. doi: 10.1109/34.868688.

[62] U. von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, Dec. 2007. ISSN 0960-3174, 1573-1375. doi: 10.1007/s11222-007-9033-z.

[63] Yu and Shi. Multiclass spectral clustering. In *International Conference on Computer Vision*, pages 313–319 vol.1, Oct. 2003. doi: 10.1109/ICCV.2003.1238361.

[64] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977. ISSN 0035-9246.

[65] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):790–799, Aug./1995. ISSN 01628828. doi: 10.1109/34.400568.

[66] F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review*, pages 65–386, 1958.

[67] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct. 1986. ISSN 1476-4687. doi: 10.1038/323533a0.

[68] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *International Conference on Computer Vision*, pages 2146–2153, Kyoto, Sept. 2009. IEEE. ISBN 978-1-4244-4420-5. doi: 10.1109/ICCV.2009.5459469.

[69] V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress. ISBN 978-1-60558-907-7.

[70] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, Apr. 2011. PMLR.

[71] J. L. Elman. Finding Structure in Time. *Cognitive Science*, 14(2):179–211, Mar. 1990. ISSN 1551-6709. doi: 10.1207/s15516709cog1402_1.

[72] Y. Bengio, P. Frasconi, and P. Simard. The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1188, San Francisco, CA, USA, 1993. IEEE. ISBN 978-0-7803-0999-9. doi: 10.1109/ICNN.1993.298725.

[73] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5 (2):157–166, 1994. ISSN 1045-9227. doi: 10.1109/72.279181.

[74] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, Nov. 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.

[75] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734, Oct. 2014. doi: 10.3115/v1/D14-1179.

[76] C. Olah. Understanding LSTM Networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, Aug. 2015.

[77] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4):291–294, Sept. 1988. ISSN 1432-0770. doi: 10.1007/BF00332918.

[78] G. E. Hinton and R. S. Zemel. Autoencoders, Minimum Description Length and Helmholtz Free Energy. In *Advances in Neural Information Processing Systems*, volume 6, pages 3–10. Morgan-Kaufmann, 1994.

[79] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In *International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390294.

[80] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010. ISSN ISSN 1533-7928.

[81] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber. Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction. In *Artificial Neural Networks and Machine Learning*, volume 6791, pages 52–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21734-0 978-3-642-21735-7. doi: 10.1007/978-3-642-21735-7_7.

[82] N. Srivastava, E. Mansimov, and R. Salakhutdinov. Unsupervised Learning of Video Representations Using LSTMs. In *International Conference on Machine Learning*, volume 37 of *ICML'15*, pages 843–852. JMLR.org, 2015.

[83] G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, July 2006. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.1127647.

[84] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, June 2015. doi: 10.1109/CVPR.2015.7298965.

[85] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*, Sept. 2016.

[86] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[87] M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv:1212.5701 [cs]*, Dec. 2012.

[88] G. Hinton, N. Srivastava, and K. Swersky. Neural Networks for Machine Learning - Lecture 6a.

[89] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*, 2015.

[90] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning*, volume 37 of *ICML'15*, pages 448–456. JMLR.org, 2015.

[91] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How Does Batch Normalization Help Optimization? In *Advances in Neural Information Processing Systems*, volume 31, pages 2483–2493. Curran Associates, Inc., 2018.

[92] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng. Building High-level Features Using Large Scale Unsupervised Learning. In *International Conference on Machine Learning*, ICML'12, pages 507–514, USA, 2012. Omnipress. ISBN 978-1-4503-1285-1.

[93] U. Shaham and R. R. Lederman. Learning by coincidence: Siamese networks and common variable learning. *Pattern Recognition*, 74:52–63, Feb. 2018. ISSN 0031-3203. doi: 10.1016/j.patcog.2017.09.015.

[94] H. Noh, S. Hong, and B. Han. Learning Deconvolution Network for Semantic Segmentation. In *International Conference on Computer Vision*, pages 1520–1528, Santiago, Chile, Dec. 2015. IEEE. ISBN 978-1-4673-8391-2. doi: 10.1109/ICCV.2015.178.

[95] W. Hu, T. Miyato, S. Tokui, E. Matsumoto, and M. Sugiyama. Learning Discrete Representations via Information Maximizing Self-Augmented Training. In *International Conference on Machine Learning*, volume 70, pages 1558–1567, Aug. 2017.

[96] Z. Jiang, Y. Zheng, H. Tan, B. Tang, and H. Zhou. Variational Deep Embedding: An Unsupervised and Generative Approach to Clustering. In *International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 1965–1972. AAAI Press, 2017. ISBN 978-0-9992411-0-3.

[97] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations*, Dec. 2014.

[98] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, volume 27, pages 2672–2680. Curran Associates, Inc., 2014.

[99] G. H. Givens and J. A. Hoeting. *Computational Statistics.* Wiley Series in Computational Statistics. Wiley, Hoboken, N.J, 2. edition, 2013. ISBN 978-0-470-53331-4.

[100] R. Jenssen. Kernel Entropy Component Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(5):847–860, May 2010. ISSN 0162-8828. doi: 10.1109/TPAMI.2009.100.

[101] J. Dieudonné. *Foundations of Modern Analysis.* Academic Press, New York, 1969. ISBN 978-0-12-215550-5. OCLC: 576465.

[102] M. Reed and B. Simon. *Methods of Modern Mathematical Physics.* Academic Press, New York, rev. and enl. edition, 1980. ISBN 978-0-12-585050-6.

[103] B. Schölkopf, A. Smola, and K.-R. Müller. Nonlinear Component Analysis as a Kernel Eigenvalue Problem. *Neural Computation*, 10(5):1299–1319, July 1998. ISSN 0899-7667. doi: 10.1162/089976698300017467.

[104] B. Scholkopf, C. Burges, and A. Smola. Advances in Kernel Methods - Support Vector Learning. *MIT Press*, Dec. 1998.

[105] S. Jayasumana, R. Hartley, M. Salzmann, H. Li, and M. Harandi. Kernel Methods on Riemannian Manifolds with Gaussian RBF Kernels. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(12):2464–2477, Dec. 2015. ISSN 0162-8828, 2160-9292. doi: 10.1109/TPAMI.2015.2414422.

[106] J. Hamm and D. D. Lee. Grassmann Discriminant Analysis: A Unifying View on Subspace-based Learning. In *International Conference on Machine Learning*, pages 376–383. ACM, 2008. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390204.

[107] J. Zhang, G. Zhu, R. W. Heath Jr., and K. Huang. Grassmannian Learning: Embedding Geometry Awareness in Shallow and Deep Learning. *arXiv:1808.02229 [cs, eess, math, stat]*, Aug. 2018.

[108] H. Anton and C. Rorres. *Elementary Linear Algebra: With Supplementary Applications.* Wiley, Hoboken, NJ, 11., international student edition, 2015. ISBN 978-1-118-67745-2. OCLC: 931593706.

[109] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009. doi: 10.1109/CVPR.2009.5206848.

[110] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[111] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov. 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

[112] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. In *Neural Information Processing Systems*, page 9, 2011.

[113] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. Aug. 2017.

[114] S. A. Nene, S. K. Nayar, and H. Murase. Columbia Object Image Library (COIL-20). Technical report, Feb. 1996. Technical Report CUCS-005-96.

[115] S. A. Nene, S. K. Nayar, and H. Murase. Columbia Object Image Library (COIL-100). Technical report, Feb. 1996. Technical Report CUCS-006-96.

[116] D. B. Graham and N. M. Allinson. Characterising Virtual Eigensignatures for General Purpose Face Recognition. In *Face Recognition: From Theory to Applications*, NATO ASI Series, pages 446–456. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-642-72201-1. doi: 10.1007/978-3-642-72201-1_25.

[117] University of Sheffield - Face Database. `https://www.sheffield.ac.uk/eee/research/iel/research/face`.

[118] T. Zhang, P. Ji, M. Harandi, R. Hartley, and I. Reid. Scalable Deep k-Subspace Clustering. *arXiv:1811.01045 [cs]*, Nov. 2018.

[119] X. Peng, S. Xiao, J. Feng, W.-Y. Yau, and Z. Yi. Deep Subspace Clustering with Sparsity Prior. In *International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1925–1931, New York, New York, USA, 2016. AAAI Press. ISBN 978-1-57735-770-4.

[120] S. E. Chazan, S. Gannot, and J. Goldberger. Deep Clustering Based on a Mixture of Autoencoders. *arXiv:1812.06535 [cs, stat]*, Dec. 2018.

[121] M. M. Fard, T. Thonet, and E. Gaussier. Deep k-Means: Jointly clustering with k-Means and learning representations. *arXiv:1806.10069 [cs, stat]*, June 2018.

[122] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, Mar. 2010.

[123] A. Joulin, L. van der Maaten, A. Jabri, and N. Vasilache. Learning Visual Features from Large Weakly Supervised Data. In *European Conference on Computer Vision*, Lecture Notes in Computer Science, pages 67–84. Springer International Publishing, 2016. ISBN 978-3-319-46478-7.

[124] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. In *International Conference on Computer Vision*, pages 843–852, 2017.

[125] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for Simplicity: The All Convolutional Net. In *International Conference on Learning Representations (workshop track)*, 2015.

[126] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. In *European Conference on Computer Vision*, Lecture Notes in Computer Science, pages 818–833. Springer International Publishing, 2014. ISBN 978-3-319-10590-1.

[127] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLOS ONE*, 10(7):e0130140, July 2015. ISSN 1932-6203. doi: 10.1371/journal.pone.0130140.

[128] D. Dheeru and E. Karra Taniskidou. *UCI Machine Learning Repository.* University of California, Irvine, School of Information and Computer Sciences, 2017.

[129] K. Ø. Mikalsen, F. M. Bianchi, C. Soguero-Ruiz, and R. Jenssen. Time series cluster kernel for learning similarities between multivariate time series with missing data. *Pattern Recognition*, 76:569–581, Apr. 2018. ISSN 00313203. doi: 10.1016/j.patcog.2017.11.030.

[130] F. M. Bianchi, L. Livi, K. Ø. Mikalsen, M. Kampffmeyer, and R. Jenssen. Learning representations for multivariate time series with missing data using Temporal Kernelized Autoencoders. *arXiv:1805.03473 [cs]*, May 2018.

[131] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, Mar. 1951. ISSN 0003-4851, 2168-8990. doi: 10.1214/aoms/1177729694.

[132] A. Rényi. On Measures of Entropy and Information. In *Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics.* The Regents of the University of California, 1961.

[133] C. Jones, J. S. Marron, and S. J. Sheather. Progress in data-based bandwidth selection for kernel density estimation. *Computational Statistics*, pages 337–381, 1996. ISSN 1613-9658.

[134] R. Cao, A. Cuevas, and W. González Manteiga. A comparative study of several smoothing methods in density estimation. *Computational Statistics & Data Analysis*, 17(2):153–176, Feb. 1994. ISSN 01679473. doi: 10.1016/0167-9473(92)00066-Z.

[135] M. C. Jones, J. S. Marron, and S. J. Sheather. A Brief Survey of Bandwidth Selection for Density Estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996. ISSN 0162-1459. doi: 10.2307/2291420.

[136] D. W. Scott. Multivariate Density Estimation and Visualization. In *Handbook of Computational Statistics*, pages 549–569. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-21550-6 978-3-642-21551-3. doi: 10.1007/978-3-642-21551-3_19.

[137] E. Gokcay and J. C. Principe. A new clustering evaluation function using Renyi's information potential. In *International Conference on Acoustics, Speech, and Signal Processing. Proceedings*, volume 6, pages 3490–3493 vol.6, June 2000. doi: 10.1109/ICASSP.2000.860153.

[138] R. Jenssen, D. Erdogmus, K. E. Hild, J. C. Principe, and T. Eltoft. Information cut for clustering using a gradient descent approach. *Pattern Recognition*, 40(3):796–806, Mar. 2007. ISSN 0031-3203. doi: 10.1016/j.patcog.2006.06.028.

[139] V. V. Vikjord and R. Jenssen. Information theoretic clustering using a k-nearest neighbors approach. *Pattern Recognition*, 47(9):3070–3081, Sept. 2014. ISSN 0031-3203. doi: 10.1016/j.patcog.2014.03.018.

[140] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral Networks and Locally Connected Networks on Graphs. *arXiv:1312.6203 [cs]*, Dec. 2013.

[141] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc., 2017.

[142] N. Tishby and N. Zaslavsky. Deep learning and the information bottleneck principle. In *IEEE Information Theory Workshop*, pages 1–5, Apr. 2015. doi: 10.1109/ITW.2015.7133169.

[143] R. Shwartz-Ziv and N. Tishby. Opening the Black Box of Deep Neural Networks via Information. *arXiv:1703.00810 [cs]*, Mar. 2017.

[144] S. Yu, K. Wickstrøm, R. Jenssen, and J. C. Principe. Understanding Convolutional Neural Networks with Information Theory: An Initial Exploration. *arXiv:1804.06537 [cs, math, stat]*, Apr. 2018.

[145] B. Efron. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1):1–26, Jan. 1979. ISSN 0090-5364, 2168-8966. doi: 10.1214/aos/1176344552.

[146] C. A. Sugar and G. M. James. Finding the Number of Clusters in a Dataset: An Information-Theoretic Approach. *Journal of the American Statistical*

*Association*, 98(463):750–763, Sept. 2003. ISSN 0162-1459, 1537-274X. doi: 10.1198/016214503000000666.

[147] R. C. Dubes. How many clusters are best? - An experiment. *Pattern Recognition*, 20(6):645–663, Jan. 1987. ISSN 0031-3203. doi: 10.1016/0031-3203(87)90034-3.

[148] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. Clustering Validity Checking Methods: Part II. *SIGMOD Record*, 31(3):19–27, Sept. 2002. ISSN 0163-5808. doi: 10.1145/601858.601862.